ChatGPT

Designing a Cycle-Accurate Emulator for the Super_Z80 Console

1. Emulator Architecture (High-Level Design)

The Super_Z80 is a single-CPU, multi-chip console, so a modular emulator architecture is ideal. A common pattern is to create a **core system class** (e.g. `Console` or `SuperZ80`) that owns all components – CPU, memory bus, GPU (video), APU (audio), input controllers, cartridge, etc. [1] [2]. This central class acts as the "motherboard," connecting components via an emulated **bus**. In a **bus-driven design**, the CPU does not access devices directly; instead, all reads/writes go through a bus interface that dispatches to RAM, VRAM, or I/O devices based on address ranges [3] [4]. This mirrors the real hardware's data bus and keeps components decoupled. By contrast, a **component-driven design** might give the CPU direct references or callbacks to each device (memory, video, etc.) for speed, at the cost of tighter coupling. For example, the CPU could hold pointers to memory arrays or function hooks for I/O; this is faster but less flexible for changes (MAME favors the former bus abstraction for flexibility, whereas older single-system emulators often used direct calls) [5] [6]. We recommend a bus-centric approach for clarity – the Super_Z80's CPU will interact with a `Bus` module for all memory and I/O operations, which in turn queries the appropriate subsystem.

Module Breakdown: A clean class/module separation makes the project manageable and extensible. At minimum, we suggest:

- **CPU Core:** Emulates the Z80 CPU (registers, instruction execution, cycle counting, interrupt handling). This could be an existing core or custom implementation integrated as a module.
- **Memory/Bus:** Manages the 64KB CPU address space and I/O ports, dispatching reads/writes to the correct memory region or device. This module implements the memory map (cartridge ROM/RAM, Work RAM, VRAM window) [7] [8] and I/O port map [9], and can include the cartridge bank switching logic.
- **PPU/Video:** Emulates the graphics subsystem (tile background planes, sprites, palette RAM, and the mixing priority logic) [10] [11]. Often this can be one class (e.g. `VideoSystem`) with internal helpers for tilemaps and sprites. It will generate the frame buffer (256×192 resolution) [12], handle scrolling registers, sprite evaluation, and generate video interrupts (VBlank, and possibly scanline compare interrupts).
- **APU/Audio:** Emulates the audio subsystem chips – the SN76489-like PSG, the YM2151 FM synth, and the 2-channel PCM DAC [13] [14]. This could be split into subcomponents (one class per chip, plus a mixer) for clarity.
- **Input:** Handles reading the two controller ports and any special inputs. This module would expose the current state of each gamepad's buttons (D-Pad, buttons, Start/Select) and respond to I/O port reads [15].
- **Cartridge/Mapper:** Represents the game cartridge, including the ROM data, optional battery-backed SRAM, and mapper logic. Different mapper types (if any) can be handled via subclassing or conditionals using the header's mapper field [16]. On the Super_Z80, bank switching is done via I/O

ports [17], so the cartridge module should respond to those writes by adjusting ROM bank mappings in the Bus.

- **Timing/Clock:** A small scheduler or timing controller to synchronize the CPU, video scanlines, and audio sample generation. This could be part of the main loop or a separate helper that ensures all components advance in lockstep cycle-accurately.

Emulation Loop Structure: Emulators generally use either an **instruction-step loop** or a **cycle-step loop** [18]. In an instruction-based loop, the CPU executes one instruction at a time (updating an internal cycle counter), and then the other components catch up to those cycles. This is simpler and was common in older emulators, but can falter if multiple hardware events occur mid-instruction. A cycle-based loop steps the smallest time unit (e.g. one clock cycle or microtick) and updates all components incrementally, which is needed for true cycle accuracy [19] [20]. For the Super_Z80, cycle accuracy is a goal, so we recommend a hybrid approach: use an instruction-step CPU core that can report timing (t-states), but insert synchronization points for critical events (scanline start/end, interrupts). Alternatively, implement a fully cycle-stepped CPU so that each clock tick is emulated in turn [21]. The cycle-stepped design is more complex but allows a straightforward whole-system timeline – every cycle, you tick the CPU, GPU, and APU as needed. This guarantees precise ordering (for example, an interrupt firing at a specific cycle will be handled correctly relative to the video beam). If using an instruction-step core, you can simulate sub-instruction timing by invoking callbacks or split the instruction execution internally into micro-operations [22] [23]. The choice depends on complexity vs. need: since Super_Z80 developers might leverage exact raster timing (due to scanline IRQs and mid-frame effects [24] [25]), a cycle-level loop is justified.

Accuracy vs. Simplicity vs. Performance: There is an inherent trade-off in emulator design [26] [27]. A **cycle-accurate** approach means every component's timing is faithfully reproduced – crucial for older 8-bit systems where games rely on exact clock timing [27]. This comes at a performance cost, as more operations are emulated (often the CPU must be stepped hundreds of thousands of times per frame). A simpler line-or frame-based approach can run faster but may miss timing nuances (e.g. a game doing mid-screen splits or precise audio sample timing could break). For Super_Z80, which is a high-end 1980s console with advanced features, we prioritize accuracy. However, we can still optimize: for example, run the CPU cycle-by-cycle but batch some work in the video system (render one scanline at a time rather than pixel-by-pixel) to balance performance. Many successful emulators (e.g. **Genesis Plus GX**) find a middle ground, achieving perfect sync for audio and video without literally simulating every single sub-cycle event, using careful scheduling and cycle counting instead [22]. We will follow that philosophy: maintain precise counters for CPU t-states and frame timing, but use higher-level operations where safe. The result is a design that ensures correctness (no dropped interrupts or misordered events) while avoiding unnecessary overhead. In summary, our architecture will mirror the real Super_Z80's layout – a CPU driving a bus of components – and our main loop will iterate in small time quanta (likely per scanline or smaller) to coordinate CPU, PPU, and APU work before moving to the next interval [4]. This sets the stage for cycle-accurate behavior, as each hardware unit advances in sync.

2. Z80 CPU Emulation Requirements

Cycle Accuracy for the Z80: To be cycle-accurate, the emulator must reproduce not just the results of each Z80 instruction, but also *when* those results occur relative to the system clock [27]. The Z80, like many 8-bit CPUs, has multi-cycle instructions and specific timing per machine cycle (fetch, memory read/write, I/O, refresh, etc.). In practice, cycle accuracy means if an instruction takes 10 T-states on real hardware, our emulation will advance the internal clock by exactly 10 cycles and coordinate any external events during

those cycles (for instance, an interrupt line going active at a certain T-state). Achieving this can be done either by a true cycle-by-cycle simulation of the CPU's internal micro-operations, or by using an instruction-level emulator that tracks cycle counts and inserts wait states or checks at the correct points [22]. The Z80's interrupt enable flip-flop delays (EI takes effect after the next instruction), memory refresh cycles, and bus request/acknowledge behavior are all aspects that a cycle-accurate emulation must handle. Fortunately, the Super_Z80's design (single 5.369 MHz Z80H [28]) doesn't include exotic wait-state hardware except perhaps for VRAM access, so standard Z80 timing will suffice. We'll ensure our CPU core counts every T-state and can report or trigger events on exact cycle boundaries.

Interrupt Behavior (IM 1 and Sources): The Super_Z80 uses Z80 Interrupt Mode 1 exclusively [29]. In IM 1, any interrupt causes the CPU to execute a fixed routine at address 0x0038 (essentially an implicit RST 38h) [30]. Our emulator must implement this by detecting when an interrupt should occur and vectoring the PC to 0x0038, after properly pushing the current PC onto the stack. The hardware has multiple interrupt sources: the primary **VBlank interrupt** (60 Hz, at end of frame) [24], a **programmable timer interrupt** for game timing, and an **optional scanline compare or sprite overflow interrupt** [31]. All these sources ultimately feed the Z80's single /INT line (wired-OR). We will emulate a **interrupt controller** simply by OR-ing these signals and driving a single interrupt request to the CPU. When the Z80 is about to finish an instruction and interrupts are enabled (IFF1=1), and if any source is requesting, we'll trigger the IM 1 sequence. This involves: setting the PC to 0x0038, resetting IFF1 (and IFF2) to 0 (interrupts off), and taking the appropriate cycles (13 T-states for Z80 interrupt acknowledge and jump in IM 1) to service it [32]. We should also record which source caused it, since the handler will likely query a status register (e.g. a VBlank flag or timer flag) to determine the cause [31]. The timer and scanline IRQ likely have acknowledgement bits in the I/O port range (0x80–0x9F for timer/system) [33] that the game will write to in order to clear the interrupt. Our emulation must implement those registers so that the interrupt doesn't continuously reassert. Special cases: The Z80 will ignore new interrupts while one is being serviced until an `EI` re-enables them, and the **EI instruction delay** (interrupts enabled one instruction after EI) must be respected. Also, if multiple sources fire in the same very short time, only one IM 1 vector occurs (the game's handler may poll all sources). Cycle accuracy here means generating the interrupt at the correct cycle (for example, exactly at the cycle after the last visible scanline for VBlank, or at a specific T-state in a scanline for the compare IRQ).

Memory and I/O Access Timing: The Z80's bus operations have defined timings (memory read/write typically 3 or 4 T-states per byte, I/O operations 4 T-states, etc.). Our emulator should at least count these cycles appropriately, which most Z80 cores do. A subtler aspect is **memory wait states or contention**. The Super_Z80 spec doesn't explicitly mention wait states, but consider VRAM: if the video hardware is reading VRAM to render and the CPU tries to read/write VRAM (0x8000–0xBFFF), the hardware might impose wait states or restrict access to avoid conflict (similar to how some systems time-share VRAM). We might implement this by simply disallowing VRAM writes outside VBlank except via DMA, per the spec's suggestion that VRAM updates happen during VBlank DMA [34]. If we do allow CPU VRAM access during active display, we could simulate a slight slowdown (e.g. adding a few cycles of wait) to mimic arbitration. Another timing consideration: **I/O port access**. The Z80 signals /IORQ and uses specific cycles for IN/OUT instructions. In emulation, beyond incrementing the cycle count correctly, we must ensure I/O writes happen at the right time relative to the CPU cycle. For example, writing to the audio chip ports will effectively latch data in hardware – if extremely fine effects depended on exact cycle (unlikely here), we'd account for it. Overall, memory and I/O timing for Super_Z80 should mirror a standard 5.369 MHz Z80 with no additional wait states except possibly for VRAM and maybe cartridge ROM (if a slow ROM, but typically consoles use fast ROM; we'll assume no wait).

Choosing a Z80 Core (Accuracy, Integration, License): We have the option to write a custom Z80 emulation or integrate an open-source core. Given the complexity of full Z80 correctness (including all undocumented instructions and flags) and our cycle accuracy goal, leveraging a proven core is wise. Some notable Z80 cores:

- **Zilog Z80 by Marat Fayzullin** – historically widely used, very fast, but it was not cycle-accurate by default and had non-permissive licensing in some cases. (MEKA used a modified Fayzullin core [35], but that code may not be MIT/BSD).
- **CZ80** – a lightweight core used in some emulators (not fully cycle accurate, but very fast).
- **MAME's Z80** – highly accurate, passes exhaustive tests, and MAME's recent licensing is BSD-3 for most cores. MAME's core might be slightly heavy and needs adapting out of the MAME framework, but it is cycle-exact and includes all quirks.
- **Z80Emu (by @redcode/Z80)** – an ANSI C core claiming full accuracy including all undocumented behaviors [36] [37]. It uses an instruction-level approach but maintains a *T-state counter* for precision [22]. This core is LGPL-3.0 licensed, which might not meet our MIT/BSD preference, but it is well-documented and passes test suites [23].
- **FUSE emulator core** (GPL) or **Z80Ex** (LGPL) – very accurate Spectrum emulator cores.
- **Genesis Plus GX's Z80** – Eke-Eke's emulator (GPL) has a Z80 for Master System/Genesis that is quite accurate and could be referenced, though license is GPL.

Considering licenses, an attractive option is to use a core under a permissive license or write a minimalist one that only implements needed features. We aim for MIT/BSD-style licensing, so we might look at the MAME core (if confirmed BSD) or other BSD implementations. Another approach is to adapt a core like the one in **GameBoy (GBZ80)** emulators – though GBZ80 is a subset with differences, not directly usable for a full Z80.

Integration and Customization: Whichever core we choose, integration involves hooking up memory and I/O. Typically, these cores allow registering callback functions for memory reads/writes and I/O port reads/writes. For example, on a memory read at 0x8000, our callback will consult the Bus: if it's VRAM window, fetch from VRAM array (with any bank offset). For an I/O write to 0x60 (audio), the callback will call the Audio subsystem to handle it. We will configure the core with our `Bus` module's interfaces. Also, we need to handle reset and interrupt signals: on reset, set PC to 0x0000 (Super_Z80 has no BIOS, it jumps straight to cartridge entry [38] [39]). On interrupt, our emulator will call the core's interrupt handling routine or set a flag that the core checks each instruction boundary. Because the Super_Z80 has some unique aspects (e.g. only IM 1, a specific clock rate), we might have to adjust the core's cycle timings if it had assumptions (like exactly 3.5 MHz clock for some reference). But generally, a Z80 core is cycle-counted in abstract "T-states," so we just ensure 5.369 MHz corresponds to the cycle count we use in scheduling.

One customization is **interrupt queuing**: since multiple sources funnel into one vector, we may need to decide an order if two happen at the same moment. Likely VBlank and timer won't coincide often (timer could be set to a different rate), but if they did, the game's ISR would still have to poll and differentiate. Our job is to ensure not to drop any – possibly set a status bit for each and have the ISR loop handle them. We also may simulate the **NMI** line if needed (not mentioned in spec, likely unused). Another customization is if the Super_Z80 had any memory paging not typical in other Z80 systems – e.g. the "simple window" for extra Work RAM. We'll have to implement that (maybe a bit flips which 16K half of RAM is at 0xC000). This isn't a CPU core change, but the Bus logic.

