# ECE 385

Spring 2014

Final Project

# Bomberman GB in VHDL

Larry Resnik (lsresni2), Matthew Grawe (grawe2)

Section AB0 at 3:00 PM - 5:50 PM

TAs: Xiao Ma (xiaoma1), Shuo Li (shuoli2)

# Contents

# 1 Purpose of Circuit

Our final project would be something that both of us would agree to work on. A few things went into considering this, but matching our strong points would be the most important above all else. Larry has a background in games programming for real-time games, image editing, high-level abstract programming, and low-level bit-wise programming. Matt has a background in signal processing such as dealing with music and images. We would also pool our knowledge of tools together such as using Mercurial, MATLAB, Python, Paint.NET, and other things for this project. We figured we could tackle any game-based project.

We chose to do a recreation of Bomberman. Specifically, we wanted to emphasize the Gameboy game, Bomberman GB. Our motivation for choosing Bomberman was threefold. First of all, we wanted to choose a project that was difficult yet manageable with our combined abilities. Secondly, we saw that Bomberman was created for this class in the past using VHDL, but its final product did not work nicely. We wanted to surpass that, so that set the high bar for us to overcome. Finally, the both of us enjoyed Bomberman in our childhoods and wanted to pay homage to it.

# 2 Bomberman, the Game

Our goal is to recreate Bomberman as much as closely as we could. With that in mind, it is important to understand how we view the game in terms of a set of rules.

Bomberman is a tile-based game, although motion and collision are done based on arbitrary units (we chose pixels as our unit of length). Bomberman would be able to place bombs down on the tile he's standing on which would eventually explode. The explosions would expand outwards in the four cardinal directions. Explosions can hurt Bomberman. Explosions cannot pierce blocks, but they can break bricks. A screenshot of gameplay from the original Bomberman GB game is shown in Figure 1. In it, one can see Bomberman, bricks (represented in this level as large rocks), blocks (represented as chimney-like structures), a bomb, and an explosion expanding outwards and hitting two bricks.

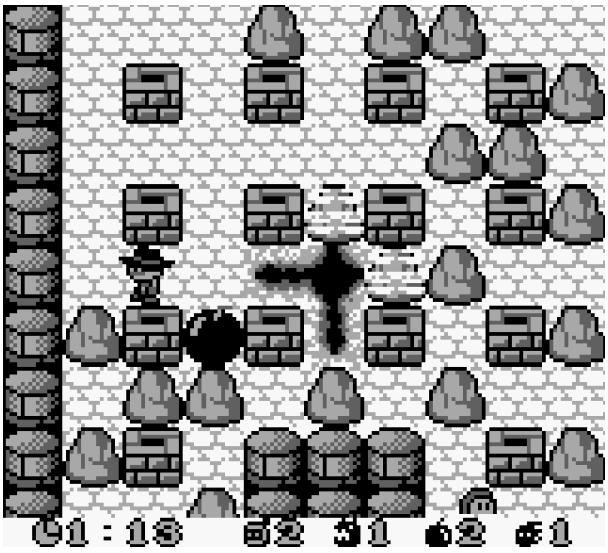


Figure 1: Bomberman GB Gameplay

The levels in Bomberman GB had varying sizes because the game supported scrolling. A secret mode in the game allowed one to fight up to three other computer-controlled Bombermen in an arena. That was a non-scrolling 11x9 area, so it was decided to copy that exact size when making our own levels.

Bomberman's motion is restricted to only being able to move on empty tiles. If you try to move in a direction blocked by a tile, Bomberman will try to walk around it towards the side he is nearest to. Because Bomberman is purposely designed to have an indestructible block in every other tile, that means Bomberman can only be in two tiles at a time maximum. We use this fact a couple times as an optimization in our code.

Bomberman has a time limit to complete his objective. The objective varies between games occasionally, so we decided to come up with a conclusion to our variant of the game as soon as we could develop the basic Bomberman functionality outlined above. In the end, we chose to make a game where Bomberman would blast his way to as many exits as he could before he ran out of time.

As for the original game itself, Bomberman GB is named as such because it was released on the Nintendo Game Boy. The plot put Bomberman into an Indiana Jones style setup, so he was equipped with a cowboy hat and whip while exploring various untraversed areas for treasure.

# 3 Description of Circuit

The final project is fairly sizable, at least in the eyes of two university students who have a month to make it all, so the project's inner workings will be explained in terms of its separate code files and then by how each of those files interact with one another.

To begin with, the circuit is composed of many separate code files called entities. They define the direction that wires connect to each other by defining them as inputs or outputs. Because entities are very concrete as opposed to abstract, we will briefly describe the function of each entity from here on out. All of this will be explained in the section labeled *Entities*.

After explaining what all of our entities are, we will explain how all of these entities function together to create our working game. The details of this synergy will be divulged in the section called *Circuit Behavior*.

As for the circuit we built ourselves, it features Bomberman walking around a 11x9 randomized tiled environment. He always spawns in the upper-left of the map and there is an exit that is always at the lower-right of the map. Unbreakable blocks are placed in pre-defined locations. Breakable bricks appear randomly throughout the level every time Bomberman reaches the exit. Bomberman can lay bombs that, after some time, explode and break bricks. If Bomberman touches a bomb explosion, he will be stunned for about a second and be unable to move. The game has a time limit and keeps track of the number of exits you've reached until you reset the game. When time is up, Bomberman is returned to the upper-left tile of the map and your score is left behind to look at. Bomberman has about a minute to reach as many exits as possible.

Bomberman can put down only one bomb. Each bomb has an explosion radius of one tile in all four cardinal directions. There are no power ups to augment these abilities.

The walking animation for Bomberman is animated. A total of 22 sprites are used in this game. Each sprite has its own purpose. A timer at the bottom of the screen is animated to lose clock symbols. When all clock symbols are gone, you only have a small amount of time left before the game is over. Each exit reached is represented by a bar above the game screen as a number of gems.

Sound effects occur when Bomberman reaches an exit, places a bomb, or a bomb explodes. The sound effects were made by hand using digital signal processing techniques. Music plays non-stop in a proper loop while playing this game. The music was a rip from the original Bomberman GB, so it and the sprites from the same game were intended to give one the feeling of playing the original game.

# 4 VHDL Code: Entity Descriptions

The term Entity is used for each VHDL code file that describes how signals flow. Instantiations of entities are called components. Entities can be defined with any number of components in them. Each entity will be described for how it works on its own.

## 4.1 Entity: SOUND

Our game has fairly developed sound capabilities. Specifically, we added support for WAV file playback as well as manual sound effect generation. We store the WAV file on the 512 KB SRAM unit located on the FPGA. We use clever counting techniques to implement basic sound synthesis. The DAC/ADC interface to the on board FPGA codec is achieved through audio_interface.vhd designed by Koushik Roy that was provided to students. Our SOUND entity has four main processes: **Delegate**, **ProcessSoundEffects**, **SoundLoop**, and **GetNextSample**.

Delegate is a control process that 'delegates' when a sound is allowed to play. For example, each sound has three control signals: sig, fin, and playing. When 'sig' is high, it is interpreted as being a 'play request'. when 'fin' is high, it means that the sound is not currently being sent to the DAC. This process is used to set the value of the 'playing' signals appropriately. The 'playing' signals are used in the ProcessSoundEffect process (described shortly) and keep the sound playing until the sound delay time ends.

ProcessSoundEffects acts as the 'timing' handler for all of the sound effects. Each sound effect has a duration, and the process does the necessary checking and assigning needed to make sure that the sounds continue to play until their duration expires. Each sound as associated 'freq' variables, which describe the frequency of the tones generated. This is done with a sawtooth wave (sawtooth generation in discussed in greater detail later in the report)

SoundLoop is a logistical process that takes the current retrieved sample from the SRAM and loads it into the DAC registers. To play both music and sound effects at the same time, we use both the left and right sound channels. Sound effects are loaded into the left channel, and the music is piped into the right channel.

GetNextSample interacts directly with the SRAM to retreive a 16 bit wave file sample. This is a fairly complicated process, because the clock, RAM sample retreival, and DAC playout mechanics all occur at different rates. Therefore, there needs to be checking code in place to detect when things are 'ready' and 'not ready', and the appropriate actions need to take place.

Since the wave file samples are stored in sequential order on the SRAM, we essentially need to send the memory of each address to the DAC, and when it completes, increment the address by one, do it again, etc. The audio_interface entity sets its output signal data_over to high when it is ok to send in the next playout sample. When this occurs, we increment the SRAM address by one. Otherwise, we keep the ram address the same. We also need to jump over the header, and loop the sound. This is as easy as checking to see if the address is equal to the last sample we wish to play, and if so, resetting the address to the first sample (past the header).

