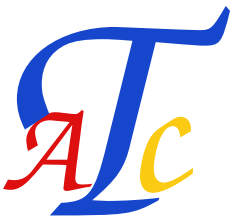

# Universidad de Córdoba

Arquitectura y Tecnología de Computadores

# Práctica 6

**Riesgos de datos en procesadores actuales**

Arquitecturas Avanzadas de Procesadores

D. Miguel Ángel Montijano Vizcaíno (el1movim@uco.es)
D. Héctor Martínez Pérez (el2mapeh@uco.es)

Curso 2025/2026

# 1 Introducción

Esta última práctica de la asignatura se centrará en dar un enfoque de actualidad a los conceptos aprendidos hasta el momento relacionados con el desarrollo y optimización de código ensamblador. En la práctica 3, ya se vio la importancia en la actualidad de desarrollar código ensamblador teniendo en cuenta la arquitectura del procesador. En cambio, esta última práctica se centrará en la optimización de estos códigos desarrollados aplicando los conocimientos adquiridos hasta la fecha.

Los objetivos principales de esta práctica son:

- Tener una visión actual de los riesgos de datos en arquitecturas modernas y el impacto que suponen.
- Aprender nuevas técnicas para solventar estos riesgos de datos para desarrollar códigos en ensamblador optimizados.

Como notación para esta práctica, pese a que el desarrollo de los códigos se lleve a cabo en MARS o MIPSim (las herramientas que se han estudiado en las prácticas anteriores), esto es aplicable para cualquier otro tipo de ISA y/o arquitectura.

# 2 El compilador. ¿Amigo o enemigo?

Durante la práctica 3, ya se adelantaron los problemas que pueden surgir a la hora de compilar códigos C/C++, etc. Pero antes de entrar en más detalle, hay varios factores fundamentales que el desarrollador debe tener en cuenta. En primer lugar, la gran mayoría de las arquitecturas modernas dispone de altas tecnologías para la anticipación de datos, como el uso de técnicas de *prefetching*, tanto para asegurar el correcto manejo de los datos como para la aceleración del procesamiento. Dicho esto, en un desarrollo de un código ensamblador en una arquitectura actual es muy extraño que el usuario tenga la necesidad de usar instrucciones *NOP*. En cambio, conforme el desarrollador se acerca más a la arquitectura y se usan técnicas avanzadas de optimización, ciertas partes que no se consideraban un cuello de botella empiezan a serlo.

A la hora de generar un código ensamblador a partir de un determinado código C/C++, el usuario compila generalmente su código implementado con algún compilador conocido como: gcc, g++. icc, etc. Al compilar nuestro código implementado, el usuario pierde completamente todo el control de las instrucciones ensamblador que se están generando, su orden, los registros usados, etc. Es natural que en aplicaciones grandes, una parte se implemente en un lenguaje más abstracto, pero esto no quita que aquellas partes donde el rendimiento sea un punto crucial, se desarrollen en ensamblador, como ya se comentó en la práctica 3.

Uno de los principales puntos a tener en cuenta es el número de registros del que dispone el procesador. Esto es un punto fundamental para un desarrollo optimizado. Recordemos que cuanto más cercano está el dato al núcleo del procesador, antes estará disponible. La gran mayoría de procesadores actuales disponen de un cierto número de registros especiales donde el procesador es capaz de llevar a cabo operaciones SIMD (Simple Instruction Multiple Data). Esto significa que el procesador será capaz de ejecutar

una única instrucción sobre múltiples datos en un único ciclo de reloj (vuestros propios PCs, seguro que tienen disponibles las extensiones SSE, AVX, AVX2, etc.).

Dicho esto, cuando nosotros desarrollamos en C/C++ usando este tipo de técnicas avanzadas de aceleración, el compilador no lleva a cabo una optimización del uso de registros del procesador utilizados; esto conlleva que nuestro código pueda ser más lento de lo esperado. Además, como se ha comentado anteriormente, al acelerar el cómputo con el uso de este tipo de técnicas, el desarrollador se encuentra con una nueva problemática: el rendimiento de su código pasa de estar limitado por el cómputo (*compute-bound*) a estarlo por los accesos a memoria (*memory-bound*). Esto implica la necesidad de acelerar la carga de datos desde memoria a registro.

# Ejercicio 1 - Desenrollado de bucles

Después de esta visión global, empecemos con una optimización básica para el ahorro de ciclos de instrucción, gracias al desenrollado de bucles. Mirando el Código 1 escrito en C, aplicando un desenrollado de bucle de factor 4, quería como en el Código 2.

