NES Audio Synthesizer

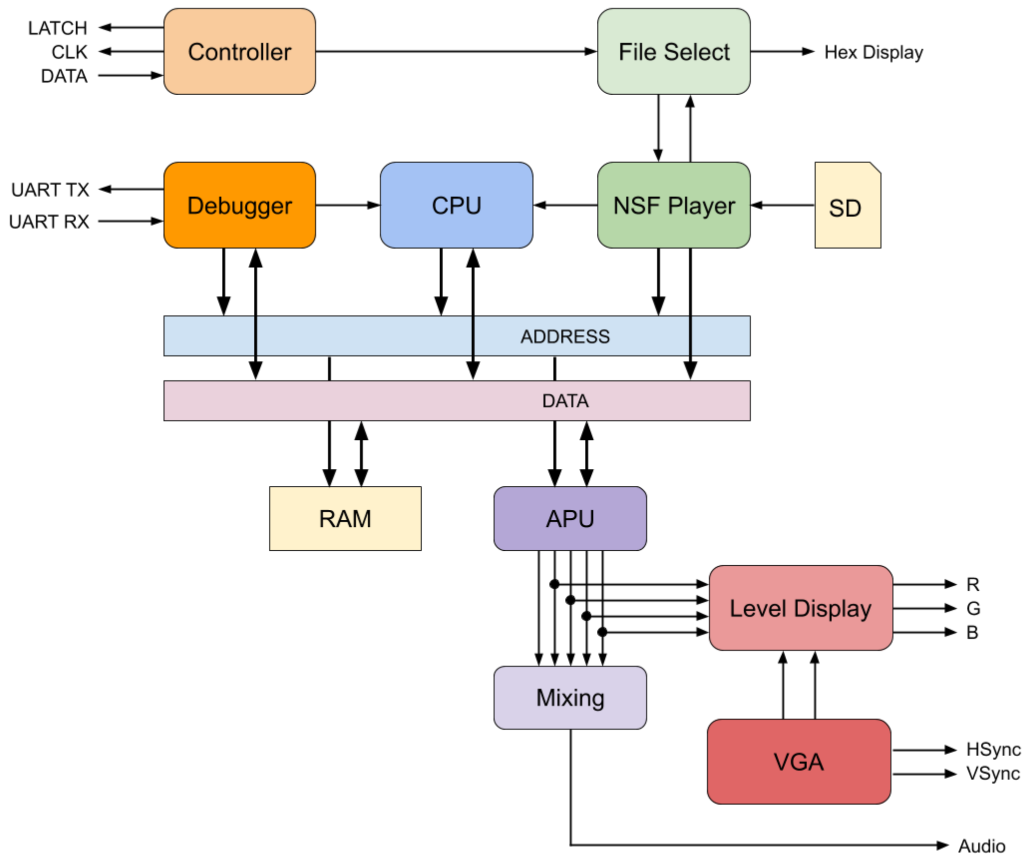
Jackson Snowden and Victor Oliveira

6.111 Fall 2020

Table of Contents

System Overview

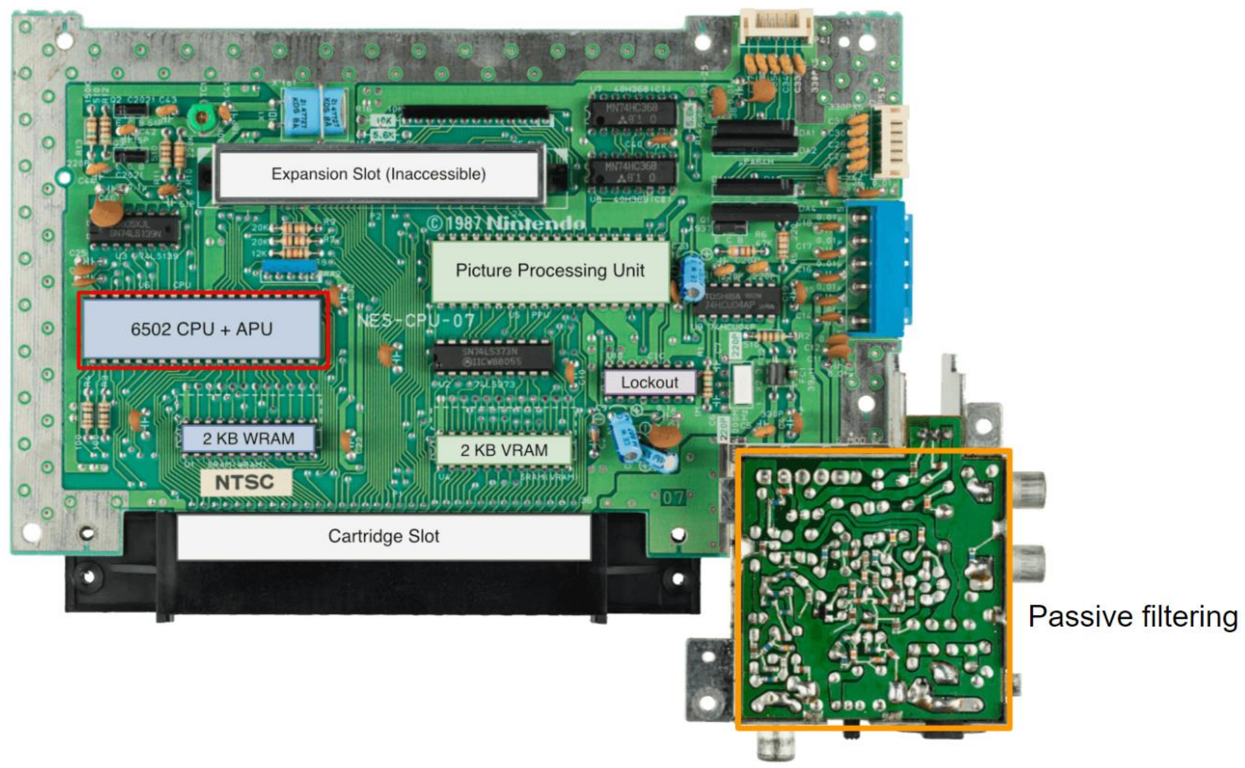
The NES Audio Synthesizer is a recreation of the original NES audio pipeline with some added features. The NES (Nintendo Entertainment System) is a video game console released in 1985 which has a characteristic sound often mimicked in modern video games. Our system aims to play music made for the NES by using original game data and recreating the same waveforms produced by the NES. This is achieved entirely in SystemVerilog.



The system contains many features beyond emulation of the NES audio pipeline, including gamepad input, SD card usage, display output, and a full debugger. Each feature in this system diagram is discussed below.

Original NES Hardware

The original NES uses a few basic components to play music: A 6502-based 8-bit CPU, a 5-channel Audio Processing Unit (APU), and an 8-bit data bus to interface with ROM storage on cartridges. The following paragraphs discuss how these components work on the NES.



CPU

The 6502 CPU is a fairly basic processor. It consists of an accumulator register (`A`) which stores the result of an arithmetic operation, two working registers (`X` and `Y`), a 16-bit program counter (`PC`) which points to the location of the next opcode in memory, an 8-bit stack pointer (`SP`) which points to the current stack index in memory, and a status register (`SR`) which indicates some results about the most recent instruction.

The CPU functions using 8-bit opcodes; only 151 of the 256 possible opcodes are valid instructions. Many of these opcodes perform the same operation (of which there are 56) but address different registers or memory. Some opcodes are also followed by 1 or 2 bytes of immediate data. As input, the CPU reads one opcode from memory per instruction cycle and may optionally read bytes from memory. As output, the CPU updates its internal registers and may optionally write bytes to memory. Opcodes may perform a wide variety of functions, including transferring data between internal registers and performing basic arithmetic. Arithmetic operations, performed by the Arithmetic Logic Unit (ALU), include addition, subtraction, bit shifting, and bit comparisons. More particular functionality of these opcodes is discussed in the CPU Emulation section.

The CPU also responds to interrupts via an external IRQ pin. When an interrupt occurs, the CPU jumps to a preset location in memory and begins executing code there. Interrupts may be disabled by certain instructions that set the interrupt disable bit in the SR; this bit is automatically set when an interrupt occurs so as to prevent repeated triggers for the same interrupt. The CPU may also be stalled by an external RDY pin, which several devices take advantage of, including the APU.

The CPU is clocked at 1.79MHz. It takes at least 2 of these cycles (and up to 8) to complete an instruction. The length of time required for a given instruction depends on its complexity; those which access memory multiple times and/or perform arithmetic to determine the output address take more cycles. In addition, any instruction may take an extra 1 or 2 cycles depending on whether it takes a branch and/or its memory access falls on one of the 256-byte page boundaries, as extra arithmetic is required to calculate the next memory and/or PC addresses.

Memory Bus

The CPU interfaces with all devices on a 16-bit address bus and the aforementioned 8-bit data bus. This provides 65,536 bytes to be addressed, though only a subset of this references actual devices. Bytes `0x0000-0x07FF` are routed to actual RAM on the NES. A program may use RAM to store arbitrary data, though the CPU performs stack operations exclusively at `0x0100-0x01FF`. Bytes `0x2000-0x2007` access the Picture Processing Unit (PPU), which is not discussed here. Bytes `0x4000-0x4017` access the APU. Bytes `0x8000-0xFFFF` access ROM storage on cartridges. Any byte ranges not mentioned are either unused or mirrors of their preceding byte ranges.

Audio

The APU receives bytes from the CPU via the memory bus and generates waveforms in response. It produces 5 independent waveforms: two square waves, one triangle wave, one noise wave, and one delta-modulation wave which can play PCM samples. These waves are passively mixed in several stages to produce a mono audio output.

Each square wave’s frequency, duty cycle, and volume may be controlled independently. In addition, the APU includes several function blocks which can automatically modulate frequency and volume over time based on preset commands. This requires relatively little CPU overhead to produce many varied waveforms.

The triangle wave’s frequency may be controlled, but its volume and general shape are constant. On the NES, its volume may be slightly modulated based on other channels’ outputs due to the complicated, non-linear mixing. This feature is not present in our implementation.

The noise wave is based on a pseudo-random number generator which may be controlled to produce different frequencies of noise. The effect is generally similar to white noise, though notably sharper and more percussive.

More particular functionality of each channel, as well as mixing, is discussed in the APU Emulation section.

Emulation

Each of the modules discussed in the Original NES Hardware section has been emulated with varying levels of abstraction to create a system that produces waveforms almost identical to those of the NES. The result is a system that can play music with an authentic sound, and without requiring any of the picture processing or user input of the original NES pipeline.

CPU

The 6502 CPU has been emulated at an instruction-level; that is, it completes instructions with the same accuracy and within the same amount of time as the original 6502, but does not perform the exact same operations on each cycle during which the instruction takes place. Instructions are instead completed on the 25MHz system clock, taking anywhere from 5 to 17 of those cycles. After completing an instruction, the CPU waits a specified amount of time that it would have taken for the original 6502 to complete the same instruction.

On reset, the `PC` is set to the 16-bit word stored at `0xFFFC-0xFFFD`. On an interrupt (IRQ), the `PC` is set to the 16-bit word stored at `0xFFFE-0xFFFF`. The addresses stored in those locations are determined by the external memory. This is how the CPU determines where to begin fetching instructions.

