Apollo 11: One Small Step Technical Guide

Supervisor: Prof. Geoffrey Hamilton
By Om Yashwant Dighe 21200292
Kenneth John Ras 21787441
02/04/2025

1. Introduction

This project is a comprehensive software emulation of the Apollo Guidance Computer (AGC), based upon the earlier build and much more complete and precise project called Virtual Apollo Guidance Computer. Designed to replicate the AGC's hardware and software with technical precision, the emulator reconstructs the computer's unique **15-bit architecture**, **real-time operating constraints**, and **interaction with a DSKY**, while integrating a modern **3D web-based simulation** to visualize mission-critical events like lunar descent and orbital maneuvers.

At its core, the emulator preserves the engineering marvel of a system that operates with only **2 kB of RAM**, **72 kB of ROM**, and a **1.024 MHz clock**, executing tasks with cycle-accurate timing (~11.7 μs per instruction). It models the AGC's memory hierarchy—including parity-checked core rope ROM and magnetic-core RAM—and its priority-driven interrupt system (e.g., RUPT_DOWNRUPT for engine burns).

The project extends beyond emulation with a **React/CesiumJS frontend** that renders Apollo 11's mission phases in 3D. Key features include:

- **Trajectory Visualization**: Precomputed Saturn V launch and LM descent paths, derived from NASA telemetry and physics models, displayed using CesiumJS's WGS84-compliant globe.
- **Real-Time Telemetry**: WebSocket integration synchronizes the AGC's register states (e.g., DSKY verb/noun displays) with spacecraft motion, replicating the Apollo Guidance Computer's role in navigation.
- **Interactive Mission Checklists**: UI components gate progress based on AGC status (e.g., program validation), while synchronized audio and particle effects enhance realism during engine burns.

By merging historical accuracy with modern web tooling, the system demonstrates how the AGC's constrained resources powered humanity's lunar exploration, while providing an immersive platform for education and technical analysis.For further information, the Virtual AGC project would be great place to start as they were our primary sources of information and we also used one of their open source tool for our project.

2. Motivation

The Apollo 11 mission was a monumental achievement for human history, engineering and space exploration. This project aims to build a web simulation of the important phases of the Apollo 11 mission phases, such as the Saturn V launch and the Lunar Module (LM) descent and landing. Motivations for this project are:

- **Technical Exploration:** To experiment with modern web technologies (React.js, CesiumJS, Node.js, WebSockets) for creating complex simulations in the browser.
- **Preserving History:** To commemorate the Apollo 11 mission by visualising its trajectory and the important events based on historical data and simulations.
- **Integration challenge:** To simulate the interaction between the Apollo Guidance Computer (AGC) and the spacecraft by linking a virtual AGC (yaDSKY2) to the simulation.
- **Systems Focus**: Emulating the AGC's 1960s-era architecture (e.g., 16-bit word lengths, priority-driven executors) in a modern Rust environment, blending historical accuracy with modern tooling.

3. Research

Developing the simulation required research into several areas:

- **Apollo Mission Data:** Sourcing accurate trajectory data, timelines, spacecraft parameters (mass, thrust, dimensions), and event timings from NASA archives, technical reports (i.e. Post-Flight Trajectory Analysis), and historical mission transcripts and journals.
- **Orbital mechanics:** Understanding the physics governing rocket launches, orbital maneuvers, gravity turns, and powered descent. This involved studying relevant equations of motion, atmospheric drag models, and gravitational influences.
- **Saturn V & LM Systems:** Learning about the staging sequence of the Saturn V rocket and the operation of the Lunar Module's descent. Learning about the basic principles of the Apollo Guidance Computer and its Display/Keyboard (DSKY) interface.
- **AGC Emulation:** Investigating existing AGC emulators (like yaAGC) and their communication protocols (i.e. yaDSKY telnet interface) to understand how to interface with them. The Virtual AGC project was a reference for most of the sources as it is great implementation already.
- **Rust Learning Curve:** Rust's syntax and functionality (e.g., iterators, result handling) differed sharply from languages like Python or JavaScript.
- **Instruction Set and Memory:** Reviewing historical AGC assembly programming guides to understand instruction mnemonics.
  Analyzed MIT's original documentation for the Block II AGC, detailing its 15-bit word architecture, core rope vs. erasable memory, and memory banking techniques.
- **DSKY Implementation:** Referencing the yaDSKY2 project's virtual DSKY implementation, which emulates the AGC-DSKY communication protocol. Studied NASA's DSKY interface diagrams to replicate the layout, including the 7-segment displays and verb/noun logic.
- **CesiumJS:** Exploring CesiumJS's capabilities for 3D geospatial visualisation, rendering celestial bodies, loading 3D models (gLTF), plotting trajectories (CZML), and managing time-dynamic simulations.

- **Web Technologies:** Researching best practices for React component design, state management, WebSocket communication for real-time updates, and integrating CesiumJS into a React application.

4. Design

4.1 System Architecture

The system consists of a client-server architecture:

- **Frontend (Client):** This is a single-page React application and is responsible for rendering the 3D visualisation using CesiumJS. It displays mission information (checklists, timers, status), and communicates with the backend via WebSockets.
- **Backend (Server):** This Express.js server is responsible for:
    - Serving the frontend application build.
    - Establishing WebSocket connections with clients/
    - Monitoring the AGC emulator's output (yaDSKY's output.txt)
    - Parsing AGC data and status.
    - Broadcasting AGC status and other relevant data to connected clients.
- **AGC Emulator (core):** The RAGC and yaDSKY processes run separately, simulating the Apollo Guidance Computer. ragc-core module is responsible for executing block 2 instructions, managing memory, priority scheduling and handling interrupts.
- **Downrupt Module:** Handles telemetry downlink using protocol channels 34/35 and is responsible for sending data to external systems via TCP.
- **DSKY Module:** The RAGC has an implementation of the yaDSKY socket protocol which allows it to communicate with the DSKY.The AGC core writes to dedicated channels (e.g., CHANNEL_DSKY/0o10) using the IoPeriph::write method and handles frontend keypress inputs.
- **Rust Ecosystem:** Using Rust's toolchain and libraries for performance, like crossbeam to provide low latency and thread-safe channels, tokio for async runtime to handle I/O and log to provide frontend-agnostic api, while clap to handle command line input.
- **Trajectory Calculation (Offline):** Python scripts pre-calculate the Saturn V launch and the LM descent trajectories based on physics models and historical data. These scripts generate CZML files which are consumed by the frontend.

4.2 Frontend Design

- **Component Structure:** The UI is broken down into React components (i.e. HomeScreen, EarthScene, MoonScene,) and reusable components (i.e. MissionChecklist, AgcConnectionIndicator).

- **State Management:** The application mainly uses React's useState and useRef hooks for managing component state, Cesium viewer instances, websocket connections, and audio playback.
- **Routing/Scene Management**: The main App component controls which scene (home, earth or moon) is currently active and has loading screens for transitions.
- **Integrating Cesium:** The EarthScene and MoonScene components encapsulate the CesiumJS viewer logic, including initialisation, CZML loading, entity tracking, and event handling (i.e. clock ticks).
- **Real-time Updates:** WebSocket messages trigger state updates in the relevant components (e.g. updating the AgcConnectionIndicator)

4.3 Backend Design

- **Websocket communication:** Uses the ws library to handle WebSocket connections and broadcasts messages to client.
- **File Watching:** Uses Node.js fs.watch to monitor output.txt for any changes. This triggers the parsing and broadcasting logic.
- **AGC Status Logic:** Parses output.txt to determine the connection status ("yaDSKY is connected.") and loaded program type (based on the pattern of the register output.)
- **Error Handling:** Basic error handling for file system operations and WebSocket communication.
- **Graceful shutdown:** Includes signal handlers (SIGINT, SIGTERM) to attempt cleanup (like clearing output.txt) before it exits.

