The Art of Digital Replication: Unveiling the Mechanics of CPU Emulation

I. Introduction to CPU Emulation

Central Processing Unit (CPU) emulation stands as a cornerstone in digital preservation, software development, and computer science education. It allows contemporary systems to mimic the behavior of often-obsolete or otherwise inaccessible hardware, breathing life into vintage software and providing invaluable platforms for research and development.

A. What is a CPU Emulator?

A CPU emulator is a sophisticated software program meticulously designed to replicate the complete functionality and behavioral characteristics of a specific hardware CPU, known as the "target" CPU, on a different computing system, referred to as the "host" system. The core mechanism involves the emulator interpreting or translating the machine code instructions, which are the native language of the target CPU, into a series of operations that the host CPU can understand and execute.[1] It is paramount to recognize that an emulator's role extends beyond merely running software; its fundamental task is to recreate the entire hardware environment that the target software anticipates.[1]

This recreation is not limited to the CPU alone. While the CPU is the brain, an emulator must often construct a comprehensive virtual hardware platform. This frequently encompasses simulating the target system's memory architecture, input/output (I/O) devices, and in many cases, specialized co-processors or graphics processing units (GPUs).[2] The necessity for such an extensive simulation arises because software, particularly operating systems or firmware, expects a complete ecosystem to interact with.[1] If these components are absent or incorrectly emulated, the target software will fail to operate as intended, or at all. Thus, CPU emulation is often a subset of system emulation, where the goal is to provide a holistic, virtualized replica of the original machine.

B. Purpose and Applications of Emulators

The motivations behind developing and utilizing CPU emulators are diverse and impactful, spanning various fields:

- **Digital Preservation:** Emulators are instrumental in preserving our digital

heritage. They enable software designed for obsolete hardware, such as classic video games, vintage operating systems, and legacy business applications, to be run on modern computers, safeguarding them from being lost to time as original hardware degrades or becomes unavailable.

- **Software Development and Testing:** Emulation provides a flexible and controlled environment for software development. Programmers can create and test software for hardware platforms that are not yet physically available in large quantities, are expensive, or are difficult to access for debugging.[1] This accelerates development cycles and allows for rigorous testing without needing to constantly transfer programs to the target hardware, and without the potential side effects that a hardware debugger might introduce.[1]
- **Cross-Platform Compatibility:** Emulators bridge the gap between different hardware architectures and operating systems, allowing software written for one specific platform to run on another. This is particularly useful for accessing platform-exclusive applications or for developers aiming to reach a wider audience without porting their software natively.
- **Security Research:** In the realm of cybersecurity, emulators offer a sandboxed environment for analyzing potentially malicious software (malware). Researchers can execute and observe malware behavior within the confines of an emulator without risking harm to the host system, allowing for detailed study of its mechanisms and exploits.
- **Education and Research:** Emulators are powerful educational tools for understanding computer architecture, operating system design, and low-level programming concepts. Architectures like the LC-3, for instance, are explicitly designed for educational purposes and are primarily interacted with via simulators or emulators.[3]

The very endeavor of constructing an emulator, especially for systems with limited or incomplete documentation, transforms into an intensive exercise in reverse engineering.[2] This process compels developers to meticulously dissect and comprehend the intricate workings of the target hardware, often leading to a far deeper understanding than could be achieved through standard usage or documentation study alone. This investigative aspect, where developers effectively become digital archaeologists, unearthing undocumented features or idiosyncratic behaviors [1], is a profound, though often understated, benefit of emulator development.

II. Core Principles of CPU Emulation

At its heart, CPU emulation is about creating a faithful software representation of a hardware entity. This involves understanding several core principles that govern how this digital mimicry is achieved and the trade-offs involved.

A. The Abstraction Layer: Mimicking Hardware in Software

Emulation works by establishing an abstraction layer. The emulator software positions itself as an intermediary between the target program's machine code and the host system's hardware. It systematically translates the instruction set architecture (ISA) and the behavioral nuances of the target CPU into a sequence of actions that the host system can perform. This complex translation process necessitates the software simulation of all fundamental CPU components, including its registers, mechanisms for memory access, the operations of the Arithmetic Logic Unit (ALU), and the logic governing control flow.[1] Essentially, the emulator creates a virtual CPU, operating entirely within the software domain of the host machine. This is often visualized as an infinite loop that fetches, decodes, and executes the target binary's instructions one by one.[4]

A practical consequence of this software-based simulation is the computational overhead incurred. Emulating hardware in software is inherently less efficient than running code natively on the hardware it was designed for. As a general guideline, it is often estimated that the host system requires significantly more processing power—roughly 8 to 10 times the "horsepower"—than the original target hardware possessed to achieve satisfactory emulation performance.[2] This disparity arises from the layers of interpretation or dynamic translation involved, and the need for the host CPU to perform many operations to simulate even a single, simple hardware function of the target. This "horsepower rule" is a critical factor influencing the feasibility of emulating certain systems, particularly newer and more powerful ones, and explains why older, less powerful systems become progressively easier to emulate as host hardware capabilities advance.

B. Key Considerations: Accuracy, Compatibility, and Performance

The development and assessment of CPU emulators revolve around three pivotal, interconnected factors:

- **Accuracy:** This refers to the fidelity with which an emulator replicates the precise behavior of the target hardware. In its most stringent form, accuracy can extend to the level of individual clock cycles, undocumented hardware features, unpredictable analog behaviors, and even known implementation bugs in the

original hardware.[1] For certain software, especially classic games or demoscene productions that exploit subtle hardware quirks for specific effects or functionality, this high degree of accuracy is not merely desirable but essential for correct operation.[6]

- **Compatibility:** This measures the breadth of software titles or applications designed for the target system that can run correctly on the emulator. While closely linked to accuracy, compatibility is not synonymous with it. Emulator developers sometimes implement "hacks" or game-specific workarounds to make popular titles run, even if these fixes deviate from a perfectly accurate hardware simulation.[6] Such measures can improve compatibility metrics but may compromise the purity of the emulation.
- **Performance:** This relates to the speed at which the emulated software executes on the host system. Users typically expect emulated programs to run at a speed comparable to, if not exceeding, the original hardware. However, a fundamental tension often exists between accuracy and performance. Emulation techniques that achieve higher accuracy, such as cycle-level simulation, tend to be more computationally intensive and can therefore result in slower performance.[5]

The concept of "accuracy" itself is not static or absolute. It often represents an evolving target, as ongoing research and reverse engineering efforts uncover more nuanced details about the original hardware's behavior. The required level of accuracy can also be context-dependent, varying based on the specific software being emulated. Some programs may function adequately with a less precise emulation model, while others, like the "Speedy Gonzales: Los Gatos Bandidos" example which soft-locks on less accurate SNES emulators due to an un-emulated hardware edge case, demand meticulous replication of subtle hardware behaviors.[6] This implies that emulator development is frequently an iterative journey, with accuracy progressively refined over time as the collective understanding of the target platform deepens.

C. High-Level Emulation (HLE) vs. Low-Level Emulation (LLE)

Emulator developers employ different strategies to replicate target systems, broadly categorized as Low-Level Emulation (LLE) and High-Level Emulation (HLE).

- **Low-Level Emulation (LLE):** This approach endeavors to simulate the target hardware components at an extremely detailed, often cycle-accurate, level. LLE directly mimics the hardware's internal logic, timing, and behavior.[2] For instance, an LLE approach to graphics might involve a software renderer that aims to produce output that is bit-for-bit identical to the original console's graphics chip.

This method is prevalent for simpler, older systems where such detail is feasible and often necessary for compatibility, or when the primary goal is the most authentic replication possible. The logic of the simulated CPU in LLE can often be directly translated into software algorithms, creating a software re-implementation that closely mirrors the original hardware design.[1]

- **High-Level Emulation (HLE):** In contrast, HLE simulates the *behavior* or *functionality* of system components or operating system (OS) calls at a much higher level of abstraction.[2] Instead of replicating every transistor or clock cycle of a graphics chip, an HLE implementation might intercept graphics API calls from the target game and translate them into equivalent calls to the host system's graphics API (e.g., DirectX or OpenGL), thereby leveraging the host's GPU for rendering. Similarly, HLE can replicate OS functionalities by translating the target system's OS calls into corresponding calls on the host OS.[1] This approach is frequently adopted for more complex components found in modern consoles or to achieve better performance, especially when LLE would be too computationally demanding. Some Game Boy Advance emulators, for example, use HLE to simulate BIOS subroutines, eliminating the need for an actual BIOS file, albeit sometimes with a minor trade-off in emulation accuracy.[7]
- **Hybrid Emulation:** Many modern emulators, particularly for complex systems like the PlayStation 3 (RPCS3) or Xbox 360 (Xenia), adopt a hybrid approach. This strategy combines LLE for critical core components where accuracy is paramount (such as certain OS services or CPU core functions) with HLE for more demanding or complex subsystems (like the GPU or audio processing units). The goal is to strike a pragmatic balance, achieving good compatibility and performance without the prohibitive development time or computational cost of full LLE for every component.

The choice between HLE and LLE involves significant trade-offs. HLE can yield substantial performance improvements and offers a practical advantage by potentially bypassing the legal complexities associated with distributing copyrighted BIOS or firmware files, as it simulates their functions rather than requiring the original files. However, this performance and convenience often come at the cost of authenticity. HLE may not perfectly replicate every nuance of the original system, leading to inaccuracies or compatibility problems, especially with software that utilizes hardware in unconventional or undocumented ways. HLE relies more heavily on the target applications adhering to standardized high-level system interfaces; if an application bypasses these or uses them in an unexpected manner, it may not function correctly

under HLE, even if similar applications work flawlessly. This can necessitate numerous game-specific tweaks and adjustments.

