NaES: Not aN Entertainment System

Pradyun Narkadamilli & Pranav Nair
GitHub

Introduction

For our final project, our goal was to design and build an SoC in SystemVerilog capable of replicating a subset of features of the NES. The NES (Nintendo Entertainment System) is an 8-bit video game console created by Nintendo in 1983. The standard NES CPU core is based on a 6502 processor, with modifications that made it consumer ready for such an SoC - the biggest addition being an APU (audio processing unit). Using an open source IP of this core, we hoped to write SystemVerilog that could model and synthesize some of the behaviors of what was a historically remarkable console. In particular, we planned to make a system capable of running NES ROMs, accepting input via a keyboard, and (ideally) being able to scroll.

NES/Famicom Game System

Much of the NES design has been well-documented by enthusiasts - after rigorous testing by enthusiasts, there are detailed explanations of its internal architecture, down to the internal bugs. However, making a one-to-one copy of such a system was infeasible

in such a small amount of time, especially while in other technical coursework. Thus, our project centers on replicating minimal behavior to emulate *launch* NES titles, the most famous of these being Donkey Kong and Super Mario Bros.

Furthermore, to comply with the hardware limitations and capabilities of the DE-10 lite, our logic adapts the NES's rendering to a VGA timing/output specification rather than directly generating composite signals and making intermediary logic. Though this will inevitably cause some issues with getting games to look perfect, it is one of the compromises we needed to make.

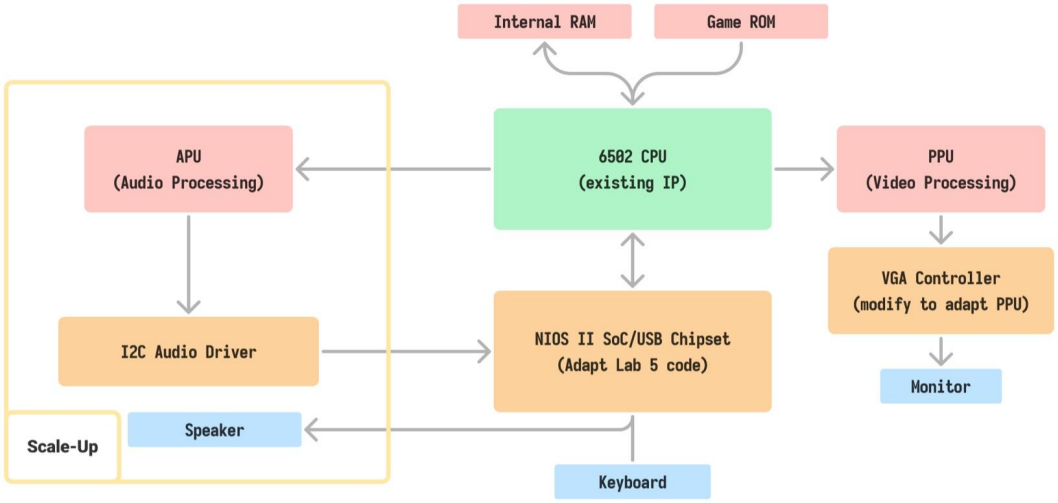
Written Description

Interaction

Note that this project does not simply emulate a *game,* but emulates the hardware used in an NES. As such, there is, by default, no code loaded onto our SoC. The user must find an appropriate game ROM to run - however to streamline this process, we have provided a python script (nes-hex.py) to convert games from the standardized iNES format to CHR-ROM and PRG-ROM blocks - we'll elaborate what these are going forward - stored in an MIF format that can be flashed to the DE-10 Lite's onboard M9K RAM blocks. Anyway, once one generates these initialization files and flashes the FPGA accordingly, they can interact with NaES using a standard keyboard, which is internally converted to the NES controller interface, and view the game on a VGA output at 512x480 resolution. The default bindings are *WASD* for directional keys, *TY* for *Select/Start*, and *GH* for *AB*. The current build (at the time of writing) supports 3-key rollover for this player, though the release on GitHub should support a second player using the right half of the keyboard, and 8-key rollover for both players combined. When flashed, the FPGA should automatically start the game of choice. Key0 has been internally routed as a reset signal for the system, should it be necessary - though note that this signal only resets program state and registers, and not the internal RAM. This may cause some unexpected behavior, but these can be avoided by re-flashing the FPGA to achieve start-up state rather than using this "hot reload".

Internal Structure

As mentioned in the introduction, this system, even at the diluted level that we produced, involves a high level of complexity, with many different behaviors that build off of each other. Thus, we chose to approach this project with an approach similar to those LEGO Bionicle models - start from a simple core, and attach more and more pieces branching outwards. In this case, our core was the CPU that we found online (we'll talk about this soon). We also abstracted away and modularized our system as much as possible, dedicated one module for each isolated behavior. As such, this description will be structured according to that abstraction. In our initial proposal, we created a block diagram depicting what we expected the completed SOC to look like, though as we'll see in the report going forward, some of these hierarchies have been reworked or elaborated as necessary.



Simplified Block Diagram

CPU

The original NES operated off of a modified 8-bit 6502 processor core (representative of computing standards at the time), but due to the scope/time constraints of this assignment, it did not make sense for us to reimplement such a CPU - such an assignment could in fact constitute its own final project.

As such, we've opted to use a readily available Verilog IP that was produced by OpenCores, and later modified/debugged by the MisTer NES Project. However, the current iteration of this project implements some extra behaviors in the CPU that make our simplified implementation more complex, as such we've opted to use an older version which is archived here. It supports all of the basic behaviors we require.

For the sake of our project we can reduce the CPU interface to a couple of key ports (outside of configuration):

- `Enable` - used to pause the CPU for batch data transfers
- `NMI` - used to initiate an non-maskable interrupt
- `W/R` - active-low Write
- `ADDR` - Bus I/O Address
- `DATA_IN` - Bus read data
- `DATA_OUT` - Bus write data (driver)

As this documentation progresses, some of these signals - especially the non-bus signals - will be used to enable and control key behaviors by the rest of the SoC, especially the PPU (picture processing unit).