Código 1: Código ejercicio 1.

```
#define MAXN 8

int main() {
  int i       = 0;
  int total   = 0;
  int A[MAXN] = {0, 1, 2, 3, 4, 5, 6, 7};
  int pointer = 0;
  for (; i < MAXN; i++) {
    total += A[pointer];
    pointer++;
  }
}
```

Código 2: Código ejercicio 2.

```
#define MAXN 8

int main() {
  int i       = 0;
  int total   = 0;
  int A[MAXN] = {0, 1, 2, 3, 4, 5, 6, 7};
  int pointer = 0;
  for (i = 0; i < MAXN - 3; i += 4) {
    total += A[pointer]; pointer++;
    total += A[pointer]; pointer++;
    total += A[pointer]; pointer++;
    total += A[pointer]; pointer++;
  }
}
```

¿Qué conseguimos con esta técnica? Que el número de ciclos por instrucción se reduzca.

**1.1** En primer lugar, implemente el Código 1 en lenguaje ensamblador MIPS, seguidamente, a dicho código aplique el desenrollado de bucle x4 (tal y como se muestra en el Código 2).

**1.2** Una vez implementados los códigos, calcule los ciclos por instrucción que generaría el primer código y el segundo.

# Ejercicio 2 - Reordenamiento del código

Este tipo de optimización se vio en la práctica anterior como método para intercalar instrucciones entre dependencias de datos, y así, poder disponer del dato correcto en una determinada etapa del procesador. Como se ha comentado, en los procesadores actuales, sería muy extraño no disponer de los datos correctos; no obstante, hay retrasos, esperas y esto se traduce en un código menos óptimo.

**2.1** Dicho esto, reordene el Código 3 para que las esperas entre las distintas dependencias de datos existentes sean las mínimas.

Código 3: Código ejercicio 2.

```
1  .data
2  A:     .word 2, 2
3  .text
4  main:
5    li    $t0, 0
6    lw    $t1, A($t0)
7    addi  $t0, $t0, 4
8    lw    $t2, A($t0)
9    add   $t3, $t2, $t1
10   addi  $t0, $t0, 4
11   addi  $t3, $t3, 2
12   sw    $t3, A($t0)
```

# Ejercicio 3 - Software Pipelining

Como última técnica de optimización, se tratará la técnica del adelantamiento de datos. Similar a la técnica del *prefetching*, donde, ya sea bien por software o por hardware, se traen a memoria caché los futuros datos que serán leídos desde la memoria principal. Cuando desarrollamos códigos en ensamblador, existe una técnica similar a nivel de registros. Pero para poder aplicarla adecuadamente, un punto muy importante a tener en cuenta es conocer la cantidad de registros que dispone la arquitectura para la cual estamos desarrollando. Esto es de extrema importancia ya que, en primer lugar necesitaremos ciertos registros para llevar a cabo las operaciones indispensables, y con los registros restante, vamos a poder aprovecharlos para llevar a cabo el adelantamiento de datos.

**3.1.** En modo de ejemplo, en MIPS disponemos de 32 registros de carácter general (aunque algunos de ellos están reservados para el procesador y no pueden ser utilizados).

Para hacer una similitud con los registros vectoriales, supongamos que los registros *$t0 - $t9* van a funcionar al doble de velocidad que el resto (simulando que son registros vectoriales como se acaba de comentar). Dicho esto, aplique en un primer lugar un desenrollado de bucle x2 al Código 4. Seguidamente, haga una precarga fuera del bucle de las direcciones de memoria $t3 y $t4. Al inicio del bucle, se cargarán las direcciones de la siguiente iteración, llevando a cabo a continuación las operaciones pertinentes con los registros precargados. Al finalizar el bucle, se llevará a cabo la precarga de la siguiente iteración futura.

Código 4: Código ejercicio 3.

```
.data
VecA:   .word 1, 2, 3, 4, 5, 6, 7
VecB:   .word 1, 2, 3, 4, 5, 6, 7
N:      .word 8
beta:   .word 1

.text
main:
        li   $t0, 0
        li   $t1, 0
        li   $t2, 0
        la   $t3, VecA
        la   $t4, VecB
        lw   $t5, N
        lw   $t6, beta

loop:   lw   $t0, 0($t3)
        lw   $t1, 0($t4)
        add  $t2, $t0, $t1
        add  $t2, $t2, $t6
        addi $t3, $t3, 4
        addi $t4, $t4, 4
        addi $t0, $t0, 1
        bne  $t0, $t5, loop
```

**3.2.** Al código final generado, aplique todas las optimizaciones y técnicas aprendidas hasta el momento para que se complete en el menor número de ciclos por instrucción posible, además de aprovechar al máximo los registros de doble velocidad.