NESsys Extensions

By David Pokora (998130883)

Introduction

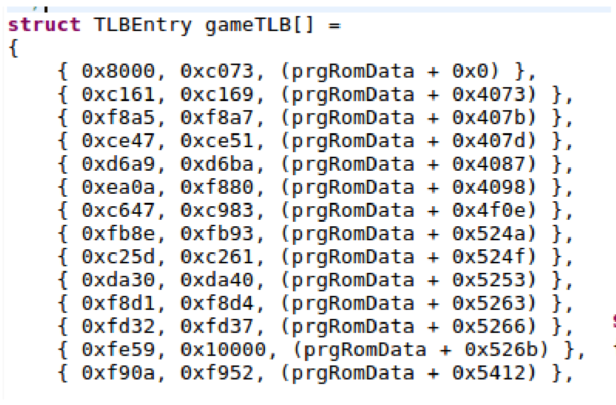
For the previous sprint, I had focused on NESgen (compiler) optimizations and updates. This sprint is more focused with NESsys (enveloping C project) updates, but it contains some NESgen optimizations too. Most of these changes are not something that I can offer in a test suite, as they are either purely functional optimizations, or visual. I offer a version of the previous sprints test suite tools nonetheless, which will simply mass compile ROMs for you to run to analyze these changes. To keep it simple however, I will include some pictures showing the changes made for every update, and briefly mention where the changes can be seen. Changes #0, #3, #6 serve as the larger changes.

#0: Compiler option: Make full PRG-ROM available as data

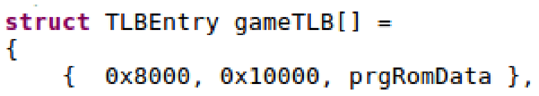
NESgen now has an option "-f" which will make all PRG-ROM data available in the resulting output. What this means is, PRG-ROM (the actual program section of the NES ROM), is split into code and data sections. NESgen currently tries to determine code sections to convert to code, and treats all other data as simply data sections/resources, which it outputs into a data array to be mapped into virtual memory by the CPU. It does this to try and include any actual data the game may use, without including all the machine code (as data) unnecessarily, and move away from holding the PRG-ROM as-is. This sprint allows you to use the specified option to instead dump the entire PRG-ROM as data. This was done for compatibility reasons, for instance, if a game had read its code section as data, and there would be any compatibility issues. This way, all data that should have been available to the ROM, will be guaranteed to be, and it simply remains an issue of code section detection issues when recompiling.

Below are two images which show the game's compiled Translation Lookaside Buffer ("TLB") with the option turned off (left) and turned on (right). As you can see, when the option is turned off, only the data in PRG-

ROM detected to be non-code sections are included, and thus you see memory being mapped to different parts of the buffer. When the option is turned on, all of the PRG-ROM is included, and can simply be mapped for 0x8000-0x10000.



```
struct TLBEntry gameTLB[] =
{
    { 0x8000, 0xc073, (prgRomData + 0x0) },
    { 0xc161, 0xc169, (prgRomData + 0x4073) },
    { 0xf8a5, 0xf8a7, (prgRomData + 0x407b) },
    { 0xce47, 0xce51, (prgRomData + 0x407d) },
    { 0xd6a9, 0xd6ba, (prgRomData + 0x4087) },
    { 0xea0a, 0xf880, (prgRomData + 0x4098) },
    { 0xc647, 0xc983, (prgRomData + 0x4f0e) },
    { 0xfb8e, 0xfb93, (prgRomData + 0x524a) },
    { 0xc25d, 0xc261, (prgRomData + 0x524f) },
    { 0xda30, 0xda40, (prgRomData + 0x5253) },
    { 0xf8d1, 0xf8d4, (prgRomData + 0x5263) },
    { 0xfd32, 0xfd37, (prgRomData + 0x5266) },
    { 0xfe59, 0x10000, (prgRomData + 0x526b) },
    { 0xf90a, 0xf952, (prgRomData + 0x5412) },
```



```
struct TLBEntry gameTLB[] =
{
    {  0x8000, 0x10000, prgRomData },
```