## 4.2 Entity: audio_interface

audio_interface is the entity that was designed by Koushik Roy that implements communication with the audio codec on the Altera board. We use this as it was intended. The only change that we made to his code was the signal that sets the sampling rate. We changed it to match it up with the wave file that plays on the SRAM. Failure to match sampling rates will cause the sound to playback too slow or too fast. Thus, this was necessary.

### 4.3 Entity: d_ff

A single D-Flip Flop. It is a remnant of our Lab 6 code. We technically don't need it now, but the clk_divider code uses the d_ff as a component.

### 4.4 Entity: clk_divider

Takes the FPGA clock and causes it to produce edges much less often. It makes the clock slow enough to correctly synchronize with a PS/2 keyboard.

This entity is an old artifact from Lab 8. We do not directly use it, but we could not easily remove it from the code either.

### 4.5 Entity: SignalSync

Using the clk_divider's slowed down clock, this entity recognizes when the PS/2 clock had a rising or falling edge. We used to use this entity, but it is currently used just to slow down our reactions to KeyboardVHDL.

### 4.6 Entity: KeyboardVHDL

Provided by the ECE 385 TAs, this entity properly recognizes a single key press from a PS/2 keyboard. It takes the main clock and internally divides it down to synchronize with the PS/2 keyboard's clock.

### 4.7 Entity: VGAController

Given to us for Lab 8, this entity synchronizes the FPGA clock to a VGA monitor and allows us to focus on discrete pixels for rendering. It will run over all 640x480 pixels in separate clock cycles. We will choose what to do with that pixel in another entity.

### 4.8 Entity: ColorTable

Based on a 2-bit input, assigns a particular RGB value mapping to it. This allows us to define entire sprites in a cheap 2-bit per color system yet still have access to all of the potential colors offered by the VGA monitor. Ironically, we still use white, black, and shades of gray in our final game. This is intentional because the original Bomberman GB was equally constrained in terms of colors.

### 4.9 Entity: SpriteTable

This entity stores all game sprites in a constant array (a ROM). Each sprite is composed of 16x16 pixels where each pixel is composed of the two bits that would be mapped by ColorTable. Any entity that wants to interact with the SpriteTable can send it a *SpriteID* signal which is simply an offset to the sprite that one wants to render. The *SpriteID* signal is actually a simply a multiply by 16 operation on the requested pixel row to access from the ROM.

One design constraint is that the SpriteTable can only output a single row of pixels at a time. Although it sounds like impossible to deal with since we can only select one of the 16 rows of pixels for a single sprite for every clock cycle, the ColorMapper entity handles this by only choosing one sprite to draw from during every clock cycle.

### 4.10 Entity: ColorMapper

Chooses what color to assign to the pixel defined by VGAController. It is a very complicated process that manages all rendering of the game. It checks if any sprites are renderable and sets the pixel color based on where the VGAController pixel is inside of the sprite.

It knows about the position of all game entities, the position and type of all level tiles, and knows the sizes of all aspects of the game. This entity brings together the ColorTable and SpriteTable entities as components.

### 4.11 Entity: Level

Defines the level that Bomberman plays in. A constant ROM exists without brick tiles so that we can put in brick tiles where we want later. When the game is reseted, an editable version of the map is propagated with brick tiles. This editable variant of the map is what we actually have access to interact with. It outputs all of its rows at once and also accepts changes to all of its rows at once.

The Level entity should only have one component of it ever initialized because there is only one map in the game to work with. The only thing that should be able to edit the level are exploding bombs and recreating the map.

The map is defined to be 11x9 pixels because this is the same size that Bomberman GB used in its multiplayer mode. The single player mode of Bomberman GB had many maps of arbitrary size which you could navigate through by way of a scrolling camera, but we did not go so far as intending to implement scrolling.

Each tile is a 4-bit number that defines the tile as Empty, Block, Brick, Broken Brick, Explosion Center, Explosion Left, Explosion Right, Explosion Up, and Explosion Down. The explosion tiles exist because it would be too complicated to write code for all explosion positions and existences as game entities. There is no support for extra firepower, so no tile defines an Explosion Extension Vertical or Explosion Extension Horizontal. Also, defining tiles as a 4-bit number allows maps to very easily be defined in hexadecimal while still being human-readable. For example the default tile map with $0_{16}$ as Empty and $F_{16}$ as Block is shown in Figure 2.

```
x"FFFFFFFFFFF",
x"F000000000F",
x"F0F0F0F0F0F",
x"F000000000F",
x"F0F0F0F0F0F",
x"F000000000F",
x"F0F0F0F0F0F",
x"F000000000F",
x"FFFFFFFFFFF"
```

Figure 2: Default Level Layout

### 4.12 Entity: MapRowAccess

A big problem we had to work around was that we could not select more than one row from an array at a time. We can't have have multiple components of the Level module for the purpose of accessing multiple rows simultaneously because each Level component would have a separate albeit identical map. To circumvent this constraint, the MapRowAccess entity was developed. It accepts all rows of the one Level component as input and allows anything to request any numbered row of those rows. Many MapRowAc-

cess components could be safely created for any sort of purpose because the MapRowAccess does not duplicate the level itself. It is essentially a large decoder. Some various uses for MapRowAccess include determining tile collision coordinates, getting the tile type below the pixel focused by VGAController, finding out what tile was affected by a bomb explosion, etc.

### 4.13 Entity: PlayerEntity

Originally Lab 8's Ball entity, the PlayerEntity defines Bomberman as a game entity. It controls his behavior in terms of how he reacts to the keyboard that makes him move. PlayerEntity also knows all about the map in order to properly react to tile collisions. Bomberman's size, position, and speed are all set in this entity. All stored values are based on pixels offsetted from the top-left of the monitor as the origin.

Tile collision detection is a complicated process that involves many conversions to-and-from pixel and tile coordinates. Care is also made to reassign the origin to the upper-left tile of the map instead of the upper-left pixel of the monitor. Failure to do so produces bogus values for the tile that Bomberman believes he is on.

### 4.14 Entity: bomb

The bomb entity encapsulates the functionality of the 'bombs' that Bomberman can place on the map. Bombs can be in three unique states: *ticking*, *exploding*, or *dead*. In the *ticking* state, the bomb is waiting to explode. In the *exploding* state, the bomb is in the process of exploding; the player is vulnerable if in the path of the explosion during this state. The final state, *dead*, is the default "off" state of the bomb; when a particular bomb instance is *dead* it is not currently placed on the map (and therefore is not ticking or exploding). The entity has an output signal that is picked up by the sound entity, which plays the appropriate sound effect for bomb placement. Similarly, bomb explosions are accompanied by another sound effect.

### 4.15 Entity: Main

The Main Entity is, as one would expect, our top level entity. It defines all important components such as PlayerEntity, SOUND, bomb, Level, and so on. It also defines the behaviors between these high level entities. Main also manages the keyboard and output to the VGA monitor.

When a bomb is laid, the bomb is given a proper location to be placed based on Bomberman's central origin. When the bomb explodes, the tiles around the bomb are checked to see if they are Bricks. Bricks become Broken Brick tiles whereas all Empty tiles become Explosion tiles specific to the direction the blast is moving. After a delay, the explosion is dissipated such that all Explosion and Broken Brick tiles become Empty tiles. This is the central aspect of how the circuit behaves as a game. All of this can only be realized by the Main entity which manages PlayerEntity, bomb, Level, and MapRowAccess.

### 4.16 Entity: HexDriver

Converts a 4-bit number to a bit vector that can be mapped to the hexadecimal display on the FPGA. This code was given to us on the day that we first started using VHDL. It is also explained in Altera's manual.

We only use HexDriver for debugging output. The final product of our game has no use for HexDriver.

# 5 Circuit Operation

The separate entities hold little meaning on their own. When connected, they create our game by acting together as a whole. This section explains how the circuit works in its entirety without delving too far into the realm of how it was coded in VHDL.

## 5.1 Behavior: Sound

Sound is a complicated subject made only harder by us using an FPGA. The game uses the SRAM for some of the sound produced and generates the rest mathematically.

### 5.1.1 WAV File Playback

The background music in our game is uploaded as a WAV file on the SRAM. To playback the wave file, we have to obtain samples from the RAM and feed them into the DAC all in real time. To do this, we maintain two separate processes **GetNextSample** and **SoundLoop**. **GetNextSample** interacts directly with the RAM. We maintain an address vector, and every other clock cycle (we wait one clock cycle for the SRAM reading operation to finish) the process will load the SRAM data at that address into a signal.

