A close-up photograph of a computer's central processing unit (CPU) cooler. The cooler is made of a dark, ribbed metal and features a prominent fan at the top. A silver metal plate is attached to the base of the cooler, which has the brand name "be quiet!" printed on it in a black, sans-serif font. Several silver screws are visible where the plate is fastened to the cooler.

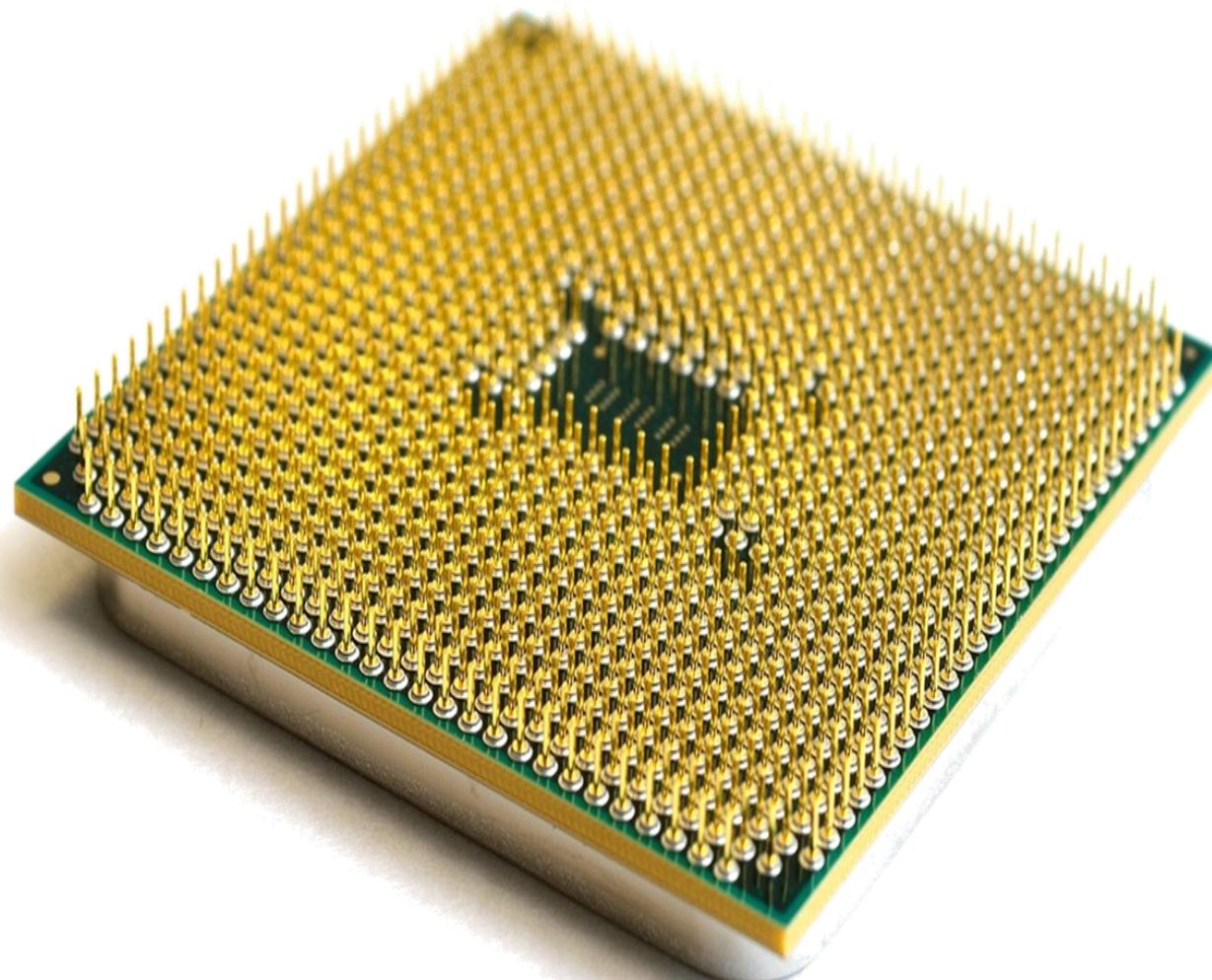
# Arquitectura de Computadoras

---

Introducción a WinMips

# Agenda

## ► Temas



### 01 Introducción

Características Generales. Arquitectura. Registros.  
Directivas del Compilador

### 02 Instrucciones

Formato de 3 operandos. Instrucciones de acceso a memoria, aritméticas, lógicas y de control

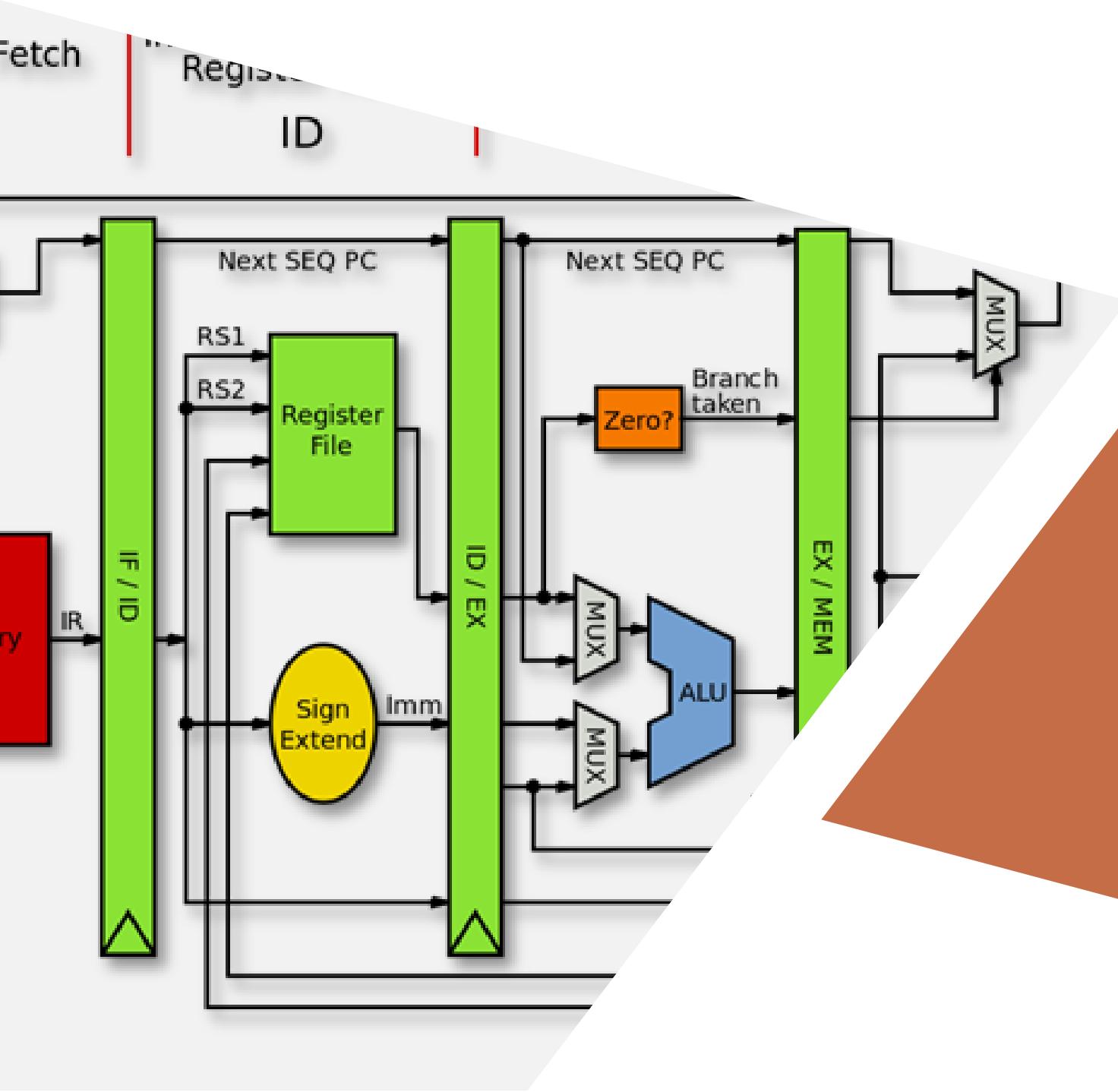
### 03 Causa/Pipeline

Funcionamiento. Tareas de cada Etapa del Cauce.  
Ciclos por Instrucción (CPI). Desglose de instrucciones típicas por etapas

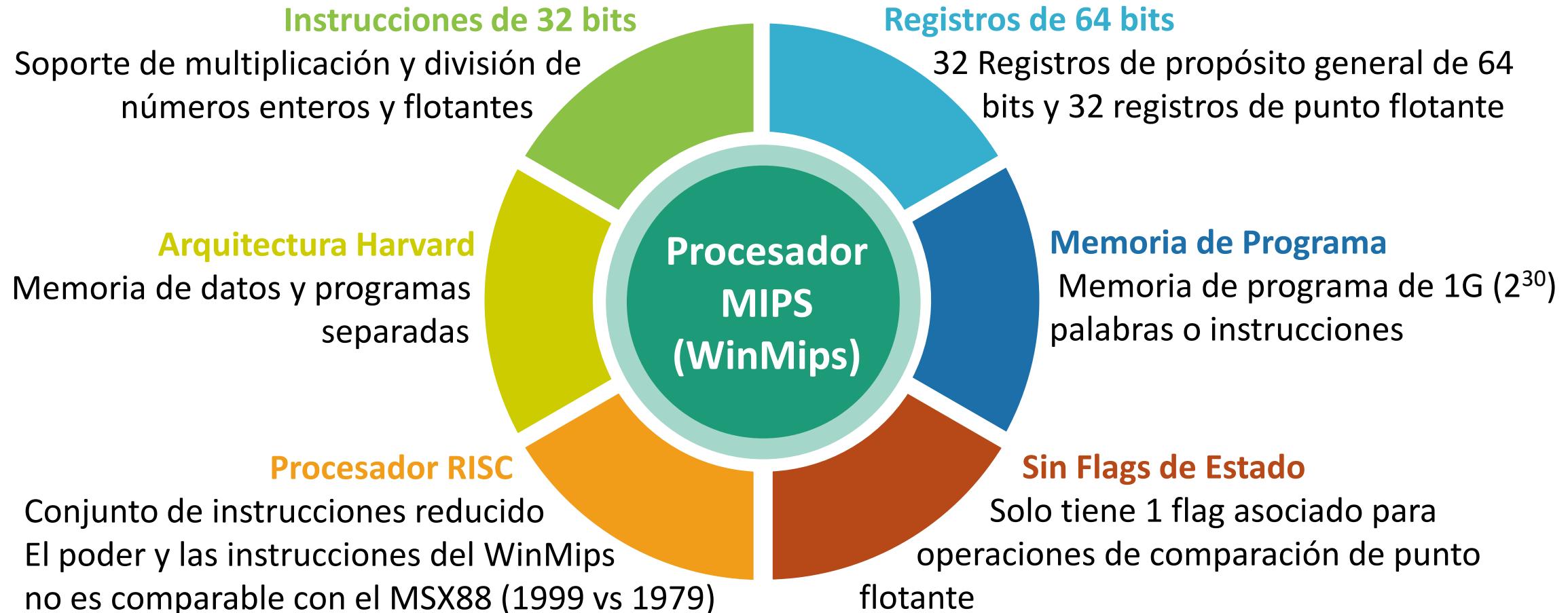
### 04 Atascos

Problemas del Cauce. Atascos de Datos RAW, WAR y WAW. Adelantamiento de Operandos. Atascos por saltos. Branch Target Buffer y Salto retardado

# Introducción a WinMips

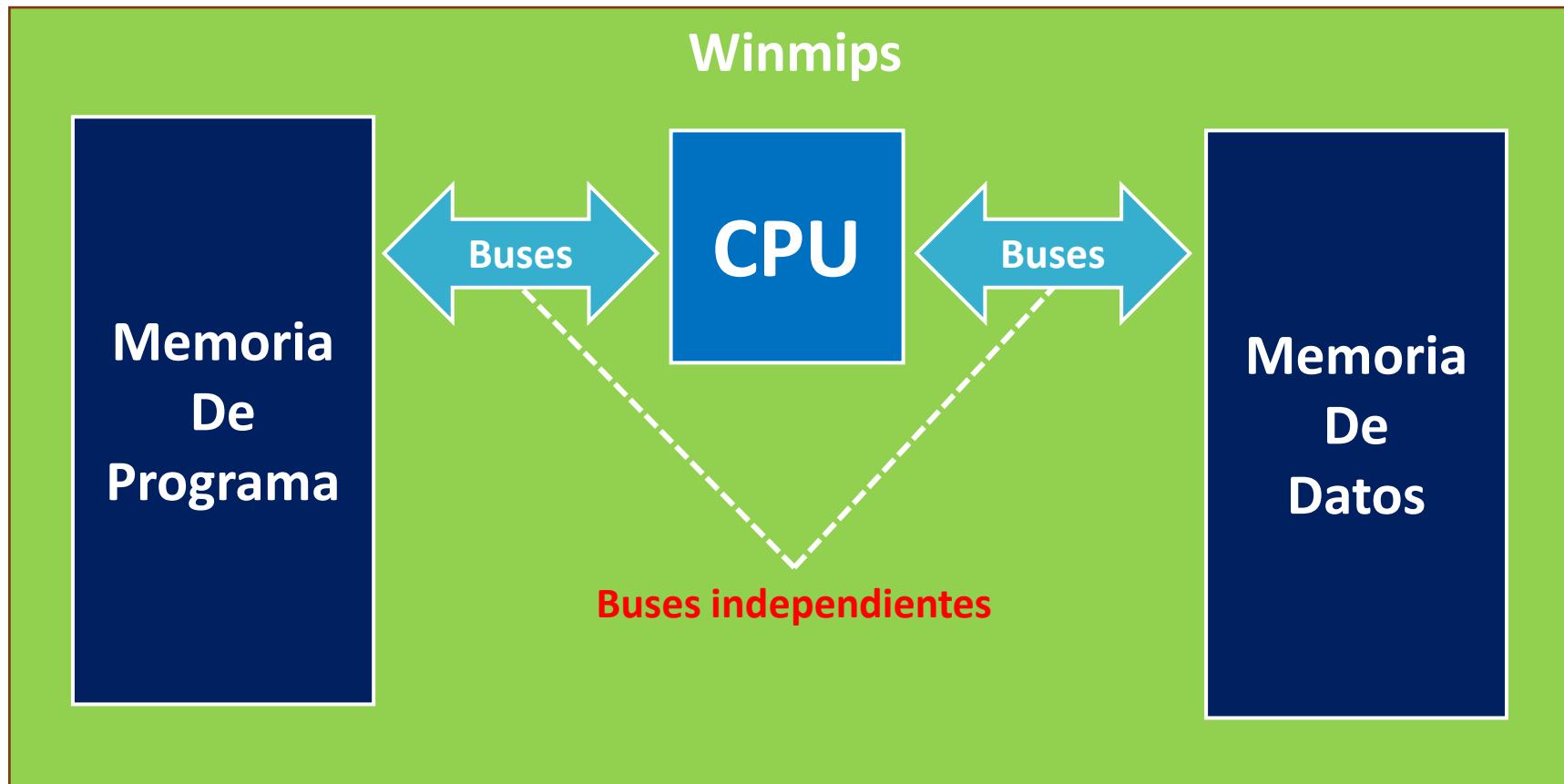


# WinMips – Características Generales



# WinMips – Características Generales

## Arquitectura Harvard



# WinMips – Características Generales

## Arquitectura Harvard vs Von Neuman

### Desventajas

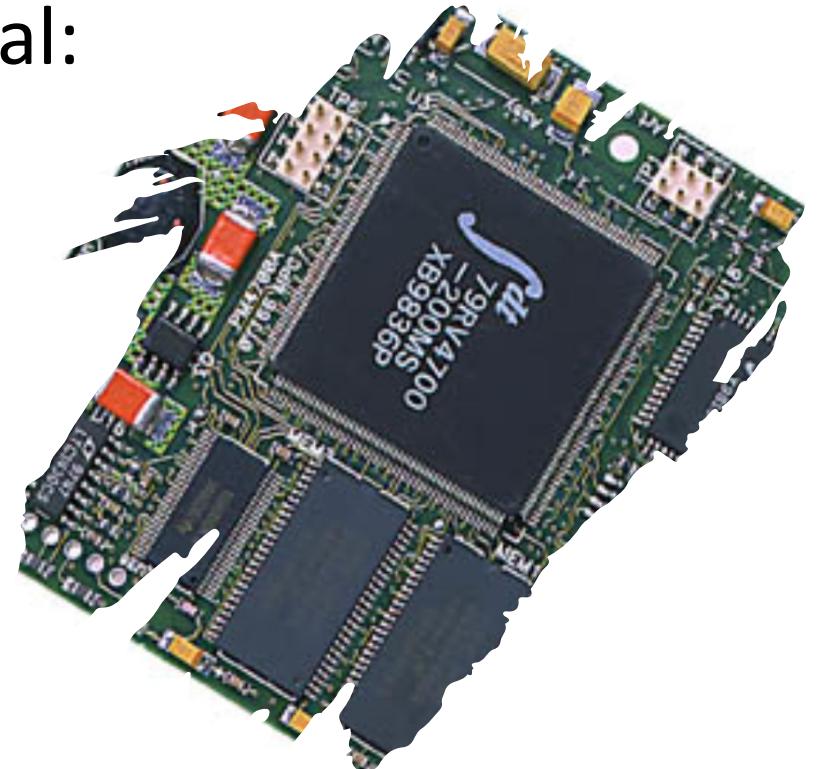
- Menos flexibilidad para almacenar datos y programas
- Menor rendimiento de memoria cache al dividirla en dos partes

### Ventajas

- Tamaño de instrucciones independientes del tamaño de la memoria de datos
- Electrónica más simple
- Acceso simultaneo a instrucciones y datos por la independencia de buses
- Acceso a una instrucción en un único ciclo

# WinMips - Registros

- 32 registros de enteros de propósito general:
  - $r\{i\}$  con  $i$  nro de registro: r0 a r31
  - r0 siempre es 0, ignora modificaciones
- 32 registros de punto flotante:
  - $f\{i\}$  con  $i$  nro de registro: f0 a f31
- No hay registro de flags de estado
- No tiene soporte de pila



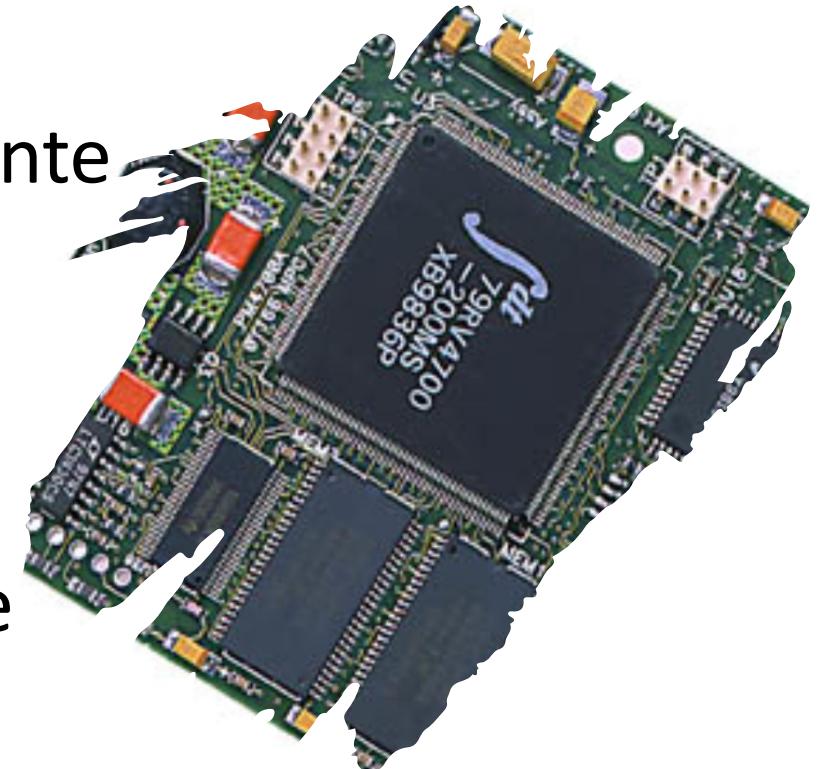
# WinMips – Directivas del compilador

.code o .text :

- Indica que el texto que sigue es código fuente
- El compilador asigna automáticamente la posición en memoria de programa

.data :

- Indica al compilador que el texto que sigue son datos
- El compilador asigna automáticamente la posición en memoria de datos



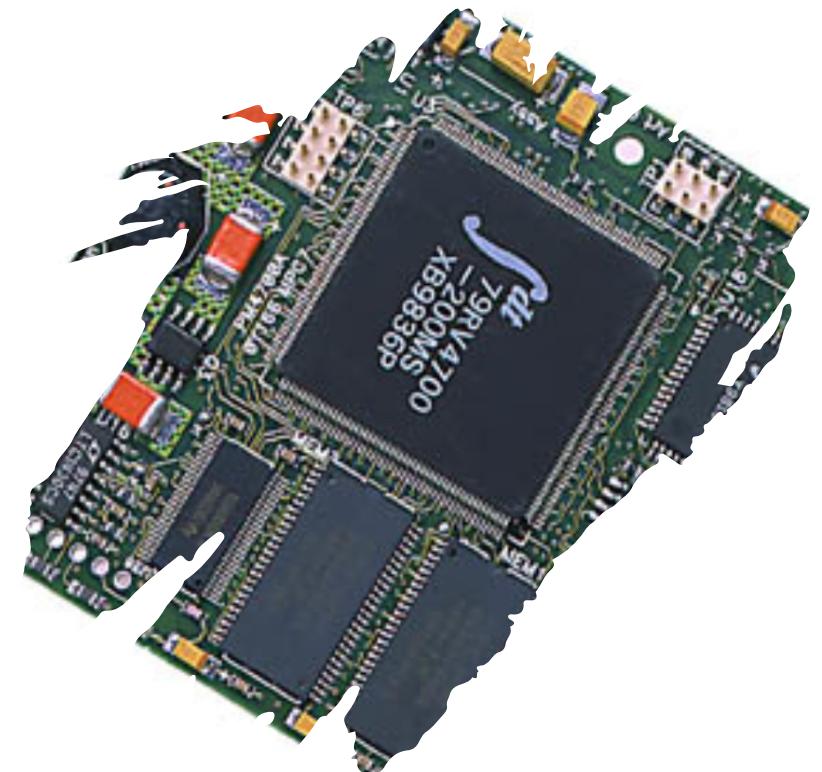
# WinMips – Directivas del compilador

.org {dirección específica}

- Ubica el texto que sigue a partir de una dirección específica. Se puede usar con código o con datos.

.space {número de bytes}

- Deja un espacio de bytes entre lo anterior y lo siguiente



# WinMips – Directivas del compilador

.ascii {string} :

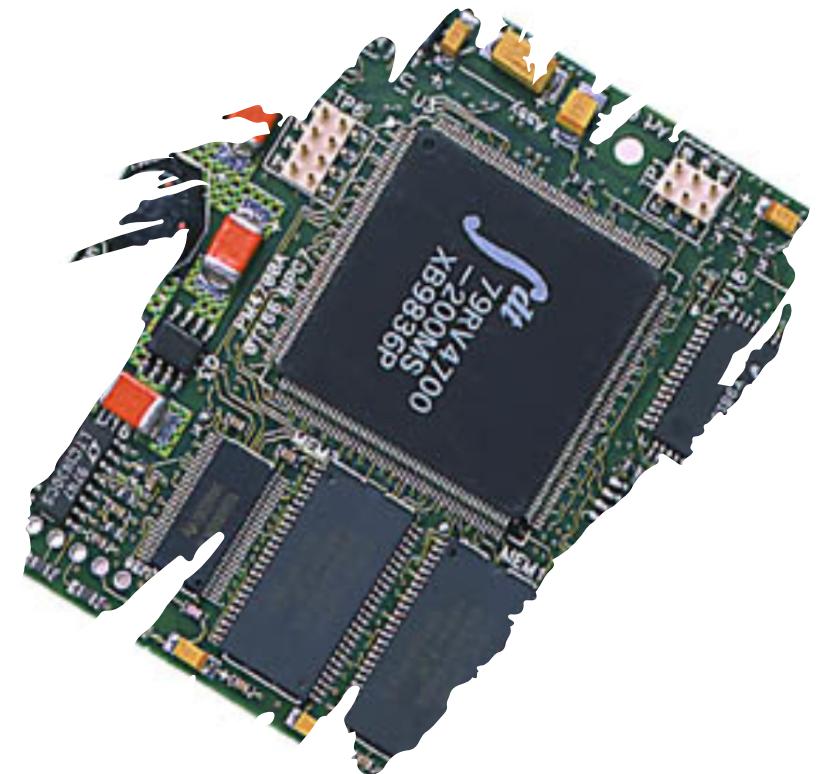
- Almacena una cadena ASCII
- Ejemplo:

mi\_variable: .ascii "hola" ;4 bytes

.asciiz {string} :

- Almacena una cadena ASCII terminada en 0
- Ejemplo:

mi\_variable: .asciiz "hola" ;5 bytes



# WinMips – Directivas del compilador

## .word:

➤ almacena número(s) de 64-bits

➤ Ejemplo:

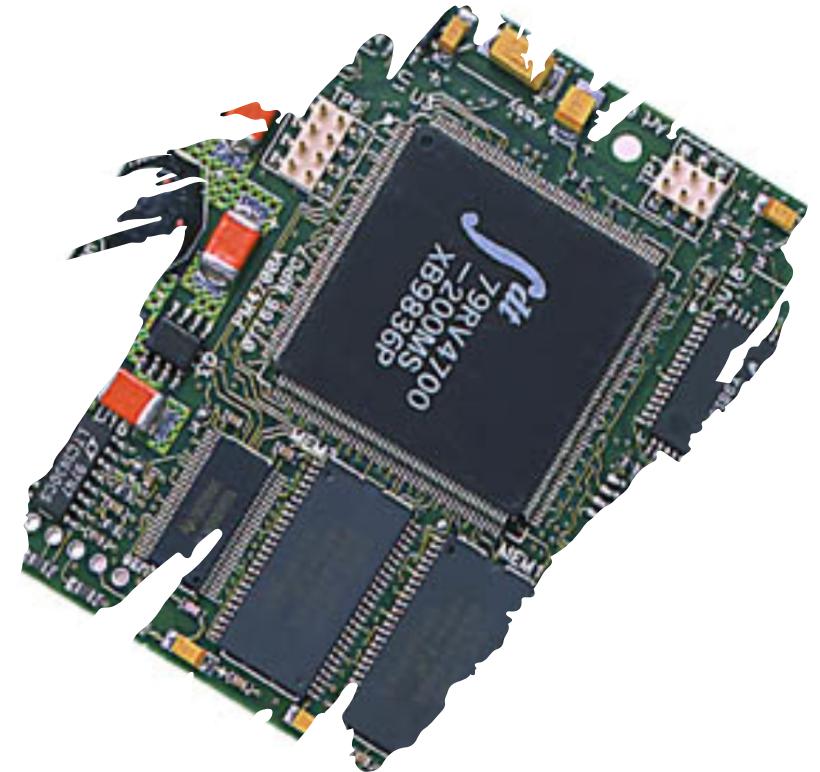
```
mi_variable: .word 1234 ;8 bytes
```

## .word 32:

➤ almacena número(s) de 32-bits

➤ Ejemplo:

```
mi_variable: .word32 1234 ;4 bytes
```



# WinMips – Directivas del compilador

## .word 16:

- almacena número(s) de 16 bits
- Ejemplo:

```
mi_variable: .word16 1234 ;2 bytes
```