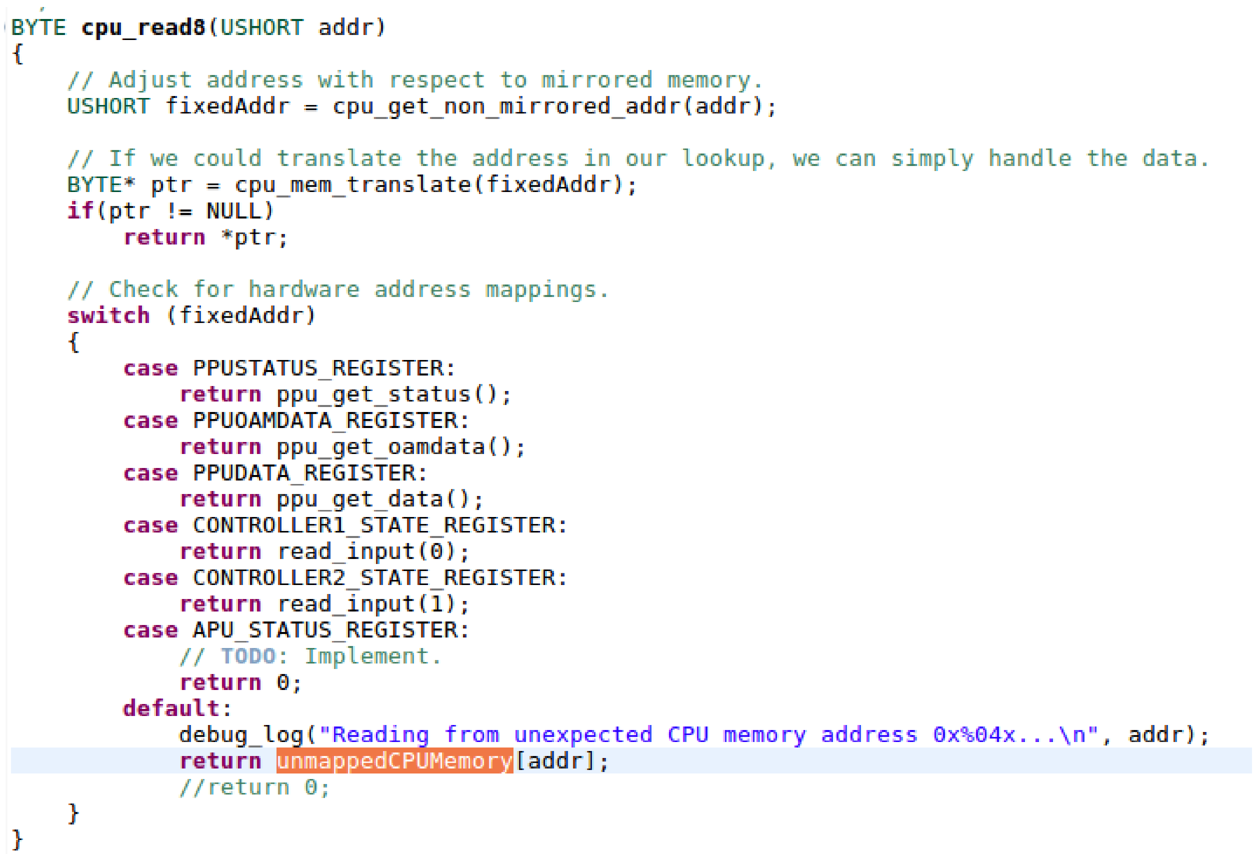
(Turned off) (Turned on)

These changes can be observed in a compiled **game.c** file, after the large **game_execute()** function. After using the test suite tool to mass compile ROMs, you can check the **/src/** folder in the test suite folder and you will see files named "<name>_old.c" and <name>_new.c" where old denotes this option being turned off, and new denotes this option being turned on.

(NOTE: Test Suite will also turn on/off the optimizations from the previous sprint between the "old" and "new" files).

