SNES Emulator

Suryansh Singh and Safwan Morshed

Introduction

This design explores emulating the Super Nintendo Entertainment System (SNES) hardware onto the Real Digital Urbana Board FPGA. The SNES is Nintendo's 1990 flagship console that boosted popular titles such as Super Mario World, Mega Man X, A Link to the Past, Donkey Kong Country 2, and much more. This lab aims to replicate the hardware on the FPGA allowing any of the SNES ROMs to be played via the board. This lab specifically focused on integrating the CPU and APU units with a custom built PPU architecture equipped with the SNES' full rendering capabilities barring graphics mode 7. Additionally, controller support was also integrated, however the BRAM space on the FPGA to support vitis was too limited given the other memories that needed to be stored, so controller support was offloaded to a Raspberry PI4 which communicated to the FPGA via GPIO.

Design Overview

The SNES' major hardware can be divided into 5 major categories (each with their respective submodule). As a general overview, the CPU handles the addressing for instructions and the memory handles the communication between the CPU and other modules through memory mapping IO as well as decoding CPU instructions to write to all the different types of memories. The memory also handles reading the ROM off an SD card and loading its contents into DDR3 on the Urbana Board. The APU acts as its own processor, which handles all the audio processing and output for the SNES, communicating with the CU via DMA (Direct Memory Access). The Controllers, as mentioned, are offloaded to the raspberry pi and communicate with the FPGA via GPIO, and are parsed in the console using hardware registers in the CPU. Lastly the PPU is responsible for all the picture processing, graphics, and rendering of the SNES games.

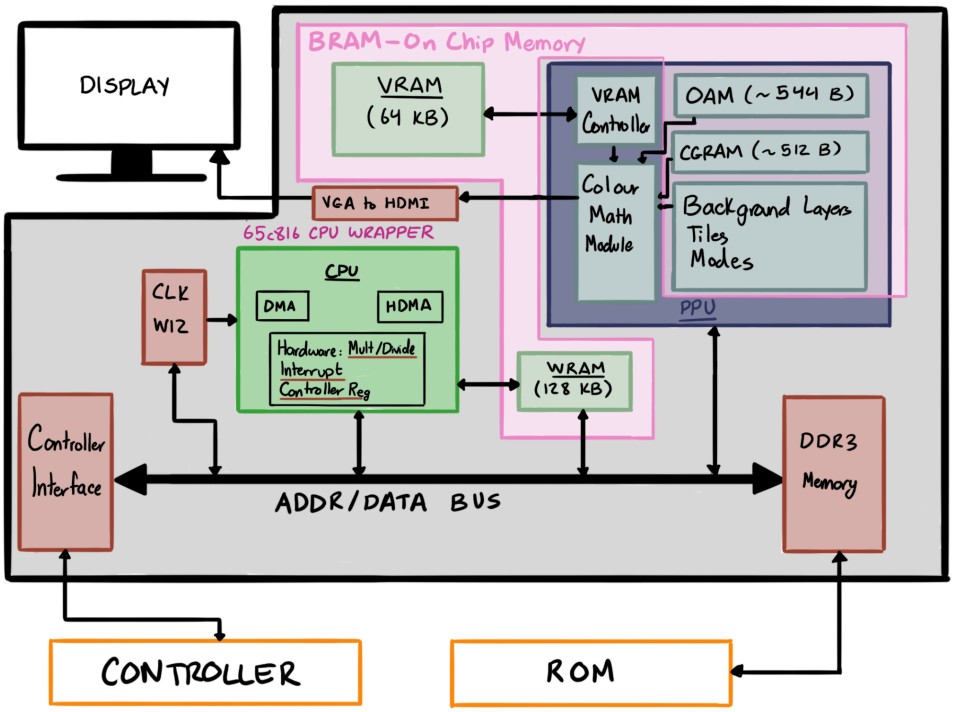

Figure 1: High Level SNES Overview

CPU

The CPU subsystem specifically is responsible for executing SNES game programs and generating the memory that the PPU, APU, DMA, and the controller access during execution. Rather than implementing the 65C816 from scratch, the project instead uses the CPU core from SNEStang as the base processor implementation and wrote a custom wrapper around that to expose the signals and pulses our SNES required. SNEStang was used because it provides an already working hardware implementation of the SNES CPU that is suitable for FPGA-based emulation, allowing the project to focus more heavily on system integration, memory mapping, video rendering, and controller support.

In this design, the SNEStang CPU core is wrapped by a logical CPU interface module so it can communicate cleanly with the rest of the Urbana board SNES system. The wrapper connects the CPU's 24-bit address bus, data bus, read/write control signals, interrupt inputs and clock enable timing to the project's memory map. CPU accesses are routed through the memory controller, which determines whether each access targets ROM, WRAM, PPU registers, APU registers, DMA registers, controller registers, or other mapped hardware. This allows the CPU to behave as if it is interacting with the original SNES memory map, while the FPGA implementation redirects each transaction to the appropriate hardware block.

The CPU does not directly control rendering or audio generation. Instead, it updates memory-mapped registers and RAM regions that are consumed by the PPU, APU, DMA, and controller subsystems. For example, writes to PPU registers configure background modes, scroll values, tilemap locations, VRAM addressing, CGRAM colors, OAM sprite data, windowing registers, and color math settings. The CPU also interacts with DMA and HDMA registers, allowing large blocks of data or per-scanline graphics updates to be transferred without requiring normal CPU stores for every byte. These interactions are essential for games such as Super Mario World, which rely heavily on timed PPU register updates during gameplay and transitions.

The CPU wrapper also exposes debug information, including the current program counter and stall state, which was especially useful during system debugging. When games failed to boot or freeze, the program counter displayed on the hex displays helped determine whether the issue was a CPU execution problem, a memory-mapping issue, or a downstream PPU/DMA timing issue. Test ROMs were used to confirm that the CPU core and basic bus behavior were functioning before debugging more complex rendering and timing problems.

Overall, the CPU subsystem served as the central control unit of the emulator. The use of the SNEStang CPU core provided a known working processor foundation, while the project-specific wrapper and memory map integrated it into the Urbana FPGA design.

Memory + DDR3

The memory subsystem connects the CPU's 24-bit address bus to the different hardware regions expected by SNES software. In the official SNES, the CPU does not access one huge block of memory, instead, different address ranges correspond to different forms of memory, including cartridge ROM, WRAM, PPU registers, APU communication registers, DMA registers, HDMA registers, controller I/O, and cartridge specific memory. To support real SNES ROMs, our FPGA design required a memory map that decoded each CPU access and routed it to the correct system. This allowed for ROM loading via an SD card without having to modify any instructions, as the memory map handles all the parsing in accordance with the FPGA design.

The CPU core issues read and write sequences using a 24-bit address, data bus, and control signals. The memory mapping logic examines each address and determines which of the above areas it should go into. Specifically, DMA/HDMA, controller, and PPU registers are implemented using the LUTs on the FPGA, whereas designated blocks of memory such as WRAM (128 KB), VRAM (64 KB), CGRAM and OAM (~1 KB total), and APU memory (64 KB), were all implemented as BRAM on the FPGA. BRAM usage had to remain mindful however, as the FPGA only supported 337 KB of total BRAM. Decoding the addresses in the memory map allowed the rest of the system to appear to the CPU like normal SNES memory instead of a divided system.

A major part of the memory map is cartridge ROM access. In the Super Nintendo architecture, HiROM, LoROM, and ExHiROM are memory mapping schemes that define how the console's 24-bit address bus routes requests to the physical ROM chip. Because the SNES reserves the lower half of many memory banks for Work RAM (WRAM) and hardware registers, standard LoROM games map their data into 32KB chunks placed in the upper halves of these banks. HiROM, on the other hand, maps data in continuous 64KB chunks within the higher address banks, making it easier for developers to manage large graphical assets without bank-crossing limitations. ExHiROM extends the HiROM concept to support massive games (over 4MB) by leveraging the highest address bit (A23) to access an extended 8MB address space.

In our FPGA design, this complex address translation is handled dynamically by the cart_header_detect.sv and cart_mapper.sv modules. Before the CPU boots, the header detector scans specific byte offsets in the raw ROM image to automatically deduce whether the game is LoROM, HiROM, or ExHiROM based on where the internal title and interrupt vectors are located. Once the CPU begins executing, cart_mapper.sv intercepts the CPU's 24-bit memory requests and uses combinational logic to calculate the exact physical offset within the ROM file. For example, it translates LoROM requests by stripping out specific address bits and applying a bitwise mask based on the detected ROM size, ensuring the physical address safely wraps within the bounds of the loaded file and preventing open-bus crashes.

Because the Urbana board's Spartan-7 FPGA lacks sufficient internal Block RAM (BRAM) to hold commercial-sized SNES games, our design relies on the board's external DDR3 memory and SD card slot. During the initial boot sequence, an SD card controller reads the raw .sfc or .smc binary file straight off the micro-SD card and writes it sequentially into the DDR3 chip. This process is orchestrated by the snes_ddr3_wrapper.sv module, which connects to the Xilinx Memory Interface Generator (MIG).

Throughout this initialization phase, the SD card is granted exclusive write access to the DDR3 memory while the SNES CPU is safely held in a reset state.

Once the entire ROM is loaded, the system asserts an initialization-complete flag, and control of the DDR3 is handed over to the ddr3_rom_reader.sv module. As the SNES CPU runs and asks for data, mem_map.sv passes the translated physical ROM address to the DDR3 reader. Because the DDR3 runs at a high frequency (often 83.33 MHz or 166.5 MHz) and the SNES CPU runs much slower (~3.58 MHz), the reader stalls the CPU using its ready signal, issues a command to the MIG to fetch a 128-bit burst of data, and caches it. It then extracts the specific 8-bit byte the CPU requested and releases the stall, seamlessly bridging the gap between modern, high-latency DDR3 memory and the cycle-accurate needs of the Ricoh 5A22 CPU.

DMA and HDMA registers are also decoded through the memory map. Normal DMA allows the CPU to configure bulk transfers from memory into PPU or APU registers, while HDMA allows per-scanline updates that are commonly used for effects such as scrolling, gradients, and window changes. The memory mapping logic provides the interface between the CPU-visible DMA registers and the DMA controller, allowing DMA activity to temporarily take over bus transactions and perform register writes on behalf of the CPU. When DMA transfers require data from ROM or RAM, the memory system must coordinate those reads with the correct backing storage, including DDR3-backed ROM data when necessary.

The design also includes address decoding for WRAM and APU communication. WRAM accesses are routed to the system's working RAM storage, while APU register accesses allow the CPU to communicate with the audio subsystem. Controller-related registers are also mapped so that game software can read player input using the same addresses it would use on SNES hardware.

Overall, the memory subsystem acts as the central routing layer of the emulator. It allows the SNEStang CPU core to interact with the rest of the custom FPGA design using the same memory layout expected by SNES software, while hiding the implementation details of ROM storage, RAM, PPU registers, DMA, APU communication, controller input, and external DDR3-backed storage.

APU

To implement audio support, we made use of SNESTang's implementation of the SPC-700 chip as it has its own independent ISA, clock cycles, and pipeline compared to the rest of the SNES which is clocked to the execution of the CPU. We had to build custom wrapper logic that could safely bridge the SNESTang SMP and DSP cores with our existing system architecture. One of the primary challenges was clock domain crossing: our main SNES CPU and memory mapper run on a high-speed clock (ui_clk) tied to the DDR3 controller, while the SNESTang APU expects to run at 25 MHz. In our apu_top.sv module, we implemented a custom lossless toggle-and-handshake synchronization mechanism for the $2140-$2143 CPU-to-APU mailbox registers. This guarantees that when our main CPU fires off audio commands, they safely cross over into the APU's slower clock domain without any consecutive writes being silently dropped. Additionally, we replaced the generic audio memory with a 64 KiB Vivado true dual-port Block

RAM (BRAM) IP, explicitly configured to handle the simultaneous read/write demands of the SMP processor.

Once the SNESTang DSP successfully generated the audio, we needed a way to physically output it through the Urbana board's 3.5mm audio jack. Because the board's audio hardware is designed to low-pass filter a single-bit digital audio stream, we couldn't just send it raw digital values. To solve this, we created the apu_pwm.sv module, which acts as a first-order sigma-delta (PDM) converter. The DSP outputs two's-complement signed 16-bit PCM samples, so our module first applies an offset-binary conversion (flipping the sign bit) to ensure absolute silence translates to a 50% pulse density. It then accumulates these values to generate high-frequency, 1-bit Pulse Width Modulation (PWM) signals for both the left and right channels, allowing the board's physical circuitry to smoothly reconstruct the SNES audio.

