جامعة الأخوين

ⵜⴰⵙⴷⴰⵡⵉⵜ ⵏ ⵓⵅⴰⵡⴰⵢⵏ

AL AKHAWAYN UNIVERSITY

School of Science and Engineering

Reverse Engineering the Nintendo 64: Building a JavaScript based Emulator.

Capstone Design Final Report

April 2025

Salma Mahidat

Supervised by:

Dr. Kevin Smith

© SCHOOL OF SCIENCE & ENGINEERING – AL AKHAWAYN UNIVERSITY

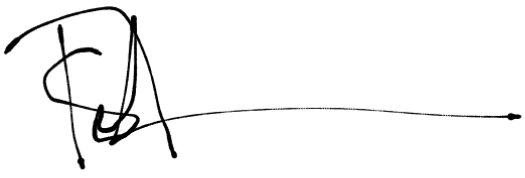
Student conduct and copyright

The supervisor and the student (the Capstoner) agree that:

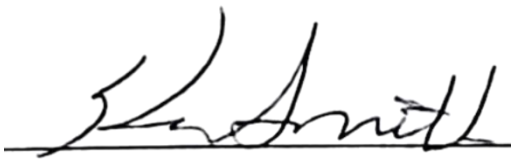
1. Permission has been obtained for any third party content (eg Data, illustrations, photographs, charts or maps).

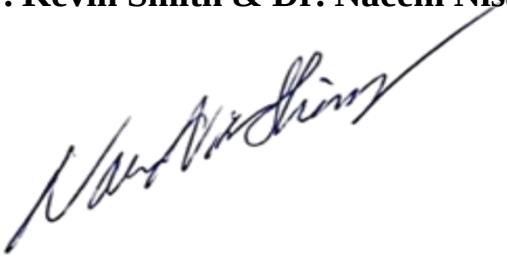
2. The results described in this report have not previously been published

Student's name and signature: Salma Mahidat



Supervisor's name and signature: Dr. Kevin Smith & Dr. Naeem Nisar Sbeikh





Reverse Engineering the Nintendo 64: Building a JavaScript based Emulator.

Capstone Report

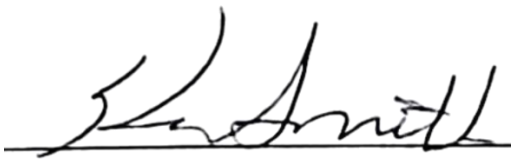
Student Statement:

I, Salma Mahidat, affirm that I have applied ethics to the design process and in the selection of the final proposed design. And that, I have held the safety of the public to be paramount and have addressed this in the presented design wherever may be applicable.

Salma Mahidat

Student's name

Approved by the Supervisor



Dr. Kevin Smith

ACKNOWLEDGEMENTS

Completing this capstone has been not only an academic challenge but also a profound journey of personal growth.

First and foremost, I would like to thank my parents and my little sister, Sara. Their unwavering belief in me, their sacrifices, and their constant encouragement have been the foundation of everything I have achieved. I am endlessly grateful for the values they taught me and the love and support they have given me throughout my life.

I am also incredibly thankful to my family and friends for their constant support, encouragement, and patience throughout this journey.

I owe my deepest gratitude to my supervisor, Dr. Kevin Smith, for his invaluable guidance, feedback, and encouragement throughout the development of this project. His support gave me direction when I needed it most and pushed me to reach higher standards.

A special thanks also goes to Professor Naeem Nisar Sheikh for encouraging me to explore this side of Computer Science and inspiring me to take on this project, as well as to all the professors at the School of Science and Engineering. Their teaching, mentorship, and support over the years have been instrumental in shaping my academic journey and preparing me for challenges like this one.

This project is a reflection not only of my efforts, but of the belief, patience, and kindness of all those who stood by me. I am forever grateful.

CONTENT

ACKNOWLEDGEMENTS **2**

LIST OF FIGURES **6**

LIST OF TABLES **7**

ABSTRACT (ENGLISH) **8**

RESUME (FRENCH) **9**

Introduction **10**

Problem Statement **11**

Project Specifications **12**

- **Objective of the Project** **12**
- **Functional Requirements** **12**
- **Non-Functional Requirements** **13**

Steeple Analysis **14**

Engineering Standards **17**

Logic Model Framework **18**

- **Target Populations** **18**
- **Underlying Assumptions** **19**
- **Resources/Challenges** **19**
  - **Resources** **19**
  - **Challenges** **19**
- **Activities** **20**
- **Outputs Of the Project** **21**
- **Outcomes** **21**

Literature Review **22**

Methodology & Capstone Design **26**

- **Nintendo 64 Architecture Overview** **26**
- **Software Stack & Tools** **41**
  - **Core Technologies** **42**
  - **Build and Runtime Environment:** **42**
  - **Libraries and Frameworks** **42**
- **Emulation Techniques & Algorithms** **42**
  - **How is the Boot process emulated?** **42**
  - **CIC Chip Behavior** **43**
  - **Regional Differences** **44**
  - **Decoding44**
  - **Dynamic Recompilation** **46**
  - **Recompilation Process** **47**

Fragment Management & Execution Context **48**
Handling Trivial vs. Non-Trivial Instructions **49**
Branch Handling & Delay Slots **50**
Caching & Performance Considerations **50**
Memory Management and Addressing **51**
Address Translation and Fast Paths **52**
Masked Writes and Bit Manipulation **53**
Memory Safety and Exception Handling **53**
RSP and RDP Emulation **54**
RSP Emulation **54**
RDP Emulation (via HLE) **55**
Graphics Math Utilities **56**
Audio System **57**
Controller Input **58**
Debugging Tools / Developer Features **59**
User Interface **60**
Key Design Decisions **62**
Why Dynamic Recompilation? **62**
Rationale for using JavaScript **65**
Modular Architecture **65**
Tradeoffs & Limitations **66**
Performance vs. Accuracy **66**
Why Cycle Accuracy Matters — And Why It Was Deferred **67**
Planned Features & Improvements **70**
Data Presentation **71**
Homebrew Testing **71**
Performance Evaluation: Project64 on Ryzen 5 5600H + RTX 3050 Laptop **73**
Performance Evaluation: JS64 on Ryzen 5 5600H + RTX 3050 Laptop **76**
Simulations, Results, & Interpretation **79**
Overview **79**
Results **79**
Correctness Validation: Homebrew Testing **80**
Technical Implications **81**
Learning Strategies **81**
Building a Foundation with Simpler Systems **81**
Learning from Existing Emulators **82**
Studying Computer Architecture **82**
Consulting Official Documentation **83**

Conclusion **83**
References **85**

LIST OF FIGURES

Figure 1 Nintendo 64 Motherboard with important parts labelled.....26
Figure 2 Main Nintendo 64 architecture diagram.....27
Figure 3 The MIPS R4300i CPU (1995).....28
Figure 4 Memory Layout of the N64 System.....31
Figure 5 Expansion Pak [9], an optional accessory sold separately (sometimes included with the game that requires it).....32
Figure 6 Jumper Pak. In the absence of the Expansion Pak, this must be present to terminate the RDRAM bus.....32
Figure 7 Architecture of the Reality Signal Processor (RSP).....34
Figure 8 Architecture of the Reality Display Processor (RDP).....36
Figure 9 Primitive view of a Super Mario 64 Scene......38
Figure 10 Rendered frame.....39
Figure 11 Overview of how the audio pipeline is often programmed.....40
Figure 12 N64 CIC Chip.....41
Figure 13 MIPS instruction formats.....44
Figure 14 N64 Controller.....58
Figure 15 js64 User Interface.....60
Figure 16 Controller Mapping Configuration.....61
Figure 17 Debug and Profiling Tools.....62
Figure 18 Performance metrics panel in js64. Toggle displays real-time FPS (frames per second), MS (millisecond latency per frame), and MB (memory usage) to monitor emulation efficiency.....62
Figure 19 Chip8 Interpreter Example.....64
Figure 20 Left - Cycle-accurate emulation with the aircraft shadow displayed. Right - Performance-optimized emulation missing the shadow due to skipped raster effects.....68
Figure 21 Output of n64-systemtest 2.0.2 validation suite.....72
Figure 22 Frame rate (FPS) performance over time during Project 64 emulation.....73
Figure 23 Frametime analysis (ms) with during Project 64 emulation.....74
Figure 24 Frame delivery quality analysis during Project 64 emulation.....76
Figure 25 Frametime analysis (ms) during js64 emulation.....77
Figure 26 Frame rate (FPS) performance over time during js64 emulation.....78
Figure 27 Frame delivery quality analysis during js64 emulation.....79

LIST OF TABLES

Table 1 MIPS instruction types, use cases, and examples 45
Table 2 Performance Benchmark: jS64 vs Project64 80

ABSTRACT (ENGLISH)

This project explores the development of a Nintendo 64 emulator, **js64**, built entirely in **JavaScript** to run within modern web browsers without requiring installation. Unlike many existing emulators that prioritize speed over fidelity, JS64 aims to find a balance between **performance**, **accuracy**, portability, and educational value.

A dynamic recompiler was developed to convert MIPS VR4300 instructions into optimized JavaScript functions at runtime, resulting in considerable performance improvements over traditional interpretation. Additionally, the emulator incorporates a high-level rendering pipeline using WebGL, and provides built-in debugging tools to support emulator development and educational analysis.

Testing with the n64-systemtest ROM resulted in a 95% correctness pass rate, validating accurate CPU and memory emulation for most standard operations. Performance evaluation against Project64 shows that although JS64's performance remains constrained by browser overhead, it achieves functional emulation of commercial games at playable framerates.

This project illustrates that current web technologies are sufficiently robust to emulate intricate systems like the Nintendo 64, paving the way for improved software preservation, accessibility, and educational opportunities.

Keywords: Emulation, Nintendo 64, JavaScript, Dynamic Recompilation, WebGL

RESUME (FRENCH)

Ce projet explore le développement d'un émulateur de Nintendo 64, js64, construit entièrement en JavaScript pour fonctionner dans les navigateurs web modernes sans avoir besoin d'installation. Contrairement à de nombreux émulateurs existants qui privilégient la vitesse à la fidélité, JS64 vise à trouver un équilibre entre les performances, la précision, la portabilité et la valeur éducative.

Un recompilateur dynamique a été développé pour convertir les instructions du MIPS VR4300 en fonctions JavaScript optimisées au moment de l'exécution, ce qui permet d'améliorer considérablement les performances par rapport à l'interprétation traditionnelle. En outre, l'émulateur intègre un pipeline de rendu de haut niveau utilisant WebGL, et fournit des outils de débogage intégrés pour soutenir le développement de l'émulateur et l'analyse pédagogique.

Les tests effectués avec la ROM n64-systemtest ont permis d'obtenir un taux de réussite de 95 %, validant une émulation précise du processeur et de la mémoire pour la plupart des opérations standard. L'évaluation des performances par rapport à Project64 montre que, bien que les performances de JS64 restent contraintes par la surcharge du navigateur, il parvient à une émulation fonctionnelle des jeux commerciaux à des taux de rafraîchissement jouables.

Ce projet montre que les technologies web actuelles sont suffisamment robustes pour émuler des systèmes complexes comme la Nintendo 64, ce qui ouvre la voie à une meilleure préservation des logiciels, à une meilleure accessibilité et à de meilleures possibilités d'éducation.

Mots clés: Émulation, Nintendo 64, JavaScript, Recompilation Dynamique, WebGL

Introduction

Video game emulation plays a vital role in preserving and extending the life of legacy hardware and software systems. As technology ages and older consoles become obsolete, emulation democratizes access to historical computing architectures, as a means to both experience classic retro games, and study old consoles' architectures. Among these systems, the Nintendo 64 (N64), released in 1990, remains a prime candidate for such efforts due to its pioneering 3D graphics, proprietary memory management, and enduring cultural impact – I consider it to be the sweet spot between technical constraints and creativity. The N64 arguably represents a sweet spot between technical limitation and creative potential. Its hardware was powerful enough to support immersive, dynamic game worlds—many of which remain visually impressive even today—yet constrained enough to necessitate inventive problem-solving. Developers could not rely solely on visual fidelity; instead, they were challenged to prioritize gameplay innovation and mechanical depth. In particular, the use of cartridges with limited storage space pushed studios to optimize design choices and focus on meaningful content, fostering a development environment where substance often took precedence over style.

Despite its commercial success and cultural impact, the N64's **complex hardware architecture** has made accurate emulation a persistent challenge. Existing emulators typically prioritize performance and playability over fidelity, often relying on high-level approximations of the hardware. This trade-off can hinder efforts related to preservation, debugging, and educational analysis.

This project seeks to address this challenge by developing an emulator that, while not being fully cycle-accurate, is browser-based in order to make N64 emulation portable and accessible. This project aims to strike a balance between performance, accuracy, accessibility, and

educational value. Leverage the web platform to enable real-time execution, integrate debugging tools, and support learning about emulator design and Nintendo 64 internals.



Problem Statement

While numerous Nintendo 64 emulators exist today, most are designed with a priority on speed and performance over accuracy. Few strike a true balance between all three due to the N64's complex architecture: the MIPS VR4300 CPU, the programmable RSP for vector processing, and the RDP for real-time graphics effects — all of which operate in parallel with tightly synchronized timing. Accurate modeling of this system is crucial for true preservation, debugging, and running edge-case software that depends on hardware quirks or exact timing. However, most existing emulators do not attempt cycle accuracy. Instead, they prioritize performance and playability using high-level emulation (HLE). While this makes them suitable for casual gaming, it introduces several graphical bugs, audio issues, and incorrect behavior in certain games.

js64 takes a different approach. It is not cycle-accurate for the moment – only the audio

implementation is. This project aims to explore what is possible with a **high-level, browser-based emulator** written in **JavaScript**, using WebGL for rendering. The goal is not to replicate every hardware detail, but to:

- make N64 emulation accessible from any modern browser with no installation required
- Allow debugging of the CPU, RSP, and RDP at a high level
- Provide an open-source code base and platform for education, research, and experimentation
- Demonstrate what modern JavaScript runtime can achieve

This project does not serve as a substitute for cycle-accurate emulators; rather, it complements them by prioritizing user-friendliness, portability, and clarity. Js64 can, with time, evolve to become cycle-accurate. For now, it acknowledges the constraints of high-level emulation to create a tool that is simple to navigate, available for exploration, and useful for understanding the broader architecture of the Nintendo 64.

Project Specifications

Objective of the Project

The primary objective of N64 is to develop a browser-based emulator capable of running homebrew and commercial games on the web using JavaScript. The project aims to emulate core components of the N64, including the CPU, GPU (RDP), sound, and controller input, all while delivering a seamless and responsive user experience. The emulator should allow users to play all N64 games directly in their browsers by loading the game ROM, without any need for installation or additional plugins.

Functional Requirements

This section lists the core features and capabilities planned for the N64. The emulator must fulfill these requirements in order to meet the objectives of the project. These are the essential

functionalities necessary for the emulator to be effective and usable.

1. Game Loading and Execution

   The emulator must allow loading and running N64 ROM files in the browser. It must correctly execute the ROM's machine code.

2. CPU Emulation

   The emulator must emulate the MIPS R4300i processor, and handle its unique features such as branch and jump instructions delay slot to ensure accuracy, and process both integer and floating-point operations.

3. Graphics Emulation (RDP)

   The emulator must use WebGL to replicate the functionality of the N64's RDP. It must support common rendering techniques used in N64 titles, such as Z-buffering, texture mapping, and anti-aliasing.

4. Sound Emulation

   The emulator must emulate the N64's audio processing unit. Audio should be synchronized with the gameplay.

5. Input Handling

   The emulator must support user input from the keyboard. The input system must map to the N64 controller layout, and must allow the customization of controls.

6. User Interface

   The emulator must include a simple and intuitive user interface.

7. Error Handling and Debugging

   The emulator must provide debugging tools for developers to track and resolve issues.

Non-Functional Requirements

This section lists the core features and capabilities planned for the N64. The emulator must fulfill these requirements in order to meet the objectives of the project. These are the essential functionalities necessary for the emulator to be effective and usable.

1. Performance

   The emulator must run N64 games with minimal input lag and at a frame rate of at least 30 FPS on most modern web browsers and devices.

2. Compatibility

   The emulator must be compatible with major web browsers, including Google Chrome, Mozilla Firefox, and Microsoft Edge.

   The emulator should be able to load and execute a wide variety of N64 ROMs, with an emphasis on ensuring compatibility with the most popular titles of the N64 library.

3. Usability

   The emulator interface must be user-friendly and intuitive. Users should be able to start playing games with little to no configuration.

   The emulator should offer in-game controls, graphical configurations, and debugging features that are readily available through the interface.

4. Reliability

   The emulator must be stable with minimized crashes or error during gameplay.

5. Maintainability

   The emulator’s codebase should be clean, modular, and well-documented to facilitate future development and maintenance.

Steeple Analysis

This STEEPLE analysis evaluates the social, technological, economic, environmental, political, legal, and ethical implications of developing js64, a modernized, browser-based N64 emulator. The aim is to assess its broader impact, ensuring alignment with sustainable, responsible, and culturally sensitive practices.

