CSE 111 Team 9 Report

Contributions and Highlights

Ryan Welter

OS Placeholder Text

Disassembler Placeholder Text

Troubleshooting/Cleanup Placeholder Text

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 Placeholder Text

GUI Color Selector Placeholder Text

Problem With Color Setting Placeholder Text

Michelle Gurovith

Extra Feature: Video Recorder Placeholder Text

GPU Placeholder Text

GUI and Pixel Filters Placeholder Text

Michael Kamensky

GUI (My Proudest Work) Placeholder Text

Memory Placeholder Text

Build Reproducibility Placeholder Text

Extra Features Photos

Graphical User Interface (GUI)

TODO

Pixel Filter

TODO

Game Recorder and Playback

TODO

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.

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
```

Working Slug Files

TODO

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.