In summary, we will likely integrate an open-source Z80 core that **passes tests and supports cycle counting**. An example is the Z80 emulator that "emulates all known behavior and achieves precision down to the T-state level" [36] [22] . Using such a core gives confidence in correctness (it should handle things like the `MEMPTR` internal register, the Q flag, and other quirks) [40] which, while rarely needed for a console, ensure full game compatibility. We'll wrap the core in our class and adapt its memory/I/O hooks to map to Super_Z80's architecture. This modular approach means if a more accurate or faster core comes along, we could swap it with minimal changes to other parts of the emulator.

3. Memory Map & Bus Emulation

Memory Map Overview: The Z80 CPU has a 16-bit address bus (64KB addressable), which the Super_Z80 maps to various physical memories and devices. According to the spec, the logical address map is:

- **0x0000–0x7FFF:** Cartridge ROM (with some portion fixed and the rest bank-switchable) [41] . This is where game program code and data reside, loaded from the cartridge.
- **0x8000–0xBFFF:** Video RAM window [41] . This 16KB region is not a separate physical RAM in the CPU's memory; instead, it provides CPU access to the console's 48KB VRAM in a banked or windowed manner (more on this below).
- **0xC000–0xFFFF:** Work RAM (fixed) [42] . The console has 32KB of work RAM total, with 16KB always mapped here. The other 16KB of RAM is not directly visible unless a mapping register toggles it in (the spec mentions a simple windowing for the remaining RAM) [8] .

Our **Bus emulation** will implement this map. The Bus module maintains references to: a RAM array (32KB), a VRAM array (48KB), and the cartridge ROM data (up to 1MB) plus any cartridge SRAM (8KB) [43] . When the CPU core issues a memory read/write, the Bus will determine which range the address falls into and index into the appropriate memory. For example, an address in 0xC000–0xFFFF indexes into work RAM array (address - 0xC000 offset into that array). An address in 0x0000–0x7FFF goes to the currently mapped cartridge ROM bank(s). The Bus will need to handle bank switching: initially, on reset, Bank 0 of the ROM is at 0x0000 (the reset/entry vector is in the cartridge since there's no BIOS [39]). If the cartridge is larger than 32KB, it will have mapper hardware. On Super_Z80, **banking is controlled via I/O ports 0x00–0x1F** [9] . Likely, a write to a specific port sets which ROM bank appears at 0x4000–0x7FFF (for instance) – this is analogous to how Sega Master System or Game Gear banking works with their memory control registers. The Bus will include variables for current bank numbers and will calculate ROM addresses as `physical_address = bank_number * bank_size + (address MOD bank_size)`. We anticipate a bank size of 16KB (common for Z80 systems) or possibly 8KB. The Cartridge header provides mapper type and ROM size [16] , so our emulator can configure the banking accordingly when a game loads.

VRAM Window and Banked VRAM Access: The Super_Z80's 48KB of video RAM cannot be all mapped at once into the 16KB 0x8000–0xBFFF window, so a scheme is needed to access the full VRAM. The spec implies a "window" – possibly meaning that only a portion of VRAM is accessible at a time. Some potential approaches: - **Fixed Window + DMA:** Perhaps 0x8000–0xBFFF always maps to a fixed 16KB segment of VRAM (say the first 16KB), and the remaining 32KB of VRAM is not directly CPU-accessible except via the DMA engine during VBlank [44] . If so, the CPU could only update those parts of VRAM (e.g. tile maps or certain tiles) in real time, and rely on DMA to transfer bulk data (like new graphics) into the non-accessible VRAM areas during blanking. - **Bank Switchable Window:** Alternatively, an I/O register could let the CPU select which *bank* of VRAM is currently paged into 0x8000–0xBFFF. For example, a simple register with 2 bits could allow switching between three 16KB banks (48KB total) – though 3 banks doesn't align to powers of

two, so possibly a scheme like always map 0x8000–0xBFFF to VRAM 0–16KB, and use the extra RAM window for the remaining main RAM instead. However, the spec explicitly labels 0x8000–0xBFFF as Video RAM, so it's likely always VRAM and perhaps the remaining 16KB of main RAM is mapped via another mechanism (maybe flipping whether 0xC000 maps the lower 16KB or upper 16KB of RAM). Since "no complex multi-page" is noted [17], any bank switching is probably a single bit toggle if at all. - **Our Emulation Plan:** We will implement at least one VRAM bank register (within 0x00–0x1F video registers range) that selects a base offset for the VRAM window. By default, it could be 0, meaning 0x8000 corresponds to VRAM address 0. If the game needs to access beyond the first 16KB, it would write to this register to, say, select VRAM addresses 0x4000–0x7FFF as the window. This is speculative, but implementing it gives flexibility. If later documentation (or actual games) clarify how VRAM is accessed, we adjust accordingly. The Bus will then use this VRAM page offset when servicing 0x8000–0xBFFF accesses.

Regardless of mapping, **VRAM is a separate memory region** and often has different timing. The Bus might enforce that VRAM writes outside of VBlank have no immediate effect if the hardware wouldn't allow it. But since the spec explicitly supports mid-frame palette changes and such [45], VRAM (or at least palette RAM, which is part of the video system) must be somewhat writable during active display. We'll assume normal CPU writes to 0x8000–0xBFFF do modify VRAM (with the considerations of potential wait states). The DMA engine will be emulated by a routine that copies a block from main RAM to VRAM when triggered (and only allow it during VBlank to respect the "VBlank-only" rule) [44]. Our Bus or Video module can execute this copy instantly at the start of VBlank, or simulate a transfer that takes some cycles.

I/O-Mapped vs Memory-Mapped Devices: The Super_Z80 uses the Z80's separate I/O address space for hardware registers (video, audio, input, etc.), rather than mapping these controls into memory addresses. The I/O port map is divided into ranges for each subsystem [33]. For instance, ports 0x00–0x1F are for video control (like background scroll registers, tile bank selects, etc.), 0x40–0x5F for controller inputs, 0x60–0x7F for audio chip registers, and so on. In our emulator, we will implement I/O read/write functions in the Bus. The CPU core, when executing an IN or OUT instruction, will call `bus.ioRead(port)` or `bus.ioWrite(port, value)`. The Bus will then dispatch based on the port number: - If port is in 0x00–0x1F, call the Video system's register write function (e.g. writing to 0x00 might be the ROM bank select register in the memory mapper block [9], 0x01 might be a VRAM page register, 0x05 might be BG scroll X for plane A, etc. – we'll define these in an enum or map in the Video module). - If 0x20–0x3F, route to Sprite/DMA control (likely writing coordinates or initiating DMA). - If 0x40–0x5F, route to Input module (reads from these ports will return controller button states). - If 0x60–0x7F, route to Audio (the YM2151 typically uses two ports: an address port and a data port; the PSG might latch via a single port, etc. We'll implement those details. For example, the Sega PSG often is written by a single 8-bit write that contains channel and volume info). - If 0x80–0x9F, route to System/Timer (writing to timer control registers, acknowledging timer interrupt, etc., and reading might return timer counts or system flags).

Using I/O ports cleanly separates control registers from the memory map, simplifying our memory emulation (we don't need "shadow addresses" for hardware registers in the RAM map). The Bus will thus have a big switch or lookup table for port handlers. This design makes adding or changing register behavior easy – just update the corresponding subsystem's handler.

Address Space vs Physical Devices: In emulation, it's important to keep the notion that the CPU's 64KB address space is an abstraction that mixes several physical memories. Our emulator will not create one giant 64KB array for memory because large parts of that space are handled by different modules. Instead, the Bus is the arbiter. This prevents accidental overlap (e.g. writing to 0x9000 should not accidentally

overwrite something in main RAM just because an array exists – it must go to VRAM or nowhere). We will likely have separate arrays: `uint8_t workRAM[32768]`, `uint8_t videoRAM[49152]`, `uint8_t cartROM[max size]`, `uint8_t cartSRAM[8192]`. The Bus logic then ensures writes land in the correct array (or if the address corresponds to no device, possibly ignore or log it as unmapped).

Also, note that the CPU's address space does **not directly map 1:1** to physical addresses in some cases. For example, if bank switching is in effect, 0x4000 in CPU space might map to 0x8000 in the actual ROM file (if bank 2 is selected). Similarly, if the extra 16KB of RAM can be swapped in, 0x8000–0xBFFF could sometimes map to RAM instead of VRAM – but since the spec didn't explicitly say that, we'll assume VRAM always and the extra RAM might not be directly usable (or perhaps it's a second half that can be swapped into 0xC000–0xFFFF, but they said fixed, so likely not). If needed, a simple toggle could allow 0xC000–0xFFFF to either show RAM bank0 or RAM bank1 (where bank0 is the default 16KB, and bank1 is the other 16KB). Because they say "simplified mapping" and "single-bit selectable" for work RAM [8], we interpret it as possibly a bit in a mapper register that if set, makes 0xC000–0xFFFF point to the alternate 16KB RAM instead. We can implement that easily: have two 16KB arrays for work RAM halves, and choose which one is currently active at 0xC000. But to avoid confusion, we might also just unify them into one 32KB and treat it normally, since the software could always access the second half if needed by toggling – effectively only 16KB can be used at a time though. We'll clarify this with further documentation, but the emulator can support it.

Mapper and Cartridge Abstraction: The emulator should be designed to accommodate different cartridge hardware without massive changes. The simplest mapper is none (32KB or smaller games, where 0x0000–0x7FFF is just the whole ROM). More advanced cartridges (up to 1MB as spec'd [46]) will use bank switching. The **cartridge class** could encapsulate this: for example, have methods `read(addr)` and `write(addr, value)` for addresses in the cartridge range. In a game with no mapper, `read` just returns `rom[addr]`. In a banked game, `read(0x4000)` will use the currently selected bank offset. The bank select registers themselves are mapped to I/O writes, which we handle in Bus by calling something like `cart.setBank(number)` on writes to port X. Alternatively, since the spec places mapper control in ports 0x00–0x1F along with video, the mapper registers could be handled by the Bus or a separate Mapper module listening on those writes. Either way, the logic will ultimately set internal variables that the cart `read` uses. This design allows supporting future special cartridge features (e.g. if a cartridge had an onboard co-processor or something, though not in spec) by subclassing the cartridge and overriding behaviors.

We will also emulate **cartridge SRAM** (battery save RAM) if present [43]. Typically, cartridges map SRAM into part of the ROM address range (often the top end, like 0x6000–0x7FFF on many systems, or via specific banks). The header's "RAM size" and flags tell us if SRAM exists [47]. We'll implement that by, say, if SRAM is present, any write to certain bank region will go to the SRAM array (and reads likewise). We should also only enable SRAM when the game expects it (the mapper type may indicate which addresses map SRAM).

In summary, our memory and bus emulation will recreate the console's memory architecture: a unified Bus interface with knowledge of all address regions, handing off to cartridge, VRAM, RAM, or I/O as appropriate. By keeping this centralized, it's easier to log or debug memory accesses. For example, we can add logs for any unmapped address access or out-of-range bank. This prevents subtle bugs where one might accidentally read wrong memory if things were hardcoded. The approach is similar to how real hardware works (a bus arbiter connecting CPU to chips) [3], and indeed in the Sega Genesis architecture, for instance, you had distinct address ranges for cartridge, work RAM, VDP registers, etc [48] [49] – our bus will emulate the Super_Z80's scheme in the same spirit.

4. Video System Emulation

The Super_Z80's video subsystem is akin to a mid-80s arcade video pipeline, with multiple layers and sprites [10]. We will emulate it at the level of scanlines to achieve cycle accuracy and correct timing for events like VBlank and scanline interrupts.

Tilemaps and Background Planes: There are two background tilemap planes (A and B), each 32×24 tiles in size covering the 256×192 screen [50]. Each tile is 8×8 pixels, 4 bits per pixel (16 colors) [50]. The planes support **fine X/Y scrolling** independently [51], which means our emulator must account for scroll values that are not multiples of tile size (we'll likely have horizontal and vertical scroll registers in the video I/O). Each tile entry also has attributes for **palette selection, priority, and flip (H/V)** [52]. In VRAM, we expect there to be tile pattern data (the bitmap for each tile index) and tilemap data (the grid of indices and attributes for each plane). The spec says VRAM is a shared pool for tile graphics and maps [53], implying the game can allocate VRAM space for tile definitions and for tilemaps as it wishes. We'll need to know from either the cartridge or assumptions how the VRAM is partitioned – many consoles fix, e.g., tile patterns in one area and maps in another. Since it's not explicitly fixed here, we might rely on conventions or possibly video registers that define base addresses of tile patterns or maps. For example, the SNES has registers to point to name tables, etc. If the Super_Z80 leaves it to software, then the game will likely write the tilemap data into VRAM at known addresses. We might deduce addresses by analyzing what the game writes via DMA.

To render backgrounds, our emulator will likely prepare a **frame buffer** array [192 rows × 256 columns] each frame. We can render one scanline at a time as the emulated beam moves. For each scanline, we determine the horizontal and vertical scroll offsets for Plane A and B (these could change at any time, but typically games update them during VBlank or via the scanline IRQ for mid-screen splits). We calculate which tile row and which line within the tile to fetch for that plane. We also consider wrapping (if the tilemap scrolls continuously, we mod the indices appropriately). We then fetch the tile index and attributes for each needed tile from the tilemap in VRAM, and then fetch the pixel data from the pattern table in VRAM (applying bitplane decoding if needed – 4bpp likely stored as bit planes or interleaved bytes). We then write the tile's pixel values to a line buffer. We do this for both planes A and B.

Parallax Scrolling and Priority: With dual planes, games achieve parallax by giving them different scroll values. Our emulator will composite the two planes. The "Priority" attribute per tile indicates how the tile stacks relative to the other plane and sprites [52]. It's common that one plane is designated background and the other foreground; for example, Plane B might be a foreground layer that can go in front of Plane A or even in front of some sprites when priority is set. The spec mentions a dedicated **Priority/Mixer** hardware block [54] which likely handles this logic: maybe each pixel has a priority value (from tile or sprite), and the mixer picks the top pixel. Likely rules: - If a sprite pixel is present and the sprite's priority (or the tile's) dictates it is above the background, it will cover background pixels unless a background tile has higher priority. - If a background tile has priority bit set, it might render in front of sprites or the other BG plane.

