ECE 540 Final Project

Winter 2019

# Invaders from Space

Theory of Operations

Team 3 “Chaos CATs”

Brian Henson
Bradon Kanyid
Grant Vesely
Jamie Williams

## Project Goal

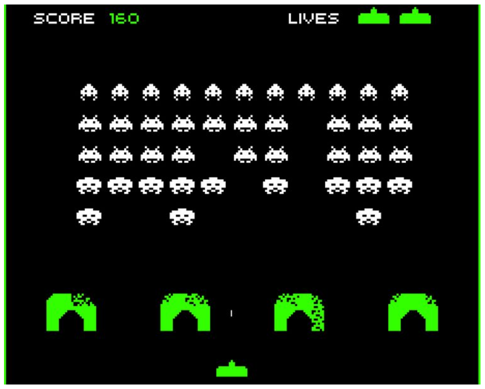
We decided to implement an upgraded version of the classic Space Invaders arcade game as our final project. Because Space Invaders is a video game, and not a formally-specified system that needs to implement certain features or interfaces to a strict protocol, any “upgrade” beyond a strict conformance to the original game would be somewhat arbitrary - however, we had to choose *some* “baseline” set of features, so we came up with the following:

- Player input via Atari paddle controllers
- 1024x768 VGA display output of the game: including the player, the enemies, and game progress information (e.g. score, lives, level, etc)
- Score calculation and display during the game (on seven segment display)
- Game options (if any) would be selected via the switches on the Nexys A7 board

We have successfully met or exceeded nearly all of our design goals. You can use an Atari paddle controller to move a ship horizontally across the bottom of the screen. You hide behind destructible bunkers as endless waves of enemies shoot at you and advance downward in a zigzag pattern. Shoot back to destroy the enemies and earn points, but as more enemies die, the remaining enemies move faster. With each new wave, the enemies shoot more rapidly. Bunkers/ score/ lives are not reset when a new wave begins, and if you die 3 times, you have to start over again. The whole system uses custom-made pixel artwork as well. The only stated goal we didn’t meet was displaying at 1024x768 resolution through VGA. Instead, our system displays at 640x480 (discussed further below).

We even managed to reach most of our stretch goals. Two-player games are supported, via a second paddle controller, and an (admittedly lackluster but functional) menu system is in place as well. After bootup, the system waits for any button press to begin playing waves. If the switches are all down when this happens, it is in 1-player mode; if any are up, it is 2-player mode. During a game, the score and wave number are tracked on-screen. One of our stretch goals, sound we were able to get working, but did not manage to integrate into the game by the end of the project.

A bonus goal we attained that we never even considered during planning was integration with an emulator using the SDL library, allowing the same codebase to be played on both a standard computer OS as well as on the FPGA board. This was extremely useful for debugging of software systems separate from the hardware.



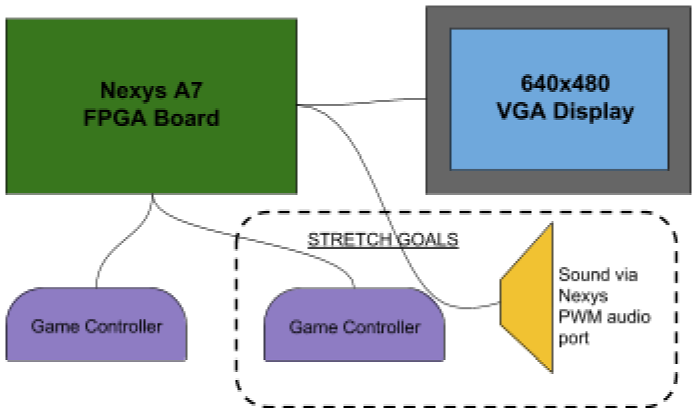
Left: The original Space Invaders Right: Our interpretation of the game

## Hardware

Our hardware goals were to create a platform consisting of the MIPSfpga soft cpu core, analog and digital inputs for controls, made up of the on-board switches as well as external Atari paddle controls, a vga controller that interacted with a dedicated sprite controller capable of displaying up to 128 individual sprites, with the limitation of a maximum of 16 sprites shown per line, and, as a stretch goal, also an audio/sound module.

A concurrent goal was to make all of the above available to the MIPSfpga via MMIO, as much as we possibly could. We believed that doing so would push a majority of the complexity to the MIPS software side, which is easier to debug and iterate on than specialized hardware described in Verilog that takes 15-20 minutes to resynthesize regardless of how much code was actually changed.

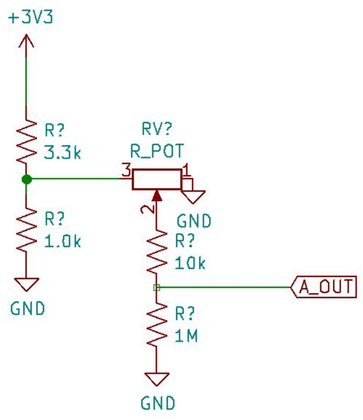
To that end, the sprite hardware has no concept of the graphics or behavior of the sprites it is displaying. Instead, the software is able to tell the hardware about the 128 sprites via a sprite table, and write the graphics into a section of RAM dedicated to sprite graphics. The combination of these two are used by the sprite rendering hardware to determine which pixels to send out to the multiplexer (basically the colorizer from Project 2). More details about the VGA/sprite hardware will be described in that section.



