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

Maestría en Sistemas de Información

Ruiz Jose Maria

Director: Colombani Marcelo Alberto

2025

Índice

Resumen **1**

Agradecimientos **2**

1 Introducción **3**
1.1 Justificación 6
1.2 Objetivos 7
1.3 Metodología de desarrollo 8
1.4 Organización del documento 9

2 Arquitectura de computadoras **10**
2.1 Introducción a la arquitectura de computadoras 10
2.2 Arquitecturas Von Neumann y Harvard 12
2.3 Tipos de arquitecturas 16
2.4 Repertorio de instrucciones 18
2.5 Filosofías CISC y RISC 23
2.6 Arquitectura x86 28
2.7 Lenguaje máquina y lenguaje ensamblador 32

3 Simulación **36**
3.1 Introducción a la simulación 36
3.2 Simulación en la educación 37
3.3 El Formalismo DEVS (Discrete Event System Specification) 39

4 Comparativa de simuladores **42**
4.1 Estudios similares 42
4.2 Simuladores bajo análisis 43
4.3 Criterios de evaluación 43
4.4 Selección de simuladores 45
4.5 Participantes en la evaluación 46
4.6 Análisis comparativo 46
4.7 Resultados 48

5 Diseño y Construcción del Simulador **51**
5.1 Requisitos de la herramienta y su justificación 51
5.2 Diseño del Simulador 59

5.3 Ciclo de la instrucción 82
5.4 Modulo de Entrada/Salida 88
5.5 Pila 90

Apéndices **100**
5.6 Anexo A: Protocolo de Entrevista Semiestructurada 100

6 Bibliografía **103**

Índice de tablas

1.1 Funcionalidades principales del simulador propuesto 7

2.1 Cuadro comparativo entre arquitecturas Von Neumann y Harvard . . 15

2.2 Aplicaciones de la simulación en distintos sectores 18

2.3 Modos de direccionamiento básicos 22

2.4 Comparativa de repertorios de instrucciones reales 23

2.5 Comparativa entre CISC y RISC 27

2.6 Hitos en la evolución x86 30

2.7 Línea de Tiempo de la Evolución de la Arquitectura x86 30

2.8 Comparación de ensambladores arquitectura x86 35

3.1 Aplicaciones de la simulación en distintos sectores 37

4.1 Criterios e indicadores de evaluación de simuladores 45

4.2 Proceso de selección de simuladores 46

4.3 Comparativa según criterios de evaluación preestablecidos 48

5.1 Activación progresiva del repertorio de instrucciones 53

5.2 Resumen de requisitos funcionales y su fundamentación pedagógica . 57

5.3 Comparativa de características entre VonSim y VonSim8 69

5.4 Estructura VonSim8: componentes principales y características 70

5.5 Registro FLAGS: descripción de las banderas 72

5.6 Bloques funcionales principales 74

5.7 Tabla de instrucciones de VonSim8 76

5.8 Saltos condicionales: instrucciones y condiciones 76

5.9 Tabla de modos de direccionamiento 80

5.10 Categorías de instrucciones y códigos de operación en VonSim8 82

5.11 Tabla de codificación de instrucciones 83

5.12 Tabla de registros del simulador 83

5.13 Tabla de Instrucciones y Códigos de Operación de la Arquitectura x86 88

5.15 Categorías de instrucciones y códigos de operación en VonSim8 95

5.16 Tabla de instrucciones y acciones 98

5.17 Tabla de codificación de instrucciones ampliado 99

Índice de figuras

2.1 Arquitectura Von Neumann 13

2.2 Arquitectura Harvard 14

2.3 Arquitectura Híbridas 15

2.4 Características repertorio de instrucciones 19

2.5 Modos de direccionamiento 21

2.6 Formato de instrucciones 23

2.7 Convergencia de filosofías 28

2.8 Diagrama esquemático microprocesador Intel 8086 29

2.9 Formato de instrucciones del Pentium x86 31

2.10 Proceso de ensamblado 34

3.1 Relación entre modelos atómicos y modelos acoplados en DEVS . . . 39

3.2 Modelo acoplado en DEVS 40

5.1 Estructura del simulador y componentes funcionales 52

5.2 Editor ensamblador 53

5.3 Ciclo de instrucción: captación y ejecución 54

5.4 Módulo genérico de entrada/salida programada 55

5.5 Métricas de rendimiento 56

5.6 Documentación on line 57

5.7 Registro FLAGS 63

5.8 Registro de estado I 63
5.9 Controles del simulador 64
5.10 Registro de 8 bits 65
5.11 Eliminación registro temporales left y rigth 65
5.12 Memoria principal 66
5.13 Resaltado registro IP y registro SP 66
5.14 Resaltado vector de interrupciones 67
5.15 Visor de instrucciones y datos del programa en memoria 67
5.16 Resaltado vector de interrupciones y registro SP 68
5.17 Memoria principal 72
5.18 Buses 73
5.19 Teclado y pantalla 74
5.20 Arquitectura general del simulador 75
5.21 Flujo del ciclo de instrucción en VonSim8 84

Resumen

Existe un consenso creciente en el uso de herramientas de simulación en la enseñanza para procesos dinámicos complejos, como las operaciones intrínsecas de la computadora, que permiten representar de forma visual e interactiva la organización y arquitectura interna de la computadora, facilitando así la comprensión de su funcionamiento por parte de los alumnos y el desarrollo de los temas por parte del docente. En este contexto, los simuladores juegan una pieza clave en el campo de la Arquitectura de Computadoras, permitiendo conectar fundamentos teóricos con la experiencia práctica, simplificando abstracciones y haciendo más rica la labor docente. La arquitectura x86 es ampliamente utilizada en computadoras de escritorio y servidores. Este documento pretende realizar una comparación de los simuladores x86 que más se adecuan en el dictado de la asignatura Arquitectura de Computadoras de la carrera Licenciatura en Sistemas, establecer los criterios de evaluación y evaluar los simuladores seleccionados de acuerdo con estos criterios.

La presente investigación surge en el marco del proyecto de investigación I/D novel PID-UNER 7065: "Enseñanza/aprendizaje de asignatura Arquitectura de Computadoras con herramientas de simulación de sistemas de cómputos". El Proyecto es llevado a cabo en la Facultad de Ciencias de la Administración de la Universidad Nacional de Entre Ríos, se vincula directamente con la asignatura Arquitectura en Computadoras que se dicta en segundo año de la carrera Licenciatura en Sistemas perteneciente a la Facultad de Ciencias de la Administración de la Universidad Nacional de Entre Ríos.

Palabras clave: x86, simulador, aprendizaje, enseñanza, arquitectura de computadoras.

Agradecimientos

Agradecimientos aquí.

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 permite afianzar los conocimientos de los temas vistos 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, simplificando abstracciones y facilitando la labor docente [17], [18], [19], [20].

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) [21], [22].

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 [21], [22], [23].

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 [24].

1.1 Justificación

Los estudiantes y docentes de la asignatura de Arquitectura de Computadoras enfrentan múltiples desafíos 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.

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.

Entre sus funcionalidades clave, permitirá: - Visualizar en detalle cada una de las etapas del ciclo de ejecución de instrucciones (fetch y 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.

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

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

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 [25], [26], [27].

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], [28], [29], [30].

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], [28], [29], [30].

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 [31], [32], [33]. 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], [30], [34].

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 [32], [35], [36].

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 [32], [35], [36].
- **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 [31], [37].

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 [38], [39]. 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” [40], 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].

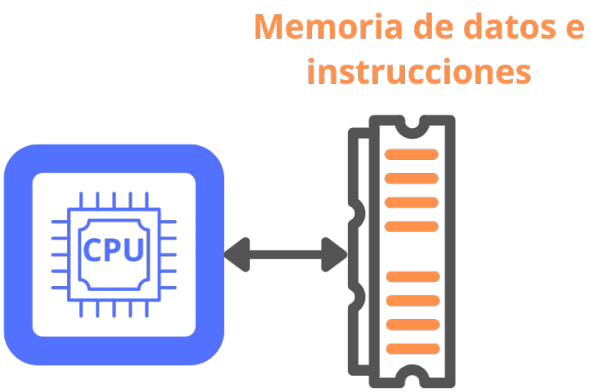

Figura 2.1: Arquitectura Von Neumann

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 [41], [42]. 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 [28]. Debido a su eficiencia, esta arquitectura se ha adoptado ampliamente en sistemas embebidos, microcontroladores y procesadores de señal digital (DSP) [43].

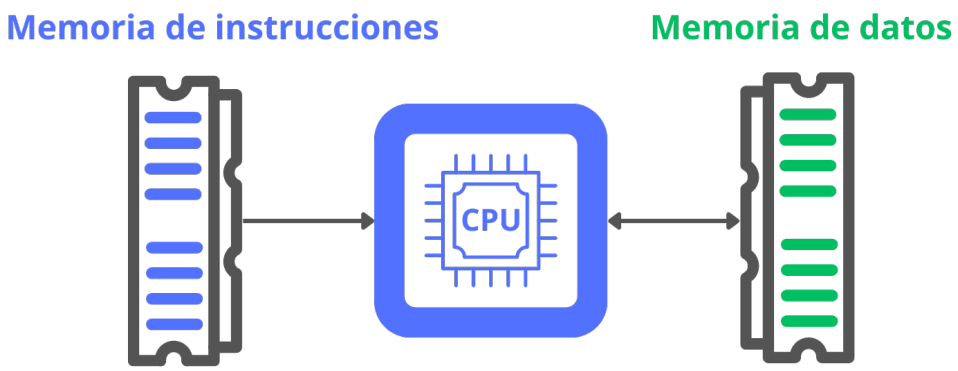

Figura 2.2: Arquitectura Harvard

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], [33], [37].

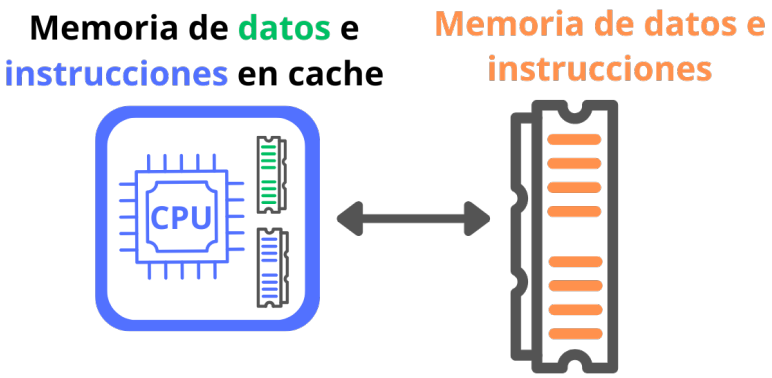

Figura 2.3: Arquitectura Híbridas

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 [44], [45].

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 [31], [33], [46], [47]. 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], [48].

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 [34], [47].

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.[31], [32], [46], [49].

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 [34], [47].

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], [33].

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], [33].
- **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.

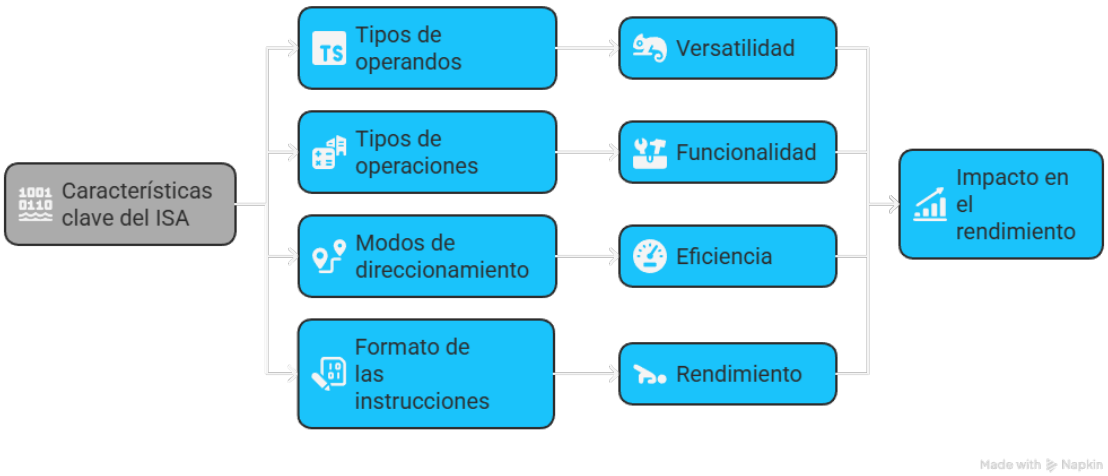

Figura 2.4: Características repertorio de instrucciones

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].