CPU Data Interfacing and Directly Interfaced Storage Elements

The CPU, as in most SoCs, interacts with the rest of the system through a primary data bus via a 16-bit addressing system and 8-bit in/out communication - the connections to which have been depicted below.

For sake of simplicity, our primary data bus was done using purely combinational logic. We connected a unified bus_out output to every element's data input, and (due to a lack of tri-state buffers), used a series of if-else statements to synthesize a priority mux to determine the data input driving the bus. For write operations by the CPU, the driver was always the CPU's `DATA_OUT`, but for read operations, we needed some kind of abstraction for element addressing.

The bus abstracts CPU memory/element accesses using the following memory map:

x0000-x07FF	SYSRAM
	(Addresses mirrored until x1FFF)
x2000 - x2007	PPU Interface
	(mirrored until x3FFF)
x4014	DMA Trigger
x4016/x4017	Controllers
x6000-x7FFF	Save RAM
x8000-xFFFF	PRG ROM

Most of these elements - at least in comparison to the rest of the system - require little architectural complexity, and should be quickly reviewed before proceeding. However, it should be noted that the main exceptions to this are the Save RAM and PPU. The PPU is an extremely complex component, which will have its own section later. Again due to the time constraints, the Save RAM was not implemented in this simplified console.

Though it should be noted that the games we used as benchmarks to test our progress - namely Donkey Kong and Super Mario Bros. - do not make use of this.

Program ROM

An iNES ROM can be decomposed into its PRG ROM and CHR ROM components - PRG ROM is where all the CPU instructions for the game are stored. It is sized to be 32KB (two 16KB banks) of M9K read-only SRAM.

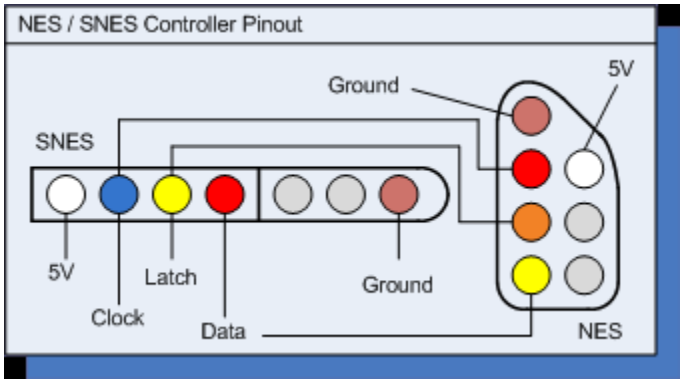
It should be noted that games can potentially use less or more memory than this. Since this was one of the first components we ironed out in the system, we were not sure if we'd be able to afford potentially "burning" valuable M9K space on extra memory, and as such have ignored games that take advantage of memory-mappers to increase the number of memory banks that can be mapped to the 16KB regions in PRG ROM. In the case of a game using only *one* 16KB bank, we simply clone it across the 32KB span.

System RAM

This is the main operating memory of the SoC. It is sized to 2 KB's worth of single-port M9K SRAM, and has read/write access from the CPU.

Controller(s)

In the original NES, the controller would connect via a pinout that looks like the following:



And latched data would be shifted out over a span of 8 reads (one per button). We required some way to emulate this behavior using a viable source of input - though the onboard switches/buttons could be used, it'd be very impractical to play games.

Instead, though a bit overkill in the grand scheme of things, we instantiated a modified version of the NIOS IIe SoC created in Lab 6, and used it to read keycodes from the MAX3241E chip via an SPI interface. It isn't exactly practical to play games with only one keystroke being read at a time, so we modified the driver code and PIOs on the system to accept 3 keycodes at a time, and output them to 3 different 8-bit PIO outputs on the platform bus.

These outputs were then passed into a submodule dedicated to controller parsing. At any given time, this submodule's combinational logic generated a potential 8-bit output sequence by validating and positioning each of the keycodes in the output sequence. When the bus indicates a write to this submodule, this composite is latched into an 8-bit shift register. For each of the next 8 reads from the bus, the most-significant bit is shifted into a read buffer that drives the bus output - this is done on the positive edge of `control1_en&&READ` , where control1_en indicates the bus "selecting" the controller for R/W operation.

Despite being rather simple to implement, the controller gave us a bit of trouble during design, primarily due to the strict timing requirements of the NES system, and an obvious lack of people piping NIOS II PIOs into their virtual NES bus. There was various experimentation done with directly wiring the shift-out to the bus, different update conditions, etc.

Clock Domains

As is common in complex SoCs, there are multiple clock domains in use. The original NES used 4 clock domains - a 1.7MHz CPU clock, a 5.369MHz PPU Clock, a 12MHz RAM clock, and 21MHz master clock used for output timing.

To simplify the design, and make it compatible with VGA, we created modified forms of the graphics logic and got rid of the PPU Clock altogether, meaning that our system also has 4 clock domains - a 50MHz system clock (primarily used to drive the NIOS SoC), a 1.7MHz CPU clock, a 12MHz clock used for RAM blocks and controller interfacing, but a 25.172 MHz graphics clock used for VGA. The latter 3 clocks are generated by a second PLL contained in the NIOS SoC, though it should be noted that this was done for no architectural reason - rather, Platform Designer automatically generates timing constraints for PLLs. When manually instantiating the PLL, we ran into various timing issues whenever a signal crossed from one clock domain to another, where at one point we were struggling to diagnose our system's (-5) slack for an unspecified logic block.

PPU

The original NES used a Ricoh 2C02 for its PPU. It was capable of rendering the NES's output video as 8px square tiles at either the NTSC 256x240 (224 visible) resolution at 60 FPS or the less common PAL standard. It cannot be overstated how difficult making this portion of the project was - while we primarily encountered timing issues with the previous sections (crossing clock domains, sampling issues, etc.), this part of the project combined our prior timing concerns with what is inherently a difficult task to tackle logically.

Though it would be possible to decompose this component in a number of ways, for the sake of this document, we will cover it in 5 broad categories:

1. Memory
2. I/O
    a. VRAM Bus
    b. DMA (Direct Memory Access)

3. Background Fetching
    a. Scrolling
4. Sprite Fetching
5. Rendering/Priority Outputs

This order is arbitrary - when developing the PPU, we regularly flitted between these sections as necessary to incrementally prototype and test the system.