Controller

Controller support was initially implemented through the use of the Lab 6.2 Microblaze design. It adapted the MAX3421E host driver to support generic USB gamepads like the cheap SNES controllers we acquired through Amazon. However, we found that the rest of our design necessitates all of the available BRAM on the Urbana Board to implement the various memory modules and buffers for the SNES. Thus, we opted to use a Raspberry Pi 4's USB ports and its GPIO headers connected to the PMOD ports of the Urbana Board to implement controller support without having to use a Microblaze core which would waste BRAM and LUTs we could've otherwise used for the rest of the design.

The final controller architecture revolves around the Pi sending packets of controller input reports to the Urbana Board and the board using combinational logic to set the values of hardware registers that correspond to Joypad input for player 1 and player 2. Data was structured as a 16-bit report packet consisting of a 12-bit button report where each set bit represents a specific button pressed on an SNES controller, a 3-bit player report which dictates if player 1 or player 2 (or 3-5 if we had chosen to implement Multitap support), and a strobe bit to indicate if the data being processed by the board should be considered valid. The Pi makes use of its built-in Linux gamepad drivers to process direct gamepad events and then using a Python script, translate these events into this packet structure.

PPU

The PPU (Picture Processing Unit) was the most complex part of the SNES emulator design because it is responsible for turning the game's graphics data, register state, background configuration, sprite data, palettes, and timing behaviour into the final video output. In the original SNES, the CPU does not directly draw pixels on the screen. Instead, the CPU writes to the PPU registers, VRAM, CGRAM, OAM, and the PPY uses those state values along with a pixel clock to generate each scanline of video. Our FPGA PPU recreated this behavior by dividing the rendering process into several stages, register control, timing generation, background and sprite pixel production, deferred priority composition, colour lookup/colour math, and final VGA/HDMI scanout.

Starting with the PPU register module, this stores all 50 memory-mapped video registers written by the CPU and DMA/HDMA system. These registers include display enables, brightness, background modes,

tilemap addresses, character data addresses, scrolling offsets, sprite configuration, mosaic settings, windowing controls, screen enable flags, CGRAM colour writes, colour math settings, and mode 7 settings. These register values are consumed by the renderer and compositor to determine which graphic layers are active, where the tile data should be fetched from, how each background is scrolled, which layers appear on the main vs subscreen, and how final colour should be blended together. The register interface was especially important because many SNES games rely on DMA and HDMA to update the PPY state mid frame during the horizontal blanking periods.

The timing module then generates the emulated SNES PPU scanline timeline. It produces the current dot (x coordinate), scanline (y coordinate), visible region flags (because the SNES has blanking regions to the right and below the screen to allow for HDMA transfers and new frame writes to happen), their respective hblank and vblank statuses, and the line start pulses. These signals are run on the PPU clock and are exported both to the PPU rendering logic and to the memory/DMA system, since HDMA must occur during hblank periods and CPU status reads depend on the blanking states. One important architectural component of the PPU is that the final composition plan runs on PPU-timing. This is because things like HDMA writes are very timing sensitive and must be read properly to produce the correct effects. Thus, all rendering takes place at PPU-time and the final RGB555 values are stored in a buffer which is read at VGA timing (the real digital imported IP that allows RGB values to be displayed to the monitor).

One of the largest timing issues faced was the fact that BRAM (which holds VRAM and OAM) has a one clock cycle read latency. VRAM specifically stores the addresses and information for all the background tiles as well as the object information, so for each pixel, 5 VRAM reads must be made, which results in a large timing delay. Thus, the pipeline was split into 3 major steps:

The first step was a fetch ahead, which fetched the tile addresses ahead of the current scanline being drawn (i.e. while line N was being drawn, data for like N+8 was being fetched)

The second step was to add windowing effects during scanline time in a registered pipeline (as the windowing combinational logic was too long to fit into one clock cycle) and get the final rendered RGB values for the main and subscreen.

The third step was then to pass those two values into the colour math module, which used the colour math settings to add / subtract the two screens together into one final RGB value, which was then passed into the final RGB buffer that was read from the VGA controller in VGA time.

Delving into more detail:

The background and sprite renderer produces candidate pixels for each visible SNES scanline. Background rendering uses the current background mode, tilemap base addresses, character base addresses, scroll registers, and VRAM data to determine the candidate color index and priority for each BG layer. Sprite rendering uses OAM data and sprite configuration registers to generate OBJ candidate pixels. Instead of immediately converting these candidates into final RGB values, the renderer writes a deferred pixel packet into a 51-bit line buffer. Each packet contains the candidate information for BG1,

BG2, BG3, BG4, and OBJ at a specific pixel, including valid bits, priority bits, and CGRAM palette indices.

A key design decision was to separate pixel production from final composition. Earlier versions of the PPU attempted to pre-render lines ahead of scanout and perform some final decisions later. This worked for simple rendering, but it broke down when windowing and HDMA effects were introduced. Window registers can change from scanline to scanline, so final masking and color math must be evaluated using the correct PPU-time register state for the scanline being displayed. To solve this, the renderer continues to produce deferred candidate pixels ahead of time, but the final priority selection and color processing are performed later on the PPU-timed path.

The deferred compositor takes the 51-bit candidate packet and determines the winning main-screen and subscreen pixels. It compares the valid and priority information from BG and OBJ candidates, applies the main/subscreen layer enable registers, and produces a compact main/sub pixel packet for the color stage. This packet includes the selected main pixel source, main CGRAM index, selected sub pixel source, sub CGRAM index, and a color-window metadata bit. The compositor was also the natural place to add layer windowing, since window masks affect whether individual BG or OBJ layers are visible on the main or subscreen. To avoid long timing paths, window-masked candidate results are registered before the final priority winner selection.

The color stage converts the selected CGRAM indices into final RGB output. It reads the main pixel color from CGRAM and, when needed, reads the subscreen color from a mirrored CGRAM read port. It then applies brightness and color math settings such as addition, subtraction, half-color math, fixed-color blending, and source-layer math enables. Color math is especially important because many SNES games use the subscreen and fixed color to create transparency, fades, lighting effects, and windowed transitions. The PPU color stage also uses color-window metadata to determine where color math is allowed or prevented and where the main screen should be clipped.

Because the FPGA did not have enough block RAM for a full RGB frame buffer, the final video path uses a smaller line-buffered architecture. The renderer first writes 51-bit candidate pixels into a multi-line buffer. The PPU-timed compositor and color stage then convert those candidates into final RGB555 pixels, which are stored in a 16-line RGB buffer implemented in distributed RAM. The VGA/HDMI scanout path is intentionally simple: it only reads final RGB pixels from this buffer and scales the SNES 256×224 image to a 512×448 region inside the VGA frame. This keeps SNES-specific effects on the PPU-timed producer side.

An important issue encountered during development was synchronization between the PPU-timed producer and the VGA-timed consumer. If the VGA scanout read a line before the PPU compositor had finished producing it, the display would flash black or show stale data. To fix this, the RGB buffer uses valid bits and line tags, and the PPU timing is phase-aligned with the VGA frame. Once the producer/consumer relationship was stabilized, the PPU could generate final RGB lines reliably without requiring a full frame buffer.

Overall, the PPU design evolved from a simpler pre-rendered scanout system into a more accurate deferred rendering pipeline. The final architecture separates the design into clear stages: register control,

PPU timing, candidate pixel generation, PPU-timed deferred composition, color/math processing, RGB line buffering, and VGA scanout. This structure allowed the emulator to support normal gameplay rendering while providing a path toward more advanced SNES features such as windowing, color math, HDMA effects, and eventually Mode 7.

FPGA Design

IMPORTANT NOTE - Many of these modules have 20+ inputs and outputs each, writing all of them would make this portion of the report exceptionally long. Exact inputs can be found in the code files, the input and outputs here are general overviews of the information in each module

CPU MODULES

Module: SCPU

Inputs: clk, reset_n, snes_ce, nmi_n, irq_n, ready_in, [7:0] cpu_rdata
Outputs: cpu_valid, cpu_we, cpu_vpa, cpu_vda, [23:0] cpu_addr, [7:0] cpu_wdata, cpu_rdy_out, cpu_brk

Description:
A wrapper module around the SNEStang P65C816 CPU core. It adapts the CPU core's native signals into the project's system bus interface by exposing a 24-bit CPU address bus, 8-bit write data bus, read data input, write-enable signal, valid address indicators, interrupt inputs, and ready/stall handshake signals. The wrapper also adds the snes_ce clock-enable input so the CPU advances at the intended SNES CPU rate rather than every FPGA fabric clock.

Purpose:
Provides a clean integration layer between the imported SNEStang 65C816-compatible CPU core and the rest of the emulator's memory map, allowing the CPU to interact with ROM, WRAM, PPU registers, APU registers, DMA, and controller logic through a unified bus interface.

Module: P65C816

Inputs: CLK, RST_N, CE, RDY_IN, NMI_N, IRQ_N, ABORT_N, [7:0] D_IN, [7:0] DBG_REG, [7:0] DBG_DAT_IN, DBG_DAT_WR
Outputs: [7:0] D_OUT, [23:0] A_OUT, WE_N, RDY_OUT, VPA, VDA, MLB, VPB, BRK_OUT, [7:0] DBG_DAT_OUT

Description:
The main 65C816-compatible CPU implementation imported from SNEStang. It contains the processor registers, instruction register, processor status flags, program counter handling, interrupt logic, microcode sequencing, ALU interface, address generation, and memory bus control. It executes SNES CPU instructions over multiple microcode-controlled states and produces the 24-bit address bus, data bus, read/write control, and valid program/data address signals required by the rest of the system.

Purpose:
Acts as the central processor for the SNES emulator. It fetches and executes game code, performs memory reads and writes, responds to interrupts, configures PPU/APU/DMA registers, and drives overall game execution. In this project, the core was reused from SNEStang and integrated into the custom Urbana FPGA system through the SCPU wrapper.

Module: mcode

Inputs: CLK, RST_N, EN, [7:0] IR, [3:0] STATE
Outputs: MCode_r M

Description:
A microcode ROM/control module for the P65C816 CPU. It stores a table of microinstructions indexed by the current instruction register and CPU state. Each microinstruction contains control fields for address generation, ALU operation, register loading, program counter updates, stack operations, bus control, byte selection, and valid-address behavior. The CPU uses this output to sequence each 65C816 instruction across one or more internal cycles.

Purpose:
Provides the control sequencing required for instruction execution. Instead of hardcoding all instruction behavior directly in random logic, the CPU uses this microcode table to determine what each cycle of each instruction should do.

Module: ALU

Inputs: [15:0] L, [15:0] R, ALUCtrl_r CTRL, w16, BCD, CI, VI, SI
Outputs: CO, VO, SO, ZO, [15:0] RES, [15:0] IntR

Description:
The arithmetic logic unit for the 65C816 CPU core. It performs logical operations, shifts, rotates, increments, decrements, additions, subtractions, comparisons, and BCD arithmetic depending on the control signals supplied by the microcode. It supports both 8-bit and 16-bit operation modes, matching the SNES CPU's variable accumulator and index register sizes. It also generates processor status outputs such as carry, overflow, sign, and zero flags.

Purpose:
Executes the arithmetic and logical operations required by CPU instructions. It is a core part of instruction execution, allowing the CPU to update registers, calculate results, compare values, and maintain processor flags.

Module: AddrGen

Inputs: CLK, RST_N, EN, [2:0] LOAD_PC, PCDec, GotInterrupt, [7:0] ADDR_CTRL, [1:0] IND_CTRL, [7:0] D_IN, [15:0] X, [15:0] Y, [15:0] D, [15:0] S, [15:0] T, [7:0] DR, [7:0] DBR, e6502
Outputs: [15:0] PC, [16:0] AA, [7:0] AB, [15:0] DX, AALCarry, JumpNoOfl

Description:
The CPU address generation module. It calculates program counter updates, effective addresses, direct-page addresses, stack-relative addresses, indexed addresses, indirect addresses, and bank values used during instruction execution. It uses control fields from the microcode to select how the low address byte, high address byte, address bank, direct-page offset, and program counter should update each cycle.

Purpose:
Generates the memory addresses required by the CPU for instruction fetches, operand reads, operand writes, stack operations, branches, jumps, and interrupt vectors. This module is essential for supporting the many addressing modes of the 65C816 processor.

