KA_BOOM

By Devin Murphy, Emma Griffiths, and Willie Zhu

Table of contents

Overview

Kaboom is inspired by the techy aesthetic of the FPGA as well as the game “Keep Talking and Nobody Explodes”. The objective of this game is to defuse the bomb before time runs out. There are various modules that must be disarmed, which the player can move between in any order they would like. If they are all disarmed before time runs out, then the bomb is defused. There is also a strike system. If you get three strikes (fail modules three times), then you will lose. The game is accompanied by a bomb defusal manual, which is purposely written in a non-straight forward manner to make defusing the bomb a bit more difficult.

Motivation

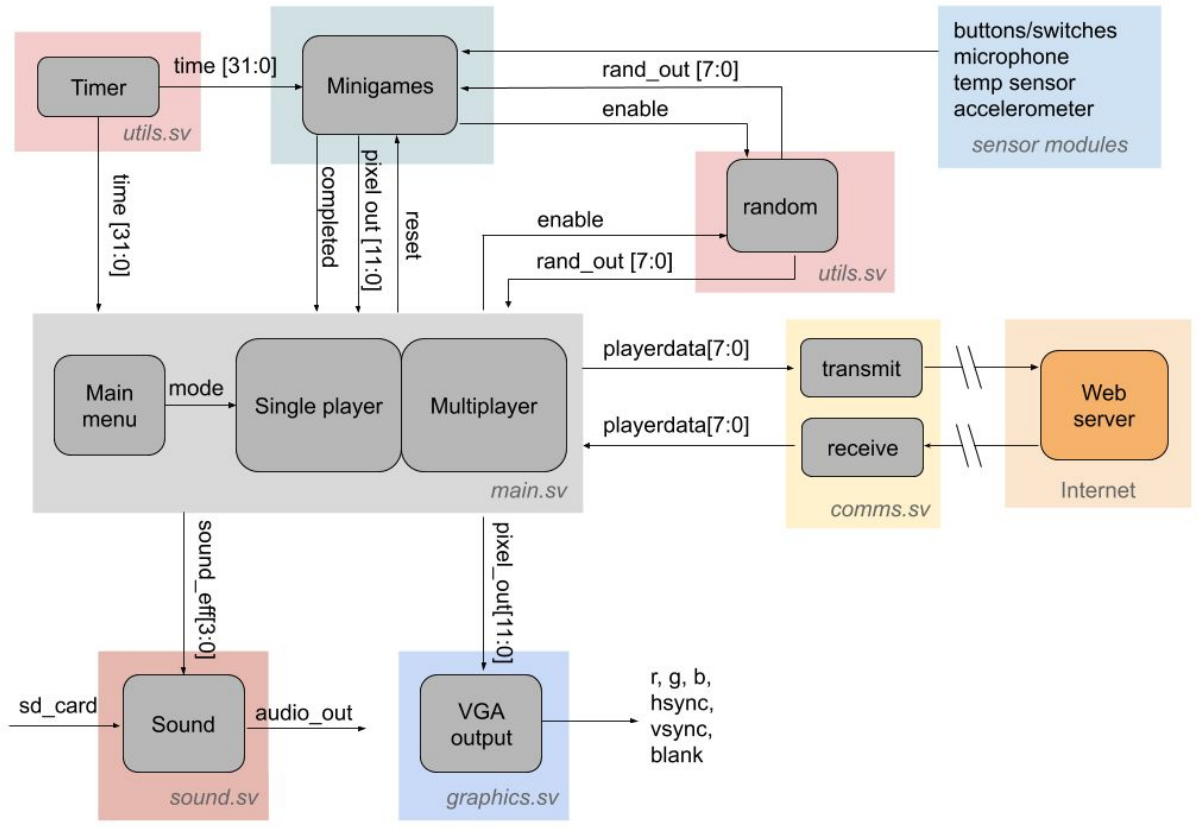
We decided that we want to make a bomb-defusing game where the FPGA itself acts as the “bomb”, where the various switches, buttons, and sensors built into the FPGA act as inputs working to defuse the bomb. This includes using the microphone, accelerometer, and the temperature sensor. We thought it would be valuable to learn how to interface with these sensors and process their outputs. The seven segment display also shows the countdown timer. We were also interested in how we could use a serial data transmission protocol to allow this game to be played in multiplayer mode over WiFi, especially because by the end of the semester we were all in different locations. We thought that it would be fun to design a game like this, between the bomb defusing manual, homescreen and end screen overall appearance, individual minigame module graphics, and sound. We all really enjoy playing these kinds of games together, and thought that making one like this would be a fun challenge!

Summary

Our team decided to make our own version of a bomb defusing game. Like in “Keep Talking and Nobody Explodes”, there is a series of modules that you must disarm following a bomb defusing manual’s instructions. In our version, the order of these modules will be in a predetermined, random order and appear one after another on the screen. All minigame modules also have elements of randomness. When a Defuser gets a strike, they will have to replay the module they are currently on. “Keep Talking and Nobody Explodes” is also a multiplayer game, in which one person will act as the Defuser and one will read the instructions from the manual without the Defuser being able to see them. We decided to add two modes to our game, both single player and multiplayer. In single player you race against the clock using the manual to defuse the bomb, and in multiplayer you race against another player to defuse the bomb faster using an esp32 module and websocket to connect the boards. The other player’s status will be displayed on your screen in multiplayer mode.

The graphics were displayed by a combination of using BRAM and storing on an SD card. The sound was also stored on the SD card. The external mic, internal accelerometer, and internal temperature sensor were all used in minigame modules.

Block Diagram



Modules

Game FSM (Devin)

module game_fsm(input clk_25mhz, system_reset, done_shuffle, my_sync, other_sync, mg_fail, mg_success, expired,play_again, input[2:0] i_op, input[1:0] strike_count_op, input [5:0][3:0] minigame_order_out, input[1:0] multiplayer, output logic [2:0] i, output logic [1:0] strike_count, output logic timer_start, start_shuffle, multiplayer_reset, mg_start, lose_start, win_start, output logic[4:0] minigame);

The game_fsm module controls much of the signals for the logic of the game itself, such as when to move on the next minigame, when to restart a minigame, the cases for winning and losing, syncing with the other player's fpga during multiplayer mode, etc. Here's a summary of the key states in the FSM:

SHUFFLE

- Initial state of the FSM. Start shuffling the order of the minigames and initialize i (the minigame success count and index for the array of minigames) and strike_count to 0. Then go to the HOME state.