Since the DAC takes longer than two clock cycles to push a sample of sound to the speakers, we have to monitor the progress of the sample as it travels through the DAC. When the sample is through, the sound driver raises a flag **data_over**. One this is detected, we choose to increment the RAM address bit by 1, causing the next wave sample in the SRAM to be accessed and fed into the DAC. In this manner, we can play back a wave file in real time.

In order for the wave file to loop, we added logic that detects when the address bit gets to the last byte in the wave file. Since data in the SRAM is 16 bits long, the *looping address* is simply the number of bytes in the wave file divided by two. Additionally, we had to offset the starting address to jump over the header of the wave file (playing back the header of a wav file over speakers can sound very unpleasant).

There are some other caveats that can make wav playback nonintuitive. For example, not all wave files are produced with the same sample rate. When creating the background music, we had to re sample the files with an external sound editor so that they would match up with the sample rate of the DAC. Additionally, the relatively tiny capacity of the SRAM meant that we had a very limited range of music duration to work with. To make the most of what we had, we chose to cleverly slice the actual Bomberman GB game music in such a way that it perfectly loops itself every 6 seconds.

### 5.1.2 Sound Synthesis

For game sound effects, we chose to generate the sounds manually using some basic signal processing theory. To generate a tone, there needs to be a way to generate oscillation, such as a square wave or sine wave. The frequency of the wave determines the pitch of the sound that is heard by the human ear. Therefore, we needed to generate the oscillation in such a way that we could control the rate at which it occurred (essentially, controlling it period). To avoid trigonometry, we opted against trying to generate a sine wave.

Our solution is elegantly simple. Consider a signal $t$. On each clock cycle, add 1 to it. As time passes, t continues to accumulate magnitude. If $t \in \mathcal{R}$, then $t$ will approach infinity. However, we are working with finite precision. Since the DAC interprets the signal vector as 2's complement, once $t$ is sufficiently high **it will rollover and become a maximally negative value**. Then $t$ will begin to increase again until hitting the maximum representable positive value for the given number of bits we choose to represent the signal sample with (in our context, 16). If we take the horizontal axis as time and the vertical as

sample magnitude, this results in the following signal shape as in Figure 3:

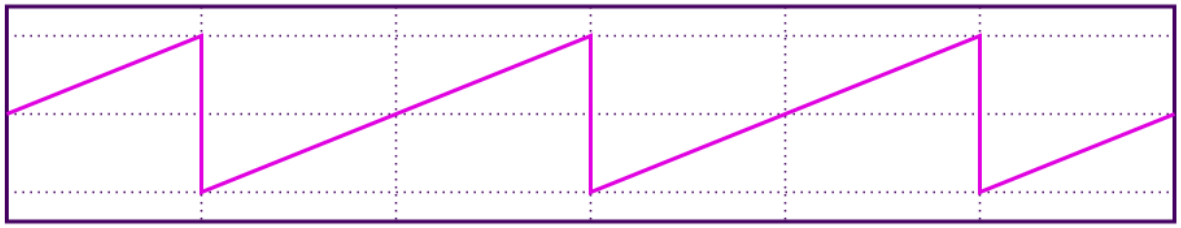


Figure 3: Sawtooth Wave. Image taken from Wikimedia Commons (freely licensed).

This signal shape is a **sawtooth wave**. We know from Fourier analysis that we can construct arbitrary signal shapes such as this one from infinite sums of pure sines and cosines. Therefore, it makes sense that we can utilize this methodology to generate tones.

#### 5.1.3 Role of Duration in Sound Effect Synthesis

Ideally, we want the duration of sound effect playback to be controllable. We achieve this by utilizing counters. Since the system clock oscillates at 50 MHz, we can calculate how many 'cycles' that occur over a given time interval $t$ as

$$50 \times 10^6 \times t = \textbf{number of cycles in t seconds}$$

If we interpret "cycles" as being counter increments, we can obtain an estimate on the number of times we need to increment the counter at a given clock frequency before the specified duration $t$ elapses. We then want to solve

$$2^n = 50 \times 10^6 \times t$$

for n to obtain the size of the vector to use in VHDL needed to measure this delay:

$$n = \log_2 \left(50 \times 10^6 t\right)$$

for example, if we want to measure a one second delay, we have

$$n = \log_2(50 \times 10^6) = 25.57 \rightarrow \textbf{26}$$

thus, for $n \geq 26$, this delay is measurable with a clock rate of 50 MHz.

### 5.2 Behavior: Rendering

Drawing a pixel to a screen is done using the ColorMapper entity that we were given back in Lab 8. The VGAController entity has a clocked process that moves across the screen and focuses on unique pixels. At each pixel, we check what we want to draw. Based on the thing to be drawn, we send RGB values for a pixel to the ColorMapper. This is the overall approach taken to rendering.

### 5.2.1 Rendering Color

Because our game has multiple colors (black, light gray, dark gray, and white), we needed to extend the Lab 8 code to recognize more than whether or a pixel was "on" or "off". The ColorTable entity was invented for that purpose. A pixel now has a two bit value associated with it. Those two bits give us $2^2 = 4$ unique colors. It's the same number of colors used in Bomberman GB, so one may even assume they used similar constraints we've put upon ourselves. These two bits are a "color code" that the ColorTable entity maps to RGB values.

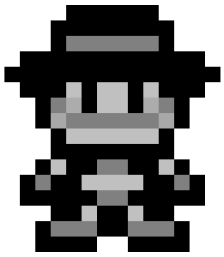
### 5.2.2 Rendering Sprites

A sprite is an image rendered on screen. Seen another way, it is a block of colors that we recognize as having a particular meaning. For example, a sprite of Bomberman has colors mapped in a format that shapes his body during one frame of animation. So if we wanted to draw Bomberman on our monitor, we would need the proper appearance of Bomberman first and foremost.

Thankfully, the people who worked on Bomberman GB already made amazing sprites for their game. With a little exercising of Fair Usage, we opted to take a small number of these sprites in order to accurately recreate Bomberman GB on the monitor. By using an emulator to play the original Bomberman GB and taking screenshots of gameplay, we could save those sprites in their 16x16 original glory on our hard drive disks. Some image editing would be required to extract specific sprites from the game's full screen capture. The next thing to do would be to get those sprites onto the FPGA.

In order to store a sprite in the FPGA, we would either need to use the SRAM or hard code the sprite into VHDL code. Since the SRAM would be constantly accessed for the music, we opted to code up the sprites as ROM data. They would be stored in the SpriteTable entity as a large array of bits. Since Bomberman GB sprites were 16x16 in dimension, each ROM entry is (16 * ColorCodeSize) bits wide with (16 * NumSprites) address lines.

When we have an image, it needs to be converted from pixels that the emulator outputs to a bit pattern that the VHDL recognizes. This was handled by using MATLAB to turn all of the pixels into numbers. We get four 16x16 matrices representing the RGBA channels of the image, but the Gameboy had the same colors for all color channels, so we only use one of those channels when producing our numbers. A sample of the work is shown in Figure 4.



```
>> G = image(:,:,3);
>> G(find(G==0)) = 1

G =

  255  255  255  255  255    1    1    1    1    1    1  255  255  255  255  255
  255  255  255  255    1    1    1    1    1    1    1    1  255  255  255  255
  255  255  255  255    1  128  128  128  128  128  128    1  255  255  255  255
  255  255    1    1    1    1    1    1    1    1    1    1    1  255  255  255
  255    1    1    1    1    1    1    1    1    1    1    1    1    1  255  255
  255  255    1    1    1  192    1  192  192    1  192    1    1    1  255  255
  255  255  255    1  128  192    1  192  192    1  192  128    1  255  255  255
  255  255  255    1  192  128  128  128  128  128  128  192    1  255  255  255
  255  255  255  255    1  192  192  192  192  192  192    1  255  255  255  255
  255  255  255  255    1    1    1    1    1    1    1    1  255  255  255  255
  255  255  255    1  192    1    1  128  128    1    1  192    1  255  255  255
  255  255    1  128    1    1  192  192  192    1    1  128    1  255  255  255
  255  255    1    1    1    1    1  128  128    1    1    1    1  255  255  255
  255  255  255  255    1    1  192    1    1  192    1    1  255  255  255  255
  255  255  255    1  128  128  128    1    1  128  128  128    1  255  255  255
  255  255  255    1    1    1    1  255  255    1    1    1    1  255  255  255
```

Figure 4: Importing the image into MATLAB for CSV Preparation

To accomplish the CSV generation, we created the following helper function in MATLAB:

```
function [img] = pixel_access(filename)
        [a map alpha]  = imread(filename);
        img = a(:,:,3);
        n = length(filename);
        filenameOut = filename(1:n-4);
        filenameOut = [filenameOut '.csv'];
```

```
        csvwrite(filenameOut, img);
end
```

Afterwards, Python was used to convert MATLAB's comma separated list of numbers into a format that VHDL understands. Only four unique numbers are outputted by MATLAB because of the limitations of the Gameboy itself. Those four numbers are mapped to our ColorMapper's 2-bit numbers. For example, the conversion of Figure 4's image into the list of bits we'd use to render it in VHDL came out as in Figure 5.

```
x"ffc003ff",
x"ff0000ff",
x"ff1554ff",
x"f000000f",
x"c0000003",
x"f022880f",
x"fc62893f",
x"fc95563f",
x"ff2aa8ff",
x"ff0000ff",
x"fc81423f",
x"f10aa04f",
x"f001400f",
x"ff0820ff",
x"fc54153f",
x"fc03c03f",
```

Figure 5: Bomberman Represented as 32x16 Bits.