## .byte

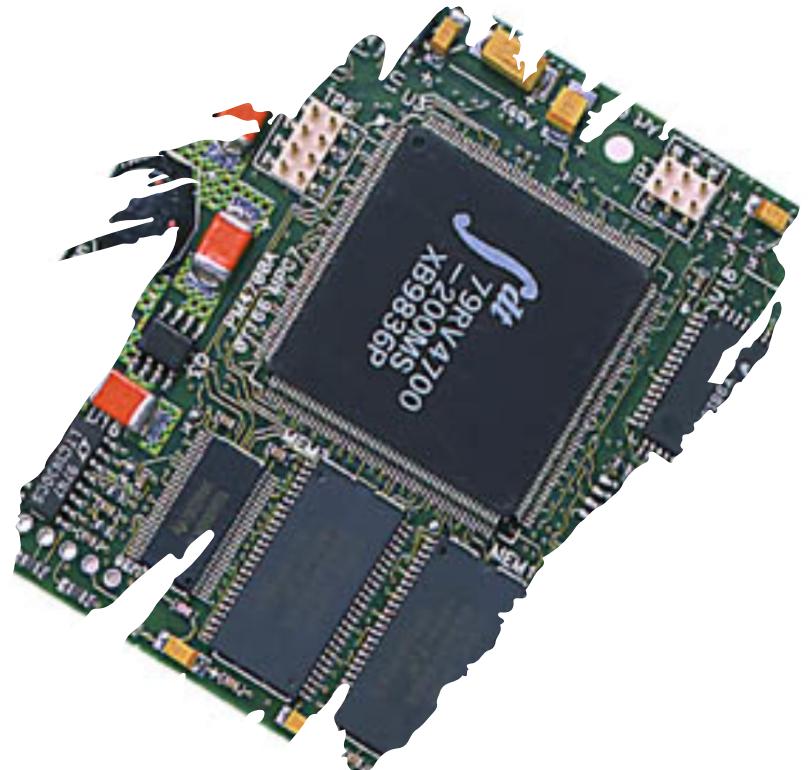
- almacena datos de 1 byte
- Ejemplo:

```
mi_variable: .byte 123 ;1 byte
```

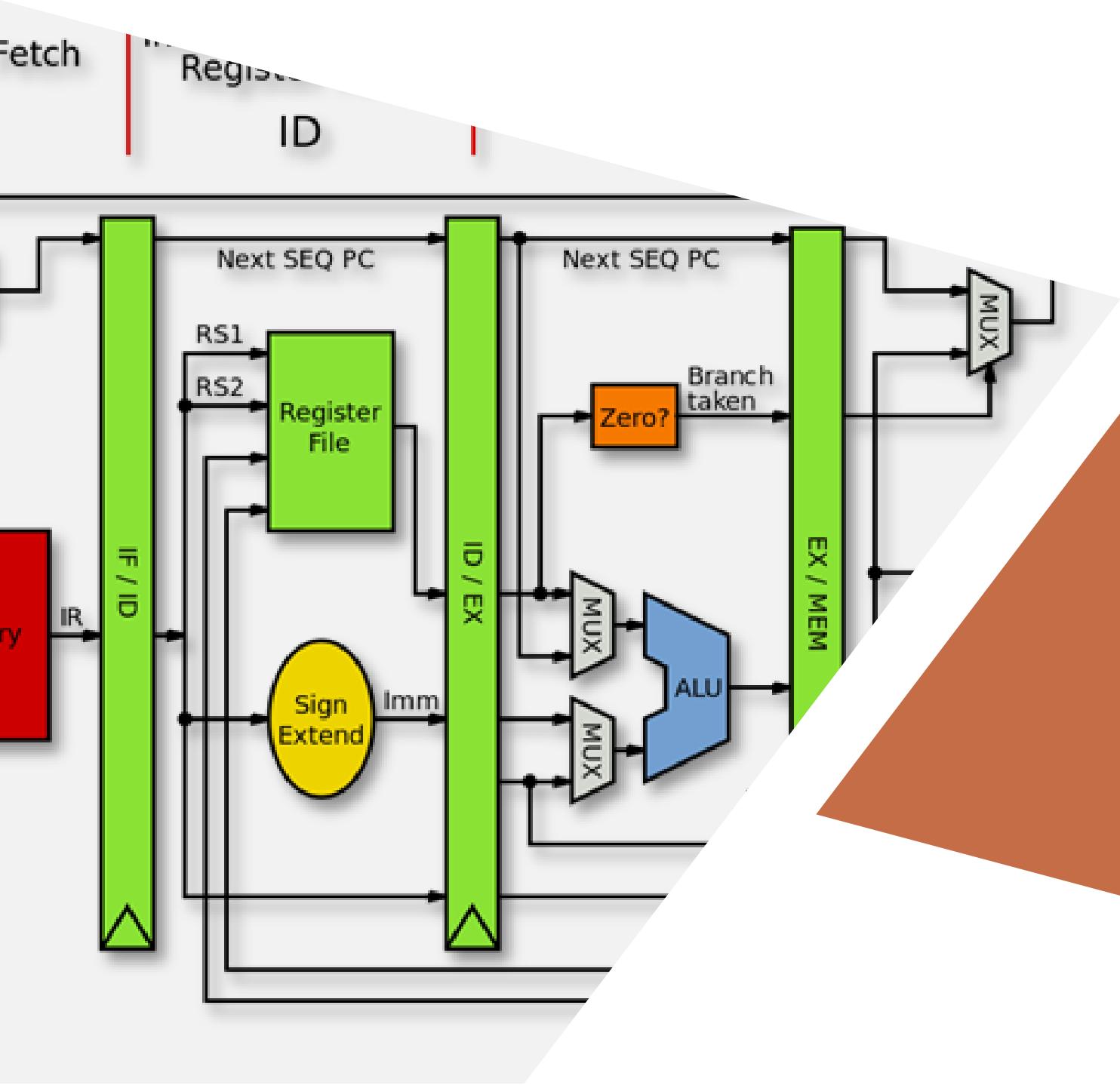
## .double

- almacena número(s) en punto flotante
- Ejemplo:

```
mi_variable: .double 3.14159 ;8 bytes
```



# Instrucciones de WinMips



# WinMips – Instrucciones

## ➤ Instrucciones de hasta 3 operandos:

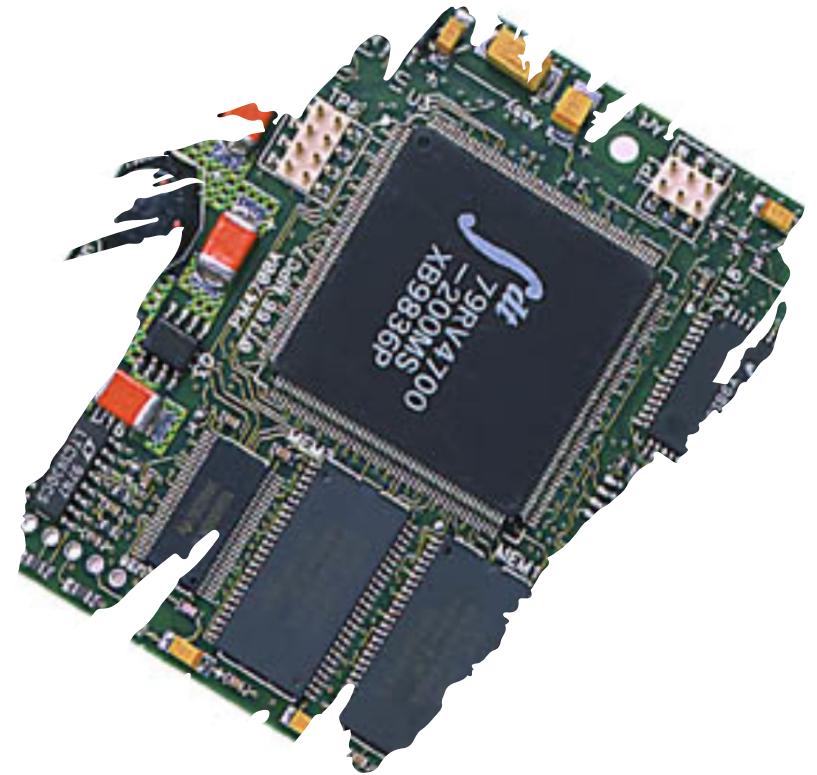
➤ {Operación } {Destino} , {Operando 1}, {Operando 2}

## ➤ Ejemplos:

**DADDI R1, R2, 5 ; R1 ← R2 + 5**

**DMUL R2, R3, R3 ; R2 ← R3 \* R3**

**DDIV R25, R4, R3 ; R25 ← R4 / R3**



# Instrucciones de Transferencia

- Solo hay 2 instrucciones para acceder a memoria de datos
- Las demás instrucciones utilizan registros y/o valores inmediatos
- **LD {reg. Dest.}, {memoria} ( {reg. Despl.} ) :**
  - Load Data: asignar un registro con un valor de memoria
  - $\{reg. Dest\} \leftarrow [\{memoria\} + \{reg. Despl.\}]$
  - Ejemplo: `LD R1, mi_var(R2)`     $R1 \leftarrow [mi\_var+R2]$
- **SD {reg. Orig.}, {memoria} ( {reg. Despl.} ) :**
  - Store Data: copia en memoria el valor de un registro
  - $[\{memoria\} + \{reg. Despl.\}] \leftarrow \{reg. Orig\}$
  - Ejemplo: `SD R1, mi_var(R2)`     $[mi\_var+R2] \leftarrow R1$

# Instrucciones de Transferencia

¿Porque sumar siempre un registro a una dirección de memoria?

- ¿y por que no?
- Si “mi\_var” es un entero de 64 bits y no quiero sumarle a su **dirección** nada, puedo usar R0 que siempre está en cero:

`LD R1, mi_var(R0)      R1 ← [mi_var]`

- Pero si “mi\_var” fuera un arreglo (ej: de enteros de 64 bits), sería interesante sumarle a la dirección un desplazamiento:

**Bucle:** `LD R1, mi_var(R2) ;recupera elemento`  
                  ;  
                  ;  
                  ;  
`DADDI R2, R2, 8 ;desplaza a prox. Elemento`  
`J Bucle           ;repite proceso`

# Instrucciones Aritméticas

**DADDI {reg. Dest.} , {reg. Oper. 1} , {inmediato 16 bits} :**

- Sumar un registro y un valor inmediato
- $\{ \text{reg. Dest} \} \leftarrow \{ \text{reg. Oper. 1} \} + \{ \text{valor inmediato} \}$
- Ejemplo:
  - **DADDI R1, R2, -1 ;suma valor negativo**
  - **DADDI R1, R0, 5 ;asignación valor inmediato**

**DADD {reg. Dest.} , {reg. Oper. 1} , {reg. Oper. 2} :**

- Sumar dos registros
- $\{ \text{reg. Dest} \} \leftarrow \{ \text{reg. Oper. 1} \} + \{ \text{reg. Oper. 2} \}$
- Ejemplo:
  - **DADD R1, R2, R3 ;suma de registros**
  - **DADD R1, R2, R0 ;asignación de registro**

➤ Instrucciones similares: DSUB, DMUL, DDIV

# Instrucciones Lógicas

**ANDI** {reg. Dest.} , {reg. Oper. 1} , {inmediato 16 bits} :

- Realizar “Y” entre un registro y valor inmediato
- {reg. Dest}  $\leftarrow$  {reg. Oper. 1} AND {valor inmediato}
- Ejemplo:
  - **ANDI R1, R2, 5 ; R1 = R2 AND 5**

**AND** {reg. Dest.} , {reg. Oper. 1} , {reg. Oper. 2} :

- Realizar “Y” entre dos registros
- {reg. Dest}  $\leftarrow$  {reg. Oper. 1} AND {valor reg. Oper. 2}
- Ejemplo:
  - **AND R1, R2, R3 ; R1 = R2 AND R3**

**Instrucciones similares:**

- “O”: OR, ORI
- “O exclusivo”: XOR, XORI

# Instrucciones de Desplazamiento

**DSLL {reg. Dest.} , {reg. Oper. 1} , {inmediato 5 bits} :**

- Desplazar a izquierda un registro según un valor inmediato
- {reg. Dest}  $\leftarrow$  {reg. Oper. 1}  $\ll$  {valor inmediato}
- Ejemplo:
  - **DSLL R1, R2, 1 ; R1 = R2 << 1**

**DSLLV {reg. Dest.} , {reg. Oper. 1} , {reg. Oper. 2} :**

- Desplazar a izquierda un registro según un valor de otro
- {reg. Dest}  $\leftarrow$  {reg. Oper. 1}  $\ll$  {valor reg. Oper. 2}
- Ejemplo:
  - **DSLLV R1, R2, R3 ; R1 = R2 << R3**

**Instrucciones similares:**

- **Desplazamiento Aritmético a derecha:** DSRL, DSRA, DSRAV

# Instrucciones de Comparación

**SLTI {reg. Dest.} , {reg. Oper. 1} , {inmediato 16 bits} :**

- comparar por menor un registro y valor inmediato
- {reg. Dest}  $\leftarrow$  {reg. Oper. 1} < {valor inmediato}
- Ejemplo:
  - **SLTI R1, R2, 2 ; R1=1 si R2<2, R1=0 si R2>=2**

**SLT {reg. Dest.} , {reg. Oper. 1} , {reg. Oper. 2} :**

- comparar por menor dos registros
- {reg. Dest}  $\leftarrow$  {reg. Oper. 1} < {reg. Oper. 2}
- Ejemplo:
  - **SLT R1, R2, R3 ; R1=1 si R2<R3, R1=0 si R2>=R3**

**Instrucciones similares:**

- **Comparación sin signo: SLTIU**

# Instrucciones de Transferencia de Control

**J {valor inmediato 26 bits o etiqueta} :**

- Transferir control a una dirección (JMP en MSX88)
- $\{IP\} \leftarrow \{\text{valor inmediato o etiqueta}\}$
- Ejemplo: **J seguir**

**JAL {valor inmediato o etiqueta} :**

- Transferir control a una dirección y guardar dirección de retorno (CALL en MSX88)
- $\{IP\} \leftarrow \{\text{valor inmediato o etiqueta}\}$
- $\{R31\} \leftarrow \{\text{dirección de retorno}\}$
- Ejemplo: **JAL subrutina**

**JR {registro} :**

- Transferir control a una dirección contenida en un registro
- $\{IP\} \leftarrow \{\text{registro}\}$

# Instrucciones de Transferencia de Control

**BEQ** {reg. Oper. 1} , {reg. Oper. 2} , {etiqueta 16 bits} :

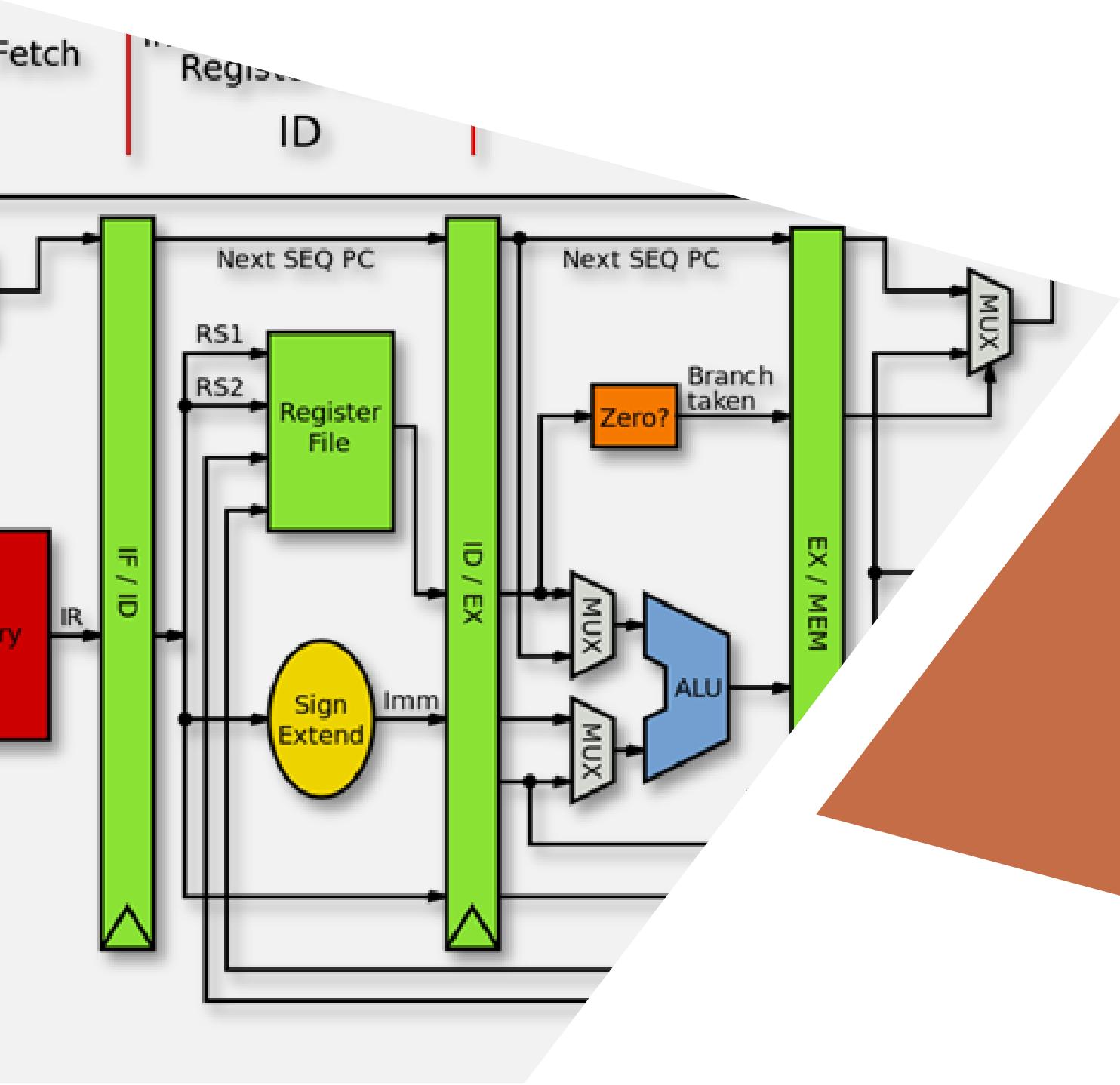
- Transferir control a una dirección si los registros son iguales
- {si reg. Oper 1 = reg. Oper 2} → {IP} ← {IP+desp. etiqueta}
- Ejemplo: **BEQ R1, R2, son\_iguales**

**BEQZ** {reg. Oper} , {etiqueta 16 bits} :

- Transferir control a una dirección si un registro es cero
- {si reg. Oper = 0} → {IP} ← {IP+ desp. etiqueta}
- Ejemplo: **BEQZ R1, es\_cero**

**Complementos:**

- **BNE para BEQ**
- **BNEZ para BEQZ**



# Cause del WinMips

# Cauce – Etapas



**Una instrucción debe pasar por las 5 etapas**

## **IF – Instruction Fetch**

- Busca instrucción en memoria de programa

## **ID – Instruction Decode**

- Decodifica instrucción y busca operandos

## **EX - Execution**

- Ejecuta la instrucción y obtiene resultados

## **MEM – Memory Access**

- Accede a memoria de datos (según la instrucción)

## **WB – Write Back**

- Escritura del resultado en el registro destino (según la instrucción)

# Cauce – Etapas



- Una instrucción debe pasar por cada bloque o etapa
- En todo momento hay 4 etapas ociosas (inactivas)
- En cada etapa, una instrucción tarda 1 ciclo de reloj
- Los tiempos de ejecución de las instrucciones son:
  - 1 instrucción ➔ 5 ciclos
  - 2 instrucciones ➔ 10 ciclos
  - 3 instrucciones ➔ 15 ciclos
  - ...
  - N instrucciones ➔  $5 \times N$  ciclos
- Ciclos por instrucción (CPI):
  - total de ciclos / total de instrucciones:  $5 \times N / N \rightarrow 5$

# Cauce – Etapas



- Si el hardware se modifica para permitir que una instrucción aproveche la etapa que deja otra instrucción, mejoramos el tiempo ya que en todo momento hay 5 etapas activas
- Los tiempos de ejecución de las instrucciones son:
  - 1 instrucción ➔ 5 ciclos =  $4 + 1$
  - 2 instrucciones ➔ 6 ciclos =  $4 + 2$
  - 3 instrucciones ➔ 7 ciclos =  $4 + 3$
  - ...
  - N instrucciones ➔  $4 + N$  ciclos
- Ciclos por instrucción (CPI):
  - total de ciclos / total de instrucciones:  $(4+N) / N$

# Cauce – Etapas



Los tiempos de ejecución y ciclos por instrucción para 10, 100 y 1000 instrucciones son:

- 10 instrucciones → CPI =  $(4 + 10) / 10 = 1,4$
- 100 instrucciones → CPI =  $(4 + 100) / 100 = 1,04$
- 1000 instrucciones → CPI =  $(4 + 1000) / 1000 = 1,004$

Para un ciclo continuo de ejecución donde N es muy grande

- N muy grande → CPI =  $(4+N) / N = \text{aprox. } 1$

La mejora permite (teóricamente) ejecutar 5 veces más rápido, es decir 1 instrucción por cada ciclo

# Cauce – Etapas



**IF**: Búsqueda de instrucción:

- Recupera instrucción de memoria de programa
- Incrementa PC

**ID**: Decodificación de instrucción:

- Si hay operandos, se accede al banco de registros para recuperarlo
- Si es operando inmediato (16 bits), se calcula el valor (extiende a 64 bits)
- Si es un salto se calcula la dirección destino y se determina si se toma o no

**EX**: Ejecución de instrucción:

- Para instrucción de proceso (aritmética, lógica, etc.), se ejecuta en la ALU
- Para acceso a memoria, se calcula la dirección efectiva
- Para saltos, se almacena la dirección en PC

**MEM**: Solo para instrucciones que acceden a memoria para leer o escribir datos

**WB**: si se produjo algún resultado, se almacena en registro destino

# Cauce – Instrucciones por etapas

LD R2, variable(R1): cargar R2 desde memoria

<b>IF</b>	Recupera instrucción de la memoria de programa
<b>ID</b>	Decodifica instrucción. Accede al registro R1 y lo almacena en un registro temporal
<b>EX</b>	Calcula dirección efectiva [variable+R1]
<b>MEM</b>	Recupera valor de la dirección [variable+R1] de memoria de datos y lo almacena en un registro temporal
<b>WB</b>	Guarda en R2 el valor leído de la memoria (lo transfiere desde el registro temporal)

# Cauce – Instrucciones por etapas

SD R2, variable(R1): guardar R2 en memoria

<b>IF</b>	Recupera instrucción de la memoria de programa
<b>ID</b>	Decodifica instrucción. Recupera valor de R2 y R1 y los guarda en registros temporales
<b>EX</b>	Calcula dirección efectiva [variable+R1]
<b>MEM</b>	Guarda valor del registro R2 en la dirección [variable+R1] de la memoria de datos
<b>WB</b>	No se necesita escribir registro. No hace nada

# Cauce – Instrucciones por etapas

DADD R3, R2, R1: sumar 2 registros

IF	Recupera instrucción de la memoria de programa
ID	Decodifica instrucción. Recupera valor de R1 y R2 y los guarda en registros temporales
EX	Calcula suma de R1 con R2 y guarda el resultado en un registro temporal
MEM	No se necesita acceder a memoria para guardar ningún registro. No hace nada
WB	Guarda resultado de suma en R3 (lo transfiere desde el registro temporal)

# Cauce – Instrucciones por etapas

BEQ R1,R2, Etiqueta: Saltar si R1 es igual a R2

<b>IF</b>	Recupera instrucción de la memoria de programa
<b>ID</b>	Decodifica instrucción. Recupera valor de R2 y R1. Determina si salta o no. Calcula la dirección de salto.
<b>EX</b>	Si determino que debe saltar en la etapa ID, Ejecuta el salto
<b>MEM</b>	No requiere acceso a memoria. No hace nada
<b>WB</b>	No se necesita escribir registro. No hace nada

# Cause - Problemas

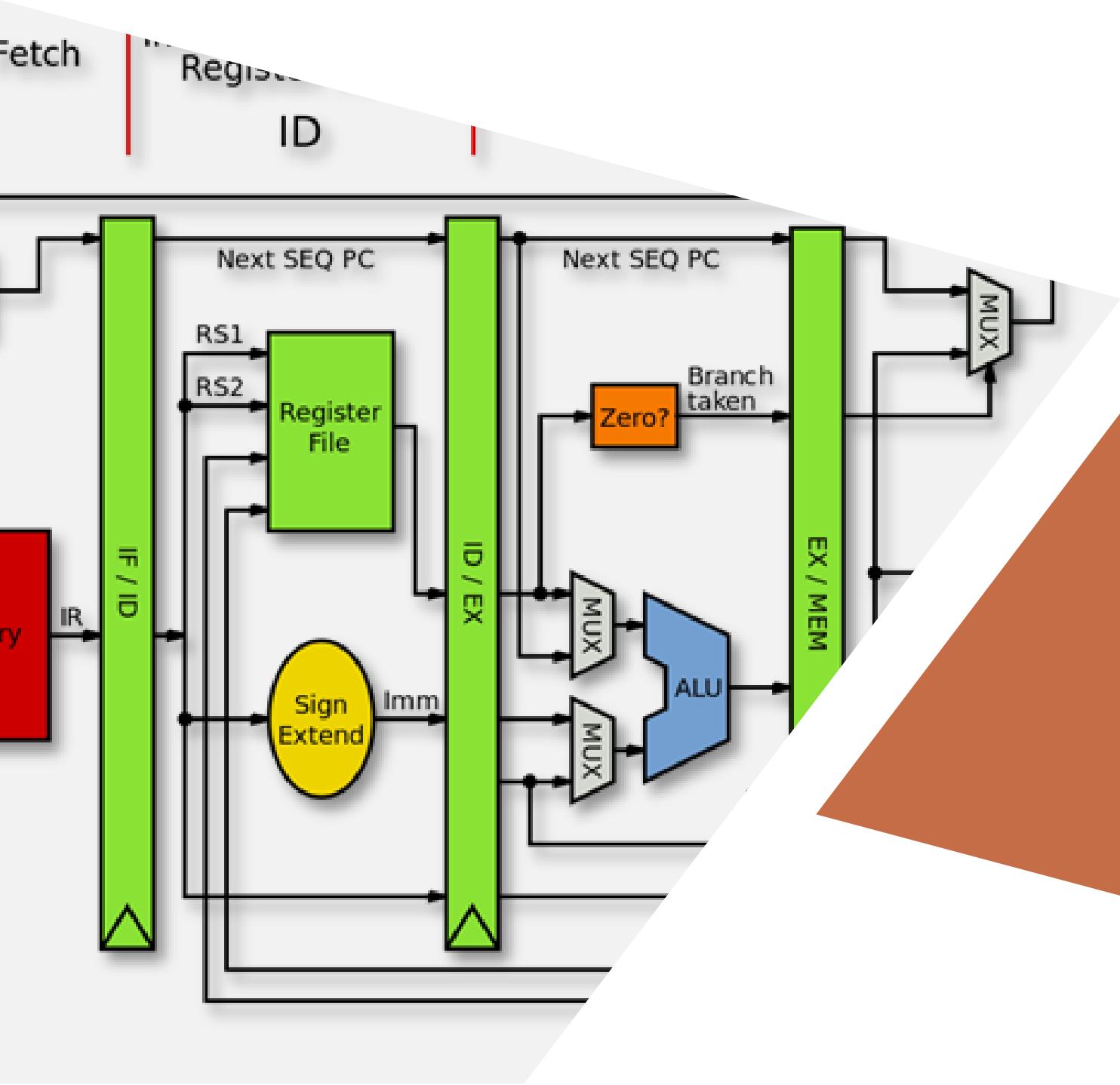
Si bien en teoría podríamos ejecutar un programa 5 veces mas rápido, en la práctica no es así:

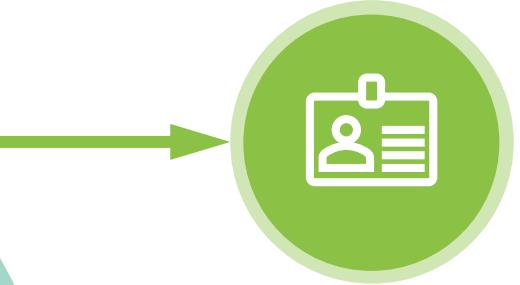
- Puede pasar que una instrucción no pueda avanzar porque su ejecución depende de que termine la ejecución de otra.
- Esta condición donde una instrucción no puede avanzar en el cauce (pipeline) es denominada atasco o “stall”.