Module: AddSubBCD

Inputs: [15:0] A, [15:0] B, CI, ADD, BCD, w16
Outputs: [15:0] S, CO, VO

Description:
A 16-bit add/subtract module with optional BCD correction. It is built from four 4-bit BCDAdder blocks chained together. The ADD control selects addition or subtraction, BCD enables decimal-mode correction, and w16 selects whether carry and overflow should be reported for 8-bit or 16-bit CPU operation.

Purpose:
Supports the 65C816 CPU's binary and decimal arithmetic instructions. It allows the ALU to perform both normal binary addition/subtraction and BCD-mode arithmetic while respecting the CPU's 8-bit and 16-bit modes.

Module: BCDAdder

Inputs: [3:0] A, [3:0] B, CI, ADD, BCD
Outputs: [3:0] S, CO, VO

Description:
A 4-bit arithmetic block that performs addition or subtraction on one nibble and optionally applies BCD correction. It first performs binary addition using adder4, then applies the appropriate correction value when decimal mode is enabled. It also outputs carry and overflow information for the nibble.

Purpose:
Provides the building block for decimal-mode arithmetic in the CPU ALU. Four instances are chained together in AddSubBCD to support 16-bit BCD addition and subtraction.

Module: adder4

Inputs: [3:0] A, [3:0] B, CI
Outputs: [3:0] S, CO

Description:
A simple 4-bit ripple-carry adder built from four bit_adder full-adder instances. Each bit's carry-out is connected to the next bit's carry-in, producing a 4-bit sum and final carry-out.

Purpose:
Provides the basic binary addition hardware used by the BCD arithmetic modules.

Module: bit_adder

Inputs: A, B, CI
Outputs: S, CO

Description:
A single-bit full adder. It takes two one-bit operands and a carry-in, then produces a one-bit sum and carry-out using combinational logic.

Purpose:
Forms the lowest-level arithmetic building block for the ripple-carry adders used in the CPU's BCD arithmetic path.

MEMORY MODULES

Module: mem_map

Inputs: clk, rst_n, CPU bus signals, DDR3 ROM read data/ready, PPU write readiness, HDMA timing pulses, APU read/write interface signals, cpu_ce, master_ce, debug_sel
Outputs: cpu_rdata, cpu_ready_in, ROM request/address signals, DMA-to-PPU write signals, DMA B-bus read signals, dma_active, APU control signals, CPU interrupt lines, debug_dma

Description:
The main address routing layer for the emulator. It handles the decoding of the 24-bit address space to redirect accesses toward ROM, WRAM, PPU, DMA/HDMA, or APU regions. Beyond translation, it coordinates ROM reads through the DDR3 interface, manages DMA-driven PPU writes, and integrates the system DMA controller with the CPU interrupt logic.

Purpose:
Provides the unified bus interface necessary for the CPU core to interact with the FPGA hardware while maintaining the memory layout required by official SNES software.

Module: snes_dma

Inputs: clk, rst_n, CPU DMA register interface, HDMA init/hblank pulses, ROM read data/ready, WRAM read data, PPU write readiness, B-bus readback data/ready, debug_sel
Outputs: CPU DMA register readback, ROM request/address, WRAM access signals, PPU register write signals, B-bus read requests, dma_active, debug_dma

Description:
A dedicated transfer engine for SNES DMA and HDMA operations. This module tracks configuration across channels to execute bulk memory transfers and per-scanline updates. It orchestrates reads from ROM or WRAM and routes the data to PPU registers via the B-bus, specifically responding to hblank timing for real-time graphics effects.

Purpose:
Enables high-speed data movement and horizontal-blanking synchronization for complex rendering tasks like scrolling, palette gradients, and window transitions.

Module: snes_ddr3_wrapper

Inputs: Differential DDR3 system clock, reset, SD card MISO, ROM read enable/address
Outputs: DDR3 physical interface signals, SD card SPI outputs, ui_clk, ROM read data/ready, RAM initialization status

Description:
The top-level interface for external ROM storage. This component manages the migration of game data from the SD

card into DDR3 memory during the boot sequence. Once initialized, it serves read requests from the memory map by fetching instructions and assets from the DDR3 physical interface.

Purpose:
Overcomes the BRAM limitations of the FPGA by utilizing external memory for large cartridge images while presenting a simplified read interface to the rest of the system.

Module: ddr3_rom_reader

Inputs: clk, reset, MIG read readiness/data signals, ROM read enable, physical ROM byte address
Outputs: MIG read command/address/enables, ROM byte data, ROM ready

Description:
A specialized read adapter that bridges the gap between single-byte SNES requests and the burst-oriented DDR3 controller. It aligns incoming addresses, manages command issuance to the MIG, and implements a caching mechanism to provide rapid access to data already present within a retrieved burst.

Purpose:
Translates burst-based DDR3 memory access into the byte-level addressing expected by the emulator, handling the underlying handshaking and alignment logic internally.

Module: sdcard_init

Inputs: clk, reset, RAM command/data FIFO readiness, SD card MISO
Outputs: RAM write command/address/data signals, RAM initialization status, SD card SPI outputs

Description:
A startup state machine responsible for the initial ROM load. It communicates with the SD card to retrieve raw binary blocks, which are then packed into 64-bit words and committed to DDR3 memory. The process continues until the entire configured address range is populated.

Purpose:
Automates the transfer of ROM files from the physical SD card to the high-speed DDR3 memory required for active emulation on the Urbana board.

Module: SdCardCtrl

Inputs: clk_i, reset_i, read/write/continue requests, block address, write data, handshake input, SD card MISO
Outputs: read data, busy flag, handshake output, error code, SD card chip select, SD clock, SD card MOSI

Description:
An SPI-based SD card protocol engine sourced from the XESS library. It manages the low-level physical signalling for card initialization, block-level data transfers, and error monitoring, providing a clean host-side interface for block request logic.

Purpose:
Encapsulates the complex SPI communication and initialization sequence of the SDHC standard, providing a reliable foundation for the higher-level ROM loading process

---

Module: ram_reader

Inputs: clk, reset, RAM read readiness/data signals, requested read address
Outputs: RAM read command/address/enables, 16-bit read data, read-data-valid flag

Description:
A utility module for word-level DDR3 access. It monitors the read address, fetches new bursts as needed, and selects the appropriate 16-bit word from the returned data. This facilitates simpler interaction with the DDR3 fabric when byte-level alignment is not the primary concern.

Purpose:
Simplifies the interface to the MIG by handling burst management and selection, making word-level data retrieval straightforward for debugging and state monitoring.

Module: rtl_ddr3_top

Inputs: Differential DDR3 system clock, reference clock, reset, SD card MISO, switches
Outputs: DDR3 physical interface signals, SD card SPI outputs, hex display controls, RAM init status, LEDs

Description:
A validation top-level for the DDR3 and SD card infrastructure. It ties together the initialization logic, physical memory interface, and board peripherals to verify that data can be loaded and read correctly before full emulator integration.

Purpose:
Serves as a diagnostic platform for isolated testing of the memory storage system, ensuring the SD-to-DDR3 path is functional through visual verification on LEDs and hex displays.

Module: pulse_cdc

Inputs: src_clk, src_rst_n, src_pulse, dst_clk, dst_rst_n
Outputs: dst_pulse

Description:
A synchronization module for crossing clock domains with pulse signals. It utilizes a toggle-and-detect strategy to ensure that single-cycle events in the source domain are accurately captured and recreated in the destination domain without metastability.

Purpose:
Ensures reliable communication of control pulses between asynchronous clock regions, preventing synchronization errors in a system with multiple frequency domains.

CONTROLLER MODULES

Module: pi_gamepad_receiver_strobe

Inputs: clk, reset, [15:0] pi_gamepad_data, [2:0] pi_report_sel
Outputs: [15:0] gamepad1, [15:0] gamepad2, [2:0] debug_report_sel, [14:0] debug_gamepad_data, debug_toggle_sync, debug_packet_edge, debug_p1_seen, debug_p2_seen, debug_p1_pulse, debug_p2_pulse

Description:
This module captures non-synchronous controller transmissions from the Raspberry Pi GPIO and translates them into stable, registered state values for the SNES. The Pi transmits 15 bits of input data alongside a packet-toggle strobe located on pi_gamepad_data[15]. The receiver synchronizes these toggle events and report selections into the FPGA clock domain, implementing a multi-cycle latching delay to ensure the data is properly assigned to either gamepad1 or gamepad2. Furthermore, the module enforces input hygiene by neutralizing contradictory directional presses, such as simultaneous Up/Down or Left/Right commands.

Purpose:
Establishes a robust communication link between the Raspberry Pi script and the FPGA core, shielding the emulator from raw GPIO signal instability and providing clean player input registers for the wider system.

Module: snes_controller_bus_bridge

Inputs: clk, rst_n, cpu_ce, master_ce, ppu_frame_level, ppu_vblank_level, ppu_hblank_level, [15:0] controller_gamepad1, [15:0] controller_gamepad2, CPU bus input signals, memory-map read data/ready
Outputs: CPU read data/ready, memory-map bus output signals, [15:0] joy1_auto_debug, [15:0] joy2_auto_debug, joy_auto_enable_debug, joy_busy_debug

Description:
Acts as an intermediary that monitors CPU memory transactions specifically targeting controller I/O. Accesses to $4016 and $4017 are parsed to return bit-serial data for manual software polling, while requests to $4218–$421B provide the standard auto-joypad formatting. The bridge also observes writes to $4016 for latching control and $4200 to supervise the auto-read status, while passing unrelated transactions through to the primary memory map. Internally, the logic maps active-high button reports into the formal SNES word structure and tracks the necessary serial indices for manual register reads.

Purpose:
Integrates the hardware register behavior required by SNES software into the existing memory architecture, allowing the processor to access Pi-sourced inputs via standard addresses using both automatic and manual polling methods.

APU MODULES

Module: apu_top

Inputs: cpu_clk, cpu_rst_n, apu_clk, apu_rst_n, enable, pal, apu_cs, [1:0] apu_pa, apu_pard_n, apu_pawr_n, [7:0] apu_cpu_di
Outputs: [7:0] apu_cpu_do, apu_ready, [15:0] audio_l, [15:0] audio_r, audio_ready

Description:
A high-level wrapper that integrates the SNEStang SMP and DSP modules into the design. It manages the synchronization of CPU-domain $2140–$2143 port accesses into the APU's independent clock domain through a ready-handshake protocol. The module links the sound processors to a shared memory space and delivers signed 16-bit stereo PCM audio data.

Purpose:
Facilitates communication between the central processor and the audio subsystem while maintaining clock domain

isolation. It serves as the primary gateway for the memory map to interface with the sound generation hardware.

Module: SMP

Inputs: CLK, RST_N, CE, ENABLE, SYSCLKF_CE, [7:0] DI, [1:0] PA, PARD_N, PAWR_N, [7:0] CPU_DI, CS, CS_N, debug inputs
Outputs: [15:0] A, [7:0] DO, WE_N, [7:0] CPU_DO, debug outputs, BRK_OUT

Description:
The audio CPU subsystem based on the SPC700 architecture, sourced from SNEStang. This module manages APU-side program execution, including internal timing, communication port logic, and IPL boot ROM routines. It directs all memory transactions within the 64 KB APU RAM space.

Purpose:
Executes audio programs independently of the main CPU. Software uploads sound code via the $2140–$2143 registers, allowing the SMP to autonomously drive the DSP for music and sound effect synthesis.

Module: DSP

Inputs: CLK, RST_N, ENABLE, PAL, SMP interface signals, APU RAM read data, debug inputs
Outputs: SMP interface data/control, APU RAM address/data/control, [15:0] AUDIO_L, [15:0] AUDIO_R, SND_RDY, debug output

Description:
The digital signal processing engine for the emulator. It manages voice state machines, BRR-encoded sample decompression, pitch modulation, and envelope generation. The logic also implements noise, echo, and stereo mixing, signaling sample readiness via the SND_RDY flag.

Purpose:
Produces the final audio streams from raw sample data and register states. It handles the complexities of voice mixing and stereo PCM generation to provide the actual sound for emulated titles.

File: dsp.vh

Inputs: None
Outputs: Macro definitions / constants used by DSP

Description:
A Verilog header file utilized by the DSP module. It contains centralized definitions for register indices, voice step constants, and various architectural macros required by the signal processor implementation.

Purpose:
Maintains a clean separation between hardware logic and static definitions, improving the maintainability and readability of the audio processing codebase.

Module: apu_pwm

Inputs: clk, rst_n, enable, [15:0] sample_l, [15:0] sample_r, sample_valid
Outputs: pwm_l, pwm_r

Description:
A sigma-delta PDM modulator designed for audio output. It processes 16-bit signed PCM data, translating the samples into offset-binary values. High-speed accumulators then generate the single-bit density streams required for external filtering.