With the ability to turn any sprite from the Bomberman GB game into a VHDL compatible format, we were able to create the SpriteTable entity. Instead of storing the sprites externally such as on the SRAM, the SpriteTable defines all of our sprites in a hard coded format just like what was shown in Figure 5. All sprites would be stored in a massive array of constant numbers (a ROM) whose size was defined as follows:

$$(\text{BitsPerColor} * \text{ImageWidth}) \text{ x } (\text{NumSprites} * \text{ImageHeight})$$

Accessing a sprite would be done by requesting its row offset and also its sprite ID. The two numbers would be multiplied together to jump around the sprite ROM to the requested sprite's row.

The Python script subsequently used after the MATLAB script extracted the sprite is written below.

```
from sys import argv

if len(argv) > 1:
        # An argument was passed. Assume it's the file to load.
        filename = argv[1]
else:
        filename = 'in.txt'

# These are the colors outputted by Matlab.
# It comes from either the sprite sheet I got online or
# from the screenshots I take in VirtualBoyAdvance.
# Some hand massaging is involved to turn the Black color from 0 to 1.
MATLAB_BLACK = 0
MATLAB_DARK_GRAY = 168
MATLAB_LIGHT_GRAY = 96
```

```
MATLAB_WHITE = 255
MATLAB_WHITE2 = 248

# The color mapping from the Matlab text document output
# to the constant colors defined in the VHDL code for ColorTable.
# ColorTable uses two bits per color.
color_mapping = {
MATLAB_BLACK:0,
MATLAB_LIGHT_GRAY:1,
MATLAB_DARK_GRAY:2,
MATLAB_WHITE:3,
MATLAB_WHITE2:2}

all_hex = []
reading = True
with open(filename, 'r') as infile:
    while reading:
        numbers = infile.readline().split(',')
        if len(numbers) > 1:
            # The very last number has a newline stuck to it.
            numbers[-1] = numbers[-1].strip()
            if not(numbers[-1]):
                # We stripped all whitespace and got an empty array. All done.
                reading = False
        else:
            # No more numbers to read from the file.
            reading = False

        if reading and len(numbers):
            # Convert all numbers from strings to integers.
            numbers = [int(i) for i in numbers]
            hexlist = []
            # Jump across every other number and merge them into single numbers.
            # Simple C code would look like the following (char = 4 bits):
            # char a, b, out;  out = (a << 2) | b;
            # Where "a" and "b" are two bits wide and "out" is four bits wide.
            for i in range(len(numbers)/2):
                hexlist.append(
                    color_mapping[numbers[i*2]] << 2 | \
                    color_mapping[numbers[i*2+1]])
            all_hex.append(hexlist)

# Print all numbers into a format that VHDL understands.
# All of the numbers are already composed of two color patterns since
# one hex number stores 4 bits, so force the hex to display in two digits.
# The string[2:] removes the "0x" from hex numbers like "0xFF".
# See SpriteTable.vhd for sample output.
for hexvals in all_hex:
    print('x"' + ''.join(['{:#x}'.format(int(h))[2:] for h in hexvals]) + '",')
```

### 5.2.3 When and Where to Render

The VGAController module would move along the monitor screen horizontally and then vertically one pixel at a time. The ColorMapper entity would be told the exact pixel we needed to pick the color for. Choosing the color of the pixel would be done by first checking if the VGA's pixel overlapped any of our

game entities. The check is a simple rectangle-point collision check where the rectangle was the game entity and the point was the VGA pixel. The collision checker looks like the following.

$$\text{CollisionX} = (\text{vgaX} >= \text{entityX}) \ \& \ (\text{vgaX} <= (\text{entityX} + \text{entityWidth} - 1))$$

$$\text{CollisionY} = (\text{vgaY} >= \text{entityY}) \ \& \ (\text{vgaY} <= (\text{entityY} + \text{entityHeight} - 1))$$

$$\text{HadCollision} = \text{CollisionX} \ \& \ \text{CollisionY}$$

If there was a collision, we would raise a flag that causes a separate VHDL process to activate. This process would calculate where within the sprite the pixel was. The calculation is entirely based on pixel offsets from the monitor's upper-left origin. The pixel is calculated as such.

$$\text{PixelX} = \text{vgaX} - \text{entityX}$$

$$\text{PixelY} = \text{vgaY} - \text{entityY}$$

This works because the VGA pixel is guaranteed to be within the image. However, each row of our sprites defined in the SpriteTable entity were composed of two bits to represent each color. There's 32 bits in each row whereas the sprite itself has 16 pixels per row. So we need to double the PixelX value and grab the two bits at 2 * PixelX and 2 * PixelX + 1. We set the sprite ID signal based on the game entity we are rendering so that we access the correct sprite from the ROM in SpriteTable.

The tiles on the map are dealt with differently because the tile to display is based on the VGA pixel's position in the map. The map is offsetted from the top-left of the screen, so we need to undo that translation to first determine the type of tile the VGA pixel is sitting on. This is unnecessary because we set the map to be a multiple of 16, so simply taking modulus 16 of the pixel will work for rendering.

$$\text{PixelX} = \text{vgaX}\%16$$

$$\text{PixelY} = (\text{vgaY} - \text{MAP\_START\_Y})\%16$$

#### 5.2.4 Rendering Animations

Animations are done by transitioning between multiple images after some period of time. The only animated game entity is Bomberman himself. Bomberman is given a state machine to track the image that would be displaying him. What we mainly need to show is his walking animation. To do so, we watch his speed during every clock cycle. As long as his speed is non-zero, we increment a timer in the state machine. If the timer reaches a pre-defined threshold, we change Bomberman's state to show the next sprite in his animation. If Bomberman ever stops, we revert the state to where he stands and faces the direction he was last moving in. Bomberman could stop because of touching a non-empty tile or because the player let released a movement key from the keyboard. The only thing to watch out for was when Bomberman changed his direction during an animation for movement in a different direction.

We wanted to have more advanced animations, but compiling the extra sprites onto the FPGA made the compilation times obnoxious. It would also require bloating the animation state machine with a lot more of more or less the same code copy-pasted elsewhere. Also, we would not get a higher grade for doing more of the same thing, so we dropped the feature so that we could develop other things.

### 5.3 Behavior: Movement

Movement is a key component to almost any real time game, but restricting Bomberman's mobility is an important aspect of the gameplay. In our game, Bomberman moves one pixel at a time in any of the four cardinal directions, but he is restricted to only be inside of empty tiles. In order to stop Bomberman from walking through blocks and bricks, we must stop him from walking into them.

### 5.3.1 Pixel and Tile Coordinate Transformations

The monitor is composed of 640x480 pixels. Any game entity can occupy any one of those points. Within those 640x480 pixels, the game map is composed of 11x9 tiles. Those tiles are 16x16 sprites, so each map tile would occupy 16x16 pixels. All of our sprites are 16x16 pixels in dimension, so we should have no problem fitting our sprites inside of tiles properly. It is easier to look at a picture of this instead of thinking about how it all the math works, so take Figure 6 for example. It is a screen shot of actual Bomberman GB Gameplay. We will be attempting to recreate its ruleset by closely examining how it functions.