Memory

For any given pixel, the PPU needs to determine what color it should be based on data from the game cartridge and the CPU's instructions. It is far too inefficient to individually address each and every pixel for every frame, so instead the NES compartmentalizes its data into sets of sprites and background tiles, establishing strict rules for determining the pattern and colors of each. Furthermore, rather than generating RGB values and encoding those to composite, the NES directly generates color in the composite format from 64 potential colors.

The most fundamental piece of this data is the palette register block, a 32-byte area of VRAM, is used for storing up to 8 palettes - 4 for different background tiles, and 4 for sprite tiles. Each byte stores a different color that can be rendered with the NTSC color set. Unless the grayscale mode is configured, all software running on the NES reads what color to output for a given pixel value from the palette.

Note that the palettes, which are normally implemented with RAM in an NES unit, are instead implemented using registers in our system for the sake of prototyping convenience and simplicity.

The astute reader can quickly discern that to index the palette and extract a color, one requires a 5-bit address. The other memories exist to supply the necessary data to

assemble this address. Internally, palettes are accessed with the following memory map (with respect to the rest of VRAM).

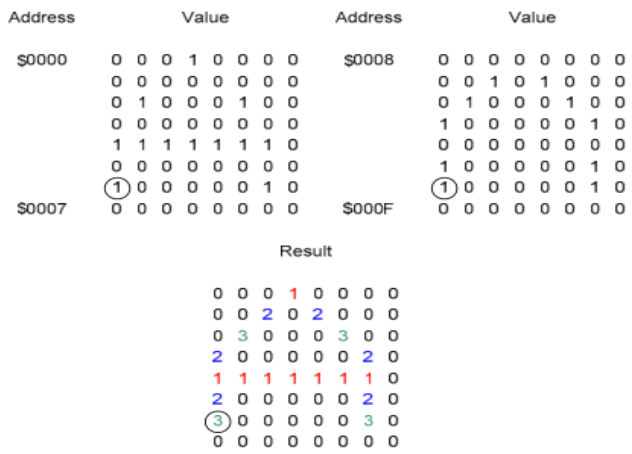
Address	Purpose
$3F00	Universal background color
$3F01-$3F03	Background palette 0
$3F05-$3F07	Background palette 1
$3F09-$3F0B	Background palette 2
$3F0D-$3F0F	Background palette 3
$3F11-$3F13	Sprite palette 0
$3F15-$3F17	Sprite palette 1
$3F19-$3F1B	Sprite palette 2
$3F1D-$3F1F	Sprite palette 3

(source)

Note that for read/write access, the addresses 10, 14, 18, 1C are mirrored to 00, 04, etc. Generally, the MSB of the palette address indicates whether one is indexing a sprite or background palette, and the other four bits are supplied by other components in VRAM.

Pattern Tables

The pattern tables, character RAM/ROM, or CHR RAM/ROM for short, is an 8KB section of VRAM (abstracted as two 4KB sections) where all the tile data for the game is stored. While most games use an 8KB ROM stored in the cartridge itself, some games/ROMs will write to a 2KB block of internal NES VRAM during runtime, and mirror its addressing across all 8 KB's worth of VRAM space. However, due to the relative infrequency of this approach in NES titles, we've foregone implementing the secondary behavior.



Pattern Table Tile Encoding

Each 8 by 8 tile occupies 16 bytes in memory. Treating each set of 8 bytes as a separate 8x8 matrix, concatenating any given bit in the second set of 8 bytes with the corresponding bit in the first set will yield a 2-bit number, representing the color of that specific pixel in the tile. Using this encoding, the pattern table is able to use 16 bytes to encode a tile with up to 4 different colors - a good example being the multicolor "A" above. This 2-bit sequence is used as the two LSBs to index the palette registers. The other 3 bits are encoded in the other two memory sections.

It should be noted that a value of "0" does not represent an opaque color, but rather transparency. This becomes pertinent when render priority between sprites and background is discussed later.

Sidebar: For execution purposes, up to 8KB of CHROM is extracted from the relevant NES ROM and initialized at startup.

Nametables

The nametables are two 1KB sections of VRAM used to store information about what pattern should be displayed by each tile of the background. Each name table is partitioned into two sections - x3C0 bytes are used to describe the tile number in the pattern tables corresponding to each of the 32x30 8px tiles on the screen. Byte 0

describes the upper leftmost tile of the screen, from which the rest of the bytes are laid out in row-major order.

The remaining memory is referred to as an **attribute table.** This section of memory stores the color palette information for every 4x4 set of tiles. If we look at these sets as a scaled version of the nametables' grid, the attribute table retains row-major order for such an arrangement. Each byte can individually address the palette used for a 2 by 2 set of tiles. According to the NESDev wiki, that data is encoded as follows:

```
7654 3210
|||| ||++- Color bits 3-2 for top left quadrant of this byte
|||| ++--- Color bits 3-2 for top right quadrant of this byte
||++------ Color bits 3-2 for bottom left quadrant of this byte
++-------- Color bits 3-2 for bottom right quadrant of this byte
```

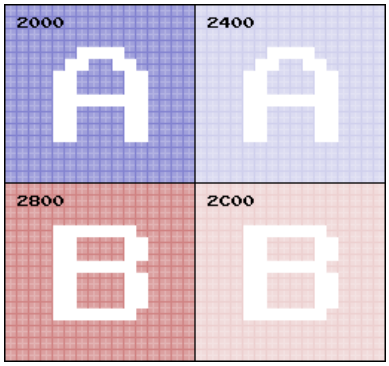
Though it may sound redundant to have two separate nametables, many games that involve "scrolling" across a level stitch together two screens and partially render each one - this is done by copying 2 screens' worth of data to the nametables, then selectively choosing a start/end position in each one to simulate a continuously moving screen. This behavior will be better discussed in the rendering section of this document.

But to summarize: For any given tile in the background, the *name table* stores what pattern should be rendered, and the *attribute table* stores what background palette we should use - this two-bit number comprises the middle two bits of our palette address.