#1: Unmapped CPU Memory fallback

NESsys now has a simple feature which accounts for another compatibility issue. Initially, NESgen/NESsys provided typical memory spaces that it deemed important to the program. This included the zero page, stack, typical RAM space, SRAM (battery backed, savable RAM), and any game data mapped in the gameTLB shown in the previous section. There are some addresses however, which may unexpectedly be accessed. Previous versions of NESsys simple returned 0 on read, and did nothing on writes. This version adds a fallback buffer called **unmappedCPUMemory** to **cpu.c/cpu.h** which serves as memory that can be used in case the recompiled program accessed memory we didn't expect it would, and we still want to provide actual memory to read/write values from/to. An example of how it is accessed for reads is shown below.



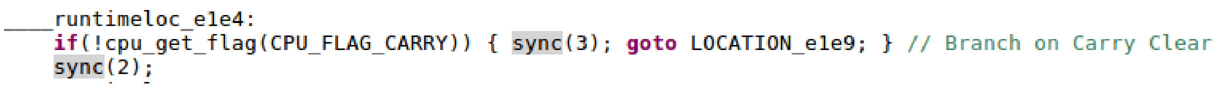
```
BYTE cpu_read8(USHORT addr)
{
    // Adjust address with respect to mirrored memory.
    USHORT fixedAddr = cpu_get_non_mirrored_addr(addr);

    // If we could translate the address in our lookup, we can simply handle the data.
    BYTE* ptr = cpu_mem_translate(fixedAddr);
    if(ptr != NULL)
        return *ptr;

    // Check for hardware address mappings.
    switch (fixedAddr)
    {
        case PPUSTATUS_REGISTER:
            return ppu_get_status();
        case PPUOAMDATA_REGISTER:
            return ppu_get_oamdata();
        case PPUDATA_REGISTER:
            return ppu_get_data();
        case CONTROLLER1_STATE_REGISTER:
            return read_input(0);
        case CONTROLLER2_STATE_REGISTER:
            return read_input(1);
        case APU_STATUS_REGISTER:
            // TODO: Implement.
            return 0;
        default:
            debug_log("Reading from unexpected CPU memory address 0x%04x...\n", addr);
            return unmappedCPUMemory[addr];
            //return 0;
    }
}
```

#2: Improved cycle accuracy

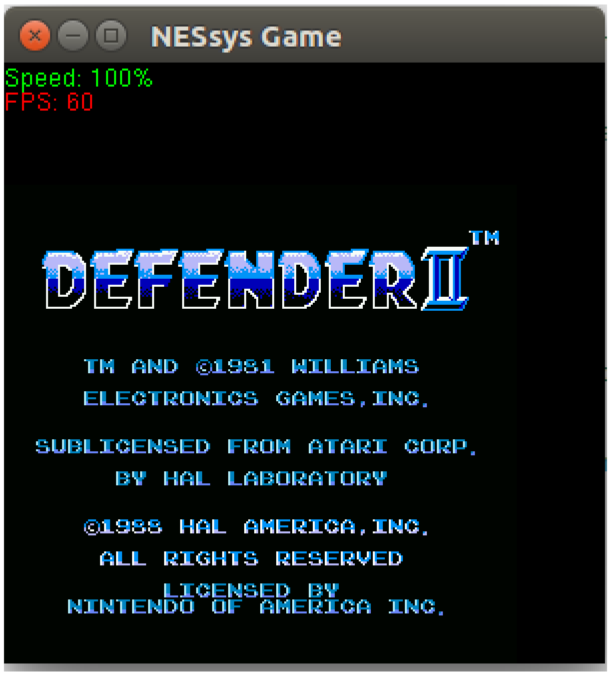
In previous versions of NESgen, the compiler did not account for the additional instruction cycle that is used when a branch is taken. Additionally, there was an error in implementation of branches, which made them cost no cycles if they had not branched at all. This is now fixed, as is shown in the pictures below. This update means that (besides the extra cycle used when crossing memory page boundaries) the instruction set should be cycle accurate. This is not to be confused with the entire system being cycle accurate, as much hardware implemented will not perform tasks on exact clock cycles that the original system would. However, this implementation improves compatibility and gets one step closer to that end.



```
____runtimeloc_e1e4:
    if(!cpu_get_flag(CPU_FLAG_CARRY)) { sync(3); goto LOCATION_e1e9; } // Branch on Carry Clear
    sync(2);
```

