SNO

SUPER NINTENDO ONLINE

DESIGN DOCUMENT

Michael Kelly
mkelly01@my.fit.edu

Keith Johnson
kjohns07@my.fit.edu

Eric Wells
wellse@my.fit.edu

Faculty Sponsor
Dr. William H. Allen
wallen@cs.fit.edu

What is SNO?

SNO is a Super Nintendo emulator written in Java. It is intended to be embedded into websites to allow users to play games developed for the Super Nintendo within their browser.

There are two primary user classes for SNO:

- **Host:** A host is someone who embeds SNO onto their website. They will be required to have access to a web server so that they may host the executable and any ROMs or save state files that they wish to share.
- **Player:** A player is a user visiting the host's website. They interact with SNO and actually play a game on the emulator.

Core Design Principles

- The software will emulate hardware functionality of the SNES directly, as opposed to recompiling ROM code or other techniques.
- Although the software shall be coded in Java, we will favor efficient, to-the-point code over over-designed, "Java-esque" code.

System Overview

SNO is comprised of two main subsystems: the Core SNES system and the Frontend Java system. The Core SNES system is a software emulation of the SNES hardware. It concerns itself only with emulating the functionality of the SNES. The Frontend Java system comprises the second half of the program, and translates the output of the SNES system to the frontend displayed to the user.

