6.111 Final Project Report

Rod Bayliss III and Brandon John

Dr. Hom

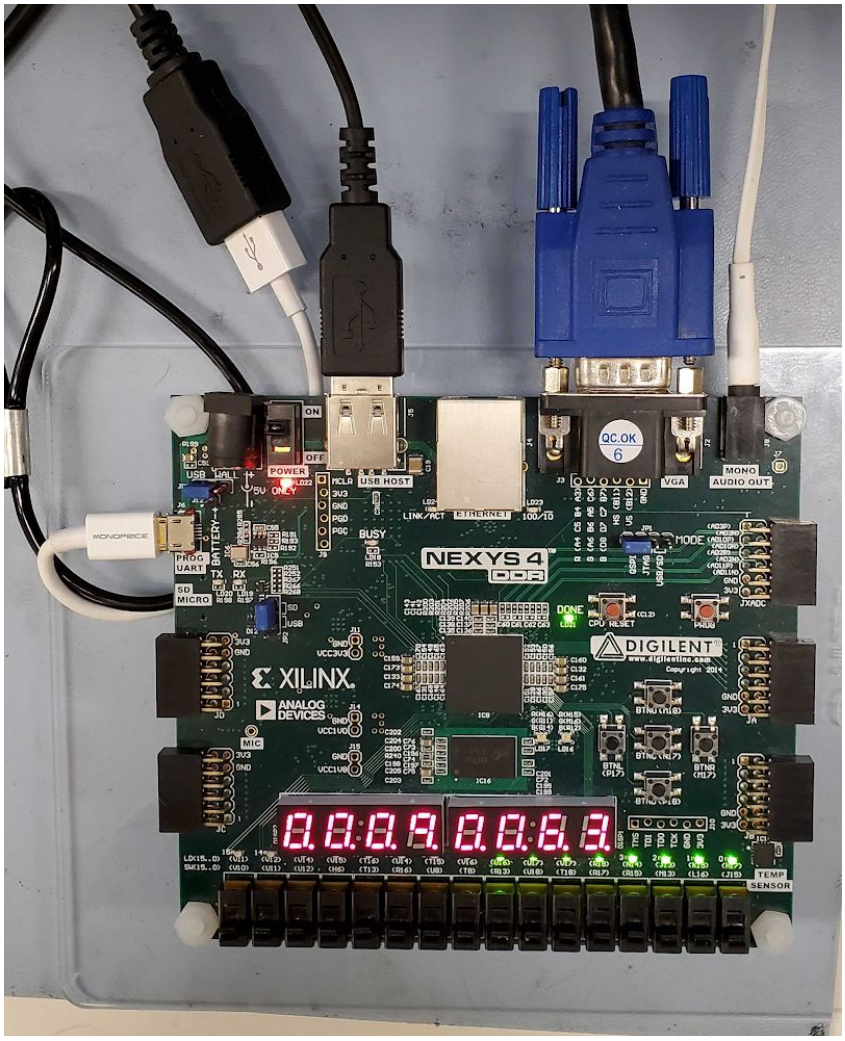
11th December 2019

Abstract

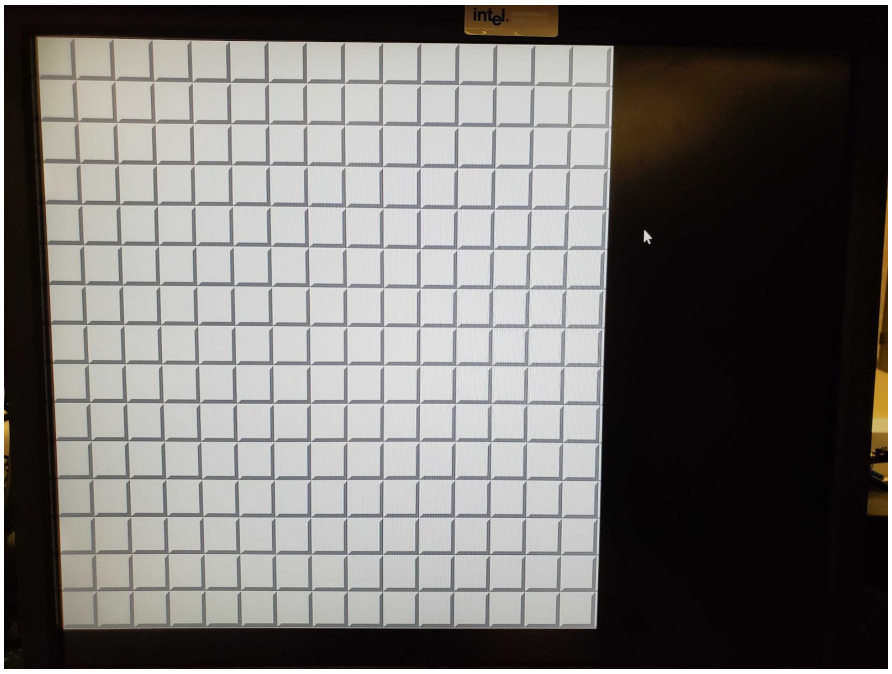
Our project was to create the classic arcade game “minesweeper” on an FPGA. Minesweeper involves a grid upon which the player can click to guess whether a cell is an empty cell or contains a mine. If they click on a mine, then the game is over. If they click on an empty cell, the cell will reveal how many neighboring cells have mines. We modeled our FPGA implementation on the classic Windows XP version. Our FPGA implementation involved several core modules: VGA to draw and interact with the game, FSMs to run the game logic, sound effects through speakers, and a USB-HID mouse to interact with the game.