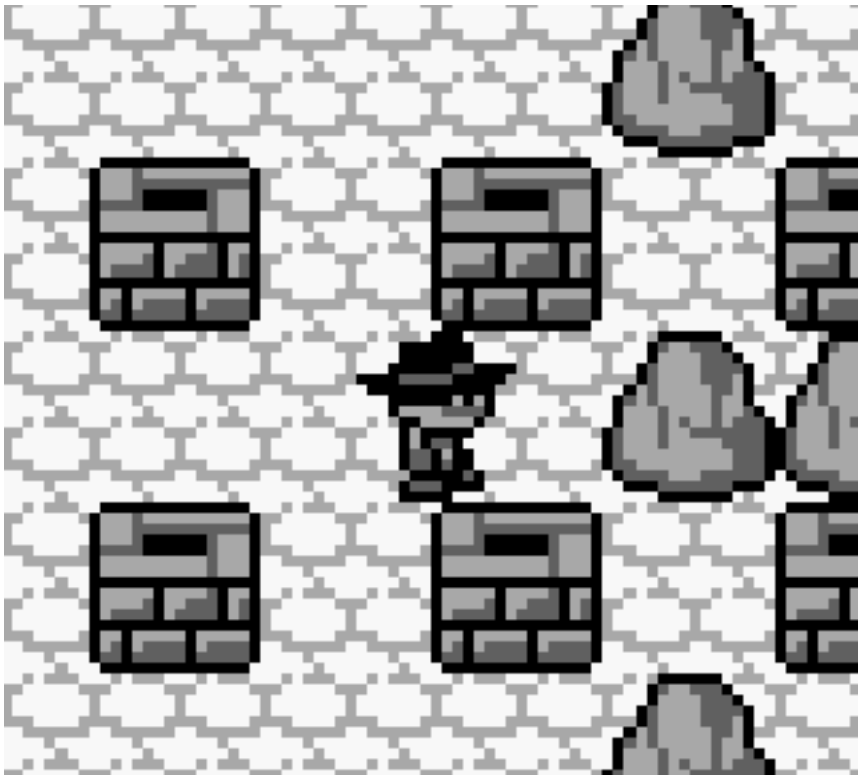


Figure 6: Bomberman GB Gameplay

In order to determine which tiles need to be collided with, we must know where we are in the map. Bomberman's location is stored in units of pixels instead of tiles so that he can have fine grained movement, so we must convert his location to a tile location. The conversion is simple because all sprites are the same size. Consider Figure 7 which outlines all collision boxes.

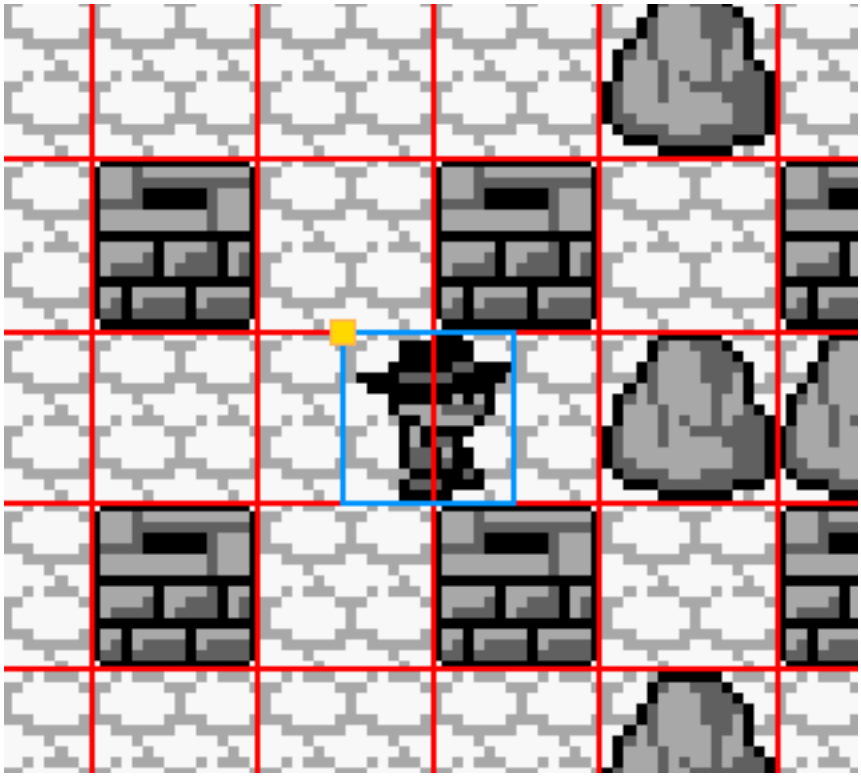


Figure 7: Grided Gameplay Screen Shot

A yellow dot was drawn in Bomberman's upper-left corner because this is an important location to pay attention to. Right now, that dot has an x-position somewhere in the middle of a tile and a y-position exactly equal to some tile's pixel-based index. Bomberman cannot move up or down in his position, so let's presume his x-position is (in decimal and binary) $54_{10} = 00110110_2$. If Bomberman were to move to the left by six pixels, we'd be at x-position $48_{10} = 00110000_2$. Notice how the lower four bits changed, but we'd still be inside of the same tile. This is true because the width of a tile is 16 pixels and we have not traveled the full length of a tile to go on to another tile. In short, it is the *upper bits* that determine the tile that a game entity exists on. Simply put, all we need to do is divide the pixel location by the sprite size (16) to get our tile location.

The opposite holds true too. To convert a tile to pixel coordinates, we would need to take the tile's location stored in the map and multiply it by 16. Not surprisingly, this will always result in the lower four bits being zero because all tiles are neatly lined up next to each other. There are no fine discrepancies between tile locations. To illustrate what I mean, consider the table in Figure 8. It provides a couple tile position values in order for you to recognize the pattern. Note that the table's "Row" word could just as easily be replaced by the word "Column".

| Tile Row | Row in Pixels (Decimal) | Row in Pixels (Binary) |
|---|---|---|
| 0 | 0 | 00000000 |
| 1 | 16 | 00010000 |
| 2 | 32 | 00100000 |
| 3 | 48 | 00110000 |
| 4 | 64 | 01000000 |

Figure 8: Sample Tile Locations

Games programmers would ordinarily be able to start recreating a game with this much information.

However, we are constrained by the fact that we are using an FPGA to do all of our calculations. The FPGA has only 12 multipliers, and we don't even know how many dividers it has! We assume it has none, but that is all right. The reason the pattern was pointed out involving the bits was because we have somewhat easy access to bit-based manipulation as opposed to multiplication and division. The key is that division by 16 is done by taking the upper 4 bits of an arbitrary string of bits. By contrast, modulus 16 is done by taking the lower 4 bits of an arbitrary string of bits. We will use the modulus operator to determine a game entity's position within a tile.

### 5.3.2 Tile Collision Detection and Reaction

Now that we know how to relate Bomberman's position to where he is in the map, we need to know whether or not he can move towards tiles or not. This part is tricky due to off-by-one errors, VHDL's 4th quadrant math, and the issue of signals not being updated until the clock cycle after they were accessed. Despite that, the general work flow goes the same way in all four directions.

Consider the scenario in the previously shown Figure 7. Bomberman exists where the yellow dot is. The tile above the yellow dot is empty, but Bomberman cannot move upwards because the tile on Bomberman's upper-right is solid. So we see that the procedure to movement will always involve a check against two tiles. If both tiles are empty, Bomberman can move in that direction. If Bomberman is exactly within a tile, the tile to his side is not even being touched, so we only need to check one tile in this case.

## 5.4 Behavior: Bombs

Bomberman could not be Bomberman without bombs. Bombs allow Bomberman to break through brick tiles and reach his destinations. Therefore, our two initial tasks before implementing bombs were to figure out how to properly do tile collision and how to make the map editable in real time. It was already explained in the previous section how we could find where we were in the map and react to the nearby tiles. We needed to make the map dynamically editable because breaking bricks involves changing bricks into empty panels at the very least.

To work around this problem, the Level entity got a brand new 2D array of the same exact size and type as the original level we had worked with. The original level was defined as a constant, but it became re-purposed as a default value for the new, editable level array. During every clock cycle, Level would check if a Load signal was high and, if so, accept new rows of tiles for all 11 tiles in all 9 rows. Essentially, we turned Level into a giant register.

Now that the level was editable in real time, we would be able to place a bomb and make the blasts break bricks. We would need to be able to produce the bomb now. The bomb would be an entity with a location, a state, and a timer. Its location obviously dictates where it is in the level such that when it explodes, it would explode in the proper location. The state would define if the bomb is not planted, if its fuse is lit and waiting to explode, and if it is currently exploding. To make the proper state transitions between those states, we would need a timer. We can't have the bomb be planted and have it instantaneously explode after all.

The bomb itself is a simple thing to create. Exploding on the other hand takes some thought. Explosions occur on multiple tiles and diminish over time. Extra firepower means we'd have to check more tiles beyond the bomb's explosion location for bricks to break or blocks to stop on. We can't explode beyond anything we hit either. Extra bombs means we'd have to account for separate timers for each bomb and make sure the explosions disappate at the time designated by when the bomb had exploded. Bomb explosions could activate other bombs, but those activated bombs have a delay of when they activate and peter out. Also, we'd need to keep track of which bombs are activated and which one's are inactive. We don't have the luxury of a growable array in VHDL, so some solution would need to be made for that.