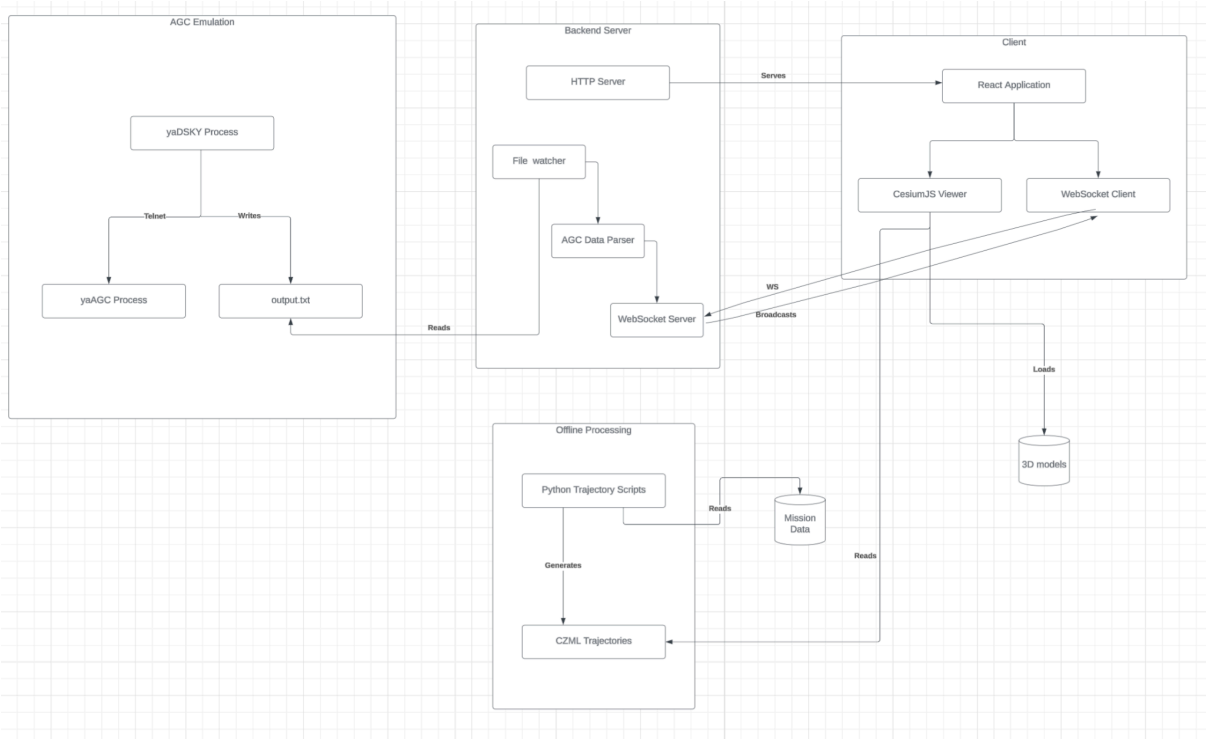


Fig. 1 System architecture Diagram

4.4 Emulator Design

The original AGC software is written in AGC assembly language and is stored on rope memory, hence it cannot be changed in operation but some key parts can be overwritten using the DSKY.

- Core Emulation
    - AGC instruction set is divided into 5 categories, each implemented as a rust module
        - Arithmetic
        - Control Flow
        - I/O
        - Load/Store
        - Interrupts
    - Memory: RAM for temporary storage and ROM for storing AGC software.
        - Register and I/O ports mapped to peripherals
        - Erasable banks (dynamic) and fixed banks (static) managed to segment memory for AGC's 15-bit architecture.
    - Decoder: Translates opcodes into executable operations
    - Interrupt: Handles peripheral interrupts (e.g. keypresses)
- Peripherals
    - DSKY (Display and Keyboard): Emulates Apollo DSKY with a 7-segment display and keypad. Uses TCP sockets (127.0.0.1:19697) for external UI interaction.
    - Manages display flashing and keypress handling
    - Downrupt: Simulates communication with external systems (e.g. telemetry)
    - Listens on 127.0.0.1:19800 for outgoing data
- yaDSKY2
    - Using open-source yaDSKY program as implementation for the DSKY.
    - Modified from the original yaDSKY code to output R3 register digits to a buffer which can be read by the frontend.
- DSKY Protocol
    - This crate defines packet format, serialization/deserialization logic and acts as a bridge between ragc-core and interfaces.
    - Each Packet is 4 bytes long and follows the format:
        - struct Packet {
          _hw_packet: bool, // Unused (reserved for future hardware-specific packets)
          io_addr: usize,   // Target I/O address (e.g., DSKY register)
          io_value: u16,   // Data value (e.g., display segments, keypress code)

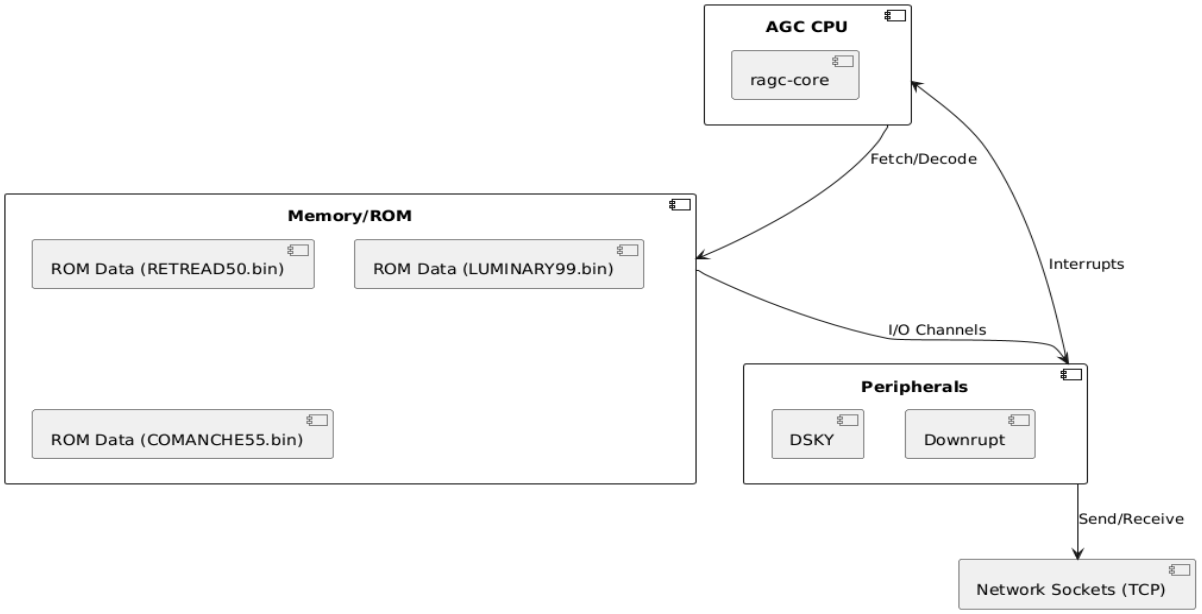


Fig. 2 Data flow diagram

4.5 Generating Trajectories

4.5.1 Defining the Goal & Constraints

The main goal was to create visually convincing flight paths for the Saturn V launch and the LM descent that could be played back smoothly in CesiumJS. The key constraint was performance: running a full, complex physics simulation in real-time in the browser was not feasible. This led directly to the decision to pre-calculate the trajectories offline. Python was a natural choice for this offline work because of its scientific libraries (Numpy, SciPy) and its data plotting capabilities.

4.5.2 Research

Mission Timelines: Liftoff, stage separations (S-IC, S-II), S-IVB burns (orbital insertion, TLI), Powered Descent Initiation (PDI), landing. This involved looking at NASA mission reports and timelines.
Spacecraft Parameters: How much did each stage weigh (structure and propellant)? How much thrust did the engines produce? This data was found in technical documents and sources like the Saturn V manual or Apollo mission summaries.
Environment: Earth's rotation speed (omega), radius (Re), gravity (g0), atmospheric density model (rho0, hscale). Similar constants were needed for the Moon (Re, g0, mu).
Locations: Launch pad coordinates (Kennedy Space Center 39A), approximate landing site coordinates (Tranquility Base).
Basic Physics: Newton's laws (F=ma), the rocket equation (how mass changes with fuel burn), gravitational force (inverse square law), and basic atmospheric drag equations.

4.5.3 Choosing the Simulation Engine

The main task was to solve the equations of motion over time. scipy.integrate.solve_ivp is the standard tool in Python for this. It takes a function defining the derivatives of the system's state and numerically integrates them over time.

4.5.4 Modeling the Dynamics (derivatives function)

This was the core of the script. This function calculated the instantaneous rate of change for the spacecraft's state (position, velocity, mass) by summing forces like gravity, thrust (including LM throttle), and atmospheric drag (for Earth launch), and accounting for mass change due to fuel consumption.

4.5.5 Implementing Guidance (approximately)

Simplified guidance rules were used: Saturn V: A time-and-altitude-based target flight path angle (gravity_turn_program) simulated the pitch-over maneuver, with the derivatives function gradually steering towards the target.LM Descent: Altitude and descent rate determined target pitch (descent_pitch_program) and throttle (descent_throttle_program) to approximate a controlled landing profile. Handling Events (Staging/Burns): Simple time checks (if/elif/else based on t) within the derivatives function controlled mission phases by changing thrust, mass flow rates, and guidance targets.

4.5.6 Converting to CZML

The simulation results (time steps, state vectors) were transformed into CZML JSON format for CesiumJS. This involved converting time to ISO 8601, calculating Cartesian positions, setting velocityReference for automatic orientation where possible, and defining availability intervals to show/hide models correctly.

4.5.7 Iteration and Debugging

The process was highly iterative: adjust Python parameters -> run simulation -> check matplotlib plots -> generate CZML -> load in CesiumJS -> observe visually -> repeat. Visual feedback in CesiumJS was crucial for refining the model and guidance heuristics until the trajectories looked somewhat accurate.

