UNIVERSIDAD DEL SUR

ASIGNATURA:

DSIC102 - Seminario de Programación de Computadoras

SEMANA 4:

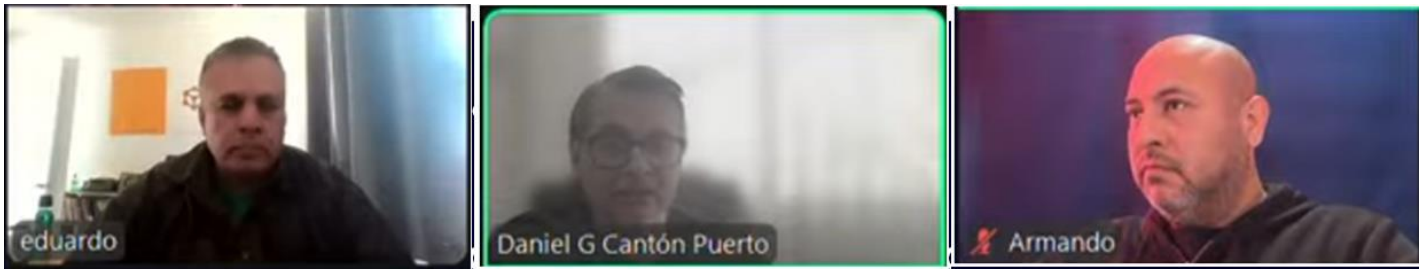
ESTRUCTURAS Y RECURSIÓN AVANZADA

REPORTE FINAL DE CURSO

DOCTORANTES:

Eduardo Méndez Villuendas
Daniel G. Cantón Puerto
Agustín A. Galindo Ozuna

Noviembre 24, 2025



Presentación:

https://www.youtube.com/watch?v=uzuAcVeF9JA

Código en github:

https://github.com/eduardomendezPhD/neumann_assembler

El paradigma de von Neumann: Un Enfoque Didáctico y Epistemológico

Eduardo Mendez-Villuendas
Universidad Autonoma del Estado de Hidalgo
Acacia Lumen Technologies-AI (ALT-AI)
Pachuca de Soto, Hidalgo, Mexico
dremen@outlook.com

Daniel G. Canton-Puerto
Universidad Autonoma de Yucatan
Merida, Yucatan. Mexico
daniel.gcanton@correo.uady.mx

Agustín A. Galindo-Ozuna
Instituto Tecnológico del Valle de Morelia
Morelia, Michoacán. México.
armando_2099@hotmail.com

***Abstract*—Este trabajo presenta el desarrollo de dos simuladores en lenguaje C que reproducen los principios fundamentales de la arquitectura de von Neumann. El primero implementa la suma de dos números con un conjunto mínimo de instrucciones (LOAD, ADD, STORE, HALT), mientras que el segundo amplía el modelo para calcular el factorial de un entero, incorporando saltos condicionales y control de flujo. Ambos programas muestran que con pocas líneas de código es posible representar el funcionamiento esencial de una máquina secuencial: memoria unificada, unidad aritmético-lógica y ciclo fetch–decode–execute. El enfoque propone la simulación como un medio pedagógico para comprender la relación entre hardware y software y reflexionar sobre la vigencia del paradigma de von Neumann frente a los nuevos modelos de arquitecturas contemporáneas.**

***Index Terms*—computational paradigm, von neumann model**

I. INTRODUCTION

La comprensión profunda del cómputo moderno requiere ir más allá del uso de lenguajes de programación o de la manipulación superficial de sistemas digitales; exige descender hasta los principios que articulan el funcionamiento de una computadora desde su núcleo arquitectónico [1]. Este trabajo busca precisamente reconstruir, desde su base más elemental, el proceso lógico y técnico que vincula el hardware, el lenguaje ensamblador y la programación estructurada, mediante el desarrollo de una computadora simulada, escrita en lenguaje C, y con elementos mínimos, que siguen el modelo clásico de von Neumann.

La propuesta no persigue únicamente un ejercicio de programación, sino la recreación del pensamiento arquitectónico que dio origen al cómputo moderno. A través de esta reconstrucción se pretende ilustrar cómo las decisiones conceptuales que dieron forma a los primeros computadores siguen influyendo, hasta hoy, el diseño de procesadores, compiladores, sistemas operativos y arquitecturas de software. El proceso de simulación permite observar la interacción directa entre los niveles físico y lógico de un sistema computacional: desde la ejecución de instrucciones binarias hasta la emergencia de estructuras de control y recursividad propias de lenguajes de alto nivel.

La arquitectura de von Neumann, formulada en 1945 en el célebre primer manuscrito de un Report de la EDVAC [2], propuso por primera vez un modelo de computadora donde los datos y las instrucciones compartían el mismo espacio de memoria. Este paradigma introdujo la noción de un flujo secuencial controlado por una unidad de control (UC) que coordina la unidad aritmético-lógica (ALU), los registros internos y la memoria principal, mediante un ciclo repetitivo de búsqueda, decodificación y ejecución de instrucciones (fetch–decode–execute). A pesar de que las arquitecturas contemporáneas (Harvard modificada, RISC, superscalar, multinúcleo o neuromórficas) se alejan en ciertos aspectos del modelo original, sus fundamentos conceptuales —la secuencialidad, el almacenamiento compartido y la abstracción del programa almacenado— siguen siendo las bases sobre las que se construyen la mayor parte de los sistemas de cómputo actuales.

Desde esta perspectiva, este trabajo se propone como un laboratorio conceptual para explorar niveles de representación: Desde el bit físico, pasando por el ensamblador y la ISA (*Instruction Set Architecture*), hasta llegar al lenguaje estructurado de alto nivel (C). La intención pedagógica es que el participante no solo observe la computadora como una máquina externa, sino que la reconstruya como un modelo conceptual y experimental, comprendiendo cómo cada capa de abstracción se sostiene sobre las anteriores. Tal experiencia, aunque simulada, reproduce el tipo de razonamiento que caracterizó a los pioneros de la informática y que permite reflexionar sobre la tension creativa entre teoría, diseño y práctica. El revisitar estos aspectos historicos no es del todo esteril, pues de cuando en cuando suelen surgir nuevos lenguajes de programacion o paradigmas de computo que integran nuevos desarrollos del hardware dentro de nuevas ontologias. El lenguaje python por ejemplo, es el resultado de volver a traer a los entornos de desarrollo, paradigmas que ya se creian superados con anterioridad. Volver a las bases siempre constituye una oportunidad de encontrar nuevas estrategias para el desarrollo de alternativas tecnologicas.

Un propósito alterno que buscamos con este documento es el de extender nuestra discusión al computo de sistemas que se apartan del paradigma establecido por las arquitecturas de von Neumann, por lo cual discutiremos posibles áreas de oportunidad que emergen como resultado de las nuevas tendencias en el computo moderno: desde Inteligencia Artificial, a redes distribuidas, y la posible revolución tecnológica que

plantean las necesidades de desarrollo del cómputo cuántico y los dispositivos de computo que demandan aplicaciones fuera de lo tradicional.

A. Motivación y propósito

El origen de este proyecto reside en una necesidad formativa fundamental: recuperar el sentido de totalidad en la comprensión del cómputo. En el contexto contemporáneo, la enseñanza de la programación suele fragmentarse entre niveles desconectados —hardware, sistemas, lenguajes, algoritmos—, lo que impide al estudiante percibir la continuidad del proceso computacional. Esta iniciativa busca restaurar la continuidad cognitiva y técnica que unifica el ciclo hardware–software–abstracción, mostrando cómo las decisiones de bajo nivel (como la codificación de instrucciones) determinan los límites y posibilidades del nivel superior (la expresión algorítmica).

El uso de C como lenguaje base no es casual. C encarna la frontera entre el control directo del hardware y la programación estructurada moderna. Es suficientemente abstracto para permitir modularidad, pero lo bastante cercano al lenguaje máquina para manipular registros y direcciones de memoria de manera explícita. Por ello, la simulación de una CPU en C no solo resulta didáctica, sino epistemológicamente coherente: C se convierte en el lenguaje de la máquina que, a su vez, ejecuta programas en un lenguaje aún más bajo, creado por el propio usuario.

El resultado final esperado —una computadora funcional, un lenguaje ensamblador propio, un traductor y un generador de código en C— reproduce el proceso histórico del desarrollo computacional: de la máquina física al lenguaje simbólico y de ahí a la programación estructurada. Al seguir este recorrido, los participantes no solo adquieren competencia técnica, sino que desarrollan una conciencia arquitectónica, una forma de pensar el cómputo como un sistema jerárquico de representaciones interdependientes.

Es imprescindible volver a las raíces del control del cómputo a nivel circuito, sobre todo en los tiempos actuales en que han surgido lenguages de programación de alto nivel, los cuales, aun cuando permitan el rápido desarrollo de soluciones de software, tal vez no lo hagan con la eficiencia óptima, por depender de capas de abstracción que nos separan del control básico a nivel máquina.

B. El modelo de von Neumann y sus derivaciones: Contexto teórico

El modelo de von Neumann no fue solo un diseño técnico, sino un cambio epistemológico. En su núcleo reside la idea de que el programa —un conjunto de instrucciones— puede tratarse como datos. Esta noción permitió la creación de máquinas reprogramables, capaces de resolver problemas generales mediante la simple modificación del contenido de la memoria. A diferencia de las máquinas de propósito fijo, la arquitectura de von Neumann introdujo la flexibilidad que daría lugar al modelo de computación universal, propuesto por Alan Turing [17].

Los modelos desarrollados se articulan alrededor de cinco componentes esenciales [2]:

i **Unidad de Control (UC)**: dirige la secuencia de operaciones y coordina los flujos de datos e instrucciones.
ii **Unidad Aritmético-Lógica (ALU)**: ejecuta operaciones elementales de cálculo y comparación.
iii **Registros**: memoria interna de alta velocidad para almacenar temporalmente operandos, resultados y direcciones.
iv **Memoria principal**: almacena tanto instrucciones como datos, accedidos de forma secuencial o directa.
v **Dispositivos de Entrada/Salida (E/S)**: permiten la comunicación entre el sistema y su entorno.

El ciclo de instrucción que articula estos componentes, por otra parte, se define por tres etapas:

i **Fetch (búsqueda)**: la UC obtiene la instrucción ubicada en la dirección señalada por el contador de programa (PC).
ii **Decode (decodificación)**: la instrucción se interpreta para determinar la operación y los operandos implicados.
iii **Execute (ejecución)**: la ALU realiza la operación y actualiza registros o memoria según sea necesario.

Aunque este proceso parece trivial, constituye la esencia de todo sistema computacional. Cada evolución tecnológica —desde los microprocesadores RISC hasta las GPUs o TPUs— puede entenderse como una variación o extensión de este esquema original. Incluso en paradigmas emergentes, como la computación *neuromórfica* o la computación en memoria, el ciclo lógico de control y ejecución mantiene una analogía con la estructura de von Neumann, lo que ha perpetuado la vigencia conceptual del modelo [3].

C. Alcances y delimitación

El presente estudio se inscribe en el ámbito de la modelación arquitectónica del cómputo, con un enfoque experimental centrado en la reproducción funcional de una máquina von Neumann mediante simulación en lenguaje C. No se trata de una emulación de hardware real ni de un intento de replicar arquitecturas comerciales, sino de un ejercicio de abstracción computacional controlada, diseñado para analizar, con precisión y transparencia, las interdependencias entre los niveles fundamentales del procesamiento digital.

En este estudio se aborda la arquitectura computacional como un sistema formal, compuesto por un conjunto finito de elementos —memoria, unidad aritmético-lógica, registros y unidad de control— articulados por un flujo de instrucciones codificadas. El primer simulador implementado se limita intencionalmente a un conjunto mínimo de operaciones (por ejemplo, **LOAD, ADD, STORE y HALT**), suficientes para demostrar la coherencia del ciclo de instrucción (fetch–decode–execute), mientras que el segundo algoritmo ejemplifica la solucion de un problema con complejidad creciente, con el calculo del factorial de un numero entero, por lo cual cubrimos desde operaciones aritméticas básicas (suma de dos numeros) hasta rutinas estructuradas y recursivas.

La restricción funcional a solamente dos algoritmos, no constituye una limitación metodológica, sino una decisión epistemológica: reproducir modelos elementales permite observar con mayor claridad las propiedades universales del proceso computacional, sin el ruido añadido de optimizaciones industriales o arquitecturas híbridas. El estudio, en consecuencia, se orienta hacia la esencia operativa del cómputo, entendida como la traducción de un conjunto de símbolos en acciones lógicas ejecutables.