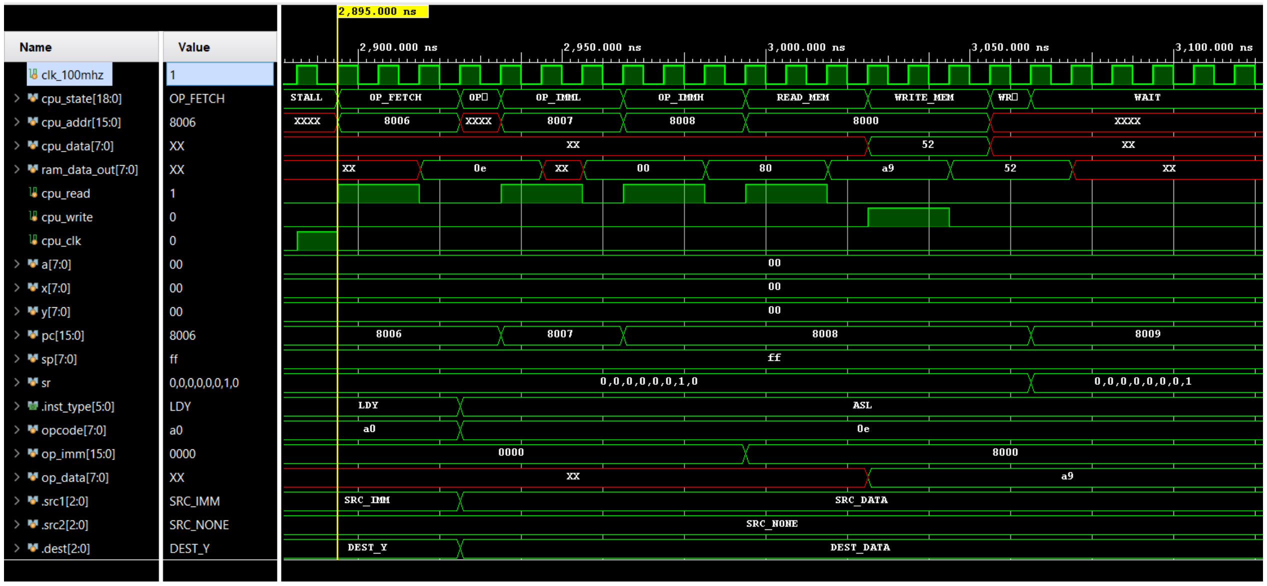
All 151 opcodes are implemented. Each opcode read from memory is decoded by the module `cpu_decode`. This module is entirely combinational, providing its result within one cycle. It outputs a human-readable struct `d_inst` which contains the following:

- Instruction type: One of 56 basic operations
- Addressing type: Absolute, indirect, zero-page, and several other variations of these
- Sources (1 and 2): Source registers for ALU input
- Destination: Destination register for ALU output
- Memory read/write flags: Set if the instruction reads or writes a byte from or to memory (a single instruction may do both)
- Data address: Location in memory to perform a read and/or write
- Indirect address: Location in memory which contains the target data address (in case of indirect addressing mode)
- Length: Number of bytes the instruction contains (1, 2, or 3)
- Cycles: Number of 1.79MHz cycles this instruction should take (2 to 7)

Most of this decoded information is passed into the module `cpu_execute`, which performs any arithmetic operations and informs the CPU of what results to write to registers or memory. This module is entirely combinational, providing its result within one cycle. It outputs a human-readable struct `e_inst` which contains the following:

- Data: The 8-bit value to write to memory
- Next `A/X/Y/PC/SP/SR`: The values to write into each of the `A/X/Y/PC/SP/SR` registers
- Extra cycles: Number of extra 1.79MHz cycles this instruction should take based on whether a branch is taken or memory access crosses a page boundary (0, 1, or 2)

All of the `d_inst` parameters determine the path taken by the overall CPU state machine, and thus how many 25MHz cycles it takes to execute. The next state is always determined by the logic `branch_state`, which takes into consideration addressing modes, memory access modes, and stack operation types. There are 19 CPU states defined in `cpu.sv`. Most of these states exist so that the CPU may wait for memory access to complete before continuing execution, which takes 3 cycles per byte accessed. No single instruction passes through all 19 states; the most lengthy instructions are those which read 2 immediate bytes, read 1 data byte, and write 1 data byte, taking 17 cycles. Below is a simulation of the CPU executing the `ASL` instruction with absolute addressing, which reads a byte from memory, performs a single left bit shift, and writes the result to the same location in memory:



This opcode `(0E)` *reads a value from memory, modifies it, and writes it back. This is indicated by* `src1` *being "SRC_DATA" and* `dest` *being "DEST_DATA".* `0xA9` *is read from* `0x8000` *and* `0x52` *is written back. It takes 17 cycles at 25Mhz.*

Stack operations represent a significant portion of the CPU state machine. A different state exists for each possible stack operation (`STACK_A`, `STACK_SR`, `STACK_PCH`, and `STACK_PCL`). Some instructions (`BRK` and `RTI`) push or pull both `SR` (1 byte) and `PC` (2 bytes) to or from the stack. In the case of a push (`BRK`), `PC` is pushed before `SR`. Thus, in the case of a pull (`RTI`), `SR` must be pulled before `PC` since pulling accesses the stack from the most recently added element. This results in a complicated determination of the next state when stack operations are involved. Due to its abstract implementation, the emulated CPU is more flexible than the original 6502. It could write up to 4 bytes to the stack (`A`, `SR`, and `PC`) at once, but no official instruction does this. It could also combine stack operations with complex (indirect) addressing modes in a single operation, but official instructions never do both.

The emulated CPU handles interrupts similarly to the 6502. An IRQ input causes the CPU to jump to the IRQ vector in memory when it goes high. In the emulated CPU, a rising edge on the IRQ input is detected and latched at any time, but it is not acted on until the CPU reaches the pre-fetch (stall) state. At this point, if an IRQ has been detected, the CPU injects a `BRK` instruction instead of reading the next opcode. A `BRK` instruction is a software-generated interrupt which pushes the `PC` and `SR` to the stack, sets the interrupt disable flag of the `SR`, and jumps to the IRQ vector. The externally generated interrupt must perform the same tasks, so it was decided to simply inject a `BRK` instruction rather than create special states to handle the interrupt routine. The CPU maintains a flag indicating whether the `BRK` instruction currently in the pipeline was software-generated or injected by the IRQ detection, which is used to determine whether to set or clear one of the flags in the `SR` register when pushing it to the stack. If a `BRK` instruction is in the pipeline, the CPU jumps to the IRQ vector only after it finishes executing the instruction.

Aside from the lack of cycle-accurate memory access, the emulated CPU includes the following functional differences with the original 6502:

- Non-maskable interrupts (NMIs): The 6502 has a separate NMI pin to trigger interrupts that cannot be disabled internally. This is not implemented here since it is only used in the NES graphics pipeline.
- Stall on invalid opcodes: in the case of invalid opcodes, the 6502 performs some undefined function or halts completely. The emulated CPU instead detects an invalid opcode and triggers an error state. In this state, the CPU stalls until a reset or IRQ occurs.
- Stall on external input: like the 6502, the emulated CPU may be stalled by an external `stall` input, but it does so only after the current instruction has finished and it is ready to fetch the next opcode. The 6502 may be stalled at any cycle during execution.
- Debugger interface: The emulated CPU includes a separate 32-bit data bus used by the debugger to read and/or modify the `A`/`X`/`Y`/`PC`/`SP`/`SR` registers as well as the instruction and cycle counters. The CPU also outputs its current status (waiting or executing) so the debugger may selectively stall.

CPU Design Insights

The `cpu_decode` module contains a case statement with 151 elements to decode each of the 151 possible opcodes. This was facilitated by using a Python script to parse some info from a website detailing each opcode and generate SystemVerilog code. While this is the most straight-forward way to decode opcodes, and perhaps the quickest, it can be done much more concisely. Certain bits in every opcode are common amongst opcodes that perform similar functions or have the same addressing modes. It is possible to split an opcode into bit-fields and perform a case selection on those smaller bit-fields. However, there are many exceptions to the regularities of these bit-fields, so it becomes a much less straight-forward operation when done this way.

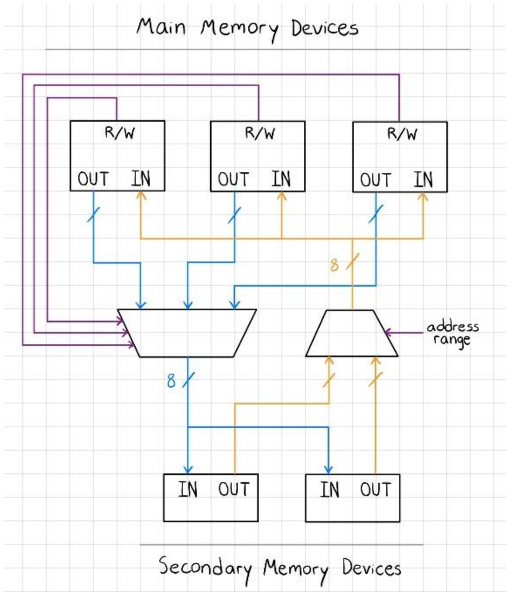
The 6502 contains a bug with indirect addressing which had to be emulated. When the location of an indirect address lies on a page boundary (i.e. `0x02FF-0x0300`), the 6502 fails to carry when incrementing the address. In effect, it reads from `0x02FF` and then `0x0200` instead of `0x0300`. This “feature” is easily missed but should certainly be emulated because programs will occasionally expect this behavior.

Where in the state machine interrupts are handled is important because interrupts set the interrupt disable bit, which disables further interrupts from being detected. Our first CPU implementation erroneously checked for the interrupt disable bit after execution when deciding whether to jump to the IRQ vector. As a result, if a `BRK` instruction occurred (either from software or injected by the IRQ handler), then the interrupt disable bit would be set during execution of that `BRK` instruction. Then, when it came time to jump to the IRQ vector, interrupts were already disabled, and no action was taken. The solution was to check the interrupt disable bit before instruction execution and act on it after execution.

Most incorrect CPU behavior was discovered through comparisons with the popular FCEUX emulator for computers. It is strongly recommended to use the debugger of this emulator or a similar tool and compare every detail, such as the values of each register after execution, the values written to memory, and the number of instructions run before a particular breakpoint. The number of instructions has been the best tool for comparison because incorrectly executed instructions usually lead to incorrect branches, and thus a different number of instructions execute before a routine ends. It is very unlikely that two CPUs would execute the same number of instructions while only one of them made a fatal error.