Social: js64 directly contributes to video game preservation by ensuring continued access to Nintendo 64 software, thus games are not lost to time nor are they locked behind obsolete, aging, hard to acquire hardware. js64 allows classic games to run directly in web browsers thus promoting greater accessibility and engagement, enabling all generations of players and enthusiasts to experience historically significant games without requiring physical consoles.

Additionally, the project's open-source nature fosters collaboration, knowledge sharing, and education within the retro gaming, emulation development, and web development communities.

Technological: Developing an emulator demands an in-depth understanding of hardware-software interactions, including CPU pipelines, precise cycle timing, memory management, instruction decoding, and optimization techniques.

Economic: Emulation provides a cost-effective alternative to acquiring legacy hardware, which has become scarce and expensive due to age and collector demand. By preserving classic games digitally, emulators reduce reliance on second-hand hardware markets, lowering the barrier to entry for users interested in retro gaming.
In the long term, open-source emulators help preserve digital heritage without the economic burden of hardware maintenance and procurement.

As a non-commercial, open-source project, JS64 also promotes long-term sustainability by removing licensing or subscription models often found in commercial emulation platforms.

Environmental: Emulation plays a role in reducing electronic waste (e-waste) by minimizing the need for physical console production and repair. As fewer people rely on outdated or broken hardware, the amount of discarded electronic components decreases.
Additionally, focusing on energy-efficient optimization in emulator design can reduce the

environmental footprint of running emulated software, ensuring lower power consumption on modern devices.

Political: js64 operates within a politically nuanced space where digital preservation intersects with strict intellectual property enforcement. Major gaming companies, notably Nintendo, have historically opposed emulation efforts, advocating for legislation like the Digital Millennium Copyright Act (DMCA), which restricts digital rights management (DRM) circumvention—even for non-commercial purposes.

However, there is increasing recognition—both among legal scholars and within the preservation community—that emulation can serve a legitimate cultural and educational role, particularly when original hardware is no longer produced or supported. While Nintendo no longer profits from selling Nintendo 64 consoles or many of its associated titles, js64 positions itself carefully as a **non-commercial, research-oriented emulator**, with the goal of fostering software preservation, educational exploration, and archival access. Academic partnerships and clear communication about the project's preservation goals can further bolster its political and institutional legitimacy.

Legal: Although js64 does not include or distribute any proprietary code or copyrighted ROMs, it operates in a domain regulated by complex international IP frameworks. Legal precedents and local laws vary, but most jurisdictions prohibit the unauthorized distribution of copyrighted software, even if it is no longer commercially available.

To mitigate risk and maintain legal defensibility, JS64 follows **clean-room reverse engineering** principles, wherein emulator behavior is reimplemented without referencing any proprietary source code. The emulator is designed to support only legally obtained ROMs, homebrew software, and public domain titles.

While some argue that emulating systems no longer sold or supported by the original

manufacturer poses minimal harm—and may in fact promote cultural preservation—this perspective remains largely ethical rather than strictly legal.
Accordingly, JS64 emphasizes transparency, legal usage guidelines, and alignment with existing DMCA exemptions for archival and educational purposes, reinforcing its commitment to responsible development.

Ethical: js64 emphasizes transparency, community accountability, and ethical use. Comprehensive documentation outlines the project's objectives, technical limitations, and any known inaccuracies in emulation. Users are explicitly encouraged to respect intellectual property rights by using only legally obtained ROMs and adhering to local laws.

While legal restrictions remain in place, there is a growing ethical consensus—particularly within academia and digital preservation circles—that emulating outdated, unsupported platforms can be justified when done non-commercially and with the intent to preserve cultural heritage. Since Nintendo no longer sells the Nintendo 64 hardware or most of its original titles, js64 aligns with the ethical principle of **minimizing cultural loss**, offering a responsible alternative to piracy while supporting educational, archival, and historical engagement.

By remaining open-source, js64 invites scrutiny, fosters trust, and empowers contributors to engage in ethical software development, ensuring that the project continues to evolve in a community-led, accountable, and preservation-focused manner.

Engineering Standards

This project follows widely recognized engineering standards to ensure code quality, portability, accuracy, and legal compliance throughout the emulator’s development.

- **IEEE 12207: Software Life Cycle Processes**

  Use structured software development processes, ensuring clear stages for design,

implementation, testing, and maintenance of the emulator.

- **ISO/IEC 25010: Software Product Quality**

  Ensure that the emulator meets key quality attributes, including functional correctness, performance, security, and maintainability.

- **ECMA-262: ECMAScript (JavaScript)**

  Conform to the ECMAScript specification to ensure that JavaScript code is maintainable, performant, and compatible across modern browsers and platforms.

- **IEEE 754: Floating-Point Arithmetic**

  Implement the floating-point operations in strict adherence to IEEE 754, ensuring accurate emulation of the Nintendo 64's FPU.

- **Web Content Accessibility Guidelines (WCAG) 2.1**

  Ensure that the emulator's UI is accessible, meeting WCAG 2.1 standards to accommodate users with disabilities.

- **IEEE 1012: Software Verification and Validation**

  Implement comprehensive testing strategies (unit, integration, and regression) to ensure functionality and prevent regressions during development.

- **Intellectual Property Compliance**

  Comply with intellectual property laws, ensuring no proprietary assets (like Nintendo ROMs, BIOS, microcode) are included in the emulator.

Logic Model Framework

Target Populations

- **Video Game Preservationists**

  Individuals interested in preserving retro games through software-based emulation, ensuring historical accuracy and a faithful gaming experience.

- **Web Developers and JavaScript Enthusiasts**

Developers interested in understanding the intricacies of emulation, especially through JavaScript, and those working with web technologies and curious about their potential.

- **Retro Gaming Community**

  Gamers looking for an authentic Nintendo 64 experience on modern devices without needing physical N64 hardware, relying instead on browser-based access.

Underlying Assumptions

- **Web-Based Emulation Can Achieve High Fidelity**

  Emulating the N64's behaviour through JavaScript and WebGL can achieve a performant, playable experience in web browsers.

- **Modern Browsers Support Efficient Emulation**

  Browsers have sufficient performance and support for technologies like JavaScript and WebGL to run complex emulation tasks efficiently.

Resources/Challenges

Resources

- Nintendo 64 Technical Documentation

  Official technical documents, particularly the VR4300 CPU and co-processor specs, to guide accurate hardware emulation.

- Open-Source Emulator Projects

  Reference projects such as *mupen64plus* and *CEN64* for architectural insights and practical solutions.

- Browser Developer Tools

  Browser-based debugging tools to profile performance, track memory usage, and visualize timing or synchronization issues.

Challenges

- Emulating Undocumented Hardware Features

Many N64 games rely on undocumented features or unique behaviour, requiring reverse-engineering or community knowledge to emulate.

- Cycle-Accurate Emulation within Web Constraints

  Achieving cycle-level emulation of the VR4300 CPU, RSP, and RDP within the performance limitations of JavaScript.

- Maintaining Real-Time Synchronization

  Ensuring precise timing and synchronization between the CPU, RSP, RDP, and audio subsystems while running on modern web browsers.

- Balancing Performance with Accuracy in Browsers

  Optimizing for performance all while ensuring the emulator runs efficiently without compromising on accuracy.

Activities

- **Implement Dynamic Recompiler for VR4300 CPU**

  Develop a dynamic recompiler for the VR4300 CPU to optimize instruction execution and improve performance by translating CPU instructions into efficient, optimized JavaScript functions.

- **Develop Efficient Memory Management**

  Develop a memory management system that simulates RDRAM access, TLB, cache, and other features while optimizing for web performance.

- **Synchronize Peripheral Devices**

  Ensure proper synchronization between controllers, audio DACs, and the CPU, ensuring that input/output behaviour matches the original hardware.

- **Develop Debugging and Profiling Tools**

  Build tools to help with debugging, monitoring CPU cycles, memory transactions, and peripheral events for testing and validation.

Outputs Of the Project

- **Fully Functional JavaScript-based Emulator**

  A working web-based emulator that can run N64 test and commercial ROMs and homebrew games.

- **Web-Based Debugging and Trace Interface**

  A real-time tool for monitoring and analyzing the emulator's performance, including CPU, memory, and RCP behavior.

- **Open-Source Emulator Codebase**

  A well-documented and modular codebase in JavaScript and WebAssembly that can be contributed to by the open-source community.

Outcomes

Short-Term

- **Basic JavaScript-Based dynamic and Memory Model**

  Initial CPU and memory models implemented, allowing simple test ROMs to be run and validated.

Intermediate

- **Accurate Graphics and Audio Emulation**

  Full emulation of the graphics pipeline (RDP) and audio processing (RSP), ensuring proper game rendering and sound reproduction.

- **Dynamic Recompiler Implementation for CPU**

  Successful integration of the dynamic recompiler for optimizing the VR4300 CPU's instruction execution, providing performance improvements.

- **Initial User Interface and Debugging Tools**

  Functional UI allowing users to load ROMs and monitor emulation performance, with real-time debugging and trace capabilities.

Long-Term

- **High Compatibility with N64 Games**

  Support for a wide range of commercial N64 games.

- **Reliable Web-Based Preservation Tool**

  A dependable, browser-accessible emulator for video game preservationists and researchers studying N64 hardware behavior and performance via dynamic recompilation.

- **Sustainable Open-Source Community**

  An active development community contributing to ongoing improvements, peripheral emulation, and long-term preservation goals.

Literature Review

The preservation of video games has become an increasingly urgent concern in digital heritage and archival circles. Unlike traditional forms of media, video games depend on complex interactions between hardware and software, making them uniquely vulnerable to technological obsolescence. Johansson (2023), in her comprehensive study on video game preservation, emphasizes that safeguarding video games is not simply a matter of storing data but of preserving an interactive experience that requires the faithful replication of the original hardware environment. Without active measures to reproduce that environment, games may become inaccessible, effectively erasing an important aspect of digital culture.

One of the most effective techniques for preserving video games is through software emulation. Emulators simulate the functionality of obsolete hardware, enabling legacy software to run on modern systems. However, emulation comes with technical and philosophical challenges, especially in how it balances accuracy with performance. Guttenbrunner, Becker, and Rauber (2010) argue that while emulation can recreate the behavior of old hardware, its implementation often requires trade-offs between fidelity and efficiency. In their evaluation of strategies for console game preservation, they identify emulation as the

most promising solution but stress that it must be carefully designed to ensure both accuracy and sustainability over time.

Emulation strategies generally fall along a spectrum between performance-oriented emulation and cycle-accurate emulation. Performance-oriented emulators focus on delivering a smooth, responsive experience on modern hardware, often using techniques such as dynamic recompilation and approximate timing models. These emulators, while offering speed and accessibility, tend to simplify or bypass hardware behavior that is not critical for typical gameplay (Emulation General Wiki, n.d.). Conversely, cycle-accurate emulators strive to replicate the behavior of original hardware at the level of individual clock cycles. This means accurately simulating the timing and interaction of every component in the system, which is essential for preserving hardware-dependent behavior and ensuring perfect compatibility (mGBA, 2017).

The complexity of cycle-accurate emulation is extensively explored by Farfeleder, Krall, and Horspool (2007), who focus on compiled simulation of inorder pipelined architectures. Their work explains that cycle-accuracy requires a complete reproduction of instruction pipelines, including stalls, hazards, and forwarding paths, which must be accurately modeled to preserve data dependencies and timing. Farfeleder et al. demonstrate that while compiled cycle-accurate simulators can significantly improve execution speed over interpreted models, they still impose high computational demands. They also highlight the need for precise modeling of memory interactions and peripheral device communication, which, if mishandled, can introduce errors in the simulation that break compatibility with software designed for the original hardware.

This tension between accuracy and performance is further illustrated in practical emulation projects. mGBA (2017) examines the trade-offs faced by emulator developers and highlights the challenges in maintaining cycle accuracy while delivering acceptable performance. They describe how highly accurate emulators, such as bsnes for the SNES, replicate hardware quirks like bus contention and instruction timing. This allows them to run games that depend on specific hardware behavior but requires modern systems with considerable processing power, even for games that originally ran on much slower hardware.

These issues become significantly more challenging when emulating complex systems such as the

Nintendo 64 (N64). McFerran (2025) summarizes Modern Vintage Gamer's investigation into why the N64 remains difficult to emulate accurately, highlighting factors such as its unique architecture and hardware quirks. The console integrates a MIPS R4300i CPU with the Reality Co-Processor (RCP), which itself includes the Reality Signal Processor (RSP) and Reality Display Processor (RDP). These components operate in parallel and rely on custom microcode to manage tasks like graphics rendering and audio processing. Because the microcode varies between games, emulators must either interpret these routines or attempt to reproduce them directly. Minor discrepancies in timing can lead to significant graphical or audio artifacts, making cycle-accurate replication particularly important for the N64.

Rodrigo Copetti (2025) provides a thorough technical analysis of the N64, emphasizing its unified memory architecture. In this design, both the CPU and co-processors share access to RDRAM, requiring careful arbitration and synchronization. Copetti explains how, if the timing of memory accesses between these processors is not faithfully reproduced, emulation can result in race conditions that never existed on the original hardware. This leads to bugs that disrupt gameplay, highlighting the necessity of precise timing in cycle-accurate emulation.

Matthew Carlson (2022) expands on the N64's architectural complexity by discussing Nintendo's design choices, including the use of custom microcode and the challenges they created for developers. Carlson notes that Nintendo's decision to offer flexible but low-level control over graphics and audio pipelines allowed developers to extract impressive performance from the hardware but made emulator development far more difficult. Each game potentially used different approaches to leverage the hardware, requiring emulator developers to accommodate a variety of unique implementations.

The official Nintendo documentation (1997) on the N64 architecture corroborates these challenges by detailing the interactions between the CPU, RSP, and RDP. It describes how these processors interact through DMA transfers and command queues, emphasizing the need for tightly controlled timing to maintain synchronization. For example, delays in processing display lists or audio buffers can result in missing frames or audio glitches, which would be unacceptable in an emulator striving for accuracy.

While the technical barriers to cycle-accurate emulation are formidable, the legal and ethical issues surrounding emulation are equally complex. Conley, Andros, Chinai, Lipkowitz, and Perez (2004) discuss the video game industry's historical opposition to emulation, citing lawsuits against emulator developers like Connectix and Bleem! as examples of the industry's efforts to protect its intellectual property. Conley et al. argue that while concerns about piracy and revenue loss are valid, they must be weighed against the value of emulation in preserving games that are no longer commercially available. They highlight how emulation provides access to legacy software, filling a preservation void that game publishers have largely ignored.

Johansson (2023) addresses this issue from the perspective of preservation communities. She notes that much of the preservation work is undertaken by hobbyists and independent archivists who operate without the support—or sometimes against the wishes—of rights holders. Johansson emphasizes that video games are part of our cultural heritage and should be preserved as such, even if doing so requires challenging current legal frameworks.

Guttenbrunner et al. (2010) further argue that while emulation offers the best chance for preserving both the software and user experience of video games, it introduces sustainability challenges. Emulators must be continuously updated to remain compatible with modern hardware and operating systems. If these projects are abandoned, the preserved games may once again become inaccessible.

Van Gorsel, Leenaars, Milic-Frayling, and Palm (2014) make the case for institutional support in digital preservation, proposing that organizations like UNESCO could play a key role in legitimizing emulation as a preservation strategy. They advocate for the development of international policies that protect emulator developers from legal risks and promote the long-term sustainability of emulation projects.

In conclusion, the literature on video game preservation, emulation accuracy, and the Nintendo 64's hardware architecture paints a complex picture. Cycle-accurate emulation offers the highest fidelity in replicating original hardware behavior but imposes significant computational demands and development complexity. Performance-optimized emulators, while more accessible, often sacrifice the very accuracy

that makes emulation a reliable preservation tool. The Nintendo 64 exemplifies these challenges, as its intricate architecture and microcode-based operations make faithful emulation especially difficult. Beyond technical considerations, legal and ethical debates continue to shape the landscape of video game preservation. A sustainable, legally sound, and technically rigorous approach to emulation will be essential for preserving video games as a vital part of our digital cultural heritage.