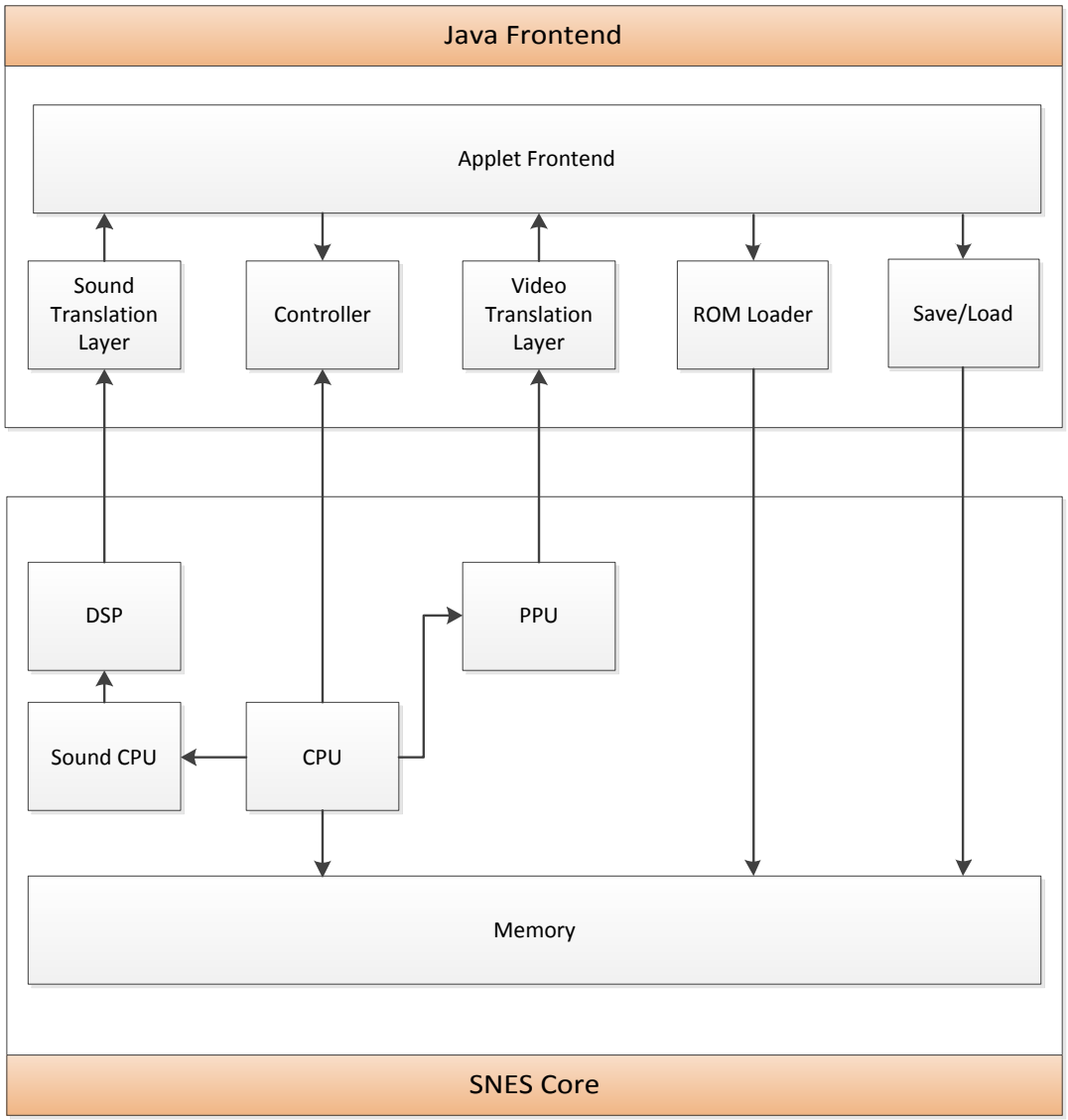


Figure 1: System Block Diagram

Block Diagram Components

- **Java Frontend**
  - **Applet Frontend:** Java frontend that the user interacts with. Handles video and audio output, user input, and other user-facing functions.
  - **Controller:** Stores the current state of the SNES controller. The frontend changes this state depending on user input, and the CPU polls this state to set the correct values in memory where the controller state is stored in the SNES.
  - **ROM Loader:** Retrieves a ROM file from either a specified URL or the user's local computer and loads it into the SNES memory.

  - **Save/Load:** Handles serialization of both the SRAM (for normal, game-managed saving) and of system state (for save states).
  - **Sound Translation Layer:** Hooks into the sound system within the SNES Core and translates the output sound into a format that can be played by the applet frontend.
  - **Video Translation Layer:** Hooks into the video system within the SNES Core and translates the video output into a format that can be displayed by the applet frontend.
- **SNES Core**
  - **CPU:** Represents the 65816 CPU within the SNES. Processes instructions in the ROM code, as well as managing timing and triggering other systems within the SNES Core when needed.
  - **DSP:** Digital Signal Processor. Generates the 16-bit sound waveform that the SNES outputs for sound. The DSP is controlled by the Sound CPU;
  - **Memory:** Stores data in the various parts of RAM (detailed later in this document).
  - **PPU:** Picture Processing Unit. Manages video output for the SNES including sprites and backgrounds currently being displayed.
  - **Sound CPU:** Represents the SPC700 CPU that runs programs that manipulate the DSP, producing sound.

Data / Memory Maps

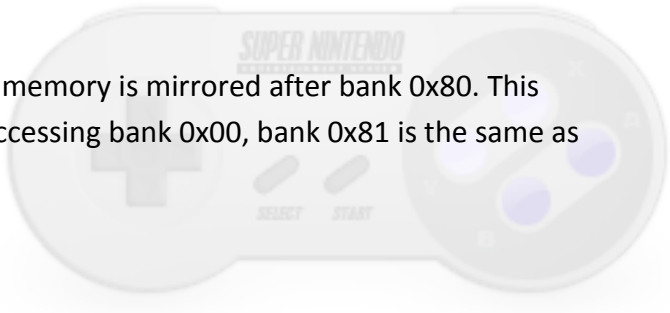
Unlike a traditional application, the data store used by an emulator is the implementation of the system's memory. In SNO, the **Memory** interface defines how other parts of the system interact with the memory, while implementing classes determine the actual layout of memory.

LoROM vs. HiROM

There are two major memory maps used by most SNES games: Mode 20 and Mode 21, which are also known as **LoROM** and **HiROM** respectively. The main difference between the two modes is that LoROM uses 32-kilobyte chunks for mapping memory addresses to ROM data, while HiROM uses 64-kilobyte chunks. HiROM removes a few sections of memory as well to allow for more overall ROM data to be mapped.

FastROM vs. SlowROM

In both HiROM and LoROM modes, the layout of memory is mirrored after bank 0x80. This means that accessing bank 0x80 is the same as accessing bank 0x00, bank 0x81 is the same as bank 0x01, and so on.



The difference between the two sections of memory is that addresses below bank 0x80 are considered **SlowROM**, while addresses within or above bank 0x80 are **FastROM**. The difference between the two is that SlowROM can only be accessed by the processor while it is running at 2.68 Mhz, while FastROM can be accessed at 2.68Mhz or 3.58Mhz.

Memory Map Diagrams

Below are memory maps for LoROM and HiROM. Please note the following:

- A block's color indicates its location within the hardware. In addition, faded colors signify memory that is mirrored from another section in memory. This means that there is another memory block with a non-faded color that this block mirrors.
- If a block has the (Mirrored) label in its text, it means that the memory addresses refer to the same thing regardless of the bank you access in. For example, WRAM in LoROM mode is mirrored across banks 0x00 through 0x3F; this means that the data will be the same in all those banks, and changes will reflect across them.
- The FastROM segment in both LoROM and HiROM is a mirror of the SlowROM segment, except for the last two banks, which hold extra ROM data that cannot be read in the SlowROM segment.