Purpose:
Converts digital PCM samples into a format compatible with the Urbana board's physical audio circuit, enabling sound output without the need for an external DAC chip.

THIS NEXT FOLDER IS THE SPC700 FOLDER FROM SNESTANG, it includes ALL the required modules for the SCP700 Processor which the APU is run off. Our portion of the project did not write these files. To reduce the page count, instead of having a module description for each of the 10+ modules in this folder, a high level overview has been provided instead

Module Group: SPC700 support folder
Inputs: CPU/APU bus control signals, APU clock/reset, APU RAM data, DSP register data, interrupt/reset control, and internal processor control signals
Outputs: APU RAM addresses/data/control, DSP register accesses, CPU/APU communication port data, processor status/debug signals

Description:
The SPC700 folder contains the lower-level modules used to implement the SNES audio CPU subsystem. In the original SNES, audio is handled by a separate Sony SPC700 processor, which runs independently from the main 65C816 CPU. The SPC700 executes audio driver code uploaded by the game, manages APU RAM, communicates with the main CPU through the $2140–$2143 ports, and controls the DSP registers used for sample playback and mixing.

In this project, these files were brought in from SNEStang as part of the APU integration. Rather than rewriting the SPC700 processor from scratch, the design reuses the existing SNEStang SPC700 implementation and connects it to the project's APU top-level wrapper, APU RAM, and DSP module. The support files include the processor core logic, instruction decoding/control, register handling, arithmetic/logic operations, memory access sequencing, timers, and bus interfaces needed for the SPC700 to run SNES audio programs.

Purpose:
Provides the internal audio CPU needed for accurate SNES sound behavior. The main CPU does not directly generate music or sound effects; instead, it sends commands and uploads audio code/data to the APU. The SPC700 support modules execute that audio code and drive the DSP, allowing games to play music and sound effects through the SNEStang DSP/audio pipeline. For this project, the SPC700 folder served as the imported processor foundation for the APU, while the main design work focused on integrating it with the memory map, clock domains, APU RAM, DSP, and final audio output.

PPU MODULES - (This module went through lots of changes, only the used modules are included here, the submitted code may have additional modules that were left in but scrapped from the final design)

Module: ppu_top

Inputs: clk, rst_n, ppu_ce, snes_ce, CPU bus signals, DMA write signals, DMA/HDMA B-bus read signals, VRAM/CGRAM/OAM read data, OBJ OAM read data, debug_sel
Outputs: VRAM/CGRAM/OAM address/write signals, video RGB output, hsync, vsync, vde, HDMA timing pulses, live PPU timing status, DMA B-bus readback data/ready, debug_ppu, debug_render_vis

Description:
The top-level integration layer for the custom PPU architecture. This module serves as the primary structural hub, linking the register file, scanline timing engine, multi-layer renderer, deferred compositor, and color math stage to the final VGA/HDMI scanout path. It orchestrates the flow of CPU and DMA transactions into the internal register state, handles memory arbitration across VRAM/CGRAM/OAM, and manages the conversion of SNES-timed pixel data into the buffered format required for digital video output.

Purpose:
Acts as the central control unit for the PPU subsystem. It coordinates the various rendering and timing submodules to ensure the emulator produces a stable, scaled video signal while maintaining the register-based behavior expected by official software routines.

Module: ppu_regs

Inputs: clk, rst_n, snes_ce, CPU address/data/write/read signals, PPU readback register selector, live H/V counters, field bit, VRAM/CGRAM/OAM readback data
Outputs: PPU control registers, background configuration, scroll registers, VRAM/CGRAM/OAM write interfaces, window registers, color-math registers, SETINI/misc video signals, CPU readback data/ready, debug register output

Description:
The memory-mapped interface for the PPU, implementing the standard SNES $2100–$213F video register range. It decodes incoming CPU and DMA transactions to update internal parameters such as screen brightness, background modes, tilemap pointers, and scroll offsets. Beyond configuration, the module facilitates readback paths for H/V counters and video memory, translating host-side requests into the appropriate internal state updates for the downstream rendering pipeline.

Purpose:
Provides the software-visible interface for video control. It translates raw bus transactions into the specific configuration signals used by the compositor, renderer, and timing logic to drive the emulated display.

Module: ppu_timing

Inputs: clk, rst_n, ppu_ce, frame_sync, nmi_en, overscan_en, interlace_en
Outputs: screen_x, screen_y, dot, scanline, field, visible-region flags, hblank, vblank, prerender status, line/frame/hblank/vblank/NMI/HDMA pulses

Description:
The master timing generator for the emulated video system. This component tracks the horizontal dot and vertical scanline positions to define the visible frame and blanking intervals. It generates critical synchronization pulses for the DMA engine and NMI logic, ensuring that per-scanline effects like HDMA are triggered at the correct intervals. It also supports phase-alignment between the internal PPU clock and the external VGA scanout logic.

Purpose:
Establishes the temporal framework for the entire PPU. Accurate scanline and dot tracking are essential for coordinating rendering tasks, HDMA scheduling, and synchronization with the physical display controller.

Module: ppu_render_new

Inputs: clk, rst_n, ppu_ce, display_screen_y, display_line_start, PPU background/scroll/tilemap registers, VRAM read data, OBJ configuration, OBJ OAM read data
Outputs: VRAM read address, OBJ OAM read address, 51-bit deferred pixel writes, final line-buffer write address/data/enable, line-valid/tag information, renderer debug signals

Description:
The primary graphics engine responsible for producing background and sprite candidates. It fetches tile and attribute data from VRAM and OAM, applying scroll offsets and configuration settings to generate pixel data for all active layers. Rather than producing immediate color values, the renderer outputs a 51-bit deferred packet for every coordinate, preserving priority and palette indices for later resolution in the PPU-timed path.

Purpose:
Decouples raw pixel generation from final display composition. By producing deferred candidates, the architecture allows timing-sensitive effects like windowing and color math to be evaluated with cycle-accurate register states during scanout.

Module: ppu_deferred_compositor

Inputs: clk, rst_n, ppu_ce, [50:0] deferred_pixel_packet, [7:0] screen_x, background mode/priority controls, main/subscreen enables, window enable/register signals
Outputs: [24:0] pixel_packet

Description:
A resolution logic module that evaluates the 51-bit deferred candidate packets to determine the winning pixels for both the main and subscreen. It applies priority logic, layer enables, and window masks to select the appropriate source for each screen position. The resulting 25-bit packet includes the final CGRAM indices and color-window metadata required for the subsequent processing stages.

Purpose:
Manages the final selection of background and object layers. This module resolves overlapping candidates and prepares the necessary color indices for the palette lookup and blending stages.

Module: ppu_colour

Inputs: clk, rst_n, ppu_ce, brightness, [24:0] pixel_packet, color-math registers, CGRAM read data, mirrored subscreen CGRAM read data
Outputs: CGRAM read address, subscreen CGRAM read address, [7:0] red, [7:0] green, [7:0] blue

Description:
The final stage of the PPU pipeline, responsible for color synthesis and blending. It retrieves RGB values from CGRAM based on the compositor's indices and applies brightness scaling alongside SNES color math operations. The logic supports transparency and lighting effects through subscreen blending and uses window metadata to enforce regional clipping and math enables.

Purpose:
Translates abstract color indices into final digital video samples. It implements the complex palette lookup and math-based blending routines that characterize the visual style of SNES titles.

Module: ppu_write_cdc

Inputs: CPU/UI clock and reset, PPU clock and reset, source-domain write request/register/data signals, destination-domain ready/accept signals
Outputs: PPU-domain write request/register/data signals, source-domain full/busy/ready status signals

Description:
A dedicated synchronization module for transferring register write events across clock boundaries. It bridges the CPU or DMA domains with the PPU register domain using a robust handshake protocol, preventing metastability and ensuring that every write command is accurately captured by the video subsystem.

Purpose:
Enables reliable communication between the high-speed system bus and the independent PPU clock region, protecting the emulator's internal state from synchronization failures during real-time execution.

Module: ppu_read_cdc

Inputs: UI/CPU clock and reset, PPU clock and reset, read request/register selector from the UI side, PPU readback data/ready
Outputs: PPU-domain read request/register selector, UI-domain read data/ready

Description:
A specialized clock-domain crossing engine for register readback operations. This component manages the transfer of read requests from the CPU into the PPU domain and orchestrates the safe return of data, ensuring that the host can query the PPU state without risking timing violations or data corruption.

Purpose:
Facilitates the retrieval of internal PPU data by the main processor and debugging tools, maintaining signal integrity across asynchronous clock boundaries within the FPGA design.

RTL Design

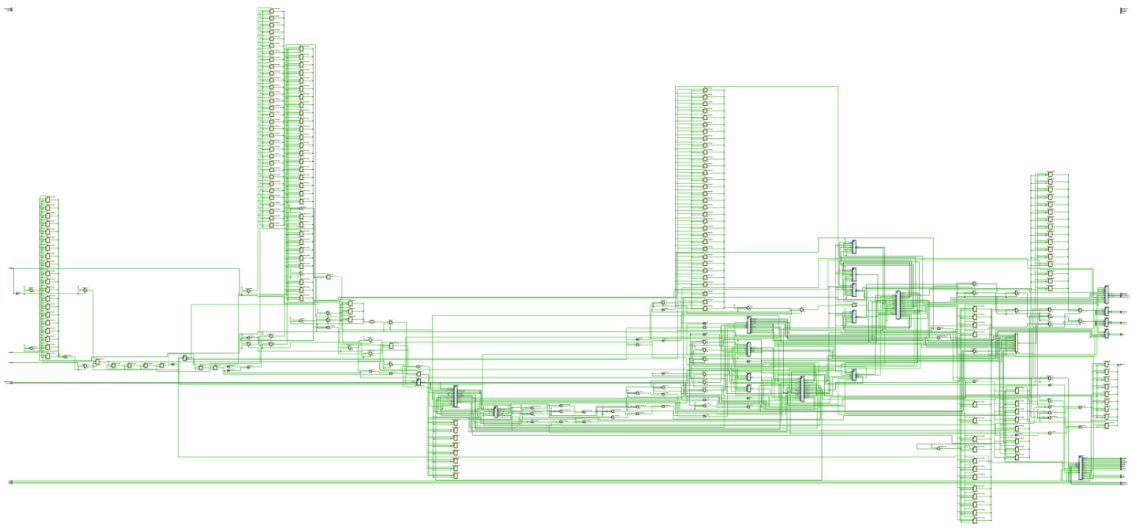

Figure 2: Vivado-generated Block Design

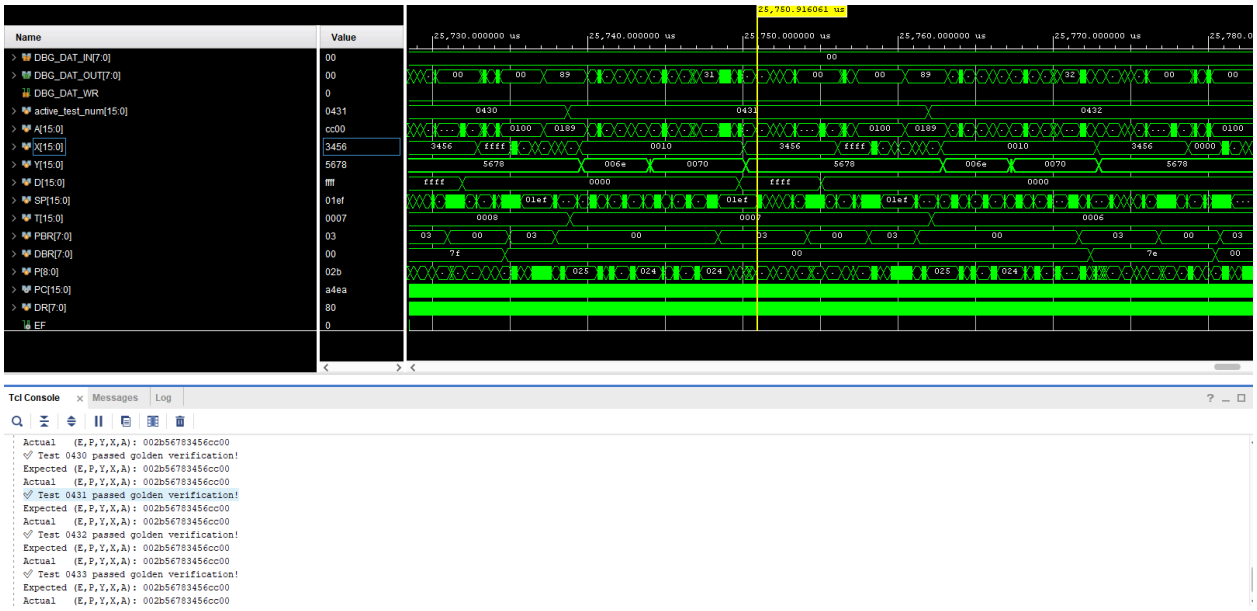

Figure 3: CPU testbench results; adapted from https://github.com/gilyon/snes-tests/tree/main/cputest