#3: Accurate frame-generated based refresh rate

Previous versions of NESgen would re-display (render) the screen at a forced 60 frames per second ("FPS"). This would mean if the system was actually generating frames slower, the program would be choking the system for resources, or if the system was generating faster, no more than 60 frames would be shown. This version instead tracks the frame-generated count per second, and will set the redisplay rate based off of the amount of frames generated last second (setting the minimum to 1FPS). This means if more or less frames are being generated now, redisplaying will occur at a proportionate time. A frame counter for FPS was also added to the application, which can be seen in this picture.



This further verifies the timing of this system. The NES is supposed to generate 60.1~ FPS when running at 100% speed. When NESsys throttles at 100%, the frames generated per second is 60 (with an occasional 61, which is explained by the .1 FPS above 60).

Changes can be seen visually, as shown above, but can also be seen by changing game speed (a feature implemented and explained later in this document) and observing. The code can be understood as follows:

- **ppu.c** implements **frameCount** and **framesPerSecond.** Every frame generated increments **frameCount** in **ppu_update()** in **ppu.c**.
- **cpu.c** has a function called **cpu_sync()** which handles tasks between instructions with a given amount of cycles updated. This is also where speed throttling is done. When a second has passed, **framesPerSecond** will be set to **frameCount**, and **frameCount** will be cleared to 0. This means **frameCount** is the active count for the current second, while **framesPerSecond** was the total frame count from the previous second.
- **NESsys.c** has a function called **update_game_display_asnyc()** (who's argument can be ignored, an argument was required for glut's timer function to callback). This function can be observed to be recursively refreshing with a delay of (1 / **framesPerSecond**) seconds.

Thus, redisplay rate is now actually based on FPS count, and FPS count can also be used to verify timing of the system is generally correct.

#4: Second Controller support (all button mapping changed)

This will be kept short because it is simple. Input for a second controller is now supported, so two individuals could play a two player game (if they could get it compile). Defender is a game that will compile but will most often crash when you die. When it doesn't it can be verified this works. Or it can be verified programmatically.

Controls are now as follows, where left side is the keyboard buttons you should use, and the right side is the corresponding NES control(s):

Player 1 (left side, letters):

WASD = Directional keys

N = B button

M = A button

J = Select

K = Start

Player 2 (right side, directional+numpad):

Directional Keys = Directional Keys

2 = B button

3 = A button

5 = Select

6 = Start

#5: PPU Greyscale Flag Supported

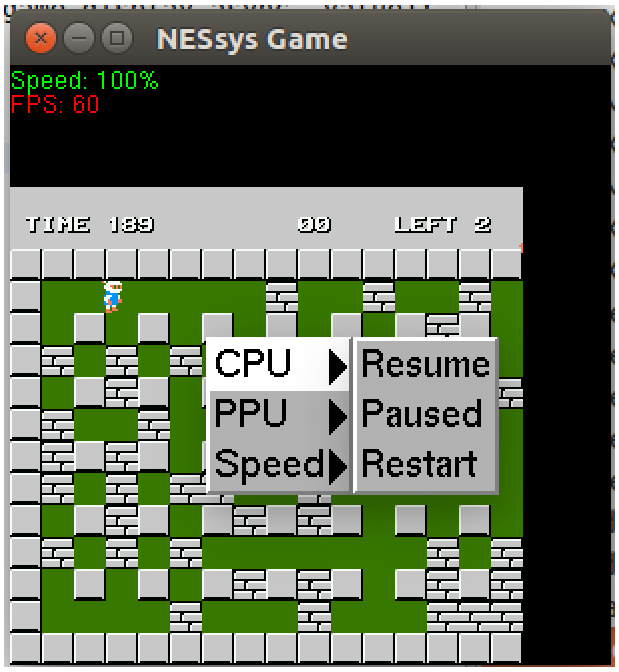
This will also be short, as to not bloat this document. There is a flag in the PPU the game can set to render the whole screen in greyscale. This is now supported. The implementation can be seen in **ppu_get_color()** in **ppu.c**. The results can be visually observed by using the menus described in the next section. A picture of this being supported is shown in the next section as well (when it is forcefully turned on).