Memory Bus

The emulated memory bus mimics the original very closely, consisting only of 16 bits for addressing and 8 bits for data. There are main devices, which have an address output and may perform reads and writes, and secondary devices, which have an address input and may be written to or read from. Each device has a data-in port and a data-out port. Instead of connecting all memory devices with wires only, each main data-out port is connected to a MUX that delivers data to all secondary devices depending on which main device is trying to read or write. Similarly, each secondary data-out port is connected to a MUX that delivers data to all main devices depending on the address range:



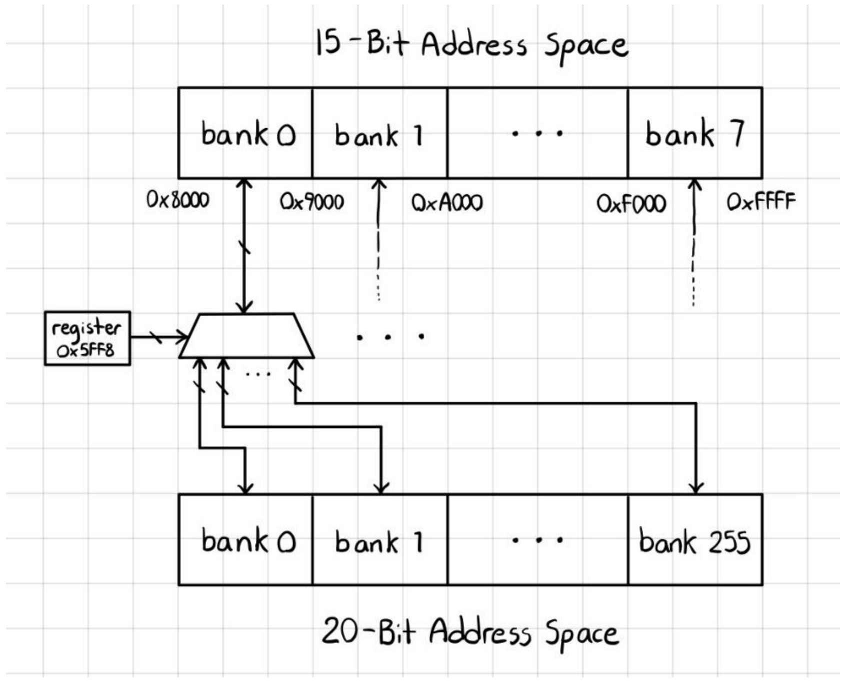
Only the data bus is shown here. The address bus follows a similar layout, but uni-directional.

When accessing any address other than `0x4000-0x4017`, main data is routed to the memory module. This module consists of two RAM blocks – one for storing data during execution (as the NES RAM does) and one for storing program data. The latter emulates an NES cartridge. Which RAM block is accessed depends on the address range, discussed in the Original NES Hardware section.

While the program data address range (`0x8000-0xFFFF`) consists of only 32kB (15 bits) of addressing space, up to 1MB (20 bits) of addressing is allowed with the use of mappers. Many types of mappers are found in NES cartridges, though our implementation uses a mapping scheme developed specifically for the NSF file format.

The 32kB address space is split into 8 banks, each 4kB in size. Similarly, the 1MB address space is split into 256 banks, each 4kB. Each of the 8 banks in the 15-bit address space may be mapped to any of the 256 banks in the 20-bit address space. Thus, a device may switch banks in order to access the entire 20-bit address space while only outputting addresses in the 15-bit range (`0x8000-0xFFFF`).

Banks are switched by writing values to a dedicated location in memory. Memory addresses `0x5FF8-0x5FFF` are reserved for this. Each of these locations corresponds to one of the banks in the 15-bit address space; `0x5FF8` is bank 0 (`0x8000-0x8FFF`), `0x5FF9` is bank 1 (`0x9000-0x9FFF`), etc. To set a bank, a program can write an 8-bit value to one of these locations, and that bank (0 to 7) in the 15-bit address space will then point to a bank (0 to 255) in the 20-bit address space determined by the value written.



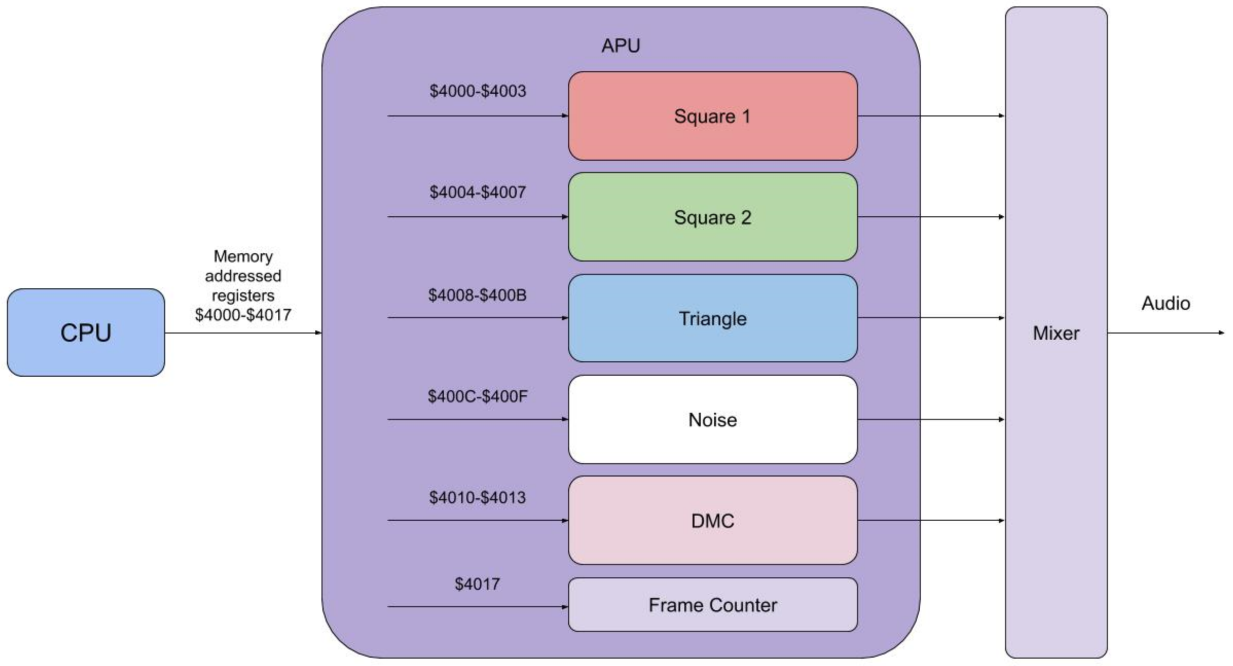
Although our implementation uses the full 20-bit addressing, only 18 bits are actually routed to the RAM block. This is because the Nexys 4 DDR does not have 1MB of block RAM, so instead only 262kB is used.

When accessing addresses `0x4000-0x4017`, main data is instead routed to the APU. All addresses except `0x4015` are write-only, and the APU uses these writes to determine its function. APU registers are discussed further in the APU Emulation section.

Memory Bus Design Insights

In our design, memory mapping was implemented fairly late. Most testing was done using a single 64kB block of RAM, one byte allocated for every address in the 16-bit address space. Programs that do not use bank switching were used for early testing. Thus, all COE files used to initialize RAM during testing were 64kB. Having a continuous block of RAM with no bank switching makes clear exactly what code is being run at any point, and how memory is being modified. This also makes it much simpler to compare results with an emulator debugger.

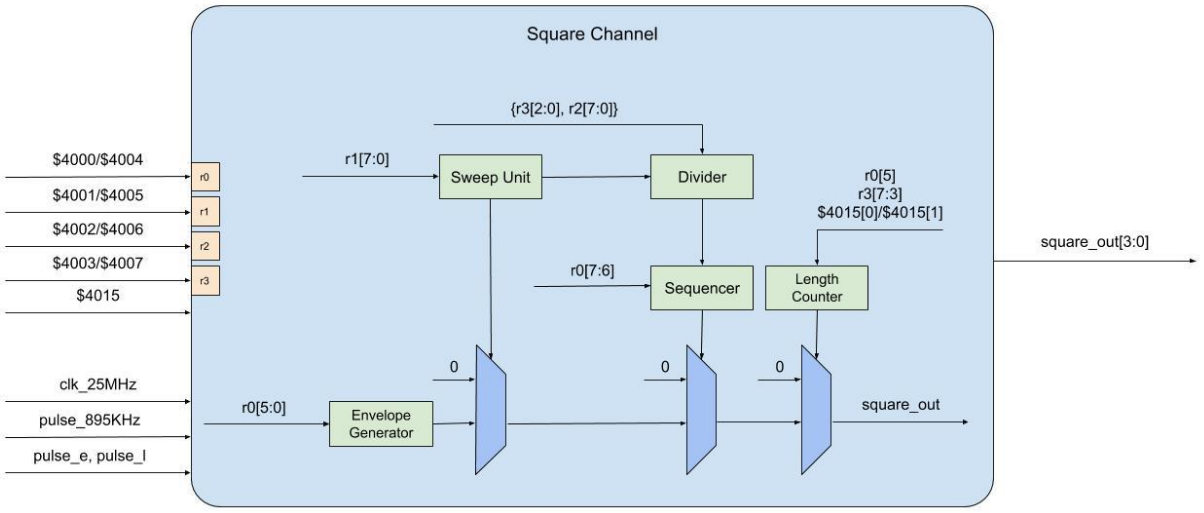
APU and Mixing



The APU is comprised of 5 sound channels: 2 Square Wave Channels, one Triangle Wave Channel, one Noise Wave Channel, and a Delta Modulation Channel (DMC) that plays samples from memory. Each channel is built with some fundamental units, some of which are shared between channels. The CPU sends relevant data to the APU through memory mapped registers, with addresses $4000 - $4017. Registers $4000-$4007 are used by the Square Channel, $4008-$400B are used by the Triangle Channel, $400C-$400F are used by the Noise Channel, and $4010-$4013 are used by the DMC. Register $4015 is used to enable or disable each Channel, and also functions as a status register that can communicate interrupt requests (IRQs) and Channel status to the CPU. Finally, register $4017 is used by the APU's frame counter, which is used to generate pulses that other channels use.

The overall system uses a 25MHz clock, which is used to generate pulses at 1.79MHz, the CPU clock speed. The units within each channel require pulses at different frequencies, which are generated by the divider module and frame counter module. The divider module simply takes in the 25MHz clock, a pulse signal, and a period P as input, and outputs a pulse when P + 1 input pulses have occurred. The frame sequencer is used to generate three signals: a 240Hz pulse (pulse_e) used by many units, a 120Hz pulse (pulse_l) used by other units, and a 60Hz pulse used as an IRQ to the CPU if bit 6 of register $4017 is clear. It has a second mode that never outputs IRQs, and outputs the other 2 pulses at 4/5 frequency when bit 7 of register $4017 is set. The frame sequencer module uses the frequencies of the American Version of the NES, and thus the APU will not work for the PAL version.

Square Channels



The Square Channels use pulse_e, pulse_l, and a pulse at half the CPU clock speed (generated by a divider with period 2). There are 2 Square Channels, where Channel 1 uses registers $4000-$4003, and Channel 2 uses registers $4004-$4007. Bit 0 of register $4015 is used to enable Channel 1, and bit 1 is used to enable Channel 2. The Square Channel is built from 5 units: an Envelope Generator, a Sweep Unit, a Length Counter, a Divider, and a Sequencer.

