NESgen Extensions

By David Pokora (998130883)

(NOTE: This sprint surrounds optimizations, as requested earlier. And as such, this sprint and the next will follow optimization phases for NESgen/NESsys.).

Introduction

For the current sprint, I had opted to focus on optimizations, as I will the next sprint as well (as requested). This is because of a lack of content for any real data type extension. This sprint generally surrounds itself with multiple smaller/medium sized optimizations that typically allow for flexibility, or aim to improve readability. Many of these features do not offer any operational difference and can only be verified by you visually, and as such, I've included some useful images to describe what I have done in this document. Typically, compiler-based optimizations were done within **iNESROMDisassembler.py** and the results can be analyzed in the output **game.c** and **game.h** files it compiles. All backend optimizations' residing locations will be described accordingly as well.

#0: Improved readability using macros for instruction functions

The day I had submitted my last sprint, I added a quick feature worth mentioning here (as other features I've added this sprint follow suit). I allow for macros to be defined for each instruction function, so that every instruction no longer has the name "MOSInstr_<3LetterInstructionAbbreviation>" but instead is given a name that is more intuitive for readability such as "StoreA" or "SetA". The previous style of output appeared as it does in the picture below, whereas the output at this stage will appear as you will see in later pictures in this document. These are compiler defined macros (and output in **game.h**) as opposed to simply renaming the functions so that even if the labels are changed across builds or customized by users, it will still work. Additionally, this allows the developer to still find an instruction implementation easily. These function name overrides can be found in **MOS6502Instructions.py** defined on a per-instruction-definition basis. The feature can be enabled or disabled by

editing the global boolean “ALLOW_FUNCTION_NAME_OVERRIDES” in **iNESROMDisassembler.py** (which is where the actual compilation occurs)**.** It is enabled by default.

```
LOCATION_80d7:
        MOSInstr_STA(0x4010); // Store Accumulator in Memor
        cpu_sync(4);
        MOSInstr_INY(); // Increment Index Y by One
        cpu_sync(2);
        MOSInstr_LDA(cpu_read8(registers.Y + 0x8951)); // L
        cpu_sync(4);
```

(old style output)

#1: Jumping to any location (foundation for runtime calculated jumps)

Although there is still the problem of implementing any kind of analysis/prediction for runtime calculated values/unconditional control flow (as mentioned in previous sprints), this sprint brings the ability to jump to any location in the converted game code by providing a memory location to jump to. Unfortunately, there was no real “clean” way to do this (as attempting to control program flow on an assembly/low level basis would differ between platforms), so this amounts to adding goto labels for every instruction that was disassembled, and adding each location to a jump table which indexes locations by their memory address. For reasons of readability, locations which we were able to map jumps to still have a name that stands out (“FUNCTION_???” or “LOCATION_???”), while locations which are not known to have a jump to – but are supported for reasons of permitting direct jumping access to in case of runtime calculated jumps – have a label name such as “___runtimeloc_???”, denoting it is a possible runtime jump location.

```
LOCATION_c53b:
    SetA(cpu_read8(0xa)); // Load Accumulator with Memory
    sync(3);
____runtimeloc_c53d:
    if(!cpu_get_flag(CPU_FLAG_SIGN)) { sync(2); goto LOCATION_c555; } // Branch on Result Plus
____runtimeloc_c53f:
    MoveYToA(); // Transfer Index Y to Accumulator
    sync(2);
LOCATION_c540:
    StoreA(cpu_read16((registers.X + 0x2) & 0xFF)); // Store Accumulator in Memory
    sync(6);
____runtimeloc_c542:
    DecrementY(); // Decrement Index Y by One
    sync(2);
```

As you can see above, every instruction has a label prefixing it, with known jumped-to locations standing out more by its lack of indentation and differing name. A runtime calculated jump (if we can assume it to work), will be converted in such a manner:

```
FUNCTION_NMI:
    sync(4);
    jumpAddress = cpu_read16(0xdb); goto Jump; // Jump to New Location
FUNCTION_IRQ:
    sync(6); return TRUE; // Return from Interrupt
LOCATION_846f:
    SetA(cpu_read8(0x4d9)); // Load Accumulator with Memory
```

Where the address to jump-to is being set as the 16-bit integer at address 0xdb, then we jump to the jump table, which resolves this jump address to a location to go to, and performs our jump for us. Addresses in this jump table appear as shown below, where known jump location labels are given a macro to label their addresses (for readability reasons), and unknown runtime locations are kept bare (to avoid bloating the code with a macro for every location).

```
    case ID_LOCATION_fbfd:
        goto LOCATION_fbfd;
    case ID_LOCATION_e3ff:
        goto LOCATION_e3ff;
    case 0xd403:
        goto ___runtimeloc_d403;
    case 0xbf56:
        goto ___runtimeloc_bf56;
```

We know this works functionally because this is the jump system used to begin execution (we call reset interrupt handler by its address, defined as macro “ID_FUNCTION_RESET” in **game.h**), or jump to handle our interrupt for vertical blanking periods (defined as macro “ID_FUNCTION_NMI”). In ROMs which are one memory bank instead of two, the ROM is mirrored in memory. This is also detected by NESgen and is properly handled in the jump table for both addresses.

