Universidad Nacional de Entre Ríos

Tecnicatura universitaria en desarrollo web

Segmentación de instrucciones

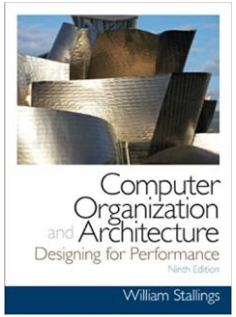
Semana 8 – Arquitectura de computadoras

Docente: José Ruiz {jose.ruiz@uner.edu.ar}

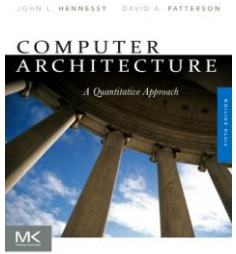
UNER virtual

Esta presentación esta basada en el libro de:

- William Stallings, Computer Organization and Architecture, 9th Edition, 2017.
  - Capítulo 16: “INSTRUCTION-LEVEL PARALLELISM”



- Hennessy-Patterson - Computer Architecture A Quantitative Approach (5th edition)
  - Capítulo 3: “ILP Instruction-Level Parallelism”

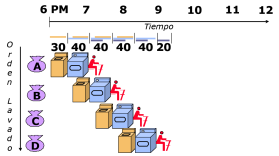


Archivos presentación y ejemplos se alojan en:

https://github.com/ruiz-jose/tudw-arq.git

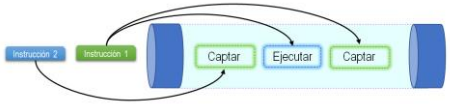
Agenda

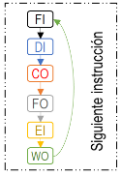
- Técnica de segmentación



- Ciclo de instrucción

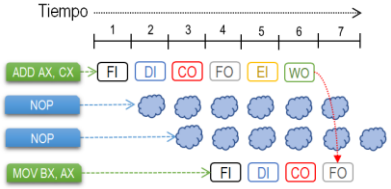
- Segmentación de instrucciones





  - Riesgos

  - Posibles soluciones



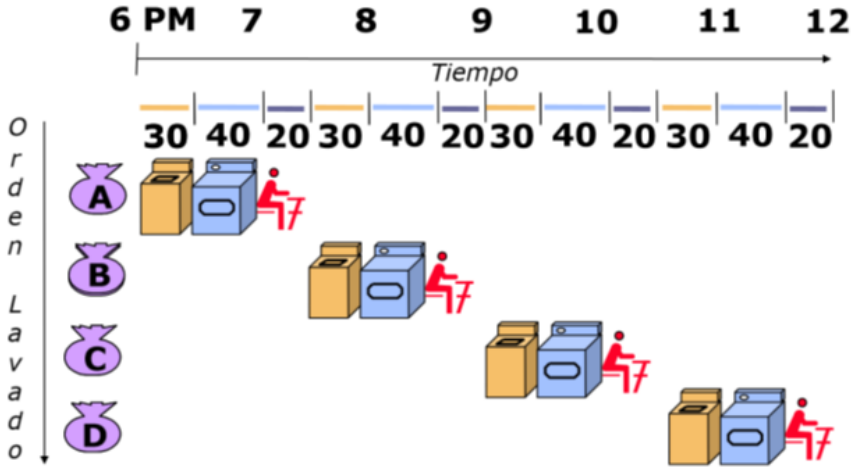
Objetivo

Comprender los conceptos de segmentación de instrucciones y los principales factores que limitan su rendimiento.

Técnica de segmentación

La técnica de segmentación se utiliza en la cadena de montaje en una fábrica.
Por ejemplo una lavandería.

Lavandería secuencial

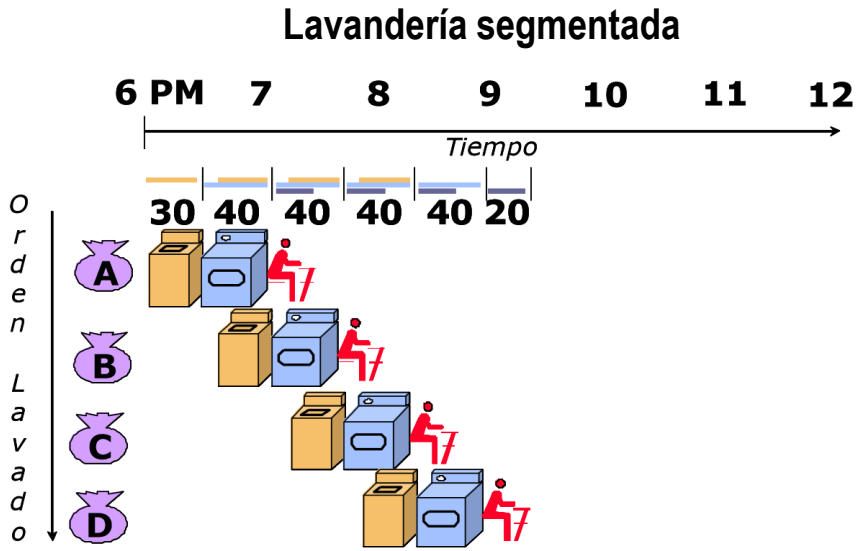


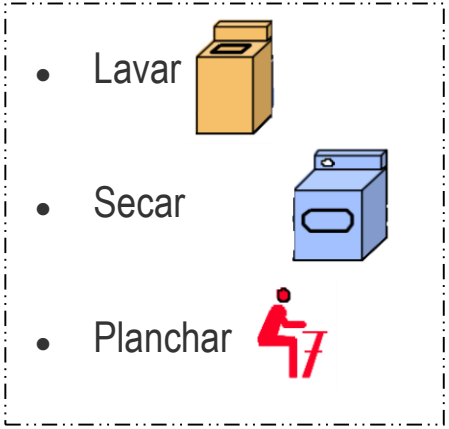
- Lavar
- Secar
- Planchar

90 minutos cada carga, total (4 cargas): 6 horas.