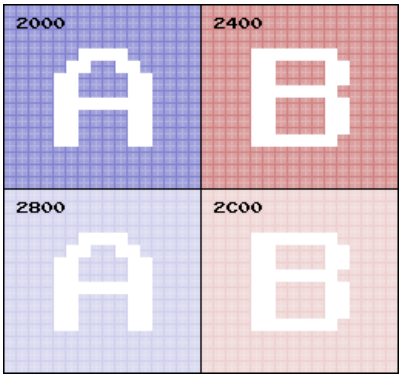
Thus for any background tile, the palette address can be assembled as `{1'b0, (palette no.), (pattern data)}`.

Nametable Mirroring

The two 1KB nametables are mirrored across 4 logical tables - though seemingly trivial, this mirroring is the basis for how scrolling works on the NES. There are two common mirroring schemes used by games (without attaching additional CHR ROM in the cartridge):



Horizontal Mirroring



Vertical Mirroring

In I/O, there is a register that configures the “base nametable” - its two-bit address refers to the logical nametables in this grid.

Sprite/Object Attribute Memory

Whereas background tiles can be stored as an array in row-major order, sprites are constantly moving, require different render orientations, and more independently accessible fine-grain control. Thus the NES has a 256-byte section of memory referred to as the *OAM*. This block stores 4 bytes’ worth of data for up to 64 different sprites. Data is encoded as follows for each sprite:

Byte 1	Topmost Y position
Byte 2	Tile number (in pattern table)
Byte 3	Attributes: render priority, horizontal flip, vertical flip, palette
Byte 4	Leftmost X position

The byte 3 palette is used to index palette registers in a similar fashion to how it’s done with nametables, but the MSB changes to 1’b1 in order to access sprite palettes.

While the NES technically supports 8x16 sprites, to simplify the prototyping of this project we have opted to only support 8x8 sprites. However, this is more than enough for most earlier NES games - Donkey Kong, Super Mario Bros. Ice Climber, Pacman,

and more games that we supported at the time of demoing this project all only require 8x8 sprites.

That should offer a “brief” overview of where pattern and render data is stored in the PPU. The coming PPU sections all in some way or another depend on interaction with these memories, thus it is best to retain a working knowledge of what they are, and how data is encoded.

At an architectural level, all of the RAM-based memories are implemented using dual-port M9K RAM blocks. While the original NES used single-port RAM, with our current experience it seemed unreasonable to design arbitration logic between commands originating from the VGA rendering and CPU accesses. Dual-port RAM allowed for a relatively elegant way to discretize these accesses while retaining memory parity.

I/O

As alluded to in the CPU memory map, the PPU is interfaced via 9 different registers (including DMA trigger). Each register has complex internal behaviors that, if fully documented, would stretch this report to the length of the NESDev wiki. For understanding purposes, we will briefly describe some registers at a high level, then separately discuss the VRAM bus and DMA.

It should be noted that internally, these registers are not actually being accessed by the CPU - rather we have a separate `bus_out` register that is used to buffer requested read data, and unified address/data ports (as opposed to creating separate entries in the primary bus for each element). Write data is directly passed to the relevant register/RAM block. Through experimentation, we found that this approach produced fewer behavioral and timing issues.

- *x2000: Control Register: Write-Only Access*
  Enables/configures various internal behaviors including NMI suppression, background pattern table address, sprite pattern table address, and base name table for background rendering.
- *x2001: Mask Register: Write-Only Access*
  Controls various render settings - enabling background/sprite rendering, whether to render background/sprites in the first 8 pixels (first tile), and to render opaque backgrounds on top of sprites or vice versa.
- *x2002: Status Register: Read-Only Access*
  Generally, the game running will use this register to determine if the PPU is in a VBlank period and if Sprite0 Hit has occurred (this event is better documented in the sprite rendering section). There are some other buggy behaviors implemented in the original NES, but for the games we are running they are irrelevant.

  When this register is read from, it clears the VBlank flag.
- *x2003: OAM Address: Write Access*
  The address stored in this register by the CPU is used to index OAM.
- *x2004: OAM Data: Read/Write Access*
  A read/write to this register performs a read/write to the memory address stored in *OAM Address*. In our system, this register actually doesn't exist - but if a read/write is requested from the corresponding address, we pipe input data to the OAM block and buffer any relevant output data into `bus_out`. On a read/write to this register, the OAM Address register is incremented.

  Note that very few games interface via this register - OAM generally needs to be written to in large quantities during vblank, and this approach would take too long (4 cycles per write in a contiguous block). Instead, DMA is regularly used.
- *x2005: Scroll Position: Write x2*

This register is *16 bits* internally - on our 8-bit system, it requires two writes to fully populate. This register specifies the X/Y offset when scrolling the background, where X/Y are defined from 0-255 or 0-239 respectively. Scrolling is a topic we will cover in more detail later in this document.

All the registers mentioned above are clocked at the CPU clock, while one port of the OAM RAM is clocked at the 12MHz RAM clock.

VRAM Bus

Due to the large amount of memory stored in the PPU (even excluding OAM), directly addressing it with the CPU is infeasible (when you consider the size of the rest of the CPU's memory map already). Instead, the PPU has its own internal VRAM bus that can be interfaced via registers x2006 (16-bit address, like x2003) and x2007 (read/write access like x2004). Though it should be noted that, like with the OAM connection, there is no literal 2007 register - rather, a series of comparators and muxes are used to generate appropriate write-enable signals and an input to the output buffer (should the 2007 register be read). All RAM ports on this bus are clocked at 12MHz, and are arranged in a memory map as follows:

Address range	Description
$0000-$0FFF	Pattern table 0
$1000-$1FFF	Pattern table 1
$2000-$23FF	Nametable 0
$2400-$27FF	Nametable 1
$2800-$2BFF	Nametable 2
$2C00-$2FFF	Nametable 3
$3000-$3EFF	Mirrors of $2000-$2EFF
$3F00-$3F1F	Palette RAM indexes