The following table provides a comparative overview of these emulation strategies:

Feature	Low-Level Emulation (LLE)	High-Level Emulation (HLE)	Hybrid Emulation
Primary Goal	Hardware component mimicry	Behavioral/functional mimicry of system calls/APIs	Balance of hardware mimicry for core parts & behavioral for complex parts
Level of Abstraction	Very low (close to hardware logic)	High (abstracts away hardware details)	Mixed; LLE for some components, HLE for others
Typical Accuracy	High to very high, potentially cycle-accurate	Variable; can be lower, focuses on functional equivalence	Aims for high accuracy in LLE parts, functional accuracy in HLE parts
Performance	Generally slower due to detailed simulation	Generally faster, leverages host capabilities	Performance varies; aims to be faster than pure LLE for complex systems
Development Complexity	Often very high, requires deep hardware knowledge	Can be lower for some parts if host APIs are suitable; complex logic	High overall complexity due to managing two different approaches
Reliance on Target System Internals	Deep understanding of hardware specifics required	More reliance on documented APIs, OS calls, or observed behavior	Mixed reliance based on component
BIOS/Firmware	Often requires	Often can bypass	May or may not

Requirement	original BIOS/firmware files	BIOS/firmware by re-implementing its functions	require BIOS depending on which parts are LLE vs. HLE
Example Components Emulated	CPU core, specific hardware timers, memory access timing	OS system calls, Graphics API calls, audio API calls	CPU core (LLE), Graphics API translation (HLE)
Pros	High authenticity, good for undocumented features/quirks	Better performance, can avoid BIOS issues, easier for complex APIs	Good balance of accuracy and performance for modern systems
Cons	Computationally expensive, complex to develop	Lower authenticity, potential compatibility issues with edge cases	Still very complex to develop and debug, potential integration challenges
Use Cases	Classic computers/consoles, software depending on precise timing	Modern consoles (GPU, OS), performance-critical components	Emulation of 6th generation consoles and newer (e.g., PS2, PS3, Xbox, Wii U)

Over time, as host computer processing power has increased exponentially, LLE has become feasible for older systems that might have initially required HLE techniques in the early days of emulation. Conversely, HLE has found new applications on less powerful platforms like smartphones and handheld devices, where its performance benefits are crucial for achieving playable frame rates.

III. The Emulator's Main Loop: The Fetch-Decode-Execute Cycle

The operational core of any CPU, and consequently any CPU emulator, is the fetch-decode-execute cycle, also known as the instruction cycle. This fundamental loop dictates how a CPU processes instructions. An emulator meticulously replicates this cycle in software.[4] As aptly described, "An emulator is, in simple words, an infinite loop to execute a...binary one by one unless something wrong happens or a user stops an emulator explicitly".[4]

A. Overview of the Instruction Cycle

The cycle comprises three primary stages, continuously repeated:

1. **Fetch:** Retrieve the next instruction from memory.
2. **Decode:** Interpret the instruction to determine the operation and its operands.
3. **Execute:** Perform the specified action.

This loop forms the heartbeat of the emulator, driving the step-by-step execution of the target program's code.

B. Fetching Instructions from Emulated Memory

The cycle commences with the **fetch** stage. The emulator reads the next machine code instruction from its simulated memory. This memory is typically an array in the host program. The address from which to fetch is dictated by a special-purpose emulated register called the Program Counter (PC).[4] The number of bytes read depends on the instruction size defined by the target architecture (e.g., two bytes for CHIP-8 instructions, 16 bits or two bytes for LC-3 instructions).

A crucial aspect of the fetch stage is the subsequent update of the Program Counter. After the instruction is fetched, the PC is typically incremented to point to the memory address of the *next* instruction in sequence.[4] For example, if an instruction is 4 bytes long, the PC will be incremented by 4 (e.g., cpu.pc = cpu.pc + 4; [4]). This ensures that, barring any control flow changes, the emulator will process instructions sequentially. The PC effectively acts as the emulator's "cursor," indicating its current position within the executable code of the target program. Its accurate management is absolutely critical; any errors in updating the PC will lead to fetching incorrect instructions, inevitably causing the emulated program to crash or behave erratically. Control flow instructions, such as jumps, branches, and subroutine calls, achieve their purpose by directly modifying the value of the PC, thereby altering the otherwise sequential path of execution.[9] Endianness, the order in which multi-byte data is stored in memory, is also a vital consideration during the fetch stage, as instructions must be reassembled correctly if they span multiple bytes.[4]

C. Decoding Instructions: Opcodes and Operands

Once an instruction is fetched from emulated memory (it is now a binary value, for example, a 16-bit integer), it enters the **decode** stage. In this stage, the emulator dissects the binary instruction to understand what operation it represents and what data it will operate on. This involves two main steps:

1. **Identifying the Opcode:** The opcode (operation code) is the portion of the instruction that specifies the particular action to be performed (e.g., add, load data, jump to a new address). Opcodes are typically encoded in a specific field of bits within the instruction word. The emulator extracts these bits, often using bitwise masking and shifting operations, to determine the instruction type.[4] For example, in LC-3, the opcode is contained in the most significant 4 bits of the 16-bit instruction.
2. **Extracting Operands:** Operands are the data values or addresses that the instruction will use. These can be:
    - **Immediate values:** Constants embedded directly within the instruction bits.
    - **Register specifiers:** Numbers indicating which CPU registers are involved (e.g., source registers for an addition, or a destination register for a load).
    - **Memory addresses or offsets:** Values used to calculate the memory location to be read from or written to.[4]

The decoding process translates the raw binary instruction into a structured representation that the execution stage can easily use. For instance, a RISC-V emulator might decode an instruction by identifying the opcode, destination register, source registers, and any immediate values, taking advantage of consistent register specifier positions in RISC-V's instruction formats to simplify this process.[4] A CHIP-8 emulator would use bit masking to isolate the different nibbles of its 2-byte opcodes to determine the instruction and its arguments (like X, Y, N, NN, or NNN). In some conceptual models, like the Little Man Computer, the decoded instruction (opcode and operand) is placed into a Current Instruction Register (CIR) before being sent to the Control Unit.

D. Executing Instructions: Simulating CPU Actions

Following decoding, the **execute** stage is where the emulator performs the action specified by the instruction. This is typically implemented using a large switch-case statement (where each case handles a specific opcode) or a table of function pointers that dispatch to the appropriate instruction-handling routine.[13] The logic within each case or function carries out the emulated CPU's behavior:

- **Register Manipulation:** Loading values into emulated registers, copying data between registers, or performing arithmetic/logical operations on register contents.
- **Memory Access:** Reading data from the emulated memory array or writing data to it, using addresses calculated based on the instruction's operands and

addressing mode.
- **ALU Operations:** Simulating arithmetic (addition, subtraction) or logical (AND, OR, XOR, NOT, shifts) operations. The host CPU's native arithmetic and logical capabilities are typically used for this.
- **Status Flag Updates:** Modifying emulated status flags (e.g., Zero, Carry, Negative, Overflow) based on the outcome of the operation, according to the target CPU's specific rules.
- **Control Flow Changes:** Modifying the emulated Program Counter (PC) for instructions like jumps, branches, or subroutine calls, thereby altering the sequence of execution.

For example, a RISC-V emulator executing an add instruction would read values from two specified source registers, add them using the host CPU's addition capability, and write the result to the specified destination register.[4] The execution of each instruction can be viewed as a transformation of the emulated CPU's state—its registers, memory contents, and flags. The fundamental job of the emulator is to apply these state transformations accurately and sequentially, precisely as dictated by the target architecture's instruction set definition.[13] This perspective of the CPU as a finite state machine, where each instruction transitions it from one valid state to the next, is a powerful model for understanding emulator design and for debugging, as errors often manifest as deviations from the expected state.

IV. Simulating Core CPU Components in Software

To faithfully replicate a target CPU, an emulator must create software equivalents of its essential hardware components. This involves representing memory, registers, the ALU's functionality, status flags, and I/O mechanisms using the constructs available in the host programming language.

A. Representing Memory

The target CPU's main memory is almost universally simulated as a contiguous block of memory in the host program, typically an array of an appropriate data type. For instance, in C or C++, this might be uint8_t memory; or in JavaScript, new Uint8Array(MEMORY_SIZE);.[11] The size of this array directly corresponds to the addressable memory space of the target CPU (e.g., 4096 bytes for CHIP-8, or 216 words for LC-3).

Interactions with this emulated memory, such as reading an instruction or

loading/storing data, involve accessing elements of this array using an address derived from the emulated PC or calculated by an instruction. Memory-mapped I/O, a common technique where device control registers are accessed as if they were memory locations, is handled by designating specific address ranges within this array to trigger interactions with emulated peripheral devices. For example, writing to a particular memory address might update the emulated display, or reading from another might fetch the status of an emulated keyboard.

Two critical details in memory emulation are access granularity and endianness.

- **Granularity:** Emulators must respect whether the target CPU is byte-addressable (each byte has a unique address) or word-addressable (each multi-byte word has a unique address). The LC-3, for example, is word-addressable, where each of its 216 memory locations holds a 16-bit word.[17] This differs from many modern systems that are byte-addressable.
- **Endianness:** This refers to the order in which multi-byte values (like 16-bit or 32-bit integers or instructions) are stored in memory. Little-endian systems store the least significant byte at the lowest memory address, while big-endian systems store the most significant byte at the lowest address. For example, CHIP-8 opcodes are two bytes long and stored big-endian [9], whereas a RISC-V emulator example describes fetching 8-bit elements and combining them according to little-endian order to form a 32-bit instruction.[4] Incorrectly handling either granularity or endianness will lead to misinterpretation of both data and instructions, causing severe emulation errors.