Methodology & Capstone Design

Nintendo 64 Architecture Overview

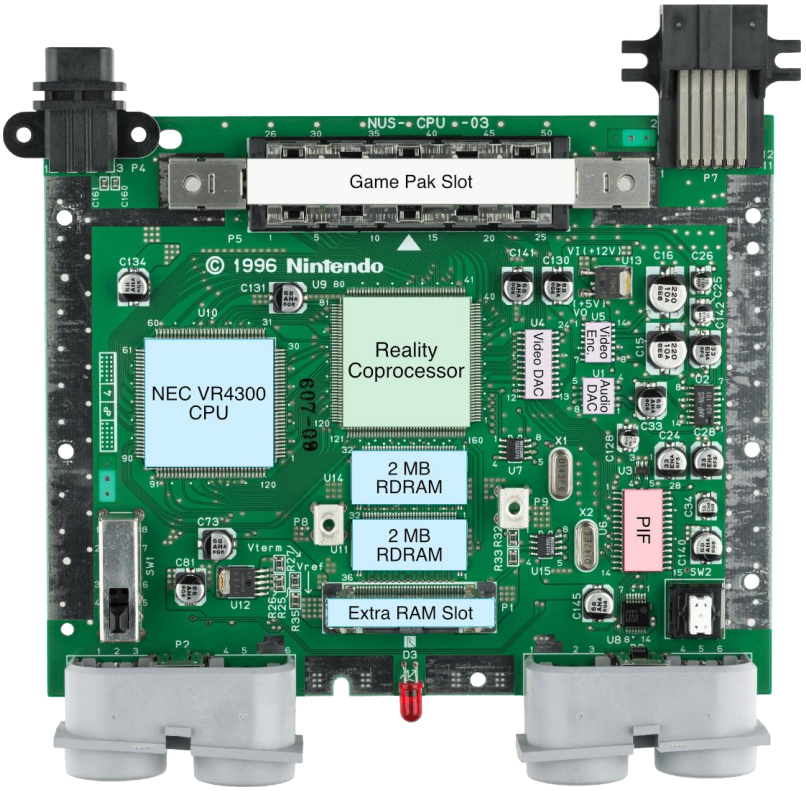


Figure 1 Nintendo 64 Motherboard with important parts labelled

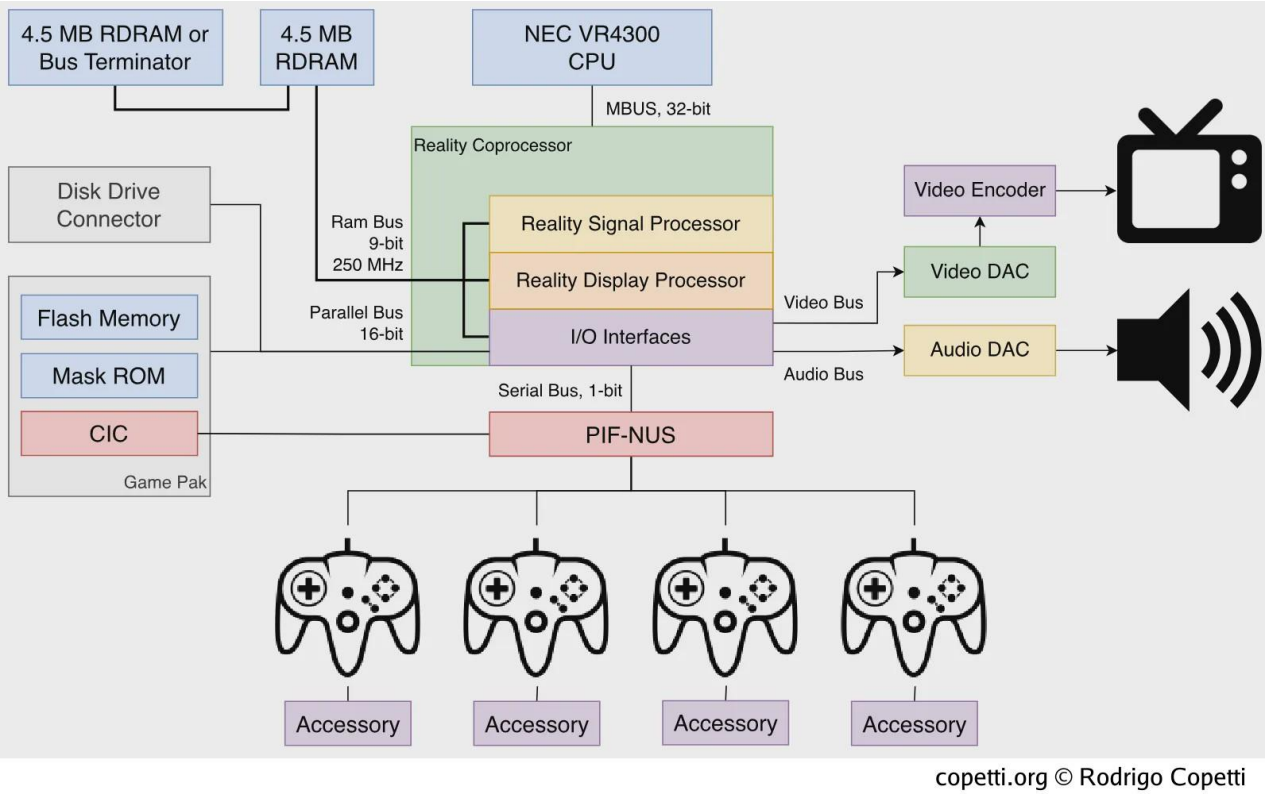


Figure 2 Main Nintendo 64 architecture diagram

The origins of the Nintendo 64's main processor begin with the MIPS R4000, MIPS' new avant-garde CPU released in 1991. The R4000's most apparent novelty was its support for 64-bit computing, resulting from widening the size of buses, registers, and computation units to handle 64-bit values with efficiency. Developers accessed these enhancements through the newly introduced MIP III instruction set.

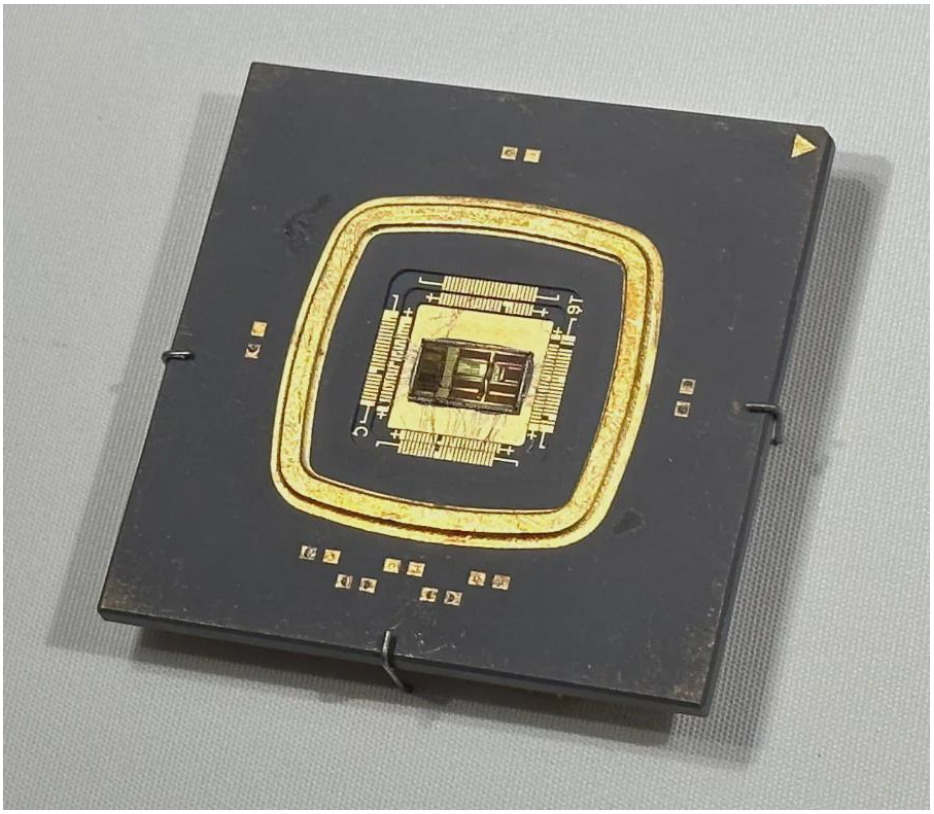


Figure 3 The MIPS R4300i CPU (1995).

For their next generation console, Nintendo looked intro bringing industrial hardware to home consoles. Thus, Nintendo partnered directly with Silicon Graphics, Inc. (SGI), a company renowned for its high-end graphics workstations and a major player in MIPS architecture development.

The R4000 was an expensive product, which made it unfeasible for a video game console. To address this, Nintendo and SGI selected a cost-effective alternative: the MIPS R4300i, a simplified, lower-cost variant. NEC was brought on board to second-source and manufacture the chip, which was eventually branded as the NEC VR4300. This processor became the heart of the Nintendo 64, operating at a clock speed of 93.75 MHz. This is a binary-compatible version of the MIPS R4300i that features

Two modes of operation:

32-bit mode: Traditional mode where the CPU behaves as a MIPS II-compatible processor. There's nothing special about this mode except that all new functions are locked out.

64-bit mode: 'Native' mode where all 64-bit extensions are available. It's also binary-compatible with 32-bit applications.

32 general-purpose registers: These are 32-bit wide in '32-bit mode' and 64-bit wide in '64-bit mode'.

The MIPS III ISA: A RISC instruction set that succeeds MIPS II. It features new opcodes that compute 64-bit words called 'doublewords'. Finally, instructions are always 32-bit long, independently of the mode.
It's worth mentioning that since MIPS II, load delay slots are gone for good, though branch delay ones still persist.

An internal 64-bit bus connected to an external 32-bit data bus: While doublewords won't degrade performance when operated internally, the CPU will still need to expend extra cycles to move 64-bit data throughout the system.

This is one of the cutbacks of the R4300i variant (the R4000 has a full 64-bit data bus).

32-bit address bus: Up to 4 GB of physical memory can be addressed.

5-stage pipeline: Up to five instructions can be allocated for execution (a detailed explanation can be found in a previous article).

24 KB L1 cache: Divided into 16 KB for instructions and 8 KB for data.

An internal Floating-point Unit (FPU) is integrated directly into the processor and identified as coprocessor 1 COP1, however it is fitted next to the main ALU and is only

accessed through the CPU's internal ALU pipeline, meaning there's no co-processing per se. It maintains a dedicated floating-point register file and supports both 32-bit and 64-bit floating-point operations, conforming to the IEEE 754 standard.

Simplified memory access

The way RAM is assembled follows the unified-memory architecture or 'UMA' where all available RAM is centralised in one place only and any component that requires RAM will access this shared location. The component arbitrating its access is, in this case, the GPU.

The reason for choosing this design comes from the fact that it saves a considerable amount of production costs while, on the other side, it increments access contention if not managed properly.

No DMA controller

Due to the unified memory architecture, the CPU no longer has direct access to RAM, so the GPU will be providing DMA functionality as well.

Memory design

The system physically contains 4.5 MB of RAM, however, it's connected using a 9-bit data bus where the 9th bit is reserved for the GPU, a design detail elaborated further in the Graphics section. As a consequence, every component except the GPU will only directly access up to 4 MB.

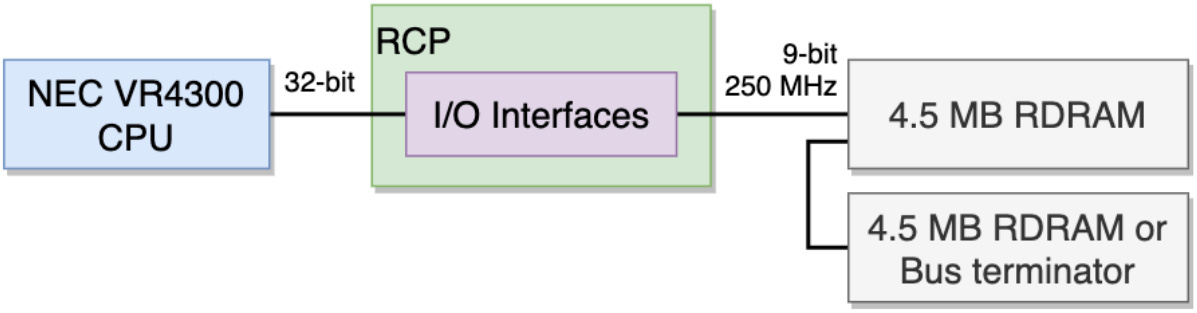


Figure 4 Memory Layout of the N64 System

The type of RAM used by the system is called Rambus DRAM (RDRAM), a competing memory technology to SDRAM at the time. RDRAM is connected in serial (where transfers are done one bit at a time) while SDRAM uses a parallel connection (transfers multiple bits at a time).

Latency and Speed

One of RDRAM's characteristics is that its latency increases with the number of memory banks installed. Given the memory configuration of the Nintendo 64, this results in relatively high latency, reported around 640 nanoseconds.

To counteract this, RDRAM is clocked at 250 MHz, approximately 2.6 times faster than the CPU clock speed, allowing it to sustain high data throughput. Nintendo claimed a theoretical data transfer rate of 500 MB/s for sequential reads or writes.

Furthermore, Nintendo chose NEC's uPD488170L memory banks for their motherboard. These chips implement a technology called 'Rambus Signaling Logic', which doubles the transfer rate. This may explain why some sources state the 'effective' rate of memory is 500 MHz.

Leaving room for improvement

Interestingly, the amount of available RAM on this console can be expanded by installing the Expansion Pak accessory: A small box that adds another 4.5 MB of RAM.

Although optional for some games, certain titles like Donkey Kong 64 or The Legend of Zelda: Majora's Mask were designed with the expansion as a requirement to run, and would display an error screen without it.



Figure 5 Expansion Pak [9], an optional accessory sold separately (sometimes included with the game that requires it)



Figure 6 Jumper Pak. In the absence of the Expansion Pak, this must be present to terminate the RDRAM bus.

Curiously enough, the RAM bus must be terminated to function, so the console always shipped with a terminator (called Jumper Pak) fitted in the place of the Expansion

Pak. Booting the system without either Pak installed results in a blank screen, as the memory subsystem cannot initialize correctly without a proper terminator

Memory management

The VR4300 includes another coprocessor called System Control Coprocessor (CP0). This unit is responsible for handling system-level functions. CP0 includes both a Memory Management Unit (MMU) and a Translation Lookaside Buffer (TLB), which together define how memory is organized, accessed, and cached.

The VR4300 can access up to 4 GB worth of 32-bit addressable memory addresses, but as we've seen, we don't have 4 GB of RAM in this console. So, the MMU takes over memory addressing and provides a useful memory map where the physical memory is mirrored multiple times. Consequently, memory locations are treated as virtual addresses. Furthermore, the TLB enables developers to define their own memory map in some mirrors without (significant) performance penalties.

At first, this may seem redundant, but each segment is connected to different circuitry (i.e. L1 cache, uncached, TLB address) so developers can optimize performance by selecting memory regions based on their use case.

Some segments are meant to discriminate 'kernel' locations from 'user' ones for security purposes. The N64 always operates in 'kernel' mode, thus, the 'non-TLB kernel cached' segment (called 'KSEG0') is the most common one for games.

Lastly, while the MMU is capable of operating in 64-bit mode, allowing for 40-bit virtual addresses (which equates to a 1 TB address space), this functionality goes largely unused in the Nintendo 64, as the system's hardware constraints make such large address spaces impractical.

Graphics

The visual output of the N64 is managed by a highly sophisticated chip know as the Reality Co-processor, designed by Silicon Graphics and running at 62.5 MHz This chip constitutes the core of the console's graphic system. Rather than simply rendering pixels to the screen, the N64's graphics architecture emphasizes programmability and geometry processing acceleration, thus offloading a significant portion of computations from the CPU.

The Reality Co-Processor is composed of two key components involved in graphics processing:

Reality Signal Processor (RSP)

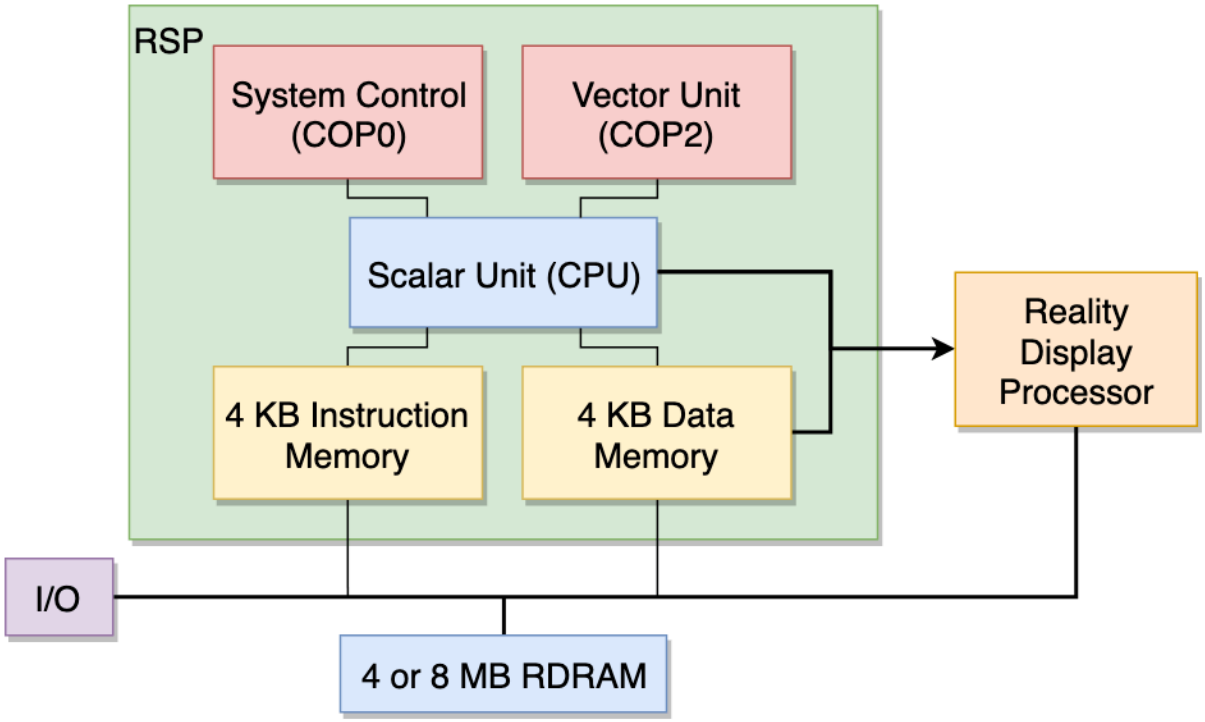


Figure 7 Architecture of the Reality Signal Processor (RSP).