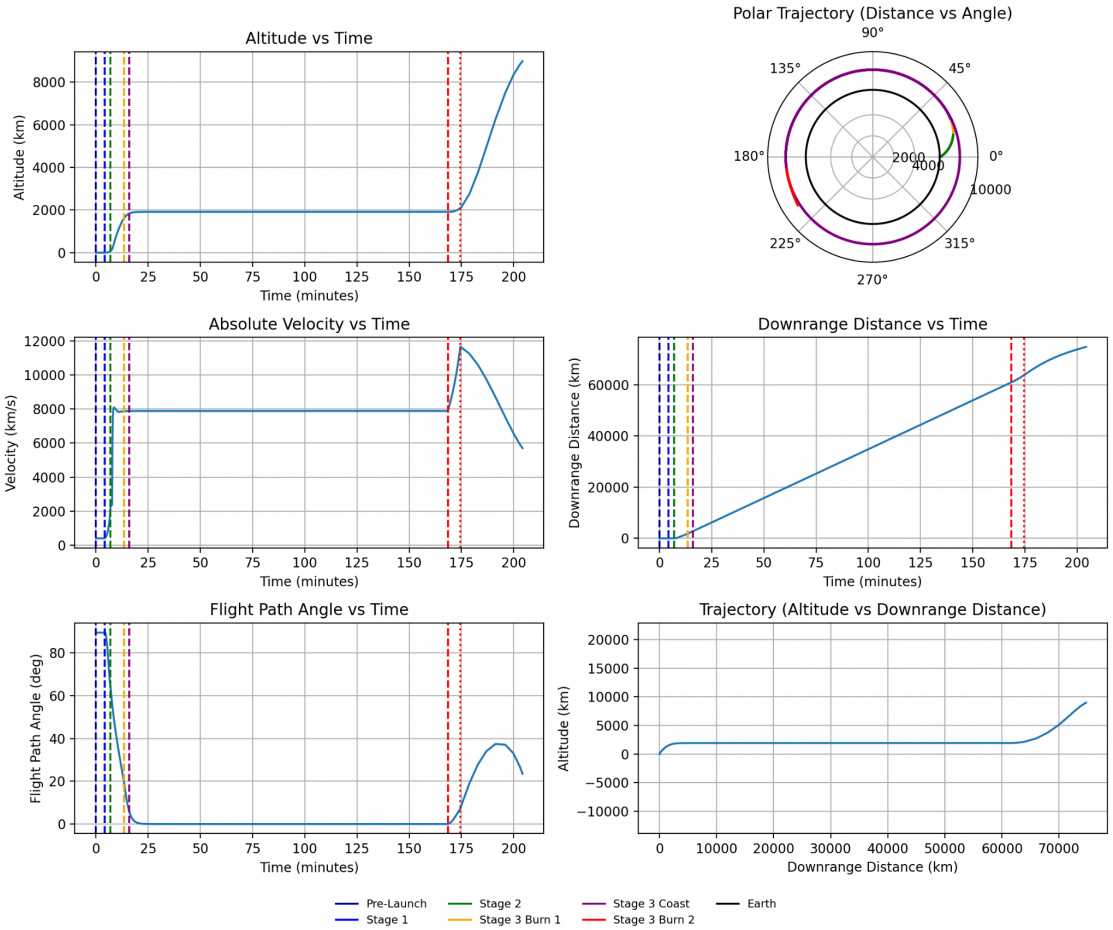


Fig 3. Python graphs of Saturn V launch & orbit trajectories and post-flight analysis

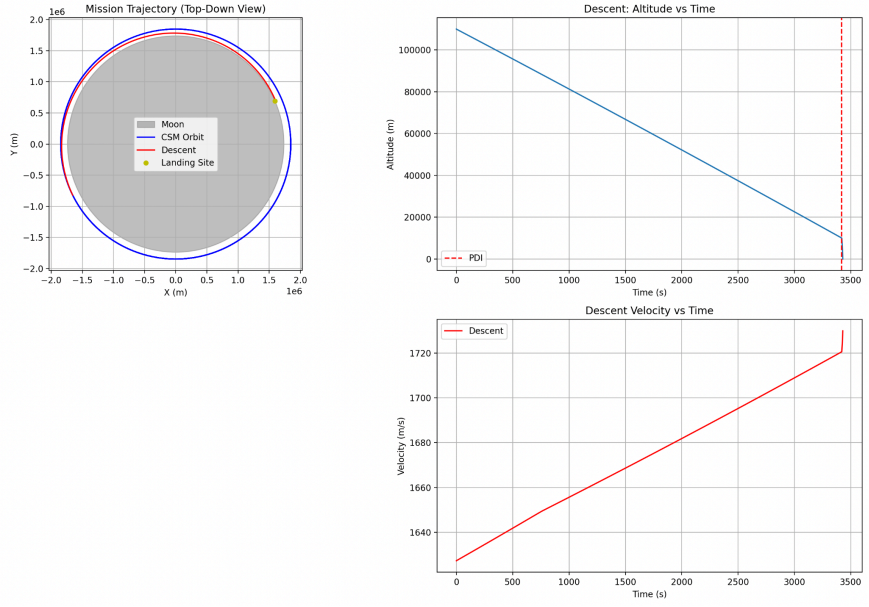


Fig 4. Python graphs of CSM and LM orbit and trajectories and post-flight analysis

4.6 UML Diagrams

4.6.1 Class Diagrams

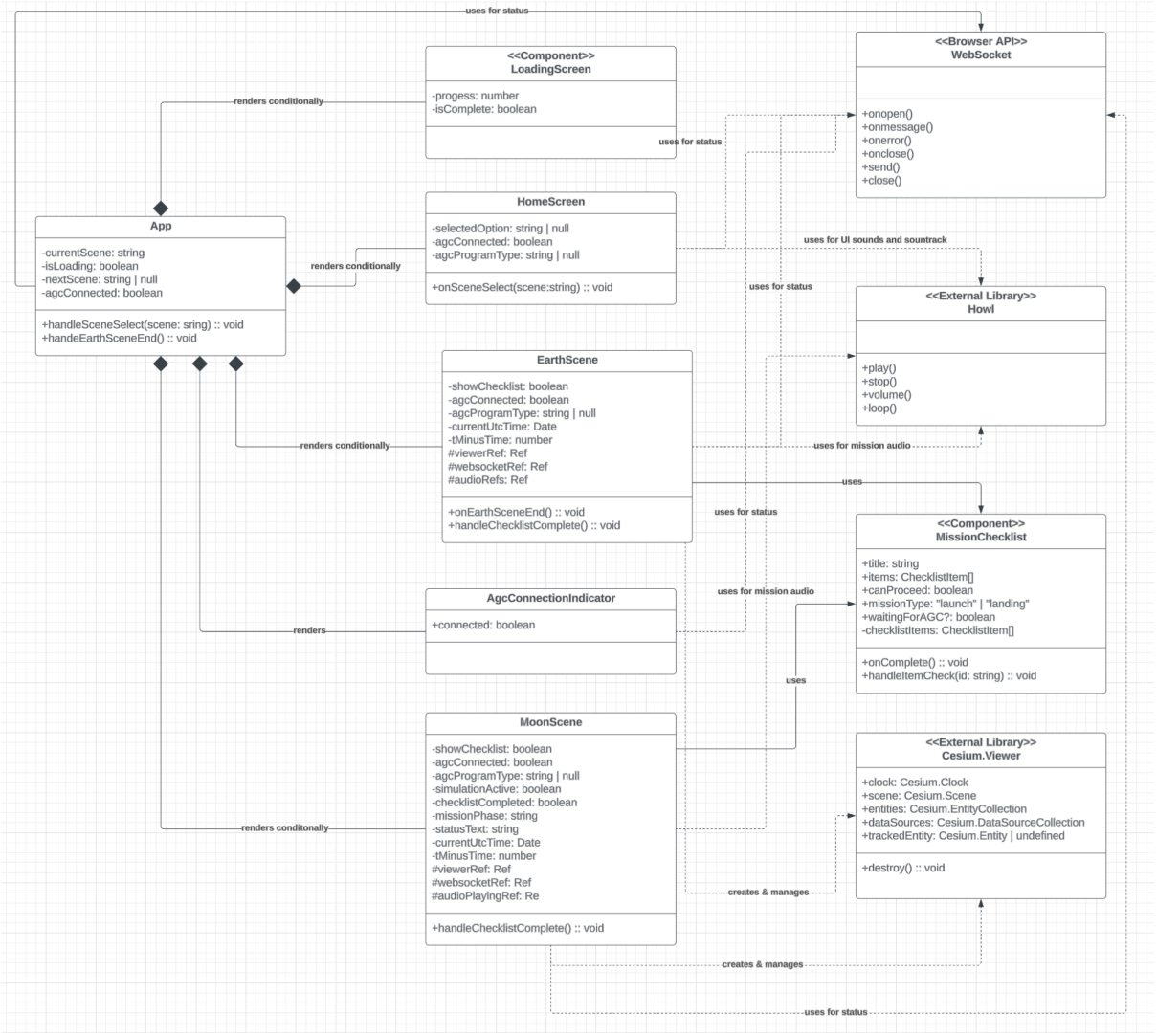


Fig. 5 Class diagram (frontend)