$3F20-$3FFF	Mirrors of $3F00-$3F1F

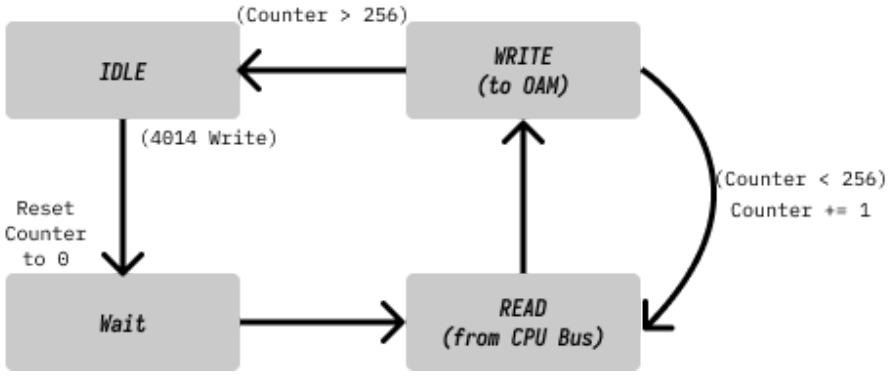
(source)

DMA

In every NES game, any kind of quickly/frequently moving game objects are always modeled as sprites - generally these will be user objects and COM/enemies. While the OAM registers detailed above offer one way to write this data between frames, it's inefficient - most of the time, games are trying to copy large batches of data at a time. This is where DMA, or Direct Memory Access, comes into play.

Generally, the CPU will trigger a DMA operation by writing an 8-bit address (call it (D)) to address x4014 on the primary data bus. When this write completes, the PPU will pause the CPU and hijack the primary data bus. It then, over the course of 512 cycles, copies 256 bytes from x(D)00 - x(D)FF on the CPU memory map to the OAM.

Internally, we made a separate module for DMA that hijacks both the CPU bus and the VRAM bus. When DMA is triggered, this module outputs a "hijack" signal to both buses, and pulls the CPU enable low (pausing it). The batch copy behavior is modeled as a simple FSM (as shown below) tied to a 16-bit counter - this felt like the leanest implementation at the time of writing. At any given time, the OAM address is simply the counter value, and the CPU bus address is x(D) concatenated with the 16-bit counter value. The OAM input is directly wired to the CPU bus's output - thus the amount of logic actually performed outside of cycle management is minimal.



Overall, while not especially complex compared to the rest of our system, DMA ultimately proves to be crucial, as it is the primary way that games populate sprite RAM between frames.

Background Fetching

So far, we've talked about how data is prepared and encoded such that the system can figure out what color each pixel should be - essentially CPU-facing logic and abstraction. This section onwards discusses the VGA-facing logic, and how that data is rendered.

The first problem we tackled was determining pertinent data to render the current background tile. Based on the memory components we knew the NES contained, our approach could be decomposed into 4 main steps:

1. Index name tables
2. Index attribute tables
3. Take name table output, index *first* set of pattern data
4. Take name table output, index *second* set of pattern data

Since we were dealing with RAM for all of these pieces of information, there would need to be some kind of wait state before each read operation to allow the output to propagate. Thus in total, we would need a minimum of 8 clock cycles to latch all the necessary data to render a tile. On the original NES, this worked out very nicely during visible scanlines - the data for the next tile could be fetched while rendering the current one.

In our system, we needed some way of tracking state to know which operation to perform at any given time, and what row/column to fetch data for. Rather than manually tracking state with an FSM and counters (as the NES did), we used the DrawX and DrawY counters of the supplied VGA controller to keep track of the current pixel position, and used the DrawX value as a form of horizontal state. This abstracted away any state logic on our end, and reduced the number of failure points.

Our VGA controller renders at double the NES resolution - as such, we still fetched all the necessary data in the first 8 pixels of each tile, but only *latched* that data to the render registers at the end of the 16th pixel of every block.

Since we decided not to use an incrementing counter, we needed some other way of keeping track of what indices to plug into the name tables. Considering we were using the same VGA controller, we chose to modify our Lab 7 prefetch logic slightly - by dividing the DrawX value by the tile width (16 in this case) and retaining our current Y position, we could simply increment the tile coordinates to find "newDrawX" and "newDrawY". These incremented coordinates would allow us to fetch the next 8 bits' worth of data per the NES resolution - as long as we double-rendered each pixel, it would look fine on VGA. The main complication this produced was that each row had to essentially be rendered twice. Our workaround was dividing NewDrawY by two to know what scanline we were on from the NES's perspective. This logic allowed us to convert from a 512x480 pixel map to a 32x240 virtual grid that was more in line with the NES's memory architecture. Simply concatenating these coordinates {Y,X} was sufficient to produce an address for the nametable. Since the attribute table was a version of the name table but with 4x4 tiles, reducing the address to {Y/4, X/4} was sufficient to produce the second address.

From there, indexing the pattern tables was significantly easier - given the tile number, we 3 least significant bits of the divided NewDrawY coordinate to figure out which of the lines in the tile we were rendering. Given the pattern table scheme, making reads to {(TILE NO.), 1'b0, (NewDrawY[2:0])} and {(TILE NO.), 1'b0, (NewDrawY[2:0])} was sufficient. At this point, we have all the data loaded to determine background colors for the next 8 NES pixels (16 VGA pixels) - whenever DrawY hits a 16th pixel, the pattern data and attribute data is latched.

Scrolling

Most games require different screens and background data as the user moves from top-to-bottom or left-to-right. To be able to render different X/Y positions without having to overwrite full nametables just to move a single block forward, the NES renders frames by using partial data from each nametable. In short, stitching together a frame from a larger overall screen. This effect is well-illustrated by this animation that we found on the NESDev wiki.

The X/Y scroll offset are provided by the scroll register we alluded to in the I/O section - the first 8 bits act as an X pixel offset, and the second 8 bits act as Y pixel offset. By treating these as an offset into a larger "mega-frame" produced by the logical nametable arrangement mentioned in the memory section, we can then index nametables and attribute tables as necessary to fetch a contiguous 256x240p background using that offset as the top-left-most point of the frame.