El lenguaje C actúa aquí como un medio intermedio entre la formalización teórica y la observación experimental. Su cercanía al hardware —a través del control explícito de memoria y punteros— lo convierte en una herramienta idónea para exponer la continuidad entre las operaciones de bajo nivel y las estructuras de control de alto nivel. Así, el entorno de simulación en C no solo sirve como instrumento de validación técnica, sino también como un espacio conceptual en el que se articulan simultáneamente la máquina, el lenguaje y la semántica del programa.

En cuanto a sus límites, el estudio no busca competir con modelos de rendimiento real ni incorporar elementos avanzados como canalización, interrupciones o paralelismo. Tampoco pretende sustituir el análisis de arquitecturas contemporáneas, sino ofrecer un marco analítico y experimental para reflexionar sobre la génesis del cómputo secuencial y su vigencia como paradigma de organización simbólica. La simulación se considera, por tanto, una plataforma para la reflexión teórica y no una herramienta de ingeniería aplicada.

Dentro de este marco, el alcance del trabajo puede resumirse como la demostración de la suficiencia funcional del modelo de von Neumann, tanto en términos de ejecución programática como de coherencia conceptual. Al integrar en un mismo entorno la representación binaria, la estructura de control y la generación de programas estructurados, la simulación busca poner en evidencia que la arquitectura clásica continúa siendo un instrumento válido para pensar el cómputo moderno y sus posibles transformaciones.

D. Relevancia del estudio de los modelos von Neumann

El desarrollo de una computadora simulada en lenguaje C, inspirada en el modelo de von Neumann, representa una contribución significativa tanto para la comprensión epistemológica del cómputo como para el análisis contemporáneo de sus fundamentos arquitectónicos. Lejos de limitarse a un ejercicio de implementación, el trabajo propone una relectura experimental del paradigma clásico, concebida como un laboratorio conceptual en el que se examina la continuidad entre los niveles físico, lógico y simbólico de la máquina universal.

Desde una perspectiva teórica, las simulaciones aquí presentadas permiten explorar cómo las abstracciones que sustentan el procesamiento secuencial —memoria compartida, unidad de control y flujo lineal de instrucciones— pueden reproducirse mediante estructuras programáticas de alto nivel sin perder su carácter estructural. Este enfoque revela la persistencia del modelo de von Neumann como arquetipo operativo que todavía organiza la mayor parte de las arquitecturas contemporáneas, incluso aquellas que intentan superarlo mediante paralelismo masivo o cómputo en memoria.

En el plano técnico, la investigación muestra la posibilidad de reconstruir un ciclo completo de procesamiento —desde la codificación binaria de instrucciones hasta la ejecución de algoritmos recursivos— dentro de un entorno controlado. La implementación en C permite observar directamente la correspondencia entre los componentes lógicos (ALU, registros, memoria) y las estructuras de control del lenguaje, haciendo visible la frontera difusa entre hardware simulado y software estructurado. Este resultado subraya la capacidad del lenguaje C para funcionar como un medio de modelado arquitectónico y no solo como un instrumento de programación.

Finalmente, desde una perspectiva epistemológica y formativa, el estudio sugiere que la reconstrucción del modelo de von Neumann posee un valor que trasciende su utilidad histórica. La experiencia de construir una máquina desde sus principios fundacionales obliga a reconsiderar la relación entre abstracción y materialidad, así como entre representación simbólica y operación física. En un momento en que la inteligencia artificial, la computación distribuida y los modelos neuromórficos parecen desplazar los fundamentos secuenciales del cómputo, volver a simular una máquina elemental ofrece una oportunidad privilegiada para interrogar el sentido y los límites de la arquitectura clásica.

Así, la contribución central de este trabajo radica en demostrar que el modelo de von Neumann sigue siendo no solo un punto de partida histórico, sino también un instrumento hermenéutico y experimental capaz de articular los distintos niveles del conocimiento computacional: desde el bit hasta el proceso lógico de control, desde la estructura física hasta la semántica algorítmica.

II. Más allá del paradigma de von Neumann

A. Expansion Lógica y Matemática

En 1928 publicó von Neumann su primer trabajo sobre Teoría de Juegos, donde aparece el teorema del minimax, germen del libro "Teoría de Juegos y comportamiento económico", que fue publicado en 1944 con Morgestern y revolucionó la Economía. En 1953 Fréchet (1878-1973) escribió una nota en inglés, donde daba cuenta de los trabajos de Borel, anteriores a los de von Neumann, sobre la Teoría de Juegos, con una valoración muy positiva, considerando a Borel como el iniciador de la Teoría de los Juegos. Esa nota y una réplica de von Neumann fueron publicadas ese mismo año en Econométrica. Hilbert, en su conferencia del Congreso Mundial de Matemáticos de 1900, enunció sus famosos 23 problemas que guiaron gran parte de la investigación matemática del pasado siglo. El sexto problema preguntaba por la posibilidad de la construcción de un sistema axiomático para la Física. von Neumann, muy influido por Hilbert, se apasionó por este problema y, en 1927, en colaboración con Hilbert y Nardheim, publicó un artículo "Sobre los fundamentos de la Mecánica Cuántica". Luego produjo otros artículos de Mecánica Cuántica y en 1932 publicó el libro

"Los fundamentos matemáticos de la Mecánica Cuántica", texto muy abstracto y elegante, pero difícil y sin aplicaciones a problemas prácticos. Utiliza el espacio abstracto de Hilbert, cuya axiomática formula. Hace una construcción deductiva de la teoría de espacios de Hilbert y utiliza la teoría espectral de los operadores en el espacio de Hilbert. Trata el problema de la medida y su interpretación estadística. Hace un estudio crítico de los formalismos usados por Heisenberg y Schrödinger, y con un isomorfismo entre dos espacios de Hilbert prueba la equivalencia entre la Mecánica de Matrices del primero y la Mecánica Ondulatoria del segundo, lo que había sido obtenido por Schrödinger por método distinto. Señala los inconvenientes del uso de la delta de Dirac, que considera poco rigurosa y cree que puede llevar a contradicciones, por lo que no incluye las ecuaciones de Dirac.

Desde el final de la primera guerra mundial hasta el inicio de la década de 1930 la lógica dominante era la lógica de orden superior de la teoría de tipos de Russell, que tenía por objeto eliminar las paradojas de la Teoría de Conjuntos. La teoría de tipos simple está desarrollada en la obra Principia Matematica [16] de Betrand Russell y Alfred Whitehead, y no es propia-mente un sistema formal bajo el punto de vista for-malista de David Hilbert, quien requería que el sistema fuese independiente de cualquier representación. La versión formal del sistema de tipos simple se debe a Kart Gödel y Alfred Tarski, quienes reformularon los axiomas lógicos de Russell y Whitehead.

Un sistema formal alternativo al de la teoría de tipos fue introducido por Ernst Zermelo en 1908, pero no fue hasta 1920 —año en que Abraham Fraenkel publicó una versión modificada del sistema de Zermelo— cuando empezó a tenerse en cuenta.

Desde 1923 hasta 1929 von Neumann publicó diversos artículos sobre la teoría de conjuntos. En un artículo de 1927 titulado Zur Hilbertschen Beweis-theorie von Neumann señalaba que:

"El edificio de la matemática clásica es inseguro y está expuesto al asalto de los escépticos en dos puntos: en el concepto de todo y en la noción de conjunto y, aunque las objeciones a todo hayan sido en cierto sentido refutadas —en la década de los veinte— no puede decirse lo mismo del concepto de conjunto".

Para superar estas deficiencias von Neumann desarrolló en este trabajo un sistema formal en seis grupos de axiomas. En los grupos I, II y III presenta un sistema lógico que incluye la aritmética de Peano, sin el principio de inducción, que aparece en el grupo IV de axiomas, dando lugar a las construcciones transfinitas de la matemática clásica.

Al final de la década de los treinta, Paul Bernays y Kurt Gödel desarrollaron una versión simplificada del sistema de von Neumann, llamado sistema de Neumann, Bernays y Gödel, que evita las paradojas lógicas distinguiendo entre conjuntos y clases. Se tiene que todo conjunto es una clase, pero no todas las clases son conjuntos. La noción de conjunto queda reservada La noción de conjunto queda reservada para aquellas clases que son miembros de otras clases. Introducen el conjunto Ω de ordinales propios, donde por inducción transfinita prueban la existencia de una única función

$$\psi : \Omega \mapsto \Omega \tag{1}$$

definida por:

1) $\psi(0) = 0$
2) $\psi(\gamma + 1) = 2^{\psi(\gamma)}$
3) $\psi(\xi) = U\{\psi(\gamma) : 0 \leq \gamma \leq \xi\}$

La function ψ se denomina función de von Neumann. La clase $\{\psi(\alpha) : \alpha \in \Omega\}$ define el universo Ω conjuntista, estructurado jerárquicamente de modo que:

1) $\psi(\alpha) < \psi(\beta)$ siempre que $\alpha < \beta$
2) Cada conjunto es un elemento de algún universo Ω y, por consiguiente, de todos los universos superiores $\psi(\beta)$.

B. Aplicaciones Técnicas

La arquitectura no von Neumann, dentro del espectro de la computación, proporciona una base idónea, especialmente para gestionar los requisitos computacionales de las tareas con uso intensivo de datos, lo que permite la ejecución eficiente y efectiva de aplicaciones basadas en datos. Esta sección explora las tres clases más extendidas de aplicaciones modernas basadas en datos. Se analiza su relación con las arquitecturas no von Neumann en términos de rendimiento computacional y campos de aplicación. Asimismo, se resume la relación entre las clases de aplicaciones y sus requisitos característicos, presentando las arquitecturas clásicas no von Neumann en la Tabla I.

En el aprendizaje automático (AA), los modelos se entrenan para reconocer características de los datos de entrada, por ejemplo, la clasificación de imágenes y el reconocimiento de objetos. Han sido una aplicación clave para arquitecturas especializadas, como los multiprocesadores de flujo, durante años debido a su mayor potencia de procesamiento en comparación con las unidades de procesamiento de propósito general [11]. Para aumentar la velocidad de procesamiento de las aplicaciones de aprendizaje automático en procesadores de flujo, las unidades de procesamiento de punto flotante de 32 y 64 bits se reemplazan por unidades de menor precisión de 16 y 8 bits, como las TPU. Estas optimizaciones computacionales han desplazado el cuello de botella del rendimiento hacia la ruta de datos, un problema tradicional de las arquitecturas de von Neumann. Arquitecturas más radicales, como la computación neuromórfica y la computación cuántica [13], pueden revolucionar la forma en que procesamos y almacenamos la información para el aprendizaje automático.

Las primeras pueden procesar redes neuronales artificiales con gran eficiencia. Cabe destacar que el aprendizaje profundo, en forma de redes neuronales de impulsos [12], es especialmente adecuado para el hardware neuromórfico. Las segundas, es decir, la computación cuántica, también pueden ofrecer una aceleración significativa gracias al paralelismo cuántico [13].

Las arquitecturas emergentes no von Neumann pueden acelerar aplicaciones científicas con altos requisitos de computación y rendimiento. Los procesadores SPARC y RISC-V permiten optimizar el movimiento y el procesamiento de

TABLE I
RESUMEN DE LAS CLASES DE APLICACIONES IDENTIFICADAS Y SU IDONEIDAD PARA EXPLOTAR DE MANERA EFICIENTE LAS ARQUITECTURAS NO CLÁSICAS (NO VON NEUMANN) DISPONIBLES EN EL CONTINUO DE LA COMPUTACIÓN.

Clase de aplicación	Quantum	Neuromorphic
Machine Learning	✓ Qubits, superposición	✓ SNN-based, aprendizaje por eventos
Vectorización / Paralelismo	✓ Entrelazamiento cuántico, codificación en amplitudes	✓ Procesamiento paralelo basado en picos
Computación científica	✓ Simulación cuántica de sistemas físicos	—
Computación general	Paralelismo cuántico	✓ Acoplamiento memoria–cómputo
Analítica de Big Data	✓ Muestreo cuántico, estimación por amplitud	✓ Acceso localizado a memoria, inferencia SNN

datos, minimizar la latencia y maximizar el rendimiento en simulaciones numéricas complejas. De forma similar, las TPU se pueden utilizar en aplicaciones científicas para acelerar las operaciones matriciales en aprendizaje profundo y entrenar grandes redes neuronales.

