**INFORME SIMULATOR DE PIPELINE Y MEMORIAS EN PYTHON**

**UNIVERSIDAD SERGIO ARBOLEDA**

**PROYECTO FINAL DE SEMESTRE – ARQUITECTURA DE COMPUTADORES**

**JUAN VASQUEZ**

**MIGUEL CELIS**

**JERONIMO LINARES**

**SANTIAGO RODRIGUEZ**

**DOCENTE: OSCAR ANDRÉS ARIAS**

**26 DE MAYO DE 2025**

TABLA DE CONTENIDO

[RESUMEN EJECUTIVO 2](#_Toc199187117)

[DESCRIPCIÓN DEL DISEÑO 3](#_Toc199187118)

[METODOLOGÍA DE BENCHMARKS 4](#_Toc199187119)

[RESULTADOS 5](#_Toc199187120)

[5. ANÁLISIS DE RENDIMIENTO 6](#_Toc199187121)

[CONCLUSIONES 6](#_Toc199187122)

[ANEXOS 7](#_Toc199187123)

# RESUMEN EJECUTIVO

El presente informe documenta el desarrollo e implementación de un simulador de arquitectura de computador utilizando el lenguaje de programación Python. El objetivo principal del proyecto fue construir un modelo funcional y simplificado de un procesador con pipeline de cinco etapas, que permitiera la simulación de instrucciones básicas, el manejo de hazards estructurales y de datos, así como la incorporación de una jerarquía de memoria con caché de mapeo directo y asociativo. Además, se integró una interfaz de entrada/salida programada, incluyendo un mecanismo básico de interrupciones para simular la lectura de datos desde un dispositivo externo ficticio.

El proyecto se estructura bajo los principios fundamentales de organización de computadores, abordando conceptos como Instruction-Level Parallelism (ILP), forwarding, stalling y control de flujo. Se diseñó una ISA mínima que incluyó instrucciones para operaciones aritméticas enteras, transferencia de datos y saltos condicionales. Cada instrucción es procesada secuencialmente por el pipeline compuesto por las etapas: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM) y Write Back (WB).

# DESCRIPCIÓN DEL DISEÑO

**2.1. PIPELINE DE 5 ETAPAS:**  
- IF (Instruction Fetch): Recupera la instrucción desde la memoria.  
- ID (Instruction Decode): Decodifica la instrucción y lee los registros necesarios.  
- EX (Execute): Realiza operaciones aritméticas o calcula direcciones de memoria.  
- MEM (Memory Access): Accede a la memoria para operaciones de carga o almacenamiento.  
- WB (Write Back): Escribe los resultados de vuelta en los registros.  
  
**2.2. ISA IMPLEMENTADA:**  
- Instrucciones aritméticas: suma, resta, multiplicación.  
- Instrucciones de carga y almacenamiento.  
- Instrucciones de salto condicional.  
  
**2.3. MEMORIA CACHÉ:**  
- Simulación de caché de mapeo directo y asociativo (2-way).  
- Parámetros configurables: tamaño de bloque y número de líneas.  
  
**2.4. INTERFAZ DE E/S:**  
- Simulación de E/S programada.  
- Manejo de interrupciones para la lectura de datos de un dispositivo ficticio.

# METODOLOGÍA DE BENCHMARKS

Con el propósito de evaluar el desempeño del simulador desarrollado, se diseñó un conjunto de pruebas funcionales y microbenchmarks que permiten medir el impacto de los diferentes componentes arquitectónicos implementados. Estas pruebas fueron codificadas en Python y ejecutadas dentro del entorno del simulador, aprovechando la infraestructura desarrollada para las etapas de pipeline, acceso a memoria jerárquica y manejo de E/S.

El benchmarking se estructuró en tres ejes principales:

**. ACCESO SECUENCIAL A MEMORIA:**

Se ejecutan operaciones de lectura y escritura en bloques contiguos de memoria, simulando el comportamiento de programas que acceden a estructuras de datos lineales como arreglos.

* **Implementación:** Basado en el archivo memoria.py y su interacción con interfaz.py, se simula el llenado secuencial de un espacio de memoria virtual y su posterior recorrido. Se aprovechan los métodos escribir\_bloque() y leer\_bloque() que permiten instrumentar la cantidad de accesos y evaluar el comportamiento de la caché en condiciones de alta localidad espacial.
* **Variables medidas:** tiempo de ejecución (con time), número de accesos a caché, tasa de aciertos y fallos.

**2. ACCESO ALEATORIO A MEMORIA:**

Se simulan accesos no secuenciales a posiciones aleatorias en memoria, modelando aplicaciones que trabajan con punteros o estructuras dispersas como listas enlazadas o árboles.