Due to time constraints, it was apparent that we could not afford to implement such power ups. We chose to work with the bare minimum Bomberman allowed: one bomb with a firepower of one. That means the one bomb Bomberman lays would have an explosion that extends only one square beyond its location.

A single bomb with minimal firepower can produce up to five explosions (four cardinal directions and its center), so we'd have to track the timing of multiple explosions on different tiles. That implies the usage of unique game entities for each bomb explosion, but checking each explosion entity during runtime is not only painful to code. Instead, a clever optimization could be made solely because no other bombs could be on the field. Having no other bombs to lay meant no need for keeping separate arrays of data for bomb and explosion timings. We also would not need to deal with cross-fire from two bombs. Therefore, we don't need to specifically identify with each explosion blast. If we could represent the explosions in a simple fashion, we could react to each explosion in an equally simple manner.

It was then decided to represent all explosions as unique map tiles. Because explosions are locked to tiles anyways, it made implementation more simple. Tiles don't store information about when a bomb blast must be cleared, so we just store that in the one bomb that had exploded. Moreover, removing bomb explosions and broken bricks after an explosion has ended is a matter of looking through all of the tiles in a map and removing the unwanted tile types.

To that respect, a certain optimization was made based on the realization that an empty tile was defined for the map to be binary 0000, a brick tile was 0001, and a block tile was 1000. The algorithm that clears blocks after an explosion has happened simply has to maintain 0001 tiles, and turn all non-1000 tiles into 0000 tiles. This has the effect of turning broken bricks and explosions into empty tiles, but ignoring other tiles.

One thing to note is that the original Bomberman did not cause bricks to instantly disappear upon being hit with a bomb. The bricks hit with explosions would turn into broken bricks. When the explosion ended, the broken bricks would become empty tiles. This stops the player from walking through just-now exploded bricks and into his own fire.

## 5.5 Behavior: Extra

One other note that didn't fit in with the lineup of behaviors was about level randomization. The game would have a clock that runs continuously. When the player's tile was the same as the exit's tile, the player would be warped back to the upper-left tile and the level would be randomized re-populated with bricks using the aforementioned clock as the random seed.

Despite the level being randomized, there are four tiles that would not be allowed to have bricks put on them. The first is the exit tile because that tile hides the tile below it. The second is the tile that Bomberman spawns on. The third and forth are the tiles below and to the right of Bomberman's spawn point. It is a standard practice for Bomberman games to do this because Bomberman must cower behind a corner to protect himself from his very first bomb blast. Otherwise, we'd put Bomberman in a situation where the only way to break out of the spawn location would be to place a bomb nearby himself and suicide.

Additionally, the game had two informative bars in it. The timer bar was initially filled with clock symbols and would diminish over time. This was done by representing all fractions of the game timer by an array of 1's (clocks) and 0's (nothing). Another bar drawn with gems would represent the number of exits reached. This one was done by shifting in 1's into an array that is initially zeros.

# 6 Running the Program

Making our game work takes some time to setup. The steps we take to setup the game is as follows.

1. Compile the latest version of our project's code.
2. Load the FPGA with the DE2_USB_API.sof to allow uploading to the FPGA's SRAM.
3. Run the DE2_Control_Panel.exe program.
4. Tell the control panel to Open the USB port.
5. In the control panel's SRAM tab, check the "File Length" check box in the "Sequential Write" section.
6. In the "Sequential Write" section, click "Write a File to SRAM" and choose our bgm32000.wav.
7. Close the USB port from the control panel.
8. Load the FPGA with the project's sof file.
9. The game will run now, but the FPGA needs to be connected to a PS/2 keyboard, a VGA monitor, and an external speaker.

# 7 Problems and Solutions

In the course of working on the project, we had to deal with fixing a lot of problems. It was a valuable, albiet time-consuming experience. This section lists some of the more frustrating things we experienced.

• The display on the monitor would have a lot of flickering pixels. Some pixels even appeared to be a color that was not even defined in ColorTable. The issue was more or less eliminated by making all accesses to the SpriteTable react to a clock edge. The problem is presumably because the time it takes to access the ROM we wrote would not be entirely deterministic without the clock governing it.

• When first attempting to implement tile collision detection and reaction, we witnessed a lot of irregular detections as though the map blocks were offsetted. By using a debug variable to follow the calculated tile position of Bomberman, it was discovered that the formula created to transform pixel-to-tile coordinates was wrong. We shifted the map itself to start on pixels that were multiples of the tile sizes (16x16) so that it made the transformation formula simpler. Secondly, we had to subtract the position Bomberman was within the map from the pixel-to-tile transformed values of Bomberman's position. Getting the correct formula made tile collision detection behave as expected.

During tile collision reaction, Bomberman would frequently stop ahead of when we expected him to stop. The way we check for collisions is done by examining the tile Bomberman is going towards. For example, pseudo code of what we do if Bomberman were moving to the right would go as follows.

```
if moving right then
    if right tile = empty tile then
        dx ← pixel speed x
    else
        dx ← 0
    end if
end if
```

We first check if the next tile is empty. If it is, we can move freely. If not, it is something Bomberman can run into. Otherwise, we would be walking into a thick tile. Setting to dx to zero would only work in our case because Bomberman's speed is set to one. However, this code did not work. As soon as Bomberman went towards a tile whose next tile was thick, Bomberman would pre-actively stop in his tracks. The problem was that Bomberman's existence was brought into the tile-based domain properly, but he still needed to move based on pixels. The solution was to change the tile collision code to the following format:

```
if moving right then
    if right tile = empty tile then
        dx ← pixel speed x
    else if bomberman pixel x + pixel speed x > distance_to(right tile) then
        dx ← pixel speed x
    else
        dx ← 0
    end if
end if
```

If the next tile is thick, but we still have the room to move towards it, then we should still move towards it. When we used this style of collision in all four directions, Bomberman's motion became very natural.

• Throughout the development, we had a recurring issue with the sound mysteriously not working sometimes. The issue seemed timing related, but we were never able to fully pinpoint the exact cause. We were able to make it occur less frequently by fixing an issue we had with some processes using falling_edge and some using rising_edge. We did this by making sure that all processes were using the same edge triggering (rising_edge).

# 8 Simulations

Using the ModelSim waveform simulator, we examined one of the more easily accessible parts of the circuit. The waveform in Figure 9 illustrates the functionality of the bomb's state machine and accompanying sound effects being activated.

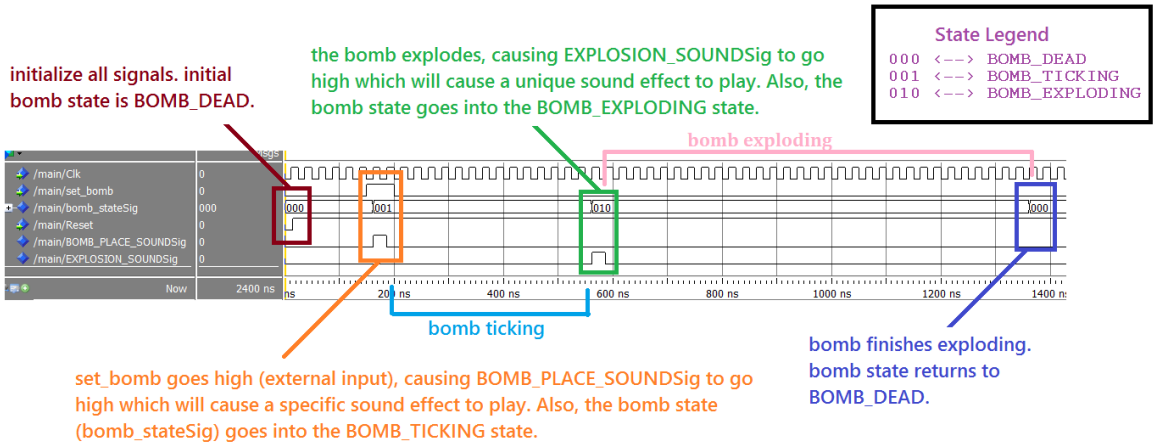



Figure 9: Simulation Waveform of Bomb Entity Functionality

As additional proof of the circuit being operational, a picture of real gameplay footage taken from the monitor is in Figure 10. Note that a fairly large amount of white area was cropped out of this image.