HOME

- This is the state for the home screen of the minigame. Stay in this state until the minigames are done shuffling and the player has made a selection of which game mode to play in (either single player or multiplayer). If multiplayer is chosen, reset the start signal for the data transmission module. If single player is chosen, start the timer.

SYNC

- If both you and your opponent have flipped switch ten, your fpgas will be considered synced and your timers will start.

START

- State to choose which minigame is playing next, and to control the start signal for that minigame

MG_S

- Single Player mode gameplay. Progresses through the minigames in their shuffled order. If at any point the timer has expired then enter lose screen. If a minigame is failed and there are less than two strikes, start that minigame over and increase the strike count. Else if the minigame fails and there are two strikes, enter the lose screen. If a minigame is won and there are less than 5 minigames won, start the next minigame and increase the index for minigame selection. Else if there are 5 minigames won enter the win screen.

MG_M

- Similar to single player mode except the opponents progress and strike count are taken into account to determine if you have won or lost.

LOSE

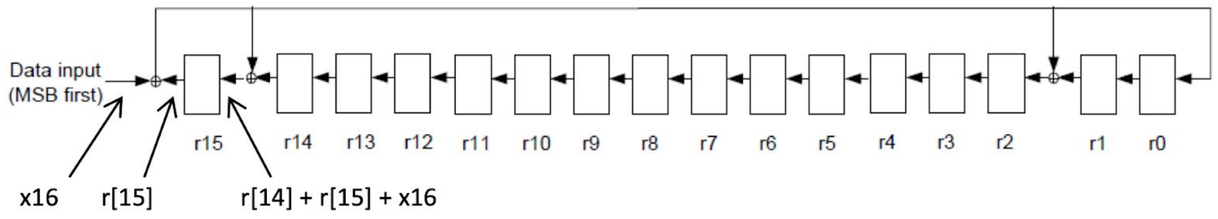
- Display lose screen. If switch 11 is flipped, start game over in shuffle state.

WIN

- Display win screen. If switch 11 is flipped start game over in shuffle state.

Random Number Generator (Devin)

```
module random_num( input clock, input data, output logic[15:0] random_number);
```



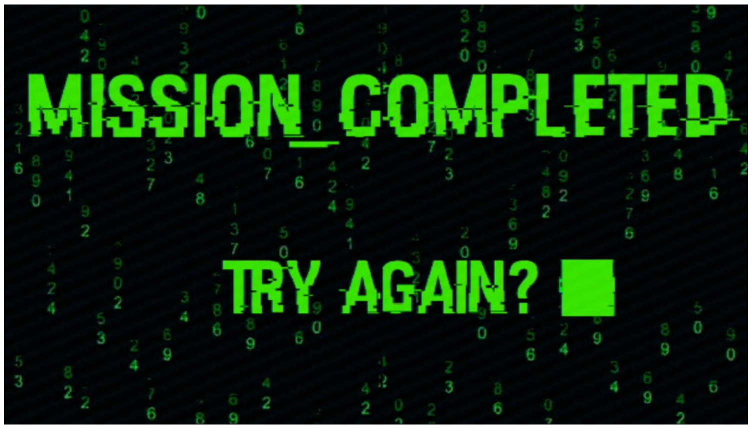
(crc generator image taken from 6.111 lpset 5 figure 3)

Using the crc we implemented in lpset 5, I created a pseudo random number generator. This takes in a high Z input as data and subsequently outputs a 16 bit pseudo random number according to the above generator every clock cycle.

SD Card Graphics (Willie)

Implementation details

A naïve method of storing graphics on the FPGA is to take up block RAM, which can be initialized from a bitstream, and store bitmaps for every image in the game. However, this approach is wildly inefficient, since even with color-mapping, an 8-bit image taking up the size of the entire screen would occupy almost half of the available BRAM on the device. This would prevent us from displaying more than a couple of graphics at most.



The player's win screen; this is drawn entirely by the graphics engine.

Since we wanted to have as many graphics on the screen as possible, as this was a core design requirement from the outset of the project, this required us to take a clever approach to storing graphics data. In the end, we settled on a texture atlas[1] and framebuffer approach,[2] which allows us much greater flexibility. Taking this approach, we are limited only by the available BRAM for the framebuffer and texture atlas; with clever design, we are able to draw an unlimited[3] number of graphics to the entire screen, while taking up less than 70% of the available BRAM.

The texture atlas/framebuffer are both implemented using Xilinx's Simple Dual-Port RAM generators; in particular, the read port of the texture atlas is exposed internally to the graphics engine, while the write port is exposed to the SD card interface. Similarly, the read port of the framebuffer is exposed externally to the VGA output, while the write port is exposed internally. By designing this read/write interface like this, each controller can remain agnostic about the other. This allows us to, for example, prioritize loading audio data over loading video data.[4] Since loading graphics can be accomplished slowly without loss of immersion,[5] but stopping an audio stream is extremely jarring, this ensures a better play experience.

For simplicity, we use a framebuffer as large as the screen itself (640 by 480 pixels), and a texture atlas slightly smaller (512 by 256 pixels),[6] each 8 bits deep. Since the texture atlas cannot hold a texture as large as the screen itself, we render backgrounds by drawing four 320 by 240 textures, one for each corner of the screen, while swapping out texture atlases between draws.

Additional RAMs are required by our current implementation to store details about each individual object to be rendered by the graphics engine. For example, object memory consists of 36-bit words that store information about the x-position, y-position, ID, and flags (currently used to turn on and off rendering for an object).

x position (9 bits)	y position (9 bits)	flags (10 bits)	atlas id (6 bits)

Similarly, we need a way to describe where each texture is located on the atlas itself. These properties are stored in 64-bit words as follows:

x position (16 bits)	y position (16 bits)	width (16 bits)	height (16 bits)

[1] Referred to as "texturemap" in the source, although we have avoided this nomenclature in this document for the sake of clarity.
[2] This is an extremely common technique. See https://en.wikipedia.org/wiki/Texture_atlas.
[3] Limited only by the storage capacity of our SD card, which was 2GB, as well as performance limitations with swapping out texture atlases.
[4] See SD card for more details.
[5] As demonstrated by numerous loading screens in almost every video game.
[6] Yes, we chose these sizes as powers of two to simplify our code.