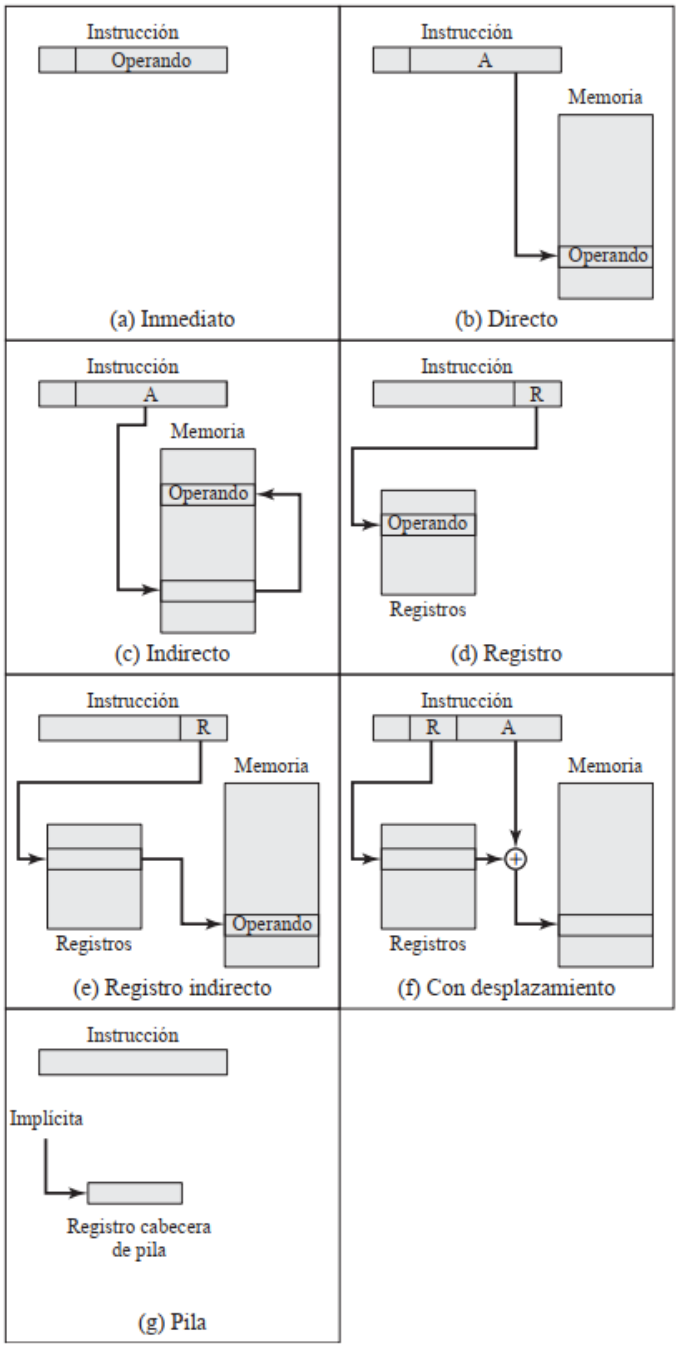

Figura 2.5: Modos de direccionamiento

- 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 ← A	No referencia a memoria	Operando de magnitud limitada
Directo	EA ← A	Es sencillo	Espacio de direcciones limitado
Indirecto	EA ← (A)	Espacio de direcciones grande	Referencias múltiples a memoria
Registro	EA ← R	No referencia a memoria	Número limitado de registros
Indirecto con registro	EA ← (R)	Espacio de direcciones grande	Referencia extra a memoria
Con desplazamiento	EA ← A + (R)	Flexibilidad	Complejidad
Pila	EA ← 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], [28]:

- **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.

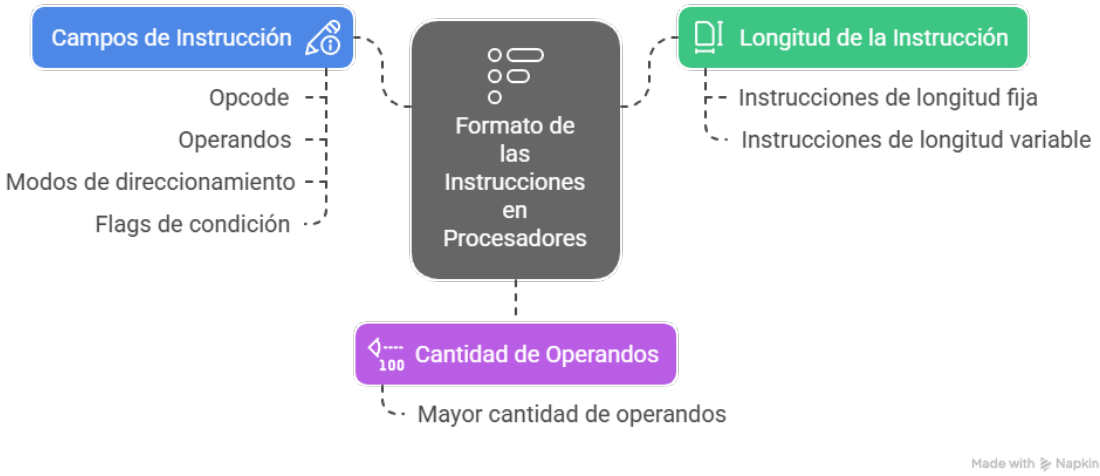

Figura 2.6: Formato de instrucciones

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], [34].

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], [34].

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 [34].

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 [46].

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 [28].

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 filosofia 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`.:

```
    # Cargar el valor de mem1 en el registro t0 (R1)
    lw t0, 0(mem1)      # t0 = MEM[mem1]

    # Cargar el valor de mem2 en el registro t1 (R2)
    lw t1, 0(mem2)      # t1 = MEM[mem2]

    # Sumar los registros t0 y t1, guardar el resultado en t2 (R3)
    add t2, t0, t1      # t2 = t0 + t1

    # Guardar el resultado en mem1
    sw t2, 0(mem1)      # MEM[mem1] = t2

```

Una arquitectura bajo la filosofia 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`:

```
  ; Cargar el valor almacenado en mem1 en el registro EAX
  MOV EAX, [mem1]

  ; Sumar el valor almacenado en mem2 al registro EAX
  ADD EAX, [mem2]

  ; Guardar el resultado de la suma de vuelta en mem1
  MOV [mem1], EAX

```

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 muestras la convergencia entre estas dos filosofías:

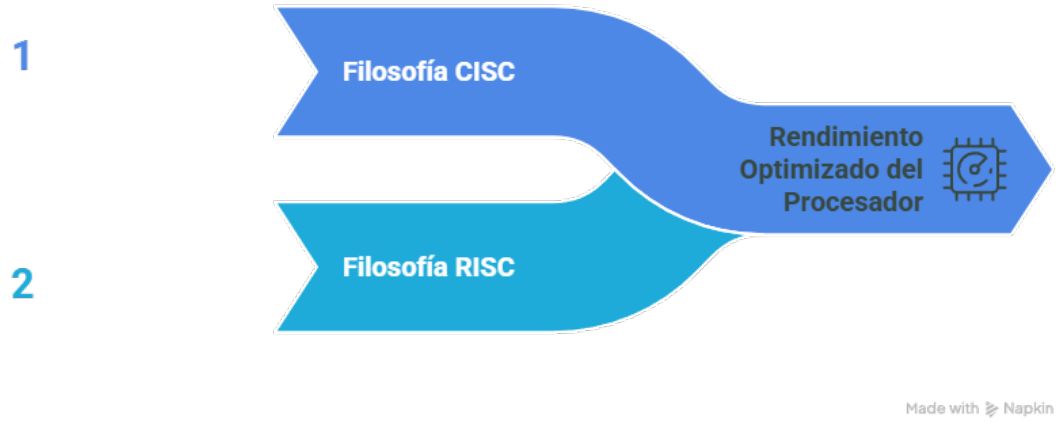

Figura 2.7: Convergencia de filosofías

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], [21], [22], [23], [50], [51].

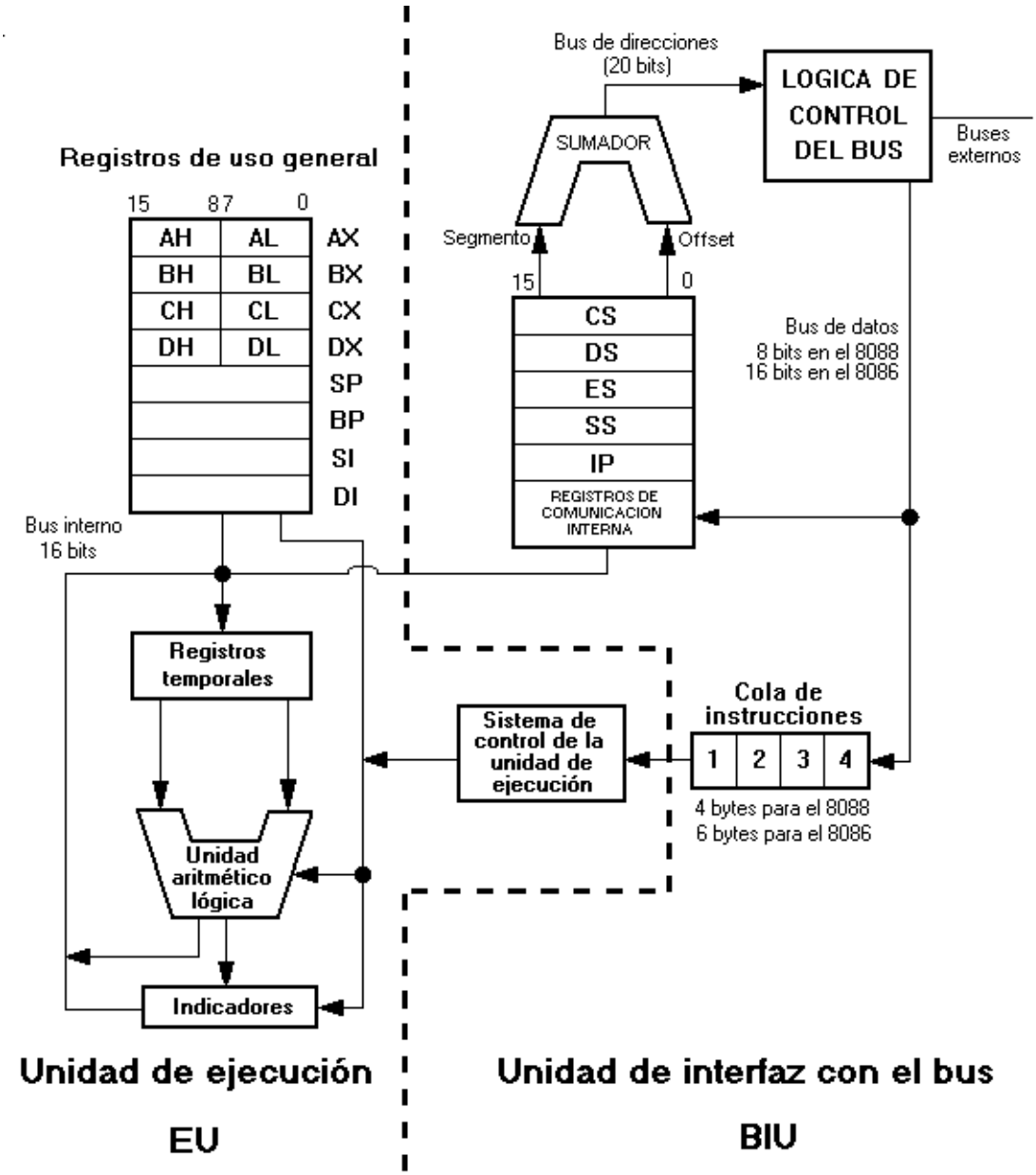

Figura 2.8: Diagrama esquemático microprocesador Intel 8086

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], [50]. 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.

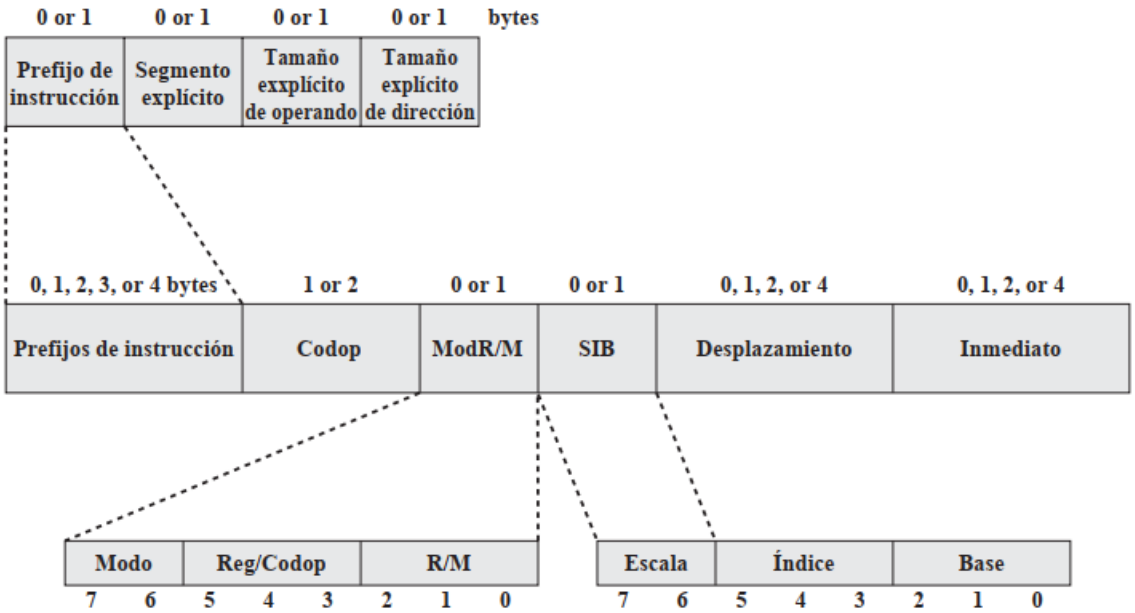

Figura 2.9: Formato de instrucciones del Pentium x86

Un ejemplo típico de instrucción es:

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

```

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], [34], [50].

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], [52].

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], [28], [33], [52].

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 [28], [33]. 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], [28]. 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.

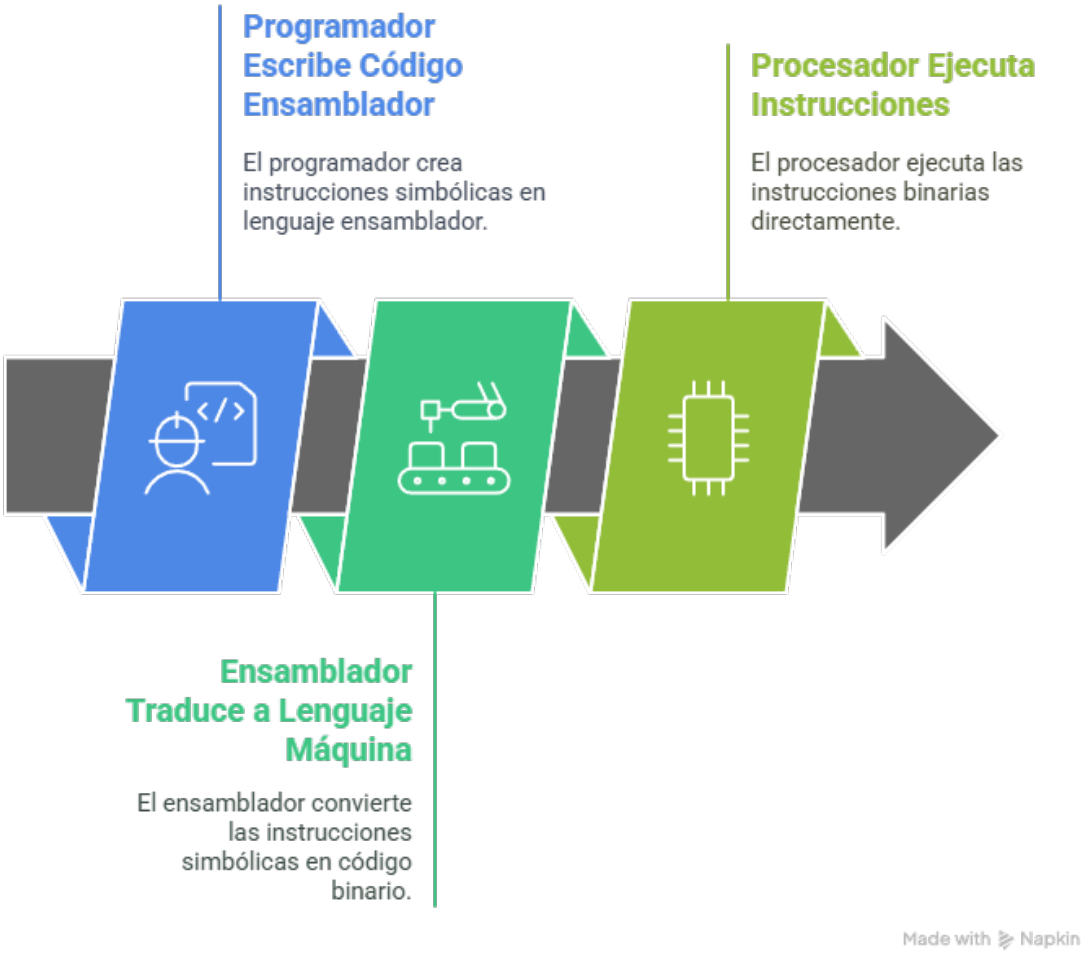

Figura 2.10: Proceso de ensamblado

2.7.2 Ensambladores x86

En el caso de la arquitectura x86, los programadores pueden elegir entre diversos ensambladores, como TASM (Turbo Assembler) [53], MASM (Microsoft Macro Assembler) [54] y NASM (Netwide Assembler) [55]. 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 [56].

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 preprocesamiento
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], [26].

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 [57].

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 [58].

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 sistémico.

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 refuercen 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], [24].

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 [26].

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

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:

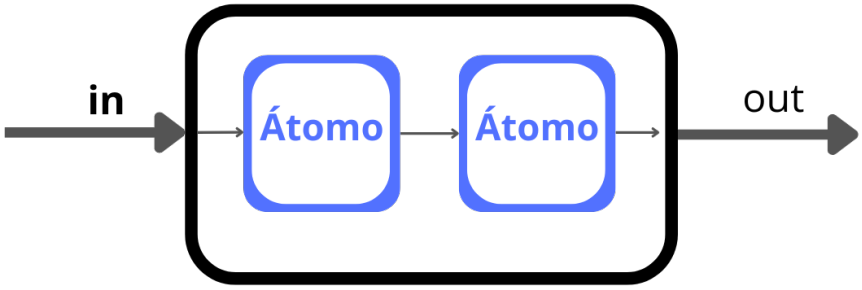

Figura 3.2: Modelo acoplado en DEVS

3.3.2 Aplicaciones del formalismo DEVS

"El formalismo DEVS encuentra aplicación en diversos ámbitos, como las redes de comunicación [59], donde permite simular el enrutamiento de paquetes y la congestión de redes; en entornos de manufactura [26], donde se modelan flujos de producción y control de calidad; y en sistemas de transporte, para la optimización de flujos vehiculares [60]. También se utiliza en la simulación de sistemas biológicos, como la propagación de enfermedades o el comportamiento de poblaciones [61]. 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 [62].

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

3.3.3 DEVS 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 DEVS como parte del entorno educativo, se potencia la capacidad de los estudiantes para abordar problemas complejos y explorar soluciones innovadoras [63].

En conclusión, el formalismo DEVS 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 [64].

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], [65].

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 direccionable 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"[64].

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 y su justificación

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 Anexo A 5.6). 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 [66].

1. **Visualización de la estructura general de la computadora:** Incluir una representación gráfica de la arquitectura básica de la computadora —compuesta por CPU, buses, memoria y dispositivos de entrada/salida— durante la ejecución de los programas. La visualización debe destacar los componentes activos en cada etapa del ciclo de ejecución, facilitando una comprensión sistémica e integrada del funcionamiento de la computadora. 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 [67]. La Figura 5.1 presenta un diagrama estructural que representa los principales módulos del simulador y sus interacciones. Este recurso visual permite integrar, de forma esquemática, los componentes funcionales implementados y su correspondencia con los objetivos pedagógicos definidos.

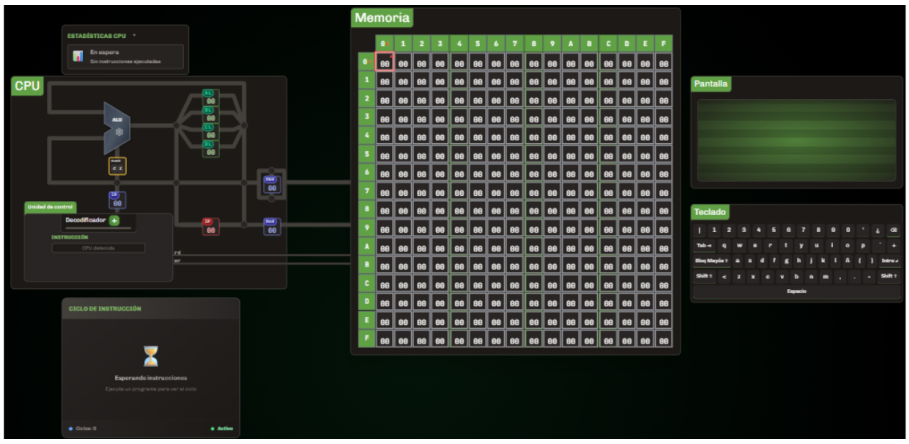

Figura 5.1: Estructura del simulador y componentes funcionales

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 [68]. El editor debe permitir al usuario escribir, editar y guardar

programas en ensamblador, así como ejecutar estos programas dentro del simulador. 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 [69].

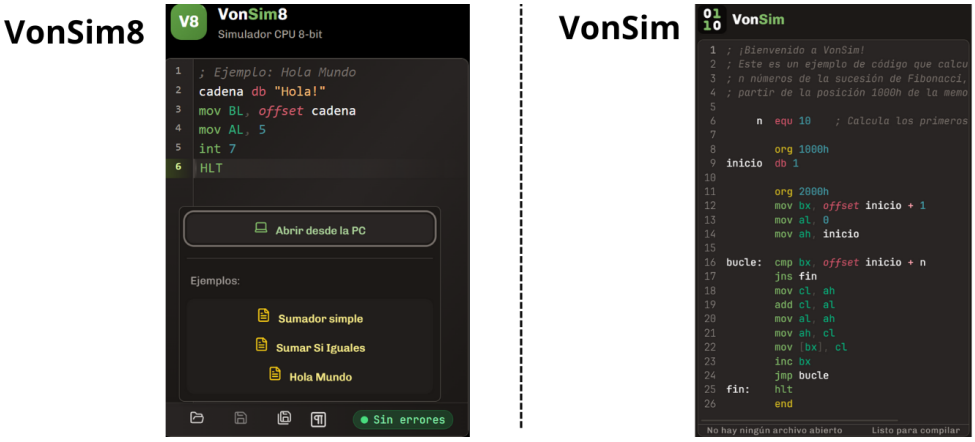

Figura 5.2: Editor ensamblador

3. **Repertorio reducido de instrucciones con activación progresiva:** Se selecciona un subconjunto esencial del conjunto de instrucciones x86, el cual se habilita en etapas secuenciales del proceso de enseñanza, en correspondencia con el avance de los contenidos curriculares. Esta decisión se fundamenta en principios de la psicología cognitiva que sugieren que la introducción escalonada de contenidos técnicos mejora la retención y reduce la sobrecarga cognitiva [70]. Esta estrategia también se encuentra respaldada por autores como Hasan [18], Null y Lobur [33], y Stallings [20], quienes proponen abordajes similares en la enseñanza de arquitecturas complejas.

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, INC, DEC, AND, OR, XOR, NOT, NEG	Manejo de periféricos e interrupciones

Tras la definición del repertorio esencial, el diseño contempla una progresión hacia la representación del ciclo de instrucción en el nivel microarquitectónico, incorporando la visualización dinámica de registros y señales de control. Esta representación incluirá la visualización dinámica de registros activos y señales de control, en correspondencia directa con la ejecución de cada instrucción. Este enfoque promueve el desarrollo progresivo de competencias, al mitigar la sobrecarga cognitiva que implicaría abordar de forma prematura el repertorio completo de instrucciones. La activación progresiva del repertorio se fundamenta en teorías de aprendizaje que sugieren que la exposición gradual a nuevos conceptos mejora la comprensión y retención [71]. Además, esta estrategia se alinea con las recomendaciones de autores como Null y Lobur [33], quienes destacan la importancia de introducir los conceptos de forma escalonada para facilitar el aprendizaje efectivo.

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 [32], [72]. 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 de las señales de control en cada etapa del proceso. Como complemento a la descripción anterior, la Figura 5.3 ilustra un ciclo de instrucción típico utilizando la operación MOV AL, BL como ejemplo. Esta representación facilita la identificación de las etapas *fetch*, *decode* y *execute*, así como los registros que participan activamente en cada fase.

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. Estas interacciones permiten emular situaciones de asincronía, esenciales para comprender mecanismos como la interrupción del flujo de ejecución. 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 [33], 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.

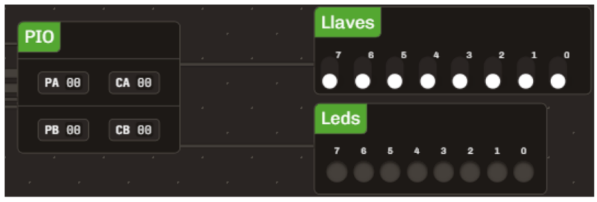

Figura 5.4: Módulo genérico de entrada/salida programada

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. Estos indicadores permiten al estudiante analizar cuantitativamente la eficiencia en la ejecución de un program, 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. **Interfaz intuitiva y experiencia de usuario:** Diseñar una interfaz gráfica de usuario (GUI) intuitiva, accesible y coherente con principios de usabilidad. La organización visual debe facilitar la navegación, la comprensión del flujo de datos y la ejecución de tareas pedagógicas sin distracciones. Se recomienda el uso de colores, etiquetas y animaciones para reforzar el aprendizaje visual, especialmente para representar cambios de estado del sistema y flujos de control [68].

8. **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 [73].

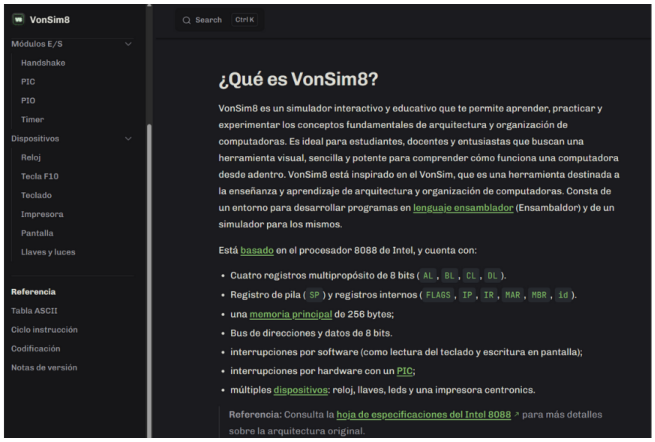

Figura 5.6: Documentación on line

Estos requisitos funcionales y pedagógicos se fundamentan en principios de diseño instruccional y psicología cognitiva, y fueron validados mediante entrevistas con docentes expertos en la materia. A través de reuniones y talleres de validación donde se analizaron prototipos tempranos y necesidades concretas del aula, garantizando así su relevancia didáctica y su adecuación al contexto educativo de la asignatura Arquitectura de Computadoras.

En conjunto, estos requisitos orientan el diseño del simulador para maximizar su impacto pedagógico y su utilidad como recurso de apoyo a la enseñanza de arquitectura x86. En la siguiente sección se describe cómo fueron implementados funcionalmente estos requisitos, detallando su correspondencia con la estructura modular del simulador. La Tabla 5.2 resume los principales requisitos funcionales del simulador, junto con los componentes específicos que los implementan.

Tabla 5.2: Resumen de requisitos funcionales y su fundamentación pedagógica

Requisito	Fundamento pedagógico
Visualización de estructura	Comprensión sistémica del hardware
Ejecución de programas en ensamblador	Desarrollo de competencias en lenguaje ensamblador
Repertorio progresivo de instrucciones	Disminución de sobrecarga cognitiva
Simulación visual de micropasos	Comprensión del flujo interno de datos
Módulo de Entrada/Salida + Vector de interrupciones	Simulación de asincronía real

5.1.1 Justificación del uso de una arquitectura simplificada en el simulador

A partir de los requisitos funcionales detallados anteriormente, se llevó a cabo un proceso colaborativo de análisis con docentes[1] de la asignatura Arquitectura de Computadoras, quienes aportaron su experiencia docente para identificar los elementos de la arquitectura x86 que resultaban prioritarios para representar, simplificar o adaptar en función de los objetivos pedagógicos del simulador. Este análisis condujo a la elección de una arquitectura simplificada de 8 bits, cuyas características se describen más adelante, justificado por su valor didáctico: una arquitectura de 8 bits permite reducir significativamente la complejidad del modelo, sin comprometer la enseñanza de conceptos fundamentales como el ciclo de instrucción, la manipulación de registros o el manejo de interrupciones, permitiendo representar procesos clave con mayor claridad y menor carga cognitiva. La menor cantidad de líneas de datos, registros y operaciones simplifica la visualización de procesos como la ejecución de instrucciones, el flujo de datos y el manejo de interrupciones, favoreciendo la comprensión por parte del estudiante en etapas iniciales del aprendizaje.

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

- **Reducir la carga cognitiva**: la simplificación del repertorio y de los componentes permite a los estudiantes enfocarse en principios fundamentales.
- **Aprendizaje progresivo**: se adopta un enfoque escalonado, empezando con un modelo simplificado y avanzando hacia representaciones más completas de x86.
- **Claridad pedagógica**: las prácticas son manejables en términos de tiempo y esfuerzo, favoreciendo un aprendizaje activo, centrado en la resolución progresiva de problemas y libre de sobrecarga cognitiva excesiva.

En este marco, el diseño del simulador contribuye a:

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

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

Diversos autores como Patterson & Hennessy [19], Tanenbaum [28] y Null [33] coinciden en que el uso de arquitecturas simplificadas, como las de 8 bits, permite a los estudiantes centrarse en los conceptos fundamentales de la arquitectura de computadores 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, favoreciendo la comprensión del funcionamiento interno del procesador. El modelo propuesto adopta una arquitectura simplificada de 8 bits, inspirada en los principios de la arquitectura x86 [45], y diseñada con un repertorio reducido de instrucciones. La elección de una arquitectura de 8 bits obedece a criterios pedagógicos, ya que simplifica el modelo sin sacrificar los principios fundamentales del repertorio x86, facilitando así la comprensión progresiva de sus componentes [74], [75], [76], [77], [78].

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

5.2 Diseño del Simulador

El simulador implementado adopta la arquitectura de Von Neumann, reconocida por su simplicidad conceptual y operativa. En este modelo, los datos y las instrucciones comparten una única memoria, lo que permite tratar las instrucciones como datos [20].

5.2.1 VonSim

VonSim[2] sirvió como referencia por su enfoque educativo e interfaz intuitiva. Sobre esta base se desarrolló VonSim8[3], adaptado para operar con registros y memoria de 8 bits y diseñado para el aprendizaje progresivo [79].

La arquitectura detallada de VonSim, con su amplio repertorio de instrucciones y componentes, ofrece una visión integral del sistema. Sin embargo, dicha riqueza funcional puede abrumar a estudiantes en etapas iniciales. Por esta razón, VonSim8 implementa una simplificación estratégica orientada a la reducción de 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

[2] VonSim: https://vonsim.github.io/
[3] VonSim8: https://ruiz-jose.github.io/VonSim8/

introdujeron diversas modificaciones en los componentes, instrucciones y funcionalidades del simulador, priorizando aquellos aspectos conceptuales que se abordan en el programa de la asignatura.

VonSim [79] es una herramienta diseñada específicamente para la enseñanza y el aprendizaje de la arquitectura y organización de computadoras. Sus principales características la posicionan como una solución educativa integral:

Características técnicas y pedagógicas de VonSim:

1. **Entorno integrado de desarrollo y simulación:** Proporciona un entorno completo que incluye editor de código ensamblador con resaltado de sintaxis y un simulador para la ejecución de programas, facilitando el proceso de aprendizaje práctico [79].

2. **Fundamento en arquitectura real:** Se basa en el procesador Intel 8088, proporcionando una referencia histórica y técnicamente relevante para el estudio de la evolución de las arquitecturas de computadoras [51].

3. **Componentes esenciales para el estudio:** Incluye cuatro registros multipropósito de 16 bits, memoria principal de 32 kB, bus de direcciones de 16 bits y bus de datos de 8 bits, abarcando los elementos fundamentales para la comprensión de la arquitectura von Neumann [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, LEDs e impresora Centronics, 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 [74].

7. **Desarrollo académico especializado:** Creado por Facundo Quiroga, Manuel Bustos Berrondo y Juan Martín Seery, con 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 apoya 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 Affero 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 [80].

En síntesis, VonSim constituye una herramienta robusta y accesible que simplifica y facilita el aprendizaje de conceptos complejos de arquitectura de computadoras mediante simulación práctica y una interfaz pedagógicamente orientada.

Principales características tecnológicas

Su desarrollo integra tecnología moderna del ecosistema web y buenas prácticas de software, proporcionando una experiencia didáctica visual, modular y accesible.

- React: Framework de interfaz moderna para construir componentes interactivos y reactividad eficiente.
- TypeScript: Tipado estático que aporta robustez y facilita el mantenimiento del código.
- React Spring: Animaciones fluidas para mejorar la experiencia visual y pedagógica.
- Jotai: Gestión de estado global, permitiendo sincronización y modularidad entre los distintos componentes.
- Tailwind CSS: Framework de estilos utilitario para lograr interfaces responsivas y coherentes.