We need to infer specifics. Since each tile has a priority flag, perhaps: - If tile priority=1, that tile is considered "foreground" and will appear above the other background plane and possibly above sprites of lower priority. - If tile priority=0, it's normal background behind everything except the other BG if that has priority. And for sprites, we might have a global or per-sprite priority (maybe per-sprite also, not stated but perhaps sprites have a single priority bit too per their attributes [55]). If sprite has a priority bit, it could determine if it goes in front of backgrounds or behind certain layers.

Without exact details, we can follow a common approach: e.g., Sega Genesis allowed sprites either behind or in front of certain BGs. SNES allowed detailed priority. We can implement a simple scheme: draw Plane A and Plane B into separate line buffers with an indication of priority per pixel (like marking if a pixel is from a priority tile). Then draw sprites and decide for each pixel: if a sprite pixel overlaps a background pixel of high priority, maybe the background stays on top (sprite is hidden). If not, sprite covers it. We also have to consider which background is "beneath" the other by default. Possibly plane B is meant as a foreground plane (like parallax in front of plane A's scenery) – the spec calls plane B "Parallax/Foreground" [56]. So likely plane B by default is drawn in front of plane A, unless a plane A tile has its priority bit set to force it in front. That would allow plane A to act as a HUD or front layer for selective tiles. We will assume: - Draw Plane A (background) entirely. - Overlay Plane B; normally it covers Plane A where it has opaque pixels. - If a Plane A tile has priority, we can treat that tile's pixels as if they were on a higher layer – meaning they should *not* be overwritten by Plane B. This effectively inverts their order locally. - Sprites: likely sprites should appear above both planes by default (since they're characters), but if a background tile's priority is set, that tile should appear in front of sprites (e.g. for when a player goes behind a wall). So a sprite pixel is drawn only if either the background tile under it doesn't have high priority, or if the sprite has its own high priority flag that should supersede.

We will refine this with testing, but the emulator can implement such rules. The **48 sprite limit, 16 per scanline** means we must emulate sprite evaluation.

Sprite Evaluation: Super_Z80 can have up to 48 sprites total [57]. Each sprite likely has attributes: X, Y position, tile index, palette, priority, flips [55]. The hardware manages a **sprite list** in VRAM or a special memory (commonly called OAM – Object Attribute Memory). The spec's I/O 0x20–0x3F region is "Sprite system & DMA control" [33], so the CPU likely writes sprite data either directly into a table via these ports or into VRAM which is then used by the Sprite Generator hardware. Many systems (NES, SNES, SMS) have a separate OAM. Possibly here, sprites share VRAM for their tile graphics but have a separate attribute memory. For emulation, we can maintain an array of 48 sprite structs internally, updated whenever the game writes to sprite registers or does a DMA (for example, maybe during VBlank the game triggers a DMA to copy a block of RAM into the sprite attribute table in VRAM). We will simulate that by either directly writing into our sprite list on those operations.

During rendering of a scanline, we must find which sprites are visible on that line. Because hardware does this every line, we will do similarly for accuracy. We iterate over all 48 sprites, and for each, compare the sprite's Y coordinate to the current line. If the line is between `Y` and `Y+height-1` of the sprite, then the sprite is on this line. We then record it along with its X position and tile data. The hardware limit is 16 sprites per scanline [58]. If more than 16 sprites cover a single line, the spec says the limit is "hardware-managed to minimize flicker" [59]. On many real systems, "flicker" is managed by dropping all sprites beyond the limit (usually those with higher index number or to the rightmost). To minimize flicker, some systems have rotating priority (e.g. PC Engine rotates evaluation start each frame to give everyone a chance; NES programmers manually shuffle sprites). The spec hint might imply an automatic rotation or an overflow interrupt so the game can handle it (since they mention a sprite overflow interrupt option [31]). If the overflow interrupt is enabled and more than 16 sprites are on a line, maybe the hardware triggers that IRQ so the game can decide (like rearrange sprites). But in terms of rendering, we will enforce: only the first 16 sprites (in whatever the hardware's priority order) will be drawn. Usually, hardware processes sprites in a fixed order, typically by OAM index (so lower index = higher priority to draw). We will assume sprite 0 is highest priority and the 17th sprite on the line (if any) simply won't be drawn (which on real hardware results in flicker as those missing sprites may appear in other frames when the order changes). To truly

minimize flicker, the hardware *could* change which 16 to draw each frame, but unless specified, we won't emulate any automatic change aside from potential overflow IRQ.

We also must consider sprite **drawing order** when multiple sprites overlap. Typically, sprites with lower index (earlier in list) draw behind those with higher index or vice versa depending on hardware. Many consoles draw in increasing index order so higher index appears on top. We'll assume the simplest: sprite #0 is evaluated first and ends up behind later sprites if they overlap (so the last sprite in the list that meets the line criteria will appear on top if at same pixel). We can adjust if needed.

Scanline vs Full-Frame Rendering: For accuracy, we will adopt a scanline-based approach. That means in the emulator main loop, for each scanline (0 to 261, if total lines ~262 for NTSC), we: 1. Calculate if this line is in the visible area (0–191 likely, since active resolution is 192 lines [12]). 2. If visible, generate the background layer pixels for this line (taking into account any mid-line changes that might have occurred via interrupts just before this line). 3. Evaluate sprites for this line, as described, and then mix sprite pixels into the line. 4. Output the completed scanline to the frame buffer (or directly to the SDL surface/texture). 5. If the current line is the line where a **scanline IRQ** is set (say the game set a register to trigger an interrupt at line N), then at the appropriate cycle (likely at the end of the previous line or beginning of this line) we fire that IRQ. 6. If this line is line 192 (just after the last visible line), that is the start of **VBlank**. At that moment, we trigger the VBlank interrupt [12]. Also, during lines 192–261 (the VBlank period ~70 lines), no new visible pixels are generated; instead the CPU is free to update VRAM via DMA or direct writes without visible effect.

This scanline iteration inherently syncs the CPU with video: we know how many CPU cycles per scanline (we will calculate that from the master clock). The Super_Z80 runs the Z80 at 5.369 MHz and has 60 Hz, 256×192 display [12]. If the total lines per frame is 262 (a common NTSC value), the CPU cycles per frame = 5,369,000 / 60 ≈ 89,483. Each line then corresponds to ~341 T-states (which is plausible, as many systems have ~228 CPU cycles per line at 3.58MHz; at 5.36MHz it would be proportionally more). We can derive exact if needed: possibly the console uses the 21.477MHz/4 for CPU and / multiple for video such that one frame equals a fixed number of CPU ticks. We'll treat 89,341 or 89,342 as needed (it might not divide evenly by 262). For our purposes, we can round to nearest whole cycles per line, and if there's a tiny remainder, distribute it (some frames might be 341 cycles for most, 342 for some – this is fine as long as average matches).

So, at the start of each line in emulation, we allow the CPU to execute for one scanline's worth of cycles (e.g. 341 cycles). Then we pause CPU, render the line with whatever data is currently in VRAM and registers, then proceed. This ensures mid-frame writes that the CPU performed during that line's time are accounted for in subsequent lines. For example, if at line 100 an interrupt occurs and the game changes scroll or palette, when we render line 101 we will use the new values, just as hardware would.

Palette Handling and Mid-Frame Changes: The console supports 128 palette entries (9-bit RGB each) [45]. The game can change these on the fly (mid-frame palette changes are supported) [60]. We will maintain an array for palette (e.g. 128 entries of RGB colors). When the CPU writes to a palette RAM I/O port (or memory-mapped palette, if that's how it works – likely palette RAM is part of the video registers, maybe accessible via port writes), we update the palette entry in our array immediately. If the write happens during active scan (not during VBlank), the hardware would immediately change the color output for any pixels rendered after that moment. Achieving *pixel-perfect* mid-line palette effects is tricky without pixel-by-pixel rendering. However, such effects are relatively advanced; if needed (for example, a gradient sky effect where the palette color for background changes every few scanlines), the usual approach is the game

triggers an IRQ at a certain scanline and then updates the palette for the next lines. Our scanline-based method can accommodate that: we can apply the palette change upon handling the IRQ, and thus when we render the next scanline, it uses the new palette. This will correctly emulate things like fades or color gradient splits at scanline granularity. If a game somehow changed palette mid-scanline (unlikely without special tricks, as Z80 is too slow to race the beam for many changes), we might not catch that exact pixel unless we went to sub-scanline rendering. But given the spec, we expect palette changes to be done on scanline boundaries via the compare IRQ or during hblank periods, which our approach can handle by splitting the render at that point. We will implement palette writes in the video module such that if a palette register is written, it updates the color immediately in the palette array (so any subsequent pixel or line uses the new color).

VBlank, Scanline IRQ, and HUD Splits: At the end of the visible frame (line 192), a VBlank interrupt occurs [12]. The emulator will handle this by queuing the VBlank IRQ to the CPU at the correct cycle. The **VBlank period** (lines 192–261) is when the game will typically update VRAM (via the DMA engine or direct writes) and prepare for the next frame's graphics. We won't render these lines (they're off-screen), but we do still run the CPU during them, and crucially, we execute any scheduled DMA at the start of VBlank. The spec allows a **HUD or splitscreen** via either a scanline interrupt or "window registers" [61]. If window registers exist, they might define a region of the screen where one layer is disabled or has fixed scroll. For example, perhaps they could designate the top 16 lines as a non-scrolling HUD. If that hardware is present, we'd emulate it by checking the current line against window settings and, if in the window, ignore scroll registers for one plane or blank it accordingly. Since it says "or window registers," possibly the hardware includes a feature to define a window inside which one plane is masked or uses alternate settings (some 16-bit consoles had window layering features). Without concrete detail, we'll primarily rely on the scanline IRQ method: The game can set the scanline compare register (maybe in the video I/O) to a line (say 180) and when the beam reaches that line, an IRQ triggers [12]. The game's ISR can then change scroll registers or other video regs on the fly (e.g. set scroll to 0 for the HUD layer, or switch palette sets). Our emulator catches that and thus renders lines 180+ with the new values, achieving the split effect.

SDL2 Rendering Pipeline: We will use SDL2 for outputting the video. The simplest approach is to use an **SDL texture** as a frame buffer. We'll allocate an SDL texture with size 256×192 (the native resolution) and a pixel format that can represent at least 9-bit color. Since 9-bit (3-3-3 bits RGB) isn't standard, we may use 16-bit (like RGB565) or 32-bit (like XRGB8888) to store colors. We can maintain an internal buffer (array of `uint32_t` for 256*192 pixels) where each pixel is in 24-bit RGB form. The palette entries (3-bit per channel) can be expanded to 8-bit per channel by multiplying by 36 or 32 (since 3-bit max 7, mapping 0–7 to 0–255 roughly by 255/7=36.4). For example, palette value R=5,G=7,B=2 would be expanded to R≈5*36, G≈7*36, B≈2*36. We'll precalc a lookup for 0–7 -> 0–255 values to speed this up. Then, as we render each pixel from tile or sprite, we index the palette and produce a 24-bit color in the buffer.

After finishing the frame (all scanlines), we use `SDL_UpdateTexture` to copy our buffer to the texture. Then use SDL's rendering functions to scale it to the window. Because the Super_Z80 is NTSC-only and fixed resolution [62], we may render at 256×192 and then scale by an integer factor to something like 512×384 or larger as desired. SDL2's software renderer can scale nearest-neighbor by default if we set logical size, or we can use SDL_RenderCopy to stretch. If performance is an issue with software scaling, we can consider using an **OpenGL renderer**: SDL2 can create a GL context, and we could then use the texture on a quad. The user requested SDL2 software renderer primarily, with OpenGL as optional, so we will implement using the SDL2 renderer API for simplicity. On modern hardware, even the "software" SDL renderer is often accelerated under the hood, but if truly in software, 256×192 is small so it's fine.

We also might consider **double buffering** the frame or synchronization: Typically, we'll generate one frame's texture, then wait for the correct time (16.67ms per frame) then present it (SDL_RenderPresent). Using `SDL_RenderPresent` with vsync enabled in the renderer creates a stable frame rate at 60Hz if possible. If not using vsync, we might throttle via SDL_Delay.

For potential future improvements (OpenGL path): We could add features like shaders (for CRT effects or scaling filters). But that can be optional and non-essential. One beneficial GL path could be to render the tile/sprite layers separately and do composition on GPU or to implement per-scanline texturing for speed. However, given the low resolution, a CPU render is absolutely fine.

Mid-Frame Timing: Our emulator will ensure that the **video beam position** is emulated in time with the CPU cycles. For example, if we decide that 1 scanline = ~341 CPU cycles, we may subdivide further: perhaps one can treat 1 pixel = ~1.33 CPU cycles (if 256 pixels ~ 341 cycles, meaning CPU runs a bit faster than pixel clock). We don't need to simulate each pixel's exact timing unless doing raster effects (like sprite multiplexing mid-line, which is not needed here since hardware manages sprites). So, per scanline is an adequate quantum. We will align the **VBlank IRQ** to occur at the moment the last visible line is done (so after line 191 rendering). In cycle terms, that's after 192 * 341 cycles from frame start (around 65500 cycles). That interrupt will be queued and when the CPU loop reaches that cycle it will vector to 0x0038. Similarly, a **scanline compare IRQ** at line N will be scheduled for N * 341 cycles from frame start (plus maybe a few cycles offset if it triggers mid-hblank). The CPU core or a scheduler can handle this by counting down cycle counters – e.g., a timer set to that cycle count triggers the interrupt. This ensures video and CPU are synchronized.

By following this approach, we will have effectively emulated the PPU in a cycle-accurate fashion relative to the CPU: each scanline's rendering occurs in the correct envelope of CPU time, and any writes the CPU did during that time only affect later scanlines. The resulting output should match what the real hardware would produce, including effects like smooth scrolling (no tearing because we emulate scanline by scanline), proper sprite flicker if too many sprites, and so on.

5. Audio System Emulation

The Super_Z80 has a rich audio system for an 8-bit console, with three sound chip components: a PSG (Programmable Sound Generator, SN76489-like), an FM synth (YM2151), and a PCM sample player [13] [14]. Our emulator's goal is to recreate the audio output faithfully and keep it synchronized with the rest of the system.

