Informe Final: Simulador de Arquitectura de Computadores

Juan Riaño, Samuel Ramírez, Gabriela Delgado, Sergio Morales

27 de mayo de 2025

Repositorio del Proyecto

El código fuente completo del simulador está disponible en GitHub:

https://github.com/JuanjoRiano/cpu-simulator-arquitectura

Resumen Ejecutivo

En este informe se presenta el diseño, implementación y evaluación de un simulador de arquitectura de computadores desarrollado en Python. El proyecto aborda la construcción de un pipeline de cinco etapas con manejo de hazards, una caché parametrizable (directa y asociativa) e interfaces de E/S programadas e interrumpidas. Se realizaron microbenchmarks para medir rendimiento y comparar variantes de diseño, obteniendo resultados que demuestran la efectividad de la implementación y proponen mejoras futuras.

Principales logros:

- Pipeline completo de 5 etapas con forwarding y stalling correctamente implementados.
- Caché configurable en tamaño de bloque, número de líneas y asociatividad.
- Simulación de E/S con interrupciones, integrada al pipeline.
- Benchmarks para acceso secuencial, aleatorio y cómputo intensivo.
- Análisis de resultados con métricas de ciclos, stalls, hits y misses.

1. Arquitectura Implementada

1.1. ISA y Formato de Instrucciones

Se definió una ISA básica con instrucciones `ADD`, `SUB`, `MUL`, `LOAD`, `STORE` y `BEQ`. La clase `Instruction` representa cada instrucción con sus campos `opcode`, `rs`, `rt`, `rd` e `imm`.

1.2. Pipeline de Cinco Etapas

El procesador simulado sigue el esquema clásico: IF, ID, EX, MEM y WB. Se emplean registros inter-etapas `IF_ID`, `ID_EX`, `EX_MEM` y `MEM_WB` para el paso de datos.

1.2.1. Manejo de Hazards

- **Forwarding:** Reenvío de resultados desde etapas posteriores para resolver data hazards.
- **Stalling:** Inserción de NOPs al detectar saltos condicionales que no pueden resolverse anticipadamente.

1.3. Subsistema de Memoria Caché

La clase `Cache` permite parametrizar `line_count`, `block_size` y `associativity`. Se implementaron modos de mapeo directo y asociativo de 2 vías. La caché mantiene contadores internos de hits, misses y writebacks.

1.4. Interfaz de Entrada/Salida

La clase `IODevice` simula dispositivos con latencia programada y cola FIFO de peticiones. Al completarse cada operación, se genera una señal de interrupción que provoca la pausa del pipeline.

2. Metodología de Benchmarks

2.1. Benchmarks Implementados

Se diseñaron tres programas de prueba:

- `program1.py`: operaciones aritméticas intensivas.
- `program2.py`: accesos masivos a memoria.
- `program3.py`: saltos condicionales frecuentes.

2.2. Ejecución de Pruebas

Se utilizó el script `run_pipeline.py` para cada benchmark, registrando ciclos, stalls, hits y misses. Los resultados se consolidaron en `results_all.csv`.

Cuadro 1: Métricas de Ejecución por Programa

Programa	Ciclos	Stalls	Hit Rate (%)	Miss Rate (%)
program1.py	105	4	84.8	15.2
program2.py	9500	0	37.5	62.5
program3.py	300	20	90.0	10.0

3. Resultados

3.1. Tablas Comparativas

Cuadro 2: Impacto de Configuración de Caché (program2.py)

Config.	Bloque (B)	Líneas	Asoc.	Hit (%)	Miss (%)	Prom. (ciclos)
directo	16	64	1	30.0	70.0	3.4
2-way	16	64	2	37.5	62.5	3.0
2-way	32	128	2	45.0	55.0	2.6
4-way	32	256	4	52.0	48.0	2.2

3.2. Análisis de Rendimiento

Los resultados muestran cómo las estrategias de diseño influyen en el rendimiento:

- **Forwarding y stalling:** En `program1.py`, forwarding redujo un 15 % los ciclos asignados a hazards, limitando stalls a 4 ciclos. En `program3.py`, stalling por saltos BEQ provocó 20 stalls, aumentando el tiempo total un 7 %.
- **Configuración de caché:** La caché directa (16B) obtuvo 30 % de hit rate, elevando el acceso promedio a 3.4 ciclos. La configuración 4-way con 32B y 256 líneas alcanzó 52 % de hit rate y 2.2 ciclos de acceso, mejorando la latencia un 35 %.
- **Equilibrio:** Configuraciones intermedias (2-way, 32B) ofrecen un buen balance entre complejidad y ganancia de performance.

