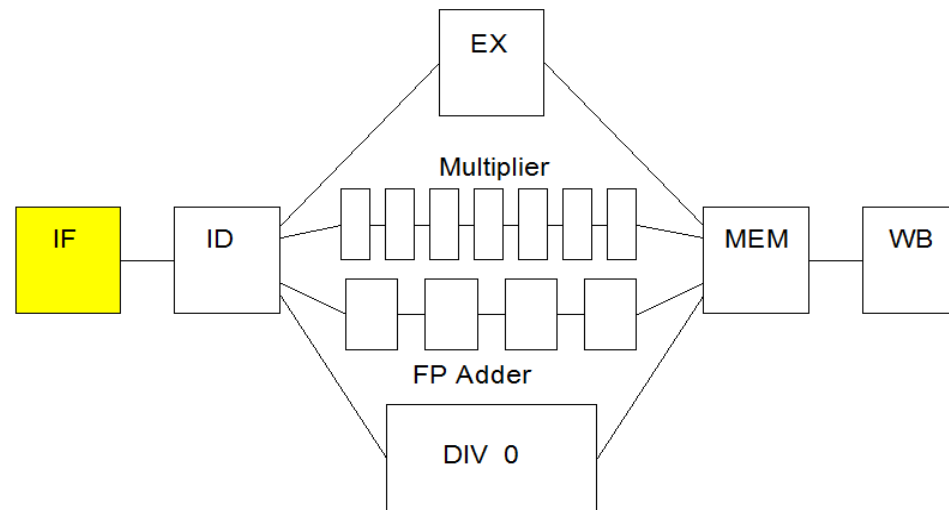


► Arquitectura de computadoras

Explicación Práctica 5

MIPS64 – Unidades de ejecución

- Para instrucciones generales (**1 ciclo**)
- Para instrucciones **aritméticas** de punto flotante:
 - a. Suma (**4 ciclos**)
 - b. Multiplicación (**7 ciclos**)
 - c. División (**24 ciclos**)

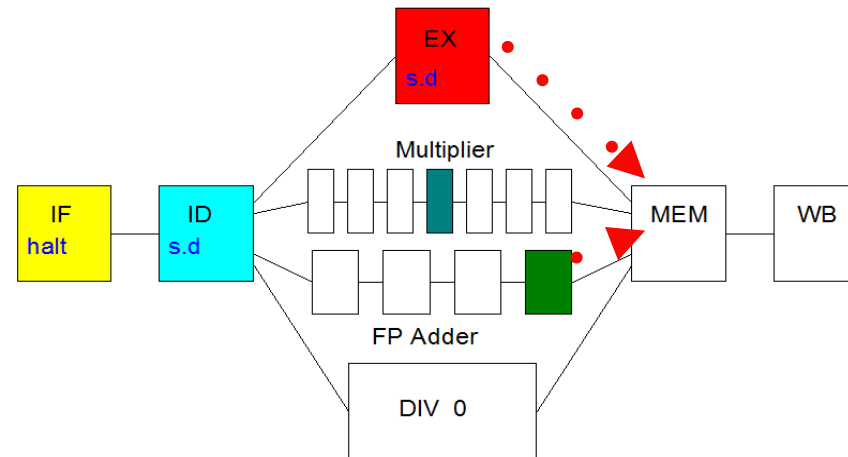


MIPS64 – Unidades de ejecución

- Al disponer de múltiples unidades:
 - Es posible ejecutar instrucciones en **menor tiempo**
 - Pero también introduce **nuevos problemas**
 - Atasco **Estructural**
 - Atasco por dependencia de datos **WAR**
 - Atasco por dependencia de datos **WAW**

MIPS64 – Atascos estructurales

- Son provocados por conflictos por los recursos
- En **WinMIPS** sólo puede suceder cuando dos instrucciones en unidades de ejecución distintas intentan acceder a la etapa **memoria** simultáneamente
- Solo puede avanzar una de las instrucciones involucradas → **tiene prioridad la primera que entró al pipeline**