Fig. David A. Patterson, John L. Hennessy . Estructura y Diseño de Computadores

En el proceso de lavado de la lavandería se puede trabajar sobre cada carga en varias etapas simultáneamente.





Total (4 cargas): 3,5 horas.

Fig. David A. Patterson, John L. Hennessy . Estructura y Diseño de Computadores

En la lavandería:
¿Se reduce el tiempo de lavado de cada carga?

Técnica de segmentación

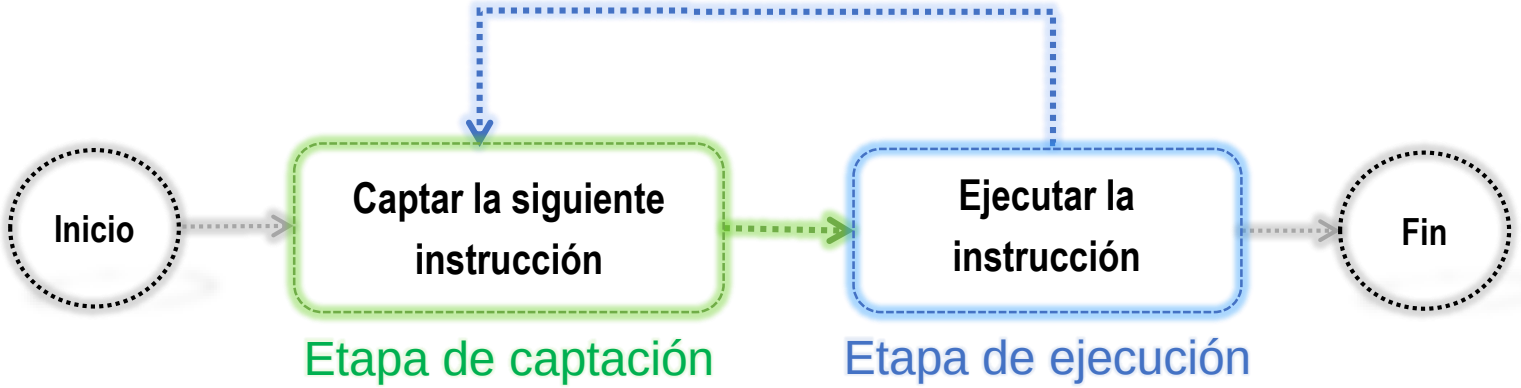
Lavandería secuencial

6 horas = 4 cargas

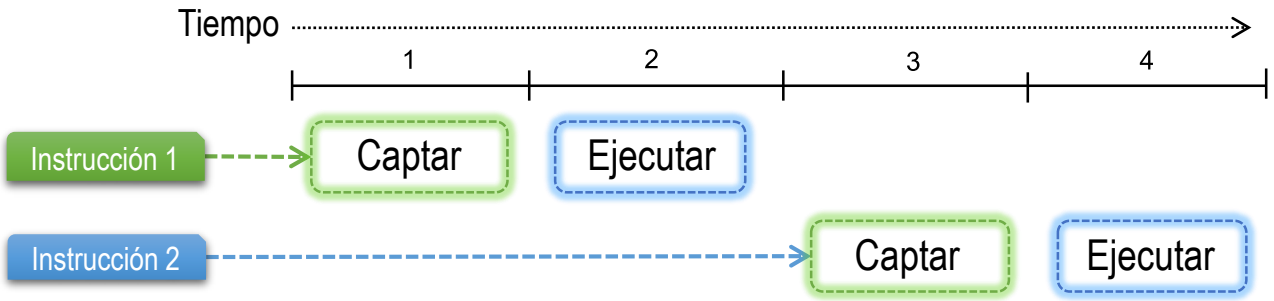
Lavandería segmentada

6 horas ≈ aproximadamente 7 cargas

Ciclo de instrucción – Etapa de captación y ejecución

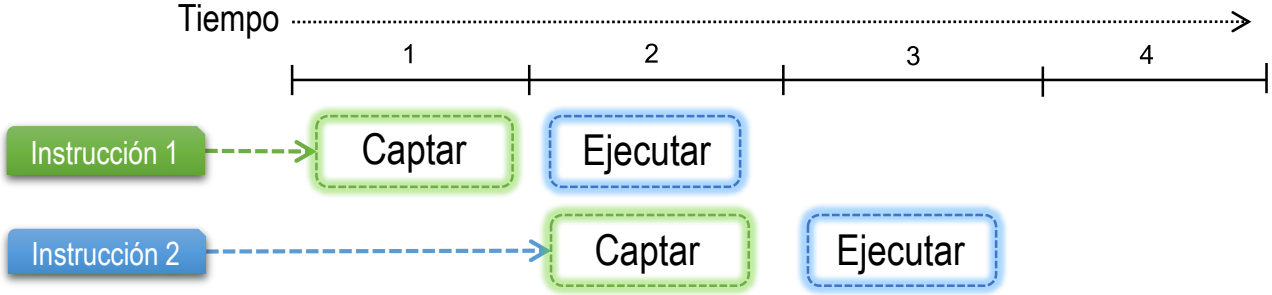


Ejecución secuencial



Pero hay periodos en la ejecución de una instrucción en los que no se accede a memoria principal, entonces este tiempo podría utilizarse para captar la siguiente instrucción en paralelo con la ejecución de la actual.