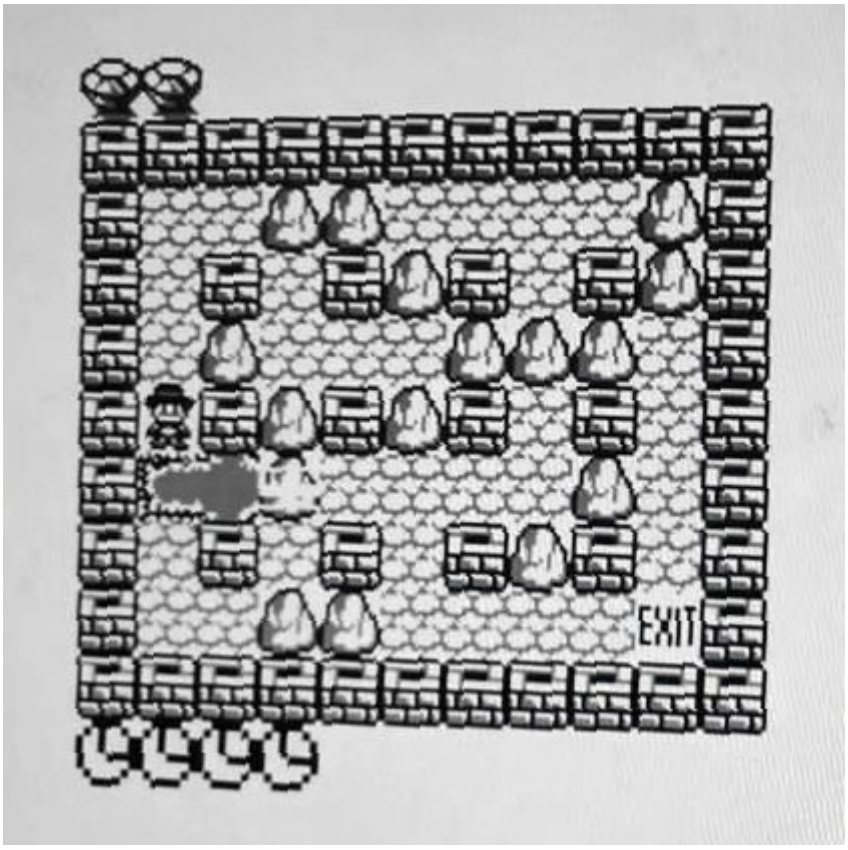



Figure 10: Gameplay on a VGA Monitor with the FPGA

# 9 Borrowed Work

Not everything we coded was done on our own efforts. We would not have gotten the project to where it was if it were not for other people's contributions.

The keyboard handler done in KeyboardVHDL was given to us by the ECE 385 TAs because we were expected to already have a working keyboard handler from Lab 7. Ours was faulty because we took a complicated approach to implementing it. This code was made Kyle Kloepper and edited by Stephen Kempf.

The sound handler was created by a previous student of ECE 385, Koushik Roy. It allowed us to interact with an external speaker.

The music for Bomberman GB was composed by Noriyuki Nakagami, Yasuhiko Fukuda, and Takashi Morio. Of that music, we only used a six second cropped loop of Stage 1's theme. The sound effects were hard-coded by using techniques learned by mimicking VIznut's "Experimental One-Line Music" videos.

The sprites for Bomberman GB are owned by Hudson Soft. The original artist is unknown. Although some sprites were extracted manually using Virtualboy Advance, two sprite sheets were used from The Shyguy Kingdom made by users Black Squirrel and WaxPoetic. Some sprites were edited in color to make scripting them into the encoding we'd use in VHDL easier to parse. The sprites we used in no particular order or intended color are shown in Figure 11.

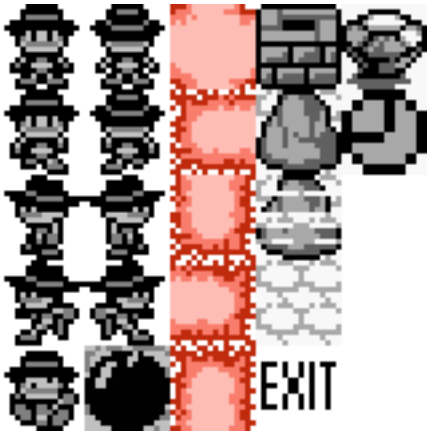



Figure 11: Sprites Used

## 10 Progress Report

Our team used hg (Mercurial) to keep our code in working order and to be able to regress the code in cases of poor planning or failed test pieces. The full report of our logs is reproduced here to illustrate the amount of time and effort put into each memorable release of our code. Note that Larry's commit count outnumbers Matt's partly because Larry was the maintainer of the hg repository and had to guide Matt in the usage of hg during the middle of this project.

## 11 Timing and Map Report

Our project was completed with the following statistics according to Quartus II.

- Total logic elements: 6,287 / 33,216 (19 %)
- Total registers: 1,660 / 33,2216 (5 %)
- Maximum Frequency: 69.65 MHz for Clk, 1017.29 MHz for clock_divider

## 12 Block Diagrams and State Machines

The project itself is very large, so the RTL diagrams do not actually fit in this report. In fact, our top level entity is so large that a number of image manipulation softwares used on its exported image have refused to work with the image due to memory constraints. However, a very general overview of the top level entity can be seen as in Figure 12.

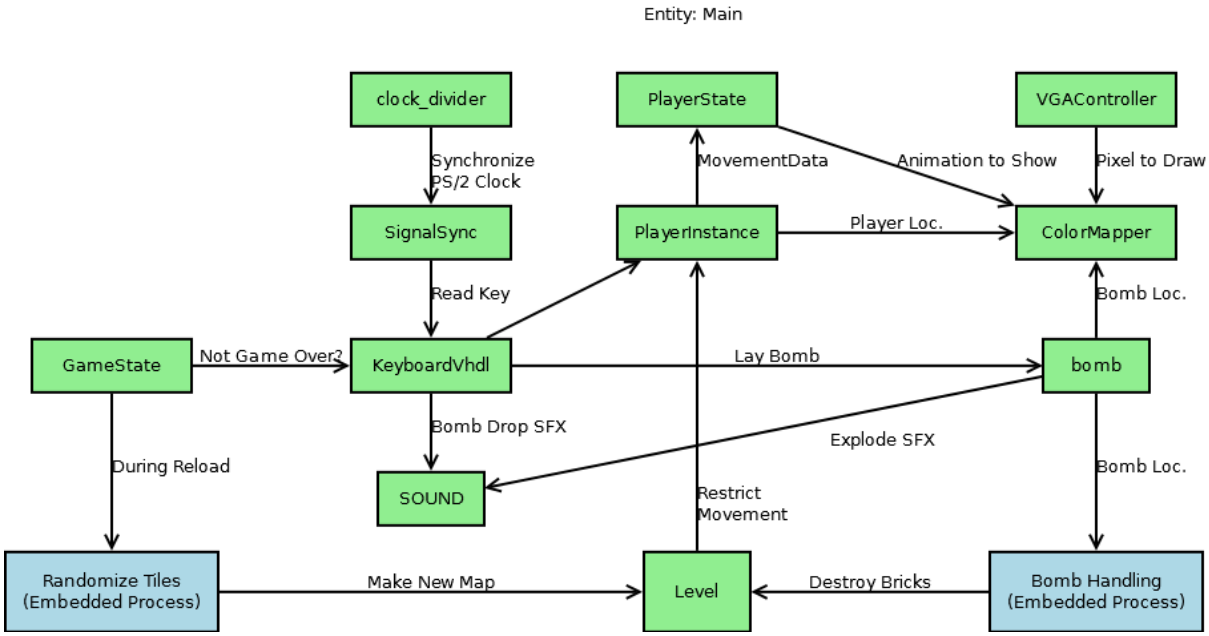



Figure 12: Mock Top Level Entity (Main)

Moreover, we also have the following state diagrams implemented in Bomberman. (audio_interface state diagram omitted because we did not change the code given to us and the entity's state diagram is extremely large).