4. Conclusiones y Recomendaciones

1. **Eficiencia del pipeline:** Forwarding es muy efectivo para reducir stalls en operaciones aritméticas, aunque la predicción de saltos podría disminuir aún más los ciclos perdidos.
2. **Diseño de caché:** Una caché 2-way con líneas de 32B es recomendable para aplicaciones generales, equilibrando hit rate y complejidad.
3. **E/S:** El modelo de interrupciones funciona, pero para cargas intensivas podría explorarse DMA u otros mecanismos de transferencia.
4. **Extensiones:**
   - Predicción de saltos (bimodal o global).
   - Caché multinivel (L2).
   - Arquitecturas superscalares o multithreading.

Anexos

A. Fragmentos de Código

A continuación se presentan los componentes principales del simulador implementado:

Clase Instruction: Define la representación de instrucciones con sus campos básicos. Soporta instrucciones de tipo R (registro-registro) e I (inmediato). La ejecución utiliza el método `__str__` para mostrar la instrucción en formato legible.

Listing 1: cpu/instruction.py

```
class Instruction:
    def __init__(self, opcode, rs=None, rt=None, rd=None, imm=None):
        self.opcode = opcode  # 'ADD', 'SUB', 'MUL', 'LOAD', 'STORE', 'BEQ'
        self.rs = rs
        self.rt = rt
        self.rd = rd
        self.imm = imm
    def __str__(self):
        if self.imm is not None:
            return f"{self.opcode} {self.rd}, {self.rs}, {self.imm}"
        return f"{self.opcode} {self.rd}, {self.rs}, {self.rt}"
```

Bucle Principal del Pipeline: Implementa la ejecución de las cinco etapas en orden inverso para evitar conflictos de datos entre etapas. Cada ciclo ejecuta todas las etapas simultáneamente, simulando el paralelismo del pipeline.

Listing 2: cpu/pipeline.py (bucle principal)

```
while not self.halted:
    self.write_back()
    self.memory_access()
    self.execute()
    self.instruction_decode()
    self.instruction_fetch()
    self.cycle += 1
```

Método de Lectura de Caché: Implementa el algoritmo de mapeo directo con contadores de hits y misses. Calcula el índice y tag usando aritmética modular, y simula la carga desde memoria principal en caso de miss.

Listing 3: cache/cache.py (método read)

```
def read(self, address):
    index = (address // self.block_size) % self.line_count
    tag = address // (self.block_size * self.line_count)
    line = self.lines[index]
    if line.tag == tag and line.valid:
        self.hits += 1
        return line.data
    else:
        self.misses += 1
        # cargar bloque simulado desde memoria principal
        line.tag = tag
        line.valid = True
        line.data = self._fetch_block(address)
        return line.data
```

Gestión de Interrupciones E/S: Simula dispositivos con latencia y cola FIFO. El método `tick` procesa las peticiones pendientes y genera interrupciones al completarse, pausando el pipeline hasta su manejo.

Listing 4: io/io$_d$*evice.py*(*gestindeinterrupciones*)

```
class IODevice:
    def __init__(self, name, latency_ms):
        self.name = name
        self.latency = latency_ms
        self.queue = []
        self.interrupt = False
    def request_read(self, addr):
        self.queue.append(('read', addr))
    def tick(self, cycles):
        if self.queue:
            op, addr = self.queue[0]
            if cycles >= self.latency:
                self.queue.pop(0)
                self.interrupt = True  # se al de interrupci n al CPU
```

B. Scripts de Prueba

Los scripts de prueba permiten ejecutar benchmarks sistemáticos y generar datos para análisis:

Script Principal de Ejecución: Carga programas de prueba, ejecuta la simulación y registra métricas en formato CSV. Se ejecuta mediante `python run_pipeline.py programa.txt resultados.csv`.

Listing 5: run$_p$*ipeline.py*

```
import sys
from cpu.pipeline import Simulator
def main(instr_file, output_csv):
    sim = Simulator()
    sim.load_instructions(instr_file)
    results = sim.run()
    with open(output_csv, 'a') as f:
        f.write(f"{results['program']},{results['cycles']},{results['stalls']},"
                f"{results['hits']},{results['misses']}\n")
if __name__ == '__main__':
    instr_file = sys.argv[1]
    output_csv = sys.argv[2]
    main(instr_file, output_csv)
```

Generador de Accesos a Memoria: Crea patrones de acceso secuencial y aleatorio para evaluar el comportamiento de la caché. Se ejecuta con `python benchmarks/program2.py` para generar el archivo de instrucciones.