Las arquitecturas cuánticas no clásicas no von Neumann también se utilizan en aplicaciones científicas, gracias a la aceleración teórica demostrada para diversos problemas científicos y al modelado nativo de muchos fenómenos científicos en programas cuánticos [13]. Técnicas cuánticas como la inversión de matrices cuánticas [14] permiten resolver sistemas de ecuaciones lineales y realizar operaciones de álgebra lineal mucho más rápido que las computadoras clásicas. La computación cuántica también ha demostrado su potencial como acelerador y para modelar procesos complejos en dinámica molecular y ciencia de materiales. Entre los ejemplos se incluyen la aceleración del cálculo de matrices de autovalores y distancias euclidianas en flujos de trabajo de dinámica molecular, el modelado de propiedades cuánticas de partículas microscópicas, la simulación molecular y la agilización del proceso de descubrimiento en el diseño de fármacos [15].

El análisis de macrodatos implica la recopilación, el almacenamiento, el procesamiento y el análisis de grandes volúmenes de datos para extraer información valiosa, identificar patrones y tomar decisiones basadas en datos en diversas industrias.

Las aplicaciones de macrodatos pueden beneficiarse del uso de arquitecturas no von Neumann para acelerar el análisis de datos. Recientemente, se han propuesto nuevos conceptos para apoyar el análisis de macrodatos en sistemas robóticos que utilizan procesadores neuromórficos para respaldar el proceso de toma de decisiones de los robots mientras realizan análisis de gran volumen de datos en la nube.

En general, se está explorando la computación neuromórfica para mejorar el análisis de macrodatos en operaciones críticas en tiempo real, especialmente en medicina e industria. Específicamente, el análisis basado en analogías reduce significativamente la complejidad computacional de las aplicaciones de macrodatos al disminuir las dimensiones espaciales y temporales de los algoritmos. Además, los multiprocesadores de flujo continuo y las unidades gráficas de propósito general se han utilizado ampliamente desde principios de siglo, especialmente en centros de computación de alto rendimiento, para permitir el análisis rápido de grandes volúmenes de datos [16]. Asimismo, se han empleado arquitecturas clásicas y especializadas, como SPARC, TPU y otras lógicas programables matriciales específicas para cada aplicación, para dar soporte al análisis de grandes volúmenes de datos mediante la optimización del hardware en función de las características específicas de los flujos de datos de entrada [16].

La inclusión del trasfondo matemático y lógico asociado a la obra de von Neumann no constituye un desvío histórico, sino un componente epistemológico esencial para comprender por qué el paradigma de arquitectura que lleva su nombre ha perdurado en la computación contemporánea. La noción de programa almacenado, la formalización del cálculo secuencial y la organización jerárquica de los estados de un sistema derivan directamente de los desarrollos lógico-matemáticos que precedieron a la arquitectura digital. Revisar estos fundamentos permite establecer que un simulador de CPU no es únicamente un ejercicio técnico, sino la materialización operativa de una filosofía del cálculo: una traducción de conceptos abstractos —conjuntos, funciones, estados, operaciones— hacia mecanismos ejecutables. Así, el recorrido histórico-matemático proporciona el andamiaje conceptual que hace inteligible la implementación moderna del modelo de von Neumann, y justifica su estudio como parte integral de un proyecto de simulación arquitectónica.

III. SIMULACIONES COMO HERRAMIENTA DIDÁCTICA

La arquitectura von Neumann, formulada a mediados del siglo XX, constituye no solo el punto de partida histórico del cómputo digital, sino también el fundamento conceptual sobre el cual se erige la mayor parte de las tecnologías contemporáneas. Comprender su funcionamiento implica penetrar en el núcleo de una idea que transformó radicalmente la relación entre el pensamiento lógico y la materia física: la posibilidad de representar operaciones mentales mediante dispositivos mecánicos y eléctricos organizados bajo un principio formal.

Sin embargo, la enseñanza y la práctica actuales de la informática han tendido a fragmentar ese principio. El desarrollo acelerado de capas de abstracción —sistemas operativos, lenguajes de alto nivel, entornos de ejecución, redes neuronales y arquitecturas distribuidas— ha producido una distancia cada vez mayor entre el usuario o programador contemporáneo y las bases arquitectónicas que sostienen el procesamiento. En consecuencia, se ha perdido en muchos casos la percepción del cómputo como un sistema jerárquico coherente, donde cada nivel de representación conserva una relación estructural con los anteriores.

La simulación de una CPU desde cero constituye un medio idóneo para restablecer esa continuidad perdida. A través de

ella, se puede reconstruir la cadena causal que conecta los elementos más simples —el bit, la operación lógica elemental, la dirección de memoria— con las expresiones algorítmicas y simbólicas que conforman el software. Tal reconstrucción no solo tiene un valor didáctico: posee una profunda dimensión epistemológica. Simular una CPU es, en cierto sentido, recrear la génesis del pensamiento computacional, no como repetición histórica, sino como experiencia contemporánea de comprensión integral [4].

A. La simulación como método epistemológico

La simulación de una CPU no debe entenderse únicamente como una técnica de modelado o una herramienta pedagógica, sino como un método de conocimiento. A diferencia de la mera observación o el análisis teórico, la simulación involucra la reconstrucción activa de un sistema; es una forma de experimentación conceptual que traduce ideas en funcionamiento. Cuando un investigador diseña la memoria, los registros y la unidad de control de una máquina simulada, no solo escribe código: formaliza relaciones lógicas que, al ejecutarse, producen evidencia empírica del comportamiento arquitectónico.

Desde esta perspectiva, la simulación actúa como un espejo del razonamiento humano: permite observar cómo los principios abstractos —aritmética binaria, lógica booleana, control secuencial— se materializan en estructuras operativas. La frontera entre teoría y práctica se disuelve. El código se convierte en un objeto de observación científica tanto como en un instrumento experimental.

De este modo, la simulación de una CPU constituye una vía de acceso privilegiada al pensamiento arquitectónico clásico, en el sentido que este pensamiento no se limita a describir el hardware, sino que explora la estructura del pensamiento mismo: cómo una operación puede ser descompuesta en pasos finitos, cómo el tiempo se discretiza en ciclos, cómo la memoria se convierte en extensión de la mente. Comprender una CPU simulada es comprender un modelo formal del razonamiento humano expresado en términos de energía, señales y símbolos.

B. La continuidad entre arquitectura y abstracción

Uno de los aspectos más notables del cómputo moderno es la aparente separación entre hardware y software. En la práctica, ambos constituyen manifestaciones complementarias de una misma lógica estructural. La simulación de una CPU revela con claridad esta continuidad: cada instrucción ejecutada en el nivel de máquina puede rastrearse hasta estructuras de control en lenguaje C, y estas, a su vez, pueden expandirse hasta los conceptos de recursión, modularidad o encapsulado propios de la programación estructurada.

Esta continuidad constituye el núcleo del pensamiento arquitectónico clásico, cuya premisa fundamental es que el todo puede comprenderse a partir de sus partes sin perder la unidad del sistema. Al diseñar una máquina simulada, el investigador reproduce el proceso intelectual que acompañó a la invención de las primeras computadoras: identificar funciones básicas, modularizarlas y articularlas mediante flujos de información coherentes.

En este contexto, la CPU no es solo un objeto técnico, sino una metáfora del pensamiento sistemático. Cada componente —la unidad aritmético-lógica, la memoria, los buses de datos— corresponde a una función cognitiva: cálculo, memoria, comunicación. La simulación pone de relieve esta analogía estructural, permitiendo observar cómo el razonamiento humano se traduce en operaciones formales y cómo estas, una vez automatizadas, devuelven conocimiento sobre el propio acto de pensar.

C. Implicaciones para la comprensión del cómputo contemporáneo

El interés por reconstruir un modelo de von Neumann mediante simulación no surge de una nostalgia por la computación clásica, sino de una necesidad analítica frente a los desafíos actuales. Paradigmas emergentes como el cómputo neuromórfico, la inteligencia artificial distribuida, o la computación cuántica, aunque difieren en implementación, conservan una herencia conceptual que remite a la secuencialidad y al control jerárquico del modelo original.

Por ejemplo, incluso en redes neuronales profundas, la propagación hacia adelante y hacia atrás responde a una lógica de control que podría interpretarse como una extensión no lineal del ciclo fetch–decode–execute. En los sistemas cuánticos, los algoritmos de compuerta y medición siguen la misma idea de un programa almacenado, aunque su soporte físico sea radicalmente distinto [7]. En ambos casos, la comprensión de las arquitecturas emergentes se ve enriquecida cuando se domina el arquetipo de von Neumann, pues este provee el lenguaje conceptual para describir relaciones de control, flujo de información y causalidad operativa.

La simulación de una CPU, por tanto, no es un ejercicio obsoleto, sino un acto de reanclaje epistemológico. Permite recordar que toda forma de cómputo —por paralela, distribuida o estocástica que sea— requiere una definición precisa de estados, transiciones y reglas de transformación. Estos principios, formulados en los albores de la informática, siguen constituyendo la gramática profunda de la tecnología digital.

D. Perspectiva educativa y cognitiva

Desde el punto de vista de la educación en ciencias de la computación, la simulación de una CPU representa una herramienta excepcional para la formación del pensamiento lógico, estructural y analítico. A diferencia del aprendizaje basado exclusivamente en lenguajes de alto nivel, el diseño de una máquina simulada obliga al estudiante o investigador a enfrentarse con las condiciones materiales del cálculo: la limitación de la memoria, el manejo de direcciones, la gestión del flujo de control y el tiempo de ejecución.

Esta experiencia concreta refuerza la comprensión de principios que a menudo se enseñan de manera abstracta. La noción de variable, por ejemplo, adquiere un nuevo significado cuando se entiende como un espacio físico de memoria direccionable; el concepto de bucle se revela como una secuencia de saltos

condicionales; y la recursividad puede interpretarse como una manipulación organizada de la pila de ejecución. En otras palabras, la simulación desmitifica la abstracción y restituye la relación entre símbolo y mecanismo.

En un nivel más profundo, el ejercicio fomenta una forma de pensamiento arquitectónico que trasciende lo técnico. Este pensamiento consiste en imaginar sistemas completos, prever sus interacciones internas y anticipar las consecuencias de sus reglas de funcionamiento. La programación deja de ser una tarea instrumental para convertirse en una práctica de diseño lógico, análoga al trabajo del arquitecto que concibe un edificio antes de construirlo. En este sentido, la educación computacional recupera su dimensión humanista: formar individuos capaces de comprender, crear y reflexionar sobre las estructuras que median entre el pensamiento y la materia.

E. La simulación como puente entre teoría y práctica

Uno de los grandes desafíos contemporáneos en la formación científica consiste en conciliar la abstracción teórica con la experiencia empírica. La simulación de una CPU proporciona un terreno donde ambas convergen. Por un lado, se basa en principios formales rigurosos: lógica booleana, álgebra de conmutación, teoría de la información. Por otro, produce resultados observables: ejecuciones, registros, salidas, errores, tiempos de cómputo.

Esta dualidad convierte a la simulación en un puente entre lo conceptual y lo operativo. En lugar de estudiar la arquitectura de computadoras como un conjunto de definiciones estáticas, el investigador la vive como un sistema dinámico en funcionamiento. Cada ciclo de ejecución se convierte en un experimento, cada modificación del código en una hipótesis verificable.

Tal enfoque aproxima la enseñanza de la computación a los métodos de las ciencias naturales, donde la teoría y la experimentación se retroalimentan. Simular una CPU es, así, una forma de practicar una epistemología experimental del cómputo, en la que el conocimiento surge de la interacción entre modelo, observación y reflexión crítica.

F. Hacia una pedagogía del pensamiento arquitectónico

En el marco más amplio de la educación superior y la investigación avanzada, este tipo de simulaciones contribuye a construir una pedagogía del pensamiento arquitectónico. Dicha pedagogía no se orienta únicamente a la adquisición de competencias técnicas, sino al desarrollo de una mirada sistémica, capaz de comprender las relaciones entre niveles, capas y dominios del saber computacional.

La construcción de una máquina simulada exige una secuencia de decisiones interdependientes: definir la representación de la memoria, diseñar el formato de las instrucciones, establecer la semántica de la ALU, diseñar la unidad de control, y finalmente, conectar todo en un ciclo funcional. Cada una de estas decisiones es, a su modo, una lección sobre el equilibrio entre simplicidad y completitud, eficiencia y legibilidad, estructura y flexibilidad.

El resultado es un proceso formativo que no solo enseña a programar o diseñar, sino a pensar arquitectónicamente: a visualizar sistemas como totalidades coherentes, a reconocer sus dependencias internas y a traducir ideas abstractas en estructuras operativas. En este sentido, la simulación de una CPU se convierte en un modelo de aprendizaje integrador, donde la teoría se hace práctica y la práctica se convierte en teoría encarnada.