The RSP functions as an autonomous processing unit, structurally resembling a reduced instruction set CPU. It comprises three main subcomponents:

- **Scalar Unit**: This is a simplified derivative of the MIPS R4000 CPU, supporting only a restricted subset of the MIPS III instruction set. It omits support for key architectural

features such as interrupts, exceptions, hardware multiplication/division, and 64-bit operations. Despite these omissions, the scalar unit serves as the primary control unit for the RSP.

- **Vector Unit**: This coprocessor is dedicated to vector arithmetic and features **32 registers**, each **128 bits wide**. These registers are internally partitioned into **eight 16-bit lanes**, allowing for the parallel execution of SIMD (Single Instruction, Multiple Data) operations. This design significantly accelerates computationally intensive tasks such as vertex transformation and lighting calculations.
- **System Control Unit**: This component manages direct memory access (DMA) operations and provides control signals to the adjacent Reality Display Processor (RDP). It enables the RSP to operate with a high degree of autonomy, minimizing CPU intervention during rendering tasks.

The RSP executes graphics operations by processing display lists, which are sequences of rendering commands generated by the main CPU and stored in RAM. These commands typically include vertex transformations, perspective projections, clipping, and per-vertex lighting.

A notable aspect of the RSP is its use of microcode—compact, specialized programs that define how graphics commands are interpreted and processed. Each microcode is typically limited to a few hundred instructions and is stored in the RSP's internal instruction memory. These microprograms are loaded into the RSP at runtime by the CPU, thereby allowing developers to customize the graphics pipeline to suit different rendering strategies.

Nintendo supplied several official microcodes optimized for varying use cases, balancing factors such as polygon throughput, lighting complexity, and texture detail. This

approach is conceptually similar to the background modes employed in the Super Nintendo Entertainment System (SNES), wherein specific configurations are used to optimize the hardware for particular visual effects.

Reality Display Processor (RDP)

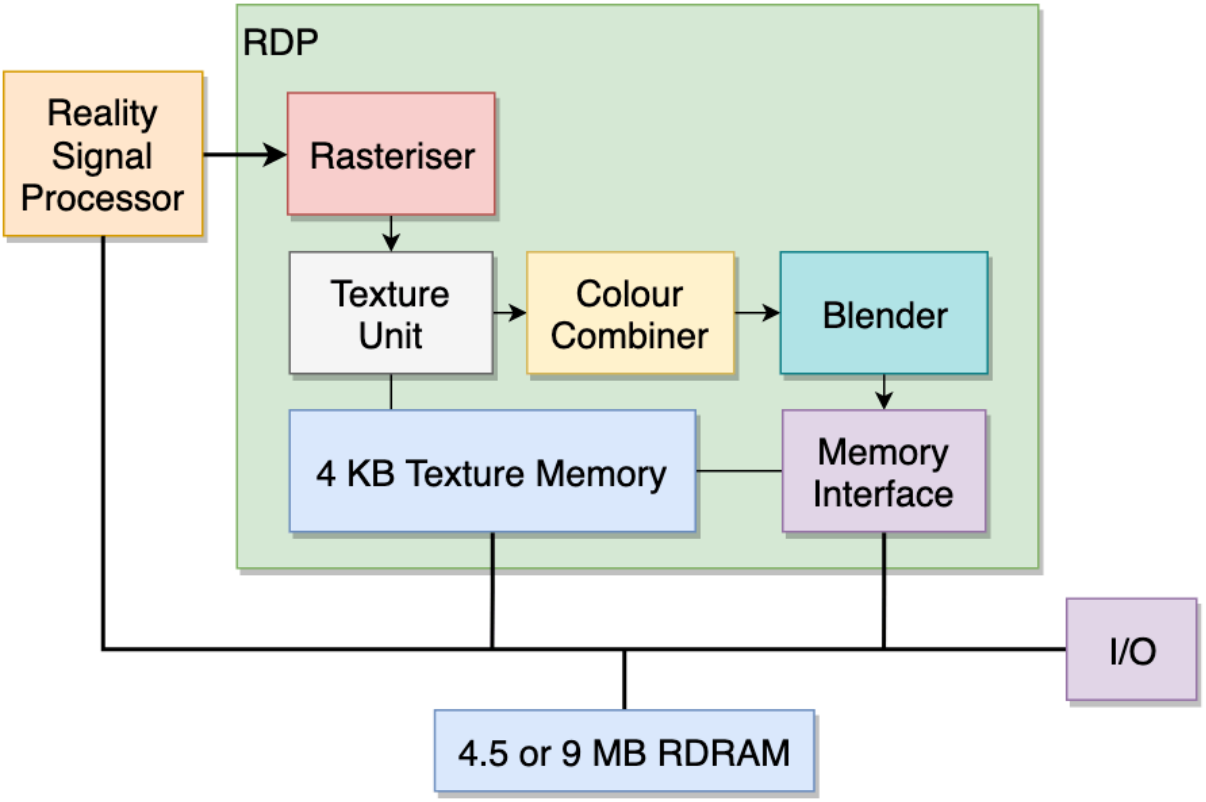


Figure 8 Architecture of the Reality Display Processor (RDP).

The RDP is a fixed-function processor responsible for rasterizing primitives, applying textures, blending colours, and composing the final frame. It handles both triangles and rectangles (the latter mainly for sprites) through a dedicated rasterization pipeline comprising:

- **Rasterizer**: Converts vertex-based primitives into pixels.
- **Texture Unit**: Maps textures stored in a 4 KB TMEM, supporting:
    - *Bilinear Filtering*: Smooths textures using triangular interpolation (three points instead of four), requiring pre-adaptation for certain textures.

- *Mip-Mapping*: Selects lower-resolution textures based on distance to reduce aliasing, with optional trilinear filtering for smoother transitions.
- *Perspective Correction*: Adjusts texture mapping based on depth values for greater visual accuracy.

- **Colour Combiner**: Blends multiple colour layers, enabling effects like shading.
- **Blender**: Composites pixels with the frame buffer to handle translucency, anti-aliasing, fog, and dithering; it also performs z-buffering.
- **Memory Interface**: Manages reading/writing to RAM and TMEM, crucial for frame buffer operations.

The RDP operates in four different modes, optimizing these blocks for various tasks. Due to constant frame buffer updates, it uniquely handles RAM, utilizing a non-standard 9-bit "byte" where the ninth bit supports z-buffering and anti-aliasing, accessible only via the Memory Interface.

Once the RDP finalizes the frame, it must be transferred to the Video Interface (VI) and subsequently to the Video Encoder for display. This transfer typically uses Direct Memory Access (DMA) to avoid CPU overhead.

The Nintendo 64 theoretically supports up to 24-bit colour (16.8 million colours) and a maximum resolution of 640×480 (or 720×576 in PAL regions). However, using these maximum specifications is often impractical due to resource constraints, leading developers to prioritize lower resolutions and colour depths to maintain performance.

Rendering a Frame: A Practical Example

To illustrate the rendering process, we can refer to Super Mario 64:

Vertex Processing



Figure 9 Primitive view of a Super Mario 64 Scene.

Initially, assets such as models and textures are stored in ROM. To maintain bandwidth efficiency, they are first loaded into RAM, with the CPU decompressing any compressed data as needed.

Rather than performing all rendering tasks itself, the CPU constructs a Display List describing the scene and delegates its execution to the Reality Co-Processor (RCP), specifically the RSP. The process includes:

Writing the Display List to RAM.

Instructing the RSP to locate and process the list.

Loading microcode to the RSP's Scalar Unit.

The RSP processes vertex transformations and lighting, sending rasterization commands to the RDP.

Pixel Processing



Figure 10 Rendered frame

At this stage, the RDP rasterizes primitives, applies textures (copied from RAM to TMEM via DMA), and processes pixel-level effects such as anti-aliasing and blending.

The RDP operates through a fixed-function pipeline, but different operational modes can be selected to optimize for specific rendering tasks. Once completed, the final frame is written to RAM's frame buffer.

The CPU then uses DMA to transfer the frame to the VI, which feeds it to the Video Encoder for display on-screen.

Audio Processing

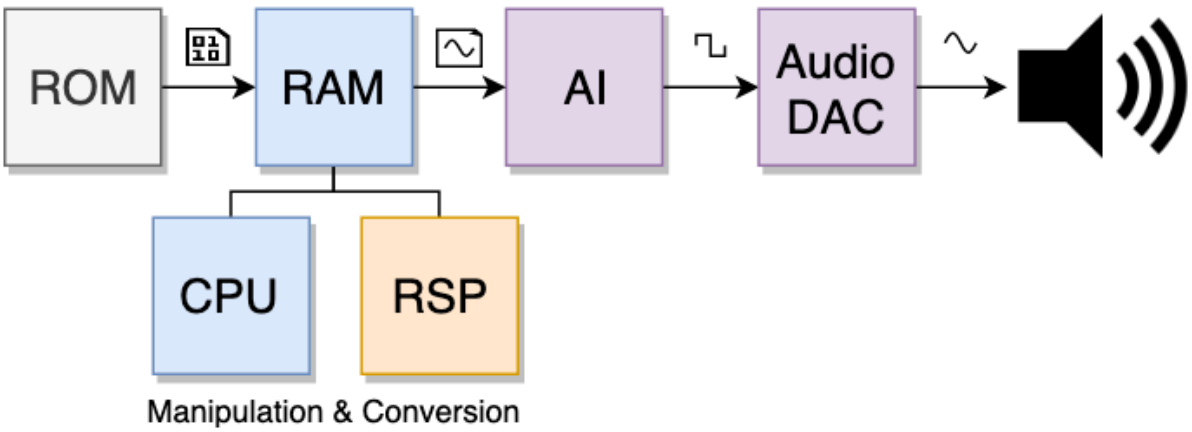


Figure 11 Overview of how the audio pipeline is often programmed.

The Nintendo 64's audio subsystem operates between two endpoints:

Source: Cartridge ROM, containing audio data readable only by the CPU.

Destination: Digital-to-Analog Converter (DAC), which requires waveform audio.

Unlike other consoles that feature a dedicated audio chip, the Nintendo 64 distributes audio processing across several components:

CPU: Transfers audio assets from ROM to RAM and prepares Audio Lists for the RSP.

RSP: Executes specialized microcode to process the audio lists stored in RAM. Its tasks include:

Decompressing ADPCM audio samples and applying audio effects.

Sequencing and mixing MIDI data using sound banks located in RAM.

The RSP produces raw waveform data, which is passed to the Audio Interface (AI). The AI then sends the waveform to the DAC for playback. The final audio output consists of two stereo channels with 16-bit resolution each.

Anti-Piracy and Region Locking



Figure 12 N64 CIC Chip

The Nintendo 64 continued the use of the SNES's security system based on the CIC (Checking Integrated Circuit) chip. Every authorized cartridge contained a CIC, and the Nintendo 64 enhanced this system by requiring each game to have a specific CIC variant. At startup and during gameplay, the PIF (Peripheral Interface) performs checksum validations to verify the presence and authenticity of the cartridge's CIC. If the PIF detects an invalid or cloned CIC, the console is forced into a permanent freeze, preventing gameplay.

Region locking was primarily implemented through physical means: cartridge shapes were slightly altered between regions, making it mechanically impossible to insert a cartridge from another market without modification.

Due to the reliance on proprietary cartridges, piracy concerns were relatively low compared to CD-based systems, although this choice contributed to significantly higher game prices.

Software Stack & Tools

The development of the js4 emulator involved a diverse set of modern web technologies and tools that enable N64 emulation entirely within a browser environment. Below are listed the key components of the software stack utilized for this project:

Core Technologies

Build and Runtime Environment:

Libraries and Frameworks

Emulation Techniques & Algorithms

How is the Boot process emulated?

To ensure accurate execution of commercial games, the boot process must be faithfully emulated. The emulator mimics the original N64's power-on behaviour by initializing the system's CPU, coprocessors, and memory in a manner consistent with real hardware expectations.

The preparation explained below allows the emulator to "trick" IPL3 into believing it is running on authentic hardware, thereby padding checksum verification and enabling the game to run normally.

The N64 has a three-stage boot sequence, known as IPL1, IPL2, and IPL3:

1. IPL1: A small 2 KB internal boot ROM embedded in the console, responsible for initializing the hardware and copying the IPL2 segment from the cartridge to RAM.
2. IPL2: The first 1 KB of the cartridge ROM, which loads and prepares the final stage.
3. IPL3, also on the cartridge, performs security validation such as checksum verification, and ultimately decrypts and launches the game's main code at address 0x80000000

In js64, IPL1 is bypassed as the emulator simulates the outcome of IPL1 by preconfiguring the CPU state.

The boot routine is implemented in the emulator through the simulateBoot(cpu0, hardware, rominfo) function. This function initializes the emulated VR4300 CPU and surrounding hardware to a state that mimics a real N64 system after IPL1 and IPL2 have completed execution.

This involves setting Control Registers such as Status, Config, Count, and Cause with values consistent with a real N64 at power-on. General-Purpose Registers r0 through r31 are set with values specific to the cartridge's CIC (Checking Integrated Circuit) chip. The program counter is set to 0xA4000040, which marks the entry point of IPL3.

CIC Chip Behavior

The CIC chip was used by Nintendo as an anti-piracy and region-locking measure. Its primary purpose is to check the authenticity of the cartridge and to supply unique seed values for security-related calculations. Each CIC version (e.g., 6102,6105) requires a specific register configuration at boot, and in some cases, auxiliary code to be injected into the RSP IMEM (instruction memory) for decryption routines.

To accommodate this, the emulator reads the CIC type from the rominfo object and applies a hardcoded set of register initializations. For example, CIC 6105 and 6106 require small programs to be written into the RSP's IMEM (Instruction Memory), which IPL3 later executes to decrypt parts of the game code. The emulator uses the line:

```
const imem = hardware.sp_mem.subRegion(0x1000, 0x1000);
```

to access the upper half of the RSP's 4 KB memory, where this decryption code is written.

Regional Differences

The emulator also accounts for regional differences (PAL vs. NTSC), as CIC seed values and timing behavior differ between television standards. This is handled by checking the tvType field in rominfo and adjusting register values accordingly to ensure accurate boot timing and video synchronization.

Decoding

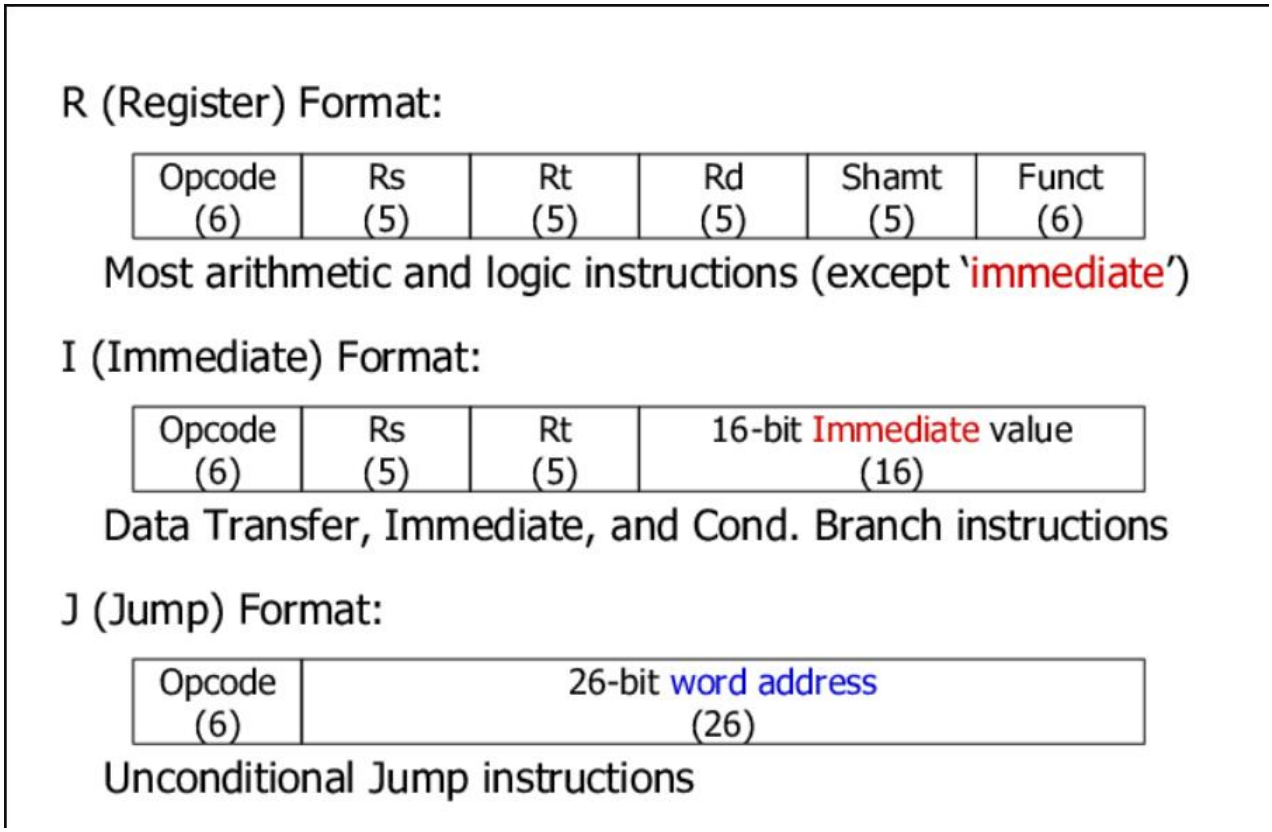


Figure 13 MIPS instruction formats

A foundational aspect of emulating the N64's CR4300 processor involves accurately decoding its 32-bit MIPS III instructions. The decoding process is essential for the CPU to interpret the raw machine code and extract relevant fields such as opcodes, registers, immediates, etc.