MIPS64 – Atascos WAR y WAW

- Los atascos WAR y WAW suceden cuando:
 - Hay más de una unidad de ejecución
 - Hay dependencia de datos entre dos instrucciones
 - 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 WAR o WAW cuando detecta una situación potencial de conflicto (aunque realmente luego no suceda)

Ejercicio 1

Analizar la ejecución paso a paso (Forwarding habilitado)

.data

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

.code

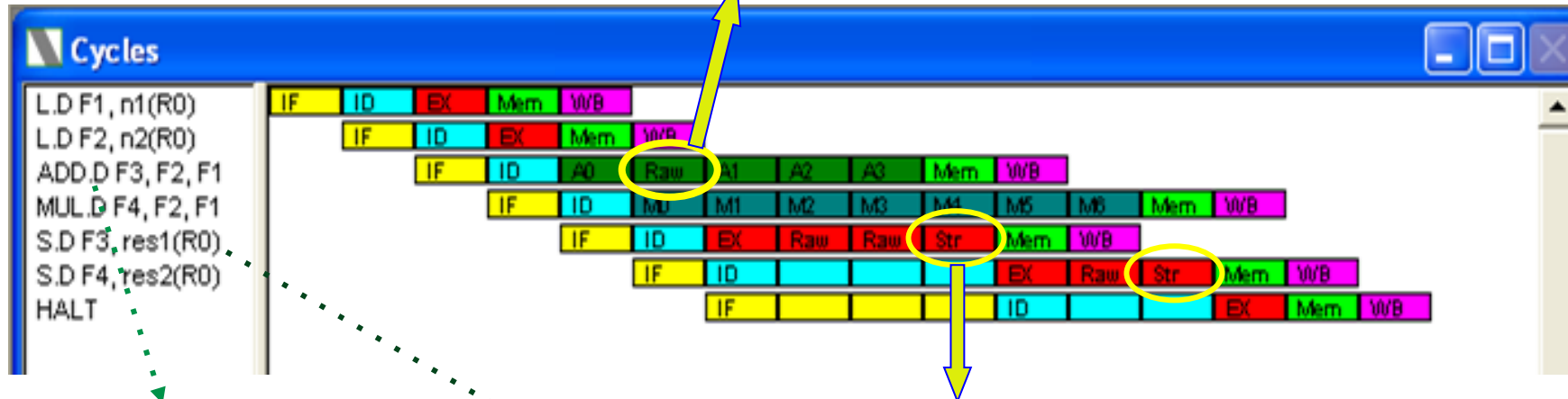
L.D F1, n1(R0)
L.D F2, n2(R0)
ADD.D F3, F2, F1
MUL.D F4, F2, F1
S.D F3, res1(R0)
S.D F4, res2(R0)
HALT

.word <n1>,<n2>... *define número(s) de 64-bits*
.word32 <n1>,<n2>... *define número(s) de 32 bits*
.word16 <n1>,<n2>... *define número(s) de 16 bits*
. byte <n1>, <n2>... *define número(s) de 8 bits*
.asciiz "abc"... *define una cadena de caracteres (c/u 1 byte)
terminada con el carácter 00H*
.double <n1>,<n2>... *define número(s) en punto flotante*

L.D = Load Double precision float
S.D = Store Double precision float

Ejercicio 1

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.

```
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
```

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

Forwarding habilitado

Ejercicio 1 (variante)

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)
L.D F2, n2(R0)
ADD.D F3, F2, F1
MUL.D F2, F2, F1
MUL.D F4, F2, F1
S.D F3, res1(R0)
S.D F4, res2(R0)
HALT
```


Ejercicio 1 (variante)

ADD.D intenta leer **F2** y no puede → **RAW**

MUL.D va a modificar **F2**. Como el **ADD.D** usa el valor de **F2** y está atascada, se atasca el **MUL.D** → **WAR**