Surgen 3 tipos de atascos:

- **Dependencia de datos:** cuando una instrucción necesita acceder a un registro pendiente de lectura o escritura por otra instrucción.
- **Dependencia de salto:** cuando una instrucción de salto determina que la siguiente del cauce no es la que se debe ejecutar.
- **Dependencia estructural:** cuando dos instrucciones intentan acceder a un mismo recurso simultáneamente (próxima práctica)

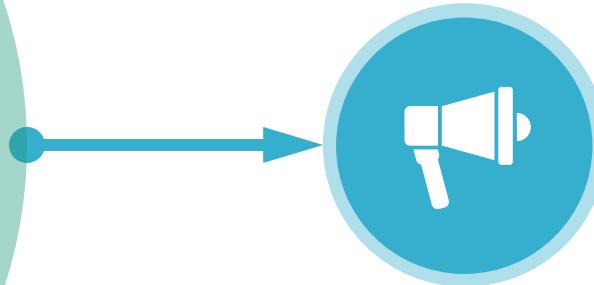
# Problemas y Soluciones del Cauce





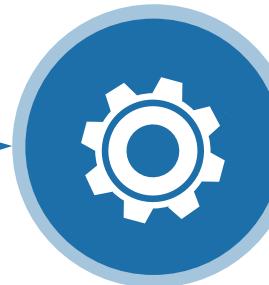
## RAW (Read After Write)

Cuando una instrucción necesita leer un registro que otra instrucción aún no escribió



## WAR (Write After Read)

Cuando una instrucción necesita escribir un registro que otra instrucción aún no leyó (próxima práctica)



## WAW (Write After Write)

Cuando una instrucción necesita escribir un registro que otra instrucción aún no escribió (próxima práctica)

# Atascos por Dependencia de Datos



Intercambio de  
valores de  
variables A y B

La instrucción LD con R2 escribe el registro en la etapa WB. Cualquier instrucción que intente leer el mismo registro antes de WB producirá un “atasco”

La instrucción SD con R2 lee el registro en la etapa ID. Si hay una instrucción que tiene pendiente escritura sobre el mismo registro producirá un “atasco”

En WB se escribe el registro al principio del ciclo y queda disponible para las demás instrucciones (no se espera a que la etapa concluya)

La instrucción SD intenta acceder a R2 en ID en el momento que la instrucción LD con R2 esta en EX. Se demora la ejecución en esta etapa hasta LD termine la ejecución de la etapa WB. Se producen 2 “raw stall”.

Importante: los registros se escriben al inicio de la etapa WB y quedan disponibles al inicio de otras etapas

# Atascos por Dependencia de Datos

Estos atascos en general se pueden resolver de 2 maneras

## Software

El programador o el compilador reordena las instrucciones para minimizar el número de atascos

## Hardware

Tenemos algún tipo de soporte de hardware que permite resolver algunas situaciones de dependencia de datos

# Atascos de Datos - Solución por Software

LD	R1, A(R0)
LD	R2, B(R0)
SD	R1, B(R0)
SD	R2, A(R0)



Intercambiamos  
de posición estas  
instrucciones  
para separar los  
accesos a R2

La instrucción SD intenta acceder a R1  
en ID en el momento que la instrucción  
LD con R1 esta en MEM. Se demora la  
ejecución en esta etapa hasta que LD  
con R1 comience la ejecución de la  
etapa WB. Se produce 1 “raw stall”

# Atascos de Datos - Solución por Hardware

- Como muchas veces los valores de los registros que requieren las instrucciones se encuentran en algún registro temporal esperando la etapa WB para ser escritos efectivamente
- Se modifica el hardware para utilizar o “adelantar” registros a las instrucciones que lo necesiten
- Ejemplos:
  - La instrucción LD deja disponible el valor de un registro al final de la etapa MEM
  - Las instrucciones aritméticas y lógicas dejan el valor disponible al final de la etapa EX
- “**Forwarding**” o adelanto de operandos brinda soporte por **hardware** para evitar en muchos casos los **atascos por dependencia de datos**.

# Atascos de Datos - Solución por Hardware

El forwarding o adelantamiento de registros modifica el cause que vimos anteriormente para que funcione de la siguiente manera:

- Facilita valores en registros temporales, antes que estén en registros efectivos. Si el valor está disponible en alguna etapa, no se atasca
- “Empuja” a las instrucciones a través del cauce hasta que no sea posible avanzar más. De esta manera permite que entren otras instrucciones en el cauce. Por ejemplo:
  - Las instrucciones aritméticas y lógicas acceden en ID a los operandos pero realmente los necesitan en EX. Puede pasar a EX donde se atascaría si el valor de los registros aún no están disponibles
  - La instrucción SD accede en ID al operando pero realmente lo necesita en la etapa MEM. Puede pasar a EX y luego a MEM donde se atascaría si aún no están disponibles.

# Atascos de Datos - Solución por Hardware

LD R1, A(R0)  
LD R2, B(R0)  
SD R2, A(R0)  
SD R1, B(R0)



Notar que:

- En el simulador WinMips, este programa se ejecuta con la opción “Enabling forwarding” que activa el funcionamiento descripto anteriormente
- El programa no tiene la optimización de software

La instrucción SD (R2) requiere el operando en ID pero no está disponible. Igualmente pasa a la etapa EX para resolver dir(A)+R0

La instrucción LD (R2) deja disponible el valor al final de la etapa MEM.

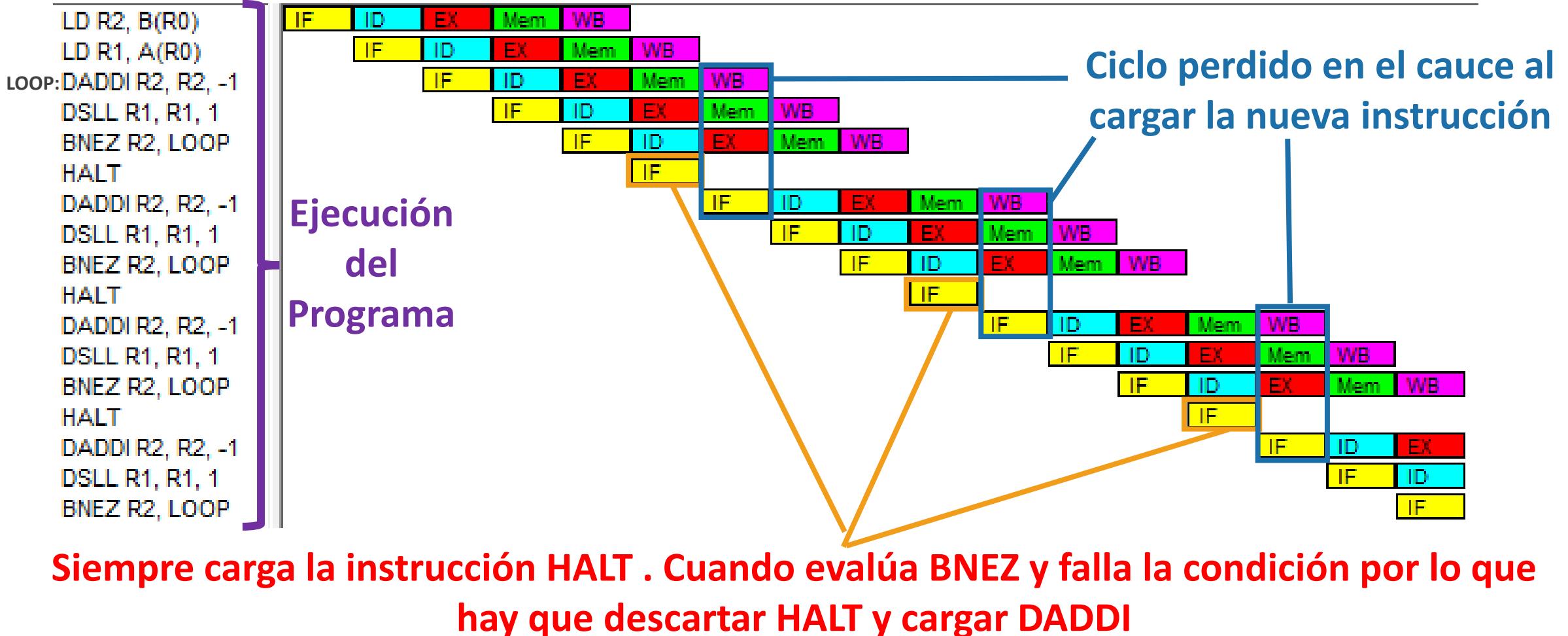
Cuando SD (R2) entra en la etapa MEM ya tiene disponible el registro dentro de la CPU, por lo que no se produce un atasco

# Atascos por Saltos - Problema

Análisis de instrucciones de salto:

- Cuando una instrucción de salto pasa a la etapa ID una nueva instrucción se carga en la etapa IF
- Cuando pasa a la etapa ID se carga una nueva instrucción en IF
- En la etapa ID se evalúa si la condición del salto es verdadera o no:
  - Si la condición de salto se cumple (hay que saltar), se descarta la instrucción en IF y se carga la instrucción de salto (“Branch taken stall”)
- Si tenemos un bucle esto puede suceder siempre, salvo la última vez. Se pierde un ciclo por cada iteración.

# Atascos por Saltos - Problema



El simulador tiene la opción “Forwarding” habilitada

# Atascos por Saltos - Solución

Los atascos por saltos implementan dos mecanismos por hardware (disjuntos) para minimizar este problema

## Branch Target Buffer

Modifica el hardware para implementar una tabla para “predecir” la instrucción que se debería ejecutar al saltar

## Delay Slot

Modifica el hardware para retardar el salto 1 instrucción , permitiendo la ejecución de la instrucción siguiente al salto

# Atascos por Saltos – Branch Target Buffer

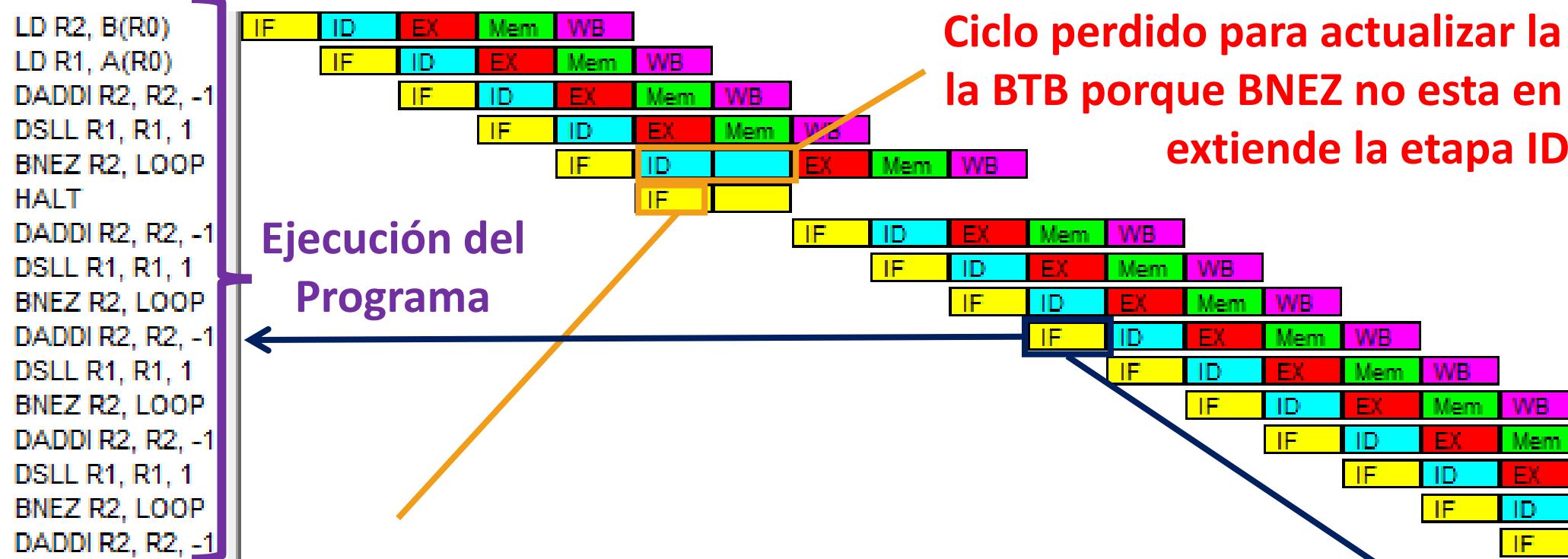
- Branch Target Buffer (BTB) introduce una modificación de hardware para intentar resolver el problema en la etapa IF
- Implementa una tabla de saltos en hardware con la información disponible en IF/ID
- La tabla contiene:
  - Dirección de la instrucción de salto:  
en IF se verifica si PC (contador del programa) esta guardado en la tabla, entonces se sabe que la instrucción es de salto sin decodificarla
  - Dirección de la instrucción a cargar en IF:  
almacena la dirección de la instrucción que se ejecutó en la ultima vez que se salto
  - Resultado de la condición del salto del último salto:  
Compara si coincide la condición del salto anterior y actual, de no ser así la instrucción cargada es la incorrecta

# Atascos por Saltos – Branch Target Buffer

## ➤ Funcionamiento de la Tabla:

- Cuando una instrucción de salto pasa a la etapa ID se carga en IF la instrucción que figura en la tabla. Si no figura en la tabla se carga en IF la siguiente instrucción al salto
- Cuando la dirección del salto no está en la tabla o no coincide la evaluación anterior:
  - Se descarta la instrucción cargada (se pierde un ciclo)
  - Se actualiza la tabla (se pierde otro ciclo)
  - Se carga la instrucción correspondiente
- Mejora: en un loop, solo se pierden 4 ciclos (2 al inicio y 2 al final), luego carga la dirección correcta siempre.

# Atascos por Saltos – Branch Target Buffer



Se carga HALT en vez de cargar la instrucción DADDI. Se perderá este ciclo porque se cargo la instrucción equivocada, pero en este momento no es posible saberlo

Ciclo perdido para actualizar la entrada de la BTB porque BNEZ no está en la tabla, se extiende la etapa ID

El simulador tiene la opción “Forwarding” y “Branch Target Buffer” habilitada

# Atascos por Saltos – Delay Slot

- Es una alternativa a la BTB (Branch Target Buffer) para evitar la pérdida de ciclos en las instrucciones de salto
- Consiste en retardar las instrucciones de salto 1 ciclo para ejecutar la instrucción siguiente y no descartarla del cauce.
- El ciclo no se pierde se cumpla o no la condición de salto.
- Esta forma de trabajar requiere reestructurar el código para asegurar que el programa se comporte normalmente:
- Solución simple: agregar NOP luego de cada instrucción de salto.  
Siempre perdemos un ciclo pero el programa funcionará normalmente.
- Solución “eficiente”: agregar alguna instrucción anterior a la de salto (en lo posible sin dependencia) para aprovechar el ciclo

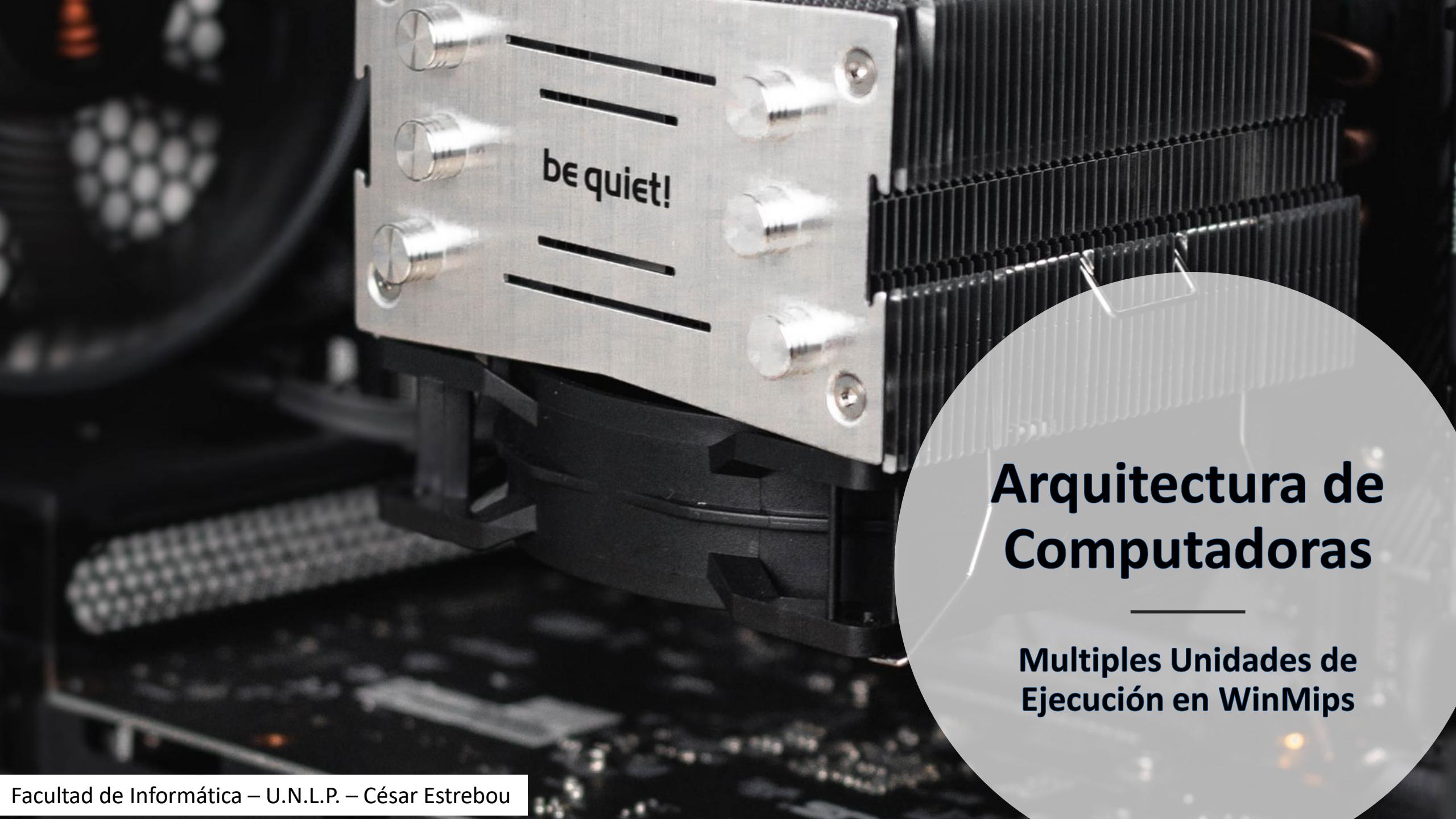
# Atascos por Saltos – Delay Slot

```
.data  
cant: .word 5  
datos: .word 1,2,3,4,5  
res: .word 0  
.code  
DADD R1, R0, R0  
LD R2, cant(R0)  
LOOP: LD R3, datos(R1)  
DADDI R2, R2, -1  
DSLL R3, R3, 1  
SD R3, res(R1)  
DADDI R1, R1, 8  
BNEZ R2, LOOP  
NOP  
HALT  
; Ejecución en 42 ciclos
```

Observar que para que el programa funcione normalmente es importante NOP.  
Si no el programa termina!

```
.data  
cant: .word 5  
datos: .word 1,2,3,4,5  
res: .word 0  
.code  
DADD R1, R0, R0  
LD R2, cant(R0)  
LOOP: LD R3, datos(R1)  
DADDI R2, R2, -1  
DSLL R3, R3, 1  
SD R3, res(R1)  
BNEZ R2, LOOP  
DADDI R1, R1, 8  
HALT  
; Ejecución en 37 ciclos
```

Selección de DADDI R1, R1, 8 para reemplazar NOP, ya que no tiene dependencias en el bucle



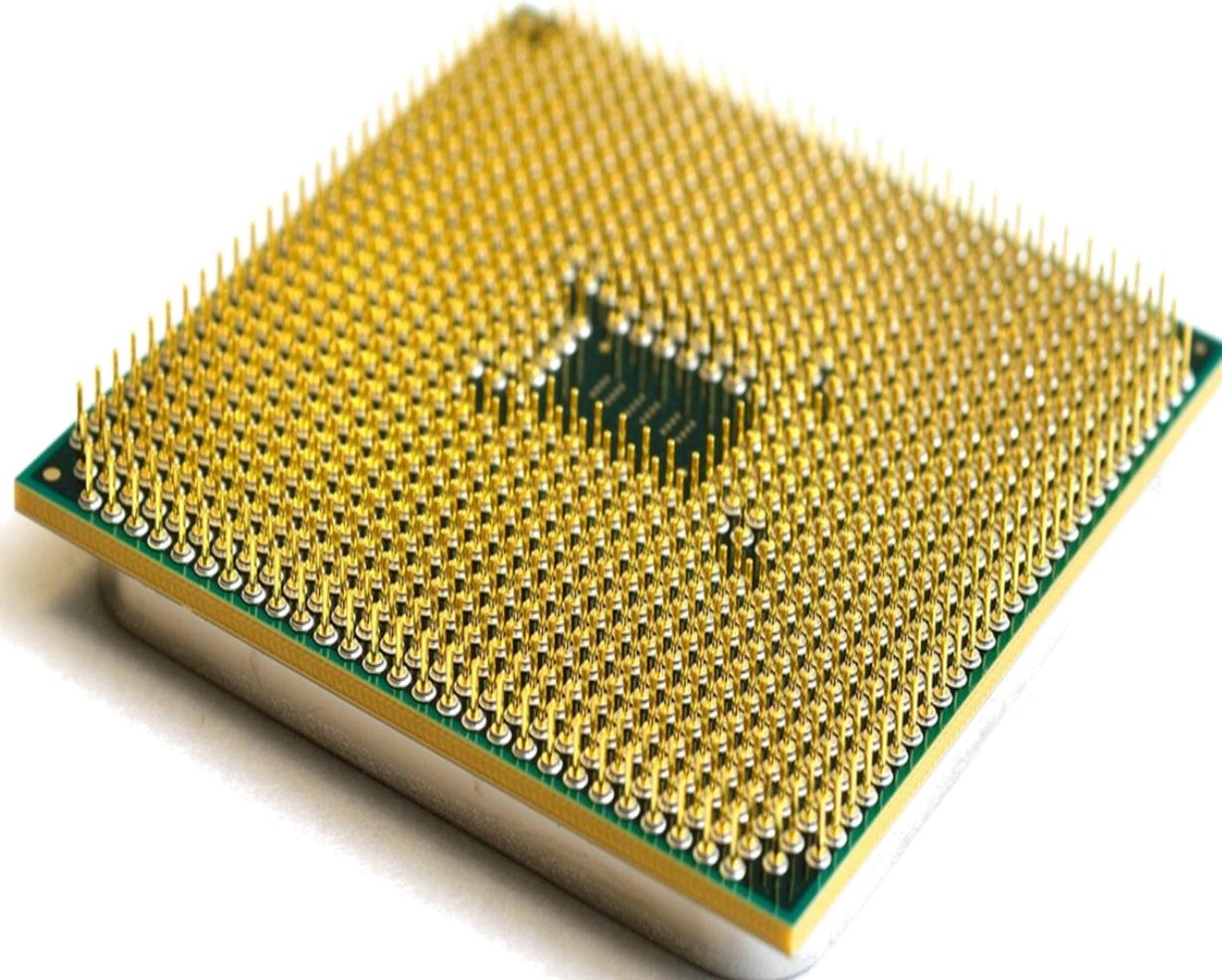
# Arquitectura de Computadoras

---

Multiples Unidades de  
Ejecución en WinMips

# Agenda

## ► Temas



### 01 Instrucciones de Transferencia

Variantes de Instrucciones de transferencia  
memoria/registro y registro/memoria

### 02 Punto Flotante

Instrucciones de punto flotante: aritméticas,  
transferencia, conversión y comparación. Flag FP

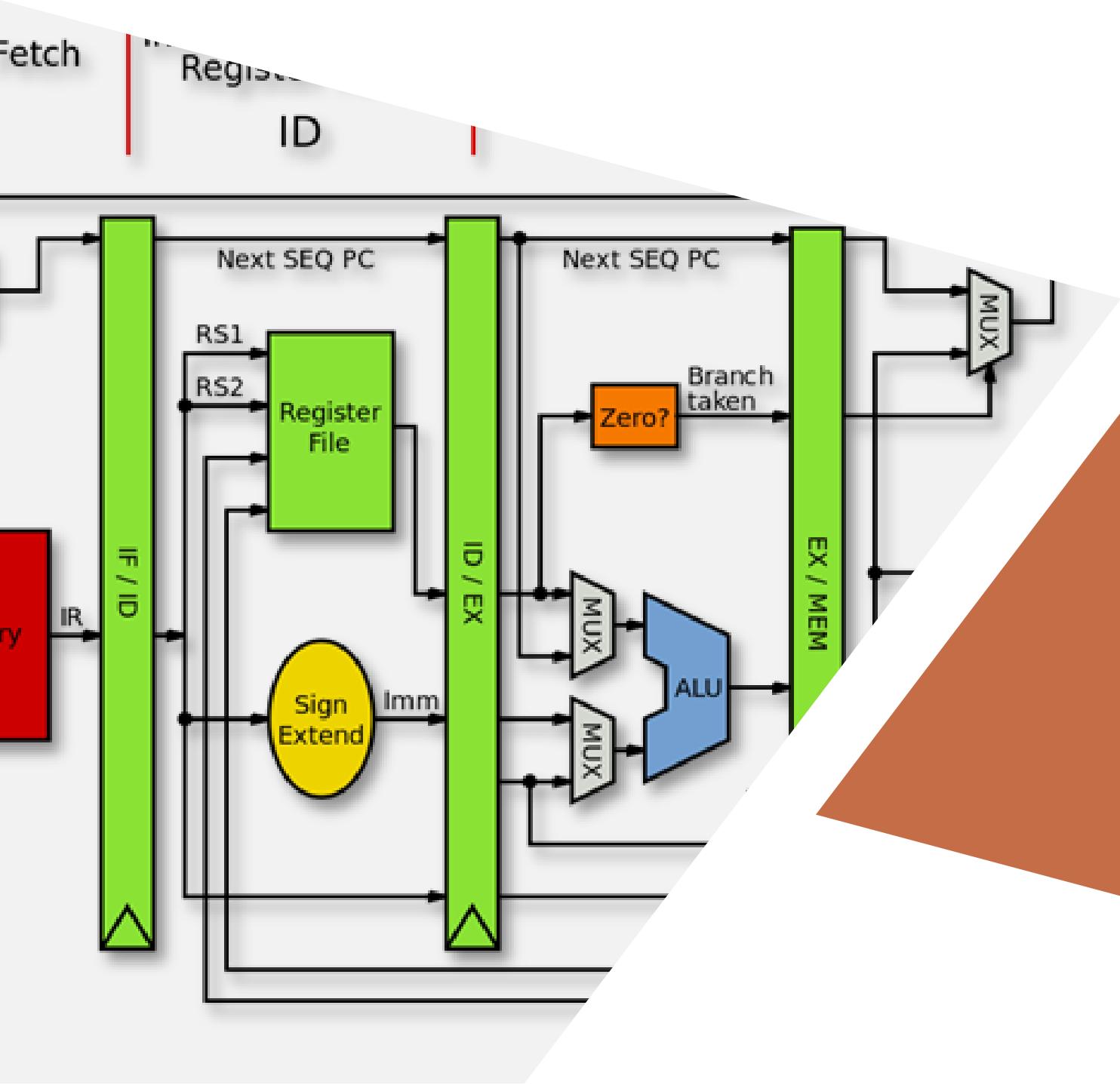
### 03 Unidades de Ejecución

Descripción y funcionamiento de unidades de  
ejecución para suma FP, multiplicación y división

### 04 Atascos WAR y WAW

Atascos de dependencia de datos WAR y WAW.  
Descripción. Condiciones para que sucedan. Ejemplos

# Instrucciones de Transferencia



# Transferencia de Memoria

- Las instrucciones LD y SD que transfieren de memoria/registro y de registro/memoria 8 bytes (64 bits)
- Winmips cuenta con instrucciones que permiten transferir 1, 2, o 4 bytes
- Las instrucciones de carga escriben los 64 bits del registro y deben completar los bits faltantes cuando el dato no los tiene
- Es necesario indicar como completar los bits del registro:
  - Si se indica que la transferencia es de números sin signo, se completan con ceros.
  - Si se indica que la transferencia es de números con signo, se completan con el valor del bit mas significativo (1 negativo, 0 positivo)

# Transferencia de Memoria - Ejemplos

Representación de números sin signo (BSS):