IV. Fundamentos teóricos

El desarrollo de una computadora simulada inspirada en la arquitectura de von Neumann requiere un análisis detallado de los principios teóricos que sustentan la noción de máquina universal y su evolución conceptual a lo largo de la historia del cómputo. Más allá de su dimensión tecnológica, el modelo de von Neumann representa una estructura epistemológica que define el modo en que la humanidad concibe, organiza y ejecuta procesos de cálculo. En esta sección se abordan los fundamentos que hicieron posible la formulación del modelo, sus componentes esenciales, las variaciones arquitectónicas derivadas de él, y las líneas de pensamiento contemporáneo que hoy le extienden o desafían.

A. Génesis del Modelo von Neumann

La arquitectura de von Neumann tiene su origen en el documento "*First Draft of a Report on the EDVAC*" (1945), redactado por John von Neumann a partir de los trabajos del grupo de investigación en el Moore School of Electrical Engineering de la Universidad de Pensilvania. Este informe sintetizó, por primera vez, los principios de diseño de una computadora digital de propósito general, estableciendo una ruptura definitiva con las máquinas de cálculo mecánicas y electromecánicas previas, como las de Charles Babbage, Alan Turing, o los sistemas de tarjetas perforadas de Hollerith.

von Neumann propuso una idea revolucionaria: Que los datos y las instrucciones pueden almacenarse en la misma memoria. Esta noción —la memoria unificada— transformó la arquitectura computacional, pues convirtió al programa en un objeto manipulable y reprogramable. En lugar de construir una máquina distinta para cada tarea, bastaba modificar el contenido de la memoria para cambiar el comportamiento del sistema.

A partir de esa concepción emerge el paradigma del programa almacenado (stored-program concept), que convirte a la computadora en una máquina simbólica universal, capaz de ejecutar cualquier algoritmo expresable en términos de operaciones lógicas y aritméticas finitas. Esta idea consolidó la unión entre los aportes teóricos de Alan Turing (la máquina universal [17]) y los avances experimentales de la ingeniería electrónica de mediados del siglo XX.

B. Componentes fundamentales de la arquitectura

El modelo de von Neumann se estructura en torno a cinco componentes esenciales, cuya interacción define el flujo de información dentro del sistema computacional:

i **Unidad Aritmético-Lógica (ALU)**: Responsable de ejecutar operaciones matemáticas elementales (suma, resta, multiplicación, comparación) y operaciones lógicas (AND, OR, NOT). En términos conceptuales, la ALU materializa el dominio formal de la lógica booleana, transformando proposiciones en estados eléctricos.

ii **Unidad de Control (UC)**: Coordina las operaciones del sistema y regula el flujo de ejecución. Su función consiste en recuperar las instrucciones de memoria, decodificarlas y activar las señales correspondientes para la ALU y los registros. Es, en esencia, el "órgano ejecutivo" del sistema.

iii **Memoria Principal**: Espacio donde se almacenan tanto los datos como las instrucciones del programa. Está dividida en celdas direccionables que contienen unidades binarias (bits) agrupadas en palabras. En el modelo original, su naturaleza es secuencial y su acceso uniforme.

iv **Registros**: Pequeñas unidades de almacenamiento de alta velocidad ubicadas dentro de la CPU, que sirven para guardar temporalmente operandos, resultados intermedios y direcciones. Destacan el Acumulador (ACC) y el Contador de Programa (PC), que respectivamente contienen los resultados de la ALU y la dirección de la próxima instrucción.

v **Dispositivos de Entrada/Salida (E/S)**: Interfases que permiten la comunicación con el entorno: recepción de datos, presentación de resultados, y control de procesos externos. En el contexto de la simulación, estas interfaces suelen representarse mediante rutinas de lectura y escritura en memoria o consola.

La interacción de estos elementos da lugar al ciclo de instrucción, eje dinámico de toda arquitectura von Neumann.

C. El ciclo de instrucción: fetch–decode–execute (FDE)

El ciclo de instrucción es la secuencia lógica mediante la cual la CPU procesa un programa. En su forma más básica consta de tres fases:

i **Fetch (búsqueda):** La Unidad de Control obtiene la instrucción almacenada en la dirección indicada por el Contador de Programa (PC). Esta instrucción se transfiere al Registro de Instrucción (IR).

ii **Decode (decodificación):** La Unidad de Control interpreta el código de operación (opcode) y determina qué componentes deben activarse.

iii **Execute (ejecución):** Se realiza la operación correspondiente: lectura o escritura en memoria, operación aritmética, salto condicional, etc. Al finalizar, el PC se actualiza y el ciclo se repite.

Este ciclo continuo constituye la esencia del comportamiento secuencial. Su simplicidad aparente encierra una enorme potencia: la posibilidad de generar comportamientos arbitrariamente complejos a partir de operaciones elementales, principio que se conserva en todas las arquitecturas posteriores, desde los microcontroladores hasta los procesadores multinúcleo.

D. El modelo de von Neumann como abstracción universal

Desde un punto de vista teórico, la arquitectura de von Neumann puede entenderse como una implementación práctica de la Máquina Universal de Turing. Ambas comparten el principio de que cualquier cálculo puede descomponerse en una secuencia finita de pasos elementales, ejecutables por una máquina que manipula símbolos bajo un conjunto de reglas.

En este sentido, el modelo no solo es un artefacto ingenieril, sino una formalización de la cognición humana: una estructura capaz de transformar información de acuerdo con un conjunto de leyes lógicas. Su valor reside en haber unificado tres nociones fundamentales —representación, memoria y control— en un sistema finito y determinista.

Este carácter universal explica por qué la arquitectura de von Neumann ha persistido como esquema conceptual durante más de siete décadas. Su simplicidad y coherencia han permitido extenderla, modularla y reinterpretarla en incontables contextos, desde las supercomputadoras hasta los teléfonos móviles y los sistemas embebidos.

E. Evoluciones y variantes arquitectónicas

Aunque el modelo de von Neumann sigue siendo la base de la computación moderna, su implementación práctica ha experimentado transformaciones profundas. Algunas de las más relevantes son:

- **Arquitectura Harvard:** Propone la separación física entre memoria de instrucciones y memoria de datos, con buses independientes. Esta división mejora el rendimiento y evita el llamado cuello de botella de von Neumann, es decir, la limitación impuesta por el acceso secuencial a una única memoria compartida [6].
- **Arquitecturas RISC (Reduced Instruction Set Computer):** Surgen en los años 1980 como respuesta a la complejidad creciente de los conjuntos de instrucciones CISC. En ellas, cada instrucción realiza una operación simple y uniforme, optimizada para ejecución rápida en hardware. Aunque difieren en implementación, conservan la estructura de control secuencial de von Neumann [7].
- **Arquitecturas superscalar y multinúcleo:** Introducen la ejecución paralela de múltiples instrucciones mediante duplicación de unidades funcionales. Si bien expanden la idea de secuencialidad, aún dependen del modelo de control central y del principio del programa almacenado [8], [9].
- **Arquitecturas neuromórficas y computación en memoria:** Inspiradas en la estructura neuronal del cerebro, buscan integrar procesamiento y almacenamiento en un mismo elemento físico (por ejemplo, los chips Loihi de Intel). Aunque se presentan como alternativas post–von Neumann, su lógica de control sigue, en muchos casos, heredando los principios de la programación estructurada y la causalidad algorítmica [10].

Estas variantes muestran que el modelo de von Neumann no ha sido reemplazado, sino reinterpretado. Cada avance tecnológico representa un intento de optimizar su ejecución

o de extender su alcance a dominios no lineales, pero ninguno ha abandonado completamente su lógica subyacente.

F. Fundamentos lógicos y matemáticos

La solidez del modelo de von Neumann descansa sobre fundamentos matemáticos precisos:

- **Lógica booleana:** Introducida por George Boole en 1854, permite representar operaciones lógicas mediante valores binarios (0 y 1). Toda operación computacional puede expresarse como una combinación de operadores booleanos (AND, OR, NOT).
- **Álgebra de conmutación:** Base para el diseño de circuitos digitales, donde los estados lógicos se implementan mediante conmutadores eléctricos o transistores.
- **Teoría de la información (Shannon, 1948):** Define la información como magnitud cuantificable y establece los fundamentos de la codificación binaria y la comunicación digital.
- **Teoría de autómatas y máquinas de Turing:** Formaliza los conceptos de estado, transición y computabilidad, demostrando que cualquier proceso lógico puede implementarse mediante una máquina finita determinista.

Estos fundamentos constituyen el marco formal que sustenta toda simulación arquitectónica. La traducción de dichos principios a un lenguaje como C permite reconstruir experimentalmente las propiedades lógicas del cómputo, observando su comportamiento en tiempo real y verificando su correspondencia con la teoría.

G. El paradigma von Neumann en la educación y la investigación contemporánea

En el ámbito académico, el estudio del modelo de von Neumann cumple una función doble: por un lado, introducir los fundamentos históricos y teóricos del cómputo digital; por otro, formar el pensamiento estructural que subyace a la ingeniería de sistemas y al diseño de algoritmos.

La simulación en C de una CPU basada en este modelo ofrece una plataforma para comprender cómo la abstracción matemática se transforma en ejecución algorítmica. En este proceso, conceptos como secuencialidad, control, flujo de datos y recursividad dejan de ser nociones abstractas para manifestarse como operaciones observables.

Además, el modelo de von Neumann sigue siendo una herramienta heurística para explorar los límites de la computación clásica frente a los nuevos paradigmas. Comprender su estructura ayuda a formular preguntas más precisas sobre los desafíos de la computación cuántica, la IA distribuida, o los sistemas adaptativos, todos ellos herederos, en mayor o menor medida, del principio universal del programa almacenado.

H. Perspectiva conceptual: Del control lógico a la cognición artificial

Desde una perspectiva más amplia, el modelo de von Neumann puede interpretarse como una metáfora del pensamiento organizado; una representación formal del modo en que la mente humana estructura, almacena y ejecuta procesos.

- La unidad de control actúa como la atención.
- La memoria como el depósito de experiencia.
- La ALU como el razonamiento, y
- El ciclo de instrucción opera como la secuencia consciente del pensamiento.

Esta analogía, lejos de ser meramente poética, revela la raíz cognitiva del diseño computacional: toda máquina de von Neumann es, en cierto sentido, una materialización del proceso lógico de la mente. Por ello, su estudio y simulación no solo sirven para comprender la tecnología digital, sino para **explorar la relación profunda entre lógica, mente y materia** [18], [19]. El pensamiento arquitectónico que surge de este análisis no se agota en la ingeniería: se proyecta hacia la filosofía del conocimiento y la epistemología de la técnica, donde el cómputo aparece como una forma contemporánea de racionalidad aplicada.

I. Síntesis de los fundamentos teóricos

En síntesis, los fundamentos teóricos del modelo de von Neumann articulan tres dimensiones complementarias:

- **Dimensión formal:** El conjunto de leyes lógicas y matemáticas que definen la posibilidad del cálculo digital.
- **Dimensión estructural:** La organización jerárquica de la máquina en componentes interconectados que ejecutan secuencialmente instrucciones codificadas.
- **Dimensión cognitiva:** La correspondencia entre los procesos de control simbólico y las formas de pensamiento sistemático que caracterizan la inteligencia humana.

Estas tres dimensiones convergen en la simulación arquitectónica, donde el acto de programar una CPU equivale a recrear en miniatura el ciclo completo de la computación: desde la formulación teórica hasta la ejecución empírica, desde la idea hasta la máquina.

V. Simuladores implementados

La decisión de construir dos simuladores consecutivos responde a una intención pedagógica explícita: iniciar con un modelo mínimo que exhiba únicamente la relación esencial entre memoria, ALU y ciclo secuencial, para posteriormente introducir gradualmente mecanismos de control de flujo como saltos condicionales y estructuras iterativas. Este tránsito desde la suma elemental hasta el cálculo del factorial permite que el lector observe cómo la complejidad algorítmica emerge de la composición de instrucciones simples dentro de un mismo marco arquitectónico, reproduciendo el proceso histórico mediante el cual las arquitecturas reales evolucionaron desde operaciones básicas hacia capacidades de programación estructurada.

Desarrollamos un modelo de simulador que materializa los principios del paradigma de von Neumann en lenguaje C, en su mas minima implementación, que suma dos numeros A y B. Despues nos avocamos a la tarea de construir un ejemplo secundario con una subrutina que evalua iterativamente el factorial de un numero entero. Nuestro objetivo no es emular procesadores comercial, sino ofrecer una abstracción funcional que permita observar la interacción entre memoria, unidad

aritmético-lógica (ALU), unidad de control (UC) y registros a través del ciclo clásico de fetch–decode–execute.

La simulación de operaciones reproduce el comportamiento de una CPU secuencial con un espacio de memoria direccionable, un conjunto reducido de instrucciones y un flujo de ejecución determinista. Cada módulo del sistema fue diseñado para mantener correspondencia directa con un componente conceptual del modelo arquitectónico original.