Ejecución segmentada

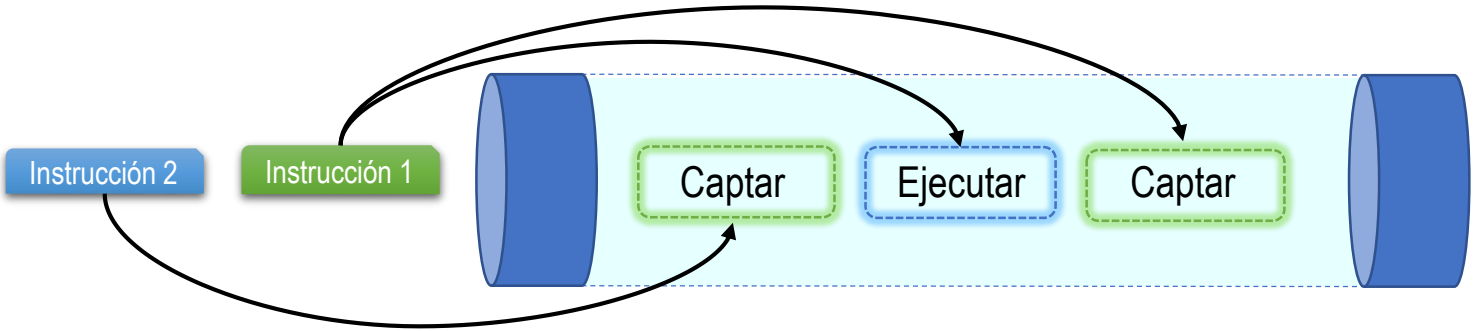


Segmentación de instrucciones

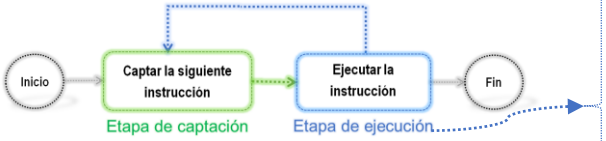
¿Qué es la segmentación de instrucciones?

Es una técnica que consiste en descomponer el ciclo de instrucción en etapas que permitan una ejecución simultánea.

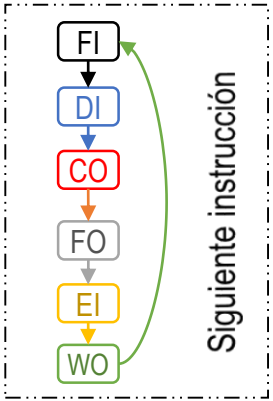
También se denomina **segmentación de cauce o Pipelining**: porque al igual que en una tubería (pipeline), en un extremo se aceptan entradas nuevas antes de que las anteriores sean salidas en el otro extremo.

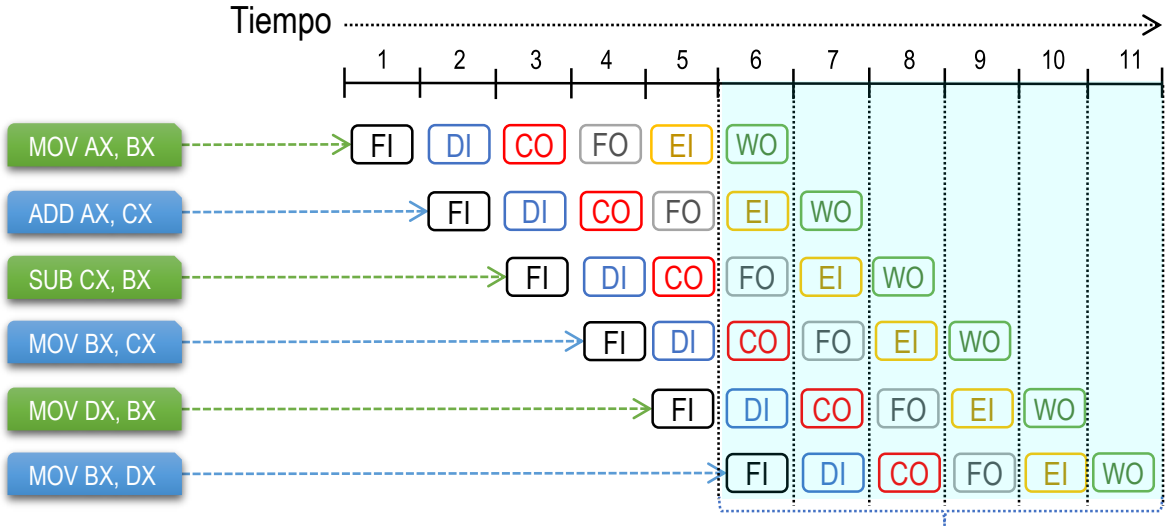


Segmentación de instrucciones



1. Captar Instrucción (**FI - Fetch Instruction**)
2. Decodificar Instrucción (**DI - Decode Instruction**)
3. Calcular Operandos (**CO - Calculate Operands**)
4. Captar Operandos (**FO - Fetch Operands**)
5. Ejecutar instrucción (**EI - Execute Instruction**)
6. Escribir Operando (**WO - Write Operand**)