B. Emulating CPU Registers

CPU registers, being small, fast storage locations within the CPU itself, are typically emulated as simple variables or a small array of variables of the appropriate data type in the host programming language.[11]

- **Program Counter (PC):** Emulated as an integer variable that holds the memory address of the next instruction to be fetched. For CHIP-8 or LC-3, this would typically be a 16-bit unsigned integer (e.g., uint16_t pc; in C++).[11]
- **Stack Pointer (SP):** Emulated as an integer variable that points to the current top of the stack within the emulated memory (or a dedicated stack array). For CHIP-8, this is an 8-bit value indexing into its 16-level stack.[11] For LC-3, register R6 often serves as the stack pointer.[10]
- **General-Purpose Registers (GPRs):** An array is a common and natural way to represent GPRs. For CHIP-8's 16 8-bit V-registers (V0-VF), this could be uint8_t

V;.[11] For LC-3's eight 16-bit registers (R0-R7), uint16_t R; would be appropriate.[11]
- **Instruction Register (IR):** A variable used to temporarily hold the currently fetched instruction during its decoding and execution phases. Its size matches the instruction width of the target CPU (e.g., uint16_t ir; for CHIP-8 or LC-3).[13]
- **Index Registers:** Special-purpose registers like CHIP-8's 16-bit I register are emulated as individual variables of the correct width (e.g., uint16_t I;).[11]
- **Status/Flags Register:** This register, which holds various condition codes or status bits, is often emulated as a single integer variable. Individual flags (like N, Z, P for LC-3's Condition Codes in the PSR, or CHIP-8's VF register when used as a flag) are then accessed or modified by manipulating specific bits within this variable.[10]

A crucial aspect of register emulation is using host language data types that precisely match the bit-width of the target CPU's registers (e.g., uint8_t for an 8-bit register, uint16_t for a 16-bit register).[11] This practice is vital for accurately simulating the behavior of the target hardware, particularly concerning arithmetic overflows, underflows, and bitwise operations. For example, if a standard C++ int (which might be 32 bits on the host) were used to emulate an 8-bit target register without explicit masking or truncation after each operation, the emulated register could hold values far exceeding its actual capacity. This would lead to behaviors that diverge significantly from the real hardware, introducing subtle and often hard-to-diagnose bugs. Adherence to correct register widths, as demonstrated in various emulator examples [11], helps prevent such issues.

C. Simulating the Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) is the component of a CPU responsible for performing arithmetic and logical operations. In an emulator, the ALU is not typically represented as a distinct software object or class. Instead, its *functionality* is implemented directly within the execution logic for each instruction that requires an ALU operation (e.g., ADD, SUB, AND, OR, XOR, NOT, SHIFT).[13] The emulator leverages the host CPU's native arithmetic and logical operators to achieve the same computational results on the emulated register values or immediate data.

- **Arithmetic Operations (e.g., ADD, SUB):** These are typically implemented using the host language's + and - operators. For example, to emulate an ADD R1, R2, R3 instruction (add contents of R2 and R3, store in R1), the emulator would perform emulated_R1 = emulated_R2 + emulated_R3;. Special attention must be paid to correctly setting status flags like Carry, Borrow (often part of Carry), and

Overflow, as the host CPU's flag behavior might not directly match the target's.[20] For instance, CHIP-8's 8XY4 (ADD Vx, Vy) instruction sets the VF register to 1 if there's a carry (unsigned overflow), and 0 otherwise.[9] LC-3's ADD instruction updates its N, Z, and P condition codes based on the result.[10]

- **Logical Operations (e.g., AND, OR, XOR, NOT):** These are implemented using the host language's bitwise operators such as & (AND), | (OR), ^ (XOR), and ~ (NOT).[20] For example, CHIP-8 features 8XY1 (Vx |= Vy), 8XY2 (Vx &= Vy), and 8XY3 (Vx ^= Vy).[9] The LC-3 architecture includes AND and NOT operations.[10] These operations usually also affect status flags (e.g., Zero and Negative flags based on the result).
- **Shift and Rotate Operations:** These are implemented using the host language's bitwise shift operators << (left shift) and >> (right shift). Rotations are more complex and may require combining shifts with bitwise OR operations to move bits from one end to the other. Status flags, particularly the Carry flag, are often affected by the bit that is shifted out of the operand.[9] For example, CHIP-8's shift instructions (8XY6 for right shift, 8XYE for left shift) store the bit shifted out into the VF register.[9]

While the core arithmetic and logical computations are often straightforward using the host language's operators, the accurate emulation of status flags is frequently the most challenging and error-prone aspect of ALU simulation.[25] Different CPUs have unique and sometimes complex rules for how their flags are set or cleared by various operations. For example, the Intel CPU architecture has a detailed set of rules for its CF (Carry), PF (Parity), AF (Adjust), ZF (Zero), SF (Sign), and OF (Overflow) flags.[24] CHIP-8's VF register serves multiple purposes depending on the instruction: it's a carry for addition, a "no borrow" flag for subtraction, and indicates pixel collision for the draw instruction.[9] LC-3 sets its N, Z, P flags based on the 16-bit two's complement interpretation of the result of an operation.[10] This means an emulator cannot simply perform, say, result = operand1 + operand2;. It must then meticulously analyze the result, and sometimes the operand1 and operand2 as well, to set the emulated flags according to the target CPU's specific architectural definition. This often requires explicit conditional logic or bit manipulation beyond the primary arithmetic or logical operation itself. The difficulty in perfectly replicating these flag behaviors is a common hurdle for emulator developers.[25]

D. Managing CPU Status Flags (Carry, Zero, Negative, Overflow, etc.)

CPU status flags (also known as condition codes or flags) are a set of single-bit values

stored in a status register (like LC-3's PSR or implicitly through CHIP-8's VF register) that reflect the outcome of the most recent ALU operation or certain other instructions (e.g., load instructions in LC-3 also set flags). These flags are crucial for conditional program execution.

The most common flags and their typical behavior are:

- **Zero Flag (Z):** Set to 1 if the result of an operation is zero; cleared to 0 otherwise.[10]
- **Negative/Sign Flag (N or S):** Set to 1 if the most significant bit (MSB) of the result is 1 (which typically indicates a negative number in two's complement representation); cleared to 0 otherwise.[10]
- **Carry Flag (C):** For addition, set to 1 if the operation generates a carry out of the most significant bit of the result (indicating an unsigned overflow); cleared otherwise. For subtraction, it's often set if a borrow is needed from the bit beyond the MSB (or, in some architectures, if no borrow occurred). It's also used in multi-precision arithmetic and bit shift/rotate operations.[9]
- **Overflow Flag (V or O):** Set to 1 if the result of a *signed* arithmetic operation is too large a positive number or too small a negative number to fit into the destination operand (excluding the sign bit); cleared otherwise. This flag indicates a signed integer overflow, which is different from the unsigned overflow indicated by the Carry Flag.[24]

In an emulator, after an instruction that affects flags is executed, the corresponding emulated flag bits must be updated. This usually involves inspecting the properties of the result value, and sometimes the operands, according to the precise rules defined by the target CPU's architecture.[24] For LC-3, a helper function like update_flags(register_index_whose_value_changed) is typically implemented to set the N, Z, and P flags based on the new value in the specified register.[27]

The accurate emulation of these status flags is not merely an academic exercise in fidelity; it is fundamental to the correct execution of most programs. The primary utility of status flags lies in enabling conditional branching instructions. These instructions (e.g., "branch if zero," "branch if negative," "branch if carry set") examine the state of one or more flags to decide whether to alter the program's flow of execution (i.e., jump to a different part of the code) or to continue sequentially.[10] Conditional branching is the bedrock upon which higher-level programming constructs like if-then-else statements, loops (for, while), and function calls are built. If an emulator fails to set the status flags correctly, conditional branches will make

incorrect decisions, leading to unpredictable program paths and almost certain failure or erroneous behavior of the emulated software.

E. Handling Input/Output (I/O) and Interrupts

CPUs do not operate in a vacuum; they interact with a variety of peripheral devices such as keyboards, displays, storage devices, and timers. Emulators must therefore simulate these interactions.

- **Input/Output (I/O):** CPUs communicate with peripherals through I/O operations. These operations can be implemented in two main ways:
  - **Memory-Mapped I/O:** Device control and data registers are assigned specific addresses in the CPU's memory map. Standard memory access instructions (like load and store) are used to interact with these registers. Reading from a specific memory address might fetch keyboard status, while writing to another might send a character to a display. LC-3 primarily uses memory-mapped I/O.[10]
  - **Port-Mapped I/O (or I/O-Mapped I/O):** The CPU has dedicated I/O instructions (e.g., IN, OUT in x86) that access a separate I/O address space, distinct from the memory address space. Emulators simulate these I/O devices as software modules. When the emulated CPU executes an instruction that accesses an I/O address (memory-mapped or port-mapped), the emulator intercepts this access and routes it to the corresponding simulated device module. This module then performs the appropriate action, which might involve interacting with the host system (e.g., reading host keyboard input, drawing to a host window).[9]
- **Interrupts:** Interrupts are signals from hardware devices (or sometimes software) that cause the CPU to temporarily suspend its current execution, save its state (at least the PC and status register), and jump to a special piece of code called an Interrupt Service Routine (ISR) to handle the event.[2] After the ISR completes, the CPU typically restores its saved state and resumes the interrupted task. Emulators must simulate this interrupt mechanism. This includes:
  - Detecting emulated interrupt conditions (e.g., a timer expiring, a key being pressed if the system uses interrupt-driven I/O).
  - Simulating the CPU's response: saving the necessary context (like PC and flags, often onto a stack), disabling further interrupts (at least of the same or lower priority), and loading the PC with the address of the appropriate ISR (often found via an interrupt vector table).[10]
  - Executing the ISR.

- Simulating the return from interrupt instruction (e.g., RTI in LC-3), which restores the saved context and re-enables interrupts.

I/O emulation serves as the crucial bridge connecting the self-contained emulated environment to the host system. It is through emulated I/O that a user can interact with the emulated software (e.g., via keyboard and mouse input mapped to the target's input devices) and observe its output (e.g., graphics rendered to a window on the host, sound played through the host's speakers). This interface between the emulated world and the host environment (sometimes managed by a dedicated CpuInterface or similar abstraction) is where many practical implementation challenges arise. Developers must address issues like mapping host inputs to the often simpler or idiosyncratic inputs of the target system, efficiently rendering the target's display within a modern graphical environment, and synchronizing emulated events with real-time host events. Bridging the "impedance mismatch" between the direct, low-level hardware access common in older systems and the abstracted, event-driven I/O models of contemporary operating systems is a significant aspect of creating a usable emulator.

