CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Sisyphus: A Flash Cartridge Prototype for the Nintendo Entertainment System Using FPGA-Based Hardware Emulation

Bachelor's thesis in Computer Science and Engineering

JOWAN ABDOU
ALI AMINI MOGHADDAM
DANIEL GÄLLBLAD
LUCAZ LINDGREN
ELLIOT MARKSTRÖM STENHAMMAR
ALBIN TÄNG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden, 2026

Bachelor's thesis, 2026

Sisyphus: A Flash Cartridge Prototype for the Nintendo Entertainment System Using FPGA-Based Hardware Emulation

JOWAN ABDOU
ALI AMINI MOGHADDAM
DANIEL GÄLLBLAD
LUCAZ LINDGREN
ELLIOT MARKSTRÖM STENHAMMAR
ALBIN TÄNG

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden, 2026

Sisyphus: A Flash Cartridge Prototype for the Nintendo Entertainment System Using FPGA-Based Hardware Emulation

JOWAN ABDOU, ALI AMINI MOGHADDAM, DANIEL GÄLLBLAD, LUCAZ LINDGREN, ELLIOT MARKSTRÖM STENHAMMAR, ALBIN TÄNG

© JOWAN ABDOU, ALI AMINI MOGHADDAM, DANIEL GÄLLBLAD, LUCAZ LINDGREN, ELLIOT MARKSTRÖM STENHAMMAR, ALBIN TÄNG, 2026.

Supervisor: Konstantinos Ioannis Sotiropoulos Pesiridis
Co-examiner: Roger Johansson
Examiner: Arne Linde

Bachelor's thesis, 2026
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg, Sweden
Telephone +46 31 772 1000

Typeset in Typst
Gothenburg, Sweden, 2026

Sisyphus: A Flash Cartridge Prototype for the Nintendo Entertainment System Using FPGA-Based Hardware Emulation

JOWAN ABDOU, ALI AMINI MOGHADDAM, DANIEL GÄLLBLAD, LUCAZ LINDGREN, ELLIOT MARKSTRÖM STENHAMMAR, ALBIN TÄNG
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis presents Sisyphus, a flash cartridge prototype for the Nintendo Entertainment System (NES) that utilizes FPGA-based hardware emulation to replace traditional game cartridges.

The system architecture employs a modular hardware design utilizing Dynamic Function eXchange (DFX) on the Diligent ZedBoard—a popular development board. This allows for runtime reconfiguration to support different memory mappers, with initial implementations focusing on NROM, UxROM, and MMC1 to provide foundational coverage of the NES game library. To ensure operational stability across asynchronous clock domains, a custom synchronization pipeline was developed to mitigate signal disturbances inherent in the prototype's physical assembly.

Results demonstrate a functional end-to-end pipeline, successfully booting, configuring, and executing NROM-based titles such as Super Mario Bros., Ice Climber, and Donkey Kong on PAL-region consoles. While the prototype exhibits graphical artifacts and significant stability issues caused by the physical assembly—with most tested titles consistently freezing or crashing—it establishes a solid architectural foundation for further research and development. Due to a hardware fault in the debug card used, UxROM and MMC1 mappers were implemented, but could not be validated on hardware.

Keywords: NES Flash Cartridge, Embedded Development, FPGA-Based Emulation, Diligent ZedBoard

Sammanfattning

Denna avhandling presenterar Sisyphus, en prototyp av en flashkassett för Nintendo Entertainment System (NES) som utnyttjar FPGA-baserad hårdvaruemulering för att ersätta traditionella spelkassetter.

Systemarkitekturen använder en modulär hårdvarudesign med Dynamic Function eXchange (DFX) på en Diligent ZedBoard—ett populärt utvecklingskort, vilket möjliggör omkonfigurering under körning för att stödja olika mappar, där de initiala implementationerna fokuserar på NROM, UxROM och MMC1 för att ge en grundläggande täckning av NES-spelbiblioteket. För att säkerställa driftsstabilitet över asynkrona klockdomäner utvecklades en anpassad synkroniseringspipeline för att mildra de signalstörningar som är inneboende i prototypens fysiska konstruktion.

Resultaten påvisar en fungerande helhetspipeline som framgångsrikt kan starta, konfigurera och köra NROM-baserade spel som Super Mario Bros., Ice Climber och Donkey Kong på konsoler i PAL-regionen. Prototypen uppvisar grafiska artefakter och betydande stabilitetsproblem orsakade av den fysiska konstruktionen—de flesta testade spel fryser eller kraschar konsekvent—men etablerar en solid arkitektonisk grund för vidare forskning och utveckling. På grund av ett hårdvarufel i det debugkort som användes kunde UxROM- och MMC1-mappar implementeras men inte valideras i hårdvara.

Preface

As this project began, though we did not have extensive experience with the Nintendo Entertainment System, hardware design, or embedded development, we strived for a goal seemingly within our grasp. Although our naivety quickly became apparent, we struggled onward with persistence and stubbornness. During this Sisyphean task, one must remember the wise words of Albert Camus:

"The struggle itself towards the heights is enough to fill a man's heart. One must imagine Sisyphus happy."

This project was made possible by the tremendous effort of the NESdev community—a group of dedicated individuals who together have spent over 30 years meticulously reverse engineering and documenting the NES hardware. All accumulated knowledge is compiled on the NESdev wiki, the most comprehensive resource in the field. However, as a community-driven wiki, it is inherently subject to change and occasionally relies on informal forum discussions rather than official documents or patents. Lacking other alternatives, this report uses the information as-is, with the understanding that it represents the current consensus of the research community. For their invaluable work, our thanks go to the NESdev community.

We would also like to extend our sincere gratitude to our supervisor, Konstantinos Ioannis Sotiropoulos Pesiridis, for his guidance, support, and dedication to assisting us whenever possible. The result of this project would have been markedly different without his counsel.

List of Abbreviations

APU	Audio Processing Unit
AXI	Advanced eXtensible Interface
BRAM	Block Random-Access Memory
CHR	Character
CIC	Checking Integrated Circuit
CIRAM	Console Internal RAM
CMOS	Complementary Metal-Oxide-Semiconductor
CPU	Central Processing Unit
DFX	Dynamic Function eXchange
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
ILA	Integrated Logic Analyzer
IP	Intellectual Property
IRQ	Interrupt Request
LUT	Lookup Table
LVCMOS	Low-Voltage Complementary Metal-Oxide-Semiconductor
NES	Nintendo Entertainment System
NMI	Non-Maskable Interrupts
OAM	Object Attribute Memory
PCAP	Processor Configuration Access Port
PCB	Printed Circuit Board
PL	Programmable Logic
PPU	Picture Processing Unit
PRG	Program
PS	Processing System
RAM	Random-Access Memory
ROM	Read-Only Memory
RP	Reconfigurable Partition
RTL	Register Transfer Level
SoC	System on a Chip
TTL	Transistor-Transistor Logic
VHDL	VHSIC Hardware Description Language

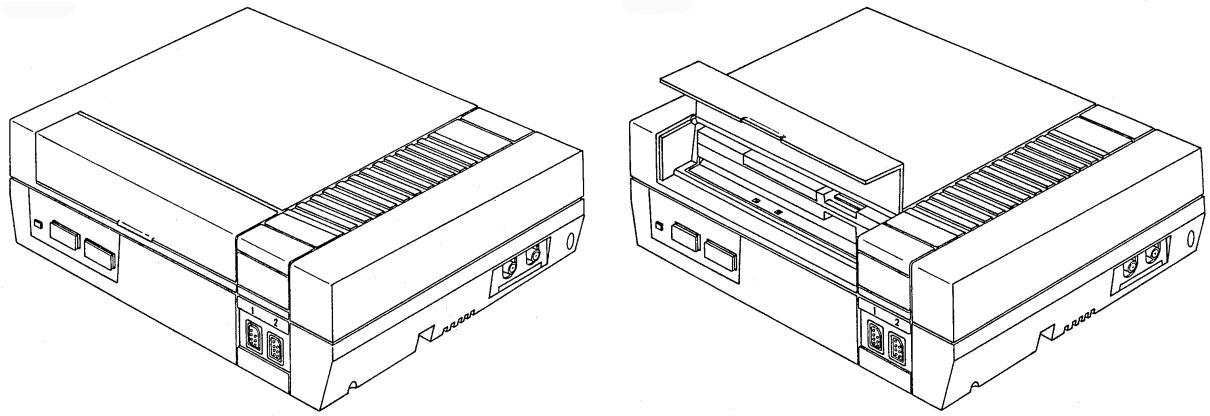
Contents

List of Figures

List of Tables

Chapter 1
Introduction

The *Nintendo Entertainment System* (NES) and its Japanese counterpart, the *Family Computer* (Famicom), represent one of the most significant landmarks in digital media history. At their peak, these consoles were in one out of every three households in both the United States and Japan, yet their legacy transcends mere commercial success. By shaping the industry's technological evolution and elevating gaming into a mainstream cultural staple, their influence remains clearly visible even decades later in gaming, art, music, fashion, and literature [1, p. 5]. All this is contained within an iconic design, seen in Figure 1, which makes it one of the most recognizable consoles of all time.



(a) Front perspective. (b) Front perspective with lid open.

Figure 1: Drawings from the original NES design patent [2].

While the original consoles have long since reached their commercial obsolescence, they did so at a pivotal moment. As software emulation became viable in the mid-1990s, the NES led the way and was able to further expand its reach, transcending the limitations of the hardware they were built upon [1, pp. 7-8].

Even today, there exists a small but dedicated community of people interested in these antiquated machines, preservation of their games, and development of new ones. To serve the needs of these people, products like the EverDrive N8 have been developed that use FPGA-based hardware emulation to emulate game cartridges, allowing the user to store a collection of games on a single SD card and run them on a console [3]. An FPGA (Field-Programmable Gate Array) is an integrated circuit whose internal logic can be configured after manufacture, making it well-suited for replicating the behavior of fixed hardware such as cartridge circuitry. However, these devices are proprietary and therefore don't allow for easy modification and experimentation by users. As such qualities are highly valued by the community, there is still a need for public research that can be built upon to eventually provide alternatives to existing solutions.

1.1 Purpose

The purpose of this project is to assess the feasibility of developing an open-source, hardware-based solution for interacting with the Nintendo Entertainment System library. The goal is the development of Sisyphus, a flash cartridge prototype designed to interface directly with the NES cartridge slot, effectively replacing a standard game cartridge. By utilizing game data loaded from a user-provided SD card, the device will execute software through FPGA-based cartridge emulation on the Diligent ZedBoard —a development board combining an ARM processor with an FPGA—enabling the playback of select titles on original console hardware. By exploring this subject, the project aims to not only determine technical viability but also address the inherent limitations of hardware emulation via programmable logic devices, while ultimately providing an open platform for further public research and collaborative development

In this context, *feasibility* is assessed against three concrete criteria: (1) whether the system can meet the strict bus timing requirements of the NES, responding to CPU and PPU memory accesses with valid ROM data within the hardware's timing windows; (2) whether the FPGA has sufficient logic and on-chip memory capacity to host the emulation logic alongside game data; and (3) whether a reliable electrical interface between the NES cartridge bus and the FPGA development board can be realized.

1.2 Scope and Limitations

The choice of the ZedBoard as the FPGA development board is the foundational hardware constraint, driven by its availability for loan rather than any specific hardware advantage. The board introduced practical limitations—a restricted GPIO count, limited on-chip memory, and a voltage mismatch with the NES cartridge bus—that were mitigated through an FMC breakout board and external level shifters. The resulting architecture is tied to the ZedBoard's pinout and is not currently portable to other platforms, though adaptation to other Zynq-7000-based boards would be feasible.

The project scope is further refined by several deliberate exclusions regarding the original console's architecture and regional variations. Although the NES and Famicom share similar internal logic, the prototype is built exclusively for the NES to avoid complications regarding differences in the cartridge's connectors and loading mechanisms. Furthermore, the scope is limited to the PAL regional standard; while the underlying logic is agnostic, the prototype is tuned to the specific clock frequencies and timing characteristics of European hardware.

Regarding mapper support, the project targets three configurations: NROM (mapper 0), UxROM (mapper 2), and MMC1 (mapper 1). However, only NROM was validated on hardware; UxROM and MMC1 were fully implemented but could not be tested on a physical console due to an unexpected hardware failure.

Chapter 2
Nintendo Entertainment System

As Sisyphus integrates directly into the NES, replacing a standard game cartridge, information about the console's functionality is necessary. This chapter describes the NES hardware behavior as it relates to Sisyphus, omitting nonessential information.

2.1 Overview

The NES is an early 8-bit home console developed and released by Nintendo, built around two purpose-built chips. The Ricoh 2A07, housing the MOS 6502 Central Processing Unit (CPU) and Audio Processing Unit (APU) [4], and the Ricoh 2C07 Picture Processing Unit (PPU). An overview of their interactions within the wider system can be seen in Figure 2.

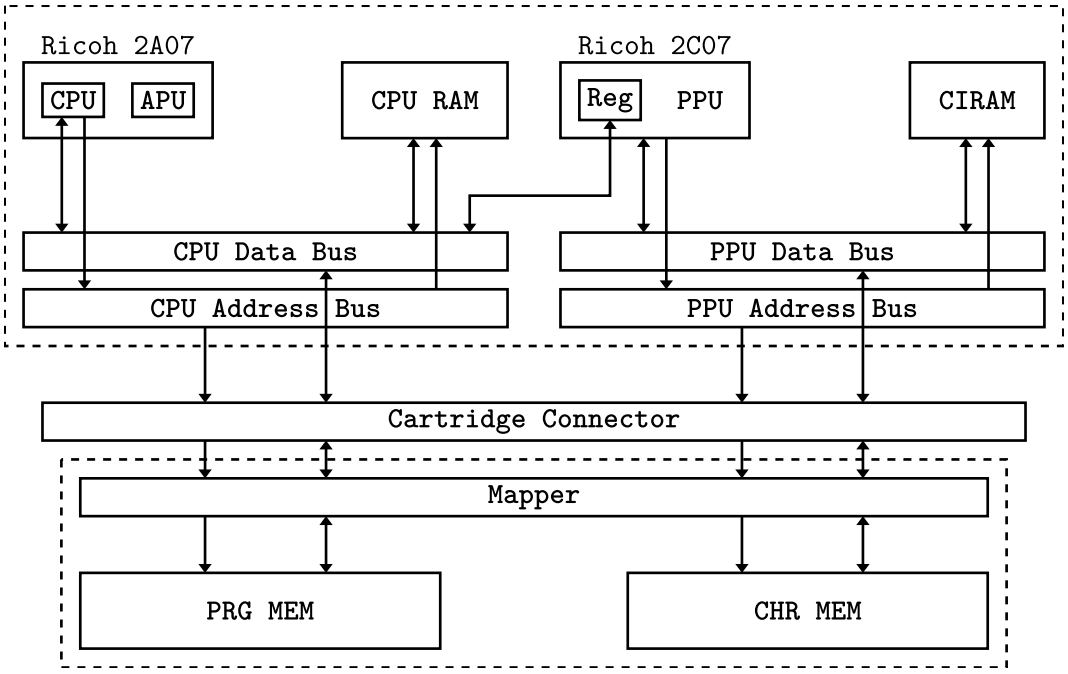


Figure 2: Architectural overview of the NES.

The CPU serves as the system's main processor, responsible for executing game code, handling controller input, and coordinating system-wide operation. The APU, integrated into the same die, produces mono audio output through five independent synthesis channels. The PPU operates as a dedicated graphics processor, autonomously generating a composite video signal to a connected CRT television scanline by scanline. The CPU and PPU operate on separate address and data buses with inter-chip communication achieved through a set of memory-mapped registers that the PPU exposes into the CPU address space.

The CPU and PPU share the same master crystal oscillator but run at different derived frequencies. In the PAL NES, the master clock is 26.6 MHz. The CPU clock is obtained by dividing the master clock by 16, yielding approximately 1.66 MHz, while the PPU clock is derived by dividing by 5, yielding approximately 5.32 MHz. This produces a

ratio of exactly 16/5, or 3.2 PPU cycles per CPU cycle [5]. Because the ratio is not an integer, the NES can power on with multiple different—non-deterministic—CPU-PPU phase alignments, and the two clock domains must be treated as asynchronous [6].

Game software is distributed on ROM cartridges connected via an edge connector that extends both the CPU and PPU buses directly to the cartridge board. This allows the CPU to address program ROM (PRG ROM) as a transparent extension of its own address space, while the PPU addresses character ROM (CHR ROM) on the same cartridge for graphics data. This architecture contrasts with the storage media used in later generations of home consoles, requiring game data to be explicitly loaded into RAM before the CPU can operate on it.

2.2 Central Processing Unit

As mentioned above, the NES is using a 6502-based architecture, a rudimentary 8-bit processor used in multiple systems during the 1980s [7]. A comprehensive pinout diagram is provided in Figure A1, with descriptions of the relevant signals detailed in Table 1.