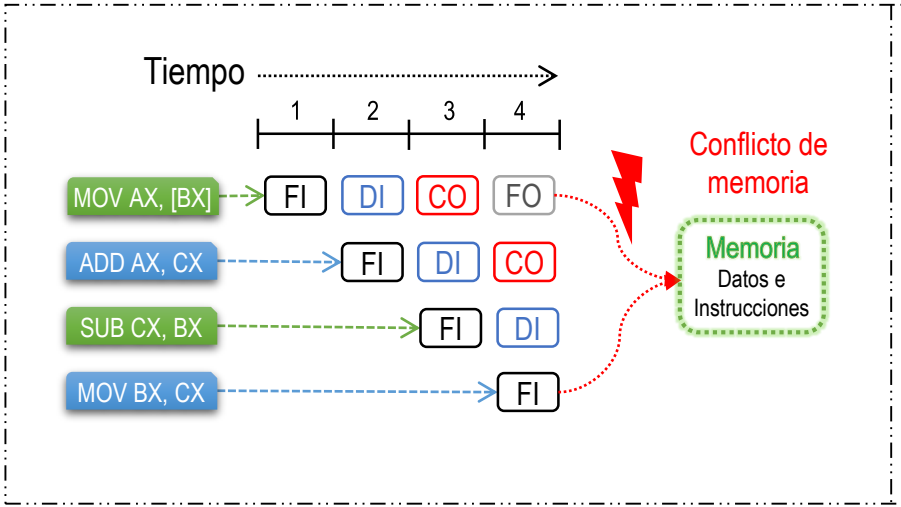


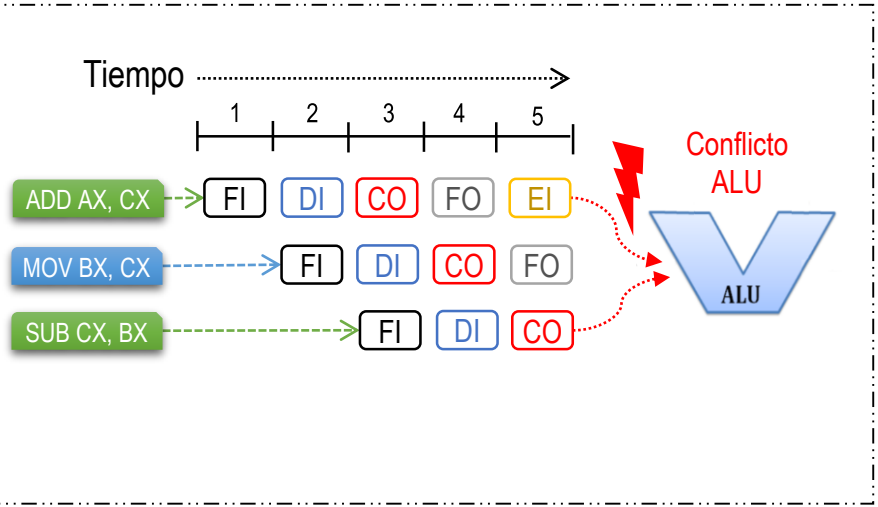
A partir del ciclo 6 finaliza una instrucción cada 1 ciclo.. **es ideal, pero ¿es real?**


Riesgos

Riesgos estructurales

Ocurre cuando dos o más instrucciones que están en el cauce necesitan utilizar el mismo recurso hardware en el mismo ciclo.