V. Emulating the CHIP-8 Architecture

CHIP-8 is a popular starting point for aspiring emulator developers due to its relative simplicity and well-documented nature. Understanding its architecture is key to successfully emulating it.

A. Overview of CHIP-8: A Simple Interpreted Language

CHIP-8 was developed in the mid-1970s by Joseph Weisbecker for microcomputers such as the COSMAC VIP and Telmac 1800.[9] It is an interpreted programming language, designed to make simple game development accessible on these early 8-bit machines. Its straightforward instruction set and architecture make it an excellent introductory project for those looking to delve into the world of emulation.[11]

It's more accurate to describe CHIP-8 as a specification for a virtual machine (VM) rather than a direct hardware CPU.[15] Various physical CPUs, like the RCA 1802, had CHIP-8 *interpreters* written for them. This means that when one builds a CHIP-8 emulator, they are, in essence, writing an interpreter for this VM's bytecode. This understanding is important because it clarifies that there isn't a single, monolithic "CHIP-8 hardware" to perfectly replicate; instead, there's a common set of behaviors, components, and opcodes defined by the CHIP-8 specification that interpreters—and

thus emulators—must implement. This also helps explain some of the historical ambiguities and variations observed across different CHIP-8 versions and their interpreters.[9]

B. CHIP-8 Components

A CHIP-8 emulator needs to simulate the following components:

- **Memory:** CHIP-8 features 4 Kilobytes (4096 bytes) of Random Access Memory (RAM).[11] Memory addresses range from 0x000 to 0xFFF.
  - The first 512 bytes (0x000 to 0x1FF) were historically reserved for the CHIP-8 interpreter on original hardware. In modern emulators, this area is typically used to store the built-in font set.[11]
  - CHIP-8 programs (ROMs) are loaded into memory starting at address 0x200 (decimal 512).[11]
  - In C++, this can be represented as: uint8_t memory;.
- **Registers:**
  - **General-Purpose Registers (V0-VF):** Sixteen 8-bit registers, denoted V0 through VF.[9] The VF register has a special role, often used as a flag for arithmetic operations (carry/no borrow) or collision detection in graphics operations.[9] Represented as: uint8_t V;.
  - **Index Register (I):** One 16-bit register used to store memory addresses, primarily for operations involving memory access like reading sprite data or loading/storing multiple V-registers.[9] Represented as: uint16_t I;.
  - **Program Counter (PC):** One 16-bit register that holds the address of the next instruction to be executed. It is initialized to 0x200.[9] Represented as: uint16_t pc;.
- **Stack:** A region of memory used to store return addresses when subroutines are called. CHIP-8 specifies a stack that can hold up to 16 16-bit values, allowing for 16 levels of nested subroutine calls.[11] Represented as: uint16_t stack;.
- **Stack Pointer (SP):** An 8-bit register that points to the current top of the stack (i.e., the index of the most recently pushed return address).[11] Represented as: uint8_t sp;.
- **Timers:**
  - **Delay Timer (DT):** An 8-bit timer used for timing game events. When set to a non-zero value, it decrements at a rate of 60 Hertz (60 times per second) until it reaches 0.[9] Represented as: uint8_t delay_timer;.
  - **Sound Timer (ST):** An 8-bit timer that functions similarly to the delay timer (decrements at 60Hz when non-zero). When its value is greater than zero, the

CHIP-8 system is expected to produce a beeping sound.[9] Represented as: uint8_t sound_timer;.

- **Display:** A monochrome (black and white) display with a resolution of 64x32 pixels.[11] Graphics are drawn using sprites. Represented as: uint32_t display[64 * 32]; (where each uint32_t might represent a pixel's color for easier rendering with modern libraries).
- **Keypad:** Input is handled via a 16-key hexadecimal keypad (keys 0 through F). These are typically mapped to keys on a modern keyboard for emulator use.[9] Represented as: uint8_t keypad;.
- **Font Set:** CHIP-8 includes a built-in set of sprite data for displaying hexadecimal characters 0 through F. Each character sprite is 8 pixels wide (though only 4 are typically used for the character shape) and 5 pixels tall. This font set is usually stored in the interpreter-reserved area of memory (e.g., starting at 0x050).[15]

C. The CHIP-8 Instruction Set (Opcodes)

The CHIP-8 architecture defines a set of 35 standard opcodes (instructions). Each opcode is two bytes (16 bits) long and is typically stored in memory in big-endian format (most significant byte first).[9] When an emulator fetches these two bytes, it decodes them to determine the specific operation to perform and any arguments required by that operation.

The arguments are often denoted using placeholders in opcode descriptions:

- NNN or addr: A 12-bit memory address.
- NN or byte: An 8-bit constant.
- N or nibble: A 4-bit constant.
- X: A 4-bit value representing one of the 16 V-registers (V0-VF).
- Y: A 4-bit value representing another of the 16 V-registers (V0-VF).

Decoding often involves examining the first nibble (4 bits) of the opcode to identify the general instruction type, and then potentially other nibbles or the entire second byte for more specific variants or arguments.[9]

It is important for developers to be aware that several CHIP-8 opcodes have historical ambiguities or have seen variations in behavior across different original interpreters. For example, the shift instructions (8XY6 and 8XYE), the jump with offset instruction (often BNNN in older docs, but BXNN in some modern interpretations), and the memory store/load instructions (FX55 and FX65) have had differing interpretations regarding which registers are affected or how the index register I is modified.[29]

Modern emulators often aim for compatibility with the widest range of ROMs by adhering to "community standard" behaviors, frequently based on well-regarded technical references like "Cowgod's Chip-8 Technical Reference" (mentioned in) or the SUPER-CHIP specification.[9] This highlights the need for careful selection of reference documentation when implementing a CHIP-8 emulator.

The following table summarizes the standard CHIP-8 instruction set:

Opcode (Hex)	Mnemonic (Conceptual)	Arguments	Description	VF Flag Effect
0NNN	SYS addr	NNN	Jump to a machine code routine at NNN. (Usually ignored in modern emulators, or only 00E0 and 00EE are implemented)	N/A
00E0	CLS		Clear the display.	N/A
00EE	RET		Return from a subroutine. (Set PC to address at top of stack, then decrement SP).	N/A
1NNN	JP addr	NNN	Jump to location NNN. (Set PC to NNN).	N/A
2NNN	CALL addr	NNN	Call subroutine at NNN. (Increment SP, put current PC	N/A

			on top of stack, then set PC to NNN).	
3XNN	SE Vx, byte	X, NN	Skip next instruction if register Vx equals NN. (PC += 4 if true, else PC += 2).	N/A
4XNN	SNE Vx, byte	X, NN	Skip next instruction if register Vx does not equal NN. (PC += 4 if true, else PC += 2).	N/A
5XY0	SE Vx, Vy	X, Y	Skip next instruction if register Vx equals register Vy. (PC += 4 if true, else PC += 2).	N/A
6XNN	LD Vx, byte	X, NN	Set register Vx = NN.	N/A
7XNN	ADD Vx, byte	X, NN	Set register Vx = Vx + NN. (Carry flag VF is not changed).	Not changed
8XY0	LD Vx, Vy	X, Y	Set register Vx = register Vy.	N/A
8XY1	OR Vx, Vy	X, Y	Set register Vx = Vx OR Vy.	N/A (Modern:

			(Bitwise OR).	Resets VF to 0)
8XY2	AND Vx, Vy	X, Y	Set register Vx = Vx AND Vy. (Bitwise AND).	N/A (Modern: Resets VF to 0)
8XY3	XOR Vx, Vy	X, Y	Set register Vx = Vx XOR Vy. (Bitwise XOR).	N/A (Modern: Resets VF to 0)
8XY4	ADD Vx, Vy	X, Y	Set register Vx = Vx + Vy.	VF is set to 1 if there's a carry (result > 255), and to 0 otherwise.
8XY5	SUB Vx, Vy	X, Y	Set register Vx = Vx - Vy.	VF is set to 0 if there's a borrow (Vx < Vy), and 1 otherwise (Vx >= Vy, "no borrow").
8XY6	SHR Vx {, Vy}	X, (Y)	Set register Vx = Vx SHR 1. (Shift Vx right by 1). (Vy is ignored by modern interpreters).	VF is set to the value of the least significant bit of Vx before the shift.
8XY7	SUBN Vx, Vy	X, Y	Set register Vx = Vy - Vx.	VF is set to 0 if there's a borrow (Vy < Vx), and 1 otherwise (Vy >= Vx, "no borrow").
8XYE	SHL Vx {, Vy}	X, (Y)	Set register Vx = Vx SHL 1. (Shift	VF is set to the value of the

			Vx left by 1). (Vy is ignored by modern interpreters).	most significant bit of Vx before the shift.
9XY0	SNE Vx, Vy	X, Y	Skip next instruction if register Vx does not equal register Vy. (PC += 4 if true, else PC += 2).	N/A
ANNN	LD I, addr	NNN	Set index register I = NNN.	N/A
BNNN	JP V0, addr	NNN	Jump to location NNN + V0. (Set PC to NNN + value of V0). (Ambiguous, some use BXNN meaning PC = VX + XNN)	N/A
CXNN	RND Vx, byte	X, NN	Set register Vx = random byte AND NN. (Random number typically 0-255).	N/A
DXYN	DRW Vx, Vy, nibble	X, Y, N	Display N-byte sprite starting at memory location I at (Vx, Vy).	VF is set to 1 if any screen pixels are flipped from set to unset (collision), and to 0 otherwise.