Name	Direction	On Cartridge Connector	Description
$\overline{\text{RST}}$	Input	No	Initializes a reset sequence ending with the program counter being loaded with the data $FFFC and $FFFD.
A0..14	Output	Yes	The lower 15-bits of address bus used to drive the target address during memory reads and writes.
A15	Output	No	Address line 16.
D7..0	Input/Output	Yes	8-bit data bus used to transfer the data during memory reads and writes.
CLK	Input	No	The master crystal oscillator clock.
M2	Output	Yes	The internal CPU clock (referred to as *Φ2* in 6502 documentation).
$\overline{\text{IRQ}}$	Input	Yes	Used by the cartridge and APU to trigger an interrupt.
$\overline{\text{NMI}}$	Input	No	Used by the PPU to trigger a non-maskable interrupt.
R/$\overline{\text{W}}$	Output	Yes	Used to indicate the memory access operation. The signal is high for a read and low for a write.

Table 1: Ricoh 2A07 partial pinout [8]. Pin directions are given from the perspective of the 2A07. See Figure A1 for the complete pinout diagram.

2.2.1 Instruction Set Architecture

The 6502 uses a small CISC-based instruction set of 56 officially defined instructions. However, as an 8-bit architecture, it theoretically supports 256 possible opcodes, 151 of which constitute the officially defined instructions. The remaining bit patterns are classified as unofficial (sometimes called "illegal") instructions, which arise from the internal logic of the chip's instruction decoder. Most unofficial opcodes are either NO-OP instructions, which consume clock cycles without altering the CPU state, or JAM instructions, which cause the processor to halt execution and drive $FFFF on the address bus until a hardware reset is initiated. Others provide useful combined operations that developers can leverage to optimize performance, which were utilized in a small number of NES games [9], [10].

2.2.2 Interrupts

Control flow is further managed through interrupts, which are classified as either hardware-triggered or software-triggered. Hardware interrupts are initiated by pulling the $\overline{\text{IRQ}}$ (Interrupt Request) or $\overline{\text{NMI}}$ (Non-Maskable Interrupt) pins low, whereas a software interrupt is explicitly invoked via the execution of the BRK instruction (opcode $00).

Maskability refers to whether or not the CPU can ignore an interrupt; an IRQ can be blocked if the Interrupt Disable (I) flag is set, while an $\overline{\text{NMI}}$ always forces an immediate jump to its vector. Regardless of source, all interrupts push the program counter and status register onto the stack in order to preserve the processor state. Since hardware-triggered IRQs and software-invoked BRK instructions share the same interrupt vector, the interrupt handler must inspect the status register for the break flag to distinguish between them. A break flag set to 1 identifies a software transfer via BRK, whereas a 0 indicates the interrupt was initiated by the hardware via IRQ [9].

2.2.3 Memory Access Timing

M2 is the primary 6502 clock coordinating CPU operations, available on the cartridge connector, as seen in Table 1. The 6502 performs operations on both edges M2, having an active pulse width of $\tau = 357$ ns [8]. On its rising edge, the address bus and R/$\overline{\text{W}}$ both become stable, with setup times of $T_{\text{ADS}} = T_{\text{RWS}} \leq 225$ ns (seen in Figure 3), and remain stable through the active portion of M2. Because R/$\overline{\text{W}}$ is held constant, any given clock cycle is unambiguously classified as either a read or a write cycle [11].

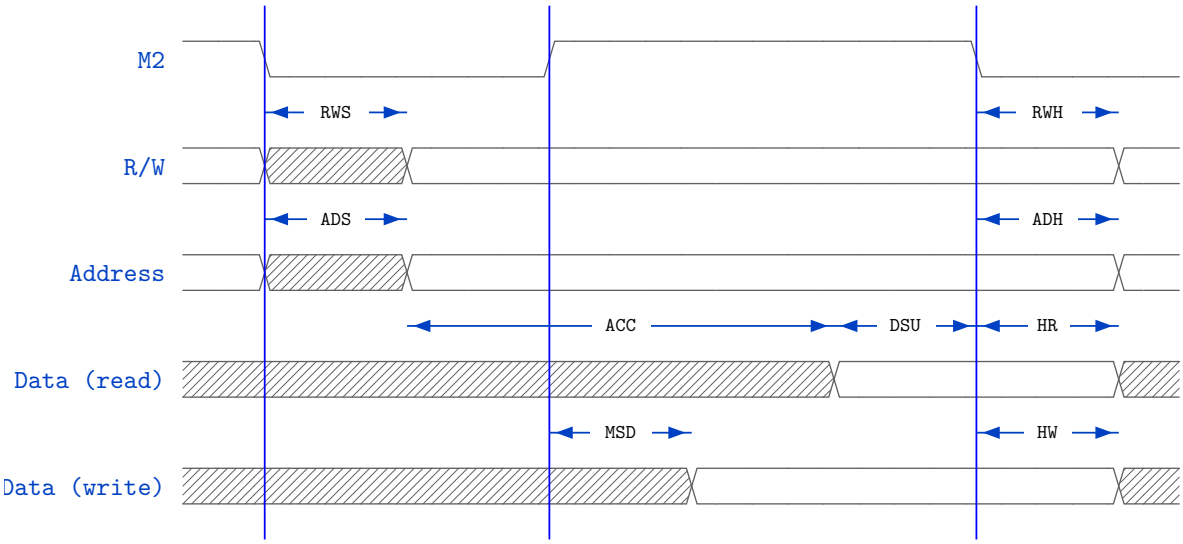


Figure 3: CPU memory access timing diagram [11].

During a write cycle, the CPU drives the data bus, ensuring that it's stable on the falling edge of M2 with a setup time $T_{\text{MSD}} \leq 175$ ns and hold time $T_{\text{HW}} \geq 60$ ns. Any external memory device must therefore latch data during this interval [11].

During a read cycle, an external memory device drives the data bus, ensuring its validity by holding it stable on the falling edge of M2 with setup time $T_{\text{DSU}} \geq 100$ ns hold time $T_{\text{HR}} \geq 10$ ns. The read access time T_{ACC} (see Figure 3) provides the memory device with at least 310 ns to respond with stable data. However, if counting from the rising edge of M2, the interval becomes roughly $\tau - T_{\text{DSU}} = 250$ ns [11].

$\overline{\text{ROMSEL}}$, which is the logical NAND of M2 and CPU address line 15, is asserted during CPU memory accesses between addresses $8000 and $FFFF while M2 is active. Consequently, it's often connected to the chip enable pin on PRG ROM. $\overline{\text{ROMSEL}}$ is the only way to determine the value of CPU address line 15 and is therefore used by cartridges as a combined address-decode and timing gate [12].

2.3 Picture Processing Unit

The CPU alone can run programs and games, but it is not sufficient to handle the graphics. This is where the second primary chip comes in: the PPU. As mentioned in Section 2.1, this unit is responsible for processing and rendering game graphics [13].

2.3.1 Architecture and Memory Mapping

Memory costs at the time of the console's production were very high. That, paired with the producer's goal of making a budget-friendly home console, meant that they needed to minimize the memory usage. These constraints led to the choice of a tile-based rendering system; rather than storing a full picture, the PPU stores a small "recipe" for it—a set of tables that generates pixels on the fly. This is achieved by dividing the screen into a grid (the nametable) and populating each cell with indices that point to a library of predefined 8×8 pixel patterns (the pattern table). This enables the console to simply remember where each pattern is placed, but not actually how the pattern is drawn.

This architecture allows the PPU to generate two distinct layers: a static background built from a grid of tiles, and sprites—the movable foreground objects and characters that slide around independently on top of the background [1, pp. 31-38]. There are 4 sets of tables that allow the PPU to achieve this:

- **Pattern tables:** These hold the primary predefined pixel art tiles. There are two tables (8 KiB total) housed on the cartridge's CHR ROM. Typically, one table is dedicated to background tiles while the other provides the tiles for sprites.
- **Nametables:** These provide the background grid layout. The NES provides 2 KiB of internal memory, called *Console Internal* RAM (CIRAM), dedicated to storing them. Each nametable can represent one screen. When the game scrolls horizontally towards the right, the PPU treats the two nametables as a continuous workspace, showing the right side of the first nametable together with the left side of the second.
- ***Object Attribute Memory* (OAM):** A small 256 B of memory that keeps track of the sprites' coordinates and orientation. The OAM, combined with the pattern table for the sprites, creates the top layer of movable objects.
- **Palette table:** A tiny memory block of 32 B that defines the colors for the tiles.

While the PPU's 14-bit address bus can theoretically access a 16 KiB range, the physical memory available is much smaller. This discrepancy results in address mirroring, a phenomenon where the same physical data "echoes" across multiple address ranges. Depending on how the game scrolls, a different number of nametables are needed. In the most demanding case, 4 nametables are needed for smooth multidirectional scrolling. Therefore, 4 KiB of RAM is reserved for the nametables. However, to save on costs, the internal CIRAM is only 2 KiB. Since most games scroll in a single direction (horizontally or vertically) and only require 2 nametables, this means that the CIRAM is mirrored and duplicated across the 4 KiB that are reserved for it.

The mirroring behavior is then dictated by the CIRAM A10 pin on the cartridge connector; by routing either the PPU's A10 address line (for vertical mirroring) or A11 address line (for horizontal mirroring) to this pin, the cartridge determines how the CIRAM is indexed. When the game uses 4 nametables, the extra 2 KiB of RAM is placed on the cartridge, and therefore the game manufacturer pays for it. In those cases, the CIRAM $\overline{\text{CE}}$ pin is used. It is connected to the PPU's CIRAM enable pin and through an inverter to the enable pin of the extra RAM on the cartridge. This way, both CIRAM and the external RAM cannot be active and enabled simultaneously. The mapper then chooses which of the two RAM chips to activate depending on which of the 4 nametables that the PPU is using. Similar to the CIRAM, the palette RAM is also mirrored since it is only 32 B but is mapped to 256 B. Other than that, 8 KiB is also reserved for the pattern tables, but since they fill the entire area, no mirroring occurs there, as is shown in Figure 4 [14].

2.3.2 Pinout and Registers

The CPU R/$\overline{\text{W}}$, data bus, and address bus lines 0 through 2 pins are directly connected to the PPU, as depicted in Table 2. This enables communication between the two chips. The PPU has eight internal IO registers, memory-mapped to the CPU address space on addresses \$2000 through \$2007 (see Figure 4). This way, the CPU can control the PPU by writing or reading from these addresses. The three address bus lines together with the $\overline{\text{CS}}$ pin are used to select one of the registers, and the R/$\overline{\text{W}}$ pin to determine whether the CPU is writing to or reading from the selected register. Using this method, the CPU can control things such as enabling/disabling background or sprite rendering, or reading/ writing to the CIRAM or OAM indirectly through PPU registers. However, writing to these memory spaces during active rendering isn't safe; therefore, the CPU can also enable generation of non-maskable interrupts ($\overline{\text{NMI}}$), which runs an $\overline{\text{NMI}}$ handler every time it is safe to update OAM or CIRAM. More specifically, the $\overline{\text{NMI}}$ handler runs every time the PPU is in its vertical blank (vblank) state [15], a brief period between rendering every frame. For the duration of this state, the vblank flag in the register is asserted, and if $\overline{\text{NMI}}$ is also enabled, then the PPU pulls the $\overline{\text{INT}}$ pin low, triggering the $\overline{\text{NMI}}$ on the CPU [13].

It is worth noting that D0–7 here is the same data bus exposed on the cartridge connector. Any cartridge hardware that drives this bus outside of a valid PRG access window risks bus contention with the PPU register interface and must therefore ensure its outputs are high impedance whenever it is not the intended bus target [16].

Name	Direction	On Cartridge Connector	Description
CPU R/$\overline{\text{W}}$	Input	Yes	Used to determine if the CPU wants to read or write to the PPU registers.
CPU D0..7	Input/Output	Yes	Used by the CPU to configure PPU registers.
CPU A2..0	Input	Yes	Used by the CPU to select among PPU registers.
$\overline{\text{CS}}$	Input	No	Generated on the mainboard to map the PPU registers to the CPU.
CLK	Input	No	The 26.6017 MHz (PAL) or 21.47727 MHz (NTSC) clock input.
$\overline{\text{INT}}$	Output	No	Connected to the CPU's $\overline{\text{NMI}}$ pin. Used to communicate that it is safe to modify the PPU registers.
VOUT	Output	No	The shifted analog video output.
$\overline{\text{RST}}$	Input	No	A reset signal, used to clear the screen upon reset.
$\overline{\text{WR}}$	Output	Yes	Used when the PPU is writing to the memory.
$\overline{\text{RD}}$	Output	Yes	Used when the PPU is reading from the memory.
PPU A13..8	Output	Yes	The upper six bits of the PPU address bus.
PPU AD7..0	Input/Output	Yes	The lower eight bits of the address bus, but also serves as the PPU data bus.
ALE	Output	No	High during memory accesses to latch the lower 8 bits of the PPU's address bus (AD pins).

Table 2: Ricoh 2C07/2C02 partial pinout [17]. Pin directions are given from the perspective of the 2C07. See Figure A2 for the complete pinout diagram.

2.3.3 Memory Access Timing

Unlike the CPU, the PPU strictly confines memory operations to the positive edge of its clock signal. Each memory access cycle spans exactly two PPU clock cycles and may be issued back-to-back, as is conventional during active rendering. This two-cycle structure is a consequence of the lower eight bits of the PPU address bus being multiplexed with the data bus to minimise pin count (see Table 2), which necessitates two distinct phases: the *address phase* and the *data phase*.

During the address phase, address lines A8..13 are loaded with the target address at the start of the access cycle and held stable throughout the remainder of the access. Simultaneously, A0..7 are driven onto the multiplexed AD0..7 bus and the Address Latch Enable (ALE) signal is asserted, prompting an external latch to capture and retain the lower eight address bits [16].

During the data phase, either $\overline{\text{RD}}$ or $\overline{\text{WR}}$ is asserted and maintained active for the entirety of the second clock cycle, indicating to the addressed memory device that the data bus carries a valid read or write transaction.

The signals ALE, $\overline{\text{RD}}$, and $\overline{\text{WR}}$ are strictly mutually exclusive; under no circumstances are two or more asserted concurrently. During a write cycle, the PPU drives the data bus for the duration of the $\overline{\text{WR}}$ assertion. During a read cycle, the addressed memory device drives the data bus for the duration of the $\overline{\text{RD}}$ assertion [16].

Finally, it is worth noting that setup and hold times are not publicly available, a consequence of the PPU's proprietary hardware architecture.

2.4 Checking Integrated Circuit Lockout Chip

The Checking Integrated Circuit (CIC) lockout chip, called *10NES*, is a custom 4-bit microcontroller-based lockout used in the NES to enforce cartridge authenticity and regional compatibility. It implements a handshake in which the console contains a "lock" CIC, and each licensed cartridge contains a complementary "key" CIC. The two run a proprietary protocol after power-up and reset, and complete a timed handshake to release the CPU and PPU $\overline{\text{RST}}$ pins, allowing them to run [18, pp. 1-18].

2.5 Cartridge Structure and Connector Interface

As explained in Section 2.1, a game cartridge is the primary storage medium for the NES. It is a plastic housing containing a printed circuit board (PCB) that plugs directly into the console's cartridge slot, providing the system with both the executable game code and its graphical data. Without a cartridge inserted, the NES has no software to run [19].

Common to all cartridges is that they contain a CIC chip, some combination of RAM and ROM, and usually a *mapper* chip [19]. As a cartridge's behavior largely depends on the mapper chip it uses, they are traditionally categorized by mapper and then further subcategorized by memory chip configuration. Each NES game is intrinsically tied to the cartridge it's designed for and not cross-compatible with other cartridges, mostly due to the different mapper behaviors.

As mentioned in Section 2.1, the NES uses a dual-bus topology. Hence, memory chips within the cartridge are either connected to the CPU or the PPU using separate address spaces, commonly referred to as memory maps, shown in Figure 4.