A. Simulador 1: Suma de dos números

1) Estructura general del simulador: El simulador está compuesto por cuatro módulos principales:

i **Memoria principal** (`memory[]`) Representada por un arreglo unidimensional de 256 bytes (`uint8_t memory[256]`). Cada posición almacena un dato o una instrucción de 8 bits. La memoria es monolítica y de acceso uniforme, cumpliendo el principio de almacenamiento compartido de von Neumann.

ii **Registros de CPU**
ACC (Acumulador): almacena resultados intermedios y operandos.
PC (Program Counter): apunta a la dirección de la siguiente instrucción.
IR (Instruction Register): contiene la instrucción actual durante la fase decode.

iii **Unidad Aritmético-Lógica (ALU)** Implementada como un conjunto de funciones que operan sobre el acumulador y operandos tomados de memoria. Soporta las operaciones básicas ADD, SUB, AND, OR, NOT. Estas rutinas son invocadas por la Unidad de Control durante la etapa execute.

iv **Unidad de Control (UC)** Implementada mediante una estructura switch–case que interpreta el opcode del registro IR. La UC orquesta las transiciones de estado del sistema y mantiene la coherencia del ciclo FDE.

2) Conjunto de instrucciones (ISA) mínimas: Para garantizar simplicidad y completitud funcional, se definió una ISA de 8 bits, compuesta por un opcode (1 byte) y un operando (1 byte). En la Tabla IV se muestran las instrucciones que interpreta el simulador. Este conjunto permite ejecutar programas estructurados y recursivos simples, demostrando la suficiencia lógica del modelo.

3) Flujo de ejecución y Pseudocódigo: El flujo operativo del simulador sigue rigurosamente el ciclo fetch–decode–execute:

i. Fetch: La UC lee memory[PC] y lo almacena en IR; incrementa PC.
ii. Decode: Se interpreta el opcode de IR; si la instrucción requiere operando, se obtiene de memory[PC++].
iii. Execute: se invoca la rutina correspondiente (por ejemplo, $ALU_{add}()$, store()), se actualizan registros y memoria, y se retorna al paso 1.

El pseudocodigo de este simulador se muestra a continuacion:

```
procedure CPU_Simulate()
    PC <- 0
    repeat
        IR <- MEM[PC]
        PC <- PC + 1
        switch IR do
         case LOAD:
             addr <- MEM[PC]; PC <- PC + 1
             ACC <- MEM[addr]
         case ADD:
             addr <- MEM[PC]; PC <- PC + 1
             ACC <- ACC + MEM[addr]
         case STORE:
             addr <- MEM[PC]; PC <- PC + 1
             MEM[addr] <- ACC
         case HALT:
             exit
        end switch
    until false
end procedure
```

Listing 1. Simulación del ciclo de CPU

4) Programa de prueba y evidencia funcional: Para verificar la coherencia del simulador se ejecutó un programa mínimo: suma de dos valores almacenados en memoria (ver Tabla II).

Tras la ejecución, `memory[0x0C] = 44`, lo que demuestra la correcta operación del ciclo completo y la interacción entre memoria, ALU y UC. La traza del contador de programa confirmó la secuencia lineal de fetch–decode–execute sin saltos erróneos. En la Fig. 1 se muestra el resultado de la ejecución del simulador utilizando diversos casos de prueba.

```
PS E:\STORAGE_MEGA\2025_UNISUR\class1> .\a.exe
cpu_sum(11, 33) = 44
cpu_sum(200, 100) = 44  (mod 256)
cpu_sum(0, 0) = 0
PS E:\STORAGE_MEGA\2025_UNISUR\class1>
```

Fig. 1. Ejecución del simulador de suma. Los números a sumar son provistos mediante una función. Se pueden modificar en la llamada de la función dentro del bloque main.

5) Correspondencia con el modelo teórico: El simulador implementado preserva una homología directa con los componentes canónicos de la arquitectura de von Neumann. Esta correspondencia asegura que las propiedades estructurales del modelo (memoria unificada, control secuencial, y ejecución determinista) se mantengan visibles y trazables en el código C. En particular, cada módulo del sistema (memoria, registros, ALU y unidad de control) cumple una función isomórfica a su contraparte teórica y participa del ciclo *fetch–decode–execute* sin ambigüedades semánticas.

A continuación se resume la relación entre los elementos conceptuales y su materialización en el simulador, así como la función operativa que desempeñan en el flujo de ejecución (Tabla III).

Esta tabla evidencia que el diseño prioriza la transparencia estructural: cada paso del ciclo y cada transferencia de datos puede rastrearse desde el código hasta el esquema conceptual, facilitando la verificación de correctitud y la discusión sobre

TABLE II
PROGRAMA DE PRUEBA Y EVIDENCIA FUNCIONAL DEL SIMULADOR VON NEUMANN

Dirección (hex)	Contenido (hex)	Significado
$0x00$	$0x01, 0x0A$	`LOAD 11` Carga el valor de la dirección 11 en ACC.
$0x02$	$0x02, 0x0B$	`ADD 33` Suma el valor de la dirección 33 al ACC.
$0x04$	$0x03, 0x0C$	`STORE 44` Guarda el resultado de ACC en la dirección 12.
$0x06$	$0xFF$	`HALT` Finaliza la ejecución del programa.
$0x0A$	$0x05$	Dato 1 = 11 (primer operando).
$0x0B$	$0x07$	Dato 2 = 33 (segundo operando).
$0x0C$	$0x0C$	Resultado esperado: 44 (decimal).

TABLE III
CORRESPONDENCIA ENTRE EL MODELO TEÓRICO DE VON NEUMANN Y LA IMPLEMENTACIÓN EN C

Componente teórico	Implementación en C	Función en el simulador
Memoria unificada (datos/instrucciones)	`uint8_t memory[]`	Almacena programa y datos; espacio direccionable único para *fetch* y operandos.
Unidad de Control (UC)	Bucle principal + `switch` por *opcode*	Orquesta el ciclo *fetch–decode–execute*; gestiona `PC`, lee `IR` y activa rutinas.
ALU	Funciones aritmético-lógicas (`add()`, `sub()`, `and()`, `or()`, `not()`)	Efectúa operaciones sobre `ACC` y operandos tomados de memoria; devuelve resultado a registros/memoria.
Registros de CPU	Variables globales `ACC`, `PC`, `IR`	Mantienen el estado interno de ejecución: acumulación de resultados, dirección de próxima instrucción, instrucción actual.
Programa almacenado	Secuencia codificada en `memory[]` (opcodes + operandos)	Define el comportamiento dinámico de la máquina; posibilita reprogramación al modificar el contenido de memoria.
Ciclo de instrucción	Secuencia en el lazo de ejecución: *fetch* → *decode* → *execute*	Garantiza la semántica secuencial y la causalidad operativa entre instrucciones.

extensiones (p. ej., modos de direccionamiento, saltos condicionales adicionales, o canalización).

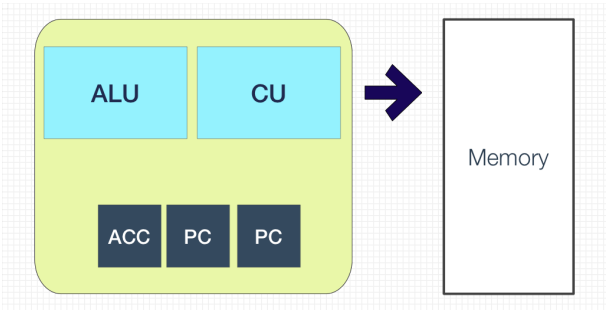


Fig. 2. Diagrama de la arquitectura von Neumann utilizada en las simulaciones.

6) Diagrama lógico: En la Fig. 2, se muestra el flujo de información entre los módulos y la relación temporal del ciclo FDE. Las flechas sólidas representan transferencia de datos; las punteadas, señales de control. La ALU opera sobre el acumulador y devuelve resultados a la memoria por medio del bus de datos.

B. Simulador 2: Factorial de un número entero

A pesar de que el nombre del simulador pareciera ser redundante, debemos notar que la función gamma puede generalizar el factorial a números no-enteros, lo cual no fue nuestro objetivo llevar a cabo.

1) Estructura general: La estructura es igual a la del primer simulador, con:

- **Memoria principal:** `uint8_t memory[256]` (8 bits, direccionamiento plano).
- **Registros:** `ACC` (acumulador), `PC` (program counter), `IR` (instruction register).
- **Ciclo de instrucción:** *fetch* → *decode* → *execute* implementado en `fetch_decode_execute()`.
- **Encapsuladores de lectura:** `fetch_u8()` y `fetch_addr()` con verificación de límites.
- **Separación código/datos:** Código en `0x00...`; datos en `0xC0+` para evitar solapes (diferencia operativa respecto al Simulador 1 original, que solapaba direcciones).

Direcciones de trabajo (datos): N=0xC0, RESULT=0xC1, COUNTER=0xC2, TEMP=0xC3, PART=0xC4, y constantes NEG1=0xF0, ONE=0xF1, ZERO=0xF2.

2) Conjunto de instrucciones (ISA) minimas: Es practicamente similar al primer simulador; Se reutiliza la misma ISA básica de 8 bits (1 byte de opcode + 1 byte de operando cuando aplica) como se muestra en la Tabla IV.

TABLE IV
CONJUNTO MÍNIMO DE INSTRUCCIONES DEL SIMULADOR FACTORIAL (ISA DE 8 BITS).

Instr.	Efecto	Usa banderas
`NOP (0x00)`	No hace nada . (seguro ante huecos de memoria)	—
`LOAD (0x01) addr`	`ACC = MEM[addr].`	—
`ADD (0x02) addr`	`ACC = ACC + MEM[addr]` (mód 256).	—
`STORE (0x03) addr`	`MEM[addr] = ACC.`	—
`JMP (0x04) addr`	`PC = addr.`	—
`JZ (0x05) addr`	Si `ACC == 0` entonces `PC = addr.`	Sí (usa `ACC==0` como Z).
`PRINT (0x06)`	Traza de iteración (solo depuración).	—
`HALT (0xFF)`	Termina ejecución.	—

3) Flujo de ejecución y Pseudocodigo: El flujo de ejecución es el de un algoritmo de factorial por producto iterativo con multiplicación emulada por suma repetida.

```
; Datos en 0xC0+: N, RESULT, COUNTER, TEMP, PART
; constantes: NEG1=0xFF, ONE=0x01, ZERO=0x00
RESULT <- 1
COUNTER <- N

LOOP:
  if COUNTER == 0 goto END
  PART <- 0
  TEMP <- RESULT

  INNER:
    if TEMP == 0 goto INNER_END
    PART <- PART + COUNTER
    TEMP <- TEMP - 1 ; (ADD NEG1)
    goto INNER

  INNER_END:
  RESULT <- PART
  PRINT ; traza: iter, RESULT, COUNTER
  COUNTER <- COUNTER - 1 ; (ADD NEG1)
  goto LOOP
END:
  HALT
```

Listing 2. Simulador 2 de CPU

4) Programa de prueba y evidencia funcional: El programa de prueba se muestra en un apéndice, fuera de este documento, mientras que la salida de la ejecución del codigo se muestra en la Fig. 3

```
PS E:\STORAGE_MEGA\2025_UNISUR\class1> gcc cpu_simulator3.c
PS E:\STORAGE_MEGA\2025_UNISUR\class1> .\a.exe
Program dump [0x00..0x33]:
0x00: 01 F1 03 C1 01 C0 03 C2 01 C2 05 33 01 F2 03 C4
0x10: 01 C1 03 C3 01 C3 05 26 01 C4 02 C2 03 C4 01 C3
0x20: 02 F0 03 C3 04 14 01 C4 03 C1 06 01 C2 02 F0 03
0x30: C2 04 08 FF
Iteration   1 -> RESULT=  6  COUNTER=  6
Iteration   2 -> RESULT= 30  COUNTER=  5
Iteration   3 -> RESULT=120  COUNTER=  4
Iteration   4 -> RESULT=104  COUNTER=  3
Iteration   5 -> RESULT=208  COUNTER=  2
Iteration   6 -> RESULT=208  COUNTER=  1

Final result:
N = 6
factorial(N) mod 256 = 208
PS E:\STORAGE_MEGA\2025_UNISUR\class1>
```

Fig. 3. Ejecución del simulador de factorial.