MIPS instructions generally fall into four broad categories, as illustrated in the table below:

Type	Use Case	Example
R-type	Register-based ops	ADD, SLL
I-type	Immediate values & branches	ADDI, LW
J-type	Jump instructions	J, JAL
COP-type	Coprocessor instructions	MFC1, CFC1

Table 1 MIPS instruction types, use cases, and examples

R-type (Register): These instructions specify all operands as registers. They typically include fields for source (rs, rt) and destination (rd) registers, as well as a function field (funct) used to identify the specific operation (e.g., ADD, SLL).

I-type (Immediate): These involve a 16-bit immediate value used in arithmetic, load/store, and conditional branch operations.

J-type (Jump): These use a 26-bit target address to support absolute jumps within the code segment.

COP-type (Coprocessor): Specially formatted instructions used to communicate with coprocessors such as the FPU (Coprocessor 1), typically using the COP1 opcode.

To decode these instructions, a collection of decoding helper functions was used, which isolate specific bitfields from the 32-bit instruction word using bitwise operations & masks. Notably:

simpleOp(i) extracts the top 6 bits (opcode), which are critical for determining the instruction type.

rs(i), rt(i), and rd(i) isolate register indices used across R- and I-type formats.

imms(i) sign-extends 16-bit immediates, essential for correctly handling signed arithmetic and branch offsets.

jumpAddress(pc, i) and branchAddress(pc, i) reconstruct full target addresses for control flow operations.

Dynamic Recompilation

A fundamental challenge in emulating a system like the Nintendo 64 lies in maintaining speed and performance while accurately reproducing the behaviour of the MIPS R4300 processor. The N64's VR4300 CPU, running at 93.75 MHz, executes millions of MIPS III instructions per second. Naïve interpretation – where each instruction is fetched, decoded, and executed in sequence by the emulator – despite its straightforwardness, quickly becomes a bottleneck, particularly in complex commercial games. Each instruction requires multiple software operations, causing games to run much slower than they did on real N64 hardware.

To address this, js64 leverages an approach known as Dynamic Recompilation (commonly abbreviated dynarec), that consists of converting machine code between architectures. By compiling during execution, the emulator

can tailor the generated code to reflect its run-time environment. In this case, blocks of MIPS machine code are translated into equivalent optimized JavaScript functions at runtime. Rather than processing instructions individually, Dynarec groups them into blocks, translates them into native JavaScript functions, and caches them for reuse. By executing precompiled code instead of interpreting each instruction on demand, the emulator significantly reduces overhead and can achieve performance closer to native execution.

Recompilation Process

When execution encounters a new code block (as indicated by the Program Counter, pc), the emulator undergoes a systematic translation process:

Instruction Fetch & Decoding – The emulator reads a sequence of 32-bit MIPS instructions from memory and decodes their opcodes, registers, and immediate values.

Dispatch to Handlers – Depending on the opcode, execution is routed to a specialized code generator function (e.g., generateADD(), generateLW(), or generateBEQ()), which translates the MIPS operation into JavaScript.

Code Emission & Optimization – The handler outputs JavaScript code that mimics the MIPS instruction.

Fragment Creation & Caching – The generated code is encapsulated in a fragment, a reusable execution unit stored in a lookup table (blockMap[address]). If execution returns to this address, the emulator reuses the compiled fragment instead of recompiling it.

Execution & Synchronization – The compiled function is executed directly, and following its execution, the Program Counter (PC) and other critical state registers are updated accordingly.

The dispatcher function, generateOp(ctx), serves as the key decision-maker throughout this process. It decides which code generation routine should be used based on the instruction type. The structure of this function, along with the dispatch table used to select handlers, is shown in Appendix A.

Fragment Management & Execution Context

All instruction translations take place within a FragmentContext, an object that holds metadata regarding the current execution block. It stores values such as:

pc – The address of the current instruction being compiled.

instruction – The raw 32-bit instruction word.

postPC – The expected PC value after execution.

bailOut – A flag set when the instruction could cause an exception or needs to be handled outside Dynarec.

needsDelayCheck – A flag indicating whether special handling for delay slots is required.

Fragments are managed efficiently to avoid unnecessary recompilation. If a block has previously been translated, the emulator retrieves it from blockMap and executes it directly. However, certain instructions, such as those that trigger hardware interrupts or manipulate the TLB, may compel the emulator to revert to interpretation to ensure correctness.

A breakdown of the FragmentContext structure and its role in managing translated code can be found in Appendix A.

Handling Trivial vs. Non-Trivial Instructions

One of the optimizations js64 utilizes is differentiating between "trivial" and "non-trivial" instructions when generating compiled code.

Trivial Instructions (Optimized Code Emission)

Simple and basic operations like register moves (ADDU, AND, OR), do not alter control flow and execute deterministically. These can be emitted with minimal overhead using lightweight boilerplate, avoiding unnecessary state updates.

For example, the generateTrivialOpBoilerplate(ctx) function (detailed in Appendix A) wraps these instructions efficiently, omitting unnecessary PC synchronization logic.

Non-Trivial Instructions (Full Synchronization Required)

Instructions that impact control flow—such as branches (BEQ, JAL), memory loads (LW, LBU), and those with potential side effects—require full PC synchronization. These operations cannot be emitted as simple inline JavaScript and must include additional logic to:

Handle delay slots (for branch instructions).

Synchronize register state (to reflect architectural correctness).

Detect potential exceptions (e.g., division by zero, TLB misses).

To account for this, js64 integrates a specialized handler function, generateGenericOpBoilerplate(ctx), which ensures non-trivial instructions execute

safely. This function incorporates additional runtime checks and register updates to uphold cycle-accurate execution, as illustrated in Appendix A.

Branch Handling & Delay Slots

MIPS architecture uses a feature called delay slots to simplify pipelining in hardware. When a branch or jump instruction is executed, the instruction immediately after it is always executed. The delay slot instruction is always executed first, regardless of whether the jump/branch instruction is taken. Such behaviour should be accurately emulated, or games may misbehave.

To handle this, branch instructions like BEQ and BNE set the delayPC flag to indicate that the next instruction must execute before the branch happens. The emulator includes logic to check and apply this delay mechanism within compiled fragments.

For instance, the generateBEQ(ctx) function (referenced in Appendix A) determines whether the branch condition is met, then appropriately updates delayPC. If the branch is taken, the next PC is redirected; otherwise, execution continues sequentially.

Caching & Performance Considerations

A key benefit of dynamic recompilation is its capability to optimize performance by caching translated blocks. However, this comes with challenges:

Self-Modifying Code: Some N64 titles change their own instructions while running. js64 must identify these instances and invalidate the relevant fragments to avoid executing stale code.

Interrupt Handling: If an external event (e.g., an RCP interrupt) takes place, execution may need to pause mid-fragment and resume in an interpreter to avoid state inconsistencies.

Memory Access Optimization: js64 optimizes memory operations by reducing redundant bounds checks where possible while maintaining accuracy.

These factors are handled through synchronization mechanisms integrated within the recompiler, guaranteeing that even with enhanced performance, the emulation remains correct.

Memory Management and Addressing

Accurate emulation of the Nintendo 64 requires precise control over how memory is represented, accessed, and protected. Unlike high-level applications where memory management is abstracted away, an emulator must reproduce the behavior of low-level hardware — including how memory is laid out, how reads and writes are performed, and how unaligned or illegal accesses are handled. This is especially important for a system like the N64, where memory-mapped I/O, hardware registers, and direct memory access (DMA) are all used extensively by games and the operating system.

In js64, memory is divided into regions that correspond to different physical devices or system memory areas, such as RDRAM, ROM, and peripheral hardware. These regions are managed through a central structure called the MemoryMap, which maps portions of the 32-bit address space to the appropriate device or memory handler. The MemoryMap itself is implemented as a table of 16,384 entries (2^{14}), indexed by the top bits of each address. When the emulator receives a read or write request, it consults this table to determine which device should handle the operation (see Appendix B).

Each memory region is implemented using the MemoryRegion class, which wraps a typed ArrayBuffer and provides typed access through Uint8Array, DataView, and other views. It also includes utility methods for zeroing memory, copying data between regions, and reading or writing values of various sizes — from 8-bit bytes to 64-bit integers. These methods support both signed and unsigned variants, allowing full compatibility with the VR4300's instruction set. For implementation details, refer to the MemoryRegion definition in Appendix B.

Address Translation and Fast Paths

The Nintendo 64 uses a segmented virtual memory model, where different ranges of the 32-bit address space are mapped to different behaviors. Some ranges are cached, some are uncached, and some are reserved for kernel operations. In js64, the emulator focuses on physical addressing for performance reasons. Most accesses are performed directly against an emulated physical memory buffer, especially in the "fast path."

The fast path is an optimization used for addresses that fall within the main RDRAM range (typically 0x80000000 and above). These accesses bypass the MemoryMap lookup entirely and read or write directly into a cached DataView, resulting in much lower overhead. To perform this translation, the address is offset by 0x80000000, effectively converting a virtual address to its physical equivalent. This conversion is handled in functions like loadU32fast and store16fast, shown in Appendix B.

If an address does not fall within this range — or if it is misaligned — the emulator falls back to the slow path, which performs a full lookup using getMemoryHandler(address). This ensures that all memory-mapped I/O and edge cases (such as unaligned accesses or peripheral memory) are still handled correctly.

The slow path functions also invoke the CPU's unalignedLoad or unalignedStore methods if needed, simulating the system's behavior in cases where memory is not accessed on natural boundaries.

In addition to fast-path optimizations, js64 fully implements the Nintendo 64's software-managed Translation Lookaside Buffer (TLB). All CP0 TLB-related registers and instructions (TLBP, TLBWR, TLBWI, etc.) are supported, enabling accurate emulation of virtual-to-physical address translation. This allows the system to correctly handle dynamic memory mapping, overlays, and OS-level exception handling for TLB misses, just as on real hardware.

Masked Writes and Bit Manipulation

Because many hardware registers on the N64 are bitfields — where only specific bits are used or modifiable — js64 includes support for masked writes. These are operations where a value is only written to memory if it passes a bitmask. For instance, writing to a register may only affect bits 4 through 7 while preserving the rest.

Functions like set32masked(offset, value, mask) in the MemoryRegion class handle this logic by reading the original value, applying the mask, and writing back the result. This is critical when emulating status registers, control flags, or DMA configurations. Code examples demonstrating this behavior are available in Appendix B.

Memory Safety and Exception Handling

When a memory access falls outside of all defined regions — or if it violates alignment rules — js64 throws an emulated exception. These exceptions are surfaced through a mechanism that interrupts the normal execution flow and signals the CPU to handle the fault. For example, reading from an undefined address will

trigger a logging message and throw an error unless the access is specifically handled by the debugger subsystem.

Additionally, certain reads and writes may be intercepted for debugging or hardware emulation purposes. For instance, reads to an uninitialized region return a sentinel value (0xdddddddd), which is easily identifiable during testing.

RSP and RDP Emulation

The Nintendo 64's graphics and signal processing subsystems are centered around two specialized coprocessors: the Reality Signal Processor (RSP) and the Reality Display Processor (RDP). These components form the heart of the N64's custom vector and rasterization pipeline, handling everything from vertex transformations to triangle rasterization and audio DSP. In js64, these two systems are emulated with different strategies due to their unique demands: the RSP is fully emulated at the instruction level, while the RDP is largely abstracted through high-level emulation (HLE) via WebGL.

RSP Emulation

The RSP is a MIPS-based coprocessor (essentially a cut-down CPU) responsible for executing tasks in parallel with the main VR4300. It is typically used for vector transformations, lighting, clipping, and audio processing. js64 implements a complete instruction-level emulator for the RSP, including support for its 32 general-purpose registers, control registers, vector unit (COP2), accumulator logic, and instruction memory execution.

The core of this implementation is the RSP class, which tracks all internal RSP state, including vector registers (vpr), general-purpose registers (gprS32 / gprU32), and a 48-bit accumulator (vAcc) used in vector arithmetic. The vector register file supports 32 registers, each 128 bits wide (8 x 16-bit elements), and

includes utilities for accessing individual signed or unsigned components. These operations are essential for supporting the N64's fixed-point math routines.

Vector operations are dispatched using executeVector, which uses the lower 6 bits of the instruction to index into a vectorTable. Each vector instruction — such as VMULF, VADD, or VGE — performs its computation and then writes the result into the appropriate accumulator element or target vector. These operations often involve saturation, clipping, or condition flag updates, all of which are accurately modeled by the emulator. Examples of the vector unit's inner workings and accumulator behavior are shown in Appendix C.

The RSP also supports delay slots, conditional branching, and DMA-style memory access. The control flow logic accounts for jumps, interrupts, and yielding behavior using fields like branchTarget, nextPC, and delayPC. DMA access is simulated through registers like SP_MEM_ADDR_REG and SP_DRAM_ADDR_REG, which control how instruction and data memory are loaded or flushed. These registers are handled through moveToControl and moveFromControl, which dispatch reads and writes to simulated memory-mapped I/O. These mechanisms are also detailed in Appendix C.

The emulation is cycle-aware: each instruction advances the RSP's program counter (pc) and may set or clear interrupt flags or halt conditions (SP_STATUS_HALT, SP_STATUS_BROKE). The step() function advances execution one instruction at a time, checking for branch targets and updating control state accordingly.

RDP Emulation (via HLE)

Unlike the RSP, js64 does not emulate the RDP instruction by instruction. Instead, it uses high-level emulation (HLE) to simulate its functionality. In a

traditional low-level emulator, the RDP's command buffer and rasterizer state would be emulated in software. However, this approach is computationally expensive and impractical in JavaScript — especially for real-time rendering.

Instead, js64 intercepts RDP command lists and processes them using WebGL, effectively translating N64 rendering operations (like triangle setup, color combining, and texture mapping) into modern GPU calls. This strategy allows js64 to offload most of the RDP's heavy lifting to the host GPU, ensuring performance while retaining compatibility with a wide range of games.

Because this emulation is high-level, it abstracts away many of the underlying hardware quirks of the RDP pipeline. While this improves speed and simplifies development, it can also lead to minor differences in rendering output compared to real hardware — particularly for games that depend on pixel-perfect blending or edge-case rasterization behaviour.

That said, the abstraction layer still respects the timing and sequencing of RDP commands, ensuring synchronization between the CPU, RSP, and RDP remains correct. This is particularly important in titles that mix CPU-driven UI with GPU-rendered output, or when synchronization with vertical retrace is necessary.

Graphics Math Utilities

js64 includes a graphics module that provides mathematical helpers used by both the RSP and HLE renderer. These include modules for 2D and 3D vector math (Vector2, Vector3, Vector4), as well as transformation utilities (Matrix4x4, Transform2D, Transform4D). These are primarily used for matrix multiplications and coordinate transformations during vertex processing.

While not directly tied to RSP execution, these components play a role in translating RSP output into a format usable by WebGL, especially in the context of vertex transforms and perspective division. Their implementation supports column-major storage and fixed-point approximations for performance reasons.

For more on the graphics math routines and their usage in js64's rendering pipeline, refer to Appendix C.

Audio System

The audio subsystem in n64js emulates the Nintendo 64's Audio Interface (AI) through a low-level approach that mirrors the behavior of the original hardware. This involves interpreting memory-mapped registers, executing DMA-based data transfers from RDRAM, and scheduling audio playback with cycle-accurate timing.

When a game writes to the audio registers—specifically the DRAM address and length registers—the emulator initiates a Direct Memory Access (DMA) transfer. This process queues stereo audio samples into a double-buffered FIFO structure, managed internally by the AI subsystem (see Appendix H.1). The actual memory transfer is deferred until the length register is written, at which point the previously latched address becomes active.

Once data is queued, the startPlayback() function reads 16-bit signed samples from emulated memory and normalizes them into 32-bit floating-point values. These samples are written into an AudioBuffer and scheduled for playback via the Web Audio API. The system calculates the duration of the playback segment and applies a dynamic rate adjustment to maintain audio-visual synchronization. If the emulator detects that the audio output is leading the main timeline, it compensates by skipping a frame (see Appendix H.2).