```
0000 d4010000 L.D F1, n1(R0)
0004 d4020008 L.D F2, n2(R0)
0008 462110c0 ADD.D F3, F2, F1
000c 46211082 MUL.D F2, F2, F1
0010 46211102 MUL.D F4, F2, F1
0014 f4030010 S.D F3, res1(R0)
0018 f4040018 S.D F4, res2(R0)
001c 04000000 HALT
0020 00000000
0024 00000000
```

Statistics	
Execution	
24 Cycles	
8 Instructions	
3.000 Cycles Per Instruction (CPI)	
Stalls	
16 RAW Stalls	
0 WAW Stalls	
1 WAR Stall	
2 Structural Stalls	
0 Branch Taken Stalls	
0 Branch Misprediction Stalls	
Code size	
32 Bytes	

Forwarding habilitado

Ejercicio 1 (variante)

Agregamos la instrucción **NOP**

.data

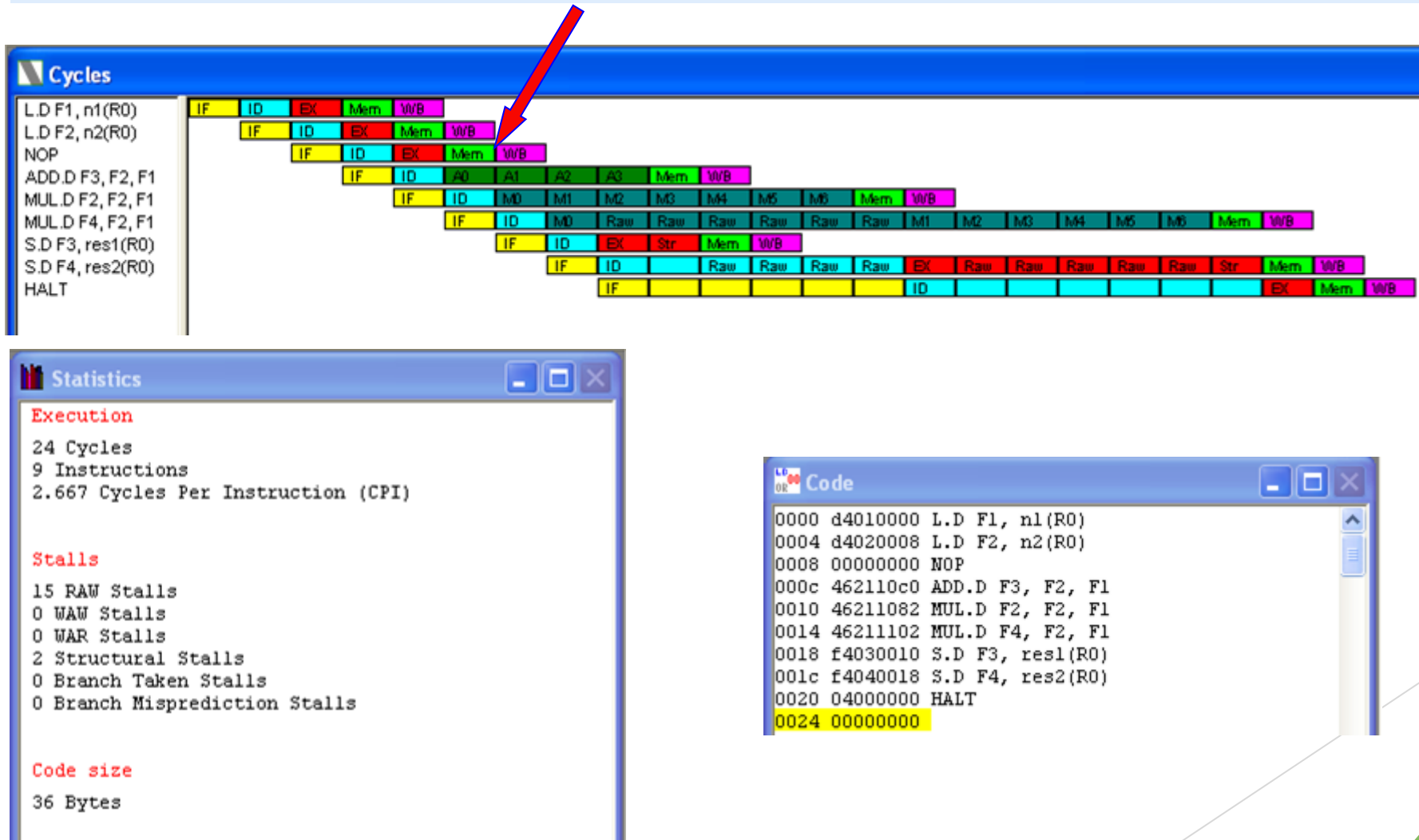
```
n1:      .double      9.13
n2:      .double      6.58
res1:    .double      0.0
res2:    .double      0.0
```

.code