- Pruebas Automatizadas: Tests unitarios, de integración y de accesibilidad (incluyendo ARIA, navegación por teclado y roles semánticos) para garantizar calidad y accesibilidad.
- Arquitectura modular y educativa
  - Componentes independientes y reutilizables, lo que facilita tanto la extensión como el mantenimiento.
  - Integración de paneles educativos, tutoriales interactivos y gamificación mediante sistemas de logros y niveles de aprendizaje.

  - Adaptabilidad a distintos niveles (básico, intermedio, avanzado) y retroalimentación visual en tiempo real.

- Simulación fiel de arquitectura de computadoras
  - CPU de 8 bits: Registros, ALU, buses, ciclo de instrucción y modos de direccionamiento.
  - Memoria RAM: Simulación de 256 bytes con direccionamiento de 8 bits.
  - Lenguaje ensamblador: Permite escribir, ejecutar y depurar programas, visualizando el flujo de instrucciones y operandos.
  - Sistema de E/S configurable: Integración de periféricos como teclado, pantalla, PIC, PIO y temporizador, siguiendo especificaciones reales de la familia Intel iAPX 88.

- Visualización y experiencia de usuario
  - Indicadores visuales avanzados para las fases de ejecución (fetch-decode-execute).
  - Animaciones de flujo de datos y cambios de estado en tiempo real.
  - Paneles flotantes y tooltips educativos para explicar conceptos clave y apoyar el aprendizaje autónomo.

- Documentación y recursos
  - Documentación exhaustiva que facilita la comprensión de cada componente y concepto abordado.
  - Recursos adicionales como tutoriales interactivos, referencia de instrucciones y guías de arquitectura.

- Accesibilidad y buenas prácticas
  - Cumplimiento de estándares WCAG 2.1 AA.
  - Enfoque en la accesibilidad y la inclusión educativa.
  - Estrategias de testing, documentación y contribución enfocadas en el impacto pedagógico.

VonSim8 se consolida como una herramienta educativa robusta y moderna, que combina simulación fiel, interactividad y accesibilidad, facilitando el aprendizaje profundo de los fundamentos de la arquitectura de computadoras.

5.2.2 Las modificaciones principales de VonSim

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 5.7: Registro FLAGS

El flag de interrupcion 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.

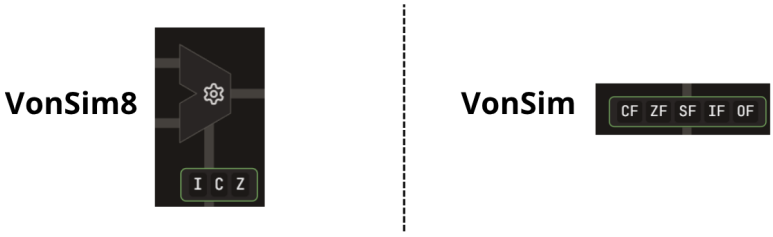

Figura 5.8: Registro de estado I

Se modifico el menu de los controles del simulador.

Figura 5.9: Controles del simulador

Se eliminaron los registros de 16 bits y se redujo el tamaño de los registros a 8 bits, lo que simplifica la representación y manipulación de datos. Esta decisión se fundamenta en la necesidad de reducir la complejidad del modelo para facilitar la comprensión de los conceptos fundamentales de la arquitectura de computadoras. Asimismo, se unificó el criterio de diseño de los registros: todos ellos disponen ahora de una entrada y una salida independientes, lo que permite una visualización más clara de la transferencia de datos entre los registros y la Unidad Aritmético-Lógica (ALU). Esta modificación es esencial para comprender el flujo de datos en el ciclo de instrucción y la interacción entre los componentes del procesador. Tambien se ocultaron los registros registro SP y (`ri` e `id`) que se habilitan automaticamente al ejecutar una instrucción que requiera su uso.

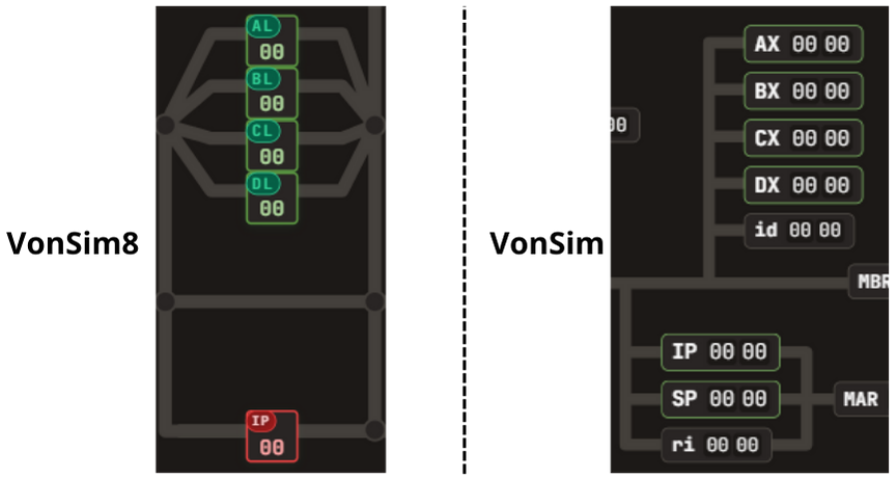

Figura 5.10: Registro de 8 bits

Se elimino el uso de los registros temporales de la ALU (`left` y `rigth`). Por una cuestion de diseño del simulador algunos registros tienen la entrada y salida de manera horizontal y otros vertical, pero su funcionamiento es el mismo.

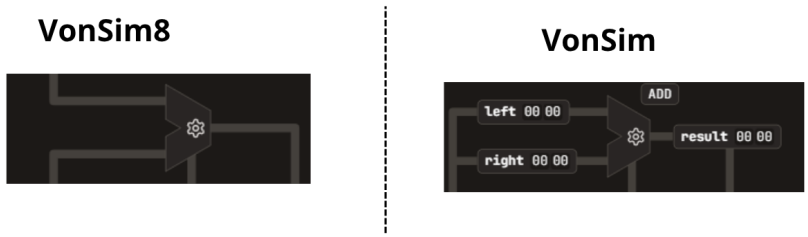

Figura 5.11: Eliminación registro temporales left y rigth

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

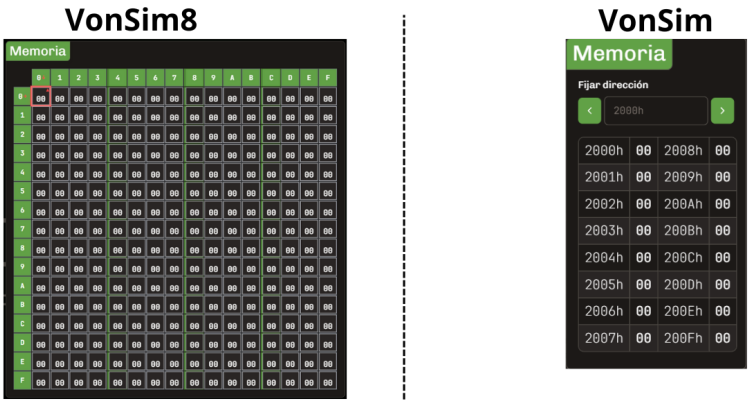

Figura 5.12: Memoria principal

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

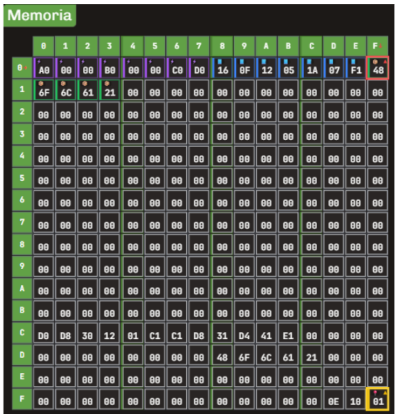

Figura 5.13: Resaltado registro IP y registro SP

El espacio destinado al vector de interrupciones en el VonSim8 es de 8 posiciones de memoria desde `0x00` a `0x07` y solamente implementa las interrupciones por software:

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

Resaltado vector de interrupciones en memoria:

Figura 5.14: Resaltado vector de interrupciones

Visor de instrucciones y datos del programa en memoria, pudiendo ver la instruccion y su tamaño en bytes que ocupa en memoria, ademas la etiqueta asociada a los datos.

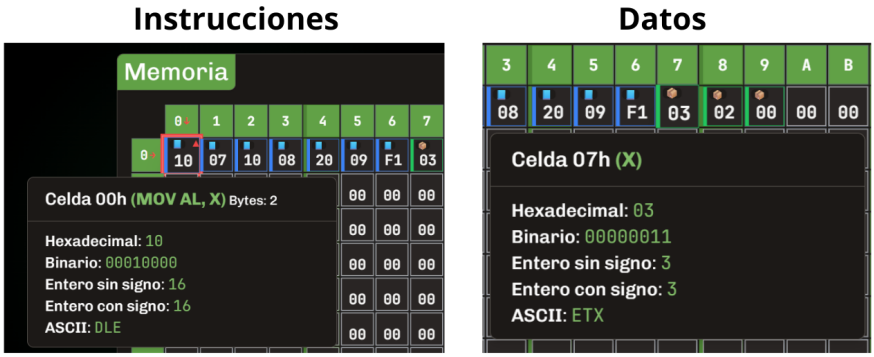

Figura 5.15: Visor de instrucciones y datos del programa en memoria

En el VonSim cuando se escribe un programa en el editor del simulador es obligatorio que la sección de codigo inicie con la directiva `org 0x2000h`, porque el simulador comienza a ejecutar la primer instrucción del programa a partir de la dirección de memoria `0x2000h`, esto se indica al comienzo del programa mediante la directiva `org 0x2000h` y los datos del programa generalmente se cargan a partir de la dirección de memoria `0x1000h` mediante la directiva `org 0x1000h`.

En VonSim8 el uso de la directiva org para definir la dirección inicial del programa es opcional. De manera predeterminada, 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 al inicio, reservando espacio para el vector de interrupciones. VonSim8 tambien permite

cargar el programa de manera similar como VonSim pero en utilizar la directiva `org 0x2000h` utiliza org `0x20h` y el simulador comienza a ejecutar la instrucción que se encuentra en la dirección `0x20h` de la memoria vez de cargar en la posición ya que el simulador asigna automáticamente la dirección 0x00 al primer byte del programa. Esto simplifica la escritura de programas y evita errores comunes relacionados con la asignación manual de direcciones.

Pero si el programa requiere una dirección de inicio específica, se puede utilizar la directiva `org` para establecerla. Por ejemplo, `org 0x20h` indicaría que el programa debe comenzar en la dirección 0x20 de la memoria. Por compatibilidad con el simulador VonSim, se mantiene la directiva `org` para aquellos usuarios que deseen especificar una dirección de inicio diferente a la predeterminada. Sino se especifica la directiva `org` en el programa el simulador comienza a ejecutar el programa desde la dirección `0x00h`, en caso de que el programa tenga instrucciones de interrupción `INT`, el simulador asigna automáticamente la dirección `0x08h` al primer byte del programa, dejando espacio para el vector de interrupciones.

Se incorporó la opción de ampliación de fuente en el editor, así como ejemplos prácticos que permiten al estudiante probar y experimentar directamente con el simulador.

Figura 5.16: Resaltado vector de interrupciones y registro SP

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.

Con el fin de facilitar el aprendizaje del simulador VonSim8 se implementó un tour de aprendizaje que guía a los usuarios a través de las características y funcionalidades del simulador. Tambien en centro de aprendizaje con explicación de ejemplos basicos.

Tabla 5.3: 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.
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.
Interrupciones implementadas	Completo soporte de interrupciones por hardware y software, con PIC.	Solo interrupciones por software: ‘INT 0‘ (fin), ‘INT 6‘ (entrada de teclado), ‘INT 7‘ (salida de cadena).
Temporales de la ALU	Registros temporales ‘left‘ y ‘rigth‘ visibles.	Eliminados, para simplificar la representación del ciclo de instrucción.
Enfoque pedagógico	Mayor fidelidad al procesador Intel 8088, con repertorio más amplio.	Simplificación estratégica para reducir carga cognitiva y favorecer aprendizaje progresivo.

5.2.3 Requisitos funcionales del simulador VonSim8

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.

Las modificaciones implementadas se alinean con los contenidos curriculares de la asignatura y están fundamentadas en los principios del aprendizaje activo [73].

5.2.4 Estructura del 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]. Se describe en la siguiente tabla 5.4 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 y su propósito pedagógico:

Tabla 5.4: 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: • 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 ‘0x00‘ y final ‘0xFF‘.
Puertos	Puertos de la CPU: • 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)

Dentro de los registros específicos que no pueden ser accedidos por el usuario, se encuentra el registro `SP` (*stack pointer*) para el funcionamiento de la pila, el registro `FLAGS` (*flags register*), el `IP` (*instruction pointer*) que almacena la dirección de la próxima instrucción a ejecutar, el `IR` (*instruction register*) que almacena el byte de la instrucción que se está analizando/decodificando en un instante dado. También hay 2 registros dedicados a la transferencia de información entre la CPU y la memoria principal: el `MAR` (*memory address register*) encargado de almacenar direcciones de memoria, y el `MBR` (*memory buffer register*) que almacena el byte que se quiere propagar o se ha recibido por el 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.

5.2.5 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.

5.2.6 ALU

La ALU (*Arithmetic Logic Unit*) permite realizar operaciones aritméticas y lógicas de 8 bits. Las operaciones disponibles son: `ADD`, `INC`, `SUB`, `DEC`, `NEG`, `NOT`, `AND`, `OR` y `XOR`. Todas estas operaciones modifican el registro `FLAGS`.

Flags

El registro `FLAGS` es un registro de 8 bits que contiene las *flags* mostradas en la siguiente tabla. Este registro no es directamente accesible por el usuario, pero puede

Tabla 5.5: Registro FLAGS: descripción de las banderas

Bit N°	Abreviatura	Descripción
0	Z	Flag de cero (Zero)
1	C	Flag de acarreo (Carry)
2	O	Flag de desbordamiento (Overflow)
3	S	Flag de signo (Sign)
4	I	Flag de interrupción (Interrupt)

ser modificado por las operaciones de la ALU y pueden realizarse saltos condicionales en base a sus valores.

Los bits restantes del registro FLAGS se encuentran reservados para futuras extensiones del simulador y no son utilizados en la versión actual.

5.2.7 Memoria

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

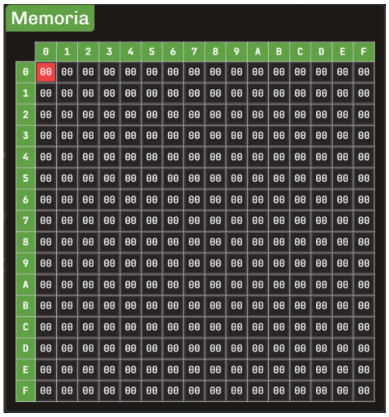

Figura 5.17: Memoria principal

5.2.8 Buses y multiplexores

El bus de datos, direcciones y control se implementa como un conjunto de líneas que permiten la comunicación entre los distintos componentes del sistema. Este bus es esencial para el intercambio de información entre la CPU, la memoria y los dispositivos de entrada/salida, y su diseño modular permite una fácil expansión en futuras versiones del simulador.

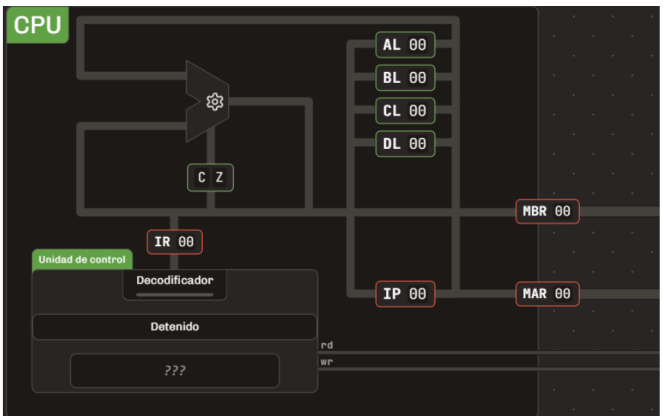

Figura 5.18: Buses

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. Estos multiplexores permiten dirigir las señales adecuadas a los registros y a la ALU (Unidad Aritmético Lógica), facilitando así el flujo de datos dentro del procesador. Un multiplexor (MUX) es 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 [81].

5.2.9 Periféricos: teclado y pantalla

El simulador incluye la representación de un teclado y una pantalla con el objetivo de emular la interacción básica entre el usuario y el sistema. El teclado se modela como un vector de 16 posiciones, cada una capaz de almacenar un carácter ASCII. La pantalla, por su parte, se representa como una matriz de 16×16 que permite visualizar caracteres, facilitando así la comprensión del manejo de entrada y salida de datos en una arquitectura computacional simplificada.

Figura 5.19: Teclado y pantalla

La selección de estos bloques funcionales se fundamenta en dos criterios principales: por un lado, la necesidad de modelar los componentes esenciales de una computadora real; y por otro, la búsqueda de modularidad y claridad conceptual con fines pedagógicos. La Tabla 5.6 resume los principales bloques que conforman la arquitectura simulada, junto con una breve descripción de su funcionalidad.

Tabla 5.6: Bloques funcionales principales

Bloque Funcional	Descripción
Unidad Central de Procesamiento (CPU)	Simulación de registros, unidad de control (UC) y unidad aritmético-lógica (ALU).
Memoria	Estructura de memoria y simulación de operaciones de lectura y escritura.
Sistema de Entrada/Salida (E/S)	Interacción con periféricos y manejo de interrupciones.
Bus de datos, direcciones y control	Modelado de la comunicación entre componentes.

La Figura 5.20 presenta una visión esquemática de la arquitectura general del simulador. En ella se distinguen los flujos de datos y de control, representando gráficamente la interacción entre los distintos módulos funcionales durante la ejecución de programas. Dicha representación facilita la comprensión estructural del sistema y su analogía con una arquitectura computacional real, simplificada para fines educativos.

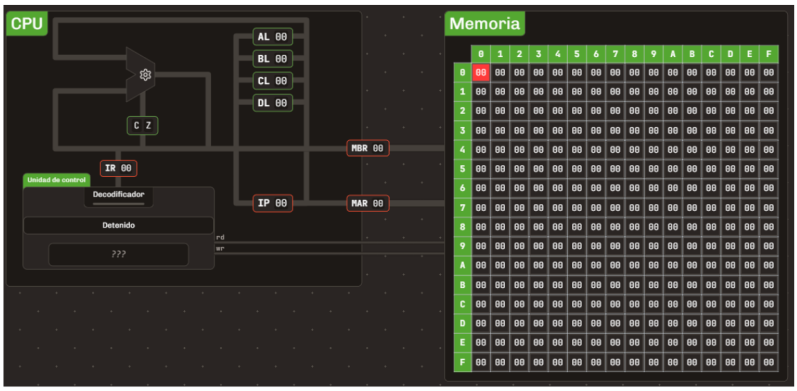

Figura 5.20: Arquitectura general del simulador

5.2.10 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, permitiendo a los estudiantes familiarizarse con las operaciones básicas sin abrumarse con la complejidad del conjunto completo de instrucciones x86.

El repertorio de instrucciones se diseñó como una abstracción deliberada inspirada en la arquitectura x86, con el objetivo de optimizar los procesos de enseñanza y aprendizaje en entornos educativos. En una etapa inicial del proceso de enseñanza, se incluyen únicamente aquellas instrucciones esenciales que permiten introducir progresivamente los contenidos básicos de la asignatura Arquitectura de Computadoras. Este enfoque progresivo permite introducir los conceptos fundamentales de manera accesible, minimizando la complejidad innecesaria que podría obstaculizar la comprensión en las etapas iniciales del aprendizaje [19], [28]. La Tabla 5.7 muestra un conjunto reducido de instrucciones que abarca las operaciones más frecuentes y pertinentes para una etapa introductoria de aprendizaje. El repertorio se centra en instrucciones de transferencia y procesamiento de datos, junto con aquellas de control de flujo. Esta selección permite introducir los conceptos básicos de la arquitectura x86 de manera accesible, evitando la complejidad del repertorio completo y alineándose con 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.

Para facilitar el aprendizaje de la programación en ensamblador e introducir el ciclo de instruccion se propone el siguente repertorio de instrucciones, luego cuando se avance sobre los modulos de entrada y salida de acuerdo al programa de estudio de la asignatura Arquitectura de Computadoras [82].

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.

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

En base a las entrevistas con los docentes y el análisis de los contenidos del curso, a continuación se establece para cada categorías de instrucciones su uso pedagógico esperado:

- **Transferencia y procesamiento de datos**: Instrucciones que permiten mover datos entre registros y memoria y realizar operaciones aritméticas. Estas instrucciones son fundamentales para la comprensión del flujo de datos en una

arquitectura computacional mostrando como se llevan a cabo operaciones aritméticas como de lenguaje de alto nivel como Python:

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

La traducción equivalente al lenguaje ensamblador sería:

```
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, permiten implementar estructuras condicionales similares a las de lenguajes de alto nivel como Python:

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

Su traduccion a ensamblador sería:

```
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
```

Su traduccion a ensamblador sería:

```
x db 2
y db 3
z db 0
          mov AL, x
          cmp AL, y
          jc EsMenor
          jmp Fin
EsMenor:  add AL, y
          mov z, AL
Fin:      hlt
```

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

```
x = 0
suma = 0

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

Su traduccion a ensamblador sería:

```
x     db 1
suma  db 0
Condicion:  cmp x, 10
            jc Bucle        ; si x < 10  salta a etiqueta Bucle:
            jmp FinBucle      ; si no salta a la etiqueta FinBucle:
Bucle:      mov BL, x
            add suma, BL
            add x, 1
            jmp Condicion      ; salta a Condicion:
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:

```python
# 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]
```

Su traduccion a ensamblador sería:

```
max     db 0
vector  db 5, 2, 10, 4, 5, 0, 4, 8, 1, 9
            mov CL, 0 ; contador
            mov BL, offset vector  ; obtiene la dirección del primer
    elemento del vector
Condicion:  cmp CL, 10
            jc Bucle          ; si x < 10  salta a etiqueta Bucle
            jmp FinBucle      ; si no salta a la etiqueta FinBucle
Bucle:      mov AL, [BL]      ; AL = vector[indice]
            cmp AL, max
            jc Proximo        ; si AL < max, salta a Proximo
            mov max, AL       ; si no, actualiza max
Proximo:    add BL, 1         ; BL = BL + 1
            add CL, 1         ; CL = CL + 1
            jmp Condicion
FinBucle:   hlt
```

La selección de instrucciones busca facilitar la comprensión de los procesos fundamentales de la arquitectura de computadoras: la ejecución de operaciones aritméticas, la transferencia de datos entre registros y memoria, y el control del flujo de ejecución. Al tratarse de un repertorio reducido e inspirado en la arquitectura x86, se garantiza un equilibrio entre rigor conceptual y simplicidad pedagógica.