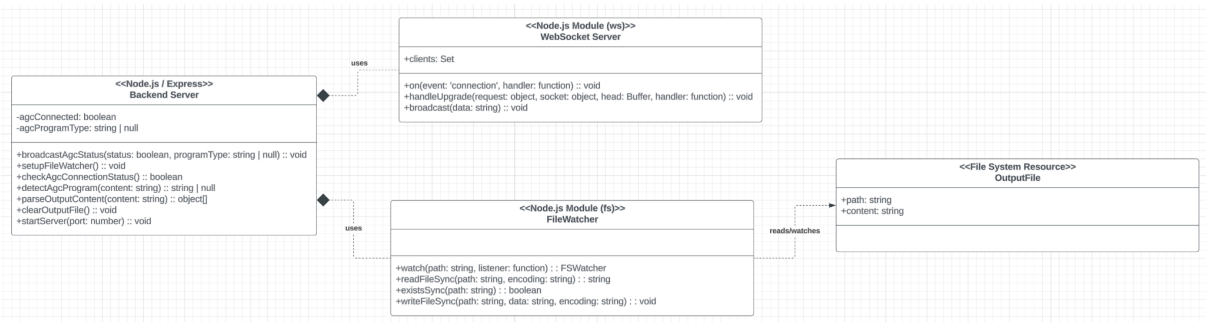


Fig. 6 Class diagram (Node.js backend)

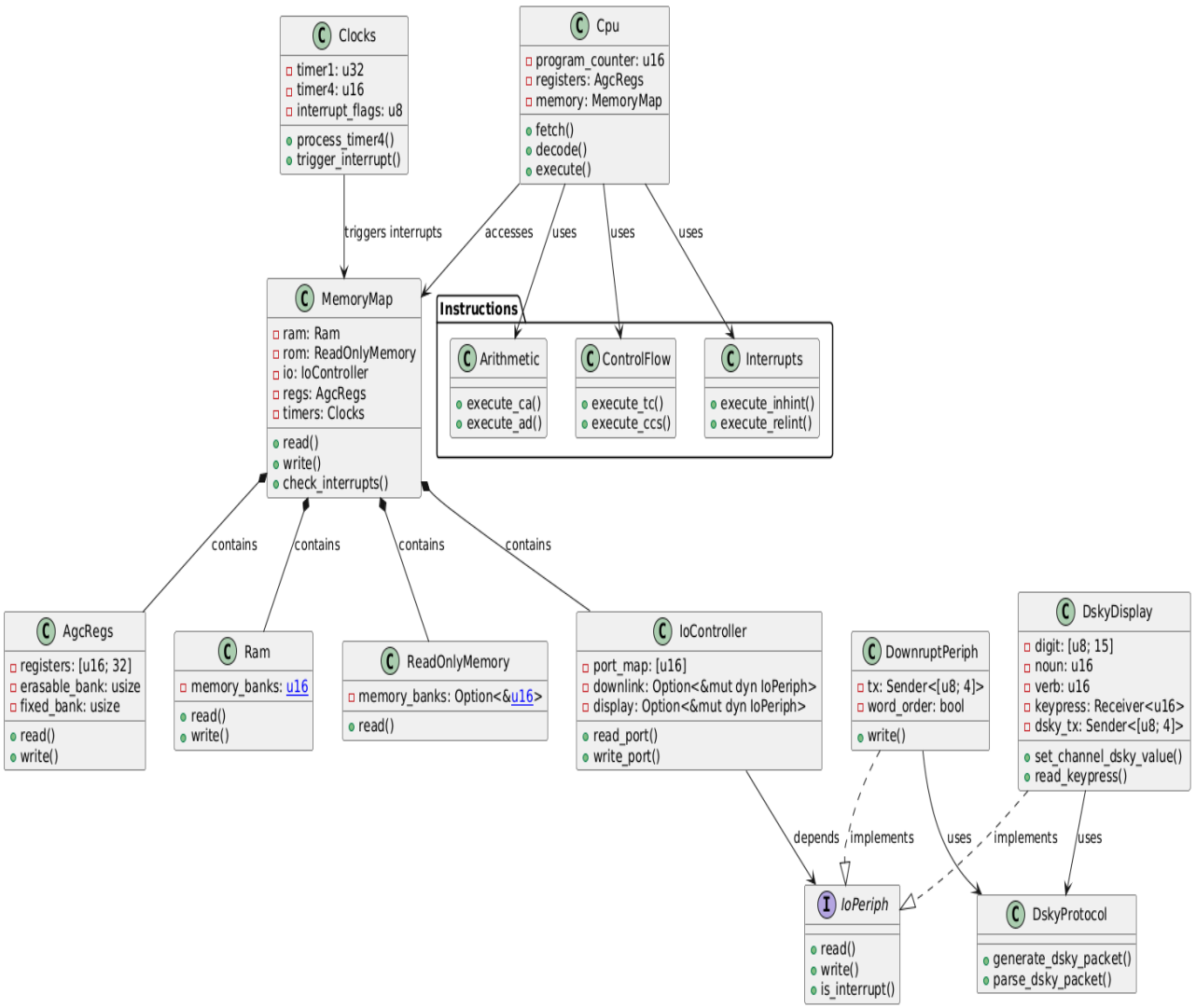


Fig. 7 Class Diagram (AGC Emulator)

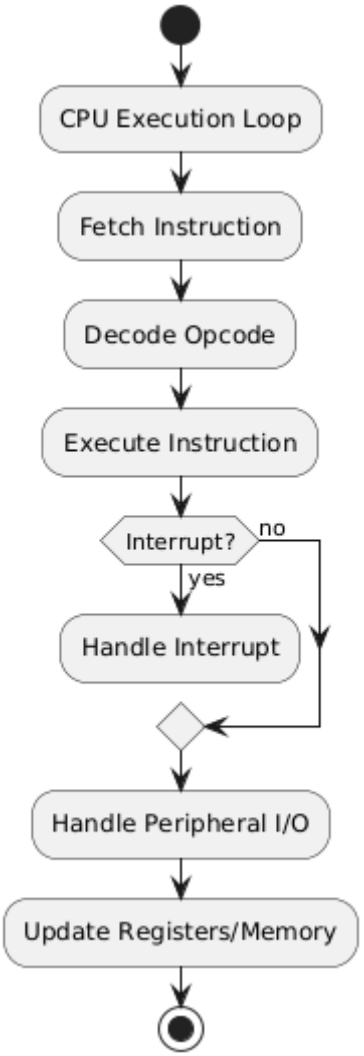


Fig. 8 Control Flow Diagram (AGC Emulator)

4.6.2 Sequence Diagram

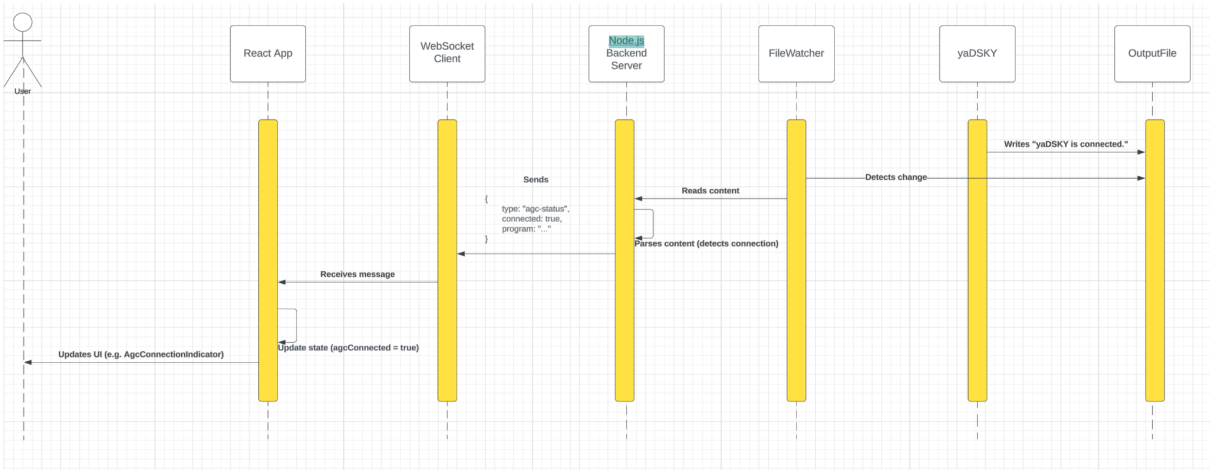


Fig. 9 AGC status sequence diagram

5. Implementation

5.1 Key Technologies

- **Frontend:** React, TypeScript, CesiumJS, Tailwind CSS, Howler.js (audio), Blender (models)
- **Backend:** Node.js, Express.js, ws (WebSockets)
- **Trajectory Generation:** Python
- **yaDSKY2:** Open Source program as an implementation of DSKY from Virtual AGC project
- **AGC Emulation:** Rust, yaDSKY protocol

5.2 Core Features Implemented

5.2.1 Frontend Features

- **Scene Switching:** Basic navigation between Home, Earth, and Moon scenes.
- **Cesium Viewer Setup**: Initialising the Cesium viewers for the Earth and the Moon, and loading 3D terrain/models and post-processing effects.
- **CZML Trajectory Loading:** Loading pre-calculated CZML files for Saturn V launch and LM descent.
- **Entity and orbit tracking:** Following the Saturn V / CSM-LM / LM entities during the simulation. Users are able to orbit around the entity, detach and free roam, and visualise the orbit trajectory.
- **Time management**: Synchronising the Cesium clock with the mission timelines that are defined in CZML files and controlling when the animation starts and stops.
- **AGC monitoring:** The file watcher on the Node.js backend monitors yaDSKY's output.txt.
- **AGC Status Communication:** Backend parses connection status and the program type, broadcasting the updates to the frontend via WebSocket.
- **Mission Checklists:** Interactive checklists that gate progress based on AGC status (e.g. program loaded)
- **Audio Integration:** Using Howler.js to play background audio, thruster sounds, and mission control chatter synchronised with simulation time.
- **Mission timers:** Displaying UTC and T-/T+ timers based on the Cesium clock.
- **Particle Effects:** Basic thruster particle effects which are visible based on the burn intervals

5.2.2 Frontend Testing

Category	Item	Type	Location	Focus
Testing: React Frontend Components	App	Unit Test	App.test.jsx	Initial rendering, ensuring HomeScreen is displayed by default.
	HomeScreen	Unit Test	HomeScreen.test.jsx	Initial rendering, scene selection logic, "Begin Experience" button visibility and state based on selection and connection status. Mocks UI components and WebSocket.
	LoadingScreen	Unit Test	LoadingScreen.test.jsx	Rendering of initial loading messages. Uses fake timers.
	MissionChecklist	Unit Test	MissionChecklist.test.jsx	Rendering based on props (title, missionType, items), state of the "Proceed" button (based on item completion, canProceed, waitingForAGC), onComplete callback execution, conditional display of status messages (required items, AGC status).