PSG (SN76489-style) Emulation: The PSG provides 3 square wave channels and 1 noise channel [14], very similar to the Texas Instruments SN76489 chip used in Master System and others. Emulating the PSG is relatively straightforward: each tone channel has a frequency tone register (10-bit frequency divider) and a volume register (4-bit volume typically). The noise channel has a shift register producing pseudo-random noise, with a couple of mode bits (white noise or periodic noise, and a frequency that may be fixed or linked to channel 3 divider). We can implement the PSG by simulating its counters each CPU cycle or using a higher-level approach: - **Cycle-by-cycle:** For each master clock tick (if we define one audio tick per CPU cycle or per N CPU cycles), decrement the tone counters; when a counter wraps, toggle that channel's output bit (square wave). Do similar for noise (shift the LFSR when its timer expires). - **Incremental step:** Because doing this at 5.37 million times per second is wasteful, we can instead accumulate time and only update the

PSG state when needed. For example, if we know how many CPU cycles per one toggle of a square wave given the tone period, we can jump in larger steps.

However, a simpler method in practice is to run the PSG at a fixed sample rate (for example, 44,100 Hz) by generating samples via interpolation. But that can cause aliasing if not careful. Many emulator authors choose to oversample the audio chip at a multiple of the audio rate or at the native chip clock then downsample. Given the SN76489 typically runs at the same clock as either the CPU or a fraction of it (on Master System, the PSG is clocked at ~3.58MHz, which is about the NTSC colorburst; on Super_Z80 presumably a similar frequency is fed), we might emulate it at that clock. The spec doesn't give the PSG clock explicitly, but one guess: if the CPU is 21.477MHz/4, maybe the PSG is driven by 21.477MHz/16 or /8. We can define it if needed or glean from a developer doc. For now, we can assume PSG clock ~ the same 5.369MHz (or maybe exactly that if tapped from CPU clock).

We'll implement the PSG by maintaining for each channel: - A frequency register (set via I/O writes, likely port 0x60 or so). - A counter that counts down at the PSG clock rate. - An output flip/flop state for the square waves. - A shift register for noise and a noise period counter. - Volume for each channel.

We can update the PSG outputs in lockstep with CPU cycles for fine accuracy: e.g., every CPU cycle, increment an accumulator. When it accumulates enough to equal one PSG clock tick (if PSG clock is not equal CPU clock), then decrement counters. This is heavy but ensures perfect timing. We can optimize by using the fact that audio doesn't need to be updated at *every* cycle – we can accumulate the waveform and only produce actual mixed samples at the audio output rate. One method: - Use an accumulator that counts CPU cycles. When it exceeds (CPU_cycles_per_audio_sample), output a sample and subtract that amount. CPU_cycles_per_audio_sample = CPU_freq / output_sample_rate. - For the SN76489 channels, we can integrate their output over that interval. But since we want accurate high-frequency behavior (e.g. if a tone flips many times within one output sample, that contributes to the average).

A simpler approach: emulate the PSG in a time-stepped way at a multiple of 44.1kHz. Honestly, given modern CPUs, directly computing 44,100 samples by stepping ~121 CPU cycles per sample (because 5.37MHz/44100 ≈121) is fine. That means for each audio sample, we step through ~121 CPU cycles' worth of PSG toggles. We can do this by looping for 121 cycles: for each, update counters, sum the output, and then divide by 121 to get an average sample value. But rather than do that integrally, we can derive formulas. However, clarity might trump micro-optimizations here, since 44k * 121 = ~5.3 million iterations per second, which is borderline but possibly okay in C++ if optimized (but remember we also are simulating CPU and video with similar complexity). Alternatively, we can decouple PSG simulation from CPU a bit: treat the PSG as running at 5.37MHz, and resample it down. Many emulators do exactly that: generate a high-frequency buffer for audio (like at 5.37MHz/ n) and then filter down.

Given time constraints, we may implement a straightforward approach that is commonly used: - Keep a fractional time accumulator. For each CPU cycle, add e.g. `(audio_rate)` to an accumulator. If it exceeds CPU_freq, then it's time to output one audio sample. Actually invert: for each CPU cycle, accumulate CPU_cycles; when it exceeds CPU_cycles_per_sample, produce a sample. But producing a sample involves computing the PSG output at that moment (which is easy, just check which channels are high/low), and YM2151's and PCM's output at that moment as well. If we do that, effectively we are sample-accurate (not cycle-accurate to less than one sample), which is usually fine because 44100 Hz is far above any audible requirement. But if needed for more fidelity (like catching tiny phase effects), one could oversample (like 88200 or 176400 Hz output, then downsample).

Since the user stresses cycle accuracy and quality, we might lean on open-source cores: - For **YM2151 (OPM)**: This chip is significantly complex to emulate. It's an 8-channel FM synthesizer with 4 operators each, envelopes, LFO, etc [63] . Emulating it from scratch is a major project; it's wiser to integrate an existing core. Known high-quality YM2151 emulators include: - The **MAME YM2151 core** (originally by Jarek Burczynski, improved over years) – very accurate in sound. If under BSD, that's good. - **Nuked OPM (YM2151)** by NukeyKT – which aims for cycle accuracy. NukeyKT's related YM2612 core (YM3438) is known for extreme accuracy, and indeed Genesis Plus GX uses it for top-notch audio [64] . Nuked OPM exists as well [65] , but it's LGPL as we saw (LGPL 2.1). - There might be simpler YM2151 cores with permissive licenses, possibly older ones. Even an older MAME core might be under a permissive license now (since MAME re-licensed, most of it is BSD-3 now, except some contributions). - Another angle: YM2151 was used in Sharp X68000 computer – its emulators might have a core (likely derived from MAME).

We prefer a core that offers a function to write to registers and a function to generate audio samples for a given number of cycles or given output tick. YM2151 runs typically at ~3.58MHz (common in arcade use – e.g., many arcade boards clocked YM2151 at 3.58 or 4 MHz). Let's assume it's clocked by the NTSC colorburst (~3.579545 MHz, exactly 6 * 597.870 kHz). If our CPU is 5.369 MHz, perhaps the YM2151 is indeed 3.579 MHz (maybe derived by dividing the same crystal by 6 instead of 4). In emulation, we need to sync it with CPU: e.g. if CPU runs 5.369 million cycles per second and YM2151 runs 3.579 million cycles per second, their ratio is ~1.5 (which interestingly is the inverse of CPU speed advantage – because CPU is 1.5× faster than a typical 3.58MHz, which matches YM speed). So in 89,483 CPU cycles per frame, YM2151 would have about (3.579/5.369)*89483 ≈ 59655 FM cycles per frame. We can integrate the YM by ticking it appropriate times per CPU cycle or using fractional math similarly.

The YM2151 core likely handles its own internal phase accumulators if you call an update function specifying how many clock cycles or samples have passed. Some cores let you drive them with a clock input per CPU tick. Others let you request audio output for X samples (given an output sample rate and internal state). We could configure the YM2151 core at an output sample rate (like 44100 Hz) and call `ym2151_update(buffer, samples)` once per frame or periodically. Internally it will step through the FM algorithm. This is a simpler integration: e.g. call it once every frame to produce 735 samples (for 60 Hz at 44100 Hz), or call every 1/60th second. However, to avoid audio buffering issues, typically you call such update more frequently (like every few milliseconds). We can tie it to our CPU loop by generating audio in smaller chunks.

So our approach: when the CPU has executed a certain number of cycles such that ~N audio samples worth of time has passed, we call the audio chips to generate those N samples and append to a buffer. We maintain a ring buffer for audio or directly use SDL's callback mechanism (though using SDL's callback with cycle-level sync is tricky; we might prefer to generate in our loop and push to SDL queue or use SDL_QueueAudio).

- **PCM (8-bit, 2 channels):** The PCM is simpler. It has two channels for one-shot sample playback [66] . This presumably works like: the game writes to registers (likely in the audio port range 0x60–0x7F) to trigger a sample on channel 1 or 2. Possibly they specify an address in ROM of the sample and maybe a length or a termination code. The hardware then reads that ROM data (likely 8-bit unsigned PCM) at a fixed playback rate (maybe 8 kHz or 16 kHz) and outputs it, mixing with the other audio. We have to emulate this by:
- Having a buffer or pointer for each PCM channel that, when triggered, starts reading bytes from the cartridge ROM. Perhaps the feature flags in the cart header might indicate presence of PCM samples

and maybe their location or sample rate [47]. If nothing is given, we assume the game knows where its samples are (likely in high ROM and possibly with a lookup table).
- We decide on a playback rate. Possibly it's fixed (maybe 8 kHz, since two channels maybe use modest bandwidth) or maybe programmable (some chips allow setting a rate or a pitch). If it's just trigger-based, likely fixed rate. We can guess 8 kHz for now (common for simple PCM in 80s).
- Once triggered, the emulator will each audio tick fetch the next byte from the ROM address and send it to DAC until some condition: either a length register's worth of bytes have been played, or a terminating zero or a specific stop command. Maybe the game is supposed to write some "end" command after writing the sample? The spec calls it "one-shot" so probably the hardware stops on its own at sample end; it might rely on data format (like first byte could be length or there is an explicit length register as part of trigger).
- For simplicity, we might embed knowledge from similar systems: The NeoGeo (which came later) had PCM with triggers, but those had their own chip (the ADPCM) with data and a stop code. The PC Engine had sample playback via CPU, not auto. The Genesis had PCM via DAC but manual. So this is custom. We might implement it as: when channel is triggered, we store the start address and perhaps the CPU or game will also provide a length. If not, maybe the data itself has a terminator (like 0x00 or a specific sample-end marker).
- We'll assume the header or data has info; but since we can't be sure, perhaps the game writes a length to another register. We might need to scour the spec or infer that from "feature flags – PCM" meaning game knows how to drive it.

Our emulator will maintain for each PCM channel: an active flag, current ROM address, remaining length (if known, or until we hit some default stop). At each audio generation step, if the channel is active, we fetch the byte from ROM, convert it to a signed sample (e.g. if stored as unsigned 0–255, convert to -128–127 or 0 centered float), and output it. Then increment the address and decrement length. If length hits zero or we reach some marker, we mark channel inactive. We also should ensure not to run past ROM end. Possibly the header "PCM samples sourced from cartridge ROM" [67] implies they're just in the ROM data, likely near end or marked.

One more: since the PCM reading is from ROM, on real hardware that might conflict with CPU if CPU also trying to use the bus. But often consoles have separate bus or time-slice for such DMA. They didn't mention a separate sound CPU, so maybe the PCM hardware can read ROM autonomously, likely during VBlank or as needed with arbitration. For emulation, we don't need to slow the CPU for this unless we want to simulate bus conflicts (rare).

Open-Source YM2151 Core Evaluation: We plan to use an existing YM2151 implementation due to complexity. Among open options, **Nuked-OPM** is very accurate but LGPL-2.1 [68]. If license is an issue, another candidate is **YM2151 from MAME** (should be BSD-3 now). The MAME YM2151 has been used in various projects and produces authentic sound (including proper LFO, envelopes, etc.). Integration wise: - We will create a `YM2151Chip` class in our audio module that contains the state of the chip. - When the CPU writes to port (say port 0x60 = YM2151 register address, 0x61 = YM2151 data as common convention), our bus will call `ym2151.writeRegister(addr, value)`. We might need to handle the two-step write (address port then data port). - The YM2151 core will update its internal registers accordingly (for example, writing to a frequency or instrument parameter). - For generating sound, we will periodically call `ym2151.generate(output_buffer, samples)` to get audio samples. Usually, these cores allow stereo output (YM2151 supports stereo panning per channel) [69], so the output is two channels.

We must decide sample rate. SDL by default often uses 48 kHz or 44.1 kHz. We can choose 44.1 kHz for simplicity (common, and divides well by 60Hz for frame syncing: 44100/60 ≈ 735 exactly (since 44100/60 = 735 * 60 = 44100 exactly, so each frame 735 samples)). That's convenient. If we use 48k, each frame is 800 samples (since 48k/60=800 exactly). Either is fine. Let's assume 44.1 kHz (CD quality).

So each frame, we need to output 735 audio samples (stereo). But generating once per frame can introduce a bit of latency if not double buffered properly, and risk drift if our frame scheduling isn't exact. A safer strategy is to generate audio more continuously – for instance, generate some samples every few scanlines or on demand when an audio buffer is empty via SDL callback. If using SDL's push method (`SDL_QueueAudio`), we can fill in chunks. Possibly easiest: use SDL audio callback to decouple audio timing from our loop, but then we must ensure our emulation keeps audio buffer fed and in sync.

To keep things simple, we can tie audio generation to our frame loop: at the end of each emulated frame, generate 735 samples and send to SDL. SDL with vsync might tie to frame rate anyway, but typically audio should run freer. Perhaps better: run the CPU and PPU as fast as needed to maintain real time, and output audio in real time increments. But we already have a good way to throttle (vsync or timers). Possibly just doing per frame is acceptable if using vsync at 60Hz, but if vsync fails or on variable refresh monitors, audio could glitch. A known solution is to use audio as the master timing: continuously produce audio and adjust video frames to match audio clock. But consoles often allow small drift.

For now, we can do: after each frame's emulation, we have 735 new samples – we push them to audio device. The audio device (if running at 44100) will consume them exactly in 1/60 second. If our emulator is a bit slow, the audio buffer might underflow (crackle). If too fast, buffer might overflow (lag). We can monitor this and adjust slight delays. Possibly SDL offers an easy way (the callback would better handle it by pulling samples exactly on time). Actually, better: implement an **SDL audio callback** that calls an `audioCallback` function which in turn runs the emulator forward enough cycles to generate the requested samples. This gets complicated because tying back into the CPU loop from audio callback (which runs on separate thread) is not thread-safe unless we run the entire emulation on the audio thread. Typically, emulators avoid that by generating audio on the main thread in lockstep with emulation and use a buffer.

Given the scope, we likely do the simpler: lock audio and video together at frame granularity. The slight drift should be negligible if using the exact 44100/60 relationship.