- Solo para representación de números positivos
- De 0 a  $2^n - 1$  números en decimal para n bits
- Para el numero 255:
  - 8 bits → **11111111**
  - 16 bits → 00000000 **11111111**
  - 32 bits → 00000000 00000000 00000000 **11111111**
  - 64 bits → 00000000 00000000 00000000 00000000 00000000 00000000 **11111111**

La solución es completar los bits faltantes con 0

# Transferencia de Memoria - Ejemplos

Ejemplos:

- LD R1, dato(R2)
  - Load Doble Word (64 bits) ➔ no agrega bits
- LBU R1, dato(R2)
  - Load Byte sin signo (8 bits) ➔ agrega 56 bits significativos en 0
- LHU R1, dato(R2)
  - Load Halfword sin signo (16 bits) ➔ agrega 48 bits significativos en 0
- LWU R1, dato(R2)
  - Load Word sin signo (32 bits) ➔ agrega 32 bits significativos en 0

Notar la letra **U** de unsigned en el nombre de la instrucción

# Transferencia de Memoria - Ejemplos

Representación de números con signo (CA2):

- Representación de números negativos y positivos
- De  $-2^n$  a  $2^n-1$  números en decimal para n bits
- Hay que tener en cuenta que la representación de menos bits también debe estar en CA2
- Si el número original es BSS, hay que tener en cuenta que el rango de los números a convertir va de 0 a  $2^{n-1}-1$ . Las representaciones que usen el bit más significativo dan resultados incorrectos

# Transferencia de Memoria - Ejemplos

## Representación de números con signo (CA2):

➤ Para el numero 127:

- 8 bits → 01111111
- 16 bits → 00000000 01111111
- 32 bits → 00000000 00000000 00000000 01111111
- 64 bits → 00000000 00000000 00000000 00000000 00000000 00000000 00000000 01111111

➤ Para el numero -128:

- 8 bits → 10000000
- 16 bits → 11111111 10000000
- 32 bits → 11111111 11111111 11111111 10000000
- 64 bits → 11111111 11111111 11111111 11111111 11111111 11111111 11111111 10000000

La solución es completar los bits con el bit mas significativo

# Transferencia de Memoria - Ejemplos

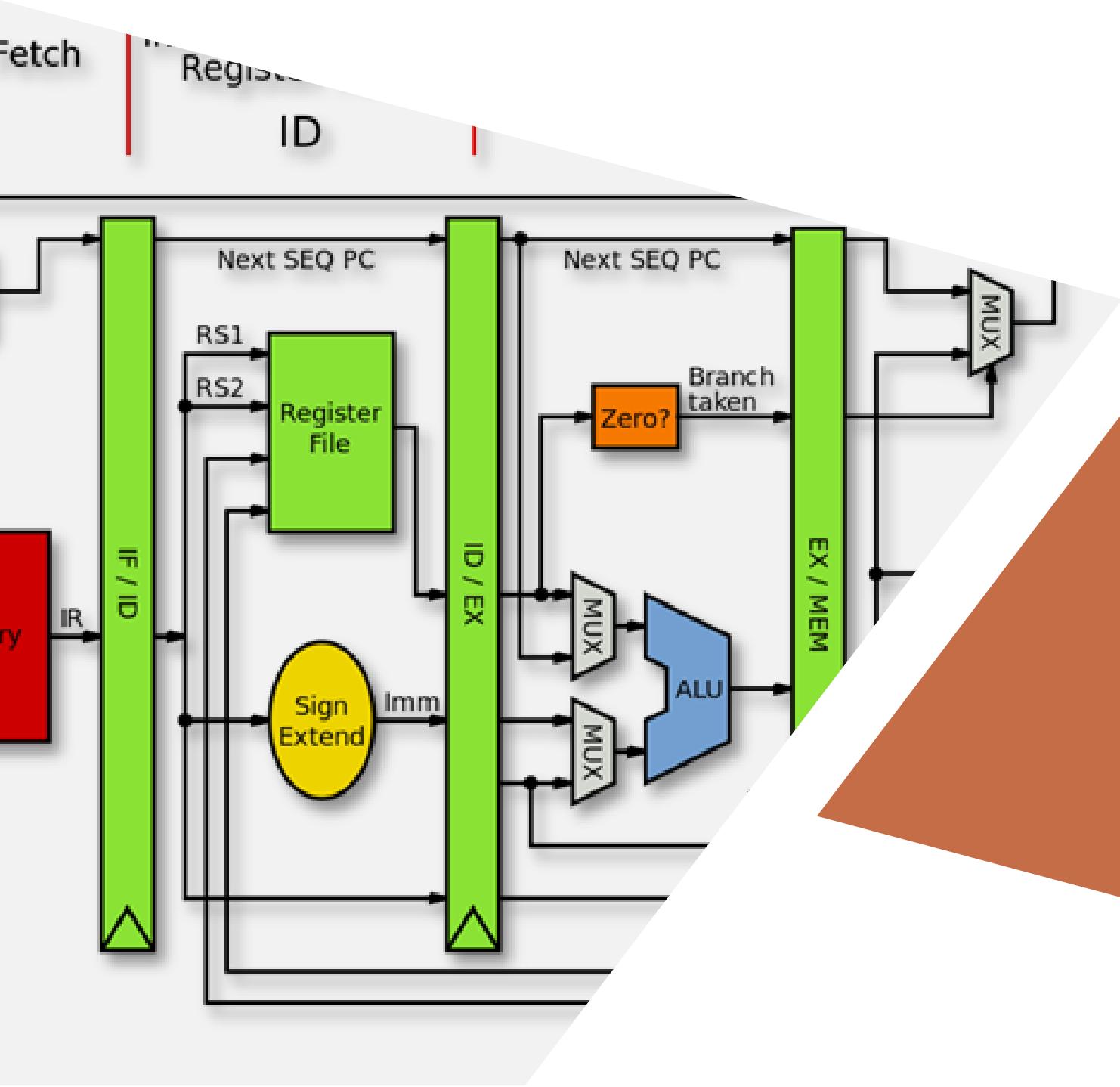
Ejemplos:

- LD R1, dato(R2)
  - Load Doble Word (64 bits) ➔ no agrega bits (no importa el signo)
- LB R1, dato(R2)
  - Load Byte con signo (8 bits) ➔ agrega 56 bits según signo
- LH R1, dato(R2)
  - Load Halfword con signo (16 bits) ➔ agrega 48 bits según signo
- LW R1, dato(R2)
  - Load Word con signo (32 bits) ➔ agrega 32 bits según signo

# Transferencia de Memoria

- Las instrucciones de almacenamiento SD truncan (recortan) el registro.
- Solo transfieren los bits menos significativos.
- Para la representación en CA2, el bit mas significativo de los copiados (el de signo) se conserva, por lo que no es necesario indicar si es con o sin signo.
- Ejemplos:
  - SD R1, dato(R2) ; Store Doubleword (64 bits)
  - SB R1, dato(R2) ; Store Byte (8 bits)
  - SH R1, dato(R2) ; Store Halfword (16 bits)
  - SW R1, dato(R2) ; Store Word (32 bits)

# Punto Flotante



# Punto Flotante

- Disponemos de 32 registros de doble precisión (64 bits) para trabajar con operaciones de punto flotante.
- Los denominamos  $f\{i\}$  donde  $i$  esta entre 0 y 31.
- Disponemos de 3 unidades de ejecución paralelas para realizar las operaciones de:
  - Suma o Resta
  - Multiplicación
  - División

# Punto Flotante - Instrucciones

L.D {reg. flot. origen}, {mem. destino}({reg. ent.})

- Copiar un valor de memoria a un registro

S.D {reg. flot. destino}, {mem. origen}({reg. ent.})

- Copiar un valor de un registro en memoria

ADD.D {reg. Destino}, {reg. Oper. 1}, {reg. Oper. 2}

- Suma 2 registros y guarda en el registro destino

SUB.D {reg. Destino}, {reg. Oper. 1}, {reg. Oper. 2}

- Resta Oper. 2 a Oper. 1 y guarda en el registro destino

MUL.D {reg. Destino}, {reg. Oper.1}, {reg. Oper. 2}

- Multiplica 2 registros y guarda en el registro destino

DIV.D {reg. Destino}, {reg. Oper.1}, {reg. Oper. 2}

- Divide Oper. 1 por Oper. 2 y guarda en el registro destino

# Punto Flotante - Instrucciones

MOV.D {reg. flot. destino}, {reg. flot. origen}

- Copia el registro flotante origen en el destino

MTC1 {reg. ent. origen}, {reg. flot. destino}

- Copia los 64 bits del reg. ent. origen en los del reg. flot. Destino

MFC1 {reg. ent. destino}, {reg. flot. origen}

- Copia los 64 bits del reg. flot. origen en los del reg. ent. destino

CVT.D.L {reg. flot. destino}, {reg. flot. origen}

- Convierte el valor flotante del reg. origen a su representación entera de 64 bits y los almacena en el reg. destino

CVT.L.D {reg. flot. destino}, {reg. flot. origen}

- Interpreta los 64 bits del reg. flot. origen como un número entero , lo convierte su representación en punto flotante y lo guarda en reg. flot. destino

# Punto Flotante - Instrucciones

Existe un flag de estado, denominado FP, asociado a instrucciones de comparaciones y saltos en punto flotante

C.LT.D {reg. flot.}, {reg. flot.}

- Pone 1 en FP si el 1er reg. es menor que el 2do. Sino pone 0

C.LE.D {reg. flot.}, {reg. flot.}

- Pone 1 en FP si el 1er reg. es menor o igual que el 2do . Sino pone en 0

C.EQ.D {reg. flot.}, {reg. flot.}

- Pone en 1 FP si los registros son iguales. Sino pone en 0

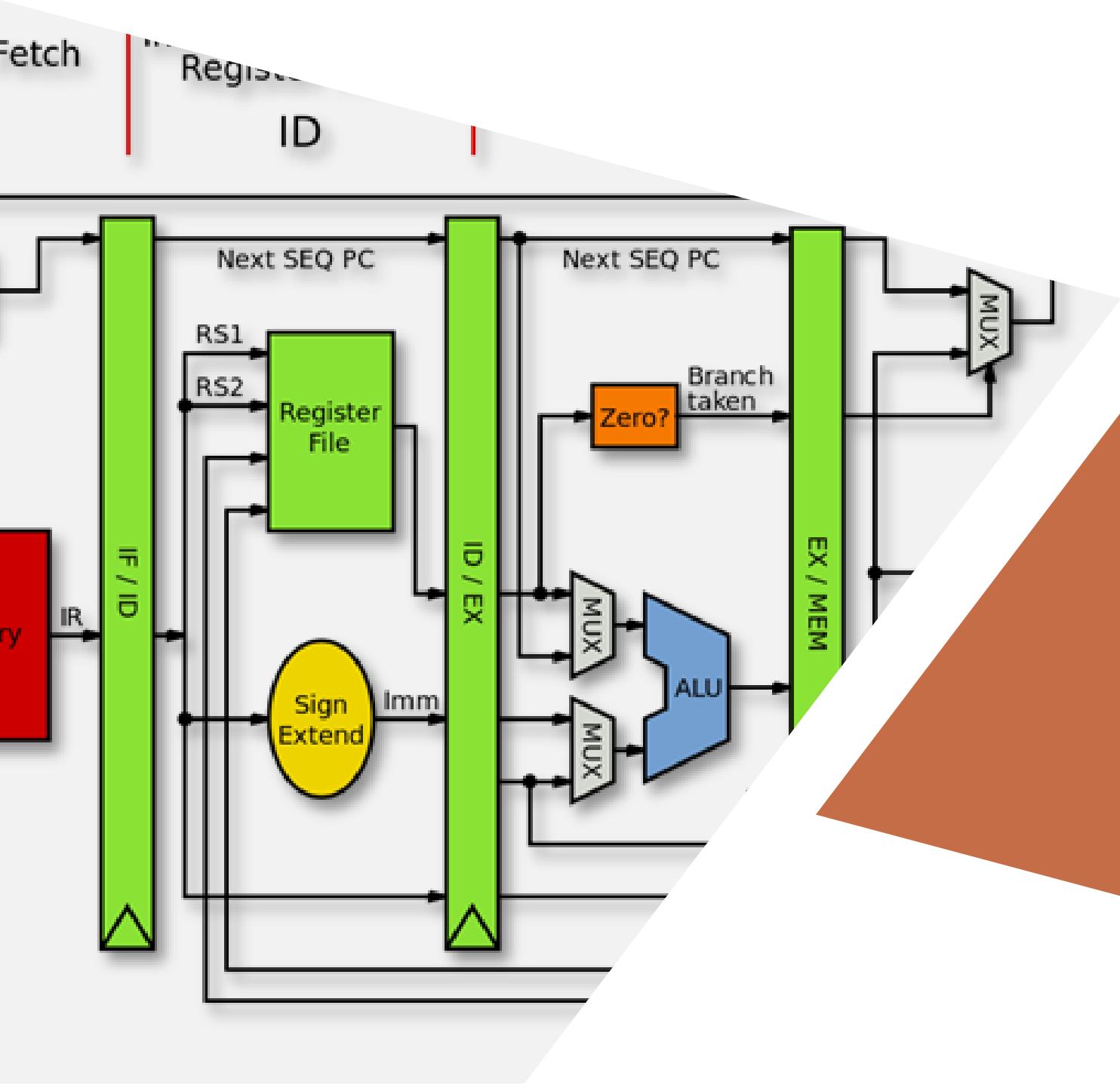
BC1F {dir. destino}

- Salta a la dir. destino si el flag FP esta en 0

BC1T {dir. destino}

- Salta a la dir. destino si el flag FP esta en 1

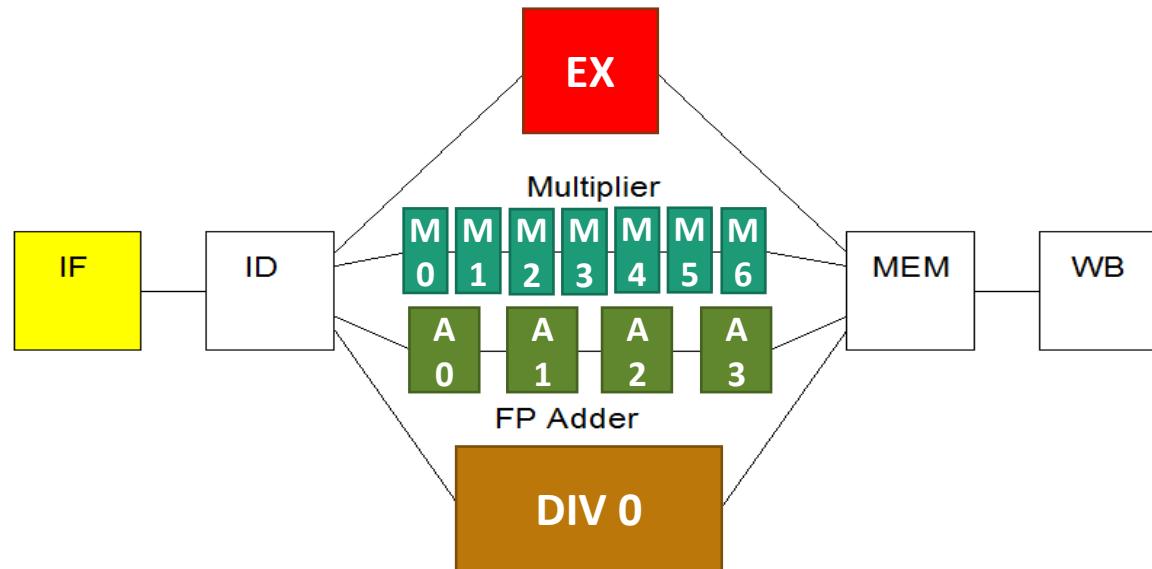
# Múltiples Unidades de Ejecución



# Unidades de Ejecución

Winmips tiene 4 unidades de ejecución:

1. Para instrucciones generales (EX, 1 etapa de 1 ciclo)
2. Para multiplicar (Multiplier, 7 etapas de 1 ciclo c/u)
3. Para sumar punto flotante (FP Adder, 4 etapas de 1 ciclo c/u)
4. Para dividir (DIV 0, 1 etapa de 24 ciclos)

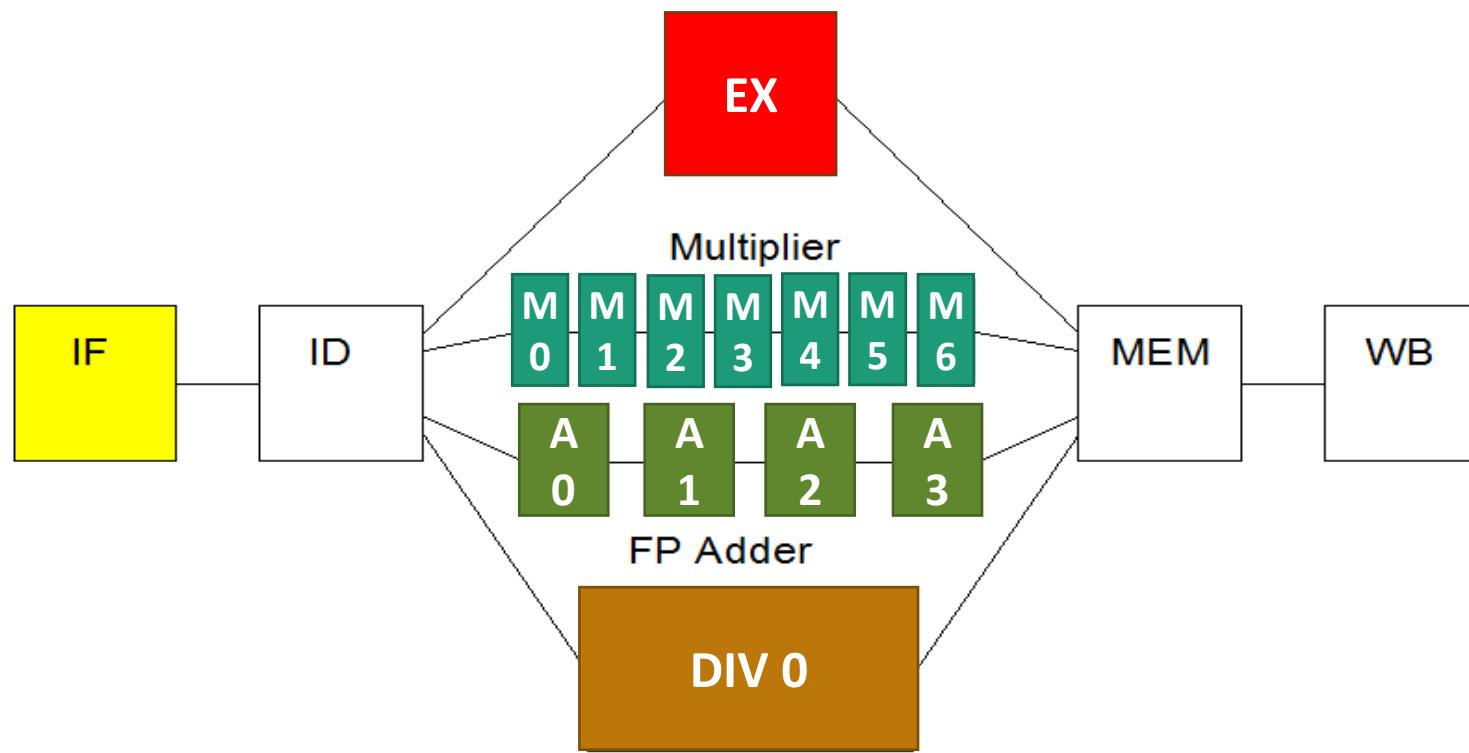


# Unidades de Ejecución

- Cada etapa de una unidad de ejecución puede tener una instrucción.
- Los operandos de una instrucción son requeridos al inicio de la primer etapa de cada unidad de ejecución. Si no están disponibles se producirá un atasco RAW
- El resultado de una instrucción se producirá al final de la última etapa de cada unidad de ejecución. Al final de esta etapa pueden adelantarse los operandos si se utiliza “forwarding”

# Unidades de Ejecución

Instrucciones por unidades de ejecución



EX, 1 ciclo

DADD, DSUB, AND, LD, SD,  
L.D, S.D, MOV.D, C.LT.D, etc.

Multiplicador, 8 ciclos

DMUL, MUL.D

Sumador PF, 4 ciclos

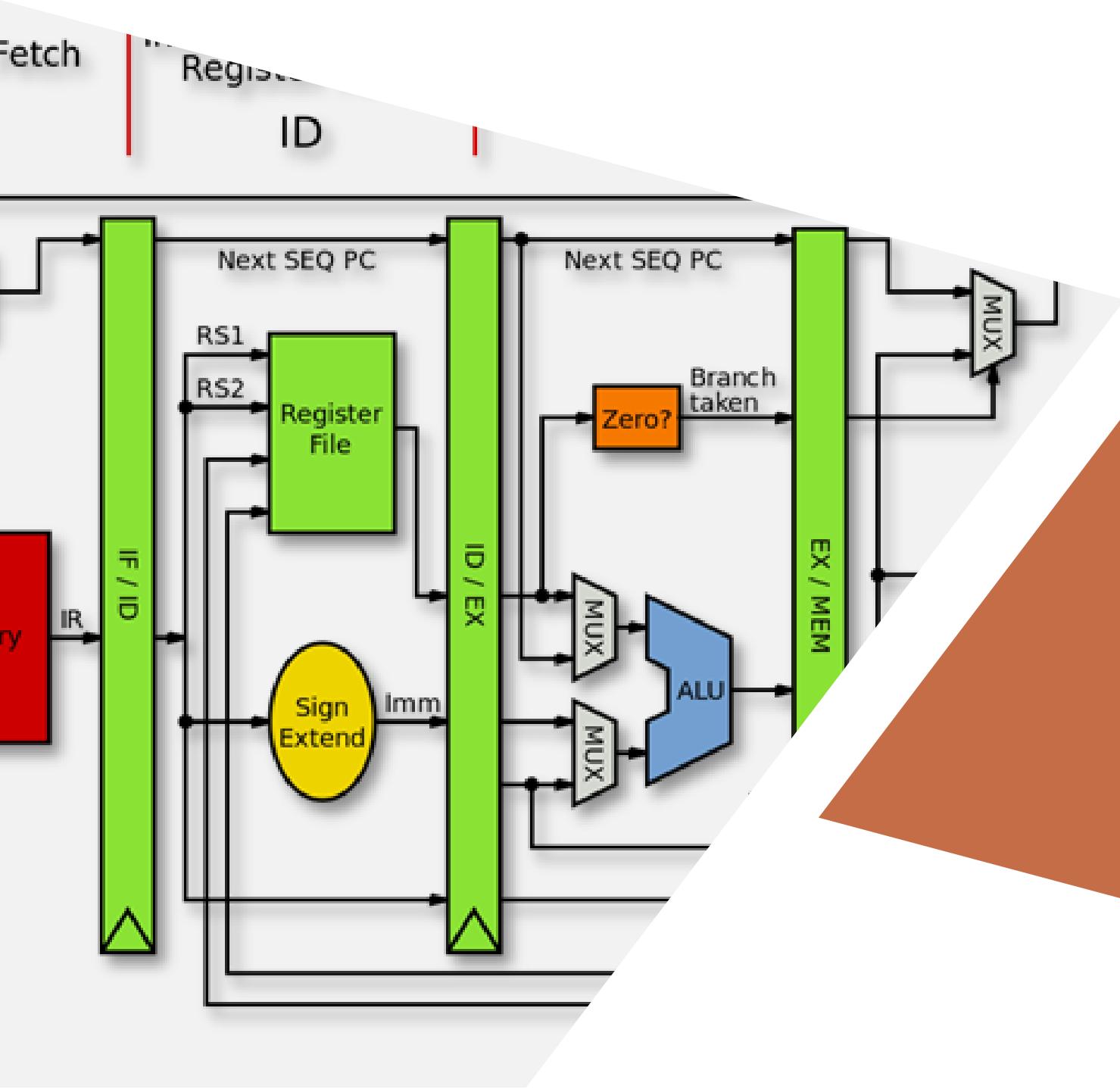
ADD.D, SUB.D

Divisor , 24 ciclos

DDIV, DIV.D

# Unidades de Ejecución

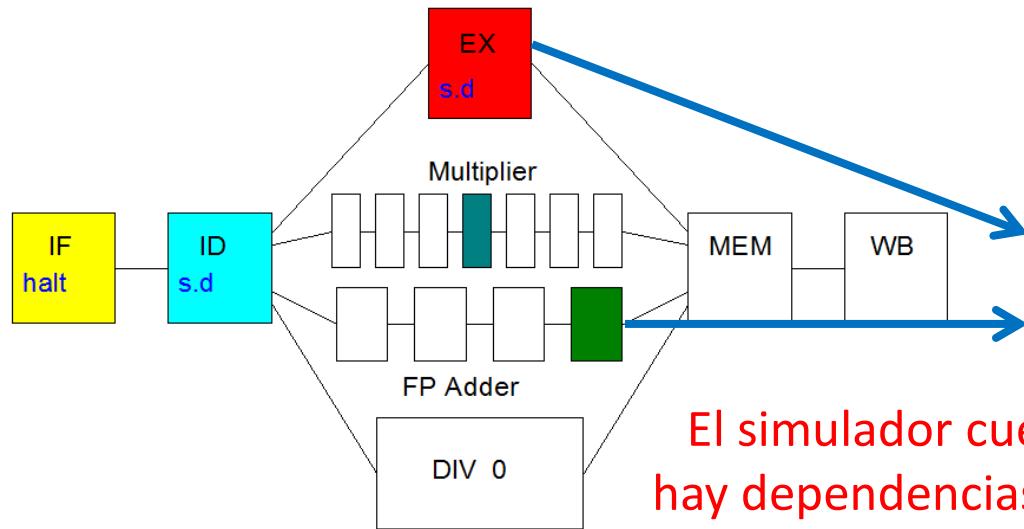
- Tener múltiples unidades de ejecución permite ejecutar mas instrucciones en el mismo tiempo.
- El uso de múltiples unidades tiene sus ventajas pero introduce nuevos problemas.
- Aparecen 3 tipos de atascos:
  - Dependencia Estructural
  - Dependencia de datos WAR
  - Dependencia de datos WAW



# Atascos Estructurales y WAR/WAW

# Atascos Estructurales

- Los atascos estructurales son provocados por conflictos por los recursos (acceder a una misma etapa del cauce)
- En Winmips puede suceder cuando dos instrucciones en unidades de ejecución distintas intentan acceder a la etapa memoria simultáneamente

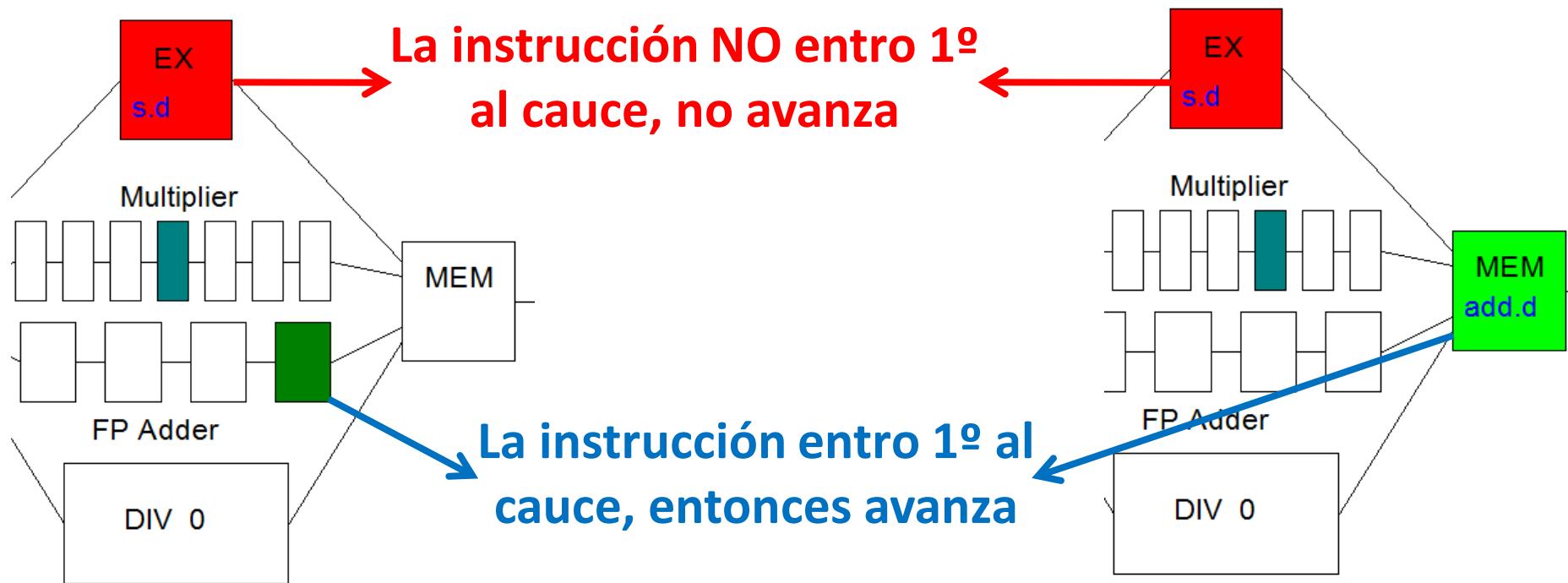


Dos instrucciones listas para pasar a la etapa de memoria. Se produce un atasco solo pasa 1