Pictures

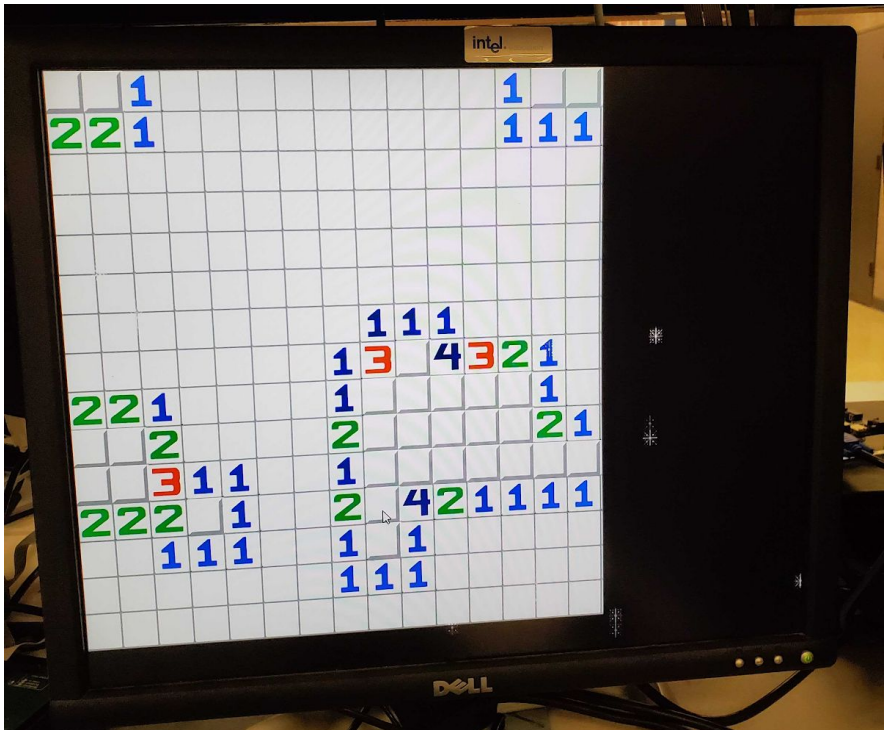
Physical Setup



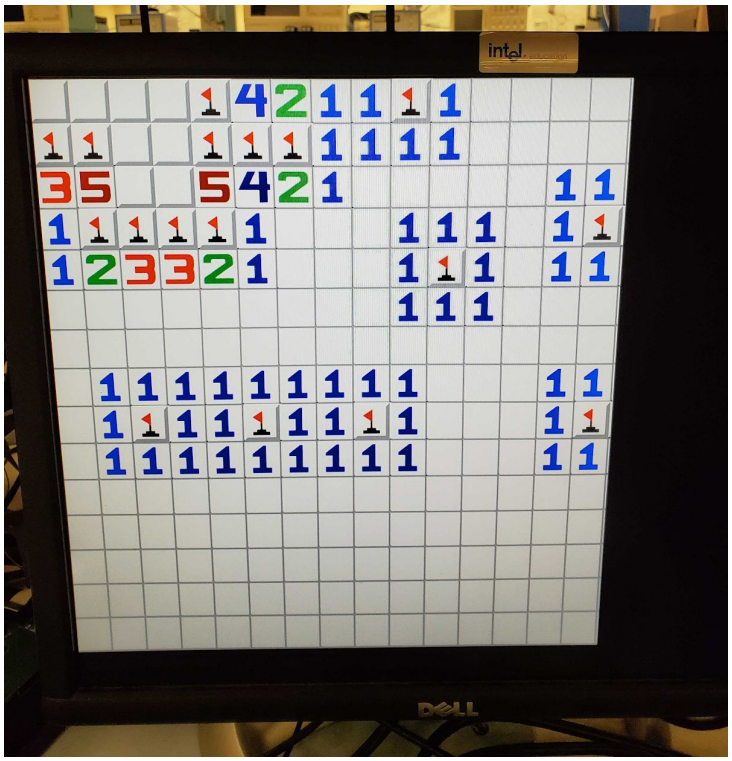
Blank Game Board



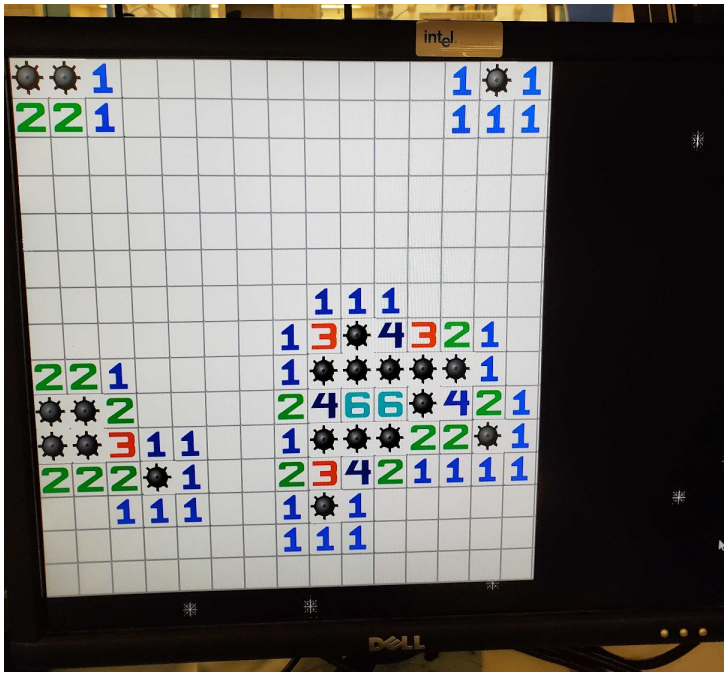
Post first-click game board with snowflakes