Mixing and Output: The final DAC mixes PSG + FM + PCM in stereo [69]. We will emulate mixing by simply summing the waveforms from each source for each channel. The PSG is likely mono (or possibly stereo capable via simple panning as in some later variants, but spec doesn't mention PSG panning so assume mono or fixed stereo). The YM2151 produces stereo (each of its 8 channels can be panned fully left, fully right, or both with varying attenuation). The PCM probably can output stereo if the DAC is stereo – maybe each of the 2 PCM channels is fixed to left or right or has a pan register? Possibly each one goes to a separate speaker (e.g., channel1 to left, channel2 to right) for simple stereo effects. The spec just says "Stereo-capable DAC" [69], implying that yes, stereo output and likely each channel can be panned. If not defined, we might just allow PCM channel 1 = left, 2 = right by convention, or maybe they both output to both channels equally by default. However, "master volume control" also exists [69] – presumably a single register to attenuate overall output.

Our mixer will likely produce 16-bit signed samples for each channel. We must be careful to avoid overflow when summing multiple sources. Typically, each source is scaled such that max combined stays within 16-

bit. For instance, YM2151 and PSG individually can reach full scale. We might either divide each by number of sources (not ideal, changes volume). A better approach is to simulate the analog behavior: the PSG and FM likely feed into a hardware mixer that doesn't clip until DAC. The DAC likely has a certain bit depth (maybe 9-bit per channel DAC to match palette? That would be odd; more likely 9-bit was just video). Possibly a 14-bit DAC or so. But for emulator output at 16-bit, we can assume if all sources at max, it might clip a bit. We could implement a soft clip or just hope games don't max everything at once (which they might not).

We will implement mixing by computing each source sample, summing, then if needed saturating to int16 range. For stereo: The YM2151 core will give us two outputs (left, right). The PSG we will give equal output to both left and right (unless we decide to simulate something like Game Gear stereo but not applicable here). The PCM – if we assign channel1 to left and channel2 to right (just a guess to utilize stereo), then we add channel1's sample to left and channel2's to right. Alternatively, maybe each PCM has its own pan flag, but not mentioned, so we'll do fixed splitting.

Audio Sync Strategies: There are a few ways to synchronize audio with the rest: - **Cycle-stepping audio chips with CPU:** We can tick the YM2151 and PSG on every CPU cycle or every few cycles. For example, every time the CPU advances one cycle, increment YM2151's internal phase counters accordingly and check if an output sample should be computed. This is extremely granular and ensures perfect sync, but is computationally expensive because the YM2151's internal algorithm per cycle is heavy. - **Scheduled tick approach:** Determine how many CPU cycles per one YM2151 clock or PSG clock. For instance, if CPU:FM clock ratio is known (e.g. CPU 5.369MHz, FM 3.579MHz, ratio 1:0.666), we can accumulate error and call YM2151's update function whenever enough cycles have passed to justify generating one sample or one small chunk. Similarly with PSG, since it might run at CPU clock or half, we can tick it accordingly. This is akin to an event-driven approach (e.g., next audio event). - **Fixed timestep (audio as separate):** Run the CPU and PPU for a fixed number of cycles then run the APU for a fixed number of cycles to catch up. Some emulators do e.g. run CPU for 1 scanline, then run audio chips for the equivalent time of 1 scanline (like 1/262 of a second). Because audio is independent aside from register writes, this can work.

We propose a compromise: Use the **per-frame mixing approach** since we identified a neat alignment (735 samples per frame at 60Hz). We will accumulate audio inputs continuously but only finalize and output each frame. Concretely, each time the CPU writes to an audio register (PSG tone, YM register, PCM trigger), we update the state immediately so the core knows of it. Then after emulating all 89k CPU cycles for the frame, we ask the audio chips to produce the audio for that frame's worth of time. If using a high-level core for YM2151, we might call something like `ym2151_generate(frame_buffer_left, frame_buffer_right, 735)` at end of frame. But these cores usually need to internally simulate the chip over those 735 samples (which is 1/60 s * YM clock). They likely integrate fine, as long as internally they know their clock and output rate. For the PSG, since we implement it, we can generate 735 samples by simulating the chip's behavior over the frame. One approach: simulate in small timestep increments (like each sample, simulate ~121 CPU cycles as discussed). Or simulate the waveform at a higher resolution and decimate. Possibly we can simply do the per-sample update: for s in 0..734, do:

```
   let cycles_per_sample = CPU_cycles_per_frame / samples_per_frame = ~89483/735
≈ 121.7
   accumulate fractional cycles; for each sample, run floor(121.7) = 121 PSG
```

```
cycles by toggling counters.
   carry fractional part to next.
   compute output value of channels (1 if square channel output high, 0 if low
for each, etc. sum noise accordingly).
```

The noise channel output is usually either high or low at any time, just like square. We can generate an analog-ish output by summing channel amplitudes with their volume.

Given the complexity, another possible improvement: use existing **SN76489 sound core** implementations. There are many simple ones (Maxim's, etc.) which might be available under MIT. For example, the SMS Power community might have one. But writing one is not too hard either.

Buffering and Drift Prevention: The emulator needs to ensure the audio output neither underflows (running out of samples, causing gaps) nor overflows (too many samples, causing latency). By matching frame samples exactly to 60Hz timing, we should, in theory, neither accumulate drift (assuming the PC's audio device is exactly 44100Hz and we sync to 60Hz properly). Realistically, PC audio clock might not be exactly tied to our frame vsync. Some small differences can accumulate. Many emulators handle this by adjusting a few ppm (parts per million) – e.g., if audio is running slightly faster than video, they might drop or duplicate a sample occasionally, or adjust the frame timing slightly to compensate.

A simpler approach is to use the audio callback: let the audio device request samples, and we generate as needed, always keeping the buffer at a safe fill. Or SDL_QueueAudio with a dynamic buffer (like always keep ~3 frames of audio queued). That way, minor differences are absorbed by the buffer. If the buffer grows too large (meaning our emu is running faster than audio consumption), we can drop a frame's audio; if it gets too small, we can add a small delay. This is advanced; for now we can likely get away with locking to vsync (which effectively ties video and audio together, assuming vsync 60Hz is stable).

We also note the spec's **Master volume control** [69] – likely an I/O register to scale overall output. We can implement that by multiplying all samples by a volume factor (0-100% likely). It's not critical but easy to add.

Finally, to avoid output distortion, we might implement clipping or limiting. If sum of channels goes beyond int16 range, we can clamp. We can also implement a simple dynamic range compression if needed, but likely not necessary if the game's sound team mixed the volumes well (the YM2151 has large dynamic range, PSG can be loud too, but presumably they balanced it).

In summary, our audio emulation will incorporate: - **PSG core:** possibly custom, ticking at sub-sample intervals to produce waveforms. - **YM2151 core:** integrated from open source, updated per frame or per small step. - **PCM player:** custom logic to fetch and play samples from ROM. - A mixing routine to combine these into stereo PCM samples at 44.1kHz. - A mechanism (likely using SDL2 audio) to output these continuously in sync with emulation.

By prioritizing known good cores (YM2151) and accurate techniques (for PSG, SN76489 is well-documented so we can get it cycle-exact easily), we ensure authentic sound. For example, the YM2151's distinctive FM timbres (like those heard in 80s arcade games) will be preserved. Genesis Plus GX's approach of integrating NukeyKT's cycle-accurate FM core is an inspiration [64]; similarly, we could integrate Nuked OPM for ultimate

accuracy, acknowledging its license (LGPL) or seeking permission. Alternatively, MAME's core under BSD would suit perfectly and provide negligible difference in quality.

We should also test audio by running known sound tests (like outputting known tones) to verify that pitch is correct (meaning our frequency ratios and clock assumptions align).

6. Timing & Synchronization

Precise timing is the backbone of a cycle-accurate emulator. We need to synchronize the CPU, video, and audio so that they progress in parallel just as in real hardware, and also ensure the emulator's output aligns with real time (~60 frames per second, ~44100 audio samples per second).

System Clock Model: The Super_Z80's master clock is derived from an NTSC colorburst crystal (21.47727 MHz) [70]. From this: - CPU clock = 21.47727 MHz / 4 = ~5.36932 MHz. - It's likely the video circuits use the same source: NTSC 60 Hz is typically 262 lines * 60 ≈ 15720 Hz horizontal frequency. Often, the pixel clock might be related to that crystal too (for 256 pixels active, etc.). - Audio YM2151 likely at 21.47727 MHz / 6 = ~3.579545 MHz (which is a common YM clock). - PSG could be at 3.579545 MHz or maybe directly CPU clock (some consoles clock SN76489 at 4x the desired frequency and internal divide by 16; if so, maybe here it's 21.477/6=3.579 which is exactly the typical SN76489 clock on SMS).

We can infer that 1 frame (60 Hz) of CPU cycles = 5.369e6/60 ≈ 89488 cycles. Let's calculate more precisely: If 21.47727 MHz/4, and NTSC is actually 59.94 Hz for color, a frame might be 21.47727e6/4/59.94 ≈ 896,250 cycles per frame? That seems too high – might be off. Actually, NTSC VBlank interrupt might be exactly 60.0 Hz if not strictly following NTSC video standard since some consoles simplify to 60. E.g., NES is ~60.1 Hz because they match CPU to PPU.

To avoid confusion, we may treat it as exactly 60 Hz (given spec literally says 60 Hz) [12]. So we'll use 5.369e6/60 = 89488.7 cycles. That's not an integer. We could adjust to 89489 cycles per frame for 59.999 Hz. The tiny difference is negligible for user; or we can alternate 89488 and 89489 cycles per frame to average out. This is common in emulators (e.g., SNES runs 536870912/8 CPU cycles per frame which is not integer per scanline).

Synchronizing CPU and PPU: We adopt the **timeline** of the video beam to orchestrate. Suppose we use 262 total scanlines (just as a guess for NTSC). Then each scanline's duration in CPU cycles would be 89489/262 ≈ 341.56. This suggests maybe 341 or 342 cycles per line, with some lines 341 and some 342 to sum correctly. Possibly the hardware might actually have a fractional cycle thing (like an extra half-cycle every other line). For emulation, a known trick is to have a fractional accumulator for timing. For example:

```
cycles_per_line = total_cpu_cycles_per_frame / total_lines = 89488.7/262.
accumulator += cycles_per_line each line, and round to nearest whole cycles,
carry the fraction.
```

This ensures over a frame you hit the right total.

Alternatively, we could define: - 192 visible lines at 342 cycles each = 192*342 = 65664 cycles - That leaves 70 vblank lines if total 262. 70*342 = 23940. Sum = 89604, which is a bit higher than 89489. If we try 341 cycles for some: - If we do 342 cycles for 192 lines and 341 cycles for 70 lines: 192*342 + 70*341 = 192*342 + 70*341 = 65664 + 23870 = 89534. We need 89489, so still 45 cycles over. If we make 5 of those vblank lines 340 cycles (just thinking hypothetically) to remove 5*1=5 cycles: 89529. We need to drop ~40 cycles more; if we make about 40 lines one cycle shorter (341->340), that's too many lines with 340.

Another approach: Perhaps the actual total lines might not be 262; some consoles had 261 or 263 for certain reasons. If 261 lines: - 89489/261 = 342.07 cycles/line, meaning mostly 342 cycles lines with a few 343 or an occasional difference. If 263 lines: - 89489/263 = 340.3 cycles/line, mostly 340 or 341 cycles.

This level of precision might not be needed if no software times things to sub-scanline except the PPU itself. Usually, in cycle-accurate emulator, you try to stick to exact hardware numbers. If unknown, we choose a consistent scheme like: every line = 342 cycles except one line = different to fix the total, etc. But since we have no specific hardware timing doc, we'll pick something stable like 342 cycles/line for all 262 lines, and accept a slight deviation in frame rate (that gives 342*262 = 89580 cycles per frame, corresponding to 5.369e6/89580 ≈ 59.96 Hz, extremely close to 60). That's probably fine. If needed, we can adjust after comparing to audio sync.

Deterministic Emulation vs Real-Time Pacing: Our emulator core loop will ideally be deterministic (given the same inputs, it produces the same output each run, unaffected by host speed). We will not insert random delays or anything. However, we do need to **pace the emulation to real time** so that audio and video output at correct speed. If we simply run the emulation loop as fast as possible, the game would finish a frame in microseconds and we'd overflow the audio buffer and video would run too fast. To prevent that, we use one of two strategies: - Use SDL_Delay or a high-resolution sleep to throttle frames to ~16.67ms each. - Use vertical sync (if we use `SDL_RenderPresent` on a vsynced renderer, it will block until the screen refresh, effectively syncing to monitor's 60Hz). - Use audio clock: ensure that audio buffer doesn't exceed a certain fill level by pausing if we're too ahead.

Often, using vsync is simplest to lock to 60Hz if the monitor is 60Hz. We might do that and also check audio. Alternatively, some emulators prefer audio master: they generate audio and play it, and they sync video to audio's timing because audio buffer underflow is more noticeable as crackle. We can combine by enabling SDL audio with callback (so it pulls samples at the needed rate) and we just ensure the emulator always stays just a bit ahead in generating audio.

For now, we will likely do: after finishing emulating a frame and rendering it, we call `SDL_Delay` for any remaining time until 16.67ms since last frame. Or rely on vsync which might do similar if triple buffered.

Slow Host Handling: If the host machine cannot run our emulator full-speed (16.67ms per frame), then frames will take longer. This will manifest as the audio buffer underrunning (causing crackle) or the video appearing slow. We have a couple of mitigation options: - **Frame Skipping:** We could detect we're running behind (e.g. frame took > 16.67ms consistently) and then choose to not render some frames (skip drawing every nth frame) to catch up. The game logic still runs, but video appears a bit choppy but faster. This is a common technique to maintain game speed at cost of visuals. - **Audio stretching:** Or we could slow down audio playback (which lowers pitch and is usually not desirable, plus if the game logic is slow, probably better to maintain correct pitch and accept slowdown as a whole). - Realistically, if cycle accurate, a slow host might not handle it well. But since this is not a commercial emulator, we might not implement

elaborate compensation beyond maybe a basic frame skip toggle or an alert to the user that performance is low.

In a debugging scenario (user intentionally running slower or stepping), we might decouple from real time entirely – e.g. run as fast as possible or pause. But for normal play, we assume we want real time.

Fast Host (Un-throttled) and Fast-Forward: On a very fast host, we want to ensure we *do* throttle, otherwise everything runs too quickly. We will definitely have a throttle by default. But we might also provide a "fast-forward" feature where the user can hold a key to temporarily disable throttling and run as fast as possible (useful for skipping cutscenes, etc.). This is easy to do: skip the delay on those frames.