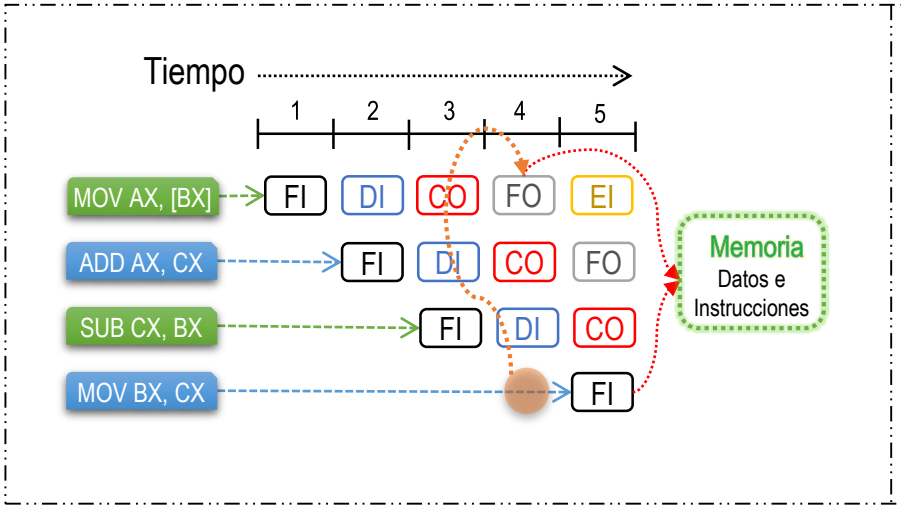


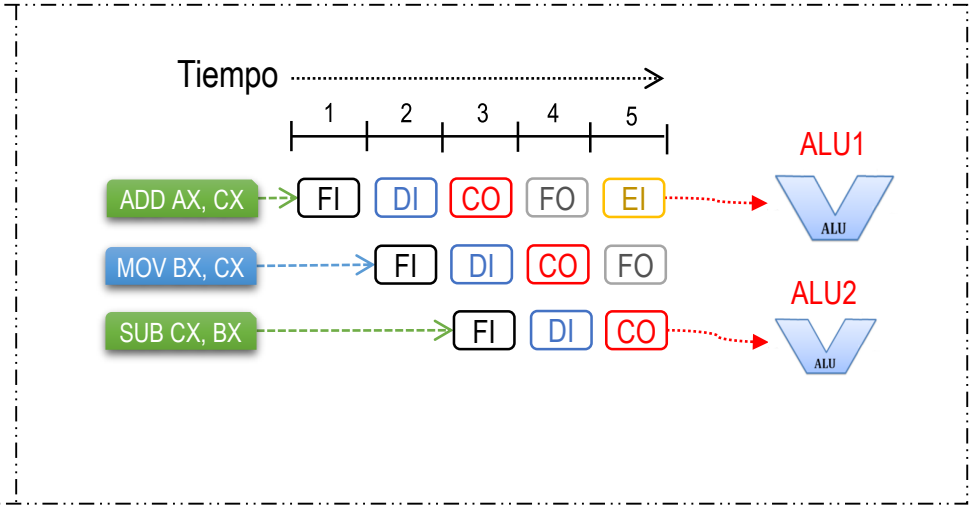


Soluciones riesgos estructurales

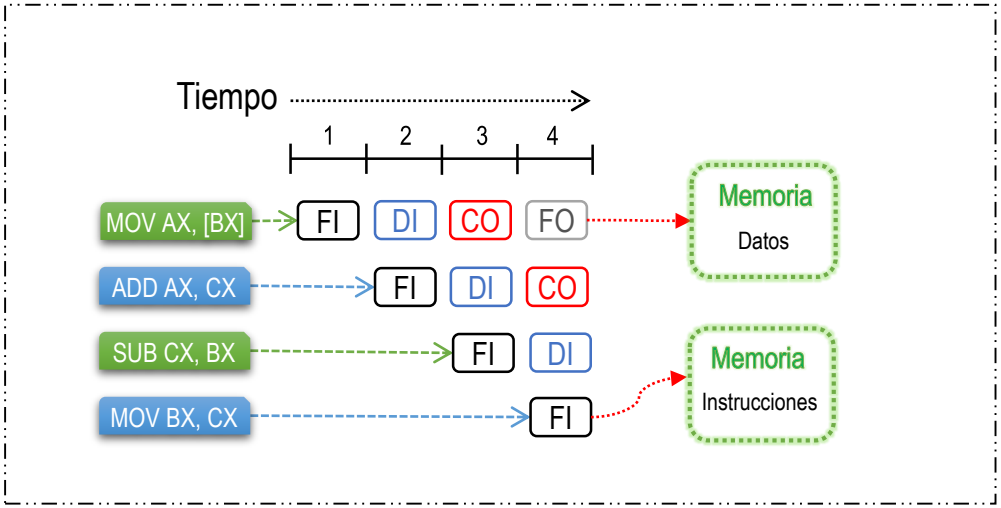
1 Insertar un ciclo ocioso por hardware.