5) Correspondencia con el modelo teórico: Es igual al Simulador 1, con énfasis en control de flujo, con:

- Memoria unificada (von Neumann): código y datos comparten memory[]; aquí se evita colisión moviendo datos a 0xC0+.
- UC (Unidad de Control): switch(IR) que interpreta opcodes y orquesta saltos (JMP, JZ) y etapas del ciclo FDE.
- ALU: implementada por ADD (suma y "resta" mediante NEG1), y transferencia LOAD/STORE.
- Registros: ACC (operador/resultado), PC (secuenciación), IR (instrucción vigente).
- Ciclo FDE: fetch_u8 $\mapsto$ decode/switch $\mapsto$ execute, con actualización de PC y lectura de operandos.

VI. CARGADORES

Un cargador (loader) es un programa cuya función es tomar un archivo con código máquina y colocarlo en memoria para ejecucion via CPU. En un sistema operativo real, un cargador reserva memoria, mapea secciones, inicializa el contador de programa y finalmente transfiere el control a la primera instrucción del programa.

En esta sección se presenta el diseño del lenguaje máquina (ensamblador) para los CPU de 8 bits simulados, así como un traductor escrito en lenguaje C que convierte el código ensamblador legible por humanos (.asm) en el código binario ejecutable (.mem y .bin).

Se procedió de la forma que a continuación se detalla:

(i) **Definición de la arquitectura (ISA mínima):** Se implementó un conjunto reducido de instrucciones compatibles con los simuladores descritos en la seccion V (LOAD (0x01), ADD (0x02), STORE (0x03), HALT (0xFF)) tal y como se describe en la table II.

- Traductor de dos pasadas:
  - Primera: registra etiquetas y calcula direcciones. El direccionamiento de memoria utilizado es directo ya que el dato se encuentra almacenado en una dirección de memoria codificada en la instrucción.
  - Segunda: genera los bytes correspondientes a cada instrucción.
- Directivas que soporta: .org, .byte, .equ, etiquetas (ETIQUETA:) y comentarios(;).
- Uso: *./assembler.x input.mem output*
- Archivos de salida: *output.mem* $\mapsto$ volcado de texto con bytes en hexadecimal (uno por línea).
  output.bin $\mapsto$ **binario crudo listo para carga directa**, sin saltos de línea, tal cual estarían en RAM.
  output.lst $\mapsto$ listado detallado con direcciones, bytes generados y tabla de símbolos. (ADDR, BYTES, SOURCE). Sirve para depurar, mostrando que bytes genero cada linea, asi como demostrar que el ensamblador valido correctamente las etiquetas, permitiendo por tanto la evaluacion de errores en forma directa, permitiendo revisar la tabla de símbolos (direcciones de A, B, RES, LOOP, etc.).

(ii) **Programas de prueba**
Se desarrollaron dos programas de prueba (*suma.asm* y *factorial.asm*). Estos se muestran en el apendice.

(iii) **Intérprete de programas** (*cpu_loader.c*)
Se diseñó un cargador simple capaz de leer el .mem y ejecutar el ciclo fetch-decode-execute, mostrando el contenido de la memoria al finalizar. Este programa se compila de la siguinte forma:
$gcc cpu_loader.c -o cpu_loader.x

Ahora incluimos la salida para cada uno de los programas. El código fuente se adjunta en los apéndices.

A. Suma de dos números

El programa (*suma.asm*) tiene la estructura siguiente:

```
; suma.asm : suma A + B -> RES
    .org 0x00
    LOAD A
    ADD B
    STORE RES
    HALT

    .org 0x20
    A: .byte 11
    B: .byte 33
    RES: .byte 0
```

Ahora ejecutamos el programa:
assembler.x suma.asm sumaout,
lo cual produce la terna de archivos *sumaout.bin , sumaout.lst, sumaout.mem*.

ADDR	BYTE VAL.	MEANING
0x00	01	LOAD opcode
0x01	20	operand: address 0x20 (A)
0x02	02	ADD opcode
0x03	21	operand: address 0x21 (B)
0x04	03	STORE opcode
0x05	22	operand: address 0x22 (RES)
0x06	FF	HALT
0x07	00	—-
0x08	00	—-
0x09	00	—-
0x0A	00	—-
0x0B	00	—-
0x0C	00	—-
0x0D	00	—-
0x0E	00	—-
0x0F	00	—-
0x10	00	—-
0x11	00	—-
0x12	00	—-
0x13	00	—-
0x14	0B	value A = 11 decimal
0x15	21	value B = 33 decimal
0x16	00	RES = 0 initially

TABLE V
DIRECCIONES, VALOR DE BYTE Y CONTENIDO/TIPO DE OPERACION.

La tabla V muestra un mapa de ocupación de variables en memoria. Se muestra la dirección, el valor, y el significado/utilidad de la instrucción en columnas distintas.

Esta información se encuentra compactada en el archivo de salida *sumaout.bin* que se muestra a continuación:

```
; LISTING FILE
; Source: suma.asm
; Generated: 2025-11-14 15:43:56
; Memory used: 0x23 bytes (0..0x22)

ADDR BYTES SOURCE
---- ---------- ---------------
             .org 0x00
0000 01 20 LOAD A
0002 02 21 ADD B
0004 03 22 STORE RES
0006 FF HALT
             .org 0x20
0020 0B .byte 11
0021 21 .byte 33
0022 00 .byte 0

SYMBOLS (3):
  A = 0x20 ( 32)
  B = 0x21 ( 33)
  RES = 0x22 ( 34)
```

Listing 3. Archivo suma.asm

La figura 4 muestra el resultado impreso en pantalla con valores de entrada en hexadecimal 0B y 21, respectivamente, lo que corresponde en decimal a 11 y 33. La suma de estos valores es 2C hexadecimal, lo cuale equivale a 44. ACC=2C es el valor almacenado en el acumulador (en este caso 44) mientras que PC=07 es el último registro que se leyó del archivo de entrada (se llamó el comando HALT).

```
(base) eduardo@DESKTOP-CS80NOM:/mnt/e/STORAGE_MEGA/2025_
UNISUR/class1/week2/code_mine$ ./cpu_loader sumaout.mem
Memory snapshot (0x20..0x22): 0B 21 2C
ACC=2C PC=07
(base) eduardo@DESKTOP-CS80NOM:/mnt/e/STORAGE_MEGA/2025_
UNISUR/class1/week2/code_mine$
```

Fig. 4. Salida del programa sum, con valores de entrada 11 y 33

B. Factorial

En esta seccion se muestra la implementacion secuencial del programa factorial, con el mismo compilador e interprete empleado para el programa suma.

```
; factorial.asm : calcula 4! = 24 y lo guarda en
      RESULT

      .org 0x00

      ; RESULT = 1
      LOAD ONE
      STORE RESULT

      ; RESULT = RESULT * 2 (doblar)
      LOAD RESULT
      ADD RESULT
      STORE RESULT

      ; RESULT = RESULT * 3
      ; ACC = 0
      LOAD ZERO
      ; ACC = ACC + RESULT + RESULT + RESULT
      ADD RESULT
      ADD RESULT
      ADD RESULT
      STORE RESULT

      ; RESULT = RESULT * 4
      ; ACC = 0
      LOAD ZERO
      ; ACC = RESULT + RESULT + RESULT + RESULT
      ADD RESULT
      ADD RESULT
      ADD RESULT
      ADD RESULT
      STORE RESULT

      HALT

      ; Datos mas lejos para no chocar con
          codigo
      ;
      .org 0x40
ONE: .byte 1
ZERO: .byte 0
RESULT: .byte 0
```

Listing 4. Programa Factorial.asm simple, que calcula el resultado de factorial(4)=4!

El listado 5 muestra un detalle de la salida de el programa *assembler.x*, despues de procesar la implementacion del factorial mediante el comando de proceso
$ assembler.x factorial.asm factorialout
El listado muestra las direcciones, los bytes, y el tipo de dato/instrucción alojado. Este listado es equivalente al del listado de salida para el programa suma

```
; LISTING FILE
; Source: factorial.asm
; Generated: 2025-11-14 16:34:36
; Memory used: 0x43 bytes (0..0x42)

ADDR BYTES SOURCE
---- ------ ---------------
                .org 0x00
0000 01 40 LOAD ONE
0002 03 42 STORE RESULT
0004 01 42 LOAD RESULT
0006 02 42 ADD RESULT
0008 03 42 STORE RESULT
000A 01 41 LOAD ZERO
000C 02 42 ADD RESULT
000E 02 42 ADD RESULT
0010 02 42 ADD RESULT
0012 03 42 STORE RESULT
0014 01 41 LOAD ZERO
0016 02 42 ADD RESULT
0018 02 42 ADD RESULT
001A 02 42 ADD RESULT
001C 02 42 ADD RESULT
001E 03 42 STORE RESULT
0020 FF HALT
                .org 0x40
0040 01 .byte 1
0041 00 .byte 0
0042 00 .byte 0

SYMBOLS (3):
  ONE = 0x40 ( 64)
  RESULT = 0x42 ( 66)
  ZERO = 0x41 ( 65)
```

Listing 5. Archivo factorialout.lst

VII. AMPLIACIONES DEL ISA

El conjunto mínimo de instrucciones empleadas en los dos programas desarrollados es extremadamente simple, sin embargo no permitia la escritura de códigos mas sofisticados. La aplicación de factorial por ejemplo, requiere del uso de ciclos (loops), para que sea elaborado un programa más eficiente, dado que el desarrollo que se muestra requiere de cálculos secuenciales de un solo paso, en vez de una lista de instrucciones recursivas. Debido a esto fue necesario ampliar la ISA, y re-escribir el programa de factoriales, el interprete del codigo, y el programa creado para su ejecucion.

Las capacidades del simulador de CPU desarrollado, en conjunto de instrucciones que el procesador es capaz de reconocer y ejecutar. La ISA actual incluye operaciones como carga, suma, almacenamiento, saltos condicionales simples y finalización. Si bien este conjunto mínimo permite ejecutar programas elementales. La ampliación de la ISA consiste en agregar nuevos mnemónicos, asignarles códigos binarios únicos y definir su comportamiento dentro del ciclo FDE del simulador. Con ello se incrementa la expresividad del lenguaje ensamblador y se reduce la cantidad de instrucciones necesarias para realizar operaciones comunes.

La tabla VI muestra la diferencia entre la implementacion de el interprete de ensamblador *assembler.c* y la version *assembler_v2.c* que incluye el reconocimiento de instrucciones de iteracion.

```
; factorial_ACC.asm : factorial general N! usando bucles
; ISA: LOAD=0x01, ADD=0x02, STORE=0x03, JMP=0x04, JZ=0x05, HALT=0xFF

      .org 0x00

      ; RESULT = 1
      LOAD ONE
      STORE RESULT

      ; COUNTER = N
      LOAD N
      STORE COUNTER

LOOP:
      ; if COUNTER == 0 goto END
      LOAD COUNTER
      JZ END

      ; PART = 0
      LOAD ZERO
      STORE PART

      ; TEMP = COUNTER
      LOAD COUNTER
      STORE TEMP

INNER:
      ; PART = PART + RESULT
      LOAD PART
      ADD RESULT
      STORE PART

      ; TEMP = TEMP - 1 (NEG1 = 0xFF)
      LOAD TEMP
      ADD NEG1
      STORE TEMP

      ; if TEMP == 0 goto INNER_END
      LOAD TEMP
      JZ INNER_END

      ; else goto INNER
      JMP INNER

INNER_END:
      ; RESULT = PART
      LOAD PART
      STORE RESULT

      ; COUNTER = COUNTER - 1
      LOAD COUNTER
      ADD NEG1
      STORE COUNTER

      ; repeat outer loop
      JMP LOOP

END:
      ; Aqui nos aseguramos que el factorial quede en ACC
      LOAD RESULT ; ACC = RESULT (N!)
      HALT

      ; DATA
      .org 0xC0
N: .byte 4 ; valor de N, cambiar para obtener otro factorial
RESULT: .byte 0
COUNTER:.byte 0
TEMP: .byte 0
PART: .byte 0
ONE: .byte 1
ZERO: .byte 0
NEG1: .byte 255 ; 0xFF = -1 en aritmetica de 8 bits
```

Listing 6. Recursive factorial

La fig. 5 muestra un diagrama de flujo con el calculo recursivo del factorial de un numero.