60Hz NTSC Timing Model: We'll program the emulator for NTSC only (as the console is NTSC-only by spec) [71] [62]. That means ~60 frames per second refresh. We should mention that if one wanted to support a theoretical PAL mode (50Hz), that'd involve different clock and line counts, but here not needed. This simplifies things.

Coordinating CPU & Device Clocks: We must ensure that events (like interrupts) happen at the right time relative to CPU execution. For example, the VBlank IRQ should occur at the exact cycle when the last visible line finishes. Because we are stepping CPU in smaller chunks (per scanline), we can schedule "at end of line 191, trigger VBlank IRQ." We will implement that by raising the interrupt flag right after we emulate the last visible scanline's CPU cycles. Similarly, if a timer interrupt is periodic (the spec's Programmable Interval Timer), we have to emulate that timer. Possibly the timer might be clocked by CPU clock or some division. We can simulate a counter that increments each CPU cycle (or every N cycles if it's prescaled) and when it matches a set value, triggers an IRQ (one of the sources at 0x0038). If the timer period is configured via registers (likely in 0x80–0x9F range), we will support that. Perhaps it's like a simple reload timer to generate an interrupt at some kHz for music timing etc.

Determinism vs Real Hardware Race Conditions: In a cycle-accurate emulator, if two events happen in the same cycle, we need a defined ordering just like hardware. For instance, if the CPU writes to a video register at the exact cycle the video hardware triggers an interrupt, which happens first? The hardware design would dictate that (maybe CPU write takes effect, then interrupt next cycle, etc.). We should attempt to mirror that ordering by how we schedule things in the code. Generally, we'll do CPU first in a cycle, and at the end of the cycle, check hardware events. This should be fine. Most interactions in this system are not extremely high-frequency (like no mid-instruction effects except perhaps if VRAM had wait states). As long as we handle interrupt deferral until instruction boundary, which Z80 dictates, we're good.

We will run CPU cycles in a loop and decrement a cycle counter. When cycle counter hits 0 for that scanline, we pause, do video stuff (which might set an interrupt flag or so), then continue.

Real-Time Clock Sync: The Timer & System Services might include a real-time clock or just the interval timer [72]. There's no mention of a battery-backed RTC, so likely just a periodic timer. We should make sure to implement that so games relying on periodic interrupts (like 60Hz or 100Hz for gameplay logic, or audio tick) can use it instead of vblank if needed. Possibly it's an adjustable rate timer which triggers IM1 aside from vblank. We can simulate it by dividing the CPU cycles or using a separate internal counter incremented each cycle.

In conclusion, we will: - Use a fixed 60Hz frame timing with ~89489 CPU cycles per frame (tweaked slightly for exact sync). - Emulate in units of scanlines or smaller to intermix CPU and video events properly. - Throttle the main loop to 60fps using SDL (vsync or delay). - Keep audio in sync by generating exactly the correct number of samples per frame, thus matching the frame time. Because 60Hz * 735 samples = 44100, our audio and video should lock well. If any drift appears (if our frame ends up 16.6ms vs audio's 16.7ms), we can adjust by slight delays. - Provide allowances for slow or fast host: likely just mention to user to lower accuracy if too slow, or skip frames to maintain audio.

Deterministic Execution: For things like replay or testing, our design could allow running unthrottled (for determinism testing or frame stepping) because the actual emulation loop is independent of wall clock except where we insert delay. If we remove those delays, it runs as fast as possible but still cycle-accurate. This is useful for automated testing or debugging.

7. Input Handling

Input on Super_Z80 is simpler than many other parts: two controller ports with standard digital buttons, read via I/O ports [15] . The emulator's job is to capture host input (keyboard/gamepad) and present it to the emulated system as the correct state bits in those I/O ports.

Controller Mapping: Each game controller likely has an 8-bit status register. Commonly, bits might be: Up, Down, Left, Right, Button1, Button2, Button3, Button4 (for the four action buttons), and perhaps Start & Select either multiplexed or on separate bits/ports (since that's more than 8 inputs). The spec lists "Start/Select" in addition to 4 buttons and D-Pad [15] , which is actually 2 + 4 + 4 directions = 10 inputs, which doesn't fit in 8 bits. Possibly they combined Start & Select into the same port by using more bits (maybe each port is actually 2 bytes, or Start/Select are read from a different port address). The I/O map shows 0x40–0x5F for Controller input [33] , that's 32 ports, which is plenty. Perhaps port 0x40 = controller1 lower 8 bits, 0x41 = controller1 additional bits, 0x42 = controller2 lower, 0x43 = controller2 extra, etc. But without exact detail, we can design a plausible scheme or allow mapping configuration. We might assume: - Port 0x40 = Controller 1 bits: bit0=Up,1=Down,2=Left,3=Right,4=Button1,5=Button2,6=Button3,7=Button4. - Port 0x41 = Controller 1 bits continued: bit0=Start, bit1=Select, others unused or 0. - Similarly 0x44 for Controller 2, 0x45 for its Start/Select. Alternatively, maybe Start/Select are on a separate "system" port in 0x80–0x9F area. But spec lumps them under Input, so likely same range.

Our emulator can define it and adjust if needed once software is run (if the game reads a certain port, we'll see).

We will implement an **Input module** that holds the current state of each controller's buttons (as booleans or bitfields). For the host, we use SDL2 events: - If using keyboard: map keys like arrow keys to D-Pad, Z/X/C/V to buttons, Enter to Start, Shift to Select (for example). - If using gamepad: query SDL_GameController or SDL_Joystick for axes and buttons, map them similarly.

SDL provides events for keydown/keyup and controller button down/up. We will update our controller state on those events. For continuous input, either polling or events both work. SDL events are fine as we just set a flag on keydown and unset on keyup.

Since the console polls inputs via I/O reads (no interrupt for button press), the emulator doesn't need to "queue" events or anything: it just needs to report the current state when CPU reads the port. This is straightforward: our Bus's `ioRead(port)` will check if port corresponds to input, and if so, returns the bitfield representing current pressed buttons for that controller.

Polling vs Interrupt Model: As noted, the console is **polled via I/O** [73] . That means the game program will frequently do IN instructions from the controller ports, typically once per frame (in VBlank) or as needed (for example, reading Start in a loop waiting for player to press it). This is in contrast to some systems that use interrupts (like an NMI on coin insert in arcades, or joypad interrupt on some computers). For us, that means we don't need to simulate any periodic input events – just maintain state.

We should ensure that input reads are effectively instantaneous. The only subtlety: if the game toggles an I/O line to latch or something (like NES has a strobe mechanism, but likely not here as it's direct I/O). The spec doesn't mention a serial or parallel protocol, so probably each bit is just wired to a pin. Possibly Start/Select might be multiplexed (on e.g. MSX or something, up/down lines dual-purposed, but unlikely with so many ports free).

Thus, our `ioRead` for controllers will simply return (for example) `controller1_state` for port 0x40, and maybe `controller1_state >> 8` for 0x41 if we split bits, or whatever scheme we decide.

Input Latency: Input latency in emulation can come from: - USB polling of the physical gamepad (if using a gamepad). - Our processing delay (negligible). - The game's polling strategy (they usually poll once per frame, meaning up to 16ms latency naturally). - Graphics output vs input capture – double buffering can add a frame.

We can't change the game's 1-frame inherent latency, but we can minimize emulator-introduced latency. Some tips: - Process input events as close as possible to the time of use. For example, we might poll SDL events at the start of each emulated frame, *before* running that frame's CPU. That way, the just-pressed keys are immediately visible to the game in the next input poll. If we did it after running the frame, the game would only see them one frame later (additional 16ms). - This timing is subtle: ideally, we incorporate all input events that occurred during the previous frame by the time we run the next. So we will poll SDL events at the top of our frame loop. - Use SDL's high-performance game controller API which can be more immediate than waiting for OS if needed (but events are fine). - Also ensure vsync doesn't introduce an extra frame of lag by presenting as soon as logic is done.

Some advanced emulators use "run-ahead" technique to reduce latency (basically simulate an extra frame in advance and then correct the state). That's beyond our scope, but since it was mentioned, we note it. It's complex and not necessary unless we aim to beat original hardware latency.

Mapping Strategies: We should allow the emulator to support keyboard and gamepad interchangeably. SDL2's GameController API allows mapping many controllers to a standardized layout (with buttons named "A","B","X","Y","Start","Back", etc.). We can leverage that so that, say, the console's Button1 corresponds to the "A" button on an Xbox controller and maybe the "Z" key on keyboard. We might choose a default mapping: - D-Pad: Arrow keys or D-pad on controller. - Button1-4: maybe map to gamepad face buttons (A,B,X,Y) or keyboard Z,X,C,V. - Start -> Enter or Start button. - Select -> Right Shift or Back/Select button.

We should also consider analog sticks: typically D-pad is digital, but if a user has a controller with only analog stick, we can treat it as digital by setting a threshold.

Input Processing Frequency: The CPU might poll input multiple times per frame (though usually no need). Our emulator's state is continuously updated with the latest, so that's fine. If the user toggles a button very quickly (within one frame), whether the game detects it might depend if it was pressed at poll time. That's how it is on hardware too (if you press and release between polls, game misses it). Emulation can optionally oversample input (like treat any press within a frame as pressed for whole frame to avoid misses), but that deviates from hardware behavior. We'll keep it authentic: if you somehow tap a key faster than one frame (16ms) such that it doesn't coincide with the game's check, it won't register – same as real (though in real it's unlikely to physically release in <16ms).

Optional Light Gun/Expansion: The spec mentions optional light gun support [73]. Light guns typically require reading an analog value or timing based on video beam. That's complex (like the NES Zapper or Master System Light Phaser). We likely won't implement it unless needed. It might require reading mouse position relative to screen and simulating the gun trigger and sensor. This is an edge case that can be skipped or added later.

Mitigating Latency: For a smoother experience, some emulators allow audio buffering or frame delays adjustments, but since we aim for correctness, we won't do hacks like disabling vsync (that can cause tearing) or sound on/off. The best we do is keep input polling timely and not add additional buffer beyond necessary. The result should be ~1 frame of latency which is baseline.

One thing: if using vsync on a typical double-buffer, the input might have been polled mid-frame, drawn at end, and only shown next vsync, effectively possibly adding 1 frame. Triple buffering or render ahead can also add. For minimal latency, one can use GPU tricks or wait until last moment to poll input before vsync. Hard to do precisely in SDL2's abstraction. But because our frame is short, it likely doesn't matter.

Focus and Pausing: If the user switches window or pauses the emulator, input should freeze. We can detect SDL_WINDOWEVENT_FOCUS_LOST and, for example, pause emulation (to avoid your character running off when window not focused). Or at least stop reading inputs (so last state holds which could be dangerous if a key was pressed).

Multiple Controllers: We should support two players. Our input config will maintain states for controller1 and controller2. If using one keyboard for both – not ideal – likely one uses keyboard and other uses second gamepad or something. SDL can handle multiple controllers by index.

Testing Input: Once implemented, we can test with a simple homebrew that draws pressed buttons or something.

Overall, input handling is relatively straightforward: **read host input events -> update controller state -> on CPU port read, return state bitfield**. This ensures the game reads the buttons as if connected to actual hardware lines.

8. Development Tooling & Debugging

Building an emulator can be greatly aided by including debugging tools. Since our goal is a custom emulator, we should plan features that help during development and could also be exposed to advanced users (like modders or developers writing homebrew for Super_Z80).

Logging & Tracing: At a minimum, adding a logging facility to output debug information is useful. For example, we can have a log of CPU instructions executed (disassembly with register state), which can help verify the CPU core is working (especially when running known test ROMs like ZEXALL for Z80). We might allow toggling this via config (because full trace slows things down). Logging I/O writes is also helpful – we can see which registers the game sets and when, to catch if our interpretations are wrong. For example, if a game writes an unexpected value to a video register, logging it could hint at an unimplemented feature.

CPU Debugger: A proper debugger allows stepping through instructions, setting breakpoints, inspecting registers and memory. We can build a simple text-based debugger console or use an existing UI library. A popular choice is **ImGui** (Dear ImGui) to create an in-application debug UI overlay. It's C++ friendly and can render on SDL's context (if using OpenGL or even software with some tweaking). With ImGui, we can create windows to show CPU registers, a disassembly of code around the current PC, memory hexdump, etc., with interactive controls (step, run, breakpoints). This does add complexity, but the payoff is huge for debugging hardware emulation issues and for developers using the emulator.

Alternatively, we could integrate with **gdbstub** – some emulators implement a gdb remote protocol so you can use GDB as a debugger for the emulated system. But that's overkill for now.

Memory and VRAM Viewer: Visualizing memory contents can help catch issues like improper banking or corruption. A memory viewer can show a hexdump of a range of addresses (with ability to switch to different memory spaces like VRAM vs main RAM vs ROM). We might incorporate this in the ImGui UI or output to a file. Similarly, a VRAM viewer specifically formatted can show the tile patterns or tile map. For example, for tile pattern graphics, since we know they are 4bpp 8x8, we can write code to interpret VRAM bytes and draw a representation of each tile in a grid (like how many emulator debuggers have a "pattern table" view or "CHR viewer"). This lets us see if tiles are loading correctly. We could also highlight which tiles are currently being drawn on screen by cross-referencing tilemap.

Sprite Viewer: We can list all sprite attributes (x, y, tile, palette, etc.) each frame. We could draw a small preview of each sprite tile as well. This helps debug sprite issues (e.g., sprite is not showing because attribute is wrong or if beyond screen).

Background Map Viewer: Similarly, show the background tilemap with scroll applied, maybe as an image.

Register Viewer: List out all important hardware registers (video registers, audio registers) and their current values. This helps ensure our writes are going through and to inspect what the game is trying to do. For example, see current scroll X/Y, current bank selection, etc.

Breakpoints/Watchpoints: Being able to break on certain conditions can save time. E.g., break when PC == some address (if we know a problematic routine) or when an I/O port is written (to catch when a certain register is set). We can implement breakpoints in CPU by checking PC after each instruction or via marking a

lookup table of addresses to break on. Watchpoints (memory read/write break) are trickier but can be done in the bus logic (check if address matches a watch before performing operation).