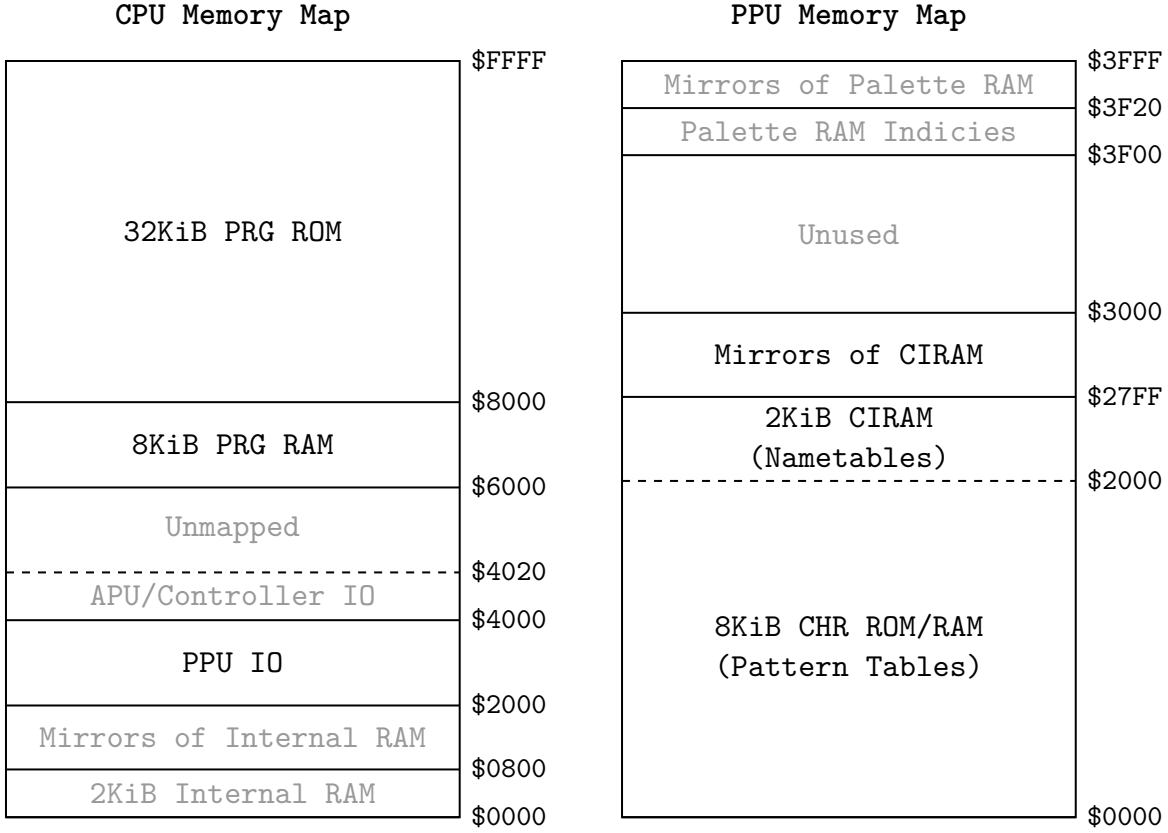


Figure 4: CPU and PPU memory maps [14], [20]. For the CPU map, all addresses at or above $4020 reside on the cartridge; for the PPU map, all addresses below $2000 reside on the cartridge, with boundaries indicated by the dotted lines. Greyed-out regions are not relevant to this report.

The cartridge connects to the console through a 72-pin edge connector called the *cartridge connector*, whose pinout is shown in Figure 5. The pins can be grouped into four main categories: CPU-related signals, shown in Table 1; PPU-related signals, shown in Table 2; CIC-related signals; and CIRAM control signals, explained in Section 2.3. Most of these are directly derived from the respective chip: 2A07, 2C07, or 10NES [12].

A critical concern when designing a cartridge is bus contention, a condition occurring when two or more devices simultaneously drive conflicting logic levels onto the same shared line. This results in a voltage fight between the drivers, producing corrupt data [21].

Only the data buses—the CPU's 8-bit D0..7 and the PPU's multiplexed AD0..7—are susceptible to this condition. This is because they are the only bidirectional buses in the system: during a read cycle, the cartridge (or other memory device) is expected to drive the bus, while during a write cycle, the CPU or PPU itself drives it. If the cartridge asserts data onto the bus at the wrong time, both sides will be simultaneously sourcing the bus, causing contention.

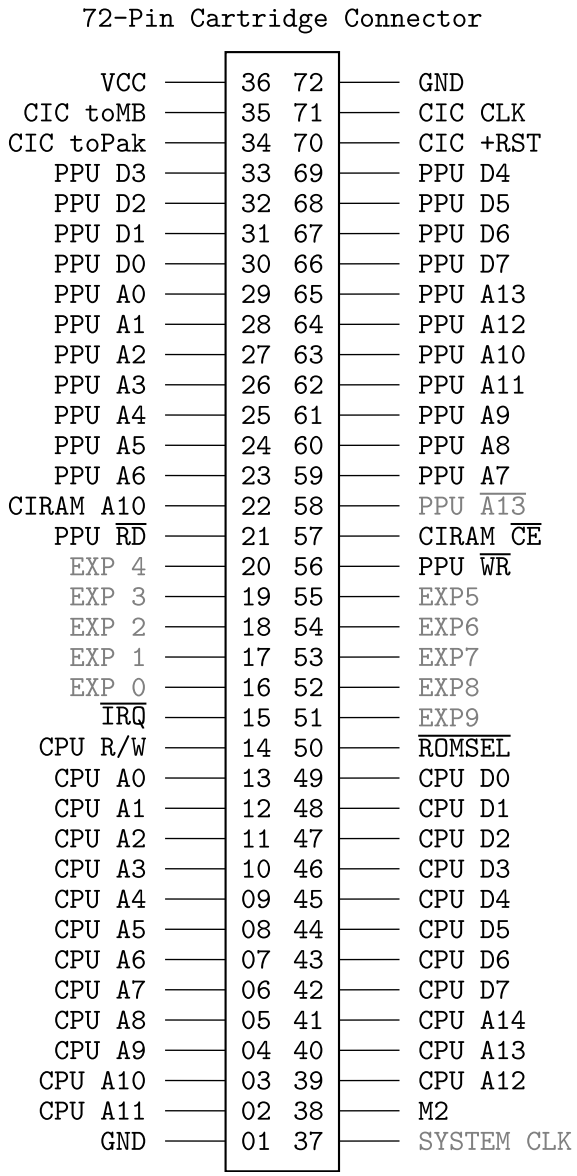


Figure 5: 72-pin NES cartridge connector pinout [12]. Greyed-out pins are not relevant to this project.

2.6 Mappers and Memory Banking

As mentioned in Section 2.5, a cartridge normally contains a mapper, a microchip whose main job is to perform memory banking meant to expand the amount of addressable memory. The CPU and PPU address buses are limited to 16-bit and 14-bit widths, providing only 64 KiB and 16 KiB of directly addressable memory, respectively. Since only a subset of these spaces is allocated to PRG and CHR memory on the cartridge, a strict size limit is enforced on games, concerning both program code and graphics data. However, cartridges can utilize oversized memory chips—those with capacities exceeding the native limits of the address buses. With oversized PRG and CHR chips, not all addresses are directly accessible via the corresponding address bus; an issue resolved by employing a mapper that performs memory banking [22].

Sisyphus targets three mapper configurations. NROM is the simplest: it contains no banking logic, meaning the full PRG and CHR ROM is permanently mapped into the address space [23]. UxROM adds switchable PRG ROM banking while keeping CHR ROM fixed, allowing games to exceed the 32 KiB PRG limit [24]. MMC1 is the most complex of the three, supporting switchable banking for both PRG and CHR ROM as well as configurable mirroring of the PPU nametable memory, enabling a wider range of game configurations [25]. Together, these three mappers cover the majority of the official NES game library [26].

Memory banking is the idea of dividing up an address space into smaller *slots* (also known as windows). Mappers typically employ a slot size between 4 KiB and 32 KiB [27]. A specific portion of the physical memory, known as a *bank*, is assigned, or "mapped", to a slot as seen in Figure 6. Because the total memory capacity exceeds the available address space, the number of banks will surpass the number of available slots. Consequently, some banks remain unassigned and are therefore not addressable from the address space, which is the primary disadvantage of memory banking.

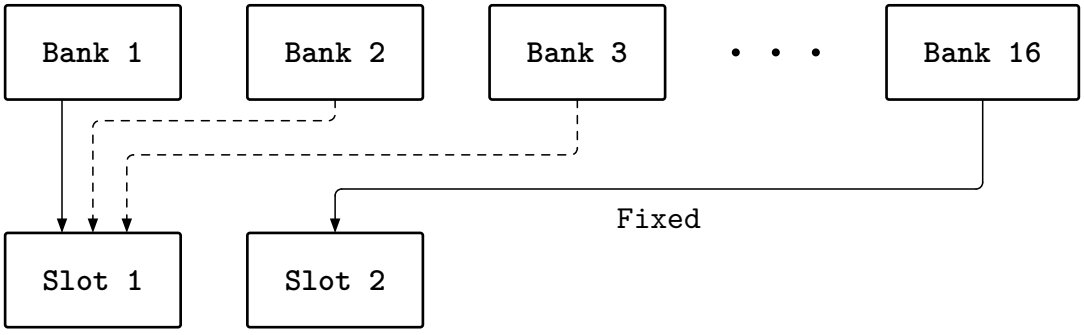


Figure 6: Example of the PRG ROM memory banking used by UxROM [24].

It is the mapper's responsibility to assign banks to slots, done through static or dynamic mapping. In static mapping, the slot-bank assignment is physically hardwired within the mapper chip, creating *fixed banks* used to ensure that interrupt vectors and critical routines are always addressable.

In dynamic mapping, the slot-bank assignment can be changed during game execution, allowing the system to swap different banks into a particular slot as needed. The mapping is controlled by registers within the mapper, memory-mapped to the CPU address space. This enables the CPU to directly write to the registers through the address range $8000 to $FFFF. Although this address range is reserved for PRG ROM —because it is read-only—the mapper can safely intercept write operations and redirect the data to a specific mapper register. Through this mechanism, the CPU maintains complete control over dynamic mapping for both PRG and CHR memory.

Banking is achieved by manipulating address lines. In a typical NES mapper setup, the address lines are divided into two groups: address offset lines, connected directly from the CPU or PPU address bus, and bank-select lines, intercepted by the mapper to produce a new set of address lines.

The UxROM mapper, whose PRG ROM memory banking is depicted in Figure 6, illustrates this concretely. Its PRG address space is split into two 16 KiB slots: a switchable low window ($8000–$BFFF) and a fixed high window ($C000–$FFFF). When the CPU writes to any address in $8000–$BFFF, UxROM latches the written value into a 4-bit bank-select register, selecting which of the 16 PRG banks is mapped into the switchable slot. To resolve a PRG read, the mapper concatenates those four bank-select bits with the lower 14 address offset lines (A13..A0), producing an 18-bit ROM address. In this arrangement, UxROM extends the addressable range from the CPU's native 32 KiB PRG window to up to 256 KiB of physical ROM [24].

2.7 NES 2.0 File Format

As emulation of the NES became increasingly popular, a standardized way of storing and distributing game data digitally became necessary. This led to the introduction of ROM files within the NES emulation community. Modern NES ROM files commonly use the NES 2.0 file format, which is backwards-compatible with the older iNES format [28]. In addition to storing the contents of cartridge memory, a ROM file using NES 2.0 also contains other important information about the game, such as the mapper number and memory size.

A NES 2.0 ROM file is divided into five main parts: a header field, a trainer field, PRG ROM data, CHR ROM data, and miscellaneous ROM data. The most relevant parts for this project are described below.

The first 16 bytes of the ROM file constitute the header, which contains information describing the cartridge hardware configuration. For the scope of this project, the following information from header fields is relevant:

- **Authenticity check:** A fixed sequence of ASCII symbols used to identify the file as a valid NES ROM image.
- **ROM size:** Specifies the size of the PRG ROM and CHR ROM regions, allowing the emulator to allocate and map memory correctly.
- **Mapper number:** Identifies the mapper hardware used by the cartridge. The mapper controls functionality such as bank switching, interrupt generation, and address translation between the CPU/PPU and cartridge memory.
- **Nametable arrangement:** Specifies whether horizontal mirroring, vertical mirroring, or mapper-controlled mirroring is used for PPU nametables.
- **Console type:** Indicates which hardware the ROM targets, such as the standard NES/Famicom, Nintendo Vs. System, PlayChoice-10, or extended console types.
- **CPU/PPU timing:** Defines the timing standard used by the game, for example, NTSC, PAL, or multi-region compatibility. Since CPU and PPU frequencies differ between systems, correct timing information is important for accurate emulation.

All information present in the header can be viewed in full in Table 3.

Byte(s)	**Contents**
0–3	File identification string: 0x4E, 0x45, 0x53, 0x1A (ASCII “NES” followed by the MS-DOS end-of-file character)
4	Least significant byte of PRG ROM size
5	Least significant byte of CHR ROM size
6	Nametable arrangement (mirroring), presence of non-volatile memory, presence of trainer field, and lower mapper bits
7	Console type, NES 2.0 format identifier, and additional mapper bits
8	Upper mapper bits and submapper number
9	Upper bits of PRG ROM and CHR ROM size
10	PRG memory size information
11	CHR memory size information
12	CPU/PPU timing mode
13	Extended console type information
14	Number of miscellaneous ROM areas
15	Default expansion device

Table 3: Contents of the NES 2.0 Header.[29]

Following the header—and optionally a trainer field, which is outside the scope of this project—the ROM file contains the PRG ROM and CHR ROM sections. The PRG ROM section stores the program code executed by the CPU, while the CHR ROM section contains the graphical tile data used by the PPU for rendering graphics [29].

Chapter 3
Digital Systems Theory

The development of Sisyphus necessitates a multidisciplinary understanding of digital systems, including hardware design, emulation, FPGAs, and electrical signal characteristics. This chapter establishes the theoretical framework required to bridge the gap between retro hardware and modern electronics.

3.1 Reconfigurable Hardware Designs using FPGAs

An FPGA is a general-purpose integrated circuit (IC) that can be configured to implement any digital circuit or system. They act as a more flexible alternative to hardware-integrated circuits, allowing designers to implement hardware designs after manufacturing by reconfiguring the FPGA.

3.1.1 Overview

FPGAs are fundamentally constructed from *logic cells*, a unit of logic, historically consisting of a 4-6 input lookup table (LUT) together with flip-flops [30, p. 11]. Multiple cells are grouped into *logic blocks* with local routing and arithmetic logic. The FPGA is then comprised of an interconnected network of these logic blocks and I/O blocks, for communication outside the FPGA.

The first generations of FPGAs were homogeneous, consisting only of a network of soft blocks, such as logic blocks. However, more modern FPGAs are heterogeneous, meaning that they contain both soft blocks and hard blocks. Hard blocks, such as memory blocks used to store memory more efficiently or multiplier blocks for optimized multiplication, have predetermined functionality. Heterogeneous FPGAs have gained popularity due to the inclusion of hard blocks, leading to greater performance and area efficiency [31, pp. 85-86].

Despite their advantages, FPGAs also have several flaws. The programmable architecture that makes them flexible also makes them larger, slower, and more power consuming than other hardware solutions [32]. Since FPGAs are fundamentally composed of logic blocks, they can only emulate the logical behavior of a circuit through binary logic implementation. As a result, they cannot replicate the physical characteristics of analog circuitry. Properties such as high impedance behavior and other analog effects are therefore difficult or impossible to emulate accurately on an FPGA.

To program an FPGA, a Register Transfer Level (RTL) description of the hardware design must first be made. An RTL description is essentially a design of the desired hardware behavior. This is usually done with a Hardware Description Language (HDL), such as VHDL or Verilog. As the name suggests, an HDL is a computer language that allows the designer to describe the desired behavior or logic of the FPGA. To simplify development, designers often use Intellectual Property (IP) cores, which are pre-designed and reusable logic modules that reduce development time and complexity [33, p. 7].

3.1.2 Development Flow

The conversion from a given RTL design into something usable by the FPGA can typically be divided into 4 stages: logic synthesis, technology mapping, routing and placement, and bitstream generation.

The RTL design is first synthesized into a gate-level representation, turning the HDL description into a network of general logic gates, such as Boolean logic gates and flip-flops. The synthesized logic is then technology-mapped, where the general logic gates are mapped to physical device resources inside the FPGA fabric, which, in a modern FPGA, typically includes things such as k-input LUTs.

After synthesis and technology mapping, the design undergoes a place-and-route process. It begins by determining an efficient physical placement for the mapped logic blocks in the FPGA fabric. Common criteria for placements are a minimum wire length and meeting set timing constraints. After placement, the routing between appropriate logic blocks occurs throughout the FPGA.

Lastly, a bitstream is generated for the design, dictating how the FPGA fabric should be configured. Once generated, the bitstream is loaded onto the FPGA device using a bitstream loader [33, pp. 68-73], [34, pp. 24-39].

3.2 Signal Integrity for Digital Circuits

Signal integrity across asynchronous domain boundaries and between logic families requires an understanding of several interrelated concepts and challenges. Metastability occurs when a digital system momentarily cannot reliably determine a logic state. Synchronizers solve this by providing extra clock cycles for resolution. Noise and voltage incompatibilities require different approaches. Schmitt triggers provide noise immunity through hysteresis, a state-dependent behavior in which switching points depend on past states. Level translation ensures voltage compatibility between different voltage domains. This section establishes these foundations.