2 Duplicación de hardware: dos ALU.



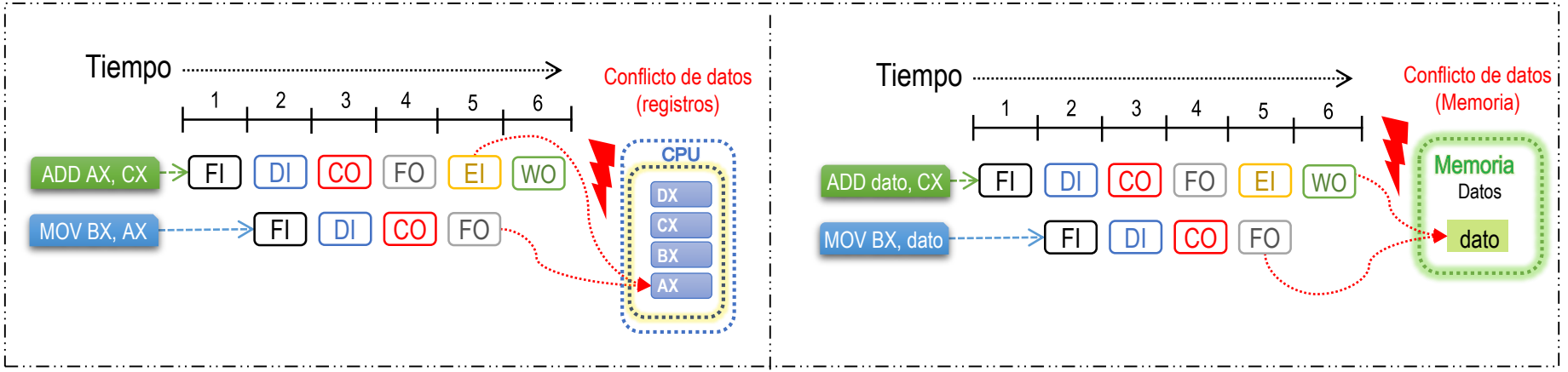


3 Separar memoria en datos e instrucciones (Arq. Harvard).



Riesgos de dependencia de datos

Ocurren cuando dos instrucciones utilizan el mismo dato en determinada etapa del cauce.