	AgcConnection Indicator	Unit Test	AgcConnectionIndicator.test.jsx	Correct display of "Connected" or "Disconnected" text based on the connected prop.
Testing: Node.js Backend Server	Server	Integration Test	server.test.js	End-to-end behavior: Server startup, watching the Rust output file (output.txt), WebSocket client connection handling, broadcasting initial status (agc-status), sending parsed register data (agc-output), detecting specific program sequences (apollo11-launch-sequence-detected).
Tools and Frameworks (Frontend & Backend)	Vitest	-	-	Test runner, assertions (expect), mocking (vi), fake timers.
	React Testing Library	-	-	Rendering React components (render), querying the DOM (screen), simulating user events (fireEvent).
	Node.js	-	-	child_process (for spawning server), fs (for file manipulation).

	ws	-	-	WebSocket client library for interacting with the backend server during tests.

5.2.3 Backend Features

- Main (Entry Point Function)
    - Initialises the emulator, handles CLI input, and runs the CPU cycle loop.
    - Key Components:
        - CLI Setup: Uses clap to parse subcommands (retread50, luminary99, comanche55) for ROM selection.
        - Signal Handling: Uses ctrlc to exit on Ctrl+C.
        - ROM Loading: Loads ROM binaries (e.g., RETREAD50_ROPE) based on CLI input.
        - Peripheral Initialisation:
            - DskyDisplay: Emulates the DSKY interface.
            - DownruptPeriph: Handles telemetry communication.
        - CPU Execution:
            - Creates a MemoryMap with ROM, peripherals, and interrupt queues.
            - Runs a real-time cycle loop (agc_cpu.step()) to emulate AGC timing (~11.7 µs per cycle).
- Instruction set:
    - Arithmetic
        - AD - add two values
        - MP - multiples 15-bit signed integers
        - DV - divides 30-bit values
        - INCR - Increments a register
    - Control Flow
        - TC - jumps to an address
        - BZF - branches if accumulator is zero
        - TCF - jumps to fixed memory address
    - Interrupts
        - INHINT - disable interrupts
        - RELINT - enable interrupts
        - RESUME - restore cpu state after interrupt
    - I/O -
        - READ/WRITE
        - WOR - performs a bitwise OR between accumulator and I/O port
        - RXOR - XORs with I/O port value
    - Load/Store -

        - XCH - exchange accumulator with memory location
        - DCA - loads a double word from memory
        - QXCH - Swaps register_return with memory address
- Memory Implementation:
    - Registers
        - Manages general-purpose registers (e.g. ACCUMULATOR, MULTIPLIER), fixed/erasable banks for memory switching
    - Ram
        - Volatile memory divided into banks (e.g erasable_bank)
        - Values masked to 15 bits
    - Rom
        - Non-volatile storage for AGC software (e.g., RETREAD50.bin). Fixed bank addressing with remapping via BANK_MAPPING
    - I/O Ports (IoController)
        - Memory-mapped I/O channels (e.g., CHANNEL_DSKY for DSKY display).
        - Integrated peripherals via IoPeriph trait.
    - Special Register
        - Implemented read only special registers
        - For sensors (OPTICAL_X, INERTIAL_Z) and control displays.
    - Edit Registers
        - Handles cyclic shifts (cycle_left, cycle_right) and bitwise operations.
    - Clocks (Timers)
        - Memory-mapped timers (TIMER1, TIMER4) for interrupts and cycle tracking.
- CPU: Executes AGC instructions and manages CPU state.
    - Components:
        - Cpu Struct:
            - Registers: ir (Instruction Register), idx_val (Index Value).
            - Flags: ec_flag (Extended Instruction), gint (Global Interrupt Enable).
            - Memory: mem (MemoryMap for RAM/ROM access).
        - Methods:
            - execute(): Dispatches instructions (e.g., CA, TC, MP).
            - step(): Processes instructions or unprogrammed sequences (interrupts).
        - Interrupt Handling: handle_interrupt() for timer/keypress events.
- Decoder: Converts raw 15-bit words into executable instructions
    - Output: Instructions struct with mnemonic (Mnemonic), opcode, and metadata
    - Functions:
        - decoder(): Main entry point for decoding.
        - decoder_simple(): Handles basic opcodes (e.g., CA, AD).
        - decoder_extended(): Parses complex instructions (e.g., READ, DIM).Output: Instructions struct with mnemonic (Mnemonic), opcode, and metadata.

5.2.4 Backend Testing

Item	Type	Location	Focus
Arithmetic Correctness	Unit Test	utils.rs (under #[cfg(test)])	Arithmetic correctness (15-bit overflow, sign extension).
CPU Behavior	CPU Integration Test	cpu.rs (cpu_tests module)	End-to-end CPU behavior (e.g., interrupt handling, restart sequences).
DSKY Display Logic	Peripheral Test	dsky.rs (dsky_unittests module)	DSKY display logic (digit mapping, channel updates).
Rust's Built-in Test Runner	-	-	Executes cargo test for modular tests.
Logging (log crate)	-	-	Traces execution flow during tests.

6. Challenges & Problems Solved

- **Integrating CesiumJS with React:** CesiumJS is a powerful library designed, however it is imperatively designed to directly manipulate a dedicated DOM element. React manages its own Virtual DOM and component lifecycle and is declarative. Mixing them can lead to conflicts where React re-renders might interfere with Cesium's rendering. As well, the Cesium.Viewer object is resource-intensive; repeatedly creating and destroying it with

component re-renders can significantly degrade performance and cause memory leaks. This was solved by treating Cesium Viewer as a persistent, externally managed resource within React's component lifecycle.

- UseRef is used to hold the Cesium.viewer instance, this ensures that the same viewer instance persists across component re-renders, preventing Cesium from re-initialising (which is costly). The viewer is initialised once within the useEffect hook. The hook's dependency array ensures this initialisation logic runs only when the component is mounted, not on every render. It waits for the container DOM element (cesiumContainerRef.current) to be available.
- Implementing the cleanup function returned by the main useEffect hook is very important. The function called viewerRef current.destroy() when the component unmounts (e.g. when switching scenes). This releases Cesium's resources (WebGL contexts, event listeners, memory), preventing memory leaks.
- Interactions with the Cesium API, such as loading CZML, adding entities, manipulating the clock, attaching event listeners like onTick, are performed imperatively inside useEffect hooks, accessing the viewer via viewerRef.current. React state is mainly used for UI elements outside of the Cesium environment (like timers and checklists), often updated based on the data from Cesium's onTick event.

- **Real-time communication:** Implementing a reliable WebSocket connection between the client and backend for broadcasting AGC status updates efficiently.

- **15-Bit Architecture Emulation:** The AGC's 15-bit word size posed compatibility issues with modern systems. By masking values to 0x7FFF and using utilities like adjust_overflow and extend_sign_bits, we emulated 15-bit arithmetic, ensuring correct truncation/sign-extension while preserving AGC-specific overflow behavior.

- **Concurrency & Peripheral Coordination:** Peripherals (DSKY, Downrupt) required parallel execution without race conditions. Thread-safe crossbeam_channel channels enabled communication (e.g., keypresses via Sender<u16>), while background threads managed display flashing, network I/O, and telemetry, decoupling CPU execution from peripheral updates.

- **ROM Loading & Memory Mapping:** Loading AGC binaries into Rust's memory model was solved via the include_transmute! macro, which safely casts ROM files (e.g., RETREAD50.bin) into static arrays. Bank switching (fixed_bank, erasable_bank) in AgcRegs dynamically mapped ROM/RAM segments, replicating the AGC's memory hierarchy.

- **Testing & Accuracy**: Validating emulator involved unit tests (e.g., overflow handling in utils.rs) and logging (log crate) to trace memory/instruction behavior. This ensured alignment with historical AGC specs while debugging edge cases like interrupt timing and register updates.

