



Herramienta de simulación para dar soporte a la enseñanza de arquitectura de computadoras

Maestría en Sistemas de Información

Ruiz Jose Maria

Director: Mg. Colombani Marcelo Alberto

Codirector: PhD José Luis Risco Martín

2025

Índice

Resumen	1
Agradecimientos	3
1 Introducción	4
1.1 Justificación	7
1.2 Objetivos	8
1.3 Metodología de desarrollo	9
1.4 Organización del documento	10
2 Arquitectura de computadoras	12
2.1 Introducción a la arquitectura de computadoras	12
2.2 Arquitecturas Von Neumann y Harvard	14
2.3 Tipos de arquitecturas	18
2.4 Repertorio de instrucciones	20
2.5 Filosofías CISC y RISC	25
2.6 Arquitectura x86	30
2.7 Lenguaje máquina y lenguaje ensamblador	34
3 Simulación	38
3.1 Introducción a la simulación	38
3.2 Simulación en la educación	39
3.3 El Formalismo DEVS (Discrete Event System Specification)	41
4 Comparativa de simuladores	45
4.1 Estudios similares	45
4.2 Simuladores bajo análisis	46
4.3 Criterios de evaluación	46
4.4 Selección de simuladores	48
4.5 Participantes en la evaluación	49
4.6 Análisis comparativo	49
4.7 Resultados	51
5 Diseño y construcción del simulador	54
5.1 Requisitos de la herramienta	54
5.2 Justificación pedagógica de la arquitectura simplificada	60

5.3	Introducción a VonSim	62
5.4	Estructura y componentes de VonSim8	64
5.5	Adaptaciones y mejoras en VonSim8	70
5.6	Repertorio de instrucciones	79
5.7	Ciclo de la instrucción	87
5.8	Modelado con xDEVS del ciclo de instrucción MOV AL, BL	93
5.9	Módulo de entrada/salida e interrupciones	97
5.10	Validación pedagógica del simulador	120
5.11	Aportes y contribuciones del simulador VonSim8	123
5.12	Resumen del capítulo	125
6	Apéndices	126
6.1	Anexo A: Protocolo de Entrevista Semiestructurada	126
6.2	Anexo B: Instrumentos de recolección de datos	128
6.3	Anexo C: Análisis de los resultados	131
6.4	Anexo D: Resultados de simulación xdevs	137
7	Trabajos futuros	161
8	Conclusiones finales	164
8.1	Principales aportes técnicos y didácticos	164
8.2	Hallazgos pedagógicos (alcance y límites)	165
8.3	Limitaciones y amenazas a la validez	165
8.4	Síntesis	165
8.5	Líneas futuras	165
9	Bibliografía	166

Índice de tablas

1.1	Funcionalidades principales del simulador propuesto	8
2.1	Cuadro comparativo entre arquitecturas Von Neumann y Harvard .	17
2.2	Aplicaciones de la simulación en distintos sectores	20
2.3	Modos de direccionamiento básicos	24
2.4	Comparativa de repertorios de instrucciones reales	25
2.5	Comparativa entre CISC y RISC	29
2.6	Hitos en la evolución x86	32
2.7	Línea de Tiempo de la Evolución de la Arquitectura x86	32
2.8	Comparación de ensambladores arquitectura x86	37
3.1	Aplicaciones de la simulación en distintos sectores	39
4.1	Criterios e indicadores de evaluación de simuladores	48
4.2	Proceso de selección de simuladores	49
4.3	Comparativa según criterios de evaluación preestablecidos	51
5.1	Activación progresiva del repertorio de instrucciones	57
5.2	Resumen de requisitos funcionales y su fundamentación pedagógica .	60
5.3	Estructura VonSim8: componentes principales y características .	65
5.4	Registro Flags: descripción de las banderas	67
5.5	Política de actualización de banderas por clase de instrucción .	68

5.6	Comparativa de características entre VonSim y VonSim8	79
5.7	Tabla de instrucciones de VonSim8	80
5.8	Saltos condicionales: instrucciones y condiciones	81
5.9	Tabla de modos de direccionamiento	85
5.10	Categoría de instrucciones y códigos de operación en VonSim8	86
5.11	Tabla de codificación de instrucciones	87
5.12	Tabla de registros del simulador	88
5.13	Modelos y componentes principales del simulador VonSim8	94
5.14	Secuencia de microoperaciones y eventos DEVS fase captación	95
5.15	Secuencia de microoperaciones y eventos DEVS fase ejecución	96
5.16	Categoría de instrucciones y códigos de operación en VonSim8	99
5.17	Módulos, direcciones y nombres del simulador	106
5.18	Líneas de interrupción y dispositivos asociados	113
6.1	Valoración de aspectos clave del simulador VonSim8	132
6.2	Evaluación de aspectos clave del simulador VonSim8	132
6.3	Evaluación de aspectos clave por los docentes	135
6.4	Evaluación del simulador educativo	135

Índice de figuras

2.1	Arquitectura Von Neumann	15
2.2	Arquitectura Harvard	16
2.3	Arquitectura Híbridas	17
2.4	Características repertorio de instrucciones	21
2.5	Modos de direccionamiento	23
2.6	Formato de instrucciones	25
2.7	Convergencia de filosofías	30
2.8	Diagrama esquemático microprocesador Intel 8086	31
2.9	Formato de instrucciones del Pentium x86	33
2.10	Proceso de ensamblado	36
3.1	Relación entre modelos atómicos y modelos acoplados en DEVS	42
3.2	Modelo acoplado en DEVS	43
5.1	Estructura general del simulador VonSim8	55
5.2	Editor ensamblador	56
5.3	Ciclo de instrucción: captación y ejecución	57
5.4	Módulo genérico de entrada/salida programada	58
5.5	Métricas de rendimiento	59
5.6	Documentación on line	59
5.7	Memoria principal	69

5.8	Buses	70
5.9	Registro Flags	71
5.10	Registro de estado I	72
5.11	Controles del simulador	72
5.12	Registro de 8 bits	73
5.13	Eliminación registro temporales left, right y result	73
5.14	Registro SP, id y ri	74
5.15	Memoria principal	74
5.16	Resaltado registro IP y registro SP	75
5.17	Resaltado vector de interrupciones	75
5.18	Visor de instrucciones y datos del programa en memoria	76
5.19	Compatibilidad directiva ORG y END	77
5.20	Editor con ampliación de fuente y ejemplos	77
5.21	Decodificador en VonSim8	78
5.22	Tour de aprendizaje en VonSim8	78
5.23	Flujo del ciclo de instrucción en VonSim8	88
5.24	Arquitectura general del simulador	98
5.25	Teclado y pantalla	104

Resumen

Esta tesis describe el diseño, construcción y validación de VonSim8, un simulador educativo de arquitectura de computadoras inspirado en principios de la arquitectura x86. El trabajo se inicia con un análisis comparativo de simuladores (Emu8086, VonSim, Simple 8-bit Assembler Simulator), que evidencia carencias pedagógicas —falta de activación progresiva del repertorio, visualización limitada del ciclo de instrucción, ausencia de métricas y barreras de acceso— y fundamenta la definición de requisitos funcionales y didácticos a partir de entrevistas a docentes y percepciones estudiantiles.

VonSim8 implementa una arquitectura de 8 bits con registros, 256 bytes de memoria, vector de interrupciones reducido y un repertorio inicial acotado y escalable. Incorpora visualización RTL paso a paso del ciclo fetch–execute, activación progresiva de instrucciones, gestión básica de interrupciones, métricas (CPI, ciclos por fase y tiempo de CPU), editor de ensamblador integrado, visor de memoria e instrucciones, tour guiado y centro de ayuda.

La validación técnica se realiza mediante modelado DEVS (`xdevs.py`) del ciclo `MOV AL, BL`, verificando coherencia funcional y temporal, estados intermedios y trazabilidad de microoperaciones. La evaluación pedagógica (encuesta a estudiantes, N=14; entrevistas a docentes, N=2) indica mejoras percibidas en la comprensión del ciclo de instrucción, el flujo de datos y el manejo básico de interrupciones, con oportunidades de refinamiento en animaciones y representación de banderas.

Las principales contribuciones son:

1. Un marco de requisitos didácticos para simuladores introductorios.
2. Una arquitectura simplificada y trazable a x86 con progresión controlada de complejidad.
3. La integración de modelado formal DEVS para una validación reproducible.
4. La instrumentación de métricas de ejecución en contexto educativo.
5. La propuesta de extensiones futuras (NASM, registros de 64 bits, DMA, pantalla gráfica, ampliación del modelado DEVS).

En síntesis, VonSim8 favorece la articulación teoría–práctica, reduce la carga cognitiva inicial y ofrece una plataforma extensible alineada con la currícula. Las líneas futuras

incluyen ampliar el repertorio, profundizar la evaluación quasi-experimental y evolucionar el modelo hacia características avanzadas (pipeline, caché, syscalls) manteniendo el enfoque pedagógico.

Agradecimientos

No habría sido posible culminar esta tesis sin el apoyo, la comprensión y el aliento de todas las personas que me acompañaron en cada etapa de este proceso.

A mi esposa, Natalia, y a mis hijos, Tomás y Thiago, por su amor incondicional, su paciencia y su apoyo constante, que me impulsaron a continuar aun en los momentos más desafiantes.

A mi director de tesis, Marcelo Colombani, por su guía, orientación y compromiso, fundamentales para la realización de este trabajo.

A mi familia, en especial a mi madre y a mis hermanos, por creer siempre en mí. Un agradecimiento especial a mi hermana Silvia, por su valioso asesoramiento y sus oportunos consejos.

A mi colega y amiga, Amalia Delduca, por su acompañamiento, el intercambio de ideas y el apoyo durante todo el desarrollo del proyecto.

A mi codirector, José Luis Risco Martín, por su colaboración, dedicación y aportes que enriquecieron este trabajo.

A mis amistades, por escucharme, acompañarme y brindarme su apoyo incondicional durante el desarrollo de esta tesis.

Finalmente, a la Universidad Nacional de Entre Ríos, y en particular a la Facultad de Ciencias de la Administración, por brindarme la oportunidad de formarme como profesional en la universidad pública.

Capítulo 1

Introducción

El uso cotidiano de dispositivos como computadoras personales, teléfonos móviles y relojes inteligentes está sustentado en arquitecturas computacionales específicas, cuya comprensión es fundamental para el desarrollo eficiente de soluciones informáticas.

Es esencial que los estudiantes de Arquitectura de Computadoras comprendan tanto la estructura como el funcionamiento interno de una computadora, y adquieran experiencia práctica con ellas. Para lograrlo, es fundamental disponer de un laboratorio bien equipado con el hardware adecuado y suficiente tiempo para que los estudiantes se familiaricen con las herramientas prácticas. En este contexto, se han desarrollado numerosos simuladores que facilitan la comprensión de la estructura y el funcionamiento de las computadoras, proporcionando valiosas experiencias de aprendizaje. Este trabajo se inscribe en dicha problemática educativa y busca contribuir con el desarrollo de una herramienta de simulación adaptada a las necesidades de la enseñanza de arquitectura de computadoras.

Esta tesis, inscrita en la Maestría en Sistemas de Información de la Facultad de Ciencias de la Administración, está directamente vinculada con el proyecto de investigación I/D novel PID-UNER 7065, titulado “Enseñanza/aprendizaje de Arquitectura de Computadoras con herramientas de simulación de sistemas de cómputo”, desarrollado en la Facultad de Ciencias de la Administración de la Universidad Nacional de Entre Ríos [1].

La asignatura Arquitectura de Computadoras forma parte del plan de estudios de la carrera de Licenciatura en Sistemas, Universidad Nacional de Entre Ríos. Su objetivo es que los estudiantes comprendan la estructura y funcionamiento de las computadoras, y la ejecución lógica de un programa a nivel de instrucciones de máquina.

Para comprender los fundamentos de la arquitectura de computadoras, resulta necesario abordar su estructura básica y funcionamiento interno, comenzando por una descripción funcional general del sistema.

Desde una perspectiva funcional, una computadora es un sistema que capta datos de entrada, los procesa de acuerdo con instrucciones codificadas, y produce salidas a través de dispositivos periféricos o almacenamiento. Esta dinámica operativa constituye la base para comprender su arquitectura interna y los principios que rigen su diseño.

El procesamiento se realiza a través del procesador o CPU, y es en este componente donde los estudiantes encuentran mayor complejidad y dificultades para comprender su funcionamiento.

A pesar de que es posible explicar las partes del procesador, su funcionamiento, la interacción de sus componentes y enseñar lenguaje ensamblador mediante prácticas, los estudiantes suelen tener dificultades para lograr una comprensión completa del funcionamiento.

Sin embargo, la utilización de simuladores favorece la consolidación de esquemas conceptuales mediante práctica guiada y visualización progresiva en las clases teóricas, por ello, los simuladores deben ser simples, intuitivos y visualmente atractivos, para que los estudiantes puedan centrarse en los conceptos de arquitectura y no en el aprendizaje de la herramienta en sí.

La simulación es un término de uso diario en muchos contextos: medicina, militar, entretenimiento, educación, etc., debido a que permite ayudar a comprender cómo funciona un sistema, responder preguntas como “qué pasaría si”, con el fin de brindar hipótesis sobre cómo o por qué ocurren ciertos fenómenos.

En este contexto, es necesario comprender con mayor profundidad qué se entiende por simulación y cómo esta técnica puede aplicarse en entornos educativos, se define simulación como el proceso de imitar el funcionamiento de un sistema a medida que avanza en el tiempo. Para llevar a cabo una simulación, es necesario construir un modelo conceptual que represente las características y comportamientos del sistema de interés. La simulación permite observar cómo dicho modelo evoluciona en el tiempo, replicando dinámicas reales o hipotéticas [2], [3], [4].

Con los avances en el mundo digital, la simulación se ha convertido en una metodología de solución de problemas indispensable para ingenieros, docentes, diseñadores y gerentes. La complejidad intrínseca de los sistemas informáticos los hace difícil comprender y costosos de desarrollar sin utilizar simulación [3].

Muchas veces en el ámbito educativo, resulta difícil transmitir fundamentos teóricos de la organización y arquitectura interna de las computadoras debido a la complejidad de los procesos involucrados. Cuando se emplean exclusivamente métodos de enseñanza tradicionales —como pizarras, libros de texto o diapositivas—, estos resultan insuficientes para representar de manera efectiva las complejidades involucradas en la arquitectura de las computadoras [5], [6], [7], [8], [9], [10], [11].

Es evidente la necesidad de utilizar nuevas tecnologías como recursos didácticos y

medios de transferencia de conocimiento, ya que ayudan a los estudiantes a relacionar conceptos abstractos con realidades concretas. Estas tecnologías permiten situar al estudiante en un contexto que imita aspectos de la realidad, posibilitando la detección y análisis de problemáticas semejantes a las que se presentan en entornos reales. Este enfoque promueve un mejor entendimiento a través del trabajo exploratorio, la inferencia, el aprendizaje por descubrimiento y el desarrollo de habilidades [12], [13], [14], [15].

Un simulador de arquitectura es una herramienta que imita el hardware de un sistema, representando sus aspectos arquitectónicos y funciones. Permiten realizar cambios, pruebas y ejecutar programas sin riesgo de dañar componentes ni depender de equipos físicos disponibles [16].

Algunas herramientas ofrecen una representación en forma visual e interactiva de la organización y arquitectura interna de la computadora, facilitando así la comprensión de su funcionamiento. En este sentido, los simuladores juegan una pieza clave en el campo de la Arquitectura de Computadores, permitiendo conectar fundamentos teóricos con la experiencia práctica, reduciendo la carga cognitiva asociada a la abstracción de procesos internos y facilitando la labor docente [17], [18], [19], [20], [21].

Dentro del estudio de las arquitecturas de computadoras, la arquitectura x86 ocupa un lugar destacado debido a su amplia difusión y compatibilidad. A continuación, se presenta una breve evolución de esta arquitectura, que justifica su inclusión en el diseño de herramientas de simulación para la enseñanza. Comenzó con el procesador Intel 8086 en 1978 como una arquitectura de 16 bits. Evolucionó a una arquitectura de 32 bits con el procesador Intel 80386 en 1985 (i386 o x86-32) y posteriormente a 64 bits con las extensiones de AMD (AMD64) y su adopción por Intel (Intel 64) [22], [23].

A pesar de la aparición de nuevas arquitecturas como ARM o RISC-V, que serán analizadas comparativamente en capítulos posteriores, la arquitectura x86 conserva una alta relevancia en contextos educativos y profesionales. Por ello, se considera pertinente su inclusión como base para el desarrollo de herramientas de simulación.

Un procesador x86-64 mantiene la compatibilidad con los modos x86 existentes de 16 y 32 bits, y permite ejecutar aplicaciones de 16 y 32 bits, como así también de 64 bits. Esta compatibilidad hacia atrás protege las principales inversiones en aplicaciones y sistemas operativos desarrollados para la arquitectura x86 [22], [23], [24].

Por ello, la enseñanza de la arquitectura x86 es de gran relevancia en la asignatura Arquitecturas de Computadoras debido a los diferentes temas que aborda.

Como alternativa al equipamiento físico, los simuladores permiten suplir limitaciones de hardware y tiempo, brindando una experiencia de aprendizaje más accesible y replicable [25].

1.1 Justificación

Los estudiantes enfrentan barreras iniciales para construir modelos mentales del flujo de datos y control; los docentes carecen de herramientas graduales que dosifiquen complejidad y proporcionen retroalimentación inmediata a la hora de abordar los complejos conceptos teóricos inherentes a la arquitectura x86. Para los estudiantes, en particular, la introducción a la arquitectura de una computadora puede resultar abrumadora debido a la abstracción y el nivel de detalle técnico requerido. Por su parte, los docentes se ven limitados en la capacidad de ilustrar estos conceptos de manera gradual y progresiva debido a la falta de herramientas didácticas específicas para esta arquitectura. Ante estos desafíos, los simuladores juegan un papel crucial como herramientas de apoyo, al permitir la exploración y experimentación con los conceptos de forma visual e interactiva.[\[26\]](#).

La necesidad de desarrollar un simulador específico para la arquitectura x86 se fundamenta en las limitaciones de los simuladores actuales, que no se adaptan de manera efectiva al plan de estudios específico de la asignatura Arquitectura de Computadoras, tal como se dicta en la Universidad Nacional de Entre Ríos. Aunque existen simuladores que apoyan la enseñanza de la arquitectura x86 en otros contextos [\[16\]](#), [\[17\]](#), estos tienden a incluir una gran cantidad de contenidos preestablecidos. Si bien dichos contenidos son relevantes, introducir la arquitectura x86 en su totalidad desde las primeras instancias del curso puede resultar contraproducente para estudiantes principiantes, debido a la complejidad técnica y a la extensa cantidad de conceptos involucrados.

Esta tesis propone un enfoque alternativo: el desarrollo de una herramienta de simulación específicamente diseñada para apoyar la enseñanza de los contenidos de la asignatura Arquitectura de Computadoras. El sistema simulará una computadora basada en la arquitectura x86, ofreciendo una representación progresiva de su estructura y funcionamiento. Abordará de forma modular los principales componentes del sistema: la unidad central de procesamiento (CPU), la memoria principal, el módulo de entrada/salida (E/S) y los buses de comunicación [\[27\]](#).

Entre sus funcionalidades clave, permitirá:

- Visualizar en detalle cada una de las etapas del ciclo de instrucción (fases de captación ‘fetch’ y ejecución ‘execute’).
- Trabajar con un repertorio limitado y escalable de instrucciones en lenguaje ensamblador.
- Ejecutar programas de forma paso a paso o completa.
- Gestionar interrupciones básicas para simular la interacción con periféricos como teclado y pantalla.

- Evaluar el rendimiento de los programas a partir de métricas observables durante la simulación.

Estas características facilitarán una comprensión progresiva de la arquitectura x86 y promoverán una experiencia de aprendizaje alineada con los objetivos del curso.

Tabla 1.1: Funcionalidades principales del simulador propuesto

Componente	Funcionalidad Principal	Propósito Didáctico
CPU	Ciclo de instrucción, ejecución paso a paso	Comprender la secuencia de ejecución
Memoria	Lectura/escritura en tiempo real	Visualizar acceso a datos
E/S	Gestión básica de teclado/pantalla	Simular interacción con periféricos
Instrucciones	Conjunto limitado y ampliable	Acompañar el avance del curso
Evaluación	Métricas de rendimiento	Analizar eficiencia de programas

Contar con un simulador adaptado específicamente a los contenidos de esta asignatura no solo facilita el proceso de aprendizaje, al presentar los conceptos de manera progresiva y alineada con la currícula, sino que también permite una experiencia de aprendizaje contextualizada. Esto fomenta un aprendizaje significativo, en el cual los estudiantes pueden conectar teoría y práctica de manera efectiva a través de una herramienta diseñada para abordar de forma gradual y específica los conceptos fundamentales del curso.

Para garantizar que el simulador sea robusto, modular, flexible y fácil de modificar o ampliar, se explorará la utilización de técnicas formales de modelado y simulación, como las redes de Petri y DEVS (Discrete Event System Specification). Estas técnicas permiten una separación conceptual entre las capas de modelado y simulación, lo cual facilita tanto la comprensión del software como su adaptación. Además, estas metodologías permiten que las simulaciones escalen de forma transparente, posibilitando su ejecución en entornos de cómputo paralelo o distribuido sin necesidad de modificar el modelo, lo que representa una ventaja significativa en términos de escalabilidad [28], [29], [30].

1.2 Objetivos

El objetivo principal de esta tesis es desarrollar una herramienta de simulación centrada en la arquitectura x86, orientada a fortalecer los procesos de enseñanza y aprendizaje de la asignatura Arquitectura de Computadoras. La herramienta estará alineada con los contenidos y objetivos formativos establecidos en la currícula

vigente. En función de este objetivo general, se plantean los siguientes objetivos específicos:

1. Estudiar y evaluar diferentes herramientas actuales de simulación destinadas a dar apoyo a la enseñanza de la arquitectura x86.
2. Diseñar e implementar una herramienta didáctica que facilite la enseñanza de los contenidos clave de la asignatura Arquitectura de Computadoras. Para ello, la herramienta deberá contemplar las siguientes funcionalidades:
 - Una visión global de la estructura y funcionamiento de la computadora.
 - Generación y ejecución de programas en ensamblador.
 - Repertorio de instrucciones x86 reducido y habilitado progresivamente.
 - Simulación visual e interactiva de micropasos de instrucciones.
 - Gestión de interrupciones y comunicación con periféricos.
 - Medidas de rendimiento de ejecución de programas.

1.3 Metodología de desarrollo

La metodología de desarrollo de este simulador específico para la arquitectura x86 sigue una serie de etapas diseñadas para garantizar una progresión coherente y eficaz desde la fase de análisis hasta el diseño e implementación del simulador, de manera que se ajuste al plan de estudios de la asignatura Arquitectura de Computadoras en la Universidad Nacional de Entre Ríos.

1. Análisis de requerimientos: En esta etapa inicial, se identifican y definen los objetivos específicos del simulador, así como los requerimientos técnicos y pedagógicos necesarios para su alineación con la currícula. Este análisis establece una base sólida y clara para todas las fases subsecuentes del proyecto, asegurando que el simulador cumpla con las necesidades educativas y técnicas del curso.
2. Revisión y recopilación de información: Se realiza una investigación sistemática sobre los simuladores existentes que abordan la arquitectura x86, considerando su aplicabilidad en contextos educativos. Este paso incluye un análisis de las características, ventajas y limitaciones de los simuladores existentes, proporcionando una comprensión más profunda del contexto educativo y permitiendo identificar áreas de mejora en relación con el objetivo del proyecto.
3. Estudio comparativo: A partir de la información recopilada, se realiza un estudio comparativo detallado de las características de los simuladores disponibles en el mercado. Esta etapa busca evaluar cuáles de sus funcionalidades pueden adaptarse o modificarse y cuáles deberían excluirse, de acuerdo con los objetivos del simulador y las necesidades específicas del plan de estudios. Los hallazgos

de este análisis comparativo constituirán una base sólida para orientar las decisiones de diseño del simulador.

4. Diseño y planificación del simulador: Con base en los hallazgos previos, se define la arquitectura, las funcionalidades y los módulos específicos del simulador. El diseño se enfoca en facilitar el aprendizaje progresivo de los estudiantes, implementando un repertorio de instrucciones que se habiliten a medida que avanzan en el curso. En esta etapa, se utilizan técnicas formales de modelado, como redes de Petri y DEVS (Discrete Event System Specification), para establecer una estructura modular, robusta y flexible que facilite tanto la comprensión como la modificación futura de la herramienta.
5. Construcción y desarrollo: En esta fase, se lleva a cabo la implementación del simulador de acuerdo con el diseño previamente definido. Cada funcionalidad se implementa y verifica de manera secuencial, asegurando su conformidad con los requerimientos técnicos y pedagógicos definidos. También se realizan pruebas parciales para asegurar la precisión y funcionalidad de cada módulo, lo que permite identificar y corregir errores tempranamente.
6. **Evaluación y ajuste:** Finalmente, se somete el simulador a una serie de pruebas con estudiantes o expertos en la materia para evaluar su efectividad en el aprendizaje de los conceptos de arquitectura de computadoras. Los resultados obtenidos en esta fase permiten realizar ajustes y optimizaciones necesarias, mejorando la herramienta y asegurando que cumpla con su propósito educativo de manera efectiva.

1.4 Organización del documento

El resto de este documento se estructura de la siguiente manera:

- En el Capítulo (2), se presenta una descripción detallada de la arquitectura x86, definiendo sus características y el conjunto de instrucciones que la componen. Esta base teórica es fundamental para comprender los aspectos que se simularán en el proyecto.
- El Capítulo (3) explora el papel de la simulación desde una perspectiva didáctica, enfatizando su valor como herramienta de apoyo en la enseñanza de Arquitectura de Computadoras. Aquí se revisan los beneficios de los simuladores en la enseñanza y los desafíos que ayudan a resolver en la formación de los estudiantes.
- En el Capítulo (4), se realiza un análisis comparativo de los simuladores actuales, evaluándolos según criterios previamente establecidos. Este análisis permite identificar las limitaciones y fortalezas de cada simulador y establecer el contexto para la propuesta de esta tesis.

- Finalmente, en el Capítulo (5), se describe el desarrollo de un simulador específico que se ajusta a los objetivos de enseñanza y aprendizaje de la arquitectura x86 en el contexto de la currícula. Este simulador está diseñado como una herramienta práctica y accesible para facilitar la comprensión de conceptos complejos en la asignatura.

Capítulo 2

Arquitectura de computadoras

Este capítulo aborda los conceptos fundamentales de la arquitectura de computadoras, incluyendo las filosofías de diseño CISC y RISC, la evolución de la arquitectura x86 y una introducción al lenguaje ensamblador. Estos temas constituyen la base necesaria para comprender el funcionamiento interno de los sistemas informáticos.

2.1 Introducción a la arquitectura de computadoras

La arquitectura de computadoras es una disciplina central en el campo de la informática que estudia el diseño, la organización y la interacción entre los componentes de un sistema computacional. Esta área abarca tanto aspectos de hardware como de software que interactúan directamente con él, proporcionando principios fundamentales para construir sistemas eficientes, robustos y adaptables. Comprender su funcionamiento resulta esencial para analizar cómo se implementan, optimizan y escalan los sistemas informáticos en diversos contextos tecnológicos [20], [31], [32], [33].

Uno de los conceptos clave en esta disciplina es la distinción entre **arquitectura de computadoras** y **organización de computadoras**. La arquitectura se refiere a los elementos visibles para el programador, como el conjunto de instrucciones, los registros y los modos de direccionamiento. La organización, en cambio, se enfoca en los detalles físicos de implementación, tales como el diseño de circuitos y los ciclos de reloj necesarios para cada operación [19], [20], [31], [32], [33].

Distinguir esta diferencia es crucial para analizar cómo los diseños arquitectónicos han evolucionado en respuesta a las crecientes demandas de rendimiento, eficiencia energética y escalabilidad. En este sentido, arquitecturas como ARM y RISC-V se han consolidado en sistemas embebidos y dispositivos móviles debido a su simplicidad estructural y bajo consumo energético [34], [35], [36]. En contraste, la arquitectura

x86 ha adoptado un enfoque híbrido que combina características de CISC y RISC, permitiéndole adaptarse a los exigentes requerimientos del mercado [20], [33], [37].

El análisis de una arquitectura de computadoras implica examinar múltiples dimensiones técnicas que inciden en su desempeño y aplicabilidad. Entre las más relevantes se encuentran:

- **Repertorio de instrucciones:** conjunto de operaciones que el procesador puede ejecutar directamente.
- **Capacidad de procesamiento:** determinada por el número de bits con los que opera la CPU (por ejemplo, 32 o 64 bits).
- **Modos de direccionamiento de memoria:** mecanismos mediante los cuales una instrucción accede a posiciones de memoria, como el direccionamiento directo, indirecto, segmentado o lineal.
- **Jerarquía de memoria y mecanismos de entrada/salida:** estructuras que influyen en la eficiencia del acceso a datos y en la interacción con dispositivos periféricos.
- **Grado de paralelismo:** capacidad de ejecutar múltiples instrucciones o tareas simultáneamente, ya sea a nivel de instrucción (ILP) o de procesos (TLP).

Estos dimensiones técnicas adquieren especial relevancia en sistemas contemporáneos aplicados a inteligencia artificial, internet de las cosas (IoT), computación en la nube y ciberseguridad, donde el equilibrio entre rendimiento, consumo energético y escalabilidad resulta determinante [35], [38], [39].

Un componente esencial en el estudio de esta disciplina es la **arquitectura del conjunto de instrucciones (ISA, por sus siglas en inglés)**, que define la interfaz entre el hardware y el software [19]. La ISA especifica las operaciones disponibles, la codificación de las instrucciones y las formas de manipular los datos. Esta interfaz es fundamental para el diseño de compiladores, sistemas operativos y herramientas de simulación, ya que permite abstraer el funcionamiento del hardware a nivel lógico y facilita la portabilidad del software.

El diseño arquitectónico implica tomar decisiones que suponen compromisos (*trade-offs*), tales como la complejidad del hardware frente al rendimiento, o la eficiencia energética frente a la flexibilidad funcional. Estas decisiones determinan la aplicabilidad de una arquitectura en distintos dominios tecnológicos. Por ejemplo:

- La arquitectura **x86** resulta adecuada para entornos que requieren alto rendimiento y compatibilidad con software legado.
- La arquitectura **ARM** se prefiere en dispositivos móviles debido a su bajo consumo energético [35], [38], [39].
- **RISC-V**, por su parte, destaca por su apertura, modularidad y flexibilidad, lo que la convierte en una alternativa atractiva para investigación, docencia y aplicaciones personalizadas [34], [40].

En síntesis, el estudio de la arquitectura de computadoras permite comprender el funcionamiento interno de los sistemas, optimizar el desarrollo de soluciones tecnológicas complejas y fomentar la innovación en ingeniería de sistemas. Su enseñanza resulta fundamental en la formación en ciencias de la computación y disciplinas afines.

Desde una perspectiva educativa, el uso de herramientas de simulación contribuye a una comprensión progresiva de los conceptos arquitectónicos, al permitir experimentar con distintas arquitecturas y observar de forma interactiva el comportamiento del hardware [41], [42]. Esta dimensión didáctica adquiere especial importancia en el desarrollo de la herramienta propuesta en esta tesis, centrada en la arquitectura x86. Dicha arquitectura, ampliamente difundida en contextos académicos e industriales, también presenta desafíos significativos desde el punto de vista pedagógico, debido a su complejidad estructural y diversidad funcional.

2.2 Arquitecturas Von Neumann y Harvard

Comprender las arquitecturas modernas requiere el análisis de dos modelos conceptuales fundamentales que sentaron las bases del diseño actual de sistemas computacionales: Von Neumann y Harvard. Estos modelos arquitectónicos no solo constituyen la base teórica de muchas arquitecturas contemporáneas, sino que también permiten identificar sus fortalezas, limitaciones y áreas de aplicación.

2.2.1 Arquitectura Von Neumann

La arquitectura Von Neumann, formalizada por John von Neumann en 1945 en su influyente documento “First Draft of a Report on the EDVAC” [43], establece un modelo computacional en el cual tanto los datos como las instrucciones residen en una única memoria y comparten un mismo bus de comunicación. Esta arquitectura se caracteriza por sus cuatro componentes fundamentales: la unidad central de procesamiento (CPU), la unidad de control, la memoria y los dispositivos de entrada/salida. La unificación del espacio de memoria facilita el diseño del sistema y la programación, sin embargo, esta unificación también origina una limitación conocida como el ‘cuello de botella de Von Neumann’, que se refiere a la imposibilidad de acceder simultáneamente a datos e instrucciones debido al uso compartido del mismo bus, lo cual reduce la eficiencia del procesamiento, particularmente en aplicaciones con uso intensivo de datos [19], [20].



Figura 2.1: Arquitectura Von Neumann

2.2.2 Arquitectura Harvard

Mientras la arquitectura Von Neumann se convertía en el paradigma dominante, paralelamente se desarrollaba un enfoque alternativo. La arquitectura Harvard tiene su origen en el diseño del Harvard Mark I, una computadora electromecánica desarrollada entre 1939 y 1944 durante la Segunda Guerra Mundial en la Universidad de Harvard bajo la dirección de Howard Aiken y con el apoyo de IBM [44], [45]. El Harvard Mark I sentó las bases para un modelo arquitectónico diferente al de Von Neumann, caracterizado por una separación física entre instrucciones y datos. En este modelo, los datos y las instrucciones residen en memorias físicamente separadas, accedidas a través de buses independientes, lo cual mejora la eficiencia del procesamiento al eliminar la competencia por el bus entre instrucciones y datos. Esta organización evita el cuello de botella característico de Von Neumann y permite un acceso paralelo que incrementa el rendimiento en escenarios críticos para la eficiencia. A continuación, se presenta una comparación sistemática entre ambos modelos, a fin de comprender mejor sus implicancias técnicas y contextos de aplicación [31]. Debido a su eficiencia, esta arquitectura se ha adoptado ampliamente en sistemas embebidos, microcontroladores y procesadores de señal digital (DSP) [46].

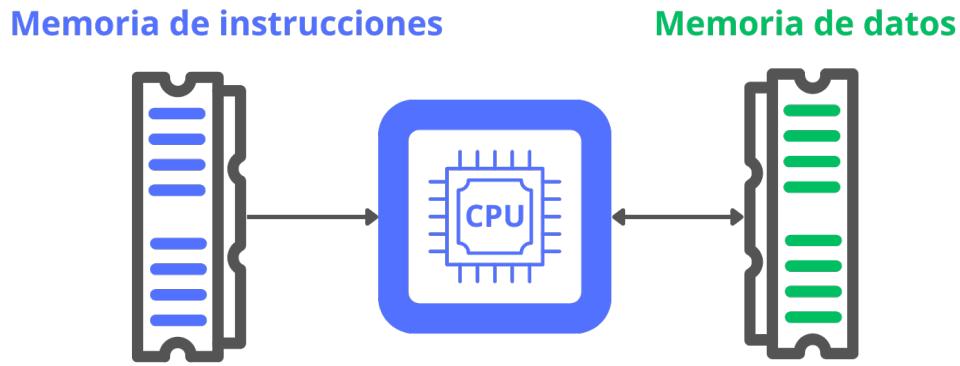


Figura 2.2: Arquitectura Harvard

2.2.3 Comparativa entre Von Neumann y Harvard

Como señalan Stallings [20] y Hennessy [19], la arquitectura Von Neumann continúa siendo una alternativa predominante cuando se priorizan la simplicidad del diseño, la flexibilidad en la asignación de memoria y la compatibilidad con software de propósito general, como ocurre en muchas computadoras personales y servidores contemporáneos. En cambio, la arquitectura Harvard ha demostrado ventajas significativas en aplicaciones que demandan procesamiento en tiempo real y eficiencia energética, como dispositivos móviles, microcontroladores y entornos de control industrial. La elección entre ambas arquitecturas responde, en última instancia, a requerimientos específicos del sistema, ya sea por su complejidad, restricciones energéticas o necesidades de rendimiento paralelo.

Para una comparación más sistemática, se pueden establecer criterios como tipo de memoria, estructura de buses, capacidad de acceso paralelo, casos de uso representativos, ventajas y limitaciones.

Ambos modelos conceptuales han tenido una influencia decisiva en el diseño de arquitecturas contemporáneas. Mientras que el modelo Von Neumann ofrece un enfoque unificado que simplifica el desarrollo de software y hardware, la arquitectura Harvard destaca por su capacidad para mejorar el rendimiento mediante el acceso paralelo a instrucciones y datos. Esta distinción resulta crucial al analizar el diseño de arquitecturas modernas como x86, que constituye el foco de esta tesis. Comprender las implicancias de estas decisiones arquitectónicas es esencial para evaluar el impacto en el rendimiento, la eficiencia energética y la escalabilidad de los sistemas actuales.

El contraste entre estos dos modelos ha dado lugar a enfoques intermedios que buscan capitalizar las ventajas de ambos. Como resultado de esta evolución, emergen las denominadas arquitecturas híbridas, las cuales integran características de ambos modelos para optimizar el rendimiento y la flexibilidad del sistema.

Tabla 2.1: Cuadro comparativo entre arquitecturas Von Neumann y Harvard

Característica	Von Neumann	Harvard
Memoria	Única para datos e instrucciones	Separada para datos e instrucciones
Buses	Bus compartido	Buses independientes
Acceso simultáneo	No	Sí
Ejemplo típico	Intel x86	AVR, PIC
Ventaja principal	Diseño más simple	Mayor rendimiento
Limitación principal	Cuello de botella	Diseño más complejo

2.2.4 Arquitecturas híbridas

Muchas arquitecturas contemporáneas implementan un enfoque híbrido, también conocido como arquitectura Harvard modificada. Este modelo emplea memorias separadas para datos e instrucciones a nivel microarquitectónico a menudo mediante la utilización de memorias caché de nivel 1 (L1) separadas para instrucciones y datos. No obstante, desde la perspectiva del programador, el modelo de memoria se mantiene unificado, facilitando el desarrollo de software sin exponer la complejidad del diseño interno. Esta dualidad permite optimizar la implementación física del procesador sin complicar el modelo de programación [19], [20], [36], [40].

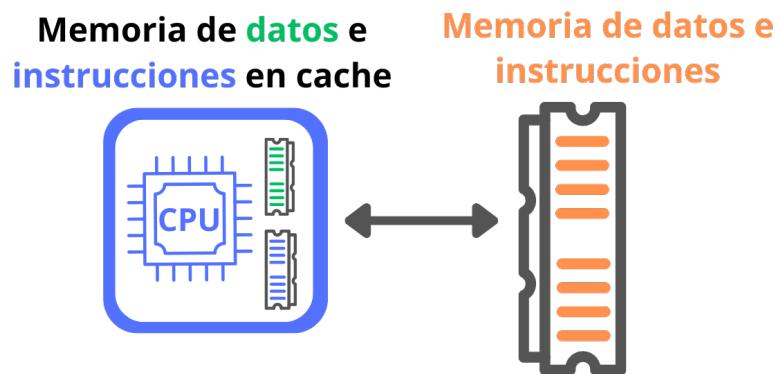


Figura 2.3: Arquitectura Híbridas

Esta aproximación híbrida se implementa en arquitecturas modernas como ARM Cortex y los procesadores Intel Core, los cuales incorporan cachés separadas para instrucciones y datos con el objetivo de optimizar el rendimiento del pipeline, lo que facilita una mayor paralelización del procesamiento y reduce los conflictos en el acceso a memoria. A pesar de que el modelo de memoria visible para el programador se

presenta como unificado, a nivel interno se implementan mecanismos característicos de la arquitectura Harvard, como el uso de memorias caché separadas para instrucciones y datos [47], [48].

La adopción de arquitecturas híbridas, como la Harvard modificada, ha permitido a los diseñadores combinar la flexibilidad del modelo Von Neumann con la eficiencia del modelo Harvard. Esta convergencia no solo optimiza el rendimiento de los sistemas, sino que también responde a las exigencias contemporáneas en términos de consumo energético y capacidad de procesamiento paralelo. En este sentido, la distinción entre ambos modelos continúa siendo un eje conceptual clave para comprender la evolución de las arquitecturas modernas y su adaptación a diferentes escenarios tecnológicos.

En síntesis, la comprensión de las arquitecturas fundamentales —Von Neumann, Harvard e híbridas— resulta esencial para el desarrollo de herramientas de simulación efectivas en la enseñanza de arquitectura de computadoras. Los conceptos explorados en esta sección proporcionan los fundamentos conceptuales esenciales para el diseño y desarrollo de la herramienta de simulación propuesta en esta tesis.

2.3 Tipos de arquitecturas

El análisis de diversas arquitecturas de computadoras, y particularmente de sus repertorios de instrucciones (Instruction Set Architecture, ISA), es esencial para comprender sus ventajas, limitaciones y áreas de aplicación. Esta evaluación comparativa permite a los diseñadores y educadores seleccionar la ISA más adecuada para sus necesidades, considerando factores como la eficiencia energética, la complejidad del hardware, la compatibilidad y el soporte educativo.

Aunque arquitecturas como PowerPC, SPARC o MIPS desempeñaron un papel central en la evolución de la computación, su adopción ha disminuido significativamente en contextos industriales y académicos, debido al desplazamiento por plataformas con mayor soporte comercial y vigencia tecnológica [20]. Su menor vigencia actual responde al surgimiento de arquitecturas más eficientes y con mejor respaldo comercial, como x86, ARM y RISC-V, que han captado la atención tanto del mercado como del ámbito educativo [34], [36], [49], [50]. Por ello, esta sección se enfoca en aquellas arquitecturas que mantienen relevancia comercial o presentan un valor pedagógico significativo en el desarrollo de simuladores educativos.

2.3.1 Arquitectura x86

La arquitectura x86, desarrollada inicialmente por Intel, ha dominado el mercado de computadoras de escritorio y servidores durante décadas, gracias a su evolución constante y soporte del ecosistema de software [19]. Su conjunto de instrucciones (ISA,

por sus siglas en inglés Instruction Set Architecture) incluye una amplia gama de operaciones, lo que otorga flexibilidad, aunque complica el diseño del hardware. Este equilibrio entre compatibilidad y rendimiento hace que x86 sea una opción preferida para entornos donde la capacidad de procesamiento es prioritaria, como en servidores y estaciones de trabajo [19], [51].

2.3.2 Arquitectura ARM

Reconocida por su alta eficiencia energética, la arquitectura ARM es la columna vertebral de dispositivos móviles y sistemas embebidos. Basada en el paradigma de conjunto de instrucciones reducidas (RISC), ARM simplifica el diseño del hardware y optimiza el consumo energético, características que la posicionan como una opción preferente para aplicaciones como smartphones y tablets. Aunque su rendimiento máximo en tareas de cómputo intensivo suele ser inferior al de x86, su equilibrio entre eficiencia energética y capacidad computacional resulta decisivo en mercados donde la autonomía y la disipación térmica son factores críticos, como los dispositivos móviles y el IoT [37], [50].