La implementación de estas instrucciones en el simulador VonSim8 está orientada a proporcionar una experiencia de aprendizaje que apoye la comprensión de los principios fundamentales de la arquitectura de computadoras, reduciendo la carga cognitiva en las fases iniciales de la formación y fortaleciendo la transición hacia repertorios más complejos. El simulador facilita que el estudiante comprenda cómo se ejecutan las operaciones aritméticas en la computadora, cómo se transfieren los datos entre registros y memoria, y cómo se controla el flujo de ejecución de un programa.

5.2.11 Modos de direccionamiento

Los modos de direccionamiento en un repertorio de instrucciones son los métodos que tiene un procesador para especificar la ubicación de los datos (operandos) que se van

a utilizar en una instrucción. Esto permite al procesador saber de dónde tomar los datos para realizar una operación y dónde guardar el resultado. A continuación, se describen los modos de direccionamiento definidos en VonSim8:

- Registro a registro (`Rx`,`Ry`): operandos son registros del procesador y `Rx` indica registro destino y `Ry` indica registro fuente.
- Directo (`[M]`): un operando es un registro y el otro operando es el contenido de una dirección de memoria `[M]`.
- Indirecto (`[BL]`): la dirección del operando se encuentra en el registro `[BL]`.
- Inmediato (`d`): un operando es un valor contenido en la instrucción.

Estos modos de direccionamiento permiten al simulador representar una variedad de operaciones, desde la manipulación directa de registros hasta el acceso a la memoria. Esto proporciona una base sólida para comprender el flujo de datos en una arquitectura computacional. La Tabla 5.9 resume los modos de direccionamiento implementados en el simulador para instrucciones de dos operandos, e incluye ejemplos de su uso.

Tabla 5.9: Tabla de modos de direccionamiento

Destino	Fuente	Ejemplo
Registro Rx	Puede ser: • registro Ry • dirección [M] • dirección registro [BL] • valor inmediato d	• Entre registro: MOV Rx, Ry • Directo: MOV Rx, [M] • Indirecto: MOV Rx, [BL] • Inmediato: MOV Rx, d
Memoria puede ser: • **dirección [M]** • **dirección registro [BL]**	Puede ser: • Registro: Ry • valor en la instrucción d	• Directo: MOV [M], Ry • Indirecto: MOV [BL], Ry • Directo-Inmediato: MOV [M], d • Indirecto-Inmediato: MOV [BL], d

Nota:
Las instrucciones de dos operando, tanto de transferencia y procesamiento tienen los mismos modos de direccionamiento.
Rx y Ry puede 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.

Ejemplo de los modos de direccionamiento:

```
x db 2
y db 3

; ejemplo modos direccionamiento
;----------------------
; carga en registro
;----------------------
; Directo
mov al, x

; Por registro
mov dl, al

; Inmediato
mov bl, 16

; Indirecto
mov cl, [bl] ; celda 16 = bl

;----------------------
; Almacenar en memoria
;----------------------
; Diecto
mov x, cl

; Indirecto
mov [bl], al

; Directo-Inmediato
mov x, 5

; Indirecto-Inmediato
mov [bl], 4

hlt
```

5.2.12 Formato de instrucciones

El formato de las instrucciones se basa en la codificación binaria, poseen una longitud variable (1, 2 o 3 bytes), donde los 4 bits de mayor peso del primer byte representa el código de operación (opcode) y los siguientes 4 bits pueden contener operandos adicionales según el modo de direccionamiento utilizado. El opcode determina la operación a realizar, mientras que los operandos especifican los datos o registros involucrados en la operación. 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.

Tabla 5.10: Categorías 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 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)
Control de flujo	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 acción correspondiente y su codificación binaria.

Considerando:

- `____`: Código de operación de la instrucción, número de 4 bits.
- `Rx` o `Ry`: Índices de registros, número entre `0` y `3`, cada índice es de 2 bits.
- `OO`: Junto con el código de operación indica si la operación es tipo B o C.
- `M`: Dirección de memoria, número de 8 bits.
- `ffff`: representan el comportamiento de la instrucción, número de 4 bits.
- `d`: Dato inmediato, número de 8 bits.
- `MMMMMMMM`: Dirección de memoria, número de 8 bits.
- `ddddddd`: Dato inmediato, número de 8 bits.

5.3 Ciclo de la instrucción

El ciclo de la instrucción es la secuencia de pasos que sigue la Unidad de Control (UC) para ejecutar cada instrucción de un programa. Es un proceso esencial para el

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 MMMMMMMM`
1	`MOV Rx, [BL]`	1	`0001 Rx01`
1	`MOV Rx, D`	2	`0001 Rx10 MMMMMMMM`
2	`MOV [M], Ry`	2	`0010 00Ry MMMMMMMM`
2	`MOV [BL], Ry`	2	`0010 01Ry`
2	`MOV [M], D`	3	`0010 1100 MMMMMMMM dddddddd`
2	`MOV [BL], D`	2	`0010 1101 MMMMMMMM`
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 MMMMMMMM`
C	`Jxx M`	1	`1100 ffff MMMMMMMM`
D	`HLT`	1	`1101 0000`

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.

funcionamiento de cualquier computadora, en el que intervienen elementos clave como registros, buses de datos, direcciones y control, y las señales de control generadas por la UC.

Las microoperaciones que lo componen se expresan con la notación de transferencia entre registros: `destino` ← `origen` .

La Figura 5.21 muestra el flujo general del ciclo, que se divide en dos etapas principales: captación (fetch) y ejecución.

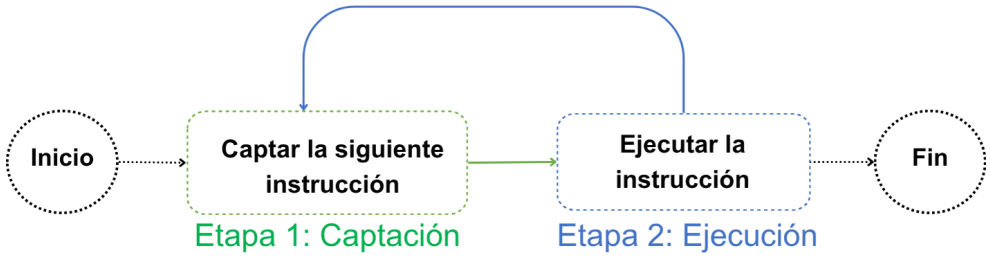

Figura 5.21: Flujo del ciclo de instrucción en VonSim8

5.3.1 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`** ← **`IP`** La UC transfiere la dirección de la próxima instrucción, almacenada en el **Puntero de Instrucción** (`IP`) al **Registro de Direcciones de Memoria** (`MAR`).

2. **`MBR`** ← **`read(Memoria[MAR])`** | **`IP`** ← **`IP`** + **`1`** La UC activa la señal de lectura a través del bus de control para obtener la instrucción indicada en la dirección del `MAR`. El valor 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`** ← **`MBR`** El contenido del `MBR` se transfiere al `IR`. A partir de este momento, la instrucción está lista para ser decodificada y ejecutada.

5.3.2 Etapa 2: Ejecución:

En esta etapa, el **decodificador de instrucciones** interpreta el valor en el registro `IR`. A partir del código de operación (opcode), lee las microinstrucciones necesarias

desde la **memoria de control** para determinar el tipo de instrucción, la cantidad de operandos y el modo de direccionamiento.

Luego, envía estas microinstrucciones al **secuenciador**, que genera las señales de control exactas para ejecutar la operación.

A continuación se explican 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 ocupa un paso y puede ser:
       * MOV: `Rx` ← `Ry`
       * ADD: `Rx` ← `Rx` + `Ry` | **update(Flags)**
       * SUB: `Rx` ← `Rx - Ry` | **update(Flags)**
       * CMP: `Rx - Ry` | **update(Flags)** (solo actualiza flags, no guarda el resultado)
  - **Modo directo (`Rx`, `[Dirección]`)**
    4. `MAR` ← `IP` – Obtener dirección del operando fuente.
    5. `MBR` ← `read(Memoria[MAR])` | `IP` ← `IP` + **1** – Leer la dirección desde memoria e incrementar IP.
    6. `MAR` ← `MBR` – Transferir la dirección al MAR.
    7. `MBR` ← `read(Memoria[MAR])` – Obtener el dato.
    8. Ejecutar la operación:
       * MOV: `Rx` ← `MBR`
       * ADD: `Rx` ← `Rx` + `MBR` | **update(Flags)**
       * SUB: `Rx` ← `Rx - MBR` | **update(Flags)**
       * CMP: `Rx - MBR` | **update(Flags)**
  - **Modo inmediato (`Rx`, `Dato`)**
    4. `MAR` ← `IP` – Obtener dirección del dato inmediato.
    5. `MBR` ← `read(Memoria[MAR])` | `IP` ← `IP` + **1** – Leer el dato e incrementar IP.
    6. Ejecutar la operación (igual que en el caso anterior).
  - **Modo indirecto (`Rx`, `[BL]`)**
    4. `MAR` ← `BL` – Obtener dirección del dato desde el registro BL.
    5. `MBR` ← `read(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 no se almacena en un registro, sino en una posición de memoria especificada directamente en la instrucción o indicada por el contenido del registro BL.

- **Modo Directo (`[Dirección]`, `Ry`)**
  4. `MAR` ← `IP` – Obtener dirección destino.
  5. `MBR` ← `read(Memoria[MAR])` | `IP` ← `IP` + **1** – Leer la dirección e incrementar IP.
  6. `MAR` ← `MBR` – Transferir dirección a MAR. Según la instrucción:
  - MOV:
    7. `MBR` ← `Ry` – Copiar Ry al MBR.
    8. `write(Memoria[MAR])` ← `MBR` – Escribir en memoria.
  - ADD, SUB, CMP:
    7. `MBR` ← `read(Memoria[MAR])` – Leer el dato.
    8. Ejecutar la operación:
       - ADD: `MBR` ← `MBR` + `Ry` | **update(Flags)**
       - SUB: `MBR` ← `MBR - Ry` | **update(Flags)**
       - CMP: `MBR - Ry` | **update(Flags)**
    9. Si es `ADD` o `SUB`: `write(Memoria[MAR])` ← `MBR` – Escribir en memoria.
- **Modo Indirecto (`[BL]`, `Ry`)**
  4. `MAR` ← `BL` – Transferir dirección de destino (en BL) a MAR. Según la instrucción:
  - MOV:
    5. `MBR` ← `Ry` – Copiar Ry al MBR.
    6. `write(Memoria[MAR])` ← `MBR` – Escribir en memoria.
  - ADD, SUB, CMP:
    5. `MBR` ← `read(Memoria[MAR])` – Leer el dato.
    6. Ejecutar la operación (igual que en el caso anterior).
    7. Si es `ADD` o `SUB`: `write(Memoria[MAR])` ← `MBR` – Escribir en memoria.
- **Modo Directo-Inmediato (`[Dirección]`, `Dato`)**
  4. `MAR` ← `IP` – Obtener dirección destino.
  5. `MBR` ← `read(Memoria[MAR])` | `IP` ← `IP` + **1** – Leer dirección e incrementar IP.
  6. `MAR` ← `IP`; `ri` ← `MBR` – Preparar para leer el dato y guardar la dirección destino en un registro intermedio (ri).
  7. `MBR` ← `read(Memoria[MAR])` | `IP` ← `IP` + **1** – Leer dato e incrementar IP. Según la instrucción:
  - MOV:
    8. `MAR` ← `ri` – Copiar dirección destino.
    9. `write(Memoria[MAR])` ← `MBR` – Escribir en memoria.
  - ADD, SUB, CMP:
    8. `MAR` ← `ri` | `id` ← `MBR` – Cargar dirección destino y guardar el valor inmediato en id.

9. `MBR` ← `read(Memoria[MAR])` – Leer el valor actual de destino.
10. Ejecutar la operación:
    - ADD: `MBR` ← `MBR` + `id` | **update(Flags)**
    - SUB: `MBR` ← `MBR - id` | **update(Flags)**
    - CMP: `MBR - id` | **update(Flags)**
11. Si es `ADD` o `SUB`: `write(Memoria[MAR])` ← `MBR` – Escribir en memoria.