- **AGC emulator interfacing:** When yaDSKY connects and starts running a program (i.e. Saturn V launch program), it begins writing the state of its internal registers (simulated core rope memory locations displayed on the DSKY) to the output.txt file. The detectAgcProgram leverages the fact that different programs initialise registers in characteristic ways shortly after the connection message appears. The code looks for the line "R3D5: 1" appearing

relatively early in the sequence after connection. Parsing the text output required careful pattern matching. Using fs.watch on yaDSKY's output.txt is effective, however it is not as robust as a direct API or network protocol.

- **Trajectory Accuracy vs. Performance.** Balancing the complexity of the physics simulation in Python (for accuracy) with the need for smooth playback in CesiumJS. Pre-calculating trajectories into CZML was the best approach for the project requirements.

- **Generating and loading .CZML files:** Making sure the CZML files were formatted correctly, included necessary metadata (clock settings, interpolation) and referenced 3D models correctly. Debugging CZML issues often involved inspecting the file structure and CesiumJS loading errors.

- **Time synchronisation:** Aligning the Cesium clock, mission event timings (from research), audio cues, and UI timers (UTC, T-minus), required careful handling of Cesium.JulianDate and JavaScript Date objects.

- **Audio synchronisation:** Triggering audio cues (engine sounds, radio) based on Cesium clock's progression required adding listeners to viewer.clock.onTick. Managing audio playback state (start/stop, looping) using Howler.js.

- **3D Model Orientation:** Correctly orienting the Saturn V and LM models along their trajectory paths, especially during launch (vertical ascent) and maneuvers (pitch, yaw and roll), required using VelocityOrientationProperty and sometimes applying fixed rotations to the model itself.

- **Performance optimisation:** Using a traditional 3D library like Three.js for rendering planet-sized bodies like the Earth or Moon has is a performance challenge, especially when running on the browser. Rendering a planet at a high-quality requires an enormous amount of data: high-resolution imagery/maps, detailed elevation maps, 3D models for models or building structures. Loading all of this into memory at once is often impossible on typical client hardware. Three.js or React Three Fiber would load a single, high-resolution texture and model for the entire planet, leading to excessive memory usage and slow rendering, especially when viewed from a distance where most detail is unnecessary. As well, drawing millions or billions of polygons for a detailed planetary surface every frame is computationally expensive. CesiumJS is specifically designed for visualising large-scale geospatial data. It natively supports streaming and rendering massive datasets using standards like 3D tiles.

7. Work Done

The emulation of AGC and physics simulation environments presented technical challenges due to the exploratory and research-intensive nature of the project. Consequently, a large part of the early-to-mid project timeline was dedicated to spikes to address these inherent exploratory requirements. These were short-term, focused technical experiments designed to de-risk important architectural or implementation decisions before committing to a final approach. Collaboration during these investigation sprints, often facilitated using tools like Live Share, was essential for validating assumptions and exploring feasibility. The areas addressed by these spikes included implementation testing/ investigations of the following:

- Testing and comparison of different 3D rendering libraries, such as Three.js and React Three Fiber.
- Rust-to-WebAssembly Compilation: Investigating the feasibility of compiling the AGC emulation to WebAssembly (WASM) for execution inside the frontend, including evaluating distributing as an npm package.
- Evaluating Javascript 3D physics libraries (Ammo.js, Rapier, Cannon.js) against manually simulating physics.
- Assessing the suitability of frontend build tools specifically Vite (with React) vs. Next.js.
- Experiments with cycle timing models to match VirtualAGC documentation.
- Instruction Set Validation: Short-lived local branches tested variations in 15-bit arithmetic logic
- Evaluating Rust concurrency models for DSKY telemetry and interrupt handling.

These spikes were critical in allowing us to validate architectural choices and determine feasibility before committing to a production-ready implementation.

While this approach limited our ability to commit regularly to the shared GitLab repo (as many spike branches were local or short-lived), this was a deliberate choice. It helped to avoid messy or misleading version history and made sure we had a clean, maintainable final product.

We recognise the value of CI/CD and tracing version history through Git, and future iterations of this project would benefit from separating spike work into dedicated branches with clearer documentation and commits.

BY OM YASHWANT DIGHE (21100292)

1. AGC Core Emulator Implementation (Rust)

- CPU Architecture:
    - Implemented the full 15-bit AGC instruction set, including:
        - Arithmetic: AD (Add), MP (Multiply), SU (Subtract), DV (Divide) with 15-bit overflow/underflow handling.
        - Control Flow: TC (Transfer Control), CCS (Conditional Branch), BZF (Branch Zero), INDEX for address modification.
        - I/O: READ, WRITE, WOR, RXOR (XOR) for peripheral communication.

        - Interrupts: Priority-based handling for RUPT_DOWNRUPT, RUPT_TIME4, and DSKY keypresses.
    - Designed cycle-accurate timing with a real-time execution loop.
- Memory Subsystem:
    - Modeled 2 kB RAM (erasable memory) with parity checking and 72 kB ROM (core rope) using memory-mapped binaries (e.g., LUMINARY131.bin).
    - Implemented bank switching logic for fixed/erasable memory segments.
- Peripheral Integration:
    - DSKY Protocol: Modified the open-source yaDSKY implementation to log the R3 register state to output.txt via dedicated I/O channels (0o10).
    - Downrupt Telemetry: Built a TCP listener for telemetry data (channels 0o34/35) using Rust's std::net module.

2. yaDSKY Register Logging

- R3 Register Capture:
    - Extended the DSKY's packet-handling logic to parse and write R3 digit values (e.g., R3D5: 1) to output.txt for frontend integration.
    - Implemented file I/O synchronization to ensure real-time updates without data races.

3. Testing & Validation

- Unit Tests:
    - Validated 15-bit arithmetic correctness (e.g., sign extension, overflow).
    - Tested memory banking logic and interrupt prioritization.

4. Tooling & CLI

- ROM Loader: Built a CLI (using clap) to load historical binaries (e.g., --retread50, --luminary99).
- Logging: Integrated log crate for debugging CPU states, memory access, and peripheral I/O.

5. Concurrency & Performance

- Thread-Safe Channels: Used crossbeam for low-latency communication between CPU, DSKY, and telemetry modules.
- Async Runtime: Leveraged non-blocking I/O operations (DSKY socket handling).

BY KENNETH JOHN RAS (21787441)

1. Backend (Node.js)

- **Backend (server.js)**
    - Set up Express server with WebSocket for real-time communication.
    - Serve static frontend files.
    - Implement file watching on output.txt.
    - Detect AGC connection status from output.

    - Detect loaded AGC program type (saturn_v, moon_landing).
    - Detect specific Apollo 11 launch sequence.
    - Parse R3D register data.
    - Broadcast all detected status and data to connected clients via WebSocket.
    - Add graceful shutdown to clear output file.

2. Orbital Calculations (Python)

- saturn_v_orbit_orientation.py:
    - Simulated Saturn V launch trajectory including stages, burns, coast.
    - Modeled physics: thrust, mass change, gravity, drag etc..
    - Implemented gravity turn pitch program.
    - Generated trajectory plots.
    - Exported trajectory, orientation, timing data to .czml for Cesium.
- moon_orbit.py:
    - Simulated Apollo 11 CSM orbit and LM descent around the Moon.
    - Modeled lunar gravity, descent thrust, mass change etc.
    - Implemented descent pitch and throttle control.
    - Simulated descent to lunar surface.
    - Detected landing event.
    - Generated trajectory plots.
    - Exported trajectories to .czml for Cesium.

3. Frontend (React/TypeScript/Cesium)

- Core Application (App.jsx, etc.):
    - Manage app state and scene transitions (Home, Earth, Moon).
    - Implement code-splitting for scenes.
    - Display loading screen during transitions.
    - Establish global WebSocket connection for AGC status.
    - Render global AGC connection indicator.
- UI Components (HomeScreen.jsx, MissionChecklist.tsx, etc.):
    - Provide scene selection options.
    - Check AGC connection/program type before starting scenes.
    - Implement reusable mission checklist based on AGC status.
    - Play background music and sounds.
- Cesium Scenes (EarthScene.tsx, MoonScene.tsx):
    - Initialise and configure Cesium viewer for specific celestial bodies (Earth, Moon).
    - Load and display trajectory data from .czml files.
    - Manage Cesium clock animation synchronized with mission timeline and AGC.
    - Handle dynamic entity orientation and tracking.
    - Play synchronized and spatial audio effects.
    - Listen for WebSocket messages (e.g., AGC liftoff) to trigger events.

    - Update UI timers and status text.

8. Results

- Successfully created a web-based visualisation platform capable of displaying pre-calculated trajectories for the Saturn V launch and LM descent.
- Replicated the Apollo DSKY display/keyboard via TCP sockets.
- Implemented ~20 key AGC instructions (e.g., CA, AD, TC, CCS) with cycle-accurate timing (~11.7 µs per cycle) with supported arithmetic, control flow, I/O, and interrupt handling
- Integrated a 3D CesiumJS view with React components for UI elements like checklists and timers.
- CLI Integration: Supported multiple ROM binaries (e.g., LUMINARY99, COMANCHE55 via CLI flags).
- Established real-time communication via WebSockets to reflect the connection status and program type of an external AGC emulator.
- Implemented mission checklists that gate simulation progress based on the AGC status, adding an interactive element.
- Synchronised basic audio cues and particle effects with the simulation time.
- Generated reasonably accurate CZML trajectory files based on simplified physics models and historical Apollo 11 data.