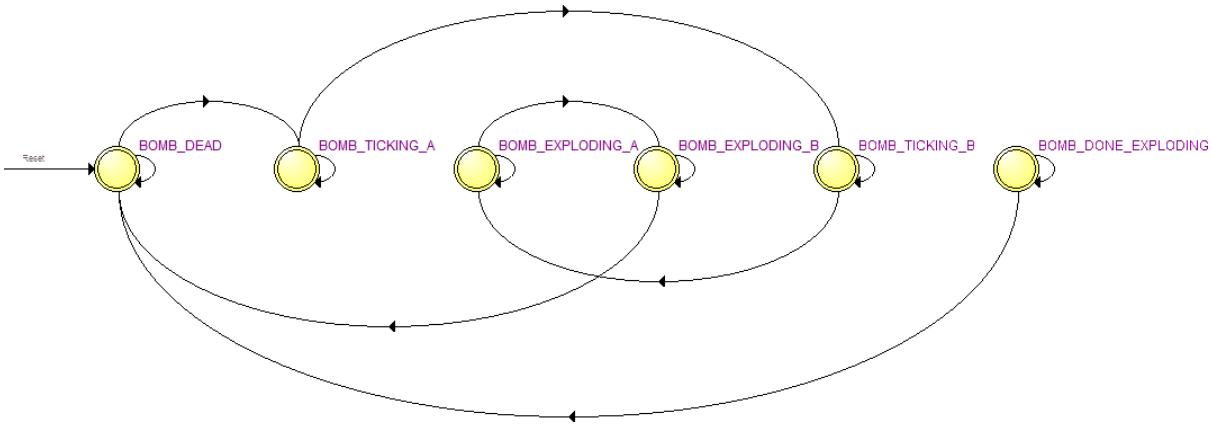



Figure 13: bomb State Machine

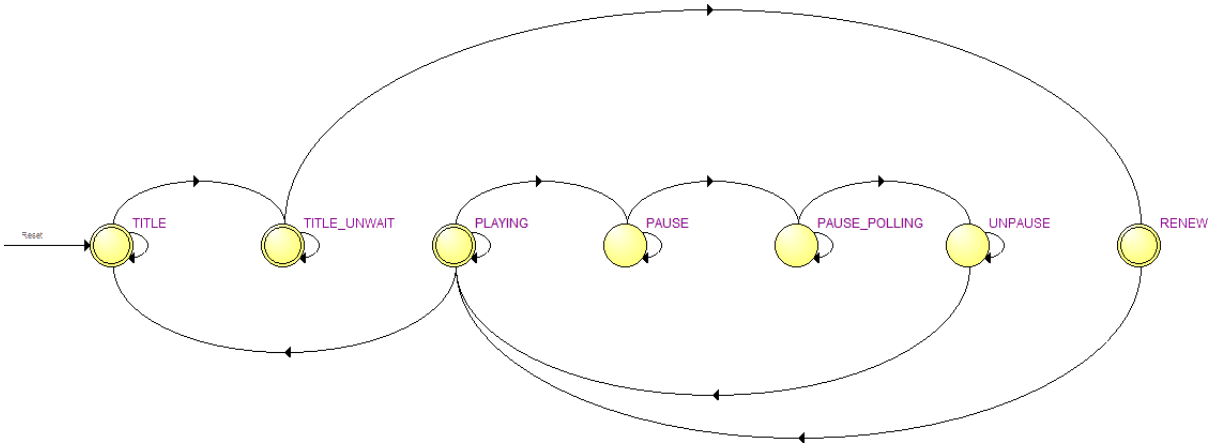



Figure 14: GameState State Machine

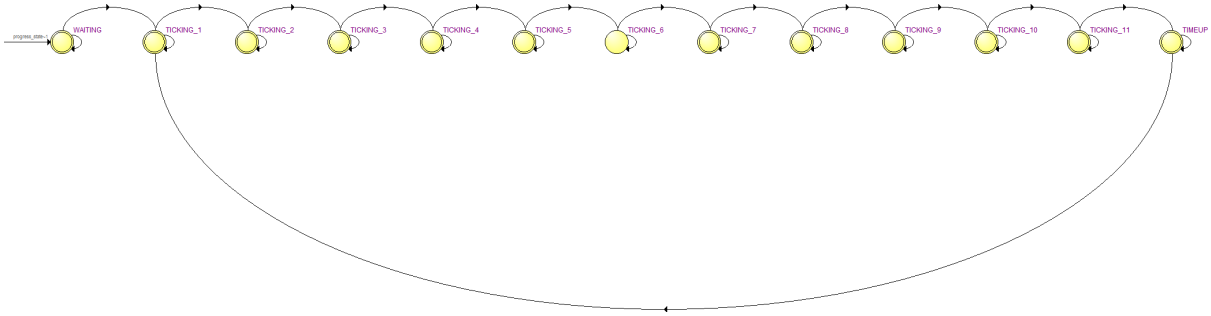



Figure 15: Main's Timer State Machine

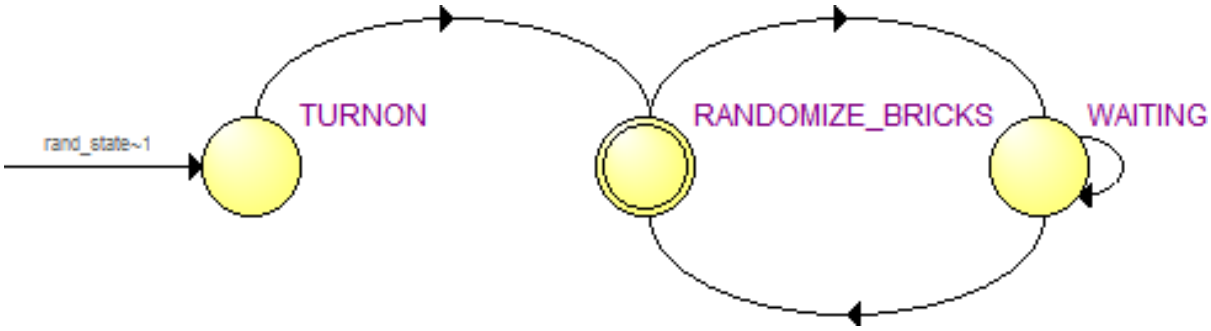



Figure 16: Main's RNG State Machine

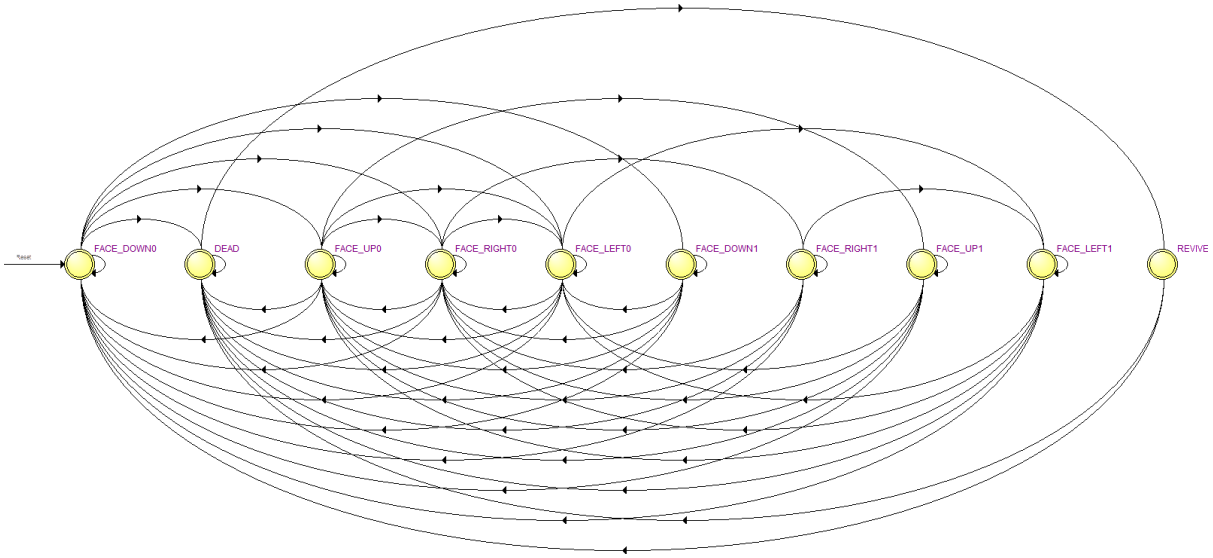



Figure 17: PlayerState State Machine

# 13 Conclusions

There were a lot of things we wanted to do but did not have the time to attempt to implement. For example, the Bouncing Ball code that we originally used as the base for Bomberman stored the location of the ball using the top-left corner of the monitor as the origin. Bomberman himself retained this representation because he was the edited from the ball's code, but everything else in the game that was not the bomb was taken to have an origin at the upper-left of the map. If all of the code worked with the map's origin, features would have been much easier to code and debug I believe. That is because math can happen simply on the logic side, but the renderer merely needs to offset what is actually stored in the memory. At the very least, we were consistent in programming all things to have their own origin at their upper-left pixel as opposed to their central pixel.

Overall, our project turned out well. We were able to implement most of the fundamental features that we had planned to do, and even did some extra features (animations, progress bar, etc). A lot was learned about design flow and the difficulties of managing a large VHDL project, such as long compile times. Wrapping up the project, we both feel a lot more comfortable working with VHDL.

The skills we have learned here will help us in future classes involving FPGA design. By putting a lot of effort into the final project, we are preparing ourselves for challenges that we may face down the road.