EX9E	SKP Vx	X	Skip next instruction if key with the value of Vx is pressed. (PC += 4 if true, else PC += 2).	N/A
EXA1	SKNP Vx	X	Skip next instruction if key with the value of Vx is not pressed. (PC += 4 if true, else PC += 2).	N/A
FX07	LD Vx, DT	X	Set register Vx = delay timer value.	N/A
FX0A	LD Vx, K	X	Wait for a key press, store the value of the key in register Vx. (Halts execution until key press).	N/A
FX15	LD DT, Vx	X	Set delay timer = Vx.	N/A
FX18	LD ST, Vx	X	Set sound timer = Vx.	N/A
FX1E	ADD I, Vx	X	Set I = I + Vx.	VF set to 1 if I overflows (I > 0xFFF), else 0 (on some interpreters, not universally

				agreed).
FX29	LD F, Vx	X	Set I = location of sprite for digit Vx. (Font characters 0-F are 5 bytes each, I points to the 5-byte character for the hex digit in Vx).	N/A
FX33	LD B, Vx	X	Store BCD representation of Vx in memory locations I, I+1, and I+2. (Hundreds digit at I, tens at I+1, ones at I+2).	N/A
FX55	LD [I], Vx	X	Store registers V0 through Vx in memory starting at location I. (I is modified by modern interpreters: I = I + X + 1 afterwards).	N/A
FX65	LD Vx, [I]	X	Read registers V0 through Vx from memory starting at location I. (I is modified by modern interpreters: I = I + X + 1	N/A

			afterwards).	

(Primary sources for table:.[9] "Modern" interpretations often align with Cowgod's CHIP-8 Technical Reference.)

D. Implementing Key CHIP-8 Operations

Beyond generic instruction execution, some CHIP-8 operations require special attention due to their interactive nature or specific behavior:

- **Graphics Rendering (Opcode DXYN - DRW Vx, Vy, N):** This is the sole instruction for drawing graphics. It reads N bytes of sprite data starting from the memory address stored in the Index Register (I). Each byte represents a row of 8 pixels. These sprite pixels are then drawn onto the 64x32 monochrome display at coordinates specified by registers VX and VY. The drawing is done using an XOR (exclusive OR) operation: if a sprite pixel is 1 and the corresponding screen pixel is 1, the screen pixel becomes 0 (erased); if the sprite pixel is 1 and the screen pixel is 0, the screen pixel becomes 1 (drawn). If a sprite pixel is 0, the screen pixel is unchanged. A crucial part of this instruction is collision detection: if any screen pixel is flipped from 1 (on) to 0 (off) as a result of the XOR operation, the VF register is set to 1; otherwise, it is set to 0.[9] While sprite coordinates (VX, VY) typically wrap around the screen if they exceed display boundaries, the actual drawing of the sprite pixels should be clipped at the screen edges.[29]
- **Timer Decrements and Sound:** The Delay Timer (DT) and Sound Timer (ST) must be managed independently of the CPU's instruction execution speed. Both timers, if set to a value greater than zero, should be decremented at a consistent rate of 60 Hertz (60 times per second).[9] This is typically handled within the emulator's main loop by checking elapsed real time. When the Sound Timer's value is greater than zero, the emulator should produce a beep or buzzing sound.[9] The actual sound generation can be implemented using host system audio libraries.
- **Input Handling:** CHIP-8 expects input from a 16-key hexadecimal keypad. An emulator must map these 16 virtual keys to keys on the host computer's keyboard (e.g., mapping the top row 1,2,3,4 to CHIP-8 keys 1,2,3,C).[12] The state of these emulated keys (pressed or not pressed) is then used by specific CHIP-8 opcodes:
    - EX9E (SKP Vx): Skips the next instruction if the key corresponding to the value in register Vx is currently pressed.

- EXA1 (SKNP Vx): Skips the next instruction if the key corresponding to the value in register Vx is *not* currently pressed.
- FX0A (LD Vx, K): Halts program execution until a key is pressed. The value of the pressed key (0-F) is then stored in register Vx, and execution resumes.[9]

E. Step-by-Step Emulation Loop for CHIP-8

A typical CHIP-8 emulation loop, integrating the fetch-decode-execute cycle with CHIP-8 specific operations, proceeds as follows:

1. **Initialization:**
   - Clear all 16 V-registers, the I register, and the stack. Set SP to -1 (or 0, depending on implementation).
   - Clear the 64x32 display buffer.
   - Clear the keypad state array.
   - Load the CHIP-8 font set (sprites for characters 0-F) into the reserved memory area (typically starting at 0x050).[15]
   - Load the CHIP-8 program (ROM file) into memory, starting at address 0x200.[11]
   - Initialize the Program Counter (PC) to 0x200.[11]
   - Initialize Delay Timer (DT) and Sound Timer (ST) to 0.
2. **Main Emulation Loop:** This loop runs continuously.
   - **Fetch Opcode:** Read the two-byte opcode from emulated memory at the address pointed to by PC (opcode = (memory[PC] << 8) | memory[PC + 1];).[11]
   - **Decode Opcode:** Analyze the fetched opcode to determine the instruction type and extract its arguments (X, Y, N, NN, NNN). This is commonly done using bitwise AND operations and shifts to isolate parts of the opcode, often within a series of switch statements.[11]
   - **Increment Program Counter:** Advance the PC by 2 to point to the next instruction (PC += 2;). This is done *before* executing the current instruction, as jump or call instructions will overwrite the PC anyway. Conditional skip instructions will further increment PC by 2 if the condition is met.[11]
   - **Execute Instruction:** Perform the action corresponding to the decoded opcode. This involves manipulating the emulated registers (V0-VF, I, SP), memory, timers, display buffer, or keypad state, and updating the VF flag as required by the specific instruction.[11]
   - **Update Timers (at 60Hz):** Independently of the CPU instruction rate, check if enough real time has passed (e.g., 1/60th of a second). If so:
     - If DT > 0, decrement DT (DT--;).

        - If ST > 0, decrement ST (ST--;). If ST is now positive, continue playing a beep; if ST just became 0, stop the beep.[12]
    - **Handle Host Input:** Process any input events from the host system (e.g., keyboard presses/releases) and update the state of the 16 emulated CHIP-8 keys in the keypad array.[11]
    - **Render Display (if needed):** If the display buffer was modified by an instruction during the execute step (e.g., 00E0 CLS or DXYN DRW), redraw the contents of the 64x32 emulated display to the host system's screen or window.[11] This is often done at a fixed rate (e.g., 60 FPS) or only when a draw instruction occurs.
    - **Control Emulation Speed:** To prevent the emulation from running too fast on modern hardware, a delay might be introduced in each loop iteration to target a specific number of CHIP-8 instructions per second (e.g., around 500-700 IPS is a common target).[12]
    - **Repeat:** Go back to the Fetch Opcode step.

This loop continues until the emulator is explicitly stopped by the user or, in some rare cases, if the CHIP-8 program executes an implicit halt (though CHIP-8 has no formal HALT instruction; programs usually enter an infinite loop).

VI. Emulating the LC-3 Architecture

The Little Computer 3 (LC-3) is another architecture often encountered in the context of emulation, primarily due to its widespread use in computer science education. It offers a more complex challenge than CHIP-8 but remains significantly simpler than commercial CPUs like x86 or ARM.

A. Overview of LC-3: An Educational Architecture

The LC-3 is an assembly language and a model computer architecture specifically designed for educational purposes.[3] Its instruction set, while relatively simple compared to contemporary ISAs, is rich enough to teach fundamental concepts of computer organization, assembly language programming, memory addressing modes, I/O handling, and even how a C compiler might translate high-level code into machine instructions.[3] Unlike CHIP-8, which was used on actual hardware, the LC-3 is predominantly encountered and utilized in simulated environments on modern computers.[3]

The educational design of LC-3 has fostered an ecosystem of supporting tools.

Readily available LC-3 simulators (like LC3Tools or PennSim), assemblers that convert LC-3 assembly code into machine code, and even rudimentary C compilers targeting the LC-3 architecture are common.[3] This suite of tools is not accidental but rather a direct consequence of its pedagogical intent. These resources are integral to how LC-3 is taught, emphasizing hands-on practice through simulation. For an aspiring emulator developer, these existing, well-tested tools serve as invaluable references. They provide a "known-good" baseline against which one's own LC-3 emulator implementation can be tested and validated, significantly aiding the debugging process.

B. LC-3 Components

An LC-3 emulator must simulate the following architectural components:

- **Memory Organization:**
  - **Word Size:** The LC-3 has a 16-bit word size. All registers and data paths are 16 bits wide.[3]
  - **Address Space:** It uses 16-bit addresses, providing an address space of 216 (65,536) unique memory locations, typically addressed from x0000 to xFFFF (hexadecimal).[3]
  - **Addressability:** Memory is word-addressable, meaning each 16-bit address refers to a 16-bit word in memory. This contrasts with byte-addressable systems where each byte has a unique address.[17]
  - **Memory Map:** Specific regions of this memory space are reserved for particular purposes, such as the Trap Vector Table (e.g., x0000-x00FF), Interrupt Vector Table, operating system routines and stack (x0200-x2FFF), user programs (x3000-xFDFF), and memory-mapped I/O device registers (xFE00-xFFFF).[10]
  - In a software emulator, this memory can be represented as an array of 16-bit unsigned integers: uint16_t memory;.[34]
- **Registers:**
  - **General-Purpose Registers (GPRs):** Eight 16-bit general-purpose registers, designated R0 through R7.[3] These can be used for arbitrary data storage and calculations by programs. Represented as: uint16_t R;.[34]
  - **Program Counter (PC):** A 16-bit register that holds the memory address of the *next* instruction to be fetched and executed.[10] Represented as: uint16_t pc;.[34]
  - **Instruction Register (IR):** A 16-bit register that holds the current instruction after it has been fetched from memory and is being decoded and executed.