9. Future Work

- **Live AGC Data Integration:** Replace file watching with direct communication to the AGC emulator (e.g., via a custom network interface for yaAGC or a different emulator) to get more detailed, real-time data (velocity, altitude, attitude, DSKY display values).
- **Dynamic Trajectory Adjustment:** Use live AGC data to influence the Cesium simulation dynamically, rather than just playing back pre-calculated CZML. This would allow for simulating off-nominal scenarios or user interaction via a virtual DSKY.
- **Enhanced Visuals:** Improve particle effects, add stage separation visuals, implement more realistic lighting and atmospheric scattering, and potentially add more detailed 3D models or environments.
- **More Mission Phases:** Simulate additional phases like Translunar Injection (TLI), Lunar Orbit Insertion (LOI) and rendezvous.
- **Interactive DSKY:** Create a virtual DSKY interface in the React app that allows users to interact with the AGC emulator.
- **Improved Audio:** Add more comprehensive and accurately timed audio clips from mission control and astronauts, potentially with spatial audio effects.

- **Error Simulation:** Introduce ways to simulate common Apollo-era issues (e.g., 1201/1202 alarms) based on emulator state.
- **Code Refinement:** Improve state management (potentially using Zustand or Redux), enhance error handling, add unit/integration tests, and further optimise performance.
- **User Experience:** Refine UI elements, add more explanatory text or annotations, and improve camera controls.
- **Deployment:** Containerised the application (frontend, backend) for easier deployment.
- **Testing & ValidationUnit Tests**:
    - Add tests for RUPT handling, timer resets, and RESTART light logic.
    - Validate edge cases (e.g., nested interrupts, overflow during AD/MP).
- **Check RUPT Requests**: Verify and prioritize pending interrupts (e.g., timers, keypresses).
- **Unimplemented Instructions**
    - Missing Instructions:
        - Divide (DV): Requires 32-bit arithmetic and overflow handling.
        - Multiply (MP): Implement using AGC's 15-bit×15-bit→30-bit logic.
        - Extended Instructions: ED (Extended Double), DTCF (Double Transfer Control).
    - Partial Implementations:
        - SU (Subtract): Verify sign handling and overflow flags.
        - INHINT/RELINT: Ensure interrupts are fully inhibited/released.

10. References

Virtual AGC Project (Primary) - https://www.ibiblio.org/apollo/index.html#gsc.tab=0

yaDSKY Open Source Project (Primary) - https://www.ibiblio.org/apollo/yaDSKY.html#gsc.tab=0

Apollo 11 Source Code (Primary) - https://github.com/chrislgarry/Apollo-11

Clap for Rust CLI - https://github.com/clap-rs/clap

Oct2Bin Tool from Virtual AGC project - https://virtualagc.github.io/virtualagc/download-old.html#gsc.tab=0

Binaries for Apollo Modules - https://github.com/virtualagc/virtualagc/tree/master

CesiumJS Sandcastle https://sandcastle.cesium.com/

Apollo Flight Journal
https://www.nasa.gov/history/afj/index.html

Post Flight Trajectory
https://www.nasa.gov/wp-content/uploads/static/history/afj/ap10fj/pdf/as-505-postflight-trajectory.pdf

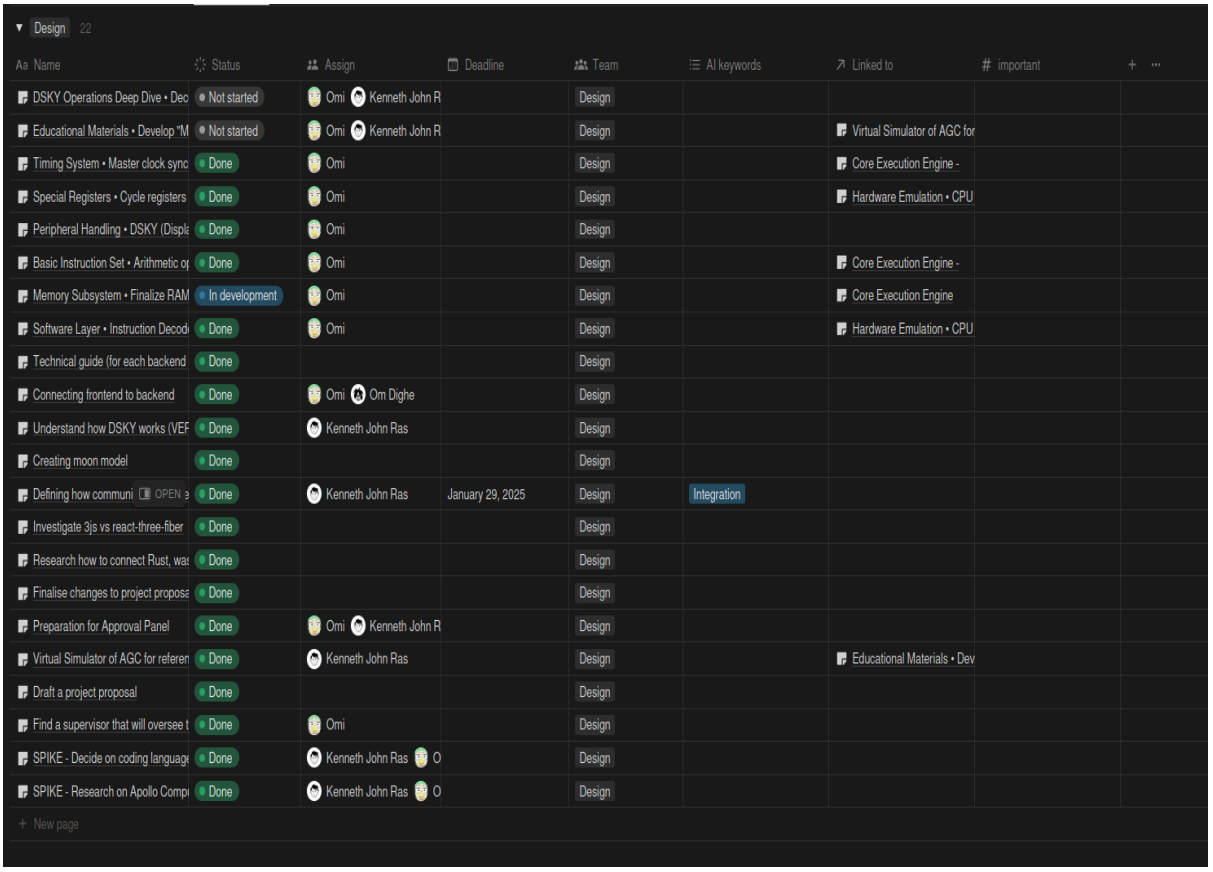
Final Descent Trajectory
https://www.researchgate.net/publication/362871268_Reconstruction_of_the_Apollo_11_Moon_Landing_Final_Descent_Trajectory

Physics and Earth Parameters
https://hypertextbook.com/facts/2002/JasonAtkins.shtml
https://nssdc.gsfc.nasa.gov/planetary/factsheet/earthfact.html
https://en.wikipedia.org/wiki/Density_of_air

Rocket geometry
https://en.wikipedia.org/wiki/Saturn_V
https://space.stackexchange.com/questions/12649/how-can-i-estimate-the-coefficient-of-drag-on-a-saturn-v-rocket-a-simulator-or

Python simulation scaffold
https://github.com/GEEGABYTE1/RocketTrajectorySim/blob/master/rocket_script_two.py