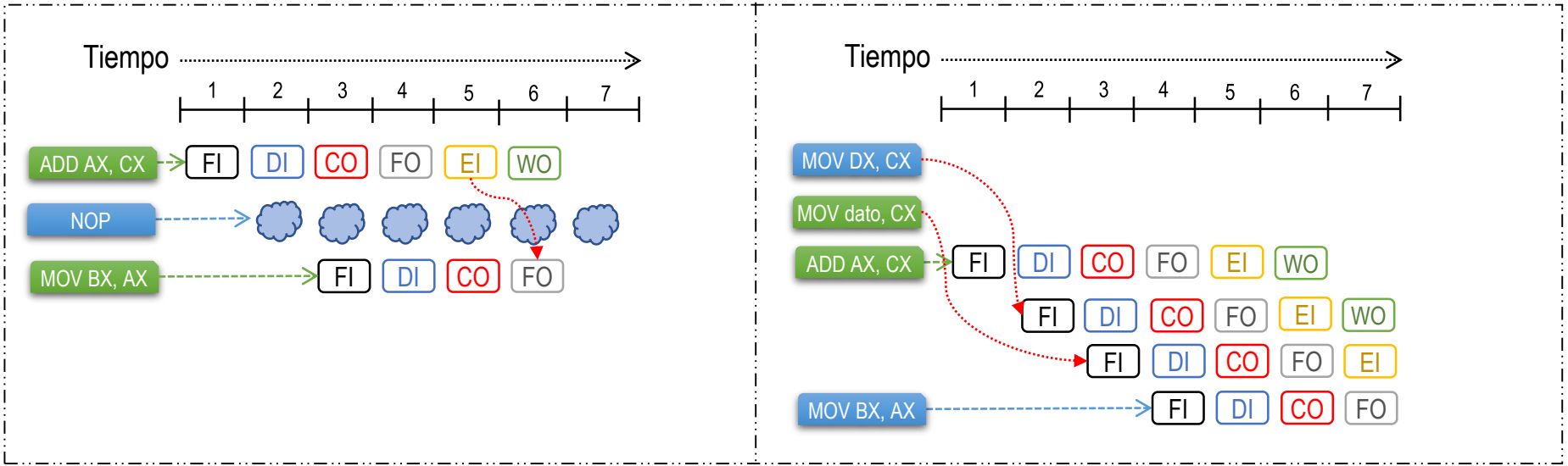


Soluciones riesgos de dependencia datos

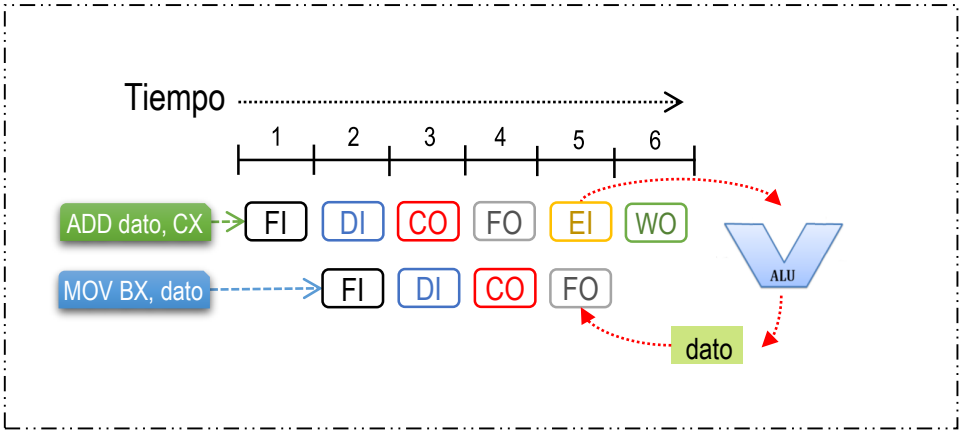
Ejecución fuera de orden

1 El compilador inserta **ciclos ociosos** (burbujas), instrucción **NOP**.

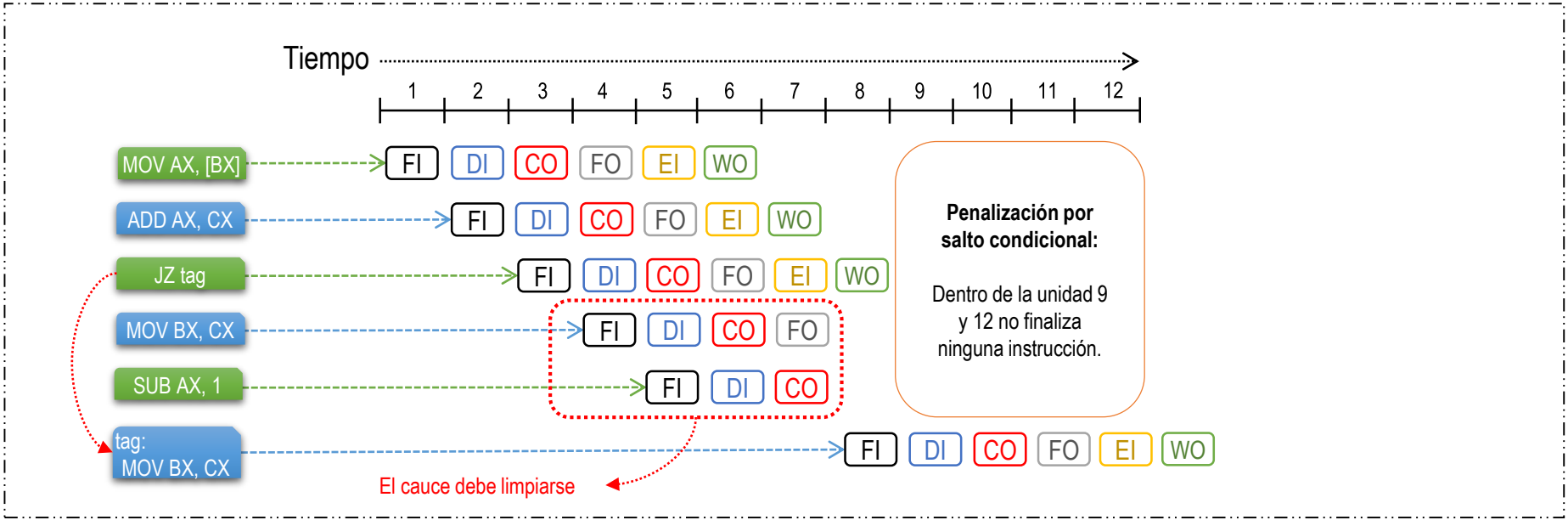
2 **Reordenación de código**: se separa lo máximo posible las instrucciones con dependencia de datos.



3 **Adelantamiento de operandos (forwarding)**: consiste en pasar directamente el resultado obtenido con una instrucción a las instrucciones que lo necesitan como operando (se hace por hardware)



El problema se presenta cuando la ejecución de una instrucción depende de cómo se ejecute otra. **Ejemplo instrucción de salto condicional**.

