NESgen Language Specification

The aim of this project is to create a compiler which, when given an NES ROM file, will disassemble it and parse the disassembly into C code. In order to provide a framework within which this converted code can execute, the project is broken into two main pieces. **NESsys** is a cross-platform C project which provides many functions that help simulate an NES system's environment, which is critical to NES ROM code. **NESgen** is a Python3.x project which takes a NES ROM file as input, and processes it into relevant C/Header files to be bundled within NESsys. The aim of such a system is to allow the user to execute converted ROMs on their machine, as any system functionality that the ROM would depend on, would be provided by the NESsys framework. Put simply, a user could convert a NES ROM into game.c/game.h files, copy them into the NESsys project (overwriting the existing templates), and compile for the target platform. The requirements are simply Python 3.5 (or similar) for the NES→C conversion, and a C compiler with GLUT/freeglut libraries to be able to execute the converted game using OpenGL on the target system. Further documentation regarding requirements and steps to build across multiple platforms have been included in the source tree as "*NESgen Documentation.docx*".

NES ROM files have a simple structure to understand, but rather difficult to parse. To begin, the header contains flags which denote much about the ROM such as the size of the program section (code/data), and the character section (a section that is loaded by the Picture Processing Unit ("PPU") directly into VRAM). Throughout its life cycle, the NES saw many hacks used to further extend the otherwise limited hardware. However, use of memory mappers which swap memory from the extended RAM/ROM/VRAM provided in the cartridge to the system are NOT supported in this project. This is the case for the same reason that code-jumps to calculated addresses at runtime likely won't be supported (since the compiler is not emulating the system, it has no runtime context, and cannot determine where these calculated jumps will occur). Memory mappers would reuse the same memory addresses for different code sections, and only further complicate this process. Memory mappers can be detected via flags in the ROM header, so as to avoid issues. Runtime calculated jumps can be detected at recompilation time. ROM files have a simple format of a header of size 0x10, an optional 0x200 byte trainer (a code section), a PRG-ROM (program code/data section) of variable length defined in the header, and a CHR-ROM (character/picture data) of variable length defined in the header. The CHR-ROM is simply data and can be broken into its respective sub-structures for NESsys. The PRG-ROM is where the bulk of recompilation will need to be done. This section can be a combination of data or code, intermittently without warning. The only way to understand what is a code section is to parse the three possible entry points into the application, and detect if any of the sequential code jumps to another location in memory (which we can calculate the file offset for by subtracting its base (0x8000)). Certain instructions can be used to detect the end of a function. Once code sections are detected, the remaining sections are simply marked as data sections, to be thrown into byte arrays in the output C files to ensure the data is always accessible to the converted code. The code segment is made up of low level processor instructions based on the MOS6502 architecture. Each instruction is made up of an 8-bit integer

that identifies what kind of operation the instruction performs (opcode), and *can* have an 8-bit/16-bit operand (argument) which is used in different ways depending on the opcode. Many of the same instructions are implemented more than once, to provide different uses of the operand (relative vs. absolute addressing for instance). Implementation to parse instructions uses an opcode lookup in MOS6502Instructions.py in the NESgen project to determine the instruction, and size/use of operand. The input language in this case meets the requirements of having distinct data types (8-bit/16-bit integers as operands, byte arrays/pointers in PRG-ROM, structures in CHR-ROM), conditional branches implemented through certain instructions, loops implemented the same way, and procedures implemented by entering a function and storing the return address on the stack, to know where to return to.

The target for recompilation as mentioned before, are .c/.h files for use in the NESsys project. The manner in which the input data will be handled is as follows: All data sections in CHR-ROM will be split into their separate substructures for use with the virtual PPU. All data sections described in PRG-ROM will be put into arrays, and their original memory addresses put into a Translation Lookaside Buffer ("TLB") to provide compatibility for converted code that wishes to access original NES memory addresses. In terms of code sections, instructions parsed will be output as NESsys function calls which simulate an instruction's behavior. Instructions which handle control flow (jumping, branching, and returning) would ideally mark sections to be split into separate C functions, although if control flow proves to be irregular in many cases, use of "goto" statements may need to replace this.

Finally, the desired usage of the NESgen tool will appear as such:

"python3.5 NESgen.py –o OUTPUTDIRECTORY\ INPUTROMFILE.NES"

Where –o denotes the output folder to output the .c/.h files into, and INPUTROMFILE.NES is the NES ROM to convert. Currently, NESgen.py can be used without the –o argument and output folder to show that it can successfully parse a provided ROM. It will output .ASM-like text that can verify its interpretation is correct. See the included "*ZeldaOriginal.asm*" (original source) and "*ZeldaParsed.ASM*" (parsed disassembly output).

NOTE: The provided Zelda ASM files are not the actual Zelda game, and thus, do not infringe on copyright. They are fan-made ROMs that look similar to the actual Zelda title screen (source: https://www.castledragmire.com/hynes/reference/resource/index.html).

You can also try using the Polar Demos ROMs or a few others on that page, as many of them do not use runtime calculated jumps which NESgen will likely not support.