Post-Implementation Utilization

LUT	23925
DSP	12
Memory (BRAM)	72.5
Flip-Flop	23532
Latches	0
Frequency (MHz)	40.923
Static Power (mW)	82
Dynamic Power (mW)	1088
Total Power (mW)	1169

Figure 4: Final SNES Utilization Values

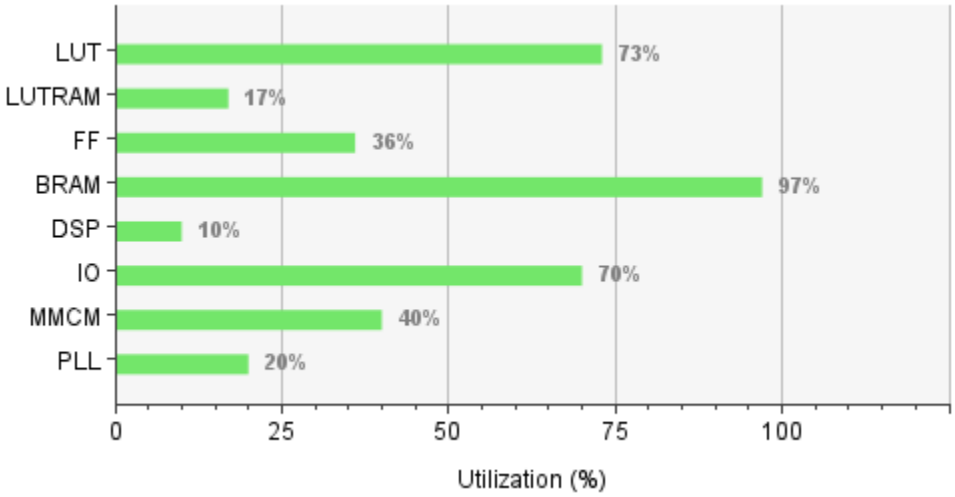

Figure 5: Final SNES Utilization graph

Setup, ROMs, & Gameplay

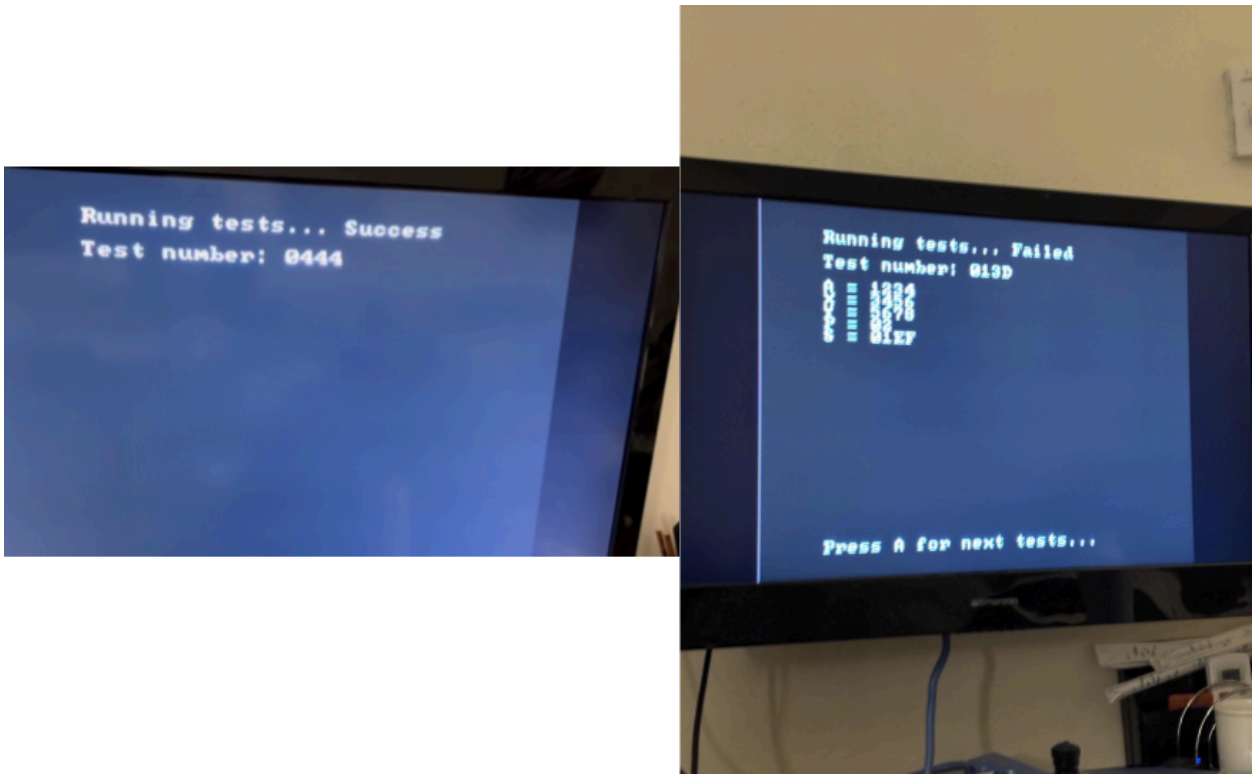

Figure 6: (left) CPU test ROM succeeding in our final design
(right) CPU test ROM failing in development
Uses https://github.com/gilyon/snes-tests/tree/main/cputest

Figure 7: (left) Setup of Urbana Board connected to Pi through GPIO and USB controllers connected to Pi (right) Demonstration of Super Mario World save selection screen with this setup

Figure X: Super Mario World gameplay from ECE385 Final Project Showcase

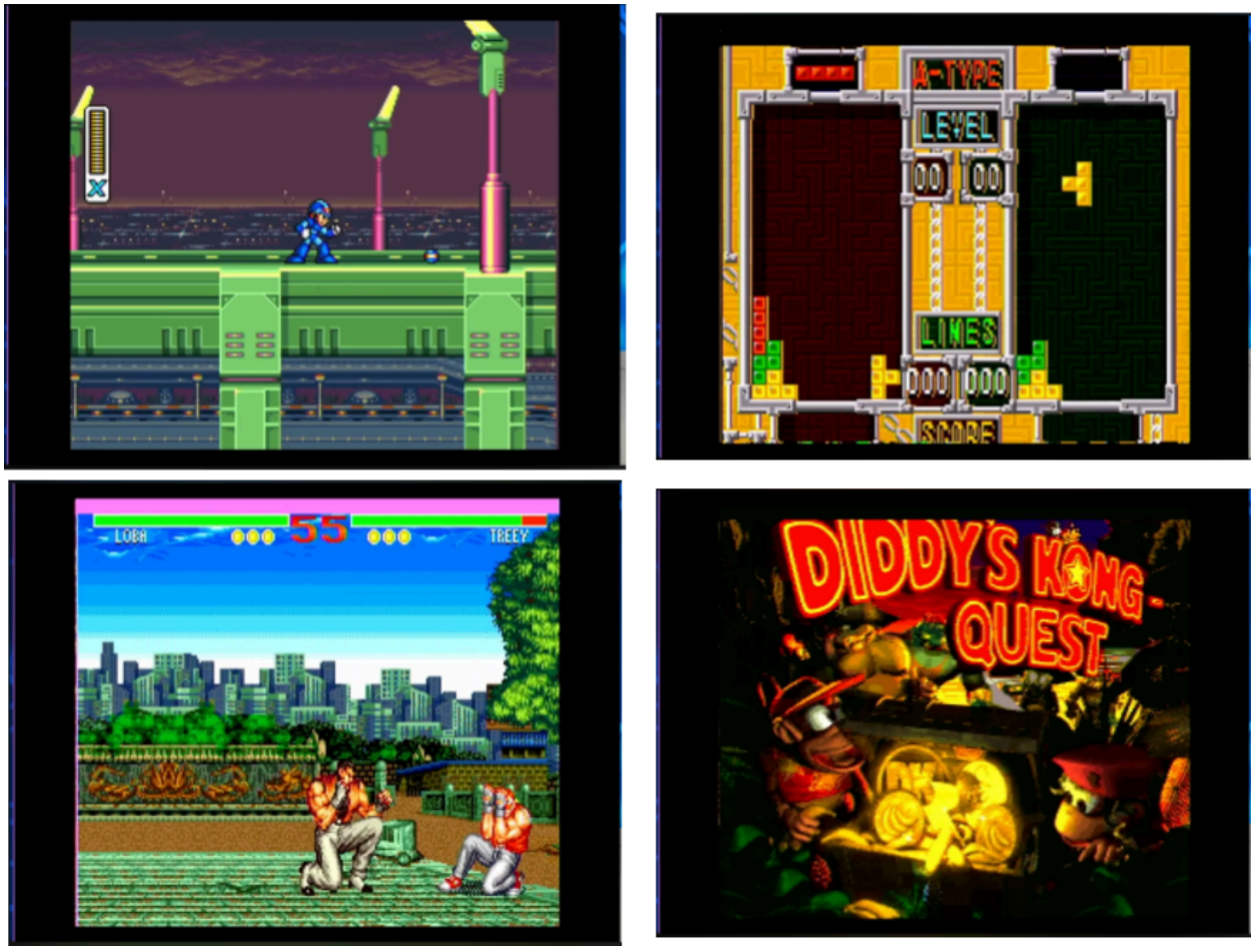

Figure 8: Gameplay from Mega Man X (top left), Tetris & Dr. Mario (top right), King of Fighters 2000 (bottom left) and Donkey Kong Country 2: Diddy's Kong Quest (bottom right) from ECE385 Final Project Showcase

Conclusion

This project was most certainly a massive undertaking as over 100 hours have been spent designing, debugging, and developing the Super Nintendo as best as we could over the span of a couple weeks. While there is much left to be improved upon including Mode 7 and expansion chip implementations, the current design as is can play several commercial SNES games near full-speed with minor graphical inconsistencies. Using Vivado and the Urbana Board's unique constraints caused us to have to think of creative ways to work around the limitations of our environment to make the best possible SNES we could. The complex architecture surrounding the PPU, APU, CPU, the various memory modules, and controller support culminated in the design we've presented here.

Acknowledgements

This project would not have been possible without the help from:

https://github.com/gilyon/snes-tests/tree/main/cputest - Main CPU test ROM we've used during development

https://github.com/nand2mario/snestang - CPU and APU implementations were used from here

https://problemkaputt.de/fullsnes.htm - probably the most comprehensive documentation regarding the SNES architecture

https://www.youtube.com/playlist?list=PLHQ0utQyFw5KCcj1ljIhExH_lvGwfn6GV - probably the most comprehensive video series on the SNES architecture (which inspired us to pursue this project in the first place)

Claude Sonnet 4.6, GPT 5.5, Gemini 3.1 Pro - a combination of these generative AI tools were used in the making of the glue logic of the CPU and APU to the rest of our design, as well as instrumenting debug signals, generating constraint files, generating test scripts, and general debugging of the PPU render pipeline, DDR3 memory control logic, CPU timing, and controller logic.

AI was used to bounce and reference ideas off, but AI was primarily used to create 2 modules, the CPU to PPU FIFO too counter CDC and the DMA channels

The following code was generated fully by AI with no human editing: (The PPU Write and Read FIFOs to prevent CDC and the DMA)

While AI was used in other components it was used in tandem with our own research and logic and to bounce ideas, no other code was fully AI generated and not human involved.