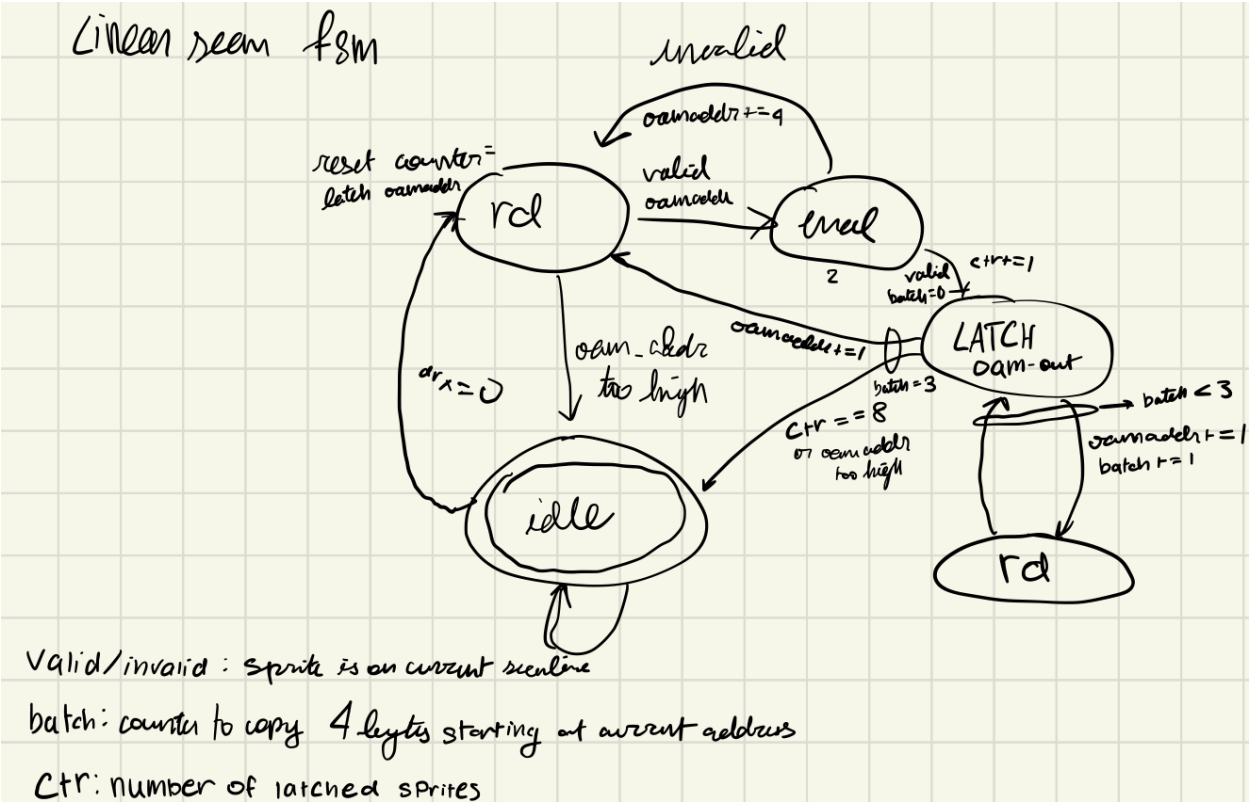
While scrolling effects can be produced by editing nametable data between frames, this produces a jittery output that would update the edge block by block - imagine if instead of smoothly moving to the right, Super Mario Bros. had a new block pop in at random frames? Thus the NES has to not only be able index data with an offset, it also has to update the on-screen blocks on 8-pixel boundaries of the *translated* pixel coordinates rather than the render coordinates. Vertical scrolling is rather simple, as you simply add the necessary vertical offset to the Y position at the beginning of each scanline, and it's consistent - the horizontal bounds of nametable fetching remain the same. However the horizontal offset can change these boundaries, and at the time of writing we haven't implemented a stable version of this logic. For now, we simply adapted our non-scrolling logic (documented above) and changed our comb block's RAM fetching triggers to be based on a translated X coordinate rather than the render X coordinate. This works for most of the screen, but the second column of the frame tends to be a bit buggy.

Sprite Fetching

Unlike nametables, where the appropriate memory address to index can be derived from the render/scroll coordinates alone, there is no direct mapping between screen position and what sprite to render - rather it's the inverse. Thus, to render sprites, we need to fetch the necessary data on the *previous* scanline, then assemble the next scanline's sprite attributes/data on the current scanline. To do so, we need to do two things: make a linear scan of the OAM to determine valid sprites, and fetch the corresponding data out of nametables.

Sprite fetching takes too long to do during the horizontal blanking period alone, and the pattern table interface is consumed by the background rendering logic. Thus we segment it into two steps: we assemble a list of up to 8 sprites on the next scanline from OAM during the visible rendering period, then once the pattern table bus has been freed in HBlank, we fetch the corresponding lines of the sprite itself.

Cutting ahead a bit, pattern fetching once the sprite data is aggregated isn't inherently difficult - we fetch data over a period of 32 VGA clock cycles in VBlank, using 4 cycles per sprite. The first two cycles latch the X position/attributes of the fetched sprite memory and the MSBs of the pattern data to sprite output unit registers, and the second two cycles latch the LSBs of the pattern data. This is a similar approach to the "pixel-based state" approach we took with the background fetching. However, fetching data from sprite OAM was not something we were able to simplify in a similar manner. Thus instead, we created an FSM to perform a linear scan on the OAM. This FSM is neatly modeled in the diagram below. While fetching data from the OAM, we cannot directly write to the sprite data cache - the NES would be using it to well... render sprites. Instead, we have *another* set of cached sprite data, called "secondary OAM", declared as a 32-byte register file. When the FSM "latches" data, it does so in the secondary OAM.



We trigger this FSM at the first pixel of the visible region, beginning our search at the current position of OAMAddr (latching this as the initialization value of our scan address). The FSM continues scanning the region of sprites until *either* we have filled our secondary OAM completely, or the scan OAM address is too high to successfully fetch a contiguous 4-byte block from OAM. The reasoning for this should be implicit from the section above about data encoding in sprite OAM.

At the end of the visible region of the visible scanlines, the temp sprite counter (number of sprites fetched), X coordinate of each sprite, and Attribute byte of each sprite are all copied to appropriate locations in the secondary OAM. The Y coordinates and tile numbers (first two bytes of each latched 4 byte block) are used in a comb block to produce the appropriate addresses for pattern table fetching. Pattern table fetching is done via the pixel-state method mentioned earlier. Through this process, the sprite output unit memory (another 32-byte register block) is populated with data.