11. Appendix

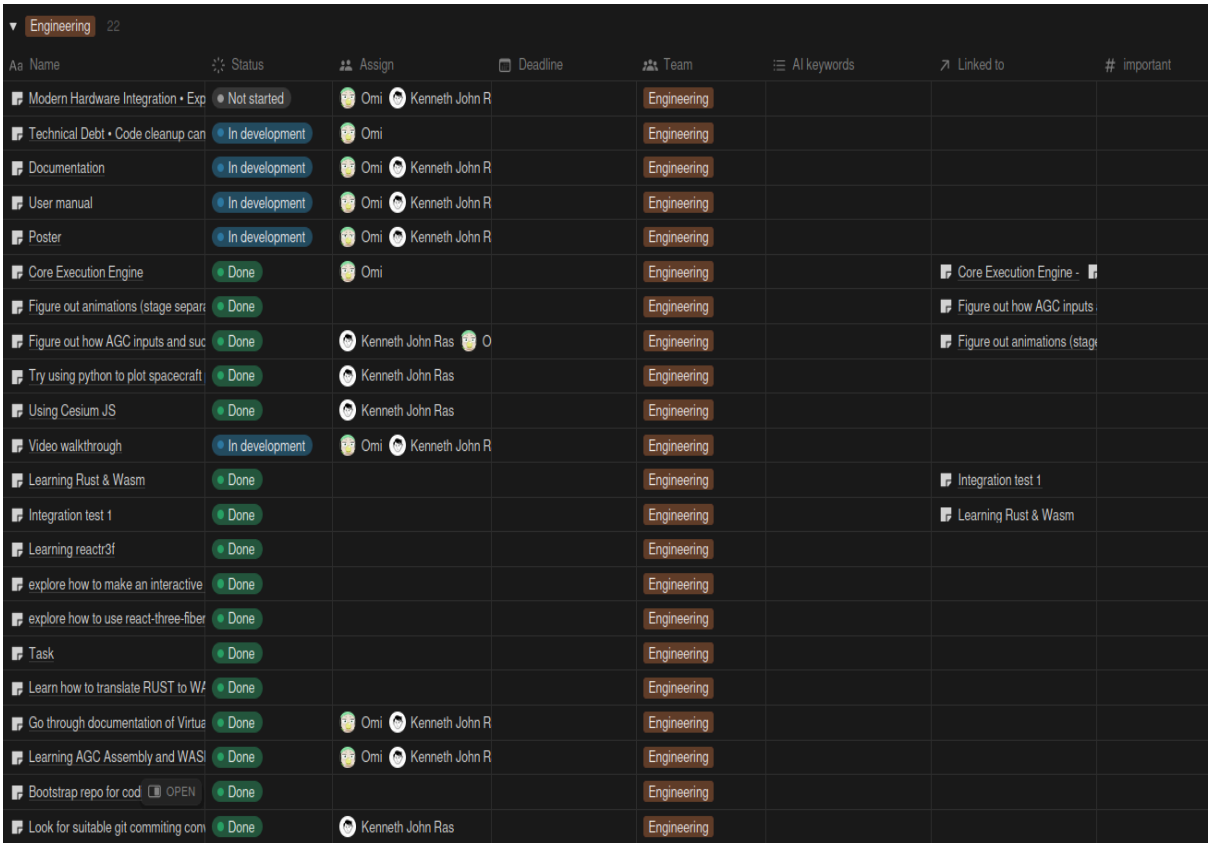


▼ Design 22

Aa Name	Status	Assign	Deadline	Team	AI keywords	Linked to	# important
DSKY Operations Deep Dive • Dec	Not started	Omi, Kenneth John R		Design			
Educational Materials • Develop "M	Not started	Omi, Kenneth John R		Design		Virtual Simulator of AGC for	
Timing System • Master clock sync	Done	Omi		Design		Core Execution Engine -	
Special Registers • Cycle registers	Done	Omi		Design		Hardware Emulation • CPU	
Peripheral Handling • DSKY (Displ	Done	Omi		Design			
Basic Instruction Set • Arithmetic or	Done	Omi		Design		Core Execution Engine -	
Memory Subsystem • Finalize RAM	In development	Omi		Design		Core Execution Engine	
Software Layer • Instruction Decod	Done	Omi		Design		Hardware Emulation • CPU	
Technical guide (for each backend	Done			Design			
Connecting frontend to backend	Done	Omi, Om Dighe		Design			
Understand how DSKY works (VEF	Done	Kenneth John Ras		Design			
Creating moon model	Done			Design			
Defining how communi OPEN	Done	Kenneth John Ras	January 29, 2025	Design	Integration		
Investigate 3js vs react-three-fiber	Done			Design			
Research how to connect Rust, was	Done			Design			
Finalise changes to project proposa	Done			Design			
Preparation for Approval Panel	Done	Omi, Kenneth John R		Design			
Virtual Simulator of AGC for referen	Done	Kenneth John Ras		Design		Educational Materials • Dev	
Draft a project proposal	Done			Design			
Find a supervisor that will oversee t	Done	Omi		Design			
SPIKE - Decide on coding language	Done	Kenneth John Ras, O		Design			
SPIKE - Research on Apollo Compu	Done	Kenneth John Ras, O		Design			

+ New page

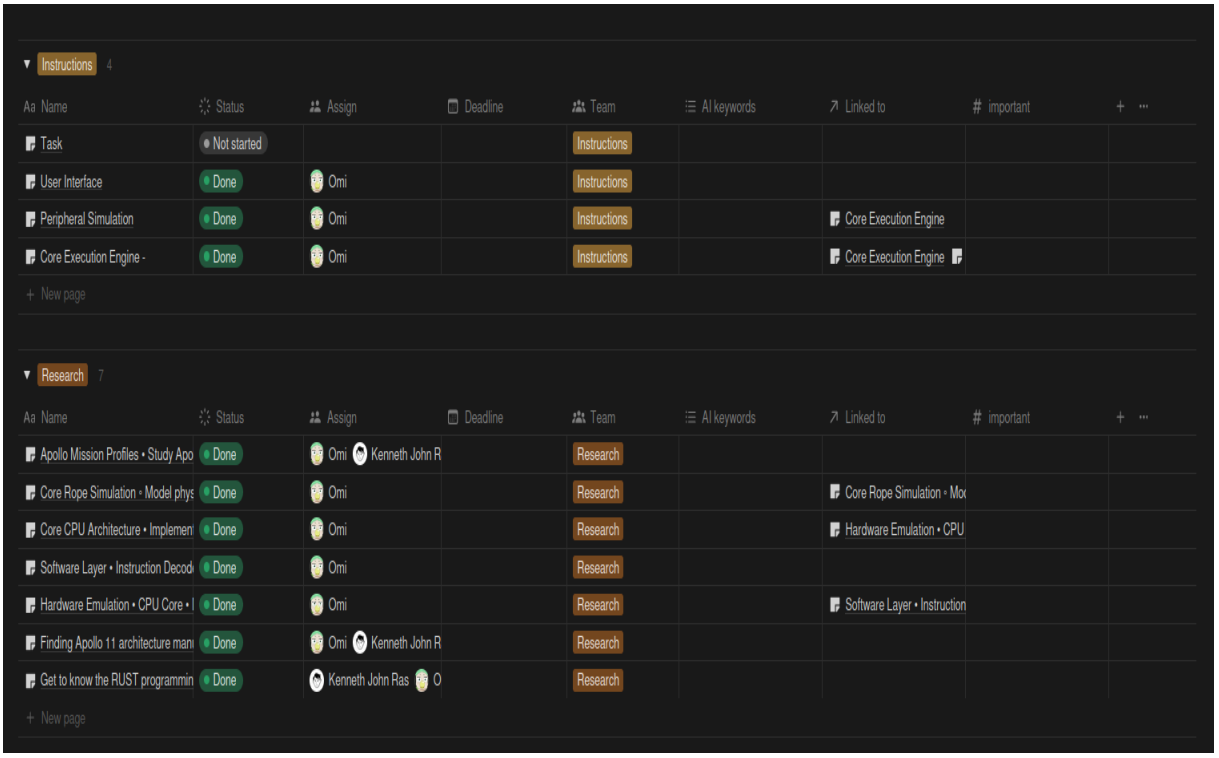
Fig. 10 Kanban board (Design)



Engineering 22

Name	Status	Assign	Deadline	Team	AI keywords	Linked to	important
Modern Hardware Integration • Exp	Not started	Omi Kenneth John R		Engineering			
Technical Debt • Code cleanup can	In development	Omi		Engineering			
Documentation	In development	Omi Kenneth John R		Engineering			
User manual	In development	Omi Kenneth John R		Engineering			
Poster	In development	Omi Kenneth John R		Engineering			
Core Execution Engine	Done	Omi		Engineering		Core Execution Engine -	
Figure out animations (stage separ	Done			Engineering		Figure out how AGC inputs	
Figure out how AGC inputs and suc	Done	Kenneth John Ras O		Engineering		Figure out animations (stag	
Try using python to plot spacecraft	Done	Kenneth John Ras		Engineering			
Using Cesium JS	Done	Kenneth John Ras		Engineering			
Video walkthrough	In development	Omi Kenneth John R		Engineering			
Learning Rust & Wasm	Done			Engineering		Integration test 1	
Integration test 1	Done			Engineering		Learning Rust & Wasm	
Learning reactr3f	Done			Engineering			
explore how to make an interactive	Done			Engineering			
explore how to use react-three-fiber	Done			Engineering			
Task	Done			Engineering			
Learn how to translate RUST to WA	Done			Engineering			
Go through documentation of Virtua	Done	Omi Kenneth John R		Engineering			
Learning AGC Assembly and WASI	Done	Omi Kenneth John R		Engineering			
Bootstrap repo for cod OPEN	Done			Engineering			
Look for suitable git commiting con	Done	Kenneth John Ras		Engineering			

Fig. 11 Kanban board (Engineering)



Instructions 4

Name	Status	Assign	Deadline	Team	AI keywords	Linked to	important
Task	Not started			Instructions			
User Interface	Done	Omi		Instructions			
Peripheral Simulation	Done	Omi		Instructions		Core Execution Engine	
Core Execution Engine -	Done	Omi		Instructions		Core Execution Engine	

+ New page

Research 7

Name	Status	Assign	Deadline	Team	AI keywords	Linked to	important
Apollo Mission Profiles • Study Apo	Done	Omi Kenneth John R		Research			
Core Rope Simulation - Model phys	Done	Omi		Research		Core Rope Simulation - Mo	
Core CPU Architecture • Implemen	Done	Omi		Research		Hardware Emulation • CPU	
Software Layer • Instruction Decod	Done	Omi		Research			
Hardware Emulation • CPU Core • I	Done	Omi		Research		Software Layer • Instruction	
Finding Apollo 11 architecture mani	Done	Omi Kenneth John R		Research			
Get to know the RUST programmin	Done	Kenneth John Ras O		Research			

+ New page

Fig. 12 Kanban board (Instructions & Research)