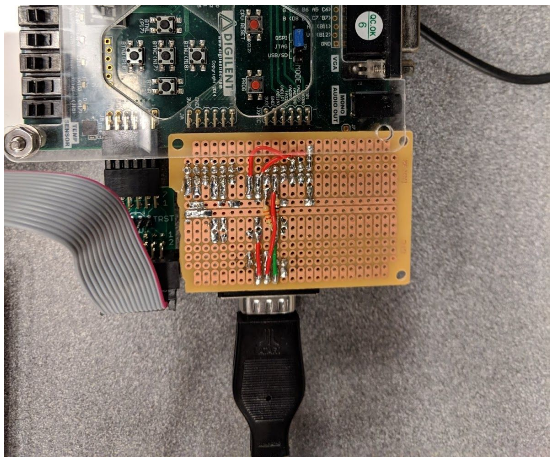
Very crude architecture blocks

## Paddle Input w/XADC - Bradon Kanyid

The XADC is an IP block in Vivado that offers up to 8 channels of ADC input at 12-bit resolution. We leveraged two of these channels, which are exposed on a special-purpose PMOD port on the Nexys A7 board. Each paddle controller consists of a ~1MΩ potentiometer, as well as a normally open button. The XADC block requires that incoming signals be biased to between 0-1V. Anything above 1V will be clipped and reported as max value (4095). The protoboard we made has a simple voltage divider at the frontend to do that biasing. The other resistors are to help scale the output to something reasonable, because the 1M potentiometer's output was very nonlinear without them.



Paddle input board schematic


Actual protoboard plugged into Nexys A7

## Video Output - VGA - Bradon Kanyid

The VGA controller consists of a very similar hierarchy of components as project 2. The output is driven by the colorizer, which now takes in a 4-bit value from the sprite controller, as well as a 12-bit value for the "background". The sprite value indexes into a fixed 16-color palette, which we defined ahead of time while creating our assets. The background is driven out directly, in the case there is no sprite pixel to display. Instead of a single "icon" as in project 2, there is a sprite controller that manages which icon to show. The only other thing of interest to mention is that we did reduce the resolution to 640x480 @ 60Hz. This required finding new constants that define the horizontal and vertical maximum values, as well as when to generate horizontal and vertical sync pulses.

The background generator was a very simple always block that just uses a portion of the y pixel position, and converts that into a 4-bit grayscale value. This made for an effective background that appeared similar to seeing the different sections of the atmosphere, and had the advantage of being very simple to generate. In hindsight, it was one of the only parts of our design that was not general. If given more time, we would have made a 480-element array that was MIP-addressable and had the CPU generate the per-line background color.

## Video Output - Sprite Controller - Bradon Kanyid

The reduction in resolution was partially to be able to lower the pixel clock to 25.175, instead of 75Mhz. Near the end, we were having timing issues, and this helped make timing closure. The timing issues were caused by the complexity of the sprite controller, which had just tons of wires and registers involved. The concept behind the sprite controller is that while there are 128 individual sprites to manage, there is a restriction that only 16 sprites can appear on any horizontal line. In order to determine this, a sprite table is traversed every HBLANK period, comparing every element's x-value and finding the 16 left-most items.

The sprite table consists of 3 attributes per sprite: x, y, and index. These attributes actually live as registers within the individual icon instances, and are writable via MIPSfpga via a secondary decoder that lives in the top-level. In order to allow simpler reads, the CPU interacts with a chunk of RAM when doing reads and writes, and the top-level secondary decoder snoops on these, determines if a write is happening for the sprite table region, and generates the correct signals so that the HWDATA signals propagate down into the individual sprite managed by the sprite controller. Since the Sprite Manager doesn't mutate any state, CPU reads of the sprite table come directly from RAM, which reflects exactly the same (synchronized) state as actually lives in the registers within the icons.

The x and y values in the sprite table are the location on-screen to show the sprite. The index is used to determine which sprite from the sprite palette to display. The sprite palette is a 4KB chunk of memory dedicated to sprite graphics. Each icon is 16x16@4 bits per pixel, so this is packed into 32 32-bit words, 2 words per horizontal line of the icon. The index is a 5-bit value that selects which sprite to display from the sprite palette. As mentioned in the VGA section, the 4-bit values in the sprite graphics data are indexes into a fixed palette that lives in the colorizer.