PPU Write CDC:

```
`timescale 1ns / 1ps

module ppu_write_cdc #(
    parameter int FIFO_DEPTH = 2048
)(
    input logic wr_clk,
    input logic rst,

    input logic wr_valid,
    output logic wr_ready,
    input logic [7:0] wr_reg,
    input logic [7:0] wr_data,

    input logic rd_clk,

    output logic ppu_we,
    output logic [23:0] ppu_addr,
    output logic [7:0] ppu_data,

    // debug
    output logic        fifo_full,
    output logic        fifo_empty,
    output logic        wr_rst_busy,
    output logic        rd_rst_busy,
    output logic [15:0] dbg_wr_count,
    output logic [15:0] dbg_rd_count,
    output logic [15:0] dbg_last_wr,
    output logic [15:0] dbg_last_rd,
    output logic [15:0] dbg_drop_count
);

    localparam  int COUNT_WIDTH = $clog2(FIFO_DEPTH) + 1;

    logic [15:0] fifo_din;
    logic [15:0] fifo_dout;
    logic fifo_wr_en;
    logic fifo_rd_en;

    assign fifo_din = {wr_reg, wr_data};
```

```
assign wr_ready = !fifo_full && !wr_rst_busy && !rst;
assign fifo_wr_en = wr_valid && wr_ready;

xpm_fifo_async #(
   .CASCADE_HEIGHT      (0),
   .CDC_SYNC_STAGES     (2),
   .DOUT_RESET_VALUE    ("0"),
   .ECC_MODE            ("no_ecc"),
   .FIFO_MEMORY_TYPE    ("auto"),
   .FIFO_READ_LATENCY   (0),
   .FIFO_WRITE_DEPTH    (FIFO_DEPTH),
   .FULL_RESET_VALUE    (0),
   .PROG_EMPTY_THRESH   (10),
   .PROG_FULL_THRESH    (FIFO_DEPTH - 16),
   .RD_DATA_COUNT_WIDTH (COUNT_WIDTH),
   .READ_DATA_WIDTH     (16),
   .READ_MODE           ("fwft"),
   .RELATED_CLOCKS      (0),
   .SIM_ASSERT_CHK      (0),
   .USE_ADV_FEATURES    ("0000"),
   .WAKEUP_TIME         (0),
   .WRITE_DATA_WIDTH    (16),
   .WR_DATA_COUNT_WIDTH (COUNT_WIDTH)
) ppu_write_fifo (
   .rst           (rst),

   .wr_clk        (wr_clk),
   .wr_en         (fifo_wr_en),
   .din           (fifo_din),
   .full          (fifo_full),
   .wr_rst_busy   (wr_rst_busy),

   .rd_clk        (rd_clk),
   .rd_en         (fifo_rd_en),
   .dout          (fifo_dout),
   .empty         (fifo_empty),
   .rd_rst_busy   (rd_rst_busy),

   .sleep         (1'b0),
   .injectsbiterr (1'b0),
   .injectdbiterr (1'b0),

   .almost_empty  (),
```

```
    .almost_full   (),
    .data_valid    (),
    .dbiterr       (),
    .overflow      (),
    .prog_empty    (),
    .prog_full     (),
    .rd_data_count (),
    .sbiterr       (),
    .underflow     (),
    .wr_ack        (),
    .wr_data_count ()
);

// Debug counters, write side
always_ff @(posedge wr_clk or posedge rst) begin
    if (rst) begin
        dbg_wr_count   <= 16'd0;
        dbg_last_wr    <= 16'h0000;
        dbg_drop_count <= 16'd0;
    end else begin
        if (fifo_wr_en) begin
            dbg_wr_count <= dbg_wr_count + 16'd1;
            dbg_last_wr  <= fifo_din;
        end

        if (wr_valid && !wr_ready) begin
            dbg_drop_count <= dbg_drop_count + 16'd1;
        end
    end
end

// Read side: FWFT means fifo_dout is valid whenever fifo_empty == 0.
typedef enum logic [1:0] {
    RD_IDLE  = 2'd0,
    RD_PULSE = 2'd1
} rd_state_t;

rd_state_t state;
logic [15:0] word_latched;

always_ff @(posedge rd_clk or posedge rst) begin
    if (rst) begin
        state       <= RD_IDLE;
        fifo_rd_en   <= 1'b0;
```

```
            ppu_we       <= 1'b0;
            word_latched <= 16'h0000;
            dbg_rd_count <= 16'd0;
            dbg_last_rd  <= 16'h0000;
        end else begin
            fifo_rd_en <= 1'b0;
            ppu_we     <= 1'b0;

            if (rd_rst_busy) begin
                state <= RD_IDLE;
            end else begin
                unique case (state)

                    RD_IDLE: begin
                        if (!fifo_empty) begin
                            word_latched <= fifo_dout;
                            fifo_rd_en   <= 1'b1;
                            state        <= RD_PULSE;
                        end
                    end

                    RD_PULSE: begin
                        ppu_we       <= 1'b1;
                        dbg_rd_count <= dbg_rd_count + 16'd1;
                        dbg_last_rd  <= word_latched;
                        state        <= RD_IDLE;
                    end

                    default: begin
                        state <= RD_IDLE;
                    end

                endcase
            end
        end
    end

    assign ppu_addr = {8'h00, 8'h21, word_latched[15:8]};
    assign ppu_data = word_latched[7:0];

endmodule
```

PPU READ CDC:

```
`timescale 1ns / 1ps
module ppu_read_cdc (
```

```
    input  logic       ui_clk,
    input  logic       ui_rst_n,
    input  logic       rd_req_ui,
    input  logic [7:0] rd_reg_ui,
    output logic [7:0] rd_data_ui,
    output logic       rd_ready_ui,
    input  logic       ppu_clk,
    input  logic       ppu_rst_n,
    output logic       ppu_req,
    output logic [7:0] ppu_reg,
    input  logic [7:0] ppu_data,
    input  logic       ppu_ready
);
    logic req_toggle_ui, req_busy_ui;
    logic [7:0] req_reg_hold_ui;
    logic ack_toggle_ppu;
    logic [7:0] data_hold_ppu;
    logic ack_m1, ack_m2, ack_seen;
    logic req_m1, req_m2, req_seen;

    typedef enum logic [1:0] {P_IDLE, P_WAIT, P_ACK} pstate_t;
    pstate_t pstate;

    always_ff @(posedge ui_clk or negedge ui_rst_n) begin
        if (!ui_rst_n) begin
            req_toggle_ui <= 1'b0; req_busy_ui <= 1'b0; req_reg_hold_ui <= 8'h00;
            ack_m1 <= 1'b0; ack_m2 <= 1'b0; ack_seen <= 1'b0;
            rd_data_ui <= 8'h00; rd_ready_ui <= 1'b0;
        end else begin
            ack_m1 <= ack_toggle_ppu;
            ack_m2 <= ack_m1;
            rd_ready_ui <= 1'b0;
            if (!req_busy_ui && rd_req_ui) begin
                req_reg_hold_ui <= rd_reg_ui;
                req_toggle_ui <= ~req_toggle_ui;
                req_busy_ui <= 1'b1;
            end
            if (req_busy_ui && (ack_m2 != ack_seen)) begin
                ack_seen <= ack_m2;
                rd_data_ui <= data_hold_ppu;
                rd_ready_ui <= 1'b1;
                req_busy_ui <= 1'b0;
            end
        end
```

```
  end

  always_ff @(posedge ppu_clk or negedge ppu_rst_n) begin
    if (!ppu_rst_n) begin
      req_m1 <= 1'b0; req_m2 <= 1'b0; req_seen <= 1'b0; ack_toggle_ppu <= 1'b0;
      data_hold_ppu <= 8'h00; ppu_req <= 1'b0; ppu_reg <= 8'h00; pstate <= P_IDLE;
    end else begin
      req_m1 <= req_toggle_ui;
      req_m2 <= req_m1;
      ppu_req <= 1'b0;
      unique case (pstate)
        P_IDLE: if (req_m2 != req_seen) begin
          req_seen <= req_m2;
          ppu_reg <= req_reg_hold_ui;
          ppu_req <= 1'b1;
          pstate <= P_WAIT;
        end
        P_WAIT: begin
          ppu_req <= 1'b1;
          if (ppu_ready) begin
            data_hold_ppu <= ppu_data;
            ppu_req <= 1'b0;
            pstate <= P_ACK;
          end
        end
        P_ACK: begin
          ack_toggle_ppu <= ~ack_toggle_ppu;
          pstate <= P_IDLE;
        end
        default: pstate <= P_IDLE;
      endcase
    end
  end
endmodule
```