To maintain correct sequencing and responsiveness, the emulator triggers an interrupt as soon as playback begins. It also schedules a future event to handle DMA completion after a calculated number of cycles, simulating hardware-timed behavior (see Appendix H.3). Although n64js currently does not emulate the RSP's audio microcode pipeline, the implemented mechanisms are sufficient to support audio output in most commercial titles.

Controller Input



Figure 14 N64 Controller

n64js emulates the Nintendo 64 controller by mapping external inputs to the console's expected memory representation using a low-level input system. The implementation primarily supports keyboard-based interaction, which is handled via JavaScript event listeners that respond to user actions in real time.

Upon loading, the emulator registers keydown and keyup events on the HTML document's body element. These events are intercepted and translated into bitwise updates to the virtual controller state. Each key corresponds to a specific N64 controller button—such as A, B, Z, or directional inputs—or analog stick

movement, which is simulated by scaling fixed values into the horizontal and vertical displacement registers (see Appendix I.1). This enables users to interact with games using standard keyboards while preserving the controller's behavioral fidelity.

Although the codebase contains preliminary infrastructure for gamepad support via the Web Gamepad API—including automatic detection of connected devices, validation, and button mapping logic—the system is not currently functional. The polling function responsible for reading gamepad state (updateInput()) exists but is not invoked during the emulator's frame execution cycle (see Appendix I.2). As such, while the emulator is structured to eventually support real-time gamepad input, only keyboard-based interaction is currently operational.

Debugging Tools / Developer Features

To facilitate inspection, analysis, and stepwise execution of emulated software, n64js includes an integrated JavaScript-based debugger. This tool provides visibility into the internal state of both the main CPU (VR4300) and the signal processor (RSP), supporting features such as disassembly views, register monitoring, memory inspection, and dynamic recompilation analysis.

The debugger operates as a DOM-based interface layered atop the emulation canvas. Upon activation, it exposes a tabbed environment through which users can navigate between the CPU state, RSP view, memory browser, dynamic recompiler statistics, and an event timeline (see Appendix J.1). Each tab displays real-time information derived from internal emulator structures and is updated at user-configurable intervals.

Key features include a real-time disassembly of executing code, interactive label editing, and highlighted execution flow. The current instruction is marked and register usage is color-coded for enhanced readability. The interface also allows the user to toggle breakpoints, step through execution, and trace recent memory accesses (Appendix J.2). A timeline visualization reflects co-processor events and DMA activity, helping correlate execution time with hardware-level actions.

While designed primarily for development and research purposes, the debugger remains minimally intrusive and can be hidden or shown dynamically during emulation. Its integration reflects a commitment to emulator transparency and ease of verification for both system behavior and performance characteristics.

User Interface

Js64 features a clean, minimalistic user interface designed for accessibility and ease of use. The main components include:

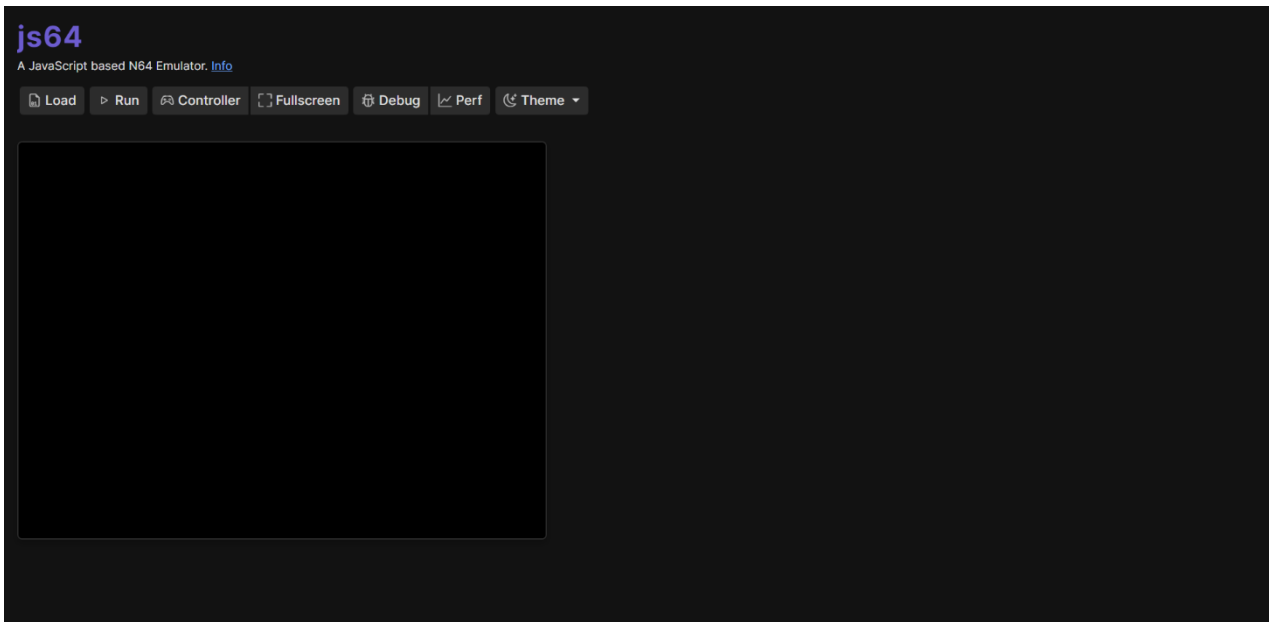


Figure 15 js64 User Interface

- ROM Loading & Execution

Buttons like **Load** and **Run** allow users to easily select and execute ROMs directly from

their local system.

- Controller Settings

A **Controller** button lets users configure input mapping to match Nintendo 64 controller layouts.

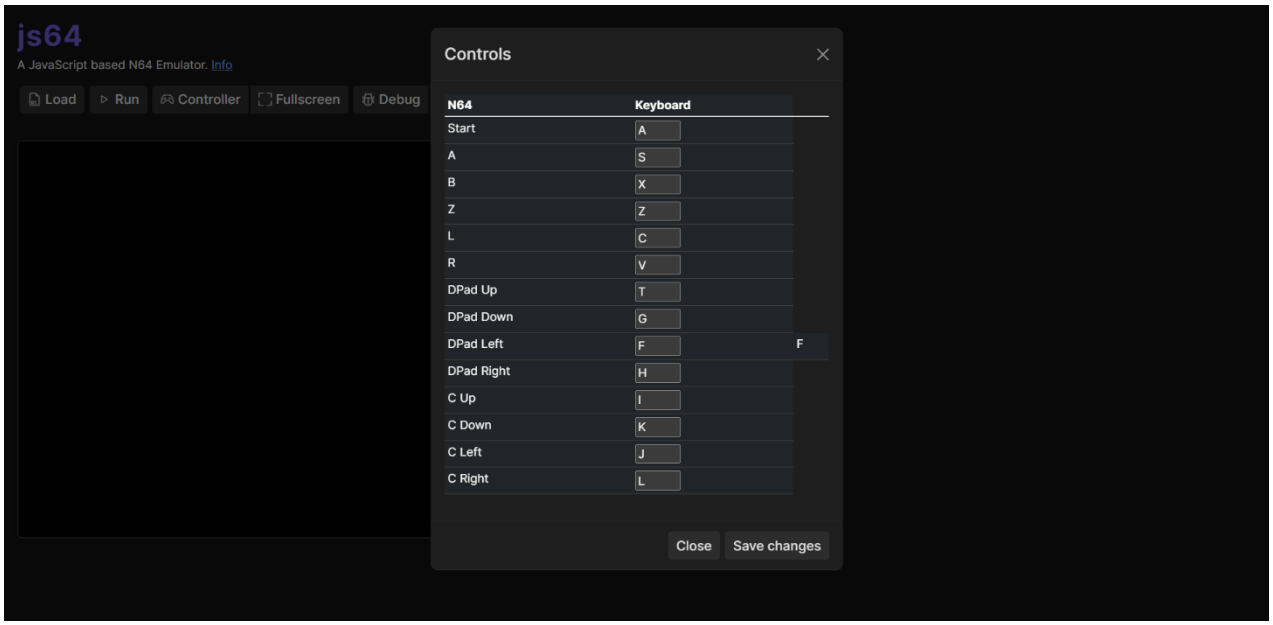


Figure 16 Controller Mapping Configuration

- Debugging & Profiling Tools

A debug menu and panel can be toggled, exposing rich real-time insights into emulator internals:

  - CPU state (PC, EPC, Cause, registers, etc.)
  - RSP state
  - Memory viewer
  - Dynarec (Dynamic Recompiler) output
  - Display list viewer
  - Texture viewer
  - Timeline analysis

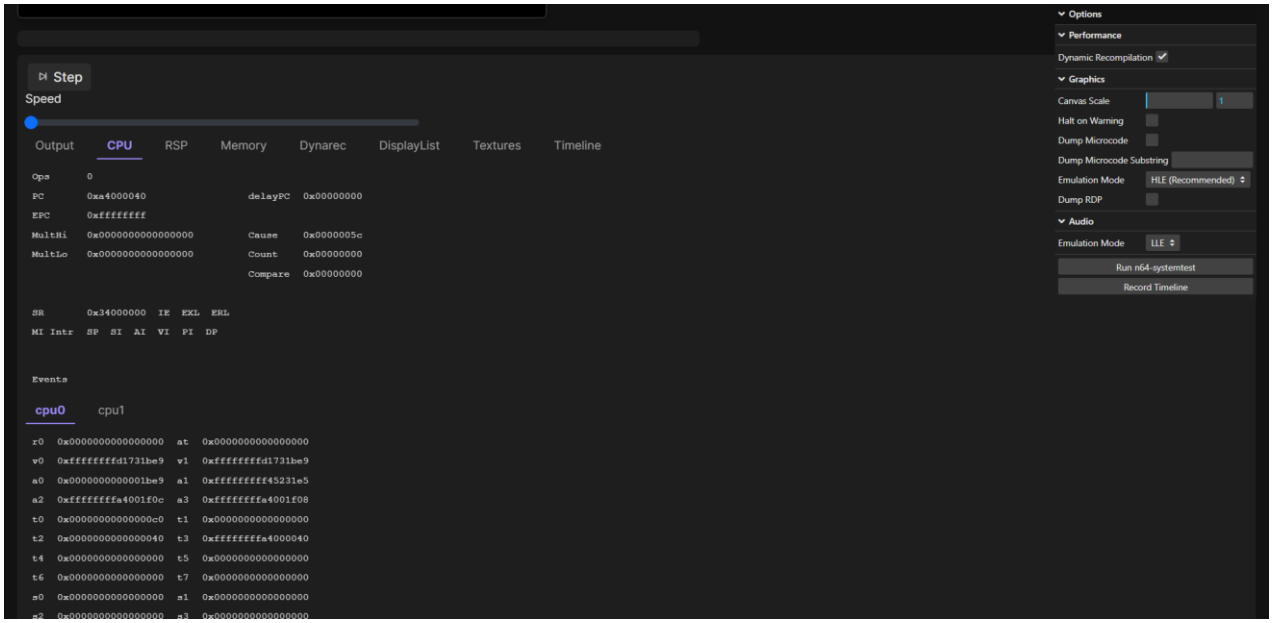


Figure 17 Debug and Profiling Tools

- Performance Monitoring

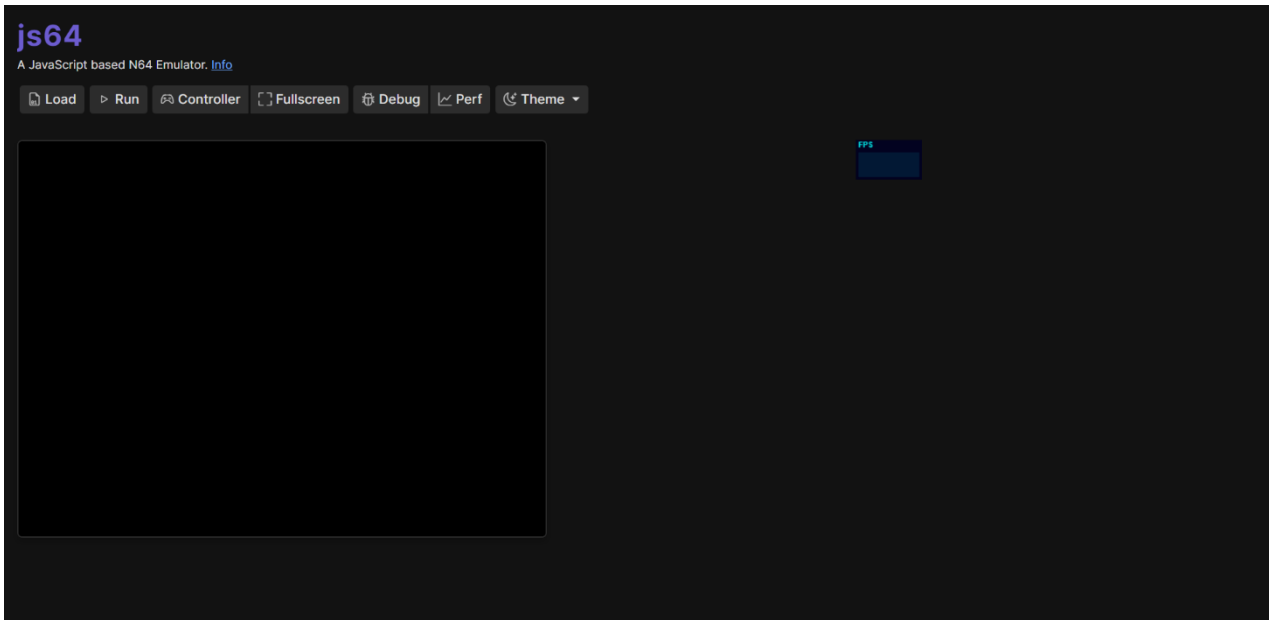


Figure 18 Performance metrics panel in js64. Toggle displays real-time FPS (frames per second), MS (millisecond latency per frame), and MB (memory usage) to monitor emulation efficiency.

- Theme Support

Key Design Decisions

Why Dynamic Recompilation?

The core of this Nintendo 64 emulator is its VR4300i's – CPU – emulation. As the central processor responsible for running game logic, managing memory, and coordinating peripheral hardware like the RSP and graphics pipeline, accurate CPU emulation is foundational. Without a reliable, and correct CPU core, nothing will function as expected – even perfectly emulated graphics or audio hardware cannot compensate for incorrect instruction decoding, memory access violations, or control flow bugs.

Emulation requires the reproduction of the physical processor, including its instruction set architecture (ISA), memory model, exceptions, memory access behaviours, interactions with co-processors, and so on.

CPU emulation fundamentally relies on two approaches. We can either interpret machine instructions one at a time, or dynamically recompile them into equivalent native code.

Interpretation decodes and executes instructions one-by-one, typically through a function pointer table to map opcodes to routines or a large switch statement. Interpreters are simple to implement, easier to debug, and ideal for High-Level Emulation (HLE) scenarios or for rapid prototyping. In earlier stages of my work, I implemented a CHIP-8 interpreter following this method, as shown in the image below:

```
static void chip8_exec_extended(struct chip8* chip8, unsigned short opcode)
{
    unsigned short nnn = opcode & 0x0fff;
    unsigned char x = (opcode >> 8) & 0x000f;
    unsigned char y = (opcode >> 4) & 0x000f;
    unsigned char kk = opcode & 0x00ff;
    unsigned char n = opcode & 0x000f;
    switch(opcode & 0xf000){

        //JP addr
        case 0x1000:
            chip8->registers.PC = nnn;
        break;

        //CALL addr
        case 0x2000:
            chip8_stack_push(chip8, chip8->registers.PC);
            chip8->registers.PC=nnn;
        break;

        //SE Vx, byte
        case 0x3000:
            if (chip8->registers.V[x]==kk){
                chip8->registers.PC += 2;
            }
        break;
```

Figure 19 Chip8 Interpreter Example

However, despite its simplicity, interpretation introduces considerable overhead due to the constant instruction dispatch and state management. For complex 64-bit architectures like the MIPS VR4300, this can quickly become a bottleneck, especially in web-based environments where performance is more constrained than native execution.

To overcome these limitations, js64 relies on Dynamic Recompilation (dynarec). Blocks of MIPS instructions are recompiled into equivalent JavaScript functions. The core concept of Dynamic Recompilation is on-demand generation of native machine code that then gets executed. When the emulator encounters a new block of code, it reads the machine code up until a break point (a jump, function call, or other flow-control change). It then translates that code from the emulated platform to the host platform, and

executes it. The result can be cached, so that the next time that code block is encountered it doesn't need to be re-translated.

Dynamic Recompilation removes much of interpreters' overhead, and can even provide opportunities for further optimization, but it needs to be re-implemented for each host system

While it does not currently operate at full cycle-accuracy, this model is sufficient for running a variety of commercial games, and the architecture remains open for future improvements.

Rationale for using JavaScript

After a re-evaluation of the technology stack, I chose JavaScript to build this emulator. Despite its reputations as a high-level, dynamic programming language not traditionally associated with systems programming, it proved itself to be a surprisingly viable alternative. Some key reasons that motivated this decision include:

Browser Accessibility: running the emulator entirely in the browser removes all barriers to entry. No installation process, no platform compatibility issues, and no dependencies beyond a modern web browser.