The Envelope Generator is responsible for determining the output volume of the Square Channel. It is also capable of volume decay, used to fade out notes. It uses bits 0 – 5 of $4000/$4004 as inputs, as well as the 240Hz pulse, pulse_e. Bits 0-3 determine the Generator's output volume if a constant volume is desired, or they determine the period P_d for the decay timer. If bit 4 is set, the volume should be constant, otherwise it will decay at a rate of 240Hz/(P_d+1). If bit 5 is set, the volume will loop back to max volume (15) once it reaches 0. The Generator is reset whenever there is a write to register $4003/$4007.

The Sweep Unit is used to sweep the Divider's period up or down, and is also responsible for muting the channel if the desired output frequency is too high or too low. It uses registers $4001/$4005 as inputs, as well as the 120Hz pulse, pulse_l. The unit sweeps the current Divider frequency by right-shifting the current period, then adding or subtracting that value from the current period. Bits 0-2 are used to determine how many bits the current period should be shifted before the addition/subtraction. Bit 3 is used to determine whether the operation is addition or subtraction, i.e. whether the sweep in frequency is down or up. If bit 3 is set. Square Channel 1 subtracts the shifted period + 1 from the current period, whereas Square Channel 2 subtracts just the shifted period. If bit 3 is 0, the shifted period is added to the current period.

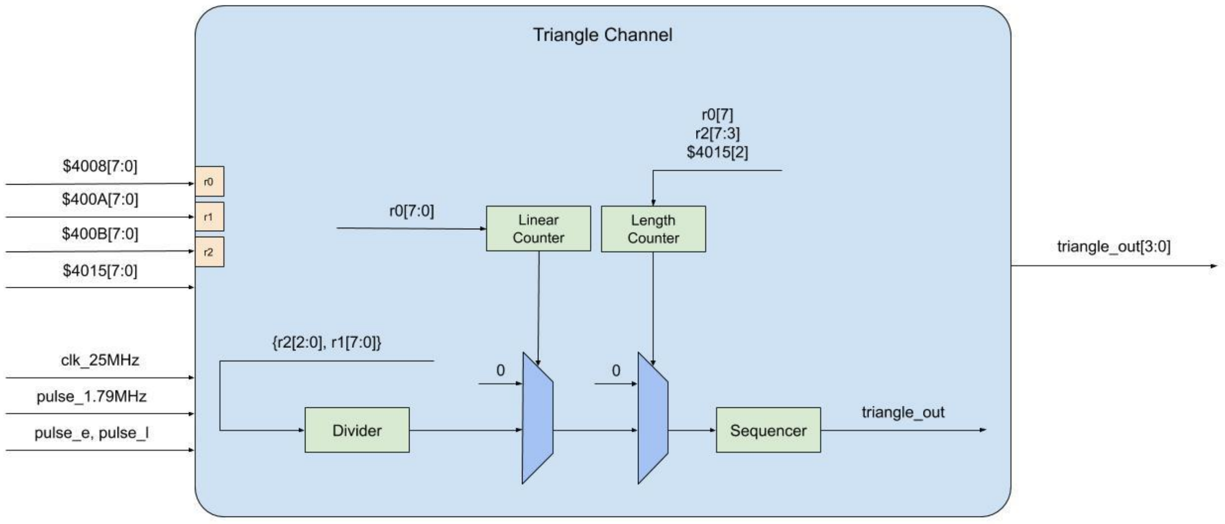
Bits 4-6 set the sweep period P_s, so the sweep frequency becomes 120Hz/(P_s + 1). Bit 7 enables the sweep function, if it is 0 then the sweep unit never updates the current period. If the current period is ever less than 8, which would create extremely high frequency notes, or if the sweep function is enabled and the target period is higher than 2047, which would create extremely low frequency notes, the sweep unit mutes the channel. The Sweep Unit is reset whenever there is a write to register $4001/$4005.

The Length Counter is used to control note duration when it is enabled and not halted. It is enabled by bit 0/1 of register $4015. When the Length Counter is disabled, it also disables the channel. Bit 5 of register $4000/$4004 will halt the Length Counter when set, allowing for indefinitely long notes. The length counter is clocked by pulse_l, which will count down from a length L when not halted, muting the channel if it reaches 0. The length L is determined by a look up table (LUT) indexed by bits 3-7 of register $4003/$4007. The values for L in the LUT seem to correspond to typical note durations, such as quarter or eighth notes, at various BPMs. The values were found at https://wiki.nesdev.com/w/index.php/APU_Length_Counter, where previous enthusiasts seem the have reverse engineered the entire operation of the NES. The Length Counter is reset whenever there is a write to register $4003/$4007.

The Divider used by the Square Channels is the same described above. In this case, it takes in an 895KHz pulse generated by dividing the 1.79MHz pulse signal by 2. The period is initialized as the concatenation of bits 0-2 of register $4003/$4007 with bits 0-7 of register $4002/$4006, extended to 12 bits ({1'b0, r3[2:0], r2}). If the sweep is enabled, in takes this as input and begins sweeping the frequency. The Divider input period P_{div} is either the original period if the Sweep Unit is disabled, or the current period based on the Sweep Unit output, which is constantly updating. The Divider's output frequency will be 895KHz/(P_{div} + 1).

Finally, the Sequencer is used to generate the square wave itself at the desired duty cycle. It uses the Divider output pulse as a clock. Bits 6-7 of register $4000/$4004 determine the desired duty cycle: 12.5% if 0, 25% if 1, 50% if 2, and 75% if 3. The Sequencer, as its name implies, generates an 8-step sequence. Based on the duty cycle, that sequence is 0 for a percentage of the steps equal to the duty cycle, and 1 for the rest. If the sequencer is outputting a 1, the channel is muted. If it is outputting a 0, the channel outputs the volume generated by the Envelope Generator. Since there are 8 steps in each sequence, the overall output frequency will be F_{div}/8, or 895KHz/(8*(P_{div} + 1)), or even better, 1.79MHz/(16*(P_{div} + 1)). The sequencer will be reset to its first step whenever there is a write to $4003/$4007.

Triangle Channel



The Triangle Channel uses pulse_e, pulse_l, and the CPU clock pulse at 1.79MHz. It uses registers $4008-$400B, though register $4009 is unused. Bit 2 of register $4015 is used to enable the Triangle Channel. The Triangle Channel is built from 4 units: a Length Counter, a Linear Counter, a Divider, and a Sequencer.

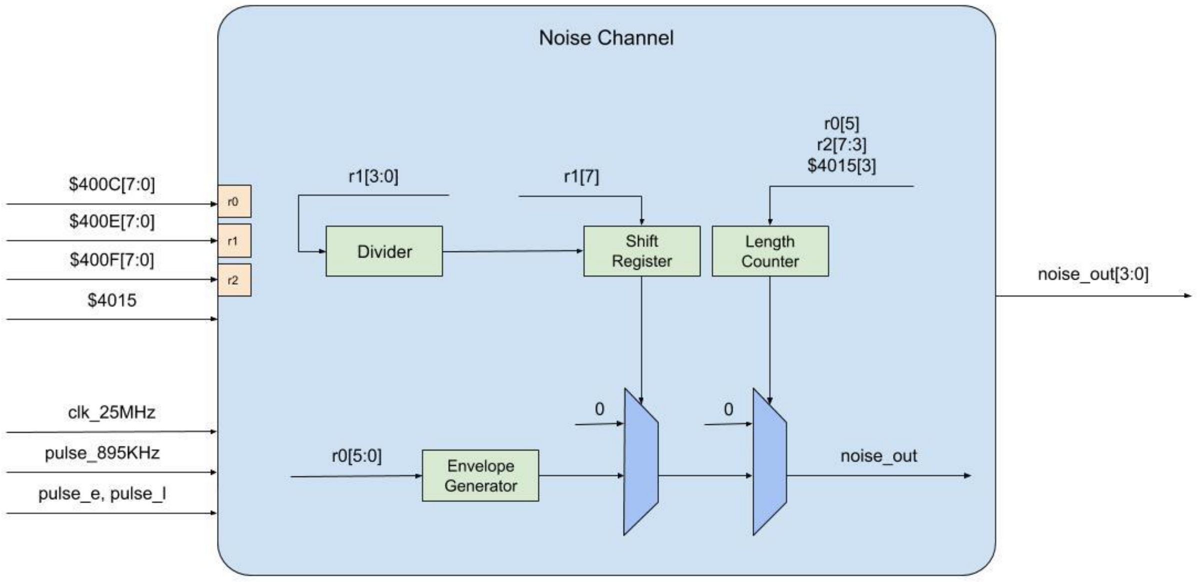
The Triangle Channel's Length Counter behaves identically to the Square Channels' Length Counter. The relevant registers are different, on the other hand. Bit 2 of register $4015 is used to enable the unit. Bit 7 of register $4008 will halt the unit. The LUT index is determined by bits 3-7 of register $400B. The unit is reset whenever there is a write to register $400B.

Unlike other units, the note duration for the Triangle Channel can also be controlled by the Linear Counter. It can count a number of pulses from 0 to 127, allowing for much more accurate duration control. It uses bits 0-7 of register $4008 as inputs, as well as pulse_e. Bits 0-6 determine the desired duration, T_l. Bit 7 halts the unit alongside halting the Length Counter. If Bit 7 is cleared, the system will begin to count down the desired duration, muting the channel once it reaches 0. The actual duration in this case will be determined by whichever had a shorter duration between the Length Counter and Linear Counter. The Linear Counter is reloaded with T_l whenever there is a write to register $400B.

The Triangle Channel's Divider behaves identically to the Square Channels' Divider, but it has a constant period. The period P_{div} is determined by the concatenation of bits 0-2 of register $400A with bits 0-7 of register $400B, extended to 12 bits ({1'b0, r2[2:0], r1}). Thus, its output frequency is 1.79MHz/(P_{div} + 1).

Finally, the Sequencer is used to generate the triangle wave itself. It uses the Divider output pulse as a clock. The Sequencer in this case generates a 32-step sequence. It begins at 0, then counts up to 15 once per Divider cycle. Once it reaches 15, it stays there for a cycle, then begins counting down to 0. Once it reaches 0, it stays there for a cycle and starts the whole process again. Since there are 32 steps in each sequence, the overall output frequency will be 1.79MHz/(32*(P_{div} + 1)). If the channel is ever muted by one of its units, it first counts back down to 0 before actually being muted. This was done to remove popping noises that became apparent when the Triangle Channel was being muted by transitioning directly to an output of 0.

Noise Channel



The Noise Channel generates noise waves used for percussion and sound effects by creating waves with pseudo-random frequencies. It uses pulse_e, pulse_l, and a pulse at half the CPU clock speed. It uses registers $400C-$400F, though register $400D is unused. Bit 3 of register $4015 is used to enable the Noise Channel. The Noise Channel is built from 4 units: an Envelope Generator, a Length Counter, a Divider, and a Shift Register.

