CSE 111 Team 9 Report

Contributions and Highlights

Ryan Welter

OS I made a significant amount of progress implementing the operating system and organizing the workflow that the CPU/GPU go through to process the .slug files. This meant that I had to become very familiar with how the .slug files are organized, how the console handles instructions, and how to initialize the system.

One particular area I spent a lot of time on was organizing how each part of the emulator interacted with each other. As everyone was working on a different component of the system, we ran into some issues where the CPU/OS would need to read/write memory in specific ways, for example when initially running the reset sequence. Our priority was that there would be enough public functions in the memory class for these needs, and enough private functions to abstract features away from classes outside of memory. At that time, our current memory implementation didn't fit this guideline, so I dug into these issues and helped organize the current implementation we have for memory access, although Michael was more involved with implementing the memory class itself.

Disassembler The extra feature I spent the most amount of time on was implementing a disassembler. This runs without running the actual console itself, breaking down the .slug files into human-readable text without spending much computing time. My thought process for picking this feature was that I had already spent a lot of time analyzing the raw .slug files and how they were structured with the OS, and so I was able to cut a lot of planning/organizing time out of the development process.

The major struggle with this feature was correctly parsing through the instructions outside of the setup and loop functions. It was originally tough to organize these instructions in a way that was understandable, since these instructions didn't belong to named functions and were only referenced by jump-type instructions. I settled on just listing their individual addresses in memory and making sure that any jump-type instructions had their immediate values printed clearly, if applicable.

Some other key details of this feature is that the "disassemble" button in the GUI only activates once a valid .slug file is selected, there are maps specific to the disassemble.cpp file that match opcodes and function codes to instruction names as strings, and some extra parsing to export the copyright message.

Troubleshooting/Cleanup Throughout the project, I was involved in keeping the code clean, removing magic numbers, double checking variable/function names, and removing redundancies. We had a period after check-in #1 where I

made a "cleanup" branch and cracked down on these bad practices so that we could have all good practices from that point forward.

Wyatt Avilla

CPU I'm particularly proud of the CPU, where I was able to offload a significant amount of computation to compile time while still maintaining readability and adherence to the spec.

Specifically, I used `contexpr` to build a jump table of the various CPU methods. Avoiding run-time initialization meant that the compiler was able to help us catch bugs with the table implementation that startup time for the emulator was faster.

In terms of readability, I used `std::variant` to allow methods with different signatures to be placed into the same jump table. This meant that the CPU method implementations matched the spec exactly while still being callable in a general way. For specific details, see `cpu.h` and `cpu.cpp`.

GitHub Actions Our project has a robust set of checks actualized by GitHub Actions. Firstly, everything in our repository is formatted. YAML and Markdown files are formatted with Prettier, and C++ source files are formatted with clang-format. Additionally, the correctness of our emulator is assessed with a compilation check followed by a set of tests against the provided "hello world" slug files. Then, our project is compiled with `-Wall -Wextra -Werror` to ensure that no compiler warnings are present. Finally, clang-tidy is run with close-to every lint rule enabled to help guard against potential bugs.

Build Reproducibility We used Git submodules to pin the exact version of the provided starter code and each of our dependencies. Although Cmake provides a `find_package` function, Git submodules allow for a finer-grained control over libraries that's agnostic to distro package managers. This approach ensures that all teammates (and potential users) utilize the same library versions and build the same binary.

Additionally, building from source allows us to manipulate the build parameters of our dependencies. If necessary, we could compile our dependencies with debug flags to give us richer stack traces or with thread sanitization to ensure concurrent safety.

GitHub Issues Integration I used GitHub issues to keep track of bugs, in-progress features, and to-dos. The "tags" and "assignees" features made it extremely easy to see everyone's contributions. For example, you're able to see who worked on what features by filtering closed issues by the "extra features" tag.

Another highlight of using GitHub issues was how easy it made checkoff 1. Opening an issue for the checkoff meant that Charles could directly reference places in our codebase that needed to be changed. Further, it gave us a centralized place for a to-do list that all team members could modify and allowed us to track the changes related to the review.

Monisha Garika

Controller For the game emulator project, I worked extensively on implementing the controller functionality in `controller.cpp` and `controller.h`. The controller was designed to handle user input through the keyboard using SDL (Simple DirectMedia Layer).

The core functionality is implemented in the `updateController()` function, which reads the current state of the keyboard using `SDL_GetKeyboardState()` and updates the state using bitwise operations. Each key press updates the corresponding bit in the controller_state variable using logical OR (|). If the state changes, it is written to memory at address 0x7000 using the `console->memory.w8u()` function, which integrates the controller with the emulator's memory system.

To assist with debugging, I included logging messages in the `updateController()` function:

I logged the memory address and value written to memory using `std::cerr` to track changes to the controller state. I also used `displayControllerState()` to print the current state of the controller in binary using `std::bitset`, which helped in identifying bit-level issues during testing.

GUI Color Selector In addition to working on the controller, I also implemented a color selection feature for the graphical user interface (GUI). The color selector allows users to customize the color of moving parts in the game after the execution stage begins. This was integrated into the `onExecute()` function in the GUI component.