Metastability is a statistical phenomenon and can therefore not be eliminated; only made extremely unlikely. The *Mean Time Between Failures* (MTBF), a common reliability metric, for a synchronizer is commonly modelled as

$$\text{MTBF} = \frac{e^{\frac{S}{\tau}}}{W \cdot F_c \cdot F_d}$$

with $S = n \cdot T_c$ (n stages, T_c = destination clock period), τ the device time constant, W the metastability window, F_c the destination clock frequency, and F_d the asynchronous event rate. MTBF, therefore, grows exponentially with added resolution time (more stages or larger T_c) and decreases linearly with event or clock rates [35], [36, pp. 392-400].

Practically, use measured τ and W and choose ≥ 2 stages for single-bit signals unless calculations justify fewer [36], [37, p. 65].

Synchronizers work when a signal crosses from one clock domain to another. The receiving flip-flop can sample while the input is changing and be forced into a metastable analog state that takes an indeterminate time to resolve. If that metastability propagates into logic, it can produce incorrect or indeterminate system behavior. Synchronizers are typically a chain of flip-flops clocked in the destination domain that give the first sampled flop extra clock cycles to resolve metastability before its output is used, dramatically reducing the probability that a metastable event causes system failure [37, p. 65].

A Schmitt trigger is a comparator circuit with positive feedback that produces hysteresis. It defines two distinct switching thresholds: a higher threshold for transitions from low to high, and a lower threshold for transitions from high to low. An input rising above the upper threshold forces the output to the logic-high state; an input falling below the lower threshold forces the output to the logic-low state, as seen in Figure 7. Because the output state is held between these thresholds, the circuit rejects small or rapid input fluctuations and noise around the switching region, yielding clean, noise-immune digital transitions often used to produce stable square-wave outputs from slow or noisy signals [38].

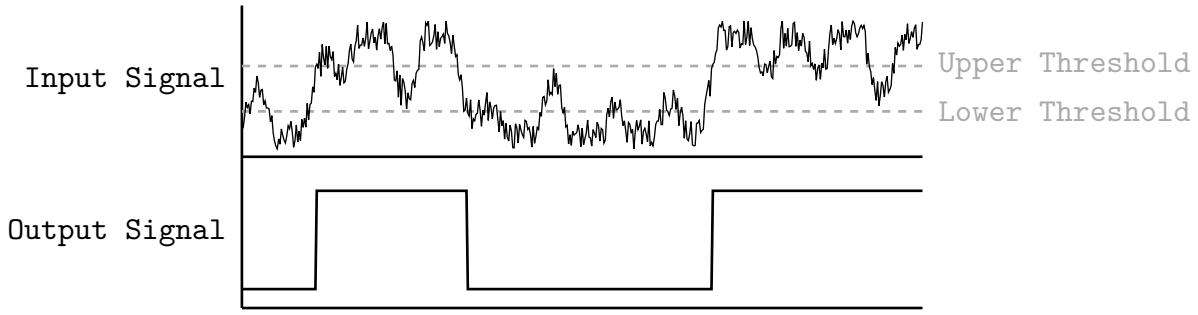


Figure 7: Schmitt trigger with upper and lower voltage thresholds.

When interfacing logic families with different supply voltages, a fundamental challenge arises: input and output threshold voltages may not align. A lower-voltage output may fall below a higher-voltage receiver's logic-high threshold, while a higher-voltage output may exceed a lower-voltage receiver's maximum input rating, risking device damage or undefined logic states. Level translation circuits address this incompatibility by converting signals between voltage domains, ensuring transitions meet the receiving logic family's VIH/VIL thresholds.

Different logic families interpret voltages differently. TTL (Transistor–Transistor Logic) considers voltages above 2.0 V as high, while CMOS (Complementary metal–oxide–semiconductor) thresholds scale with supply voltage so that the same absolute voltage may mean different logic values across different parts. This asymmetry explains why a 3.3 V LVCMOS output can reliably drive a 5 V TTL input, but not vice versa—a commonly exploited but fragile design pattern that requires careful management [39].

Chapter 4
Implementation

This chapter describes the design and implementation of Sisyphus. It begins with an overview of the development environment and system architecture, establishing the tools used and how the system's components interrelate. Subsequent sections detail the specific components of Sisyphus.

4.1 Hardware Platform and Development Toolchain

Sisyphus is built using the Diligent ZedBoard, a low-cost evaluation and development board popular among hobbyists and professionals alike. It boasts a wide set of on-board peripherals; most crucial to Sisyphus is the SD-card reader and FMC LPC connector. At the core of the ZedBoard sits the AMD Zynq-7000 all programmable SoC. It combines an ARM Cortex A9 processor with an Artix-7 FPGA on a single chip [40]. This allows the SoC to be used for both software and hardware development, perfect for Sisyphus, as it significantly simplifies system design. It allows the two domains to communicate directly via the AXI4 (Advanced eXtensible Interface) interconnect without requiring an external communication interface. The ZedBoard was primarily chosen for its status as a user-friendly beginner board and ease of availability.

Because the Zynq-7000 is manufactured by Xilinx, now owned by AMD, hardware and software design development was done using the vendor-provided Vitis and Vivado Design Suites.

Vivado enables hardware design and synthesis for the FPGA fabric of the Zynq-7000. It provides a block design environment, the *IP Integrator*, where IP cores—often vendor-supplied pre-packaged hardware designs—can be instantiated and connected graphically, as well as a full RTL synthesis and implementation flow for custom HDL modules. Verilog was chosen as the HDL due to its lower learning curve compared to VHDL and its popularity in the FPGA development space.

The block design approach allows the *Processing System* (PS)—the ARM Cortex-A9 —to be configured and exported alongside the *Programmable Logic* (PL)—the Artix-7 FPGA—as a unified hardware platform. Once the design is finalized and implemented, Vivado exports a hardware handoff file (.xsa), which encapsulates the bitstream and all necessary hardware metadata required by the Vitis software toolchain.

Vitis uses this .xsa file to generate a Board Support Package (BSP), providing the low-level drivers and runtime libraries needed to target the specific hardware configuration. On top of this BSP, application projects can be built targeting either a bare-metal standalone execution environment or a real-time operating system such as FreeRTOS. Together, Vivado and Vitis form a tightly integrated design suite spanning the full development cycle from hardware synthesis to software deployment, well-suited to the hybrid PS/PL architecture of the Zynq-7000.

The ZedBoard's AMD Zynq-7000 SoC serves as the backbone of Sisyphus, with the PS and PL serving two fundamentally different problem domains, reflecting a deliberate hardware-software partitioning decision. Inspired by the EverDrive partitioning, the PL is responsible for all timing-critical operations—namely, interfacing with the NES cartridge connector and handling real-time mapper emulation. The PS is instead responsible for all high-level, non-timing-critical tasks. This includes loading an NES 2.0 ROM file from the SD card, parsing the header, and reconfiguring the FPGA accordingly.

4.2 Physical Assembly and Logic Level Translation

As seen in Figure 8, an *AMD FMC XM105 Debug Card* is connected to the ZedBoard FMC LPC connector. It exposes all 68 user-defined, single-ended LA pins used as GPIO. Because 56 of the 72 signals exposed on the cartridge connector (see Figure 5) are essential for cartridge emulation, it was not possible to rely on the 40 onboard PMOD I/O pins provided by the ZedBoard. The LA pins connect to the NES through a *perforated prototyping board* (perfboard), housing intercepting bus transceivers meant to bridge the different voltage logic levels used by the NES and the ZedBoard. View the schematic in Figure D1. The assembly is finally connected to a cartridge breakout board.

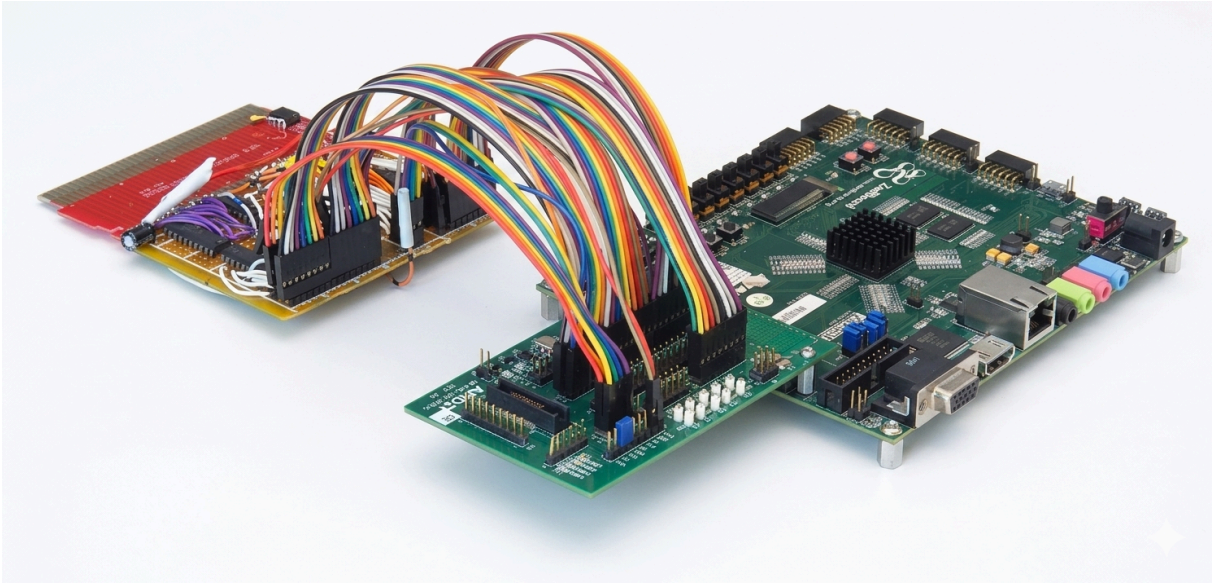


Figure 8: Sisyphus physical assembly, including the ZedBoard, AMD FMC XM105 Debug Card, perfboard, and cartridge breakout board.

On the cartridge breakout board sits an ATtiny13A microcontroller responsible for the CIC lockout bypass, explained in Section 2.4. It implements the open-source CIC algorithm by Krikzz [41]. The ATtiny13A was programmed via In-System Programming (ISP) using an Arduino as the programmer. The Arduino was configured as an ISP with the official Arduino IDE example sketch [42]. The Arduino and ATtiny13A were connected on a breadboard, and the CIC binary was flashed to the ATtiny13A with the required fuse settings to enable operation from the NES-provided external clock.

Level shifting was done using the SN74LVC245A (see Figure A3 for pinout), a common octal bus transceiver with 3-state outputs [43]. The output voltage is bounded by the transceiver's single supply rail. Therefore, signals originating from the NES are downshifted to 3.3 V at the ZedBoard interface, whereas signals travelling in the reverse direction are not upshifted and arrive at the NES unchanged at 3.3 V. Under most circumstances, driving a 5 V logic device with a 3.3 V signal would be unreliable. However, because the NES was designed using 5 V TTL and the ZedBoard 3.3 V CMOS—both using a T_{IL} of 0.8 V and T_{IH} of 2.0 V—the particular combination of logic families makes it viable.

The bus transceiver signal direction is governed by the DIR pin. The majority of signals on the NES cartridge connector, illustrated in Figure 5, are unidirectional and thus have their corresponding DIR pins statically tied to either GND or VCC. The CPU and PPU data buses are an exception, as their direction is determined by whether the cartridge is being read from or written to; consequently, the DIR pins of the transceivers handling these buses are driven dynamically by the cartridge emulation logic. These signals are henceforth designated DIR_{CPU} and DIR_{PPU}, and are discussed further in Section 4.7.3.

4.3 Programmable Logic Block Design and DFX Partitioning

The PL design is organized as two distinct layers: a persistent top block design and a reconfigurable partition (RP) containing the current cartridge block design; swapped at runtime via *Dynamic Function eXchange* (DFX) partial bitstreams. DFX is an AMD-specific technology allowing partial reconfiguration of the FPGA fabric [44]. A DFX Decoupler IP sits at the boundary between the static region and the cartridge RP. When decoupling is asserted, it drives all signals entering the RP to safe default values, preventing the partially configured fabric from interfering with the static region. The decoupler is controlled by the PS via AXI4. The whole system is using a 100 MHz fabric clock referred to as F_{CLK}. The four figures below present this hierarchy top-down, beginning with a complete system overview shown in Figure 9.

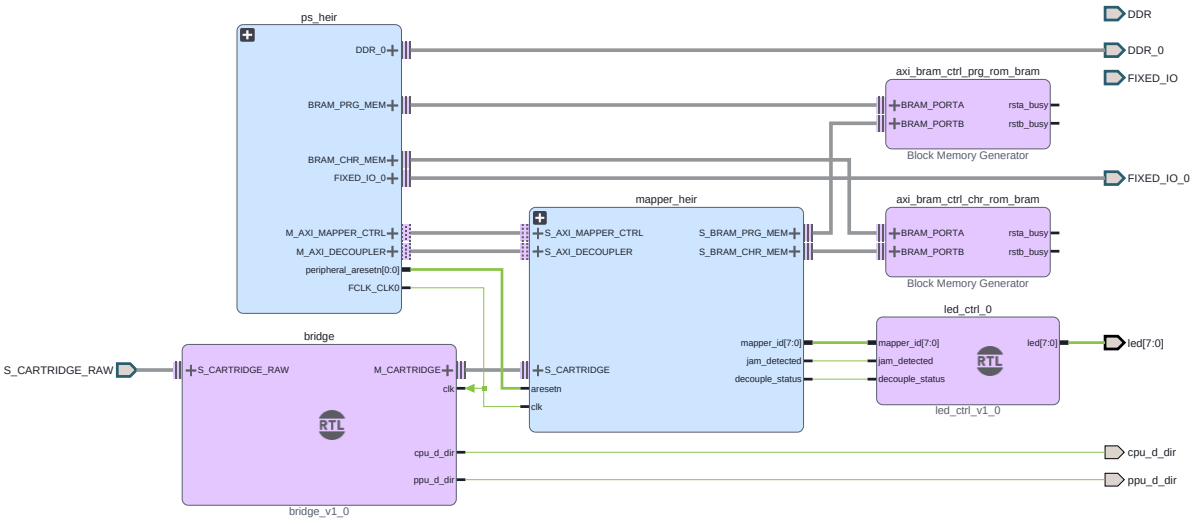


Figure 9: High-level overview of the block design exported from Vivado.

Both the PRG ROM and CHR ROM are allocated as true dual-port *Block RAM* (BRAM) in the static partition. BRAM is synchronous on-chip dedicated fabric memory implemented using the Block Memory Generator IP in the IP Integrator [45]. It is connected to both AXI BRAM Controller IPs and the cartridge RP, enabling both the PS and cartridge RP to access memory. Given the synchronous nature of BRAM, a dedicated bridge module (whose logic is explained later in Section 4.7) is used to ensure correct clock domain crossing between the PL fabric and the NES clock domains. The module forwards the synchronized signals through the DFX decoupler to the current cartridge RP. It is also responsible for driving the DIR_{CPU} and DIR_{PPU} signals introduced earlier in Section 4.2.

Figure 10 represents the boundary of the PS domain: all timing-critical NES interfacing is absent here. The Zynq PS exposes its general-purpose AXI master port, clocked by F_{CLK}. This connects to an AXI SmartConnect IP that fans out to four downstream AXI slaves: CHR and PRG MEM BRAM controllers, mapper configuration interface, and DFX decoupler control register.