– **Modo Indirecto-Inmediato (`[BL]`, `Dato`)**

4. `MAR` ← `IP` – Obtener dirección del dato inmediato.
5. `MBR` ← `read(Memoria[MAR])` | `IP` ← `IP` + **1** – Leer dato e incrementar IP. Según la instrucción:

∗ MOV:

6. `MAR` ← `BL` – Copiar dirección de destino.
7. `write(Memoria[MAR])` ← `MBR`

∗ ADD, SUB, CMP:

6. `MAR` ← `BL` | `id` ← `MBR` – Cargar la dirección destino y guardar el valor inmediato en id.
7. `MBR` ← `read(Memoria[MAR])` – Leer el valor actual de destino.
8. Ejecutar la operación:
    - ADD: `MBR` ← `MBR` + `id` | **update(Flags)**
    - SUB: `MBR` ← `MBR - id` | **update(Flags)**
    - CMP: `MBR - id` | **update(Flags)**
9. Si es `ADD` o `SUB`: `write(Memoria[MAR])` ← `MBR` – Escribir en memoria.

Instrucciones con un operando `JMP` y `Jxx`:

- **Salto a (`Dirección`)** Tanto incondicional `JMP` como condicionales `Jxx` tienen estos pasos:

  4. `MAR` ← `IP` – Obtener la dirección del salto.
  5. `MBR` ← `read(Memoria[MAR]); IP` ← `IP` + **1** – Leer la dirección de destino e incrementar IP.

  Según la instrucción:

  – JMP:

  6. `IP` ← `MBR`

  – Jxx:

  6. `IP` ← `MBR` si se cumple la condición del flag `xx`; en caso contrario, continúa con la siguiente instrucción.

Instrucciones sin operandos

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

5.4 Modulo de Entrada/Salida

El simulador puede configurarse para conectar módulos de entrada/salida al bus y otros dispositivos. Estos se encuentran agrupados, y cada uno de estos grupos puede conectarse solo si lo desea el usuario:

- Teclado y pantalla: un teclado y una pantalla.
- PIC: un PIC, además de la tecla F10 para interrumpir, y un timer con su reloj.
- PIO: un PIO que puede conectarse a llaves y luces o a una impresora Centronics.
- Handshake: Una impresora Centronics conectada por un Handshake.

Para poder utilizar los dispositivos de entrada y salida es necesario presentar nuevas instrucciones al conjunto de instrucciones del procesador.

Tabla 5.13: Tabla de Instrucciones y Códigos de Operación de la Arquitectura x86

Código operación	Instrucciones	Nemónico	Acción
MOV	Transferencia de datos	MOV destino, origen	1- Copiar entre registros 2- Cargar a registro 3- Almacenar en memoria
ADD	Aritmética	ADD destino, origen	1- Sumar 2- Restar 3- Comparar
JMP	Control de flujo	JMP destino	Salto incondicional JMP. Saltos condicionales Jxx. Llamadas a rutinas CALL y retorno RET
PUSH, POP, OUT, IN	Pila y E/S	PUSH, POP, OUT, IN	Poner en la pila PUSH. Retirar de la pila POP. Enviar un byte al puerto del dispositivo de E/S. Recibir un byte del dispositivo de E/S
INT , IRET	Interrupciones	INT , IRET	Llamar a una rutina de tratamiento de interrupción INT. Retornar de una rutina de tratamiento de interrupción IRET
NOP , HLT	Control	NOP , HLT	No opera NOP. Detiene el CPU HLT

5.4.1 Subrutinas

El procesador también implementa subrutinas. Estas son pequeños fragmentos de código que pueden ser llamados desde cualquier parte del programa. Para ello, se

utiliza la instrucción [`CALL`]. Esta instrucción almacena el `IP` en la pila, y luego realiza un salto a la dirección de la subrutina, modificando el `IP` para que este apunte a la primera instrucción de la subrutina. Para volver de la subrutina, se utiliza la instrucción [`RET`], que desapila la dirección apilada previamente por `CALL` y restaura el `IP`, volviendo a el punto de ejecución posterior a la llamada a la subrutina.

Ejemplo de subrutina:

```
mov al, 1
mov bl, 2
mov cl, 3
call sum3
; ax = 6
hlt

; suma al, bl y cl
sum3: add al, bl
      add al, cl
      ret
```

5.4.2 Interrupciones

El simulador tiene predefinido el vector de interrupción en ciertas posiciones de reservadas de memoria, dentro de dicho vector encontraremos las rutinas del sistema predefinidas para interactuar con el teclado y monitor. En el caso de las interrupciones por software, esta es dada por el operando de la instrucción `INT` número. Una vez interrumpido, el procesador ejecutará la rutina de interrupción asociada a ese número de interrupción. La dirección de comienzo de esta rutina estará almacenada en el vector de interrupciones. Este vector ocupa las celdas `00h` hasta `07h` de la memoria principal, y cada elemento del vector tiene 1 byte de largo – el primer elemento se encuentra en `0h`, el segundo en `1h`, el tercero en `2h`, y así. Cada elemento corresponde con la dirección de inicio de la rutina de interrupción.

El procesador admite interrupciones por hardware y por software, que pueden ser emitidas por el PIC o por la instrucción `INT` respectivamente. Para ejecutar interrupciones por hardware, el procesador debe estar habilitado para recibir interrupciones. Esto es, `IF=1` (la *flag* de interrupciones activada).

Ambas interrupciones deben propocionar un número de interrupción. En el caso de las interrupciones por software, esta es dada por el operando de la instrucción `INT`. En el caso de las interrupciones por hardware, esta es dada por el PIC. El número de interrupción debe ser un número entre `0` y `7`.

Una vez interrumpido, el procesador ejecutará la rutina de interrupción asociada a ese número de interrupción. La dirección de comienzo de esta rutina estará almacenada en

el vector de interrupciones. Este vector ocupa las celdas `00h` hasta `07h` de la memoria principal, y cada elemento del vector tiene 1 byte de largo – el primer elemento se encuentra en `0h`, el segundo en `1h`, el tercero en `2h`, y así. Cada elemento corresponde con la dirección de inicio de la rutina de interrupción.

Específicamente, el procesador:

1. obtiene el número de la interrupción (0-7),
2. apila el registro `FLAGS`,
3. inhabilita las interrupciones `I=0`,
4. apila el registro `IP`,
5. obtiene la dirección de la rutina de interrupción del vector de interrupciones,
6. modifica el `IP` para que apunte a la dirección de la rutina de interrupción.

Y así se comienza a ejecutar la rutina de interrupción. Estas tienen el mismo formato que una subrutina salvo que terminan en `IRET` en vez de `RET`.

Llamadas al sistema

El simulador permite realizar llamadas al sistema o *syscalls*. En el simulador, estas llamadas son realizadas idénticamente a las interrupciones. Así, para realizar una *syscall* basta con interrumpir a la CPU con el número de interrupción correspondiente. Estos números son:

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

Las direcciones del vector de interrupciones asociadas a estos números están protegidas por el sistema, impidiendo que el usuario las modifique.

El contenido de estas rutinas se encuentran almacenadas en el monitor del sistema en las direcciones `A0h`, `B0h` y `C0h` respectivamente.

5.5 Pila

En esta etapa se incorporaron las instrucciones de manejo de pila y de interrupciones, ampliando la funcionalidad del simulador para cubrir un conjunto más completo de operaciones propias de la arquitectura x86.

El procesador implementa la pila como método de almacenamiento accesible por el usuario y por la misma CPU para su correcto funcionamiento. Esta es del estilo *Last In, First Out* (LIFO), es decir, el último elemento en entrar es el primero en salir. La pila se encuentra en la memoria principal, comenzando en la dirección más alta de la misma (`FFh`) y creciendo hacia las direcciones más bajas (`FEh`, `FCh`, etc.). El tope de la pila se guarda en el registro `SP`. Todos los elementos de la pila son de 8 bits.

5.5.1 Instrucciones IN y OUT

Para la comunicacion con los modulos de entrada/salida se utilizan puertos, que son direcciones especiales en el espacio de direcciones del procesador. En este simulador, se implementan dos instrucciones para manejar la comunicación con los puertos: - `OUT puerto, AL`: Esta instrucción 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 enviar datos a dispositivos externos, como impresoras o pantallas. - `IN AL, puerto`: Esta instrucción 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.

5.5.2 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

```
1 org 10h
2 cadena db ‘Hola!‘
3
4 org 20h
5 mov BL, offset cadena
6 mov AL, 5
7 int 7
8 ; Se imprime Hola! (sin las comillas) en la pantalla.
9 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.

5.5.3 Teclado

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 correspona será guardado en la dirección de memoria almacenada en BL según su representación en ASCII.

```
1  org 10h
2  car db 0
3
4  org 20h
5  mov bl, offset car
6  int 6
7  ; El usuario escribe un carácter
8  hlt
9
10 ; El carácter escrito se almacenó en 'car'.
11 ; Por ejemplo, si el usuario presionó la tecla 'a', entonces
12 ; se almacena el valor 61h en 'car'.
```

5.5.4 Puertos de E/S

La memoria de entrada/salida se encuentra aislada de la memoria principal. Es decir, para acceder a ella se utilizan exclusivamente las instrucciones [`IN`] y [`OUT`]). Cuando se quiere acceder a un módulo de entrada/salida, la [CPU] activa la señal `IO/M`, lo que causa que un selector de chips (*chip select*) lea la dirección del bus del direcciones y envíe la señal de lectura/escritura al módulo correspondiente.

El rango de direcciones de la memoria de entrada/salida es de `00h` hasta `FFh` (256 direcciones).

Estos son las direcciones de entrada/salida disponibles en el simulador, todos de 8 bits:

Dirección	Nombre	Módulo
10h	CONT	[Timer]
11h	COMP	[Timer]
20h	EOI	[PIC]
21h	IMR	[PIC]
22h	IRR	[PIC]
23h	ISR	[PIC]
24h	INT0	[PIC]
25h	INT1	[PIC]
26h	INT2	[PIC]
27h	INT3	[PIC]
28h	INT4	[PIC]
29h	INT5	[PIC]
2Ah	INT6	[PIC]
2Bh	INT7	[PIC]
30h	PA	[PIO]
31h	PB	[PIO]
32h	CA	[PIO]
33h	CB	[PIO]
40h	DATA	[Handshake]
41h	STATE	[Handshake]

5.5.5 Llaves y luces

Las llaves o interruptores están conectados al puerto PA/CA del PIO y son dispositivos de entrada. Al cambiar su estado, estos se reflejarán en PA (si el PIO está configurado correctamente). Si se altera PA, los cambios no se verán reflejados en las llaves (no se mueven solas, por así decirlo).

```
1  ; Leer el valor de las llaves como una contraseña hasta que el usuario la
       adivine
2
3  clave db 15                ; Contraseña esperada: 00001111 (en decimal 15)
4  mensaje_ok db ‘Bienvenido!‘ ; Mensaje a mostrar si la contraseña es
       correcta
5
6  ; Configurar PA (Puerto A) como entrada
7  mov al, 15                 ; 00001111b: configura los primeros 4 bits de PA
       como entrada
8  out 32h, al                ; Escribe en CA para configurar PA
9
10 bucle:
11     in al, 30h             ; Lee el valor actual de las llaves desde PA
12     cmp al, clave          ; Compara el valor leído con la contraseña
```

```
13     jz Mostrar_Mensaje     ; Si coincide, salta a Mostrar_Mensaje
14     jmp bucle              ; Si no coincide, vuelve a intentar
15
16 Mostrar_Mensaje:
17     mov bl, offset mensaje_ok ; BL apunta al mensaje de éxito
18     mov al, 11                ; Longitud del mensaje (Bienvenido! tiene 11
       caracteres)
19     hlt                       ; Detiene la ejecución del programa
```

Las luces o LED están conectadas al puerto PB/CB del PIO y son dispositivos de salida. La única forma de cambiar su estado es modificando PB. Estos cambios se reflejarán en las luces si el PIO está configurado correctamente, de lo contrario, las luces se verán 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
```

Actualmente el codigo de operacion de las instrucciones son 4 bits, esto reduce a complejidad del simulador y el numero de instrucciones que se pueden implementar, para no tener que ampliar el tamaño de las instrucciones, pero poder mostrar más instrucciones, por ejemplo logicas y aritmeticas de 1 operando, se implementaron instrucciones con el codigo de operacion 15, entonces las instrucciones logicas con código operación 15 (AND, OR y XOR) tienen los mismos modos de direccionamiento que las instrucciones aritméticas (ADD, SUB y CMP) pero comparten el mismo código de operación.

5.5.6 Etapa de ejecución de instrucciones

- **Subrutinas**
  - **CALL `Dirección`**
    4. **`MAR`** ← **`IP`** – Obtener dirección destino.
    5. **`MBR`** ← **`read(Memoria[MAR])`** | **`IP`** ← **`IP`** + **`1`** – Leer la dirección e incrementar IP.