Game Board with Flags

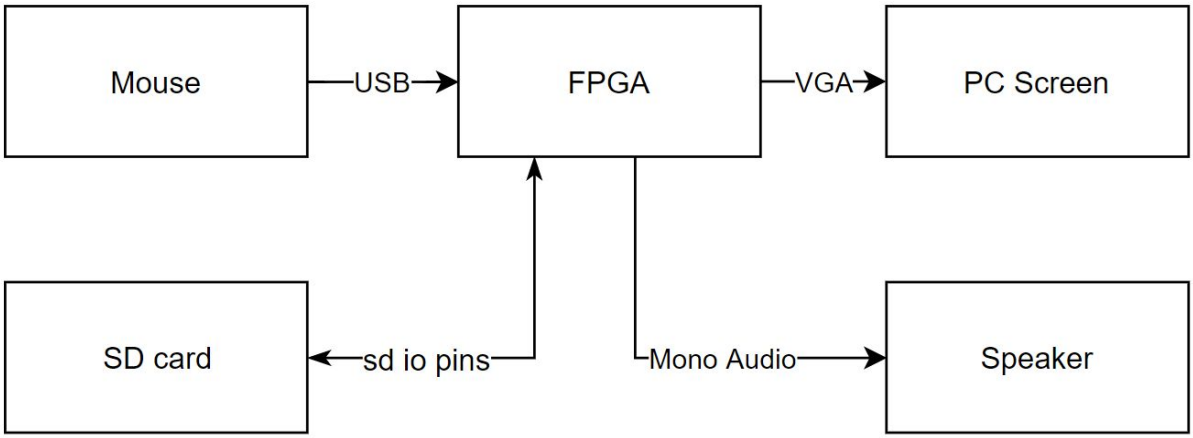


Post Game Screen With Snowflakes



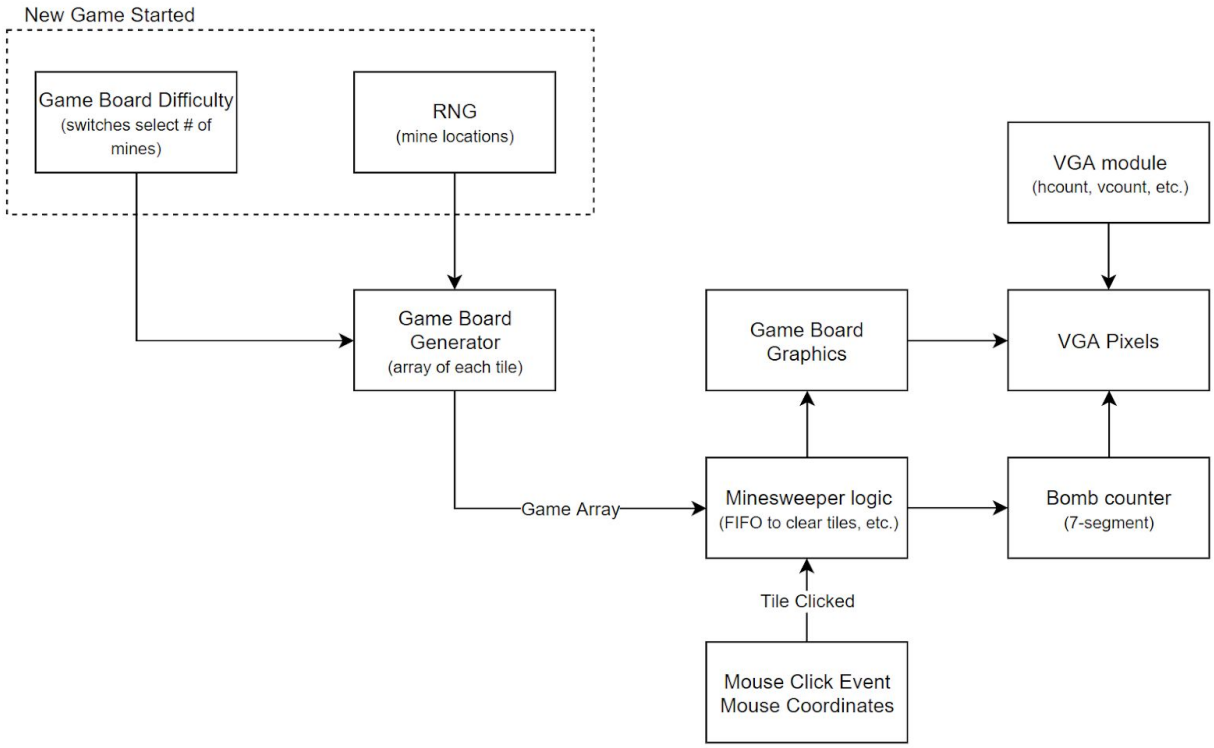
Block Diagrams and Module Descriptions

Hardware Block Diagram



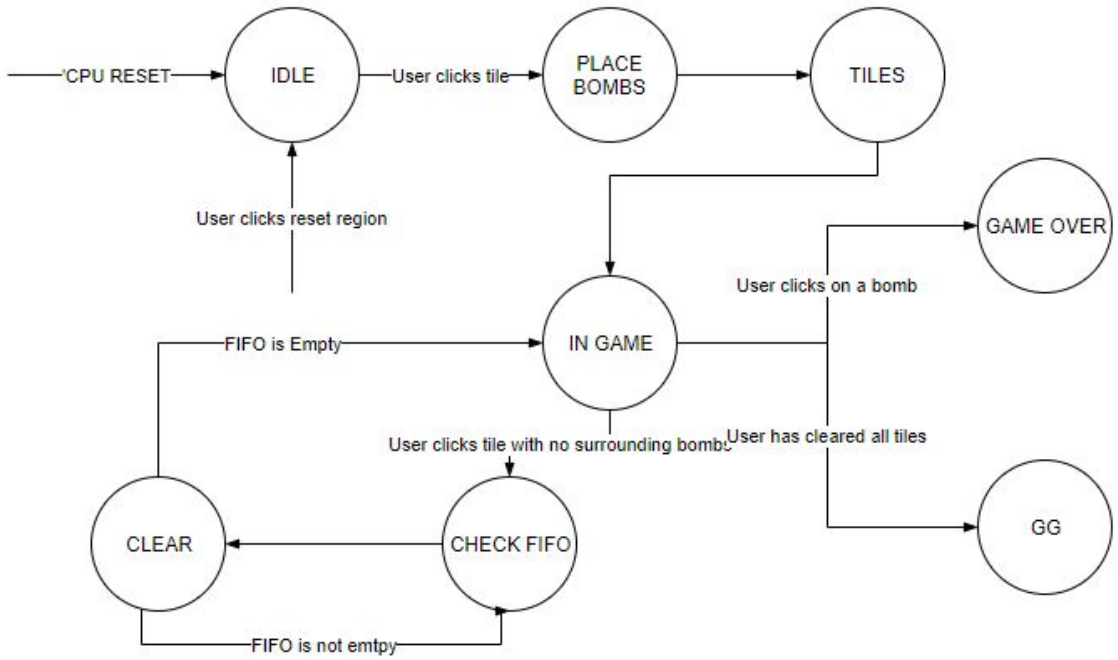
The system only needs a modest amount of IO to support this project. On the input side we have a USB mouse (plugged into the USB Host port) and an SD card (into the SD card slot). The outputs are similarly straightforward, we will output audio over via the 3.5mm headphone jack and video through the VGA port.

Minesweeper.sv Block Diagram (Rod)



My main role in the project was implementing the minesweeper logic and VGA generation of the game. First, the game status is stored through several 15x15 arrays. `Bomb_locations` is a 15x15 array of bits that are 1 if a bomb is located at the specified tile. This array is populated via RNG which I'll get into later. Tile_status is a 15x15 array of 2 bit numbers where 0 means the tile has not been cleared, 1 means the tile has been cleared successfully, and 3 if the tile has been flagged. `Tile_numbers` is the final 15x15 array of 4 bit numbers. This array keeps track of each non-bomb tile's number of adjacent bombs. Additional game information given to the user is the number of flags and a 1 Hz timer fed to the seven segment display. Because most humans can't intuitively read hex super fast I fed these numbers into a hex to decimal converter that made the values more legible when displayed on the seven segment.

Minesweeper State Machine