To implement this, I created a `wxColourDialog` to present a color picker to the user. The selected color is retrieved and passed to the `console->filter.setColor()` function to update the game's display. I also added logging using `std::cerr` to track issues with creating the color dialog and to confirm the color selection process. If the user confirmed the color, the selected RGB values were applied to the game's display filter.

The color selection feature significantly improved the user experience by allowing customization and better visual engagement during gameplay.

Problem With Color Setting Initially, the color picker was working correctly, and the RGB values were being read and set, but the graphical elements still displayed the default gray color. This happened because SDL textures don't

automatically reflect color changes unless explicitly updated. After setting the color, the texture needed to be locked, updated, and rendered for the change to appear visually. Fixing it involved updating the texture using `SDL_RenderCopy()` and refreshing the renderer.

Michelle Gurovith

Extra Feature: Video Recorder I am extremely proud of implementing the video recorder feature, which allowed the game to be recorded and viewed as a video. I used SDL(Simple DirectMedia Link), which is a type of graphics library that we discussed that renders VRAM content. The way this works is that it creates an SDL window, and `startRecording()` starts enabling recording, and `stopRecording()` disables recording. Then, `saveRecording()` and `loadRecording()` allow the video recording to be saved and watched.

My favorite part of this feature was creating a movable progress bar, where the user could move a yellow box at any point in game execution. This works because when a user clicks the progress bar, the `dragging_process` is set to true, which allows the user to adjust the progress box to their liking. Additionally, I added a button called View Recording, which, after the execution of the game, allows the user to press the button to see the video. The View Recording button gets temporarily disabled after the user confirms that the next slug file that they choose is the one that they want. Once the game execution is finished, the View Recording button is enabled.

I had to fix an issue regarding chosen colors not showing up as needed when the game video was played. I realized that the issue was due to the `VideoRecorder` class not storing information regarding color, and there wasn't a way for the color information to go from GPU to the video recorder. The way I fixed this was by creating `filter.cpp` and `filter.h`, where I made a separate Filter class where color information was stored. Then I made sure convertFrameToRGBA() uses the filter's colors in the code. Then, in the `loadRecording` and `saveRecording` methods, I made code to save and load the color information. By fixing this issue, when a person picks a color from the catalog, it is reflected correctly in the video recording.

GPU I was responsible for making the GPU portion of the Banana Emulator. For the emulator to show the games in motion, I used SDL. The way I created the GPU included the initialization process, which included initializing SDL for rendering, creating a window, and creating a renderer and texture to hold pixel data. After this, the rendering process involves checking for interactions with the user, such as checking if the window is closed. It also included getting pixel data from VRAM, converting grayscale pixels into ARGB format, and rendering each frame onto the screen.

GUI and Pixel Filters I helped enhance a couple of GUI and pixel-related parts of the final project. One thing I decided to do was to disable the color black from being used in the game. The reason why I wanted to do this is because I noticed that whenever I would apply a solid black color to be my pixel color, the color-changing pixels would fully blend in with the black background, which was problematic. I also ensured that the default color would be grey on the color catalog so that if a person didn't choose a specific color, grey would automatically be used in the game. Additionally, I created a visual message that the solid black color cannot be used in the game if someone decided to press on solid black. The message then redirects the user to the color catalog to choose a different color for the game.

Michael Kamensky

GUI (My Proudest Work) I am most proud of my work on the GUI. Since we use wxWidgets, I followed my group's precedent and did a git submodule for the library. This was tricky because the library has many more dependencies than SDL, and I had to modify the CMAKE code significantly, including multithreading for building and caching, so the program could build in a reasonable amount of time. After adjusting the cmake to include wxWidgets, I started writing the code. My first attempt failed. I did not know how to use wxWidgets and could not compile the code with the competing main functions.

Eventually, I settled for this type of code structure. I decided to use a class structure for the code. My class initializes a window with a dark gray theme, featuring a title, an image, and three buttons: "Select File," "Execute," and "View Recording." The code is structured with a class-based approach, where the `MyFrame` class handles the GUI layout, event bindings, and user interactions. The GUI dynamically adjusts its design based on window-resizing events, ensuring a responsive user experience. Furthermore, the application includes mechanisms for file selection, color customization, and video playback, enhancing usability and interactivity.

It was cool to see my work visually. When working with memory, I could not see the differences in my code, even though it was essential and functional. It was cool to see the window, the buttons' color, and the themes that I added get applied to the end product. It made it more fun to work on and explore possibilities

Memory I was responsible for making the Memory; I followed the code and precedent set by the professor in one of our classes. This Memory class does a solid job handling memory operations while also considering endianness, which is crucial when working with low-level memory management. How we deal with endianness is particularly interesting in functions like `l16u`, `l32u`, `w16u`, and `w8u`.

When reading multi-byte values (16-bit and 32-bit), we manually reconstruct

them byte by byte using bit shifts, ensuring consistency regardless of the underlying system's endianness. For example, in `l32u`, we load four individual bytes and shift them into their correct positions, enforcing a little-endian layout. Similarly, when writing 16-bit values in `w16u`, we break the value into two bytes and store them separately, preserving the intended byte order.