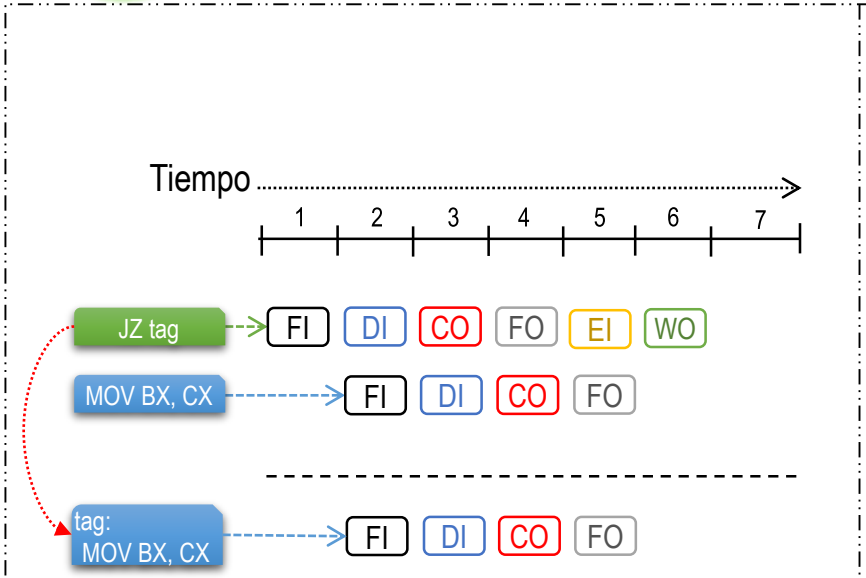


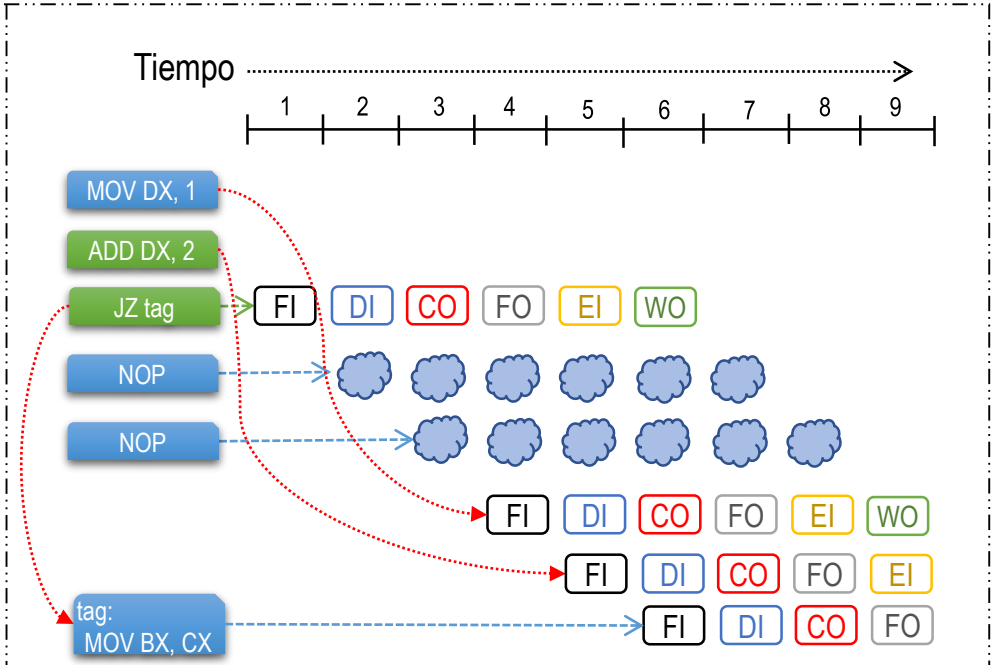
Soluciones riesgos de control

1 **Flujo múltiple**: se implementa duplicando las partes iniciales del cauce y se captan los dos caminos (siguiente instrucción e instrucción destino del salto).

ejecución especulativa

2 **Salto retardado:** en el período de penalización de una instrucción de salto, el compilador trata de situar instrucciones útiles (que no dependan del salto) en ese periodo. Si no es posible, se utilizan instrucciones NOP (requiere reordenamiento de código).





3 **Desenrollado de bucles (Loop Unrolling)**: la técnica consiste en duplicar las declaraciones del cuerpo del bucle una o muchas veces, para implementarlo es necesario, que las declaraciones dentro del loop independientes entre si.

❖ **Loop normal**

```
int x;
for (x = 0; x <  4; x++) {
   vector[x] = vector[x] * 2;
}
```

❖ **Loop desenrollado**

```
vector[0] = vector[0] * 2;
vector[1] = vector[1] * 2;
vector[2] = vector[2] * 2;
vector[3] = vector[3] * 2;
```

Como resultado de esta modificación, el nuevo programa tiene que hacer solo 20 iteraciones, en lugar de 100.

❖ **Loop normal**

```
int x;
for (x = 0; x < 100; x++) {
   delete(x);
}
```

❖ **Loop desenrollado**

```
int x;
for (x = 0; x < 100; x += 5 ) {
  delete(x);
  delete(x + 1);
  delete(x + 2);
  delete(x + 3);
  delete(x + 4);
}
```

❖ **Ventajas**:

- El objetivo del desenrollado de bucles es aumentar la velocidad de un programa reduciendo las instrucciones que controlan el bucle, esto reduce la penalización por saltos, los bucles se pueden reescribir como una secuencia repetida de declaraciones independientes similares.

❖ **Desventajas**:

- Aumento del tamaño del código del programa, que puede provocar un aumento de errores en la memoria caché de instrucciones, lo que puede afectar negativamente al rendimiento.

4 **Predicción de saltos**: existen diferentes técnicas:

❖ **Estáticas**: se toma una decisión en función del código del programa.

Compilador

- El compilador predice la siguiente instrucción basado en la dirección destino del salto:
  - **"hacia atrás"** → estructura **bucle** (salto tomado).
  - **"hacia adelante"** → estructura **Si-Finsi** (salto no tomado).
- El compilador añade un bit de predicción al opcode de la instrucción.

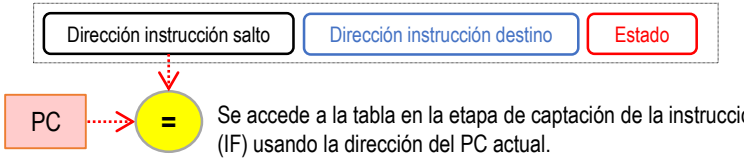
En la etapa de decodificación

- Salto no tomado (**non-taken**): el CPU continua la búsqueda de la siguiente instrucción después del salto. **(Si acierta el cauce no tiene retardo).**
- Salto tomado (**taken: tomado**): el CPU continua la búsqueda de la instrucción a partir de la dirección destino. **(el cauce tiene un ciclo de retardo).**

Se debe agregar recursos a la etapa de decodificación para poder calcular si debe saltar o no, y enviar la dirección destino al PC .

❖ **Dinámicas**: dependen de la historia de la ejecución, trabaja en tiempo de ejecución.

- **Tabla de historia de saltos:** es una memoria cache que almacena por cada instrucción de salto la dirección de la instrucción de salto, la dirección destino de la instrucción y el historial del salto (si salto o no).



1. Si el PC no se encuentra en la tabla prosigue según la predicción estática.
   - Si el resultado es no tomado (non-taken), no se almacena en la tabla
   - Si el resultado es tomado (taken), se ingresa una entrada a la tabla.

2. Si el PC se encuentra en la tabla, se obtiene de la tabla cuál es la siguiente instrucción a ejecutar, que puede ser la siguiente en el orden secuencial o la del destino del salto.
   - Si la predicción falla (salto tomado o no tomado), se modificará la entrada correspondiente de la tabla para predecir los futuros saltos correctamente.

Preguntas?