Rendering/Priority Outputs

Suppose that by this point, all the necessary background and sprite data has been fetched. Even with all the pattern data and palette data fetched, what color should the PPU render at each position? Should it be from the background pattern, or the PPU pattern? What if there isn't a valid pixel at that position? This logic is all handled by a set of priority muxes.

The first priority mux exists to determine sprite priority - it determines the sprite to use and outputs appropriate color and attribute data. Generally, sprites with a lower index in the OAM should be given priority over other sprites. Since we aggregate secondary OAM/sprite output unit memory with a linear scan, it is already in priority order - thus simply finding the first valid sprite in our output unit memory (OUM?) is sufficient. Considering that for every sprite in our OUM, there is a byte declaring its X position, a naive implementation of this logic would be to use a set of comparators and give the first sprite with an X position (SpriteX) where the render pixel's X position (non-translated) is between SpriteX and SpriteX+7 (inclusive). However, this fails to take into account the fact that the NES treated pixels in the pattern data with a value of 0 as transparent. Thus, the priority condition is adjusted to reflect the first pixel in range with a non-zero pixel value. The comb block synthesizing this priority logic outputs the relevant sprite's attribute data, and two bits from the pattern data (essentially a two-bit number) describing the color of the current pixel in the sprite. Before being passed into the next mux, pattern data and attribute data is used to index the palette registers, and find the final VGA-compatible (12-bit) color that would be outputted should the sprite be rendered.

From this point, the PPU must determine render priority between the sprite data and background data. Assuming that the background obeys the same 0-pattern transparency rule and that there could potentially be *no* valid sprites, there are 4 cases to consider. If only *one* of the potential outputs (BG vs Sprite) has a nonzero pattern value for the current pixel, then that one should be selected by default. If *neither* has an opaque

pattern, then the universal background color, stored in palette register 0, is rendered. If *both* are opaque, then there is a bit in the sprite's attribute byte describing whether to give it priority over the background or not - this is used as the priority condition.

Disclaimer

There are quite a few intricacies in this document ignored, such as mask conditions, the sprite 0 hit flag, etc. However if we were to fully document all of these things, this document would easily cross 50-60 pages, even with this diluted version of the NES. This goes to show how truly sophisticated of an SoC the NES was. For sake of (relative) concision, many of these intricacies were omitted. For those curious in the full extent of our design process, a list of resources have been provided at the end of this document - it encompasses every document we used to design our system.

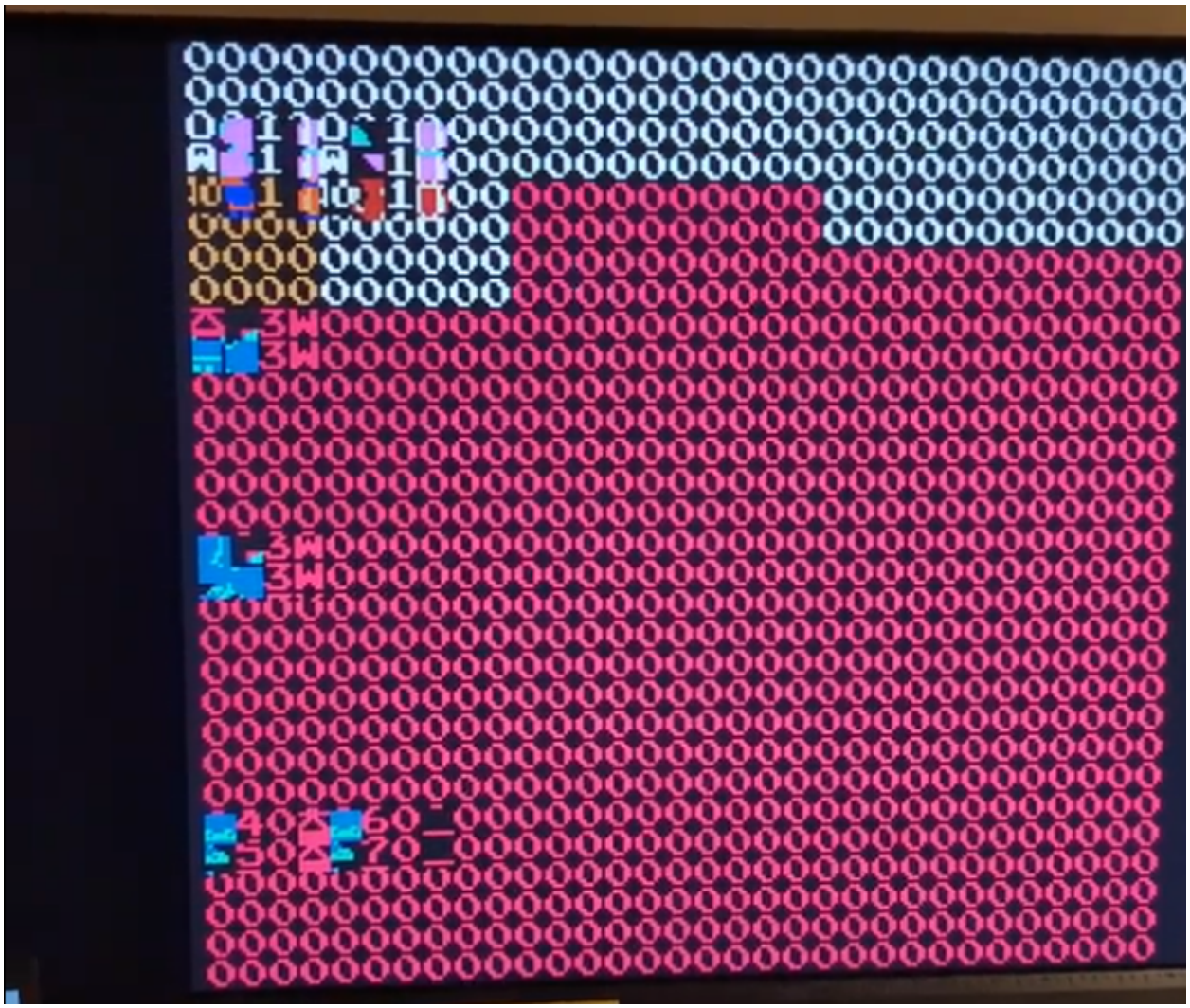
Software Description

Our software component for this project consisted mainly of modifying Lab 6 code for additional PIO blocks. The implemented functions of `Maxreg_wr`, `Maxreg_read`, `Maxbytes_wr`, and `Maxbytes_read` remain in use for SPI protocols. However, the `main.c` file has been updated to support 3-key rollover from keyboard input. This requires the use of three separate `setKeycode` function calls, each addressing a separate PIO. To scale up in the future and support an ideal 16-key rollover, we would, adhering to the logic thus far, create another 10 PIOs and have 10 more function calls in main.c.