As a compromise between BRAM utilization and performance, we chose to use an 8-bit framebuffer and texture atlas, even though the Nexys is capable of outputting 12-bit color. We settled on this compromise for the sake of simplicity and development time.

Usage details

The graphics engine exposes a number of inputs relating to loading texture atlases and rendering objects. When loading in a texture atlas, it exposes a selector for choosing the atlas index,[7] as well as an input that loads the selected texture atlas into BRAM when asserted. When finished loading, this module asserts `texturemap_load_ack`.

To render objects, the module exposes a SRAM-like interface to its internal object memory (`new_object_waddr, new_object_we,` and `new_object_properties`). It also exposes two flags: the first, `render_dirty`, signals whether the framebuffer should be cleared (set to 0x00) entirely before rendering the scene; the other, `should_render`, starts a full-scene render when asserted.

SD Card Sound (Willie)

Implementation details

Sounds are stored on the SD card[8] and played back through the sound engine. The sound engine consists of a single FIFO (designed to use built-in FIFO primitives instead of BRAM, in order to conserve resources). The write port of this FIFO was exposed to the SD card controller, while the read port was exposed internally and fed into a PWM generator. To prevent buffer underruns, we used a 2048-byte FIFO (8 bits per byte). Accordingly, we also used unsigned 8-bit PCM wav files to store sound data. We also included an empty signal that triggered at a fill level of 50%, and made the SD card controller read in fresh data whenever this signal was asserted. This ensured that we always had a sufficient amount of audio data buffered in the FIFO.[9]

Sounds were played back at 44.1 kHz, and passed through a Sallen-Key Butterworth Low-pass 4th Order Filter. The frequency response of this filter has a knee at approximately 20 kHz, which is well-suited for audio purposes.[10] In order to modulate the amplitude output of the filter, we used a PWM clock equal to the system clock frequency, which gave us ample headroom for an 8-bit DAC.

Usage details

The sound controller exposes three inputs: a play signal, a stop signal, and a sound ID selector. When the play input is asserted, the controller latches the current sound ID value,

[7] Due to the way we store these atlases, there is a **hard limit** of 240 atlases. In our final build, we used approximately 20.
[8] For more information, see SD card.
[9] Since the SD card stores data in 512-byte blocks, we found it easiest to stick with a multiple of this size.
[10] For reference, the upper range of human hearing is approximately 28 kHz, although this is often rounded down to 20 kHz in audio applications.

then starts reading into its FIFO. When the stop input is asserted, the controller flushes its FIFO and stops reading from the SD card.

SD card controller (Willie)

Implementation details

In order to store graphics and audio data, we used the built-in SD card slot on the Nexys 4 DDR board. SD cards expose a simple SPI interface, which we used to read blocks of 512-byte audio and video data into BRAM.

During the initialization process, the controller first read in the audio header, which was stored in the first 512-byte block of the SD card. This header includes offsets for the different sounds in the game, which were used to calculate where the SD card should read from whenever the sound engine was used. Because of the limited size of the audio header, we have a hard limit of 64 sound samples.[11]

After reading in this header, the SD controller arbitrates requests from the graphics and sound engines (giving preference to the sound engine, because audio interruptions are very apparent, while visual interruptions are much less noticeable). When a request is taken off the queue, the SD controller begins a full block read; after reading in the entire block, the controller sends an acknowledge signal to let the other module know it is finished.

In order to produce a hex file that can be written to the SD card and read from the FPGA, we used a simple python script to package and link all the assets together. The first `0x20000` bytes were reserved for the audio and graphics headers. The audio header has already been described above; for the graphics header, each 512-byte block following the first one contains details about the objects contained in each texture atlas (specifically, the x- and y-positions of the upper left pixel of each texture, as well as its width and height). While the audio header is only read once during initialization, each graphics header must be loaded when the corresponding texture atlas is loaded into memory. Due to the size of each graphics header, there is a hard limit of 240 texture atlases, though this can easily be extended by increasing the data offset for the first texture atlas.[12]

Graphics (All)

FPGA and Strikes

module FPGA_graphics(input vclock_in, input reset_in, input [10:0] hcount_in, input [9:0] vcount_in, input [2:0] mg_completed, input [1:0] strikes, output logic [11:0] pixel_out);

[11] In our final build, we used five sound samples.
[12] In our final build, we used around 20 texture atlases.

This module takes in the strike count and the number of modules completed, and generates a cartoon FPGA board in the top left with circles that turn green when a module is completed. It also generates the strikes as red circles in the bottom right corner. We used a pipelined circle_blob module to create these graphics. To draw the opponents FPGA in multiplayer mode, we just change the mg_completed input and the location of the FPGA cartoon. The FPGA cartoon is drawn using COE files and ROMs, similar to the method we used in lab 3.

Networking (Willie)

Implementation details

To connect the FPGAs to the internet, we use ESP32s which are configured to relay information sent from the user's FPGA over a serial TX connection, and pass through messages received from a remote web server over an RX connection. Every time the player completes a module, receives a strike, or completes the game, the FPGA sends this information to the ESP32. Similarly, whenever an opponent completes a module, receives a strike, or completes a game, that serial message is routed to every other FPGA, updating the displays accordingly.

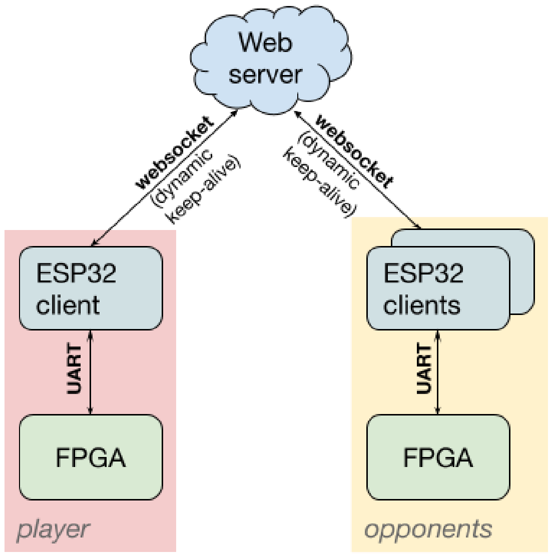


Diagram of ESP32 connections.

The ESP32s maintain a websocket connection with a Node.js server running Express. Importantly, the ESP32s are required to respond to regular heartbeat packets sent from the server; if they fail to reply within 5 seconds, the websocket is terminated. This ensures that stale connections are automatically closed.