The sprite sorter creates 128 procedural blocks, where each manages a specific index of the sprite table. A load signal loads in the sprite table as it exists at the beginning of the sort, and this triggers the sort to happen. In fact, sorting is constantly happening, but once it is sorted, the next iteration of sort makes no changes. Individual x values aren't changing in the sprite table every line, but in order to only find the pertinent x-values, the icons report that their x-position is 640 (off-screen) if they are not in the vertically visible range. In other words, if the y pixel position is above the y position of a sprite, or below (y + 15) position of the sprite , it cannot be visible, so it is ignored by the sorter by pushing it off-screen. Every clock cycle, the sorter compares elements 0 and 1, 2 and 3, 4 and 5, etc, and swapping if necessary. Then a secondary pass happens that compares 1 to 2, 3 to 4, 5 to 6, etc. This double swap per cycle effectively means that for an array of N elements, it will be sorted in N/2 cycles. This type of sorting would be very/slow to do in software, but the parallel nature of hardware makes this efficient, at the cost of hardware complexity. For arrays larger than ours (say if we wanted to do 512 sprites), I'm not sure that this method would have been effective.

The sorted indexes of the sprites are then used to index the individual icon instances, which then output their sprite pixel. In the event that there are overlapping sprites, the earlier sprite table entry takes priority and wins out. There is a large assign statement at the bottom of the sprite manager that effectively determines if there is a sprite pixel active (which is used by the colorizer), and another large assign that determines which pixel to show. This took some abuse of the ternary operator, and may have been better suited to priority encoding, in hindsight.

## Audio - Jamie Williams

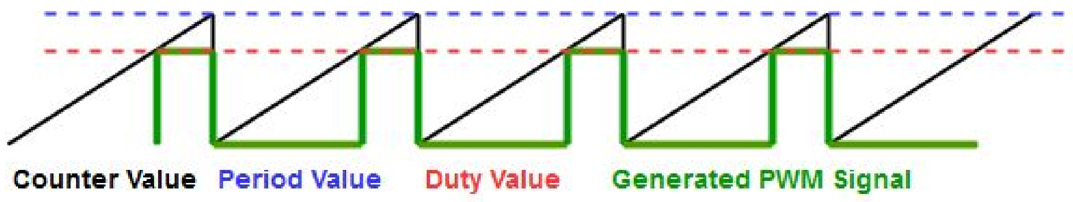
**Verilog Modules:**

One of the stretch goals that we had for the project as to add audio output to game, in the form of sound effects and/or background music. We decided to implement sound in hardware, using the Nexys board's built-in audio jack. The audio output of the board accepted a PWM signal as the input, and passed through a filter on the board before reaching the audio jack. This accepted two one-bit signals: the PWM input, and another signal that acted as an audio enable. (More information about the filter used can be found in the board reference manual located here: https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual#mono_audio_output )

To generate the necessary PWM signal to create output, we created a PWM generator module in Verilog. This module was designed to produce PWM output would produce a tone of a certain frequency (as a 1-bit reg output). A simplified version of the code is as follows:

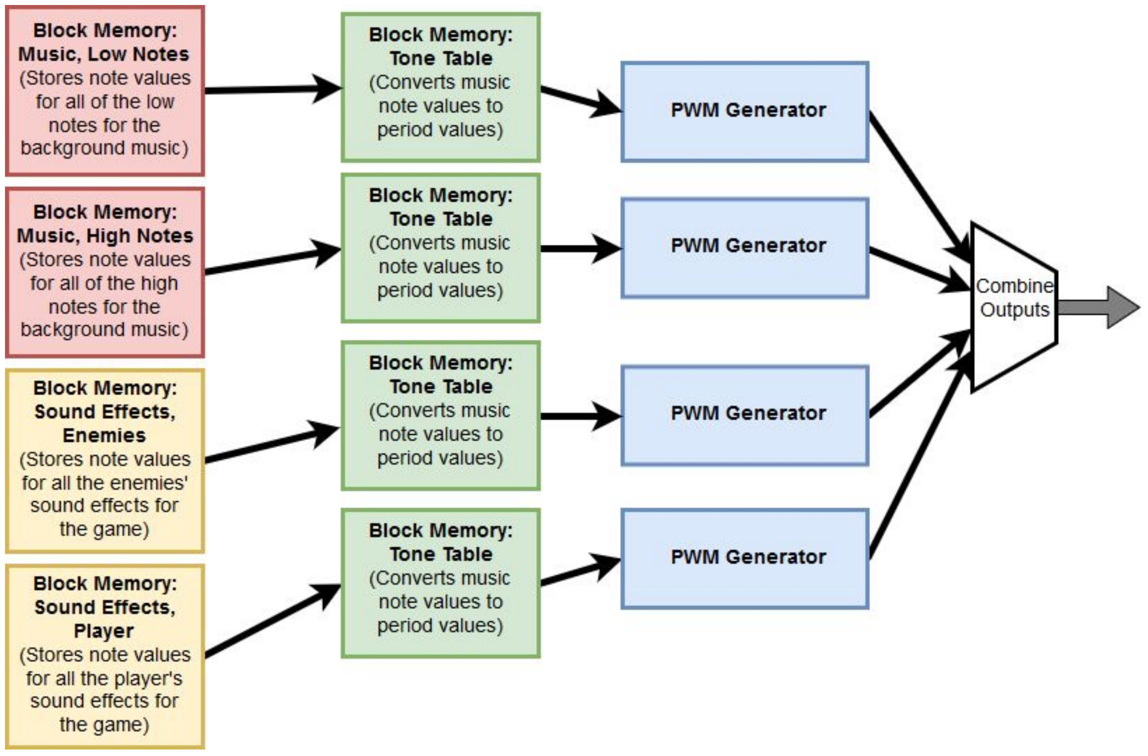
```
module pwm_generator(
      input          clock,       // 100 MHz clock
      input [31:0]   period,      // Determines the frequency
      input [31:0]   duty_period, // Determines the volume of the sound
      output reg     PWM_out);    // PWM signal
      reg[31: 0] count_val = 0;
      always @ (posedge clock)begin
            if(count_val < period)         // Increase and reset counter
                  count_val <= count_val + 1;
            else
                  count_val <= 0;
            if (count_val < hold_duty) // Output based on counter and duty
                  PWM_out <=1;
            else
                  PWM_out <=0;
      end
endmodule
```

The input “period” describes the maximum value of the counter, which is calculated by the following: **Counter Period = Clock Frequency/ Target Output Frequency**. So, for example, to output a frequency at 440Hz (an “A” note), the period value used would be 227,273 (100*10^6/440). The duty value determines the volume of a sound, and is just the period value bit shifted left by some amount (or, in other words, it is period value divided by some amount). Here is an illustration of the PWM wave generated:



The PWM generator module is instantiated multiple times inside of a sound generator verilog module. The outputs of the modules are combined (using OR logic) in order to produce a combination of tones simultaneously (as in a musical chord).

The sound generator module was created to provide input to the PWM generator module, and to control their outputs. In the sound module, there are several memory blocks, that work together to pass the correct period values to the PWM modules.

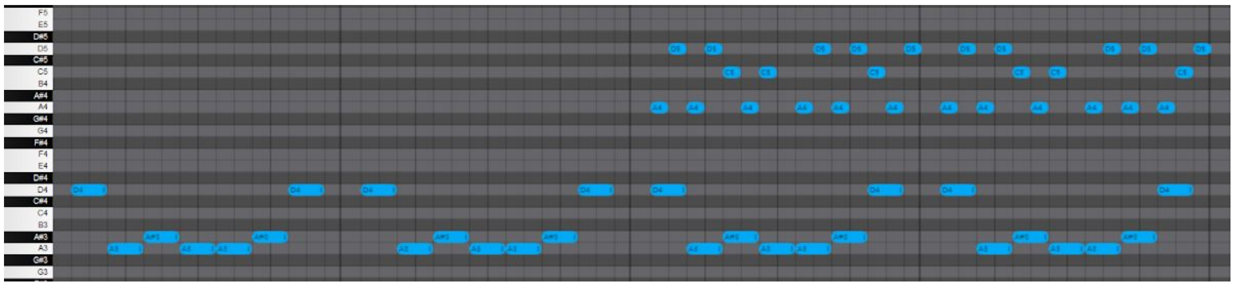

Block Memory: Music, Low Notes (Stores note values for all of the low notes for the background music)
Block Memory: Music, High Notes (Stores note values for all of the high notes for the background music)
Block Memory: Sound Effects, Enemies (Stores note values for all the enemies' sound effects for the game)
Block Memory: Sound Effects, Player (Stores note values for all the player's sound effects for the game)
Block Memory: Tone Table (Converts music note values to period values)
Block Memory: Tone Table (Converts music note values to period values)
Block Memory: Tone Table (Converts music note values to period values)
Block Memory: Tone Table (Converts music note values to period values)
PWM Generator
PWM Generator
PWM Generator
PWM Generator
Combine Outputs

The sound values are stored as specially formatted “music notes” so that they are better able to be read and modified by users. These music notes are translated to period values using a memory block called the tone table. Then, these period values are fed into the PWM generators. The music and sound effects are selected by reading the entries in the music or sound effect memories. The logic determines the starting value to read from (always at zero, in the case of the music memories) and the address accessed is incremented by a counter, which counts up continuously if enabled.

There are three separate counter for the sound module. One for the two parts of the background music (which are always in synch), and then one for each of the two sound effect blocks (which do not have to be in synch). The music operates at a counter that runs sixteen times a second (to be easily convertible into actual music note time scales), while the sound effects counters run at one hundred times a second (used to be convenient for the fast increases and decreases in tone used by the sound effects).

**Composition:**

For this project, I (Jamie Williams) composed a unique background music comprised of a high part and a low part, and designed to be repeated indefinitely. A copy of this music is stored as an MP3 file in the media resources folder in the GitHub repository (and can be seen in this video: https://www.youtube.com/watch?v=4iv5EspsE-w ). The music looks like this:



Brian composed some crude sound effects for the player exploding, the player shooting, and an enemy exploding. Each is simply a starting frequency that decreases by a set amount every 0.06, 0.02, or 0.03 seconds (respectively) before stopping.

**Integration Challenges:**

Unfortunately, sound output did not make it into the final version of the project. Although the sound was working when in it's own project, it broke the AHB/JTAG system when combined with the rest of the the Verilog code, preventing us from loading our software into the program RAM. We did not have enough time to find the problem and fix it before the project was due.

My best guess is that the block memories interfered with the memory accesses between the game software and the VGA modules. However, I was not able to confirm that.

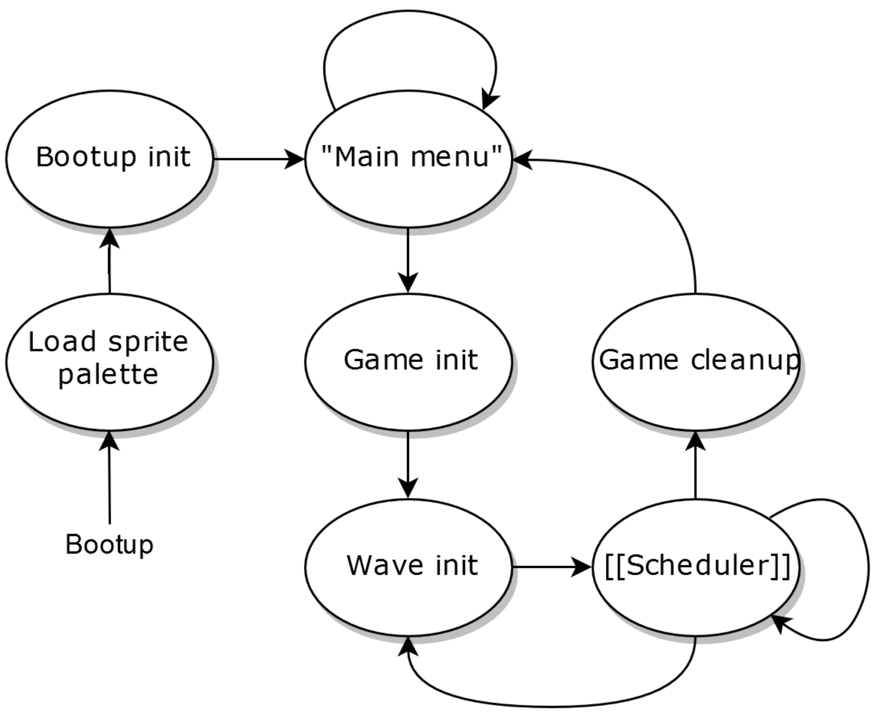
# Software

Our project is, in total, over 3,000 lines of C code. It is highly parameterized with #define statements, to the point that nearly everything related to the layout of on-screen objects and the timing of all evens is easily reconfigurable from a single place, "params.h". Similarly, the file “mfp_memory_mapped_registers.h” contains many more #define statements and macros that are used to access the memory-mapped peripherals on the MIPS core. This file originally came from MIPSfpga-plus on Github, but it has been heavily modified and expanded for our system.

The file pair “util.c/h” contains what you would expect: a series of application-generic functions used throughout the rest of the project, such as clipping an input value to fall within a specified floor and ceiling, or converting an input to base ten for displaying in the real world. The file pair “interface.c/h” contain functions for accessing the memory-mapped peripherals or the SDL2 backend, depending on the compilation mode. The file pair “load_sprites.c/h” contains the arrays of sprite pixel data to be written to the palette RAM as well as the functions for doing so. All “*.S” files are assembly code provided with the original MIPSfpga package and untouched by us.

## Game Logic - Brian Henson, Grant Vesely

The remaining files to discuss are “main.c”, “game.c/h”, and “wave.c/h”. These contain/implement the actual game logic, though the vast majority of the code is in wave. “Main” takes care of loading the sprite palette, bootup init operations, and running the main menu. Once an option (1-player or 2-player) is selected, it moves into “game”, where game-start, game-end, and most wave-start init operations are performed. “Game” is also responsible for preserving the score/lives/bunkers from wave to wave, and looping and running waves until the player has a Game Over. “Wave” contains all the game logic for running a single wave of enemies: the scheduler loop that keeps track of upcoming events and when to run them, as well as all the functions that perform those events. The overall program execution flow is illustrated in the grossly simplified flowchart below.



Simplified high-level program flow

In main.c, the sprites are loaded into the sprite palette RAM, and all bootup init operations are performed. Notably, this means setting the x-values of sprites that never move along the x-axis, as well as setting the palette indices for sprites that never change or animate. It also hides all sprites by moving them below the bottom edge of the screen, since the main menu is intentionally blank. It then waits at the “main menu”, flashing the LEDs until any button is pressed. It also continually displays the raw XADC input readings of both paddles during this waiting period; this was mainly used as a debugging aid. If all switches are low at that moment a button is pressed, it runs 1-player mode; otherwise it runs 2-player. It runs a game by calling play_one_game(int numplayers).