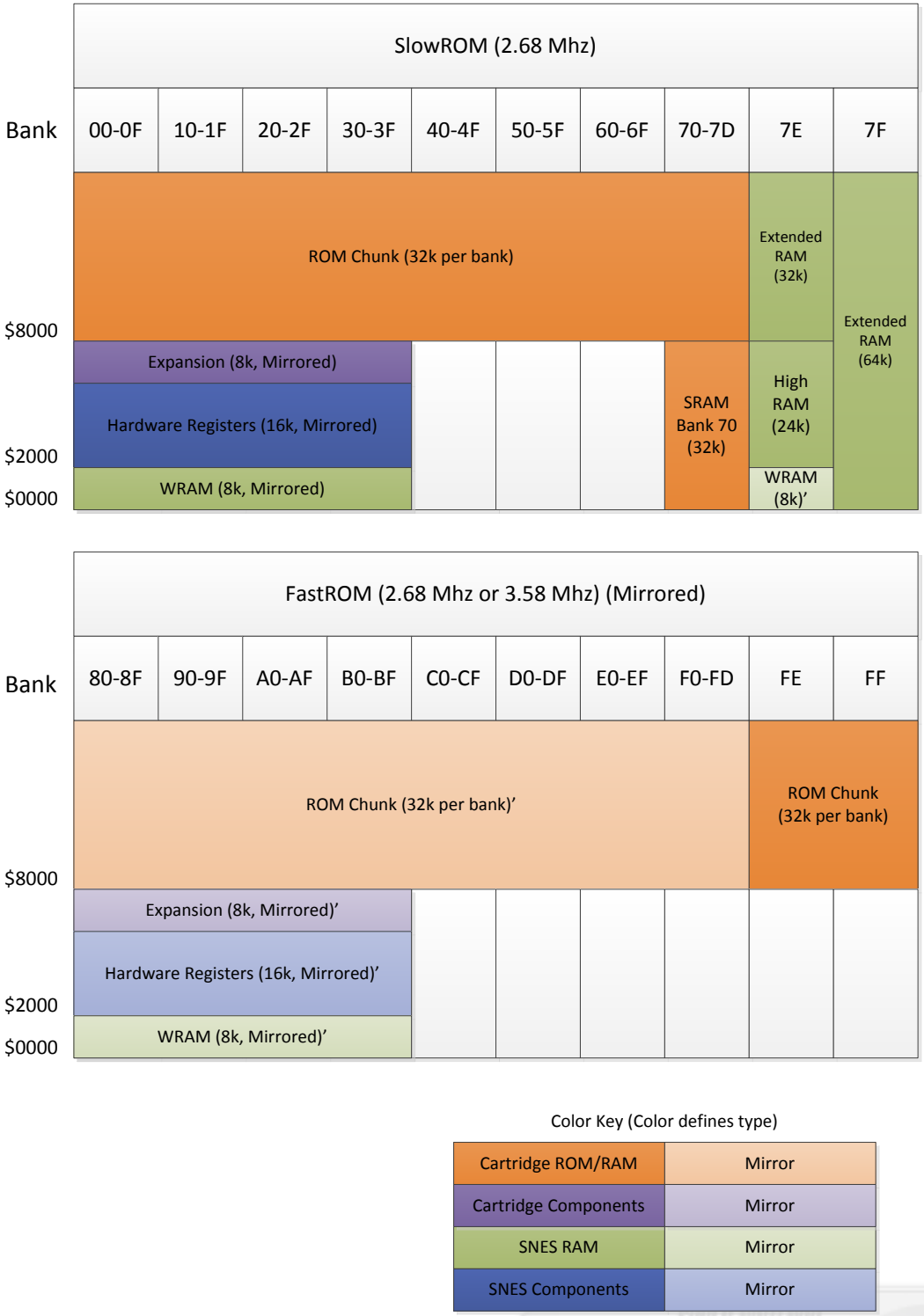


Figure 2: LoROM Memory Map

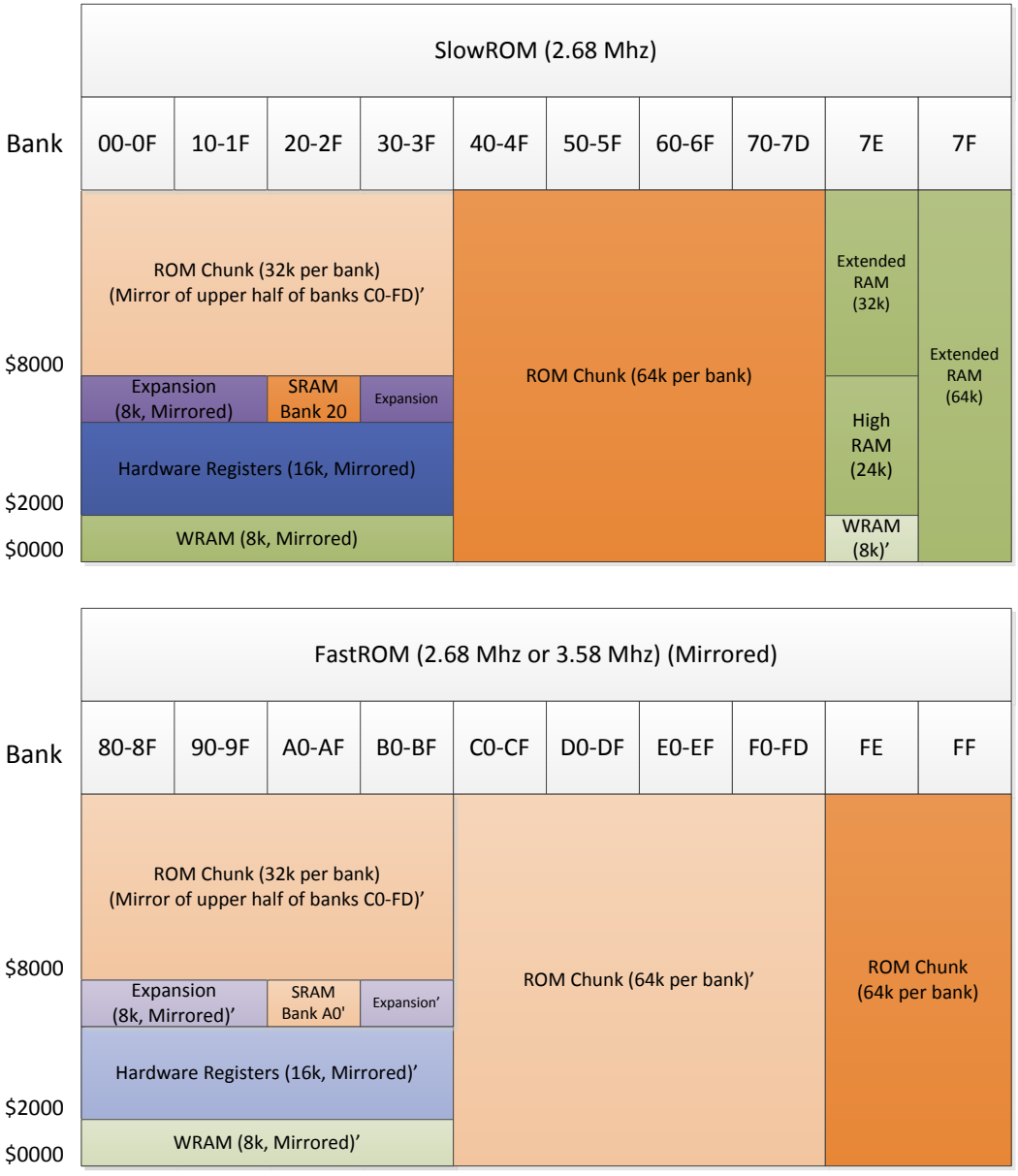


Figure 3: HiROM Memory Map

Sectors of Memory

- **WRAM:** An 8k sector of RAM that is mirrored wherever it appears. In other words, no matter what bank you access WRAM from; you will always be accessing the same 8k of RAM.
- **Hardware Registers:** Game code can use the CPU to communicate with hardware like the PPU or the Sound CPU by storing data in certain addresses within this sector. Instead of writing the data to memory, the CPU sends the data to the appropriate piece of hardware.
- **Expansion:** Special addresses reserved for communicating with expansion hardware included on certain game cartridges, such as the Super FX chip.
- **SRAM:** SRAM is a volatile form of RAM that requires a very low charge to retain data. Game cartridges use SRAM and a small battery to store saved games on the cartridge. This area of memory maps to the cartridge SRAM.
- **ROM Chunk:** Maps to ROM memory on a game cartridge. This is where the actual code, sound, and graphics for a game are mapped.
- **High RAM:** Normal system RAM used by the SNES.
- **Expansion RAM:** Normal system RAM used by the SNES.

Implementation Details

CPU

Each instruction for the SNES CPU will be an anonymous inner class that implements the Instruction interface. Each class instance will contain the logic for the instruction as well as the argument count; arguments are passed to the instruction as method parameters. The instructions will be grouped into classes according to their function such that similar functions will be part of the same class. The instructions are static member variables of the class they belong to.