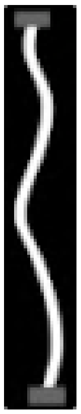
Note that this design is entirely agnostic of the number of clients connected; when an ESP32 requests a broadcast, its message is sent to all other connected ESP32s. This means that n-player multiplayer is possible; the ESP32 simply needs to be updated to consider, for example, only the maximum number of modules completed. In this case, the opponent's FPGA display on the screen would match that of the player closest to winning. The ESP32 would also have to count, for instance, the total number of other players who have failed, waiting for all (n - 1) players to fail before sending the message that the current player has won. Due to the flexible design of the web stack, this modification is trivial. This implies that a 6.111-wide game of Kaboom, one where every student of the class would participate, is possible.

Minigames

Wire Cutting (Emma)

```
module wire_cutting (input clock_in, input reset_in, input [10:0] hcount_in, input
[9:0]  vcount_in, input hsync_in, input vsync_in, input blank_in, input [5:0]
cut_switch, input [2:0] incolor1, input [2:0] incolor2, input [2:0] incolor3, input
[2:0] incolor4, input [2:0] incolor5, input [2:0] incolor6, output logic completed,
output logic failed, output logic phsync_out,   output logic pvsync_out, output logic
pblank_out, output logic [11:0] pixel_out);
```

This module takes in 6 3 bit numbers from the randomizer, one for the color of each of the 6 wires. These three bit numbers are mapped to colors via case statements. The instructions to disarm this module are based on the first wire color and the number of wires of that color. To count the number of wires that were the first color, I wrote a short module that takes in two input colors and returns 1 if they are the same, and 0 otherwise. I then put all the combinations of the first wire and others through this module ((1,2), (1,3), (1,4), (1,5), (1,6)) and added the results + 1 (to account for the first wire). I then used an always_comb to assign the wire that the user needed to cut for each first wire color / number of wires of that color combination. If the defuser cuts the correct wire (flips the correct switch), then an always_ff block will signal that the module was completed, while if the user cuts a wrong wire, the always_ff will signal that the module was failed. If no wires are cut, the module remains in the playable state.



The graphics for this module were based on one wire image, where the wire itself was white:

This image was stored in BRAM using COEs generated from this python script. Once the PNG/JPEG you want to use is in the same folder as the script, you can run "python image_to_coe.py "ImageName.FileType". The COE files will then be generated and added to the folder. The names the files save as can be altered in the python script directly.

The wire was intentionally made white so that the module that stored the wire image could take in a wire input color, then check for pixels in the image that were white and replace them with the input color. This made it easy to generate wires of multiple random colors without taking up too much BRAM.

A delay was also added between when the Defuser cut the wire, and when the completed/failed signals were updated. This was to allow time for the user to see that the graphics update and cut wires turn black. This was mostly just to make the module slightly more immersive and so that a new module didn't start immediately after flipping a switch.

Uncovered Button (Emma)

```
module button_game (input clock_in, input reset_in, input [10:0] hcount_in, input
[9:0]  vcount_in, input hsync_in, input vsync_in, input blank_in, input confirm, input
[1:0] rand_button_color, input rand_strip_color, input rand_has_text, input
pushed_button, input [3:0] ones, input [3:0] tens, input [3:0] minutes, output logic
failed, output logic completed, output logic phsync_out, output logic pvsync_out,
output logic pblank_out, output logic [11:0] pixel_out);
```

This module takes in one 2 bit number, and two 1 bit numbers from the randomizer. The two bit number assigns the button to a color using a case statement, and one of the 1 bit numbers is used to assign a color strip under the button to one of two colors. The last 1 bit random number is used to determine whether the button will be blank or have text (say "Hold"). This module also takes in the minutes, tens, and ones position of the timer. The instructions to disarm this module are based on all of the randomized inputs. Depending on the button color, strip color, and whether the button has text the user must press the pushed_button when there is a certain number in the timer and release when there is a different number in the timer.

A few always_ff blocks were used to keep track of the times the user pushed/released the button. Upon reset, a previous_button logic was set to the input pushed_button value. A counter was used, and once it got to 1_000_000 clock cycles, I checked to see if the previous_button was 0 and the current pushed_button was 1. If this was true, I stored the inputted timer's minutes, tens, and ones as the pushed minutes, tens, ones. I also did this in reverse (checking if the button was previously 1 and now 0 after 1_000_000 clock cycles) to store the timer's values when the button was released. The count to one million was added as an additional debounce precaution and to ensure that I was actually seeing a transition from not pushed/pushed/released.

The game logic was written in an always_ff. There was a different push/release button time condition for each color/strip/text combination. If the Defuser pushed and released the button at the correct times according to the rules, then the always_ff will signal that the module was completed. If the user does not push or release the button at the correct time, then a failed signal will be raised.

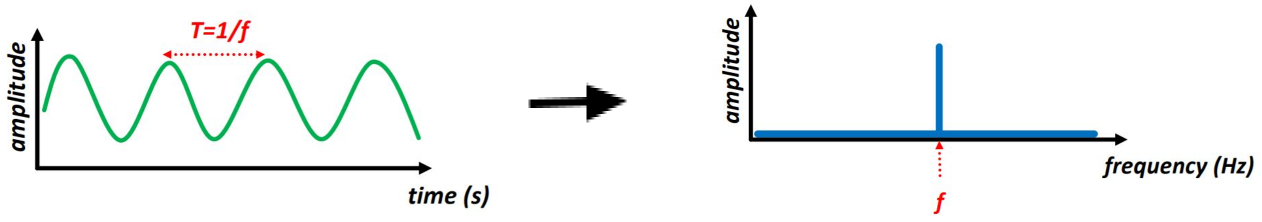
The graphics for this module were a combination of blobs and BRAM stored COEs. The button with text was stored in BRAM as a white button with black text. The same method that was used with the wires was used here to change the button color as well depending on the random color chosen. The blank colored button was created using a circle blob module that takes in a radius, color, x and y top left corner position and displays a colored circle blob. The color strip was a rectangular blob.

Microphone/Covered Button (Emma)

```
module micr_minigame( input  clock_in,  input reset_in, input [10:0] hcount_in,  input
[9:0] vcount_in, input hsync_in, input vsync_in, input blank_in, input button, input
[23:0] sqrt_data, input sqrt_valid, input sqrt_last, input [2:0] minigame_number,
output logic completed, output logic failed, output logic phsync_out, output logic
pvsync_out, output logic pblank_out, output logic [11:0] pixel_out);
```

The microphone/covered button minigame was played by using the external microphone. Most of the microphone audio data processing was done in the top level with the 100mHz clock, and just the final outputs that the game logic was based off of (sqrt_data, sqrt_valid, sqrt_last) were inputted into the actual mic minigame module.