In game.c, the sprites are all manipulated to transition from the blank main menu screen to display a normal wave: score, bunkers, players, extra lives, and labels are all set to the correct x and y position and the correct palette index. It then enters a loop where it stays until the player gets a Game Over. On each pass, bullets are cleared, enemies are populated and positioned, the wave # is incremented and displayed, and the wave-start fanfare is played. The game is completely (visually) set up to run, but pauses briefly until the fanfare is done. Then it calls run_one_wave(int wavenum, int numplayers). Once the player gets a Game Over, the game cleanup operations begin. The Game Over sound is played, and the game is still visible & frozen while it plays. Then it simply hides all sprites by moving them off the bottom edge of the screen, and returns to the main menu.

When a wave begins, the scheduler loop is initialized. The smallest unit of time we need to be concerned with in our design is 0.005s, or 1/200th of a second, henceforth a “gametick”. This is measured via reading a continually-running timer peripheral added to the AHB system.

The “gametime” represents the number of gameticks since the wave began. All events are scheduled to run by setting a gametime value at which it will run next, henceforth a “target” value. Both repeating and non-repeating events are handled by this same system. On any given gametick, any number of targets might match the gametime and execute their respective tasks.

There are 7 different events/sets of targets that are scheduled this way. First I will describe the non-repeating events. These are: 1) animate enemy explosions (each enemy has a separate target value); 2) manage the player shot cooldown (each player has a separate target value); and 3) animate the player explosion (each player has a separate target value). The player explosion animation also takes care of respawning operations, and causes the player to lose the game if they attempt to respawn but have no lives.

Next I will describe the repeating events. Number 4 is labelled “move player”, and it updates the players’ X positions to match the XADC inputs, but it also handles reading the other inputs as well: shooting, pause, and self-destruct (previously “quit”). Shooting is triggered via the paddle buttons. Pause is mapped to the UP button on the Nexys board, and the up arrow on SDL2. Self-destruct is mapped to the DOWN button on the Nexys board, and the down arrow on SDL2. Both pause and self-destruct (and unpause) are activated on their respective button releases. This is intentional, because if pause were state-based, it would trigger multiple times in sequence and cause pause-stuttering. Finally, this event also runs a check of whether the player’s new position is overlapping with an enemy bullet, in the function collision_ebullet().

Number 5 is “move bullets”, and it does exactly that. It moves enemy bullets down, player bullets up, and hides them if they move off the screen. It then checks all forms of collision, by running collision_ebullet() (player-to-enemybullet), collision_pbullet() (enemy-to-playerbullet), and collision_bunker() (bunker-to-anybullet).

Number 6 is “move enemies”, and it does exactly that. The zigzag enemy movement is calculated, applied, and then collision_pbullet() is ran. This also handles destroying the bunkers or causing a Game Over if the enemies get too low on the screen. Notably, the time between the executions of this block is not constant; as enemies die, the delay between executions decreases.

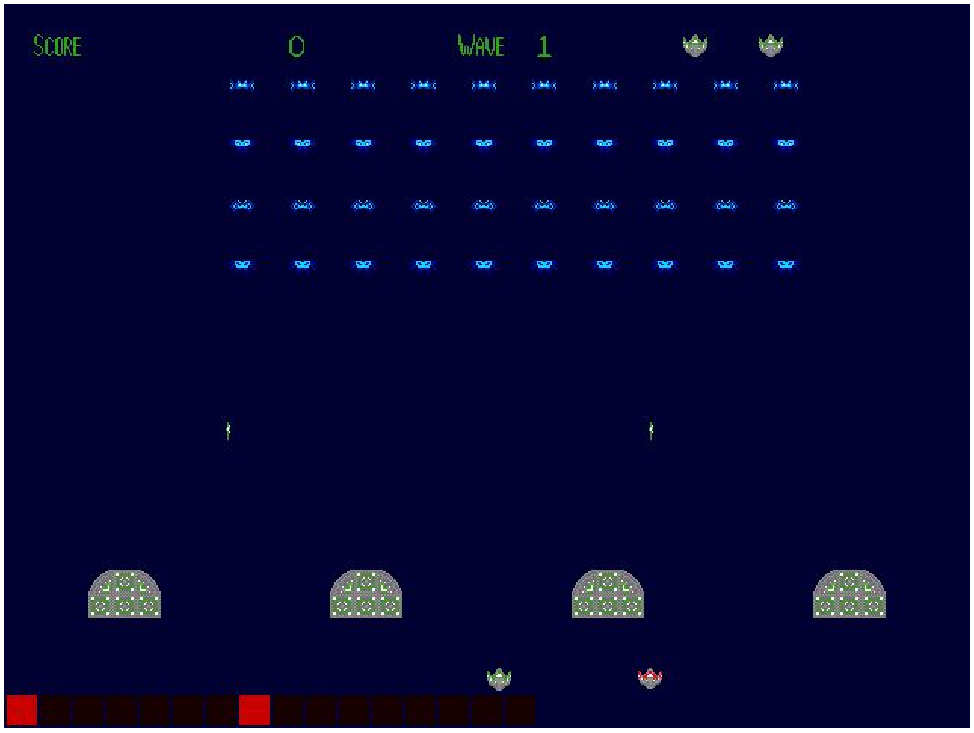
Finally, number 7 is “enemy shooting”, and this is also not a constant frequency. The time between one execution and the next is randomized between some floor and ceiling. What happens in this block is that either 1 or 2 columns that contain living enemies are chosen, and then the bottommost living enemy in either column generates a bullet. Very simple.