2.3.3 Arquitectura RISC-V

Como arquitectura de código abierto, RISC-V ofrece una alternativa personalizable a los modelos propietarios, destacándose en entornos académicos y de desarrollo especializado. Su ISA flexible permite a los desarrolladores personalizar sistemas según necesidades específicas, haciéndola especialmente atractiva para investigación, educación y aplicaciones embebidas. Basada en principios RISC, RISC-V combina eficiencia energética con un diseño de hardware simplificado, y su creciente ecosistema la posiciona como una fuerte competidora frente a arquitecturas establecidas como ARM. No obstante, RISC-V enfrenta desafíos para su adopción masiva, en parte debido a la falta de estándares unificados, la fragmentación de su ecosistema y la limitada presencia de proveedores comerciales consolidados, lo que dificulta su despliegue en entornos productivos críticos. [34], [35], [49], [52].

2.3.4 Comparativa entre arquitecturas

Las características distintivas de cada arquitectura condicionan su idoneidad para diversas aplicaciones. Por ejemplo, mientras x86 sobresale en entornos de alto rendimiento, ARM domina en dispositivos móviles gracias a su eficiencia energética. Por su parte, arquitecturas como RISC-V han encontrado aplicaciones relevantes en sistemas embebidos, plataformas educativas y diseños personalizados, aunque su presencia comercial difiere notablemente. La selección adecuada de

una arquitectura impacta significativamente en el éxito de un proyecto, desde el diseño hasta su implementación final. Además, comprender las diferencias entre estas arquitecturas, en particular sus repertorios de instrucciones y principios de diseño, resulta fundamental en el ámbito educativo, dado que facilita el desarrollo de herramientas didácticas que simulan sus principios operativos y ayudan a los estudiantes a visualizar el funcionamiento real de los sistemas computacionales [37], [50].

Tabla 2.2: Aplicaciones de la simulación en distintos sectores

Sector	Aplicación	Beneficio principal
Automotriz	Pruebas de colisión virtuales	Reducción de costos y aumento de seguridad
Aeroespacial	Simuladores de vuelo	Entrenamiento sin riesgo
Medicina	Simulación de cirugías	Entrenamiento sin comprometer pacientes
Educación	Simuladores para arquitectura de computadoras	Comprensión de procesos abstractos

En el contexto de la enseñanza de arquitectura de computadoras, estas arquitecturas permiten abordar distintos niveles de complejidad y estilos de diseño, lo que resulta clave para la construcción de simuladores educativos efectivos.

2.4 Repertorio de instrucciones

El repertorio de instrucciones, o Instruction Set Architecture (ISA), es el conjunto de operaciones que un procesador puede ejecutar, incluyendo su representación binaria y el conjunto de reglas que definen la interacción entre el software y el hardware. El ISA define la interfaz entre el hardware y el software, abarcando instrucciones aritméticas, lógicas, de control y de manipulación de datos, así como los modos de direccionamiento y los formatos de instrucción. Por su influencia directa en el rendimiento, la eficiencia energética y la versatilidad del sistema, el ISA constituye un componente esencial en el diseño de arquitecturas de computadoras [19], [20], [36].

2.4.1 Características clave del ISA

Entre las características fundamentales a considerar en el diseño de un repertorio de instrucciones se encuentran las siguientes [19]:

- **Tipos de operandos:** representan los datos que las instrucciones pueden manipular, como enteros, números en punto flotante, caracteres y direcciones de memoria. Un ISA eficiente debe soportar una amplia variedad de operandos para maximizar su versatilidad.
- **Tipos de operaciones:** incluyen las operaciones que el procesador puede realizar, como aritméticas (suma, resta), lógicas (AND, OR), de control (saltos, llamadas a subrutinas) y de manipulación de datos (almacenamiento, carga). Diversos autores destacan que un ISA bien diseñado debe lograr un equilibrio entre funcionalidad, simplicidad y eficiencia de implementación, aspectos fundamentales en el diseño de arquitecturas modernas [19], [36].
- **Modos de direccionamiento:** determinan cómo se especifican los operandos en las instrucciones. Entre los modos más comunes se encuentran el inmediato, directo, indirecto, mediante registros, con desplazamiento y basado en pila. Cada uno ofrece distintos niveles de eficiencia, flexibilidad y complejidad, siendo fundamentales para optimizar el acceso a datos y la ejecución de instrucciones.
- **Formato de las instrucciones:** que definen las reglas para acceder a los operandos dentro de las instrucciones, se exploran con mayor detalle en la siguiente subsección.



Figura 2.4: Características repertorio de instrucciones

El diseño de un repertorio de instrucciones eficiente y versátil es un desafío complejo que requiere un equilibrio entre funcionalidad, rendimiento y facilidad de uso. La selección adecuada de operandos, operaciones y modos de direccionamiento, junto con un formato de instrucción bien estructurado, son aspectos fundamentales para lograr una arquitectura de computadoras efectiva y adaptable a diversas aplicaciones. Estas características no solo definen las capacidades funcionales de un procesador, sino que también condicionan la manera en que las instrucciones interactúan con la memoria.

y los registros. A continuación, se profundiza en los modos de direccionamiento, uno de los elementos que más influye en la flexibilidad y eficiencia del repertorio de instrucciones.

2.4.2 Modos de direccionamiento

Los modos de direccionamiento definen los mecanismos mediante los cuales una instrucción especifica la ubicación de sus operandos, permitiendo así al procesador acceder a los datos en memoria o registros en tiempo de ejecución. A continuación, se describen los modos de direccionamiento más comúnmente implementados en las arquitecturas modernas [19], [20]:

- a) **Inmediato:** el operando está directamente incluido en la instrucción, permitiendo acceso rápido a valores constantes. Es eficiente para operaciones simples, aunque limitado a operandos pequeños.
- b) **Directo:** la instrucción contiene la dirección de memoria del operando. Es fácil de usar, pero está restringido por el rango de direcciones accesibles.
- c) **Indirecto:** la instrucción apunta a una dirección que contiene la ubicación real del operando, lo que amplía el rango de direcciones a costa de un acceso adicional a memoria.
- d) **Registro:** el operando se encuentra en un registro del procesador, proporcionando acceso extremadamente rápido, pero limitado por la cantidad de registros disponibles.
- e) **Registro Indirecto:** similar al modo indirecto, pero la dirección efectiva se obtiene a partir del contenido de un registro, lo que ofrece un buen equilibrio entre velocidad de acceso y capacidad de direccionamiento.
- f) **Con Desplazamiento:** combina una dirección base con un valor de desplazamiento, ideal para estructuras como arrays y matrices.
- g) **Pila:** el operando está en la parte superior de la pila, útil para gestionar subrutinas y el paso de parámetros.

Para complementar la descripción anterior, la Figura 2.5 presenta una representación esquemática de los modos de direccionamiento, mostrando gráficamente cómo se calcula la dirección efectiva (EA) en cada caso [20].

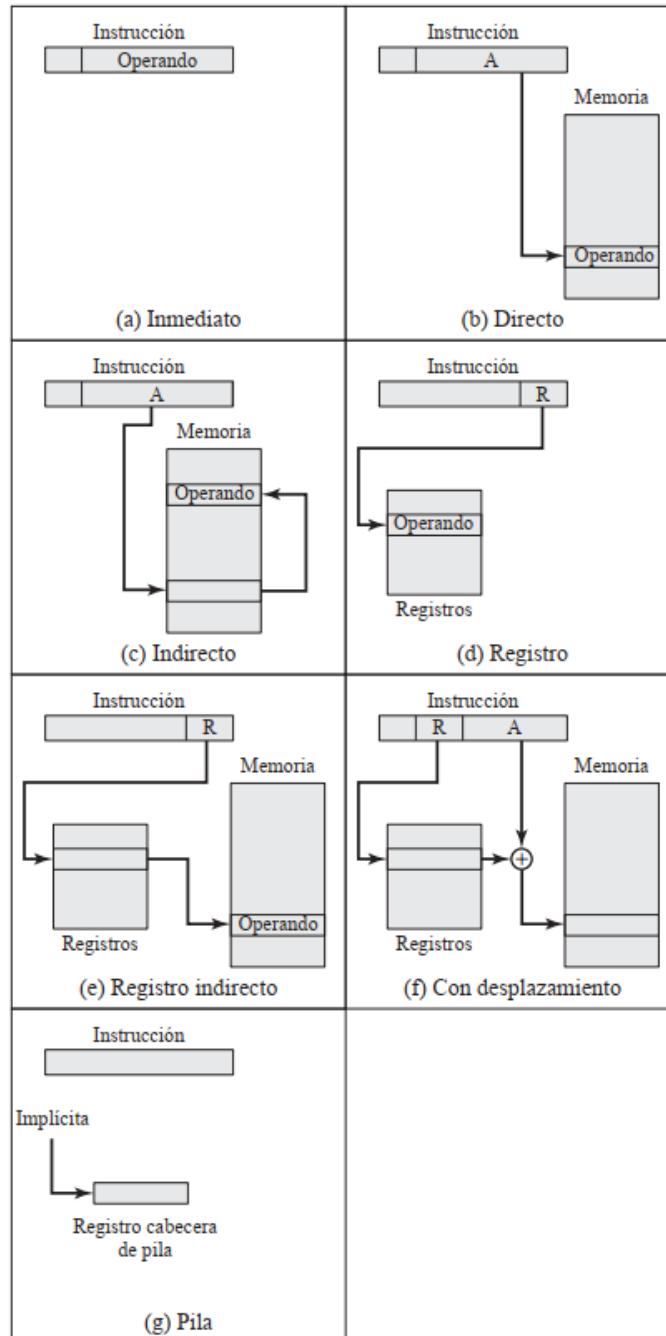


Figura 2.5: Modos de direccionamiento

- A = contenido de un campo de dirección en la instrucción
- R = contenido de un campo de dirección en la instrucción que referencia a un registro
- EA = dirección real (efectiva) de la posición que contiene el operando que se referencia

La tabla 2.3 detalla el cálculo de la dirección para cada modo de direccionamiento.

Tabla 2.3: Modos de direccionamiento básicos

Modo	Algoritmo	Ventaja	Desventaja
Inmediato	Operando $\leftarrow A$	No referencia a memoria	Operando de magnitud limitada
Directo	EA $\leftarrow A$	Es sencillo	Espacio de direcciones limitado
Indirecto	EA $\leftarrow (A)$	Espacio de direcciones grande	Referencias múltiples a memoria
Registro	EA $\leftarrow R$	No referencia a memoria	Número limitado de registros
Indirecto con registro	EA $\leftarrow (R)$	Espacio de direcciones grande	Referencia extra a memoria
Con desplazamiento	EA $\leftarrow A + (R)$	Flexibilidad	Complejidad
Pila	EA \leftarrow puntero de pila	No referencia a memoria	Aplicabilidad limitada

2.4.3 Formato de las instrucciones

El formato de las instrucciones especifica la disposición y codificación de los elementos que conforman una instrucción, como el código de operación (opcode), los operandos, los modos de direccionamiento y otros campos auxiliares. Esta organización impacta directamente en la facilidad de decodificación y en el rendimiento del procesador. Este formato afecta la rapidez de decodificación y la eficiencia general del procesador [19], [31]:

- **Longitud de la instrucción:** puede ser fija o variable. Las instrucciones de longitud fija permiten una decodificación más rápida y simplifican la lógica de control del procesador. En cambio, las instrucciones de longitud variable permiten una codificación más eficiente del espacio de memoria, a costa de una mayor complejidad en la etapa de decodificación.
- **Cantidad de operandos:** las instrucciones pueden trabajar con diferentes números de operandos (de 0 a 3 o más). Una mayor cantidad de operandos incrementa la expresividad de las instrucciones, pero también puede derivar en una mayor complejidad de codificación y en un mayor uso de recursos del procesador.
- **Campos de instrucción:** incluyen el opcode y campos adicionales como operandos, modos de direccionamiento y flags de condición. Estos campos determinan cuántas y qué tipo de operaciones puede ejecutar el procesador en un ciclo de reloj.

La Figura 2.6 muestra un ejemplo representativo de formato de instrucción, donde se visualizan los campos que la componen y su disposición en el código binario.

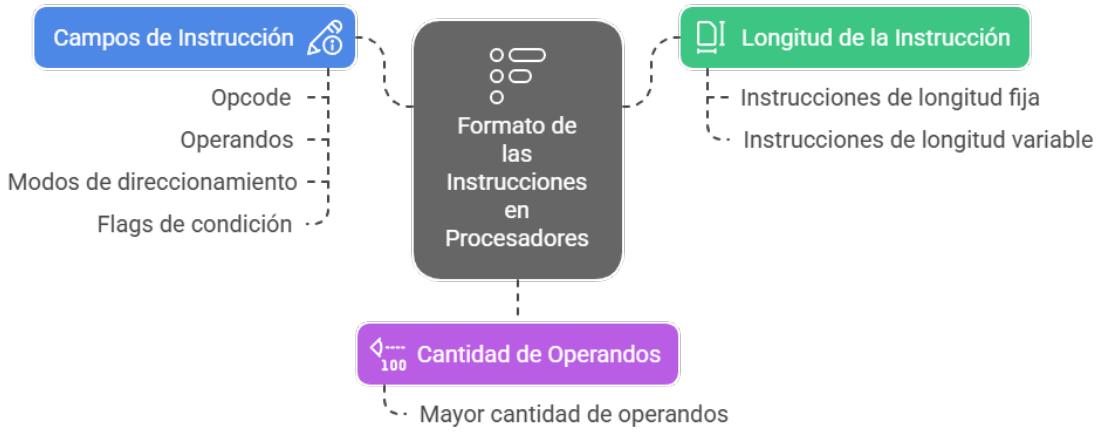


Figura 2.6: Formato de instrucciones

2.4.4 Comparativa de repertorios de instrucciones

La siguiente tabla 2.4 resume las características principales de los repertorios de instrucciones en tres arquitecturas ampliamente utilizadas: x86, ARM y RISC-V. Se consideran aspectos como la longitud de las instrucciones, la cantidad de operandos, su complejidad y los modos de direccionamiento que permiten.

Tabla 2.4: Comparativa de repertorios de instrucciones reales

Arquitectura	Longitud instrucción	Nº operandos	Tipos de operandos	Modos de direccionamiento
x86	Variable	0-3+	Complejos	Muchos
ARM	Fija (32 bits)	3	Simples	Limitados
RISC-V	Fija (32 bits)	3	Simples	Extensible

2.5 Filosofías CISC y RISC

El diseño del repertorio de instrucciones es una decisión estratégica clave en la arquitectura de procesadores, ya que determina no solo el rendimiento del sistema,

sino también la complejidad del hardware y del software, en particular del compilador. Dos de las filosofías más influyentes en este campo son **CISC (Complex Instruction Set Computing)** y **RISC (Reduced Instruction Set Computing)**. Mientras que **CISC** prioriza la reducción del número de instrucciones necesarias para realizar tareas complejas mediante operaciones multifuncionales, **RISC** simplifica el conjunto de instrucciones con el objetivo de maximizar la velocidad y la eficiencia energética. En esta sección se analizan ambos enfoques y sus implicaciones en el diseño de procesadores [19], [37].

El debate entre las filosofías CISC y RISC se remonta a fines de la década de 1970, cuando se comenzaron a cuestionar los beneficios reales de los repertorios de instrucciones complejos. Mientras las primeras generaciones de computadoras buscaban reducir el número de instrucciones por programa, investigaciones posteriores demostraron que un conjunto reducido y eficiente de instrucciones podía mejorar significativamente el rendimiento al simplificar la ejecución y optimizar el uso del hardware.

La evolución de los procesadores ha llevado a un enfoque más equilibrado, donde las arquitecturas modernas combinan elementos de ambas filosofías. Las arquitecturas modernas tienden a incorporar elementos de ambas filosofías. Por ejemplo, x86 adopta técnicas de ejecución interna propias de RISC para aumentar su rendimiento, mientras que procesadores RISC como ARM han introducido extensiones complejas para tareas específicas, acercándose parcialmente al enfoque CISC [19], [37].

2.5.1 CISC

Las arquitecturas **CISC**, como la **x86**, se caracterizan por su enfoque en reducir el número de instrucciones requeridas para completar operaciones complejas. Esto se logra mediante la inclusión de instrucciones que combinan múltiples operaciones en un solo ciclo. Como resultado, los programadores necesitan escribir menos líneas de código para alcanzar un objetivo específico.

Sin embargo, este diseño implica ciertas desventajas. La **decodificación y ejecución** de instrucciones CISC requiere un hardware considerablemente más complejo, y las instrucciones de longitud variable, típicas de estas arquitecturas, pueden aumentar el tiempo de decodificación. Esto genera cuellos de botella en el pipeline y limita el rendimiento.

Un ejemplo representativo es la arquitectura x86, que ha incorporado técnicas internas de ejecución similares a RISC —como la descomposición de instrucciones mediante microcódigo— con el fin de mejorar el rendimiento sin abandonar su repertorio complejo. Utiliza microcódigo para descomponer las instrucciones complejas en operaciones más simples, parecidas a las de un procesador RISC. Aunque esta estrategia mejora la eficiencia de ejecución en algunos casos, el diseño sigue siendo más costoso en términos de consumo energético y complejidad [37].

En consecuencia, el diseño del repertorio de instrucciones —incluyendo operaciones, modos de direccionamiento y formatos— constituye la interfaz crítica entre hardware y software, afectando tanto la eficiencia de ejecución como la expresividad de los programas. Su diseño influye directamente en la eficiencia del procesamiento y en la forma en que los programas interactúan con la arquitectura subyacente, lo que refuerza su relevancia en el estudio de la arquitectura de computadoras.

2.5.2 RISC

Las arquitecturas basadas en RISC, en contraste con CISC, se caracterizan por emplear instrucciones simples y de longitud fija. Esta simplificación facilita la decodificación y permite que muchas instrucciones se ejecuten en un solo ciclo de reloj. Además, esta filosofía favorece la implementación de técnicas avanzadas como el pipelining y la predicción de ramas, optimizando así el rendimiento.

A nivel de hardware, RISC prioriza la eficiencia energética, una característica crucial en dispositivos móviles y sistemas embebidos. Por ello, procesadores como los basados en ARM han dominado estos mercados, especialmente en dispositivos móviles, debido a su bajo consumo energético. La simplicidad y el bajo CPI (ciclos por instrucción) han sido factores determinantes en su adopción [49].

2.5.3 Comparativa entre CISC y RISC

Las diferencias entre CISC y RISC son evidentes tanto a nivel de diseño como de implementación. En las arquitecturas RISC, las instrucciones tienen una longitud fija, lo que simplifica la decodificación, reduce la latencia y mejora la predictibilidad del rendimiento. Además, este formato mejora la eficiencia del uso de la memoria caché, al ocupar menos espacio y facilitar accesos más rápidos.

En cambio, las arquitecturas CISC, como x86, emplean instrucciones de longitud variable, lo que les permite ofrecer una mayor flexibilidad y un repertorio más amplio de operaciones. Sin embargo, esta flexibilidad conlleva un mayor tiempo de decodificación y una complejidad adicional en la implementación del pipeline. Esto puede causar problemas como interrupciones en el flujo debido a errores de predicción de ramas, aunque se mitiguen mediante técnicas avanzadas como la predicción dinámica de saltos y el prefetching [31].

Por ejemplo, en RISC, los modos de direccionamiento son simples y permiten un acceso más rápido a los operandos, reduciendo la latencia en el pipeline [20]. En CISC, los modos de direccionamiento más complejos proporcionan flexibilidad a costa de una mayor latencia, lo que impacta negativamente en el rendimiento general del sistema.

Ejemplos de instrucciones

Para ilustrar la diferencia entre ambas filosofías, se presenta el siguiente ejemplo: cargar dos valores de memoria, sumarlos y almacenar el resultado en una dirección de memoria.

Una arquitectura bajo la filosofía RISC es RISC-V, que utiliza instrucciones simples y de longitud fija. En este caso, la instrucción para cargar un valor de memoria en un registro es **lw** (load word), y la instrucción para almacenar el resultado es **sw** (store word). La suma se realiza con la instrucción **add**:

```

1 # Cargar el valor de mem1 en el registro t0 (R1)
2 lw t0, 0(mem1)      # t0 = MEM[mem1]
3
4 # Cargar el valor de mem2 en el registro t1 (R2)
5 lw t1, 0(mem2)      # t1 = MEM[mem2]
6
7 # Sumar los registros t0 y t1, guardar el resultado en t2 (R3)
8 add t2, t0, t1      # t2 = t0 + t1
9
10 # Guardar el resultado en mem1
11 sw t2, 0(mem1)      # MEM[mem1] = t2
12

```

Una arquitectura bajo la filosofía CISC es x86, que utiliza instrucciones más complejas y de longitud variable. En este caso, la instrucción para cargar un valor de memoria en un registro es **MOV**, y la instrucción para almacenar el resultado es también **MOV**. La suma se realiza con la instrucción **ADD**:

```

1 ; Cargar el valor almacenado en mem1 en el registro EAX
2 MOV EAX, [mem1]
3
4 ; Sumar el valor almacenado en mem2 al registro EAX
5 ADD EAX, [mem2]
6
7 ; Guardar el resultado de la suma de vuelta en mem1
8 MOV [mem1], EAX
9

```

La tabla 2.5 sintetiza las principales diferencias estructurales y operativas entre las filosofías CISC y RISC, destacando sus implicancias en el diseño del hardware y el rendimiento general del sistema.

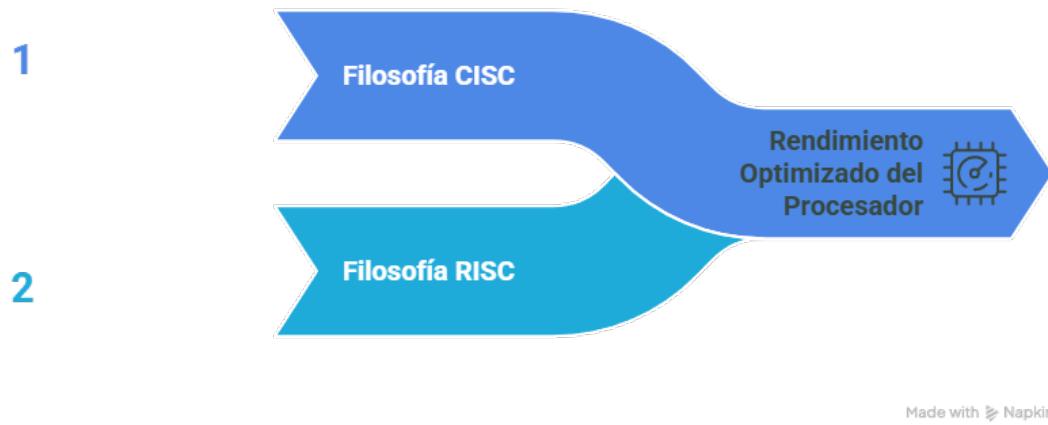
Tabla 2.5: Comparativa entre CISC y RISC

Aspecto	CISC	RISC
Objetivo principal	Minimizar el número de instrucciones para operaciones complejas	Simplificar el conjunto de instrucciones para optimizar velocidad y eficiencia energética
Tipo de instrucciones	Instrucciones complejas, longitud variable	Instrucciones simples, longitud fija
Decodificación y ejecución	Requiere hardware más complejo, posibles cuellos de botella en el pipeline	Decodificación más sencilla, facilita el uso de técnicas avanzadas como pipelining
Longitud de instrucciones	Longitud variable, puede aumentar el tiempo de decodificación	Longitud fija, simplifica la decodificación y mejora la predictibilidad del rendimiento
Eficiencia energética	Menor eficiencia energética en comparación con RISC	Mayor eficiencia energética, especialmente en dispositivos móviles
Modos de direccionamiento	Flexibilidad a costa de mayor latencia	Acceso más rápido a los operandos, menor latencia

Convergencia de filosofías

A pesar de sus diferencias, las arquitecturas modernas tienden a integrar características de ambas filosofías. Por ejemplo, los procesadores x86 adoptan técnicas propias de RISC para mejorar la eficiencia energética y el rendimiento. Esta convergencia refleja cómo los avances en diseño de procesadores buscan combinar lo mejor de cada enfoque, maximizando la flexibilidad y la eficiencia para adaptarse a las necesidades actuales y futuras del mercado.

La Figura 2.7 muestra la convergencia entre estas dos filosofías:



Made with Napkin

Figura 2.7: Convergencia de filosofías

En síntesis, las filosofías CISC y RISC representan enfoques contrastantes pero complementarios en el diseño de arquitecturas de procesadores. Su comprensión no solo es esencial para analizar el rendimiento y la eficiencia energética de los sistemas modernos, sino también para formar una base sólida en la enseñanza de arquitectura de computadoras, especialmente en contextos donde se emplean simuladores didácticos.

2.6 Arquitectura x86

La arquitectura x86, reconocida por su amplia adopción en computadoras personales, estaciones de trabajo y servidores, se introdujo en 1978 con el procesador Intel 8086, basado en una arquitectura de 16 bits. Desde entonces, ha evolucionado en capacidad y complejidad, con hitos clave como la introducción del Intel 80386 (32 bits) en 1985 y la extensión a 64 bits con AMD64 en 2003. Esta evolución ha permitido mejoras significativas en el rendimiento, el direccionamiento de memoria y la compatibilidad con aplicaciones exigentes. [20], [22], [23], [24], [53], [54].

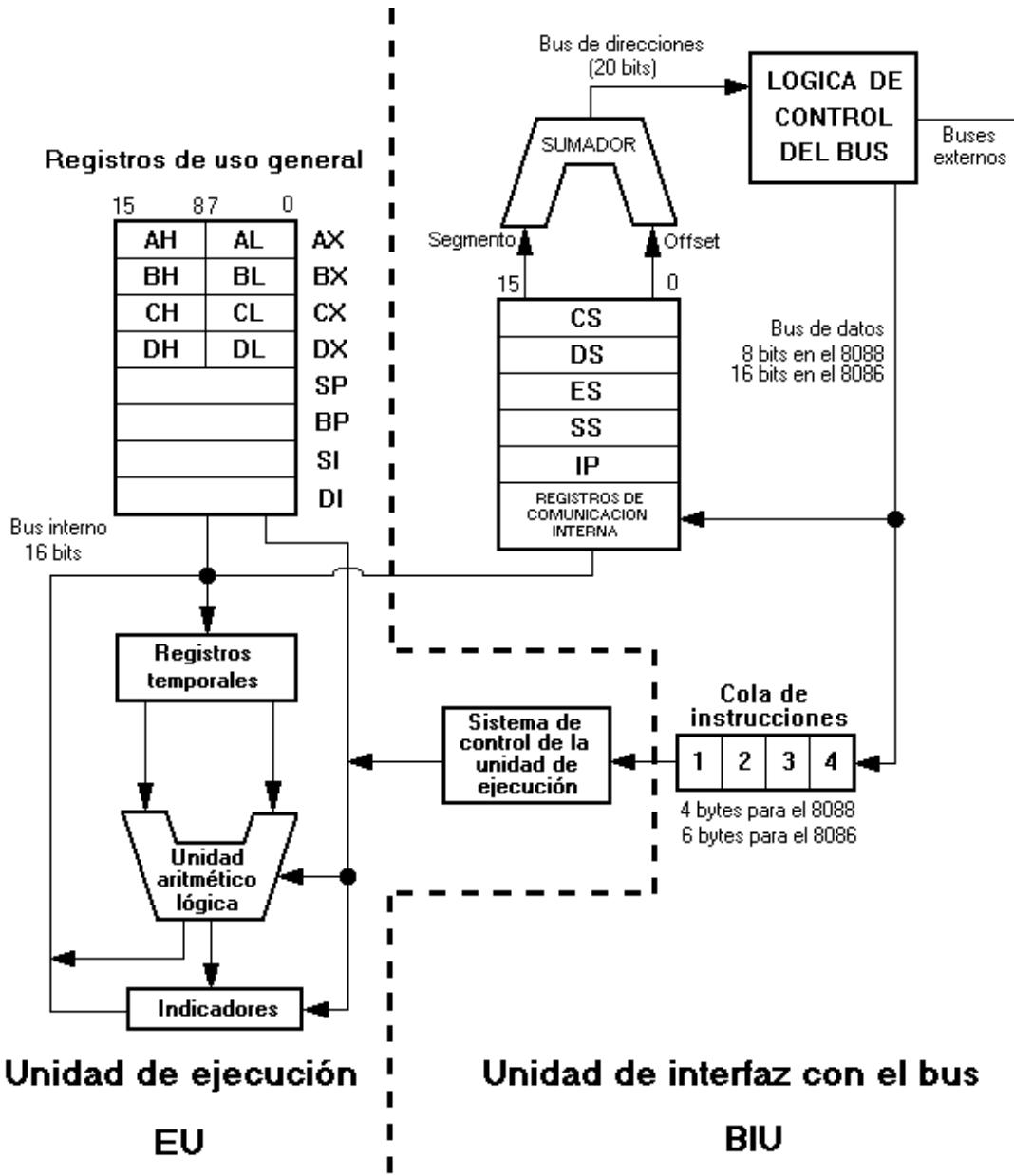


Figura 2.8: Diagrama esquemático microprocesador Intel 8086

2.6.1 Evolución de la arquitectura x86

Uno de los pilares del éxito de la arquitectura x86 ha sido su retrocompatibilidad, permitiendo la ejecución de aplicaciones de 16, 32 y 64 bits en un mismo sistema. Dicha propiedad no solo ha garantizado la continuidad del ecosistema x86, sino que también ha protegido las inversiones en software y sistemas operativos, una característica fundamental en entornos empresariales y académicos.

A continuación, se presenta la tabla 2.6 que resume los hitos clave en la evolución de los procesadores x86:

Tabla 2.6: Hitos en la evolución x86

Procesador	Año de Lanzamiento	Número de Bits	Extensiones de 64 bits
Intel 8086	1978	16	Arquitectura inicial
Intel 80386	1985	32	Memoria virtual
AMD64	2003	64	Extensiones de 64 bits

La tabla 2.7 muestra cómo la evolución de x86 ha estado marcada por avances tecnológicos que han impulsado la informática hacia nuevas fronteras:

Tabla 2.7: Línea de Tiempo de la Evolución de la Arquitectura x86

Año	Procesador	Innovación
1978	Intel 8086	Introducción de la arquitectura x86, 16 bits
1982	Intel 80286	Modos de operación adicionales
1985	Intel 80386	Arquitectura de 32 bits, memoria virtual
1989	Intel 80486	Unidad de punto flotante integrada, mejor caché
1993	Intel Pentium	Ejecución superescalar, predicción de saltos
1995	Intel Pentium Pro	Ejecución fuera de orden, caché L2 integrada
2003	AMD64	Extensiones a 64 bits, mayor acceso a memoria
2006	Intel Core	Optimización de rendimiento y eficiencia energética

2.6.2 Repertorio de instrucciones x86

La arquitectura x86 destaca por su complejidad y flexibilidad, reflejada en un repertorio de instrucciones extenso y de longitud variable. Esto contrasta con arquitecturas RISC, donde predominan instrucciones de longitud fija y decodificación sencilla [19], [53]. Aunque esta flexibilidad implica una mayor capacidad expresiva y compatibilidad hacia atrás, también introduce desafíos de diseño, tales como la necesidad de decodificadores complejos, técnicas de predicción de instrucciones y ejecución fuera de orden para lograr un rendimiento competitivo.

Estructura de una instrucción x86

Una instrucción típica de x86 puede incluir los siguientes componentes [20]:

- **Prefijos:** modifican la operación principal de la instrucción. Por ejemplo, el prefijo 0x66 cambia el tamaño del operando.
- **Código de operación (Opcode):** indica la operación a realizar. Por ejemplo, 0x89 corresponde MOV.
- **Modificadores de dirección (ModR/M y SIB):** definen registros y direccionamiento. El byte **SIB** (Scale, Index, Base) es especialmente útil para operaciones complejas, como el acceso a matrices.
- **Desplazamiento e inmediato:** Agregan flexibilidad en el manejo de datos, aunque aumentan la complejidad.

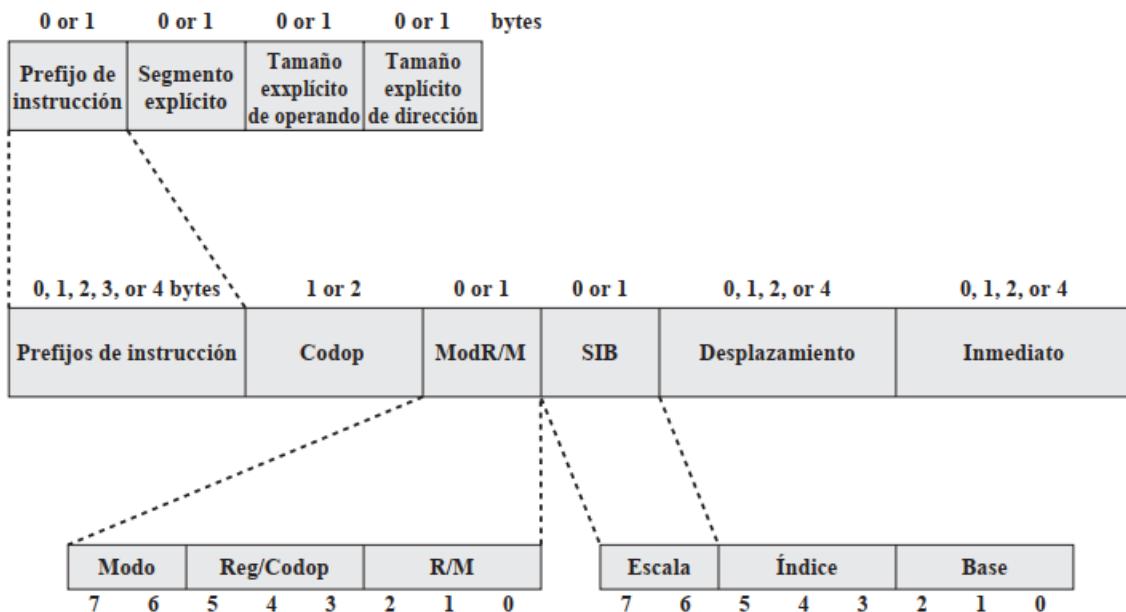


Figura 2.9: Formato de instrucciones del Pentium x86

Un ejemplo típico de instrucción es:

```

1 ; Carga en el registro AX el valor almacenado en la dirección de memoria
2 ; que resulta de sumar el contenido de los registros BX y SI
3 ; más el desplazamiento 16.
4 MOV AX, [BX+SI+16]
5

```

Esta instrucción utiliza varios componentes, que el procesador debe decodificar antes de ejecutarla. Aunque esta flexibilidad es una ventaja en términos de funcionalidad, requiere técnicas avanzadas, como predicción de saltos y paralelización, para mantener la eficiencia en procesadores modernos [19], [37], [53].

2.7 Lenguaje máquina y lenguaje ensamblador

El lenguaje máquina es el conjunto de instrucciones que un procesador puede entender y ejecutar directamente. Cada procesador tiene su propio conjunto de instrucciones, que se representan en forma de números binarios. Estas instrucciones son específicas para cada arquitectura y están diseñadas para realizar operaciones básicas como sumar, restar, mover datos entre registros y acceder a la memoria. El lenguaje máquina es el nivel más bajo de programación y está compuesto por secuencias de bits que representan operaciones y operandos específicos del procesador [19], [55].

El procesador ejecuta directamente las instrucciones codificadas en lenguaje máquina, sin requerir traducción desde niveles superiores de abstracción. Sin embargo, la escritura manual de código en lenguaje máquina es un proceso extremadamente laborioso, propenso a errores y difícil de mantener. Cada instrucción debe representarse como una cadena precisa de ceros y unos. Esta codificación depende de las reglas específicas del procesador, que incluyen los modos de direccionamiento, los formatos de instrucción y la organización de la memoria [20], [31], [36], [55].

Por ejemplo, si un estudiante o desarrollador deseara sumar dos números en lenguaje máquina, tendría que especificar manualmente cada secuencia binaria correspondiente a la operación de suma, así como las direcciones de memoria donde se encuentran los operandos. Este enfoque no solo es tedioso, sino que también aumenta la probabilidad de errores, especialmente cuando se requiere modificar o depurar el código.

Ante las limitaciones del lenguaje máquina en términos de legibilidad y mantenibilidad, se desarrolló un lenguaje de bajo nivel con mayor legibilidad que el lenguaje máquina que permitiera al programador escribir instrucciones de forma más comprensible: el lenguaje ensamblador. Este lenguaje permite a los programadores escribir instrucciones más comprensibles mediante mnemónicos simbólicos, que actúan como representaciones legibles de las instrucciones en lenguaje máquina. Cada arquitectura de procesador define su propio conjunto de instrucciones (ISA, Instruction Set Architecture), lo que implica que el lenguaje ensamblador asociado debe ajustarse a la codificación binaria, modos de direccionamiento y sintaxis específicos de dicha ISA [20].

En el ámbito educativo, el lenguaje ensamblador se destaca como una herramienta fundamental para comprender cómo se comunican el software y el hardware [31], [36]. Permite a los estudiantes visualizar la ejecución de instrucciones individuales, analizar el uso de registros y explorar la estructura de la memoria, convirtiéndose en un recurso valioso para este propósito.

Un programa en lenguaje ensamblador suele estar compuesto por instrucciones que especifican un mnemónico, uno o más operandos, y eventualmente el modo de direccionamiento. Por ejemplo:

¹ ; Carga el valor inmediato 5 en el registro AX.

```
2 MOV AX, 5
3
4 ; Sumar los registros BX y AX, guarda el resultado en AX.
5 ADD AX, BX
6
```

Estas líneas indican que el valor 5 se mueve al registro AX y luego se suma el contenido de BX. A través de este tipo de instrucciones, el estudiante puede visualizar de forma explícita cómo opera el procesador sobre sus registros y memoria.

2.7.1 Ensamblador

El ensamblador es un programa que traduce las instrucciones simbólicas escritas en lenguaje ensamblador a lenguaje máquina, es decir, las convierte en las secuencias binarias que el procesador puede interpretar y ejecutar. Este proceso de traducción es prácticamente directo, ya que existe una correspondencia uno a uno entre las instrucciones en ensamblador y las instrucciones en lenguaje máquina [20], [31]. En contraste, los lenguajes de programación de alto nivel, como C o Python, suelen generar múltiples instrucciones máquina por cada línea de código fuente, lo que los distancia más de la arquitectura subyacente [19].

La Figura 2.10 muestra el proceso de traducción de un programa en lenguaje ensamblador a lenguaje máquina. En este proceso, el ensamblador toma cada línea de código en ensamblador y la convierte en su representación binaria correspondiente, generando así un archivo ejecutable que puede ser cargado y ejecutado por el procesador.



Made with Napkin

Figura 2.10: Proceso de ensamblado

2.7.2 Ensambladores x86

En el caso de la arquitectura x86, los programadores pueden elegir entre diversos ensambladores, como TASM (Turbo Assembler) [56], MASM (Microsoft Macro Assembler) [57] y NASM (Netwide Assembler) [58]. Aunque cada ensamblador tiene características y sintaxis particulares, todos comparten el objetivo fundamental de convertir las instrucciones ensamblador en código binario ejecutable por los procesadores x86 [59].

A continuación, se presenta una tabla comparativa 2.8 que resume las principales características de tres ensambladores ampliamente utilizados en la arquitectura x86.

La información compilada permite visualizar diferencias relevantes en términos de sintaxis, compatibilidad, funcionalidades adicionales y contexto de uso, lo que resulta particularmente útil al momento de seleccionar herramientas adecuadas para entornos educativos o de desarrollo de bajo nivel.

Tabla 2.8: Comparación de ensambladores arquitectura x86

Característica	TASM	MASM	NASM
Desarrollador	Borland	Microsoft	Simon Tatham et al.
Año de lanzamiento	1985	1981	1996
Sistema operativo	MS-DOS, Windows	MS-DOS, Windows	Multiplataforma (Windows, Linux, macOS)
Sintaxis	Sintaxis similar a Intel con extensiones	Sintaxis de Intel con soporte avanzado	Sintaxis de Intel, modular y extensible
Soporte de macros	Macros y directivas avanzadas	Macros y directivas extensivas	Macros avanzadas y preprocessamiento
Compatibilidad	Compatibilidad con x86 antiguo	Compatibilidad con x86 antiguo	Compatibilidad con x86, x86-64 y otros
Capacidades adicionales	Integración con herramientas Borland	Integración con Visual Studio	Soporte para múltiples formatos (binario, ELF, etc.)
Licencia	Comercial	Comercial	Código abierto
Uso actual	Menos común, usado en entornos heredados	Ampliamente usado en desarrollo Windows	Popular en sistemas y software libre

Capítulo 3

Simulación

En este capítulo se analiza el papel de la simulación desde una perspectiva didáctica, destacando su relevancia como herramienta de apoyo en la enseñanza de Arquitectura de Computadoras. Se abordan los beneficios que ofrecen los simuladores en el proceso educativo y los desafíos que ayudan a superar en la formación de los estudiantes.

3.1 Introducción a la simulación

La simulación constituye una herramienta esencial en múltiples dominios, incluidos la medicina, la defensa, el entretenimiento y particularmente la educación, debido a su capacidad para representar procesos complejos y facilitar la toma de decisiones en entornos seguros y controlados. Su principal valor radica en su capacidad para modelar sistemas complejos, generar hipótesis, realizar análisis predictivos y explorar escenarios de manera segura y eficiente.

Banks define la simulación como el proceso de replicar el comportamiento de un sistema a lo largo del tiempo mediante un modelo conceptual que representa sus características y dinámicas principales [2]. Estos modelos evolucionan simulando las interacciones entre sus componentes, lo que permite estudiar su respuesta ante diferentes variables y escenarios [4].

La posibilidad de analizar sistemas complejos sin intervenir directamente en ellos convierte a la simulación en una herramienta indispensable en el contexto actual, marcado por el avance de la tecnología y la creciente complejidad de los sistemas. Además, la simulación permite optimizar diseños, prever comportamientos y reducir los costos de desarrollo antes de implementar soluciones reales [3], [29].

3.1.1 Aplicaciones de la simulación en la industria

En sectores como la industria automotriz, la simulación es fundamental para el diseño y prueba de sistemas de seguridad, como airbags y frenos. Gracias a modelos virtuales, se realizan pruebas de colisión y análisis de rendimiento sin necesidad de recurrir a costosas pruebas físicas. Asimismo, la simulación permite optimizar diseños de motores, analizar el flujo aerodinámico y prever el comportamiento de materiales en condiciones extremas, contribuyendo a mejorar tanto la eficiencia como la seguridad de los vehículos [60].

En la aviación, los simuladores de vuelo son esenciales para entrenar pilotos, replicando condiciones reales de vuelo sin riesgos. Durante el diseño de aeronaves, estas herramientas permiten evaluar la aerodinámica y el rendimiento en diversos entornos, reduciendo significativamente el tiempo y los costos de desarrollo mientras incrementan la seguridad [61].

Estos principios generales encuentran aplicaciones concretas en diversos sectores industriales, donde la simulación cumple un papel clave tanto en el diseño como en el entrenamiento, la evaluación y la toma de decisiones.