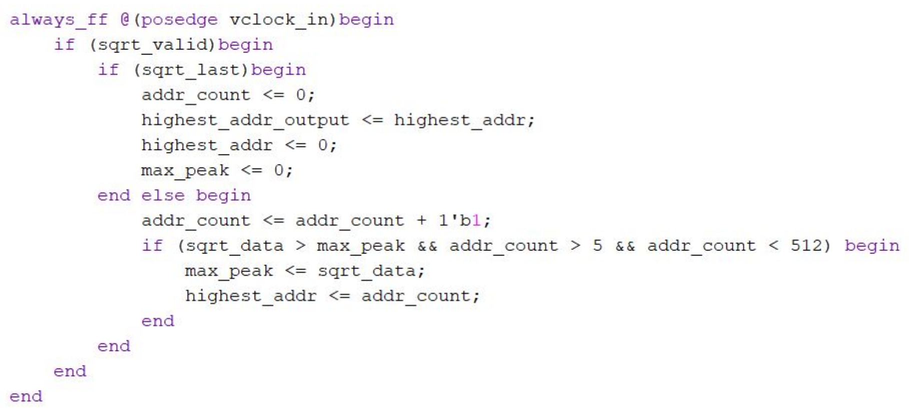
First, the audio was put through an analog-digital converter IP (ADC), similar to what was done in lab 5a. This measures the microphone output voltage and converts to a digital signal. This data was then put through a Fast Fourier Transform IP (FFT) to convert the time-domain signal into its frequency domain representation, as shown below.



(images taken from 6.111 lecture 10, slide 31)

A LAST signal from the audio sampler tells the FFT when we are at the end of a frame (on the last sample). An fft_ready signal is also used to tell when the FFT is ready for a sample. The FFT outputs a 32 bit complex number, where the first 16 are the real component and the last 16 are the imaginary. These components were split, squared, and summed ($Re(X)^2+Im(X)^2$). Another IP (CORDIC IP, for general mathematical operations) was used to take the square root and calculate the magnitude of the frequency components. Between the split/square/sum and CORDIC IP, a First-In-First-Out IP (FIFO) was used as an extra precaution that held data between the two just in case they could not process data at the same rates and to resolve short data buildups. It was these signals from the CORDIC square root IP (sqrt_data, sqrt_valid, sqrt_last) that were input into the actual game logic module. Joe goes through this IP pipeline in lecture 10 and in this source code. The code is also very well commented and tells you exactly how you need to implement each IP. He used a BRAM as well to store the data and to display the data as a bar graph, but since I didn't need these things for my purposes I didn't end up using that part of the code.

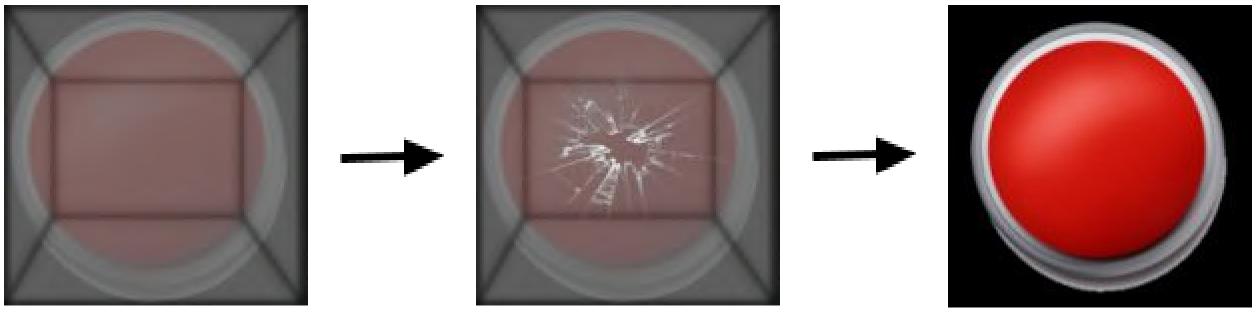
The module begins by displaying a glass covered button on the screen and a rectangular color strip to the right. The user must hold a correct note long enough to shatter the glass and push the button. The graphics also update when the note is held long enough, showing the covering cracked, and then gone (all these images created in inkscape and stored in BRAM via COEs). Once the cover is gone, the user can press the corresponding button on the FPGA to complete the module. The instructions to disarm this module are based off of which number module it is. Depending on which module it is, a set of three color possibilities will be assigned to a rectangular blob color strip to the right of the button in a case statement: one to show that the user is below the necessary frequency, one for within the frequency range, and one for above the necessary frequency.



```
always_ff @(posedge vclock_in)begin
    if (sqrt_valid)begin
        if (sqrt_last)begin
            addr_count <= 0;
            highest_addr_output <= highest_addr;
            highest_addr <= 0;
            max_peak <= 0;
        end else begin
            addr_count <= addr_count + 1'b1;
            if (sqrt_data > max_peak && addr_count > 5 && addr_count < 512) begin
                max_peak <= sqrt_data;
                highest_addr <= addr_count;
            end
        end
    end
end
```

An always_ff was used to determine which microphone input frequency corresponded to the highest amplitude (as seen to the left). As long as sqrt_valid is asserted, we continue. We go through the samples, one address at a time, keeping track of the addr_count (through playing around with the module, I've learned that each delta 1 address corresponds to about 24Hz). I check to see if the amplitude at that address is the new max amplitude, if the address corresponds to a frequency above ~120Hz to minimize low frequency noise, and if the address is less than 512 (since this is halfway through the samples and after halfway we just see a reflection). I update the max amplitude and corresponding address as needed. Once we get to the end of the samples (sqrt_data is asserted) is when I finally output the highest amplitude address and reset the rest to zero.

This highest amplitude address was compared to a predetermined desired address (that I made sure was in voice range), and if it was within 1, a count started. If it remained in range long enough, the glass broke, then was removed, and the user could press the necessary button to complete the module.