Here is the state machine that makes the game work well. First, any cpu reset event or user clicking on the right portion of the screen will send the game into the idle state, no matter what state the FPGA is presently in. After a user click on a tile, the game goes into the PLACE_BOMBS state. Here, the game iterates through the game array in the same way that VGA iterates through the screen (i.e. row by row, then resetting at the top row). On each cycle the RNG number is updated and if it is above a certain threshold, a bomb is marked on the current tile. There are some other considerations such as the first click not being on a bomb, and not placing a bomb in the same tile twice. After setting the threshold of the RNG number to place a bomb reasonably high (i.e. if(random number)>50,000, place bomb [random is between 0 and 65535]), we were able to get rather varied game boards that didn't have bomb concentrations in the top left of the game board (a potential consequence of placing the bombs in this manner). After all bombs have been placed and the bomb_locations array updated, the FSM goes into the TILES state. Here, each tile checks the number of surrounding bombs and marks the location appropriately. A lot of edge cases had to be handled here as the top row of tiles have no tiles above them so you could reference the bomb_locations array poorly and run into some bad times. After both of these are completed, the FSM finally enters the IN_GAME state.

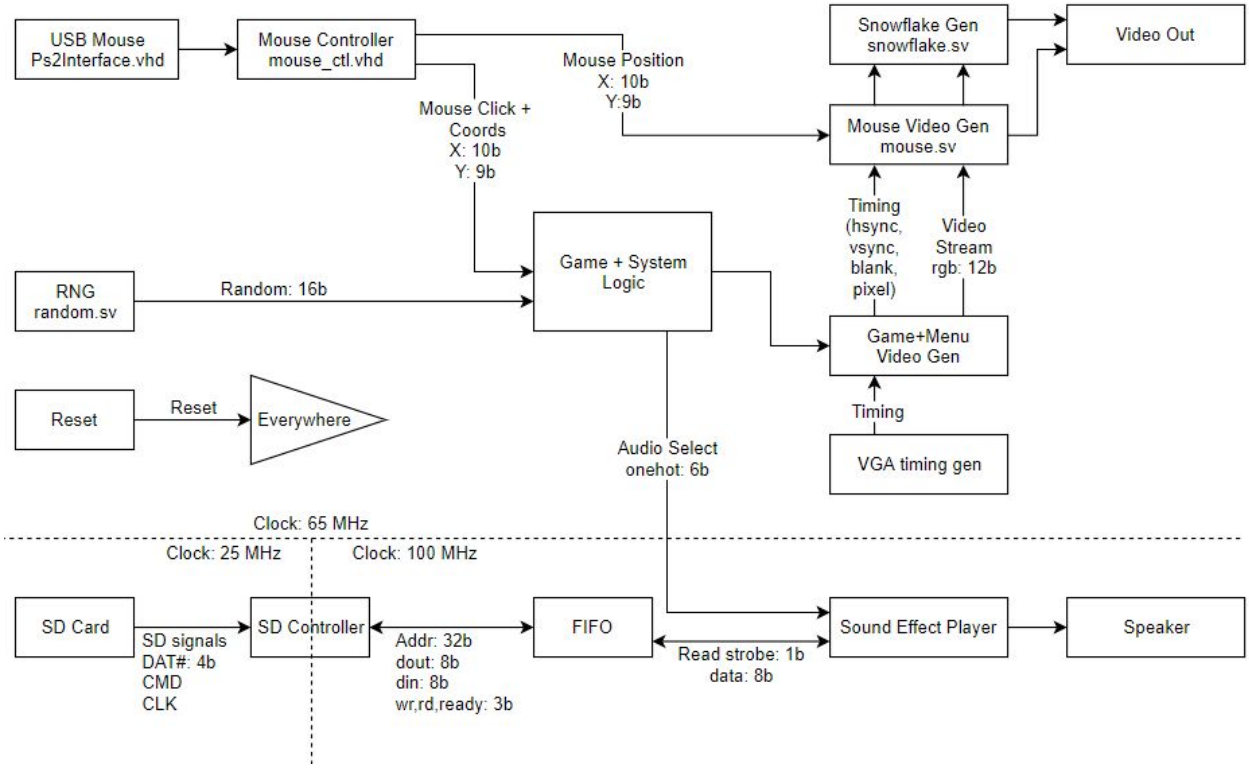
In the IN_GAME state, the game simply clears tiles that are marked with a non-zero tile number, and flags tiles that are right clicked. Constantly, the minesweeper module is "binning" the mouse position into the tile it is located in by dividing the raw position by the tile size (48 pixels). If, however, the user clicks on a tile with zero adjacent bombs, the game needs to clear