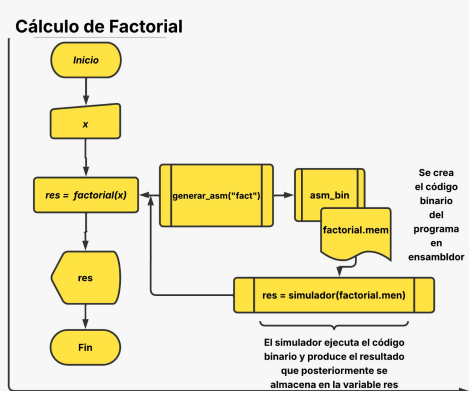


Fig. 5. Diagrama de flujo del cálculo del factorial de un número con el simulador

Al igual que el programa assembler, tambien se agregaron ajustes al programa *cpu_loader.c* para dar lugar a la segunda version *cpu_loader_v2.c*. El software se incluye en los apendices.

```
(base) eduardo@DESKTOP-CS80NOM:/mnt/e/STO
$ ./cpu_loader.x factorialout.mem
Memory snapshot (0x20..0x22): FF 00 00
ACC=18 PC=21
(base) eduardo@DESKTOP-CS80NOM:/mnt/e/STO
$ ./cpu_loader_v2.x factorialBout.mem
Memory snapshot (0x20..0x22): 01 C3 05
ACC=18 PC=35
```

Fig. 6. Comparacion de factoriales de 4, para las dos versiones del programa factorial. En ambos casos, la respuesta por supuesto es 18 hexadecimal, lo cual equivale a 24

VIII. LENGUAJES DE ALTO NIVEL

En general no es recomendable desde luego, producir todo el código como lenguaje ensamblador, o en lenguaje de alto nivel, sino que es necesario encontrar un balance de acuerdo a aspectos de costo/beneficio en donde solamente las subrutinas de mayor uso sean optimizadas al límite, mientras que tareas de uso general, y que no requieren de alta eficiencia, puedan ser implementadas con lenguaje de alto nivel.

El proceso que conecta el código fuente con la ejecución final en la CPU simulada puede resumirse como una tubería secuencial de transformación: el programa escrito en C se procesa primero mediante el módulo `c_to_asm.c`, que genera automáticamente el archivo ensamblador correspondiente. Este archivo es posteriormente traducido a formato binario (`.mem`) por el ensamblador. El cargador (*loader*) toma dicho archivo y construye la imagen de memoria que finalmente es interpretada instrucción por instrucción por el simulador mediante el ciclo *fetch–decode–execute*. En conjunto, este flujo `C` → `C-to-ASM` → `ASM` → `.mem` → `Loader` → `Simulador` constituye una cadena coherente que reproduce, en miniatura, las etapas esenciales de un sistema de compilación real. (See fig.7)

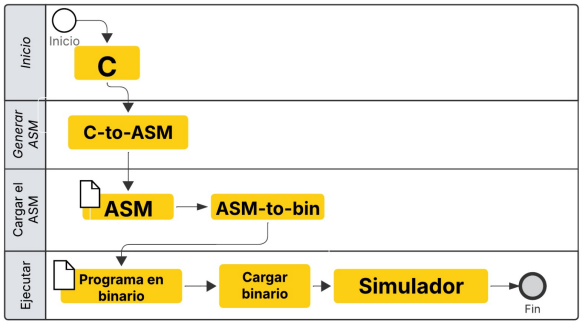


Fig. 7. Diagrama de flujo de codigo a ensamblador.

Un compilador es un traductor que convierte un lenguaje de alto nivel (por ejemplo, C o Python) a lenguaje de bajo nivel (ensamblador o código máquina). Los compiladores de uso profesional tienen múltiples fases: análisis léxico, análisis sintáctico, generación de código intermedio, optimización, y finalmente generación de código objeto. Un ligador (linker) por otra parte, combina múltiples archivos objeto y resuelve direcciones de funciones, variables y referencias externas.

En este proyecto se implementó una mini-etapa de compilación mediante el programa *c_to_asm.c*. Este programa, escrito en C, genera automáticamente archivos .asm utilizando parámetros y estructuras de control del lenguaje C. El código C se convierte así en una representación más cercana al lenguaje máquina de nuestra CPU simulada. En la Fig. 5 se observa el flujo del programa para calcular el factorial de un número, generando el código ensamblador correspondiente que posteriormente pasará a binario y finalmente será interpretado por el simulador.

A. Compilado Básico de C a Ensamblador

Se generó un programa en C que crea los programas en lenguaje ensamblador. Este módulo se puede concebir como un generador de librerías, que toma los parámetros pasados por el usuario y los personaliza para su uso posterior. De esta manera, funcionan los lenguajes de alto nivel, creando una capa de abstracción que evita que el desarrollador tenga que realizar manualmente la comunicación directa con el procesador, permitiendo al desarrollador enfocarse optimizar su solución.

La librería desarrollada en este proyecto tiene como objetivo la generación de código ensamblador (funciones individuales y simples).

B. Lenguaje de Alto Nivel

El uso de una subrutina principal (main2link.c), representa el set de programas escritos en lenguaje de alto nivel que pasaran argumentos al codigo en lenguaje fuente cuando asi se requiera.

La función principal del archivo `main2link.c` es coordinar la ejecución de módulos ensamblados para la CPU simulada de 8 bits. Para ello, se apoya en la función `load_module`, cuyo propósito es cargar en memoria el contenido de un archivo `.mem`. Estos archivos contienen los

TABLE VI
DIFERENCIAS PRINCIPALES ENTRE `assembler.c` Y `assembler_v2.c`.

Aspecto	`assembler.c`	`assembler_v2.c`
ISA soportada	LOAD, ADD, STORE, HALT	**+ JMP, JZ** (salto incondicional y condicional)
Cálculo de tamaño (Pass 1)	Sólo instrucciones de 2 bytes: LOAD, ADD, STORE	**También**: JMP, JZ son de 2 bytes
Emisión de instrucciones (Pass 2)	Genera código únicamente para LOAD/ADD/STORE/HALT	**Agrega emisión para JMP y JZ**
Manejo de símbolos	Igual: resolución básica de etiquetas y números	Idéntico; no cambia
Compatibilidad	Sólo programas lineales sin ciclos	**Permite programas con bucles y condicionales**

bytes en formato hexadecimal generados por el ensamblador. La función abre el archivo correspondiente, limpia la memoria simulada mediante `memset`, y copia secuencialmente cada byte leído en las posiciones 0 a 255 del arreglo `memory`. Esta operación no ejecuta el código; simplemente construye la imagen de memoria que luego interpretará el simulador a través de la instrucción `fetch_decode_execute()`.

El programa solicita al usuario dos valores, `N1` y `N2`, que representan los argumentos para calcular dos factoriales. Para obtener el primer factorial, se carga el módulo ensamblado `factorial.mem`, se coloca el valor `N1` en la dirección `0x20` (variable `N` del módulo), y se ejecuta el simulador mediante `fetch_decode_execute()`. Una vez que el código ensamblado alcanza la instrucción `HALT`, el resultado queda almacenado en la dirección `0x21`. Dicho valor se lee y se guarda en la variable `FACT1`. El mismo procedimiento se repite para `N2`, reutilizando exactamente el mismo módulo ensamblado de factorial.

Para calcular la suma de ambos factoriales se carga el módulo `suma.mem`. Este módulo utiliza las direcciones `0x20` y `0x21` como variables `A` y `B`. El programa escribe en dichas direcciones los valores `FACT1` y `FACT2`, ejecuta nuevamente `fetch_decode_execute()`, y finalmente recupera el resultado almacenado en la dirección `0x22`. De esta forma, la resistencia del sistema modular queda demostrada: un programa en C puede cargar, parametrizar y ejecutar múltiples módulos ensamblados independientes, emulando llamadas a funciones y combinando resultados sin necesidad de extender la ISA ni modificar el simulador subyacente.

C. Linker: Enlace entre Alto y Bajo Nivel

Para que el conjunto de código ensamblador con el código objeto main.o pueda integrarse, se necesita hacer linking, lo cual se automatiza mediante un script en bash que hace lo siguiente (ver código en appendix):

Compila el ensamblador propio desarrollado para la CPU de 8 bits.
Compila la CPU simulada en un objeto (`cpu_sim.o`) para reutilizar la rutina `fetch_decode_execute()` desde C.
Compila `main2link.c` y lo enlaza con el simulador para producir el ejecutable.
Ensambla por separado los módulos `factorial.asm` y `suma.asm`, generando los archivos `factorial.mem` y `suma.mem`.
Ejecuta `./main2link`, el cual:
Carga `factorial.mem`, asigna `N1`, ejecuta el simulador y recupera el valor `RESULT`.
Vuelve a cargar `factorial.mem`, asigna `N2`, ejecuta nuevamente y obtiene el segundo `RESULT`.
Carga `suma.mem`, coloca `FACT1` y `FACT2`, ejecuta la operación y recupera la suma final.

IX. RESULTADOS EXPERIMENTALES Y VALIDACIÓN

El sistema final integra de manera coherente las tres capas desarrolladas:

(i) El simulador de CPU de 8 bits,
(ii) El ensamblador simbólico que produce imágenes `.mem`, y
(iii) Un programa en C que actúa como *driver* de alto nivel para controlar la ejecución completa.

La validación se realizó ejecutando programas aritméticos y un caso recursivo (factorial), verificando que el flujo C $\rightarrow$ ASM $\rightarrow$ BIN $\rightarrow$ simulador produce resultados correctos dentro de las limitaciones propias de una arquitectura de 8 bits. Debido a que la CPU diseñada opera en el rango 0–255, todas las operaciones exhiben aritmética modular y desbordamiento natural, permitiendo observar explícitamente fenómenos de saturación que suelen quedar ocultos en sistemas modernos. Las direcciones, datos y opcodes se representan en hexadecimal: el espacio `00h–FFh` corresponde exactamente al rango decimal 0–255, y valores como `0x2A`, `0xC0` o `0xF3` resultan más legibles porque cada par hexadecimal describe directamente un byte. Esta representación estándar en sistemas de bajo nivel simplifica la inspección de memoria, la verificación del código máquina y la lectura de archivos `.mem`, haciendo más clara y trazable la depuración del simulador.

La fig.8 muestra el programa de suma de factoriales, que llama a las funciones escritas en ensamblador. El conjunto de ejecuciones de la izquierda observa el resultado esperado, mientras que el conjunto de ejecuciones del lado de la derecha muestra un error, explicable debido a que el maximo valor que puede ser almacenado es de 255 decimal. El ultimo numero ”correcto” es $5! = 120$, mientras que para el factorial de 6 tenemos $6! = 720 \mapsto 720‘ \ mod \ 256 = 720 - 2256 = 208$, lo mismo para el factorial de 7, $7! = 5040 \mapsto 5040 \ mod \ 256 = 5040 - 19256 = 176$, etc. Cada celda de memoria es de

1 byte, capaz de representar solo valores entre 00 y FF en hexadecimal. Si un cálculo supera ese rango, el resultado se recorta a los 8 bits inferiores. Es decir, el desbordamiento es una limitación natural de tu arquitectura de 8 bits.

El programa en C pudo invocar correctamente las subrutinas en ensamblador porque ambos utilizan las mismas direcciones fijas de memoria para las variables (como N, RESULT, A, B, etc.). Al mantener estas ubicaciones constantes, el programa en C puede escribir valores distintos antes de cada ejecución, permitiendo que el ensamblador reciba inputs variables sin necesidad de recompilar el código. Este codigo, disponible en github (ver recursos adicionales), posibilita la escritura de ciclos y toda clase de implementaciones de alto nivel, dentro o fuera de el programa escrito en alto nivel.

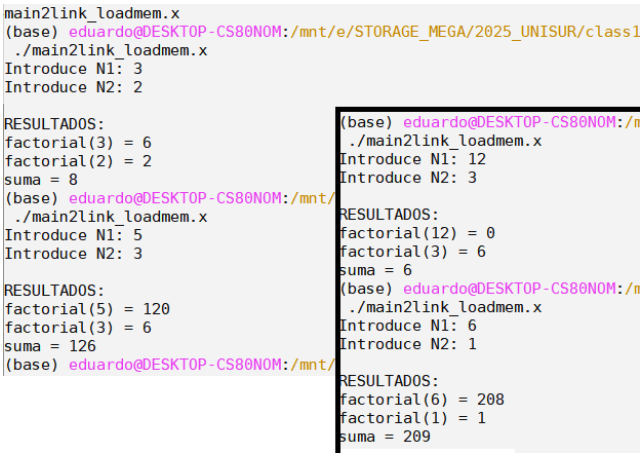


Fig. 8. Ejecución de archivo compuesto con main driver que llama a dos funciones en ensamblador.

El *driver* en C permite dos modalidades de integración: (A) invocar directamente módulos escritos en ensamblador, reutilizando codigo del cargador e intérprete existentes, o (B) cargar imágenes binarias ya ensambladas y ejecutarlas desde C como bloques funcionales independientes. Ambos esquemas demostraron ser operativos y equivalentes en funcionalidad, lo que confirma que el sistema permite combinar código de alto nivel con rutinas de bajo nivel sin requerir un enlazador tradicional.