Tabla 3.1: Aplicaciones de la simulación en distintos sectores

Sector	Aplicación	Beneficio principal
Automotriz	Pruebas de colisión virtuales	Reducción de costos y aumento de seguridad
Aeroespacial	Simuladores de vuelo	Entrenamiento sin riesgo
Medicina	Simulación de cirugías	Entrenamiento sin comprometer pacientes
Educación	Simuladores para arquitectura de computadoras	Comprensión de procesos abstractos

Estos ejemplos ilustran cómo la simulación contribuye significativamente a la optimización de procesos, la reducción de riesgos y la mejora continua en el desarrollo de sistemas complejos. Su uso no solo ha transformado sectores productivos, sino que también ofrece un modelo replicable en contextos educativos especializados, como la enseñanza de arquitectura de computadoras.

3.2 Simulación en la educación

En contextos educativos, la simulación se ha consolidado como una estrategia pedagógica eficaz para facilitar la comprensión de fenómenos complejos, especialmente en disciplinas que requieren alto nivel de abstracción y razonamiento sistemático. A

través de simuladores, los estudiantes pueden interactuar con sistemas virtuales y experimentar escenarios realistas, lo que mejora la comprensión de ideas abstractas y favorece la aplicación práctica de conocimientos teóricos [5].

En contraste con enfoques instruccionales tradicionales centrados en la transmisión de información, los simuladores favorecen metodologías activas basadas en el aprendizaje por descubrimiento, la resolución de problemas y la construcción significativa del conocimiento, las herramientas de simulación integran tecnologías que vinculan conceptos teóricos con situaciones reales. Esto promueve una pedagogía interactiva, basada en la resolución de problemas y el aprendizaje por descubrimiento, estimulando la exploración y el razonamiento inferencial [6].

En definitiva, la simulación enriquece la experiencia de aprendizaje al proporcionar una plataforma dinámica y participativa que facilita tanto la experimentación como la asimilación profunda de los contenidos.

3.2.1 El rol de la simulación en la enseñanza de Arquitectura de Computadoras

En la carrera de Licenciatura en Sistemas, la asignatura Arquitectura de Computadoras persigue varios objetivos esenciales:

- Comprender la estructura y funcionamiento de las computadoras.
- Conocer las diferentes arquitecturas de sistemas microprocesadores.
- Evaluar medidas de rendimiento y comparar arquitecturas.
- Analizar el impacto de la tecnología de las computadoras en contextos sociales y económicos.

Enseñar los fundamentos teóricos de la organización y arquitectura interna de las computadoras puede ser un reto debido a la complejidad de los procesos involucrados. Los estudiantes necesitan desarrollar altos niveles de abstracción para construir modelos mentales que les permitan entender conceptos como la ejecución de instrucciones, la gestión de memoria o la interacción entre componentes del sistema.

En este contexto, los simuladores se configuran como mediadores didácticos que permiten representar gráficamente procesos abstractos, facilitando la manipulación de parámetros y el análisis de resultados en un entorno seguro, repetible y sin restricciones físicas. Estas herramientas permiten a los alumnos experimentar con configuraciones y parámetros, observar su impacto en el rendimiento del sistema y explorar escenarios hipotéticos sin necesidad de hardware físico.

Además, la simulación actúa como un puente entre la teoría y la práctica, facilitando que los docentes refuerzen conceptos abstractos con experiencias concretas. En conjunto, estas ventajas hacen de la simulación una metodología pedagógica invaluable, promoviendo la experimentación y el aprendizaje activo en la enseñanza de Arquitectura de Computadoras [7], [12], [25].

3.3 El Formalismo DEVS (Discrete Event System Specification)

El formalismo DEVS es una metodología modular y jerárquica que permite modelar y analizar sistemas representables como sistemas de eventos discretos, continuos o híbridos. Desarrollado por Bernard P. Zeigler en la década de 1970, este enfoque amplía el concepto de las máquinas de Moore al añadir una estructura que permite representar el comportamiento de sistemas mediante eventos temporizados que provocan cambios de estado, capturando así tanto la dinámica interna como las interacciones externas del sistema [29].

3.3.1 Estructura del formalismo DEVS

El formalismo DEVS se basa en la representación de sistemas como una colección de componentes que interactúan entre sí a través de eventos. Cada componente tiene un estado interno y puede recibir eventos externos que provocan cambios en su estado. Estos eventos pueden ser temporizados, lo que significa que el sistema puede reaccionar a eventos en momentos específicos, o pueden ser desencadenados por condiciones específicas. Esta estructura permite capturar tanto el comportamiento interno como la interacción externa del sistema modelado, ver figura 3.1.



Figura 3.1: Relación entre modelos atómicos y modelos acoplados en DEVS

La estructura formal del modelo atómico DEVS se presenta en la ecuación (3.1):

$$M = \langle X, S, Y, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, \tau \rangle \quad (3.1)$$

Donde:

- X : entradas
- S : estados internos
- Y : salidas
- δ_{int} : transición interna
- δ_{ext} : transición externa
- λ : función de salida
- τ : tiempo de avance del estado

Esta estructura facilita el modelado de sistemas informáticos, permitiendo representar componentes como registros o unidades de control mediante modelos atómicos que se comunican a través de eventos discretos.

DEVS describe el comportamiento de un sistema real utilizando eventos de entrada y salida, así como transiciones entre estados definidos. Un sistema en este formalismo se compone de dos tipos principales de modelos:

- **Modelos atómicos:** representan las unidades fundamentales de comportamiento.
- **Modelos acoplados:** integran modelos atómicos y/o otros modelos acoplados, permitiendo la construcción jerárquica de sistemas más complejos.

Esta organización modular facilita el análisis y la gestión de sistemas, permitiendo probar subsistemas de manera aislada antes de integrarlos en un modelo completo.

La siguiente figura 3.2 ilustra la organización modular del formalismo DEVS, mostrando cómo se integran modelos atómicos dentro de modelos acoplados:

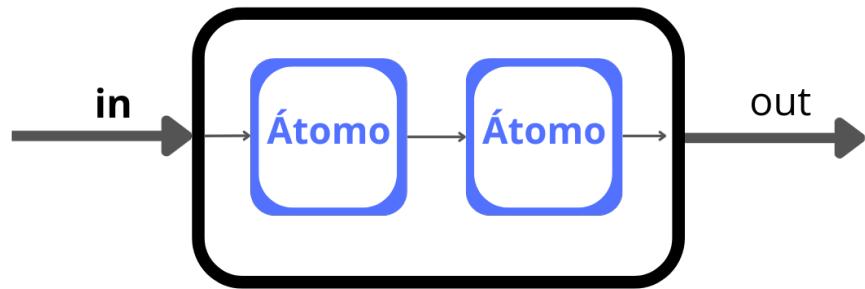


Figura 3.2: Modelo acoplado en DEVS

3.3.2 xDEVS

xDEVS es un framework de modelado y simulación de eventos discretos basado en DEVS/PDEVS que unifica una interfaz común entre múltiples lenguajes (C, C++, Java, Python, Rust, entre otros). Esta interoperabilidad permite reutilizar modelos y comparar motores, al tiempo que habilita ejecuciones secuenciales, paralelas y distribuidas, con énfasis en rendimiento y verificabilidad. En este trabajo se utiliza `xdevs.py`, la implementación en Python, por su equilibrio entre legibilidad, portabilidad y soporte de ejecución en tiempo virtual y real [62], [63], [64].

xDEVS separa explícitamente la capa de **modelado** (modelos atómicos y acoplados con puertos y acoplamientos) de la capa de **simulación** (simuladores y coordinadores que orquestan el avance temporal y la confluencia de eventos). Esta organización refleja la estructura formal DEVS y favorece la construcción jerárquica de sistemas complejos, manteniendo claridad en el flujo de información.

3.3.3 Aplicaciones del formalismo DEVS

“El formalismo DEVS encuentra aplicación en diversos ámbitos, como las redes de comunicación [65], donde permite simular el enrutamiento de paquetes y la congestión de redes; en entornos de manufactura [29], donde se modelan flujos de producción

y control de calidad; y en sistemas de transporte, para la optimización de flujos vehiculares [66]. También se utiliza en la simulación de sistemas biológicos, como la propagación de enfermedades o el comportamiento de poblaciones [67]. En el ámbito de la educación, DEVS se ha implementado en simuladores para la enseñanza de arquitectura de computadoras, permitiendo a los estudiantes explorar y comprender conceptos complejos mediante la visualización y manipulación de modelos [68].

Estas aplicaciones destacan su versatilidad para optimizar sistemas complejos en escenarios del mundo real.

xDEVS en la enseñanza de la Arquitectura de Computadoras

La implementación de entornos de simulación basados en DEVS en la enseñanza de arquitectura de computadoras aporta múltiples ventajas que enriquecen el proceso de aprendizaje:

- **Representación visual:** ofrece diagramas y representaciones dinámicas que ayudan a los estudiantes a visualizar y comprender procesos internos, como la ejecución de instrucciones y la gestión de recursos.
- **Interactividad:** permite modificar configuraciones y parámetros, fomentando la experimentación y mostrando el impacto directo de estas variables en el rendimiento del sistema.
- **Exploración de escenarios:** posibilita simular escenarios hipotéticos y evaluar el comportamiento de sistemas complejos sin la necesidad de hardware físico.

Estas funcionalidades enriquecen la experiencia educativa al integrar la teoría con la práctica y fomentar una participación activa en el análisis de los principios fundamentales de la arquitectura computacional. Al adoptar xDEVS como parte del entorno educativo, se potencia la capacidad de los estudiantes para abordar problemas complejos y explorar soluciones innovadoras [69], [70], [71].

En conclusión, el formalismo xDEVS no solo es una herramienta valiosa para el modelado y análisis de sistemas, sino que también representa un recurso poderoso para facilitar la enseñanza de conceptos complejos, como los que se encuentran en la arquitectura de computadoras.

Capítulo 4

Comparativa de simuladores

Este capítulo presenta un análisis comparativo de simuladores basados en la arquitectura x86, con el objetivo de determinar su adecuación para su integración en la asignatura Arquitectura de Computadoras de la Licenciatura en Sistemas de Información.

La selección y evaluación de estos simuladores se fundamenta en criterios específicos diseñados para medir su efectividad en un entorno educativo. El objetivo principal es identificar las herramientas que mejor respalden el proceso de enseñanza y aprendizaje. Los criterios definidos abarcan aspectos clave para la enseñanza de arquitectura de computadoras: facilidad de uso, funcionalidades del entorno de programación, calidad de los recursos de apoyo, mecanismos de ejecución de programas, precisión en la emulación de la arquitectura x86, características técnicas del software y su alineación con los contenidos curriculares.

Los resultados de esta investigación fueron publicados en el XVII Congreso de Tecnología en Educación y Educación en Tecnología (2022), en el trabajo titulado Herramientas de software para dar soporte en la enseñanza y aprendizaje de la arquitectura x86 [72].

4.1 Estudios similares

Existen antecedentes de estudios comparativos que evalúan simuladores aplicados a la enseñanza en cursos de arquitectura de computadoras: - “A survey and evaluation of simulators suitable for teaching courses in computer architecture and organization”, 2009 [17]: este estudio analiza simuladores considerando dos categorías principales. La primera, relacionada con las características de simulación, incluye criterios como granularidad, usabilidad, disponibilidad, presentación visual y flujo de simulación. La segunda categoría evalúa la cobertura de los contenidos

establecidos en los planes de estudio. - “Survey and evaluation of simulators suitable for teaching for computer architecture and organization Supporting undergraduate students at Sir Syed University of Engineering & Technology”, 2012 [18]: este trabajo evalúa aspectos como la usabilidad, disponibilidad, fundamentos de arquitectura informática, jerarquía de sistemas de memoria, comunicación e interfaz, y diseño de sistemas de procesadores.

A diferencia de los estudios mencionados, este trabajo propone una evaluación centrada exclusivamente en simuladores de arquitectura x86, mediante el uso de criterios diseñados ad hoc para analizar tanto las funcionalidades de simulación como su adecuación a los contenidos específicos de la asignatura Arquitectura de Computadoras dictada en la Licenciatura en Sistemas de la Universidad Nacional de Entre Ríos.

4.2 Simuladores bajo análisis

Un simulador de arquitectura es una herramienta de software que emula el hardware de un sistema de cómputo, permitiendo representar aspectos arquitectónicos y funcionales del mismo. Estos simuladores ofrecen un entorno controlado para realizar pruebas, modificaciones y ejecución de programas sin riesgo de dañar componentes físicos o enfrentar limitaciones de hardware [16].

Algunos simuladores destacan por proporcionar una representación visual e interactiva de la organización y arquitectura interna de una computadora, facilitando la comprensión de su funcionamiento. Algunos ejemplos relevantes de simuladores son: Assembly Debugger (x86), Simple 8-bit Assembler Simulator, Microprocessor Simulator, Simulador de ensamblador de 16 bits y Emu8086. Estas herramientas juegan un papel fundamental en el aprendizaje de la arquitectura de computadoras, al conectar conceptos teóricos con experiencias prácticas y simplificar abstracciones complejas, además de servir como soporte en la labor docente [17], [18], [19], [20], [73].

4.3 Criterios de evaluación

Los criterios de evaluación se definieron con el objetivo de realizar un análisis integral y sistemático de los simuladores seleccionados. A continuación, se presentan estos criterios junto con sus respectivos indicadores y escalas:

- **Usabilidad:** evalúa la facilidad de uso del simulador.
 - **Indicadores:**

- * Facilidad de aprendizaje (tiempo necesario para familiarizarse con la herramienta).
 - * Interfaz de usuario (claridad y organización).
 - * Documentación y ayuda (accesibilidad y calidad de tutoriales y guías).
 - **Escala:** Difícil - Media - Fácil.
- **Editor:** analiza las funcionalidades para escribir y depurar código ensamblador.
 - **Indicadores:**
 - * Capacidad de edición (resaltado de sintaxis, puntos de interrupción, etc.).
 - * Manejo de errores de sintaxis.
 - * Opciones de almacenamiento (guardar y cargar programas).
 - **Escala:** Baja - Media - Alta.
- **Documentación:** valora la disponibilidad y calidad de los recursos de aprendizaje proporcionados.
 - **Indicadores:**
 - * Manual de usuario.
 - * Tutoriales de aprendizaje.
 - * Exhaustividad en la descripción del repertorio de instrucciones.
 - **Escala:** Mínima - Media - Completa.
- **Ejecución de simulación:** mide la facilidad para controlar y observar la ejecución de programas.
 - **Indicadores:**
 - * Control de simulación (pausa, reanudación, retroceso).
 - * Visualización del flujo de ejecución.
 - * Configurabilidad (ajuste de parámetros como la velocidad del reloj).
 - **Escala:** Baja - Media - Alta.
- **Nivel de especificación de la Organización y Arquitectura del sistema simulado:** determina la precisión en la representación de la arquitectura x86.
 - **Indicadores:**
 - * Fidelidad en la representación de la arquitectura.
 - * Completitud del conjunto de instrucciones implementadas.
 - * Inclusión y funcionalidad de memoria y módulos de E/S.
 - **Escala:** Mínima - Media - Completa.
- **Características del producto software:** evalúa las propiedades generales del simulador.
 - **Indicadores:**
 - * Tipo de licencia (open source o privativa).

- * Frecuencia de actualizaciones.
- * Plataforma (aplicación web o de escritorio)
- **Escala:** Mala - Buena - Muy buena.
- **Cobertura de los contenidos preestablecidos en la currícula:** mide el grado en que el simulador abarca los contenidos de la asignatura.
 - **Indicadores:**
 - * Alineación con los tópicos del currículum.
 - * Profundidad en el tratamiento de los temas.
 - **Escala:** Baja - Media - Alta.

La Tabla 4.1 resume los criterios, indicadores y escalas utilizadas.

Tabla 4.1: Criterios e indicadores de evaluación de simuladores

Criterio	Indicadores	Escala
Usabilidad	Facilidad de aprendizaje, interfaz, documentación	Difícil - Media - Fácil
Funcionalidad del editor	Sintaxis, manejo de errores, guardar/cargar	Baja - Media - Alta
Calidad de la documentación	Manuales, tutoriales, repertorio de instrucciones	Mínima - Media - Completa
Ejecución de simulación	Control de simulación, visualización del flujo, configurabilidad	Baja - Media - Alta
Especificación de arquitectura x86	Fidelidad de la arquitectura, repertorio, memoria y E/S	Mínima - Media - Completa
Propiedades técnicas y de distribución	Licencia, actualizaciones, plataforma	Mala - Buena - Muy buena
Alineación con contenidos curriculares	Cobertura de tópicos, profundidad del tratamiento	Baja - Media - Alta

4.4 Selección de simuladores

Mediante una exploración exhaustiva de fuentes disponibles en línea, foros académicos y repositorios educativos, se identificaron los siguientes simuladores de arquitectura x86: Assembly Debugger (x86), Simple 8-bit Assembler Simulator, Microprocessor Simulator, Simulador de ensamblador de 16 bits, Emu8086, VonSim, Orga1 y Qsim. Estos simuladores fueron seleccionados por su relevancia en el ámbito educativo y su potencial para facilitar la enseñanza de la arquitectura x86.

La selección se basó en una evaluación preliminar que consideró el tiempo necesario para su análisis y el grado de cumplimiento de los criterios definidos, priorizando aquellos simuladores que ofrecieran un balance adecuado entre funcionalidad, usabilidad, documentación y alineación con los contenidos curriculares de la asignatura Arquitectura de Computadoras. De esta preselección, se eligieron tres herramientas que, a priori, cumplían con la mayor cantidad de criterios evaluativos: **Emu8086**, **VonSim** y **Simple 8-bit Assembler Simulator**.

Tabla 4.2: Proceso de selección de simuladores

Simulador	Exploración Previa	Evaluación Final
Assembly Debugger (x86)	✓	✗
Simple 8-bit Assembler Simulator	✓	✓
Microprocessor Simulator	✓	✗
Simulador de ensamblador de 16 bits	✓	✗
Emu8086	✓	✓
VonSim	✓	✓
Orga1	✓	✗
Qsim	✓	✗

4.5 Participantes en la evaluación

La evaluación fue llevada a cabo por un equipo conformado por tres docentes de la asignatura Arquitectura de Computadoras —Marcelo A. Colombani, José M. Ruiz y Amalia G. Delduca—, quienes aportaron su experiencia en el uso pedagógico de simuladores. Asimismo, se contó con la participación de un asesor externo, Marcelo A. Falappa, quien aportó una perspectiva independiente y validó tanto la metodología como los resultados obtenidos.

4.6 Análisis comparativo

A continuación, se presenta un análisis detallado de los simuladores seleccionados, basado en los criterios previamente establecidos:

4.6.1 Simple 8-bit Assembler Simulator

- **Usabilidad:** Nivel medio. Todos los componentes se muestran en una sola pantalla, lo que puede resultar abrumador para usuarios principiantes.
- **Editor:** Nivel bajo. Incluye notificaciones de errores de sintaxis al ensamblar, pero carece de resaltado de sintaxis, puntos de interrupción (breakpoints) y opciones para guardar o cargar programas.
- **Documentación:** Nivel mínimo. Consta solo de un manual de instrucciones implementadas.
- **Ejecución de simulación:** Nivel medio. Permite ajustar la velocidad del reloj de la CPU y proporciona controles básicos de simulación.
- **Nivel de especificación:** Nivel mínimo. Simplifica la arquitectura x86 a un CPU de 8 bits con 256 bytes de memoria y sin soporte para operaciones de entrada/salida (IN/OUT).
- **Desarrollo del producto:** Nivel bueno. Licencia MIT, última actualización en 2015, desarrollado como una plataforma web.
- **Cobertura de contenidos:** Nivel bajo. No incluye memoria independiente para módulos de entrada y salida, rutinas de interrupciones ni representación visual del ciclo de instrucción.

4.6.2 VonSim

- **Usabilidad:** Nivel medio. Utiliza solapas para presentar los componentes, lo que puede ser confuso para usuarios iniciales.
- **Editor:** Nivel medio. Proporciona notificaciones de errores de sintaxis, resaltado de código y puntos de interrupción mediante software.
- **Documentación:** Nivel medio. Incluye un manual de uso y un tutorial interactivo.
- **Ejecución de simulación:** Nivel medio. Permite ajustar la velocidad del reloj de la CPU y ofrece controles básicos de simulación.
- **Nivel de especificación:** Nivel medio. Representa una simplificación del procesador 8088 con arquitectura de 16 bits y memoria direccional de 16 KiB.
- **Desarrollo del producto:** Nivel muy bueno. Licencia GNU Affero General Public License v3.0, última versión en 2020, con amplia evidencia de uso académico.
- **Cobertura de contenidos:** Nivel medio. Implementa dispositivos internos y externos, pero carece de visualización del ciclo de instrucción y métricas de rendimiento.

4.6.3 Emu8086

- **Usabilidad:** Nivel fácil. Inicialmente muestra el editor y permite activar los componentes del simulador a medida que se cargan programas.
- **Editor:** Nivel alto. Incluye notificaciones de errores de sintaxis, resaltado de código, puntos de interrupción y opciones para guardar/cargar programas.
- **Documentación:** Nivel completo. Ofrece un manual de instrucciones con ejemplos, un tutorial de aprendizaje y una guía de uso detallada.
- **Ejecución de simulación:** Nivel alto. Proporciona control avanzado de la simulación, como retroceder una instrucción (“step back”).
- **Nivel de especificación:** Nivel completo. Detalla la arquitectura del procesador 8086, con memoria direccionable de 1 MiB y soporte para interrupciones de software y hardware.
- **Desarrollo del producto:** Nivel bueno. Licencia privativa, última actualización en 2023, desarrollado para plataformas de escritorio.
- **Cobertura de contenidos:** Nivel alto. Emula el arranque (bootstrapping) de una IBM PC desde un disco flexible (floppy disk) y soporta todos los modos de direccionamiento.

Tabla 4.3: Comparativa según criterios de evaluación preestablecidos

Criterio de Evaluación	Simple 8 bit	VonSim	Emu8086
Usabilidad	Medio	Medio	Fácil
Editor	Bajo	Medio	Alto
Documentación	Mínima	Media	Completa
Ejecución de simulación	Medio	Medio	Alta
Nivel de especificación x86	Mínima	Media	Completa
Características del producto	Buena	Muy buena	Buena
Cobertura de contenidos	Baja	Media	Alta

4.7 Resultados

La asignatura promueve el uso de simuladores para apoyar la enseñanza y el aprendizaje, permitiendo aplicar los contenidos desarrollados en máquinas reales. Emu8086 es la herramienta más adecuada para esta finalidad, ya que facilita la implementación de programas en hardware real. Sin embargo, su dependencia de

MS-DOS complica su ejecución en sistemas operativos actuales, requiriendo el uso de emuladores de MS-DOS, lo que añade complejidad al proceso de enseñanza y aprendizaje.

Desde 2018, la asignatura utiliza la versión 4.08 de Emu8086. La herramienta tiene un periodo de evaluación gratuito de 14 días, después del cual se debe adquirir una licencia. Esto es un inconveniente, ya que se busca que los estudiantes puedan acceder a las herramientas de forma libre y gratuita.

Utilizar lenguaje NASM (Netwide Assembler) garantiza soporte tanto para Linux como Windows a través de herramientas libres como GCC (GNU Compiler Collection), generando programas para la arquitectura x86 de 16, 32 y 64 bits.

Emu8086 destaca por su interfaz dinámica, que muestra componentes como la pila, flags, teclado y pantalla solo cuando son necesarios, a diferencia de otros simuladores que presentan todos sus componentes desde el inicio.

Emu8086 sobresale particularmente en los aspectos vinculados a la edición, documentación y control de la ejecución. Su editor permite establecer puntos de ruptura, retroceder una instrucción, y guardar o recuperar programas desde la interfaz. Además, ofrece una documentación extensa, que incluye un repertorio de instrucciones con ejemplos, un tutorial para el aprendizaje del lenguaje ensamblador y un manual detallado del entorno de desarrollo. Estas características lo posicionan como una herramienta completa en términos de acompañamiento a los procesos de enseñanza y aprendizaje.

En el criterio de evaluación cuatro, Emu8086 se destaca por ofrecer una mayor cantidad de controladores para gestionar el flujo de ejecución, como la capacidad de retroceder la ejecución de una instrucción y recargar el programa actual.

En cuanto al nivel de especificación, Emu8086 representa con gran precisión la arquitectura x86, incluyendo soporte para interrupciones del sistema operativo MS-DOS. Esta característica permite simular de manera realista programas que podrían ejecutarse en un entorno compatible, constituyendo una ventaja significativa frente a los otros simuladores analizados.

En el criterio de evaluación seis VonSim se destaca del resto debido a que es licencia libre y posee una comunidad que respalda el proyecto.

En cuanto al último criterio, ninguna de las herramientas evaluadas cubre todos los contenidos que se pretende desarrollar con la ayuda de una herramienta, quedando excluido pasos del ciclo de instrucción y medidas de rendimientos (tiempo de CPU y CPI: ciclo por instrucción).

A partir del análisis comparativo, se destacan las siguientes observaciones clave:

- **Emu8086** presenta la interfaz más intuitiva y completa, con amplia documentación y una simulación precisa de la arquitectura x86. Sin embargo,

su licencia privativa y la necesidad de emuladores para su ejecución en sistemas actuales constituyen limitaciones relevantes.

- **VonSim**, con su licencia libre y actualización reciente, representa una alternativa interesante desde una perspectiva de software abierto, aunque su cobertura de contenidos y nivel de especificación son limitados en comparación.
- **Simple 8-bit Assembler Simulator** resulta insuficiente para cubrir los objetivos curriculares de la asignatura, debido a su bajo nivel de complejidad, escasa documentación y capacidades limitadas de simulación.

En conclusión, si bien cada simulador ofrece ventajas puntuales, ninguno logra satisfacer plenamente los requerimientos pedagógicos y técnicos de la asignatura en su totalidad. Por ello, se recomienda continuar utilizando Emu8086 de manera transitoria, mientras se avanza en el desarrollo de un simulador propio que integre sus fortalezas, opere con licencia libre y sea compatible con entornos modernos. Esta iniciativa permitirá una mayor adecuación curricular, accesibilidad tecnológica y sostenibilidad en el tiempo.

4.7.1 Publicación

Este análisis comparativo fue publicado en el XVII Congreso de Tecnología en Educación y Educación en Tecnología (2022), bajo el título “Herramientas de software para dar soporte en la enseñanza y aprendizaje de la arquitectura x86”[\[72\]](#).

Además, durante este proceso se estableció contacto con un desarrollador de VonSim, logrando implementar mejoras significativas, como animaciones de ejecución y documentación en línea, disponibles en su última versión publicada en agosto de 2023.

Capítulo 5

Diseño y construcción del simulador

En este capítulo se describe el diseño y desarrollo de una herramienta de simulación específica para la arquitectura x86, orientada a facilitar la enseñanza de los principios de arquitectura de computadoras. Se detalla la justificación del diseño, los pasos seguidos para su construcción y los métodos utilizados para validar su funcionalidad.

En el capítulo anterior se analizaron y evaluaron las herramientas de simulación existentes para la arquitectura x86. Esta revisión exhaustiva permitió identificar las limitaciones de las soluciones actuales y fundamentar la necesidad de desarrollar una herramienta específica (véase el capítulo 4).

A partir de esta necesidad, se establecieron un conjunto de requisitos funcionales y pedagógicos que guiaron de manera integral el diseño, la implementación y la validación del simulador. Estos requisitos no solo responden a las limitaciones observadas en herramientas existentes, sino que se alinean con los objetivos educativos previamente definidos.

5.1 Requisitos de la herramienta

Esta sección expone los requisitos que orientaron el diseño del simulador, clasificados en dos dimensiones complementarias: pedagógica y funcional. La primera se vincula con los objetivos formativos definidos en el capítulo introductorio (1), mientras que la segunda refiere a las características técnicas necesarias para garantizar su implementación eficaz. La definición de los requisitos se apoyó en principios pedagógicos y técnicos, complementados con una validación empírica realizada mediante entrevistas semiestructuradas a docentes expertos (ver Apéndice: Anexo A 6.1). Estas entrevistas revelaron, entre otros aspectos, la necesidad de incorporar

visualizaciones gráficas del ciclo de instrucción, soporte para interrupciones y periféricos, y un repertorio reducido de instrucciones con activación progresiva, para evitar la sobrecarga cognitiva en los estudiantes. Asimismo, se identificaron limitaciones en herramientas existentes, como el Emu8086, cuya interfaz y funcionalidades no satisfacen completamente las necesidades pedagógicas. Este proceso permitió identificar necesidades auténticas del aula y carencias específicas en las herramientas existentes, aportando una base empírica rigurosa para la formulación pedagógica y técnica de los requisitos [74].

- 1. Visualización de la estructura general de la computadora:** Representar gráficamente la CPU, los buses, la memoria y los dispositivos de entrada/salida. Esta visualización debe destacar los componentes activos en cada etapa del ciclo de instrucción, facilitando una comprensión sistémica e integrada del funcionamiento de la computadora [75]. El uso de representaciones gráficas como recurso didáctico está respaldado por estudios que demuestran su efectividad para facilitar la comprensión de conceptos abstractos en disciplinas técnicas [76]. La Figura 5.1 ilustra la estructura general del simulador, esta representación gráfica facilita la comprensión de cómo los componentes del simulador trabajan en conjunto durante la ejecución de un programa.

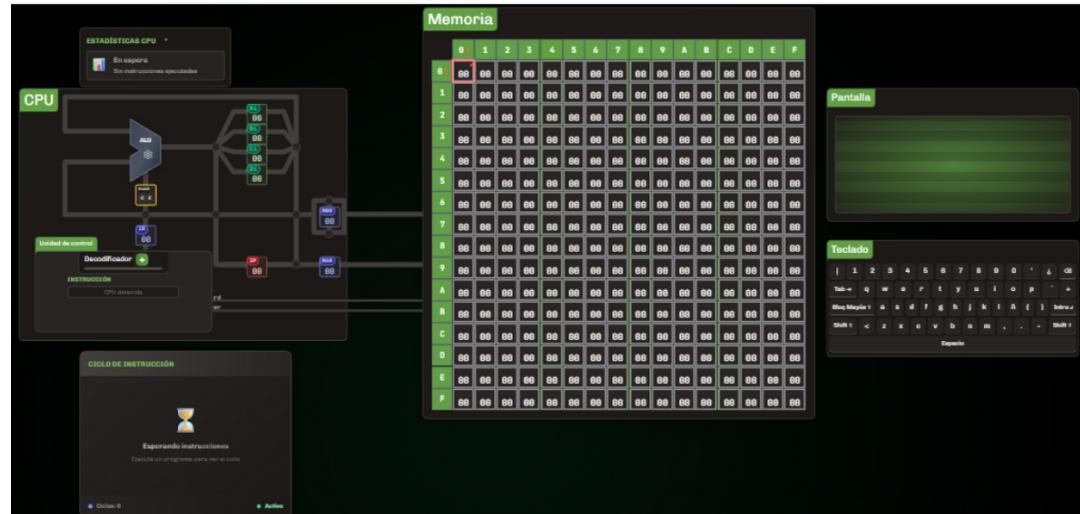


Figura 5.1: Estructura general del simulador VonSim8

- 2. Soporte para la generación y ejecución de programas en ensamblador:** Incorporar la posibilidad de ejecutar programas escritos en lenguaje ensamblador tanto de forma paso a paso como en ejecución continua. Esta funcionalidad posibilita el análisis detallado de cada instrucción, fortaleciendo competencias en trazado y depuración de código ensamblador, fundamentales para comprender la relación entre software y hardware. Para

apoyar este proceso, se propone la inclusión de un editor de ensamblador que incorpore funciones como resaltado de sintaxis y autocompletado. Estas características mejoran la experiencia del usuario y facilitan la escritura y comprensión del código, en consonancia con principios de diseño de interfaces que priorizan la usabilidad y la accesibilidad [75]. El editor debe permitir al usuario escribir, editar, guardar y ejecutar programas en ensamblador dentro del simulador, además de ofrecer ejemplos predefinidos como apoyo didáctico. La incorporación de entornos de desarrollo integrados (IDEs) en contextos educativos ha demostrado ser eficaz para la enseñanza de lenguajes de programación, según diversos estudios [77].

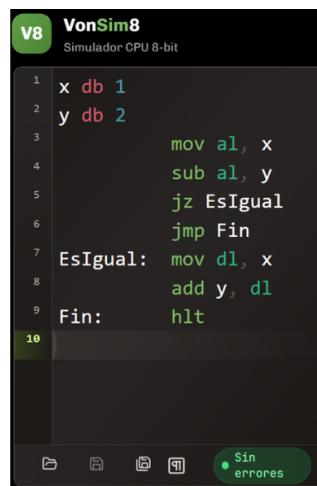


Figura 5.2: Editor ensamblador

3. **Repertorio reducido de instrucciones con activación progresiva:** Se selecciona un subconjunto esencial del conjunto de instrucciones x86, el cual se habilita de manera secuencial a lo largo del proceso de enseñanza, en estrecha correspondencia con el avance de los contenidos curriculares. Esta decisión responde a principios de la psicología cognitiva, que sostienen que la introducción gradual de contenidos técnicos favorece la retención y disminuye la sobrecarga cognitiva [78]. Diversos autores, como Hasan [18], Null y Lobur [36], y Stallings [20], respaldan este enfoque, recomendando la incorporación escalonada de conceptos en la enseñanza de arquitecturas complejas. La activación progresiva del repertorio de instrucciones promueve el desarrollo paulatino de competencias, evitando que los estudiantes se enfrenten prematuramente a la totalidad del conjunto instruccional. Este método se fundamenta en teorías de aprendizaje que destacan los beneficios de la exposición gradual a nuevos conceptos para mejorar la comprensión y la retención [21]. En la siguiente tabla 5.1 se resume la activación progresiva de instrucciones y sus objetivos didácticos asociados.

Tabla 5.1: Activación progresiva del repertorio de instrucciones

Fase	Instrucciones activadas	Objetivo didáctico
Inicial	MOV, ADD, SUB, HLT	Comprensión del ciclo de instrucción básico
Intermedia	CMP, JMP, Jxx	Introducción a control de flujo
Avanzada	CALL, RET, INT, IRET, CLI, STI, IN, OUT, POP, PUSH	Manejo de periféricos e interrupciones

4. **Simulación visual e interactiva de micropasos de instrucciones:** Se implementa una visualización interactiva del flujo de datos basada en el modelo de Nivel de Transferencia entre Registros (Register Transfer Level, RTL). Este enfoque permite representar con precisión el desplazamiento de datos entre registros, buses y unidades funcionales del procesador, así como las señales de control involucradas en cada fase del ciclo de instrucción [35], [79]. Stallings [20] propone utilizar el modelo RTL para representar el ciclo de instrucción, desde la captura (fetch) hasta la ejecución (execute), facilitando la visualización del recorrido de datos y señales de control en cada etapa del proceso. Como complemento a la descripción anterior, la Figura 5.3 muestra un ciclo de instrucción típico utilizando la operación `MOV AL, BL`. En la etapa de captación (*fetch*), la dirección de la instrucción se carga desde el registro IP al MAR, y el contenido de la memoria se transfiere al MBR y luego al IR. En la etapa de ejecución (*execute*), los datos se mueven desde el registro BL al registro AL.

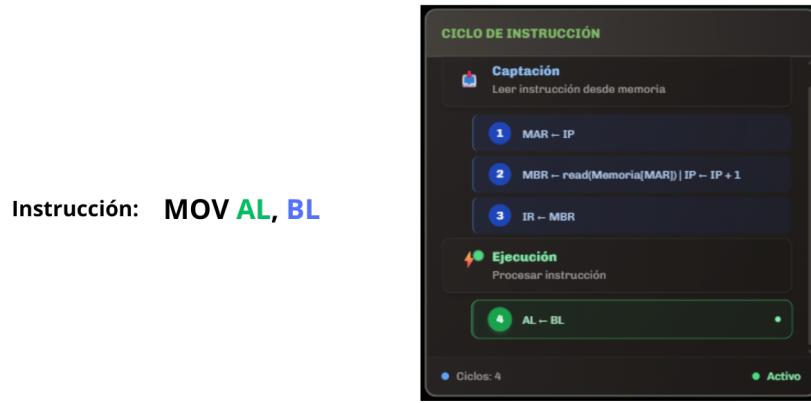


Figura 5.3: Ciclo de instrucción: captación y ejecución

5. **Gestión básica de interrupciones y periféricos:** Incluir un vector de interrupciones predefinido que simule eventos externos, como la entrada de

datos mediante teclado o la salida de información a través de un monitor. El vector de interrupciones predefinido permite simular eventos externos, como la entrada de datos mediante teclado. Por ejemplo, al recibir una interrupción de teclado, el simulador activa la señal INTR, detiene la ejecución actual y transfiere el control a la rutina de tratamiento de interrupciones correspondiente. Desde el punto de vista pedagógico, esta funcionalidad ofrece al estudiante la posibilidad de explorar de manera interactiva conceptos clave como la asincronía, el manejo de eventos y la interrupción del flujo secuencial, todos ellos característicos del diseño de arquitecturas modernas y fundamentales para el entendimiento de sistemas reales. Su inclusión se alinea con las recomendaciones de autores como Null y Lobur [36], quienes destacan el valor de abordar estos conceptos en etapas tempranas de la formación. Además, se incorpora un módulo genérico de entrada/salida programada (Programmed Input/Output, PIO), que actúa como interfaz entre la CPU y los dispositivos periféricos. Este módulo permite simular operaciones mediante instrucciones como IN y OUT, facilitando la interacción del estudiante con dispositivos representados gráficamente, como interruptores y teclas. De esta forma, se promueve una comprensión más tangible de los mecanismos subyacentes al intercambio de información entre el procesador y los dispositivos externos.

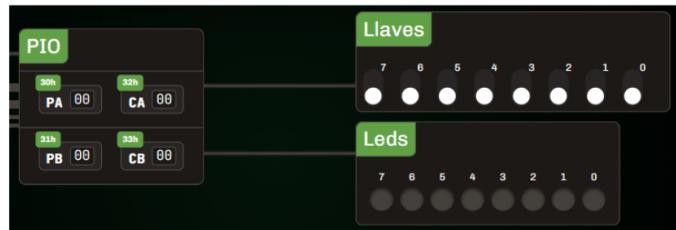


Figura 5.4: Módulo genérico de entrada/salida programada

6. Métricas de rendimiento: Incluir indicadores clave como tiempo de ciclo, tiempo de CPU y ciclos por instrucción (CPI), generados automáticamente a partir de la ejecución de los programas. Asimismo, se permite configurar la frecuencia del CPU dentro de un rango de valores (1–10 Hz). Estos indicadores permiten al estudiante analizar cuantitativamente la eficiencia en la ejecución de un programa, facilitando comparaciones entre diferentes implementaciones. Su inclusión apunta a fortalecer la comprensión de aspectos clave del rendimiento del procesador, promoviendo una formación integral que contemple tanto aspectos funcionales como métricos del comportamiento del sistema [19].



Figura 5.5: Métricas de rendimiento

7. Documentación y recursos de apoyo: Proporcionar documentación clara y accesible que explique el funcionamiento del simulador, sus componentes y las instrucciones disponibles. Esta documentación debe incluir ejemplos prácticos, guías de uso y recursos adicionales para facilitar la comprensión y el aprendizaje autónomo. La inclusión de tutoriales interactivos y ejemplos prácticos es fundamental para guiar al estudiante en el uso efectivo del simulador, promoviendo un aprendizaje activo y reflexivo [80].

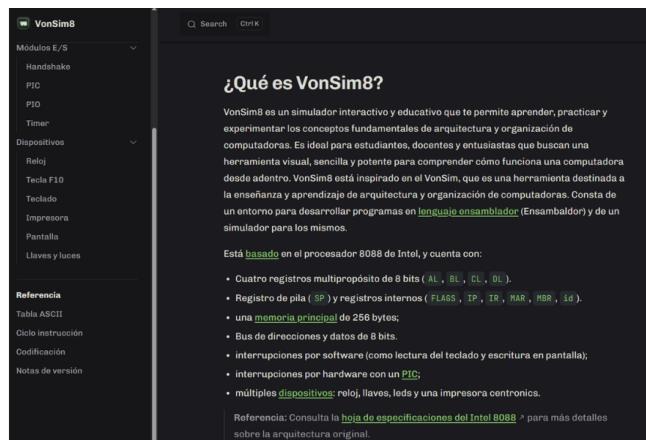


Figura 5.6: Documentación on line

En conjunto, estos requisitos constituyen la base del diseño del simulador, garantizando tanto su pertinencia pedagógica como su viabilidad técnica. Su formulación responde a la necesidad de contar con un recurso didáctico que facilite la enseñanza y el aprendizaje de la arquitectura x86, integrando aspectos visuales, interactivos y de análisis del rendimiento. La Tabla 5.2 presenta un resumen de los principales requisitos funcionales, junto con su fundamentación pedagógica y técnica.

Tabla 5.2: Resumen de requisitos funcionales y su fundamentación pedagógica

Requisito	Fundamento pedagógico / técnico
Visualización de estructura general	Facilita comprensión sistemática mediante representaciones gráficas de hardware.
Soporte para programas en ensamblador	Desarrolla competencias en trazado y depuración de lenguaje ensamblador; mejora usabilidad.
Repertorio reducido y activación progresiva	Disminuye sobrecarga cognitiva al introducir instrucciones de manera escalonada.
Simulación visual de micropasos (RTL)	Permite comprender el flujo interno de datos y señales de control durante el ciclo de instrucción.
Gestión de interrupciones y periféricos	Simula asincronía y manejo de eventos, favoreciendo la comprensión de sistemas reales.
Métricas de rendimiento	Promueve análisis cuantitativo de eficiencia (CPI, ciclos, tiempo de CPU).
Documentación y recursos de apoyo	Fomenta aprendizaje autónomo y activo mediante guías, tutoriales y ejemplos.

La definición de los requisitos funcionales y pedagógicos permitió identificar la necesidad de simplificar la arquitectura x86 para adaptarla a los objetivos educativos. Esta simplificación asegura que los estudiantes puedan concentrarse en los conceptos fundamentales sin verse abrumados por la complejidad técnica. A continuación, se detalla la justificación de esta decisión.

5.2 Justificación pedagógica de la arquitectura simplificada

A partir de los requisitos funcionales detallados anteriormente, se realizó un proceso colaborativo de análisis con docentes¹ a cargo de la asignatura Arquitectura de

¹Docentes: Marcelo A. Colombani y Amalia G. Delduca

Computadoras. Su experiencia permitió identificar los elementos de la arquitectura x86 que debían representarse, simplificarse o adaptarse según los objetivos pedagógicos del simulador. Como resultado, se optó por una arquitectura simplificada de 8 bits, cuya elección se justifica por su valor didáctico: reduce la complejidad del modelo sin comprometer la enseñanza de conceptos esenciales como el ciclo de instrucciones, la manipulación de registros o la gestión de interrupciones. Esta simplificación permite representar procesos clave con mayor claridad y menor carga cognitiva, favoreciendo la comprensión de los estudiantes en las etapas iniciales del aprendizaje.