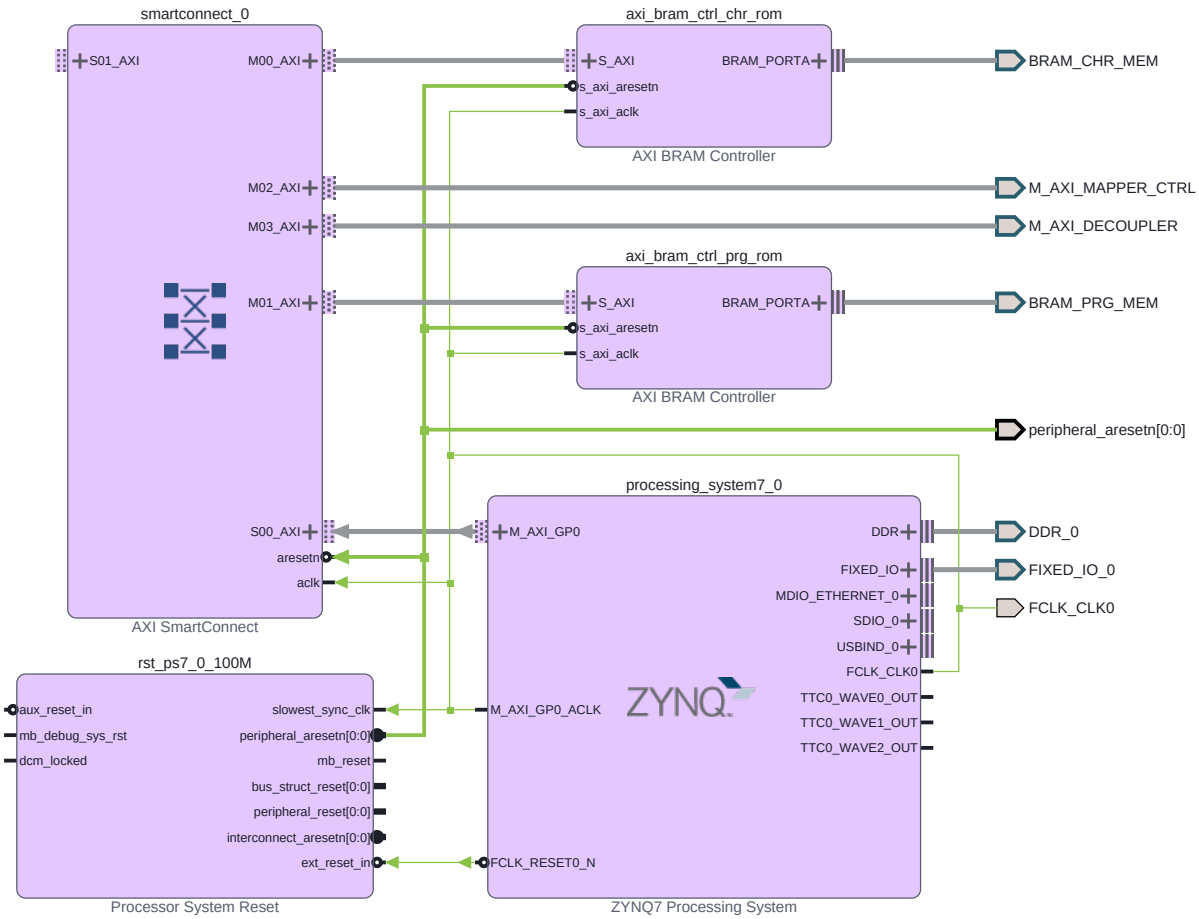


Figure 10: Processing system block design hierarchy exported from Vivado.

Figure 11 shows the mapper hierarchy. It contains the DFX decoupler and cartridge RP, together with jam and reset detection modules. Both modules monitor the CPU address bus via the cartridge connector. The reset detection is based on the $FFFD address (location of the reset vector), and the jam detection is based on the $FFFF address

(see Section 2.2.1). When JAM is detected across multiple consecutive M2 cycles, the module asserts its detected output, which propagates to the LED controller connected to the ZedBoard LEDs for external observation.

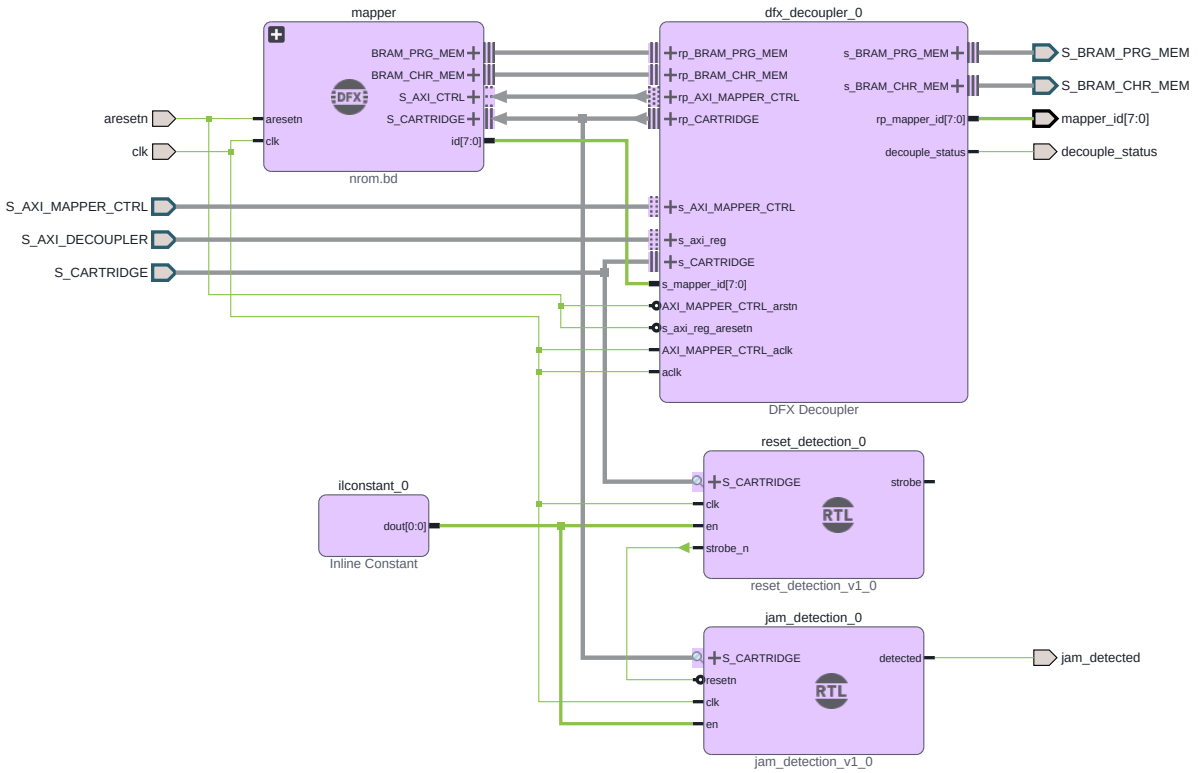


Figure 11: Cartridge block design hierarchy exported from Vivado.

Each cartridge RP comprises two primary components: a mapper module responsible for implementing the requisite mapper logic, and an AXI4-compatible mapper configuration IP managed by the PS (see Figure 12). Since cartridges within a given mapper family, such as NROM, may differ in hardwired attributes including nametable mirroring and PRG/CHR memory capacity, the PS must be capable of configuring these parameters on a per-title basis.

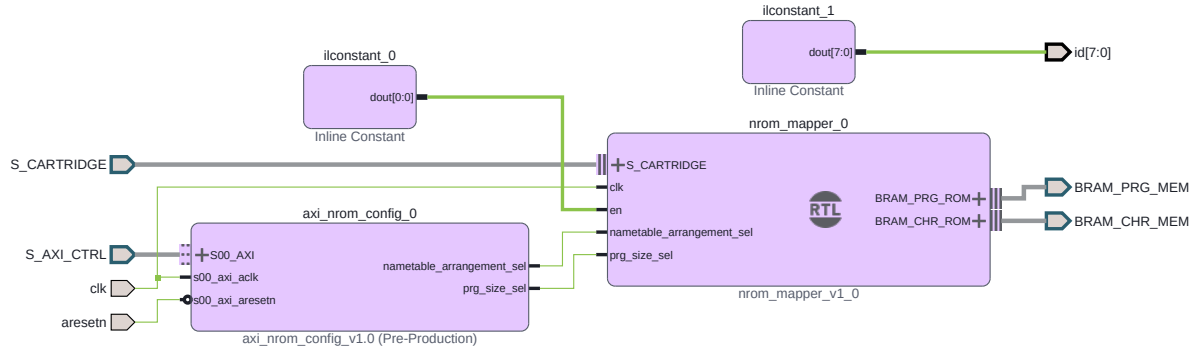


Figure 12: NROM cartridge block design exported from Vivado.

4.4 Cartridge Emulation

As described in Section 2.5, a cartridge typically consists of a mapper chip together with some combination of PRG and CHR memory. In Sisyphus, the logic for each mapper is implemented as a dedicated RTL module within the corresponding cartridge block design, as explained in Section 4.3. This section further describes the behavior and functionality of the mappers currently supported by Sisyphus: NROM, UxROM, and MMC1.

4.4.1 NROM

Being the simplest mapper——technically not even a mapper, since no mapper chip is present——NROM contains minimal logic: PRG ROM mapped at $8000–$FFFF, CHR ROM mapped at $0000–$1FFF, and hard-wired nametable mirroring, where $\text{CIRAM}_{\text{A10}}$ is connected to either PPU_{A10} (vertical mirroring) or PPU_{A11} (horizontal mirroring) [23]. The memory fields used by NROM can be viewed in Table 4.

Field	Bank Sizes	Address Range	Description
PRG ROM	16 KiB or 32 KiB	$8000-$FFFF	Stores the game's program code and logic.
PRG RAM	—	$6000–$7FFF	Writable memory for save data and temporary game state.
CHR ROM/ RAM	8 KiB	$0000-$1FFF	Stores tile and sprite graphics used by the PPU.

Table 4: NROM Memory Fields [23].

4.4.2 UxROM

While UxROM shares the physical signal mapping used by NROM, it additionally supports 16 KiB PRG bank switching. CPU writes to the cartridge address range ($8000–$FFFF) to update the currently selected PRG bank.

The lower 16 KiB CPU region ($8000–$BFFF) is switchable, while the upper 16 KiB region ($C000–$FFFF) is fixed to the last PRG ROM bank. This ensures that common game code and reset vectors remain permanently accessible. PRG ROM addressing is performed by combining the selected bank number with the lower CPU address bits, allowing the CPU to access data within the active 16 KiB bank.

Notably, UxROM does not utilize CHR ROM. Instead, it employs a larger PRG ROM and stores character data in 8 KiB of CHR RAM, allowing graphics data to be written dynamically at runtime [24]. UxROM memory fields can be seen in Table 5.

Field	Bank Sizes	Address Range	Description
PRG ROM	16 KiB + 16 KiB fixed	$8000-$FFFF	Stores the game's program code and logic.
PRG RAM	—	$6000–$7FFF	Writable memory for save data and temporary game state.
CHR RAM	8 KiB	$0000-$1FFF	Stores tile and sprite graphics used by the PPU.

Table 5: UxROM Memory Fields [24].

4.4.3 MMC1

MMC1—the mapper chip present in the SxROM cartridge family—extends the capabilities of NROM and UxROM by providing flexible PRG, CHR, and nametable control through a register-based system rather than simple bank switching.

CPU writes to $8000–$FFFF are treated as configuration inputs. A write with bit 7 set resets the internal shift logic; otherwise, data is shifted bit-by-bit into a 5-bit register. After five writes, the value is committed to one of four internal registers based on address range: control ($8000–$9FFF), CHR bank 0 ($A000–$BFFF), CHR bank 1 ($C000–$DFFF), or PRG bank ($E000–$FFFF) [25].

The control register selects nametable mirroring mode and PRG/CHR banking mode. PRG memory can be mapped as either a single 32 KiB region or split into two 16 KiB regions, where one half is fixed, and the other is switchable. CHR banking allows either a single 8 KiB bank or two independent 4 KiB banks, as shown in Table 6.

Field	Bank Sizes	Address Range	Description
PRG ROM	16 KiB + 16 KiB fixed or 32 KiB	$8000-$FFFF	Stores the game's program code and logic.
PRG RAM	8 KiB	$6000–$7FFF	Writable memory for save data and temporary game state.
CHR ROM/ RAM	4 KiB + 4 KiB or 8 KiB	$0000-$1FFF	Stores tile and sprite graphics used by the PPU.

Table 6: MMC1 Memory Fields [25].

4.5 Dynamic Loading of Games

To facilitate the execution of various games without requiring manual reprogramming, Sisyphus utilizes a dynamic loading process that bridges the PS file system with the real-time hardware emulation in the PL. At boot, the PS mounts the SD card and scans for compatible ROM files, automatically selecting the first valid file it encounters. Once identified, the system parses the NES 2.0 header independently from the rest of

the game file to extract essential metadata, such as the mapper ID and the memory sizes. This isolated parsing allows the system to dynamically allocate the exact amount of memory needed to store the PRG and CHR data, eliminating the need for a fixed, maximum-size buffer. Additionally, while the parser is designed to support the NES 2.0 format, the system maintains backwards compatibility by ignoring higher-order bytes when processing older iNES files, as the first eight bytes remain consistent across both formats.

The game's PRG code and CHR data are extracted from the ROM file and written into BRAM within the static partition via the AXI BRAM Controller, mapping the data to the appropriate memory addresses. Additionally, the hardware is updated via AMD's Dynamic Function eXchange (DFX) to match the specific mapper architecture required for the game. This reconfiguration is handled through the Device Configuration Interface and the Processor Configuration Access Port (PCAP) to perform a non-secure DMA (Direct Memory Access) transfer of the partial bitstream. The process concludes with the mapper being configured based on the extracted metadata through the memory-mapped interface.

4.6 Hardware Verification Test ROMs

To verify the correctness of the cartridge emulation logic independently of any commercial game ROM, custom test ROMs were developed and then assembled as NROM-compatible binaries. All programs are using an isolated hardware design based on the same custom NROM hardware implementation and synchronization methodology described in Section 4.4.1 and Section 4.7. However, NROM is augmented with a section of the PRG ROM region that is writable—the LED segment at address $8000 whose value was directly connected to the eight general-purpose LEDs of the ZedBoard. This provides a simple output for observing program state. The hardware platform also includes a JAM detection mechanism (based on the CPU address behavior mentioned in Section 2.2.1) that outputs $FF on the LEDs during a CPU stall.

Below follows a description of the most significant test programs developed.

4.6.1 Simple Store

The *simple-store* test is the most minimal. After reset, the CPU loads the immediate value $55 (01010101) into the accumulator and writes it to address $8000, where the LED byte is initialized to $AA (10101010). A successful write causes all eight LEDs to toggle simultaneously, confirming that the full CPU read/write cycle—address decoding, data propagation, $\overline{\text{ROMSEL}}$ assertion, and the write-enable path through the mapper—functions correctly. The program then enters an infinite loop, holding the result stable for observation. The cycle-by-cycle signal trace is annotated in the source, making this test well-suited for *Integrated Logic Analyzer* (ILA)-based verification; each CPU signal's expected state is known at every clock cycle, allowing direct comparison against captured waveforms. The PPU is held idle throughout the program.

4.6.2 Continuous Store

The *continuous-store* test extends the simple-store concept into a continuous, time-observable pattern. Rather than a one-shot write, the program alternates the LED value between $55 and $AA every 2.2 seconds. Each iteration reads the current LED value, performs an exclusive OR operation with $FF to toggle all bits, writes it back, and then calls a delay subroutine. Because correct execution requires sustained, repeated bus transactions over thousands of cycles, this test is sensitive to intermittent signal integrity issues that a single-shot test would not catch. A steadily blinking LED pattern at the expected frequency indicates sustained correct operation of the CPU signals. The PPU is held idle throughout the program.

4.6.3 Rendering Control

The *rendering-control* test uses $\overline{\text{NMI}}$-driven frame timing and interrupt source discrimination in a single program. The reset handler performs a full system initialization sequence: it waits for two PPU warm-up vblanks and zero-initializes the 2 KiB of internal CPU RAM together with CIRAM. $\overline{\text{NMI}}$ is subsequently enabled by writing to $2000, and the CPU enters an infinite loop. During normal operation, rendering is controlled interactively via the gamepad. Each $\overline{\text{NMI}}$—firing once per PPU vblank—polls the D-pad and controls the background rendering accordingly: right enables background rendering by writing $08 to $2001, left disables it by writing $00. This allows the tester to control PPU output in real time. LED bit 0 mirrors the rendering status, while LED bits 6–3 toggle every 25 frames, providing a secondary confirmation that $\overline{\text{NMI}}$ is firing at the expected frequency. The IRQ handler serves as an auxiliary check; after an interrupt, it inspects the B flag in the processor status register to distinguish a hardware IRQ from a software BRK, writing either $55 or $AA to the LEDs.

4.7 Synchronization and Data Bus Control

During initial hardware testing, signal instability was observed using the Vivado ILA IP core, preventing proper game execution. The findings are documented in Section 5.3. This section details the signal synchronization measures developed as a result.