This approach guarantees that our memory operations remain predictable across different architectures, preventing issues where byte order might otherwise cause misinterpretations of stored values. It's a simple but effective strategy to keep our system clean and effective.

Build Reproducibility As mentioned above in the GUI section, I followed Wyatt Avilla's precedent and used a GitHub module to import and build the wxWidgets library. This ensures that future users have access to the correct binary libraries that our program needs with no extra effort from the user.

Extra Features

Disassembler

Sample of disassembled Slug instructions.

```
8200        ADDI ZERO,r1,128
8204        AND r4,r1,r2
8208        JR r31,ZERO,ZERO
820c        SLL ZERO,ZERO,ZERO
8210        ADDI ZERO,r1,64
8214        AND r4,r1,r2
8218        JR r31,ZERO,ZERO
821c        SLL ZERO,ZERO,ZERO
8220        ADDI ZERO,r1,32
8224        AND r4,r1,r2
```

Graphical User Interface (GUI)

Pixel Filter

Game Recorder and Playback

File Organization

Our file organization strategy is straightforward. From the top down, we used the `build/` directory to organize cmake-related build files and to keep them easily git-ignoreable. Next, we used the `external/` directory to keep the git submodules for the external libraries we depend on. The `src/` directory (of course) contains our source files. Also, we used the `report/` directory to organize the automatically generated report pdf and it's associated images. Finally, the starter code for the project is included in `starter-code/` as a git submodule for easy reference and use by the CI.

Figure 1: Graphical User Interface

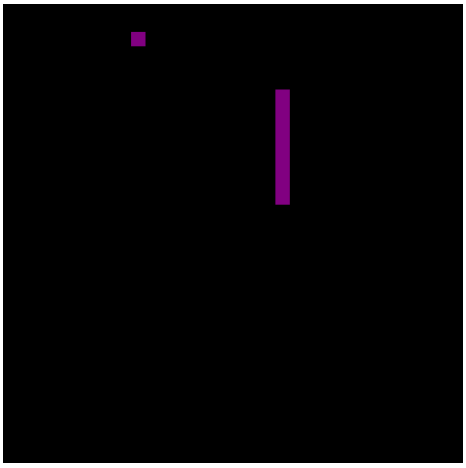

Figure 2: Snake With a Purple Pixel Filter Applied

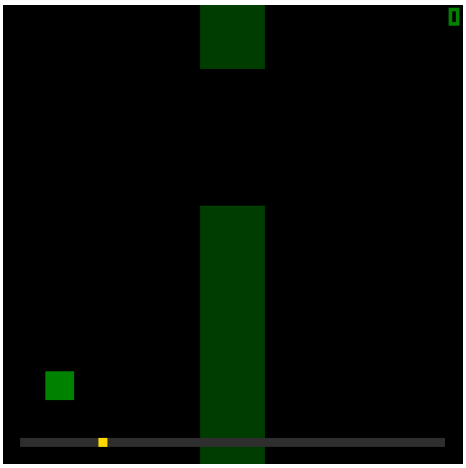

Figure 3: Playback of a Recorded Flappy Bird Game

File Tree

```
.
├── build
│   ├── CMakeCache.txt
│   ├── CMakeFiles
│   ├── cmake_install.cmake
│   ├── compile_commands.json
│   ├── emulator
│   ├── external
│   └── Makefile
├── CMakeLists.txt
├── external
│   ├── SDL
│   └── wxWidgets
├── helper_tools
│   └── opcode_analyzer.py
├── report
│   ├── report.md
│   └── report.pdf
├── readme.md
├── src
│   ├── banana.png
│   ├── bit_definitions.h
│   ├── console.cpp
│   ├── console.h
│   ├── controller.cpp
│   ├── controller.h
```

```
│   ├── cpu.cpp
│   ├── cpu.h
│   ├── disassembler.cpp
│   ├── disassembler.h
│   ├── filter.cpp
│   ├── filter.h
│   ├── gpu.cpp
│   ├── gpu.h
│   ├── gui.cpp
│   ├── gui.h
│   ├── main.cpp
│   ├── memory.cpp
│   ├── memory.h
│   ├── os.cpp
│   ├── os.h
│   ├── vr.cpp
│   └── vr.h
└── starter-code
    ├── bananaslug_documentation.pdf
    ├── games
    ├── gpu
    ├── hws
    ├── LICENSE
    ├── README.md
    └── tests
```

Cmake Flags

The CMAKE_BUILD_TYPE flag allows users to select the desired build mode. Setting it to Debug enables debugging symbols (`-ggdb -O0`), making it easier to troubleshoot issues. The Release mode applies high-level optimizations (`-O3 -Werror`) for maximum performance while treating warnings as errors. For a balance between debugging and optimization, RelWithDebInfo (`-O2 -g`) retains debug symbols without sacrificing too much speed. MinSizeRel (`-Os`) optimizes for smaller binary size. Additionally, we provide a Headless mode (`-O3 -Werror -DHEADLESS_BUILD`), which removes the GUI components for systems that don't require a graphical interface. By default, if no build type is specified, we set it to Release to ensure the best runtime performance.