La arquitectura x86 se distingue por su elevada complejidad, derivada de su extenso repertorio de instrucciones y sus múltiples características avanzadas. En respuesta, el diseño del simulador se fundamenta en tres principios pedagógicos centrales:

- **Reducir la carga cognitiva:** simplificar el repertorio y los componentes permite a los estudiantes enfocarse en los principios fundamentales.
- **Aprendizaje progresivo:** se adopta un enfoque escalonado, comenzando con un modelo simple y avanzando hacia representaciones más completas de x86.
- **Claridad pedagógica:** las prácticas resultan manejables en términos de tiempo y esfuerzo, favoreciendo un aprendizaje activo y centrado en la resolución progresiva de problemas.

En este marco, el diseño del simulador contribuye a:

- **Comprensión fundamental:** los estudiantes pueden concentrarse en el ciclo de instrucciones, la interacción de componentes y el flujo básico de datos.
- **Análisis crítico:** comparar el modelo simplificado con la arquitectura x86 real favorece un aprendizaje más reflexivo y profundo.
- **Experimentación práctica:** proporciona un entorno accesible para explorar conceptos y corregir errores.

Diversos autores como Patterson & Hennessy [19], Tanenbaum [31] y Null [36] coinciden en que el uso de arquitecturas simplificadas, como las de 8 bits, permite a los estudiantes centrarse en los fundamentos de la arquitectura de computadoras sin verse abrumados por la complejidad técnica de arquitecturas reales. Este enfoque hace posible observar la transferencia de datos entre registros y la activación de señales de control en cada etapa, facilitando la comprensión del funcionamiento interno del procesador.

El modelo propuesto se inspira en los principios de la arquitectura x86 [48], pero implementa un repertorio reducido de instrucciones en una arquitectura de 8 bits. Esta elección responde a criterios pedagógicos: simplifica el modelo sin sacrificar los principios esenciales del repertorio x86, y facilita una comprensión progresiva de sus componentes [81], [82], [83], [84], [85].

Aunque la simplificación a 8 bits reduce la fidelidad del modelo respecto a la arquitectura x86 real, esta decisión permite a los estudiantes concentrarse en los principios fundamentales, como el ciclo de instrucciones y la interacción entre registros. Para abordar esta limitación, el simulador incluye funcionalidades avanzadas que pueden activarse progresivamente, como la gestión de interrupciones y el repertorio completo de instrucciones.

En síntesis, la definición de estos requisitos integra aspectos funcionales, pedagógicos y técnicos en una herramienta que no solo simula el comportamiento del sistema, sino que además facilita activamente los procesos de enseñanza y aprendizaje en arquitectura de computadoras. La combinación entre visualización, ejecución progresiva y análisis de rendimiento ofrece un entorno didáctico rico que responde tanto a las necesidades del aula como a los desafíos de la disciplina.

5.3 Introducción a VonSim

VonSim² [86] es una herramienta diseñada específicamente para la enseñanza y el aprendizaje de la arquitectura y organización de computadoras, que sirvió de referencia por su enfoque educativo e interfaz intuitiva. A partir de esta herramienta se desarrolló VonSim8³ [87], adaptado para operar con registros y memoria de 8 bits, y diseñado para favorecer el aprendizaje progresivo.

VonSim ofrece una arquitectura detallada con un amplio repertorio de instrucciones y componentes. Aunque esta riqueza funcional es valiosa, puede resultar abrumadora para estudiantes en etapas iniciales. Por esta razón, VonSim8 implementa una simplificación estratégica para reducir la carga cognitiva en los primeros niveles de aprendizaje, promoviendo una asimilación progresiva de los conceptos fundamentales de la arquitectura de computadoras. A partir de esta base, se introdujeron diversas modificaciones en los componentes, instrucciones y funcionalidades del simulador, priorizando aquellos aspectos conceptuales que se abordan en el programa de la asignatura.

Las siguientes características posicionan a VonSim como una solución educativa integral:

²VonSim: <https://vonsim.github.io/>

³VonSim8: <https://ruiz-jose.github.io/VonSim8/>

1. **Entorno integrado de desarrollo y simulación:** incluye un editor de código ensamblador con resaltado de sintaxis y un simulador para la ejecución de programas, facilitando el aprendizaje práctico. [86].
2. **Fundamento en arquitectura real:** basado en el procesador Intel 8088, ofrece una referencia histórica y técnicamente relevante. [54].
3. **Componentes esenciales para el estudio:** incorpora cuatro registros multipropósito de 16 bits, memoria principal de 32 kB, bus de direcciones de 16 bits y bus de datos de 8 bits, entre otros. [20].
4. **Gestión completa de interrupciones:** Implementa tanto interrupciones por software (entrada/salida de datos) como interrupciones por hardware mediante un controlador de interrupciones programable (PIC), cubriendo aspectos fundamentales de la operación del sistema [19].
5. **Simulación de periféricos:** incorpora dispositivos como reloj, llaves, luces e impresora, inspirados en los especificados por la familia iAPX 88 de Intel, permitiendo simular interacciones complejas con el sistema.
6. **Enfoque pedagógico mediante simplificaciones estratégicas:** no pretende ser un emulador fiel del 8088, sino una herramienta educativa que implementa simplificaciones deliberadas (repertorio de instrucciones reducido y codificación simplificada) para facilitar la comprensión en contextos educativos [81].
7. **Desarrollo académico especializado:** fue creado por Facundo Quiroga, Manuel Bustos Berrondo y Juan Martín Seery, con la colaboración de Andoni Zubimendi y César Estrebou, específicamente para las cátedras de Organización de Computadoras y Arquitectura de Computadoras de la Facultad de Informática de la Universidad Nacional de La Plata, garantizando su alineación con objetivos curriculares específicos.
8. **Fundamento en experiencia previa:** se basa en el simulador MSX88, desarrollado en 1988 por Rubén de Diego Martínez para la Universidad Politécnica de Madrid, aprovechando décadas de experiencia acumulada en simuladores educativos.
9. **Accesibilidad y sostenibilidad:** distribuido bajo licencia GNU Afferro General Public License v3.0 con código fuente disponible en GitHub, y documentación bajo licencia CC BY-SA 4.0, facilitando su estudio, modificación y mejora continua [88].

5.3.1 Stack tecnológico

El proyecto VonSim está desarrollado íntegramente en TypeScript, lo que permite aprovechar el tipado estático, lograr mayor robustez del código y contar con mejor

soporte para autocompletado y detección temprana de errores durante el desarrollo.

La organización del código sigue una arquitectura de monorepositorio, compuesto por diversos paquetes especializados que cumplen funciones específicas:

- vonsim/assembler: ensamblador que traduce el código en lenguaje ensamblador a binario ejecutable.
- vonsim/simulator: motor que ejecuta los programas ensamblados.
- vonsim/app: aplicación web que proporciona la interfaz gráfica de usuario e integra el simulador.
- vonsim/common: utilidades compartidas entre los distintos módulos.
- eslint-config-vonsim: paquete para la configuración de reglas de estilo y buenas prácticas mediante ESLint.
- vonsim/scripts y vonsim/tsconfig: paquetes de soporte con scripts de desarrollo y configuraciones específicas para TypeScript.
- vonsim/docs: módulo destinado a la gestión de la documentación del proyecto.

Para el desarrollo y la ejecución del proyecto se utilizan herramientas modernas como Node.js v22 y el gestor de paquetes pnpm v10. Asimismo, el repositorio incluye una serie de scripts predefinidos para facilitar las tareas de instalación, compilación y despliegue: pnpm install, pnpm dev, pnpm docs:dev y pnpm build.

5.4 Estructura y componentes de VonSim8

En esta sección se describe la estructura del simulador VonSim8. El diseño de los registros se concibió con un propósito pedagógico: facilitar la comprensión de los modos de direccionamiento y del ciclo de instrucción, elementos fundamentales en el estudio de la Arquitectura de Computadoras [20]. En la tabla 5.3 se describen los componentes principales del simulador, junto con sus características y funcionalidades específicas. Esta tabla proporciona una visión general de la arquitectura del simulador, destacando los elementos clave que componen su estructura:

Entre los registros específicos, no accesibles directamente por el programador, se incluyen los siguientes:

- **SP (*Stack Pointer*)**: Responsable de la gestión de la pila, permitiendo el seguimiento de las direcciones de memoria asociadas a las operaciones de apilamiento y desapilamiento.
- **Flags (*Flags Register*)**: Registro que almacena las banderas de estado, utilizadas para reflejar el resultado de operaciones aritméticas y lógicas, así como para controlar el flujo del programa.

Tabla 5.3: Estructura VonSim8: componentes principales y características

Componente	Características
Arquitectura	Von Neumann: memoria compartida para datos e instrucciones.
Registros generales	4 registros de propósito general de 8 bits ('AL', 'BL', 'CL', 'DL').
Registros específicos	6 registros de propósito específico: <ul style="list-style-type: none"> • IP (Instruction Pointer) • IR (Instruction Register) • SP (Stack Pointer) • Flags (registro de estado) • MAR (Memory Address Register) • MBR (Memory Buffer Register)
Acceso a registros	Los registros de propósito general pueden ser accedidos y modificados por el programador. Los específicos son gestionados por la CPU.
Memoria	Memoria principal de 256 bytes, direccionada por un bus de direcciones de 8 bits. Cada posición almacena un byte. La memoria se organiza en celdas de 16 bytes, con dirección inicial '0x00h' y final '0xFFh'.
Puertos	Puertos de la CPU: <ul style="list-style-type: none"> • Bus de direcciones de 8 bits (MAR) • Bus de datos de 8 bits (MBR) • Señal de lectura (rd) y escritura (wr), cada una de 1 bit • Señal IO/M (1 bit) para distinguir acceso a memoria o E/S • Señal de petición de interrupción (INTR, 1 bit) • Señal de reconocimiento de interrupción (INTA, 1 bit)

- **IP (Instruction Pointer):** Contiene la dirección de memoria de la próxima instrucción a ejecutar, asegurando la secuenciación correcta del programa.
- **IR (Instruction Register):** Almacena el byte correspondiente a la instrucción que está siendo decodificada y ejecutada en el momento.

Adicionalmente, se disponen de dos registros esenciales para la transferencia de datos entre la CPU y la memoria principal:

- **MAR (Memory Address Register):** Encargado de almacenar las direcciones de memoria que se desean acceder.

- **MBR (Memory Buffer Register)**: Almacena temporalmente el byte de datos que se transfiere hacia o desde la memoria principal a través del bus de datos.

Adicionalmente, se incluyen dos registros auxiliares: **ri**, destinado al almacenamiento temporal de direcciones, e **id**, orientado al almacenamiento temporal de datos. Estos registros cumplen una función de apoyo en la ejecución de instrucciones.

5.4.1 Unidad de Control

La unidad de control es responsable de coordinar todas las operaciones de la CPU. Se encarga de:

- **Decodificación de instrucciones**: Interpreta el código de operación de cada instrucción.
- **Generación de señales de control**: Activa las señales necesarias para ejecutar microoperaciones.
- **Secuenciación**: Controla el orden de ejecución de las operaciones.

Memoria de Control

La memoria de control almacena microinstrucciones que guían la ejecución de cada instrucción en el simulador. Una representación visual de esta memoria, organizada en filas (microinstrucciones) y columnas (microoperaciones y señales de control), facilita la comprensión del papel que desempeña en la coordinación de la unidad de control [20].

Secuenciador

El secuenciador complementa la memoria de control mostrando cómo se controla la secuencia de microoperaciones y las señales de control generadas en cada fase del ciclo de instrucción.

5.4.2 Unidad Aritmético-Lógica (ALU)

La ALU (*Arithmetic Logic Unit*) realiza operaciones aritméticas y lógicas, como ADD y SUB. Durante el ciclo de instrucción, la unidad de control genera señales específicas que activan las operaciones de la ALU. Por ejemplo, al ejecutar una instrucción ADD, la unidad de control activa las señales necesarias para transferir los operandos desde los registros hacia la ALU y almacenar el resultado en el registro de destino. Todas estas operaciones modifican el registro Flags.

Registro de Banderas (Flags)

El registro de banderas **Flags** almacena las banderas de estado que reflejan el resultado de operaciones aritméticas y lógicas. Por ejemplo, después de una operación de suma, la bandera **Z** (Zero) se activa si el resultado es cero, lo que permite al procesador tomar decisiones condicionales basadas en este estado.

El registro **Flags** almacena banderas de estado resultantes de operaciones aritméticas, lógicas, de control y de gestión del flujo. Estas banderas permiten a la Unidad de Control modificar el flujo mediante saltos condicionales o habilitar/deshabilitar servicios (interrupciones). La Tabla 5.4 resume las banderas activas en esta versión.

Tabla 5.4: Registro Flags: descripción de las banderas

Bit N°	Abreviatura	Descripción
0	Z	Zero: 1 si el resultado = 0x00
1	C	Carry: acarreo (suma) o préstamo (resta)
2	O	Overflow: desbordamiento aritmético con signo
3	S	Sign: copia del bit 7 del resultado
4	I	Interrupt Enable: 1 habilita atención de interrupciones hardware

Los bits restantes del registro se reservan para extensiones futuras (no implementados en esta versión).

Política de actualización de banderas

Convenciones operativas:

- Z: se activa ($Z = 1$) cuando el resultado es 0x00; en otro caso $Z = 0$.
- C:
 - ADD: 1 si hay acarreo desde el bit 7 (suma sin signo).
 - SUB y CMP: 1 si destino < fuente (borrow); 0 en caso contrario.
- O:
 - ADD/SUB/CMP: se actualiza según regla de overflow con signo.
- S:
 - ADD/SUB/CMP: se actualiza según el signo del resultado (bit 7).
- I:

- CLI/STI: CLI pone I=0; STI pone I=1.
- INT: apila Flags y fuerza I=0.
- IRET: restaura el registro Flags (todas las banderas).

La Tabla 5.5 detalla la política de actualización de banderas según la clase de instrucción ejecutada:

Tabla 5.5: Política de actualización de banderas por clase de instrucción

Clase	Z	S	C	O	I
MOV	–	–	–	–	Preserva
ADD	Sí	Sí	Sí (acarreo)	Sí	Preserva
SUB	Sí	Sí	Sí (borrow)	Sí	Preserva
CMP	Sí	Sí	Sí (borrow)	Sí	Preserva
JMP/Jxx	–	–	–	–	Preserva
IN/OUT	–	–	–	–	Preserva
PUSH/POP	Preserva	Preserva	Preserva	Preserva	Preserva
CALL/RET	Preserva	Preserva	Preserva	Preserva	Preserva
IRET	Restaura	Restaura	Restaura	Restaura	Restaura
CLI/STI	–	–	Preserva	–	I=0/I=1
INT	Preserva	Preserva	Preserva	Preserva	I=0
HLT	–	–	–	–	Preserva

Nota:

El símbolo – indica que la instrucción no modifica la bandera.

5.4.3 Memoria principal

La memoria principal se modela como una matriz de 16×16 expresada en formato hexadecimal, lo que permite almacenar hasta 256 bytes de datos. Esta capacidad resulta suficiente para la mayoría de los programas de ejemplo utilizados en el curso, y su diseño simplificado facilita la comprensión de los conceptos fundamentales asociados a la memoria principal en una computadora.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
3	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
4	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
5	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
6	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
7	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
8	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
9	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figura 5.7: Memoria principal

5.4.4 Buses y multiplexores

Los buses de datos, direcciones y control se modelan como un conjunto de líneas que permiten la comunicación entre los distintos componentes del sistema. Estos buses resultan esenciales para el intercambio de información entre la CPU, la memoria y los dispositivos de entrada/salida. Además, su diseño modular favorece la posibilidad de expansión en futuras versiones del simulador.

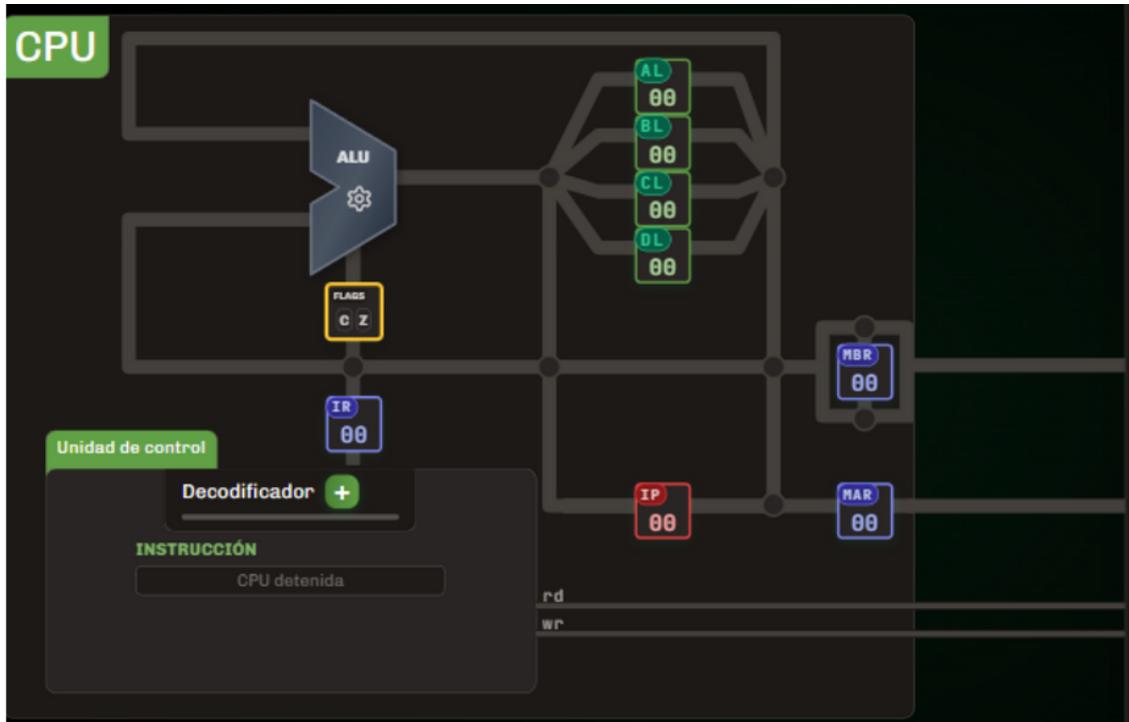


Figura 5.8: Buses

Dentro de los buses internos de la CPU se incluyen representaciones gráficas de multiplexores, componentes digitales esenciales que permiten seleccionar entre múltiples fuentes de datos o direcciones durante el ciclo de instrucción. Los multiplexores dirigen las señales hacia los registros y la ALU (Unidad Aritmético-Lógica), facilitando el flujo de datos dentro del procesador. Un multiplexor (MUX) funciona como un conmutador digital que conecta datos de una de n fuentes a la salida. Están dotados de entradas de control capaces de seleccionar una y solo una de las entradas de datos para permitir su transmisión desde la entrada seleccionada hacia dicha salida [89].

5.5 Adaptaciones y mejoras en VonSim8

Si bien VonSim ofrece una base sólida para la enseñanza de arquitectura de computadoras, su complejidad funcional puede resultar abrumadora para estudiantes en etapas iniciales. Por esta razón, se implementaron modificaciones estratégicas en VonSim8, orientadas a simplificar el modelo y alinearlo con los objetivos pedagógicos de la asignatura. A continuación, se describen las principales mejoras realizadas.

Las modificaciones implementadas se alinean con los contenidos curriculares de la asignatura y están fundamentadas en los principios del aprendizaje activo [80].

1. Simplificación del repertorio instruccional para una introducción gradual;
2. Reducción a registros y memoria de 8 bits, coherente con la escala de enseñanza;
3. Interfaz gráfica esquemática que muestra el flujo de ejecución;
4. Funciones interactivas para observar explícitamente el ciclo de instrucción y la interacción de componentes;
5. Ocultamiento inicial de las banderas O (overflow) y S (signo), y visualización dinámica de la bandera I (interrupciones);
6. Menú de controles modificado y adaptado al uso pedagógico;
7. Registros con entrada y salida independientes; ocultamiento automático de SP, ri e id;
8. Eliminación de los registros temporales left, right y result en la ALU;
9. Resaltado dinámico en memoria de las posiciones apuntadas por IP y SP;
10. Vector de interrupciones de 8 posiciones (0x00h–0x07h) y mapeo directo INT→dirección;
11. Visor de instrucciones y datos del programa, con tamaño en bytes y etiquetas;
12. Compatibilidad con directivas ORG y END (por defecto 0x00h o 0x08h; compatibilidad con org 0x20h);
13. Decodificador con memoria de control y secuenciador para microoperaciones y señales de control;
14. Tour de aprendizaje, centro de ayuda y ejemplos integrados.

A continuación se detallan los cambios más relevantes implementados en VonSim8, junto con capturas de pantalla que ilustran las modificaciones realizadas:

En el registro Flags de VonSim se ocultaron inicialmente las banderas O (overflow) y S (signo), dado que en los primeros ejercicios de ensamblador solo se emplean números enteros positivos. No obstante, estas banderas pueden habilitarse posteriormente desde el menú de configuración del simulador, conforme se abordan ejercicios de mayor complejidad.



Figura 5.9: Registro Flags

El flag de interrupción I solo se muestra cuando el programa lo requiere, por ejemplo, al ejecutar una instrucción de interrupción como INT o IRET. Esto permite a los estudiantes observar cómo se activa y desactiva este flag en función de las operaciones realizadas.



Figura 5.10: Registro de estado I

Se modificó el menú de los controles del simulador.



Figura 5.11: Controles del simulador

En lugar de utilizar registros de 16 bits completos, se emplea únicamente la parte baja de cada registro, lo que simplifica tanto la representación como la manipulación de los datos. Esta decisión responde a la necesidad de reducir la complejidad del modelo, facilitando así la comprensión de los conceptos fundamentales de la arquitectura de computadoras. Además, se ha unificado el criterio de diseño de los registros: todos cuentan ahora con una entrada y una salida independientes, lo que permite visualizar de manera más clara la transferencia de datos entre los registros y la Unidad Aritmético-Lógica (ALU). Esta modificación resulta esencial para comprender el flujo de datos durante el ciclo de instrucción y la interacción entre los distintos componentes del procesador.

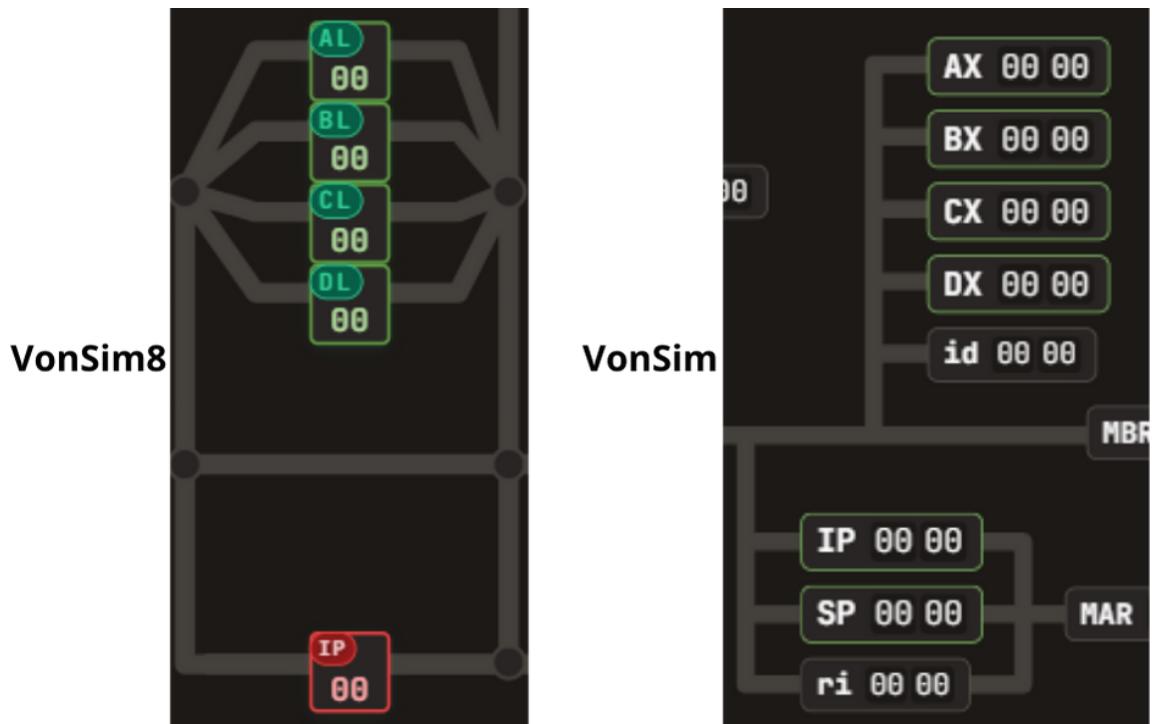


Figura 5.12: Registro de 8 bits

La eliminación de los registros temporales `left`, `right` y `result` en la ALU responde a la necesidad de simplificar el flujo de datos y reducir la carga cognitiva de los estudiantes. En lugar de utilizar registros intermedios, el simulador muestra directamente las operaciones realizadas en los operandos, lo que facilita la comprensión del proceso aritmético y lógico.

Figura 5.13: Eliminación registro temporales `left`, `right` y `result`

Los registros `SP`, `ri` e `id` se mantienen ocultos y solo se habilitan automáticamente cuando una instrucción requiere su utilización.

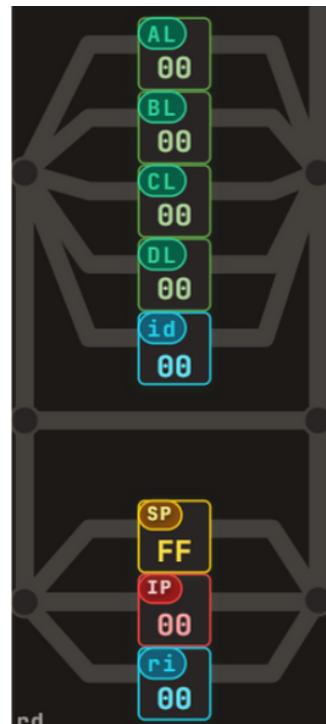


Figura 5.14: Registro SP, id y ri

La memoria principal se modela como una matriz de 16×16 expresada en hexadecimal, lo que permite almacenar hasta 256 bytes de datos.

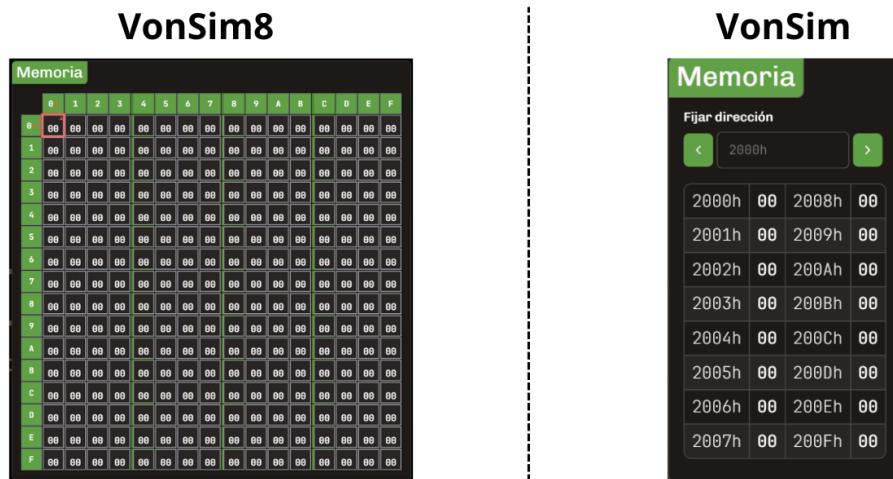


Figura 5.15: Memoria principal

Resalto de la dirección de memoria apuntada por el registro IP en memoria y también resalta la dirección de memoria apuntada por el registro SP.

Memoria																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	C9	02	01	F1	00	00	00	00	00	00	00	00	00	00	00	
1	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
2	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
3	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
4	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
5	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
6	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
7	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
8	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
9	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
B	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
C	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
D	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
E	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	01	

Figura 5.16: Resultado registro IP y registro SP

Se resalta en color naranja en la memoria principal el espacio destinado al vector de interrupciones en el VonSim8 es de 8 posiciones de memoria desde 0x00h a 0x07h:

Memoria							
	0	1	2	3	4	5	6
0	00	80	00	00	00	C0	D0

Figura 5.17: Resultado vector de interrupciones

Para determinar la dirección de la rutina de tratamiento de interrupción en VonSim, es necesario multiplicar el número de interrupción por 4, ya que cada dirección de rutina ocupa 4 bytes. En cambio, en VonSim8 no es necesario realizar esta multiplicación, dado que cada dirección de rutina de interrupción corresponde a un solo byte. Por lo tanto, la interrupción INT 0 se encuentra en la dirección 0x00h, la interrupción INT 6 en la dirección 0x06h, y así sucesivamente.

Se incorporó un visor de instrucciones y datos del programa en memoria, que permite visualizar la instrucción, el tamaño en bytes que ocupa en memoria y la etiqueta asociada a los datos.

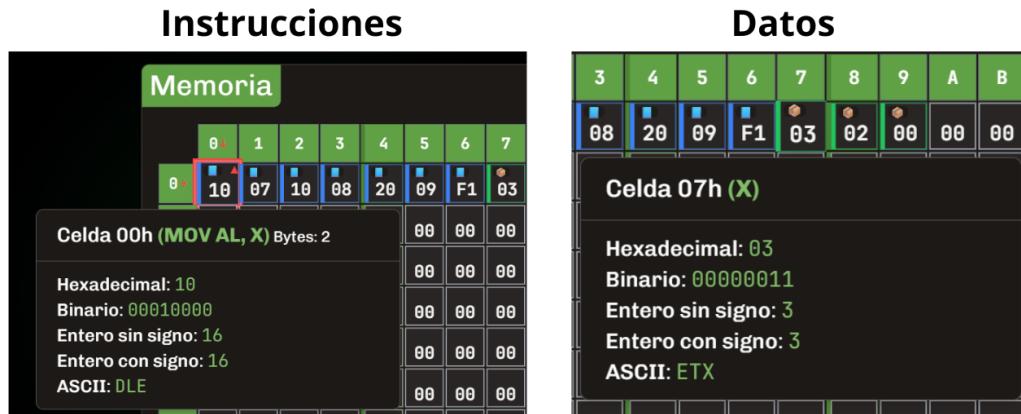


Figura 5.18: Visor de instrucciones y datos del programa en memoria

En VonSim, cuando se escribe un programa en el editor del simulador, es obligatorio que la sección de código inicie con la directiva **ORG 0x2000h** y que termine con la directiva **END**. Esto se debe a que el simulador comienza a ejecutar la primera instrucción a partir de la dirección de memoria **0x2000h**. De manera complementaria, los datos del programa suelen cargarse a partir de la dirección **0x1000h** mediante la directiva **ORG 0x1000h**.

En VonSim8, el uso de la directiva **ORG** para definir la dirección inicial es opcional. Por defecto, si el programa no incluye esta directiva, la primera instrucción se carga en la dirección **0x00h**. En caso de contener instrucciones de interrupción (INT), el simulador asigna automáticamente la dirección **0x08h** como punto de inicio, reservando espacio para el vector de interrupciones.

Por una cuestión de compatibilidad se permite cargar programas de manera similar a VonSim, pero en lugar de cargar el programa en la dirección **0x2000h** se debe cargar en la **0x20h**. En este caso, el simulador comienza a ejecutar las instrucciones a partir de la dirección **0x20h**. Para mantener compatibilidad con VonSim, se conserva esta directiva, permitiendo a los usuarios establecer direcciones personalizadas.

VonSim8

```

1 ORG 10h
2 x db 3
3 y db 2
4 z db 0
5
6 ORG 20h
7 mov al, x
8 add al, y
9 mov z, al
10 hlt
11 END

```

VonSim

```

1 ORG 1000h
2 x db 3
3 y db 2
4 z db 0
5
6 ORG 2000h
7 mov al, x
8 add al, y
9 mov z, al
10 hlt
11 END

```

Figura 5.19: Compatibilidad directiva ORG y END

Además, en VonSim8 se incorporó la opción de abrir ejemplos prácticos para que el estudiante pueda experimentar directamente con el simulador. También se desarrolló un tour de aprendizaje que guía al usuario a través de las principales funcionalidades, junto con un centro de aprendizaje que ofrece explicaciones y ejemplos básicos.

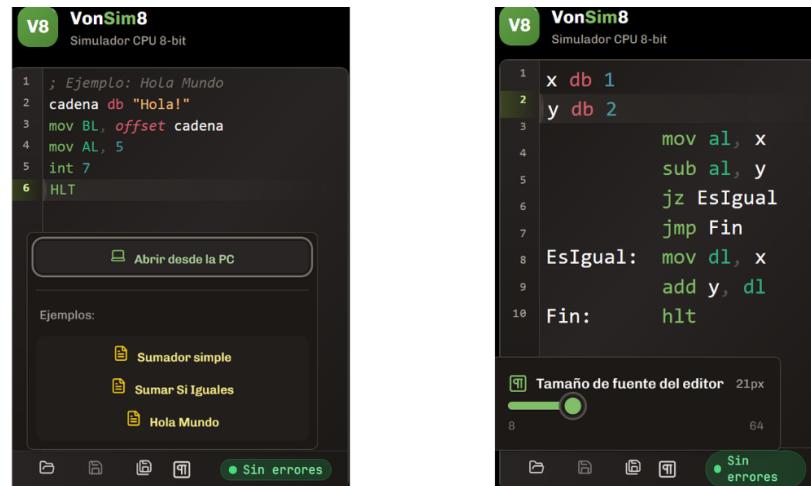


Figura 5.20: Editor con ampliación de fuente y ejemplos

Se ha mejorado el diseño de la unidad de control incorporando, dentro del decodificador, una memoria de control y un secuenciador. Estos componentes permiten representar el ciclo de instrucción de manera detallada, mostrando las microoperaciones y las señales de control generadas en cada etapa del proceso.



Figura 5.21: Decodificador en VonSim8

VonSim8 incorpora un tour guiado e interactivo que introduce didácticamente el entorno de trabajo. Este recorrido orienta al usuario en la identificación de los componentes principales —panel de registros, memoria, repertorio de instrucciones y consola de ejecución—, explicando su función y su articulación dentro del ciclo de ejecución de un programa. De este modo, el estudiante se familiariza progresivamente con la interfaz y con la representación estructural y funcional de los procesos internos del sistema.

Complementariamente, la herramienta incluye una guía de aprendizaje progresiva con ejercicios y ejemplos de código ensamblador que ponen en marcha el simulador. Cada ejemplo permite observar, instrucción por instrucción, los efectos sobre registros y memoria, favoreciendo un aprendizaje exploratorio basado en la observación y la experimentación, e integrando teoría y práctica.

Ambos recursos reducen la curva de aprendizaje y promueven la autonomía del usuario. Su diseño responde a enfoques constructivistas, en los que la interacción activa con el entorno es clave para la comprensión conceptual. Así, VonSim8 trasciende el rol de simulador técnico para operar como un entorno de aprendizaje guiado que integra explicaciones visuales, retroalimentación inmediata y ejemplos aplicados, reforzando la comprensión de la arquitectura de Von Neumann y de la ejecución de instrucciones a nivel de hardware.

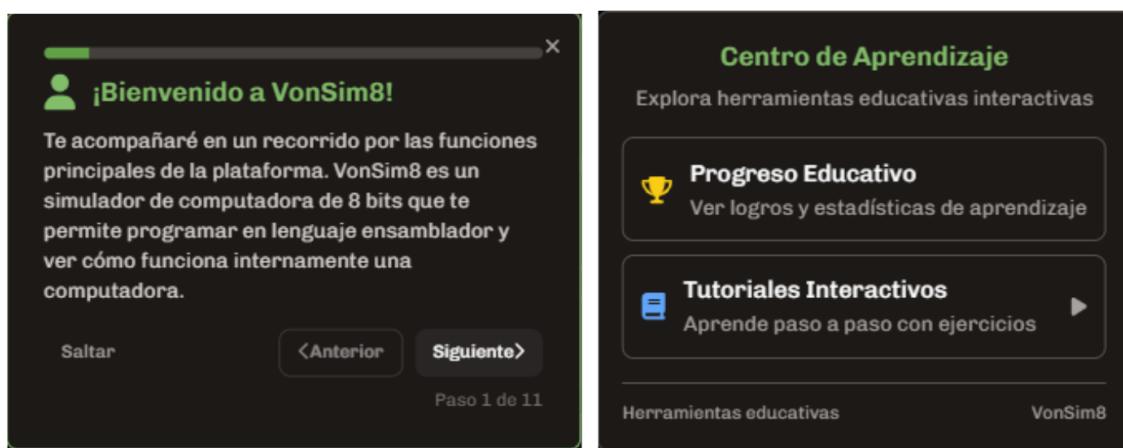


Figura 5.22: Tour de aprendizaje en VonSim8

A fin de sintetizar las diferencias más relevantes entre VonSim y VonSim8, se presenta la siguiente tabla comparativa. Esta permite visualizar de manera inmediata los cambios introducidos en el diseño y su impacto en la enseñanza de los conceptos fundamentales de arquitectura de computadoras.

Tabla 5.6: Comparativa de características entre VonSim y VonSim8

Característica	VonSim	VonSim8
Registros	Registros de 16 bits (AX, BX, CX, DX)	Registros de 8 bits, con entrada y salida independientes. Registros SP, RI e ID se ocultan y habilitan automáticamente.
Registro FLAGS	Incluye todas las banderas del 8088 (O, S, Z, C, etc.)	Se ocultan inicialmente las banderas O (overflow) y S (signo). El flag I (interrupción) se muestra dinámicamente.
Controles del simulador	Menú de controles estándar	Menú de controles modificado y adaptado.
Directiva org	Obligatoria. El código comienza en '0x2000h' y los datos en '0x1000h'.	Opcional. Por defecto, el código inicia en '0x00h'. Si hay interrupciones, comienza en '0x08h'. Compatible con 'org 0x20h'.
Memoria principal	32 KB, organizada en páginas del 8088.	Matriz de 16×16 bytes (256 bytes), expresada en hexadecimal, con resaltado dinámico de IP y SP.
Resaltado de IP y SP	No resalta dinámicamente IP y SP en memoria.	Resalta en memoria la posición apuntada por IP y SP.
Vector de interrupciones	Vector de interrupciones ocupa varias posiciones, cada rutina ocupa 4 bytes.	Vector de interrupciones ocupa 8 posiciones ('0x00h' a '0x07h'), cada rutina ocupa 1 byte.
Dirección de rutina de interrupción	Se multiplica el número de interrupción por 4 para obtener la dirección.	La dirección de la rutina de interrupción coincide con el número de interrupción (ej. INT 6 en '0x06').
Visor de instrucciones y datos	No incluye visor detallado de instrucciones y datos.	Incluye visor de instrucciones y datos, mostrando tamaño y etiquetas.
Temporales de la ALU	Registros temporales 'left', 'right' y 'result' visibles.	Eliminados, para simplificar la representación del ciclo de instrucción.
Unidad de control	Decodificador básico sin memoria de control ni secuenciador.	Decodificador mejorado con memoria de control y secuenciador, permite visualizar microoperaciones y señales de control.
Enfoque pedagógico	Mayor fidelidad al procesador Intel 8088, con repertorio más amplio.	Simplificación estratégica para reducir carga cognitiva y favorecer aprendizaje progresivo.
Editor y recursos didácticos	Editor estándar, sin ampliación de fuente ni tour de aprendizaje.	Editor con ampliación de fuente, ejemplos prácticos, tour de aprendizaje y centro de ayuda.

5.6 Repertorio de instrucciones

El simulador VonSim8 implementa un repertorio reducido de instrucciones, diseñado para facilitar la comprensión de los conceptos fundamentales de la arquitectura de computadoras. Este repertorio incluye instrucciones aritméticas, lógicas, de transferencia de datos y control de flujo, lo que facilita que los estudiantes se familiaricen con las operaciones básicas, evitando la complejidad del repertorio completo de instrucciones x86.

El repertorio de instrucciones del simulador fue concebido como una abstracción

pedagógica basada en la arquitectura x86, orientada a optimizar los procesos de enseñanza y aprendizaje en el ámbito educativo. En las primeras etapas del curso, se seleccionan únicamente las instrucciones esenciales, lo que permite introducir de forma gradual y accesible los contenidos fundamentales de la asignatura Arquitectura de Computadoras. Este enfoque progresivo facilita la comprensión de los conceptos clave, evitando la complejidad innecesaria que podría dificultar el aprendizaje inicial [19], [31]. La Tabla 5.7 presenta un conjunto reducido de instrucciones que abarca las operaciones más relevantes para una etapa introductory, centradas en la transferencia y procesamiento de datos, así como en el control de flujo. Esta selección estratégica garantiza la accesibilidad de los conceptos básicos de la arquitectura x86 y se ajusta a los objetivos pedagógicos del curso.

Las instrucciones del simulador VonSim8 se dividen en dos categorías principales: las instrucciones de transferencia y procesamiento de datos, y las instrucciones de control de flujo. Las primeras permiten mover datos entre registros y memoria, realizar operaciones aritméticas y lógicas, y manipular el contenido de los registros. Las segundas permiten alterar el flujo de ejecución del programa mediante saltos condicionales e incondicionales, así como la detención del procesador.

Con el objetivo de favorecer el aprendizaje de la programación en ensamblador y la comprensión del ciclo de instrucción, se presenta a continuación un repertorio de instrucciones alineado con el programa de estudio de la asignatura Arquitectura de Computadoras [90].

Tabla 5.7: Tabla de instrucciones de VonSim8

Instrucciones	nemónico	Acción
Transferencia de datos	MOV	Copiar
Procesamiento de datos	ADD	Sumar
	SUB	Restar
	CMP	Comparar
Control de flujo	JMP	Salto incondicional
	Jxx	Salto condicional si xx=1
	HLT	Detiene CPU

El parámetro **xx** en las instrucciones Jxx hace referencia a diferentes combinaciones de banderas de estado (flags), tales como cero (Z), acarreo (C), signo (S) y desbordamiento (O). La negación de un flag se indica con la letra N, lo que amplía la flexibilidad de control del flujo de ejecución en programas condicionales.

Con base en las entrevistas realizadas a los docentes y el análisis de los contenidos del curso, a continuación se presenta el uso pedagógico esperado para cada categoría de instrucciones.