Temperature Sensor/Fingerprint Scanner (Devin)

```
module minigame_1( input vclock_in, input reset_in, input [10:0] hcount_in, input
[9:0] vcount_in, input [1:0] random,input [3:0] sw, input btnu, btnl, btnd, btnr,
input vsync_in, input [12:0] temp_in, output logic [11:0] pixel_out,output logic
success, fail);
```

The first step to getting this minigame working was integrating the onboard temperature sensor. Luckily Gim put together a demo project using the temperature sensor to display the temperature on the 7 segment display. I used the VHDL code from this project to wire up the temperature sensor to our top level, and then pipe the 13 bit output to this module. The temperature sensor was able to run at 25 Mhz, our system clock, so I just changed the input clock frequency parameter in the VHDL.

This minigame has 3 square blobs. The first changes color after your finger increases the temperature output by 12’d4, after which you must take your finger off and wait for the sensor to cool down. I chose 12’d4 by using the ILA to look at the output of the fingerprint scanner and seeing it’s reaction to leaving my finger on the scanner for various amounts of time. When the sensor cools down by the same 12’d4, the second square should change color, and you have to put your finger back on the temperature to change the color of the third and final square. To complete your “fingerprint scan”, you must use the color of this last square to determine what combination of switches and buttons you must input.

(For reference, Gim’s code: https://web.mit.edu/6.111/volume2/www/f2020/temperature/)

Accelerometer (Willie)

Implementation details

The Nexys 4 DDR has an onboard Analog Device ADXL362 accelerometer. In our design, an abstracted accelerometer control module outputs the current orientation/pose of the board itself. Minigames can read the current X/Y pose of the Nexys 4 DDR board, which can be in one of five positions: centered (flat on a surface), or tilted to the right, left, forwards, or backwards (we treat having the FPGA upside down as being in the center position, since it is unreasonable to ask the player to turn the FPGA upside down).

To read data from the accelerometer IC, we use an SPI connection clocked at 1 MHz. Since data from the onboard accelerometer can be quite noisy, we additionally pass pose data through an IIR filter with a decay factor of 0.125. In particular, we use the difference equation

$$y[n] = (1-d)\cdot x[n] + d\cdot y[n-1],\ d = 0.125$$

where y[n] is the filter response at timestep n. This gives us a low-pass filter with cutoff frequency and improves our noise immunity.

$$f_c = \frac{-ln(d)}{2\pi}$$

Usage details

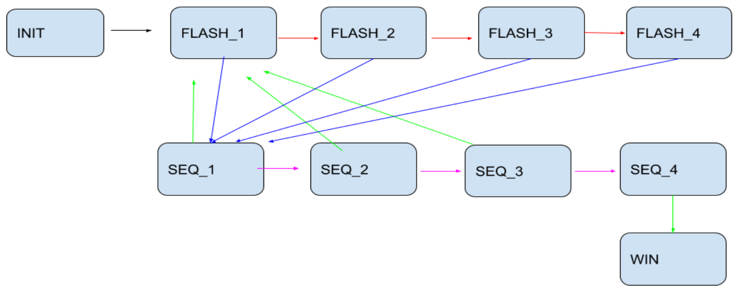
This output produces one output, which is an enum set to the FPGA board's current orientation (either `center`, `top`, `bottom`, `left`, or `right`). In our current implementation, this orientation is used in the Simon Says minigame. In lieu of using the buttons, the player simply tilts the entire board in the right direction.

Simon Says (Devin)

```
module minigame_2( input vclock_in, input reset_in, input [10:0] hcount_in, input
[9:0] vcount_in, input [1:0] random, input btnu, btnl, btnd, btnr, input vsync_in,
output logic [11:0] pixel_out, output logic [11:0] timer_count, output logic led_r,
output logic  led_g,output logic  led_b, output logic [3:0] state,output logic
success, fail );
```

This was inspired by an actual minigame in “Keep Talking and Nobody Explodes”. A random number will determine a random sequence of 4 blinking led colors. The first color will blink, and then the user has to tilt the accelerometer in the correct direction associated with that color. If this is correct, then the first color will blink again, followed by the second color. The user continues inputting the correct directions using the accelerometer until they correctly input the whole sequence of 4 blinks.

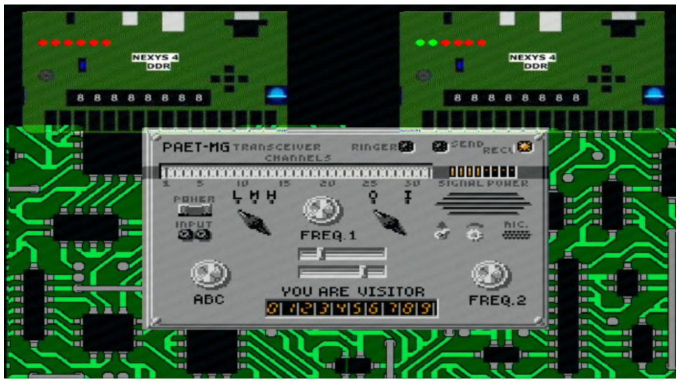
The FSM for this minigame was a bit complicated at first. I was unsure of how to recreate this effect of the FPGA sort of “remembering” the order of LED blinks and also waiting for user input after 1 blink, then 2 blinks, then 3 blinks etc. The way I resolved this was by having a “FLASH” state and a “SEQ” state for each light, as well as a counter that counted every correct user input. This way, when I am in the FLASH state for light 1 for example, I can determine whether to flash the next light (entering the FLASH state for light 2) or wait for user input based on how many correct user inputs there have already been (entering the SEQ state for light 1).



Assuming the player always entered the correct sequences, the above FSM would enter states as follows: INIT-> FLASH_1 -> SEQ_1 -> FLASH_1 -> FLASH_2 -> SEQ_1 -> SEQ_2 -> FLASH_1 -> FLASH_2 -> FLASH_3 etc.

Morse Code (Willie)

In the Morse code minigame, players are tasked with decoding the Morse code displayed on the screen via a blinking LED; after matching the decoded word with an entry in a “decryption table” found in the manual, they are tasked with sending a particular binary number (as determined by the switches on the bottom of the board) and “transmitting” it by pushing a button.[13]



The Morse code minigame in action

[13] Like most of the other minigames in our project, this was also inspired by KTANE, which features an almost identical module. Our codewords were, however, heavily inspired by 6.111 ;)

A mapping from ASCII to Morse code for alphabetical characters is accomplished with a small lookup table. For simplicity, we store the code words themselves in a single port ROM, initialized by a .coe file included with the module itself. The implementation of the morse code itself is largely trivial; refer to the source code for details.