Ease of Deployment: the emulator can be hosted on any static web server and access instantly.

Familiar Development Environment: JavaScript has a large ecosystem of tools and libraries, and its debugging experience—especially in browsers like Chrome or Firefox—is excellent. This made development and troubleshooting easier compared to native alternatives.

Modular Architecture

Given the complexity of the project and the variety of hardware subsystems involved, a modular architecture was instrumental. Each component – including the CPU, RSP, memory map, peripherals, graphics subsystems, and debugging tools – is encapsulated in its own JavaScript module.

This layout serves several important purposes, including:

- Maintainability: Code is easier to reason about, debut, and extend.
- Testability: Individual modules can be tested independently.
- Performance Optimization: It's easier to focus optimization efforts without affecting unrelated functionality
- Scalability: Future features described in future work can be added without rewriting core systems.
- Compatibility with JavaScript: Given JavaScript's single-threaded and event-driven nature in browsers, modularity also reduces the risk of shared-state bugs or timing regressions.

Tradeoffs & Limitations

Performance vs. Accuracy

One of the most significant design decisions was to abandon the original plan of writing a native desktop application, cycle-accurate emulator in C in favour of a browser-based emulator using JavaScript and WebGL. While the initial choice involved precise and faithful modelling of the console's hardware down to the clock cycle – instrumental to ensure a perfect Nintendo 64 console experience – the complexity and low-level nature of such an undertaking proved to be unsustainable. Challenges included frequent crashes, difficulty of debugging low-level memory interactions, and a steep increase in development time.

In contrast, the web-based approach allowed for faster iteration, accessibility (no installation needed), and easier debugging, at the cost of some accuracy. This emulator relies for the most part on HLE – high level accuracy – , and involves various shortcuts taken in order to replicate the console's internal behaviour rather than emulating it to the transistor.

Why Cycle Accuracy Matters — And Why It Was Deferred

While cycle-accuracy emulation is crucial for a faithful Nintendo 64 experience, it comes at a steep cost in performance and complexity. For example, in the game Air Strike Patrol, a shadow is drawn beneath the player's relies on cycle-precises scanline effect. In emulators that prioritize speed over accuracy, this shadow is often missing. As shown in the screenshots below, the left image (outlined in green) demonstrates the correct behaviour with the shadow present, while the right image (outlined in red) shows the shadow missing. Many emulators that favour speed simply skip these effects, unintentionally degrading gameplay.



Figure 20 Left - Cycle-accurate emulation with the aircraft shadow displayed. Right - Performance-optimized emulation missing the shadow due to skipped raster effects

Similarly, the Nintendo 64's architecture — with its tightly integrated CPU, RSP, RDP, and memory subsystems — makes it particularly hard to emulate accurately without deep knowledge of hardware interactions. Although a future version of this emulator or a separate native build may aim for this level of fidelity, the current web-based implementation prioritizes accessibility, development speed, and maintainability.

To achieve cycle accuracy, it will be necessary to create a detailed CPU, RSP, and RDP pipeline model, where each instruction is executed in lockstep with a simulated hardware clock. Memory accesses, cache behavior, bus contention, and timing of co-processor operations would all need to be synchronized at the cycle level. This would also involve carefully reproducing hardware quirks and edge cases observed in the original Nintendo 64. Techniques like building a cycle scheduler to manage events and using extensive hardware testing (e.g., comparing against real console behaviour with test ROMs) would be essential to validate accuracy.

JavaScript & WebGL vs. C & SDL2

During the early stages of development, the project was meant to be written in C, using SDL2 for graphics, input, and audio. The reason why I went through with this technology stack is due to the low-level control over hardware emulation it offers, making it perfect for cycle-accurate timing. However, the complexity of the project quickly became overwhelming especially within the limited time frame of a capstone project.

After exploring alternative approaches, I chose to work with JavaScript, which while traditionally not associated with low-level systems programming, offered a range of APIs that could be leveraged for an emulation project. JS provided unexpected flexibility in handling low-level management though tools like ArrayBuffer and DataView, and WebGL allowed efficient rendering within the browser.

The decision to move to JS was surprising at first given its reputation. However, it's an astonishing demonstration of the power of today's web browsers and JS engines. This made it a practical solution that ultimately enhanced the project's development speed, compatibility, and accessibility, even if some level of accuracy had to be sacrificed for performance and speed.

Incomplete Hardware Emulation

Some components of the N64 hardware remain unimplemented or are only partially implemented.

Graphics are handled using HLE, with GBI0 mostly implemented, but GBI1 and GBI2 only partially supported, and LLE rendering not implemented

at all. This leads to graphical inaccuracies in some games, such as missing effects, flickering, or broken geometry.

Audio emulation includes a basic LLE path, but HLE audio is not yet implemented, which may lead to silence or incorrect sound playback in some games.

Controller support is functional through static key bindings, but more advanced features like configurable controls and Gamepad API integration are not yet implemented.

Save system supports various storage types (Mempak, Eeprom 4k/16k, SRAM, FlashRAM) with persistence via localStorage, but lacks savestate functionality and import/export features, which are often expected by modern emulator users.

Planned Features & Improvements

While js64 is already capable of running some Nintendo 64 commercial titles, there are several planned improvements to enhance functionality, compatibility, and usability:

- **Fix Graphics Issues:**

  While many games are playable, there are visible rendering errors in some titles. Upcoming enhancements aim to refine the HLE graphics pipeline and fix edge cases in the RDP emulation to achieve a visual experience closer to that of the original hardware.

- **Save Game Import/Export:**

  A key objective is to support the loading and saving of native N64 save files (EEPROM, SRAM, FlashRAM). This would enable users to transfer their saves between js64 and other emulators or even actual N64 consoles.

- **Savestates:**

  The addition of save states would allow users to quickly save and restore game progress at any point, regardless of the game's internal save system—this is an essential enhancement for both casual gamers and testers.

- **Gamepad Support:**

  Introducing support for external controllers (through the Gamepad API) would significantly enhance the gaming experience.

Data Presentation

This section comprises two main parts:

It presents the results of functional correctness testing using the n64-systemtest homebrew ROM.

Second, it analyzes performance benchmarks collected from both JS64 (browser-based emulator) and Project64 (native emulator).

Together, these results provide a comprehensive view of the emulator's current accuracy and real-time performance.

Homebrew Testing

```
n64-systemtest 2.0.2 (base=1 timing=0 cycle=0 cp0-hazards=0)
Finished in 2.05s. Base: Failed 147 of 3554 tests (95% success rate)
Heap range: 8023fab0 to 80300000
Test 'LW with address not sign extended' failed: Exception expecte
d but none seen

Test 'SW with address not sign extended' failed: Exception expecte
d but none seen

Test 'cart-writing: Temp value  decay' with '20' failed: a == b expec
ted, but a=0xbadc0ffe b=0x1234567. Expect that value is gone afte
r 110 loop iterations

Test 'cart-writing: Temp value  decay' with '10' failed: a == b expec
ted, but a=0xffe0000 b=0x1234567. Expect that value is gone after
110 loop iterations

Test 'cart-writing: Temp value  decay' with '8' failed: a == b expect
ed, but a=0xfe000000 b=0x1234567. Expect that value is gone after
 110 loop iterations

Test 'cart-writing: Temp value  decay' with '40' failed: a == b expec
ted, but a=0xffffffff b=0x1234567. Expect that value is gone afte
r 110 loop iterations
```

Figure 21 Output of n64-systemtest 2.0.2 validation suite

To assess the functional correctness of the emulator, the **n64-systemtest** homebrew suite was executed. This test ROM includes over 3,500 low-level checks targeting CPU behaviour, memory handling, and exception triggering. The emulator achieved a **95% pass rate**, with 147 tests failing.

Notable failures involved:

- Incorrect handling of load/store operations with non-sign-extended addresses.
- Inaccurate behaviour in memory decay simulations during cartridge writes, where temporary values persisted longer than expected.

These results suggest the core architecture is implemented correctly for the most part, but some hardware-specific edge cases—particularly those involving memory timing and exception conditions—require further refinement.

Performance Evaluation: Project64 on Ryzen 5 5600H + RTX 3050 Laptop

FPS (Frames Per Second) and frametime data were captured using *CapFrameX* during a standardized gameplay session of Super Mario 64 EU ROM.

FPS (Frames Per Second) Analysis

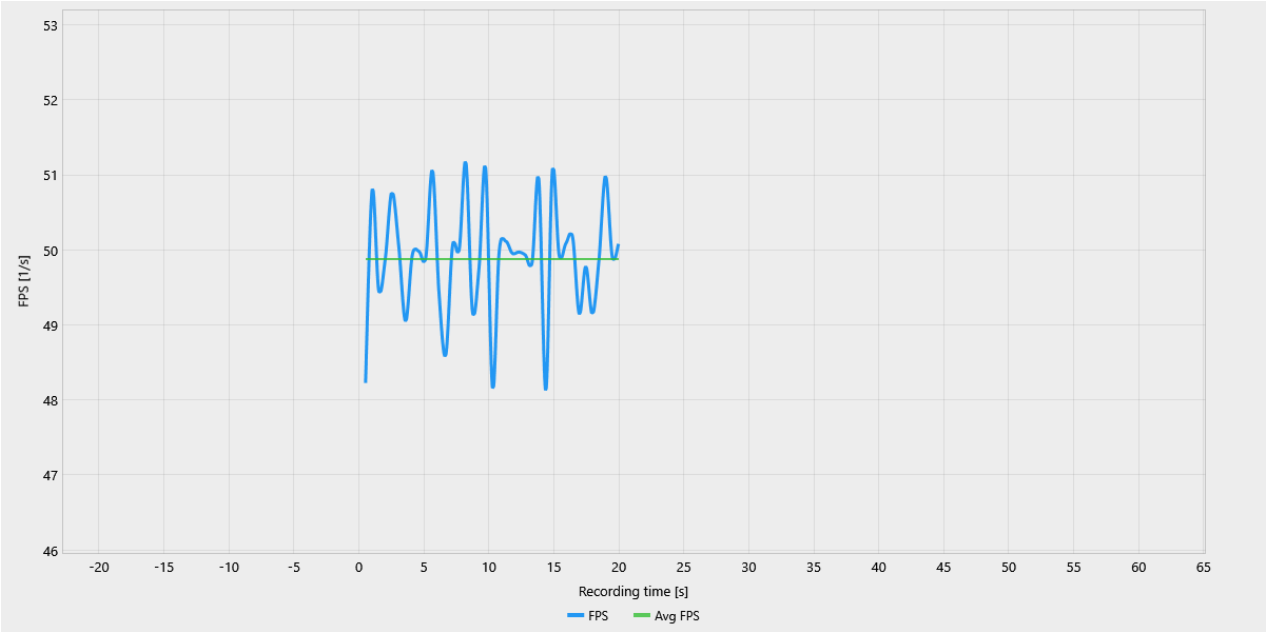


Figure 22 Frame rate (FPS) performance over time during Project 64 emulation

- **Average FPS:** Approximately **50 FPS**, fluctuating between **46–53 FPS**.
- **Target Comparison:** This is below the Nintendo 64's original target of **60 FPS**, suggesting that Project64 was unable to consistently maintain full-speed emulation.
- **Key Contributing Factors:**
    - **CPU Bottleneck:** N64 emulation remains heavily single-threaded; thus, the Ryzen 5 5600H's multi-core advantage may not be fully utilized by Project64's dynamic recompiler.

- **GPU Underutilization:** Although the RTX 3050 is a capable GPU, N64 emulation workloads are rarely GPU-bound. Minor driver overheads could still affect performance.
- **Configuration Settings:** Synchronization methods, resolution scaling, and graphics plugin behavior may also impact observed framerates.

- **FPS Stability:**
  - Moderate fluctuations (~7 FPS variance) were recorded.
  - Severe stutters were absent, but the framerate was not perfectly locked to a consistent value.
- **Recording Timeline:**
  - Recording data spanned for 20 seconds.

Overall, gameplay remained playable but did not achieve console-native framerate stability.

Frametime Analysis

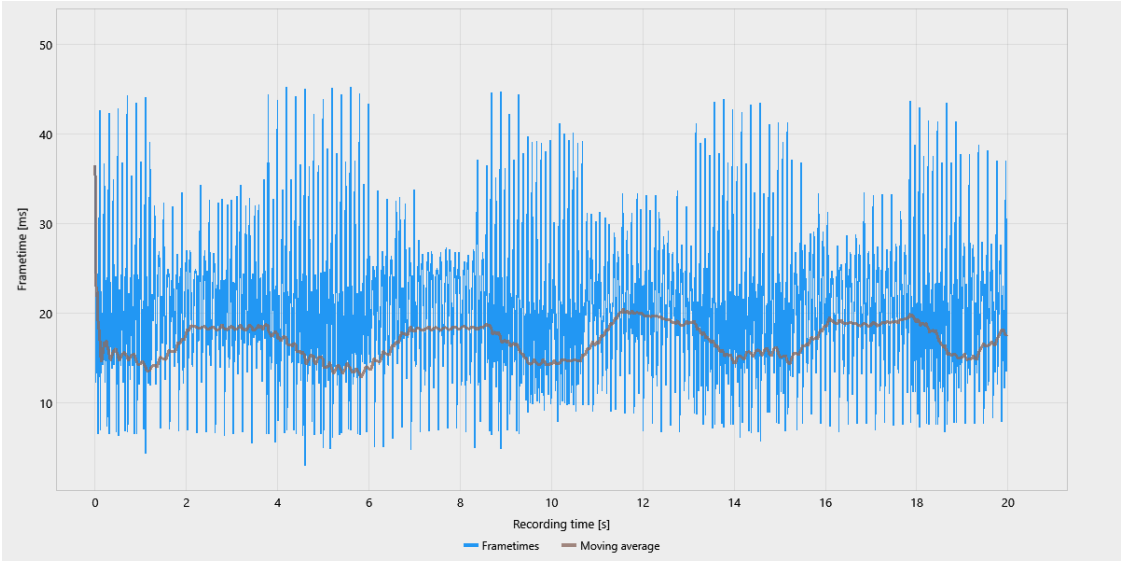


Figure 23 Frametime analysis (ms) with during Project 64 emulation

- **Average Frametime:** Frame delivery times ranged primarily between **16 ms and 22 ms**.
    - A perfect 60 FPS experience would require ~16.67 ms per frame.
- **Frametime Stability:**
    - The collected data showed frequent frametime spikes, with some frames requiring up to **40–50 ms** to render.
    - The moving average revealed periodic cycles of increased frametime, possibly linked to scene complexity or intermittent emulation overhead.
    - Despite the overall smoothness, frametime variability suggests the presence of **microstutters** during gameplay.
- **Potential Causes for Irregularities:**
    - Real-time shader compilation, particularly when using plugins such as *GLideN64*.
    - Background system processes introducing CPU scheduling noise.
    - Resource management fluctuations within the emulator during dynamic scenes.

The frametime analysis reinforces the FPS findings, confirming that while Project64 delivers mostly smooth emulation, achieving perfect frame pacing under all conditions remains a technical challenge.

Smoothness, Low FPS, and Stuttering Breakdown

Here's a visual breakdown of the gameplay experience based on smoothness, low FPS occurrences, and stuttering events:

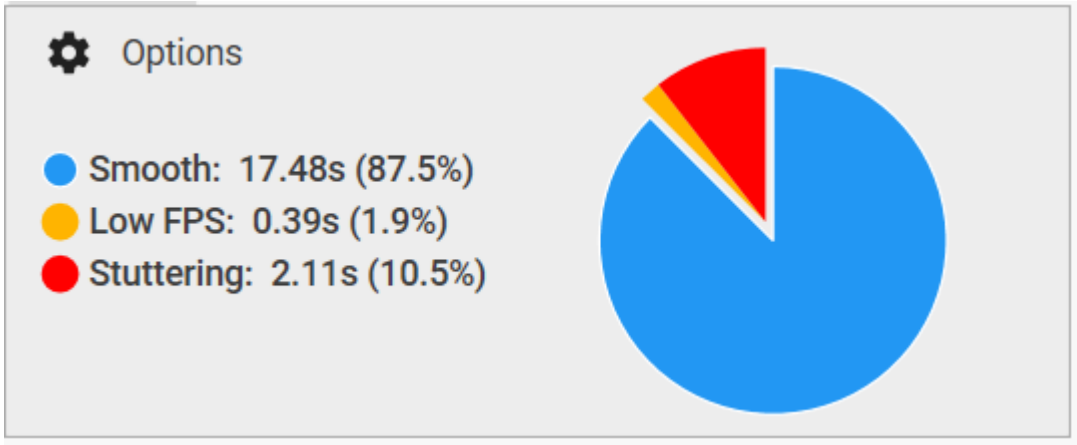


Figure 24 Frame delivery quality analysis during Project 64 emulation

- **Smooth**: 17.48 seconds (87.5%)
- **Low FPS**: 0.39 seconds (1.9%)
- **Stuttering**: 2.11 seconds (10.5%)

This chart confirms that the majority of gameplay time remained smooth, with **occasional microstutters** and very few **low FPS incidents**.