Tabla 5.8: Saltos condicionales: instrucciones y condiciones

Instrucción	Acción
JZ Dirección	Salta a _Dirección_ si Z = 1
JNZ Dirección	Salta a _Dirección_ si Z = 0
JC Dirección	Salta a _Dirección_ si C = 1
JNC Dirección	Salta a _Dirección_ si C = 0
JS Dirección	Salta a _Dirección_ si S = 1
JNS Dirección	Salta a _Dirección_ si S = 0
JO Dirección	Salta a _Dirección_ si O = 1
JNO Dirección	Salta a _Dirección_ si O = 0

- **Transferencia y procesamiento de datos:** instrucciones que permiten mover datos entre registros y memoria, así como realizar operaciones aritméticas. Estas instrucciones son fundamentales para comprender el flujo de datos en una arquitectura computacional, mostrando cómo se ejecutan operaciones aritméticas de manera análoga a las implementadas en lenguajes de alto nivel como Python:

```
x=2
y=3
z=0
z = x + y
```

La traducción equivalente en lenguaje ensamblador es:

```
1 x db 2
2 y db 3
3 z db 0
4 mov AL, x ;Se carga el valor de x (2) en AL
5 add AL, y ;Se suma el valor de y (3) a AL (2) = 5
6 mov z, AL ;Se guarda el valor del registro AL (5) en z
7 hlt
```

- **Control de flujo:** Instrucciones que permiten alterar el flujo de ejecución del programa mediante saltos condicionales e incondicionales, así como la detención del procesador. Estas instrucciones son esenciales para comprender cómo se controlan las decisiones y el flujo de ejecución en un programa. Por ejemplo, lo que posibilita la implementación de estructuras condicionales análogas a las de lenguajes de alto nivel, como Python:

```
x=2
y=3
z=0
if x == y:
    z = y + x
```

La traducción equivalente en lenguaje ensamblador es:

```
1 x db 2
2 y db 3
3 z db 0
4     mov AL, x
5     cmp AL, y
6     jz EsIgual
7     jmp Fin
8 EsIgual: add AL, y
9     mov z, AL
10 Fin:   hlt
```

```
x=2
y=3
z=0
if x < y:
    z = y + x
```

La traducción equivalente en lenguaje ensamblador es:

```
1 x db 2
2 y db 3
3 z db 0
4     mov AL, x
5     cmp AL, y
6     jc EsMenor
7     jmp Fin
8 EsMenor: add AL, y
9     mov z, AL
10 Fin:   hlt
```

Por ejemplo, la estructura iterativa `while` en Python:

```
x = 0
suma = 0

while x < 10:
    suma = suma + x
    x = x + 1
```

La traducción equivalente en lenguaje ensamblador es:

```

1 x      db 1
2 suma   db 0
3 Condicion: cmp x, 10
4         jc Bucle      ; si x < 10 salta a etiqueta Bucle:
5         jmp FinBucle  ; si no salta a la etiqueta FinBucle:
6 Bucle:   mov BL, x
7         add suma, BL
8         add x, 1
9         jmp Condicion    ; salta a Condicion:
10 FinBucle: hlt

```

Tratamiento de vectores: El simulador permite trabajar con vectores y matrices, lo que facilita la comprensión de cómo se manejan estructuras de datos más complejas en una arquitectura computacional. Por ejemplo, el siguiente código en Python busca el máximo de un vector:

```

# Búsqueda del máximo en un vector
vector = [5, 2, 10, 4, 5, 0, 4, 8, 1, 9]
maximo = 0

for i in range(len(vector)):
    if vector[i] > maximo:
        maximo = vector[i]

```

La traducción equivalente en lenguaje ensamblador es:

```

1
2 max      db 0
3 vector  db 5, 2, 10, 4, 5, 0, 4, 8, 1, 9
4         mov CL, 0 ; contador
5         mov BL, offset vector ; dirección inicial del vector
6 Condicion: cmp CL, 10
7         jc Bucle      ; si x < 10 salta a etiqueta Bucle
8         jmp FinBucle  ; si no salta a la etiqueta FinBucle
9 Bucle:   mov AL, [BL]      ; AL = vector[indice]
10        cmp AL, max
11        jc Proximo    ; si AL < max, salta a Proximo
12        mov max, AL    ; si no, actualiza max
13 Proximo: add BL, 1       ; BL = BL + 1
14        add CL, 1       ; CL = CL + 1
15        jmp Condicion
16 FinBucle: hlt

```

La selección de instrucciones está diseñada para facilitar la comprensión de los procesos esenciales de la arquitectura de computadoras, como la ejecución de

operaciones aritméticas, la transferencia de datos entre registros y memoria, y el control del flujo de ejecución. Al emplear un repertorio reducido e inspirado en la arquitectura x86, se logra un equilibrio entre el rigor conceptual y la simplicidad pedagógica.

La implementación de estas instrucciones en el simulador VonSim8 tiene como objetivo ofrecer una experiencia de aprendizaje que favorezca la asimilación de los principios fundamentales de la arquitectura de computadoras, disminuyendo la carga cognitiva en las etapas iniciales y fortaleciendo la transición hacia repertorios más avanzados. Esta selección de instrucciones y modos de direccionamiento, inspirada en x86 pero pedagógicamente acotada, proporciona al estudiante un modelo mental preciso sobre la ejecución de operaciones aritméticas, la transferencia de datos entre registros y memoria, y la gestión del flujo de ejecución.

5.6.1 Modos de direccionamiento

Los modos de direccionamiento especifican cómo el procesador determina la ubicación de los operandos de una instrucción. Estos mecanismos permiten identificar de dónde se obtienen los datos necesarios para ejecutar una operación y dónde se almacena el resultado. En VonSim8 se implementan los siguientes modos de direccionamiento:

- **Registro a registro (Rx, Ry)**: ambos operandos corresponden a registros del procesador; Rx es el registro destino y Ry el registro fuente.
- **Directo ([M])**: uno de los operandos es el contenido de una dirección de memoria [M].
- **Indirecto ([BL])**: la dirección del operando se encuentra almacenada en el registro [BL].
- **Inmediato (d)**: uno de los operandos es un valor inmediato incluido en la propia instrucción.

La Tabla 5.9 resume los modos de direccionamiento disponibles en el simulador para instrucciones de dos operandos e incluye ejemplos ilustrativos de su uso.

Estos modos de direccionamiento permiten al simulador ejecutar una variedad de operaciones, desde la manipulación directa de registros hasta el acceso flexible a la memoria. Constituyen una base esencial para comprender el flujo de datos en una arquitectura computacional.

Ejemplo de los modos de direccionamiento:

```
1 x db 2
2 y db 3
3
4 ; Ejemplo modos direccionamiento
```

Tabla 5.9: Tabla de modos de direccionamiento

Destino	Fuente	Ejemplo
Registro Rx	Puede ser: <ul style="list-style-type: none"> • registro Ry • dirección [M] (1) • dirección registro [BL] (2) • valor inmediato d (3) 	<ul style="list-style-type: none"> • Entre registro: MOV Rx, Ry • Directo: MOV Rx, [M] • Indirecto: MOV Rx, [BL] • Inmediato: MOV Rx, d
Memoria puede ser: <ul style="list-style-type: none"> • dirección [M] • dirección registro [BL] 	Puede ser: <ul style="list-style-type: none"> • Registro: Ry • valor en la instrucción d 	<ul style="list-style-type: none"> • Directo: MOV [M], Ry • Indirecto: MOV [BL], Ry • Directo-Inmediato: MOV [M], d • Indirecto-Inmediato: MOV [BL], d

Nota:

Las instrucciones de dos operandos, tanto de transferencia y procesamiento tienen los mismos modos de direccionamiento.

Rx y Ry pueden ser un registro de propósito general: AL, BL, CL y DL.

Notas numéricas:

¹ La notación [M] indica el contenido de la dirección de memoria.

² La notación [BL] indica el contenido de la dirección del registro BL.

³ La notación d indica dato inmediato.

Símbolos:

* La notación M indica dirección de memoria.

```

5 ;-----
6 ; carga en registro
7 ;-----
8 ; Directo
9 mov al, x
10
11 ; Por registro
12 mov dl, al
13
14 ; Inmediato
15 mov bl, 16
16
17 ; Indirecto
18 mov cl, [bl] ; celda 16 = bl
19
20 ;-----
21 ; Almacenar en memoria
22 ;-----
23 ; Directo
24 mov x, cl
25
26 ; Indirecto

```

```

27 mov [bl], al
28
29 ; Directo-Inmediato
30 mov x, 5
31
32 ; Indirecto-Inmediato
33 mov [bl], 4
34
35 hlt

```

5.6.2 Formato de instrucciones

El formato de las instrucciones en VonSim8 se fundamenta en una codificación binaria de longitud variable (1, 2 o 3 bytes). Los 4 bits de mayor peso del primer byte corresponden al código de operación (opcode), mientras que los 4 bits restantes pueden contener información adicional sobre los operandos, dependiendo del modo de direccionamiento utilizado. El opcode determina la operación a ejecutar, y los operandos especifican los datos o registros implicados en la instrucción.

Las instrucciones del simulador VonSim8 se agrupan en dos categorías principales: instrucciones de transferencia y procesamiento de datos, e instrucciones de control de flujo.

Tabla 5.10: Categoría de instrucciones y códigos de operación en VonSim8

Categoría	Instrucción	Código operación	Operandos	Acción
Transferencia de datos	MOV	0, 1, 2	2	Copiar entre registros, cargar de memoria a registro, almacenar en memoria
Procesamiento de datos	ADD	3, 4, 5	2	Operación aritmética: operando1 ← operando1 + operando2
	SUB	6, 7, 8	2	Operación aritmética: operando1 ← operando1 - operando2
	CMP	9, 10, 11	2	Comparación: operando1 - operando2 (no actualiza el destino)
	JMP / Jxx	12	1	Salto incondicional JMP, condicionales Jxx
	HLT	13	0	HLT: detiene el CPU

Cada instrucción se codifica en un formato binario específico, que incluye un código de operación (opcode) y, en algunos casos, operandos adicionales. La Tabla 5.11 presenta una lista de las instrucciones implementadas en el simulador, junto con su codificación binaria.

La tabla 5.12 presenta los códigos binarios y decimales correspondientes a los registros de propósito general AL, BL, CL y DL, los cuales pueden ser utilizados como Rx y Ry en las instrucciones del simulador.

Tabla 5.11: Tabla de codificación de instrucciones

CodOp	Instrucción	Byte	Codificación
0	MOV Rx, Ry	1	0000 RxRy
1	MOV Rx, [M]	2	0001 Rx00 MMMMMMM
1	MOV Rx, [BL]	1	0001 Rx01
1	MOV Rx, D	2	0001 Rx10 DDDDDDD
2	MOV [M], Ry	2	0010 00Ry MMMMMMM
2	MOV [BL], Ry	2	0010 01Ry
2	MOV [M], D	3	0010 1100 MMMMMMM DDDDDDD
2	MOV [BL], D	2	0010 1101 DDDDDDD
3	ADD Rx, Ry	1	0011 RxRy
4	ADD Rx, -	Idem	0100 --- ----
5	ADD [M], -	Idem	0101 --- ----
6	SUB Rx, Ry	1	0110 RxRy
7	SUB Rx, -	Idem	0111 --- ----
8	SUB [M], -	Idem	1000 --- ----
9	CMP Rx, Ry	1	1001 RxRy
A	CMP Rx, -	Idem	1010 --- ----
B	CMP [M], -	Idem	1011 --- ----
C	JMP M	1	1100 0000 MMMMMMM
C	Jxx M	1	1100 ffff MMMMMMM
D	HLT	1	1101 0000

Nota:

Las instrucciones de dos operandos, tanto de transferencia y procesamiento tienen los mismos modos de direccionamiento. Considerando:

_____ : Código de operación de la instrucción, 4 bits.

Rx o Ry: Índices de registros de propósito general, 0 a 3, 2 bits cada uno.

M: Dirección de memoria 8 bits.

D: Dato inmediato 8 bits.

ffff: representa el comportamiento de la instrucción 4 bits.

5.7 Ciclo de la instrucción

El ciclo de la instrucción es la secuencia de pasos que realiza la Unidad de Control (UC) para ejecutar cada instrucción de un programa. Este proceso es fundamental para el funcionamiento de cualquier computadora, ya que involucra elementos clave como registros, buses de datos, direcciones y señales de control generadas por la UC.

Las microoperaciones que lo componen se expresan mediante la notación de transferencia entre registros: **destino** \leftarrow **origen**.

Tabla 5.12: Tabla de registros del simulador

Registros R	Binario	Decimal
AL	00	0
BL	01	1
CL	10	2
DL	11	3

Nota:

Los registros AL, BL, CL y DL corresponden a registros de propósito general de 8 bits.

La Figura 5.23 ilustra el flujo general del ciclo, el cual se divide en dos etapas principales: captación (fetch) y ejecución.

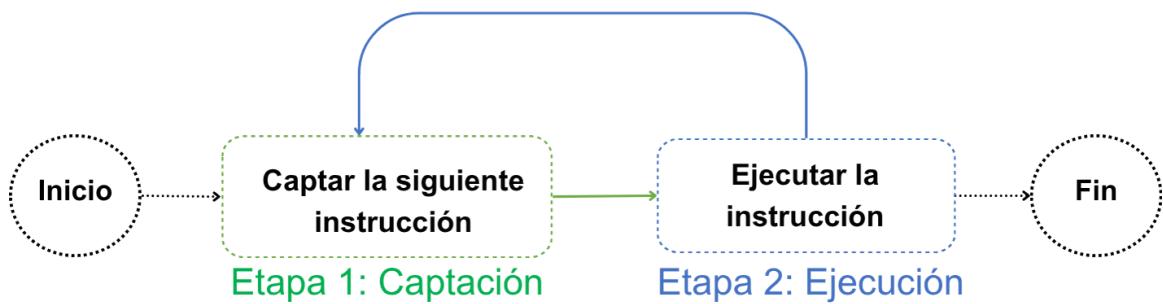


Figura 5.23: Flujo del ciclo de instrucción en VonSim8

5.7.1 Etapa 1: Captación

Esta etapa es igual para todas las instrucciones. Su objetivo es leer la instrucción desde la memoria y cargarla en el **Registro de Instrucciones** (IR). Consta de tres microoperaciones:

1. **MAR \leftarrow IP** La UC transfiere al **Registro de Direcciones de Memoria** (MAR) la dirección de la próxima instrucción, almacenada en el **Puntero de Instrucción** (IP).
2. **MBR \leftarrow read(Memoria[MAR]) |⁴ IP \leftarrow IP + 1** La UC activa la señal de lectura para leer la instrucción ubicada en la dirección contenida en el MAR. El valor

⁴“|”: microoperaciones concurrentes en el mismo paso (no agregan ciclos).

leído se guarda en el **Registro de Datos de Memoria** (MBR) y, al mismo tiempo, el IP se incrementa para apuntar a la siguiente instrucción u operando.

3. **IR** \leftarrow **MBR** El contenido del MBR se transfiere al IR, dejando la instrucción lista para ser decodificada y ejecutada.

5.7.2 Etapa 2: Ejecución

En esta etapa, el **decodificador de instrucciones** interpreta el valor en el registro IR. A partir del código de operación, lee las microinstrucciones necesarias en la **memoria de control** para determinar el tipo de instrucción, la cantidad de operandos y el modo de direccionamiento.

Luego, estas microinstrucciones se envían a **secuenciador**, que genera las señales de control precisas para ejecutar la operación.

A continuación se detallan las instrucciones más comunes:

Instrucciones con dos operandos MOV, ADD, SUB y CMP:

- Destino en registro (Rx)
 - Modo entre registros (Rx, Ry)
 4. La ejecución se realiza en un solo paso:
 - * MOV: $Rx \leftarrow Ry$
 - * ADD: $Rx \leftarrow Rx + Ry$ | update(Flags)
 - * SUB: $Rx \leftarrow Rx - Ry$ | update(Flags)
 - * CMP: $Rx - Ry$ | update(Flags) (solo actualiza flags, no guarda el resultado)
 - Modo directo (Rx, [Dirección])
 4. **MAR** \leftarrow **IP** – Obtener dirección del operando fuente.
 5. **MBR** \leftarrow **read(Memoria[MAR])** | **IP** \leftarrow **IP** + 1 – Leer la dirección desde memoria e incrementar IP.
 6. **MAR** \leftarrow **MBR** – Transferir la dirección al MAR.
 7. **MBR** \leftarrow **read(Memoria[MAR])** – Obtener el dato.
 8. Ejecutar la operación:
 - * MOV: $Rx \leftarrow MBR$
 - * ADD: $Rx \leftarrow Rx + MBR$ | update(Flags)
 - * SUB: $Rx \leftarrow Rx - MBR$ | update(Flags)
 - * CMP: $Rx - MBR$ | update(Flags)
 - Modo inmediato (Rx, Dato)
 4. **MAR** \leftarrow **IP** – Obtener dirección del dato inmediato.

5. $MBR \leftarrow \text{read}(\text{Memoria}[MAR]) \mid IP \leftarrow IP + 1$ – Leer el dato e incrementar IP.
6. Ejecutar la operación (igual que en el caso anterior).
- **Modo indirecto (R_x , [BL])**
 4. $MAR \leftarrow BL$ – Obtener dirección del dato desde el registro BL.
 5. $MBR \leftarrow \text{read}(\text{Memoria}[MAR])$ – Leer el dato.
 6. Ejecutar la operación (igual que en el caso anterior).
- **Destino en memoria ([Dirección] o [BL])** En este caso, el resultado de la operación se almacena en una dirección de memoria especificada en la instrucción o indicada por el contenido del registro BL.
 - **Modo Directo ([Dirección], Ry)**
 4. $MAR \leftarrow IP$ – Obtener dirección destino.
 5. $MBR \leftarrow \text{read}(\text{Memoria}[MAR]) \mid IP \leftarrow IP + 1$ – Leer la dirección e incrementar IP.
 6. $MAR \leftarrow MBR$ – Transferir dirección a MAR. Según la instrucción:
 - * MOV:
 7. $MBR \leftarrow Ry$ – Copiar Ry al MBR.
 8. $\text{write}(\text{Memoria}[MAR]) \leftarrow MBR$ – Escribir en memoria.
 - * ADD, SUB, CMP:
 7. $MBR \leftarrow \text{read}(\text{Memoria}[MAR])$ – Leer el dato.
 8. Ejecutar la operación:
 - ADD: $MBR \leftarrow MBR + Ry \mid \text{update}(\text{Flags})$
 - SUB: $MBR \leftarrow MBR - Ry \mid \text{update}(\text{Flags})$
 - CMP: $MBR = Ry \mid \text{update}(\text{Flags})$
 9. Si es ADD o SUB: $\text{write}(\text{Memoria}[MAR]) \leftarrow MBR$ – Escribir en memoria.
 - **Modo Indirecto ([BL], Ry)**
 4. $MAR \leftarrow BL$ – Transferir dirección de destino (en BL) a MAR. Según la instrucción:
 - * MOV:
 5. $MBR \leftarrow Ry$ – Copiar Ry al MBR.
 6. $\text{write}(\text{Memoria}[MAR]) \leftarrow MBR$ – Escribir en memoria.
 - * ADD, SUB, CMP:
 5. $MBR \leftarrow \text{read}(\text{Memoria}[MAR])$ – Leer el dato.
 6. Ejecutar la operación (igual que en el caso anterior).
 7. Si es ADD o SUB: $\text{write}(\text{Memoria}[MAR]) \leftarrow MBR$ – Escribir en memoria.
 - **Modo Directo-Inmediato ([Dirección], Dato)**
 4. $MAR \leftarrow IP$ – Obtener dirección destino.
 5. $MBR \leftarrow \text{read}(\text{Memoria}[MAR]) \mid IP \leftarrow IP + 1$ – Leer dirección e incrementar IP.

6. $\text{MAR} \leftarrow \text{IP} \mid \text{ri} \leftarrow \text{MBR}$ – Preparar para leer el dato y guardar la dirección destino en un registro intermedio (ri).
7. $\text{MBR} \leftarrow \text{read}(\text{Memoria}[\text{MAR}]) \mid \text{IP} \leftarrow \text{IP} + 1$ – Leer dato e incrementar IP. Según la instrucción:
 - * MOV:
 8. $\text{MAR} \leftarrow \text{ri}$ – Copiar dirección destino.
 9. $\text{write}(\text{Memoria}[\text{MAR}]) \leftarrow \text{MBR}$ – Escribir en memoria.
 - * ADD, SUB, CMP:
 8. $\text{MAR} \leftarrow \text{ri} \mid \text{id} \leftarrow \text{MBR}$ – Cargar dirección destino y guardar el valor inmediato en id.
 9. $\text{MBR} \leftarrow \text{read}(\text{Memoria}[\text{MAR}])$ – Leer el valor actual de destino.
 - 10. Ejecutar la operación:
 - ADD: $\text{MBR} \leftarrow \text{MBR} + \text{id} \mid \text{update}(\text{Flags})$
 - SUB: $\text{MBR} \leftarrow \text{MBR} - \text{id} \mid \text{update}(\text{Flags})$
 - CMP: $\text{MBR} - \text{id} \mid \text{update}(\text{Flags})$
 - 11. Si es ADD o SUB: $\text{write}(\text{Memoria}[\text{MAR}]) \leftarrow \text{MBR}$ – Escribir en memoria.
- Modo Indirecto-Inmediato ([BL], Dato)
 4. $\text{MAR} \leftarrow \text{IP}$ – Obtener dirección del dato inmediato.
 5. $\text{MBR} \leftarrow \text{read}(\text{Memoria}[\text{MAR}]) \mid \text{IP} \leftarrow \text{IP} + 1$ – Leer dato e incrementar IP. Según la instrucción:
 - * MOV:
 6. $\text{MAR} \leftarrow \text{BL}$ – Copiar dirección de destino.
 7. $\text{write}(\text{Memoria}[\text{MAR}]) \leftarrow \text{MBR}$
 - * ADD, SUB, CMP:
 6. $\text{MAR} \leftarrow \text{BL} \mid \text{id} \leftarrow \text{MBR}$ – Cargar la dirección destino y guardar el valor inmediato en id.
 7. $\text{MBR} \leftarrow \text{read}(\text{Memoria}[\text{MAR}])$ – Leer el valor actual de destino.
 - 8. Ejecutar la operación (igual que en el caso anterior).
 - 9. Si es ADD o SUB: $\text{write}(\text{Memoria}[\text{MAR}]) \leftarrow \text{MBR}$ – Escribir en memoria.

Instrucciones con un operando JMP y Jxx:

- Salto a (Dirección) Tanto incondicional JMP como condicionales Jxx tienen estos pasos:
 4. $\text{MAR} \leftarrow \text{IP}$ – Obtener la dirección del salto.
 5. $\text{MBR} \leftarrow \text{read}(\text{Memoria}[\text{MAR}]); \text{IP} \leftarrow \text{IP} + 1$ – Leer la dirección de destino e incrementar IP.

Según la instrucción:

- JMP:
 6. $\text{IP} \leftarrow \text{MBR}$
- Jxx:
 6. $\text{IP} \leftarrow \text{MBR}$ si se cumple la condición del flag xx; en caso contrario, continúa con la siguiente instrucción.

Instrucciones sin operandos

- HLT:
 4. Detiene la ejecución de la CPU.

Al contabilizar los pasos del ciclo RTL, cada paso equivale a un ciclo de CPU. La sección siguiente (5.7.3) describe cómo formaliza este conteo para definir CPI y tiempo de CPU.

5.7.3 Estadísticas de CPU en VonSim8

En VonSim8 cada “paso” del ciclo RTL equivale a 1 ciclo de CPU. De este modo, el CPI de cada instrucción se obtiene sumando 3 ciclos fijos de captación (pasos 1–3) más los pasos de ejecución propios del modo de direccionamiento. La notación “|” indica microoperaciones concurrentes dentro del mismo paso (no suma ciclos). Con frecuencia configurada f (Hz), el tiempo de CPU viene dado por $T_{\text{CPU}} = \text{ciclos_totales} / f$.

En consecuencia:

- Cada paso equivale a 1 ciclo.
- La etapa de captación aporta $n_{fetch} = 3$ ciclos (pasos 1–3).
- La etapa de ejecución aporta $n_{execute}$ ciclos según el modo de direccionamiento y la instrucción (pasos 4, 5, ...).

Definiciones y fórmulas:

- CPI por instrucción:

$$CPI_{instr} = n_{fetch} + n_{execute}$$

- Tiempo por instrucción (con frecuencia configurada f_{CPU} en Hz):

$$t_{instr} = \frac{CPI_{instr}}{f_{CPU}}$$

- Para un programa de N instrucciones ejecutadas:

$$CPI_{prog} = \frac{\sum_{k=1}^N CPI_k}{N} ; \quad t_{CPU} = \frac{\sum_{k=1}^N CPI_k}{f_{CPU}}$$

Ejemplos (contabilizando pasos tal como se listan en esta sección):

- MOV rx, ry (entre registros): $n_{fetch} = 3$, $n_{execute} = 1 \implies CPI = 4$ y $t_{instr} = 4/f_{CPU}$.
- MOV Rx, [Dirección] (registro \leftarrow memoria directa): pasos 4–8 $\implies n_{execute} = 5 \implies CPI = 3 + 5 = 8$, $t_{instr} = 8/f_{CPU}$.
- JMP Dirección: pasos 4–6 $\implies n_{execute} = 3 \implies CPI = 3 + 3 = 6$, $t_{instr} = 6/f_{CPU}$. En saltos condicionales Jxx solo se cuentan los pasos realmente ejecutados (si no se toma el salto, no se ejecuta la carga de IP).

Notas:

- La frecuencia f_{CPU} es configurable (1–10 Hz), por lo que el tiempo por ciclo es $T = 1/f_{CPU}$.
- Estas métricas son las que muestra el panel “Estadísticas CPU” del simulador (ciclos totales, CPI y tiempo de CPU).

5.8 Modelado con xDEVS del ciclo de instrucción MOV AL, BL

Con el propósito de validar la arquitectura del simulador VonSim8 y analizar su comportamiento interno, se modela la instrucción MOV AL, BL como un caso de estudio representativo del flujo básico de ejecución a nivel de hardware.

VonSim8 adopta el modelo de Von Neumann, caracterizada por la utilización de un bus único compartido para instrucciones y datos, lo cual introduce el clásico cuello de botella estructural (limitación en el acceso concurrente a datos e instrucciones). En este contexto, la Unidad de Control (UC) debe sincronizar con precisión los accesos concurrentes a memoria y al bus.

Este modelo aplica el formalismo DEVS (Discrete EVents System Specification), introducido en el capítulo 3.3, para representar la ejecución a nivel de Transferencia entre Registros (RTL) como una secuencia de eventos discretos temporizados. La

simulación se construye sobre la plataforma xDEVS, utilizando la biblioteca xdevs.py (Python), cuya elección se justificó en la sección 3.3.2.

De acuerdo con el formalismo DEVS, cada componente funcional del procesador —como los registros, la Unidad de Control (UC), la memoria y el bus— se modela como un modelo atómico, mientras que las interacciones entre ellos se describen mediante un modelo acoplado jerárquico. Esta estructura jerárquica permite capturar la concurrencia de eventos y la propagación temporal de señales entre componentes, manteniendo la trazabilidad de cada transición de estado.

5.8.1 Componentes del simulador VonSim8

Los componentes que intervienen en el ciclo completo de instrucción (Fases de Captación y Ejecución) se modelan como modelos atómicos o acoplados dentro de la plataforma xDEVS, cada uno con sus puertos de entrada, salida y retardos (tiempos de avance τ) definidos.

La tabla 5.13 resume los modelos y componentes principales del simulador VonSim8, detallando sus funciones, señales de entradas/salidas y retardos asociados.

Tabla 5.13: Modelos y componentes principales del simulador VonSim8

Modelo	Descripción	Entradas / Salidas	Retardo
IP (Instruction Pointer)	Contiene la dirección de la próxima instrucción a ejecutar. Envía su valor al MAR.	salida: addr_out; entrada: ip_write	$\tau_{ip} = t_{ip}$
MAR (Memory Address Register)	Almacena la dirección recibida desde el IP y la utiliza para acceder a la memoria unificada.	entrada: addr_in; salida: addr_out (a MEM)	$\tau_{mar} = t_{mar}$
MEM (Memoria unificada)	Memoria compartida que contiene instrucciones y datos. Devuelve o almacena la palabra solicitada.	entrada: addr, rw; salida: data_out	$\tau_{mem} = t_{mem}$
MBR (Memory Buffer Register)	Registro intermedio que conecta la memoria con los registros internos.	entrada: data_in; salida: data_out	$\tau_{mbr} = t_{mbr}$
IR (Instruction Register)	Registro de instrucción que recibe la palabra desde el MBR y la expone a la UC para su decodificación.	entrada: inst_in; salida: opcode, operandos	$\tau_{ir} = t_{ir}$
REG_BANK (Banco de Registros)	Modelo acoplado que agrupa los registros AL, BL, CL y DL. Cada uno se modela como componente atómico capaz de leer o escribir datos a través del bus compartido. El banco administra las señales de control (enable_in, enable_out) y coordina las operaciones entre los registros bajo las órdenes de la UC.	entradas: enable_in[n], enable_out[n]; salida: data_bus; (n = AL, BL, CL, DL)	$\tau_{regbank} = t_{regbank}$
UC (Unidad de Control)	Coordina el ciclo de captación y ejecución, generando las señales de control y arbitraje del bus.	entradas: IR; salidas: enable_out, enable_in, mem_read, mem_write	$\tau_{control} = t_{control}$
BUS / Árbitro	Canal compartido de comunicación entre IP, MAR, MEM y el banco de registros. Garantiza acceso exclusivo mediante señales de control.	señales req / grant para acceso exclusivo al bus	$\tau_{bus} = t_{bus}$

5.8.2 Fases del ciclo de instrucción

El ciclo de instrucción se compone de dos fases principales, controladas por la Unidad de Control (UC) y modeladas mediante transiciones de estado y funciones temporizadas en el formalismo DEVS:

1. Fase de captación (Fetch): lectura de la instrucción desde la memoria unificada.
2. Fase de ejecución (Execute): transferencia del contenido del registro fuente BL al registro destino AL.

5.8.3 Fase de captación (Fetch)

La fase de captación tiene como objetivo obtener la instrucción almacenada en la memoria unificada, utilizando el valor del Instruction Pointer (IP) como dirección base, y transferirla al Instruction Register (IR) para su posterior decodificación. Este proceso marca el inicio del ciclo de instrucción y determina la secuenciación temporal de accesos a memoria, condicionada por el modelo Von Neumann (memoria y bus compartidos).

En VonSim8, esta fase reproduce la restricción estructural propia de la arquitectura Von Neumann (bus y memoria compartidos), modelando el acceso exclusivo durante la lectura de la instrucción.

La tabla 5.14 presenta la secuencia de microoperaciones y eventos DEVS que modelan esta fase.

Tabla 5.14: Secuencia de microoperaciones y eventos DEVS fase captación

Paso	Microoperación	Descripción	Funciones DEVS
1	UC → IP	La UC genera una señal de lectura hacia el Instruction Pointer.	$\lambda(UC) \rightarrow \delta_{ext}(IP)$
2	IP → MAR	El IP emite la dirección actual de instrucción hacia el MAR.	$\lambda(IP) \rightarrow \delta_{ext}(MAR)$
3	UC → MEM; IP ← IP+1	La UC activa la señal mem_read para iniciar la lectura de la instrucción y, de forma concurrente, incrementa el valor del IP en una unidad.	$\lambda(UC) \rightarrow \delta_{ext}(MEM); \lambda_{UC} \rightarrow \delta_{ext}(IP)$
4	MEM → MBR	La memoria unificada devuelve la instrucción solicitada después de su retardo interno (τ_{mem}).	$\delta_{int}(MEM) \rightarrow \lambda(MEM) \rightarrow \delta_{ext}(MBR)$
5	MBR → IR	El MBR transfiere la instrucción recibida al registro IR.	$\lambda(MBR) \rightarrow \delta_{ext}(IR)$
6	IR → UC	El IR genera un evento de salida notificando a la UC que la instrucción está lista para decodificación.	$\lambda(IR) \rightarrow \delta_{ext}(UC)$

Durante esta fase, el bus se arbitra mediante un protocolo de petición-concesión (req/grant), garantizando exclusión mutua en el acceso y previniendo colisiones.

5.8.4 Fase de Ejecución (Execute): MOV AL, BL

En la fase de ejecución, la UC decodifica el contenido del IR y activa las señales necesarias para que el Banco de Registros (REG_BANK) transfiera el dato desde el registro fuente BL hacia el registro destino AL a través del bus compartido.

El REG_BANK se modela como un componente acoplado que contiene cuatro modelos atómicos: AL, BL, CL y DL. Cada registro posee sus propias entradas (enable_in, enable_out, data_in, data_out), mientras que el banco actúa como mediador, gestionando la habilitación del registro fuente y destino según las señales emitidas por la UC.

La tabla 5.15 muestra la secuencia de microoperaciones que implementan esta instrucción.

Tabla 5.15: Secuencia de microoperaciones y eventos DEVS fase ejecución

Paso	Microoperación	Descripción	Funciones DEVS
1	UC: decodifica(IR)	La UC interpreta el opcode MOV y determina los operandos AL y BL.	$\lambda(UC) \rightarrow \delta_{ext}(REG_BANK)$
2	UC → REG_BANK.enable_out(BL)	La UC habilita la salida del registro BL en el Banco de Registros.	$\lambda(REG_BANK) \rightarrow \delta_{ext}(BL)$
3	BUS ← BL	El valor de BL se transfiere al bus compartido.	$\lambda(BL) \rightarrow \delta_{ext}(BUS)$
4	UC → REG_BANK.enable_in(AL)	La UC habilita la entrada del registro AL en el Banco de Registros.	$\lambda(UC) \rightarrow \delta_{ext}(REG_BANK)$
5	AL ← BUS	El registro AL captura el dato presente en el bus y actualiza su valor interno.	$\lambda(BUS) \rightarrow \delta_{ext}(AL)$

Las microoperaciones se ejecutan conforme a las funciones DEVS de transición interna δ_{int} , transición externa δ_{ext} y de salida (λ).

5.8.5 Validación experimental

La implementación en xdevs.py reproduce 11 microoperaciones (6 en Fetch y 5 en Execute) y, para este caso de estudio, produce 14 ciclos totales y 45 transiciones DEVS. Se mantiene esta métrica de forma consistente en el cuerpo del texto y el Anexo D (6.4). Los resultados confirmán:

- **Correctitud funcional:** la transferencia $AL \leftarrow BL$ se ejecuta correctamente ($AL: 0x01h \rightarrow 0x0Ah$).
- **Coherencia temporal:** cada componente respeta sus retardos τ especificados.
- **Sincronización del bus:** el acceso exclusivo se garantiza mediante señales de control.

La traza completa de ejecución, que incluye todos los eventos λ , δ_{int} y δ_{ext} con sus marcas temporales, se presenta en el Anexo D (6.4), donde se evidencia la coordinación correcta entre la UC, el bus y los registros involucrados.

5.8.6 Conclusión

El modelado del ciclo de instrucción MOV AL, BL mediante el formalismo DEVS, implementado sobre la plataforma xDEVS con la biblioteca xdevs.py, demuestra la viabilidad de representar con rigor y precisión el comportamiento temporal de la arquitectura VonSim8 a nivel de transferencia entre registros (RTL). La descomposición del procesador en modelos atómicos (IP, MAR, MEM, MBR, IR, REG_BANK) y su orquestación mediante un modelo acoplado jerárquico permiten capturar tanto la concurrencia de eventos como las dependencias estructurales inherentes a la arquitectura de Von Neumann.

Los resultados experimentales validan la correctitud del modelo: la ejecución de la instrucción requiere 14 ciclos distribuidos en 6 microoperaciones de captación (FETCH) y 5 de ejecución (EXECUTE), generando un total de 45 transiciones DEVS ($\lambda, \delta_{int}, \delta_{ext}$) con un tiempo real de simulación de 3.36 ms. La traza temporal evidencia cómo la Unidad de Control sincroniza el acceso exclusivo al bus compartido y a la memoria unificada, reproduciendo fielmente el cuello de botella estructural de Von Neumann.

Desde una perspectiva de validación arquitectural, este enfoque posibilita la instrumentación de métricas cuantitativas de rendimiento —CPI (Cycles Per Instruction), latencia de acceso a memoria, tiempo de ocupación del bus— que constituyen indicadores objetivos para evaluar el comportamiento del sistema y detectar inconsistencias en el diseño. La modularidad del modelo DEVS facilita, además, la extensión futura del simulador mediante la incorporación de nuevos componentes (ALU, pipeline, cache) sin comprometer la estructura existente.

Desde una perspectiva pedagógica, el modelo ofrece una representación explícita y observable de las dependencias funcionales entre componentes internos del procesador, permitiendo a los estudiantes comprender cómo las restricciones arquitecturales —bus único, acceso secuencial a memoria, coordinación mediante señales de control— impactan directamente en el rendimiento del sistema. De este modo, la visualización de las transiciones de estado y la propagación temporal de eventos no solo valida el modelo, sino que refuerza el aprendizaje de conceptos fundamentales como el ciclo de instrucción, la sincronización de componentes y las limitaciones estructurales de las arquitecturas clásicas.

5.9 Módulo de entrada/salida e interrupciones

El simulador permite configurar la conexión de diversos módulos de entrada/salida y otros dispositivos al bus principal, agrupados en las siguientes categorías:

- Teclado y pantalla.

- Un módulo PIO, que puede conectarse a LEDs e interruptores.
 - Módulo Handshake, con posibilidad de conexión a una impresora, con o sin controlador PIC.
 - Un controlador PIC, que interactúa con la tecla F10 para generar interrupciones, junto con un temporizador y su reloj asociado.

La Figura 5.24 muestra una visión general de los dispositivos que pueden conectarse al simulador.

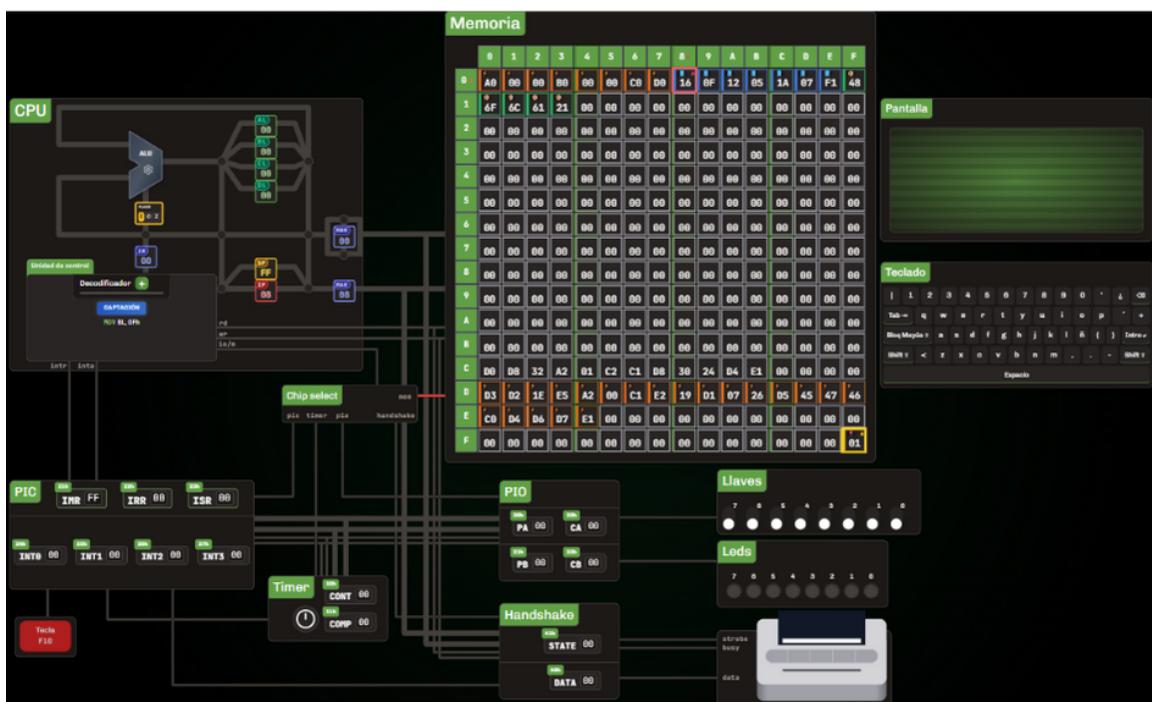


Figura 5.24: Arquitectura general del simulador

El componente chipSelect se encarga de activar el dispositivo correspondiente en cada momento. Para ello, recibe las señales de control del CPU junto con el bus de direcciones, y genera las señales de selección de chip (CS) necesarias para habilitar el dispositivo adecuado.

Para interactuar con los módulos de entrada y salida, es necesario incorporar nuevas instrucciones al repertorio del procesador. Estas instrucciones permiten la comunicación con los puertos de E/S y la gestión de interrupciones. La Tabla 5.16 detalla las categorías de instrucciones, sus códigos de operación y las acciones que realizan.

El simulador utiliza un código de operación de 4 bits para las instrucciones, lo que simplifica la arquitectura del sistema pero limita el número máximo de instrucciones

Tabla 5.16: Categoría de instrucciones y códigos de operación en VonSim8

Categoría	Instrucción	Código operación	Acción
Transferencia de datos	MOV	0, 1, 2	Copiar entre registros, cargar a registro, almacenar en memoria
Procesamiento de datos	ADD	3, 4, 5	Operación aritmética: operando1 \leftarrow operando1 + operando2
	SUB	6, 7, 8	Operación aritmética: operando1 \leftarrow operando1 - operando2
	CMP	9, 10, 11	Comparación: operando1 - operando2 (no actualiza el destino)
Control de flujo	JMP / Jxx / CALL / INT	12	Salto incondicional JMP, salto condicionales Jxx, subrutina CALL, llamar rutina de interrupción INT
Gestión de flujo	HLT / RET / IRET / CLI / STI	13	Detener CPU HLT, retorno subrutina RET, retornar de interrupción IRET, deshabilita interrupciones CLI, habilita interrupciones STI
Manejo de pila y E/S	OUT / IN / PUSH / POP	14	Enviar a puerto OUT, recibir desde puerto IN, poner en la pila PUSH, retirar de la pila POP
Miscelánea	AND / OR / XOR / NOT / NEG / INC / DEC	15	Operaciones lógicas y aritméticas

implementables a 16 opciones diferentes. Con el objetivo de ampliar el repertorio de instrucciones sin incrementar el tamaño de la codificación, se adoptó una estrategia de agrupación para el código de operación 15.

Bajo esta implementación, las instrucciones lógicas (AND, OR y XOR) comparten el código de operación 15 con las instrucciones aritméticas de un operando (INC, DEC, NEG y NOT). Esta decisión de diseño permite mantener la compatibilidad con los modos de direccionamiento establecidos para las instrucciones aritméticas de dos operandos (ADD, SUB y CMP), garantizando consistencia en la interfaz del simulador mientras se maximiza la funcionalidad dentro de las limitaciones impuestas por el esquema de codificación de 4 bits.

Esta solución representa un compromiso eficaz entre la simplicidad arquitectural y la capacidad funcional del simulador, permitiendo una mayor diversidad de operaciones sin comprometer la claridad pedagógica del diseño.

5.9.1 Etapa de ejecución de instrucciones

Finalmente, se detallan los pasos correspondientes a las instrucciones restantes del repertorio de instrucciones:

- Subrutinas

– **CALL Dirección**

4. $\text{MAR} \leftarrow \text{IP}$ – Obtener dirección destino.
5. $\text{MBR} \leftarrow \text{read}(\text{Memoria}[\text{MAR}]) \mid \text{IP} \leftarrow \text{IP} + 1$ – Leer la dirección e incrementar IP.
6. $\text{ri} \leftarrow \text{MBR} \mid \text{SP} \leftarrow \text{SP} - 1$ - Guardar la dirección en ri y decrementar SP.
7. $\text{MAR} \leftarrow \text{SP} \mid \text{MBR} \leftarrow \text{IP}$ - Preparar para apilar.
8. $\text{write}(\text{Memoria}[\text{MAR}]) \leftarrow \text{MBR} \mid \text{IP} \leftarrow \text{ri}$ - Guardar IP en la pila y saltar a la subrutina.