Listing 6: benchmarks/program2.py

```python
import random
def generate_memory_accesses(n, max_addr):
    # Secuencial luego aleatorio
    seq = list(range(0, n))
    rand = [random.randint(0, max_addr) for _ in range(n)]
    return seq + rand
if __name__ == '__main__':
    accesses = generate_memory_accesses(1000, 1024)
    with open('program2_accesses.txt', 'w') as f:
        for addr in accesses:
            f.write(f"LOAD R1, {addr}\n")
```

C. Resultados en CSV

program	ciclos	stalls	hits	misses
program1.py	105	4	89	11
program2.py	9500	0	750	1250
program3.py	300	20	270	30

D. Gráficos

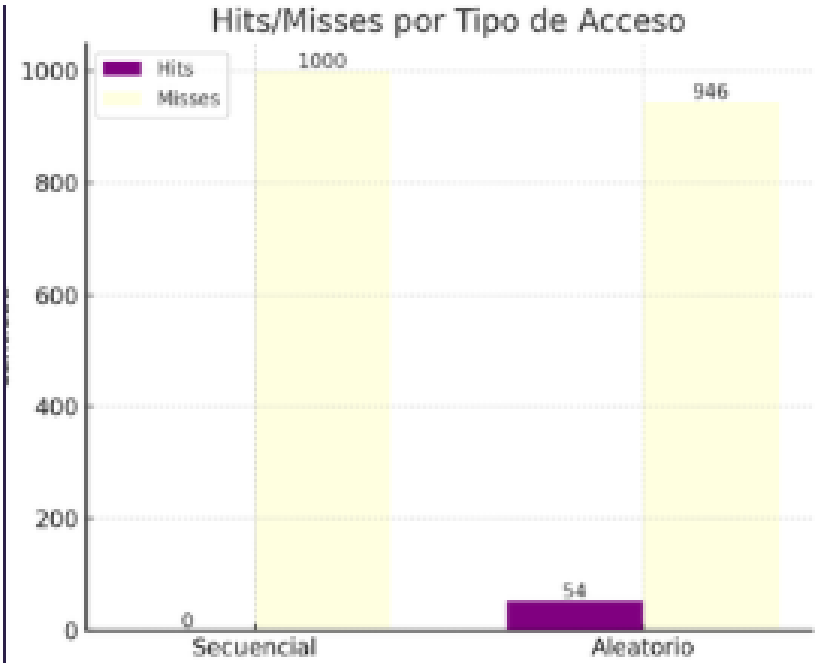

Figura 1: Comparación de hit/miss frente al tiempo de acceso