En conjunto, estos experimentos validan la corrección de la ISA, el funcionamiento del ensamblador y la consistencia del simulador, al tiempo que muestran que el sistema soporta programas completos que mezclan control de alto nivel con operación explícita a nivel de máquina.

X. Discusión

Los dos simuladores desarrollados demuestran que es posible reproducir el comportamiento esencial de un sistema computacional completo utilizando menos de un centenar de líneas en C. El primer modelo, dedicado a la suma de dos números, constituye una representación mínima del ciclo fetch–decode–execute, mientras que el segundo, encargado de calcular el factorial de un entero, amplía el conjunto de instrucciones para incluir saltos condicionales y bucles, evidenciando la emergencia del control de flujo dentro del paradigma de von Neumann.

En conjunto, ambos programas conforman una secuencia pedagógica coherente: el primero ilustra la ejecución lineal y la manipulación básica de datos, y el segundo introduce la iteración y la auto-referencia del programa sobre su propia memoria. Esta progresión permite observar cómo la complejidad algorítmica se apoya en una misma estructura elemental de memoria y registros, reafirmando la unidad del modelo computacional subyacente.En la Tabla VII se puede observar una comparación entre estos simuladores.

El valor formativo de esta aproximación radica en su transparencia estructural: cada instrucción puede seguirse paso a paso, y cada salto o modificación del acumulador se relaciona directamente con el estado lógico del sistema. Este entorno controlado no sólo facilita la exploración experimental del comportamiento del hardware simulado, sino también la reflexión teórica sobre la relación entre código, estructura y ejecución.

En etapas posteriores, ambos simuladores servirán como base para implementar un ensamblador simbólico, con lo cual se completará el tránsito desde la representación binaria hacia el nivel simbólico del software, cerrando el ciclo hardware–software propuesto en el seminario.

Por ultimo, estos ejercicios ponen de manifiesto que incluso las arquitecturas más complejas derivan de principios sorprendentemente simples. Al construir y analizar simuladores mínimos, no sólo se internaliza la lógica fundamental del procesamiento secuencial, sino que desarrolla una comprensión más profunda de cómo las abstracciones de más alto nivel lenguajes, compiladores, ensambladores y sistemas operativos que emergen de estos bloques primitivos.

Esta perspectiva integrada resulta especialmente valiosa en un contexto educativo, donde comprender el “cómo” y el “por qué” detrás de la ejecución permite trascender la mera utilización instrumental de las herramientas. Al hacer explícitas las conexiones entre el comportamiento del simulador, las decisiones de diseño de la ISA y los patrones de ejecución observados, el aprendizaje se transforma en un proceso consciente y fundamentado.

XI. Conclusiones

En síntesis, la reconstrucción de una arquitectura de von Neumann mediante simulación en C se justifica por su doble valor: analítico y formativo. Analítico, porque permite explorar los fundamentos universales del procesamiento digital, identificar sus invariantes y reinterpretarlos en el contexto de las tecnologías emergentes. Formativo, porque ofrece una experiencia de aprendizaje que une abstracción y materialidad, pensamiento y operación, lógica y experimentación.

La Implementación de un ensamblador en C reforzá la importancia de establecer reglas claras en el diseño del conjunto de instrucciones (ISA), ya que cualquier ambigüedad en los mnemónicos, en los campos de las instrucciones o en los tamaños de palabra se refleja directamente en errores durante la traducción o ejecución.

TABLE VII
COMPARACIÓN ENTRE EL SIMULADOR DE SUMA Y EL SIMULADOR DE FACTORIAL (8 BITS).

Aspecto	**Suma de dos números**	**Factorial (N!)**
Núcleo	von Neumann 8-bit, `memory[256]`, `ACC`, `PC`	Igual; además `IR` y separación código/datos
ISA	`LOAD`, `ADD`, `STORE`, `HALT`	Igual + `NOP`, `JMP`, `JZ`, `PRINT`; decremento con `ADD NEG1`
Control de flujo	Secuencial (sin saltos)	Dos bucles (externo e interno) con `JZ`/`JMP`
Memoria (datos)	Direcciones `10, 11, 12` (sin norma explícita)	Datos a partir de `0xC0` (evita colisiones con el código)
Robustez	Sin comprobaciones ni trazas	`fetch_u8/fetch_addr` con checks, `dump_program`, `PRINT` con iteración
Complejidad	O(1) instrucciones	O($N\cdot$ multiplicaciones por suma repetida)
Overflow 8-bit	Posible según operandos	Intrínseco a $N!$: $6! \to 208$, $7! \to 176$, $N \geq 10 \Rightarrow 0 \bmod 256$
Evidencia	Valor final en `memory[12]`	Trazas por iteración + valor final en `RESULT`

Utilizar un conjunto reducido de instrucciones permitió enfocarse en la comprensión acerca del almacenamiento del programa en lenguaje máquina, proporcionando una visión al funcionamiento de los compiladores de lenguajes de alto nivel. Sin embargo, las instrucciones utilizadas en este simulador difieren de las utilizadas en los compiladores comerciales, ya que en este proyecto se utilizaron instrucciones de 8 bits a diferencia de los compiladores actuales que utilizan 64.

Por otra parte, el desarrollo de una capa de abstracción que lleve un programa en ensamblador a código binario, interpretado por el simulador, nos permite entender cómo el procesador ejecuta las instrucciones y el manejo de memoria. Esto es fundamental para el desarrollo de sistemas que requieran una interacción directa con el hardware, tales como los sistemas en tiempo real o los compiladores.

En un tiempo en que la complejidad tecnológica tiende a ocultar los principios elementales que la sostienen, volver a construir una máquina desde cero no es un acto de simplificación, sino de esclarecimiento. Es una forma de recordar que toda forma de inteligencia artificial, toda red, todo sistema de cómputo, descansa —en última instancia— sobre un modelo conceptual de máquina. Y comprender ese modelo, a través de la simulación, es comprender los cimientos mismos del pensamiento computacional moderno.

Finalmente, este trabajo culmina presentando un entorno de construcción completamente autónomo que prescinde de opciones de compilacion estandard del mercado como lo es el *Netwide Assembler*(NASM) empleado en sistemas LINUX para 16, 32 (IA-32) y 64 bits (X86_64). Empleamos por tanto todas las herramientas generadas para este laboratorio: *Un ensamblador propio para la CPU de 8 bits, un simulador/loader dedicado y un linker elemental capaz de fusionar módulos* `.mem` *en una única imagen ejecutable*. El esquema utilizado no sólo garantiza un flujo de compilación coherente y totalmente controlado, sino que también establece las bases para futuras extensiones de la arquitectura; por ejemplo, la incorporación de instrucciones `CALL`/`RET` o saltos indirectos permitiría evolucionar el linker hacia un sistema capaz de gestionar llamadas reales entre módulos—como que `main` invoque al módulo `factorial` y este retorne adecuadamente—lo cual formaría parte de un programa de "trabajo futuro" orientado a comparar eficiencias y ampliar las capacidades del ecosistema."'

Es preciso destacar que a diferencia de un enlazador tradicional, `main2link.c` no resuelve símbolos ni combina objetos, sino que actúa simplemente como un *driver* que carga y ejecuta módulos `.mem` directamente sobre la CPU simulada.

En conjunto, el experimento realizado demuestra que los lenguajes de alto nivel no sustituyen, sino que complementan la comprensión arquitectónica del cómputo. La construcción de módulos ensamblados a partir de plantillas escritas en C evidencia cómo las abstracciones de alto nivel permiten estructurar soluciones complejas sin perder de vista la semántica de la máquina subyacente. El proceso `C` $\to$ `ASM` $\to$ `BIN` $\to$ `simulador` reafirma que todo programa, independientemente de su nivel de abstracción, termina dependiendo de un modelo operativo preciso, aquí representado por la arquitectura de von Neumann recreada. Así, la integración entre lenguajes de alto nivel y un simulador pedagógico de CPU permite comprender cómo las capas superiores del software expresan, reorganizan y simplifican los patrones fundamentales de control, datos y ejecución que gobiernan el funcionamiento de toda computadora. En síntesis, se pone de manifiesto que la formación sólida en computación exige transitar entre abstracciones sin perder el hilo que conecta las ideas de alto nivel con su realización física. La complementariedad entre C, ensamblador y simulación computacional no solo enriquece la comprensión del proceso de ejecución, sino que fortalece la capacidad de diseñar software más robusto, eficiente y alineado con los principios fundamentales de la arquitectura computacional.

RECURSOS ADICIONALES

Se adjunta (A) Código en lenguaje C, para el set de las dos implementaciones con simulacion en lenguaje C. El código fue compilado y ejecutado desde la terminal con VSCode, en Windows. Se empleó ANSI C en la implementación de los simuladores. (B) Se adjuntan los programas para leer/depurar en lenguaje ensamblador simple (version desarrollada ex profeso para este estudio). (C) Se adjunta el programa para

ejecutar el archivo de salida de el lector/depurador de ensamblador. (D) Codigo de interface entre lenguaje de alto nivel (en C), que hace uso de las subroutinas de lenguaje ensamblador generadas (A a C).
El codigo se puede descargar de:
https://github.com/eduardomendezPhD/neumann_assembler

REFERENCES

[1] Flynn, Michael, "Computer architecture", Wiley Encyclopedia of Computer Science and Engineering, Wiley Online Library, 2007.

[2] Nicholas Enticknap, VON NEUMANN ARCHITECTURE, Editor(s): Nicholas Enticknap,Computer Jargon Explained, Elsevier, 1989, Pages 128-129, ISBN 9781483135533, https://doi.org/10.1016/B978-1-4831-3553-3.50066-9.

[3] Indiveri, Giacomo; Liu, Shih-Chii. Memory and Information Processing in Neuromorphic Systems. Proceedings of the IEEE, 103(8), 1379–1397. 2015

[4] López Benjumea A. La simulación, una herramienta para el aprendizaje de los conceptos físicos.2016

[5] González Tamames, P., 2022. Algoritmos cuánticos controlados por medida.

[6] Arikpo, I.I., Ogban, F.U. and Eteng, I.E., 2007. von neumann architecture and modern computers. Global Journal of Mathematical Sciences, 6(2), pp.97-103.

[7] Aletan, S.O., 1992, April. An overview of RISC architecture. In Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing: technological challenges of the 1990's (pp. 11-20).

[8] Silc, J., Robic, B. and Ungerer, T., 1999. Processor Architecture: From Dataflow to Superscalar and Beyond; with 34 Tables. Springer Science & Business Media.

[9] Solihin, Y., 2015. Fundamentals of parallel multicore architecture. CRC Press.

[10] Davies, M., Srinivasa, N., Lin, T. H., Chinya, G., Cao, Y., Choday, S. H., ... & Wang, H. (2018). Loihi: A neuromorphic manycore processor with on-chip learning. IEEE Micro, 38(1), 82–99. https://doi.org/10.1109/MM.2018.112130359

[11] Kripke, S. The Church-Turing 'Thesis' as a Special Corollary of Gödel's Completeness Theorem. In Computability: Turing, Gödel, Church, and Beyond; Copeland, B.J., Posy, C., Shagrir, O., Eds.; MIT Press: Cambridge, MA, USA, 2013; pp. 77–104.

[12] Gandy, R. The Confluence of Ideas in 1936. In The Universal Turing Machine: A Half-Century Survey; Herken, R., Ed.; Kammerer & Unverzagt: Berlin, Germany, 1988; pp. 55–111.

[13] Pinghui Mo, Chang Li, Dan Zhao, Yujia Zhang, Mengchao Shi, Junhua Li, and Jie Liu. Accurate and efficient molecular dynamics based on machine learning and non von neumann architecture. npj Computational Materials, 8(1):1–15, 2022.

[14] Bojia Duan, Jiabin Yuan, Ying Liu, and Dan Li. Quantum algorithm for support matrix machines. Physical Review A, 96(3):032301, 2017.

[15] Sandeep Suresh Cranganore, Vincenzo De Maio, Ivona Brandic, Tu Mai Anh Do, and Ewa Deelman. Molecular dynamics workflow decomposition for hybrid classic/quantum systems. In 18th IEEE International Conference on e-Science, e-Science 2022, Salt Lake City, UT, USA, October 11-14, 2022, pages 346–356. IEEE, 2022.

[16] Bertrand Russell and Alfred North Whitehead, Principia Mathematica. Cambridge University Press (tres tomos, priera edición 1910, 1912 y 1913).

[17] Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, 2(42), 230–265. https://doi.org/10.1112/plms/s2-42.1.230

[18] Pylyshyn, Z. W. (1984). Computation and Cognition: Toward a Foundation for Cognitive Science. Cambridge, MA: MIT Press.

[19] Dyson, G. (2012). Turing's Cathedral: The Origins of the Digital Universe. New York: Pantheon Books.