Step-through Tools: Provide step (execute one CPU instruction), step over (execute one instruction but if it's a call, run the subroutine fully), and step out (run until returning from current call). For Z80, step and step-out are straightforward enough.

GUI Integration: Using **Dear ImGui** is a common choice for emulator UIs because it's lightweight and immediate mode (no heavy scene graph, easy to embed). The MEKA emulator (by Omar Cornut) in fact now uses a custom UI, and interestingly, Omar Cornut is the author of Dear ImGui as well. In the MEKA GitHub README, he mentions MEKA provides "competent debugging and reverse engineering tools" despite old code [74]. That shows the value of such tools for enthusiasts and devs. Many modern emulators (higan/bsnes, Mesen, Dolphin, etc.) have elaborate debugging interfaces – we don't need to match those, but even a basic one helps us ensure correctness.

If ImGui is used, we can overlay it on the SDL window (requires using an OpenGL backend or writing an ImGui SDL renderer for software). Alternatively, open a separate window for debug. Simpler: an optional console mode where pressing a certain key (say backtick) drops into a debugger prompt in the terminal, where one can type commands to disassemble memory or set breakpoints. This is easier to implement (just parse text commands), but less user-friendly than a GUI. However, it requires the emulator to be run from a console to interact.

Logging should be toggleable at runtime or via config. For example, pressing a debug hotkey could increase log verbosity or dump the CPU state. We might log to a file too.

Performance Monitoring: While not directly asked, as developers we might want to measure how many ms per frame each subsystem is taking. We could instrument some high-level timing. If using ImGui, we could present CPU usage percent or host FPS vs target FPS.

Rewind/Rollback: Not requested, but sometimes helpful in debugging to "rewind" to an earlier state if you overshoot an issue. Implementing rewind means saving state periodically, which is heavy but possible. Many emulators support it for user convenience. It's complex and memory heavy, so likely skip.

Hardware Trace Debugging: For deeper issues, sometimes you need to trace hardware signals (like when exactly an interrupt triggers vs when CPU acknowledges). We could add trace logs for interrupts: e.g., log when VBlank IRQ is raised and cleared, when CPU enters an ISR, etc. Also, logging every memory access is possible but huge – maybe only for certain ranges or if we suspect a bus error.

Integration Example (ImGui): We could create a debug overlay that shows: - CPU registers (AF, BC, DE, HL, IX, IY, SP, PC, IFF, IR, etc for Z80). - Perhaps cycle count or current scanline. - Buttons: Step, Continue, Reset, etc. - A disassembly view: we can disassemble a block of memory at PC and highlight current PC. - Memory editor: could reuse ImGui's example memory editor widget (ImGui has examples including a hex editor). - PPU view: a small texture showing the content of VRAM tile patterns or current frame buffer or both. - Sprite list table. - Possibly audio channel levels (like showing what frequency or note each channel is playing, might be overkill but fun).

We should allow toggling the debug UI on/off so normal users not needing it can hide it and not incur any performance cost from rendering it.

Project Structure for Debug: It's wise to separate the core emulation logic from debug GUI logic. Perhaps have a compile-time flag or runtime mode that includes the debugging UI. We might have something like a `Debugger` class that has hooks into the CPU (for breakpoints and step control) and references to memory. This class can implement the features above. If we integrate ImGui, the Debugger would create and manage ImGui windows.

Examples to Follow: The FCEUX (NES emulator) or Mesen (NES) have very comprehensive debuggers with all those features (they even let you view nametables, pattern tables, etc.). For inspiration, we could see how they structure it, though FCEUX is old and uses custom GUI, Mesen uses Qt.

Because this is a research/design document, we don't need to implement, but to plan. We should emphasize how having these tools will prevent "emulator complexity underestimation" pitfalls (like, it's easy to get black screen and have no clue without a debugger – with one, you can see if CPU is stuck, etc.).

Logging and Tracing Example: We might say we could implement logging macros that print to a debug console or file. For instance, after each instruction, we log something like `PC:OPCODE A=xx F=xx BC=xxxx DE=xxxx HL=xxxx SP=xxxx (cycles so far: n)` if a debug flag is on. Or logs for certain events like `VBlank IRQ signaled at line 192` or `DMA transfer 128 bytes from $C000 to VRAM $2000`.

Given unlimited user patience (since they ask for comprehensiveness), we can pack a lot here. We should also mention verifying our emulator against known tests: - Running Z80 instruction test suite (like ZEXALL) to ensure CPU correctness (requires implementing some I/O to output results; maybe easier to run a known ROM and check sum). - If possible, code small test ROMs to check video (like put a pixel at a known VRAM location and see if it appears). - Use reference emulators or hardware if available to cross-check behavior. For instance, if our emulator supports dumping memory, we can compare it to MAME's output at certain times.

In sum, adding these tooling elements will significantly ease development and give our emulator an edge for developers. Emulating the entire Super_Z80 accurately is complex, but with step-by-step debugging, we can fix issues iteratively. This is why projects like MEKA still maintain debug features for Sega 8-bit research [74].

9. Build System & Project Structure

A well-organized project structure and robust build system are crucial for managing an emulator's complexity. We will use **CMake** to handle building the project, as it is cross-platform and can manage optional components (like whether to include the debugger UI, etc.).

Directory Layout: We propose a modular directory layout, reflecting the components:

```
/src
   /cpu        (CPU core implementation, e.g. z80.cpp, z80.h, possibly a subdir
if using external core)
   /video      (Video/PPU implementation, tile and sprite logic, e.g.
video.cpp/.h)
   /audio      (Audio/APU implementation, e.g. audio.cpp/.h and maybe subfiles
for YM2151, PSG, PCM)
   /input      (Input handling, e.g. input.cpp/.h for controller state)
   /cartridge  (Cartridge and mapper logic, e.g. cart.cpp/.h)
   /emu        (High-level emulator coordination: main loop, timing,
synchronization)
   /ui         (Debugging UI, ImGui integration, if enabled)
   /SDL        (Platform-specific SDL interface code, or could be in emu)
```

Alternatively, some may group by functionality:

```
   /core   (all core logic without external dependencies)
   /frontend (SDL2 main, event loop, etc.)
```

But given our list, above breakdown by subsystem is clear.

We should separate **emulator core vs platform-dependent code**. The core (CPU, PPU, APU, etc.) should ideally have no SDL or OS-specific calls. This allows portability (maybe later someone wants to port the core to a web or to another UI). The SDL-specific parts (window management, audio output, reading host input) should be in their own module (perhaps in `main.cpp` or an `SDLPlatform` class). This also allows headless operation possibly (like running the emulator without video for test or using a different front-end).

CMake Configuration: Using CMake, we can have options: - `ENABLE_DEBUGGER` (ON/OFF to include ImGui and debug UI). - `USE_EXTERNAL_Z80` (maybe choose between our built-in vs an external library). - Possibly `ENABLE_TESTS` for including test ROM integration or unit tests.

CMakeLists can find SDL2 (via `find_package(SDL2)`), and find OpenGL and any other libs (like if we include Nuked-OPM as a source, or if we incorporate ImGui as source files directly since ImGui is just compiled as part of project).

We want to ensure portability: target OS likely Windows, Linux, macOS. SDL2 covers these. We should avoid any non-portable system calls in our code.

Third-party Libraries: We have three main potential third-party code: - Z80 CPU core (if we use one) – license matters as discussed. If it's permissive, we can include the .c/.h in our `cpu` directory or as a git submodule. - YM2151 core – similar handling. - Possibly SN76489 core – but that one we might write ourselves (it's small). If not, some available (Maxim's, or the one from Game Music Emu library). - ImGui – it's MIT license, we can include it as source (a few .cpp files) and compile directly. - SDL2 – is external, user must have it installed or we ship it.

We should isolate third-party code in the tree, e.g. `src/thirdparty/` with subfolders. If they are large or we prefer not to merge, we can use submodules or ask user to install (but bundling ensures correct version and simplifies build for newbies).

We also need to ensure third-party licenses are compatible. We specifically aim for MIT/BSD/Zlib for cores, which is good. If we ended up using any LGPL component (like Nuked OPM), that requires dynamic linking to avoid infecting our code or open sourcing our whole project if we static link (LGPL allows dynamic linking usage in closed code though). We can have CMake pick either MAME's BSD YM2151 or Nuked's if license is okay, depending on user preference.

Project Modularization: Possibly build as static libraries for subsystems during development: - e.g., build libcpu.a, libvideo.a, libaudio.a, then link into the final binary. This can speed development if working on one part often. - But static libraries per component might be overkill for a small project – we can just compile all objects into one executable. It's not like multiple link targets are needed at runtime.

However, if we foresee wanting to separate the emulator core as a library (for example, to use in a GUI application or different front-ends), then making the core a static or shared library is useful. For instance, one could build `libsuperz80core.a` with everything except main(), and then have `superz80-sdl` project linking it. This way someone could later build a Qt GUI linking the core, etc., without duplicating logic. This separation also helps license management – if core is LGPL and front-end is not, dynamic linking could isolate them. But we try to keep all permissive to avoid complexity.

CMake and Dependencies: We will use **FetchContent** or git submodules for dependencies if appropriate. For example, we could have CMake fetch ImGui from its repo or just vendor it. SDL2 we assume provided by user's system (or we could include it, but typically not needed as it's available via package managers).

Building for Multiple Platforms: - Windows: We can provide a Visual Studio project via CMake generation or instruct use of vcpkg for SDL. - Linux: Make sure to link with `-lSDL2` etc. - Mac: Similar, plus maybe handle app bundle. CMake covers most. Possibly use C++17 or C++20 for nicer features, making sure all compilers are okay.

Isolating Third-Party Code: Keep their code in separate directory and do not modify it if possible (makes updating easier). If we do modify, document it or apply patches in CMake via patch files.

We should include their license texts in our project distribution to comply with terms. E.g., include MAME's license file if using code from it, ImGui's license, etc., in a `LICENSES` folder.

File Organization: Each module likely has a class or set of functions: - CPU: maybe `Z80CPU` class encapsulating registers and execute step function. Or if using a C core, have a wrapper class. - Bus: a `Bus` or `Memory` class that has methods read8, write8 for memory and in8, out8 for I/O, and holds references to mem arrays and device objects. Possibly one could integrate bus logic in the CPU's memory callbacks, but a Bus class is neat. - Video: possibly one `VideoSystem` class that holds VRAM, palette RAM, tile and sprite logic, and has methods like `renderScanline(int line, uint32_t* framebuffer_line)`. - Audio: perhaps separate classes for each chip (`PSG`, `YM2151Chip`, `PCMPlayer`) and an `AudioSystem` class to combine them and output samples. Or `AudioSystem` directly contains instances of those chip classes. - Input: a class or just static structure for controller states. Possibly a function to update state from SDL

events. - Cartridge: a `Cartridge` class with ROM data and bank regs, and method `loadROM(file)`. Could also incorporate the header parsing (to know mapper type, features like FM present or not – maybe if a game says no FM, we could disable YM2151 to save performance). - Timer: Perhaps part of system or separate if more complex. Might just be handled in an area of the code that on each CPU cycle increments a counter and triggers at match. - Main Emulation Loop: We might have an `Emulator` class that ties everything together. For example:

```
class Emulator {
  Z80CPU cpu;
  VideoSystem video;
  AudioSystem audio;
  InputSystem input;
  Cartridge cart;
  Bus bus;
  // ... other state like cycle counters, current scanline, etc.

  void reset();
  void insertCartridge(Cartridge c);
  void runFrame();
  ...
};
```

The `Emulator` can initialize all subsystems, link the CPU's memory access to the Bus (e.g., by providing static callbacks or passing pointer to Bus into CPU if core supports that), and then in `runFrame()` implement the frame loop: for (line=0; line<total_lines; ++line) { run CPU for cycles_per_line; video.renderLine(line); if (line==vblank_start) trigger IRQ... etc. } And also accumulate audio.

That structure is clean and encapsulated, making it easier to call from main.

CMake Build Types: Provide debug and release builds. Debug with logging, assertions on (like if an invalid memory access happens, break), Release with optimizations. Possibly a RelWithDebInfo to allow debugging a release-speed run.

Testing and Continuous Integration: Could set up some basic automated tests, though emulating full frames is not easily assertable without reference output. But we could include known checksum tests (like run CPU through test ROM and check a memory location matches expected result). This might be too much for now, but is good if one wants to ensure nothing regresses.

Portability: Emphasize using standard C++ and SDL for portability. Avoid platform-specific code; if needed (like file paths, maybe use std::filesystem or SDL_RWops). For threading if used (maybe audio callback runs on separate thread), ensure proper locking if needed for shared data (we might need a mutex around input state if the SDL event thread writes it while CPU thread reads – though SDL events are usually pumped in main thread, so okay).

Build and Packaging: If we want to distribute to users, we'd compile to an executable and bundle with SDL2.dll on Windows, etc. But as a dev project, just instruct how to build from source.

Optional Tools: Possibly have separate targets: - one target for the emulator GUI - one for a command-line disassembler or ROM analyzer if needed - one for running tests.

We can keep it simple and just have one binary that does all (with a UI and a hidden debug console maybe).

Examples of Good Practice: Many emulator projects on GitHub can be references for structure (e.g., SameBoy's structure for multi-platform, GenesisPlusGX for modular code albeit in C, etc.). We should emulate best practices: e.g., document code, ensure not to mix code and data sections incorrectly (some consoles might have alignment needs but probably not here), etc.

In summary, our build system will ensure a smooth compile on all platforms and easy toggling of optional components, making development and maintenance easier. A well-structured codebase with clear separation (CPU vs PPU vs APU, etc.) means new contributors or ourselves later can pinpoint where an issue might be (e.g., sprite glitch likely in video module, etc.). Also isolating third-party code prevents confusion and ease of license compliance [35] (like listing credit to Marat for Z80 core usage as in MEKA's credits, if we used it).

10. Known Pitfalls

Building an emulator, especially a cycle-accurate one, is a complex endeavor often fraught with hidden challenges. Being aware of common pitfalls can save a lot of debugging time:

- **Underestimating Complexity:** Perhaps the biggest pitfall is thinking that writing an emulator is straightforward. Emulating an entire console involves not just the CPU, but all the timing interactions of video, audio, and I/O. Many emulator developers find that a simple implementation that "runs something" is not necessarily accurate. For cycle accuracy, every edge case must be handled: this significantly increases complexity. It's important to plan incremental progress (e.g. get something working roughly, then refine timing) and not give up when things get tricky. Having the debugging tools (as discussed) is crucial because without them, a black screen is an impenetrable problem. The inclusion of debug facilities and careful logging is the remedy to this pitfall.