Challenges/Difficulties

When we started out the project, we used a 65 Mhz clock and kind of just hoped that using 1024 x 768 graphics would be fine on our FPGA's memory. When we realized that using graphics of this resolution would mean we couldn't have as many detailed images as we hoped (and also took up too much of the BRAM), we had to redesign our system around the 25 MHz clock for 680 x 480 graphics. This meant that we had to resize a lot of our images and generate new COEs.

Potential Improvements

One thing we would have liked to do if we had more time is stored the player's fastest time on the SD card and include a text renderer that could be used to display the time during the end screen. We would also like players to be able to choose an order to solve the modules in, rather than just displaying them back to back in a predetermined (random) order. This would require a much more complicated FSM. We would also like to expand multiplayer mode to more people and also incorporate sabotage elements (such as making your opponents screen go black for 5 seconds). Adapting multiplayer mode to work with 3+ players instead of 2 would require some method of identifying which ESP32 sent which signals, as right now the server is entirely agnostic.

Right now our graphics are half implemented using the ROM and COE method and half implemented using the SD card and texture loaders. Ideally, we'd like to convert all of our graphics to be on the SD card so that the graphics pipeline is the same for all images. Other potential improvements to the graphics design might include the use of colormaps, which would allow us to reach the full 12-bit color capability of the Nexys board while keeping BRAM usage low. However, the design would need to either associate each pixel with a particular colormap, which is memory-intensive enough that it might not actually be an improvement. Alternatively, one could shrink the texture atlas size and increase the bit depth for the framebuffer, though this would have negative performance implications, since smaller texture atlases mean more frequent atlas swaps.

Other potential improvements could include the integration of DDR memory, which would allow larger framebuffers, as the available video memory would be orders of magnitude larger.[14]

[14] In the case of the Nexys 4 DDR, this size is 128MiB.

Another improvement is the use of double-buffering, which is a standard practice in game development. In a double-buffered system, one framebuffer is selected to be displayed at a time, while the other framebuffer can be drawn to. Then, whenever a vsync occurs, the framebuffers are swapped, and rendering for the next frame can begin. This reduces “tearing,” which occurs because a framebuffer is being drawn to and displayed on the screen at the same time.

Finally, due to a lack of development time, we did not add an output enable to the graphics engine itself. This would have been useful when drawing backgrounds, since the user would see each corner rendering sequentially. By turning off the output until this background drawing is finished, all the user sees is a black screen until the background is fully rendered. This would probably look a little nicer.

Conclusions and Advice

As tedious as it may seem, we recommend trying to seriously calculate how much memory your system is going to need from the get go. We realized a lot of other system design choices needed to be based around this, and it resulted in some lost time down the line when we had to change numerous parts of our system to increase memory availability. In addition, we recommend trying to become familiar with git and start integrating early on in the process, especially if you are working remotely. Integrating our modules took a lot of time, and vivado doesn’t seem to be very git friendly.

Sources

All source code for this project can be found here:
https://github.com/devinmur29/kaboom_project

KA_BOOM

BOMB DEFUSAL MANUAL

Version F20

Verification Code: 6111

Defusing Bombs

A bomb will explode when the countdown timer reaches 0:00 or when three strikes have been recorded. The only way to defuse the bomb is to disarm all of its modules before its countdown timer expires.

Modules

Each bomb is composed of 6 random modules that must be disarmed. The modules must be solved in a predetermined order.

Instructions for disarming modules are found later in the manual.

Modules will appear one at a time in a predetermined random order. If a module is failed and there are less than 3 strikes recorded, the module will restart and a strike will be added.

All modules must be disarmed in order to defuse the bomb.

Strikes

When you make a mistake, a strike will be recorded and displayed on the screen and the current module will continue. The bomb will explode once the third strike has been recorded.

Multiplayer

In multiplayer mode, you will race against another Defuser to defuse the bomb in the shortest amount of time.

The status of each Defuser will be displayed on the screen. The game is over once someone finishes defusing.

On the Subject of Wires

One wire, two wire, red wire, blue--hold on, how many are there again?

These wires are carrying detonation signals between components of the bomb. But which ones will trigger the bomb, and which ones are harmless? Looks like you'll have to find out.

- **Only <u>one</u> wire needs to be cut to disarm the module.**
- **Wire ordering goes left to right, and starts from switch 9.**
- **To cut a wire, flip the corresponding switch.**

If the first wire is...

Red:	**Green:**	**Blue:**
1) If there are four red wires, cut the third wire.	1) If there are five green wires, cut the third wire.	1) If there are six blue wires, cut the fifth wire.
2) If there are two red wires, cut the fifth wire.	2) If there are three green wires, cut the second wire.	2) If there is one blue wire, cut the third wire.
3) If there is one red wire, cut the sixth wire.	3) If there are six green wires, cut the sixth wire.	3) If there are four blue wires, cut the sixth wire.
4) If there are six red wires, cut the first wire.	4) If there is one green wire, cut the first wire.	4) If there are three blue wires, cut the first wire.
5) If there are three red wires, cut the fourth wire.	5) If there are four green wires, cut the fifth wire.	5) If there are two blue wires, cut the fourth wire.
6) If there are five red wires, cut the second wire.	6) If there are two green wires, cut the fourth wire.	6) If there are five blue wires, cut the second wire.
Yellow:	**Purple:**	**Gray:**
1) If there are six yellow wires, cut the second wire.	1) If there are four purple wires, cut the second wire.	1) If there are five gray wires, cut the first wire.
2) If there is one yellow wire, cut the sixth wire.	2) If there are two purple wires, cut the first wire.	2) If there are three gray wires, cut the fifth wire.
3) If there are four yellow wires, cut the fourth wire.	3) If there is one purple wire, cut the fifth wire.	3) If there are six gray wires, cut the second wire.
4) If there are three yellow wires, cut the third wire.	4) If there are six purple wires, cut the third wire.	4) If there is one gray wire, cut the fourth wire.
5) If there are two yellow wires, cut the fifth wire.	5) If there are three purple wires, cut the sixth wire.	5) If there are four gray wires, cut the third wire.
6) If there are five yellow wires, cut the first wire.	6) If there are five purple wires, cut the fourth wire.	6) If there are two gray wires, cut the sixth wire.

On the Subject of the Uncovered Button

This won't be as easy as the ones you'll find at Staples.