4.7.1 CPU Signal Synchronization

As explained in Section 2.2.3, all CPU-related signals are guaranteed to be stable while M2 is high. This means they can be latched on the positive edge of M2 or the falling edge of $\overline{\text{ROMSEL}}$, ensuring signals are stable throughout the read or write cycle. However, since M2 is an asynchronous signal relative to F_{CLK}, it must be passed through a 2-stage flip-flop synchronizer to mitigate metastability, as described in Section 3.2. The number of stages was determined using the MTBF formula:

$$\text{MTBF} = \frac{e^{\frac{S}{\tau}}}{W \cdot F_c \cdot F_d}$$

AMD does not publish device-specific values for τ and W for the Zynq-7000. Therefore, representative values from published FPGA metastability measurements at comparable process nodes give $\tau \approx 50$ ps and $W \approx 3$ ps. Assuming 2 stages needed and setting $S = 2 \cdot 10$ ns $= 20$ ns, $F_c = F_{\text{CLK}} = 100$ MHz, and $F_d = \text{M2} = 1.66$ MHz yields:

$$\text{MTBF} = \frac{e^{400}}{498} \approx 10^{171} \text{ s}$$

After synchronization, the output M2_{sync} is passed through a one-cycle hold debounce circuit to produce $\text{M2}_{\text{filtered}}$. The debounce circuit filters out false edges, serving a similar purpose as the Schmitt trigger buffer, described in Section 3.2. While some Zynq-7000 I/O natively implement Schmitt trigger buffers, the HP (High Performance) I/O banks used for the FMC LPC connector do not—making the fabric-based debounce solution necessary. The pipeline introduces a total delay of 4 F_{CLK} cycles (40 ns) on both the rising and falling edges of $\text{M2}_{\text{filtered}}$ relative to the raw M2 signal (visualized in Figure 13). This delay must fit within the T_{HW} write data hold time of 60 ns to ensure that data latched on the falling edge of $\text{M2}_{\text{filtered}}$ is held stable by the CPU (described in Section 2.2.3). This is not a consideration for R/$\overline{\text{W}}$ and the address bus, as they are reliably latched on the rising edge of $\text{M2}_{\text{filtered}}$. $\overline{\text{ROMSEL}}$ pass through the same pipeline as M2 to produce $\overline{\text{ROMSEL}}_{\text{filtered}}$.

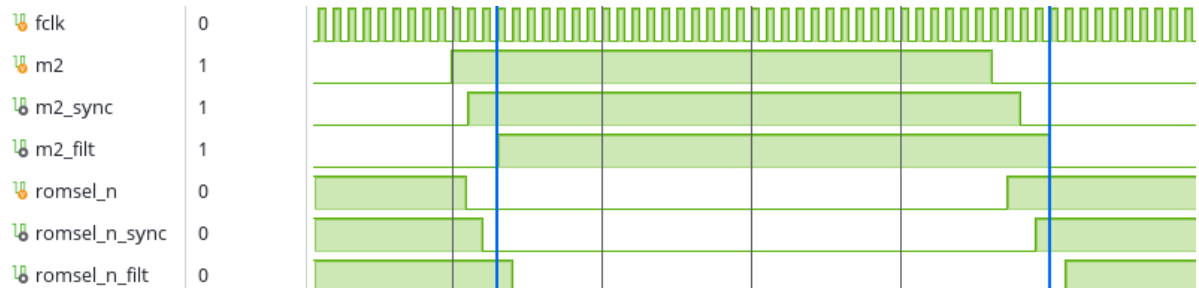


Figure 13: CPU simulation waveform visualizing synchronization timing. The bars indicate when signals are latched.

4.7.2 PPU Signal Synchronization

The PPU clock is not exposed on the cartridge connector, so $\text{M2}_{\text{filtered}}$-based latching used for the CPU signals cannot be directly applied. Instead, the PPU's $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes serve as the timing reference for the PPU domain, as they delineate the write and read phases of each PPU memory access cycle as described in Section 2.3.3.

As on the CPU side, $\overline{\text{RD}}$ and $\overline{\text{WR}}$ are asynchronous with respect to F_{CLK} both signals are therefore passed through a 2-stage flip-flop synchronizer and a one-cycle debounce circuit, producing $\overline{\text{RD}}_{\text{filtered}}$ and $\overline{\text{WR}}_{\text{filtered}}$, respectively. Similar to before, the pipeline introduces a total delay of 4 F_{CLK} cycles (visualized in Figure 14). The 2-stage count is justified by the same MTBF analysis as in Section 4.7.1, with F_{d} now corresponding to the PPU clock rate of approximately 5.32 MHz, resulting in MTBF $\approx 3.3 \cdot e^{170}$ s. The PPU address and data buses are latched on the falling edge of either $\overline{\text{RD}}_{\text{filtered}}$ or $\overline{\text{WR}}_{\text{filtered}}$.

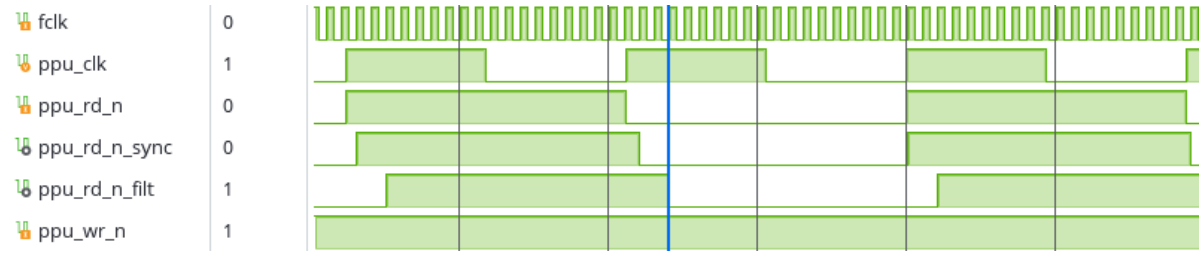


Figure 14: PPU simulation waveform visualizing synchronization timing. The bar indicates when signals are latched.

4.7.3 Preventing Bus Contention

As discussed in Section 2.5, the CPU and PPU data buses are bidirectional: they are driven by the console or cartridge depending on the type of memory access cycle. Simultaneous assertion by both parties would cause bus contention, corrupting the transferred data. In Sisyphus, this is prevented through dynamic control of $\mathrm{DRV}_{\mathrm{CPU}}$ and $\mathrm{DRV}_{\mathrm{PPU}}$, which govern the direction of their respective bus transceivers via the relations

$$\mathrm{DIR}_{\mathrm{CPU}} = \neg\mathrm{DRV}_{\mathrm{CPU}} \quad \text{and} \quad \mathrm{DIR}_{\mathrm{PPU}} = \neg\mathrm{DRV}_{\mathrm{PPU}}$$

where the inversion reflects the physical orientation of the transceivers on the perfboard.

The CPU data bus direction is controlled by

$$\mathrm{DRV}_{\mathrm{CPU}} = \mathrm{R}/\overline{\mathrm{W}} \wedge \neg\overline{\mathrm{ROMSEL}}_{\mathrm{filtered}}$$

This expression ensures that Sisyphus asserts the CPU data bus only during a write cycle with the address falling within the PRG ROM range. A consequence of this formulation is that PRG RAM is not supported, as Sisyphus will not drive the data bus for addresses in the 6000–7FFF range: a limitation that could be addressed by extending the drive condition to encompass the PRG RAM address range.

The PPU data bus direction is controlled by

$$\mathrm{DRV}_{\mathrm{ppu}} = \neg\overline{\mathrm{RD}}_{\mathrm{filtered}} \wedge \overline{\mathrm{PPU\ A13}}_{\mathrm{filtered}}$$

where $\overline{\mathrm{PPU\ A13}}_{\mathrm{filtered}}$ denotes the filtered, inverted PPU address line 13. This ensures that Sisyphus drives the PPU data bus only while $\overline{\mathrm{RD}}_{\mathrm{filtered}}$ is asserted, and the address falls within CHR ROM range.

Chapter 5
Results

This chapter presents the empirical findings and performance metrics of Sisyphus, detailing both the physical construction of the hardware and its operational efficiency.

5.1 Product Overview

Sisyphus constitutes an initial functional prototype of an FPGA-based NES flash cartridge, successfully demonstrating the core system pipeline from ROM loading to gameplay. To operate, Sisyphus requires a FAT-formatted user-provided SD card including a BOOT.bin executable (containing the bare-metal PS application), one partial bitstream per supported mapper, and any NES ROM files the user wishes to run.

At boot, the PS mounts the SD card and reads the first encountered .nes file, parsing its NES 2.0 header to determine the mapper number, among other necessary information to configure the mapper emulation logic. It initializes BRAM with corresponding PRG and CHR data; the PL is subsequently reconfigured via DFX partial bitstreams to instantiate the appropriate mapper. Once configured, the PL responds to all data signals from the NES in real time, correctly emulating the behavior of the original cartridge mapper. However, the implementation exhibits two primary issues: graphical glitches manifested as sprite tearing along screen edges, and game execution consistently halting related to the cartridge insertion angle.

5.2 Open-Source Distribution and Licensing

The software is distributed under the MIT License, a permissive open-source license that permits free modification and redistribution, provided that the original license terms are preserved in all derivative works [46]. The complete source code for Sisyphus, including test ROMs, schematics, and supplementary resources, is publicly available at https://git.chalmers.se/lucaz/sisyphus. The code is dual-hosted on both GitLab and GitHub to preserve academic integrity and enable future development. The Chalmers GitLab instance holds the official software regarding this thesis, while a standalone GitHub repository exists for community engagement and ongoing contributions, available at https://github.com/lucaz/sisyphus.

5.3 Functional Testing and Observed Behavior

Initial testing was conducted using the simple-store test program described in Section 4.6. During these tests, the ZedBoard consistently did not return the correct CPU data for the driven address. After introducing the synchronization pipeline across the asynchronous clock domains described in Section 4.7, both the simple-store and continuous-store test programs executed without interruption, and the CPU was able to run the test programs without observable disturbances.

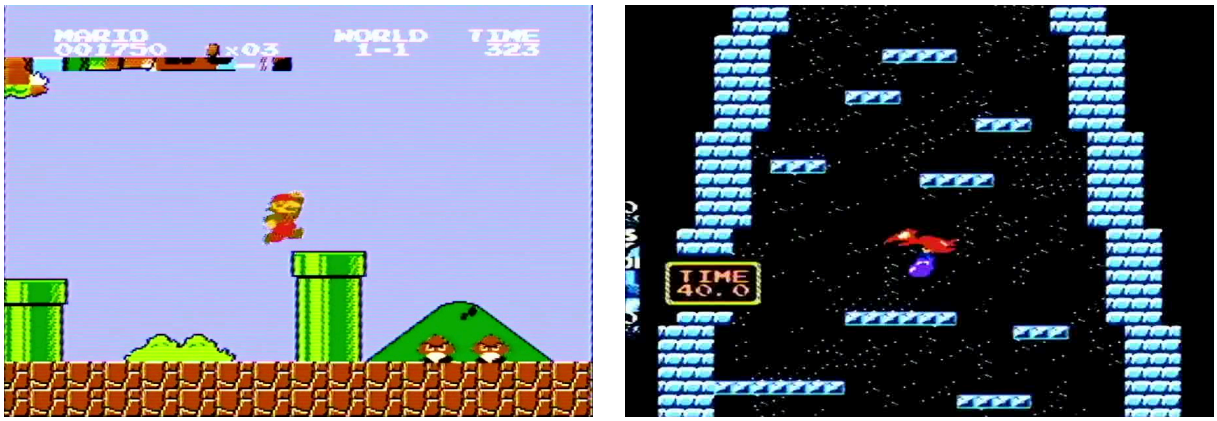
Following these tests, the rendering-control test ROM was executed. When PPU rendering was enabled, the CPU consistently executed either a JAM instruction or a software interrupt through the BRK instruction, as indicated by the ZedBoard LEDs described in Section 4.6.

In addition to the hardware verification tests described above, Sisyphus exhibited undesirable behavior when running game ROMs. Small variations in cartridge insertion angle frequently caused the CPU to halt immediately after reset due to a JAM instruction. This behavior is not exclusive to Sisyphus, as similar behavior was observed when testing an original Super Mario Bros. cartridge mounted in a customized plastic housing. In these tests, the game initially failed to render the background correctly unless physical pressure was applied to the housing. Subsequent tests were occasionally executed without issue under otherwise unchanged conditions.

However, due to an unforeseen hardware issue with the AMD debug card that prevented the system from booting while the card was connected, testing was ultimately restricted to NROM. The resulting behavior and visual artifacts from this limited testing—such as the sprite tearing in Super Mario Bros. and Ice Climber shown in Figure 15—are documented in Table 7 and Appendix B.

Game Title	**Mapper**	**Notes**
Super Mario Bros.	NROM	Minor graphical glitches (Figure 15). Consistently freezes or crashes.
Ice Climber	NROM	Minor graphical glitches (Figure 15). Consistently freezes or crashes.
Donkey Kong	NROM	No graphical glitches. Rarely freezes or crashes.

Table 7: Compatibility and functional status across NROM games.

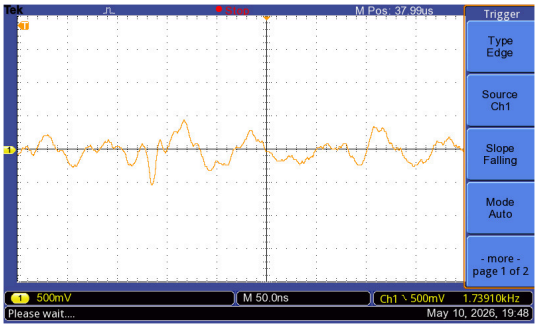


(a) Super Mario Bros. (Vertical) (b) Ice Climber (Horizontal)

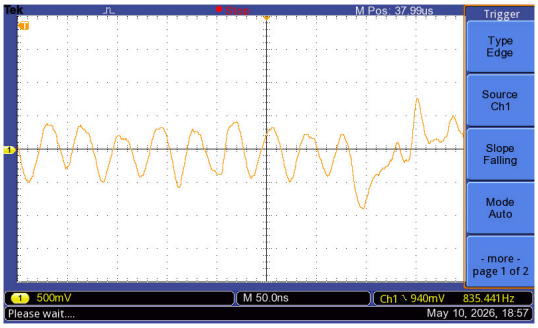
Figure 15: Consistent graphical artifact in Super Mario Bros. and Ice Climber. Note the difference in mirroring modes and the location of the glitches.

5.4 Ground Bounce Characterization

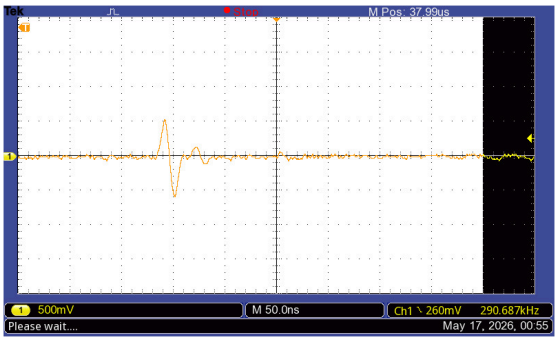
Ground bounce measurements were taken to characterize the residual instability further. Oscilloscope probes were placed between the Ricoh 2A07 GND pin and the CPU data bus transceiver GND pin with PPU while running Super Mario Bros. Without the 100 µF decoupling capacitors fitted between the cartridge connector power and ground lines, a voltage difference of up to 2 V was measured between these points. Connecting the large capacitors reduced the peak difference to approximately 1 V. Adding a 100 nF capacitor directly across the transceiver supply had no further benefit. The waveforms for both configurations, alongside a reference measurement taken from an original Super Mario Bros. cartridge, are shown in Figure 16.



(a) Sisyphus With Capacitors.



(b) Sisyphus Without Capacitors.



(c) Original Super Mario Bros. Cartridge.

Figure 16: Ground bounce measured while running Super Mario Bros.

5.5 ZedBoard Resource Utilization

Resource utilization was exported from Vivado following a complete implementation run with all three mappers included. Table 8 presents the overall utilization figures for the Artix-7 FPGA on the ZedBoard.

Resource	Utilization	Available	Utilization %
Slice LUTs	1570	53200	2.95
LUT as Logic	1563	53200	2.94
LUT as Memory	7	17400	0.04
Slice Registers	1575	106400	1.48
Block RAM Tile	96	140	68.57
IO	63	200	31.50
BUFG	1	32	3.13

Table 8: Total ZedBoard resource utilization exported from Vivado.

Table 9, Table 10, and Table 11 report per-mapper utilization within their respective DFX Pblocks. Each Pblock was sized to accommodate the largest expected mapper implementation; the figures represent an upper bound for the reconfigurable region.

Resource	Utilization	Available	Utilization %
Slice LUTs	212	2000	10.60
Slice Registers	141	4000	3.53
Unique Control Sets	18	500	3.60
Slice	116	500	23.20

Table 9: NROM Pblock utilization exported from Vivado.

Resource	Utilization	Available	Utilization %
Slice LUTs	211	2000	10.55
Slice Registers	141	4000	3.53
Unique Control Sets	18	500	3.60
Slice	137	500	27.40

Table 10: UxROM Pblock utilization exported from Vivado.

Resource	Utilization	Available	Utilization %
Slice LUTs	274	2000	13.69
Slice Registers	141	4000	3.53
Unique Control Sets	18	500	3.60
Slice	151	500	30.20

Table 11: SxROM Pblock utilization exported from Vivado.

5.6 Cartridge Compatibility and Game Coverage

Sisyphus currently imposes two hardware-level memory constraints that apply across all supported mappers: PRG RAM is not supported in any form, and the BRAM allocated for PRG ROM and CHR ROM is limited to 256 KiB and 128 KiB, respectively. These limits are a direct consequence of the Zynq-7000 BRAM budget; as shown in Table 8, BRAM is the most utilized resource at 68.6 %, leaving little headroom for expansion without architectural changes.