Debugging and Testbenching

Generally, the edge case for being able to test a design with simulations and testbenches is that it is either primarily graphical (VGA output) or involves a great deal of RAM logic. In our case, both were true - until the point that our background rendering was working, we primarily debugged using the onboard hex displays to show important information like the PC and OAM address. Once the background rendering worked, however, we would pass our debug data to the background renderer as a pattern table index - as shown below, this allowed us to graphically see how certain conditions were

changing throughout a single frame. When debugging our system, a majority of bugs were purely due to timing issues or mismatches with the timing requirements of the original NES (and what the games we ran were expecting). It was more efficient for us to procedurally test these timing compatibilities by *running* a game and monitoring internal state, rather than creating our own analogous test in assembly - both from a time-cost and stimulus standpoint.



Displaying sprite data on each scanline via the background renderer

Design Resources and Statistics

LUT	**DSP**	**MEM**	**FLIP-FLOP**
7314	0	373,888	3591
Frequency	**Static Power**	**Dynamic Power**	**Total Power**
151.42 MHz	102.42 mW	0.71 mW	112.45 mW

Conclusion

Our expectation going into this project was to create a minimal NES SoC capable of playing Donkey Kong by the time of our demo, with the potential to expand functionality in the future - we believe we've more than exceeded that goal. At the time of writing, our NaES is capable of playing Ice Climber, Pacman, and Donkey Kong without any issues - we're sure there are other games that can be played successfully as well, we just haven't tested too many yet. Super Mario Bros. is able to play, albeit with a slight rendering bug.

Due to proximity to the deadline and other finals, the scrolling implementation we made was a bit buggy. We've noticed that when games attempt to scroll horizontally (e.g Mario), the second block of each scanline is a copy of the first column. We were not able to debug this prior to demo, but do have some ideas on how to improve the scrolling logic to eliminate this issue. While there are some issues in other games - noticeable examples including flickering of the status bar in Super Mario Bros. and *significant* screen tearing in Kung Fu - we have been able to attribute these issues to the fact that our logic is based on a *VGA* clock at a higher resolution, rather than the intended NTSC clock. A possible solution for this would be to directly render frames in composite, then create a discrete module to adapt a composite frame buffer to VGA. The immediate concern with doing this is screen tearing - however if we were to make this change, our NES would likely become more cycle-accurate, allowing for smoother emulation of

many games. Luckily, most of our PPU render logic is designed to be compatible with the original NES frame timings, and as such would require minimal modification.

Though not exactly a bug, we noticed during our demo with Prof. Cheng that he was having trouble making fast movements when playing Super Mario Bros. due to rollover limitations. As mentioned throughout this document, rollover is something that we would like to extend as soon as possible - though now it seems to be more critical to user experience than we initially thought.

Outside of just debugging issues with the NES, we believe that there are many ways for us to viably extend the behavior of NaES over the summer. A goal that could be quickly achieved might be to support ROM memory mappers (MMCs). In actual NES cartridges, these mappers allowed games to internally switch between different memory banks, allowing for larger and more complex games. Considering that memory availability tends to be the main setback with implementing such games and that our M9K usage is only at 28%, we could easily instantiate more ROM banks in the top-level module to support mapped games.

Another improvement would be implementing the audio processor (APU) - many games in the NES era were characterized by their music, and we think this would greatly improve user experience. However the APU requires more memory priority handling and bus hijacking for the delta modulation channels and sample loading - this would increase the complexity of our bus design by a non-trivial amount. So while this would be a long-term goal, we can't see ourselves reliably completing this subsystem soon.

Overall, we found this project to be greatly educational regarding digital design and working with FPGAs. While we didn't work with C code or Platform Designer as much as some of the other projects we saw, there was much more work put into reliably crossing clock domains, bus architecture, FSM design, etc. - skills that are all essential to pursue a successful career in hardware design. Furthermore, we *enjoyed* our project - it

was exhilarating to see Donkey Kong pop up on the screen for the first time in black-and white, to see Mario run around (even when decapitated and inverted), and to finally see people enjoying our system at the project showcase. As an experience, even as infuriating as it was to be “flying blind” while debugging, we think that creating NaES will be an essential experience in our future careers and pursuits within the field of digital hardware design.

References/Resources/Acknowledgements

95% of the reason we were able to complete this project on such a tight timeline, especially with other difficult courses, was because of some of the great documentation and resources we found online by enthusiasts who were passionate enough to thoroughly document their findings and process creating emulators. Though note that many of these resources are written from the perspective of a programmer trying to create a software emulator, rather than a hardware clone.

NESDev Wiki

A great resource for specific behavioral analysis of the NES’s behavior. It describes everything in a great deal of granularity. Various images and tables throughout this document were found in this wiki.

https://www.nesdev.org/wiki/

Nintendo Entertainment System Documentation v1.0 - Patrick Diskin

The NESDev wiki can be quite expansive for someone just starting to grasp the internals of an NES. This document acts as a comprehensive “getting started” guide to understanding the NES’s internal structure. It was our starting point as well.

https://archive.org/details/NESDoc

JavidX9 NES Emulator Series

Very detailed and well-explained series about making an emulator in C++. We were able to relate some of his logic - especially that regarding render priority - to how we might

design equivalent digital logic.
https://www.youtube.com/channel/UC-yuWVUplUJZvieEligKBkA

We'd be remiss without specially thanking Ian Dailis. As someone who made a similar project when he was in our position, he went above and beyond his commitments as a CA to guide us. We're beyond grateful for his commitment and guidance.