An uncovered button will appear in the center of the screen. To disarm this module, follow the instructions in the order they are listed. Once the button is pressed, a colored strip will appear below that may include necessary information regarding disarming this module.

1. If the button is white and has no text, press it when there is a 7 in any position of the timer.
2. If the button says "Hold" and is green, press it at any time.
3. If the button does not say "Hold" and is blue, press it at any time.
4. If the button has text and is blue, press it when there is an 8 in any position of the timer.
5. If the button is not blank and is red, press it when there is a 6 in any position of the timer.
6. If the button is blank and is green, press it when there is a 9 in any position of the timer.
7. If the button doesn't say "Hold" and is red, press it any time.
8. If the button has text and is white, press it at any time.

Releasing a Held Button

1. If the button has text, is white, and has a yellow colored strip, release it when there is a 3 in any position of the timer. Otherwise, if the colored strip is not yellow, release when there is a 7 in any position.
2. If the button says "Hold" and is red, release it when there is a 3 in any position of the timer.
3. If the button is not blank and is blue, release it when there is a 4 in any position of the timer.
4. If the button has text, is green, and has a cyan colored strip, release it when there is a 2 in any position of the timer. Otherwise, if the colored strip is yellow, release it when there is a 5 in any position.
5. If the button is blank, is red, and has a yellow colored strip, release it when there is a 4 in any position of the timer. Otherwise, if the colored strip is cyan, release it when there is a 2 in any position.
6. If the button does not say "Hold" and is green, release it when there is a 5 in any position of the timer.
7. If the button has no text, is blue, and has a cyan colored strip, release it when there is a 5 at any position in the timer. Otherwise, if the colored strip is not cyan release it when there is a 1 in any position.
8. If the button is blank and is white, release it when there is a 2 in any position of the timer.

On the Subject of the Covered Button

Seems like it's time for your grand aria. I hope you've warmed up your singing voice, though, because this is no time to blow it...

A glass covered button will appear in the center of the screen. Using force to break the glass will cause the bomb to explode, so you need to instead use your voice to shatter it. A colored strip will appear to the right of the button that provides feedback as to what frequency you need to feed into the mic to shatter the glass. Once you hold the correct frequency long enough, the glass will shatter, and you must push the uncovered button to disarm this module.

1. If this is the third module, a blue strip means you are within the correct frequency range, while yellow means you are too high and green means you are too low.
2. If this is the second module, a yellow strip means you are below the correct frequency range, while a red strip means you are too high and a cyan strip means that you are just right.
3. If this is the fifth module, a purple strip means that you are above the correct frequency range, while a white strip means you are too low and a red strip means you are within range.
4. If this is the first module, a red strip means that you are below the correct frequency range, while a green strip means that you are within range and a blue strip means that you are too high.
5. If this is the sixth module, a white strip means that you are within the correct frequency range, while a green strip means that you are too high and an orange strip means that you are too low.
6. If this is the fourth module, a cyan strip means that you are above the necessary frequency range, while a purple strip means that you are too low and an orange strip means that you are within range.

On the Subject of the Fingerprint Scanner

This module prevents us from disarming the bomb by robot. Guess there's some things we fleshy sacks of meat do better, after all.

Defusing the bomb requires using your fingerprint to scan in, but you have to do it right. You need to be sure that you don't scan for too long and that you enter the correct passcode.

First, turn off switches 0-3. Then, place your finger on the fingerprint scanner (temperature sensor). You will see three squares on the screen. Once the first changes color, take your finger off the scanner. Then, wait for the second square to change color. Finally, place your finger back on the scanner and wait for the third color to change. You will need to take note of this color and look it up using the rules below. You must first flip a switch and then press two buttons.

Flipping switches:

1. If the color is not white, red, blue, cyan, or green, flip switch two.
2. If the color is purple, blue or red, flip switch three.
3. If the color is not cyan, blue, purple, yellow, or red, flip switch zero.
4. If the color is gray, white, or cyan, flip switch one.

Pressing buttons:

1. If switch zero is flipped and the color is not yellow, press buttons BTNU and BTND.
2. If switch three is flipped, press buttons BTNL and BTND.
3. If switch one is flipped, press buttons BTNL and BTNR.
4. If switch two is flipped and the color is not green, press buttons BTNU and BTNR.

On the Subject of Simon Says

Just like the games I remember playing at summer camp. Ah, to be young again!

A blinking light will provide all of the information needed to solve this module. Simply copy what the blinking light tells you to do and you should be able to disarm this module. The light will blink, you press the needed button, and the light will add a blink to its pattern until the module is completed.

1. If the light does not blink purple, orange, green, blue or white, then slant the board towards you.
2. If the light blinks blue or yellow, then slant the board away from you.
3. If the light does not blink red, green, white or blue, then slant the board to the left.
4. If the light blinks orange, green or white, then slant the board to the right.

On the Subject of Morse Code

"The numbers, Mason. What do they mean? Where are they broadcast from?"

Looks like someone is sending messages to the bomb via Morse Code! You'll have to decipher their code word if you want to have a chance at defusing this module.

- **Use the following table to match the Morse Code message with its proper reply.**
- **Once you've figured it out, use switches 0-3 to enter your response and press the center button to transmit.**

If the word is:	Reply with:
DEVIN	0000
MORPHING	0001
WILLIE	0010
ZOO	0011
EMMA	0100
GRYFFIN	0101
GIM	0110
HOME	0111
JOE	1000
STAIN	1001
MEIJER	1010
VIVADOO	1011
VERILEAF	1100
BITSTRAW	1101
XYLYNX	1110
NEXUS	1111

Morse Code Reference

1. A short flash represents a dot.
2. A long flash represents a dash.
3. There is a long gap between letters.
4. There is a very long gap before the word repeats.

A	• ▬	N	▬ •
B	▬ • • •	O	▬ ▬ ▬
C	▬ • ▬ •	P	• ▬ ▬ •
D	▬ • •	Q	▬ ▬ • ▬
E	•	R	• ▬ •
F	• • ▬ •	S	• • •
G	▬ ▬ •	T	▬
H	• • • •	U	• • ▬
I	• •	V	• • • ▬
J	• ▬ ▬ ▬	W	• ▬ ▬
K	▬ • ▬	X	▬ • • ▬
L	• ▬ • •	Y	▬ • ▬ ▬
M	▬ ▬	Z	▬ ▬ • •