Performance Evaluation: JS64 on Ryzen 5 5600H + RTX 3050 Laptop

To assess the real-world performance of JS64, frametime and FPS data were captured using CapFrameX during a standardized 20-second gameplay session of Super Mario 64 EU ROM.. The results provide insights into the emulator's smoothness, stability, and efficiency under typical workloads.

Frametime Analysis

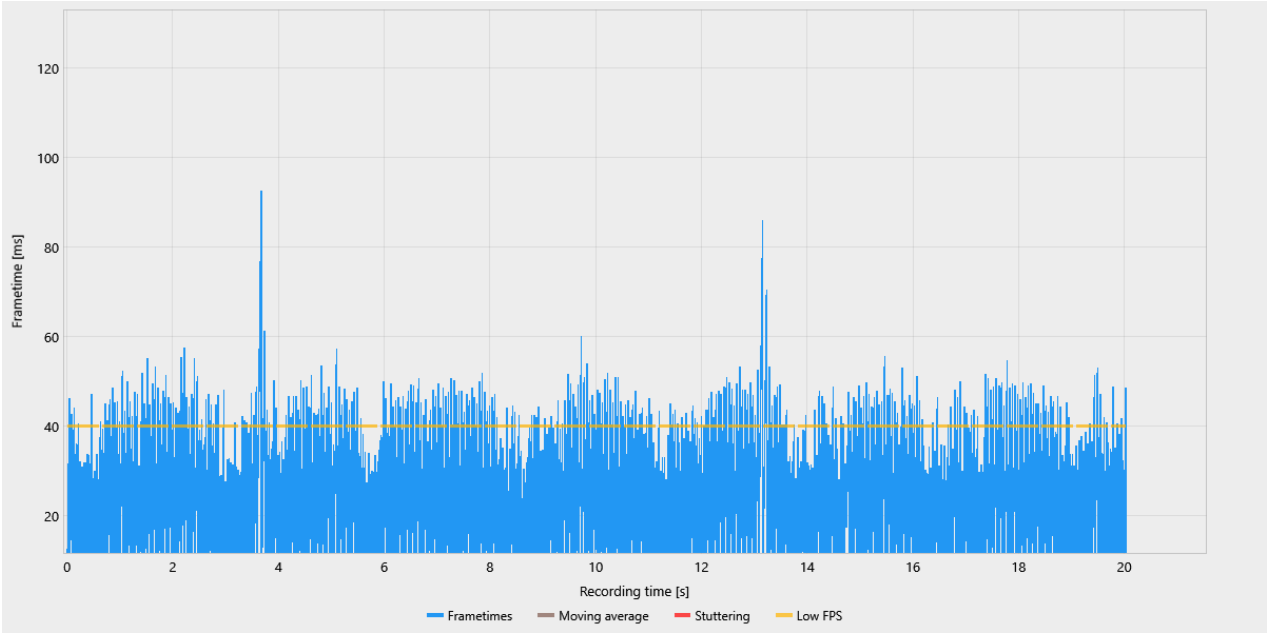


Figure 25 Frametime analysis (ms) during js64 emulation

Frametime measurements reflect the consistency of frame delivery, with lower and more stable values indicating smoother performance:

- Average Frametime: Ranged between 20–60 ms, with frequent spikes reaching 100–120 ms during scene transitions or complex rendering tasks.
- Target Benchmark: For a perfect 60 FPS experience, frametimes should stabilize at ~16.67 ms. JS64's higher averages confirm it struggles to maintain full-speed emulation.
- Critical Observations:
    - The moving average (smoothed over 0.6s) revealed cyclical frametime increases, suggesting periodic bottlenecks in the emulator's main loop.
    - Severe stuttering was detected for 19.43s (97% of the session), with frametimes often doubling or tripling the ideal value.

FPS (Frames Per Second) Metrics

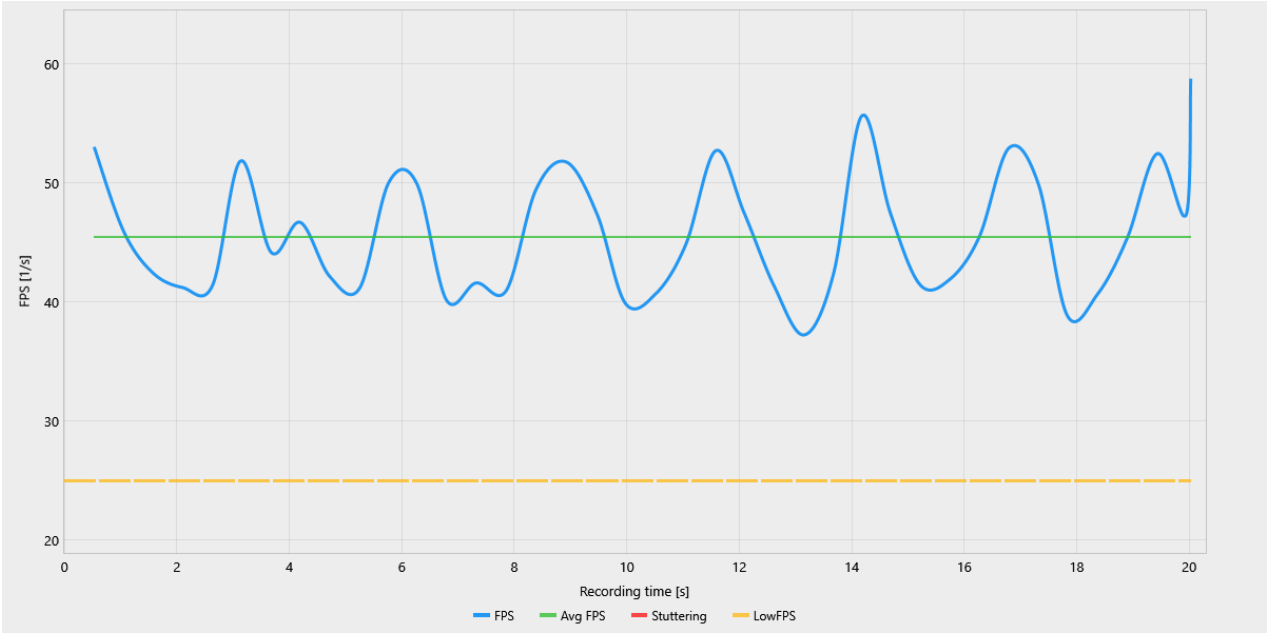


Figure 26 Frame rate (FPS) performance over time during js64 emulation

FPS data complements frametime analysis by quantifying rendering throughput:

- **Average FPS**: **~30 FPS**, with fluctuations between **20–50 FPS**.
- **Performance Bottlenecks**:
    - **JavaScript Interpreter Overhead**: The absence of dynamic recompilation forces *JS64* to execute N64 instructions sequentially, incurring significant CPU penalties.
    - **Garbage Collection Pauses**: Spikes in frametimes often correlated with JavaScript engine cleanup cycles, interrupting emulation continuity.
    - **GPU Underutilization**: Despite the RTX 3050's capability, *JS64*'s rendering pipeline rarely saturated GPU resources, as N64 workloads remain CPU-bound.

Smoothness Breakdown

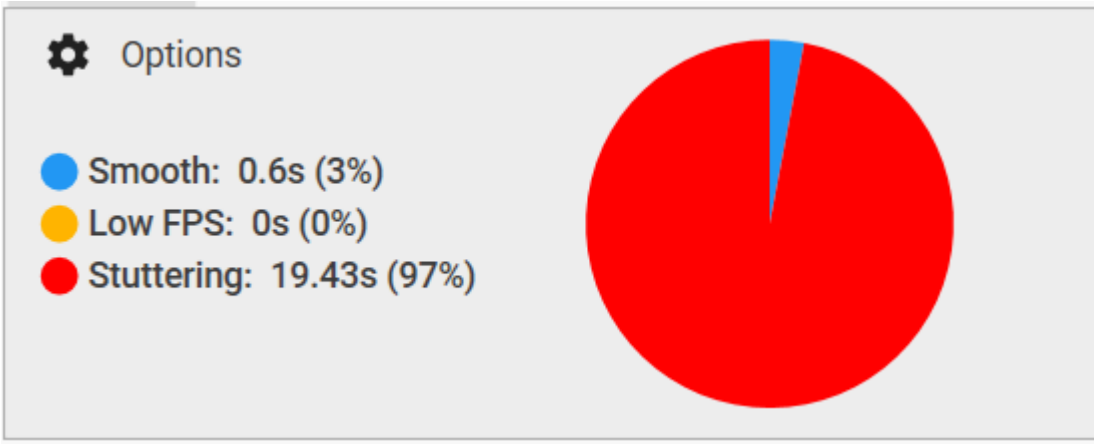


Figure 27 Frame delivery quality analysis during js64 emulation

The session was categorized into three key performance states:

- **Smooth**: **0.6s (3%)** — Periods where frametimes adhered to the 16.67 ms target.
- **Low FPS**: **0s (0%)** — No sustained drops below playable thresholds.
- **Stuttering**: **19.43s (97%)** — Dominant state, characterized by erratic frametimes and inconsistent frame pacing.

Simulations, Results, & Interpretation

Overview

To evaluate js64's emulator's functionality and performance, a series of tests and benchmarks were conducted based on both homebrew test ROMs and commercial games. Additionally, considering before js64 my capstone was to be a C based native emulator, I compared js64's performance to Project 64 – a mature, native emulator – to assess strengths and limitations.

Tests were run on a Ryzen 5 5600H CPU and an RTX 3050 GPU, using the Chrome browser for js64 and a native environment for Project 64.

Results

Correctness Validation: Homebrew Testing

The results of the homebrew testing demonstrate that JS64 successfully reproduces the core behavior of the Nintendo 64's VR4300 CPU and memory model. A **95% pass rate** across thousands of hardware tests indicates a high level of functional correctness, especially considering the complexity of the system and the use of a JavaScript-based dynamic recompiler.

The few remaining inaccuracies are primarily tied to highly specialized edge cases — such as memory access patterns relying on undocumented hardware behavior — that are rarely encountered during normal game execution. Thus, for most commercial games and homebrew applications, JS64 is expected to provide reliable and accurate performance.

Nevertheless, achieving 100% correctness would require deeper simulation of low-level timing details, stricter address alignment enforcement, and more precise replication of memory bus decay characteristics. These are important future targets if cycle-accuracy or edge-case compatibility is desired.

Overall, these results validate that the architectural design choices made in JS64 — particularly the dynamic recompiler approach — are effective at replicating the core execution behavior of the original hardware while balancing performance and browser constraints.

Metric	JS64 (Browser + Dynarec)	Project64 (Native Dynarec)
Average FPS	~30 FPS	~50 FPS
Smooth Gameplay	3% of session	87% of session
Stuttering	97% of session	10% of session
Frametime Stability	Poor	Good

Table 2 Performance Benchmark: jS64 vs Project64

- **JS64** uses a **dynamic recompiler**: it dynamically translates MIPS VR4300 machine code into JavaScript functions at runtime, improving over pure interpretation.
- Despite the dynarec, **performance is still limited** by:
    - JavaScript's execution overhead versus native machine code.
    - Browser scheduling, garbage collection pauses, and WebGL bottlenecks.

Compared to Project64's native dynamic recompiler, JS64's JavaScript dynarec achieves **functional success** but still suffers from **browser-imposed performance ceilings**.

Technical Implications

The data underscores two fundamental challenges for *JS64*:

1. **JavaScript Limitations**: The language's single-threaded nature and lack of low-level memory access hinder cycle-accurate emulation.
2. **Browser Overhead**: Even with hardware acceleration, browser-based execution introduces unpredictable latency from rendering APIs and background processes.

While *JS64* demonstrates functional correctness, its performance profile currently lags behind native emulators. Future optimizations could mitigate these bottlenecks.

Learning Strategies

When I first thought about developing an emulator to learn more about low-level programming for my capstone, I was very intimidated by the idea. Emulators are notoriously complex, and I had heard about the N64's intricate architecture. The following is how I approached the learning process:

Building a Foundation with Simpler Systems

To ease into emulator development, I joined multiple forums and communities focused on systems programming and emulator design. My favourites being Reddit's /r/emudev, and the Emulator Development Discord Server. I asked questions, shared progress, and learned from the community.

I was advised to dip my foot in emulator development through simpler projects for simpler architectures. Thus, I started by writing a CHIP-8 emulator in C. CHIP-8 has a simple instruction set and minimal hardware features, which made it a perfect introduction to emulation concepts such as opcode decoding, memory management, and basic CPU emulation.

I then proceeded to creating a Super Nintendo Entertainment System (SNES) emulator, also in C. These projects helped build my confidence, and laid the technical groundwork for tackling a more complex system like the N64.

Learning from Existing Emulators

I spent a significant amount of time reading through the source code of popular N64 emulators, such as:

- **CEN64** – known for its cycle accuracy and low-level realism
- **Mupen64** – a widely-used plugin-based emulator
- **Project64** – a feature-rich emulator with years of development

Reverse-engineering parts of these projects and observing how they implemented key systems like the RSP and the RDP was very helpful in shaping and creating js64.

Studying Computer Architecture

To supplement my hands-on learning, I completed CS301: Computer Architecture from Saylor Academy. The course covered:

- The MIPS instruction set
- Pipelining and parallel execution
- Memory access and caching mechanisms
- Hardware/software integration principles

These concepts were directly relevant to understanding the Nintendo 64's R4300i processor and system architecture.

Consulting Official Documentation

Official documentation from Nintendo and NEC was instrumental for the development of js64. I consistently referred to official hardware documentation and trusted community resources. These included processor manuals, memory layout guides, and technical references for the RSP and RDP components.

When emulator's behaviour differed or was unclear, these sources served as the ultimate reference for aligning my implementation with the actual hardware.

Conclusion

The development of js64 demonstrates that emulation of a complex 1990s console like the Nintendo 64 is possible entirely within modern web browsers using normal web technologies: JavaScript and WebGL. Through the user of a dynamic recompiler, js64 overcomes many performance limitations associated with classic interpreted emulators, delivering a playable experience for homebrew and commercial ROMs.

Homebrew testing results show that js64 reproduces the behavior of the VR4300 CPU and memory management system with **high fidelity**, achieving a **95% pass rate** on low-level hardware tests. While its performance remains below that of native emulators like Project64 — largely due to JavaScript runtime constraints and browser overhead — JS64 successfully balances

functional correctness with accessibility.

Beyond technical achievement, JS64 fulfils a larger mission: offering an educational resource for gaining insights into low-level system architecture, CPU pipelines, memory management, and the concepts of dynamic recompilation. It encourages the democratization of emulation technology by rendering intricate system internals available online, eliminating the necessity for specialized hardware or software installations.

While there is room for future improvement — including more accurate RDP emulation, savestate functionality, and expanded controller support — this project establishes a solid foundation. In doing so, it demonstrates the valuable role that web technologies can play in software preservation, making retro gaming more accessible, and promoting educational efforts in computer science and digital heritage areas.

References

Carlson, M. (2022, February 23). Nintendo 64: Architecture and History - Matthew Carlson - medium. *Medium*. https://medium.com/@matthewfcarlson/nintendo-64-architecture-and-history-8a01cf503a6a

Conley, J., Andros, E., Chinai, P., Lipkowitz, E., & Perez, D. (2004). *Use of a game Over: Emulation and the Video game Industry, a white paper*. Northwestern Pritzker School of Law Scholarly Commons. https://scholarlycommons.law.northwestern.edu/njtip/vol2/iss2/3

Copetti, R. (2025, February 2). *Nintendo 64 Architecture*. The Copetti Site. https://www.copetti.org/writings/consoles/nintendo-64

Emulation accuracy - Emulation General Wiki. (n.d.). https://emulation.gametechwiki.com/index.php/Emulation_accuracy

Emulation Accuracy, Speed, and Optimization - mGBA. (2017, April 30). https://mgba.io/2017/04/30/emulation-accuracy

Farfeleder, S., Krall, A., & Horspool, N. (2007). Ultra fast cycle-accurate compiled emulation of inorder pipelined architectures. *Journal of Systems Architecture*, *53*(8), 501–510. https://doi.org/10.1016/j.sysarc.2006.11.003

Guttenbrunner, M., Becker, C., & Rauber, A. (2010). Keeping the Game Alive: Evaluating Strategies for the Preservation of Console Video Games. *International Journal of Digital Curation*, *5*(1), 64–90. https://doi.org/10.2218/ijdc.v5i1.144

Johansson, C. (2022). *Video Game Preservation and Emulation from three perspectives: developers, archivists and gamers (Master's Thesis)*. Södertörn University.

McFerran, D. (2025, January 10). Why is N64 so hard to emulate in 2025? Modern Vintage Gamer investigates. *Time Extension*. https://www.timeextension.com/news/2025/01/why-is-n64-so-hard-to-emulate-in-2025-modern-vintage-gamer-investigates

N64 Introductory Manual. (1997). http://n64.icequake.net/doc/n64intro/kantan/step1/index1.html

Van Gorsel, M., Leenaars, M., Milic-Frayling, N., & Palm, J. (2014). Evaluation and strategies of digital preservation: UNESCO's role in facing the technical challenges. *ICA Annual Conference*.