Because cartridges using the same mapper span a wide range of memory configurations, these constraints do not affect all titles uniformly. Table 12 summarizes the primary configurations supported by each of the three implemented mappers.

Mapper	Cartridge	PRG ROM	CHR MEM	PRG RAM	Supported
NROM	NROM	16 KiB	8 KiB	None	Yes
NROM	NROM	32 KiB	8 KiB	None	Yes
UxROM	NROM	128 KiB	8 KiB	None	Yes
UxROM	NROM	256 KiB	8 KiB	None	Yes
MMC1	SxROM	256 KiB	128 KiB	None	Yes
MMC1	SxROM	256 KiB	128 KiB	32 KiB	No
MMC1	SxROM	512 KiB	128 KiB	None or 32 KiB	No

Table 12: Cartridge configuration support across implemented mappers.

According to the mapper frequency distribution listed in Table C1, NROM, UxROM, and SxROM together account for 1202 games, approximately 49 % of the entire surveyed library. MMC1 is the single most common mapper in the catalogue, with around a 28 % share, reflecting its use across a large portion of Nintendo's first-party output. However, as described above, the memory constraints imposed by the current BRAM budget mean that this headline figure overstates effective compatibility: SxROM titles requiring PRG RAM, or those using the 512 KiB PRG ROM variant, are excluded.

Chapter 6
Discussion

This chapter encapsulates the points worth discussing within the works of the project. About the final results and how they compare to the initial goals of the project, along with possible future improvements that were not implemented due to constraints.

6.1 Analysis of Observed Behavior

As observed in Section 5.3, the test sequence carried out during this project revealed a coherent chain of causation linking the perfboard's physical construction to the instability observed whenever the PPU began rendering. The root cause is believed to be the increased current draw caused by the PPU transition from idle to active rendering. In its idle state, the PPU makes no back-to-back memory accesses; once background rendering is enabled, the PPU performs two consecutive memory cycles every PPU clock period. This sustained switching activity increases the dynamic current demand of both the PPU itself and the bus transceivers on the address and data lines.

According to Faraday's law, any inductance in a current path produces a voltage drop proportional to the rate of change of current: $V = L\frac{\mathrm{d}I}{\mathrm{d}t}$. This inductive voltage drop manifests as a momentary rise in local ground potential—ground bounce—exacerbated by the perfboard's lack of a proper ground plane. This was confirmed by oscilloscope measurement seen in Figure 16, which showed a peak voltage differential of up to 2 V between the cartridge connector ground and the CPU ground without decoupling capacitors, reduced to approximately 1 V after large 100 µF capacitors were added across the cartridge connector power lines.

A significant enough common ground differential between the Ricoh 2A07 IC and the corresponding data bus transceiver can cause data corruption. If the CPU latches a corrupt byte interpreted as an opcode, program execution will immediately deviate. This will either result in a spurious interrupt due to a BRK instruction (opcode \$00) or eventually halt the CPU due to an illegal JAM instruction—in line with observations from Section 5.3.

While these issues were partly alleviated by the synchronization mechanism explained in Section 4.7, the remaining issues can only be solved by introducing a continuous ground plane, controlled-impedance traces, and appropriately placed decoupling capacitors, discussed later in Section 6.4. Additionally, the consistent sprite tearing observed along screen edges in Super Mario Bros. and Ice Climber, shown in Figure 15, is not easily attributable to ground bounce. Because the tearing is a persistent and spatially consistent, a more plausible explanation is a timing error arising from $\mathrm{CIRAM}_{\mathrm{A10}}$ being derived from the latched PPU address rather than the raw signal.

6.2 Limitations of FPGA-Based Cartridge Emulation

Hardware and the emulation of hardware have been at the forefront throughout the development of Sisyphus. Naturally, this has meant utilizing FPGAs and their capability to be reconfigured, as described in Section 3.1. However, although hardware emulation is a prevalent use case for FPGAs, they are not without inherent limitations.

Replicating the functional behavior of the original NES cartridges has resulted in a clash, stemming from a contrast in computer architecture choices of 1980s devices and modern FPGAs. The CHR and PRG ROMs on cartridges operate unclocked and are asynchronous. In comparison, FPGAs like the one used in this project rely on strictly synchronous memory. This meant that Sisyphus had to synchronize the NES's asynchronous CPU and PPU to the FPGA's synchronous BRAM. This produced a complex and unique problem that had to be bypassed using creative measures, such as a 2-stage synchronizer, described in Section 4.7. While being effective, the synchronization produces a delay of 40 ns. This all stems from the fact that the FPGA uses synchronized BRAM, making it an unignorable limitation of the technology. A natural alternative to avoid the above-mentioned complication is to simply use external asynchronous RAM. However, although this would bypass the need for a synchronizer, it would instead introduce other challenges, for example, how to send the data to the RAM, and would make the system more complicated.

Another limiting factor is the fact that the BRAM is limited and can not be expanded. As mentioned in Section 5.6, BRAM is the most utilized resource and even limits the possibility to support SxROM cartridges with large PRG ROMs. However, this is specific to our hardware, and the simplest solution is to use an FPGA with more BRAM.

Furthermore, a general problem for FPGAs is the issue of emulation only being digital, as outlined in section Section 3.1, meaning that analog logic cannot be emulated correctly. This means that FPGAs cannot natively represent high-impedance states. As a consequence, tri-state logic behavior, something necessary for NROM, cannot be achieved internally. Therefore, a one-to-one copy of it is not possible. Instead, Sisyphus's implementation of NROM circumvents this issue by avoiding the usage of a tri-state buffer and instead mimicking the logic of a tri-state buffer.

In summary, it is undeniable that FPGAs are not without their limitations, yet they remain highly practical for hardware emulation. For this project, the requirements were within the margins of the FPGA's above-discussed limitations. However, one could conclude that expanding further on this would likely mean that deviations need to be taken.

6.3 Practical Application

In its current state, it is difficult to argue for Sisyphus as a practical alternative to existing solutions. Game execution remains too unstable for reliable use, and mapper support covers only a fraction of the library. Beyond the mappers implemented, a hard architectural constraint further limits compatibility: the ZedBoard provides 630 KiB of BRAM when organized as byte-addressable. This restricts not only which mappers

can be supported, but also which cartridge configurations within already-supported mappers are viable, as seen with MMC1 in Table 12. This is the biggest limitation preventing Sisyphus from becoming a useful product.

Compared to the EverDrive, Sisyphus is both more expensive and less functional, with the EverDrive offering a cheaper, more polished experience, including save states, integrated game selection menus, and substantially broader compatibility. However, Sisyphus is not intended to compete with the EverDrive. Its practical utility is better understood through its potential as an open platform for the development community rather than as a consumer product. Because it's based on the ZedBoard using standard development tools, the barrier to contribution is substantially lower than for proprietary alternatives. Furthermore, AMD Zynq-7000 SoC provides access to AMD's full IP library alongside standard software libraries, making development faster and more approachable, particularly for those already familiar with the platform.

6.4 Future Work

To provide greater flexibility for developers using Sisyphus, a broader range of cartridge configurations should be supported. Given that mapper usage within the NES library is highly non-uniform, it is reasonable to prioritize implementation based on the prevalence of each configuration, targeting those used by the largest number of games first. Following this approach, the next logical candidates are cartridges using mappers 3 and 4, which—when combined with the already implemented mappers—would increase coverage to approximately 80% of the NES library, as illustrated in Figure 17.

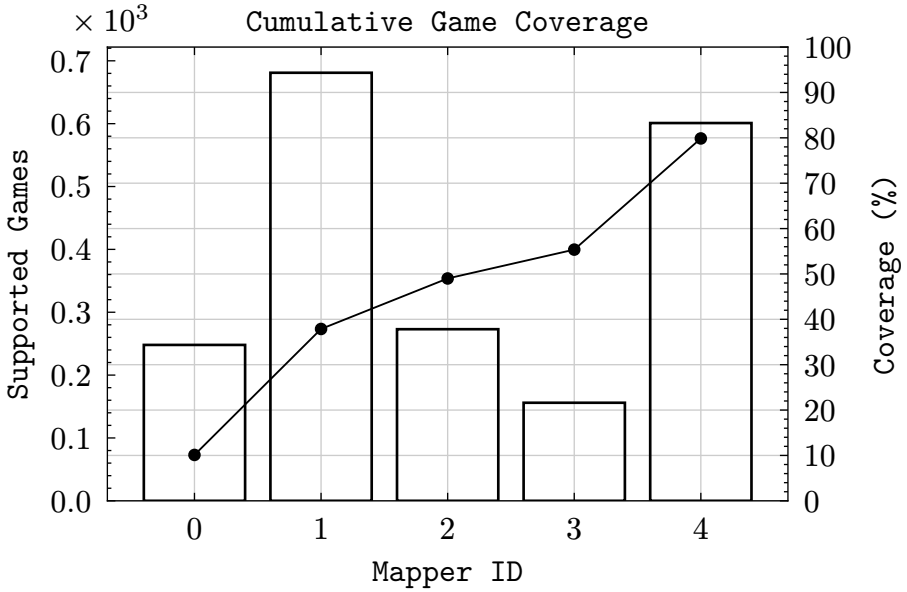


Figure 17: Cumulative game coverage of mappers 0, 1, 2, 3, and 4 . See Table C1 for the complete dataset. [26]

Another metric for deciding subsequent cartridge implementations is to prioritize unique or technically interesting mappers. This approach would provide developers with a broader set of distinctly different options when using Sisyphus, enabling them to select configurations that better match specific technical requirements or design goals.

Furthermore, the current design, built around a cartridge breakout board and perfboard combination, is needlessly complex and difficult to reproduce without significant manual labor. A dedicated PCB featuring an integrated edge connector with dedicated footprints for level shifters would be more compact, improve signal integrity, and be more convenient to work with. Furthermore, extending the PCB to a fully fledged Mezzanine Card—directly incorporating an FMC LPC male connector—would render the currently used AMD FMC XM105 Debug Card obsolete. While more convenient to use, that would be significantly harder to develop and manufacture.

6.5 Software License and Ethical Aspects

The MIT License was chosen primarily due to its common use for open-source software. Its widespread ubiquity ensures that both individual developers and corporate entities are already familiar with its terms, reducing legal friction for adoption. Furthermore, our choice was influenced by the project's dependencies. Specifically, we utilized code authored by Krikzz [41] for bypassing the CIC. As Krikzz's original work is distributed under the MIT license, adopting the same license made sense in terms of compatibility and honors the original author's contribution while maintaining a consistent licensing structure across the codebase.

The primary purpose of Sisyphus is to provide a versatile cartridge emulation platform for user-provided ROM files, a functionality that inherently raises concerns regarding software piracy. Because the platform allows users to load ROMs they may not legally own, there is a risk that some individuals might bypass legitimate purchases in favor of unauthorized downloads. While such use is contrary to the project's intent, the potential impact on the retro gaming market—where piracy could diminish the already limited sales of physical media—is a significant consideration. Therefore, we discourage piracy and expect users to load only ROMs they legally own. When ROMs are purchased legitimately, copyright holders receive their intended revenue, so lawful use does not financially disadvantage them. Ultimately, while our goal is to foster the retro gaming community and attract new enthusiasts, the project cannot regulate or control individual user behavior.

Furthermore, while unauthorized distribution is illegal, many argue that digital preservation has been instrumental in safeguarding titles that are no longer commercially available and whose physical iterations are succumbing to degradation. When approached responsibly, these preservation efforts help sustain the retro gaming ecosystem and ensure that gaming history remains accessible for future generations.

6.6 Use of Generative AI

Generative artificial intelligence—specifically, large language models (LLMs) such as ChatGPT, Gemini, and Claude—played an important role in the development of Sisyphus and the composition of this thesis. These tools were utilized primarily for technical support, including code generation and debugging, as well as for assisting in understanding complex topics to establish a foundation for further research. Additionally, LLMs assisted in refining the prose of this document through iterative writing and paraphrasing. To ensure academic integrity, all AI-generated suggestions were critically evaluated and verified for accuracy against primary sources in accordance with Chalmers AI policy [47].

Chapter 7
Conclusion

The purpose of the project was to assess the feasibility of developing an open-source, hardware-based NES flash cartridge and to provide an open platform for further research. Evaluated against that goal, the project demonstrates that the proposed architecture is conceptually sound. A functional end-to-end pipeline was achieved, and core sub-systems, including the synchronization pipeline and NROM mapper logic, operate correctly in isolation. However, it also exposes significant practical limitations that prevent Sisyphus from serving as a ready alternative to existing solutions. Full operational viability, meaning stable and sustained game execution, remains undemonstrated, where signal integrity problems in the physical perfboard assembly cause most tested titles to crash, and the two other implemented mappers (UxROM and MMC1) could not be validated on hardware due to a hardware fault in the debug card.

The current physical assembly of Sisyphus is therefore the primary barrier for achieving sufficient signal stability, and the most immediate objective is replacing it with a custom, dedicated PCB featuring a proper ground plane and controlled-impedance traces. Additionally, broader cartridge compatibility is restricted by the ZedBoard's limited BRAM capacity. Despite these limitations, Sisyphus constitutes a proof of concept that validates the architectural approach to FPGA-based cartridge emulation, establishing a solid foundation. Such a foundation could ultimately contribute to broader efforts aimed at the preservation and continued accessibility of retro gaming consoles.

Bibliography

[1] N. Altice, *I Am Error: The Nintendo Family Computer / Entertainment System Platform*. The MIT Press, 2015, pp. 5–38. doi: 10.7551/mitpress/9484.003.0003.

[2] M. Yukawa, "Video game control unit." [Online]. Available: https://patents.google.com/patent/USD299726S

[3] "Everdrive N8 Pro." [Online]. Available: https://www.nesdev.org/wiki/Everdrive_N8_Pro

[4] "CPU." [Online]. Available: https://www.nesdev.org/wiki/CPU

[5] "Cycle reference chart." [Online]. Available: https://www.nesdev.org/wiki/Cycle_reference_chart

[6] "PPU frame timing." [Online]. Available: https://www.nesdev.org/wiki/PPU_frame_timing

[7] "Team6502." [Online]. Available: https://www.team6502.org/

[8] "CPU pinout." [Online]. Available: https://www.nesdev.org/wiki/CPU_pinout

[9] "6502 Instruction Set." [Online]. Available: https://www.masswerk.at/6502/6502_instruction_set.html

[10] "CPU unofficial opcodes." [Online]. Available: https://www.nesdev.org/wiki/CPU_unofficial_opcodes

[11] Synertek, "SY6500 8-Bit Microprocessor Family Datasheet," technical report. [Online]. Available: https://www.princeton.edu/~mae412/HANDOUTS/Datasheets/6502.pdf

[12] "Cartridge connector." [Online]. Available: https://www.nesdev.org/wiki/Cartridge_connector

[13] P. Diskin, "Nintendo Entertainment System Documentation," technical report, 2004. [Online]. Available: https://archive.nesdev.org/NESDoc.pdf

[14] "PPU Memory Map." [Online]. Available: https://www.nesdev.org/wiki/PPU_memory_map

[15] "PPU registers." [Online]. Available: https://www.nesdev.org/wiki/PPU_registers

[16] B. Taylor, "2C02 technical reference," technical report, 2004. [Online]. Available: https://www.nesdev.org/2C02%20technical%20reference.TXT

[17] "PPU pinout." [Online]. Available: https://www.nesdev.org/wiki/PPU_pinout

[18] K. Nakagawa, "System for determining authenticity of an external memory used in an information processing apparatus." United States Patent, Trademark Office (USPTO), pp. 1–18, 1989.

[19] K. Nakagawa and M. Yukawa, "Memory cartridge and information processor unit using such cartridge." [Online]. Available: https://patents.google.com/patent/US4865321A

[20] "CPU Memory Map." [Online]. Available: https://www.nesdev.org/wiki/CPU_memory_map

[21] "Bus conflict." [Online]. Available: https://www.nesdev.org/wiki/Bus_conflict

[22] "Mapper." [Online]. Available: https://www.nesdev.org/wiki/Mapper

[23] "NROM." [Online]. Available: https://www.nesdev.org/wiki/NROM

[24] "UxROM." [Online]. Available: https://www.nesdev.org/wiki/UxROM

[25] "MMC1." [Online]. Available: https://www.nesdev.org/wiki/MMC1

[26] "NES Cartridge Database." [Online]. Available: https://nescartdb.com/

[27] "Comparison of Nintendo Mappers." [Online]. Available: https://www.nesdev.org/wiki/Comparison_of_Nintendo_mappers

[28] "iNES." [Online]. Available: https://www.nesdev.org/wiki/INES

[29] "NES 2.0." [Online]. Available: https://www.nesdev.org/wiki/NES_2.0

[30] M. Z. M. H. Farooq U., "FPGA Architectures: An Overview," *Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*. Springer New York, New York, NY, p. 11, 2012. doi: 10.1007/978-1-4614-3594-5_2.