all the tiles until a boundary layer of only tiles with numbers are reached. This places us into the CLEAR state. Here, each adjacent tile to the originally clicked tile is cleared. If any of these 8 tiles also have zero adjacent bombs, they are added to a FIFO. After all 8 of the original adjacent tiles have been cleared and processed, the game goes into the CHECK_FIFO state. Here, if the FIFO is empty, the game returns to the in-game state. Otherwise, the FSM returns to the CLEAR state with the new tile to check being one of the fifo tiles. Coming up with this implementation and debugging was one of the more time-consuming final bits of the game but it was extremely satisfying once implemented. Finally, once a user clears all the tiles or clicks on a bomb, the game goes into the GG or GAME_OVER states respectively. This stops the timer and displays whether you won or lost on the seven segment display.

VGA Generation

The last big block in the minesweeper.s file is the tile_drawer, which given the tile_status and tile_numbers array actually draws the screen. I used the MATLAB script from lab 3 to generate the image and rgb coe files from the 48x48 pixel images of the tiles I had. On each cycle, image_addr determines based on the hcount and vcount location which tile the VGA pointer is currently. This image_addr is then fed concurrently to all of the BRAMs of the image_rom and rgb_roms. From there, given tile_status and tile_numbers, the module chooses which rgb_rom output to give to the screen. I.e if the current tile has been uncovered, set `pixel_out <= uncovered_rgb`, if it's a flag, `pixel_out <= flag_rgb`. Because each BRAM access takes 2 clock cycles and the multiplication and division I used to come up with image_addr takes around 2 clock cycles, I had to pipeline the VGA signals significantly in order to time everything correctly.

Top_level.sv Block Diagram (Brandon)



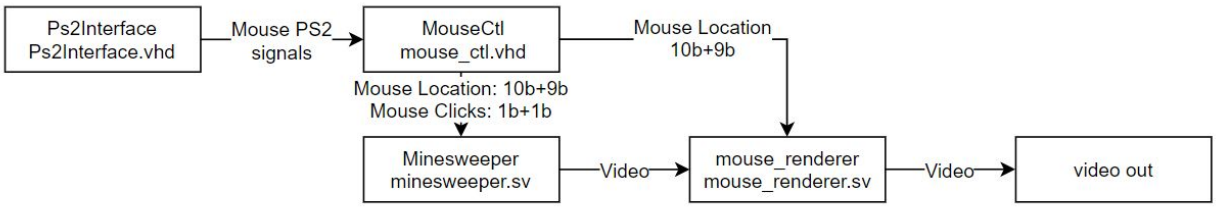
Most of Top_Level.sv is just linking other modules together, but it does have a few important pieces of functionality that we are not discussing elsewhere in this report.

First, this module includes the clock divider, which converts the input 100mhz clock into a 65mhz clock for the video subsystem, and a 25mhz clock for the SD card subsystem. We ran the sound effect player at 100mhz because this was an integer multiple of the SD card subsystem and could therefore guarantee the timing between clock domains to be consistent. The only interface between the 65MHz game/video system and the 100mhz sound system was a one-hot sound initiator bus with few precise timing limitations.

Top_Level.sv also instantitates some modules from utils.sv, specifically the xvga timing generator, debounce modules for input sanitization, and the seven segment display module. These were all written or provided previously in the semester, so I will not go into detail on how they work.

The final functionality of top_level.sv is that it muxes the output video between the “holiday theme” version with snowflakes and the regular output based on the state of sw[14]. This is a simple if/else inside the provided video output block.

Mouse.sv, mouse_ctl.vhd, Ps2Interface.vhd (Brandon)

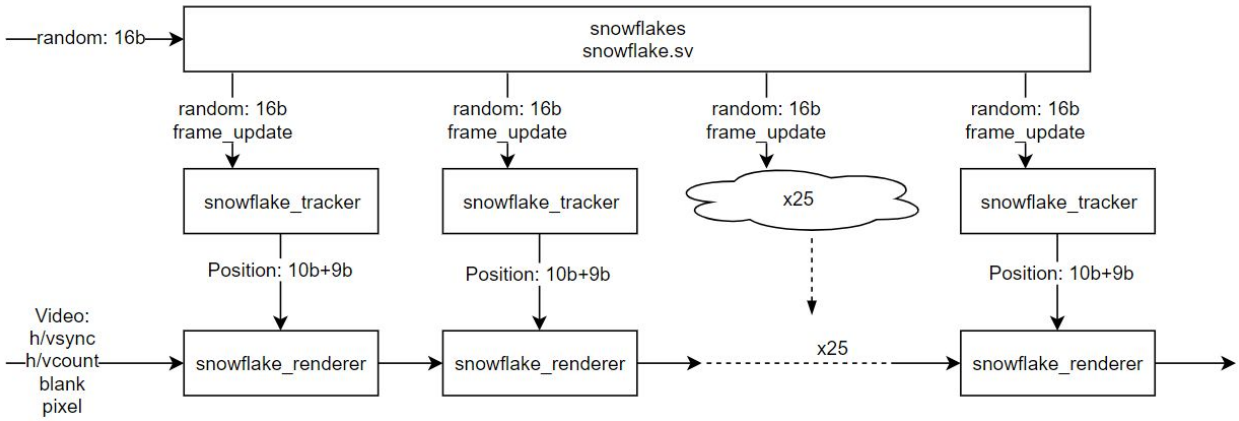


The mouse system for our design can be broken into two parts. The first part is the mouse controller/physical interface, and the second is the graphics engine behind it. While I started out attempting to write my own interface, I quickly realized that this would be a very tedious process without much to be learned, as it was mostly just a large state machine to handle all the commands that had to be sent to the mouse. So I decided to look around for other options. There was one verilog version provided on the course website, but it seemed to have bugs or something and I couldn't get it to work. I therefore decided to look elsewhere, and eventually settled on a verified and feature complete version written in vhdl (mouse_ctl.vhd, Ps2Interface.vhd). This then meant that I had to learn a little vhdl to make some modifications to this implementation, specifically for the system frequency, screen resolution, and output data format.

The second part of this implementation was the mouse renderer; I wrote this part from scratch (mouse.sv). All this module ended up having to do was draw the mouse over the pixels provided from the minesweeper module, and pipeline the timing signals appropriately. This amounted to a 2-stage pipeline where the first calculated the relative position of the current pixel compared to the mouse pointer location, and then also determined whether or not the pixel was within the bounding box defining the mouse. The second stage then either passed on the value of "pixel_in" from the minesweeper module or overwrote with either a black or white pixel, based on the relative location and status of the bounding box. It also wrote out the delayed hcount, vcount, and clock signals, they had to be delayed such that the pixels could stay in sync with the timing signals and the screen would render correctly. I wrote the pixel overwrite code with some manual if-else conditions, because I realized that the synthesis and implementation times would be quicker this way than if I allocated an entire bram to holding the state of a super tiny .coe for the mouse icon. Manually writing the conditions didn't take long either, and I now have a standalone mouse renderer that doesn't rely on any external files. Overall this portion was a resounding success.

The original implementation of mouse.sv also had a functioning "integrator" written, which would take mouse movement deltas directly from the ps2 interface and convert that into mouse movement. This had some interesting edge cases to consider, namely interactions with edges and corners. However I ended up removing this portion as it was no longer necessary once I found the functioning vhdl version.

Snowflake.sv (Brandon)



Our “holiday theme” stretch goal ended up involving two parts. The first was the jingle bells theme (covered in detail in the sound effect section), and the second was snowflakes randomly falling over the screen. These snowflakes ended up being a fairly massive implementation. At the center of the snowflake implementation is the snowflakes module. This module takes in the vga generation signals (h/vcount, h/vsync, blank, pixel) and then outputs the pipeline-delayed version of these. It also takes in a 16 bit random number on every 65mhz clock cycle. This module also creates a 60fps clock for snowflake frame updates. All of the actual snowflake generation happens in individual modules that are then chained together with a generate loop, which allows us to parametrically determine how many snowflakes we will include..

The first of these modules is the snowflake_tracker. This takes in a 60hz update strobe and the random number, then generates the location of the falling snowflakes. The major challenge in this section was creating randomness out of a only-kinda-random source. Specifically, all of the snowflakes had to fall out of sync and in different locations, but if I sent them all the same random numbers then they would all fall perfectly in sync. To overcome this I ended up sending the 60hz update strobes out of phase with each other. The snowflake tracker then took latched the random number every time it got an update strobe, meaning that the individual snowflakes were getting different random numbers and thus able to each fall independently.

Snowflake_tracker has 4 main states in its state machine. The first is a “wait” state that determines when the snowflake is ready to fall. It waits for a random number in a small range to appear, such that on each frame update the flake has a 2100/65536 chance (3.2%) chance of falling. This allows the flakes to fall in random clumps and not have a constant number of them displayed on the screen at any given time. The exact chance of falling was fairly arbitrary, I was just aiming for any given flake to wait on average 2 seconds before falling.

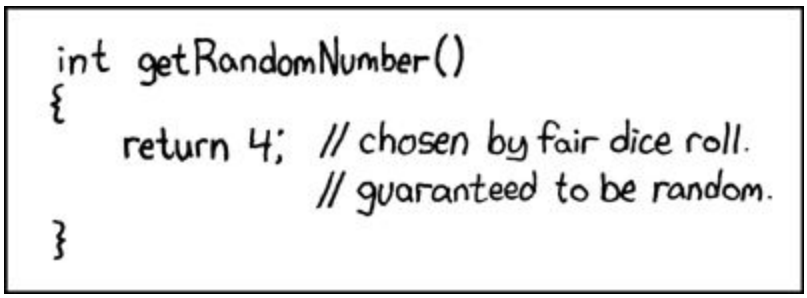
After this first state, the next two states last for one cycle each. The second determines the velocity of the snowflake (they can fall slowly or quickly, or somewhere in between) based on the value of the current random number. The third state determines where on the screen the

snowflake will start falling. This just selects some random position horizontally then applies that to the snowflake x-axis output.

In the fourth state, I update the snowflake's position for each frame update. This simply involves adding the value in velocity to the y axis position, then making sure that the snowflake is still in frame. Once the snowflake leaves the frame it is reset to the first state where the snowflake begins the whole cycle over again.

The other important section of the snowflake generation is the snowflake_renderer. This acts very similarly to the mouse renderer, in that it pipelines the vga timing signals and overrides the pixel signal as needed. I once again chose to manually implement these pixels because it was quicker to do than to create a .coe then deal with instantiating bram, etc., it also meant synthesis and implementation times were quicker.

Random.sv (Brandon)



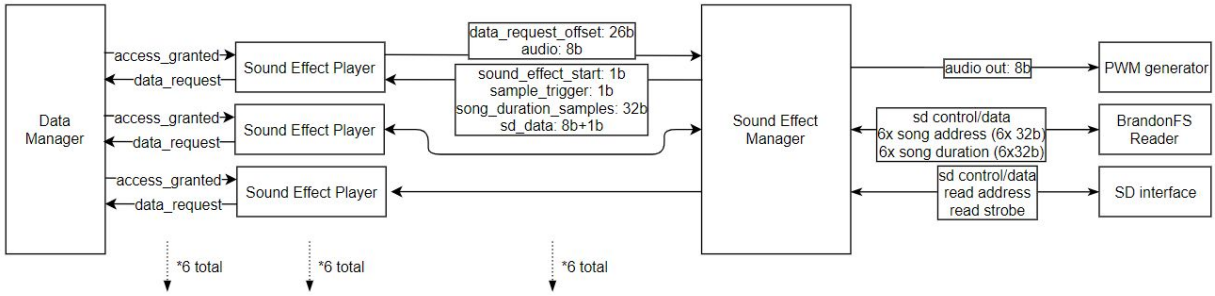
https://xkcd.com/221/

One major requirement for a game like this is that we needed a reliable source of randomness. Otherwise, every time we turned on the board the game would be the exact same. It also turns out that getting truly random numbers is quite difficult. We played around with a few different implementations, and ended up with a pseudorandom number generator (PRNG) linear-feedback shift register (LFSR). Pretty much any implementation we could pick would be a PRNG, though we did consider reading the LSB of an analog input connected to some noisy source - maybe the accelerometer or the temperature sensor. Unfortunately this did not create *enough* randomness fast enough, we would only get 1-2 bits at 1MHz max. To get around this limitation, we had one other trick up our sleeves: we were able to select the seed for our PRNG at random, then relying on the PRNG for the remaining calculations. This way we could use a computationally simple and deterministic PRNG that only had to be "good enough" at being random in the short term, and we could then set its seed based on some actual randomness. To further simplify we set the seed as a constant (`31'h4606ad21 [see https://xkcd.com/221/` for validation on this]), and then just started reading random numbers from this deterministic list at a non-deterministic time, namely when the user first clicked the game board. We would then repeat this process for every game generation based on when the user clicked a button. Even if a user could accurately time their first click after startup to within 100ms, they would still only have a 1/1,000,000 chance of getting the same board twice. We decided that for this project, this was more than acceptable for our purposes.

To actually implement the LFSR, I found a few whitepapers on the optimal taps for different size LFSRs. I picked a 31 bit LFSR, and then only used the lower 16 bits, giving our

numbers a bit more apparent randomness, and allowing the output numbers to hit each value more than once before the whole thing looped around. Then, according to https://www.xilinx.com/support/documentation/application_notes/xapp052.pdf, I picked bits 30 and 27 as my optimal taps. In combination with the randomness in timing from the user, we got an LFSR that ended up working really well.

Sound_effects.sv (Brandon)



The sound effects were by far my most massive undertaking for this project. Getting a single sound effect to play from a .coe was fairly simple (I had pretty much already done this with lab 5, sans the pre-filled COE). Getting multiple sound effects to play at the same time and with decent a decent fidelity and long length was much harder.

My final architecture involved loading our sound effects from an SD card with a custom filesystem (BrandonFS), six separate sound effect player modules (one set of code instantiated 6 times), a data manager to prevent concurrent access to the SD card, and a FileSystem interface that dealt with BrandonFS.

My first task was to figure out our system requirements. I quickly realized that I was definitely aiming to complete our concurrent sound effect stretch goal, so decided to not bother implementing a single .coe file version. Instead, I decided to jump straight to a dynamic multi-file SD card solution.

In order to use an external SD card with multiple sound effects, I needed a way to know where each song/sound effect started and ended in memory. The easy option would have been to hardcode these, but this would mean that the user could not update sound effects without redoing synthesis and implementation. I also knew that using a standard filesystem like FAT would be simultaneously complete overkill and also would be way too hard for a final project of this scope. So, I decided to write my own! I first determined some minimum requirements:

- Hold 6 files
- No file names, just index (0..5)
- Include start and length of each file
- All files sector-aligned (512 bytes)
- Header easily parsed
- 32 bit addressing

From here I created the following definition of my filesystem:

```
Outline:
Sector 0: Include all FS meta data
Sector 1: Free for future use.
Sectors 2..N: File 1
Sectors N+1.. : File 2
...


Sector 0 bytes:
0..15: ASCII "Brandon'sFS 1.0" + \x00
16..19: A for file 0
20..23: B for file 0
24..27: A for file 1
28..31: B for file 1
32..35: A for file 2
36..39: B for file 2
40..43: A for file 3
44..47: B for file 3
48..51: A for file 3
52..55: B for file 3
56..59: A for file 5
60..63: B for file 5
64..511: Don't care.


A: 32bit starting sector of file n. In bytes. Multiple of 512.
B: 32bit length of file n. In bytes, not necessarily a multiple of
512.

All multi-byte values are big-endian


Order of files per current Minesweeper implementation:
1. Music
2. Bomb
3. Win
4. Place flag
5. Music 2: Holiday.
6. Unplace flag
```

I also created a python script to auto-write the SD card, so that I didn't have to manually copy data around in HxD each time I needed to change something. This python script can be found in our github repo under "`custom_fs\fs_write.py`". To run, use a command like:

```
python fs_write.py Underclocked_mono.wav Music_Thing_mono.wav
Gentle_Rain_mono.wav Paper_Towel_beat_mono.wav
```

WARNING!! This script can corrupt your hard drive!!! Only run it if you are confident that you know what you are doing.