This register is not directly programmable by LC-3 assembly instructions but is a crucial part of the conceptual CPU model.
    - **Processor Status Register (PSR):** A 16-bit register containing important status information about the currently executing process. This includes:
        - **Privilege Mode (PSR):** 1 bit indicating whether the CPU is in User mode (1) or Supervisor mode (0).
        - **Priority Level (PSR[10:8]):** 3 bits specifying the priority level of the current process.
        - **Condition Codes (PSR[2:0]):** The N, Z, and P flags.[10]
- **Condition Codes (CC):** Three 1-bit flags that are part of the PSR:
    - **N (Negative):** Set to 1 if the result of the last operation that modified a GPR was negative.
    - **Z (Zero):** Set to 1 if the result was zero.
    - **P (Positive):** Set to 1 if the result was positive (and non-zero). These flags are set by most computational instructions (ADD, AND, NOT) and data movement instructions that load a value into a register (LD, LDI, LDR, LEA).[3] Exactly one of these three flags is set at any given time.

C. The LC-3 Instruction Set Architecture (ISA)

The LC-3 ISA defines 15 distinct types of instructions (the 16th opcode is reserved).[3] Each instruction is 16 bits wide.

- **Opcode:** The most significant 4 bits (bits) of an instruction specify its opcode, determining the operation to be performed.[3]
- **Instruction Categories:** LC-3 instructions are generally grouped into three categories [10]:
    1. **Operate Instructions:** Perform computations (e.g., ADD, AND, NOT). These typically use registers as operands, though ADD and AND can also use a small immediate value.
    2. **Data Movement Instructions:** Transfer data between memory and registers (e.g., LD, ST, LDR, STR, LDI, STI) or compute an address and load it into a register (LEA).
    3. **Control Instructions:** Alter the flow of program execution (e.g., BR (branch), JMP (jump), JSR (jump to subroutine), TRAP (system call), RTI (return from interrupt)).
- **Instruction Formats:** The remaining 12 bits of an instruction (bits [11:0]) are used to specify operands, such as destination registers (DR), source registers (SR1, SR2), immediate values (e.g., imm5), offsets for memory addressing (e.g.,

PCoffset9, offset6), or trap vector numbers (trapvect8). The interpretation of these bits depends on the specific opcode.[10]

The following table provides a summary of the LC-3 instruction set, with details based primarily on [[10] (Appendix A)]:

Opcode (Binary)	Name	Assembly Example(s)	Format (Bits [15:0])	Description of Operation	Addressing Modes Used	CC Affected
0001	ADD	ADD DR, SR1, SR2 ADD DR, SR1, imm5	0001 DR SR1 0 00 SR2 0001 DR SR1 1 imm5	If bit=0, DR = SR1 + SR2. If bit=1, DR = SR1 + SEXT(imm5).	Register, Immediate	N,Z,P
0101	AND	AND DR, SR1, SR2 AND DR, SR1, imm5	0101 DR SR1 0 00 SR2 0101 DR SR1 1 imm5	If bit=0, DR = SR1 & SR2. If bit=1, DR = SR1 & SEXT(imm5).	Register, Immediate	N,Z,P
0000	BR	BRn target BRzp target BRnzp target	0000 n z p PCoffset9	If specified condition code(s) (n,z,p corresponding to N,Z,P flags) is set, PC = PC + SEXT(PCoffset9).	PC-Relative	Unchanged

				(PC is already incremented).		
1100	JMP	JMP BaseR RET (is JMP R7)	1100 000 BaseR 000000	PC = BaseR. For RET, PC = R7.	Register Direct	Unchanged
0100	JSR/JSRR	JSR target JSRR BaseR	0100 1 PCoffset11 0100 0 00 BaseR 000000	R7 = PC (incremented PC). If bit=1 (JSR), PC = PC + SEXT(PCoffset11). If bit=0 (JSRR), PC = BaseR.	PC-Relative, Register Direct	Unchanged
0010	LD	LD DR, target	0010 DR PCoffset9	DR = Mem. (PC is already incremented).	PC-Relative	N,Z,P
1010	LDI	LDI DR, target	1010 DR PCoffset9	DR = Mem]. (PC is already incremented).	Indirect (via PC-Relative)	N,Z,P
0110	LDR	LDR DR, BaseR, offset6	0110 DR BaseR offset6	DR = Mem.	Base+Offset	N,Z,P

1110	LEA	LEA DR, target	1110 DR PCoffset9	DR = PC + SEXT(PCoffset9). (PC is already incremented). Does not access memory for operand.	PC-Relative (for address calculation)	N,Z,P
1001	NOT	NOT DR, SR	1001 DR SR 111111	DR = ~SR (bitwise NOT).	Register	N,Z,P
1000	RTI	RTI	1000 0000000000000	If in supervisor mode, pop PC and PSR from supervisor stack. Else, privilege mode exception.	N/A (Stack)	N,Z,P (restored from stack)
0011	ST	ST SR, target	0011 SR PCoffset9	Mem = SR. (PC is already incremented).	PC-Relative	Unchanged
1011	STI	STI SR, target	1011 SR PCoffset9	Mem] = SR. (PC is already incremented).	Indirect (via PC-Relative)	Unchanged

0111	STR	STR SR, BaseR, offset6	0111 SR BaseR offset6	Mem = SR.	Base+Offset	Unchanged
1111	TRAP	TRAP trapvect8 (e.g. GETC, OUT, PUTS, HALT)	1111 0000 trapvect8	R7 = PC (incremented PC). PC = Mem. (Enters supervisor mode).	Trap Vector	Unchanged (R7 loaded)

(Note: SEXT denotes sign-extension to 16 bits, ZEXT denotes zero-extension. PC refers to the incremented PC after fetch).

D. LC-3 Instruction Cycle in Detail (Focus on Address Evaluation & Operand Fetch)

The LC-3 instruction cycle can be expanded into roughly six phases, providing a more detailed view than the basic Fetch-Decode-Execute model, especially for instructions involving memory access.[27]

1. **Fetch:**
    - The Program Counter (PC) contains the address of the next instruction. This address is placed into the Memory Address Register (MAR) (conceptually).
    - A memory read operation is initiated. The instruction at Mem is fetched and placed into the Memory Data Register (MDR) (conceptually).
    - The content of MDR is copied into the Instruction Register (IR).
    - The PC is incremented by 1 (since LC-3 instructions are one word long) to point to the next sequential instruction.[27]
2. **Decode:**
    - The opcode (IR[15:12]) is examined by the control unit to identify the instruction type.
    - Other relevant fields from the IR (e.g., register specifiers, offsets, immediate values) are identified based on the opcode.[38]
3. **Evaluate Address:**
    - This phase is crucial for instructions that access memory (LD, ST, LDR, STR, LDI, STI) or compute an address (LEA, BR, JSR).

- The effective memory address (EA) or target address is calculated based on the instruction's addressing mode. This might involve:
  - Adding a sign-extended offset from the IR to the (already incremented) PC.
  - Adding a sign-extended offset from the IR to the contents of a base register (BaseR).
  - For indirect addressing, an initial address is calculated (often PC-relative), and the value at this memory location is then used as the final effective address.
- The calculated EA is conceptually stored in the MAR.[17]
- A critical detail here is **sign extension**. Many LC-3 instructions use offsets (e.g., PCoffset9 for LD/ST/LEA/BR, imm5 for ADD/AND, offset6 for LDR/STR) that are shorter than 16 bits. These offsets must be sign-extended to 16 bits before being used in address calculations or arithmetic operations.[10] Sign extension ensures that negative offsets (indicated by a 1 in the most significant bit of the offset field) are correctly interpreted. For example, a 9-bit PCoffset allows branching approximately $\pm$256 words from the current PC. If an offset representing a backward branch (e.g., 111111111 for -1) were not sign-extended, it would be treated as a large positive value, leading to incorrect program flow. The sign_extend function shown in some LC-3 emulator implementations [27] handles this by checking the MSB of the short offset; if it's 1, the higher bits of the 16-bit word are filled with 1s.

4. **Fetch Operands:**
   - If the instruction requires reading an operand from memory (e.g., LD, LDR, LDI), a memory read is performed using the address in MAR. The fetched data is placed in MDR and then typically transferred to a destination register (DR).[27]
   - If operands are located in general-purpose registers (e.g., for ADD R1, R2, R3), their values are fetched from the emulated register file.[27]
   - If an operand is an immediate value embedded in the instruction (e.g., imm5 in ADD R1, R2, #val), it is extracted from the IR and, if necessary (as with imm5), sign-extended.[27]
5. **Execute:**
   - The ALU performs the specified arithmetic or logical operation using the fetched operands.
   - For control flow instructions, the PC might be updated with a new target address.
   - For TRAP instructions, system-specific actions are taken.[27]

6. **Store Result:**
    - If the instruction produces a result that needs to be stored, it is written to the designated destination:
        - A general-purpose register (DR) for operate instructions or load instructions.
        - A memory location (specified by MAR) for store instructions (ST, STR, STI).
    - Crucially, if the instruction writes to a GPR (ADD, AND, NOT, LD, LDI, LDR, LEA), the Condition Codes (N, Z, P) in the PSR must be updated based on the value written to the register.[10]

E. Implementing LC-3 Addressing Modes

The "Evaluate Address" and "Fetch Operands" stages are intimately tied to the LC-3's addressing modes. Here's how they are typically implemented in an emulator:

- **PC-Relative Addressing (used by LD, ST, LEA, BR, JSR):**
    - **Evaluate Address:** The effective address (EA) or target branch address is calculated by taking the current (already incremented) Program Counter (PC), extracting the 9-bit offset (PCoffset9 from IR[8:0]) or 11-bit offset (PCoffset11 from IR[10:0] for JSR), sign-extending this offset to 16 bits, and adding it to the PC. EA = current_PC + sign_extend(offset_from_IR); .[10]
    - **Fetch Operands (for LD):** operand = memory[EA];
    - **Store Result (for ST):** memory[EA] = value_from_SR;
    - The LEA (Load Effective Address) instruction is a special case. It calculates EA = current_PC + sign_extend(PCoffset9) but then, instead of using EA to access memory, it loads the *value of EA itself* into the destination register DR: DR = EA;.[10] This distinction is vital: LEA computes an address and stores that address, it does not perform a memory lookup for its final operand. This is often a point of confusion for those new to LC-3, as they might expect it to behave like other load instructions.
- **Base+Offset Addressing (used by LDR, STR):**
    - **Evaluate Address:** The EA is calculated by taking the 16-bit value from a specified base register (BaseR, one of R0-R7, identified by IR[8:6]), extracting the 6-bit offset (offset6 from IR[5:0]), sign-extending this offset to 16 bits, and adding it to the value from BaseR. EA = R + sign_extend(offset6_from_IR); .[10]
    - **Fetch Operands (for LDR):** operand = memory[EA];
    - **Store Result (for STR):** memory[EA] = value_from_SR;
- **Indirect Addressing (used by LDI, STI):**
    - **Evaluate Address (Phase 1 - Pointer Address):** First, an intermediate

address (pointer address) is calculated using PC-relative addressing: pointer_address = current_PC + sign_extend(PCoffset9_from_IR);
    - **Evaluate Address (Phase 2 - Final EA):** The value stored at this pointer_address in memory is the actual effective address of the data: Final_EA = memory[pointer_address]; .[10]
    - **Fetch Operands (for LDI):** operand = memory[Final_EA];
    - **Store Result (for STI):** memory[Final_EA] = value_from_SR;
- **Immediate Addressing (used by ADD, AND):**
    - **Fetch Operands:** The operand is a 5-bit immediate value (imm5 from IR[4:0]) embedded in the instruction. This value is sign-extended to 16 bits before being used in the ALU operation. immediate_operand = sign_extend(imm5_from_IR); .[10]
- **Register Direct Addressing (used by ADD, AND, NOT, JMP BaseR, JSRR BaseR):**
    - **Fetch Operands:** The operand(s) are directly contained in one or two specified general-purpose registers (SR1 from IR[8:6], SR2 from IR[2:0], or BaseR from IR[8:6]). The emulator simply reads the values from its emulated register array. operand1 = R; operand2 = R; (if applicable) .[10]
- **Trap Vector Addressing (used by TRAP):**
    - **Evaluate Address:** The 8-bit trap vector number (trapvect8 from IR[7:0]) is zero-extended to 16 bits. This forms an address that points into the Trap Vector Table (typically starting at memory location x0000). trap_vector_table_entry_address = zero_extend(trapvect8_from_IR);
    - The content of memory[trap_vector_table_entry_address] is the starting address of the system service routine. The PC is loaded with this address. Before the jump, the current PC (already incremented) is saved in R7 to allow return from the trap routine.[10]

F. Handling Condition Codes (N, Z, P) after Operations

After any LC-3 instruction that writes a new value into one of the general-purpose registers (R0-R7)—specifically ADD, AND, NOT, LD, LDI, LDR, and LEA—the three condition code flags (N, Z, P) in the Processor Status Register must be updated.[10] These flags reflect the nature of the 16-bit value that was just written, when interpreted as a two's complement signed integer.

The logic for updating these flags, often encapsulated in a helper function (e.g., update_flags(uint16_t result_value) or by passing the register index update_flags(int

dr_index) [27]), is as follows:

1. **If result_value == 0:**
    - Set Z (Zero flag) to 1.
    - Set N (Negative flag) to 0.
    - Set P (Positive flag) to 0.
2. **Else if (result_value >> 15) & 1 is true (i.e., the most significant bit, bit 15, is 1):**
    - This indicates a negative number in two's complement representation.
    - Set N (Negative flag) to 1.
    - Set Z (Zero flag) to 0.
    - Set P (Positive flag) to 0.
3. **Else (the value is not zero and the most significant bit is 0):**
    - This indicates a positive number.
    - Set P (Positive flag) to 1.
    - Set N (Negative flag) to 0.
    - Set Z (Zero flag) to 0.

It is guaranteed that exactly one of the N, Z, or P flags will be set after any instruction that modifies them.[10] These flags are then used by the conditional branch (BR) instructions to make decisions about program flow.

VII. Advanced Emulation Concepts and Considerations

While the core fetch-decode-execute cycle and component simulation form the basis of most emulators, particularly for simpler architectures like CHIP-8 and LC-3, several advanced concepts come into play for more complex systems or when higher performance and accuracy are desired.

A. Cycle-Accurate Emulation

Cycle-accurate emulation represents a more rigorous level of fidelity, aiming to replicate not just the outcome of instructions but also the precise number of CPU clock cycles each instruction takes to execute on the original hardware.[1] This extends to emulating the exact timing of memory accesses, I/O events, and interactions between different hardware components (like CPU and GPU) within those cycles. Such precision is critical for software that has strict timing dependencies. This is often the case with advanced demo-scene productions, which push hardware to its limits, or certain games where gameplay mechanics, audio synchronization, or graphical effects rely on the exact timing characteristics of the original system.

Achieving true cycle accuracy is an exceedingly complex and computationally demanding endeavor. It requires an intimate understanding of the target hardware's microarchitecture, including details such as instruction pipelines, cache behavior, memory wait states, bus arbitration, and undocumented quirks—information that is often not publicly available or is difficult to reverse engineer. While essential for some platforms [1], for simpler architectures like CHIP-8 or the typical educational use of LC-3, instruction-level accuracy (correctly executing each instruction) is usually sufficient. The significant development overhead and performance penalty associated with cycle accuracy make it a specialized pursuit, generally reserved for emulators of systems where such meticulous timing is demonstrably necessary for compatibility with key software titles or for specific research purposes.

B. Dynamic Recompilation (Just-In-Time - JIT)

Dynamic recompilation, often referred to as Just-In-Time (JIT) compilation, is an advanced emulation technique employed to enhance performance, especially when emulating more powerful target CPUs. Instead of interpreting each target instruction one by one every time it's encountered, a JIT-based emulator translates blocks of the target machine code into machine code that can be natively executed by the host CPU. This translation happens at runtime, "just in time" before the code block is needed.

Once translated, these blocks of native host code are cached. Subsequent executions of the same target code block can then directly run the cached native version, bypassing the interpretation overhead entirely. This can lead to substantial speed improvements, making it feasible to emulate complex architectures at or near their original speeds. However, JIT compilation introduces its own complexities, including the need to develop a recompiler, manage the translated code cache, and handle issues like self-modifying code (where the target program alters its own instructions in memory, potentially invalidating translated blocks). For relatively simple ISAs like CHIP-8 (with only 35 opcodes [9]) or LC-3 (with 15 opcodes [3]), the overhead of interpretation is generally low enough on modern host CPUs that the complexity of implementing a JIT recompiler would likely outweigh any performance benefits. The "8-10x horsepower" rule [2] implies that current hosts can interpret these simple instructions very rapidly, making straightforward interpreters perfectly adequate.

C. Debugging Tools and Techniques for Emulators

Developing an emulator is a complex task, and robust debugging capabilities are

invaluable, not just for the emulator developer but also for users who might want to analyze or debug the target software running within the emulator. Building these tools directly into the emulator can provide insights that are difficult or impossible to achieve with debuggers on native hardware. Common debugging features include:

- **Step-by-Step Execution:** Allowing the user to advance the emulation one instruction at a time, observing the state of the system after each step.[20] Some emulators offer finer-grained stepping, such as "step over" or "step into" subroutines.[32]
- **Register and Memory Viewers/Editors:** Displaying the current values of all emulated CPU registers (GPRs, PC, SP, I, flags) and allowing inspection and even modification of emulated memory contents.[18] This provides a live window into the emulated machine's state.
- **Breakpoints:** The ability to pause emulation when the PC reaches a specific memory address, or when certain conditions are met (e.g., a particular memory location is written to, known as a watchpoint).[16] This allows developers to examine the system state at critical junctures.
- **Disassembler:** Translating the machine code at the current PC (or a specified memory region) back into human-readable assembly language mnemonics. This helps in understanding what the target program is doing at a low level.
- **Logging and Tracing:** Recording a history of executed instructions, register state changes, memory accesses, or other significant events.[40] This trace can be invaluable for post-mortem analysis of bugs or complex program behavior.

Debugging *within* an emulator offers a unique level of control and visibility. Because the emulator *is* the (virtual) hardware, it can non-intrusively inspect and modify any aspect of the CPU's state, memory, or I/O at any point in time.[1] It can pause execution with perfect precision and, in some advanced cases, even support features like reverse execution or "undoing" instructions.[40] This comprehensive oversight is extremely powerful for understanding intricate bugs in either the emulated software or the emulator itself, enabling a fine-grained analysis that often surpasses the capabilities of native hardware debuggers.

VIII. Conclusion and Further Exploration

CPU emulation is a fascinating intersection of computer architecture, software engineering, and digital history. It allows us to preserve the past, develop for the future, and learn intricate details about how computers fundamentally operate.

A. Recap of Key Emulation Principles

The journey of emulating a CPU revolves around several core tenets. At its heart is the meticulous software implementation of the **fetch-decode-execute cycle**, where the emulator continuously retrieves instructions from a simulated memory, discerns their meaning, and performs the corresponding actions. This requires creating software representations of all essential CPU components: **memory** (typically an array), **registers** (variables of appropriate types), and the **ALU's functionality** (using host CPU operations while carefully managing **status flags**). The accuracy of this simulation, from instruction outcomes to potentially cycle-level timing, directly impacts compatibility with target software. Different strategies, such as **Low-Level Emulation (LLE)** focusing on hardware mimicry and **High-Level Emulation (HLE)** focusing on behavioral replication, offer trade-offs between authenticity and performance.

For specific architectures like **CHIP-8**, emulation involves managing its unique 4KB memory layout, 16 V-registers (including the multi-purpose VF flag), index register I, stack, timers, 64x32 monochrome display, and 16-key input, all while interpreting its 35 distinct 2-byte opcodes. Key operations include XOR-based sprite drawing and 60Hz timer decrements.