– **RET**

4. $\text{MAR} \leftarrow \text{SP}$ – Obtener dirección de retorno desde la pila.
5. $\text{MBR} \leftarrow \text{read}(\text{Memoria}[\text{MAR}])$ – Leer la dirección de retorno.
6. $\text{IP} \leftarrow \text{MBR} \mid \text{SP} \leftarrow \text{SP} + 1$ – Restaurar IP y actualizar SP.

- **Interrupciones**

– **INT Dirección**

4. $\text{MAR} \leftarrow \text{IP}$ – Obtener dirección destino.
5. $\text{MBR} \leftarrow \text{read}(\text{Memoria}[\text{MAR}]) \mid \text{IP} \leftarrow \text{IP} + 1$ – Leer la dirección e incrementar IP.
6. $\text{ri} \leftarrow \text{MBR} \mid \text{SP} \leftarrow \text{SP} - 1$ - Guardar la dirección en ri y decrementar SP.
7. $\text{MAR} \leftarrow \text{SP} \mid \text{MBR} \leftarrow \text{Flags}$ – Preparar para apilar los registros de estado.
8. $\text{write}(\text{Memoria}[\text{MAR}]) \leftarrow \text{MBR} \mid \text{update}(\text{Flags I=0})$ – Guardar los registros de estado y desactivar interrupciones.
9. $\text{MBR} \leftarrow \text{IP} \mid \text{SP} \leftarrow \text{SP} - 1$ – Guardar IP en la pila.
10. $\text{MAR} \leftarrow \text{SP}$ – Preparar para apilar.
11. $\text{write}(\text{Memoria}[\text{MAR}]) \leftarrow \text{MBR}$ - Guardar en la pila.
12. $\text{MAR} \leftarrow \text{ri}$ - Obtener dirección rutina.
13. $\text{MBR} \leftarrow \text{read}(\text{Memoria}[\text{MAR}])$ – Leer la dirección.
14. $\text{IP} \leftarrow \text{MBR}$ – Asigno dirección de la rutina de interrupción.

– **IRET**

4. $\text{MAR} \leftarrow \text{SP}$ – Obtener dirección de retorno desde la pila
5. $\text{MBR} \leftarrow \text{read}(\text{Memoria}[\text{MAR}])$ – Leer la dirección de retorno.
6. $\text{IP} \leftarrow \text{MBR} \mid \text{SP} \leftarrow \text{SP} + 1$ – Restaurar IP y actualizar SP.
7. $\text{MAR} \leftarrow \text{SP}$ – Preparar para leer los registros de estado.
8. $\text{MBR} \leftarrow \text{read}(\text{Memoria}[\text{MAR}])$ – Leer los registros de estado
9. $\text{Flags} \leftarrow \text{MBR} \mid \text{SP} \leftarrow \text{SP} + 1$ – Restaurar los registros de estado y actualizar SP.

– **CLI**

4. $\text{update}(\text{Flags I=0})$ – Desactivar interrupciones.

– **STI**

4. $\text{update}(\text{Flags I=1})$ – Activar interrupciones.

- E/S
 - OUT
 - * Dirección destino en registro **OUT DL, AL**
 4. **MAR** \leftarrow **DL** – Obtener dirección destino.
 5. **MBR** \leftarrow **AL** – Preparar el contenido a escribir.
 6. **write(E-S[MAR])** \leftarrow **MBR** – Escribir el contenido en el puerto de E/S.
 - * Dirección destino inmediato **OUT inmediato, AL**
 4. **MAR** \leftarrow **IP** – Obtener dirección del dato inmediato.
 5. **MBR** \leftarrow **read(Memoria[MAR])** | **IP** \leftarrow **IP + 1** – Leer el dato e incrementar IP.
 6. **MAR** \leftarrow **MBR** – Preparar la dirección destino.
 7. **MBR** \leftarrow **AL** – Preparar el contenido a escribir.
 8. **write(E-S[MAR])** \leftarrow **MBR** – Escribir el contenido en el puerto de E/S.
 - IN
 - * Dirección destino en registro **IN AL, DL**
 4. **MAR** \leftarrow **DL** – Obtener dirección del puerto de entrada.
 5. **MBR** \leftarrow **read(E-S[MAR])** – Leer el contenido del puerto.
 6. **AL** \leftarrow **MBR** – Almacenar el contenido en AL.
 - * Dirección destino inmediato **IN AL, inmediato**
 4. **MAR** \leftarrow **IP** – Obtener dirección del dato inmediato.
 5. **MBR** \leftarrow **read(Memoria[MAR])** | **IP** \leftarrow **IP + 1** – Leer el dato e incrementar IP.
 6. **MAR** \leftarrow **MBR** – Preparar la dirección del puerto de entrada..
 7. **MBR** \leftarrow **read(E-S[MAR])** – Leer el contenido del puerto.
 8. **AL** \leftarrow **MBR** – Almacenar el contenido en AL.
- Pila
 - POP Rx
 4. **MAR** \leftarrow **SP** – Obtener dirección de la pila.
 5. **MBR** \leftarrow **read(Memoria[MAR])** – Leer el contenido de la pila.
 6. **Rx** \leftarrow **MBR** | **SP** \leftarrow **SP + 1** – Almacenar el contenido en Rx y actualizar SP.
 - PUSH Ry
 4. **MBR** \leftarrow **Ry** | **SP** \leftarrow **SP - 1** – Preparar el contenido a escribir y actualizar SP.
 5. **MAR** \leftarrow **SP** – Preparar la dirección de la pila.
 6. **write(Memoria[MAR])** \leftarrow **MBR** – Escribir el contenido en la pila.
- Operaciones Lógicas y Aritméticas

- Operaciones con dos operandos (AND, OR, XOR)
- Siguen los mismos pasos de ejecución que las operaciones aritméticas ADD y SUB, con la diferencia en la operación realizada por la ALU.
- Operaciones con un operando (NOT, NEG, INC, DEC)
 - * Destino en registro (Rx)
 - INC Rx
 4. $Rx \leftarrow Rx + 1 \mid update(Flags)$ - Incrementar el valor del registro y actualizar los flags.
 - DEC Rx
 4. $Rx \leftarrow Rx - 1 \mid update(Flags)$ - Decrementar el valor del registro y actualizar los flags.
 - NOT Rx
 4. $Rx \leftarrow NOT Rx \mid update(Flags)$ - Realizar la operación lógica NOT y actualizar los flags.
 - NEG Rx
 4. $Rx \leftarrow CA2 Rx \mid update(Flags)$ - Realizar la operación de complemento a dos y actualizar los flags.
 - * Destino en memoria ([Dirección] o [BL])
 - Modo Directo ([Dirección])
 4. MAR $\leftarrow IP$ – Obtener dirección destino.
 5. MBR $\leftarrow read(Memoria[MAR]) \mid IP \leftarrow IP + 1$ – Leer la dirección e incrementar IP.
 6. MAR $\leftarrow MBR$ – Transferir dirección a MAR.
 7. MBR $\leftarrow read(Memoria[MAR])$ – Leer el dato.
 8. Ejecutar la operación: - INC: $MBR \leftarrow MBR + 1 \mid update(Flags)$
- DEC: $MBR \leftarrow MBR - 1 \mid update(Flags)$ - NOT: $MBR \leftarrow NOT MBR \mid update(Flags)$ - NEG: $MBR \leftarrow CA2 MBR \mid update(Flags)$
 9. $write(Memoria[MAR]) \leftarrow MBR$ – Escribir en memoria.
 - Modo Indirecto ([BL])
 4. MAR $\leftarrow BL$ – Transferir dirección de destino (en BL) a MAR.
 5. MBR $\leftarrow read(Memoria[MAR])$ – Leer el dato.
 6. Ejecutar la operación (igual que en el caso anterior).
 7. $write(Memoria[MAR]) \leftarrow MBR$ – Escribir en memoria.

5.9.2 Pila y subrutinas

El procesador implementa una pila como método de almacenamiento, accesible tanto por el usuario como por la CPU para su funcionamiento interno. La pila opera bajo

el esquema *Last In, First Out* (LIFO), es decir, el último elemento en ingresar es el primero en salir. Está ubicada en la memoria principal, comenzando en la dirección más alta (FFh) y creciendo hacia direcciones más bajas (FEh, FCh, etc.). El tope de la pila se gestiona mediante el registro SP, y cada elemento almacenado ocupa 8 bits.

Además, el procesador permite el uso de subrutinas, que son fragmentos de código reutilizables y pueden ser invocados desde cualquier parte del programa. Para llamar a una subrutina se utiliza la instrucción [CALL], que apila el valor actual de IP y salta a la dirección de la subrutina, modificando el IP para apuntar a la primera instrucción de la misma. El retorno se realiza mediante la instrucción [RET], que desapila la dirección previamente guardada y restaura el IP, permitiendo continuar la ejecución justo después de la llamada.

Ejemplo de subrutina:

```
1      mov al, 1
2      mov bl, 2
3      mov cl, 3
4      call sum3
5      ; ax = 6
6      hlt
7
8      ; suma al, bl y cl
9      sum3: add al, bl
10         add al, cl
11         ret
```

5.9.3 Interrupciones

VonSim8 incorpora un teclado y una pantalla como periféricos básicos de E/S, utilizados mediante rutinas de sistema invocadas por interrupciones.



Figura 5.25: Teclado y pantalla

El vector de interrupciones está preasignado en memoria desde `0x00h` hasta `0x07h`. Cada entrada ocupa 1 byte y almacena la dirección de inicio de la rutina asociada al número de interrupción. Las interrupciones pueden ser:

- De software: instrucción `INT n` ($n \in [0,7]$).
- De hardware: generadas por el PIC a través de la línea INTR (requieren $I = 1$) véase [5.9.10](#).

Interrupciones de software (números reservados):

- `INT 0`: termina la ejecución del programa (equivalente a `HLT`);
- `INT 6`: lee un carácter del teclado;
- `INT 7`: escribe una cadena de caracteres en pantalla.

Estas entradas están protegidas y no pueden modificarse por el usuario. Las rutinas residen en el monitor del sistema en `A0h`, `B0h` y `C0h`, respectivamente.

Secuencia de atención de una interrupción:

1. determinar el número de la interrupción (0-7);
2. apilar `Flags` y forzar $I = 0$;
3. apilar `IP`;
4. cargar $IP \leftarrow \text{MEM}[n]$ (dirección desde el vector) y ejecutar la rutina;
5. finalizar con `IRET`, que restaura `IP` y luego `Flags`.

No se admite anidamiento: si llega una nueva solicitud mientras otra está en servicio, queda pendiente en `IRR` hasta que se envía `EOI`.

5.9.4 Pantalla

La pantalla es un dispositivo de salida que permite mostrar caracteres. La forma de comunicarse con la pantalla es mediante una llamada al sistema. Esto es así por simplicidad, ya que una pantalla real es mucho más compleja.

Con la llamada INT 7 se escribe una cadena de caracteres en la pantalla. Recibe dos parámetros:

AL: longitud de la cadena a imprimir BL: dirección de memoria donde empieza la cadena

Ejemplo de hola mundo en lenguaje ensamblador para el simulador VonSim8:

```

1 cadena DB 'Hola!'
2 MOV BL, offset cadena
3 MOV AL, 5
4 INT 7
5 ; Se imprime Hola! (sin las comillas) en la pantalla.
6 HLT

```

Hay tres caracteres especiales:

- el carácter de retroceso (BS, 8 en decimal) borra el carácter previo;
- el carácter de salto de línea (LF, 10 en decimal) imprime, en efecto, un salto de línea — útil para no imprimir todo en una sola línea;
- el carácter de *form feed* (FF, 12 en decimal) limpia la pantalla.

5.9.5 Teclado

El teclado se modela como un vector de 16 posiciones, cada una capaz de almacenar un carácter ASCII. La pantalla, por su parte, permite visualizar caracteres, facilitando así la comprensión del manejo de entrada y salida de datos en una arquitectura computacional simplificada.

El teclado es un dispositivo de entrada que permite al usuario ingresar caracteres al sistema. La forma de comunicarse con el teclado es mediante una llamada al sistema. Esto es así por simplicidad, ya que un teclado real es mucho más complejo.

Con la llamada INT 6 se detiene la ejecución del código hasta que se presione una tecla en el teclado. El carácter que corresponda será guardado en la dirección de memoria almacenada en BL según su representación en ASCII.

```

1 car db 0
2 mov bl, offset car

```

```

3 int 6
4 hlt
5
6 ; El carácter escrito se almacenó en 'car'.
7 ; Por ejemplo, si el usuario presionó la tecla 'a', entonces
8 ; se almacena el valor 61h en 'car'.

```

5.9.6 Puertos de E/S

La memoria de entrada/salida está completamente separada de la memoria principal. Para interactuar con ella, se emplean exclusivamente las instrucciones [IN] y [OUT]. Cuando se requiere acceder a un módulo de entrada/salida, la [CPU] activa la señal IO/M, lo que provoca que un selector de chips (chip select) interprete la dirección presente en el bus de direcciones y envíe la señal de activación al módulo correspondiente.

El rango de direcciones asignado a la memoria de entrada/salida abarca desde 00h hasta FFh, lo que permite un total de 256 direcciones. A continuación, se presentan las direcciones de entrada/salida disponibles en el simulador, todas de 8 bits:

Tabla 5.17: Módulos, direcciones y nombres del simulador

Módulo	Dirección	Nombre
Timer	10h	CONT
	11h	COMP
PIC	20h	EOI
	21h	IMR
	22h	IRR
	23h	ISR
	24h	INT0
	25h	INT1
	26h	INT2
	27h	INT3
PIO	30h	PA
	31h	PB
	32h	CA
	33h	CB
Handshake	40h	DATA
	41h	STATE

5.9.7 Instrucciones IN y OUT

La comunicación con los módulos de entrada/salida se realiza a través de puertos, que son direcciones específicas dentro del espacio de direcciones del procesador. En este simulador, se implementan dos instrucciones fundamentales para gestionar esta comunicación:

- **OUT puerto, AL:** envía el contenido del registro AL al puerto especificado. El puerto es una dirección de 8 bits, lo que permite un total de 256 puertos diferentes (de 00h a FFh). Esta instrucción se utiliza para transmitir datos a dispositivos externos, como impresoras o pantallas.
- **IN AL, puerto:** recibe un byte desde el puerto especificado y lo almacena en el registro AL. Al igual que la instrucción OUT, el puerto es una dirección de 8 bits.

5.9.8 Módulo PIO (Leds e Interruptores)

El módulo de entrada/salida programada (Programmed Input-Output, PIO) actúa como interfaz entre la CPU y los dispositivos periféricos genéricos. Su diseño está basado en el controlador PPI 8255 de Intel, específicamente en su modo 0, pero incorpora modificaciones orientadas a simplificar su funcionamiento para fines educativos [54], [91].

El módulo cuenta con dos puertos bidireccionales de 8 bits (A y B) que pueden ser configurados de manera independiente. Su arquitectura incluye cuatro registros accesibles:

- PA (dirección 30h en el espacio de memoria E/S): registro de datos del puerto A
- PB (dirección 31h en el espacio de memoria E/S): registro de datos del puerto B
- CA (dirección 32h en el espacio de memoria E/S): registro de configuración del puerto A
- CB (dirección 33h en el espacio de memoria E/S): registro de configuración del puerto B

La configuración de cada puerto se define a través de sus respectivos registros de control (CA y CB). Cada bit del registro de configuración determina la dirección del bit correspondiente en el puerto de datos: un valor 0 configura el bit como salida, mientras que un valor 1 lo configura como entrada. Por ejemplo, si CA = 00001111b, los cuatro bits más significativos del puerto A funcionarán como salidas,

y los cuatro bits menos significativos como entradas. El puerto B opera de manera idéntica mediante su registro CB.

Este módulo PIO puede conectarse a diferentes tipos de dispositivos periféricos, como LEDs y interruptores, o a dispositivos más complejos como una impresora, proporcionando así una interfaz versátil para la comunicación con el mundo exterior.

Leds

Los diodos emisores de luz (LEDs) funcionan como dispositivos de salida y están conectados al puerto B del módulo PIO. Su control se realiza mediante la manipulación del registro de datos PB (dirección 31h) en conjunto con el registro de configuración CB (dirección 33h).

Para que los LEDs respondan correctamente, es necesario configurar previamente el puerto B como salida escribiendo el valor apropiado en el registro CB. Una vez configurado correctamente el PIO, cualquier modificación en el valor del registro PB se reflejará inmediatamente en el estado de los LEDs correspondientes. En caso de que la configuración del puerto sea incorrecta o se omita este paso, los LEDs permanecerán apagados independientemente de los valores escritos en PB.

Esta configuración permite controlar individualmente cada LED mediante los bits del puerto, ofreciendo flexibilidad para crear patrones luminosos o indicadores visuales en aplicaciones educativas y de demostración.

Las luces o LEDs están conectadas al puerto PB/CB del PIO y funcionan como dispositivos de salida. Su estado solo puede modificarse alterando el valor del puerto PB. Estos cambios se reflejarán en las luces si el PIO está configurado correctamente; de lo contrario, las luces permanecerán apagadas.

```
1 ; Enciende las luces (una sí, una no): 1010 1010b
2 ; 31h = PB --> puerto de datos para las luces (LEDs)
3 ; 33h = CB --> puerto de control para las luces
4
5 ; Configura todos los bits de PB como salida para controlar las luces
6 mov al, 0           ; 0000 0000b: todos los bits de PB en modo salida
7 out 33h, al        ; Escribe en CB para configurar PB como salida
8
9 ; Enciende las luces alternadas: 1010 1010b (170 decimal)
10 mov al, 170         ; 1010 1010b: enciende LEDs pares, apaga impares
11 out 31h, al        ; Escribe el valor en PB para actualizar las luces
12
13 hlt                ; Detiene la ejecución del programa
```

interruptores

Los interruptores (también denominados llaves) funcionan como dispositivos de entrada conectados al puerto A del módulo PIO. Su operación se gestiona mediante el registro de datos PA (dirección 30h) y el registro de configuración CA (dirección 32h).

Para que los interruptores operen correctamente, es fundamental configurar previamente el puerto A como entrada mediante el registro CA. Una vez establecida la configuración apropiada, cualquier cambio en el estado físico de los interruptores se reflejará automáticamente en los bits correspondientes del registro PA, permitiendo al programa leer su estado actual.

Es importante destacar que la comunicación es unidireccional desde los dispositivos hacia el procesador: aunque es posible escribir valores directamente en el registro PA mediante software, estos cambios no alterarán el estado físico de los interruptores. Los dispositivos de entrada mantienen su estado independientemente de las modificaciones realizadas por programa en sus registros asociados, garantizando así la integridad de la información proveniente del mundo exterior.

```
1 ; Leer el valor de las llaves como una contraseña
2 ; hasta que el usuario la adivine
3
4 clave db 15          ; Contraseña esperada: 00001111 (en decimal 15)
5 mensaje_ok db 'Bienvenido!' ; Mensaje si la contraseña es correcta
6
7 ; Configurar PA (Puerto A) como entrada
8 mov al, 15    ; 00001111b: configura los primeros 4 bits de PA como entrada
9 out 32h, al      ; Escribe en CA para configurar PA
10
11 bucle:
12   in al, 30h        ; Lee el valor actual de las llaves desde PA
13   cmp al, clave      ; Compara el valor leído con la contraseña
14   jz Mostrar_Mensaje ; Si coincide, salta a Mostrar_Mensaje
15   jmp bucle         ; Si no coincide, vuelve a intentar
16
17 Mostrar_Mensaje:
18   mov bl, offset mensaje_ok ; BL apunta al mensaje de éxito
19   mov al, 11           ; Longitud del mensaje (tiene 11 caracteres)
20   int 7
21   hlt                  ; Detiene la ejecución del programa
```

5.9.9 Módulo handshake (Impresora)

El módulo Handshake es un controlador especializado diseñado para facilitar la comunicación con impresoras que utilizan el protocolo Centronics. Su diseño se inspira en el controlador PPI 8255 de Intel, operando en modo “1”, pero incluye modificaciones específicas para simplificar su uso y comprensión en entornos educativos.

El módulo cuenta con una arquitectura simplificada basada en dos registros de 8 bits:

- **Registro de datos:** ubicado en la dirección 40h del espacio de memoria de E/S.
- **Registro de estado:** ubicado en la dirección 41h del espacio de memoria de E/S.

La estructura de estos registros se organiza de la siguiente manera:

Datos = DDDD DDDD
Estado = I ___ _SB

Funcionamiento del registro de datos

El registro de datos almacena el carácter que se desea imprimir, codificado en formato ASCII. Una característica destacada del módulo Handshake es su capacidad de automatización: cada vez que la CPU escribe un valor en este registro, el controlador genera automáticamente un flanco ascendente en la señal strobe, iniciando el proceso de impresión sin necesidad de intervención adicional por parte del software.

Funcionamiento del registro de estado

El registro de estado gestiona las señales de control y supervisión del protocolo de comunicación. Sus principales componentes son:

- **Bits S y B (strobe y busy):** los dos bits menos significativos controlan las señales de comunicación, con diferencias clave respecto a la implementación en el PIO:
 - El bit busy es de solo lectura y refleja automáticamente el estado de la impresora

- El bit strobe permanece normalmente en 0 y es gestionado automáticamente por el módulo. Cuando la CPU escribe un 1 en el bit strobe, se genera un flanco ascendente que transmite los datos almacenados en el registro de datos, retornando automáticamente a 0.
- **Bit I (interrupción):** El bit más significativo controla el sistema de interrupciones del módulo. Cuando este bit está habilitado ($I=1$) y la impresora se encuentra disponible ($B=0$), el módulo Handshake genera una interrupción por hardware a través de la línea INT2 del controlador PIC, notificando al sistema que la impresora está lista para recibir nuevos datos.

Esta implementación ofrece una interfaz intuitiva para el control de impresoras, automatizando aspectos críticos del protocolo de comunicación y manteniendo la flexibilidad necesaria para comprender los fundamentos de la interacción con dispositivos periféricos.

Ejemplo de uso del módulo Handshake con sondeo

Para imprimir utilizando el módulo Handshake, se deben seguir los siguientes pasos:

1. Verificar que el buffer no esté lleno (flag busy).
2. Escribir el carácter en el registro de datos.

Además de los caracteres ASCII comunes, el módulo admite caracteres especiales útiles para la impresión: - Salto de línea (LF, 10 en decimal): Imprime un salto de línea, evitando que todo el texto se imprima en una sola línea. - Form feed (FF, 12 en decimal): Limpia la impresora, equivalente a arrancar una hoja.

A continuación, se presenta un ejemplo en ensamblador para imprimir la cadena “Hola” utilizando el módulo Handshake:

```

1 ; Imprime el string 'Hola' en la impresora usando sondeo
2
3 dato DB 'Hola', 0          ; String a imprimir, terminado en (carácter nulo)
4
5 HS_DATA    EQU 40h        ; Dirección del registro de datos del Handshake
6 HS_STATUS  EQU 41h        ; Dirección del registro de estado del Handshake
7
8 ; Deshabilita las interrupciones del Handshake (bit 7 en 0)
9 IN  AL, HS_STATUS
10 AND AL, 0111111b         ; Fuerza el bit 7 a 0 (sin interrupciones)
11 OUT HS_STATUS, AL
12
13 ; Inicializa el puntero al string

```

```

14 MOV BL, OFFSET dato ; BL apunta al primer carácter del string
15
16 ; - Bucle principal: espera espacio en el buffer e imprime
17 Sondeo:
18   IN AL, HS_STATUS
19   AND AL, 00000001b ; Lee el flag busy (bit 0) 1=lleno, 0=libre
20   JZ ImprimirCadena ; Si busy=0, hay espacio y puede imprimir
21   JMP Sondeo ; Si busy=1, espera hasta que haya espacio
22
23 ImprimirCadena:
24   MOV AL, [BL] ; Carga el siguiente carácter del string
25   CMP AL, 0 ; Es el final del string carácter nulo
26   JZ fin ; Si sí, termina el programa
27
28   OUT HS_DATA, AL ; Envía el carácter al Handshake
29
30   INC BL ; Avanza al siguiente carácter del string
31   JMP Sondeo ; Repite el proceso para el próximo carácter
32
33 fin:
34   HLT ; Detiene la ejecución

```

Para mejorar la lectura del programa se utilizan constantes en ensamblador “EQU” para definir las direcciones de los puertos del módulo Handshake.

5.9.10 Módulo PIC (Controlador de Interrupciones)

El Programmable Interrupt Controller (PIC) es un módulo que actúa como intermediario entre los dispositivos que generan interrupciones y la CPU. Dado que la CPU dispone de una única línea de entrada para interrupciones, el PIC se encarga de recibir solicitudes de múltiples dispositivos y multiplexarlas en esta línea única.

Este módulo está basado en el PIC 8259A de Intel, aunque se han realizado modificaciones para simplificar su funcionamiento y adaptarlo a fines educativos.

Líneas de interrupción

El PIC dispone de 4 líneas de interrupción, denominadas INT0 a INT3 (aunque no todas son utilizadas). Cada línea está asociada a un registro de 8 bits en la memoria de E/S. Por ejemplo, la línea INT0 corresponde a la dirección 24h, la línea INT1 a la dirección 25h, y así sucesivamente hasta la línea INT3, que utiliza la dirección 26h. En estos registros se almacena el número de interrupción asociado a cada línea.

Cuando un dispositivo o módulo solicita una interrupción, el PIC envía a la CPU el número de interrupción almacenado en el registro correspondiente, desacoplando así el número de línea del número de interrupción.

Las líneas están conectadas a los siguientes dispositivos:

Tabla 5.18: Líneas de interrupción y dispositivos asociados

Línea	Módulo/Disp.
INT0	[Tecla F10]
INT1	[Timer]
INT2	[Handshake]
INT3	-

Registros y puerto EOI

El PIC cuenta con tres registros adicionales para gestionar las interrupciones y un puerto de comando para finalizar su atención. Cada bit de estos registros corresponde a una línea de interrupción; el bit menos significativo (bit 0) corresponde a la línea INT0 y el más significativo (bit 3) a la línea INT3.

- **Registro IMR (Interrupt Mask Register)**: Ubicado en la dirección 21h de la memoria de E/S, este registro permite enmascarar (inhabilitar) líneas de interrupción. Si un bit está en 1, la línea correspondiente está enmascarada y no generará interrupciones en la CPU. Si el bit está en 0, la línea está habilitada. Este registro puede ser modificado por la CPU.
- **Registro IRR (Interrupt Request Register)**: Ubicado en la dirección 22h de la memoria de E/S, este registro indica las interrupciones pendientes. Si un bit está en 1, la línea correspondiente tiene una interrupción pendiente. Este registro es de solo lectura para la CPU y es gestionado por el PIC.
- **Registro ISR (In-Service Register)**: Ubicado en la dirección 23h de la memoria de E/S, este registro muestra qué interrupción está siendo atendida en un momento dado. Si un bit está en 1, la línea correspondiente está en servicio. Este registro también es de solo lectura para la CPU y es gestionado por el PIC.
- **Puerto EOI (End Of Interrupt)**: Ubicado en la dirección 20h de la memoria de E/S, es un puerto de solo escritura. Cualquier byte escrito en 20h se interpreta como señal de fin de interrupción: el PIC limpia el bit correspondiente en el ISR y, si existen solicitudes pendientes no enmascaradas en el IRR, vuelve a activar INTR según su política de prioridad. La lectura de 20h no está definida.

Funcionamiento

Cuando una línea de interrupción se activa, el PIC la registra en el IRR. Si la línea no está enmascarada y no hay otra interrupción en servicio (es decir, si ISR = 00h), el PIC envía una señal de interrupción a la CPU activando la línea INTR.

El proceso de atención de la interrupción sigue estos pasos:

1. La CPU responde a la señal INTR enviando un pulso por la línea INTA.
2. Al recibir la señal, el PIC marca la línea como *in-service* en el registro ISR y la elimina del registro IRR.
3. El PIC envía a la CPU, a través del bus de datos, el número de interrupción correspondiente a la línea activa.
4. La CPU envía un segundo pulso por la línea INTA.
5. El PIC desactiva la línea INTR.

Para finalizar la atención de la interrupción, la CPU escribe el byte EOI (End of Interrupt) en la dirección 20h de la memoria de E/S. Al recibir este byte, el PIC desmarca la línea como in-service en el registro ISR. Si hay interrupciones pendientes, el PIC reactiva la línea INTR y repite el proceso.

Este PIC no admite interrupciones anidadas. Si ocurre una nueva interrupción mientras otra está siendo atendida, la nueva solicitud se encola en el registro IRR y se procesará una vez que la interrupción en curso haya finalizado, sin importar su prioridad.

Prioridades

Cuando hay múltiples interrupciones pendientes, el PIC atiende primero la de mayor prioridad. La prioridad de cada línea está determinada por su número de interrupción: las líneas con números más bajos tienen mayor prioridad. Por ejemplo, la línea INTO tiene mayor prioridad que la línea INT1.

Ejemplo de uso del módulo Handshake con interrupciones

A continuación, se presenta un ejemplo en ensamblador que utiliza el módulo Handshake junto con el PIC para imprimir la cadena “Hola” mediante interrupciones por hardware. En este ejemplo, se configura el PIC para que solo la línea INT2 (asociada al módulo Handshake) esté habilitada, y se define una rutina de interrupción que se ejecuta automáticamente cuando la impresora está lista para recibir un nuevo carácter.

```
1 ;
2 ; PROGRAMA Impresión de string usando Handshake con interrupciones
3 ; DESCRIPCIÓN Imprime el string 'Hola' en la impresora utilizando el módulo
4 ; Handshake con interrupciones por hardware (INT2)
5 ;
6 ;
7 ; SECCIÓN DE DATOS
8 mensaje    db 'Hola', 0      ; String a imprimir, terminado en carácter nulo
9 restantes   db 4            ; Contador de caracteres restantes por imprimir
10 puntero    db 0            ; Puntero al siguiente carácter (8 bits)
11 ;
12 ; CONSTANTES DE HANDSHAKE
13 HS_DATA     EQU 40h        ; Registro de datos del Handshake (puerto E-S)
14 HS_STATUS   EQU 41h        ; Registro de estado del Handshake (puerto E-S)
15 ;
16 ; CONSTANTES DE INTERRUPCIONES
17 ID          EQU 2          ; ID de la interrupción para Handshake (0-7)
18 IMR         EQU 21h        ; Registro de máscara de interrupciones del PIC
19 EOI         EQU 20h        ; Puerto para enviar End Of Interrupt al PIC
20 INT2        EQU 26h        ; Puerto para configurar la línea INT2
21 ;
22 ; PROGRAMA PRINCIPAL
23 ;
24 ;
25 ;
26 ; 1) CONFIGURACIÓN INICIAL
27 cli           ; Deshabilitar interrupciones globales
28 ;
29 ; 2) CONFIGURACIÓN DEL HANDSHAKE
30 ; Habilitar interrupciones del Handshake (bit 7 = 1)
31 in  al, HS_STATUS       ; Leer estado actual del Handshake
32 or  al, 10000000b       ; Activar bit 7 (habilitar interrupciones)
33 out HS_STATUS, al       ; Escribir configuración al Handshake
34 ;
35 ; 3) CONFIGURACIÓN DEL PIC (Controlador de Interrupciones)
36 ;
37 ; 3.1) Configurar máscara de interrupciones - Solo habilitar INT2
38 mov al, 11111011b       ; Máscara: habilita solo INT2 (bit 2=0),
39 ; resto deshabilitado
40 out IMR, al             ; Aplicar máscara al PIC
41 ;
42 ; 3.2) Asignar ID de interrupción a la línea INT2
43 mov al, ID               ; Cargar ID de interrupción (2)
44 out INT2, al             ; Configurar línea INT2 con este ID
45 ;
46 ; 3.3) Configurar vector de interrupción en memoria
47 mov bl, ID               ; BL = posición en tabla de vectores (ID=2)
```

```
48 mov [bl], int2_handler ; Almacenar dirección de rutina en vector[2]
49
50 ; 4) INICIALIZACIÓN DE VARIABLES
51 mov al, offset mensaje ; AL = dirección del primer carácter del string
52 mov puntero, al ; Guardar en variable puntero
53
54 ; 5) ENVIAR PRIMER CARÁCTER PARA INICIAR EL PROCESO
55 ; Esperar que la impresora esté lista
56 esperar_listo:
57     in al, HS_STATUS
58     and al, 00000001b ; Verificar bit busy
59     jnz esperar_listo ; Si busy=1, esperar
60
61 ; Enviar primer carácter
62 mov bl, puntero ; Cargar puntero
63 mov al, [bl] ; Obtener primer carácter
64 cmp al, 0 ; Es string vacío
65 jz fin ; Si está vacío, terminar
66
67 out HS_DATA, al ; Enviar primer carácter
68 inc bl ; Avanzar puntero
69 mov puntero, bl ; Guardar puntero actualizado
70 dec restantes ; Decrementar contador
71
72 ; 6) HABILITAR INTERRUPCIONES Y ESPERAR
73 sti ; Habilitar interrupciones globales
74
75 ; 7) BUCLE DE ESPERA
76 ; El programa principal espera hasta que se impriman todos los caracteres
77 bucle_espera:
78     cmp restantes, 0 ; Quedan caracteres por imprimir
79     jnz bucle_espera ; Si quedan, seguir esperando
80
81 ; 8) FINALIZACIÓN
82 fin:
83 hlt ; Detener ejecución del programa
84
85
86 ; RUTINA DE INTERRUPCIÓN INT2 - HANDSHAKE
87 ; DESCRIPCIÓN Se ejecuta automáticamente cuando la impresora está lista
88 ; para recibir un nuevo carácter (busy = 0)
89 ; ENTRADA Variable puntero = dirección del siguiente carácter a imprimir
90 ; SALIDA Carácter enviado a la impresora, puntero actualizado
91
92 org 80h
93 int2_handler:
94     ; PRESERVAR CONTEXTO
```

```

95  push al          ; Guardar registros que se van a modificar
96  push bl

97
98  ; VERIFICAR SI HAY MÁS CARACTERES
99  cmp restantes, 0    ; Quedan caracteres por imprimir
100 jz fin_interrupcion ; Si no quedan, terminar

101
102 ; OBTENER SIGUIENTE CARÁCTER
103 mov bl, puntero      ; BL = puntero al siguiente carácter
104 mov al, [bl]          ; AL = carácter apuntado por BL
105 cmp al, 0            ; Es el carácter nulo (fin de string)
106 jz fin_interrupcion ; Si es 0, terminar

107
108 ; ENVIAR CARÁCTER A LA IMPRESORA
109 out HS_DATA, al      ; Escribir carácter en el Handshake

110
111 ; ACTUALIZAR PUNTEROS Y CONTADORES
112 inc bl              ; Avanzar al siguiente carácter
113 mov puntero, bl      ; Guardar puntero actualizado
114 dec restantes       ; Decrementar contador de caracteres restantes

115
116 fin_interrupcion:
117 ; - ENVIAR EOI AL PIC
118 mov al, 20h           ; Señal de fin de interrupción
119 out EOI, al           ; Notificar al PIC

120
121 ; RESTAURAR CONTEXTO
122 pop bl                ; Restaurar registros preservados
123 pop al

124
125 ; RETORNO DE INTERRUPCIÓN
126 iret                  ; Retorno de interrupción

```

Tecla F10

La tecla F10 es un dispositivo que habilita una forma rápida y práctica de ejecutar una interrupción por hardware. Está conectada a la línea INT0 del [PIC]. Puede accionarse presionando la tecla F10 el teclado físicamente o haciendo clic en el “botón rojo de interrupción” en la interfaz gráfica.

```

1 ; Programa Contador de pulsaciones de la tecla F10 usando interrupciones
2
3 ; 1) Definiciones y variables
4
5 cantidad db 0    ; Variable almacena la cantidad de veces que se presionó F10
6

```

```

7 ID EQU 1      ; ID de la interrupción para F10 (puede ser 0-7)
8 IMR EQU 21h    ; Dirección del registro IMR (máscara de interrupciones)
9 EOI EQU 20h    ; Dirección para enviar End Of Interrupt al PIC
10 INTO EQU 24h   ; Dirección para configurar la línea INTO (F10)
11
12 ; 2) Inicialización del PIC y vector de interrupción
13
14 ; 2.1) Habilitar solo la interrupción de F10 (INTO)
15 mov al, 11111110b ; Habilita solo INTO (bit 0 en 0),
16 ; el resto deshabilitado
17 out IMR, al
18
19 ; 2.2) Configurar el ID de la interrupción para INTO
20 mov al, ID       ; Cargar el ID elegido para F10
21 out INTO, al
22
23 ; 2.3) Asociar el vector de interrupción con la subrutina atenderf10
24 mov bl, ID       ; BL = ID de la interrupción
25 mov [bl], atenderf10 ; Vector de interrupción: dirección de la rutina
26
27 ; 3) Bucle principal (espera activa)
28
29 loop: jmp loop    ; Espera indefinida
30 ; (el programa queda esperando interrupciones)
31
32 hlt                ; (Opcional) Detiene la CPU si sale del bucle
33
34 ; 4) Rutina de atención de la interrupción F10
35
36 org 50h            ; Dirección de la subrutina de atención
37
38 atenderf10:
39   inc cantidad      ; Incrementa el contador cada vez que se presiona F10
40   mov al, 20h        ; Código de End Of Interrupt (EOI)
41   out EOI, al       ; Notifica al PIC que terminó la atención
42   iret              ; Retorna de la interrupción

```

5.9.11 Módulo timer

El timer es un módulo que dispone de dos registros internos:

- Registro **CONT**: ubicado en la dirección 10h de la memoria E/S.
- Registro **COMP**: ubicado en la dirección 11h de la memoria E/S.

Este módulo está basado en el PIT 8253 de Intel, aunque se han realizado modificaciones para simplificar su funcionamiento. El registro COMP permite establecer el valor de comparación con CONT. Si se escribe un valor de 0 en COMP, el timer se desactiva y no genera interrupciones. Cualquier otro valor habilita el timer y permite su funcionamiento normal.

El reloj es un dispositivo que hace tic cada segundo. Al hacer tic, incrementa el registro CONT del [timer] en uno.

```
1 ; Programa Imprime 'Hola' a los 10 segundos de iniciado y luego termina.
2 ; Utiliza la interrupción del TIMER (ID = 5).
3
4 mensaje db 'Hola'           ; Mensaje a imprimir
5 imprimio db 0              ; Flag para saber si ya imprimió
6
7 ; Definición de direcciones de registros de dispositivos
8 CONT equ 10h                ; Registro de conteo del timer
9 COMP equ 11h                ; Registro de comparación del timer
10
11 EOI equ 20h                 ; End Of Interrupt (para PIC)
12 IMR equ 21h                 ; Interrupt Mask Register (PIC)
13 INT1 equ 25h                ; Registro de vector de interrupción 1
14
15 ; Habilitar interrupciones del timer
16 ; IMR = 1111 1101b (solo habilita interrupciones del timer y teclado)
17 mov al, 11111101b           ; Habilita interrupciones del timer (bit 1 en 0)
18 out IMR, al
19
20 ; Configurar vector de interrupción del timer
21 mov al, 5                  ; ID de interrupción del timer
22 out INT1, al               ; Asigna rutina de atención a la posición 5
23
24 ; Instalar rutina de interrupción en el vector
25 mov bl, 5                  ; Vector de interrupción 5
26 mov [bl], imp_msj          ; Apunta a la rutina imp_msj
27
28 ; Configurar timer para 3 segundos
29 mov al, 10                 ; Valor de comparación (10 segundos)
30 out COMP, al
31
32 mov al, 0                  ; Reinicia el contador del timer
33 out CONT, al
34
35 ; Esperar a que se imprima el mensaje
36 loopinf: cmp imprimio, 0 ; Ya imprimió
37     jz loopinf             ; Si no, sigue esperando
38
39 hlt                      ; Termina el programa
```

```
40  
41 ; Rutina de interrupción del timer  
42 org 50h  
43 imp_msj:  
44     mov bl, offset mensaje ; Dirección del mensaje  
45     mov al, 4              ; Servicio de impresión  
46     int 7                 ; Llama a la interrupción de impresión  
47     mov imprimio, 1        ; Marca que ya imprimió  
48     mov al, 20h             ; Señal de fin de interrupción  
49     out EOI, al            ; Notifica al PIC  
50     iret                  ; Retorna de la interrupción
```

5.10 Validación pedagógica del simulador

Con el objetivo de evaluar rigurosamente la eficacia pedagógica del simulador VonSim8 en la enseñanza de arquitectura de computadoras, se realizó una prueba piloto durante el segundo cuatrimestre de 2025. Esta experiencia estuvo dirigida a estudiantes de segundo año de la Licenciatura en Sistemas y se centró en medir el nivel de comprensión alcanzado respecto a conceptos fundamentales, como el ciclo de instrucción, la manipulación de registros y la gestión de interrupciones.

5.10.1 Objetivos de la validación

- Analizar el impacto del simulador en la comprensión de los principios esenciales de la arquitectura de computadoras.
- Identificar mejoras en la habilidad de los estudiantes para analizar y depurar programas en lenguaje ensamblador.
- Evaluar la percepción de usabilidad y utilidad didáctica de la herramienta desde la perspectiva de los usuarios.

5.10.2 Población de estudio

La validación se realizó sobre una muestra de 14 estudiantes matriculados en la asignatura Arquitectura de Computadoras y 2 docentes a cargo. La participación fue voluntaria, asegurando la representatividad del perfil académico habitual en la materia.

5.10.3 Instrumentos de recolección de datos (ver Apéndice: Anexo B 6.2)

Para la validación se emplearon instrumentos diseñados para medir la calidad didáctica y recoger opiniones cualitativas tanto de estudiantes como de docentes. Los instrumentos, adaptados al contexto universitario y alineados con los objetivos pedagógicos del capítulo, fueron los siguientes:

- **Cuestionario de conocimiento:** Aplicado a estudiantes, utiliza una escala tipo Likert (por ejemplo, 1-5) para evaluar la satisfacción y percepción de eficacia en aspectos clave del simulador, tales como visualización, editor, progresividad de instrucciones, métricas, documentación y manejo de periféricos. Incluye además un apartado para comentarios abiertos.
- **Entrevistas semiestructuradas a docentes:** Este instrumento permitió recoger valoraciones expertas sobre el potencial pedagógico del simulador, así como comparaciones con herramientas previas y sugerencias de mejora. Las preguntas se organizaron en categorías como usabilidad, claridad conceptual, impacto en la motivación estudiantil y utilidad de la documentación.