The script requires admin mode on Windows, though could be easily extended to work on other operating systems by changing the physical disk address. It assume physical disk 2, this can be changed manually. Check which disk you need using `` `wmic diskdrive list brief` `` on Windows, then modify the script appropriately. The script has some basic sanity checks, mainly that the targeted SD card has the correct "`Brandon'sFS 1.0`" magic number at the beginning.

With this filesystem plan out of the way, I needed to figure out how I was going to buffer data from the SD card on the FPGA. The first plan for this was to simply load all sound effects into the DDR ram memory located on the nexys board on boot, and then just grab samples as necessary. I came close to getting this set up, but then realized preloading all sound data was not necessary (see challenges/lessons learned for more details on this DRAM system). The new plan was to make a fifo using the fifo generator for every sound effect player, and then load data as needed from an SD card. This was possible because we only needed to play samples at 48KHz, and could read in 512-byte sectors to memory way faster than that.

This new fifo-based plan did have a significant flaw though - I only had 1 SD card, so I had to find a way to only read 1 sector at a time. To do this I used the provided sd_controller (sd_controller.v) to handle the SD interface, then created a "Data Manager" module which managed access to the SD card. It assigned a priority to every module that could talk to the SD card, and then only granted them access to the SD card one sector at a time. This then meant that every module requiring SD access had to have a "data_request" output signal. This signal told the data manager which modules needed more data, and let it appropriately allocate access to the sd interface.

The first module that needed access to the SD card was the brandon_fs_reader module. This module is the first to talk to the SD on boot, and reads in the first sector which contains all of the offsets and durations needed by the sound effect player modules. This one is fairly simple and tedious; I probably should have found a better way to write it as my final implementation was 190 lines long. All this module does is request to read Sector 0, parses each received byte one at a time and stores in the appropriate offset or duration register, then goes on to the next byte. After reading in this first sector the module disables itself and leaves the offset and duration register outputs constant until the next "RESET" pulse.

The most important module in this implementation is the sound_effect_player module. It takes data from the SD card as it becomes available and adds it to the fifo. It then takes out 1 sample at a time from the fifo at each sample_trigger strobe and outputs it to the audio output. It also manages all of the edge cases surrounding stopping and starting audio files, making sure they don't get out of sync, etc. The audio output is not signed since the source 8 bit audio files are not signed.

Once all six sound_effect_player modules calculate their current audio output, it is time to add them together. This was tricky because I needed to find a way to manually implement clipping in the case that the audio files had too large of an amplitude at any point. Just taking the average wouldn't work because this would result in audio being really quiet. I ended up just adding all 6 outputs together, then removing most of the DC bias. If the sum was less than 0x27B, I set the output to 0. If the sum was greater than 0x37A, I set the output to 0xFF. And in the remaining cases, the output was just the sum-0x27B. This worked because every sound output had a DC bias of 0x80, and therefore my final output would also have a DC bias of 0x80. A better way to implement this would have been to use signed numbers. Then I wouldn't have had to deal with all the weird limit cases and could have just checked if the sum was over 0x7F or under -0x80.

Challenges/Lessons Learned

One major challenge and simultaneous lesson learned that Brandon had to deal with was that projects that involve interfacing with pre-existing and not-clearly-documented modules are very hard. It is impossible to write a testbench when you don't actually know what the expected behaviour is supposed to be. Likewise, interfacing with some built-in hardware can be very difficult, such as DRAM. Brandon spent many, many hours researching how to use the MIG (Memory Interface Generator), how to find pinouts for the XDC file, etc. While he came very close to getting it all working, he decided to abort this option because the FIFO generator option was so much easier to use and didn't carry the baggage of undocumented interfaces. In reality, that DRAM task could have been a project on its own just from the background research perspective.

Another challenge Rod learned was that implementing recursion or really any process that takes an unknown amount of resources or steps is very challenging. When designing the "CLEAR" portion of the Minesweeper module, he had to find a way to step through all of the neighboring pieces. This lends itself to a recursive algorithm quite neatly, but this then required a FIFO to keep track of the steps. This came with plenty of its own challenges, such as data being added to the fifo not being available at the output on the next cycle, or just generally keeping track of what was in the fifo and what was not.

Finally, we both learned that pipelining VGA signals is incredibly important and also surprisingly difficult. We spent many hours debugging off-by-one or off-by-a-few errors. One caused the grid to be 3 pixel off from the rest of the screen, another caused the bombs to be rendered 4 pixels off in the other direction.

Future Work

If we were to continue this project, I would like to spend some more time refining the sound effect system. Can we get the fpga to detect when an SD card has been inserted? Handle hot plugging? Store the data in the DRAM so that the card *could* be removed at any point?

The “theme” system could also be extended, to perhaps have a rainy day with an appropriate backing track in addition to raindrops falling on the screen.

Another interesting thing to focus on would be extending the game to have new features, such as the multiplayer mode we originally proposed as a stretch goal.

We could also just start implementing more Windows XP built-in games, and make it a full on XP Game Emulator. Perhaps the next steps could be a few forms of solitaire?

Appendix

Github Link

https://github.com/wiresboy/6.111-Minesweeper