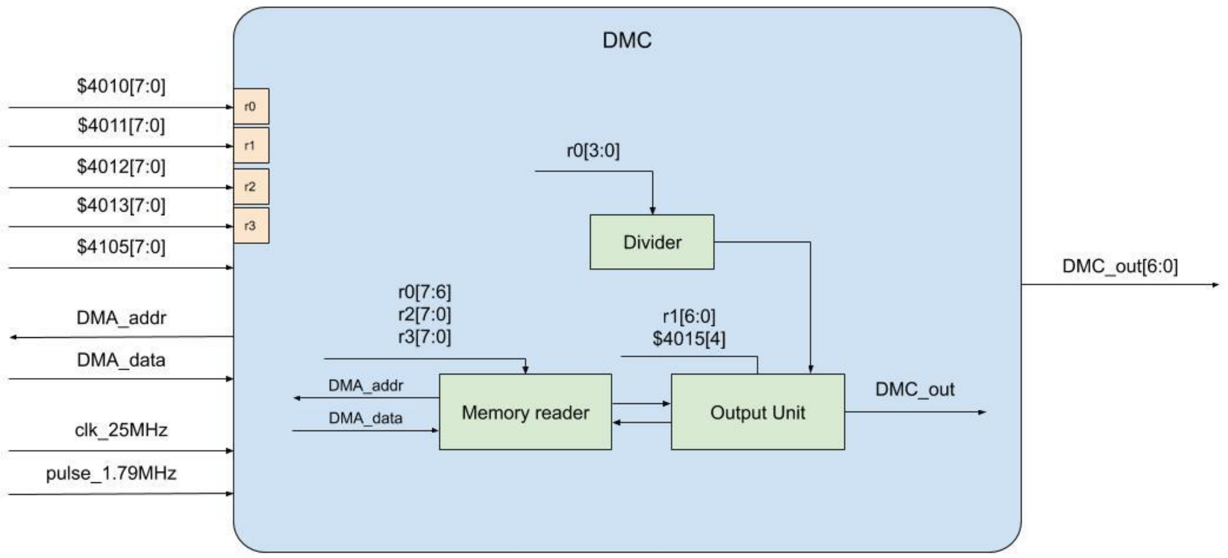
The Noise Channel's Envelope Generator behaves identically to the Square Channels' Envelope Generator. The relevant registers are different, on the other hand. Bits 0-3 of register $400C is used to determine the constant volume or P_d. Bit 4 of register $400C determines if the output should be constant or decay. Bit 5 of register $400C determines if the volume should loop.

The Noise Channel's Length Counter behaves identically to the Square Channels' Length Counter. The relevant registers are different, on the other hand. Bit 3 of register $4015 is used to enable the unit. Bit 5 of register $400C will halt the unit. The LUT index is determined by bits 3-7 of register $400F. The unit is reset whenever there is a write to register $400F.

The Noise Channel's Divider behaves identically to the Square Channels' Divider, but it has a constant period. The period P_{div} is determined by the entries of a LUT. The index for this LUT is determined by bits 0-3 of register $400E, while its entries were found at https://wiki.nesdev.com/w/index.php/APU_Noise. Its output frequency can be calculated, but isn't extremely relevant as the goal is to generate a wave with random frequency.

Finally, the Shift Register is used to make the wave's frequency pseudo-random. This unit comprises a simple 15-bit right-shift register loaded with the value 1 at power-up. It has 2 modes of operation based on bit 7 of register $400E. Whenever this unit receives a pulse from the Divider, it calculates the XOR value of bit 0 and either bit 6 (if the mode bit is set) or bit 1 (if the mode bit is clear), then it shifts the bits in the register to the right once. It fills the new empty bit, bit 14, with the XOR calculated previously. This creates pseudo-randomness. If bit 0 of the shift register is set, the channel is muted, otherwise it outputs the volume generated by the Envelope Generator.

DMC



The DMC reads and plays audio samples from memory. This requires the DMC to have direct memory access (DMA), making it much more complex than the other channels. It only uses the CPU clock pulse at 1.79MHz, and uses the registers $4010-$4013. Bit 4 of register $4015 is used to enable the DMC. The DMC is built from 3 units: a Divider, a Memory Reader, and an Output Unit.

The DMC's Divider behaves identically to the Noise Channel's Divider. The period P_{div} is determined by the entries of a LUT. The index for this LUT is determined by bits 0-3 of register $4010, while its entries were found at https://wiki.nesdev.com/w/index.php/APU_DMC. Its output frequency can be calculated, but isn't extremely relevant as the goal is play samples rather than waveforms.

The Memory Reader is the most complex unit in the APU. It runs at 25MHz until it needs to access memory, at which point it runs at 1.79MHz. In order to access memory, the DMC must first stall out the CPU so that it doesn't try to read data it should not be reading. The unit will need to read memory whenever the output unit requests more data or when the unit is enabled. Writing a 1 to bit 4 of register $4015, however, does nothing if the channel was already enabled or if the current sample is not done playing. This unit uses registers $4012 and $4013. Register $4012 sets the starting address A of the sample to be played. The actual address will be $C000 + (A << 5). Register $4013 sets the length L_{samp} of the sample in bytes. The actual length will be (L_{samp} << 4) + 1. The NSF Player module described in later sections is responsible for placing the correct data in the correct memory locations so that the DMA can access it.

When the channel is enabled, it starts the memory loading process by first sending the CPU a stall signal. The CPU can take up to 7 cycles at 1.79MHz to finish an instruction, and it won’t stall until then, so the Memory Reader sends a stall signal for 8 cycles. Once the penultimate stall cycle starts, the Memory Reader starts reading from memory at the location mentioned above. After it reads the data from memory, it stores it in a sample buffer, increments the current memory location by 1, and decrements the bytes_remaining counter (initially loaded with L_{samp}) by 1. It also returns control of memory to the CPU and stops stalling it. As it plays the sample, it will continue to repeat this process whenever the current sample buffer has been used up by the Output Unit, always incrementing the memory and decrementing the bytes_remaining counter. If the memory location reaches the highest value $FFFF, it loops back to $8000. If the bytes_remaining counter reaches 0, it will either loop back to the starting memory address and reload its bytes_remaining counter if bit 6 of register $4010 is set, or send the CPU an IRQ if bit 7 of register $4010 is set. Once the system is done playing the current sample, it sends a signal to the Output Unit telling it there are no more samples.

The Output Unit is responsible for playing the sample using Delta Modulation. It uses the Divider output pulse as a clock, and uses register $4011. It starts with a load value of 0. The load value can be set to the value in bits 0-6 in register $4011 at any time by writing to register $4011. Otherwise, the load will be modified by the sample that was read from memory by the Memory Reader. When the unit receives a pulse from the Divider, it reads bit 0 from the current sample. If the bit is 1 and the current load is < 126, it adds 2 to the load. If the bit is 0 and the current load is > 1, it subtracts 2 from the load. The load should never leave the 0 – 127 range.

After processing the current bit, the Output Unit shifts the sample right so it can read the next bit. After all bits have been processed, the unit checks if there are any samples left from the Memory Reader. If there are, the Output Unit reads the sample and tells the Memory Reader to read the next sample from memory. If there are no samples left, it silences itself by preventing any change to the load. This method plays out the desired samples, but the current implementation has a strange bug that also generates high frequency sounds alongside the samples. It seems like the load is going up and down at high frequencies even when the Output Unit should be silenced, but hours of troubleshooting have borne no fruit. Thus, due to lack of time, this issue persists.

Mixing

The output of each channel is mixed before being played through PWM. The Square, Triangle, and Noise channels output a 4-bit number, while the DMC outputs a 7-bit number. A LUT approximation was used for the sake of convenience and due to limited processing speed. A 31 entry LUT was used to mix the outputs of the 2 Square Wave Channels, while a 203 entry LUT was used to mix the outputs of the Triangle Wave Channel, Noise Wave Channel, and DMC.

The entries of the Square Channel Mixer LUT were calculated with the formula $sqr\_tbl[n] = round\left(255 * \left(\frac{95.52}{\frac{8128}{n}+100}\right)\right)$, where n is the sum of the outputs of each Square Channel. The entries of the TND Mixer LUT were calculated using the formula $tnd\_tbl[n] =$

$round\left(255 * \left(\frac{163.67}{\frac{24329}{n}+100}\right)\right)$, where n is the sum of the output of the DMC, twice the output of the Noise Channel, and thrice the output of the Triangle Channel. The entries of these LUTs will add up to a value between 0 and 255, and thus they are perfect as inputs to an 8-bit audio PWM module. These approximations are based on others' work reverse engineering the NES, and the original equations were found at https://wiki.nesdev.com/w/index.php/APU_Mixer.

File System and Playback

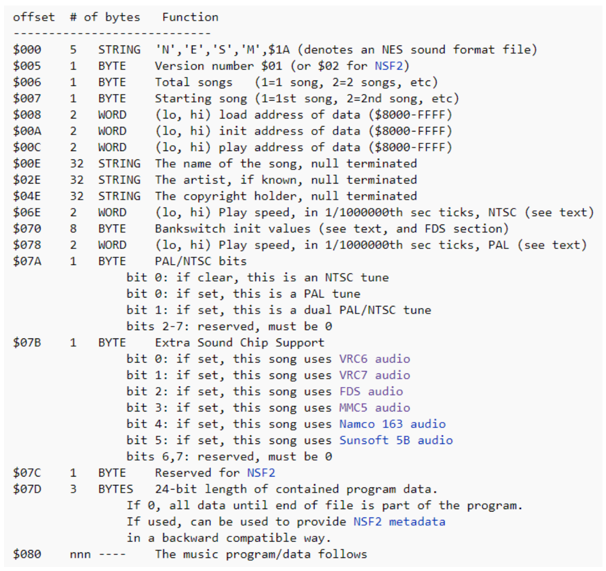
Data used for playing songs is stored in files on an SD card. An individual file and song may be selected by the user. To facilitate this, a file interpreter and file select system were implemented. The file interpreter includes SD card access, file parsing, and playback control, while the file select system includes user input from an NES gamepad.

File Structure

Our implementation utilizes data stored on an SD card in raw binary format along with the instructor-provided `sd_controller` module. Data on the card is structured as a collection of files, each file starting at the beginning of an SD card sector (a block of 512 bytes), with a single string “`END DATA`” in the last sector to indicate there are no more files.

The files used are in NES Sound Format (NSF). NSF files are a repackaging of audio code found in original game ROMs. The code is prepended with a header, which includes a constant string “`NESM`” used to identify the beginning of a file when parsing the SD card. The NSF header follows a regular format which specifies how many songs are contained in the file, how to initialize memory banks before playing songs, the addresses of certain subroutines in the code, the speed at which songs should be played, and some human-readable metadata about the file. The header also contains some flags indicating whether the file uses any of the many NES expansion audio chips, none of which are emulated in our implementation.