DMA - ONLY CHANNEL 0 – This is the entire DMA file, but only channel 0 was fully AI generated with no human input, channels 1 - 7 were adapted from the AI code and added on.

```
`timescale 1ns / 1ps

module snes_dma (
    input  logic        clk,
    input  logic        rst_n,

    input  logic        cpu_reg_we,
    input  logic        cpu_reg_re,
    input  logic [15:0] cpu_offs,
    input  logic [7:0]  cpu_wdata,
    output logic [7:0]  cpu_rdata,
    output logic        cpu_rdata_valid,

    input  logic        hdma_init_pulse,
    input  logic        hdma_hblank_pulse,

    output logic        dma_rom_req,
    output logic [23:0] dma_rom_addr,
    input  logic [7:0]  dma_rom_rdata,
    input  logic        dma_rom_ready,

    output logic        dma_wram_en,
    output logic        dma_wram_we,
    output logic [16:0] dma_wram_addr,
    output logic [7:0]  dma_wram_wdata,
    input  logic [7:0]  dma_wram_rdata,

    output logic        dma_ppu_we,
    output logic [7:0]  dma_ppu_reg,
    output logic [7:0]  dma_ppu_data,
    input  logic        ppu_wr_ready,

    output logic        dma_bbus_rd_req,
    output logic [7:0]  dma_bbus_rd_reg,
    input  logic [7:0]  dma_bbus_rd_data,
    input  logic        dma_bbus_rd_ready,

    output logic        dma_active,

    input  logic [2:0]  debug_sel,
    output logic [15:0] debug_dma
);

    function automatic logic is_dma_reg(input logic [15:0] offs);
```

```
    return ((offs == 16'h420B) || (offs == 16'h420C) ||
            (offs >= 16'h4300 && offs <= 16'h437F) ||
            (offs >= 16'h2180 && offs <= 16'h2183));
endfunction

function automatic logic abus_is_wram(input logic [23:0] addr);
    logic [7:0] bank;
    logic [15:0] offs;
    begin
        bank = addr[23:16];
        offs = addr[15:0];
        return ((bank == 8'h7E) || (bank == 8'h7F) ||
                (((bank <= 8'h3F) || (bank >= 8'h80 && bank <= 8'hBF)) && offs <= 16'h1FFF));
    end
endfunction

function automatic logic [16:0] abus_to_wram_addr(input logic [23:0] addr);
    begin
        if (addr[23:16] == 8'h7E)      return {1'b0, addr[15:0]};
        else if (addr[23:16] == 8'h7F) return {1'b1, addr[15:0]};
        else                           return {4'b0000, addr[12:0]};
    end
endfunction

function automatic logic abus_is_lorom(input logic [23:0] addr);
    logic [7:0] bank;
    begin
        bank = addr[23:16];
        return addr[15] && ((bank <= 8'h3F) || (bank >= 8'h80 && bank <= 8'hBF));
    end
endfunction

function automatic logic [23:0] lorom_phys(input logic [23:0] addr);
    return {2'b00, addr[22:16], addr[14:0]};
endfunction

function automatic logic [2:0] unit_len(input logic [2:0] mode);
    begin
        unique case (mode)
            3'd0: unit_len = 3'd1;
            3'd1, 3'd2, 3'd6: unit_len = 3'd2;
            default: unit_len = 3'd4;
        endcase
    end
```

```
endfunction

function automatic logic [1:0] b_delta(input logic [2:0] mode, input logic [1:0] phase);
   begin
      unique case (mode)
         3'd0: b_delta = 2'd0;
         3'd1: b_delta = {1'b0, phase[0]};
         3'd2: b_delta = 2'd0;
         3'd3: b_delta = {1'b0, phase[1]};
         3'd4: b_delta = phase;
         3'd5: b_delta = {1'b0, phase[0]};
         3'd6: b_delta = 2'd0;
         3'd7: b_delta = {1'b0, phase[1]};
         default: b_delta = 2'd0;
      endcase
   end
endfunction

function automatic logic [2:0] first_ch(input logic [7:0] mask);
   begin
      if     (mask[0]) first_ch = 3'd0;
      else if (mask[1]) first_ch = 3'd1;
      else if (mask[2]) first_ch = 3'd2;
      else if (mask[3]) first_ch = 3'd3;
      else if (mask[4]) first_ch = 3'd4;
      else if (mask[5]) first_ch = 3'd5;
      else if (mask[6]) first_ch = 3'd6;
      else              first_ch = 3'd7;
   end
endfunction

logic [7:0] mdmaen_r, hdmaen_r;
logic [7:0] dmap_r [0:7], bbad_r [0:7], a1tl_r [0:7], a1th_r [0:7], a1b_r [0:7];
logic [7:0] dasl_r [0:7], dash_r [0:7], dasb_r [0:7], a2al_r [0:7], a2ah_r [0:7];
logic [7:0] ntrl_r [0:7], unused_r [0:7];
logic [7:0] hdma_active_mask;
logic       hdma_do [0:7];
logic       hdma_repeat [0:7];
// Decoded HDMA line counter. The raw NTRL byte uses 00h/80h for
// 128 lines depending on bit 7; keeping this decoded counter avoids
// treating 80h as an immediate table reload.
logic [7:0] hdma_line_count [0:7];
logic [16:0] wmadd_r;
```

```
logic [2:0] reg_ch;
logic [3:0] reg_idx;
assign reg_ch = cpu_offs[6:4];
assign reg_idx = cpu_offs[3:0];

typedef enum logic [5:0] {
   S_IDLE, S_MSEL, S_MBYTE, S_AREAD, S_AWAIT_WRAM, S_AWAIT_ROM,
   S_BWRITE, S_BWRITE_DONE, S_BREAD, S_BWAIT_WRAM, S_BWAIT_PPU,
   S_AWRITE, S_AWRITE_DONE, S_MADV,
   S_HINIT, S_HINIT_CH, S_HSCAN, S_HHEADER, S_HAFTER_HEADER,
   S_HPTR_LO, S_HAFTER_PTR_LO, S_HPTR_HI, S_HAFTER_PTR_HI,
   S_HXFER, S_HAFTER_AREAD, S_HAFTER_BREAD, S_HAFTER_BYTE,
   S_HDEC, S_HDONE
} state_t;

state_t state, return_state;
logic [2:0] active_ch, h_ch;
logic [23:0] cur_addr, op_addr;
logic [16:0] remaining;
logic [1:0] phase, h_phase, addr_step;
logic [2:0] xfer_mode, h_len;
logic [7:0] bbad_base, op_breg, op_wdata, op_rdata, byte_r;
logic reverse_dir;
logic pending_hinit, pending_hblank;

wire [7:0] active_mask_bit = (8'h01 << active_ch);
wire [15:0] h_a2 = {a2ah_r[h_ch], a2al_r[h_ch]};
wire [15:0] h_das = {dash_r[h_ch], dasl_r[h_ch]};
wire [23:0] h_table_addr = {a1b_r[h_ch], a2ah_r[h_ch], a2al_r[h_ch]};
wire [23:0] h_ind_addr = {dasb_r[h_ch], dash_r[h_ch], dasl_r[h_ch]};
wire [23:0] h_data_addr = dmap_r[h_ch][6] ? h_ind_addr : h_table_addr;

// Hold the CPU as soon as an MDMA start has been accepted, even during
// the one cycle before the FSM leaves IDLE. Seeing the CPU address sit
// on the STA $420B instruction during this time is expected.
assign dma_active = (state != S_IDLE) || (mdmaen_r != 8'h00);

always_comb begin
   cpu_rdata = 8'h00;
   cpu_rdata_valid = cpu_reg_re && is_dma_reg(cpu_offs);
   if (cpu_offs == 16'h420B) cpu_rdata = mdmaen_r;
   else if (cpu_offs == 16'h420C) cpu_rdata = hdmaen_r;
   else if (cpu_offs == 16'h2180) cpu_rdata = dma_wram_rdata;
   else if (cpu_offs == 16'h2181) cpu_rdata = wmadd_r[7:0];
```

```
    else if (cpu_offs == 16'h2182) cpu_rdata = wmadd_r[15:8];
    else if (cpu_offs == 16'h2183) cpu_rdata = {7'b0, wmadd_r[16]};
    else if (cpu_offs >= 16'h4300 && cpu_offs <= 16'h437F) begin
      unique case (reg_idx)
        4'h0: cpu_rdata = dmap_r[reg_ch];
        4'h1: cpu_rdata = bbad_r[reg_ch];
        4'h2: cpu_rdata = a1tl_r[reg_ch];
        4'h3: cpu_rdata = a1th_r[reg_ch];
        4'h4: cpu_rdata = a1b_r[reg_ch];
        4'h5: cpu_rdata = dasl_r[reg_ch];
        4'h6: cpu_rdata = dash_r[reg_ch];
        4'h7: cpu_rdata = dasb_r[reg_ch];
        4'h8: cpu_rdata = a2al_r[reg_ch];
        4'h9: cpu_rdata = a2ah_r[reg_ch];
        4'hA: cpu_rdata = ntrl_r[reg_ch];
        4'hB, 4'hF: cpu_rdata = unused_r[reg_ch];
        default: cpu_rdata = 8'h00;
      endcase
    end
  end

  always_comb begin
    dma_rom_req = 1'b0;
    dma_rom_addr = lorom_phys(op_addr);
    dma_wram_en = 1'b0;
    dma_wram_we = 1'b0;
    dma_wram_addr = 17'h00000;
    dma_wram_wdata = op_wdata;
    dma_ppu_we = 1'b0;
    dma_ppu_reg = op_breg;
    dma_ppu_data = op_wdata;
    dma_bbus_rd_req = 1'b0;
    dma_bbus_rd_reg = op_breg;

    unique case (state)
      S_AREAD, S_AWAIT_WRAM, S_AWAIT_ROM: begin
        if (abus_is_wram(op_addr)) begin
          dma_wram_en = 1'b1;
          dma_wram_addr = abus_to_wram_addr(op_addr);
        end else if (abus_is_lorom(op_addr)) begin
          dma_rom_req = 1'b1;
          dma_rom_addr = lorom_phys(op_addr);
        end
      end
```

```
      S_AWRITE, S_AWRITE_DONE: begin
        if (abus_is_wram(op_addr)) begin
          dma_wram_en = 1'b1;
          dma_wram_we = (state == S_AWRITE);
          dma_wram_addr = abus_to_wram_addr(op_addr);
          dma_wram_wdata = op_wdata;
        end
      end
      S_BREAD, S_BWAIT_WRAM: begin
        if (op_breg == 8'h80) begin
          dma_wram_en = 1'b1;
          dma_wram_addr = wmadd_r;
        end
      end
      S_BWAIT_PPU: begin
        dma_bbus_rd_req = 1'b1;
        dma_bbus_rd_reg = op_breg;
      end
      S_BWRITE, S_BWRITE_DONE: begin
        if (op_breg == 8'h80) begin
          dma_wram_en = 1'b1;
          dma_wram_we = (state == S_BWRITE);
          dma_wram_addr = wmadd_r;
          dma_wram_wdata = op_wdata;
        end else if (op_breg <= 8'h3F) begin
          dma_ppu_we = (state == S_BWRITE) && ppu_wr_ready;
          dma_ppu_reg = op_breg;
          dma_ppu_data = op_wdata;
        end
      end
      default: begin
        if (state == S_IDLE && cpu_reg_we && cpu_offs == 16'h2180) begin
          dma_wram_en = 1'b1;
          dma_wram_we = 1'b1;
          dma_wram_addr = wmadd_r;
          dma_wram_wdata = cpu_wdata;
        end else if (state == S_IDLE && cpu_reg_re && cpu_offs == 16'h2180) begin
          dma_wram_en = 1'b1;
          dma_wram_addr = wmadd_r;
        end
      end
    endcase
  end
```

```
  integer i;
  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      mdmaen_r <= 8'h00;
      hdmaen_r <= 8'h00;
      hdma_active_mask <= 8'h00;
      wmadd_r <= 17'h00000;
      pending_hinit <= 1'b0;
      pending_hblank <= 1'b0;
      for (i = 0; i < 8; i = i + 1) begin
        dmap_r[i] <= 8'h00; bbad_r[i] <= 8'h00;
        a1tl_r[i] <= 8'h00; a1th_r[i] <= 8'h00; a1b_r[i] <= 8'h00;
        dasl_r[i] <= 8'h00; dash_r[i] <= 8'h00; dasb_r[i] <= 8'h00;
        a2al_r[i] <= 8'h00; a2ah_r[i] <= 8'h00; ntrl_r[i] <= 8'h00;
        unused_r[i] <= 8'h00; hdma_do[i] <= 1'b0; hdma_repeat[i] <= 1'b0; hdma_line_count[i] <=
8'd0;
      end
      state <= S_IDLE; return_state <= S_IDLE;
      active_ch <= 3'd0; h_ch <= 3'd0; cur_addr <= 24'h000000; op_addr <= 24'h000000;
      remaining <= 17'd0; phase <= 2'd0; h_phase <= 2'd0; addr_step <= 2'd0;
      xfer_mode <= 3'd0; h_len <= 3'd0; bbad_base <= 8'h00; op_breg <= 8'h00;
      op_wdata <= 8'h00; op_rdata <= 8'h00; byte_r <= 8'h00; reverse_dir <= 1'b0;
    end else begin
      if (hdma_init_pulse)   pending_hinit <= 1'b1;
      if (hdma_hblank_pulse) pending_hblank <= 1'b1;

      if (cpu_reg_we && is_dma_reg(cpu_offs) && state == S_IDLE) begin
        if (cpu_offs == 16'h420B) begin
          if (cpu_wdata != 8'h00) mdmaen_r <= cpu_wdata;
        end else if (cpu_offs == 16'h420C) begin
          hdmaen_r <= cpu_wdata;
          hdma_active_mask <= hdma_active_mask & cpu_wdata;
        end else if (cpu_offs == 16'h2181) begin
          wmadd_r[7:0] <= cpu_wdata;
        end else if (cpu_offs == 16'h2182) begin
          wmadd_r[15:8] <= cpu_wdata;
        end else if (cpu_offs == 16'h2183) begin
          wmadd_r[16] <= cpu_wdata[0];
        end else if (cpu_offs == 16'h2180) begin
          wmadd_r <= wmadd_r + 17'd1;
        end else if (cpu_offs >= 16'h4300 && cpu_offs <= 16'h437F) begin
          unique case (reg_idx)
            4'h0: dmap_r[reg_ch] <= cpu_wdata;
            4'h1: bbad_r[reg_ch] <= cpu_wdata;
```

```
          4'h2: a1tl_r[reg_ch] <= cpu_wdata;
          4'h3: a1th_r[reg_ch] <= cpu_wdata;
          4'h4: a1b_r[reg_ch] <= cpu_wdata;
          4'h5: dasl_r[reg_ch] <= cpu_wdata;
          4'h6: dash_r[reg_ch] <= cpu_wdata;
          4'h7: dasb_r[reg_ch] <= cpu_wdata;
          4'h8: a2al_r[reg_ch] <= cpu_wdata;
          4'h9: a2ah_r[reg_ch] <= cpu_wdata;
          4'hA: ntrl_r[reg_ch] <= cpu_wdata;
          4'hB, 4'hF: unused_r[reg_ch] <= cpu_wdata;
          default: ;