- **Incomplete CPU Emulation:** If the Z80 core is not fully accurate, games may break in weird ways. For example, not implementing the subtle behavior of the **MEMPTR register** or the **Q flag** might not affect basic games, but some self-checks or copy-protection could fail [40] [23]. Undocumented opcodes or behaviors (like how certain flags are affected by specific operations) are often used in real code (especially in demos or clever programming). Using a well-tested core or thoroughly testing your own against test suites avoids this. Another CPU-related pitfall is not handling the **EI/DI and interrupt enabling delay** properly, which can lead to an interrupt firing one instruction too early or late, causing race conditions in games. For example, on Z80 after an `EI`, an interrupt is not accepted immediately; forgetting this could cause an ISR to run when it shouldn't, possibly crashing the game. Ensuring the CPU respects that (often by setting a one-instruction delay flag) is necessary.

- **Interrupt Timing & Ordering:** Timing of interrupts is a common source of bugs. For instance, the VBlank interrupt should trigger at a very specific point – if it's off by even a few cycles, the game's code that expects it might do things too early or too late. This can manifest as screen tearing (if game updates scroll at the wrong time) or missed input reads. Moreover, if multiple interrupts

coincide (e.g., our optional scanline interrupt at the same moment as VBlank), we must know which is serviced first or if one is masked. Perhaps the design is that scanline IRQ wouldn't trigger on the last line because VBlank supersedes it – our emulator should mirror whatever the hardware would do (likely only one interrupt line so they wouldn't "coincide" per se, one would just be lost or delayed). Not properly masking or prioritizing interrupts can cause unpredictable behavior. Testing with scenarios where two interrupt sources are active is important.

- **Video Rendering Issues:** Tile and sprite rendering can have many pitfalls:

- **Off-by-one Errors in Coordinates:** It's easy to mis-handle the fine scroll offsets or the exact pixel at which a new tile starts. For example, when X scroll is not a multiple of 8, you need to fetch an extra tile for the partially visible column. If you miscalculate, you might see a vertical seam or missing column on scrolling backgrounds. Similarly, for vertical scrolling and wraparound, if you off-by-one the tile index, you show the wrong row. Thoroughly testing scrolling (like scrolling a checkerboard pattern) helps catch this.
- **Sprite Overlap and Priority:** Combining sprite and background layers is tricky. A common mistake is to draw sprites in the wrong order. If we draw sprite #0 on top of sprite #1 when hardware intended the opposite, the result will be incorrect sprite priority. Another mistake could be not implementing the priority flags correctly – e.g., a background tile that should appear in front of a sprite might be drawn behind. This could make certain game elements invisible or flicker incorrectly. Ensuring that our priority handling matches the spec (like how the **Priority/Mixer** block decides pixel output [54]) is crucial. We should test scenarios like a sprite going behind a tree (tree tile with priority bit) to ensure the sprite disappears appropriately.
- **Sprite Clipping & Count Limits:** Many older consoles have quirks like sprites at the edge of the screen may partially show or not at all depending on hardware clipping. We should decide how Super_Z80 handles sprites near boundaries (likely anything outside 0-255 range might be considered off-screen and not drawn). Also, the 16-per-line limit must be exact. If we were to draw 17th sprite, the hardware would typically drop it entirely (not partially). If we accidentally draw it or drop the wrong one (say we drop the first when hardware drops the last or vice versa), visuals differ [59]. The game might even rely on that (for example, some games intentionally draw more sprites to cause flicker and create transparency effects – if we handle differently, it might look wrong).
- **Mid-frame changes:** If a game uses the scanline interrupt to change something like the horizontal scroll mid-frame (split-screen effect), a pitfall is not implementing that timing. For example, on line 100 the game triggers an IRQ and changes scroll for plane B to 0. If our emulator doesn't actually apply it at the correct moment, the bottom half of the screen might not shift as intended. We must ensure that after handling the IRQ (which presumably happens in the few cycles at end of line 99 or start of 100), the rendering of line 100 onward uses the updated scroll. Not doing so could result in static HUDs not working or parallax failing.
- **Palette and Color Issues:** Mistiming or not implementing palette changes can cause wrong colors on scanline effects. Another pitfall is color depth mismatches – e.g., if we mistake how palette index bits map to palette entries. With 128 palette entries (7 bits index, maybe one bit for high/low bank or priority use) [45], if we mis-read a tile's palette selection bits, we could be using wrong colors. A single bit off can make, say, all sprites appear with incorrect palette, which is noticeable (like all characters being green when they should be blue). We need to confirm how 4bpp tile data plus a palette select yields the final 7-bit palette index. Usually it's tile pixel (0-15) plus a palette base from tile attributes. If we off-by-one that base, colors shift by one palette group.

- **DMA Implementation:** If the game relies on VRAM DMA to transfer tiles during VBlank, a pitfall is not emulating it or doing it outside of VBlank. If our emulator mistakenly lets DMA occur mid-frame, a game might work in emulator but on real hardware it would have corrupted graphics (since real hardware wouldn't allow that). Conversely, if we *don't* implement DMA and the game's not writing to VRAM anywhere else, then in emulator nothing would appear (because game assumes DMA did it). So we must implement the DMA engine properly. Also, some games might start a DMA and assume it's done by end of VBlank; if we were to simulate it taking longer or incorrectly lock out CPU (if it was a blocking DMA), it could hang. We plan to simulate DMA as instantaneous at VBlank [44] – that should be fine as long as we don't allow CPU to see partially updated data mid-frame.

- **Audio Synchronization and Quality:** Common issues in audio include:

- **Buffer Underruns/Overruns:** If our audio generation isn't perfectly in step with playback, crackling will result. For example, if we generate too few samples and the audio device runs out, you hear stutter. Or if too many, you get latency. We need to tune the buffering. Many emulators have initial issues here, often fixed by slight resampling or timing adjustments. Using the exact 44100/60 alignment helps, but if we got the frame rate slightly wrong (59.96 vs 60.00) over time a drift would occur. If we detect crackle, we may need to adjust (like stretch a sample every few seconds).
- **Incorrect Channel Implementation:** If we mis-implement the PSG or YM2151, sound will be off. For instance, a common pitfall is the noise channel's pseudo-random generator – if you use a different polynomial than the real SN76489, the noise will sound different (e.g., white noise vs periodic might not match exactly). Also volume levels between chips need to be balanced. If our PSG is way too loud compared to YM2151, one will drown out the other; the real hardware has specific relative gains. We might need to empirically adjust volume scaling for PSG and PCM to mix well with FM (or consult known values from other emulators or measurements).
- **Lack of Audio Filters:** Real hardware often has analog filtering (like low-pass filters) that shape the sound. Emulating that is usually a low priority, but sometimes needed for authenticity (for example, NES's harsh high-frequency harmonics are usually toned down by a filter). For Super_Z80, not specified, but maybe the DAC or output amp had some filtering. Not doing this is not critical, but might result in slightly brighter or noisier sound than real hardware. This is an advanced nuance; initial emulator can skip, but mention it if audiophiles compare outputs.

- **Synchronization of Audio to Video:** If we don't keep audio and video in lockstep, either can lag. E.g., if video runs ahead, you might see an explosion before you hear it or vice versa. A pitfall is to focus on one and neglect the other – we must ensure minimal skew. Our plan to generate audio per frame helps because it ties them, but if frames are not exactly real-time, they could drift from true 44100 Hz by a hair. Frequent resync or slight sample adjustments can solve that if needed.

- **Memory Mapping Mistakes:** If the memory map is handled incorrectly, games will crash or behave wrongly. For example, if we forget to bank-switch the ROM when the game writes to the mapper register, it will keep executing the wrong data once it expects a new bank. We should verify bank switching by observing program flow (the game likely writes to mapper then jumps to an address >32KB expecting new code there). If our mapping is off by one (like if banks are 16KB but we assumed 8KB, addresses will be wrong). Also, the "remaining Work RAM window" – if not implemented but a game tries to use that extra 16KB, it might overwrite VRAM window by accident in our emu, causing weird issues. We should either implement it or if we choose to always map full 32KB (if that's what hardware effectively does), confirm that's correct. Similarly for VRAM mapping – if a game can switch which 16KB of VRAM is accessible and we don't do it, the game might try to

write to VRAM addresses that we silently drop or put in wrong place, resulting in missing graphics. We must pay attention to any I/O writes that look like they might affect VRAM mapping and incorporate them.

- **Cartridge/Mapper Pitfalls:** Some potential issues:

- If different games use different mapper types, and we haven't implemented them, those games won't work. For instance, maybe some games have larger ROM and use bankswitch at 0x4000, others might bankswitch multiple segments or have SRAM enabled at some address. If we only implement a basic mapper, certain titles might fail (e.g., not saving to SRAM). It's wise to at least implement the standard banking and SRAM. Perhaps all Super_Z80 carts use the same basic mapper (since spec indicates unified approach), but if a "mapper type" field exists in header [75], there might be a few (like maybe one for extra SRAM, one for different bank sizes, etc.). We should handle at least those anticipated.

- Another pitfall is not resetting the mapper state on soft reset. If a game resets (calls the reset vector without power cycle), some systems keep bank state, others reset to bank0. Spec says reset maps bank0 at 0x0000 [76], so we must ensure to do that on resets (both power-on and if the game's code triggers a reset vector). If we miss that, a soft reset (common in games via start menu) might jump to 0x0000 expecting the startup code in bank0 but our emulator left it on another bank, causing a crash. This is subtle but important.

- **Emulator Performance Pitfalls:** Accuracy often comes at the cost of speed. Pitfalls here include:

- If we simulate everything cycle-by-cycle in the most naive way, the emulator might run too slowly on typical hardware (especially the YM2151, which is heavy to do at 3.5MHz * every operator).
- We should be careful with our loops, avoid too much overhead per cycle. Using efficient data structures (e.g., using arrays and indices rather than costly map lookups for memory dispatch, because that happens millions of times).
- If we integrate debug logging but forget to disable it in release, performance will tank due to I/O overhead of printing millions of lines. Always guard debug prints with conditions or compile-out in release.

- Certain tasks could be done in bulk instead of per cycle. For instance, instead of checking 48 sprites * 192 lines = 9216 checks every frame (which is actually fine), be mindful not to do something like scanning all sprites for each pixel which would be far more (48*49152 pixels potentially). We choose algorithms smartly (we did line-based which is good).

- **Testing and Validation Pitfalls:** Without real hardware or a reference, it's possible to think something is working when it's subtly off. One should test with as many sources as possible: known test ROMs (for CPU), and compare output of our emulator on some games with known emulator or videos of real hardware. For instance, if on a test game an effect looks slightly wrong, that hints at a subtle timing bug we might not notice otherwise.

By anticipating these pitfalls: - We will use testing tools (like the Z80 test suite, graphical tests if available). - Keep the design flexible to adjust (like if we find timing off, we can easily tweak cycles per line, etc., thanks to centralized definitions). - Acknowledge that initial versions may have issues and iterative refinement is normal.

In conclusion, building a cycle-accurate Super_Z80 emulator is a challenging project that demands attention to myriad details. By adhering to a good architecture, using proven cores, leveraging debugging tools, and learning from prior art (MAME, Genesis Plus GX, MEKA) [35], we greatly improve our chances of success. Each component needs careful implementation and lots of testing. There will be times when games don't work and we have to sleuth out why – whether it's an unimplemented feature or a one-cycle timing error. With the structured approach and awareness of the common failure points outlined above, we can methodically address issues and move towards an emulator that is both accurate and robust, bringing the hypothetical Super_Z80 console to life in software.

Sources:

- Super_Z80 Hardware Specification [28] [57] [14]
- Lukasz Zapart's notes on emulation accuracy and main loops [4] [27]
- David Tyler's discussion of instruction vs cycle stepping [18]
- Z80 CPU core documentation (redcode's Z80) on instruction granularity vs T-state precision [22]
- Reddit commentary on Genesis Plus GX audio accuracy (using Nukeykt's core) [64]
- MEKA emulator documentation highlighting debugging tool importance [74].

---

[1] [2] General emulator component architecture and design questions : r/EmuDev
https://www.reddit.com/r/EmuDev/comments/8fd5g1/general_emulator_component_architecture_and/

[3] [4] [26] [27] [48] [49] Emulating video game consoles
https://lukezapart.com/emulating-video-game-consoles

[5] [6] CPU emulator organization - The Rust Programming Language Forum
https://users.rust-lang.org/t/cpu-emulator-organization/121204

[7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [24] [25] [28] [29] [31] [33] [34] [38] [39] [41] [42] [43] [44] [45] [46] [47] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [66] [67] [69] [70] [71] [72] [73] [75] [76]
super_z280_hardware_specification.md
file://file_0000000000b871f895fed07d7d97ed17

[18] [19] Bringing emulation into the 21st century - David Tyler's Blog
https://blog.davetcode.co.uk/post/21st-century-emulator/

[20] [21] A new cycle-stepped Z80 emulator
https://floooh.github.io/2021/12/17/cycle-stepped-z80.html

[22] [23] [36] [37] [40] GitHub - redcode/Z80: Highly portable Zilog Z80 CPU emulator written in ANSI C
https://github.com/redcode/Z80

[30] Interrupts - SpecNext Wiki
https://wiki.specnext.dev/Interrupts

[32] Interrupt Behaviour of the Z80 CPU
http://www.z80.info/interrup.htm

[35] [74] GitHub - ocornut/meka: MEKA - Sega 8-bit emulator with debugging/hacking tools
https://github.com/ocornut/meka

64 Question about 22050Hz and 44100Hz in Sega Genesis emulator ...
https://www.reddit.com/r/retrogaming/comments/k4rnj6/question_about_22050hz_and_44100hz_in_sega/

65 68 nukeykt/Nuked-OPM: Cycle accurate Yamaha YM2151 ... - GitHub
https://github.com/nukeykt/Nuked-OPM