The code found in an NSF file may consist of any collection of opcodes (i.e. an arbitrary program), but it is guaranteed to contain an `INIT` routine which may be called once to initialize memory with the chosen song and a `PLAY` routine which may be called at regular intervals to achieve playback. The locations of these subroutines in the code is contained in the file header. Additionally, the code section may contain PCM data which is read by the DMA included in the APU for playing samples. Code may be up to 1MB in size, in agreeance with the 20-bit address space afforded by the 256 banks discussed in the Memory Bus section.



```
offset  # of bytes   Function
----------------------------
$000    5   STRING  'N','E','S','M',$1A (denotes an NES sound format file)
$005    1   BYTE    Version number $01 (or $02 for NSF2)
$006    1   BYTE    Total songs   (1=1 song, 2=2 songs, etc)
$007    1   BYTE    Starting song (1=1st song, 2=2nd song, etc)
$008    2   WORD    (lo, hi) load address of data ($8000-FFFF)
$00A    2   WORD    (lo, hi) init address of data ($8000-FFFF)
$00C    2   WORD    (lo, hi) play address of data ($8000-FFFF)
$00E    32  STRING  The name of the song, null terminated
$02E    32  STRING  The artist, if known, null terminated
$04E    32  STRING  The copyright holder, null terminated
$06E    2   WORD    (lo, hi) Play speed, in 1/1000000th sec ticks, NTSC (see text)
$070    8   BYTE    Bankswitch init values (see text, and FDS section)
$078    2   WORD    (lo, hi) Play speed, in 1/1000000th sec ticks, PAL (see text)
$07A    1   BYTE    PAL/NTSC bits
                bit 0: if clear, this is an NTSC tune
                bit 0: if set, this is a PAL tune
                bit 1: if set, this is a dual PAL/NTSC tune
                bits 2-7: reserved, must be 0
$07B    1   BYTE    Extra Sound Chip Support
                bit 0: if set, this song uses VRC6 audio
                bit 1: if set, this song uses VRC7 audio
                bit 2: if set, this song uses FDS audio
                bit 3: if set, this song uses MMC5 audio
                bit 4: if set, this song uses Namco 163 audio
                bit 5: if set, this song uses Sunsoft 5B audio
                bits 6,7: reserved, must be 0
$07C    1   BYTE    Reserved for NSF2
$07D    3   BYTES   24-bit length of contained program data.
                If 0, all data until end of file is part of the program.
                If used, can be used to provide NSF2 metadata
                in a backward compatible way.
$080    nnn ----    The music program/data follows
```

Description of an NSF file header. Source: https://wiki.nesdev.com/w/index.php/NSF

NSF Player and SD Reader

The `nsf_player` module is a large state machine responsible for parsing all files on the SD card, selecting a file based on user input, and managing playback. To do this, it interfaces with most other major modules, including the CPU, memory, SD controller, and file select. The NSF player is a main device on the memory bus.

On reset, the NSF player scans sectors sequentially on the SD card and increments the `num_files` counter every time it encounters the string “`NESM`” found in a file header. It also stores the SD sector associated with the header of each file. It stops scanning once it encounters the string “`END DATA`”. At this point, it waits for a file selection event and outputs the number of files it found (0 to 16) as well as a `scan_done` flag for use by other modules.

When an external `file_select` pulse occurs, the module latches the `file` input (0 to 15) and points the SD controller to the sector associated with that file’s header. Then, all RAM (addresses `0x0000-0x7FFF`) is initialized to 0. The file header is read from the SD card and some information is stored in registers for later use, including the offset address for loading code data, the `INIT` and `PLAY` routine addresses, playback speed, and an initialization value for each of the 8 memory banks.

After reading the header, the module enters the `READ_DATA` state. In this state, the module transfers every byte of data from the SD card to a corresponding location in the 262kB RAM block of the memory module (with the offset determined by the file header). For each byte received from the SD card, the memory address is incremented by 1. Code data from the NSF file is intended to be loaded into a contiguous block of memory and then accessed by the CPU with bank mappers. However, the NSF player utilizes the same 16-bit address bus as other main memory modules, so it must also use bank switching in order to write more than 4kB of code data into RAM contiguously. The NSF player uses only bank 0 mapping; that is, it only writes to memory in the `0x8000-0x8FFF` range. For every SD byte received in the `READ_DATA` state, the module checks if the current memory address is at the end of a bank (i.e. the last 12 bits are `0xFFF`). If it is, it stops reading the sector, increments the current bank by 1, writes the new bank value to `0x5FF8`, and then re-reads the sector where it left off. Similarly, if the current byte counter indicates the end of an SD sector, it increments the current sector and signals to the SD controller to begin a new sector read.

After loading file data into RAM, the NSF player loads further initialization values at the following locations:

- `0x5000-0x5024`: Custom routine used for handling playback
- `0x5080`: 0
- `0x5081`: Song selection (0-255)
- `0x5082`: Region (0 for NTSC, 1 for PAL)
- `0x5FF8-0x5FFF`: Initial values for banks 0-7, specified by the file header
- `0xFFFC-0xFFFD`: **`0x5021`** (custom reset vector value)
- `0xFFFE-0xFFFF`: **`0x5000`** (custom IRQ vector value)

After loading these values into memory, the NSF player enters its IRQ generation state, where it generates an IRQ pulse at the playback rate specified by the NSF file header. This rate is most often 60Hz. The IRQ pulse is fed into the CPU, and playback begins on the first IRQ. In this state, the NSF player watches for a pulse on the `file_select` input to select a new file. It also watches for a pulse on the `play_pause` input, which causes the player to toggle the CPU stall output. When the stall output is high, the CPU is stalled and ceases playback.

The custom handling routine loaded into `0x5000-0x5024` was developed to allow the CPU to playback songs rather autonomously, requiring only an external interrupt to keep timing. It consists of the following instructions:

```
0x5000: SEI       ; Set interrupt disable flag
0x5001: LDX #$FF  ; Load X with 0xFF
0x5003: TXS       ; Transfer X to SP (initialize the stack)
0x5004: LDA $5080 ; Load A with the value stored at 0x5080
0x5007: CMP #$81  ; Check if A == 0x81
0x5009: BNE $5011 ; If A != 0x81, go to 0x5011
0x500B: JSR PLAY  ; Call PLAY routine and return
0x500E: JMP $5021 ; Go to 0x5021
0x5011: LDA #$81  ; Load A with 0x81
0x5013: STA $5080 ; Store A at 0x5080
0x5016: LDA $5081 ; Load A with the value at 0x5081 (song selection)
0x5019: LDX $5082 ; Load X with value at 0x5082 (region selection)
0x501C: LDY #$00  ; Clear Y
0x501E: JSR INIT  ; Call INIT routine and return
0x5021: CLI       ; Clear interrupt disable flag
0x5022: JMP $5022 ; Jump to 0x5022 (i.e. loop infinitely)
```

Address `0x5021` is pointed to by the reset vector while address `0x5000` is pointed to by the IRQ vector. Thus, on reset, the CPU goes to `0x5021`, and on IRQ, it goes to `0x5000`. The NSF player ensures these vectors contain the correct values by writing to 0xFFFC-0xFFFF in the `WRITE_VECTORS` state. This section of code also contains the addresses of the `PLAY` and `INIT` routines derived from the file header.

On reset, the CPU goes to `JMP $5022`; this instruction is stored at `0x5022`, so it jumps to itself and goes nowhere. This is equivalent to an infinite loop that can only be broken by a reset or interrupt.

On interrupt, the memory byte at `0x5080` is read. Its value (`0x00` or `0x81`) indicates whether the INIT routine has already been run, which must happen only once each time a song is loaded. If the value at `0x5080` is `0x00`, then it branches to `0x5011`. At `0x5011`, the value `0x81` is written to `0x5080` to ensure it will not branch here again, the song selection is loaded into `A`, and the `INIT` routine is called. The `INIT` routine will eventually return the CPU to the next instruction (`0x5021`). If the value at `0x5080` is `0x81`, it will instead call the `PLAY` routine, which will also eventually return the CPU to the next instruction (`0x500E`). Regardless of which routine is called, the CPU will end at `0x5022`, where it waits for the next interrupt.

It is expected that both the `INIT` and `PLAY` routines end with an `RTS` instruction, which will ultimately return the CPU to `0x5022` where it may wait for an interrupt. It is also expected that the `INIT` routine will properly initialize memory to play a particular song based on the value of `A` when `INIT` is called.

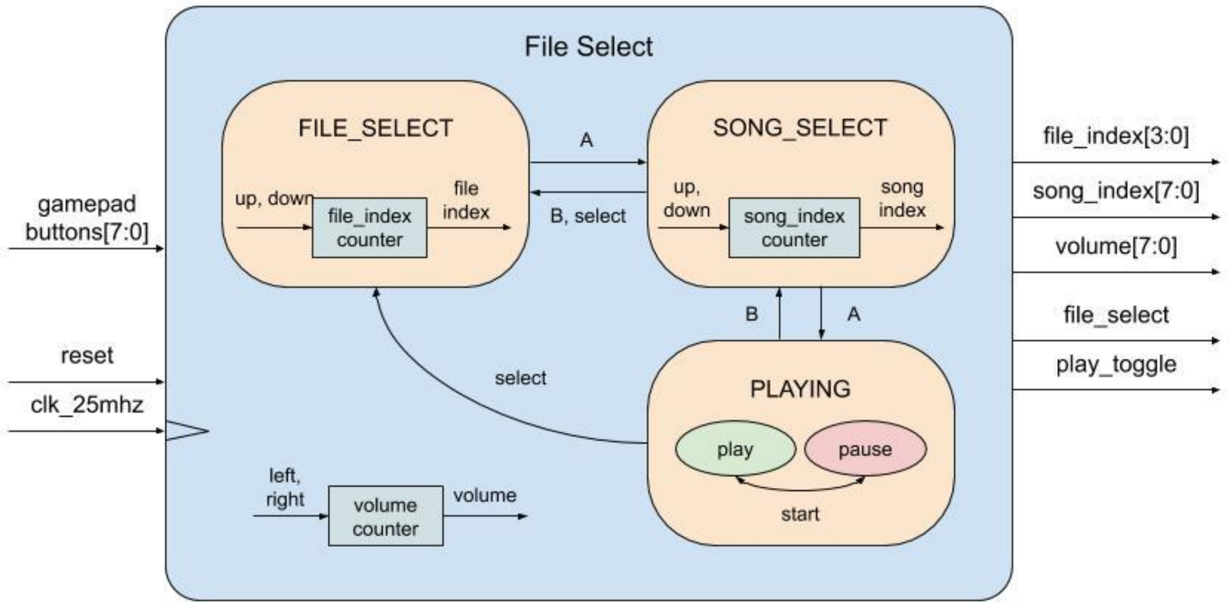
NSF Player Design Insights

Initializing RAM to 0 in the NSF player pipeline seems unnecessary considering the state of RAM on the original NES is not defined on reset. A program (i.e. an NSF file) is not supposed to assume the RAM contains any particular values prior to writing those values. However, one file (Super Mario Bros. 3) refused to play in our initial tests. After extensive debugging and comparisons with a working emulator, it appeared that Super Mario Bros. 3 was failing because it would halt if a certain location in RAM did not contain 0 on startup. Thus, it was decided all RAM should be cleared prior to loading a file. It is unclear if this is a fault in the NSF file or if our CPU is somehow failing to run some instructions which are meant to clear the RAM on startup.

The end of sector condition (i.e. when the 512th SD byte has been reached) is actually checked one cycle after the end of memory bank condition. Thus, if the end of the SD sector is encountered at the same time as the end of a memory bank, it will switch banks before starting a new sector read. It is important not to act on both conditions in the same cycle because, if they do both occur, that would result in either not incrementing the sector or not incrementing the bank depending on what order in which they are checked.