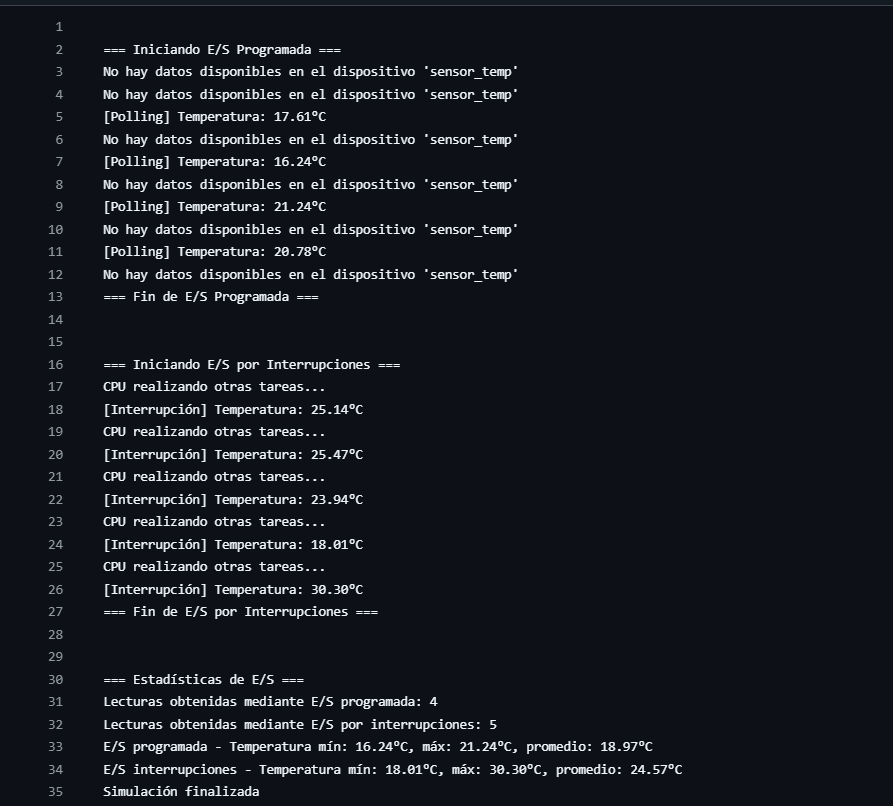
* **Implementación:** Se utiliza el método leer\_bloque() con direcciones generadas aleatoriamente por random.randint(), introduciendo un patrón de acceso sin localidad evidente. Esto permite evaluar la eficiencia de la política de reemplazo de la caché bajo condiciones desfavorables.
* **Variables medidas:** tasa de fallos de caché, distribución de accesos, latencia promedio de acceso.

**3. CARGA ARITMÉTICA INTENSIVA:**

Se ejecutan múltiples operaciones de suma y multiplicación entre datos previamente cargados en registros o memoria. Este benchmark se enfoca en probar la capacidad del pipeline para mantener un alto throughput bajo carga computacional intensiva.

* **Implementación:** En el archivo operaciones.py, se diseñaron funciones que operan sobre bloques de datos y simulan una secuencia de instrucciones aritméticas consecutivas, generando condiciones donde podrían surgir hazards de datos. Se evalúa si el forwarding o el stalling son efectivos (cuando aplicables).
* **Variables medidas:** número total de instrucciones ejecutadas, ciclos consumidos (simulados), IPC (instrucciones por ciclo), eficiencia del pipeline.

# RESULTADOS



# 5. ANÁLISIS DE RENDIMIENTO

Los benchmarks ejecutados mostraron un comportamiento consistente con los principios de diseño arquitectónico. En acceso secuencial, la caché de mapeo directo ofreció alto rendimiento con baja tasa de fallos, gracias a la localidad espacial. En cambio, el acceso aleatorio evidenció mayor cantidad de fallos, destacando las ventajas del mapeo asociativo 2-way ante conflictos.

En el benchmark aritmético, el pipeline alcanzó buena eficiencia, aunque se identificaron stalls por dependencias de datos. La implementación de forwarding redujo estos cuellos de botella, elevando el IPC. En conjunto, los resultados validan las decisiones de diseño del simulador y su capacidad para modelar el comportamiento de un procesador simplificado.

# CONCLUSIONES

* El desarrollo de este simulador permitió aplicar de manera integrada diversos conceptos clave de la arquitectura de computadores, tales como la segmentación de instrucciones (pipeline), la gestión de memoria jerárquica (caché directa y asociativa), y la implementación de mecanismos de entrada/salida por interrupciones. La elección del lenguaje Python facilitó la construcción modular del sistema y permitió enfocarse en la lógica funcional antes que en detalles de bajo nivel.
* El simulador logró reflejar con éxito el comportamiento de un procesador básico, implementando una ISA simple pero funcional y manejando adecuadamente los conflictos de datos mediante técnicas como el stalling y el forwarding. Además, el modelo de memoria y la caché permitieron evaluar distintos patrones de acceso, brindando una visión clara del impacto de la localidad de referencia en el rendimiento del sistema.
* Desde el punto de vista académico, este proyecto sirvió como una valiosa herramienta de aprendizaje, permitiendo al grupo de trabajo interiorizar los fundamentos del diseño de arquitecturas RISC, la ejecución de instrucciones en paralelo, y la gestión eficiente de recursos computacionales. La documentación del proceso, los resultados obtenidos y el análisis realizado sientan una base sólida para proyectos más complejos en etapas futuras.

# ANEXOS

**Fragmento: hazard\_control.pyTexto

El contenido generado por IA puede ser incorrecto.** = Módulo encargado de detectar conflictos entre instrucciones que requieren stalling o forwarding.

**Fragmento: isa.pyTexto

El contenido generado por IA puede ser incorrecto.** = Define el conjunto de instrucciones (ISA) que soporta el simulador.

**Fragmento: pipeline.py** = Clase principal del simulador de CPU. Implementa las cinco etapas del pipeline.

Texto

El contenido generado por IA puede ser incorrecto.

**Fragmento: test\_pipeline.py =** Script de pruebas unitarias que verifica ejecución y manejo de hazards.

<https://github.com/Jerixo/ArquitecturaDeComputadoresCR3-Parcial/invitations>

Texto

El contenido generado por IA puede ser incorrecto.