5.10.4 Análisis de resultados (ver Apéndice: Anexo C 6.3)

La validación pedagógica del simulador VonSim8 se realizó mediante dos instrumentos principales: una encuesta de retroalimentación aplicada a estudiantes y entrevistas semiestructuradas realizadas a docentes. A continuación, se presenta un análisis integral de los resultados obtenidos a partir de ambos enfoques.

1. Encuesta de retroalimentación a estudiantes

La encuesta fue respondida por 14 estudiantes y permitió evaluar la claridad de los contenidos teóricos, la calidad del material didáctico y la utilidad del simulador. Los resultados reflejan una valoración positiva de la experiencia educativa, destacándose los siguientes aspectos:

- **Resultados cuantitativos:**
 - Los recursos visuales obtuvieron la puntuación más alta (promedio 4.21), seguidos por la claridad de los modos de direccionamiento (4.14) y la conexión entre teoría y práctica (4.14). Estos resultados evidencian la efectividad del simulador para facilitar la comprensión de conceptos clave.

- La visualización de las microoperaciones fue el ítem con menor puntuación (3.57), lo que sugiere la necesidad de optimizar este aspecto para mejorar la experiencia de aprendizaje.
- **Resultados cualitativos:**
 - Los estudiantes destacaron como aspectos más útiles la posibilidad de observar paso a paso la ejecución de instrucciones y los cambios en registros y memoria, así como el carácter práctico e intuitivo del simulador.
 - Las principales dificultades mencionadas incluyeron la interpretación de las banderas, las animaciones de lectura/escritura en memoria y las instrucciones de salto, lo que indica áreas de mejora en la documentación y los recursos de apoyo.
- **Conclusión:** El simulador VonSim8 fue percibido como una herramienta pedagógica eficaz, que facilita la comprensión del ciclo de instrucción y promueve un aprendizaje activo. Se recomienda continuar perfeccionando las representaciones visuales y ampliar los materiales de apoyo para abordar las dificultades identificadas.

2. Entrevistas semiestructuradas a docentes

Las entrevistas realizadas a dos docentes especializados en la enseñanza de Arquitectura de Computadoras permitieron recoger valoraciones cualitativas sobre el simulador VonSim8. Los resultados se agrupan en las siguientes categorías:

- **Usabilidad y claridad conceptual:**
 - Los docentes valoraron positivamente la claridad de las explicaciones teóricas (promedio 4.20) y la utilidad de los recursos visuales (4.50), destacando su efectividad para enseñar conceptos abstractos como el ciclo de instrucción y el flujo de datos.
 - Se identificaron oportunidades de mejora en la interfaz del simulador, especialmente para estudiantes principiantes, y en la representación de las microoperaciones.
- **Impacto en la motivación estudiantil:**
 - Ambos docentes coincidieron en que el simulador incrementa la motivación de los estudiantes al ofrecer una experiencia práctica e interactiva, que conecta teoría y práctica de manera efectiva.
- **Sugerencias de mejora:**
 - Los docentes recomendaron optimizar la representación de las banderas y las animaciones de lectura/escritura en memoria, así como mejorar la accesibilidad de la interfaz para estudiantes con poca experiencia.

- **Conclusión:** Los docentes consideraron que VonSim8 es una herramienta didáctica eficaz, que facilita la enseñanza de la arquitectura x86 y promueve un aprendizaje progresivo. Las sugerencias recogidas servirán de base para orientar futuras mejoras en el desarrollo del simulador.

3. Análisis integral

La integración de los resultados de ambos instrumentos permite obtener una visión completa del impacto pedagógico del simulador VonSim8:

- **Fortalezas:**
 - El simulador facilita la comprensión de conceptos complejos como el ciclo de instrucción, el flujo de datos y la interacción entre registros y memoria.
 - Los recursos visuales y la conexión entre teoría y práctica son altamente valorados tanto por estudiantes como por docentes.
 - La activación progresiva del repertorio de instrucciones y las métricas de rendimiento contribuyen a un aprendizaje gradual y reflexivo.
- **Áreas de mejora:**
 - Optimizar la representación de las microoperaciones y las animaciones de lectura/escritura en memoria.
 - Mejorar la accesibilidad de la interfaz para estudiantes principiantes.
 - Ampliar los materiales de apoyo y las explicaciones sobre aspectos técnicos como las banderas y las instrucciones de salto.
- **Conclusión general:** El simulador VonSim8 se consolida como una herramienta educativa integral, que combina visualización interactiva, ejecución progresiva y análisis de rendimiento. Los resultados de la validación confirman su efectividad pedagógica y destacan su potencial para seguir mejorando en función de las sugerencias recogidas. En conclusión, la triangulación de datos cuantitativos (encuestas a estudiantes) y cualitativos (entrevistas a docentes) validó la pertinencia pedagógica de VonSim8. Las fortalezas identificadas ratifican las decisiones de diseño, mientras que las oportunidades de mejora proporcionan una hoja de ruta clara para futuras iteraciones.

5.11 Aportes y contribuciones del simulador VonSim8

El simulador VonSim8 constituye una valiosa contribución al ámbito educativo en la enseñanza de arquitectura de computadoras, destacándose por los siguientes aspectos:

1. **Simplificación pedagógica de la arquitectura x86:** VonSim8 implementa una arquitectura simplificada de 8 bits, diseñada para reducir la complejidad inherente a la arquitectura x86. Esta simplificación permite a los estudiantes concentrarse en conceptos fundamentales, como el ciclo de instrucciones, la interacción entre registros y la gestión de interrupciones, sin verse abrumados por detalles técnicos avanzados. La selección de un repertorio reducido de instrucciones se fundamenta en principios de la psicología cognitiva, que destacan la importancia de introducir gradualmente conceptos técnicos para mejorar la retención y reducir la sobrecarga cognitiva [21], [78].
2. **Activación progresiva del repertorio de instrucciones:** El simulador habilita de manera escalonada un repertorio reducido de instrucciones, en correspondencia con el avance de los contenidos curriculares. Este enfoque progresivo facilita la asimilación de conceptos más complejos, mitigando la sobrecarga cognitiva y promoviendo un aprendizaje gradual y efectivo.
3. **Visualización interactiva del ciclo de instrucciones:** VonSim8 incorpora una representación visual basada en el modelo de Nivel de Transferencia entre Registros (RTL), que permite observar el flujo de datos y las señales de control en cada etapa del ciclo de instrucción. Esta funcionalidad refuerza la conexión entre teoría y práctica, facilitando la comprensión de procesos internos del procesador mediante una representación clara y dinámica.
4. **Simulación de periféricos y gestión de interrupciones:** El simulador incluye un módulo de entrada/salida programada (PIO) y un vector de interrupciones predefinido, que emulan interacciones con dispositivos externos como teclados y monitores. Estas características permiten explorar conceptos clave como la asincronía y el manejo de eventos, esenciales para comprender sistemas reales.
5. **Entorno integrado de desarrollo y simulación:** VonSim8 ofrece un editor de ensamblador con funciones como resaltado de sintaxis, autocompletado y ejemplos predefinidos, junto con un simulador que permite ejecutar programas paso a paso o de manera continua. Este entorno integrado mejora la experiencia del usuario y fomenta un aprendizaje práctico y autónomo.
6. **Métricas de rendimiento y análisis cuantitativo:** El simulador proporciona indicadores clave como ciclos por instrucción (CPI), tiempo de CPU y tiempo de ciclo, que permiten a los estudiantes analizar la eficiencia de sus programas. Estas métricas promueven una comprensión integral del rendimiento del procesador y su impacto en la ejecución de programas.
7. **Documentación y recursos de apoyo:** VonSim8 incluye una documentación clara y accesible, complementada con tutoriales interactivos y ejemplos prácticos que guían al estudiante en el uso del simulador. Este recurso fomenta un aprendizaje activo y reflexivo, facilitando la adquisición de competencias técnicas.

8. **Compatibilidad y accesibilidad:** El simulador es de código abierto y se distribuye bajo licencias que permiten su estudio, modificación y mejora continua. Su diseño accesible y sostenible asegura su utilidad en diversos contextos educativos.

En resumen, VonSim8 se presenta como una herramienta educativa integral que combina visualización interactiva, ejecución progresiva y análisis de rendimiento. Su diseño responde a las necesidades pedagógicas y técnicas de la enseñanza de arquitectura de computadoras, promoviendo un aprendizaje activo, reflexivo y centrado en la comprensión de los principios fundamentales de la disciplina.

5.12 Resumen del capítulo

En este capítulo se ha descrito el diseño y desarrollo del simulador VonSim8, destacando sus características pedagógicas y técnicas. Las modificaciones implementadas, como la simplificación de la arquitectura y la visualización interactiva, están orientadas a facilitar la enseñanza de la arquitectura de computadoras. Estas mejoras aseguran que el simulador no solo sea una herramienta funcional, sino también un recurso educativo efectivo.

La efectividad pedagógica del simulador fue evaluada mediante una prueba piloto con estudiantes de la asignatura Arquitectura de Computadoras, tal como se detalló en la sección (5.10). Los resultados de dicha validación, que incluyeron encuestas y entrevistas, confirmaron el valor de la herramienta para la comprensión de conceptos clave.

Asimismo, la validación formal mediante el modelado xDEVS permitió demostrar la correctitud funcional y temporal del simulador VonSim8, evidenciando la sincronización precisa entre los componentes y la reproducción fiel de las restricciones arquitecturales de Von Neumann. Este enfoque no solo aporta rigor científico al desarrollo, sino que también facilita la instrumentación de métricas objetivas de rendimiento y la extensión futura del simulador. La integración del formalismo DEVS refuerza el valor pedagógico de la herramienta, al ofrecer una representación explícita y trazable de los procesos internos del procesador.

Capítulo 6

Apéndices

6.1 Anexo A: Protocolo de Entrevista Semiestructurada

Objetivo:

Relevar necesidades, experiencias y percepciones de docentes especializados en la enseñanza de Arquitectura de Computadoras, con el fin de fundamentar los requisitos funcionales y pedagógicos de una herramienta de simulación orientada a la arquitectura x86.

Tipo de entrevista: Semiestructurada

Duración estimada: 45–60 minutos

Participantes: Docentes universitarios con experiencia en la enseñanza de asignaturas vinculadas a Arquitectura de Computadoras

Modo de registro: Grabación de audio (previo consentimiento informado) y notas del entrevistador

6.1.1 Introducción (a cargo del entrevistador)

- Breve presentación personal y del objetivo de la entrevista.
- Explicación sobre la confidencialidad de los datos.
- Solicitud de consentimiento para grabar la entrevista.
- Aclaración sobre la posibilidad de no responder a alguna pregunta o interrumpir la entrevista en cualquier momento.

6.1.2 Datos generales del entrevistado

- Universidad o institución en la que trabaja:

- **Asignatura(s) que dicta relacionadas con arquitectura de computadoras:**
- **Años de experiencia docente en el área:**
- **Nivel en el que dicta la asignatura:** (Grado, Posgrado, Técnico, Otro)

6.1.3 Preguntas principales

A. Enseñanza y dificultades

- ¿Cuáles considera que son los principales desafíos que enfrentan los estudiantes al aprender los conceptos de arquitectura de computadoras?
- ¿Qué contenidos o temas observa que generan mayores dificultades de comprensión?
- ¿Cómo aborda actualmente la enseñanza del lenguaje ensamblador y el ciclo de instrucción?

B. Experiencia con herramientas de simulación

- ¿Utiliza o ha utilizado herramientas de simulación en sus clases? ¿Cuáles?
- ¿Qué ventajas ha encontrado en el uso de estas herramientas?
- ¿Qué limitaciones o dificultades ha identificado en las herramientas actualmente disponibles?

C. Requisitos deseables en una herramienta

- En su opinión, ¿qué funcionalidades debería tener una herramienta de simulación para ser útil en el proceso de enseñanza?
- ¿Considera importante que la herramienta incluya visualizaciones gráficas del ciclo de instrucción o del flujo de datos entre componentes?
- ¿Cree que el soporte para interrupciones y periféricos (por ejemplo, teclado o pantalla) aporta valor al aprendizaje?
- ¿Considera que la posibilidad de activar progresivamente instrucciones del repertorio x86 según el avance del curso puede beneficiar el proceso de enseñanza-aprendizaje?
- ¿Qué importancia le asigna a la incorporación de métricas de rendimiento (como ciclos por instrucción, tiempo de CPU, etc.) en una herramienta educativa?

D. Interfaz y accesibilidad

- ¿Qué aspectos de la interfaz considera prioritarios en una herramienta pensada para estudiantes?

- ¿Debería contemplarse la accesibilidad para personas con discapacidad? ¿De qué forma?

6.1.4 Cierre

- ¿Desea agregar algo más que no hayamos preguntado?
- Agradecimiento por el tiempo y colaboración.

Nota: El análisis posterior de las entrevistas será realizado de manera confidencial y con fines exclusivamente académicos.

6.2 Anexo B: Instrumentos de recolección de datos

En este anexo se presentan los instrumentos empleados para la validación pedagógica del simulador VonSim8. Se incluyen encuestas, pruebas objetivas, guía de entrevistas y plantilla de registro de uso, con sus respectivos ítems, escalas y pautas de codificación. Las encuestas y pruebas se aplican en dos momentos (pre y post) para medir la ganancia de aprendizaje.

6.2.1 1) Encuesta de retroalimentación – Unidad ciclo de instrucción y simulador VonSim8 estudiantes

Sección 1: Experiencia previa

1. Antes de esta unidad, ¿qué tanto conocías el ciclo de instrucción?
2. ¿Habías usado simuladores educativos de este tipo antes?

Sección 2: Calidad del material didáctico

(Escala 1–5: 1 = Totalmente en desacuerdo, 5 = Totalmente de acuerdo)

3. La explicación de las etapas de captación (fetch) y ejecución (execute) fue clara.
4. Los ejemplos con instrucciones (MOV, ADD, SUB, JMP) me ayudaron a entender las microoperaciones.
5. La explicación de los modos de direccionamiento (inmediato, directo, indirecto, entre registros) fue suficiente y comprensible.
6. Las actividades propuestas (análisis, ejercicios, simulaciones) me ayudaron a consolidar los conceptos.
7. Las tablas, esquemas e imágenes del material fueron útiles para mi aprendizaje.

Sección 3: Valor del simulador VonSim8

(Escala 1–5: 1 = Totalmente en desacuerdo, 5 = Totalmente de acuerdo) 8. El simulador me permitió visualizar con claridad los cambios en registros y memoria durante la ejecución de los programas. 9. El simulador facilitó la conexión entre la teoría y la práctica. 10. El simulador es intuitivo y fácil de usar. 11. Ver las microoperaciones y los componentes internos en el simulador mejoró mi comprensión del ciclo de instrucción.

Sección 4: Reflexión y sugerencias

(Respuesta abierta, párrafo largo) 12. ¿Cuál fue el aspecto más útil o interesante de la unidad (material o simulador)? 13. ¿Qué parte del material o del simulador te resultó más confusa o difícil? 14. Después de completar esta unidad, ¿cuánto sientes que mejoraron tus conocimientos sobre el ciclo de instrucción? Gracias por tu tiempo. Tus respuestas sinceras nos ayudarán a mejorar esta unidad y el simulador VonSim8 para futuros estudiantes.

6.2.2 2) Entrevista semiestructurada a docentes sobre el simulador VonSim

Objetivo:

Recoger valoraciones cualitativas de los docentes sobre la calidad del material didáctico y la utilidad del simulador VonSim8 en la enseñanza del ciclo de instrucción y conceptos relacionados con la arquitectura de computadoras.

Sección 1: Experiencia previa

1. ¿Cuánto tiempo llevas enseñando temas relacionados con la arquitectura de computadoras?
2. ¿Has utilizado simuladores educativos en tus clases anteriormente? Si es así, ¿cuáles y cómo ha sido tu experiencia con ellos?
3. Antes de utilizar el simulador VonSim8, ¿qué tan familiarizado/a estabas con el ciclo de instrucción y los conceptos relacionados?

Sección 2: Calidad del material didáctico

4. ¿Cómo evalúas la claridad y completitud de las explicaciones proporcionadas en el material didáctico sobre las etapas del ciclo de instrucción (captación y

ejecución)?

5. ¿Consideras que las tablas, esquemas e imágenes del material son útiles para enseñar conceptos abstractos como modos de direccionamiento y microoperaciones? ¿Por qué?
6. ¿Qué opinas de los ejemplos prácticos (MOV, ADD, SUB, JMP) incluidos en el material? ¿Crees que son adecuados para ilustrar las microoperaciones y el flujo de datos?
7. ¿El material didáctico permite a los estudiantes conectar la teoría con la práctica de manera efectiva? ¿Podrías dar un ejemplo?
8. ¿Qué opinas de las actividades propuestas (análisis, ejercicios, simulaciones)? ¿Crees que son apropiadas para consolidar los conceptos clave?

Sección 3: Valor del simulador VonSim8

9. ¿Cómo evalúas la capacidad del simulador para facilitar la visualización de los cambios en registros y memoria durante la ejecución de los programas?
10. ¿Crees que el simulador ayuda a los estudiantes a comprender mejor el ciclo de instrucción y las microoperaciones? ¿Por qué?
11. ¿Qué opinas de la representación gráfica del flujo de datos y las señales de control en el simulador? ¿Es clara y didáctica?
12. ¿Cómo describirías la interfaz del simulador? ¿Es intuitiva y fácil de usar para los estudiantes?
13. ¿Consideras que la activación progresiva del repertorio de instrucciones es útil para evitar la sobrecarga cognitiva en los estudiantes? ¿Por qué?
14. ¿Qué tan útiles consideras las métricas de rendimiento (CPI, tiempo de CPU, etc.) para enseñar conceptos relacionados con la eficiencia del procesador?

Sección 4: Reflexión y sugerencias

(Respuesta abierta, párrafo largo)

15. ¿Qué aspectos del simulador VonSim8 consideras más útiles para la enseñanza del ciclo de instrucción y la arquitectura de computadoras?
16. ¿Qué aspectos del simulador o del material didáctico crees que podrían mejorarse?

17. ¿Cómo percibes el impacto del simulador en la motivación y comprensión de los estudiantes?
18. ¿Recomendarías el uso del simulador VonSim8 en otros cursos o contextos educativos? ¿Por qué?

6.3 Anexo C: Análisis de los resultados

6.3.1 Encuesta de retroalimentación estudiantes

La encuesta de retroalimentación sobre la unidad **Ciclo de Instrucción y simulador VonSim8** fue respondida por un total de 14 estudiantes. El objetivo de este instrumento fue evaluar la claridad de los contenidos teóricos, la calidad del material didáctico, la utilidad del simulador y el impacto percibido en el aprendizaje. A continuación, se presenta un análisis detallado de los resultados obtenidos.

1. Experiencia previa de los estudiantes

En cuanto al conocimiento inicial sobre el ciclo de instrucción, el 64,3 % de los estudiantes manifestó poseer poco o nulo conocimiento, mientras que el 35,7 % indicó tener un nivel moderado o alto. Estos resultados reflejan que la mayoría de los participantes partía de una base conceptual limitada, lo que justifica la necesidad de incluir actividades introductorias y recursos visuales que faciliten la comprensión de los conceptos fundamentales.

Respecto al uso de simuladores educativos, el 64,3 % de los estudiantes indicó no haber utilizado previamente herramientas de este tipo. Este dato evidencia que el simulador VonSim8 representó una experiencia novedosa para la mayoría del grupo, lo que pone en relieve la importancia de una guía docente clara y materiales de apoyo que orienten su uso.

2. Valoración del material didáctico

Los resultados obtenidos muestran una valoración positiva del material teórico y de las actividades propuestas. Los promedios de calificación fueron los siguientes:

En conjunto, estos resultados evidencian un alto grado de satisfacción con el material didáctico, destacándose especialmente los recursos visuales y los ejemplos aplicados como elementos clave para la comprensión de los contenidos.

3. Valoración del simulador VonSim8

Tabla 6.1: Valoración de aspectos clave del simulador VonSim8

Aspecto evaluado	Promedio	Interpretación
Claridad de los conceptos teóricos (fetch/execute)	3.86	Buena comprensión general, aunque con margen de mejora en la claridad conceptual.
Ejemplos con instrucciones (MOV, ADD, SUB, JMP)	4.07	Los ejemplos prácticos facilitaron la comprensión de las microoperaciones.
Explicación de los modos de direccionamiento	4.14	Los estudiantes valoraron positivamente la claridad en la exposición de estos conceptos.
Actividades (análisis, ejercicios, simulaciones)	4.07	Las actividades fueron percibidas como útiles para consolidar los aprendizajes.
Tablas, esquemas e imágenes	4.21	Los recursos visuales fueron altamente valorados como apoyo al aprendizaje.

El uso del simulador VonSim8 fue valorado de manera positiva, aunque con ciertos aspectos a mejorar. Los resultados fueron los siguientes:

Tabla 6.2: Evaluación de aspectos clave del simulador VonSim8

Aspecto evaluado	Promedio	Interpretación
Visualización de registros y memoria	3.79	Buena valoración general, aunque algunos estudiantes señalaron dificultades para interpretar los cambios mostrados.
Conexión entre teoría y práctica	4.14	El simulador facilitó la aplicación práctica de los contenidos teóricos.
Facilidad de uso e intuición	3.93	El simulador fue percibido como accesible y comprensible, aunque se sugiere optimizar la interfaz.
Visualización de microoperaciones	3.57	Aspecto menos valorado, posiblemente por la complejidad visual o falta de familiaridad inicial.

Los resultados indican que el simulador cumplió adecuadamente su función pedagógica, ayudando a los estudiantes a comprender los procesos internos del procesador. No obstante, se observan oportunidades de mejora en la representación visual de las microoperaciones y en la claridad de las animaciones.

4. Análisis cualitativo de las respuestas abiertas Aspectos más útiles o interesantes

Los estudiantes destacaron como elementos más valiosos:

- La posibilidad de visualizar paso a paso la ejecución de instrucciones dentro del ciclo de instrucción.
- La observación directa de los cambios en los registros y la memoria durante la ejecución.
- La comprensión del funcionamiento interno del procesador y de las microoperaciones.
- El carácter práctico e intuitivo del simulador y las actividades de aplicación.

Aspectos más difíciles o confusos

Entre las principales dificultades mencionadas se encuentran:

- El funcionamiento de las banderas (flags) y su activación durante la ejecución.
- La lectura y escritura en memoria, particularmente las animaciones del simulador.
- El lenguaje ensamblador y las instrucciones de salto (JMP).
- La interpretación de las microoperaciones, percibida por algunos como rápida o poco clara.

Estos comentarios sugieren que, si bien el contenido teórico fue bien comprendido, ciertos detalles técnicos del simulador requieren explicaciones adicionales o recursos interactivos complementarios.

5. Impacto percibido en el aprendizaje

En la pregunta final, donde se consultó cuánto mejoraron sus conocimientos sobre el ciclo de instrucción, los estudiantes otorgaron un promedio de 4.00/5. Esto refleja que la mayoría percibió una mejora significativa en su comprensión del tema, atribuida a la combinación entre el material teórico, las actividades prácticas y el uso del simulador VonSim8.

6. Conclusiones generales

Los resultados de la encuesta permiten concluir que la experiencia educativa fue altamente positiva, tanto en lo pedagógico como en el uso de la tecnología. El diseño de la unidad, basado en la combinación de teoría, práctica y simulación, favoreció el aprendizaje significativo del ciclo de instrucción.

El simulador VonSim8 se consolidó como una herramienta didáctica eficaz para vincular los conceptos teóricos con su aplicación práctica, permitiendo a los estudiantes observar de forma dinámica el comportamiento de los registros, la memoria y las microoperaciones. Sin embargo, se identificaron áreas de mejora relacionadas con la claridad de las animaciones, la representación de las banderas y la comprensión de la memoria.

En síntesis, los resultados confirman que la implementación del simulador VonSim8 y el enfoque didáctico adoptado contribuyen de manera efectiva al desarrollo de competencias conceptuales y procedimentales en el estudio de la arquitectura de computadoras.

6.3.2 Entrevistas a docentes

Las entrevistas semiestructuradas realizadas a dos docentes especializados en la enseñanza de Arquitectura de Computadoras tuvieron como objetivo relevar sus percepciones, experiencias y necesidades en relación con el uso de herramientas de simulación, particularmente el simulador VonSim8. A continuación, se presenta un análisis detallado de los resultados obtenidos.

1. Experiencia previa de los docentes

Ambos docentes entrevistados cuentan con una amplia experiencia en la enseñanza de temas relacionados con la arquitectura de computadoras, destacando un promedio de más de 10 años en el área. En cuanto al uso de herramientas de simulación, ambos participantes indicaron haber utilizado previamente simuladores como Emu8086, Simple 8-bit Assembler Simulator y VonSim, aunque señalaron limitaciones en las herramientas existentes, como interfaces poco intuitivas y falta de soporte para funcionalidades avanzadas.

2. Valoración del material didáctico

Los docentes evaluaron positivamente el material didáctico asociado al simulador VonSim8, destacando los siguientes aspectos:

En conjunto, los docentes destacaron la calidad del material didáctico, especialmente los recursos visuales y los ejemplos prácticos, como elementos clave para la enseñanza.

3. Valoración del simulador VonSim8

Tabla 6.3: Evaluación de aspectos clave por los docentes

Aspecto evaluado	Promedio	Interpretación
Claridad de las explicaciones teóricas	4.2	Las explicaciones sobre el ciclo de instrucción fueron consideradas claras y completas.
Utilidad de los recursos visuales	4.5	Las tablas, esquemas e imágenes fueron altamente valoradas como apoyo al aprendizaje.
Ejemplos prácticos (MOV, ADD, SUB, JMP)	4.3	Los ejemplos facilitaron la conexión entre teoría y práctica.
Actividades propuestas	4.1	Las actividades fueron percibidas como útiles para consolidar los conceptos clave.

Tabla 6.4: Evaluación del simulador educativo

Aspecto evaluado	Promedio	Interpretación
Visualización de registros y memoria	4.0	Buena valoración general, aunque se sugieren mejoras en la claridad de las animaciones.
Conexión entre teoría y práctica	4.4	El simulador fue considerado eficaz para vincular los conceptos teóricos con su aplicación práctica.
Facilidad de uso e intuición	3.9	Aunque accesible, algunos docentes sugirieron optimizar la interfaz para estudiantes principiantes.
Representación gráfica del flujo de datos	4.1	La representación gráfica fue valorada como clara y didáctica, aunque con margen de mejora en detalles técnicos.

El simulador VonSim8 fue valorado de manera positiva por los docentes, quienes resaltaron su utilidad pedagógica y su capacidad para facilitar la comprensión de conceptos complejos. Los resultados fueron los siguientes:

Los docentes coincidieron en que el simulador cumple adecuadamente su función pedagógica, aunque identificaron oportunidades de mejora en la interfaz y en la representación de microoperaciones.

4. Análisis cualitativo de las respuestas abiertas

Aspectos más útiles o interesantes

Los docentes destacaron como elementos más valiosos del simulador:

- La visualización gráfica del ciclo de instrucción y el flujo de datos entre componentes.

- La posibilidad de activar progresivamente instrucciones del repertorio x86, evitando la sobrecarga cognitiva.
- La conexión entre teoría y práctica, facilitada por las simulaciones interactivas.

Aspectos más difíciles o confusos

Entre las principales dificultades mencionadas se encuentran:

- La representación de las banderas (flags) y su activación durante la ejecución.
- La claridad de las animaciones en la lectura y escritura de memoria.
- La necesidad de mejorar la accesibilidad de la interfaz para estudiantes con poca experiencia.

Estos comentarios sugieren que, aunque el simulador es efectivo, ciertos detalles técnicos y de diseño requieren ajustes para optimizar su uso en el aula.

5. Impacto percibido en la enseñanza

Los docentes coincidieron en que el simulador VonSim8 tiene un impacto positivo en la enseñanza, destacando su capacidad para:

- Mejorar la comprensión de conceptos abstractos como el ciclo de instrucción y las microoperaciones.
- Incrementar la motivación de los estudiantes al ofrecer una experiencia práctica e interactiva.
- Facilitar la enseñanza de temas avanzados como interrupciones y periféricos.

6. Conclusiones generales

Los resultados de las entrevistas permiten concluir que el simulador VonSim8 es una herramienta didáctica eficaz, valorada positivamente por los docentes por su capacidad para vincular teoría y práctica, y por su enfoque progresivo en la enseñanza de la arquitectura x86. Sin embargo, se identificaron áreas de mejora relacionadas con la interfaz, la representación de banderas y la claridad de las animaciones.

En síntesis, el simulador VonSim8 y el material didáctico asociado contribuyen de manera significativa al desarrollo de competencias conceptuales y procedimentales en el estudio de la arquitectura de computadoras, consolidándose como un recurso valioso para la enseñanza en este campo.

6.4 Anexo D: Resultados de simulación xdevs

Este anexo presenta la implementación del modelo DEVS del simulador VonSim8 construida con la biblioteca xdevs.py (Python) y resume los resultados de su ejecución para la instrucción MOV AL, BL. El objetivo es aportar evidencia empírica que respalde la validación del modelo teórico descrito en el Capítulo 5, sección “Modelado con xDEVS del ciclo de instrucción MOV AL, BL” 5.8.

6.4.1 Interpretación de métricas

DEVS ofrece trazabilidad temporal de grano fino mediante microtransiciones ($\lambda, \delta_{int}, \delta_{ext}$). En esta traza, un único “paso” pedagógico del ciclo RTL (captación/ejecución) puede descomponerse en varias microtransiciones y, por tanto, en múltiples eventos del simulador. Por diseño, la interfaz de VonSim8 contabiliza los ciclos según esos pasos pedagógicos del ciclo de instrucción y con ellos calcula los acumulados y el CPI; en paralelo, este anexo reporta tiempo real y número de eventos de la simulación DEVS, magnitudes distintas que no deben confundirse con los ciclos pedagógicos.

6.4.2 Supuestos de la simulación

- Datos y registros de 8 bits (saturación/mascarado a 0xFF).
- Memoria unificada con latencia de lectura fija $\tau_{mem} = 1$ ciclo; no se modelan escrituras en este escenario.
- Bus interno del banco de registros con propagación y estabilización total 2 ciclos en la transferencia fuente→bus→destino; arbitraje sin contención (acceso exclusivo).
- Repertorio reducido: se implementa y ejecuta únicamente MOV AL, BL; no se actualizan banderas (flags) ni estado de la ALU.
- Memoria precargada con opcode 0x01 en dirección 0x00; IP inicial en 0x00.
- Valores iniciales: AL=0x01, BL=0x0A (CL=0x00, DL=0x00), MBR=IR=0x00.
- Planificador DEVS xdevs.py con avance de tiempo discreto; resolución temporal en ciclos abstractos (no tiempo físico).
- Métricas: CPI como suma de ciclos de FETCH y EXECUTE en la UC; “Eventos DEVS” como conteo de llamadas a $\delta_{ext}/\delta_{int}$ a través del coordinador instrumentado.

6.4.3 Configuración y procedimiento de ejecución

- Entorno recomendado: Linux x86_64, Python ≥ 3.10 , xdevs.py instalado vía pip.

- Semilla/aleatoriedad: no aplica (modelo determinista).
- Parámetros de corrida: número de iteraciones del coordinador fijado en `num_iters=30`, suficiente para completar la instrucción y emitir métricas.

6.4.4 Repositorio y trazabilidad

El código fuente de la simulación está disponible en: xdevs-vonsim8¹ [92].

```
from xdevs.models import Atomic, Coupled, Port
from xdevs.sim import Coordinator

# =====
# MODELOS ATÓMICOS: Registros y componentes individuales
# =====

class SharedBus(Atomic):
    """
    Modelo atómico para el bus compartido.
    Garantiza acceso exclusivo mediante señales de control y
    modela el retardo tau_bus.
    """

    def __init__(self, name: str = "BUS"):
        super().__init__(name)
        self.current_value: int = 0x00
        self.locked: bool = False
        self.requester: str = ""

        # Puertos de arbitraje
        self.req = Port(str, name="req")
        # Identificador del componente solicitante
        self.add_in_port(self.req)
        self.grant = Port(bool, name="grant")
        self.add_out_port(self.grant)

        # Puertos de datos
        self.data_in = Port(int, name="data_in")
        self.add_in_port(self.data_in)
        self.data_out = Port(int, name="data_out")
        self.add_out_port(self.data_out)
```