Overall, this code base is massive & seems complex at first, but excellent commenting, encapsulation with functions, variable names, and parameterization makes it surprisingly readable. Frankly, I’m very proud of every aspect of it (planning, execution, annotation, features) and encourage you to look through it if you want to understand it better.

## Emulator - Grant Vesely

Early on in the project, Bradon came up with the idea of writing a software “emulator” for the hardware, so that we could start testing our software before the hardware was ready. This turned out to be a very wise idea, as we didn’t get fully-functional hardware until the evening before the demo - however, because we had been testing our software game logic against the emulator, we were able to completely finish the game within a mere six hours of the hardware being completed.

The emulator was built using the SDL2 media library as a backend, which exposes a C interface through which you can create windows, get input, display graphics, play audio, and do a variety of other useful things in a platform-independent fashion. Code compiled against SDL2 runs on Windows, macOS, and Linux with only minimal modifications, which was very useful for our team as two of our members were running Windows, while the other two were running Linux.



A screenshot from the emulator. The red squares represent hardware LEDs.

The architecture of the emulator itself was fairly simple: we wrote functions that abstracted all of our hardware-specific logic, such as the code that would read from MMIO registers to get the currently-pressed buttons, into C functions (say, read_butts()). We then wrote both code that interacted with SDL2 (for instance, by calling the function that returned the state of all buttons pressed on the keyboard, and then re-structuring that return value into the form that the game logic was expecting), and code that actually used the hardware features of the MIPSfpga, and used #ifdef “fences” to force only one implementation to be used for a particular compile target. During compilation, defining the SDL2 preprocessor macro would cause the SDL2 implementations to be used in the bodies of those functions - if the macro was left undefined, the “real” code was used instead. Aside from a few #ifdef guards around code not in functions (such as to customize the constants used for the game loop timing calculations, as our hardware timer had a speed of 50 MHz, while our software timer had a resolution of 1 KHz; or global variables to store pointers to SDL2’s context data for holding the emulator window), that’s all there was to our emulator.

## Sprite Design - Jamie Williams

For our game, Invaders from Space, one of our stretch goals was to have better graphics and animations than the original arcade game. Therefore, one of the things that had to get done was designing new sprites for all of the necessary game elements.

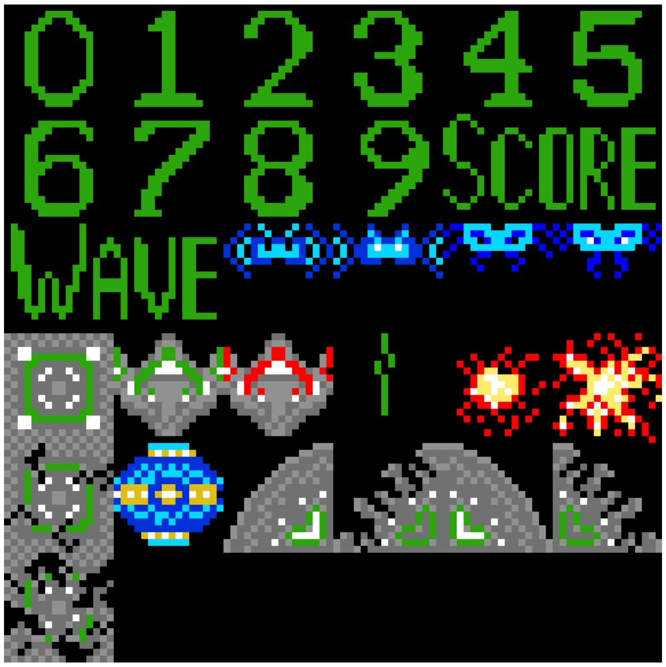
Due to the style of VGA output and sprite controller design we were using, there were several constraints to our graphics design:

- All sprites had to be made of 16x16 pixel blocks (Larger designs, like the “Score” and “Wave” labels, were made of two of these sprite blocks next to each other)
- There could only be 32 unique block designs (The maximum number that would fit in the sprite table used by the sprite controller)
- There could only be a total of 15 colors (plus a value transparent) used in the sprites (As there were only four bits of color data saved per pixel)

Each sprite was made using the web application Piskel, and then exported as a C array. The array’s data was then modified from the full hex color down to a index value for the color palette. Then, these arrays were put into the C file called “load_sprites.c” which stored the arrays and moved them to a memory location where they could be accessed by the sprite controller.