For the **LC-3**, the focus shifts to its 16-bit word-addressable memory, eight GPRs, program counter, and processor status register holding the crucial N, Z, P condition codes. Its 15-instruction ISA features more complex addressing modes (PC-relative, base+offset, indirect, immediate) that require careful implementation during the evaluate address and fetch operand stages of the instruction cycle, with particular attention to sign extension of offsets and immediate values.

B. Next Steps for Aspiring Emulator Developers

For those inspired to embark on their own emulator development journey, several paths and resources can prove beneficial:

1. **Start Simple:** Begin with a well-documented and relatively simple architecture. CHIP-8 is an excellent first choice due to its small instruction set and straightforward design.[12] The LC-3, while more complex, is also a good target given its educational focus and available documentation.[3]
2. **Consult Technical References:** Seek out reliable technical documentation for the target architecture. For CHIP-8, resources like "Cowgod's Chip-8 Technical Reference" are invaluable. For LC-3, the textbook "Introduction to Computing

Systems: From Bits & Gates to C/C++ & Beyond" by Patt and Patel is the definitive source, and its associated simulators offer a reference implementation.[3]

3. **Join Communities:** Online communities such as the EmuDev subreddit (r/EmuDev) or forums dedicated to specific systems can provide support, guidance, and a wealth of shared knowledge from experienced developers.
4. **Iterative Development:** Don't try to implement everything at once. Start with the basic CPU loop, memory, and a few simple instructions. Test thoroughly at each stage. Gradually add more instructions, peripherals, and debugging features. For CHIP-8, getting a simple ROM to display something is a major milestone. For LC-3, running basic arithmetic or memory test programs is a good start.
5. **Utilize Existing Simulators for Testing:** For architectures like LC-3, use existing, trusted simulators (e.g., LC3Tools, PennSim [32]) to run test programs and compare the state of your emulator (registers, memory, flags) against the reference output after each instruction. This is an extremely effective way to catch bugs.
6. **Focus on Debugging Tools Early:** Incorporate basic debugging features into your emulator from the outset, such as printing register states, memory dumps, and the current instruction being executed. This will save immense time and frustration later.[29]
7. **Study Open-Source Emulators:** Examine the source code of existing open-source emulators for the architecture you are interested in, or for similar systems. This can provide practical insights into how others have solved common problems (e.g., GitHub repositories for CHIP-8 or LC-3 emulators [40]).

The process of writing an emulator is a challenging yet deeply rewarding endeavor. It offers unparalleled insight into the inner workings of computers and provides a tangible connection to the history of computing. With patience, persistence, and a methodical approach, creating a functional CPU emulator is an achievable goal that significantly enhances one's understanding of the digital world.

Works cited

1. Emulator - Wikipedia, accessed June 4, 2025, https://en.wikipedia.org/wiki/Emulator
2. Principles of Software Emulation - joe bertolami, accessed June 4, 2025, https://bertolami.com/files/emulation.pdf
3. Little Computer 3 - Wikipedia, accessed June 4, 2025, https://en.wikipedia.org/wiki/Little_Computer_3
4. CPU with Two Instructions - Writing a RISC-V Emulator in Rust, accessed June 4, 2025, https://book.rvemu.app/hardware-components/01-cpu.html

5. High/Low level emulation - Emulation General Wiki, accessed June 4, 2025, https://emulation.gametechwiki.com/index.php/High/Low_level_emulation
6. Emulation accuracy - Emulation General Wiki, accessed June 4, 2025, https://emulation.gametechwiki.com/index.php/Emulation_accuracy
7. Emulator - Wikipedia, accessed June 4, 2025, https://en.wikipedia.org/wiki/Emulator#High-_and_low-level_emulation
8. FDE Cycle A Level | OCR Computer Science Revision Notes, accessed June 4, 2025, https://www.savemyexams.com/a-level/computer-science/ocr/17/revision-notes/1-the-characteristics-of-contemporary-processors-input-output-and-storage-devices/1-1-structure-and-function-of-the-processor/fetch-decode-execute-cycle/
9. CHIP-8 - Wikipedia, accessed June 4, 2025, https://en.wikipedia.org/wiki/CHIP-8
10. www.cs.colostate.edu, accessed June 4, 2025, https://www.cs.colostate.edu/~cs270/.Spring23/resources/PattPatelAppA.pdf
11. Writing a Chip-8 Emulator from Scratch in JavaScript | Tania's Website, accessed June 4, 2025, https://www.taniarascia.com/writing-an-emulator-in-javascript-chip8/
12. CHIP-8 Emulation - O. Valadares' Blog, accessed June 4, 2025, https://otavio.dev/2024/12/08/chip-8-emulation/
13. Good tutorial on writing a simple computer emulator? : r/compsci - Reddit, accessed June 4, 2025, https://www.reddit.com/r/compsci/comments/1vg3zg/good_tutorial_on_writing_a_simple_computer/
14. How a CPU works: Bare metal C on my RISC-V toy CPU - Florian Noeding's blog, accessed June 4, 2025, https://florian.noeding.com/posts/risc-v-toy-cpu/cpu-from-scratch/
15. Building a CHIP-8 Emulator [C++] - Austin Morlan, accessed June 4, 2025, https://austinmorlan.com/posts/chip8_emulator/
16. Next level CPU emulating : r/EmuDev - Reddit, accessed June 4, 2025, https://www.reddit.com/r/EmuDev/comments/1iqt0or/next_level_cpu_emulating/
17. LC3-1 The LC-3, accessed June 4, 2025, https://www.unm.edu/~zbaker/ece238/slides/lc3-1.pdf
18. Programming Model 1, accessed June 4, 2025, https://cs.newpaltz.edu/~phamh/aos/lab/Lab04-ProgrammingModel1
19. daytonpe/computer-emulator: Simulates a simple computer system consisting of a CPU and Memory which run in separate processes. - GitHub, accessed June 4, 2025, https://github.com/daytonpe/computer-emulator
20. CPU Emulator Tutorial - Phoenix!, accessed June 4, 2025, https://phoenix.goucher.edu/~kelliher/f2023/cs220/cpuEmulatorTutorial
21. Simple CPU Simulator - Computer Science - Kalamazoo College, accessed June 4, 2025, https://www.cs.kzoo.edu/cs101/labs/cpu/cpuSimulator.html
22. Verilog code for Arithmetic Logic Unit (ALU) - FPGA4student.com, accessed June

4, 2025, https://www.fpga4student.com/2017/06/Verilog-code-for-ALU.html
23. VHDL code for Arithmetic Logic Unit (ALU) - FPGA4student.com, accessed June 4, 2025, https://www.fpga4student.com/2017/06/vhdl-code-for-arithmetic-logic-unit-alu.html
24. what conditions set the overflow and carry flags?, accessed June 4, 2025, https://stackoverflow.com/questions/28164227/what-conditions-set-the-overflow-and-carry-flags
25. Using native assembly language to emulate status flag state : r/EmuDev - Reddit, accessed June 4, 2025, https://www.reddit.com/r/EmuDev/comments/11yb66a/using_native_assembly_language_to_emulate_status/
26. A new cycle-stepped 6502 CPU emulator - The Brain Dump, accessed June 4, 2025, https://floooh.github.io/2019/12/13/cycle-stepped-6502.html
27. Write your Own Virtual Machine - Justin Meiners, accessed June 4, 2025, https://www.jmeiners.com/lc3-vm/
28. Building a CHIP-8 Emulator in Rust - An Advanced Adventure - DEV Community, accessed June 4, 2025, https://dev.to/trish_07/building-a-chip-8-emulator-in-rust-an-advanced-adventure-1kb4
29. Guide to making a CHIP-8 emulator - Tobias V. Langhoff, accessed June 4, 2025, https://tobiasvl.github.io/blog/write-a-chip-8-emulator/
30. CHIP-8 extensions and compatibility, accessed June 4, 2025, https://chip-8.github.io/extensions/
31. Building and understanding a Virtual Machine in 2hrs (CHIP-8 Emulator) - YouTube, accessed June 4, 2025, https://www.youtube.com/watch?v=jWpbHC6DtnU
32. Guide to Using LC3Tools, accessed June 4, 2025, https://acsa.ustc.edu.cn/ics/download/lc3/GuideToUsingLC3Tools.pdf
33. LC-3 Edit and PennSim Tutorial - cs.wisc.edu, accessed June 4, 2025, https://pages.cs.wisc.edu/~markhill/cs252/Fall2009/includes/lc3editguide.html
34. Implementing LC-3 in Fortran - Tutorials, accessed June 4, 2025, https://fortran-lang.discourse.group/t/implementing-lc-3-in-fortran/6150
35. en.wikipedia.org, accessed June 4, 2025, https://en.wikipedia.org/wiki/Little_Computer_3#:~:text=The%20LC%2D3%20instruction%20set,they%20can%20be%20operated%20upon.
36. The LC-3 - UT Computer Science, accessed June 4, 2025, https://www.cs.utexas.edu/~fussell/courses/cs310h/lectures/Lecture_10-310h.pdf
37. Introduction to Computer Engineering Chapter 5 The LC-3 - CS/ECE 252, accessed June 4, 2025, https://ece252.ece.wisc.edu/ch05_inclass_01_lc3_overview.pdf
38. The LC3 Datapath (Chapter 5, Appendix B,C), accessed June 4, 2025, https://cs2461-2020.github.io/lectures/Datapath.pdf

39. ECE 120: Introduction to Computing How Can We Fetch and Perform a Load or a Store? Break Instruction Processing into Steps, accessed June 4, 2025, https://lumetta.web.engr.illinois.edu/120-S19/slide-copies/098-instruction-processing.pdf
40. complx-tools/liblc3: LC3 Simulator backend written in C++. Supporting the original and 2019 revision of the LC3 - GitHub, accessed June 4, 2025, https://github.com/complx-tools/liblc3
41. lc3-vm/index.lit at master - GitHub, accessed June 4, 2025, https://github.com/justinmeiners/lc3-vm/blob/master/index.lit