#6: Menus added supporting many new options

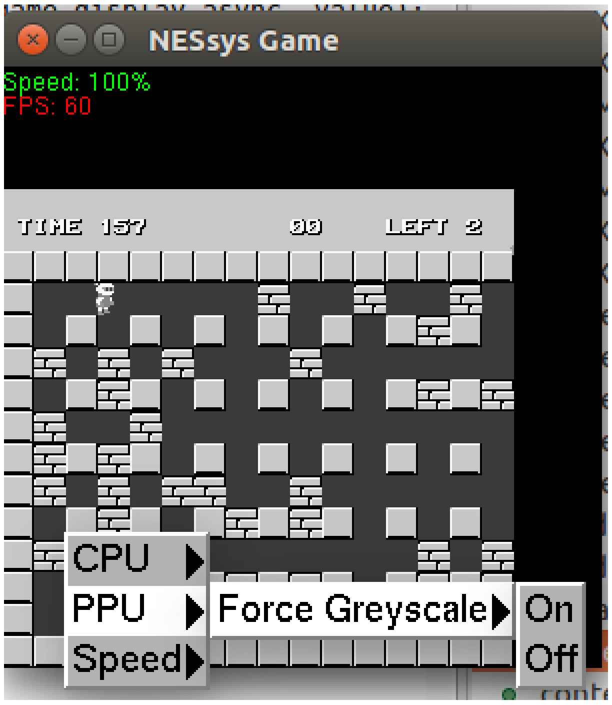
This sprint adds a right-click context menu for the application that offers features implemented in this sprint (besides restart which was implemented last sprint). Features include:

- Resuming/Pausing
  - Implemented in **cpu.c/cpu.h** as **cpuPaused** (BOOL) which simply forces continuation of the speed throttling loop run after every instruction in **cpu_sync()**.
- Restarting (also accessible by hitting F5)
  - Implemented in **cpu.c/cpu.h** as **cpuRestarting** (BOOL) which simply causes **cpu_sync()** to return TRUE, which tells the **sync()** macro from **game.h** that its function **game_execute()** should be exited (which will happen recursively out of interrupts, function calls, etc), all the way back to **game_start()** in **NESsys.c**.
- Customizable Speed Throttling
  - Implemented in **cpu.c/cpu.h** as **cpuSpeedMultiplier** (DOUBLE) which simply multiplies the amount of instructions allowed to be executed given the fractions of the seconds between which it is executed. Implementation can be observed in **cpu_sync**().
- Forcing Greyscale
  - Implemented in **ppu.c/ppu.h** as **forceGreyscale** as seen in **ppu_get_color()**.