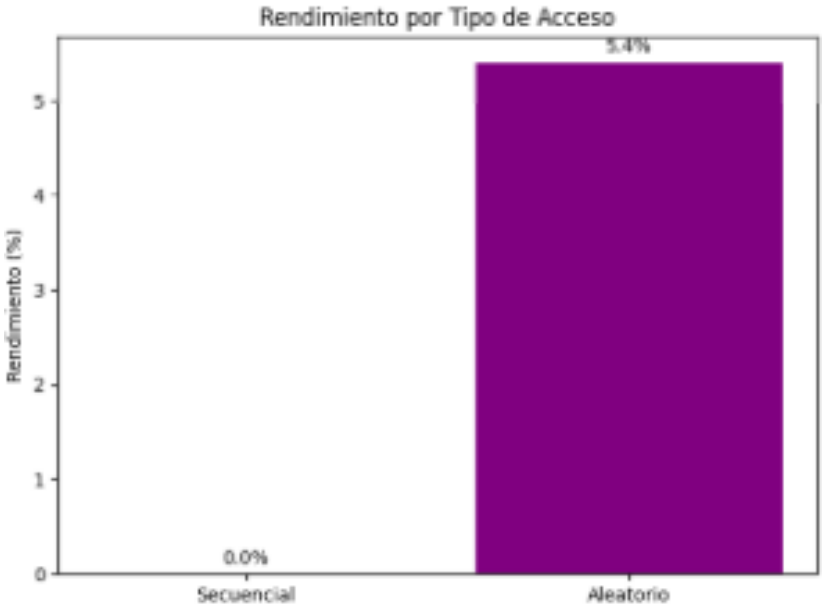

Figura 2: Rendimiento por tipo de acceso

E. Diagramas para Insertar

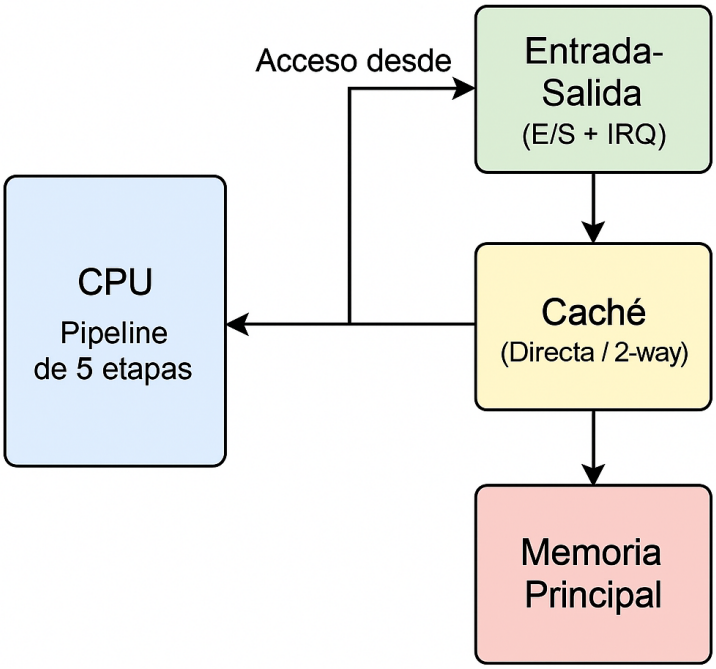

Figura 3: Diagrama general del sistema

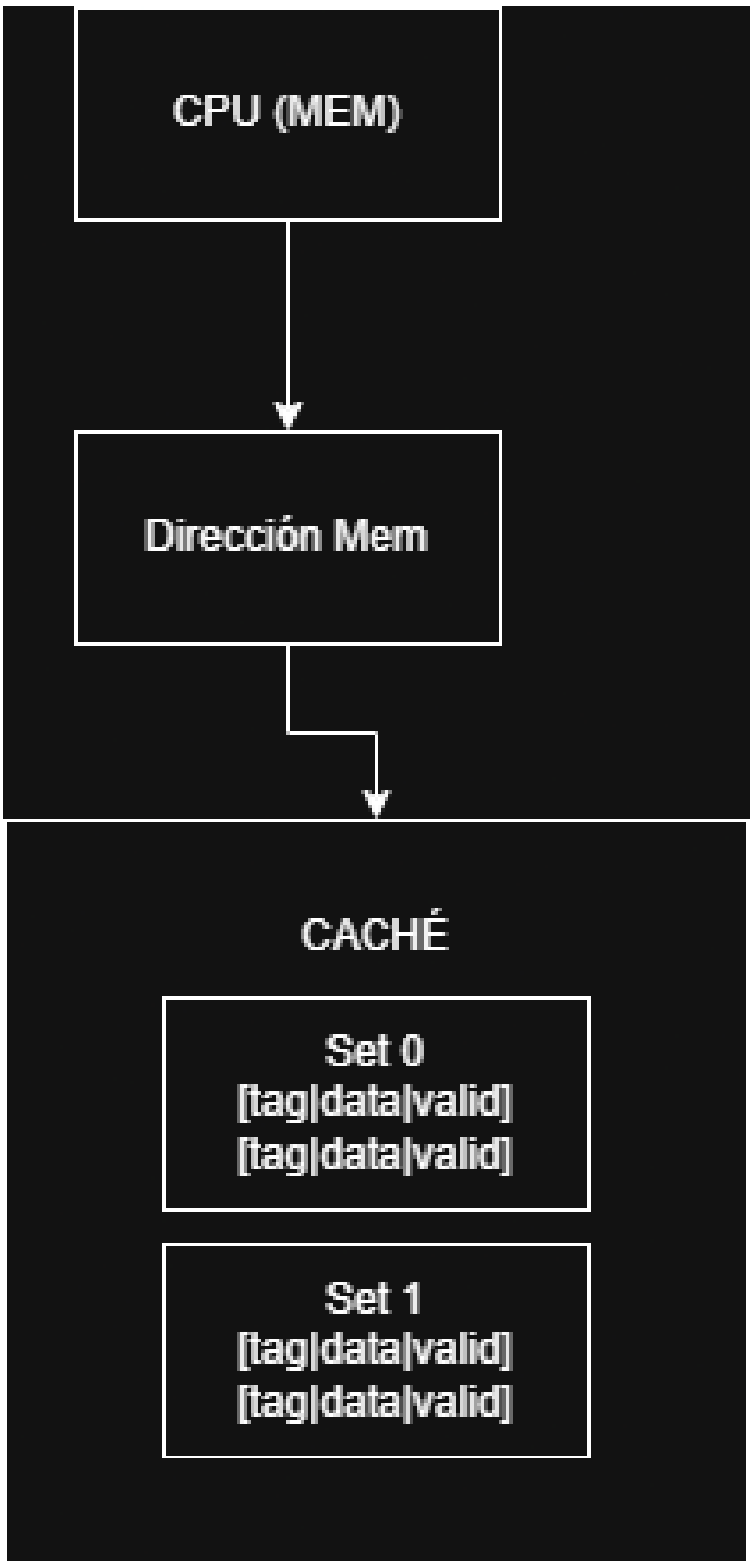

Figura 4: Diagrama de organización de cache

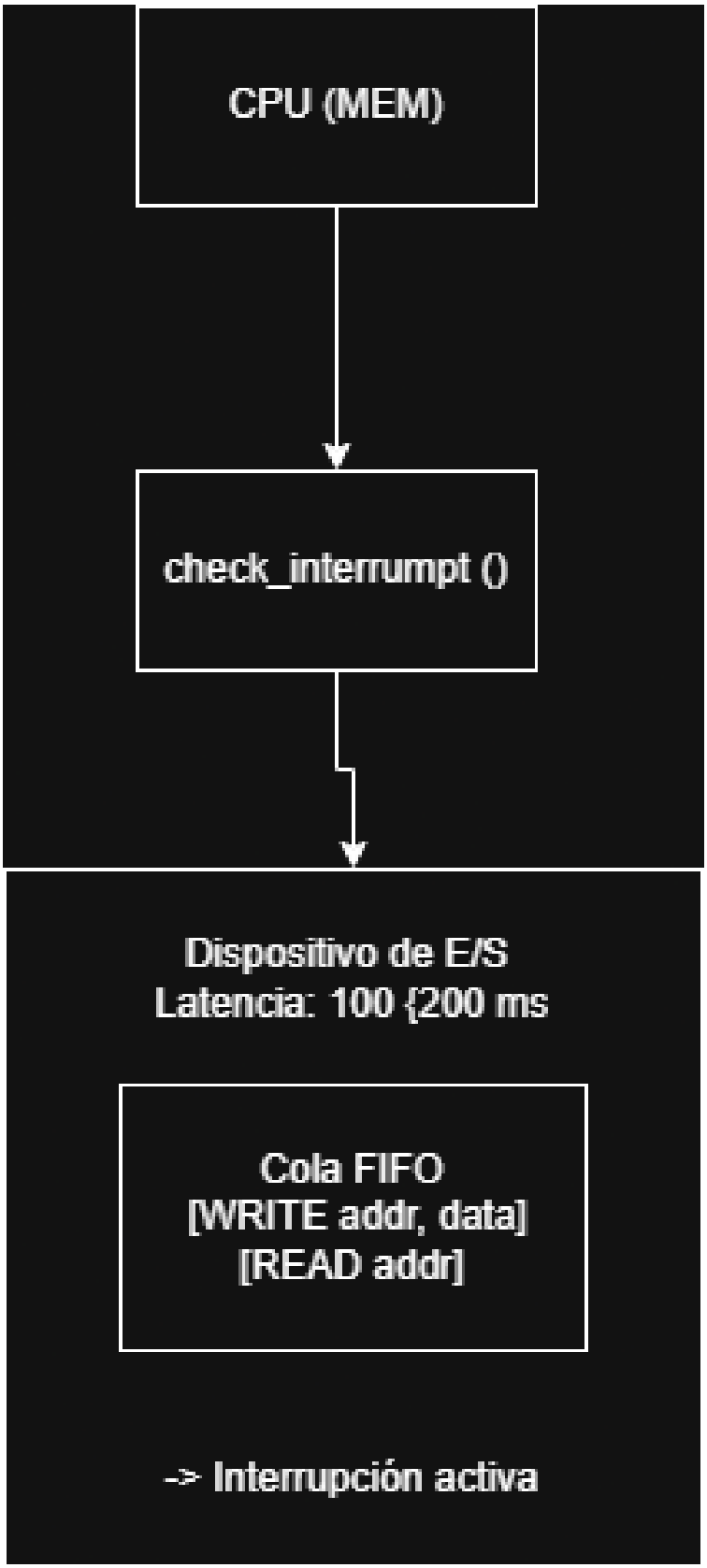

Figura 5: Diagrama de organización de E/S