Handling memory banks while writing file data to RAM is not truly necessary. The alternative is to use a separate 20-bit memory bus between the NSF player and the 1MB RAM block. This separate bus was avoided in order to keep the memory structure streamlined and true to the original hardware, but in hindsight, it would have been much simpler to expand the memory bus and avoid the complicated state machine required to handle bank switching.

File Select and NES Gamepad Controller



The file select systems allows users to select what songs they wish to play from whichever files are present on the SD card. It also allows them to control the music playback volume, pause and unpause playback, and choose a different song if the user wishes to do so. This was done by utilizing a simple finite state machine with three states: “FILE_SELECT”, “SONG_SELECT”, and “PLAYING”. The state machine is controlled directly through the buttons of an NES Gamepad. Each state will process the button inputs differently in order to implement its intended behavior.

The NES Gamepad itself is a simple 8-bit shift register with pull-up resistors. Pushing a button connects the shift register’s input pins to ground, setting the input signal to 0. The shift register also has a latch and pulse input, and a single output. Once the shift register receives a high latch signal, it latches the current state of all 8 buttons. After latching, it will begin to output the button states through its single output pin, shifting one state out each input pulse. The order it outputs the button states is: “A”, “B”, “Select”, “Start”, “Up”, “Down”, “Left”, “Right”. Finally, the controller waits until the next latch signal to begin outputting again. Communication from the FPGA was handled by the controller_poll module, which sends a latch signal at 500Hz, then reads out the button states at 50KHz, storing them sequentially in a buffer. Once all states are in, the module outputs the negated contents of the buffer since the NES Gamepad has active low output.

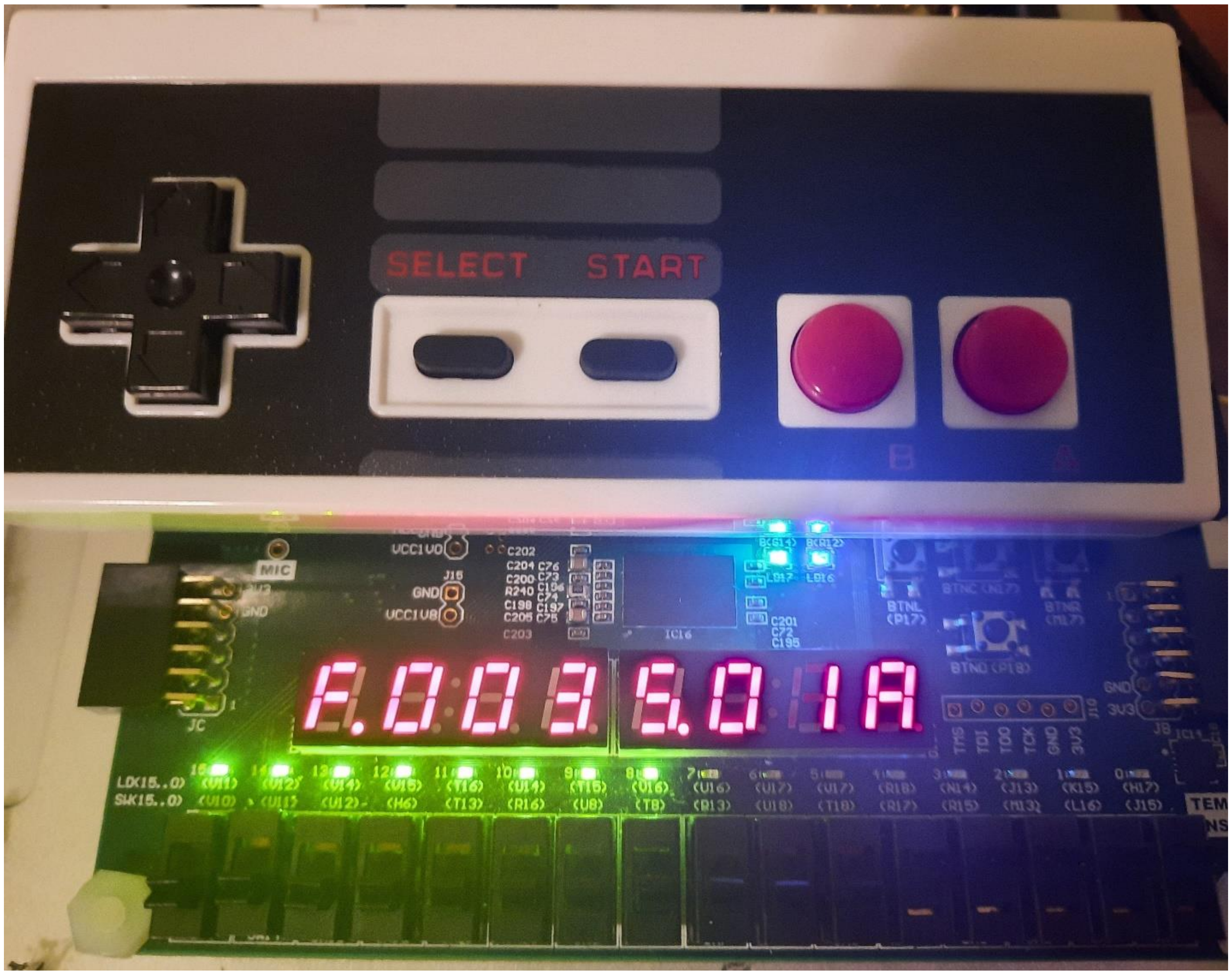
In order for the FPGA to communicate with the NES Gamepad, they were connected via 5 wires soldered to the Gamepad’s connector. The white Vcc wire was connected to 3.3V and the black Ground wire was connected to GND. The latch signal, the brown wire in this case, was connected to jb[0], and the pulse signal, the orange wire in this case, was connected to jb[1]. Finally, the output signal, the yellow wire in this case, was connected to ja[0].

When the file select system is in the “FILE_SELECT” state, it allows users to select what file they wish to load. The Up and Down buttons on the NES Gamepad control the file_index variable, which goes from 0 to the total number of files (minus 1) in the SD card, a value the NSF Player module determines. The Select and B buttons do nothing in this state, while the Start button is used to reset the file_index to 0, and the A button is used to confirm the desired file. Pressing A also moves the system to the next state, the “SONG_SELECT” state, and resets the song_index variable to 0.

In the “SONG_SELECT” state, the Up and Down buttons are used to control the current song_index variable, which goes from 0 to the total number of songs (minus 1) on the loaded NSF file, a value the NSF Player module reads from the loaded file. Here, the Select and B buttons are used to return to the previous state, the “FILE_SELECT” state. The Start button is used to reset the song_index to 0, and the A button is used to confirm the desired song. Pressing A also moves the system to the next state, the “PLAYING” state, and sends the NSF Player module the file_select signal, which will let the system start playback of the desired song.

Finally, when the system is in the “PLAYING” state, the Up, Down, and A buttons do nothing. The Select button will return the system to the “FILE_SELECT” state, while the B button will return the system to the “SONG_SELECT” state. Here, the Start button is used to pause or unpause audio playback without restarting the song.

At any point in time, the Left button is used to decrease the system volume, and the Right button is used to increase the system volume. The current volume, which varies from 1/8 of the max volume to the max volume, is displayed via the FPGA LEDs above the switches. The current file_index is displayed through the 4th digit (from left to right) of the hex display, while the current song_index is displayed through the 7th and 8th digits of the hex display.

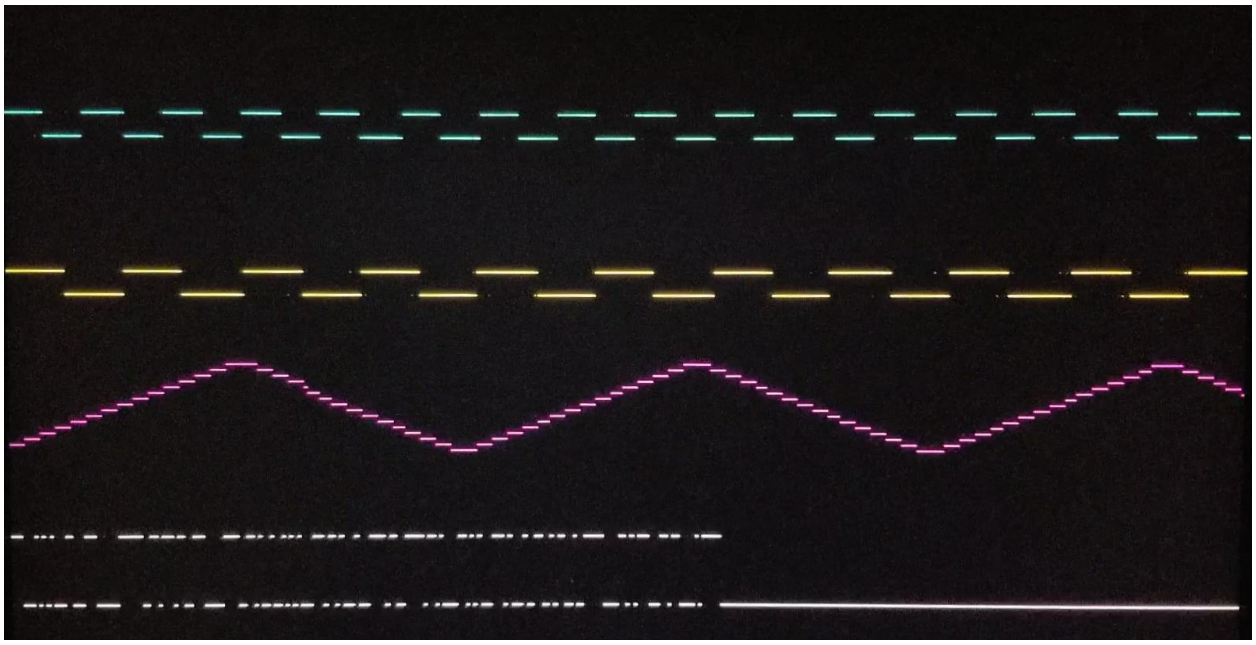


Display

VGA was used to display the waveforms synthesized by the APU as well as the current file select state. The VGA module from Lab 3 was modified to use a 25MHz clock and display at a resolution of 640x480. Originally, the goal was to implement an entire file select GUI over VGA, but the hex display was used instead due to time constraints.

Waveform Visualizer

In order to better demonstrate the output of our APU implementation, a simple waveform visualizer was implemented. The `level_display` module samples and displays four 4-bit waveforms from the APU (two square waves, one triangle wave, and one noise wave) in real time. Display data is output in the form of a 640x480 60Hz VGA signal.



Each wave is sampled using the `level_sample` module. This module waits for a rising edge (a transition from zero to non-zero) on a channel, then collects 320 samples. It collects samples at a rate specified by an external pulse input. In this case, all channels are sampled at 15kHz. This is fast enough that it completes in the time required to display the waveform but slow enough that it captures many peaks and troughs of the waveform rather than a small local area. In order to allow more time for sampling, the module begins sampling on every other frame rather than every frame. This results in a maximum sampling period of 840,000 clock cycles (2 frames at 640x480). Because the system runs at 25MHz, this means the sampling rate must be greater than 9.5kHz in order to capture 320 samples within 2 frames.

