NESgen Backend

By David Pokora (998130883)

(Similar to last sprint: I apologize for the length of this document, it is a bit longer in order to describe all aspects of the project since it is untraditional and I want to make sure everything is understood correctly. If you are uninterested in the details behind some of the aspects of it, I would suggest skimming through it or seeing what headers/sections I provide that may give you more direct answers you seek).

Introduction

In the previous sprint I had elaborated about NESgen (the NES ROM to .C/.H compiler)'s intermediate representation ("IR") which it had parsed a given ROM to. At that stage the parsed output was presented in an assembly like form. This stage however, focuses on how the intermediate representation which was parsed is converted into the .c/.h code to be used and compiled within NESsys (the NES system wrapper). I will first explain the decisions made for this compiled output, and then explain how the compiled will look as a result of those decisions based off the data in the ROM.

System Implementation

The goal of NESgen was not to simply take a NES ROM and store it as a hardcoded array to be run as a static interpreter, but to eliminate any instruction interpretation such that we would not need to look up an opcode, the instruction length, and rely on the machine transferring control flow as an NES would (changing program counter register). Instead, NESgen parses the code/data sections in such a way that it interprets control flow such that we can reconstruct the ROM with C-like control flow principles (goto, function calls) (virtualized/emulated program counter is nonexistent). Because ROMs are so low level and do not follow many conventions a compiler would (since they are almost exclusively written by hand with an assembler), there are many cases to attempt to support that will be hard not to overlook, requiring the most flexible output to remain in a similar layout (unless higher level analysis is done). What this means is one cannot simply replace all jumps with function calls, as some jumps do not use the stack, but function calls require the stack to store the return address on most machines. The control flow created with these jumps could be irregular such that they cannot be implemented cleanly/efficiently in any other such way than goto statements (with static recompilation). For this reason jump/branch instructions which do not use the stack (JMP) and do not return to caller are replaced with goto statements, while any jumps that do use the stack (JSR) are replaced with a function call and return to the caller. Because all game code is stored in one function, this function takes an input value, telling what goto label to jump to, and uses an appropriate jump table (switch statement) to reach the desired location in the code. (This is all visible in **game.c**'s game_execute() once compiled). Similarly, because it is extremely difficult, or at times impossible to have a perfectly accurate higher level analysis of all variables used/stored in memory without runtime context, NESsys still uses the same finite amount of registers as an NES would to compute and its original intended memory model (virtualized). Most instructions are mapped to functions which perform the actions that the original instruction would have done with respect to CPU flags. The compiler by default (but optionally) outputs macros for the original function names that simulate an instruction to give more readable names, and be customizable as the compiler changes. Jumps and branches do not call a function but use direct "*if(condition) goto label;*" statements. In between all the game's lines of code will be a function call to sync() with an integer argument. This is a function which tells the CPU that a given amount of cycles has occurred, so it should call on the PPU to process (as it is meant to in parallel, 3x faster than the CPU), and handle any interrupt requests, as well as throttle the speed of the machine. It is not cycle accurate (does not account for an extra cycle for cross boundary memory page accesses), although it does handle the appropriate amount of base-cycles per instruction. Because it is not 100% cycle accurate and certain states are not updated in between specific cycles for an instruction, some games obviously will not render 100% as the NES is. This is hard, and also not the purpose of this compiler. The framebuffer drawn in PPU code is double buffered to avoid showing frames that are in progress/torn, since the OpenGL framework used calls on drawing asynchronously. (Debating changing OpenGL frameworks for unrelated reasons later, but will see).

NESsys provides a virtual Central Processing Unit ("CPU"), Picture Processing Unit ("PPU"), and virtual memory. The memory either maps to data (much of the data is shared by the PPU rendering code (direct), while game's can also map to this data through its virtual memory model (see: **memory.c**). This way, NESsys has higher level structures (struct) which can be virtually mapped for the game code to accessed on a low level basis (byte*) by its original intended memory address). This way, the CPU provides its virtual memory space, and the PPU can provide its own virtual memory space. This is how data sections in the ROM are implemented. The PRG-ROM (NES program)'s data sections are stored as arrays in the compiled **game.c** along with their respective addresses in a Translation Lookaside Buffer ("TLB") (which **memory.c** accesses to map game data into CPU address space). Unlike PRG-ROM whose data sections can be located in various places and require a TLB, the CHR-ROM (character ROM/video resources) is all one large data block loaded into the PPU's address space at a static location, and as such, it is stored as a single byte array called "chrRom" in **game.c**. Additionally, the CPU and PPU can communicate through various special registers (see: **ppu.c**).

Compiled Output (Implementation)

The compiled **game.c** will have (in order):

- A macro which defines whether we should use **game.c** (the compiled game), or **game_base.c** (an interface for it, which was used to design **game.c**'s layout (used for development reasons).
- The game_execute() function, which returns a BOOL which is TRUE when the function has exited a function or interrupt.
    - Function calls (which call game_execute at a specific location) ignore the return status and simply return back one step. (returns to caller).
    - Returning from an interrupt will check return status and if true, return true recursively via the sync() macro. (returns to interrupt).
- game_execute() takes an argument "functionID" for a goto label to jump to, which it uses a jump table for, to reach the correct code location for function calls/interrupts. The jump table (switch statement) is visible at the top of the function. It is then followed by the earlier described game code.
- At the end of the function is the game's TLB which maps all game data to specific addresses, followed by the actual game data in arrays, including the CHR-ROM data which is the final byte array in the source file.

game.h is simply a header file for **game.c** and is trivial in design, and as such, I won't take up more space to explain it.

Description of Scripts

The scripts contained in NESgen are described:

- **NESgen.py**: Main entry point
- **iNESROM.py**: Provides information about the ROM, and location/size of PRG-ROM/CHR-ROM.
- **NESMemory.py**: Memory/file address translation.

- **MOS6502Instructions.py**: MOS6502 Instruction Set
- **PRGROM.py:** Parses the NES ROM's underlying PRG-ROM into code/data sections, storing code sections (a list of instructions) separately based off of change in control flow (jumps into a section will split it, instructions which statically jump or return mark end of code section area).
- **iNESROMDisassembler.py:** Uses **PRGROM.py** to generate ASM/C code.

The source files contained in NESsys are described:

- **cpu.c:** Implements CPU registers, stack (return address no longer stored on it since we don't handle program counter), and a function to simulate x CPU cycles, which causes PPU to cycle 3*x times, then checks for interrupt requests to handle, and finally, throttle's execution speed.
- **game.c:** The NES ROM compiled into C source to be executed within NESsys.
- **input.c:** Handles keyboard input. Keys are Z,X, ENTER, BACKSPACE, UP, DOWN, LEFT, RIGHT. Both controllers implemented, only one mapped for proof of concept.
- **instructions.c:** Functions which modified the CPU registers accordingly as the MOS6502 instruction normally would.
- **memory.c:** Memory mapping, reading, writing implementation for both CPU/PPU (and map's game data sections into CPU address space as well, and game's CHR-ROM into PPU address space).
- **NESsys.c:** Main entry point, handles actual OpenGL rendering, window, callbacks, etc.
- **platform.c:** Used to implement universal features/includes used throughout the program, including cross-platform wrappers.
- **ppu.c:** Picture Processing Unit, handles PPU's special register operations, and rendering.
- **tests.c**: A few random tests used when implementing certain features. Not heavily maintained. Was simply to test before ROMs were working.

It is worth noting that **NESsys.h** contains many configuration variables, such as debug logging, whether to use **game.c** or the template **game_base.c**, and testing mode which runs the tests in **tests.c** instead of the game. Also contains a multiplier for game speed.

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

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.

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
    - 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 ~/Desktop/testROMs/Motion.NES ../NESsys/src/game.c ../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.
- 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.