Tabla 5.15: Categorías 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 ← operando1 + operando2
	SUB	6, 7, 8	Operación aritmética: operando1 ← operando1 - operando2
	CMP	9, 10, 11	Comparación: operando1 - operando2 (no actualiza el destino)
Control de flujo	JMP / Jxx / CALL / INT	12	Salto incondicional JMP, condicionales Jxx, llamada CALL, INT: llamar rutina de interrupción
Gestión de flujo	RET / IRET / CLI / STI / HLT	13	RET: retorno, IRET: retornar de interrupción, NOP: no opera, HLT: detiene el CPU
Manejo de pila y E/S	PUSH / POP / OUT / IN	14	PUSH: poner en la pila, POP: retirar de la pila, OUT: enviar a puerto, IN: recibir desde puerto
Miscelánea	AND / OR / XOR / NOT / INC / DEC	15	Operaciones lógicas y aritméticas

6. `ri` ← `MBR` | `SP` ← `SP - 1`
7. `MAR` ← `SP` | `MBR` ← `IP`
8. `write(Memoria[MAR])` ← `MBR` | `IP` ← `ri`

– **RET**

4. `MAR` ← `SP`
5. `MBR` ← `read(Memoria[MAR])`
6. `IP` ← `MBR`| `SP` ← `SP + 1`

- **Interrupciones**

– **INT `Dirección`**

4. `MAR` ← `IP` – Obtener dirección destino.
5. `MBR` ← `read(Memoria[MAR])` | `IP` ← `IP + 1` – Leer la dirección e incrementar IP.
6. `ri` ← `MBR` | `SP` ← `SP - 1`
7. `MAR` ← `SP` | `MBR` ← `Flags`
8. `write(Memoria[MAR])` ← `MBR` | **update(Flags I=0)**
9. `MBR` ← `IP` | `SP` ← `SP - 1`
10. `MAR` ← `SP`
11. `write(Memoria[MAR])` ← `MBR` | `IP` ← `ri`

– **IRET**

4. `MAR` ← `SP`
5. `MBR` ← `read(Memoria[MAR])`
6. `IP` ← `MBR` | `SP` ← `SP + 1`
7. `MAR` ← `SP`

8. `MBR` ← `read(Memoria[MAR])`
9. `Flags` ← `MBR` | `SP` ← `SP - 1`

– **CLI**

4. **update(Flags I=1)**

– **STI**

4. **update(Flags I=0)**

- **E/S**
  - **OUT**
    - Direccion destino en registro **OUT DL, AL**
      4. `MAR` ← `DL` – Dirección destino.
      5. `MBR` ← `AL` – Contenido a escribir.
      6. `write(E-S[MAR])` ← `MBR`.
    - Direccion destino inmediato **OUT inmediato, AL**
      4. `MAR` ← `IP` – Obtener dirección del dato inmediato.
      5. `MBR` ← `read(Memoria[MAR])` | `IP` ← `IP + 1` – Leer el dato e incrementar IP.
      6. `MAR` ← `MBR` – Dirección destino.
      7. `MBR` ← `AL` – Contenido a escribir.
      8. `write(E-S[MAR])` ← `MBR`
  - **IN**
    - Direccion destino en registro **IN AL, DL**
      4. `MAR` ← `DL` – Dirección destino.
      5. `MBR` ← `read(E-S[MAR])` – Contenido a leer.
      6. `AL` ← `MBR`
    - Direccion destino inmediato **IN AL, inmediato**
      4. `MAR` ← `IP` – Obtener dirección del dato inmediato.
      5. `MBR` ← `read(Memoria[MAR])` | `IP` ← `IP + 1` – Leer el dato e incrementar IP.
      6. `MAR` ← `MBR` – Dirección destino.
      7. `MBR` ← `read(E-S[MAR])` – Contenido a leer.
      8. `AL` ← `MBR`
- **Pila**
  - **POP Rx**
    4. `MAR` ← `SP`
    5. `MBR` ← `read(Memoria[MAR])`
    6. `Rx` ← `MBR` | `SP` ← `SP` + 1
  - **PUSH Ry**
    4. `MAR` ← `SP`
    5. `MBR` ← `Ry` | `SP` ← `SP` - 1

6. `write(Memoria[MAR])` ← `MBR`

Se implementaron operaciones lógicas con dos operandos AND, OR y XOR que tienen los mismos pasos de ejecución que las operaciones ADD y SUB, donde cambia únicamente la operación que realiza la ALU.

Se agregaron operaciones aritméticas y lógicas de un solo operando: - **INC** `Rx` 4 - `Rx` ← `Rx` + **1 | update(Flags)**

- **DEC** `Rx` 4 - `Rx` ← `Rx` **- 1 | update(Flags)**
- **NOT** `Rx` 4 - `Rx` ← **NOT** `Rx` **| update(Flags)**
- **NEG** `Rx` 4 - `Rx` ← **CA2** `Rx` **| update(Flags)**

El formato ampliado para las instrucciones incluye los siguientes casos tabla 5.17:

Tabla 5.16: Tabla de instrucciones y acciones

Tipo	Ejemplo	Acción
A: Entre registros	MOV Rx, Ry	Rx ← Ry
	ADD Rx, Ry	Rx ← Rx + Ry
	SUB Rx, Ry	Rx ← Rx - Ry
	CMP Rx, Ry	Rx - Ry *
B: Cargar a registro	MOV Rx, [M]	Rx ← Mem[Dirección]
	MOV Rx, [BL]	Rx ← Mem[BL]
	MOV Rx, d	Rx ← dato
	ADD Rx, [M]	Rx ← Rx + Mem[Dirección]
	ADD Rx, [BL]	Rx ← Rx + Mem[BL]
	ADD Rx, d	Rx ← Rx + dato
	SUB Rx, [M]	Rx ← Rx - Mem[Dirección]
	SUB Rx, [BL]	Rx ← Rx - Mem[BL]
	SUB Rx, d	Rx ← Rx - dato
	CMP Rx, [M]	Rx - Mem[Dirección] (*)
	CMP Rx, [BL]	Rx - Mem[BL] (*)
	CMP Rx, d	Rx - dato (*)
C: Almacenar en memoria	MOV [M], Ry	Mem[Dirección] ← Rx
	MOV [BL], Ry	Mem[BL] ← Rx
	MOV [M], d	Mem[Dirección] ← dato
	MOV [BL], d	Mem[BL] ← dato
	ADD [M], Ry	Mem[Dirección] ← Mem[Dirección] + Rx
	ADD [BL], Ry	Mem[BL] ← Mem[BL] + Rx
	ADD [M], d	Mem[Dirección] ← Mem[Dirección] + dato
	ADD [BL], d	Mem[BL] ← Mem[BL] + dato
	SUB [M], Ry	Mem[Dirección] ← Mem[Dirección] - Rx
	SUB [BL], Ry	Mem[BL] ← Mem[BL] - Rx
	SUB [M], d	Mem[Dirección] ← Mem[Dirección] - dato
	SUB [BL], d	Mem[BL] ← Mem[BL] - dato
	CMP [M], Ry	Mem[Dirección] - Rx
	CMP [BL], Ry	Mem[BL] - Rx
	CMP [M], d	Mem[Dirección] - dato
	CMP [BL], d	Mem[BL] - dato
D: control de flujo	JMP M	IP ← Dirección
	Jxx M	Si xx = 1 entonces IP ← Dirección
	HLT	Detiene el procesador

*Nota *:*
La instrucción CMP no almacena el resultado de la operación en el operando destino.

Tabla 5.17: Tabla de codificación de instrucciones ampliado

Tipo	Operación	Codificación	Parámetros
Transferencia y procesamiento	A: entre registros	_____XXYY	XX = Registro destino, YY = Registro fuente
	B: Cargar a registro	_____XX00 MMMMMMMM	XX00 = Registro destino y M = Dirección de memoria
		_____XX01	XX01 = Registro destino y direccion en registro [BL]
		_____XX00 dddddddd	XX10 = Registro destino y d = Dato inmediato
	C: Almacenar en memoria	_____00YY MMMMMMMM dddddddd	YY = Registro fuente, 'M' = Dirección de memoria, 'd' = Dato Inmediato
Control	D: control de flujo	_____ffff MMMMMMMM	ffff = funcionalidad, M = Dirección de memoria

Nota:
d = dato inmediato, no puede ser destino de la instrucción.

Apéndices

5.6 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

---

5.6.1 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.

5.6.2 Datos generales del entrevistado

- **Nombre y Apellido:** (opcional si se desea anonimato)
- **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)

5.6.3 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?

5.6.4 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.

Capítulo 6

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, "PBB-Cache: 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.

[14] F. García-Carballeira, 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. Đordevic, "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.

[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.

[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] 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`

[22] AMD, "Developer guides, manuals & ISA documents." Sep. 2024. Accessed: May 11, 2025. [Online]. Available: `https://www.amd.com/en/search/documentation/hub.html`

[23] P. Abel, *IBM PC Assembly Language and Programming*, 5th ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[24] D. Skrien, "CPU Sim 3.1: A tool for simulating computer architectures for computer organization classes," *Journal on Educational Resources in Computing (JERIC)*, 2001.

[25] J. L. Peterson, *Petri net theory and the modeling of systems*. Prentice Hall PTR, 1981.

[26] 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.

[27] B. P. Zeigler, A. Muzy, and E. Kofman, *Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations*. Academic Press, 2018.

[28] A. S. Tanenbaum, *Structured computer organization*. Pearson Education India, 2016.

[29] M. J. Murdocca and V. Heuring, *Principles of computer architecture*. Pearson Education, 2000.

[30] R. E. Bryant and D. R. O'Hallaron, *Computer systems: A programmer's perspective*. Pearson, 2015.

[31] 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

[32] S. Harris and D. Harris, *Digital design and computer architecture*. Morgan Kaufmann, 2015.

[33] L. Null, *Essentials of computer organization and architecture with navigate advantage access*, 6th ed. Burlington, MA: Jones & Bartlett Learning, 2023.

[34] Patterson *et al.*, *Computer organization and design: The hardware/software interface-5th*. Morgan Kaufmann, 2014.

[35] D. A. Patterson and J. L. Hennessy, *Computer organization and design ARM edition: The hardware software interface*. Morgan kaufmann, 2016.

[36] L. Belli *et al.*, "IoT-enabled smart sustainable cities: Challenges and approaches," *Smart Cities*, vol. 3, no. 3, pp. 1039–1071, 2020.

[37] D. A. Patterson and J. L. Hennessy, *Computer organization and design RISC-v edition: The hardware software interface*. Morgan Kaufmann, 2017.

[38] A. Akram and L. Sawalha, "A survey of computer architecture simulation techniques and tools," *Ieee Access*, vol. 7, pp. 78120–78145, 2019.

[39] M. Menchón, M. Tosini, and O. Goñi, "Herramientas de software educacional para el aprendizaje de arquitectura de procesadores."

[40] J. von Neumann, "First draft of a report on the EDVAC," Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia, PA, Technical Report, 1945.

[41] P. E. Ceruzzi, *A history of modern computing*. MIT press, 2003.

[42] M. R. Williams, *A history of computing technology*. Prentice-Hall Englewood Cliffs, NJ, 1998.

[43] T. Noergaard, *Embedded systems architecture: A comprehensive guide for engineers and programmers*. Newnes, 2012.

[44] Arm Ltd., *Arm architecture reference manual: Armv9-a, for Armv9-a architecture profile*. Arm Ltd., 2021.

[45] Intel Corporation, *Intel® 64 and IA-32 architectures optimization reference manual*, April 2021. Intel Corporation, 2021.

[46] J. L. Hennessy and D. A. Patterson, *Computer organization and design RISC-v edition: The hardware software interface*. Elsevier Science & Technology Books, 2017.

[47] 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

[48] 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

[49] D. A. Patterson and J. L. Hennessy, *Computer organization and design ARM edition: The hardware software interface*. Morgan kaufmann, 2016.

[50] B. B. Brey, *The intel microprocessors: Architecture, programming, and interfacing*, 8th ed. Pearson Education, 2013.

[51] 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

[52] K. R. Irvine and L. B. Das, *Assembly language for x86 processors*. Prentice Hall, 2011.

[53] B. International, *Turbo assembler user's guide*. Borland International, 1993.

[54] M. Corporation, *Microsoft macro assembler 6.1 reference*. Microsoft Press, 1992.

[55] The NASM Project, *The netwide assembler (NASM) manual*. 2023.

[56] R. Hyde, *The art of assembly language*. No Starch Press, 2010.

[57] 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.

[58] F. Jentsch and M. Curtis, *Simulation in aviation training*. Routledge, 2017.

[59] R. M. Fujimoto, "Parallel and distributed simulation systems," in *Proceeding of the 2001 winter simulation conference (cat. No. 01CH37304)*, IEEE, 2001, pp. 147–157.

[60] 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.

[61] 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.

[62] F. A. Calvo Valdes, J. F. Roldan Ramirez, and A. San Miguel Sanchez, "Simulador del procesador MIPS sobre el formalismo DEVS," 2010.

[63] 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

[64] 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

[65] P. BEHROOZ, *Computer Architecture: From Microprocessors to Supercomputers*. Oxford University Press Inc, 2005.

[66] A. Huberman *et al.*, "Qualitative data analysis a methods sourcebook," 2019.

[67] 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.

[68] W3C Web Accessibility Initiative, "Accessibility principles." 2021. Available: https://www.w3.org/WAI/fundamentals/accessibility-principles/

[69] 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.

[70] 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.

[71] J. Sweller, P. Ayres, and S. Kalyuga, *Cognitive load theory*. New York: Springer, 2010. doi: 10.1007/978-1-4419-8126-3.

[72] 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.

[73] C. C. Bonwell and J. A. Eison, "Active learning: Creating excitement in the classroom," *ASHE-ERIC Higher Education Report*, vol. 1, 1991.

[74] 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.

[75] Z. Majid, "Design of SAP-1 controller and simulation using mentor graphics," PhD thesis, Universiti Teknologi MARA (UiTM), 1999.

[76] A. Morlan, "Building a simple 8-bit CPU in verilog (SAP-1)." 2021. Available: https://austinmorlan.com/posts/fpga_computer_sap1/

[77] 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

[78] D. Silber, "TinyCPU - a simple CPU implemented in verilog." https://www.eecis.udel.edu/~silber/tinycpu/#/home.

[79] J. M. S. Facundo Quiroga Manuel Bustos Berrondo, "VonSim - simulador de arquitectura de von neumann." https://vonsim.github.io/, 2020.

[80] Open Source Initiative, "Open source licenses: Understanding GPL, MIT, and Apache licenses." 2024. Accessed: Jan. 15, 2024. [Online]. Available: https://opensource.org/licenses/

[81] M. M. Mano and M. D. Ciletti, *Digital design: With an introduction to the verilog HDL, VHDL, and SystemVerilog*. Pearson, 2017.

[82] 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.