El simulador cuenta un “Structural Stalls” sin tener en cuenta si hay dependencias de datos. Aunque una instrucción este atascada en una unidad y no pueda avanzar igualmente lo cuenta

# Atascos Estructurales

- Cuando sucede un atasco estructural solo una de las instrucciones avanza
- Tiene prioridad la primera instrucción que entro en el cauce



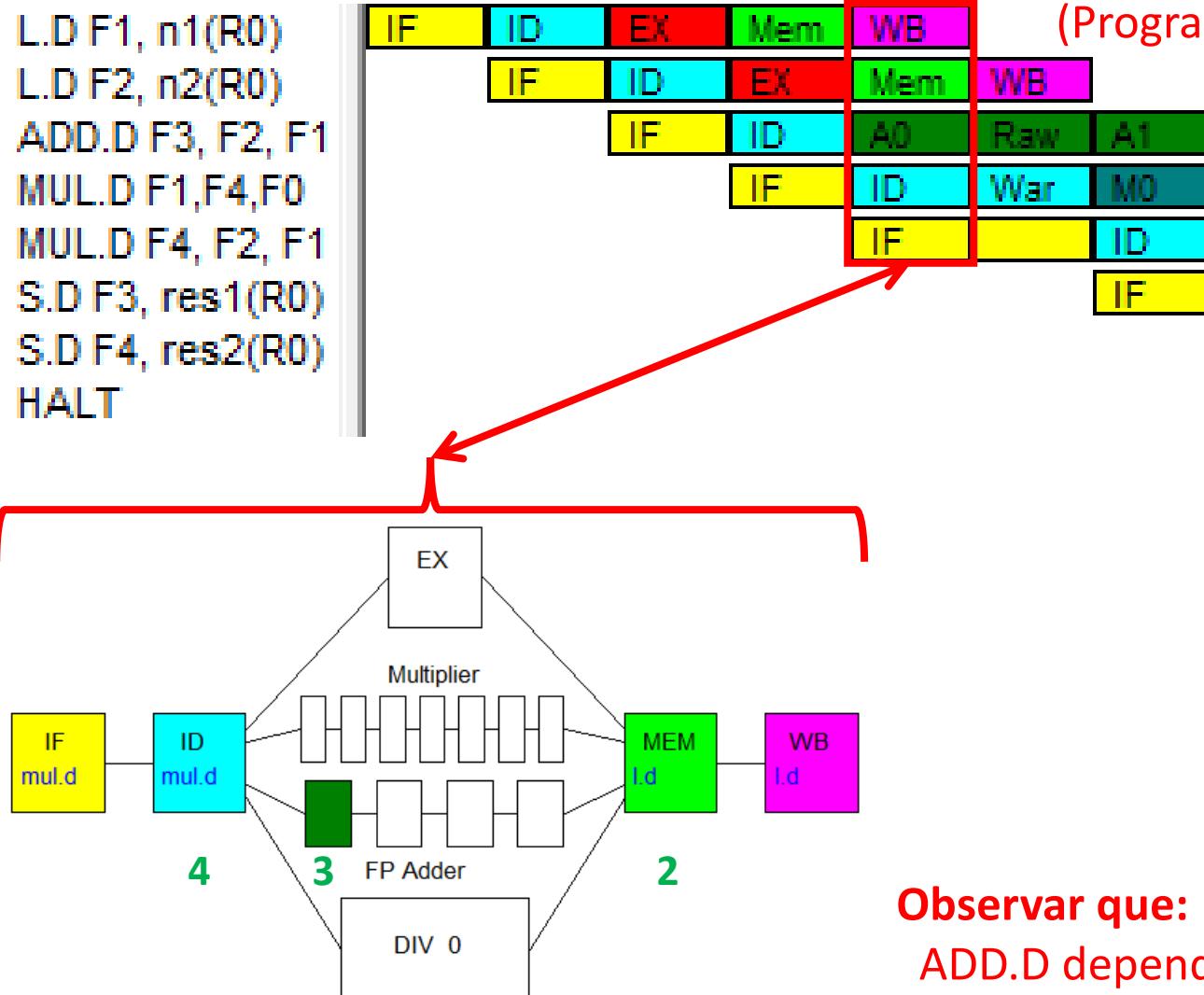
# Atascos WAR y WAW

Los atascos WAR y WAW suceden cuando:

- hay dependencia de datos entre dos instrucciones (igual que RAW)
- existe mas de una unidad de ejecución
- una instrucción que entra al cauce puede sobrepasar a una instrucción anterior, escribiendo un registro pendiente de lectura (WAR) o escritura (WAW)
- el simulador produce atascos cuando detecta una situación potencial de dependencia WAR/WAW
- muchas veces si la instrucción continuara el atasco WAR/WAW no sucede. Bloquea preventivamente ya que no tiene hardware que determine cuanto tarda una instrucción en cada unidad de ejecución

# Atascos WAR y WAW

- 1 L.D F1, n1(R0)
- 2 L.D F2, n2(R0)
- 3 ADD.D F3, F2, F1
- 4 MUL.D F1,F4,F0
- 5 MUL.D F4, F2, F1
- 6 S.D F3, res1(R0)
- 7 S.D F4, res2(R0)
- 8 HALT



(Programa ejecutado con forwarding habilitado)

## Situación 1:

- ADD.D (3) depende de F2
- ADD.D necesita F2 al inicio de A0
- L.D F2 (2) carga F2 al final de Mem

**Resultado:** RAW por F2 de ADD.D

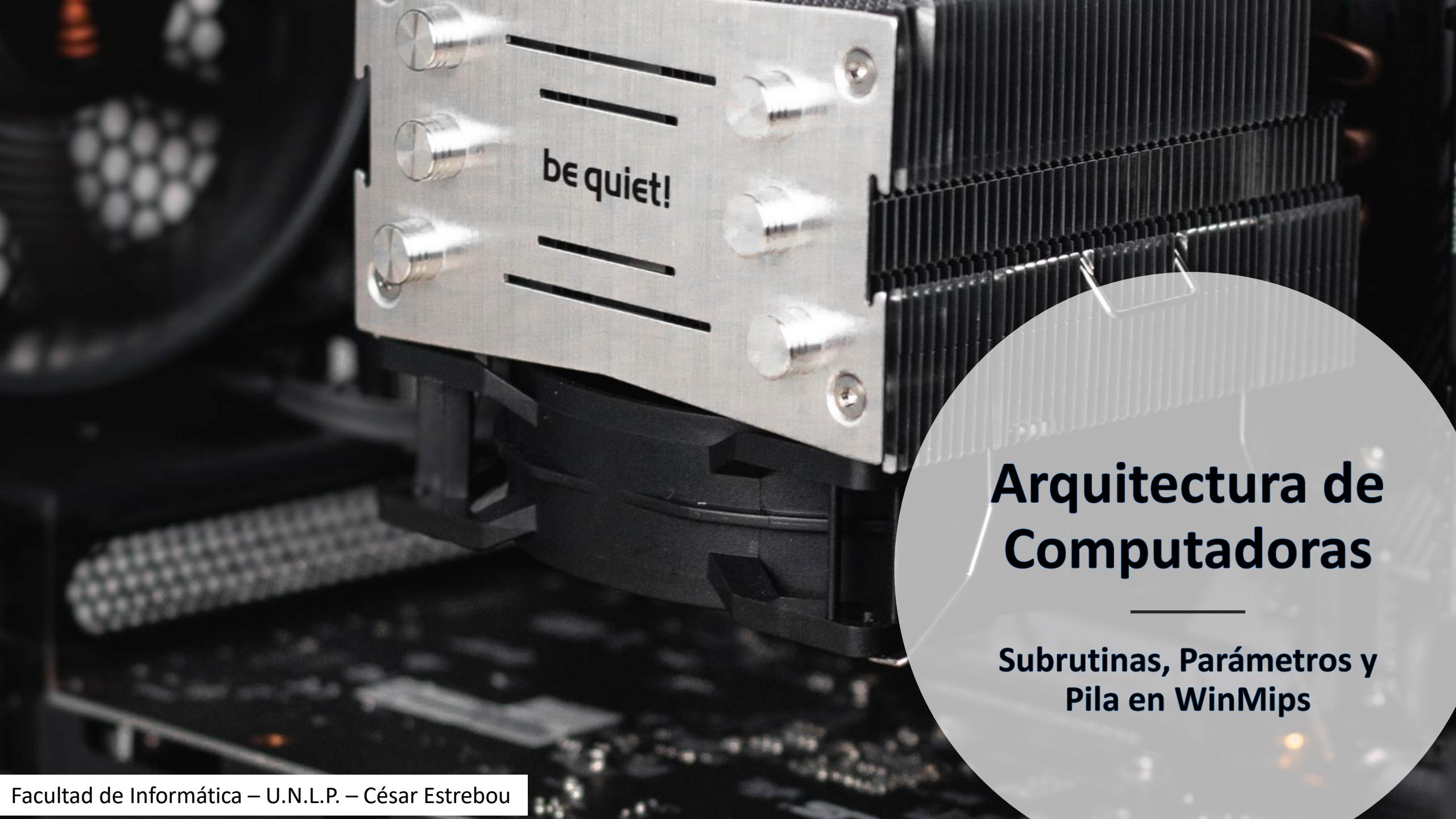
## Situación 2:

- MUL.D (4) escribirá F1
- ADD.D que debe leer F1 generará RAW por F2 de L.D y no avanzará
- El simulador no tiene forma de saber cuantos ciclos RAW provocará ADD.D

**Resultado:** WAR (potencial) por F1 de ADD.D por prevención

## Observar que:

ADD.D depende de L.D y provoca 1 raw, pero si dependiera de DIV provocaría 24 raw

A close-up photograph of a computer's central processing unit (CPU) cooler. The cooler is made of a dark metal with a ribbed heat sink and a large, black, circular fan. A silver metal bracket is attached to the side of the heat sink, featuring the 'be quiet!' logo in a stylized font. The background is dark and out of focus.

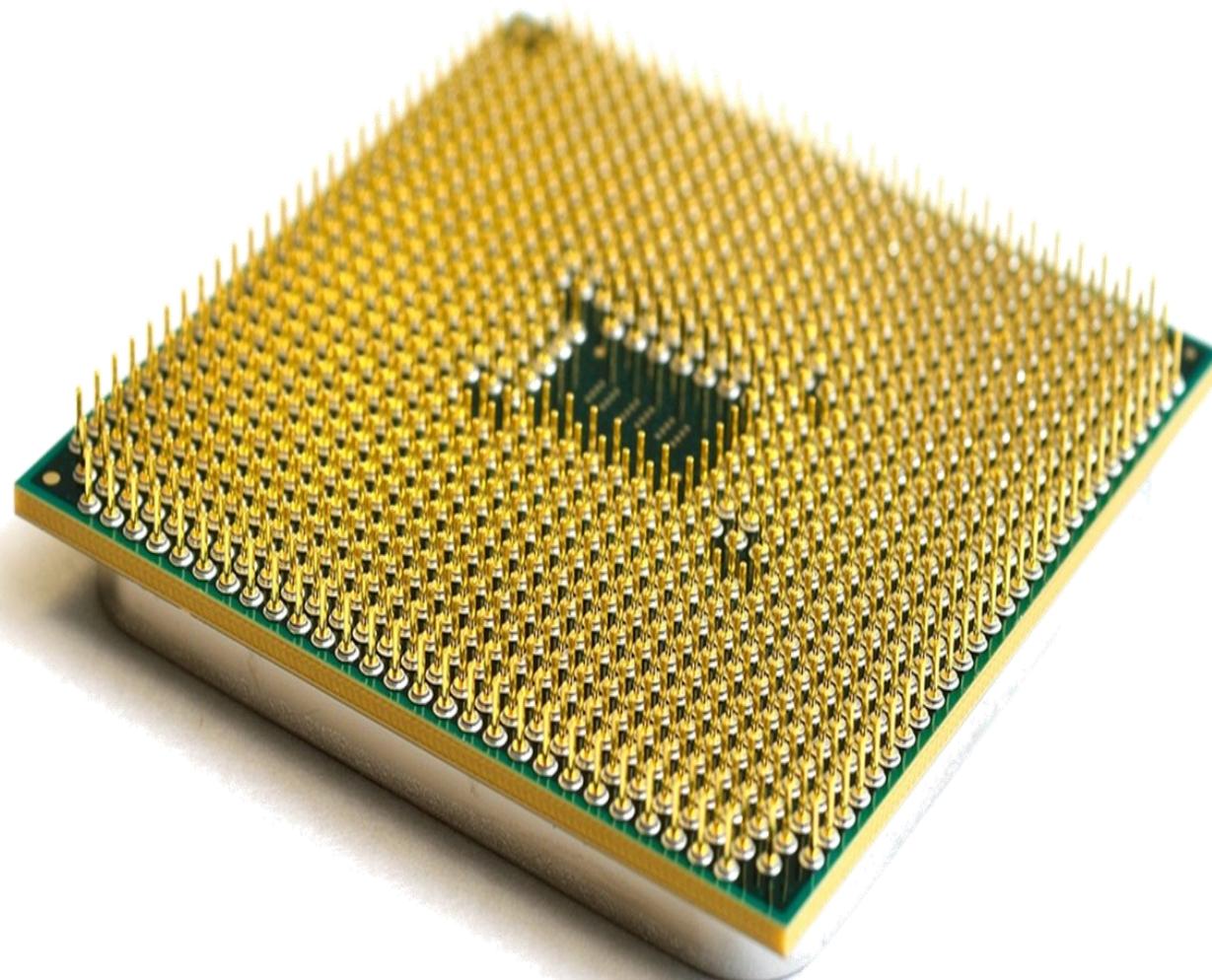
# Arquitectura de Computadoras

---

Subrutinas, Parámetros y  
Pila en WinMips

# Agenda

## ► Temas



### 01 Introducción

Características Generales. Arquitectura. Registros.  
Directivas del Compilador

### 02 Instrucciones

Formato de 3 operandos. Instrucciones de acceso a memoria, aritméticas, lógicas y de control

### 03 Causa/Pipeline

Funcionamiento. Tareas de cada Etapa del Cauce.  
Ciclos por Instrucción (CPI). Desglose de instrucciones típicas por etapas

### 04 Atascos

Problemas del Cauce. Atascos de Datos RAW, WAR y WAW. Adelantamiento de Operandos. Atascos por saltos. Branch Target Buffer y Salto retardado

```
.data
nt: .word 0xF987
es: .asciiz "Hello World"
ey: .asciiz "Press any key to continue"

ub: .double 32.786 ; a double precision number
: .byte 0 ; coordinates
: .byte 0
ol: .byte 255,0,255,0 ; the colour magenta

CONTROL: .word32 0x10000
ATA: .word32 0x10008

.text

lwu $t8,DATA($zero) ; $t8 = address of DATA register
lwu $t9,CONTROL($zero) ; $t9 = address of CONTROL register

daddi $v0,$zero,1 ; set for unsigned integer
ld $t1,int($zero)
sd $t1,0($t8) ; write integer to memory
sd $v0,0($t9) ; write to CONTROL register

daddi $v0,$zero,2 ; set for signed integer
ld $t1,int($zero)
sd $t1,0($t8) ; write integer to memory
sd $v0,0($t9) ; write to CONTROL register

daddi $v0,$zero,3 ; set for floating point
l.d f1,dub($zero)
s.d f1,0($t8) ; write floating point to memory
```

# Subrutinas, Parámetros y Convenciones

# Subrutinas - Conceptos

Mecanismo de llamado:

- MSX88 → **CALL {dirección subrutina}**
- WINMIPS → **JAL {dirección subrutina}**

Mecanismo de retorno:

- MSX88 → **RET**
- WINMIPS → **JR R31** ; JAL deja la dir. de retorno en R31

Parámetros por registro:

- MSX88 → **AX, BX, CX, DX**
- WINMIPS → **R1 a R30**

Parámetros por pila:

- MSX88 → **PUSH {reg. 16 bits}, POP {reg. 16 bits}**
- WINMIPS → **No hay pila, pero simulamos PUSH y POP**

# Subrutinas - Parámetros por Valor

Parámetros por valor:

- MSX88 → **MOV {reg. Destino}, {dir. memoria}**  
**MOV {reg. Destino}, {valor literal}**
- WINMIPS → **LD {reg. Destino}, {dir. memoria}(r0)**  
**DADDI {reg. Destino}, R0, {valor literal}**

Ejemplos:

**LD r1, dato(r0)** →  $r1 = \underbrace{[ \text{dir. dato} + r0 ]}_{\text{referencia}}$  contenido

**DADDI r1, r0 , 5** →  $r1 = 5 + r0 = 5$

# Subrutinas - Parámetros por Referencia

Parámetros por referencia:

- MSX88 → **MOV {reg. Destino}, OFFSET {dir. memoria}**
- WINMIPS → **DADDI {reg. Destino}, r0, {dir. memoria}**

Ejemplo:

$$\text{DADDI } r1, \text{ r0, dato} \rightarrow r1 = \underbrace{\text{dir. dato + r0}}_{\text{referencia}}$$

# Subrutinas - Ejemplo

**.data**

**result: .word 0**

**.text**

**daddi r4, r0, 10 ; Primer parámetro en r4**

**daddi r5, r0, 20 ; Segundo parámetro en r5**

**jal sumar ; Se invoca a la subrutina “sumar”**

**sd r2, result(r0) ; Resultado de “sumar” en r2**

**halt**

sumar: **dadd r2, r4, r5 ; subrutina “sumar” (r31 = dir. Retorno)**

**jr r31 ; Retorna a la instr. sig. al “JAL sumar”**

```
.data
nt: .word 0xF987
es: .asciiz "Hello Worl
ey: .asciiz "Press any key to c

ub: .double 32.786 ; a double
:
:
ol: .byte 255,0,255,0 ; the colour magenta

ONTROL: .word32 0x10000
ATA: .word32 0x10008

.text

lwu $t8,DATA($zero) ; $t8 = address of DATA register
lwu $t9,CONTROL($zero) ; $t9 = address of CONTROL register

daddi $v0,$zero,1 ; set for unsigned integer
ld $t1,int($zero)
sd $t1,0($t8) ; write integer to DATA register
sd $v0,0($t9) ; write to CONTROL register

daddi $v0,$zero,2 ; set for signed integer
ld $t1,int($zero)
sd $t1,0($t8) ; write integer to DATA register
sd $v0,0($t9) ; write to CONTROL register

daddi $v0,$zero,3 ; set for floating point
l.d f1,dub($zero)
s.d f1,0($t8) ; write floating point to DATA register
```

# Convenciones y Pila

# Subrutinas – Problemas/Dificultades

**Al utilizar subrutinas pueden surgir algunas dificultades y/o problemas que debemos solucionar. Estos son:**

- 1. Caos en el uso de registros:** Podemos usar “todos” los registros para cualquier tarea. Es necesario mantener el “orden” reservando registros para funciones específicas. Ejemplo: parámetros, cálculos, uso de pila, etc.
- 2. Nombres de registros poco significativos:** al reservar registros para distintas funciones sería interesante darles nombres significativos. Si usamos r29 como puntero de pila podríamos denominarlo SP (Stack Pointer)

# Subrutinas - Problemas

**Al utilizar subrutinas pueden surgir algunas dificultades y/o problemas que debemos resolver. Estos son:**

**3. Anidamiento de subrutinas:** Cuando el programa principal invoca a una subrutina deja en r31 la dirección de retorno. Invocar una nueva subrutina desde esta deja la dirección de retorno en r31, sobrescribiendo la dirección anterior. Es necesario emplear un mecanismo para permitir los anidamientos.

**4. Efectos laterales:** Cuando se invoca una subrutina es posible que esta utilice registros que estén en uso desde donde se la llama. Es necesario un mecanismo que preserve los valores de los registros.

# Subrutinas - Convenciones

Para resolver estos problemas/dificultades adoptamos convenciones:

- Agrupar registros para funciones específicas y nombrarlos según esta función:
  - Usamos el mismo criterio y los nombres que usa el compilador de C para MIPS.
- Implementar una pila:
  - Destinar un registro como puntero de pila e implementar las operaciones de “PUSH” y “POP”.
- Utilizar la pila para:
  - Asegurar la dirección de retorno antes de invocar una subrutina.
  - Asegurar los valores de registros (según la convención de C) antes de invocar a una subrutina.

# Subrutinas - Convención, Registros

Registro	Nombre	Uso	Ref.
r0	\$zero o \$0	Siempre cero, no se modifica	1
r1	\$at	Assembler Temporary – Reservado para el ensamblador	
r2 - r3	\$v0 y \$v1	Valores de retorno de la subrutina llamada	1
r4 - r7	\$a0 - \$a3	Argumentos (parámetros) pasados a la subrutina llamada	1
r8 - r15	\$t0 - \$t7	Registros Temporales de propósito general. Cuando se retorna de	1
r24 - r25	\$t8 - \$t9	una subrutina, no se garantiza la permanencia del valor anterior.	
r16 - r23	\$s0 - \$s7	Registros Salvados. Si una subrutina los utiliza debe salvar los valores y reponerlos antes de retornar	1, 2
r26 - r27	\$k0 - \$k1	Para uso del Kernel del sistema operativo	
r28	\$gp	Global Pointer: Puntero a zona de memoria estática del programa	2
r29	\$sp	Stack Pointer: Puntero al tope de la pila	1, 2
r30	\$fp	Frame Pointer: Puntero al marco de pila en una subrutina actual	2
r31	\$ra	Return Address : Dirección de retorno de subrutina actual	1, 2

1 – registros que usamos en los ejercicios de la práctica.

2 – registros que deben salvarse antes de ser utilizados en una subrutina

# Subrutinas – Convención, Ejemplo

```
.data  
    result: .word 0  
  
.text  
    daddi r4, r0, 10  
    daddi r5, r0, 20  
    jal sumar  
    sd r2, result(r0)  
    halt
```

```
sumar: dadd r2, r4, r5  
        jr r31
```

Convención  
programa  
equivalente

```
.data  
    result: .word 0  
  
.text  
    daddi $a0, $zero, 10  
    daddi $a1, $zero, 20  
    jal sumar  
    sd $v0, result($zero)  
    halt
```

```
sumar: dadd $v0, $a0, $a1  
        jr $ra
```

¿Cuál es mas legible?

# Subrutinas - Convención, Registros

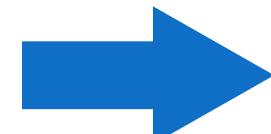
## Preservación de registros:

- Una subrutina debe guardar los valores de los registros “garantizados” antes de modificarlos: **\$ra, \$s0..\$s7, \$fp, \$sp, \$gp**
- Si todas las subrutinas siguen la convención garantizamos la preservación de los valores luego de la invocación a otras subrutinas, evitando los efectos laterales
- Para garantizar la preservación de los valores de los registros es necesario un mecanismo adicional que permita salvar los valores. Necesitamos una pila

# Convención - Pila

- No hay instrucciones específicas para manipular y mantener la pila
- Todas las subrutinas usarán, por convención, el registro \$sp (stack pointer - r29) como puntero de la pila
- Por convención, se inicializa \$sp con un valor alto de la memoria de programa y decrementa al apilar (0x400 por defecto en WimMips)
- Se definen las operaciones de “PUSH” y “POP” para apilar y desapilar registros en memoria de datos:

**push \$t1**

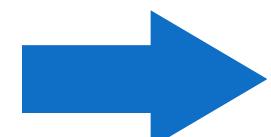


**daddi \$sp, \$sp, -8**

**sd \$t1, 0(\$sp)**

**Notar cero  
en vez de  
etiqueta**

**pop \$t1**



**ld \$t1, 0(\$sp)**

**daddi \$sp, \$sp, 8**

# Convención - Pila

**“Apilado” consecutivo de varios (3) registros:**

**; Método 1**

```
daddi $sp, $sp, -8  
sd    $s1, 0($sp)  
daddi $sp, $sp, -8  
sd    $s2, 0($sp)  
daddi $sp, $sp, -8  
sd    $s3, 0($sp)
```

**; Método 2**

```
daddi $sp, $sp, -24 ; 8*3  
sd    $s3, 0($sp)  
sd    $s2, 8($sp)  
sd    $s1, 16($sp)
```

**; Método 3, acceder directamente en cualquier orden**

```
daddi $sp, $sp, -24 ; 8*3  
sd    $s1, 0($sp)  
sd    $s2, 8($sp)  
sd    $s3, 16($sp)
```

**Los mismos métodos pueden usarse para restaurar los registros**

# Convención - Subrutina

Así, toda subrutina se dividirá en **tres partes**: **prólogo**, **cuerpo** y **epílogo**

**subrut:** **daddi \$sp, \$sp, -tamaño\_frame** ; Reserva espacio en la pila  
**sd \$ra, 0(\$sp)** ; Guarda la dirección de retorno

**prólogo** **sd \$s0, 8(\$sp)** ; Guarda el registro \$s0  
**sd \$s1, 16(\$sp)** ; Guarda el registro \$s1

:

**cuerpo** ... instrucciones ... ; Cuerpo de la subrutina

:

:

**epílogo** **ld \$ra, 0(\$sp)** ; Recupera la dirección de retorno

**ld \$s0, 8(\$sp)** ; Recupera el registro \$s0

**ld \$s1, 16(\$sp)** ; Recupera el registro \$s1

:

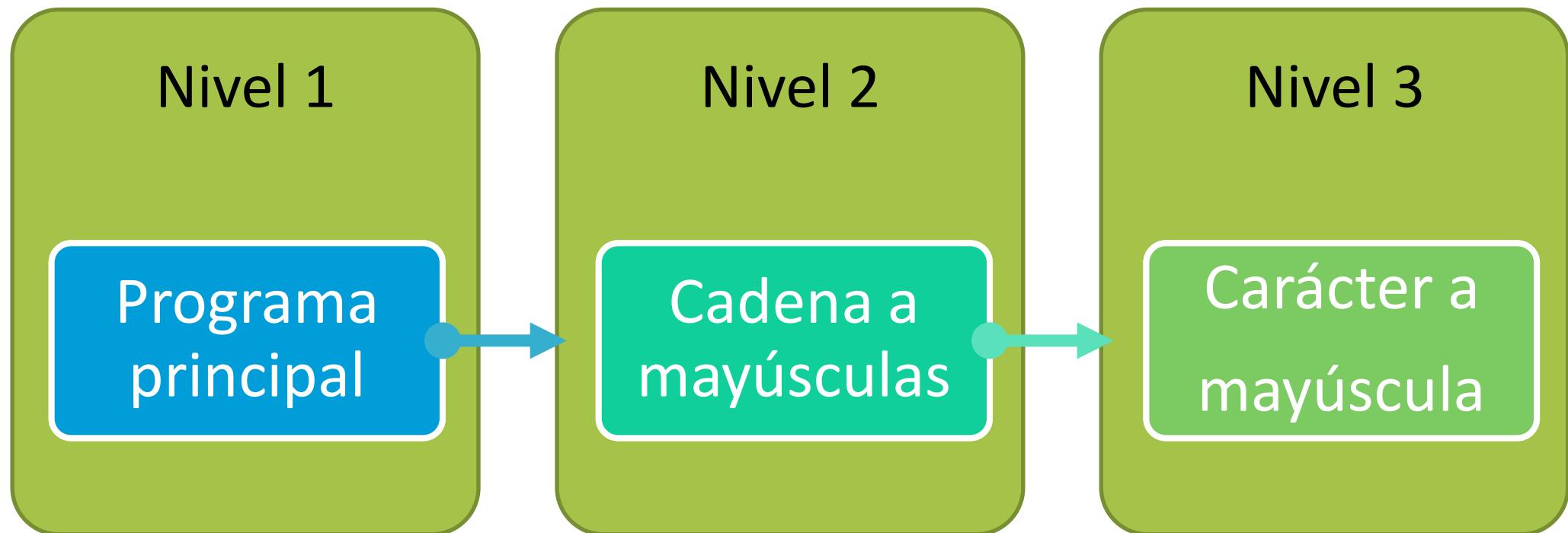
:

**daddi \$sp, \$sp, tamaño\_frame** ; Restaura el tope de la pila

**jr \$ra** ; Retorna al punto de llamada

# Ejemplo de Subrutina

Construir un programa que convierta una cadena de caracteres a mayúsculas. Utilizar una subrutina para convertir la cadena que a su vez utilice otra subrutina que convierta un carácter a mayúsculas.