I created 31 unique sprites for the game, and used only 9 of the possible 15 different colors. The list of sprites created for the game are:

- Two different player ships (green, for single player, and a red for a second player)
- Two types of swarm enemies, each with two frames of animation
- One special enemy with no animation (unused in the final game)
- Ten digits for the score and wave count (0 to 9)
- Four pixel blocks for the “Score” and “Wave labels (two blocks each)
- One explosion animation, with two frames
- One bullet (used for both the player and the enemies)
- Seven blocks for the bunker tiles
    - A solid block, with three frames of animation
    - Two curved blocks (left and right), each with two frames of animation



Compilation of all 31 sprites created for the game. All but the special enemy were used.

## Demonstration of Completed Project:

https://www.youtube.com/watch?v=m1VFT0_nl0g

## Miscellany

### Issues:

There are only 3 known bugs remaining with the design. Firstly, the wave counter is only a single digit, and we currently have no good way of supporting/displaying waves 10 or beyond. Instead, it keeps proceeding through the sprite palette until it eventually wraps around back to 0 at wave 32 and 1 at wave 33.

Secondly, the enemy movement pattern is imperfect. There is no distinction between moving downward along the right side of the screen versus the left side; when it is done moving down, it simply checks whether the leftmost living enemy is all the way to the left , and if yes, it begins moving right. If not, it begins moving left. Because of this logic, killing the leftmost enemies as it is moving down along the left edge will cause it to move left again instead of heading right. Even when the bug occurs, however, the game continues to be playable, so fixing it was a low priority.

Lastly and most importantly, there are some number of flaws in the sprite controller logic that we didn't have time to resolve. This manifests as a 12-sprite per line limit. If there are 13-16 sprites on the same horizontal line, the 13th through 16th will be very glitchy and unstable. We were able to accomodate this error by simply reducing the number of bunkers for the player to hide behind, so that only 9 bunkers + 1 player shot + 2 enemy shots = 12 sprites would be displayed on the same line at a time. If the player's score exceeds 99,999 points without losing a life, however, 13 sprites will be displayed on the header line and the second life indicator will be affected by this bug.

## Feature "near misses":

As it stands, our "menu" is basically nothing more than an extra state in the code that prevents the game from starting until a button is pressed, and checking to see how many players should be added. If we had had more time, we would have attempted to implement a full graphical main menu that allowed you to use the paddle controllers to select the number of players, and maybe display a high score.

Although we actually had a working hardware sound module, we weren't able to integrate it with the rest of our design before demo day. It was a PWM sound module, so we wouldn't have been able to play traditional sound clips, but if we had gotten integration working, we could have added a few basic bleeps and buzzes to the game alongside MIDI-style tone generator music.

Finally, all of our enemy waves were identical, except that their firing rate increases as you get to higher levels. If we had had the time, we would have implemented a unique enemy ship with novel movement and/or firing patterns to make the game more interesting.

## Hindsight:

The reason we decided to use software-loaded sprites was that it would give us faster turnaround time for tweaking and changing the artwork. We never took advantage of this upside, because we liked the way the sprites looked on its first iteration. Therefore, rather than try to implement this flexible reprogrammability, we could have done the artwork hardcoded in block RAM and saved effort in both the hardware & software development.

We started on the emulation code a significant amount of time after we had started writing the game logic, such that we had several hundred lines of code that needed to be tested by the time the emulator was actually functional. Additionally, because the code was structured so that the entire game logic was implemented all in one go, this lead to more time spent debugging and writing "stub" functions than if we had started out with "the simplest thing that works" (initialize the emulator/MIPSfpga) and then incrementally grown it by modifying the design (add the timing loop, then make sure that the timing loop works with one target, then multiple targets, then aperiodic targets, then draw a single sprite to the screen with a hardcoded position, then connect the position to code in one of the targets, then put it in an array, etc.) and ensuring that the code was completely correct at every iteration. In short, we “took too large of a bite” of the project before testing, instead of writing & testing smaller portions one by one.

We also didn’t make the emulator emulate hardware at as low of a level as we could have. For instance, in hardware, the sprite table (which stores information on a single sprite to be drawn to the screen) is MMIO registers, and the software does “game-y” things, such as checking for bullet/player collisions, directly on these registers. As we accessed this array through preprocessor constants, we could have used #ifdef fences to actually create an array behind that constant for the software implementation. Instead, we defined a bunch of helper functions that essentially just read or wrote sprite data, adding unnecessary complexity to our code.

## Team Picture:

Team members from left to right: Bradon Kanyid, Grant Vesely, Brian Henson, Jamie Williams