Any ROM recompiled using NESgen which uses runtime calculated jumps will give a warning of possible mis-analysis upon recompilation. Any runtime calculated jump which jumps to a location that was not anticipated/recompiled as code will throw an error in NESsys.

#2: Labelling known memory access locations

In previous versions, any loading and storing of values was solely done using numerical memory addresses and not interpreted/analyzed any further. This sprint brings an optimization which analyzes any read/write locations to determine if they correspond to known locations (such as memory mapped PPU registers, input registers, etc). When NESgen detects a location it's familiar with, it will now output the load/store using the appropriate macro/label making it more readable. As such, if we wanted to find where input is read, we can now search for a given macro. You can see a few of the supported locations below, which prove easy to read.

```
LOCATION_8804:
    SetA(cpu_read8(CONTROLLER1_STATE_REGISTER)); // Load Accumulator with Memory
    sync(4);


____runtimeloc_81c0:
    StoreA(PPUSCROLL_REGISTER); // Store Accumulator in Memory
    sync(4);


LOCATION_8181:
    SetA(cpu_read8(PPUSTATUS_REGISTER)); // Load Accumulator with Memory
    sync(4);
```

#3: Consolidating PRG-ROM data sections into one location, and adding support to backup/restore it → game restart.

Previous iterations of this application output all data sections detected in PRG-ROM into separate byte arrays, and mapped them in the Translation Lookaside Buffer ("TLB") accordingly. These data sections were stored as raw byte buffers since there is no way to guarantee analysis of all data. This sprint instead consolidates all data sections' data into one array, and maps to the appropriate location inside the array accordingly. The primary motivation behind this is cleanliness, since multiple separate arrays were unnecessarily bloating, but it also allowed for us to easily backup and restore all program data sections' data. This is useful because data in these data sections could change at runtime, and backing up from one location is much more efficient and doesn't require lookups in the TLB. At this point, game restart functionality was also implemented, which can be triggered by clicking F5 when running a converted game. It restores all PRG-ROM data, and reloads the pattern tables with the CHR-ROM data, as well as resets all CPU/PPU values used at runtime. The picture below on the left shows *just the beginning* of the declarations for PRG-ROM data from a previous build. The picture below on the right is the current build which consolidates the data. (Both are pictures of an output **game.h**).

```
// --------------------------------
// Data
// --------------------------------
extern BYTE DATA_8000[];
extern BYTE DATA_95c1[];
extern BYTE DATA_fe68[];
extern BYTE DATA_9c44[];
extern BYTE DATA_ab05[];
extern BYTE DATA_8486[];
extern BYTE DATA_d441[];
extern BYTE DATA_af88[];
extern BYTE DATA_82ca[];
extern BYTE DATA_984b[];
extern BYTE DATA_bc8c[];
extern BYTE DATA_c7cd[];
extern BYTE DATA_87ce[];
extern BYTE DATA_d90d[];
extern BYTE DATA_9dd1[];
```

```
// --------------------------------
// Data
// --------------------------------
extern BYTE prgRomData[];
UINT prgRomDataSize;
extern BYTE chrRom[];
UINT chrRomSize;
```

#4: Optimized PPU nametable mirroring

On the NES, there is memory to render 2 NES screens at a time, but PPU address space allows for 4. As such, games will map these 2 screens into 4 locations, so each screen will appear twice in memory. These structures that constitute the screen are called Nametables. Games can have either Horizontal or Vertical mirroring (and will support Vertical or Horizontal scrolling respectively). Games can use memory mappers to provide more than 2 physical nametables (but recall: we do not support memory mappers in this project). In previous builds, we allocated 4 nametables and redirected any read/write operations to a different table's memory address before performing the lookup for the redirected address in the TLB. This essentially is performing two redirect operations. This sprint's build instead allocates 2 physical nametables, and also allocates 4 pointers to nametables (which it sets to the 2 physical tables. This saves memory space, and allows for mirroring/mapping without any prior redirect. Instead, we map to the data at these pointers which we adjust to the same memory location for two of the tables.

Diagrams and explanation of nametable mirroring can be found here:
https://wiki.nesdev.com/w/index.php/Mirroring#Nametable_Mirroring

While an example of scrolling through two nametables can be found here:
https://wiki.nesdev.com/w/index.php/PPU_scrolling#The_common_case

Sidenote: A few bugs fixed

It's also worth noting this sprint fixes many bugs. One or two instructions would did not set a flag correct. PPU background rendering was not rendering the final nametable tile column (8px) properly. PPU mirroring bugs were fixed in this sprint when mirroring was optimized/redone. An issue with zero page addressing in some instructions that were recompiled were not correct. Tests were not updated and would've failed.

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). Although this sprint offers no real executable functional difference (as most are visual), source files can be analyzed to have the described changes implemented, while the compiled binaries can prove that execution has maintained integrity and all ROMs provided prior still work. This version of testSuite outputs the source files into the **/testSuite/src/** folder with and without optimizations. 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. Since some optimizations are backend implementations, I'll be specific: the optimizations that are enabled/disabled are #0, #1, and #2. 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.

NOTE: 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. 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). This will be looked at further depending on time permitted. The next sprint is likely to target audio support.

Instructions from previous sprint:

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] -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 #0, #1, #2.
- 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, where as 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.