¹xdevs-vonsim8: <https://github.com/ruiz-jose/xdevs-vonsim8>

```
# Puerto de release
self.release = Port(bool, name="release")
self.add_in_port(self.release)

self.pending_grant = False
self.pending_data = False

def initialize(self):
    self.passivate()

def deltext(self, e: float):
    self.continuef(e)

# Solicitud de acceso al bus
if self.req and not self.req.empty():
    if not self.locked:
        self.requester = self.req.get()
        self.locked = True
        self.pending_grant = True
        self.hold_in("GRANTING", 1) # tau_bus = 1 ciclo

# Liberación del bus
if self.release and not self.release.empty():
    if self.release.get():
        self.locked = False
        self.requester = ""

# Recepción de datos
if self.data_in and not self.data_in.empty():
    self.current_value = self.data_in.get() & 0xFF
    self.pending_data = True
    self.activate()

def deltint(self):
    if self.pending_grant:
        self.pending_grant = False
    if self.pending_data:
        self.pending_data = False
    self.passivate()

def lambdaf(self):
    if self.pending_grant:
        self.grant.add(True)
    if self.pending_data:
```

```
        self.data_out.add(self.current_value)

    def exit(self):
        pass


class Register(Atomic):
    """Modelo atómico genérico para un registro de 8 bits con
    señales indexadas."""

    def __init__(self, name: str, initial_value: int = 0x00):
        super().__init__(name)
        self.value: int = initial_value
        self.reg_name: str = name
        # Nombre del registro (AL, BL, CL, DL)

        # Puertos de entrada/salida
        self.data_in = Port(int, name="data_in")
        self.add_in_port(self.data_in)
        self.enable_in = Port(str, name="enable_in")
        # Recibe nombre de registro
        self.add_in_port(self.enable_in)
        self.enable_out = Port(str, name="enable_out")
        # Recibe nombre de registro
        self.add_in_port(self.enable_out)

        self.data_out = Port(int, name="data_out")
        self.add_out_port(self.data_out)

        self.pending_write = False
        self.pending_read = False
        self.pending_value = 0x00

    def initialize(self):
        self.passivate()

    def deltext(self, e: float):
        self.continuef(e)

        # Capturar dato entrante (puede llegar antes del enable_in)
        if self.data_in and not self.data_in.empty():
            self.pending_value = self.data_in.get() & 0xFF

        # Procesar señales de habilitación indexadas
```

```
if self.enable_in and not self.enable_in.empty():
    target_reg = self.enable_in.get()
    # Solo actuar si la señal es para este registro
    if target_reg == self.reg_name:
        self.pending_write = True
        self.activate()

if self.enable_out and not self.enable_out.empty():
    target_reg = self.enable_out.get()
    # Solo actuar si la señal es para este registro
    if target_reg == self.reg_name:
        self.pending_read = True
        self.activate()

def deltint(self):
    if self.pending_write:
        self.value = self.pending_value
        print(f" [{self.name}] delta_int: Escritura completada.
              Valor={self.value:02X}")
        self.pending_write = False

    if self.pending_read:
        print(f" [{self.name}] delta_int: Lectura completada.
              Valor={self.value:02X}")
        self.pending_read = False

    self.passivate()

def lambdaf(self):
    if self.pending_read:
        print(f" [{self.name}] lambda: Emitiendo dato
              ={self.value:02X} por data_out")
        self.data_out.add(self.value)

def exit(self):
    pass

class SimpleRegister(Atomic):
    """Modelo atómico para registros simples (MBR, IR) con señales
    booleanas."""

    def __init__(self, name: str, initial_value: int = 0x00):
        super().__init__(name)
```

```
self.value: int = initial_value

# Puertos de entrada/salida
self.data_in = Port(int, name="data_in")
self.add_in_port(self.data_in)
self.enable_in = Port(bool, name="enable_in")
self.add_in_port(self.enable_in)
self.enable_out = Port(bool, name="enable_out")
self.add_in_port(self.enable_out)

self.data_out = Port(int, name="data_out")
self.add_out_port(self.data_out)

self.pending_write = False
self.pending_read = False
self.pending_value = 0x00

def initialize(self):
    self.passivate()

def deltext(self, e: float):
    self.continuef(e)

# Procesar señales de habilitación y datos
if self.enable_in and not self.enable_in.empty():
    enabled = self.enable_in.get()
    if enabled and self.data_in and not self.data_in.empty():
        self.pending_value = self.data_in.get() & 0xFF
        self.pending_write = True
        self.activate()

    if self.enable_out and not self.enable_out.empty():
        enabled = self.enable_out.get()
        if enabled:
            self.pending_read = True
            self.activate()

# Escritura directa sin señal enable
if not self.enable_in and self.data_in and
not self.data_in.empty():
    self.pending_value = self.data_in.get() & 0xFF
    self.pending_write = True
    self.activate()
```

```
def deltint(self):
    if self.pending_write:
        self.value = self.pending_value
        print(f" [{self.name}] delta_int: Escritura completada.
              Valor={self.value:02X}")
        self.pending_write = False

    if self.pending_read:
        print(f" [{self.name}] delta_int: Lectura completada.
              Valor={self.value:02X}")
        self.pending_read = False

    self.passivate()

def lambdaf(self):
    if self.pending_read:
        print(f" [{self.name}] lambda: Emitiendo
              dato={self.value:02X} por data_out")
        self.data_out.add(self.value)

def exit(self):
    pass

class InstructionPointer(Atomic):
    """Modelo atómico para el registro IP (Instruction Pointer)."""

    def __init__(self, name: str = "IP"):
        super().__init__(name)
        self.value: int = 0x00

        self.addr_out = Port(int, name="addr_out")
        self.add_out_port(self.addr_out)
        self.ip_write = Port(int, name="ip_write")
        self.add_in_port(self.ip_write)
        self.read_request = Port(bool, name="read_request")
        self.add_in_port(self.read_request)

        self.pending_output = False
        self.pending_increment = False

    def initialize(self):
        self.passivate()
```

```
def deltext(self, e: float):
    self.continuef(e)

    if self.read_request and self.read_request.get():
        self.pending_output = True
        self.activate()

    if self.ip_write:
        increment = self.ip_write.get()
        if increment:
            self.pending_increment = True
            self.activate()

def deltint(self):
    if self.pending_increment:
        old_val = self.value
        self.value = (self.value + 1) & 0xFF
        print(f" [IP] delta_int: Incremento IP.
{old_val:02X} → {self.value:02X}")
        self.pending_increment = False

    if self.pending_output:
        self.pending_output = False

    self.passivate()

def lambdaf(self):
    if self.pending_output:
        print(f" [IP] lambda: Emitiendo dirección IP=
{self.value:02X} por addr_out")
        self.addr_out.add(self.value)

def exit(self):
    pass

class MemoryAddressRegister(Atomic):
    """Modelo atómico para el registro MAR
(Memory Address Register)."""

    def __init__(self, name: str = "MAR"):
        super().__init__(name)
        self.address: int = 0x00
```

```
        self.addr_in = Port(int, name="addr_in")
        self.add_in_port(self.addr_in)
        self.addr_out = Port(int, name="addr_out")
        self.add_out_port(self.addr_out)

        self.pending_addr = None

    def initialize(self):
        self.passivate()

    def deltext(self, e: float):
        self.continuef(e)
        if self.addr_in:
            self.pending_addr = self.addr_in.get() & 0xFF
            self.activate()

    def deltint(self):
        if self.pending_addr is not None:
            self.address = self.pending_addr
            print(f" [MAR] delta_int: Dirección almacenada.
                MAR={self.address:02X}")
            self.pending_addr = None
        self.passivate()

    def lambdaaf(self):
        if self.pending_addr is not None:
            print(f" [MAR] lambda: Emitiendo dirección
                MAR={self.pending_addr:02X} hacia MEM")
            self.addr_out.add(self.pending_addr)

    def exit(self):
        pass

class Memory(Atomic):
    """Modelo atómico para la memoria unificada."""

    def __init__(self, name: str = "MEM"):
        super().__init__(name)
        self.storage: dict[int, int] = {}
        self.pending_addr: int | None = None
        self.pending_read: bool = False

        self.addr = Port(int, name="addr")
```

```
    self.add_in_port(self.addr)
    self.rw = Port(bool, name="rw") # True=read, False=write
    self.add_in_port(self.rw)
    self.data_in = Port(int, name="data_in")
    self.add_in_port(self.data_in)

    self.data_out = Port(int, name="data_out")
    self.add_out_port(self.data_out)

    self.pending_operation = None

def initialize(self):
    # Cargar programa: MOV AL, BL (opcode 0x01)
    self.storage[0x00] = 0x01
    self.passivate()

def delttext(self, e: float):
    self.continuef(e)

    # Capturar dirección
    if self.addr and not self.addr.empty():
        self.pending_addr = self.addr.get() & 0xFF

    # Capturar señal de lectura
    if self.rw and not self.rw.empty():
        self.pending_read = self.rw.get()

    # Si tenemos ambos, iniciar operación
    if self.pending_addr is not None and self.pending_read:
        self.pending_operation = ("read", self.pending_addr)
        print(f" [MEM] delta_ext: Solicitud de lectura en
              dirección {self.pending_addr:02X}")
        self.hold_in("READING", 1) # tau_mem = 1
        self.pending_addr = None
        self.pending_read = False

def deltint(self):
    if self.pending_operation:
        op_type, addr_val = self.pending_operation
        data = self.storage.get(addr_val, 0x00)
        print(f" [MEM] delta_int: Lectura completada.
              MEM[{addr_val:02X}]={data:02X}")
        self.pending_operation = None
    self.passivate()
```

```
def lambdaaf(self):
    if self.pending_operation and self.pending_operation[0]
    == "read":
        addr_val = self.pending_operation[1]
        data = self.storage.get(addr_val, 0x00)
        print(f" [MEM] lambda: Emitiendo dato={data:02X}
por data_out")
        self.data_out.add(data)

def exit(self):
    pass

class ControlUnit(Atomic):
    """Modelo atómico para la Unidad de Control (UC)."""

    def __init__(self, name: str = "UC"):
        super().__init__(name)

        # Entradas
        self.ir_in = Port(int, name="ir_in")
        self.add_in_port(self.ir_in)

        # Salidas de control - Fase FETCH
        self.ip_read = Port(bool, name="ip_read")
        self.add_out_port(self.ip_read)
        self.ip_inc = Port(bool, name="ip_inc")
        self.add_out_port(self.ip_inc)
        self.mem_read = Port(bool, name="mem_read")
        self.add_out_port(self.mem_read)
        self.mbr_enable = Port(bool, name="mbr_enable")
        self.add_out_port(self.mbr_enable)
        self.ir_enable = Port(bool, name="ir_enable")
        self.add_out_port(self.ir_enable)
        self.ir_read = Port(bool, name="ir_read")
        self.add_out_port(self.ir_read)

        # Salidas de control - Fase EXECUTE
        # (señales indexadas para REG_BANK)
        self.reg_enable_out = Port(str, name="reg_enable_out")
        # Nombre del registro: "AL", "BL", etc.
        self.add_out_port(self.reg_enable_out)
        self.reg_enable_in = Port(str, name="reg_enable_in")
        self.add_out_port(self.reg_enable_in)
```

```
# Estado interno
self.phase = "IDLE"
self.instruction_code = 0x00
self.micro_step = 0

# Contadores para métricas
self.total_cycles = 0
self.fetch_cycles = 0
self.execute_cycles = 0

# Tabla de decodificación
self.instruction_set = {
    0x01: {"opcode": "MOV", "dst": "AL", "src": "BL"}
}

def initialize(self):
    self.phase = "FETCH1"
    self.micro_step = 0
    # Contar el ciclo inicial
    self.total_cycles = 1
    self.fetch_cycles = 1
    self.hold_in(self.phase, 1)

def deltint(self):
    # Máquina de estados de la UC
    if self.phase == "FETCH1": # UC → IP
        print(f"\n{'-'*78}")
        print(f"  FASE FETCH - Paso 1/6: UC → IP")
        (solicita dirección)")
        print(f"{'-'*78}")
        self.phase = "FETCH2"
        cycles = 1
        # No sumar aquí porque ya se contó en initialize()
        self.hold_in(self.phase, cycles)

    elif self.phase == "FETCH2": # IP → MAR
        print(f"\n{'-'*78}")
        print(f"  FASE FETCH - Paso 2/6: IP → MAR")
        (transfiere dirección)")
        print(f"{'-'*78}")
        self.phase = "FETCH3"
        cycles = 1
        self.fetch_cycles += cycles
        self.total_cycles += cycles
```

```
        self.hold_in(self.phase, cycles)

    elif self.phase == "FETCH3": # UC → MEM; IP++
        print(f"\n{'-'*78}")
        print(f"  FASE FETCH - Paso 3/6: UC → MEM (mem_read);"
              f" IP ← IP+1")
        print(f"{'-'*78}")
        self.phase = "FETCH4"
        cycles = 2
        self.fetch_cycles += cycles
        self.total_cycles += cycles
        self.hold_in(self.phase, cycles)

    elif self.phase == "FETCH4": # MEM → MBR
        print(f"\n{'-'*78}")
        print(f"  FASE FETCH - Paso 4/6: MEM → MBR"
              f" (instrucción leída)")
        print(f"{'-'*78}")
        self.phase = "FETCH5"
        cycles = 1
        self.fetch_cycles += cycles
        self.total_cycles += cycles
        self.hold_in(self.phase, cycles)

    elif self.phase == "FETCH5": # MBR → IR
        print(f"\n{'-'*78}")
        print(f"  FASE FETCH - Paso 5/6: MBR → IR"
              f" (carga instrucción)")
        print(f"{'-'*78}")
        self.phase = "FETCH6"
        cycles = 2
        self.fetch_cycles += cycles
        self.total_cycles += cycles
        self.hold_in(self.phase, cycles)

    elif self.phase == "FETCH6": # IR → UC
        print(f"\n{'-'*78}")
        print(f"  FASE FETCH - Paso 6/6: IR → UC (recibe opcode)")
        print(f"{'-'*78}")
        # Transición directa a EXEC1 sin ciclo intermedio
        self.phase = "EXEC1"
        cycles = 1
        self.fetch_cycles += cycles
        self.total_cycles += cycles
```

```
        self.hold_in(self.phase, cycles)

    elif self.phase == "EXEC1": # Decodificación
        print(f"\n{'-'*78}")
        print(f"  FASE EXECUTE - Paso 1/5: Decodificación")
        print(f"{'-'*78}")
        decoded = self.instruction_set.get(self.instruction_code,
                                             {"opcode": "NOP", "dst": "", "src": ""})
        print(f"  [UC] Instrucción decodificada:
              {decoded['opcode']} {decoded['dst']},{decoded['src']}")
        print(f"  [UC] Microoperaciones planificadas: BL →
              BUS → AL")
        self.phase = "EXEC2"
        cycles = 1
        self.execute_cycles += cycles
        self.total_cycles += cycles
        self.hold_in(self.phase, cycles)

    elif self.phase == "EXEC2": # AL ← BUS
        print(f"\n{'-'*78}")
        print(f"  FASE EXECUTE - Paso 2/5: UC →
              REG_BANK.enable_out(BL)")
        print(f"{'-'*78}")
        self.phase = "EXEC3"
        cycles = 1
        self.execute_cycles += cycles
        self.total_cycles += cycles
        self.hold_in(self.phase, cycles)

    elif self.phase == "EXEC3":
        print(f"\n{'-'*78}")
        print(f"  FASE EXECUTE - Paso 3/5: BUS ← BL
              (dato disponible)")
        print(f"{'-'*78}")
        self.phase = "EXEC4"
        cycles = 2
        # 2 ciclos: 1 para propagar en bus, 1 para estabilizar
        self.execute_cycles += cycles
        self.total_cycles += cycles
        self.hold_in(self.phase, cycles)

    elif self.phase == "EXEC4":
        print(f"\n{'-'*78}")
        print(f"  FASE EXECUTE - Paso 4/5: UC →
```

```
REG_BANK.enable_in(AL)")
print(f"\n{'-'*78}")
self.phase = "EXEC5"
cycles = 1
self.execute_cycles += cycles
self.total_cycles += cycles
self.hold_in(self.phase, cycles)

elif self.phase == "EXEC5":
    print(f"\n{'-'*78}")
    print(f"  FASE EXECUTE - Paso 5/5: AL ← BUS
        (captura completada)")
    print(f"\n{'-'*78}")
    # Incrementar contador del último ciclo
    cycles = 1
    self.execute_cycles += cycles
    self.total_cycles += cycles
    self.phase = "DONE"
    print(f"\n{'-'*78}")
    print(f"  Ciclo de instrucción MOV AL, BL completado")
    print(f"\n{'-'*78}")
    # Usar hold_in para contar el último ciclo,
    # luego passivate
    self.hold_in(self.phase, cycles)

elif self.phase == "DONE":
    # Finalizar después del último ciclo
    # (ya contado en EXEC5)
    self.passivate()

else:
    self.passivate()

def delttext(self, e: float):
    self.continuef(e)
    if self.ir_in:
        self.instruction_code = self.ir_in.get() & 0xFF

def lambdaf(self):
    # Generar señales de control según la fase
    if self.phase == "FETCH1":
        self.ip_read.add(True)

    elif self.phase == "FETCH3":
```

```
        self.mem_read.add(True)
        self.ip_inc.add(True)

    elif self.phase == "FETCH4":
        self.mbr_enable.add(True)

    elif self.phase == "FETCH5":
        self.ir_enable.add(True)

    elif self.phase == "FETCH6":
        self.ir_read.add(True)

    elif self.phase == "EXEC2":
        self.reg_enable_out.add("BL")

    elif self.phase == "EXEC4":
        self.reg_enable_in.add("AL")

def exit(self):
    pass

# =====
# MODELO ACOPLADO: Banco de Registros (REG_BANK)
# =====

class RegisterBank(Coupled):
    """
    Modelo acoplado para el banco de registros AL, BL, CL, DL.
    Actúa como mediador entre la UC y los registros individuales,
    gestionando las señales de habilitación indexadas y el bus interno.
    """

    def __init__(self, name: str = "REG_BANK"):
        super().__init__(name)

        # Crear registros atómicos individuales
        self.al = Register("AL", 0x01)
        self.bl = Register("BL", 0x0A)
        self.cl = Register("CL", 0x00)
        self.dl = Register("DL", 0x00)

        # Agregar como componentes
        self.add_component(self.al)
```

```
        self.add_component(self.bl)
        self.add_component(self.cl)
        self.add_component(self.dl)

        # Puertos de control desde UC (señales indexadas)
        self.reg_enable_in = Port(str, name="reg_enable_in")
        # Recibe nombre de registro destino
        self.add_in_port(self.reg_enable_in)
        self.reg_enable_out = Port(str, name="reg_enable_out")
        # Recibe nombre de registro fuente
        self.add_in_port(self.reg_enable_out)

        # Acoplamientos para enable_in desde UC hacia registros
        self.add_coupling(self.reg_enable_in, self.al.enable_in)
        self.add_coupling(self.reg_enable_in, self.bl.enable_in)
        self.add_coupling(self.reg_enable_in, self.cl.enable_in)
        self.add_coupling(self.reg_enable_in, self.dl.enable_in)

        # Acoplamientos para enable_out desde UC hacia registros
        self.add_coupling(self.reg_enable_out, self.al.enable_out)
        self.add_coupling(self.reg_enable_out, self.bl.enable_out)
        self.add_coupling(self.reg_enable_out, self.cl.enable_out)
        self.add_coupling(self.reg_enable_out, self.dl.enable_out)

        # Bus interno: todos los registros pueden
        # enviar/recibir datos
        # Cada registro fuente puede enviar al data_in de cualquier
        # registro destino
        for src_reg in [self.al, self.bl, self.cl, self.dl]:
            for dst_reg in [self.al, self.bl, self.cl, self.dl]:
                if src_reg != dst_reg:
                    self.add_coupling(src_reg.data_out,
                                      dst_reg.data_in)

# =====
# MODELO ACOPLADO: Sistema completo VonSim8
# =====

class VonSim8System(Coupled):
    """Modelo acoplado que integra todos los componentes del
    simulador VonSim8."""

    def __init__(self, name: str = "VonSim8"):
```

```
super().__init__(name)

# Crear componentes atómicos
self.ip = InstructionPointer("IP")
self.mar = MemoryAddressRegister("MAR")
self.mem = Memory("MEM")
self.mbr = SimpleRegister("MBR", 0x00) # Usa SimpleRegister
self.ir = SimpleRegister("IR", 0x00) # Usa SimpleRegister
self.uc = ControlUnit("UC")

# Crear modelo acoplado: Banco de Registros
# (incluye bus interno)
self.reg_bank = RegisterBank("REG_BANK")

# Contador de eventos DEVS
self.devs_events = 0

# Agregar componentes al modelo acoplado
self.add_component(self.ip)
self.add_component(self.mar)
self.add_component(self.mem)
self.add_component(self.mbr)
self.add_component(self.ir)
self.add_component(self.uc)
self.add_component(self.reg_bank)

# =====
# ACOPLAMIENTOS FASE FETCH
# =====
# UC → IP
self.add_coupling(self.uc.ip_read, self.ip.read_request)
# IP → MAR
self.add_coupling(self.ip.addr_out, self.mar.addr_in)
# MAR → MEM
self.add_coupling(self.mar.addr_out, self.mem.addr)
# UC → MEM (read signal)
self.add_coupling(self.uc.mem_read, self.mem.rw)
# UC → IP (increment)
self.add_coupling(self.uc.ip_inc, self.ip.ip_write)
# MEM → MBR
self.add_coupling(self.mem.data_out, self.mbr.data_in)
# UC → MBR (enable write from MEM)
self.add_coupling(self.uc.mbr_enable, self.mbr.enable_in)
# MBR → IR
```

```
        self.add_coupling(self.mbr.data_out, self.ir.data_in)
        # UC → MBR (enable out to IR)
        self.add_coupling(self.uc.ir_enable, self.mbr.enable_out)
        # UC → IR (enable write from MBR)
        self.add_coupling(self.uc.ir_enable, self.ir.enable_in)
        # UC → IR (enable read to send to UC) - en FETCH6
        self.add_coupling(self.uc.ir_read, self.ir.enable_out)
        # IR data → UC
        self.add_coupling(self.ir.data_out, self.uc.ir_in)

        # =====
        # ACOPLAMIENTOS FASE EXECUTE
        # (con REG_BANK que contiene bus interno)
        # =====
        # UC → REG_BANK (señales indexadas)
        self.add_coupling(self.uc.reg_enable_out,
                          self.reg_bank.reg_enable_out)
        self.add_coupling(self.uc.reg_enable_in,
                          self.reg_bank.reg_enable_in)

class CPUSystem(Coupled):
    """Wrapper para compatibilidad con el main anterior."""
    def __init__(self, name: str = "CPUEnvironment"):
        super().__init__(name)
        self.vonsim8 = VonSim8System("VonSim8")
        self.add_component(self.vonsim8)

if __name__ == "__main__":
    import time

    # Clase para contar eventos DEVS
    class EventCountingCoordinator(Coordinator):
        def __init__(self, model):
            super().__init__(model)
            self.event_count = 0

        def _inject_event_counter(self, model):
            """Inyecta contador en deltint y deltext de todos
            los modelos atómicos."""
            if hasattr(model, 'deltint'):
                original_deltint = model.deltint
                def counted_deltint():
                    self.event_count += 1
                    return original_deltint()
                model.deltint = counted_deltint
```

```
        self.event_count += 1
        return original_deltint()
    model.deltint = counted_deltint

    if hasattr(model, 'delttext'):
        original_delttext = model.delttext
        def counted_delttext(e):
            self.event_count += 1
            return original_delttext(e)
        model.delttext = counted_delttext

    # Recursivamente para modelos acoplados
    if hasattr(model, 'components'):
        for comp in model.components:
            self._inject_event_counter(comp)

    def initialize(self):
        """Override para injectar contadores antes de
        inicializar."""
        self._inject_event_counter(self.model)
        super().initialize()

# =====
# ENCABEZADO
# =====
print("\n" + "-" * 80)
print(" " * 15 + "SIMULACIÓN DEVS - VONSIM8 (Von Neumann 8-bit)"
+ " " * 20 )
print("-" * 80)
print(" Instrucción: MOV AL, BL (opcode 0x01)")
print(" Formalismo: DEVS (Discrete Event System Specification)")
print("-" * 80 + "\n")

# =====
# EJECUCIÓN
# =====
print("Ejecutando simulación... \n")

env = CPUSystem("VonSim8Environment")
coord = EventCountingCoordinator(env)
coord.initialize()

start_time = time.time()
coord.simulate(num_iters=30)
```

```
simulation_time = time.time() - start_time
coord.exit()

vonsim8 = env.vonsim8

# =====
# RESULTADOS
# =====
print("-" * 60)
print("  RESULTADOS DE LA SIMULACIÓN")
print("-" * 60)

# Estado de registros (compacto)
print(f"\n  Registros:")
print(f"    IP: 0x00 → 0x{vonsim8.ip.value:02X} ")
print(f"    AL: 0x01 → 0x{vonsim8.reg_bank.al.value:02X}")
{'Transferido' if vonsim8.reg_bank.al.value == 0xA
else 'Error'}")
print(f"    BL: 0x0A → 0x{vonsim8.reg_bank.bl.value:02X} ")

# Verificación compacta
print()
if vonsim8.reg_bank.al.value == 0xA:
    print("  MOV AL,BL ejecutado correctamente")
else:
    print("  ERROR en la transferencia")

# Métricas compactas (valores calculados dinámicamente)
print(f"\n  Métricas:")
print(f"  • CPI: {vonsim8.uc.total_cycles} ciclos (FETCH: {vonsim8.uc.fetch_cycles} + EXECUTE: {vonsim8.uc.execute_cycles})")
print(f"  • Tiempo real: {simulation_time*1000:.2f} ms")
print(f"  • Eventos: {coord.event_count} transiciones DEVS")
print()

# =====
# PIE DE PÁGINA
# =====
print("-" * 80)
print("Simulación completada | Arquitectura Von Neumann validada")
print("-" * 80 + "\n")
```

Un ejemplo de resultado de la simulación es el siguiente:

SIMULACIÓN DEVS - VONSIM8 (Von Neumann 8-bit)

Instrucción: MOV AL, BL (opcode 0x01) Formalismo: DEVS (Discrete Event System Specification)

Ejecutando simulación...

FASE FETCH - Paso 1/6: UC → IP (solicita dirección)

- [IP] λ : Emitiendo dirección IP=00 por addr_out
- [MAR] λ : Emitiendo dirección MAR=00 hacia MEM
- [MAR] δ_{int} : Dirección almacenada. MAR=00

FASE FETCH - Paso 2/6: IP → MAR (transfiere dirección)

- [MEM] δ_{ext} : Solicitud de lectura en dirección 00

FASE FETCH - Paso 3/6: UC → MEM (mem_read); IP \leftarrow IP+1

- [IP] δ_{int} : Incremento IP. 00 \rightarrow 01
- [MEM] λ : Emitiendo dato=01 por data_out
- [MEM] δ_{int} : Lectura completada. MEM[00]=01
- [MBR] δ_{int} : Escritura completada. Valor=01

FASE FETCH - Paso 4/6: MEM → MBR (instrucción leída)**FASE FETCH - Paso 5/6: MBR → IR (carga instrucción)**

- [MBR] λ : Emitiendo dato=01 por data_out
- [MBR] δ_{int} : Lectura completada. Valor=01
- [IR] δ_{int} : Escritura completada. Valor=01

FASE FETCH - Paso 6/6: IR → UC (recibe opcode)

- [IR] λ : Emitiendo dato=01 por data_out
- [IR] δ_{int} : Lectura completada. Valor=01

FASE EXECUTE - Paso 1/5: Decodificación

- [UC] Instrucción decodificada: MOV AL,BL
- [UC] Microoperaciones planificadas: BL → BUS → AL

FASE EXECUTE - Paso 2/5: UC → REG_BANK.enable_out(BL)

- [BL] λ : Emitiendo dato=0A por data_out
- [BL] δ_{int} : Lectura completada. Valor=0A

FASE EXECUTE - Paso 3/5: BUS ← BL (dato disponible)

FASE EXECUTE - Paso 4/5: UC → REG_BANK.enable_in(AL)

- [AL] δ_{int} : Escritura completada. Valor=0A

FASE EXECUTE - Paso 5/5: AL ← BUS (captura completada)

Ciclo de instrucción MOV AL, BL completado

RESULTADOS DE LA SIMULACIÓN

Registros:

- IP: 0x00 → 0x01
- AL: 0x01 → 0x0A Transferido
- BL: 0x0A → 0x0A

MOV AL,BL ejecutado correctamente

Métricas:

- * CPI: 14 ciclos (FETCH: 8 + EXECUTE: 6)
- * Tiempo real: 2.76 ms
- * Eventos: 45 transiciones DEVS

Simulación completada Arquitectura Von Neumann validada

El resultado de la simulación confirma que el modelo DEVS implementado en xdevs.py reproduce correctamente el ciclo de instrucción MOV AL, BL, validando así el enfoque teórico presentado en el Capítulo 5. Los registros involucrados (IP, AL, BL) muestran los valores esperados antes y después de la ejecución, y las métricas de rendimiento (CPI, tiempo real, eventos) proporcionan información adicional sobre la eficiencia del modelo.

Capítulo 7

Trabajos futuros

El desarrollo del simulador VonSim8 ha permitido establecer una base sólida para la enseñanza de los principios fundamentales de la arquitectura de computadoras. Sin embargo, existen múltiples oportunidades para expandir y mejorar sus capacidades, alineándose con las necesidades de formación avanzada y los avances tecnológicos en el ámbito de la simulación. A continuación, se describen las principales líneas de trabajo propuestas para el futuro:

1. Compatibilidad con el lenguaje NASM

Una de las extensiones más relevantes consiste en habilitar el soporte para el lenguaje ensamblador NASM (Netwide Assembler). Este lenguaje es ampliamente utilizado en entornos profesionales y educativos, especialmente en la programación de bajo nivel para arquitecturas x86 y x86-64. La integración de NASM en el simulador permitiría a los estudiantes trabajar con un lenguaje estándar de la industria, facilitando la transición hacia entornos reales de desarrollo. Además, se planea incorporar un ensamblador interno que traduzca el código NASM a las instrucciones del simulador, asegurando una experiencia fluida y consistente.

2. Ampliación a 16 registros de 64 bits

Actualmente, el simulador opera con registros de 8 bits, lo que resulta adecuado para la enseñanza introductoria. Sin embargo, para abordar conceptos avanzados, se propone implementar un conjunto de 16 registros de 64 bits, similar al modelo utilizado en arquitecturas modernas como x86-64. Esta ampliación permitiría explorar temas como la manipulación de datos de mayor tamaño, la optimización de operaciones aritméticas y lógicas, y el uso eficiente de registros en aplicaciones complejas.

3. Implementación de interrupciones de Linux x86

La incorporación de un sistema de interrupciones basado en las llamadas al sistema de Linux (syscalls) para arquitecturas x86 representa una oportunidad

para conectar el simulador con entornos operativos reales. Esto permitiría a los estudiantes comprender cómo las aplicaciones interactúan con el sistema operativo a través de interrupciones, abordando conceptos como la gestión de archivos, la entrada/salida estándar y la asignación de memoria. Además, esta funcionalidad facilitaría la simulación de programas que dependen de servicios del sistema operativo, enriqueciendo la experiencia de aprendizaje.

4. Módulo de entrada/salida con DMA

Se propone desarrollar un módulo de entrada/salida basado en el uso de acceso directo a memoria (DMA, por sus siglas en inglés). Este módulo permitiría simular la transferencia de datos entre dispositivos periféricos y la memoria principal sin la intervención directa de la CPU, replicando un mecanismo clave en arquitecturas modernas. La implementación de DMA ofrecería a los estudiantes una comprensión más profunda de los procesos de entrada/salida eficientes y su impacto en el rendimiento del sistema.

5. Simulación de una pantalla gráfica de 16x16 píxeles

Finalmente, se plantea la incorporación de un módulo gráfico que permita simular una pantalla de 16x16 píxeles. Este componente ofrecería una representación visual básica para explorar conceptos como la manipulación de gráficos, la representación de datos en memoria y la interacción entre la CPU y dispositivos de salida. Además, este módulo podría integrarse con el sistema de interrupciones y el módulo DMA, proporcionando un entorno más completo y realista para la simulación.

6. Extensión del modelado DEVS y simulación avanzada

A partir de la validación obtenida mediante el modelado DEVS del ciclo de instrucción MOV AL, BL, una línea de trabajo futuro relevante consiste en ampliar la instrumentación y el análisis de la simulación. Esto incluye la incorporación de nuevas instrucciones del repertorio x86, la modelización de componentes adicionales como la ALU, unidades de pipeline o memoria cache. Asimismo, se propone profundizar en la recolección de métricas cuantitativas (CPI, latencia, ocupación del bus) y en el desarrollo de herramientas de visualización que permitan analizar la traza temporal de eventos DEVS. Estas extensiones no solo fortalecerán la validez del simulador VonSim8 como herramienta de investigación y docencia, sino que también abrirán el camino para explorar fenómenos avanzados de arquitectura de computadoras bajo un enfoque formal y reproducible.

7.0.1 Conclusión

Estas líneas de trabajo futuro no solo expanden las capacidades técnicas del simulador, sino que también fortalecen su valor pedagógico al abordar temas avanzados de arquitectura de computadoras y programación de bajo nivel. La

implementación de estas mejoras permitirá que el simulador evolucione hacia una herramienta más versátil y alineada con las necesidades de formación en entornos académicos y profesionales. Asimismo, estas extensiones abren la posibilidad de colaborar con la comunidad educativa y de desarrollo, promoviendo la adopción y mejora continua del simulador.

Capítulo 8

Conclusiones finales

Esta tesis desarrolló y validó VonSim8, un simulador educativo orientado a introducir conceptos de arquitectura de computadoras con foco en x86 desde una arquitectura de 8 bits trazable y progresiva. La evidencia reunida indica que la simulación, cuando está diseñada con criterios pedagógicos (simplificación deliberada, activación gradual del repertorio y visualización RTL del ciclo de instrucción), constituye un recurso pertinente para articular teoría y práctica.

8.1 Principales aportes técnicos y didácticos

- Progresión controlada de complejidad: repertorio inicial reducido y activación gradual, alineado con la currícula, para disminuir carga cognitiva extrínseca.
- Visualización RTL paso a paso: trazabilidad de datos y señales de control en fetch/execute, favoreciendo la formación de modelos mentales del ciclo de instrucción.
- Interrupciones y E/S básicas: periféricos introductorios (teclado/pantalla) y vector de interrupciones acotado que permiten abordar asincronía de manera accesible.
- Métricas de ejecución: CPI, ciclos por fase y tiempo de CPU para promover análisis cuantitativo del desempeño.
- Validación formal con xDEVS: modelado DEVS/PDEVS de componentes y del ciclo MOV AL, BL en xdevs.py, con medición de transiciones y coherencia temporal.
- Apertura y sostenibilidad: distribución libre que habilita adopción, auditoría y mejoras por la comunidad académica.

8.2 Hallazgos pedagógicos (alcance y límites)

- Encuesta a estudiantes ($N=14$) e entrevistas a docentes ($N=2$) sugieren mejoras percibidas en claridad del ciclo de instrucción, flujo de datos y comprensión de microoperaciones.
- Las valoraciones destacan recursos visuales y conexión teoría-práctica; se observan oportunidades de mejora en la representación de banderas, microoperaciones y claridad de animaciones.
- Dado el muestreo no probabilístico y el tamaño muestral, los resultados son indicativos; requieren replicación con diseños cuasi-experimentales y muestras mayores.

8.3 Limitaciones y amenazas a la validez

- Validez externa: contexto único y muestra acotada dificultan generalización.
- Fidelidad del modelo: la arquitectura de 8 bits prioriza claridad sobre exhaustividad; ciertos fenómenos de x86 real quedan fuera de alcance.

8.4 Síntesis

El prototipo VonSim8 cumplió los objetivos formativos propuestos: visualización del ciclo de instrucción a nivel RTL, repertorio progresivo, métricas básicas e interrupciones introductorias. La validación con xDEVS aportó rigor estructural y temporal al modelo y sentó bases para instrumentación y reproducibilidad. Los resultados educativos son prometedores pero preliminares; su consolidación exige estudios con mayor potencia estadística y refinamientos de interfaz y animaciones.

8.5 Líneas futuras

Las extensiones técnicas y pedagógicas delineadas (por ejemplo, compatibilidad NASM, registros de 64 bits, syscalls Linux, DMA y pantalla 16×16 , así como ampliación del modelado DEVS) se detallan en el capítulo Trabajos futuros (véase [7](#)). Su abordaje permitirá mantener el enfoque pedagógico mientras se incrementa la fidelidad y el alcance del simulador.

Capítulo 9

Bibliografía

- [1] M. A. Colombani, M. A. Falappa, A. G. Delduca, and J. M. Ruiz, “PID novel 7065: Enseñanza/aprendizaje de asignatura Arquitectura de Computadoras con herramientas de simulación de sistemas de cómputos.” Feb. 2022. Accessed: Jul. 10, 2024. [Online]. Available: https://proyectos.uner.edu.ar/aplicacion.php?ah=st668e6d47663eb&ai=gestion_extinv%7C%7C23000105
- [2] J. Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol, *Discrete-event system simulation*, 5th ed. Prentice Hall, 2010.
- [3] A. M. Law, *Simulation Modeling & Analysis*, 5th ed. New York, NY, USA: McGraw-Hill, 2015.
- [4] S. Robinson, *Simulation: The Practice of Model Development and Use*, 2nd edition. Wiley, 2014.
- [5] C. Lion, “Los simuladores. Su potencial para la enseñanza universitaria,” *Cuadernos de Investigación Educativa*, vol. 2, no. 12, pp. 53–66, 2005.
- [6] G. Contreras, R. G. Torres, and M. S. R. Montoya, “Uso de simuladores como recurso digital para la transferencia de conocimiento,” *Apertura: Revista de Innovación Educativa*, vol. 2, no. 1, pp. 86–100, 2010.
- [7] A. Garcia-Garcia, J. C. Saez, J. L. Risco-Martin, and M. Prieto-Matias, “PBBCache: An open-source parallel simulator for rapid prototyping and evaluation of cache-partitioning and cache-clustering policies,” *Journal of Computational Science*, vol. 42, p. 101102, 2020.
- [8] M. D. Grossi, E. M. Jiménez Rey, A. C. Servetto, and G. Perichinsky, “Un simulador de una maquina computadora como herramienta para la enseñanza de la arquitectura de computadoras,” in *I jornadas de educación en informática y TICs en argentina*, 2005.
- [9] E. Herruzo, J. Benavides, E. Saez, M. Montijano, and J. Paloamres, “Desarrollo de simuladores de arquitectura de computadores y su aplicación en la enseñanza,” in *Congreso de tecnologías aplicadas a la enseñanza de la electrónica (TAEE'2002)*, 2002.
- [10] R. de Diego Martinez, “MSX88: Una herramienta para la enseñanza de la estructura y funcionamiento de los ordenadores,” *Actas del Congreso URSI*, 1994.

- [11] R. Concheiro, M. Loureiro, M. Amor, and P. González, “Simula3MS: Simulador pedagógico de un procesador,” 2005.
- [12] B. Nova, J. C. Ferreira, and A. Araújo, “Tool to support computer architecture teaching and learning,” in *Engineering Education (CISPEE), 2013 1st International Conference of the Portuguese Society for*, IEEE, 2013, pp. 1–8.
- [13] B. Mustafa, “Evaluating A System Simulator For Computer Architecture Teaching And Learning Support,” *Innovation in Teaching and Learning in Information and Computer Sciences*, vol. 9, no. 1, pp. 100–104, 2010, doi: [10.11120/ital.2010.09010100](https://doi.org/10.11120/ital.2010.09010100).
- [14] F. García-Carballera, A. Calderón-Mateos, S. Alonso-Monsalve, and J. Prieto-Cepeda, “WepSIM: An online interactive educational simulator integrating microdesign, microprogramming, and assembly language programming,” *IEEE Transactions on Learning Technologies*, vol. 13, no. 1, pp. 211–218, 2020.
- [15] P. Prasad, A. Alsadoon, A. Beg, and A. Chan, “Using simulators for teaching computer organization and architecture,” *Computer applications in engineering education*, vol. 24, no. 2, pp. 215–224, 2016.
- [16] Z. Radivojevic, M. Cvetanovic, and J. Dordevic, “Design of the simulator for teaching computer architecture and organization,” in *2011 Second Eastern European Regional Conference on the Engineering of Computer Based Systems*, IEEE, 2011, pp. 124–130.
- [17] B. Nikolic, Z. Radivojevic, J. Djordjevic, and V. Milutinovic, “A Survey and Evaluation of Simulators Suitable for Teaching Courses in Computer Architecture and Organization,” *IEEE Transactions on Education*, vol. 52, no. 4, pp. 449–458, Nov. 2009, doi: [10.1109/TE.2008.930097](https://doi.org/10.1109/TE.2008.930097).
- [18] A. Akram and L. Sawalha, “A survey of computer architecture simulation techniques and tools,” *IEEE Access*, vol. 7, pp. 78120–78145, 2019, doi: [10.1109/ACCESS.2019.2917698](https://doi.org/10.1109/ACCESS.2019.2917698).
- [19] J. L. Hennessy and D. A. Patterson, *Computer architecture: A quantitative approach*, 6th ed. Boston: Morgan Kaufmann, 2017.
- [20] W. Stallings, *Computer organization and architecture: Designing for performance*, 11th ed. Boston, MA: Pearson, 2021.
- [21] J. Sweller, P. Ayres, and S. Kalyuga, *Cognitive load theory*. New York: Springer, 2010. doi: [10.1007/978-1-4419-8126-3](https://doi.org/10.1007/978-1-4419-8126-3).
- [22] Intel Corporation, *Intel® 64 and IA-32 architectures software developer's manual*. 2025. Available: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>
- [23] AMD, “Developer guides, manuals & ISA documents.” Sep. 2024. Accessed: May 11, 2025. [Online]. Available: <https://www.amd.com/en/search/documentation/hub.html>
- [24] P. Abel, *IBM PC Assembly Language and Programming*, 5th ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.
- [25] D. Skrien, “CPU Sim 3.1: A tool for simulating computer architectures for computer organization classes,” *Journal on Educational Resources in Computing (JERIC)*, 2001.

- [26] R. E. Mayer, *Multimedia learning*, 2nd ed. Cambridge University Press, 2009.
- [27] S. A. Ambrose *et al.*, *How learning works: Seven research-based principles for smart teaching*. Jossey-Bass, 2010.
- [28] J. L. Peterson, *Petri net theory and the modeling of systems*. Prentice Hall PTR, 1981.
- [29] B. Zeigler, H. Prähofer, and T. G. Kim, “Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems,” vol. 2, Jan. 2000.
- [30] B. P. Zeigler, A. Muzy, and E. Kofman, *Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations*. Academic Press, 2018.
- [31] A. S. Tanenbaum, *Structured computer organization*. Pearson Education India, 2016.
- [32] M. J. Murdocca and V. Heuring, *Principles of computer architecture*. Pearson Education, 2000.
- [33] R. E. Bryant and D. R. O'Hallaron, *Computer systems: A programmer's perspective*. Pearson, 2015.
- [34] A. Waterman and K. Asanović, *The RISC-V instruction set manual, volume i: User-level ISA, version 2.0*. University of California, Berkeley, 2014. Available: <https://riscv.org/technical/specifications>
- [35] S. Harris and D. Harris, *Digital design and computer architecture*. Morgan Kaufmann, 2015.
- [36] L. Null, *Essentials of computer organization and architecture with navigate advantage access*, 6th ed. Burlington, MA: Jones & Bartlett Learning, 2023.
- [37] Patterson *et al.*, *Computer organization and design: The hardware/software interface-5th*. Morgan Kaufmann, 2014.
- [38] D. A. Patterson and J. L. Hennessy, *Computer organization and design ARM edition: The hardware software interface*. Morgan kaufmann, 2016.
- [39] L. Belli *et al.*, “IoT-enabled smart sustainable cities: Challenges and approaches,” *Smart Cities*, vol. 3, no. 3, pp. 1039–1071, 2020.
- [40] D. A. Patterson and J. L. Hennessy, *Computer organization and design RISC-v edition: The hardware software interface*. Morgan Kaufmann, 2017.
- [41] A. Akram and L. Sawalha, “A survey of computer architecture simulation techniques and tools,” *Ieee Access*, vol. 7, pp. 78120–78145, 2019.
- [42] M. Menchón, M. Tosini, and O. Goñi, “Herramientas de software educacional para el aprendizaje de arquitectura de procesadores.”
- [43] J. von Neumann, “First draft of a report on the EDVAC,” Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia, PA, Technical Report, 1945.
- [44] P. E. Ceruzzi, *A history of modern computing*. MIT press, 2003.
- [45] M. R. Williams, *A history of computing technology*. Prentice-Hall Englewood Cliffs, NJ, 1998.
- [46] T. Noergaard, *Embedded systems architecture: A comprehensive guide for engineers and programmers*. Newnes, 2012.

- [47] Arm Ltd., *Arm architecture reference manual: Armv9-a, for Armv9-a architecture profile*. Arm Ltd., 2021.
- [48] Intel Corporation, *Intel® 64 and IA-32 architectures optimization reference manual*, April 2021. Intel Corporation, 2021.
- [49] J. L. Hennessy and D. A. Patterson, *Computer organization and design RISC-v edition: The hardware software interface*. Elsevier Science & Technology Books, 2017.
- [50] ARM Holdings, “The relentless evolution of the arm architecture,” ARM Holdings, 2025. Available: <https://newsroom.arm.com/blog/evolution-of-arm-architecture-evolution-40-years>
- [51] I. Corporation, “Intel xeon processor scalable family: Performance and efficiency for modern data centers,” Intel, 2023. Available: <https://www.intel.com/content/www/us/en/products/docs/processors/xeon-accelerated/4th-gen-xeon-scalable-processors.html>
- [52] D. A. Patterson and J. L. Hennessy, *Computer organization and design ARM edition: The hardware software interface*. Morgan kaufmann, 2016.
- [53] B. B. Brey, *The intel microprocessors: Architecture, programming, and interfacing*, 8th ed. Pearson Education, 2013.
- [54] Intel Corporation, *Intel 8086 family user’s manual*. Intel Corporation, 1979. Accessed: May 17, 2025. [Online]. Available: https://bitsavers.org/components/intel/8086/9800722-03_The_8086_Family_Users_Manual_Oct79.pdf
- [55] K. R. Irvine and L. B. Das, *Assembly language for x86 processors*. Prentice Hall, 2011.
- [56] B. International, *Turbo assembler user’s guide*. Borland International, 1993.
- [57] M. Corporation, *Microsoft macro assembler 6.1 reference*. Microsoft Press, 1992.
- [58] The NASM Project, *The netwide assembler (NASM) manual*. 2023.
- [59] R. Hyde, *The art of assembly language*. No Starch Press, 2010.
- [60] A. Stork, C.-A. Thole, S. Klimenko, I. Nikitin, L. Nikitina, and Y. Astakhov, “Towards interactive simulation in automotive design,” *The Visual Computer*, vol. 24, pp. 947–953, 2008.
- [61] F. Jentsch and M. Curtis, *Simulation in aviation training*. Routledge, 2017.
- [62] J. L. Risco-Martín, S. Mittal, K. Henares, R. Cardenas, and P. Arroba, “xDEVS: A toolkit for interoperable modeling and simulation of formal discrete event systems,” *Software: Practice and Experience*, vol. 53, no. 3, pp. 748–789, 2023, doi: <https://doi.org/10.1002/spe.3168>.
- [63] Iscar-UCM, “xDEVS: Toolkit for interoperable modeling and simulation of formal discrete event systems.” <https://github.com/iscar-ucm/xdevs>, 2023.
- [64] Iscar-UCM, “Xdevs.py: Python implementation of xDEVS for discrete event simulation.” <https://github.com/iscar-ucm/xdevs.py/tree/main>, 2023.
- [65] R. M. Fujimoto, “Parallel and distributed simulation systems,” in *Proceeding of the 2001 winter simulation conference (cat. No. 01CH37304)*, IEEE, 2001, pp. 147–157.
- [66] F. J. Barros, “Modeling formalisms for dynamic structure systems,” *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 7, no. 4,

- pp. 501–515, 1997.
- [67] B. P. Zeigler, R. Jammalamadaka, and S. R. Akerkar, “Continuity and change (activity) are fundamentally related in DEVS simulation of continuous systems,” in *International conference on AI, simulation, and planning in high autonomy systems*, Springer, 2004, pp. 1–13.
- [68] F. A. Calvo Valdés, J. F. Roldan Ramírez, and A. San Miguel Sanchez, “Simulador del procesador MIPS sobre el formalismo DEVS,” 2010.
- [69] F. A. Calvo Valdés, J. F. Roldán Ramírez, and A. San Miguel Sánchez, “Simulador del procesador MIPS sobre el formalismo DEVS,” *Revista de Simulación*, 2010, Accessed: Sep. 19, 2024. [Online]. Available: <https://hdl.handle.net/20.500.14352/46063>
- [70] G. Llorente de la Cita and L. Lázaro-Carrasco Hernández, “Diseño e implementación del kernel de xDEVS, versión distribuida,” Universidad Complutense de Madrid, Facultad de Informática, Trabajo de Fin de Grado, 2016.
- [71] R. E. Bryant, “Formal verification of pipelined Y86-64 microprocessors with UCLID5,” School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, CMU-CS-18-122, 2018.
- [72] M. A. Colombani, J. M. Ruiz, A. G. Delduca, and M. A. Falappa, “Herramientas de software para dar soporte en la enseñanza y aprendizaje de la arquitectura x86,” 2022. Accessed: Jul. 10, 2024. [Online]. Available: <http://sedici.unlp.edu.ar/handle/10915/139908>
- [73] P. BEHROOZ, *Computer Architecture: From Microprocessors to Supercomputers*. Oxford University Press Inc, 2005.
- [74] A. Huberman *et al.*, “Qualitative data analysis a methods sourcebook,” 2019.
- [75] W3C Web Accessibility Initiative, “Accessibility principles.” 2021. Available: <https://www.w3.org/WAI/fundamentals/accessibility-principles/>
- [76] J. Sorva, “Notional machines and introductory programming education,” *ACM Transactions on Computing Education (TOCE)*, vol. 13, no. 2, pp. 1–31, 2013, doi: [10.1145/2483710.2483713](https://doi.org/10.1145/2483710.2483713).
- [77] M. McCracken *et al.*, “A multi-national, multi-institutional study of assessment of programming skills of first-year CS students,” *SIGCSE Bulletin*, vol. 33, no. 4, pp. 125–140, 2001, doi: [10.1145/572139.572181](https://doi.org/10.1145/572139.572181).
- [78] National Academies of Sciences, Engineering, and Medicine, *How people learn II: Learners, contexts, and cultures*. Washington, DC: National Academies Press, 2018. doi: [10.17226/24783](https://doi.org/10.17226/24783).
- [79] T. Newhall, K. C. Webb, I. Romea, E. Stavis, and S. J. Matthews, “ASM visualizer: A learning tool for assembly programming,” in *Proceedings of the 56th ACM technical symposium on computer science education v. 1*, in SIGCSETS 2025. New York, NY, USA: Association for Computing Machinery, 2025, pp. 840–846. doi: [10.1145/3641554.3701793](https://doi.org/10.1145/3641554.3701793).
- [80] C. C. Bonwell and J. A. Eison, “Active learning: Creating excitement in the classroom,” *ASHE-ERIC Higher Education Report*, vol. 1, 1991.
- [81] Y. N. Patt and S. J. Patel, *Introduction to computing systems: From bits and gates to c and beyond*, 3rd ed. New York: McGraw-Hill Education, 2019.

- [82] Z. Majid, “Design of SAP-1 controller and simulation using mentor graphics,” PhD thesis, Universiti Teknologi MARA (UiTM), 1999.
- [83] A. Morlan, “Building a simple 8-bit CPU in verilog (SAP-1).” 2021. Available: https://austinmorlan.com/posts/fpga_computer_sap1/
- [84] A. Gualdrón Gamarra and J. P. Pinilla, “Plataforma para la emulación y reconfiguración de arquitecturas CISC y RISC.” 2015-01-27. Available: <http://hdl.handle.net/20.500.11912/2004>
- [85] D. Silber, “TinyCPU - a simple CPU implemented in verilog.” <https://www.eecis.udel.edu/~silber/tinycpu/#/home>.
- [86] J. M. S. Facundo Quiroga Manuel Bustos Berrondo, “VonSim - simulador de arquitectura de von neumann,” <https://vonsim.github.io/>, commit 9ff63978, 2020.
- [87] J. Ruiz, “VonSim8 - simulador simplificado de arquitectura von neumann de 8 bits,” commit 60a0da4, Oct. 2025. Accessed: Nov. 09, 2025. [Online]. Available: <https://ruiz-jose.github.io/VonSim8/>
- [88] Open Source Initiative, “Open source licenses: Understanding GPL, MIT, and Apache licenses.” 2024. Accessed: Jan. 15, 2024. [Online]. Available: <https://opensource.org/licenses/>
- [89] M. M. Mano and M. D. Ciletti, *Digital design: With an introduction to the verilog HDL, VHDL, and SystemVerilog*. Pearson, 2017.
- [90] Facultad de Ciencias de la Administración, UNER, “Programa de la asignatura Arquitectura de Computadoras.” Consejo Directivo, Facultad de Ciencias de la Administración, UNER; <https://digesto.uner.edu.ar/documento.frame.php?cod=170316>, Mar. 2025.
- [91] A. P. Godse and D. A. Godse, *Microprocessor and interfacing*. Technical Publications, 2020.
- [92] J. Ruiz, “Xdevs-vonsim8: Modelo DEVS del simulador VonSim8,” commit a51b371, Oct. 2025. Accessed: Nov. 09, 2025. [Online]. Available: <https://github.com/ruiz-jose/xdevs-vonsim8>