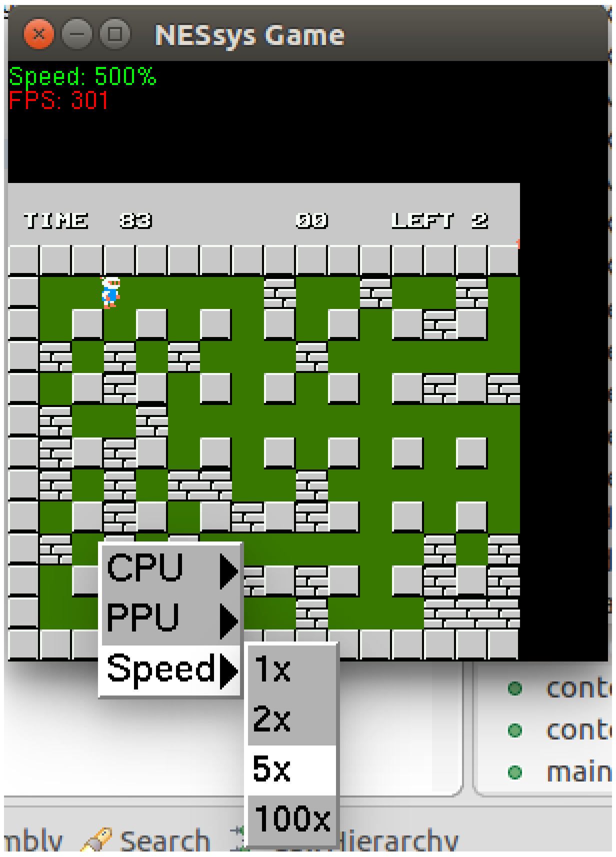
All menu creation can be seen in **NESsys.c** in **main()**, while events handled by these menus can be seen in **context_menu_cpu_handler()**, **context_menu_speed_handler()**, and **context_menu_greyscale_handler()**.



(CPU Resume/Pause/Restart options)



(Greyscale options, with forced greyscale turned on)



(Speed Throttling options, with speed set as 5x, as can be seen by the "Speed: 500%" label, and the corresponding FPS)

NOTE: These menus had to separate On/Off and Resume/Pause as separate

Conclusion

Although there is always still much to do (could've used a better OpenGL framework, implemented resizing of the viewport, fixing a bug where higher to lower speed changes may stall for a second, and many ROMs not being supported), it leaves room for any future updates. It's not a project I will likely update after this point, as the compiler aspect of the project has been explored, and any additional higher level analysis could always be done but it's best to move on at this point, as it was never meant to be a feasible emulator, and the concepts learned here could be applied on more modern systems if one was to really try to tackle some challenges with static recompilation, especially since most executables would follow some compiler's convention. There may be some smaller functionality to fix and implement here, and verify every aspect of every instruction works, but otherwise, as a compiler, all that could be done to support more ROMs would be to find ways to detect code sections (because we handle jumping to any location in a found/recompiled code section), and to recompile any kind of jump tables, or return-oriented programming ("ROP") techniques used or other kinds of untraditional conventions for changing the program counter.

I think the point has been made here, static recompilation is less feasible than dynamic recompilation or interpretation given lack of context, but nonetheless it was fun to undertake a less traditional approach to a fun problem, and get to play a little with a system I knew nothing technically of beforehand.

Test Suite (Mass Compilation of Supported ROMs) (UNIX only)

This sprint offers similar test suite tools as the previous sprint, which will mass compile .NES ROM files to .c/.h files (and save them in the **/testSuite/src/** folder), while also compiling those source files into executables (and save them in the **/testSuite/bin/** folder). Source files can be analyzed to reflect optimization #1, while the compiled binaries offer proof to many others. The rest will have to be seen in code as described throughout this document. This version of testSuite outputs the source files into the **/testSuite/src/** folder with and without optimizations from the last sprint (and optimization #0 from this sprint). The files named "**?_new.c**" (or .h) will contain optimizations while the files named "**?_old.c**" (or .h) will not be compiled with any optimizations. I've included the files output from my run of the script, so you don't need to run it yourself if you don't want to.

Disclaimer from previous sprint: There are a few actual games that now actually recompile and will run but have issues. Most games are obviously not supported, (as mentioned in the previous sprints) or have issues because virtually all games have some runtime values that are improperly predicted. "Bomberman" will load but has no enemies, while "Defender II" will play fine until you die, at which point it will start corrupting memory, and you'll have to hit F5 to restart the game (or use the context menu implemented in this sprint). These may be issues surrounding some runtime calculation, but may also be a bug in current CPU/PPU implementation (especially since we are not 100% cycle accurate, and do not support all hacks and features of the system).