# Ejemplo de Subrutina - Upcase

; Pasar un carácter a mayúscula.

; Parámetros:

; - \$a0 -> carácter

; - \$v0 -> carácter en mayúscula

; No se utiliza la pila porque no se usan registros que deban ser salvados

; Nota: 0x61 a 0x7A → "a" a "z"            0x41 a 0x5A → "A" a "Z"

;        0x61 → 01100001<sub>2</sub> → "a"        0x41 → 01000001<sub>2</sub> → "A"

**upcase:**    DADD \$v0, \$a0, \$zero

                SLTI \$t0, \$v0, 0x61

                BNEZ \$t0, salir

                SLTI \$t0, \$v0, 0x7B

                BEQZ \$t0, salir

                DADDI \$v0, \$v0, -0x20

salir:        JR \$ra

; copia carácter en registro de salida

; compara con 'a' minúscula

; no es un carácter en minúscula

; compara con el car sig a 'z' minúscula (z=7AH)

; no es un carácter en minúscula

; pasa a minúscula (pone a '0' el 6to bit o bit5)

; retorna al programa principal

# Ejemplo de Subrutina - UpcaseStr

; Parámetros: - \$a0 -> inicio de cadena (puntero o referencia al primer carácter)

; Se utiliza la pila para guardar:

; - \$ra -> porque se invoca a otra subrutina

; - \$s0 -> para guardar la dirección de inicio de la cadena y recorrerla

upcaseStr:	<b>DADDI</b> \$sp, \$sp, -16 ; Reserva lugar en la pila -> 2 x 8 <b>SD</b> \$ra, 0(\$sp) ; Salva porque llama a subrutina <b>SD</b> \$s0, 8(\$sp) ; Salva para preservar puntero a cadena cuando invoque upcase
upcaseStrLOOP:	<b>DADD</b> \$s0, \$a0, \$zero ; copia la dirección de la cadena <b>LBU</b> \$a0, 0(\$s0) ; recupera el car actual como argumento para upcase <b>BEQ</b> \$a0, \$zero, <i>upcaseStrFIN</i> ; Si es el fin de la cadena, termina <b>JAL</b> <i>upcase</i> <b>SB</b> \$v0, 0(\$s0) ; guarda el caracter procesado en la cadena <b>DADDI</b> \$s0, \$s0, 1 ; avanza al siguiente caracter <b>J</b> <i>upcaseStrLOOP</i>
<i>upcaseStrFIN:</i>	<b>LD</b> \$ra, 0(\$sp) ; restaura registros a valores iniciales <b>LD</b> \$s0, 8(\$sp) <b>DADDI</b> \$sp, \$sp, 16 <b>JR</b> \$ra

# Ejemplo de Subrutina - UpcaseStr

El siguiente ejemplo sencillo convierte una cadena de caracteres a caracteres en mayúsculas. Programa principal:

```
.data
    cadena: .asciiz "Caza"
.text
    ; La pila comienza en el tope de la memoria de datos 0x400 es la dirección
    ; más alta configurada en el WinMIPS
    DADDI $sp, $0, 0x400
    ; pasa referencia a cadena como primer argumento para upcaseStr
    DADDI $a0, $0, cadena
    JAL upcaseStr
...
    HALT
```

# Arquitectura de computadoras

*Genaro Camele*

# RISC

*Reduced instruction set computing*

# RISC

*Definición*

Hasta ahora vimos un procesador **CISC** (SX88) donde teníamos muchas instrucciones complejas

Ahora veremos un procesador **RISC** con un conjunto reducido de instrucciones llamado **MIPS**

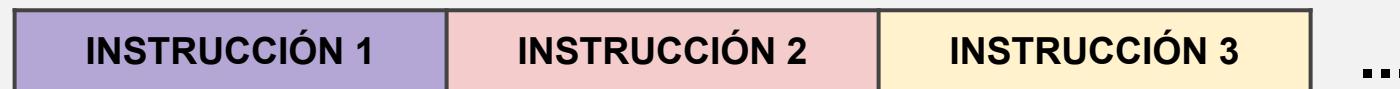
---

- El hardware se organiza de una manera llamada **Segmentación de cause o Pipeline**: realiza más de una operación al mismo tiempo!
- Las instrucciones se organizan en **fases** de manera que esto sea posible
- Explota el **parallelismo** entre las fases de distintas instrucciones

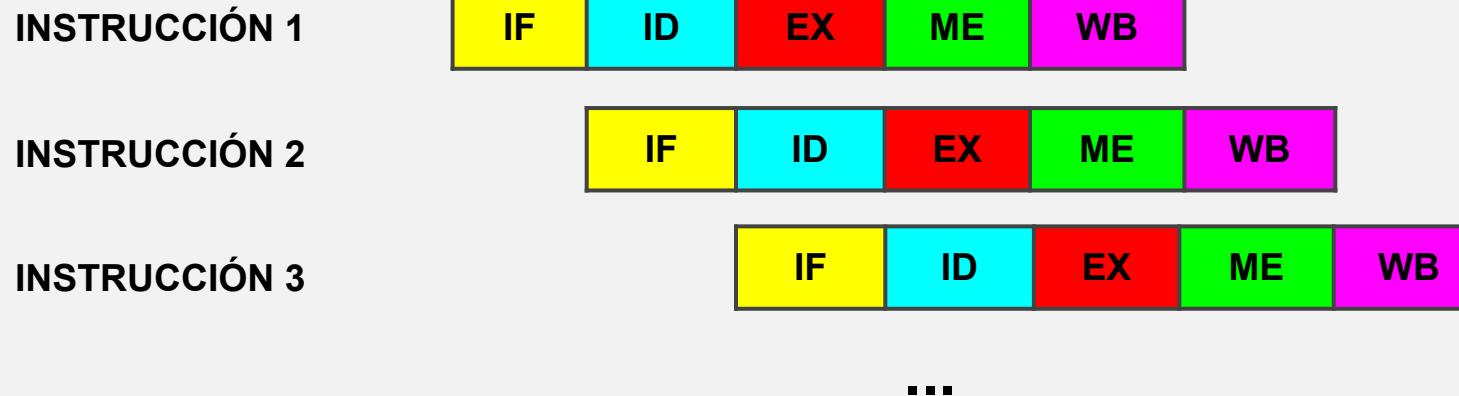
# RISC

*Definición*

## Ejecución secuencial



## Ejecución segmentada



Las instrucciones duran lo mismo. Pero podemos ejecutarlas paralelamente

# RISC

*Fases de una instrucción*

- Las instrucciones se organizan en **fases** de manera que esto sea posible



- Se accede a memoria por la instrucción
- Se incrementa el PC (antes conocido como el IP)

*Instruction Fetch*



- Se decodifica la instrucción
- Se accede al banco de registros por los operandos. Se atasca si no están disponibles
- Se calcula el valor del operando inmediato

*Instruction Decoding*

- Si es un salto, se calcula el destino y si se toma o no (requiere acceder al banco de reg.)



- Si es una instrucción de cómputo, se ejecuta en la ALU
- Si es un acceso a memoria, se calcula la dirección efectiva
- Si es un salto, se realiza (se modifica el registro PC)

*Execution*

# RISC

*Fases de una instrucción*

- Las instrucciones se organizan en **fases** de manera que esto sea posible



ME

- Si es un acceso a memoria, se lee/escribe el dato

*Memory access*

WB

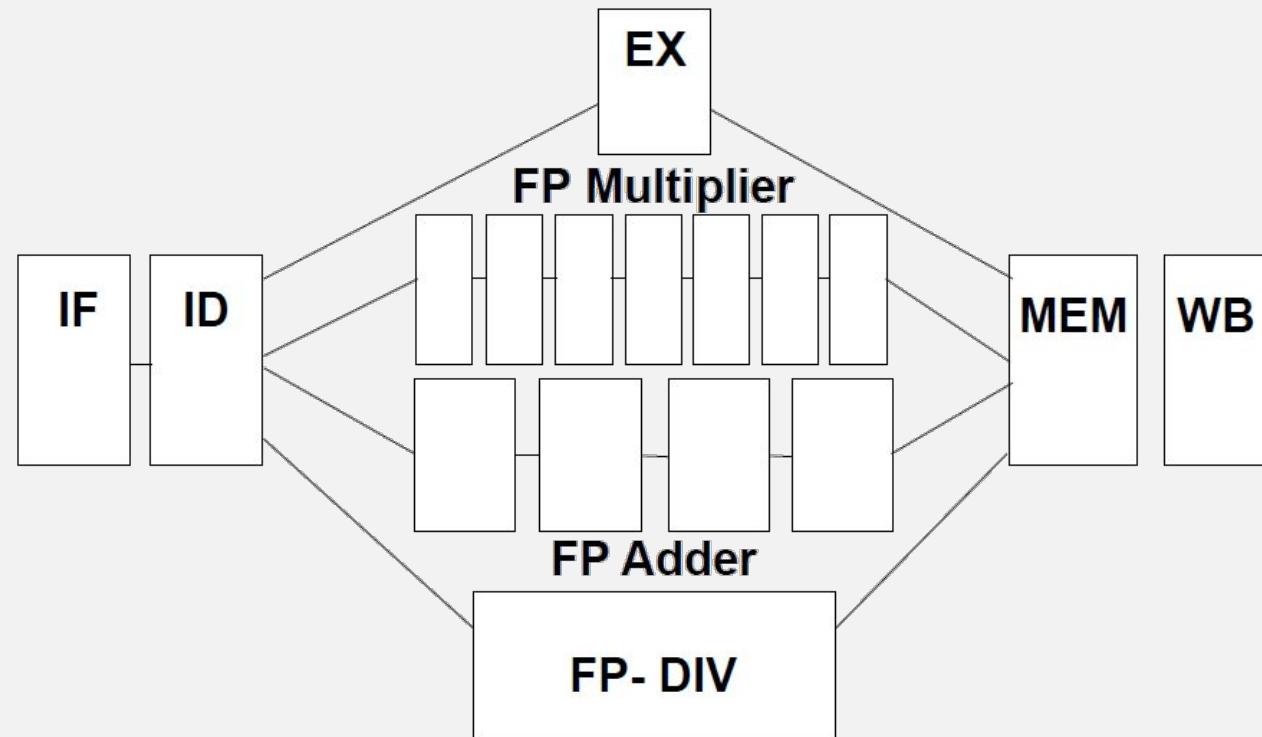
- Se almacena el resultado (**si lo hay**) en los registros

*Write Back*

# RISC

*Fases de una instrucción*

- No todas las etapas tardan lo mismo
- Por ende, no todas pueden manejarse en paralelo



# Atascos

*Tipos*

Se pueden dar situaciones que impiden a la siguiente instrucción que se ejecute en el ciclo que le corresponde

## DEPENDENCIA DE DATOS

- **RAW:** Read After Write
- **WAR:** Write After Read
- **WAW:** Write After Write

## ESTRUCTURALES

Provocados por conflicto de recursos

## DEPENDENCIA DE CONTROL

Provocados al esperar la decisión de otra instrucción anterior (por ej.: si se realiza o no un salto)

Queremos evitarlos a toda costa

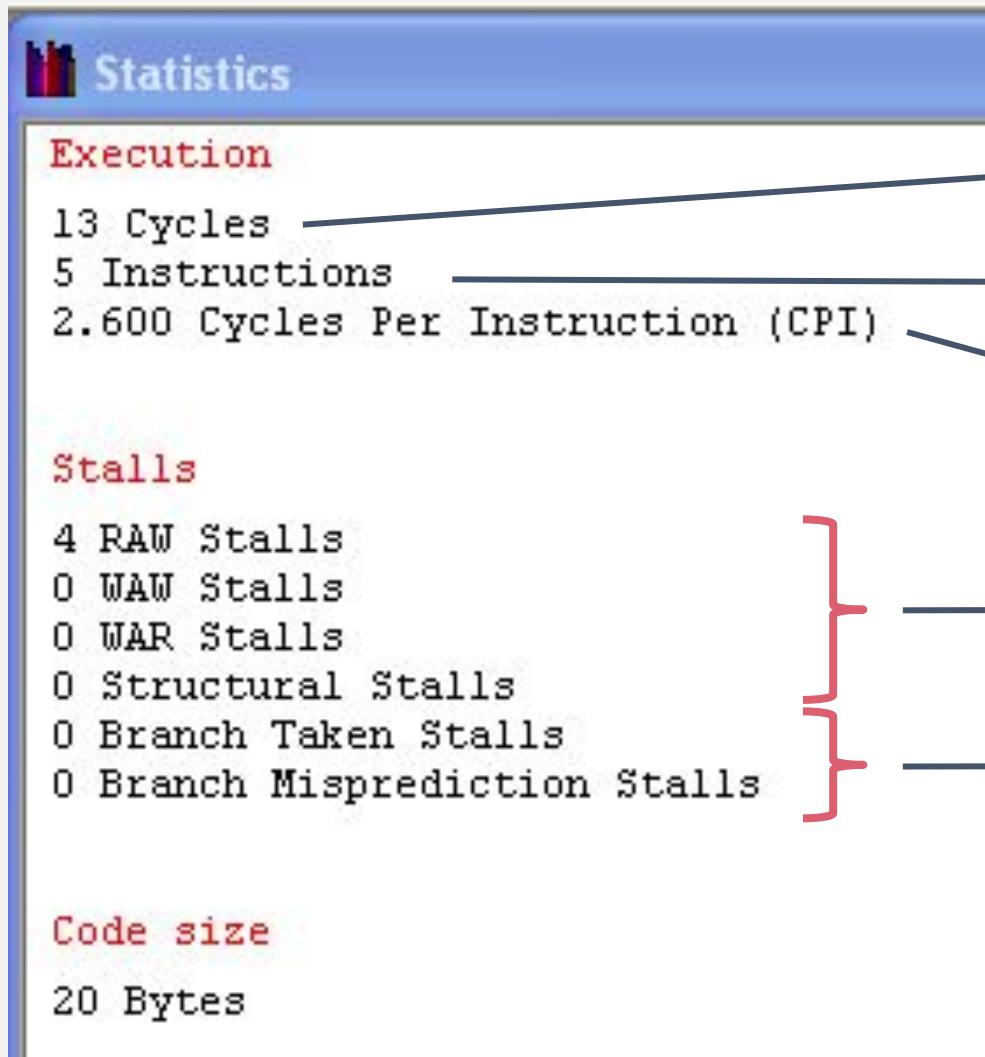
# MIPS

Simulador



# MIPS

*Simulador*



Cantidad de ciclos de ejecución

Cantidad de instrucciones

Ciclos por instrucción -> Ciclos / Instrucciones

Atascos

Penalización por saltos

# MIPS

*Registros y variables*

Tenemos 32 registros de uso general (de 64 bits): **R0 ... R31**

**RO SIEMPRE vale 0**

Tenemos 32 registros de punto flotante (de 64 bits): **F0 ... F31**

---

También podemos definir **variables**!

```
.data  
A: .byte 1  
B: .word16 2  
C: .word32 3  
D: .word 2882  
TABLA: .byte "Hola"
```

<b>.byte</b>	<i>8 bits</i>
<b>.word16</b>	<i>16 bits</i>
<b>.word32</b>	<i>32 bits</i>
<b>.word</b>	<i>64 bits</i>

# MIPS

## *Instrucciones*

Veamos algunas instrucciones

- Sumar dos registros: *DADD R1, R2, R3*  $R1 = R2 + R3$
- Sumar un registro con un valor inmediato: *DADDI R1, R2, 5*  $R1 = R2 + 5$
- Hacer un XOR con un valor inmediato: *XORI R6, R6, 0xFD*  $R6 = R6 \text{ XOR } 253$
- Tenemos la resta! *DSUB R1, R2, R3*  $R1 = R2 - R3$
- Tenemos multiplicación y división!
  - *DMUL R1, R2, R3*  $R1 = R2 * R3$
  - *DDIV R1, R2, R3*  $R1 = R2 / R3$

Para el resto! Ver el set de  
instrucciones!

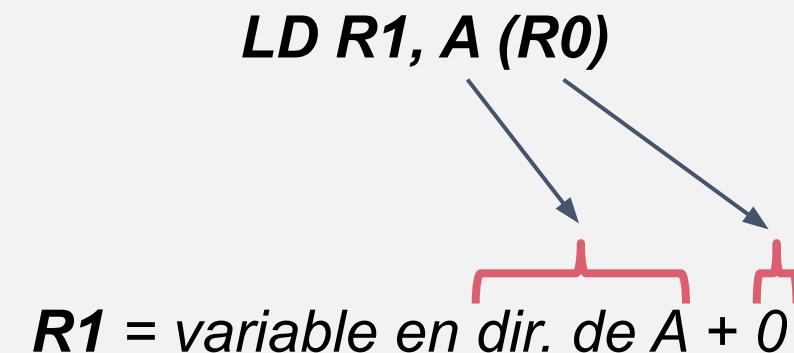
# MIPS

## Instrucciones

Veamos cómo nos manejamos con variables

- Para empezar, **todas** las operaciones aritmético-lógicas **deben** hacerse con **registros**
- Si quiero usar variables, debo cargarlas antes en un registro
- Las variables se definen en un bloque `.data` y arrancan en la dirección 0
- Para tomar una variable se usa **LD <DESTINO>, <VARIABLE> (DESPLAZAMIENTO)**

```
.data  
A: .word 5  
B: .word 8  
  
.code  
LD R1, A (R0)
```



# MIPS

## Instrucciones

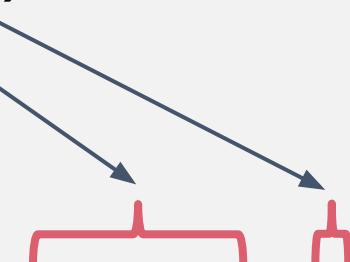
Veamos cómo nos manejamos con variables

- Para guardar un valor en memoria se utiliza el mismo mecanismo de desplazamiento
- La sintaxis es **SD <REGISTRO ORIGEN>, <VARIABLE> (DESPLAZAMIENTO)**

```
.data  
A: .word 0  
  
.code  
DADDI R1, R0, 5  
SD R1, A (R0)
```

**SD R1, A (R0)**

*A = valor de R1 en variable en dir. de A + 0*



# MIPS

*Ejemplo de multiplicación*

Escribir un algoritmo que multiplique dos variables y guarde el resultado en la variable *RES*

**.data**

NUM1: .word 5

NUM2: .word 6

RES: .word 0

**.code**

LD R1, NUM1 (R0)

LD R2, NUM2 (R0)

DMUL R3, R1, R2

SD R3, RES (R0)

HALT

Ponerlo en un archivo .s y  
cargarlo en WinMIPS

# Atascos

*Dependencia de datos*

# Atascos

*RAW*

*RAW* significa “Read After Write”

Se produce cuando una instrucción necesita leer un dato que todavía no está disponible

**.data**

NUM1: .word 15

```
DADDI R1, R0, 8  
DADDI R2, R1, 10  
LD R7, NUM1 (R0)  
HALT
```



**.code**

; Inicializamos un registro y le sumamos 10

DADDI R1, R0, 8

DADDI R2, R1, 10

; Después hacemos otra cosa

LD R7, NUM1 (R0)

HALT

Se atasca el pipeline y  
se alarga la etapa ID!

# Atascos

*Soluciones al RAW*

Podemos solucionarlo de dos formas:

## Por software

DADDI R1, R0, 8

DADDI R2, R1, 10

LD R7, NUM1 (R0)

**vs**

DADDI R1, R0, 8

**NOP**

DADDI R2, R1, 10

LD R7, NUM1 (R0)

**Con NOPs**

**vs**

DADDI R1, R0, 8

LD R7, NUM1 (R0)

DADDI R2, R1, 10

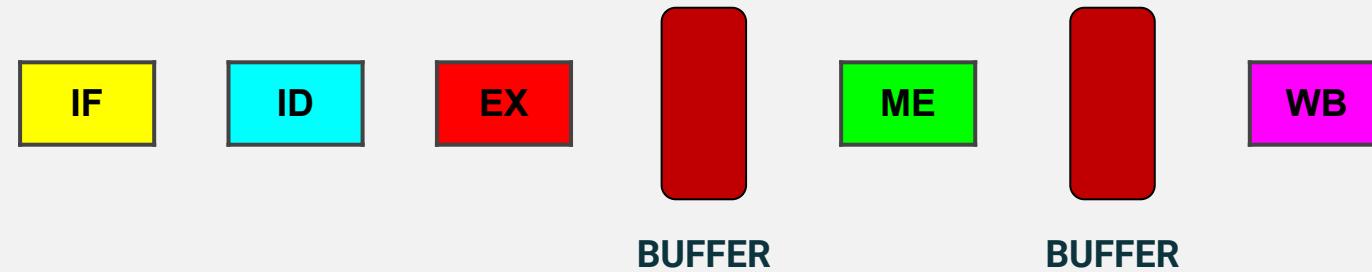
**Ordenando sentencias**

# Atascos

*Soluciones al RAW*

## Por hardware

Si ya tenemos los valores necesarios, podemos “adelantarlos”



En estos buffers se almacenan los valores para que los puedan usar en las próximas instrucciones

De esta manera no hace falta esperar a las etapas **MEM** y **WB** para usar los valores!

Este adelantamiento de operando lo llamamos **Forwarding**

Esta técnica, además, permite postergar la “necesidad” de los operandos

# Atascos

*Forwarding*

## Sin forwarding

*LD R5, NUM (R0)*



*DADD R2, R5, R4*



## Con forwarding

*LD R5, NUM (R0)*



*DADD R2, R5, R4*



**.code**

...

*LD R5, NUM (R0)*

*DADD R2, R5, R4*

...

# Atascos

*Dependencia de control*

# Atascos

*Dependencia de control*

Tenemos dos tipos de saltos

**Incondicionales**: salta siempre

**Condicionales**: salta dependiendo de que se cumpla una condición

```
; Condicional  
LOOP: DADDI R2, R2, -1  
DADDI R1, R1, R1  
BNEZ R2, LOOP  
DADDI R7,R0,R1
```

```
; Incondicional  
LOOP: ; Hace algo...  
      ; Hace otra cosa...  
J LOOP
```

```
; Incondicional con registro  
DADD R1, R2, R3  
JR R1
```

# Atascos

*Branch Taken Stall*

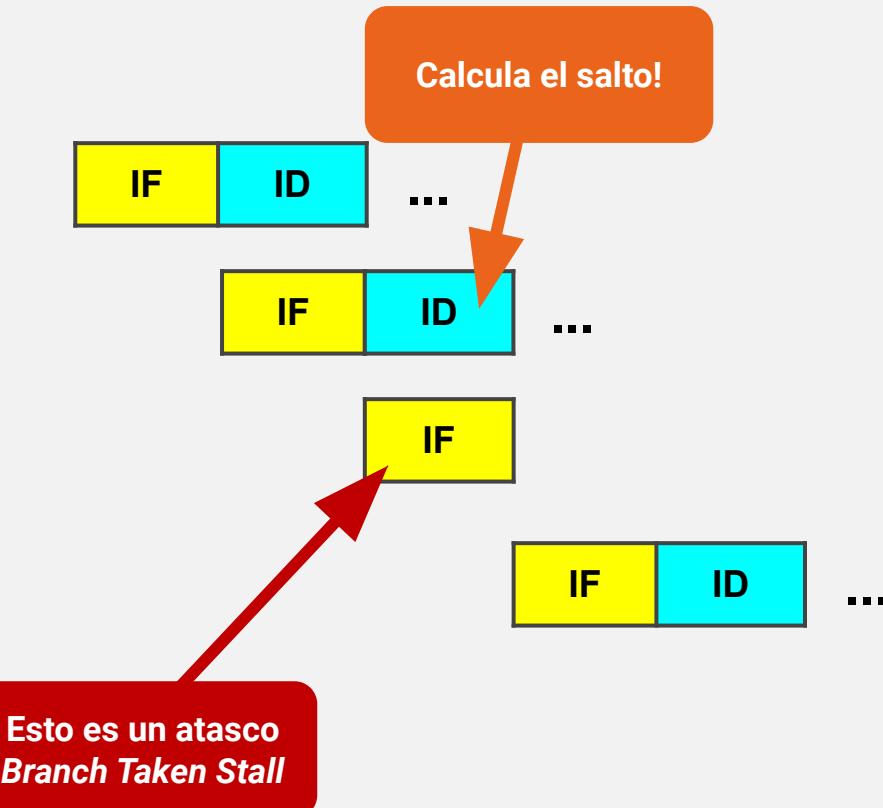
; Supongamos R2 = 5

LOOP: DADDI R2, R2, -1

**BNEZ** R2, LOOP ; R2 = 4

DADDI R7,R0,R1

LOOP: DADDI R2, R2, -1



; Condicional  
LOOP: DADDI R2, R2, -1  
**BNEZ** R2, LOOP  
DADDI R7,R0,R1

# MIPS

*Ejemplo de saltos*

Calcular  $2^B$  en MIPS

**.data**

B: .word 5

**.code**

DADDI R1, R0, 1

LD R2, B (R0)

LOOP: DSLL R1, R1, 1 ; Desplazo a la izquierda

DADDI R2, R2, -1 ; Cant. de desplazamientos que faltan

BNEZ R2, LOOP ; Si no es 0 salto a LOOP

HALT

Analizar en WinMIPS

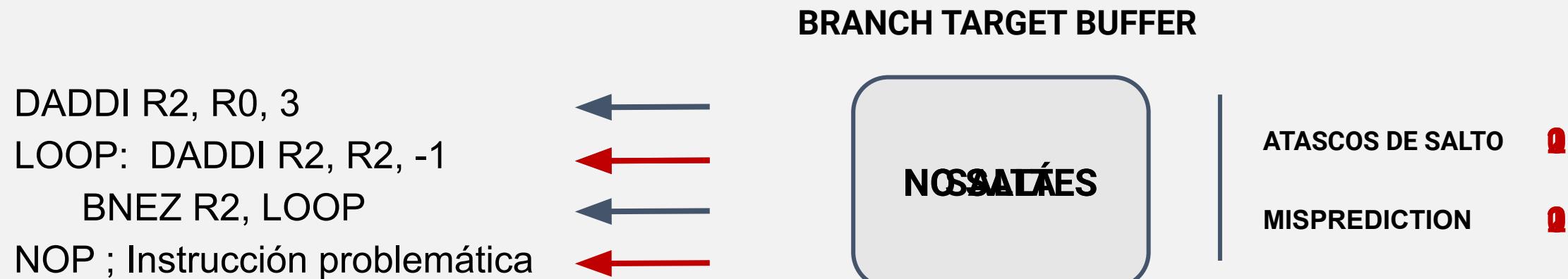
# Atascos de control

*Branch Target Buffer (BTB)*

Al igual que con los atascos de dependencia de datos, tenemos diferentes técnicas para evitar los atascos por saltos

La primera que vamos a ver es la denominada **Branch Target Buffer** que consiste en tener un flag que indica que si se debe saltar incondicionalmente o no dependiendo de qué hizo antes (es decir, *predice*)

Cada vez que ese flag/buffer se actualiza cuenta como un **atasco de salto!** Cada vez que le erramos a la predicción cuenta como atasco!



# Atascos de control

*Branch Target Buffer (BTB)*

¿Cuándo conviene activar el *BTB*?