        endcase
      end
    end else if (cpu_reg_re && cpu_offs == 16'h2180 && state == S_IDLE) begin
      wmadd_r <= wmadd_r + 17'd1;
    end

    unique case (state)
      S_IDLE: begin
        // If the CPU just wrote MDMAEN, start the scheduler on the
        // very next cycle instead of waiting for a later IDLE poll.
        if (cpu_reg_we && cpu_offs == 16'h420B && cpu_wdata != 8'h00) begin
          mdmaen_r <= cpu_wdata;
          state <= S_MSEL;
        end else if (pending_hinit && hdmaen_r != 8'h00) begin
          pending_hinit <= 1'b0;
          state <= S_HINIT;
        end else if (pending_hinit) begin
          pending_hinit <= 1'b0;
        end else if (pending_hblank) begin
          pending_hblank <= 1'b0;
          if (hdma_active_mask != 8'h00) begin
            h_ch <= 3'd0;
            state <= S_HSCAN;
          end else if (mdmaen_r != 8'h00) begin
            state <= S_MSEL;
          end
        end else if (mdmaen_r != 8'h00) begin
          state <= S_MSEL;
        end
      end

      S_MSEL: begin
        if (pending_hinit && hdmaen_r != 8'h00) begin
```

```
                pending_hinit <= 1'b0; state <= S_HINIT;
              end else if (pending_hinit) begin
                pending_hinit <= 1'b0;
              end else if (pending_hblank && hdma_active_mask != 8'h00) begin
                pending_hblank <= 1'b0; h_ch <= 3'd0; state <= S_HSCAN;
              end else if (mdmaen_r == 8'h00) begin
                state <= S_IDLE;
              end else begin
                active_ch <= first_ch(mdmaen_r);
                cur_addr <= {a1b_r[first_ch(mdmaen_r)], a1th_r[first_ch(mdmaen_r)],
a1tl_r[first_ch(mdmaen_r)]};
                remaining <= ({dash_r[first_ch(mdmaen_r)], dasl_r[first_ch(mdmaen_r)]} == 16'h0000) ?
                        17'd65536 : {1'b0, dash_r[first_ch(mdmaen_r)], dasl_r[first_ch(mdmaen_r)]};
                reverse_dir <= dmap_r[first_ch(mdmaen_r)][7];
                addr_step <= dmap_r[first_ch(mdmaen_r)][4:3];
                xfer_mode <= dmap_r[first_ch(mdmaen_r)][2:0];
                bbad_base <= bbad_r[first_ch(mdmaen_r)];
                phase <= 2'd0;
                state <= S_MBYTE;
              end
            end

            S_MBYTE: begin
              if (pending_hinit && hdmaen_r != 8'h00) begin
                pending_hinit <= 1'b0; state <= S_HINIT;
              end else if (pending_hinit) begin
                pending_hinit <= 1'b0;
              end else if (pending_hblank && hdma_active_mask != 8'h00) begin
                pending_hblank <= 1'b0; h_ch <= 3'd0; state <= S_HSCAN;
              end else if (remaining == 17'd0) begin
                state <= S_MADV;
              end else begin
                op_breg <= bbad_base + {6'b000000, b_delta(xfer_mode, phase)};
                if (((bbad_base + {6'b000000, b_delta(xfer_mode, phase)}) == 8'h80) &&
abus_is_wram(cur_addr)) begin
                  // Real SNES WRAM-to-WRAM DMA through port 2180h is not possible.
                  state <= S_MADV;
                end else if (reverse_dir) begin
                  op_addr <= cur_addr;
                  return_state <= S_MADV;
                  state <= S_BREAD;
                end else begin
                  op_addr <= cur_addr;
                  return_state <= S_BWRITE;
```

```
                state <= S_AREAD;
              end
            end
          end

          S_AREAD: begin
            if (abus_is_wram(op_addr)) state <= S_AWAIT_WRAM;
            else if (abus_is_lorom(op_addr)) state <= S_AWAIT_ROM;
            else begin op_rdata <= 8'h00; if (return_state == S_BWRITE) op_wdata <= 8'h00; state <=
return_state; end
          end
          S_AWAIT_WRAM: begin
            op_rdata <= dma_wram_rdata;
            if (return_state == S_BWRITE) op_wdata <= dma_wram_rdata;
            state <= return_state;
          end
          S_AWAIT_ROM: begin
            if (dma_rom_ready) begin
              op_rdata <= dma_rom_rdata;
              if (return_state == S_BWRITE) op_wdata <= dma_rom_rdata;
              state <= return_state;
            end
          end

          S_BWRITE: begin
            if (op_breg == 8'h80) begin
              wmadd_r <= wmadd_r + 17'd1;
              state <= S_BWRITE_DONE;
            end else if (op_breg <= 8'h3F) begin
              if (ppu_wr_ready) state <= S_BWRITE_DONE;
            end else begin
              state <= S_BWRITE_DONE;
            end
          end
          S_BWRITE_DONE: state <= (return_state == S_BWRITE) ? S_MADV : return_state;

          S_BREAD: begin
            if (op_breg == 8'h80) begin
              state <= S_BWAIT_WRAM;
            end else if (op_breg <= 8'h3F) begin
              state <= S_BWAIT_PPU;
            end else begin
              // Unimplemented B-bus region; preserve transfer timing and
              // write an open-bus-style zero to the requested A-bus target.
```

```
      op_rdata <= 8'h00;
      op_wdata <= 8'h00;
      state <= S_AWRITE;
   end
end
S_BWAIT_WRAM: begin
   op_rdata <= dma_wram_rdata;
   op_wdata <= dma_wram_rdata;
   wmadd_r <= wmadd_r + 17'd1;
   state <= S_AWRITE;
end
S_BWAIT_PPU: begin
   if (dma_bbus_rd_ready) begin
      op_rdata <= dma_bbus_rd_data;
      op_wdata <= dma_bbus_rd_data;
      state <= S_AWRITE;
   end
end

S_AWRITE: state <= S_AWRITE_DONE;
S_AWRITE_DONE: state <= return_state;

S_MADV: begin
   if (remaining != 17'd0) begin
      remaining <= remaining - 17'd1;
      {dash_r[active_ch], dasl_r[active_ch]} <= remaining[15:0] - 16'd1;
      unique case (addr_step)
         2'b00: begin
            cur_addr[15:0] <= cur_addr[15:0] + 16'd1;
            {a1th_r[active_ch], a1tl_r[active_ch]} <= cur_addr[15:0] + 16'd1;
         end
         2'b10: begin
            cur_addr[15:0] <= cur_addr[15:0] - 16'd1;
            {a1th_r[active_ch], a1tl_r[active_ch]} <= cur_addr[15:0] - 16'd1;
         end
         default: begin
            {a1th_r[active_ch], a1tl_r[active_ch]} <= cur_addr[15:0];
         end
      endcase
      if ({1'b0, phase} + 3'd1 >= unit_len(xfer_mode)) phase <= 2'd0;
      else phase <= phase + 2'd1;
   end
   if (remaining <= 17'd1) begin
      mdmaen_r[active_ch] <= 1'b0;
```

```
      state <= ((mdmaen_r & ~active_mask_bit) != 8'h00) ? S_MSEL : S_IDLE;
    end else begin
      state <= S_MBYTE;
    end
end

S_HINIT: begin
    hdma_active_mask <= hdmaen_r;
    h_ch <= 3'd0;
    state <= S_HINIT_CH;
end
S_HINIT_CH: begin
    if (hdmaen_r[h_ch]) begin
      a2al_r[h_ch] <= a1tl_r[h_ch];
      a2ah_r[h_ch] <= a1th_r[h_ch];
      ntrl_r[h_ch] <= 8'h00;
      hdma_do[h_ch] <= 1'b0;
      hdma_repeat[h_ch] <= 1'b0;
      hdma_line_count[h_ch] <= 8'd0;
    end
    if (h_ch == 3'd7) state <= (mdmaen_r != 8'h00) ? S_MSEL : S_IDLE;
    else h_ch <= h_ch + 3'd1;
end

S_HSCAN: begin
    if (!hdma_active_mask[h_ch]) begin
      if (h_ch == 3'd7) state <= S_HDONE;
      else h_ch <= h_ch + 3'd1;
    end else if (hdma_line_count[h_ch] == 8'd0) begin
      op_addr <= h_table_addr;
      return_state <= S_HAFTER_HEADER;
      state <= S_AREAD;
    end else if (hdma_do[h_ch]) begin
      h_phase <= 2'd0;
      h_len <= unit_len(dmap_r[h_ch][2:0]);
      state <= S_HXFER;
    end else begin
      state <= S_HDEC;
    end
end

S_HAFTER_HEADER: begin
    {a2ah_r[h_ch], a2al_r[h_ch]} <= h_a2 + 16'd1;
    if (op_rdata == 8'h00) begin
```

```
        hdma_active_mask[h_ch] <= 1'b0;
        hdma_do[h_ch] <= 1'b0;
        hdma_repeat[h_ch] <= 1'b0;
        ntrl_r[h_ch] <= 8'h00;
        hdma_line_count[h_ch] <= 8'd0;
        state <= S_HDEC;
    end else begin
        ntrl_r[h_ch] <= op_rdata;
        hdma_line_count[h_ch] <= (op_rdata[6:0] == 7'd0) ? 8'd128 : {1'b0, op_rdata[6:0]};
        // Fullsnes defines 81h..FFh as repeat. 80h is the
        // 128-line non-repeat/pause encoding, despite bit 7=1.
        hdma_repeat[h_ch] <= op_rdata[7] && (op_rdata[6:0] != 7'd0);
        hdma_do[h_ch] <= 1'b1;
        if (dmap_r[h_ch][6]) state <= S_HPTR_LO;
        else begin
            h_phase <= 2'd0;
            h_len <= unit_len(dmap_r[h_ch][2:0]);
            state <= S_HXFER;
        end
    end
end

S_HPTR_LO: begin
    op_addr <= {a1b_r[h_ch], a2ah_r[h_ch], a2al_r[h_ch]};
    return_state <= S_HAFTER_PTR_LO;
    state <= S_AREAD;
end
S_HAFTER_PTR_LO: begin
    dasl_r[h_ch] <= op_rdata;
    {a2ah_r[h_ch], a2al_r[h_ch]} <= h_a2 + 16'd1;
    state <= S_HPTR_HI;
end
S_HPTR_HI: begin
    op_addr <= {a1b_r[h_ch], a2ah_r[h_ch], a2al_r[h_ch]};
    return_state <= S_HAFTER_PTR_HI;
    state <= S_AREAD;
end
S_HAFTER_PTR_HI: begin
    dash_r[h_ch] <= op_rdata;
    {a2ah_r[h_ch], a2al_r[h_ch]} <= h_a2 + 16'd1;
    h_phase <= 2'd0;
    h_len <= unit_len(dmap_r[h_ch][2:0]);
    state <= S_HXFER;
end
```

```
S_HXFER: begin
   if ({1'b0, h_phase} >= h_len) begin
      state <= S_HDEC;
   end else begin
      op_breg <= bbad_r[h_ch] + {6'b000000, b_delta(dmap_r[h_ch][2:0], h_phase)};
      op_addr <= h_data_addr;
      if (dmap_r[h_ch][7]) begin
         return_state <= S_HAFTER_BYTE;
         state <= S_BREAD;
      end else begin
         return_state <= S_HAFTER_AREAD;
         state <= S_AREAD;
      end
   end
end

S_HAFTER_AREAD: begin
   op_wdata <= op_rdata;
   return_state <= S_HAFTER_BYTE;
   state <= S_BWRITE;
end

S_HAFTER_BREAD: begin
   op_wdata <= op_rdata;
   state <= S_AWRITE;
end

S_HAFTER_BYTE: begin
   if (dmap_r[h_ch][6]) {dash_r[h_ch], dasl_r[h_ch]} <= h_das + 16'd1;
   else                 {a2ah_r[h_ch], a2al_r[h_ch]} <= h_a2 + 16'd1;
   h_phase <= h_phase + 2'd1;
   state <= S_HXFER;
end

S_HDEC: begin
   if (hdma_active_mask[h_ch] && hdma_line_count[h_ch] != 8'd0) begin
      hdma_line_count[h_ch] <= hdma_line_count[h_ch] - 8'd1;

      // Keep NTRL readable/debuggable. Zero means the next
      // HBlank will fetch a new table header.
      ntrl_r[h_ch][6:0] <= (hdma_line_count[h_ch] == 8'd1)
                           ? 7'd0
                           : (hdma_line_count[h_ch] - 8'd1);
```

```
                // Non-repeat entries transfer on the first line only;
                // repeat entries transfer on every line until the count
                // expires and a new table header is fetched.
                if (!hdma_repeat[h_ch]) hdma_do[h_ch] <= 1'b0;
              end
              if (h_ch == 3'd7) state <= S_HDONE;
              else begin
                h_ch <= h_ch + 3'd1;
                state <= S_HSCAN;
              end
            end

            S_HDONE: begin
              state <= (mdmaen_r != 8'h00) ? S_MSEL : S_IDLE;
            end

            default: state <= S_IDLE;
          endcase
        end
      end

      always_comb begin
        unique case (debug_sel)
          // State/handshake: SSssss P R W D MMMMMM
          //   state[5:0], ROM request, ROM ready, PPU FIFO ready, PPU write, MDMAEN[5:0]
          3'd0: debug_dma = {state[5:0], dma_rom_req, dma_rom_ready, ppu_wr_ready, dma_ppu_we,
mdmaen_r[5:0]};
          3'd1: debug_dma = remaining[15:0];
          3'd2: debug_dma = cur_addr[15:0];
          3'd3: debug_dma = {cur_addr[23:16], op_breg};
          3'd4: debug_dma = {dmap_r[active_ch], bbad_r[active_ch]};
          3'd5: debug_dma = {a2ah_r[h_ch], a2al_r[h_ch]};
          3'd6: debug_dma = {hdma_active_mask, hdmaen_r};
          3'd7: debug_dma = {h_ch, h_phase, reverse_dir, dma_bbus_rd_req, dma_bbus_rd_ready, byte_r};
          default: debug_dma = 16'hDEAD;
        endcase
      end
    endmodule
```