The number of samples collected is 320 because the horizontal display resolution is 640, a multiple of 320. Thus, the module may determine which sample to display by simply dividing the current horizontal pixel count by 2 and indexing the sample buffer with the result. The 4-bit samples themselves are multiplied by 4 in order to scale them up to a possible 64 pixels in height, which makes them much more visible on screen.

Because waveforms are displayed in real time, they must be displayed as soon as they are available. However, using the same buffer to both capture and display the samples results in screen tearing because sampling will need to occur for more than a frame's worth of time, and sampling will inevitably alter the values while the waveform is in the process of being displayed. To avoid this, two buffers were used: one to store samples, and one to store the displayed waveform. Sampling may occur while the waveform is being displayed, but the display buffer is only updated once every other frame, outside the visible area of the waveform.

File Select Display

File select system state is displayed on the monitor through the title_blob module. When the state is "FILE_SELECT", "Select a File" is displayed. When the state is "SONG_SELECT", "Select a Song" is displayed. When the state is "PLAYING", " Now Playing " is displayed (including spaces). This module takes in a coordinate (x_in, y_in) that represents the top left corner of the displayed message. The module interfaces with a font ROM loaded with a COE file containing data to display characters indexed by their ASCII representation. For example, inputting $41 into the ROM's address will cause it to output data that the title_blob module uses to display the character "A". The ROM outputs 64 bits, which encodes 8 rows of 8 bits, each of which informs the module whether that relative bit position should be empty or not.

The title_blob module is designed to display a single line of text. The number of characters to display can be modified by changing the parameter NUM_ITEMS, but users must also change the parameter SIZE_ITEMS to allocate enough bits for NUM_ITEMS. Therefore, the relevant space on the screen would be the space where hcount is greater or equal to x_in but less than WIDTH*NUM_ITEMS, and where vcount is greater than or equal to y_in but less than HEIGHT, where HEIGHT and WIDTH are both 8 by default to match the output of the ROM. The relevant positions within this space would be curnt_x = hcount – x_in, and curnt_y = vcount – y_in.

Each text character is encoded into ASCII by 1 byte. This means that the character being examined at any point is the character at index [(NUM_ITEMS – (curnt_x >> 3))* - 1]. In other words, if the curnt_x is from 0 to 7, character 0 is being examined; if curnt_x is from 8 to 15, character 1 is being examined, and so on and so forth. The module then sends the current character into the ROM, and reads the relevant data to determine whether the pixel should be empty or not. As mentioned above, the ROM outputs 64 bits, which must be split into 8 rows of 8 bits. The y index of this array will be dependent on the current relative y position, which is curnt_y. Since only a single line of text is displayed, only 8 bits of height are significant, and thus only the 3 least significant bits of curnt_y will be useful. Meanwhile, the most significant bits of curnt_x were used to determine what the current character is, but the 3 least significant bits determine the x position of the pixel within the character.

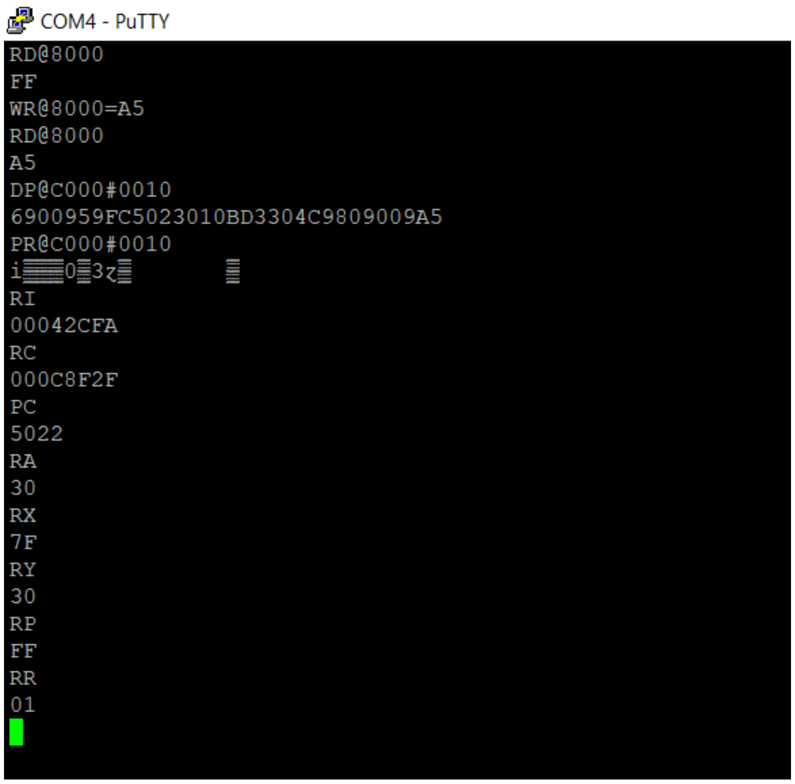
Since each row is separated by 8 bits within the ROM data, the y coordinate will increment every 8 bits, thus each row starts at bits 0, 8, 16, and so on. This means the top 3 bits of the pixel coordinate will be encoded by curnt_y[2:0]. Within each row, the x coordinate will be used to determine what the current pixel is. Thus, the current pixel's coordinate within the 64 bits coming from the ROM will be {curnt_y[2:0], curnt_x[2:0]}. This was attempted, but all characters were displayed upside down, so the correct coordinate is {~curnt_y[2:0], curnt_x[2:0]}.

The title_display module looks at each pixel position using this logic to determine whether said position should be empty or not. The ROM, however, takes 2 cycles to output, therefore some pipelining was utilized to ensure there was enough time to process each pixel. The module was originally designed in a fashion that would allow it to interface with the NSF Player module and RAM to display file names. The NSF Player would write the titles of each file within the SD card into the RAM, which the display module would then read and display when in the "FILE_SELECT" state. Time constraints required that the display module be simplified into its current state.

Debugger

In order to ensure proper functionality of the CPU and NSF player modules, a capable debugger was developed. The debugger serves no purpose in the final audio pipeline of our system but was crucial during the entire development process. The debugger is a main memory device, able to read from or write to any location in memory. It shares a separate 32-bit data bus with the CPU to allow reading or writing of the CPU's `A`/`X`/`Y`/`PC`/`SP`/`SR` registers, instruction counter, and cycle counter. It also may stall or reset the CPU. It interfaces with a serial terminal via bi-directional UART at 38,400bps.

There are 25 debugger commands, each consisting of 2 characters. For example, the "read A" command (RA) returns the value of the CPU's A register. The command is executed as soon as a valid sequence of 2 characters is sent through the terminal. In the case of an invalid command, the debugger simply prints "INVALID". Every character typed is sent back through the terminal in order to provide the user feedback of what commands have been sent. All command parameters and all printed results are in hexadecimal, although the print command (PR) prints each byte from memory directly as a single char rather converting it to a 2-char hexadecimal representation. Most commands either write a single value to a single register or read a value from a single register or memory location. However, the load (LD), dump (DP), and print (PR) commands take a length argument of up to 65,535 bytes, allowing the user to write or read that many bytes to or from memory at once. This is especially useful for reading whole blocks of memory, in which case the memory contents are returned as a continuous string of bytes.



```
RD@8000
FF
WR@8000=A5
RD@8000
A5
DP@C000#0010
6900959FC5023010BD3304C9809009A5
PR@C000#0010
i▒▒▒0▒3z▒        ▒
RI
00042CFA
RC
000C8F2F
PC
5022
RA
30
RX
7F
RY
30
RP
FF
RR
01
```

The commands shown here are: RD, WR, DP, PR, RI, RC, PC, RA, RX, RY, RP, and RR. All of them except WR return a result on the next line.

Most debugger states consist of waiting for user input. This is done by waiting for the UART receive module to send a `valid` pulse, which indicates a new byte is available. Each byte received via UART is passed through a decoder that converts the ASCII char to the hexadecimal number it represents (‘0’ through ‘9’ are 0-9, while ‘A’-‘F’ and ‘a’-‘f’ are 10-15). This number conversion is used anytime the debugger expects a number parameter, such as an address, length, or data value. Similarly, each byte produced as a command result is converted to its 2-char ASCII representation. The largest results (CPU counter values) may be 4 bytes, so the hex-to-ASCII conversion outputs 8 ASCII chars. The actual number of chars displayed (2, 4, or 8) depends on the command.

Most debugger states print something when user input occurs. Printing is achieved through dedicated `PRINT` and `WAIT_PRINT` states. A 128-bit shift buffer may be loaded with up to 16 chars before entering the `PRINT` state. The upper-most char of this buffer is printed, the buffer is left-shifted to remove the sent char, and the system waits for the UART transmit module to finish transmission. After this, the process repeats if there are any non-zero values left in the buffer. After printing, the system returns to `return_state`, which should be defined before first entering the `PRINT` state.

Aside from accessing registers and memory, the debugger provides several CPU control commands. It may stall (ST), run/unstall (RN), single step (SS), or set a breakpoint (BK). The single step command steps through one CPU instruction by unstalling the CPU, waiting for the CPU to complete the instruction, then stalling again. The breakpoint command works similarly, but takes an address argument. It unstalls the CPU, waits for the CPU PC to reach the specified address, then stalls it. This results in the CPU being stalled before the instruction at the specified address is executed.

Possible Improvements

Jackson

The CPU likely still needs improvements because there are select songs that will not play. This is most likely due to some instruction behaving improperly only in edge cases and could be resolved with more debugging time. Many improvements were made during development by stepping through assembly routines on both our debugger and the FCEUX debugger and comparing outputs. With enough time, this process could be repeated for all songs that fail to play and eventually reveal an issue.

In addition, the CPU could be expanded to support unofficial instructions. This is not necessary for audio playback since no NSF files are known to use unofficial instructions, but there are some NES games and other executables that make use of them. If unofficial instructions were implemented, the CPU would be functionally complete compared to the original 6502.

The NSF player could be expanded to read the names of files from NSF headers as well. This is a fairly trivial addition but was not included since there is no GUI to display names.

The waveform visualizer always displays the first sample on the left edge of the screen, so waveforms appear anchored there. It would be more effective to display the first sample in the center of the screen, which would require sampling both before and after the trigger point.

Victor

The most glaring issue is that the DMC generates high frequency sound as it plays samples. This can significantly impact the output sound quality, and can be fairly painful if the user tries to listen to the music at high volume. Based on ILA observations, the cause of this problem seems to lie somewhere in the DMC’s Output Unit. If this issue can be resolved, the output sound will be a nearly perfect recreation of the sound generated by the original NES hardware.

Originally, the VGA display was intended to allow users to see all files within the SD card. Each file has data about its file name, which would be read by the NSF Player, saved into RAM, and displayed on the monitor during file selection. The display would then act as a GUI, highlighting the current user selection, thus allowing users to select a particular file. Once a file was selected, the monitor would display all songs within that file. The NSF Player already outputs the total number of songs within the current file, thus the monitor would simply need to display the correct number of choices. Again, the user would be able to select a particular song, which would then initiate audio playback. At this point, the monitor would display the waveforms as it currently does. These functions were not implemented due to lack of time, and thus could be improved in the future.