Analicemos los siguientes algoritmos (en pseudocódigo)

```
For i := 1 to 100000000 do begin  
    ; Hace algo  
end;
```

```
For i := 1 to 100000000 do begin  
    if i es par then  
        ; Hace una cosa  
    else  
        ; Hace otra cosa  
    end;
```

La hostia! Vamos a tener dos atascos al principio y dos al final

Vamos a tener dos atascos en cada iteración (porque vamos a fallar la predicción)

# Atascos de control

*Delay Slot*

La otra técnica se denomina **Delay Slot**

Simplemente consiste en ejecutar **siempre** la siguiente instrucción a un salto

DADDI R2, R0, 3

LOOP: DADDI R2, R2, -1

BNEZ R2, LOOP

HALT

Se terminaría el programa  
ya que se ejecuta el HALT en  
la primera iteración!

- Buscamos ubicar instrucciones que **no dependen del salto**
- De esta manera aprovechamos una instrucción y no tenemos atascos de salto!
- Cómo último recurso usar sentencias NOP

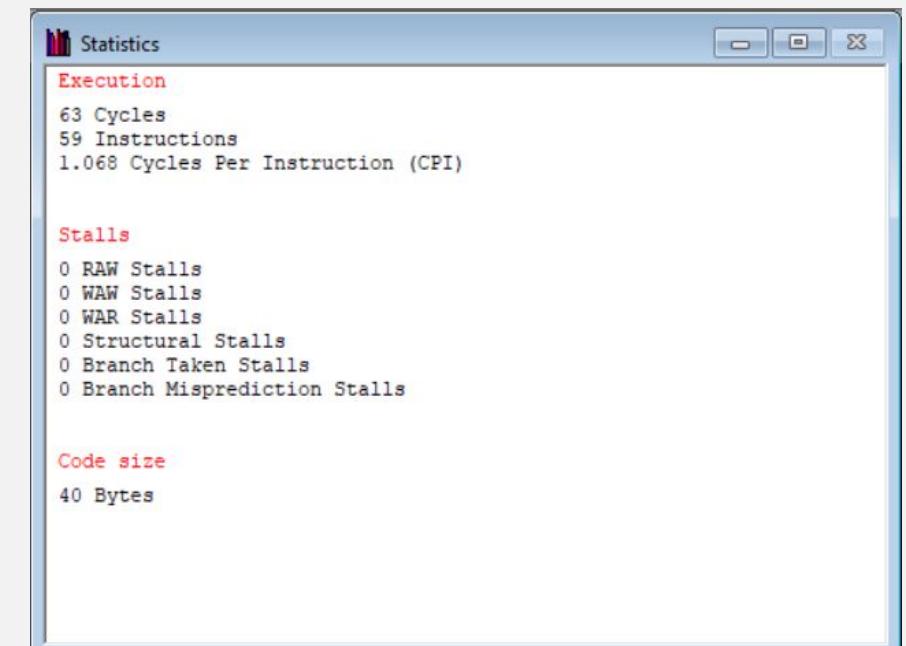
# Atascos de control

## *Delay Slot*

Analizar el siguiente código con el **Delay Slot** deshabilitado/habilitado

```
.data  
cant: .word 8  
datos: .word 1, 2, 3, 4, 5, 6, 7, 8  
res: .word 0  
  
.code  
DADD R1, R0, R0 ; Inicializa R1 = 0  
LD R2, cant (R0) ; R2 = cant  
LOOP: LD R3, datos (R1) ; R3 = elemento de datos en la posición R1  
        DADDI R2, R2, -1 ; Resta 1 a la cantidad de elementos a procesar  
        DSLL R3, R3, 1 ; Multiplica por dos el elemento actual  
        SD R3, res (R1) ; Almacena el resultado en la tabla de resultados  
        DADDI R1, R1, 8 ; Avanza a la siguiente posición  
        BNEZ R2, LOOP ; Si quedan elementos sigo iterando  
  
NOP  
HALT
```

¿Para qué?



Con Forwarding activado

# Atascos de control

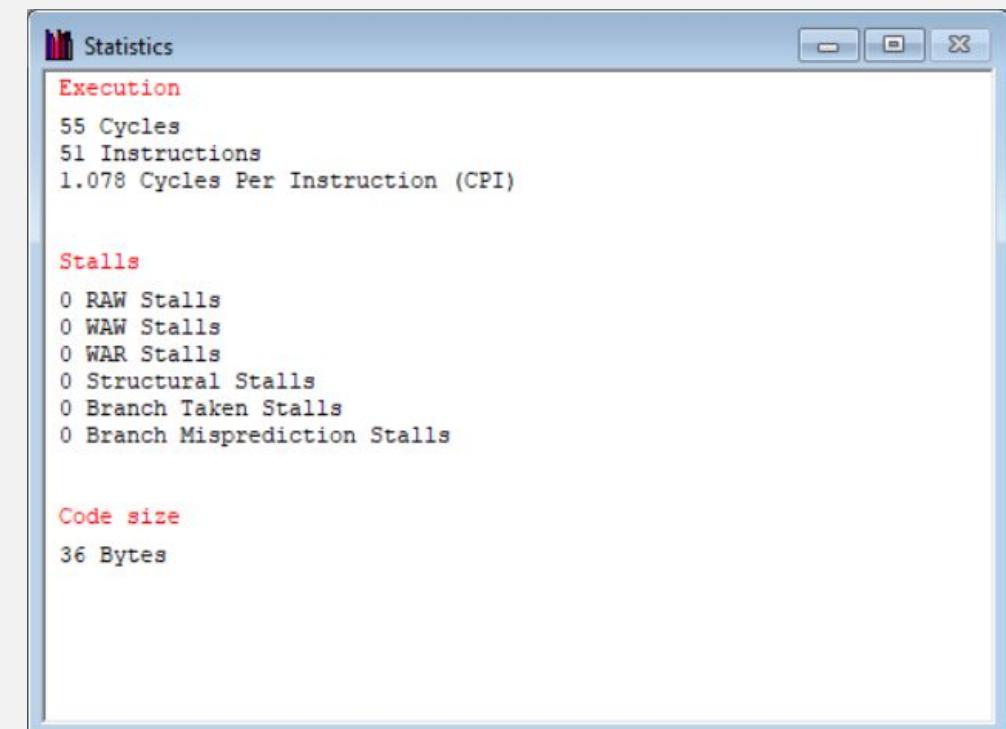
*Delay Slot*

Podemos usar la técnica del reordenamiento de sentencias para dejar algo útil!

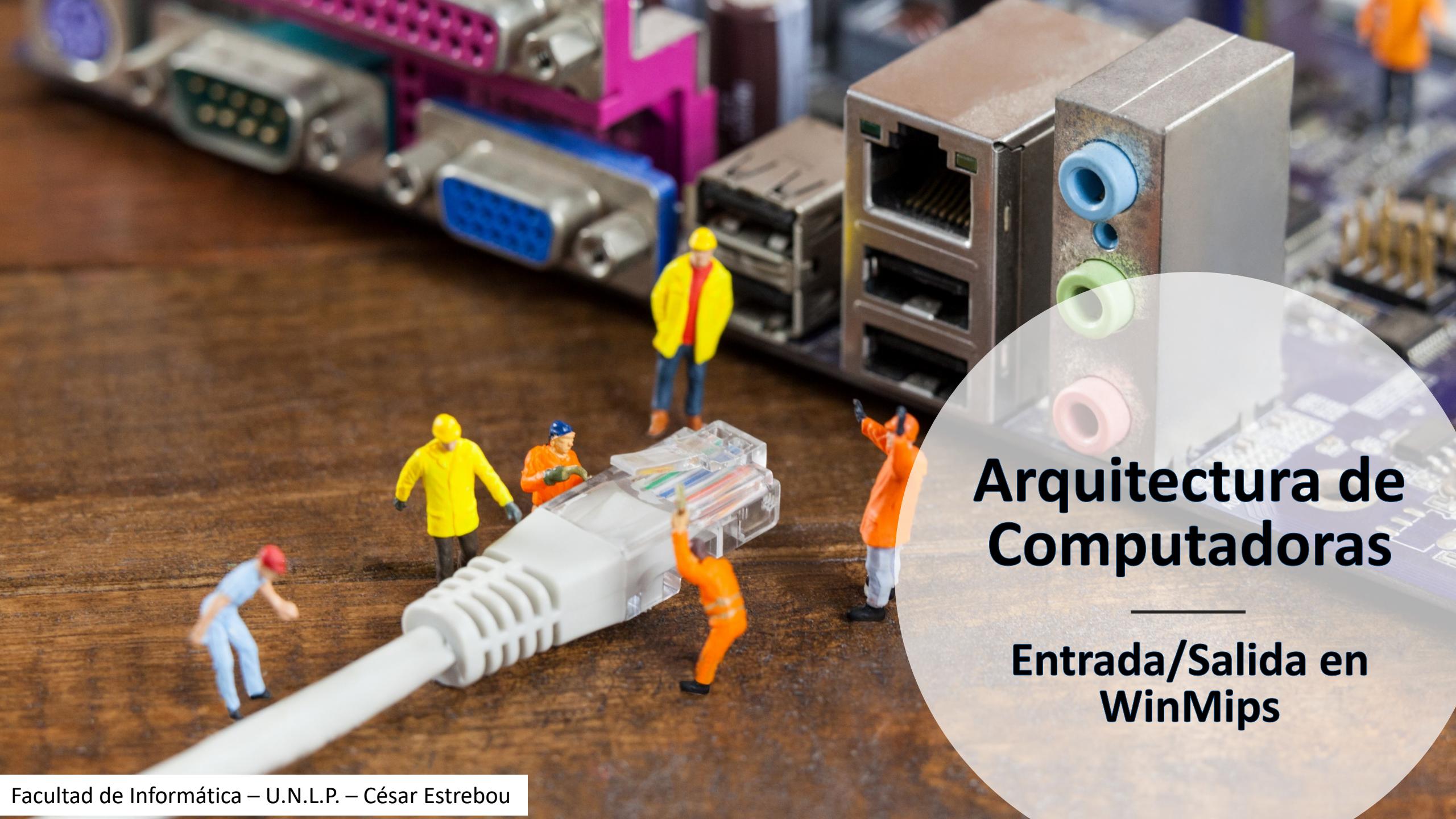
Recordar que esta sentencia no debe que afectar el salto!

```
.data
cant: .word 8
datos: .word 1, 2, 3, 4, 5, 6, 7, 8
res: .word 0

.code
DADD R1, R0, R0 ; Inicializa R1 = 0
LD R2, cant (R0) ; R2 = cant
LOOP: LD R3, datos (R1) ; R3 = elemento de datos en la posición R1
        DADDI R2, R2, -1 ; Resta 1 a la cantidad de elementos a procesar
        DSLL R3, R3, 1 ; Multiplica por dos el elemento actual
        SD R3, res (R1) ; Almacena el resultado en la tabla de resultados
        BNEZ R2, LOOP ; Si quedan elementos sigo iterando
DADDI R1, R1, 8 ; Avanza a la siguiente posición
HALT
```



Con Forwarding activado



# Arquitectura de Computadoras

---

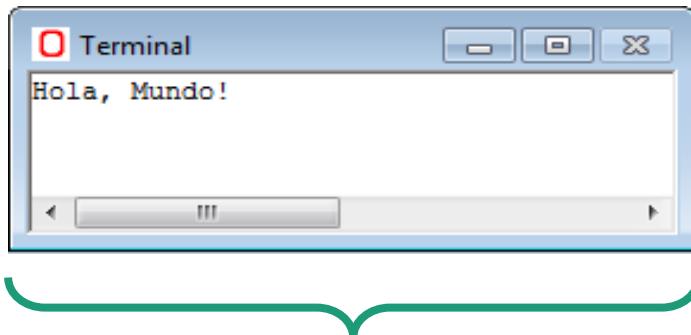
## Entrada/Salida en WinMips



# Entrada/Salida en WinMips

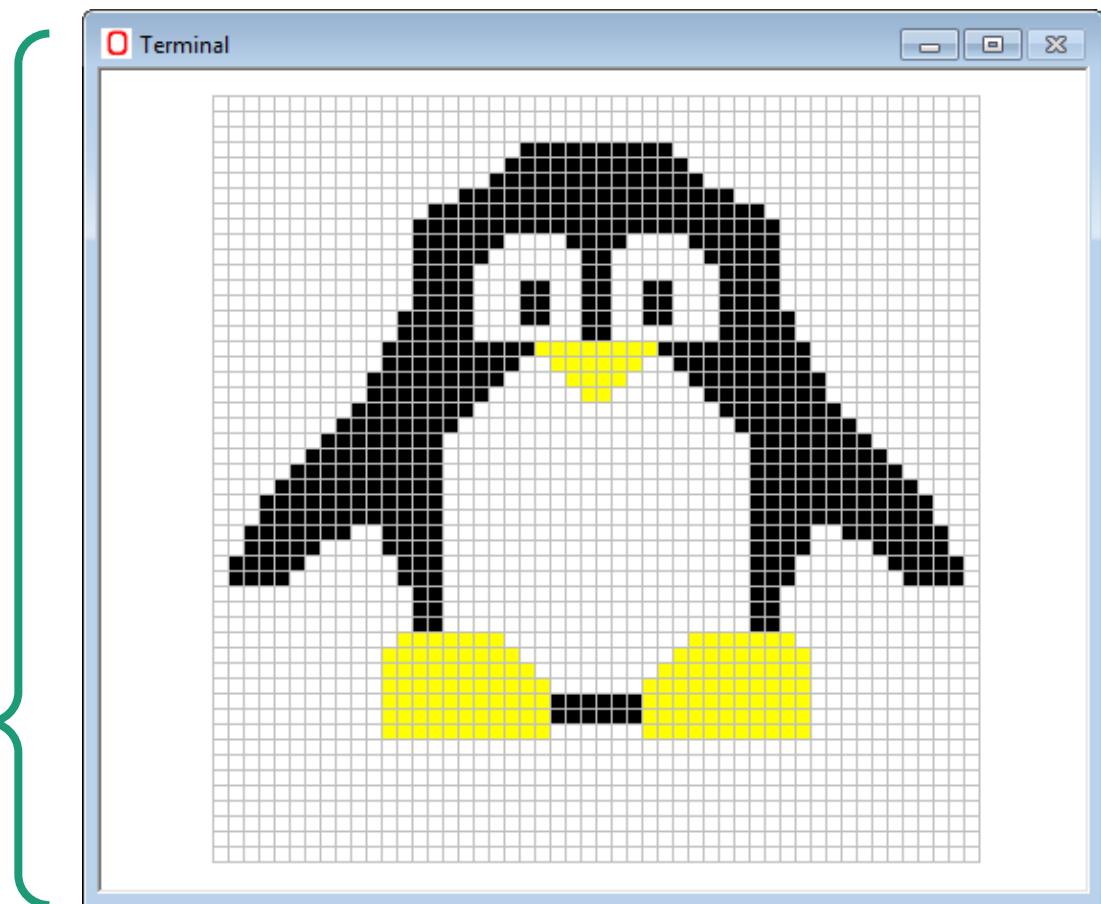
# Entrada / Salida en Winmips

Winmips posee una terminal que permite hacer algunas operaciones de **entrada / salida** por **teclado / pantalla**



Entrada / Salida en modo texto:  
leer y escribir números y  
cadenas

Salida en modo gráfico: pintar  
pixeles en una pantalla de 50 x 50



# Entrada / Salida en Winmips

En el MSX88 tenemos espacios de memoria y entrada/salida bien diferenciados y se acceden con instrucciones diferentes:

- MOV → Memoria
- IN/OUT → Entrada/Salida

En el Winmips accedemos al espacio de entrada/salida a través del espacio de memoria, por lo que usamos la misma instrucción en ambos casos:

- LD/SD → Memoria
- LD/SD → Entrada/Salida

Estas operaciones de Entrada/Salida son similares a las interrupciones por software del MSX88

Cada operación esta identificada con un número, igual que en el MSX88

# Entrada / Salida en Winmips

- Existen dos direcciones de memoria en el winmips destinadas para entrada/salida.
- Al leer/escribir estas direcciones NO se lee ni escribe la memoria sino, que se lee/escribe el espacio de entrada/salida.
- La interacción con estos puertos nos permite realizar operaciones similares a las que hacíamos con las interrupciones por software del MSX88.

# Puertos de Entrada / Salida en Winmips

## Dirección CONTROL

- Ubicado en la dirección 0x10000 (65536)
- Ocupa 8 bytes (64 bits)
- Se utiliza para almacenar un valor asociado a una operación:
  - Leer o escribir números enteros, flotantes y cadenas de texto
  - Escribir un pixel en pantalla gráfica
  - Borrar pantalla de texto o gráfica
- Solo tiene sentido escribirlo

## Dirección DATA

- Ubicado en la dirección 0x10008 (65544)
- Ocupa 8 bytes (64 bits)
- Se utiliza para leer/escribir un valor (parámetro de entrada/salida) que depende de lo que se escribe en el puerto de control

# Operaciones de Entrada

Una operación típica de entrada (teclado):

- Produce una entrada desde el teclado (leer un número entero o flotante o un carácter)
- Como no requiere parámetros (al menos para las operaciones que actualmente implementa Winmips) no se escribe DATA, sólo se lee luego de escribir CONTROL
- La operación de entrada se “ejecuta” cuando se escribe el valor de operación en el puerto de CONTROL Y Bloquea la ejecución hasta que finalice la operación de entrada
- Deja el valor de entrada en el registro DATA

# Operaciones de Entrada

- Leer un número entero o flotante desde el teclado:
  - Se escribe un 8 en CONTROL
  - Luego de ingresado el número desde la consola, este valor queda disponible en DATA
- Leer una tecla (código ascii) sin eco desde el teclado :
  - Se escribe un 9 en CONTROL
  - Luego de ingresado el carácter desde la consola, este carácter queda disponible en DATA

# Operaciones de Salida

Una operación típica de salida (pantalla):

- Produce algún tipo de salida o efecto en pantalla de texto o pantalla gráfica
- Puede requerir un parámetro. Cuando éste se necesita se escribe en DATA (siempre antes que el control)
- Se “ejecuta” cuando se escribe el valor de operación en el puerto de CONTROL. Inmediatamente aparece el resultado en pantalla

# Operaciones de Salida

- Escribir un número entero sin signo:
  - Se escribe el número natural en DATA
  - Se escribe un 1 en CONTROL y en la consola se muestra el número
- Escribir un número entero con signo:
  - Se escribe el número entero en DATA
  - Se escribe un 2 en CONTROL y en la consola se muestra el número
- Escribir un número flotante:
  - Se escribe el número flotante en DATA
  - Se escribe un 3 en CONTROL y en la consola se muestra el número

# Operaciones de Salida

- Escribir una cadena de texto (que termina en cero):
  - Se escribe la dirección de comienzo de la cadena en DATA
  - Se escribe un 4 en CONTROL y en la consola se muestra la cadena de texto
- Limpiar pantalla alfanumérica:
  - No requiere DATA
  - Se escribe un 6 en CONTROL y la consola se muestra limpia en modo texto
- Limpiar pantalla gráfica:
  - No requiere DATA
  - Se escribe un 7 en CONTROL y la consola se muestra limpia en modo gráfico

# Operaciones de Salida

## Escribir un pixel en pantalla gráfica:

- Primero se escribe en DATA la información del pixel:
  - DATA      ← color del pixel (4 bytes) en RGB (cantidad de rojo, verde, azul y un cuarto valor ignorado)
  - DATA+4    ← coordenada Y del punto (1 byte)
  - DATA+5    ← coordenada X del punto (1 byte)
- Luego se escribe un 5 en CONTROL
- Se dibuja un pixel de color RGB en la coordenada (X,Y) indicada (DATA+5, DATA+4)
- La información del pixel se puede escribir parcialmente. Debe estar completa al momento de escribir CONTROL

# Ejercicio 1

```
.data
texto:      .asciiz  "Hola Mundo !!" El mensaje a mostrar
CONTROL:    .word32  0x10000  Direcciones de mapeo, observar que CONTROL es
DATA:       .word32  0x10008  una variable con la dirección y no el puerto de control
```

```
.text
1 lwu $s0, DATA($zero) ; $s0= dirección de DATA
2 addi $t0, $zero, texto ; $t0= dirección de comienzo de texto
3 sd $t0, 0($s0) ; DATA recibe el puntero de mensaje
4 lwu $s1, CONTROL($zero) ; $s1= dirección de CONTROL
5 addi $t0, $zero, 6 ; $t0= 6, función 6 = limpiar pantalla
6 sd $t0, 0($s1) ; CONTROL recibe 6 y limpia pantalla terminal
7 addi $t0, $zero, 4 ; $t0= 4, función 4 = escribir cadena
8 sd $t0, 0($s1) ; CONTROL recibe 4 y produce salida del mensaje
9 halt
```

*Nota: las instrucciones con valores inmediatos permiten 16bits, no es posible utilizar valores mayores a 65535, por eso CONTROL y DATA se cargan desde memoria*

# Ejercicio 6

```

.data
coorX: .byte 24 ; coordenada X de un punto
coorY: .byte 24 ; coordenada Y de un punto
color: .byte 255, 0, 255, 0 ; R,G,B,? → color magenta
CONTROL: .word32 0x10000
DATA: .word32 0x10008

.text
1 lwu $s6, CONTROL($zero) ; $s6= dirección de CONTROL
2 lwu $s7, DATA($zero) ; $s7= dirección de DATA
3 daddi $t0, $zero, 7 ; $t0= función 7, limpiar pantalla gráfica
4 sd $t0, 0($s6) ; Escribe registro de control, limpia pantalla
5 lbu $s0, coorX($zero) ; $s0= coordenada X
6 sb $s0, 5($s7) ; Byte 5 DATA = coordenada X
7 lbu $s1, coorY($zero) ; $s1= coordenada Y
8 sb $s1, 4($s7) ; Byte 4 de DATA = coordenada Y
9 lwu $s2, color($zero) ; $s2= color para pixel (4 bytes)
10 sw $s2, 0($s7) ; Byte 0 a 3 = color para pixel
11 daddi $t0, $zero, 5 ; $t0 = función 5, dibujar pixel
12 sd $t0, 0($s6) ; CONTROL recibe 5 y dibuje pixel
13 halt

```

DATA 

255	0	255	?	24	24	?	?
R	G	B		Y	X		

# Arquitectura de computadoras

*Genaro Camele*

# Punto Flotante

# Punto Flotante

*En MIPS*

MIPS utiliza IEEE 754 para números en punto flotante

Contamos con 32 registros: desde F0 (siempre vale 0) hasta F31

Tiene un único tipo de dato que es el *.double*

Algunos ejemplos de instrucciones son:

- Carga - *L.D F1, NUM1 (R0)*
- Suma - *ADD.D F1, F2, F3*
- Resta - *SUB.D F1, F2, F3*
- Multiplicación - *MUL.D F1, F2, F3*
- División - *DIV.D F1, F2, F3*
- Almacenamiento - *S.D F1, RES (R0)*

$$\text{RES} = (\text{A} + \text{B}) * \text{C}$$

## ***.data***

*A: .double 5.5*

*B: .double 8.6*

*C: .double 10.0*

*RES: .double 0.0*

## ***.code***

*L.D F1, A (R0)*

*L.D F2, B (R0)*

*L.D F3, C (R0)*

*ADD.D F1, F1, F2*

*MUL.D F1, F1, F3*

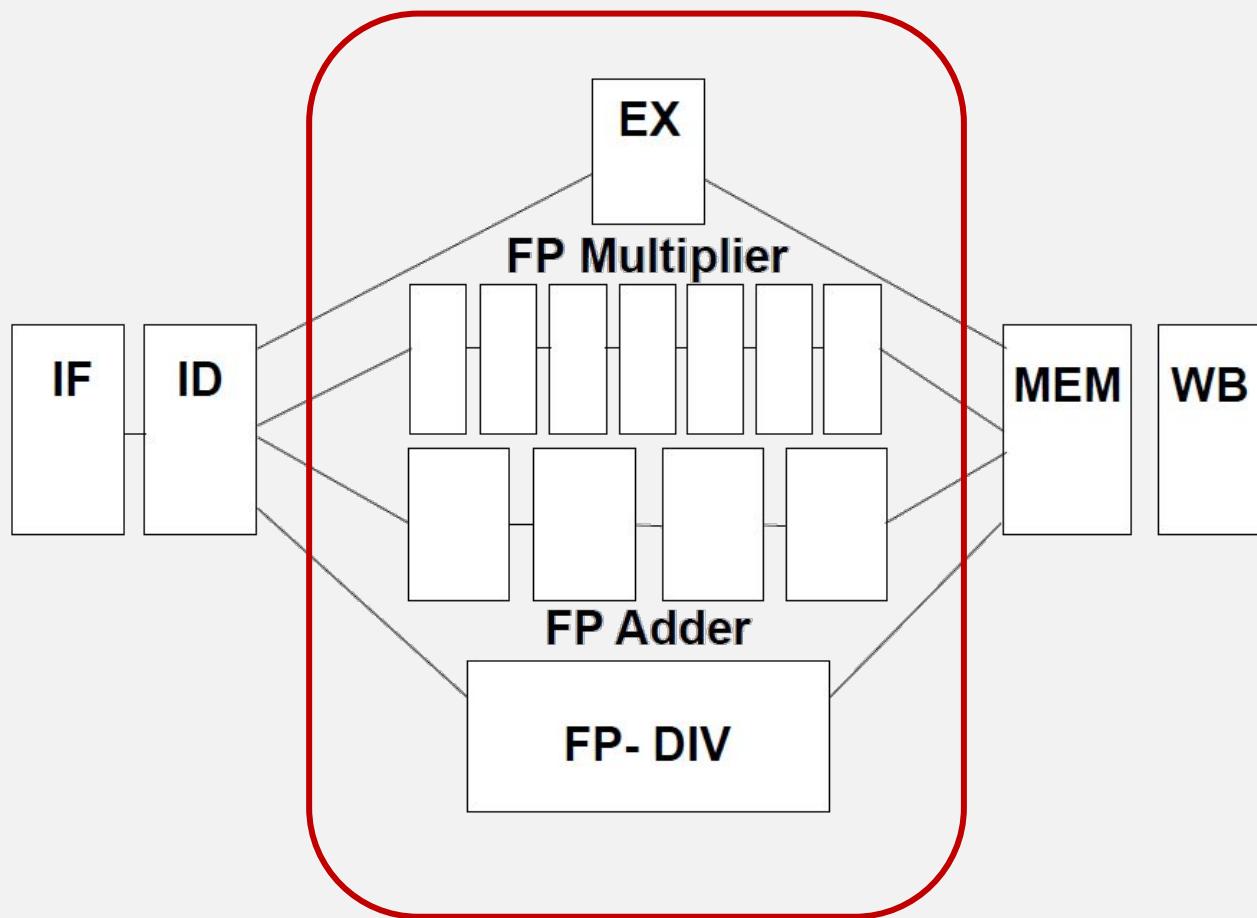
*S.D F1, RES (R0)*

*HALT*

# Punto Flotante

*Instrucciones pesadas*

Recordar que: **no todas las etapas tardan lo mismo**



- Generales = 1 ciclo
- Multiplicar en Pto. F. = 7 ciclos
- Sumar en Pto. F. = 4 ciclos
- Dividir en Pto. F. = 24 ciclos

**Esto nos va a generar nuevos problemas**

# Punto Flotante

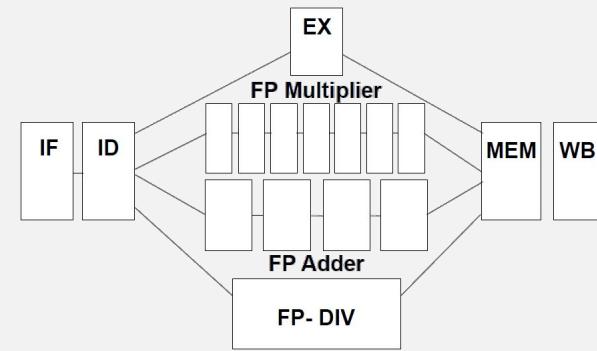
*Instrucciones pesadas*

Esta arquitectura nos permite tener múltiples instrucciones en la etapa *EX*

Podemos ejecutar múltiples instrucciones en menos tiempo!