Instructions from previous sprints:

These tests are not compatible with Windows, although NESgen and NESsys can both compile and execute on Windows. They also rely on NESgen to generate C code, and NESsys to compile into executable. As such, you will need requirements for both. This includes python3.5, GLUT/freeglut, and CMake. If you need more information on how to install there on OSX/Linux, please read "**NESgen Documentation.docx**" in **"<root>/NESgen/".** It will describe how to install the requirements on each platform. These tests were tested on “Ubuntu 16.04” and “macOS Sierra 10.12.3” and worked well.

Test Suite included is located in the directory /testSuite/ in this sprint directory. It provides a **testSuite.py** python file, along with multiple fan created NES ROMs. **testSuite.py** batch compiles the .NES→.C→Executable. It does this by using NESgen to generate C files, which it copies into **“NESsys/src/*”**, and calls on **“NESsys/compile.sh**” to compile the NESsys project with the given game C files into an executable, which it then copies from the “**NESsys/bin/***” folder into the “**/testSuite/bin/***” folder in this directory, and then uses “chmod +x” to set the appropriate executable permissions. It also copies the **game.c/game.h** source files for each game it compiles into **/testSuite/src/*** (for analysis). There is also a /testSuite/pics/ folder to show all the roms running.

To use testSuite, **cd** into the testSuite directory, use Python 3.5 (or similar), and simply call the **testSuite.py** class. For example: “*python3.5 testSuite.py*”. It should call the appropriate scripts by calling **“../../NESgen/NESgen/NESgen.py”**. Because it uses relative paths, it is important keep the original folder structure when testing/marking. The **testSuite.py** file calls **python3.5** (**hardcoded**). If you want to use a different version, you will need to change that in the **testSuite.py** script. I included all the binaries I compiled in Ubuntu when I ran it, in case you wish to avoid installing all the prerequisites and compiling yourself, so they should be executable on a similar OS.

Note: **testsuite.py** may require appropriate permissions on some systems if it is not permitted to “chmod” or copy files. **Also note:** there is a bug which occurs on some systems where the file does not flush before it is copied to the appropriate directory. As such, you may have a binary named one thing that is actually another. I added a delay between compilation rounds in case of this. However, in case of any such issue, as mentioned, I’ve included the executables/source files built with my machine which you can analyze on their own.

Using NESgen/NESsys (without Test Suite automating it)

- To keep it short. NESgen is a python program used with Python 3.5. It can compile NES .ROM files into **game.c** and **game.h** files to compile with NESsys (the provided C project) which wraps some NES system functionality
    - Basic usage: **NESgen.py [-n] [-f] -i <input.NES> -c <game.c> -h <game.h>**
    - Here is an example I used to compile a ROM on my machine into NESsys, after **cd**’ing into the NESgen directory:
        - ***“python3.5 NESgen.py -i ~/Desktop/testROMs/Motion.NES -c ../NESsys/src/game.c -h ../NESsys/src/game.h”***
    - This provided example put the files directly into my NESsys folder, although you can place them elsewhere if you just intend to analyze them.
    - The –n option will compile without optimizations from the previous sprint.
    - The –f option will compile using all PRG-ROM data, instead of just predicted data sections, as outlined in optimization #0 in this sprint.
- NESsys is a project which provides an environment to the compiled game C files. It emulates important system tasks that these games depend on to time certain actions and such.
    - Once a **game.c/game.h** is placed in the NESsys/src/ folder, you can compile it using the **compile.sh** (UNIX) or **compile.bat** (Windows) scripts in the NESsys/ folder, which automate the compilation for you. They rely on CMake to find an appropriate compiler on the system. For instance, on a Windows machine, it may use Visual Studio dependencies to compile, whereas on Linux it may use GCC. The compiled executable can be found in the NESsys/bin/ afterward. For windows machines this will have an .exe extension. For UNIX it will have no extension. On

Windows, you will need to copy your installed **freeglut.dll** into the same folder as the executable you compiled for it to run, otherwise it will throw an error.

Source of .NES files: https://www.castledragmire.com/hynes/reference/resource/index.html

- Fan made, open source using legal tools. Not copyright infringing.