The CPU class will store these instructions in a jump table. A jump table, in the context of SNO, is an array that stores references to the static member variables that store CPU instructions. The array index represents the opcode for each instruction. Thus, to perform an instruction, the CPU simply reads the opcode and retrieves the element at that index in the jump table.

Sound

The sound system is divided into two components: the SPC-700 sound CPU and a DSP. The SPC-700 is a co-processor that communicates with the main CPU through four 8-bit IO ports that are mapped to four locations in memory. The SPC-700 will be implemented using a similar class setup as the main CPU, but with a different instruction set and different memory bank.

The DSP generates a 16-bit sound wave by reading the state of a set of 8 “voices” stored in the sound system’s memory.

Video

The PPU in the SNES stores the current state of 4 BG layers that are filled with tiles, as well as a set of 128 sprites that can be displayed on screen along with the BG layers. The PPU uses this data to determine the color of each position on the screen and sends this data to the television.

Our PPU implementation will use an internal graphics buffer upon which the sprites and background data will be drawn once per frame. This buffer will then be transferred to the frontend, which will handle display of the newly-drawn buffer. The PPU will both store the video data and process it, rather than storing the data in the memory module.

User Interface

Figure 4 shows a mockup of our main user interface. It includes a web page with SNO embedded into it, along with a dropdown box for selecting a game to play. The black box that displays a screen from Super Metroid is the only part of the screen that is actually displayed by the applet; the other page elements are extra features of the web page itself, not of SNO.

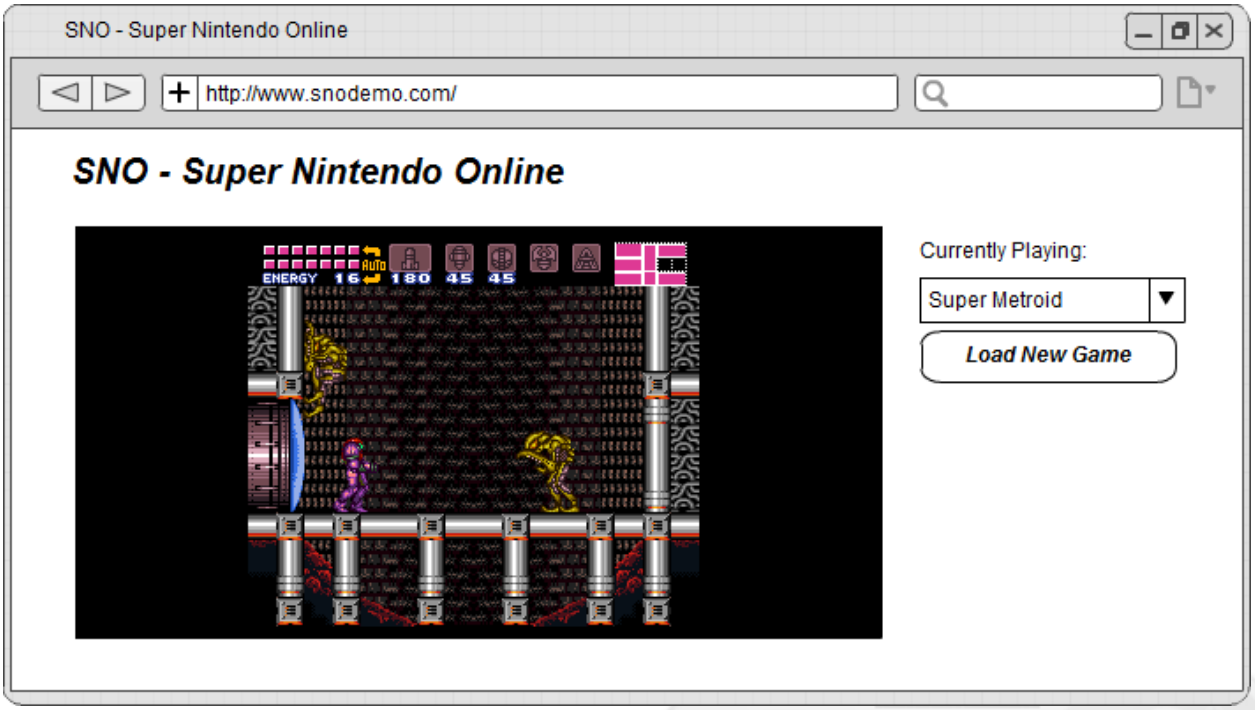


Figure 4: User Interface Mockup