No todo es color de rosas. Introduce los siguientes atascos:

- Dependencia Estructural
- Dependencia de datos WAR
- Dependencia de datos WAW



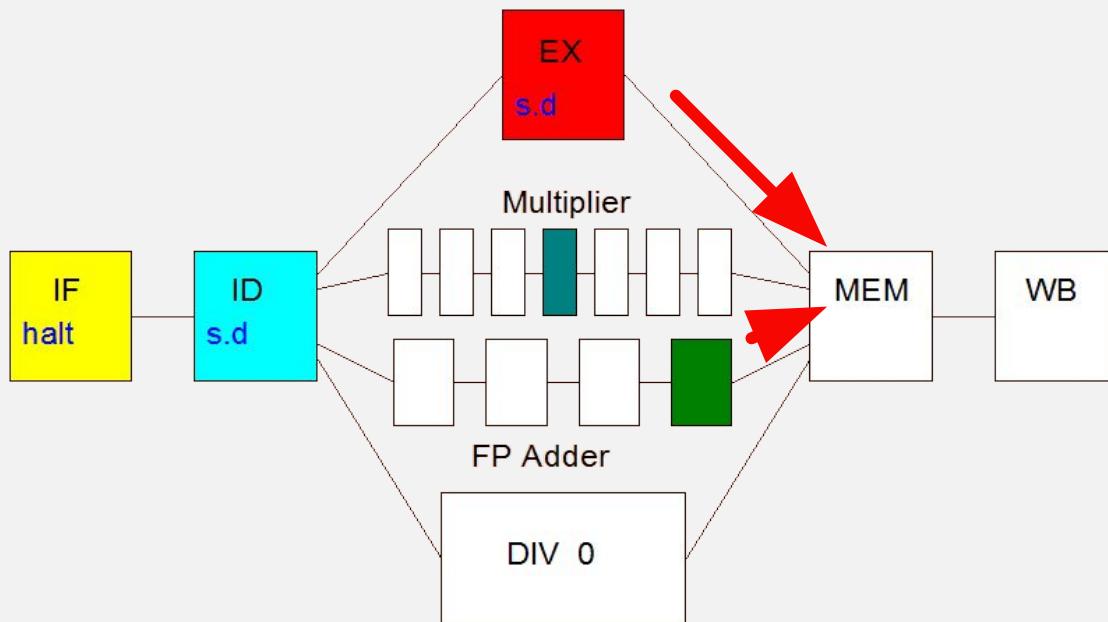
Atascos  
Str., WAR y WAW

# Atascos estructurales

*Definición*

Los atascos estructurales son provocados por conflictos por los recursos

En el MIPS sucede cuando dos instrucciones intentan acceder a la etapa **MEM** simultáneamente



Dos instrucciones listas para pasar a la etapa de memoria. Se produce un atasco estructural y solo pasa una de ellas

Tiene prioridad la primera instrucción que entró en el cauce

# Atascos WAR y WAW

*Definición*

Se producen cuando

- Hay **dependencia de datos** entre dos instrucciones (igual que *RAW*)
- Una instrucción puede sobrepasar a una instrucción anterior, queriendo escribir un registro pendiente de lectura (WAR) o escritura (WAW)
- El simulador produce atascos cuando detecta una situación potencial (**puede que realmente no suceda**) de dependencia WAR o WAW.

# Atascos WAR y WAW

*Ejemplo*

Analizar en el simulador el siguiente algoritmo

```
.data
n1: .double    9.13
n2: .double    6.58
res1: .double   0.0
res2: .double   0.0
```

```
.code
L.D F1, n1 (R0) ; F1 = n1
L.D F2, n2 (R0) ; F2 = n2
ADD.D F3, F2, F1 ; F3 = F2 + F1
MUL.D F4, F2, F1 ; F4 = F2 * F1
S.D F3, res1 (R0) ; Guarda la suma en res1
S.D F4, res2 (R0) ; Guarda la multiplicacion en res2
HALT
```

Analizar en el simulador (con Forwarding activado) los atascos que genera

# Atascos WAR y WAW

Ejemplo

No tiene disponible F2

ADD.D F3,F2,F1 y S.D F3, res1(R0) están listas para pasar a la etapa MEM.  
S.D F3, res1(R0) debe esperar que ADD.D F3,F2,F1 pase a la siguiente

Cycles

Statistics

Execution  
16 Cycles  
7 Instructions  
2.286 Cycles Per Instruction (CPI)

Stalls

4 RAW Stalls  
0 WAW Stalls  
0 WAR Stalls  
2 Structural Stalls  
0 Branch Taken Stalls  
0 Branch Misprediction Stalls

Code size  
28 Bytes

Code

```
0000 d4010000 L.D F1, n1(R0)
0004 d4020008 L.D F2, n2(R0)
0008 462110c0 ADD.D F3, F2, F1
000c 46211102 MUL.D F4, F2, F1
0010 f4030010 S.D F3, res1(R0)
0014 f4040018 S.D F4, res2(R0)
0018 04000000 HALT
001c 00000000
```

# Atascos WAR y WAW

*Ejemplo 2*

Agregamos la instrucción **MUL.D F2, F2, F1**

```
.data
n1:    .double    9.13
n2:    .double    6.58
res1:   .double    0.0
res2:   .double    0.0
```

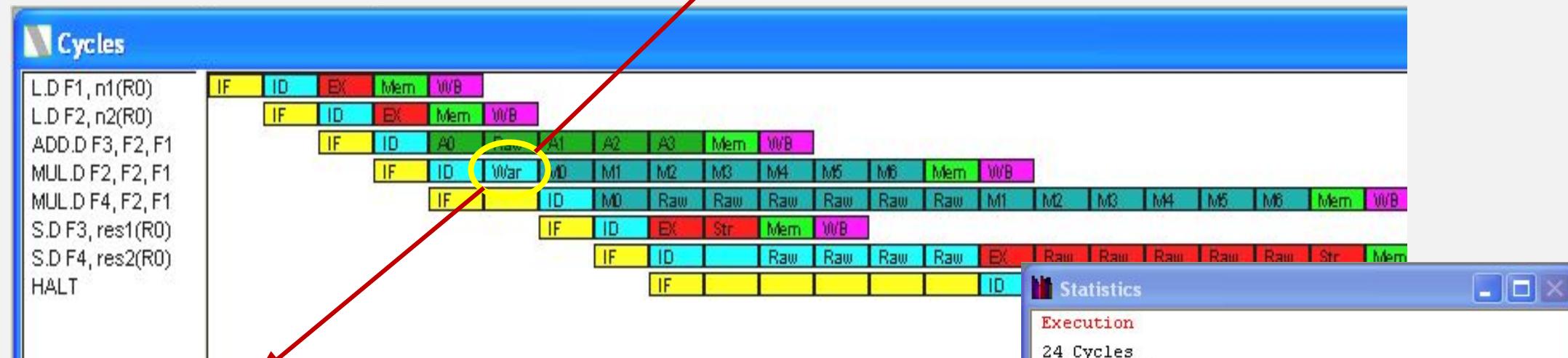
```
.code
L.D F1, n1 (R0) ; F1 = n1
L.D F2, n2 (R0) ; F2 = n2
ADD.D F3, F2, F1 ; F3 = F2 + F1
MUL.D F2, F2, F1 ; F2 = F2 * F1
MUL.D F4, F2, F1 ; F4 = F2 * F1
S.D F3, res1 (R0) ; Guarda la suma en res1
S.D F4, res2 (R0) ; Guarda la multiplicacion en res2
HALT
```

Anализар en el simulador (con  
Forwarding activado) los  
atascos que genera

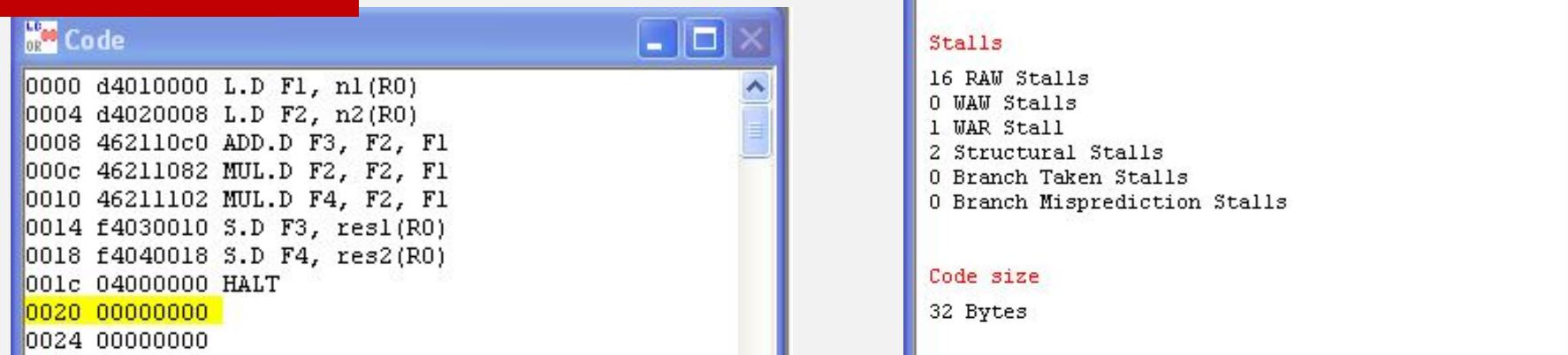
# Atascos WAR y WAW

Ejemplo 2

MUL.D quiere modificar F2, pero esta está siendo leída por ADD.D



Se puede solucionar con un NOP en el medio



# Subrutinas

# Subrutinas

Al igual que en MSX88, en MIPS también podemos definir **subrutinas**

## .data

NUM1: .word 5

NUM2: .word 8

RES: .word 0

## .code

LD R1, NUM1 (R0)

LD R2, NUM2 (R0)

JAL **SUMAR**; Llama a la subrutina

SD R3, RES (R0)

HALT

**SUMAR**: DADD R3, R1, R2

**JR R31**; Retorna al punto donde fue llamado

## Cosas importantes

- No hay manejo implícito de la pila!
- La dirección de retorno **siempre** estará en **R31**
  - Esto sí se hace implícitamente en el *JAL*
- Los registros se pueden **sobrescribir**, incluído **R31**
  - Vamos a tener que salvarlos en la pila

# Convenciones

Ante la cantidad de registros y consideraciones que debemos tener en cuenta, se establecieron **convenciones**:

AHORA	DESCRIPCIÓN	ANTES
\$zero	Siempre tiene el valor 0 y no se puede cambiar	(r0)
\$ra	<i>Return Address</i> – Dir. de retorno de subrutina. Debe ser salvado	(r31)
\$v0-\$v1	Valores de retorno de la subrutina llamada	(r2-r3)
\$a0-\$a3	Argumentos pasados a la subrutina llamada	(r4-r7)
\$t0-\$t9	Registros temporarios	(r8-r15 y r24-r25)
\$s0-\$s7	Registros que deben ser salvados	(r16-r23)
\$sp	<i>Stack Pointer</i> – Puntero al top	<ul style="list-style-type: none"><li>● En <b>verde</b> aquellos que, en caso de usarse, <b>deben ser salvados</b></li><li>● En <b>azul</b> aquellos que podemos sobrescribir sin ningún problema</li></ul>
\$fp	<i>Frame Pointer</i> – Puntero de pi	
\$at	<i>Assembler Temporary</i> – Reser	
\$k0-\$k1	Para uso del kernel del sistem	
\$gp	<i>Global Pointer</i> – Puntero a zon	

- En **verde** aquellos que, en caso de usarse, **deben ser salvados**
- En **azul** aquellos que podemos sobrescribir sin ningún problema

# Convenciones

Veamos el ejemplo de recién, pero ahora con convenciones

```
.data  
NUM1: .word 5  
NUM2: .word 8  
RES: .word 0
```

```
.code  
LD R1, NUM1 (R0)  
LD R2, NUM2 (R0)  
JAL SUMAR; Llama a la subrutina  
SD R3, RES (R0)  
HALT
```

**SUMAR**: DADD R3, R1, R2

**JR R31**; Retorna al punto donde fue llamado



```
.data  
NUM1: .word 5  
NUM2: .word 8  
RES: .word 0
```

```
.code  
LD $a0, NUM1 (R0)  
LD $a1, NUM2 (R0)  
JAL SUMAR; Llama a la subrutina  
SD $v0, RES (R0)  
HALT
```

**SUMAR**: DADD \$v0, \$a0, \$a1

**JR \$ra**; Retorna al punto donde fue llamado

# Convenciones

*Preservación de los registros*

Las subrutinas deben garantizar el guardado de los registros que correspondan

De esta manera, una subrutina podrá llamar a otra sabiendo que esta no modificará el valor de estos registros

Para poder mantener esa garantía, es necesario guardar los registros en la **pila**. Pero MIPS **no tiene pila!**

Pero existe un registro que por **convención** todas las subrutinas usarán como puntero al tope de la pila!

## Usemos el registro \$sp

PUSH \$t1



**daddi** \$sp, \$sp, -8 ; “Subo” una celda de memoria  
**sd** \$t1, 0 (\$sp) ; Almaceno el dato

POP \$t1



**ld** \$t1, 0 (\$sp) ; Extraigo el dato  
**daddi** \$sp, \$sp, 8 ; “Bajo” una celda de memoria

# Subrutinas

Veamos un ejemplo de una **subrutina** que pasa todo un string a mayúsculas

```
.data  
cadena: .asciiz "Caza"  
  
.code  
; La pila comienza en el tope de la memoria de datos  
DADDI $sp, $0, 0x400 ; bus 10 bits  $\square 2^{10} = 1024 = 0x400$   
  
; Guarda como primer argumento para upcaseStr  
; la dirección de cadena  
DADDI $a0, $0, cadena  
JAL upcaseStr  
HALT
```

Recordar que nosotros debemos inicializar la pila

**upcaseStr:** DADDI \$sp, \$sp, -16 ; Reserva lugar en pila -> 2 x 8  
SD \$ra, 0 (\$sp) ; Guarda en pila \$ra  
SD \$s0, 8 (\$sp) ; Guarda en pila \$s0  
DADD \$s0, \$a0, \$zero ; Copia la dirección de inicio de la cadena  
**LOOP:** LBU \$a0, 0 (\$s0) ; Toma car. actual  
BEQ \$a0, \$zero, FIN ; Si es el fin, termina  
JAL upcase  
SB \$v0, 0 (\$s0) ; Guarda el carácter procesado en la cadena  
DADDI \$s0, \$s0, 1 ; Avanza al siguiente carácter  
J LOOP  
; Recupera los datos salvados en la pila  
**FIN:** LD \$ra, 0 (\$sp)  
LD \$s0, 8 (\$sp)  
DADDI \$sp, \$sp, 16  
JR \$ra

# Subrutinas

**upcase** solo se encarga de pasar un único carácter a mayúsculas

; Pasa un carácter a mayúscula.

; Parámetros: \$a0 -> carácter

; Retorna \$v0 -> carácter en mayúscula

; No se utiliza la pila porque no se usan registros que deban ser salvados

**upcase:** DADD \$v0, \$a0, \$zero

SLTI \$t0, \$v0, 0x61 ; Compara con 'a' minúscula

BNEZ \$t0, salir ; No es un carácter en minúscula

SLTI \$t0, \$v0, 0x7B ; Compara con el carácter sig a 'z' minúscula (z=7AH)

BEQZ \$t0, salir ; No es un carácter en minúscula

DADDI \$v0, \$v0, -0x20 ; Pasa a minúscula

**salir:** JR \$ra ; Retorna

# Arquitectura de computadoras

*Genaro Camele*

# Entrada/Salida

*Comencemos con salida*

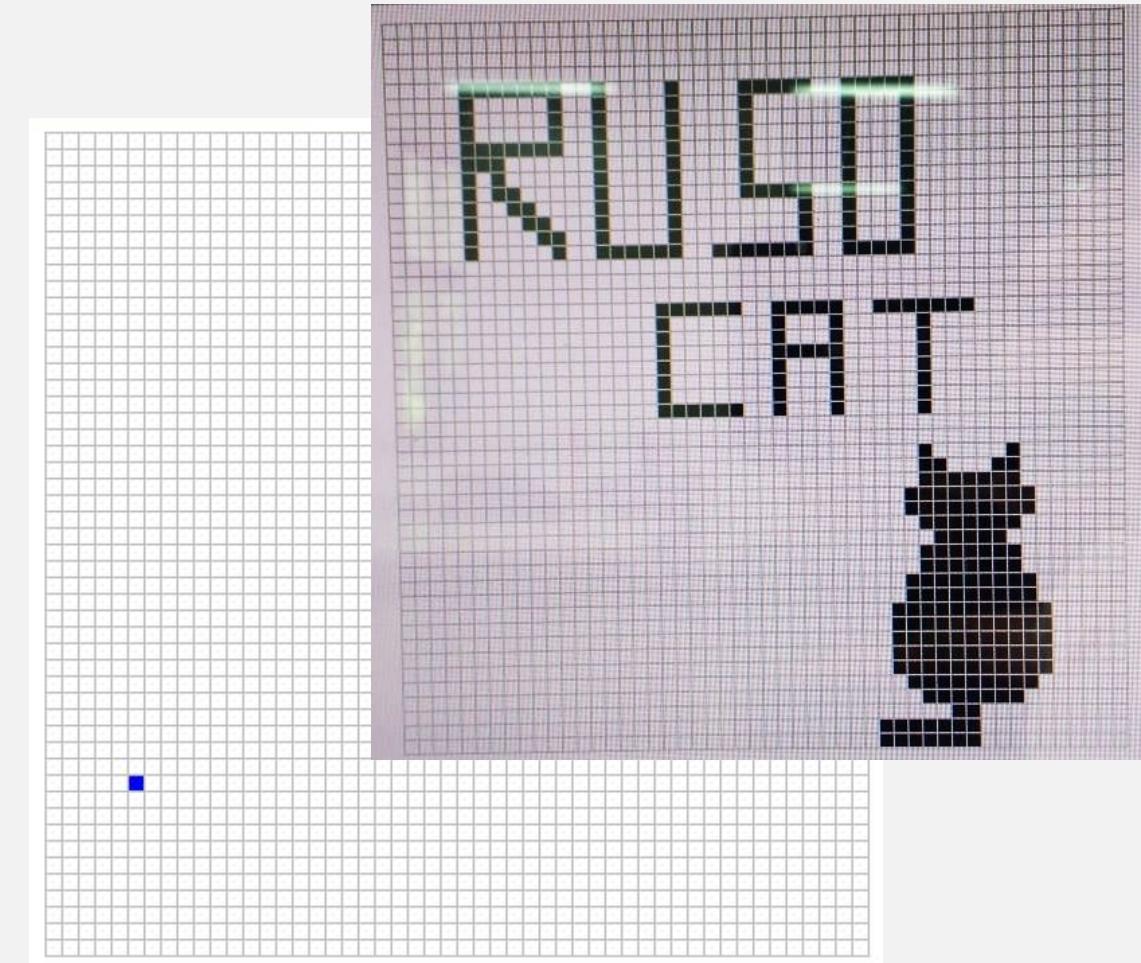
# Entrada/Salida

*Pantalla alfanumérica y gráfica*

MIPS cuenta con dos “pantallas”

Una **alfanumérica**, donde podemos imprimir texto

Y otra **gráfica** donde podemos pintar píxeles



# Entrada/Salida

*¿Cómo se usa?*

Existen dos “registros” (es decir, dos celdas de memoria comunes)

**CONTROL** sirve para enviar códigos de operaciones

**DATA** sirve para enviar o recibir datos

Como son celdas de memoria se leen y escriben con instrucciones de memoria: LD/L.D/LBU/SD/S.D...

Ejemplo mandando algo a CONTROL, por ejemplo, el valor 1

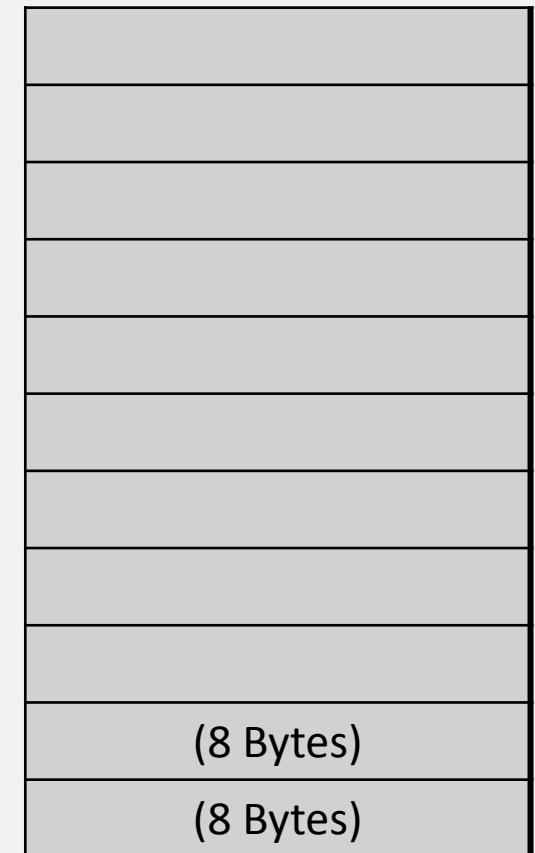
```
DADDI $t1, $zero, 1  
SD $t1, 0x10000 ($zero)
```

0x10000 no entra  
como dirección fija  
(máx. 2 bytes)

```
.data  
CONTROL: .word 0x10000
```

```
.code  
LD $s0, CONTROL ($zero)  
DADDI $t1, $zero, 1  
SD $t1, 0 ($s0)
```

**MEMORIA**



# Entrada/Salida

*Pantalla alfanumérica*

## Pantalla alfanumérica

Si se quiere **imprimir** un string (**NO** un carácter, un string completo!)

**DATA** -> Dirección del string

**CONTROL** -> El valor 4

Si se quiere **imprimir** un número

**DATA** -> El dato

**CONTROL** ->

- 1 -> Imprime un **entero sin signo**
- 2 -> Imprime un **entero con signo**
- 3 -> Imprime un **flotante**

Si se quiere **limpiar** la pantalla

**CONTROL** -> El valor 6

**.data**

**CONTROL**: .word 0x10000  
**DATA**: .word 0x10008

**.code**

LD \$s0, **CONTROL** (\$zero) ; \$s0 = CONTROL  
LD \$s1, **DATA** (\$zero) ; \$s1 = DATA

DADDI \$t0, \$zero, -85

SD \$t0, 0 (\$s1) ; Mando el dato a DATA

DADDI \$t0, \$zero, 2

SD \$t0, 0 (\$s0) ; CONTROL = 2

DADDI \$t0, \$zero, 6

SD \$t0, 0 (\$s0) ; CONTROL = 6 (limpia)

HALT

# Entrada/Salida

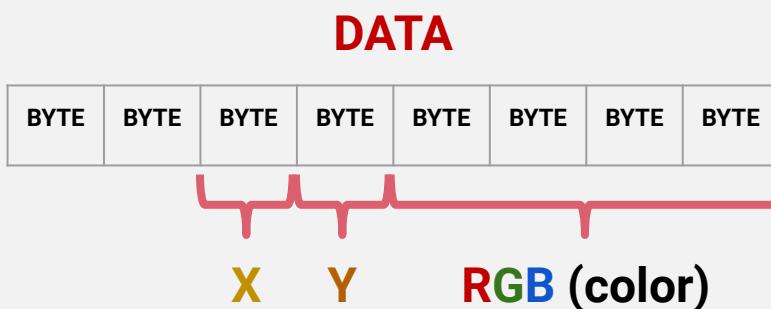
Pantalla gráfica

## Pantalla gráfica

Si se quiere **pintar un píxel**

**DATA** -> Color y coordenadas

**CONTROL** -> El valor 5



Si se quiere **limpiar la pantalla**

**CONTROL** -> El valor 7

### .data

**PIXEL**: .byte 0, 185, 135, 0, 23, 10, 0, 0

**CONTROL**: .word 0x10000

**DATA**: .word 0x10008

### .code

LD \$s0, **CONTROL** (\$zero) ; \$s0 = CONTROL

LD \$s1, **DATA** (\$zero) ; \$s1 = DATA

LD \$t0, **PIXEL** (\$zero)

SD \$t0, 0 (\$s1) ; Mando el dato a DATA

DADDI \$t0, \$zero, 5

SD \$t0, 0 (\$s0) ; CONTROL = 5

DADDI \$t0, \$zero, 7

SD \$t0, 0 (\$s0) ; CONTROL = 7 (limpia)

HALT

# Entrada/Salida

Pantalla gráfica. Ejemplo 2

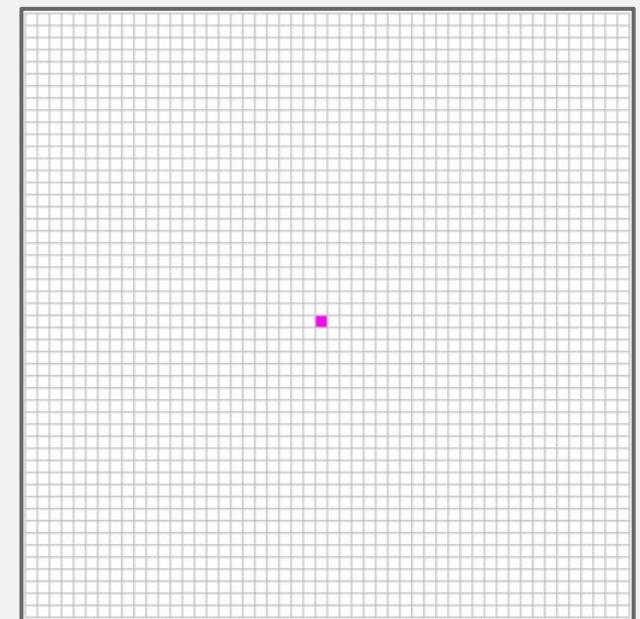
## .data

```
coorX: .byte 24 ; Coordenada X
coorY: .byte 24 ; Coordenada Y
color: .byte 255, 0, 255, 0 ; Máximo rojo + máximo azul = magenta
CONTROL: .word 0x10000
DATA: .word 0x10008
```

## .code

```
ld $s0, CONTROL ($zero)           ; $s0 = dirección de CONTROL
ld $s1, DATA ($zero)              ; $s1 = dirección de DATA
daddi $t0, $zero, 7                ; $t0 = 7 -> función 7: limpiar pantalla gráfica
sd $t0, 0 ($s0)                  ; CONTROL = 7 (limpia la pantalla gráfica)
lbu $t0, coorX ($zero)            ; $t0 = valor de coordenada X
sb $t0, 5 ($s1)                  ; DATA + 5 recibe el valor de coordenada X
lbu $t1, coorY ($zero)            ; $t1 = valor de coordenada Y
sb $t1, 4 ($s1)                  ; DATA + 4 recibe el valor de coordenada Y
```

```
lwu $t2, color ($zero) ; $t2 = color
sw $t2, 0 ($s1) ; Pongo color en DATA
daddi $t0, $zero, 5
sd $t0, 0 ($s0) ; Pinta el píxel
HALT
```



# Entrada/Salida

*Entrada*

# Entrada/Salida

## Teclado

Podemos leer un número o un carácter:

Si se quiere **leer un número (entero o flotante)**

**CONTROL** -> El valor 8

**DATA** ->

- **Muestra** el carácter presionado
- Termina de leer cuando presiona *Enter*
- Si el dato ingresado no es un número se guarda 0. Tomar el valor (**Hexadecimal**) con LD o L.D desde **DATA**

Si se quiere **leer un carácter**

**CONTROL** -> El valor 9

**DATA** ->

- **NO** muestra el carácter presionado
- No espera al *Enter*
- Tomar el valor (**ASCII**) con LBU desde **DATA**

**.data**

**CONTROL**: .word 0x10000

**DATA**: .word 0x10008

**NUM**: .double 0.0

**CARACTER**: .byte 0

**.code**

LWU \$s0, **CONTROL** (\$zero) ; \$s0 = CONTROL

LWU \$s1, **DATA** (\$zero) ; \$s1 = DATA

DADDI \$t0, \$zero, 8

SD \$t0, 0 (\$s0) ; CONTROL = 8

L.D f1, 0 (\$s1) ; Tomo número en f1

S.D f1, **NUM** (\$zero) ; Guardo en variable

DADDI \$t1, \$zero, 9

SD \$t1, 0 (\$s0) ; CONTROL = 9

LBU \$t1, 0 (\$s1) ; Tomo carácter en \$t1

SB \$t1, **CARACTER** (\$zero) ; Guardo en variable

HALT