[31] M. Z. M. H. Farooq U., "Heterogeneous Architectures Exploration Environments," *Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*. Springer New York, New York, NY, pp. 85–86, 2012. doi: 10.1007/978-1-4614-3594-5_4.

[32] F. Umer, M. Zied, and M. Habib, "FPGA Architectures: An Overview," *Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*. Springer New York, pp. 7–48, 2012. doi: 10.1007/978-1-4614-3594-5_2.

[33] M. Gokhale and P. S. Graham, *Reconfigurable computing : accelerating computation with field-programmable gate arrays*. Springer, 2005, pp. 7–73.

[34] M. Z. M. H. Farooq U., "FPGA Architectures: An Overview," *Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*. Springer New York, New York, NY, pp. 24–39, 2012. doi: 10.1007/978-1-4614-3594-5_2.

[35] Abbey1, "Managing mean time between failure in Xilinx devices." 2020.

[36] J. Fan, X. Ye, J. Kim, B. Archambeault, and A. Orlandi, "Signal integrity design for high-speed digital circuits: Progress and directions," *IEEE Transactions on Electromagnetic Compatibility*, vol. 52, no. 2, pp. 392–400, 2010, doi: 10.1109/TEMC.2010.2045381.

[37] M. Crews and Y. Yuenyongsgool, "Practical design for transferring signals between clock domains," *EDN*, vol. 48, no. 4, p. 65, 2003.

[38] K. M. R. G. Venkata Rao K.; Rama Sudha, *Pulse and Digital Circuits*. New Delhi: Pearson Education / Pearson India, 2010, p. .

[39] M. Mccaughey, E. Maier, and C. Spurlin, "Voltage Translation Between 3.3-V, 2.5-V, 1.8-V, and 1.5-V Logic Standards," technical report, 2002.

[40] Diligent, "ZedBoard (Zynq™ Evaluation and Development) Hardware User's Guide," technical report, 2014. [Online]. Available: https://files.digilent.com/resources/programmable-logic/zedboard/ZedBoard_HW_UG_v2_2.pdf

[41] krikzz, "avrciczz." [Online]. Available: https://github.com/krikzz/avrciczz/tree/259adceb3e6ca97b0ec9911e0194912cc0bb9739?

[42] "Arduino as ISP and Arduino Bootloaders." [Online]. Available: https://docs.arduino.cc/built-in-examples/arduino-isp/ArduinoISP/

[43] Texas Instruments, "SN74LVC245A Octal Bus Transceiver With 3-State Outputs," 2015, [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74lvc245a.pdf

[44] AMD, "Vivado Design Suite User Guide: Dynamic Function eXchange." [Online]. Available: https://docs.amd.com/r/en-US/ug909-vivado-partial-reconfiguration/Introduction

[45] AMD, "Block Memory Generator v8.4," technical report, 2021. [Online]. Available: https://docs.amd.com/v/u/en-US/pg058-blk-mem-gen

[46] Open Source Initiative, "The MIT License." [Online]. Available: https://opensource.org/license/mit

[47] "Regulations for the use of AI tools in thesis work," [Online]. Available: https://www.chalmers.se/en/education/your-studies/masters-and-bachelors-thesis/regulations-for-the-use-of-ai-tools/

[48] "List of mappers." [Online]. Available: https://www.nesdev.org/wiki/List_of_mappers

Appendix A
Supplementary Component Diagrams

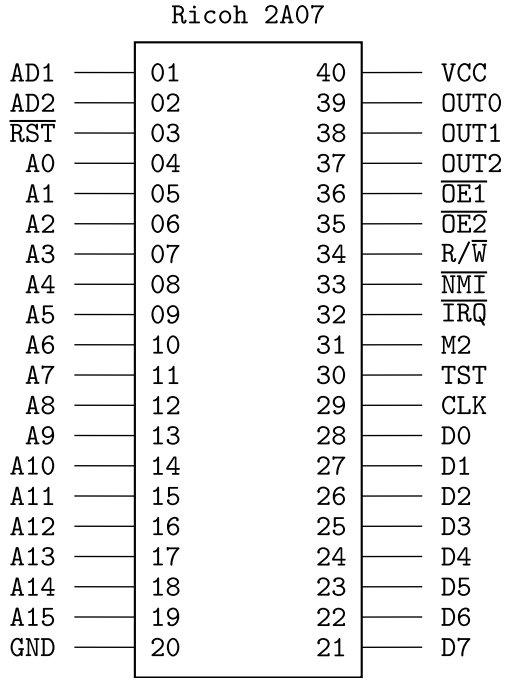


Figure A1: Ricoh 2A07 pinout [8].

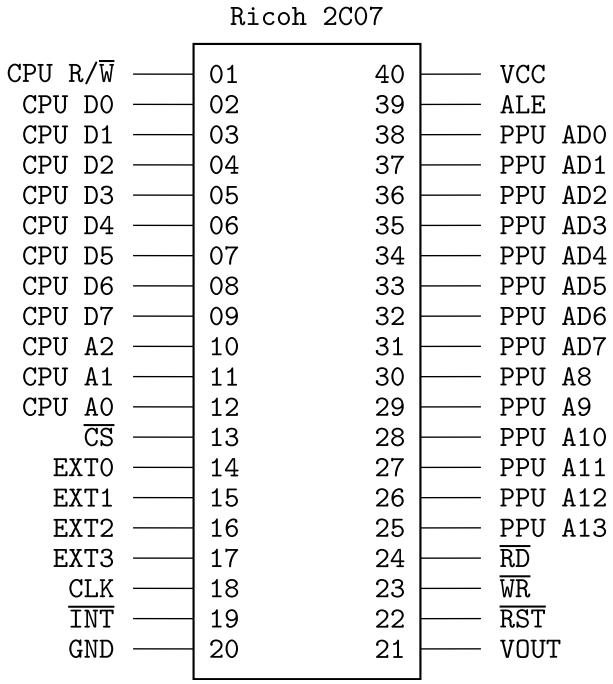


Figure A2: Ricoh 2C07 pinout [17].

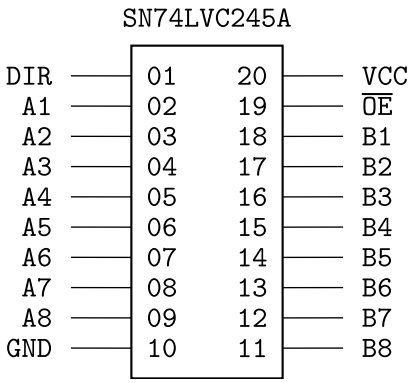


Figure A3: SN74LVC245A bus transceiver pinout [43].

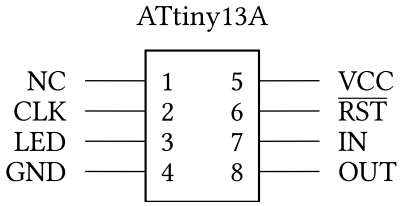


Figure A4: ATtinyA13 pinout as connected using Krikzz CIC implementation [41].

Appendix B
Miscellaneous Graphical Anomalies

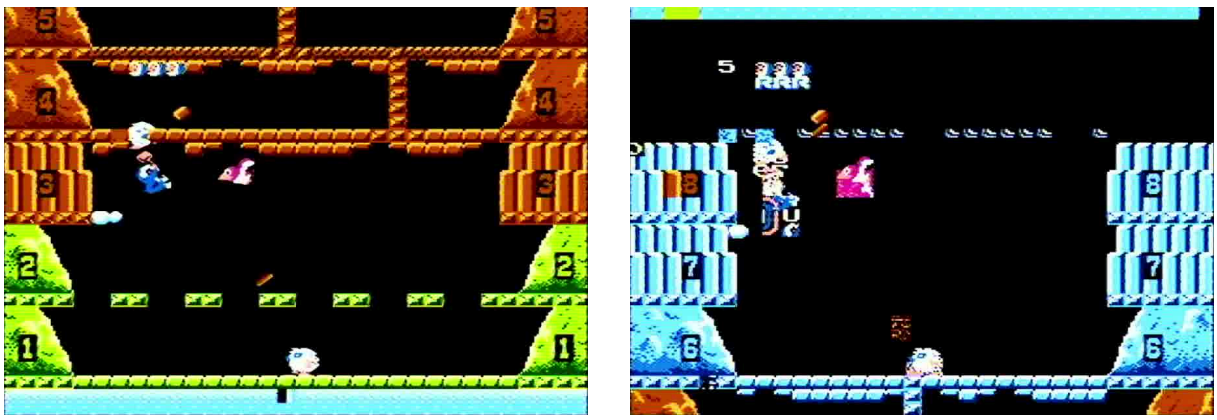
The anomalies detailed in this section do not represent consistent or repeatable behavior, documenting instead unique, isolated occurrences. These examples are intended to illustrate the types of irregularities that may emerge when using Sisyphus, rather than offer diagnostic insights as to their underlying causes.

Figure B1: Glitch where the sprite of a Goomba is replaced by that of Bowser. Gameplay remains fully functional.

Figure B2: Super Mario Bros. crash example 1.

Figure B3: Super Mario Bros. crash example 2.

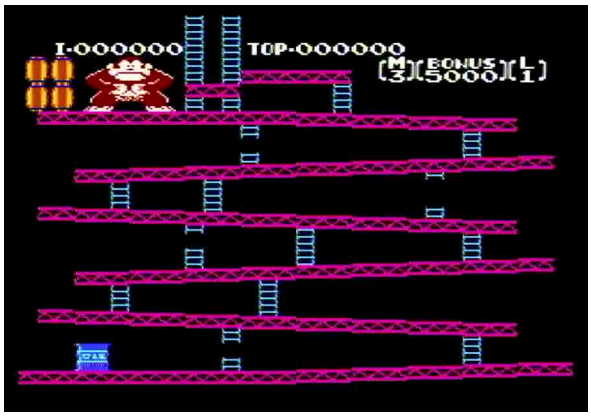
Figure B4: Super Mario Bros. crash example 3.



(a) Before (b) After

Figure B5: Ice Climber crash.

(a) Normal menu



(b) Normal gameplay

(c) Menu glitch

Figure B6: Donkey Kong glitch that superimposes gameplay onto start menu.

Appendix C

Comprehensive List of Mappers

ID	Common Designation(s)	Frequency	Share %
0	NROM	248	10.11
1	SxROM, MMC1	681	27.76
2	UxROM	273	11.13
3	CNROM	156	6.36
4	TxROM, MMC3, MMC6	601	24.5
5	ExROM, MMC5	25	1.02
7	AxROM	75	3.06
9	PxROM, MMC2	11	0.45
10	FxROM, MMC4	3	0.12
11	Color Dreams	34	1.39
13	CPROM	1	0.04
16	Bandai EPROM (24C02)	15	0.61
18	Jaleco SS8806	15	0.61
19	Namco 163	20	0.82
21	VRC4a, VRC4c	2	0.08
22	VRC2a	2	0.08
23	VRC2b, VRC4e	10	0.41
24	VRC6a	1	0.04
25	VRC4b, VRC4d	6	0.24
26	VRC6b	2	0.08
32		6	0.24
33		9	0.37
34	BNROM, NINA-001	5	0.2
37		3	0.12
38		1	0.04
41		1	0.04
47		1	0.04
48		2	0.08
64	RAMBO-1	5	0.2
65		3	0.12
66	GxROM, MxROM	17	0.69
67		2	0.08
68	After Burner	4	0.16
69	FME-7, Sunsoft 5B	15	0.61

ID	Common Designation(s)	Frequency	Share %
70		2	0.08
71	Camerica/Codemasters	16	0.65
72		3	0.12
73	VRC3	1	0.04
75	VRC1	6	0.24
76	Namco 109 variant	1	0.04
77		1	0.04
78		2	0.08
79	NINA-03/NINA-06	16	0.65
80		7	0.29
82		5	0.2
85	VRC7	2	0.08
86	JALECO-JF-13	1	0.04
87		10	0.41
88		3	0.12
89		1	0.04
92		1	0.04
93		2	0.08
94	Senjou no Ookami	1	0.04
95		1	0.04
96		1	0.04
97		1	0.04
105	NES-EVENT	1	0.04
118	TxSROM, MMC3	7	0.29
119	TQROM, MMC3	8	0.33
140		3	0.12
141		1	0.04
144		1	0.04
146		1	0.04
152		4	0.16
154		1	0.04
158		1	0.04
159	Bandai EPROM (24C01)	4	0.16
168		1	0.04
180	Crazy Climber	1	0.04
184		3	0.12
185	CNROM with protection diodes	8	0.33

ID	Common Designation(s)	Frequency	Share %
189		1	0.04
193		2	0.08
206	DxROM, Namco 118 / MIMIC-1	45	1.83
207		1	0.04
210	Namco 175 and 340	12	0.49
228	Action 52	1	0.04
232	Camerica/Codemasters Quattro	5	0.2
234		2	0.08

Table C1: Frequency distribution of NES games by mapper [26], [48]. Note that games with multiple versions (for example regional releases) are considered separate entries.

Appendix D
Schematic

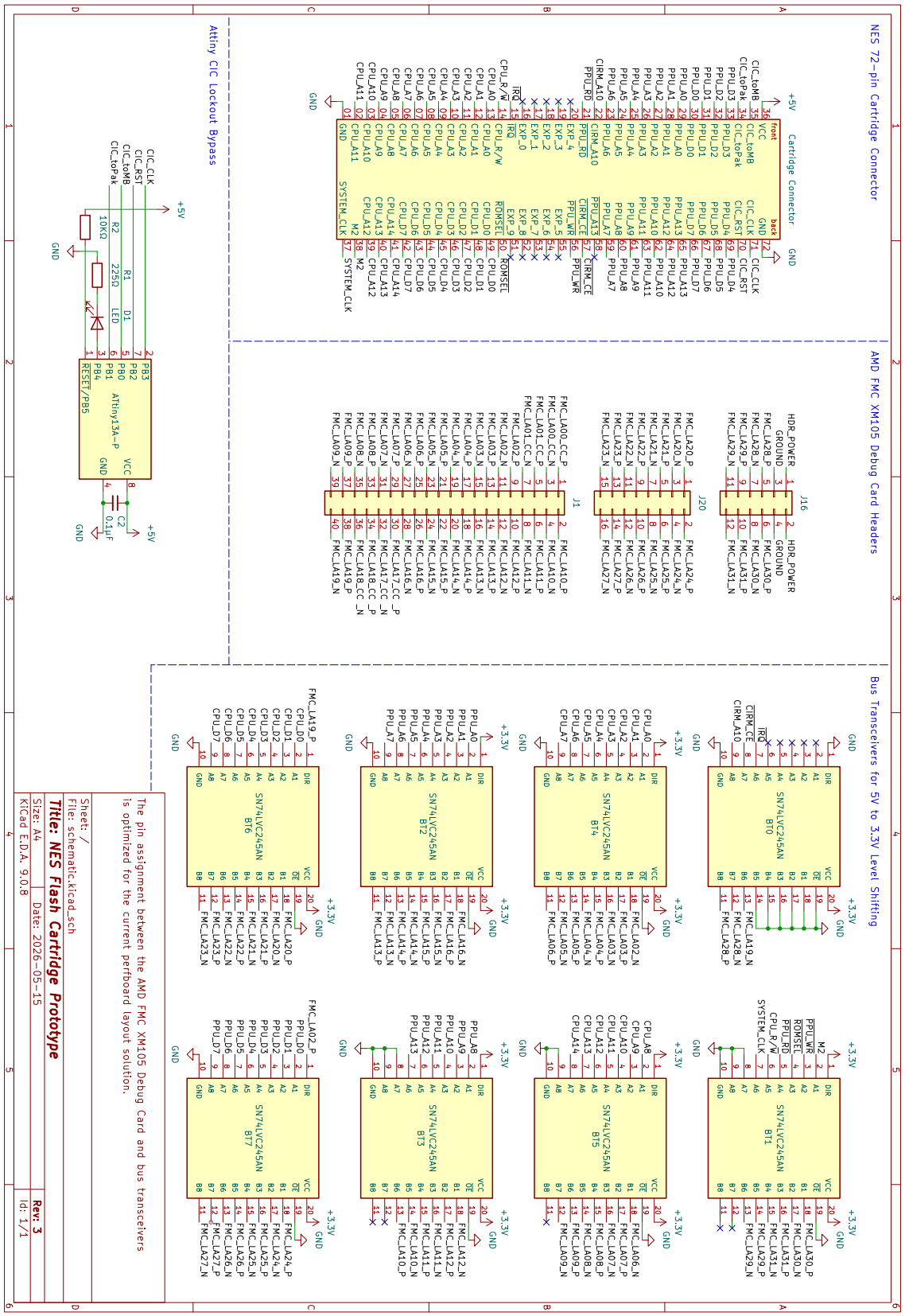


Figure D1: Schematic of Sisyphus.

COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY