NESgen Parser

By David Pokora (998130883)

(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 my changes/structures and simply wish to test output, you can skip to the “Verifying Parsed Structures” section).

Description of Parser

In the previous sprint, I had generally underlined the criteria under which this project will be expected to perform. I had provided a very simple set of scripts for python to parse a given NES ROM, and its underlying PRG-ROM (Program ROM)’s code/data. The submission for this sprint however, is an update to those scripts which parses to a more comprehensive intermediate representation (“IR”), which will be used directly for code generation. In the previous sprint, I had simply followed jump/branch instructions and marked the start of code sections, only to re-run from the beginning-to-end of the ROM, outputting ASM-like text of instruction/data without actually storing it in any IR. All of the data of the PRG-ROM was kept as an array of bytes and data was simply output.

The submission for this sprint offers much more comprehensive parsing, as mentioned. The PRG-ROM is now parsed into structures which describe the contents well enough that the original byte buffer is no longer needed, and all parsed ASM output you will see in this sprint is purely generated from the IR. The code section structures in this sprint maintain the original offset/address information, along with a list of instructions, and information regarding any references made to it (which is not needed at the moment but is useful for additional features I may add in the future). These code sections are split in two if there is a jump that occurs into the body of them. This allows us to assert that the start of any code section will be the only places where a jump occurs to. Using this logic, we can precede any sections of code with “goto” labels, which will allow us to recreate the original control flow of the ROM in our converted or parsed output. This means you will now see labels used for jumps in the ASM output instead of the integer literals in the previous spring which represented relative/absolute addresses. The original control flow will also be maintained in the C output by replacing jumps with goto statements, and comprehending where we should place the labels which we will go to. The data section structures created simply record the original offset/address/bytes of data sections, which will allow us to output them as arrays into the C output, and add the address to a Translation Lookaside Buffer ("TLB") to provide the code with proper access to the data as originally intended.

For purposes of explaining every piece of the parser, I'll quickly reiterate from last sprint how each structure is discovered/parsed: Each instruction is of variable length, where the first byte is a unique identifier that denotes what the instruction is (and indicates how its operand will be used). By parsing this one byte, you will know how long the operand is, and how it should be used in accordance with the instruction. Some instructions do the same things, but simply address their values differently (using operand as immediate value, or as a pointer). A PRG-ROM is a collection of code/data sections, in no particular form, and most often intermittently. To detect code sections, you simply follow the 3 interrupt entry points which you obtain from three 16-bit pointers (MOS6502 is a little endian architecture) at the end of PRG-ROM. Following code jumps and branches will lead you to other sections, and certain instructions will denote the end of an interrupt or section, or a jump to another code location which does not return to the caller, which signifies there is no guarantee code will follow so you stop parsing sequentially at that point and keep following branches you've yet to traverse. Anything which is not a code section is simply a data section. Because it is assembly, it is all low level and data sections can constitute any kind of data (high level interpretation of data is unnecessary). And finally, the PRG-ROM is parsed from the generic NES .ROM format (which is formally called iNES). iNES has a 0x10 size header which contains a signature, and flags to denote rendering modes, and the size of PRG-ROM/CHR-ROM and whether an optional code stub to help start the game in a specific context (called a trainer) is used. Trainers are 512 bytes immediately after the header. PRG-ROM follows sequentially after the trainer data, and CHR-ROM after that. This explains the basic parsing process.

It is worth noting that the only other part of the NES ROM that will be output into the final C files which is not shown/handled in these scripts is the CHR-ROM (video/PPU resources) mentioned in the previous sprint. This is simply because there is no real conversion/processing/parsing to do in that regard. It will be stored in a byte array as-is in C (or as a separate binary file), and simply loaded into GPU memory as it was originally intended.

Description of Scripts

The following scripts have not been significantly changed (with respect to parsing):

- iNESROM.py (Generic NES .ROM file parsing)
- NESMemory.py (memory/file address translation features)
- MOS6502Instructions.py (MOS6502 instruction set)
- NESgen.py (main entry point)

The following are a high level description of the list of changes and general purposes of underlying objects so you can understand what is going on:

- **PRGROM.py**
    - This is a new class, it parses the NES ROM's underlying PRG-ROM into code/data sections.
    - **Components:**
        - **PRGROMCodeSectionType:** An enum which describes the type of code section, used to differentiate "goto" labels for stylistic reasons.
        - **PRGROMCodeSection:** Describes a block of code which is jumped to, and maintains all instructions that follow until the next code or data section. Any place code can jump to will have a respective code section.
        - **PRGROMDataSection:** Describes/stores a block of data/bytes in a non-code section of the ROM. Could be unused, but is included nonetheless for compatibility reasons.
        - **PRGROM:** This is where PRG-ROM parsing is done into the two types of structures listed above. A dictionary of {MemoryAddress : SectionClass} is maintained here for quick lookup.
- **iNESROMDisassembler.py**
    - This class was entirely rewritten. It uses a PRGROM() from PRGROM.py to parse a supplied ROM's PRG-ROM section.
    - It determines the name of the labels based off of whether we detected the jump to each code section as a function (returns to caller) or simple code location (doesn't).
        - Interrupts are required to have their labels so we can handle them. Because their labels may reside at the same address, special care is given to them.
    - Jump instructions now use these labels instead of integer literals signifying relative/absolute addresses.

Verifying Parsed Structures

Verifying parsed structures in this case can be done as it was in the previous sprint. I recommend turning off the "wrap text" feature on your text editor, since all the data sections which bloat the parsed output are actually on one line, and will then be too intrusive to the comparison.

I have provided a *testSuite.py* script in the */testSuite/* folder in this sprint folder which will take all of the .NES files in the folder, parse them and output the parsed data as "*Filename_Parsed.ASM*" which you can compare to the original source I included for each .NES, aka "*Filename.ASM*". There are some ROMs I included which call an "include FILE.NAM" or something of the sort to include a data section. I did not include these since they mean extra analysis for you, but you can obtain them from their source which I list at the end of the paper if you are really interested in making sure every single data section is correct (although it should be apparent from the data that isn't included from another file, Zelda is a good example for this since it doesn't include any external data files). To verify the output is correct, I would mainly focus on the code sections and the instructions themselves. The instructions should match up. The original source code may use macros for certain operands so you may not be able to compare operands directly unless you go find the macro assignment, but you will know the parsed output is correct simply based off the fact that the instructions are the correct type, and any instructions that follow have the correct operands. If there was any size mismatch, all the following instructions would be incorrect. If there was a value mismatch, it would be apparent in areas where macros/labels are not used. This version of the parser uses labels for jumps/branches, so it is more like the original source in this sprint.

To execute the testSuite, use Python 3.5 (or similar) and simply call the *testSuite.py* class. 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 directly. If you want to use a different version, I would recommend changing that in the script. This was tested on Ubuntu 16.04 and worked. In case of any compatibility issue (since calling OS/directory functions in python is not the most flexible for all environments), I included all the files it generated when I ran it.

Note: PRG-ROMs are loaded into 0x8000-0xFFFF. PRG-ROMs can be 0x4000 or 0x8000 bytes in size. If they are 0x4000, they are mirrored twice in that address space to fill it up (they are at 0x8000 and 0xC000), so if you see a memory address reference at 0xC030, it is the same as referencing 0x8030. (In case you are curious why there is some sort of an address inconsistency in any output or tests you may run). I output PRG-ROM size in the header of my parsed output if you're curious if this is the case.

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

- Fan made, open source using legal tools. Not copyright infringing.
- (Included “Zelda Title Screen Simulator” and “Motion”)