```
L.D F1, n1(R0)
L.D F2, n2(R0)
NOP
ADD.D F3, F2, F1
MUL.D F2, F2, F1
MUL.D F4, F2, F1
S.D F3, res1(R0)
S.D F4, res2(R0)
HALT
```

Ejercicio 1 (variante)

F2 esta disponible ser leído por el **ADD.D**, por lo que ya no se provoca el **RAW**.
Por lo tanto, tampoco se provoca el **WAR** para el **MUL.D**.



MIPS64 – Subrutinas

- El soporte para la invocación a **subrutinas** es **mucho más reducido** que el que provee la arquitectura x86
- **No hay un manejo implícito de la pila** en la **invocación a las subrutinas** (instrucciones CALL y RET del x86)
- **No hay instrucciones explícitas** para **apilar y desapilar** valores en la pila (instrucciones PUSH y POP del x86)

MIPS64 – Subrutinas

```
result:      .data
              .word 0

              .text
daddi r4, r0, 10
daddi r5, r0, 20
jal sumar    # Se llama a la subrutina "sumar"
sd r2, result(r0)
halt

sumar:       dadd r2, r4, r5      # subrutina "sumar"
              jr r31             # Retorna al punto donde se llamó a "sumar"
```

MIPS64 – Subrutinas

Problemas

- **Registro de retorno**
 - Si el programa principal invoca a una subrutina y dentro de esa subrutina se necesita invocar a una segunda subrutina, el segundo jal **sobrescribirá el valor contenido en el registro r31** con la nueva dirección de retorno
- **Sobreescritura de registros**
 - Si una subrutina utiliza registros que la subrutina que la invocó estaba usando puede suceder que **los valores de los mismos no se conserven al retornar**

MIPS64 – Convención

- Para solucionar estos problemas se estableció una **convención** que define lo siguiente:
 - Una **función en particular** para c/u de los registros
 - El mecanismo para la **invocación a subrutinas**
 - La implementación de una **pila** para almacenar valores temporalmente

MIPS64 – Convención

Los registros: sus nombres y sus usos

\$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 tope de la pila. Debe ser salvado	(r29)
\$fp	<i>Frame Pointer</i> – Puntero de pila. Debe ser salvado	(r30)
\$at	<i>Assembler Temporary</i> – Reservado para ser usado por el ensamblador	(r1)
\$k0-\$k1	Para uso del kernel del sistema operativo	(r26-r27)
\$gp	<i>Global Pointer</i> – Puntero a zona de memoria estática. Debe ser salvado	(r28)

MIPS64 – Convención

```
.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

```
.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
```

MIPS64 – Convención

Preservación de registros

Una subrutina debe **garantizar que preserva los valores originales** de los registros: **\$s0..\$s7**, **\$ra**, **\$sp**, **\$fp**, **\$gp**

Así, una subrutina puede llamar a una **segunda subrutina** sabiendo que esta **no modificará el valor** de estos registros

Para lograr esto **es necesario contar con algún lugar** donde almacenar los valores originales de los registros antes de modificarlos

¿Cómo es posible hacer esto si todos los registros tienen funciones asignadas?

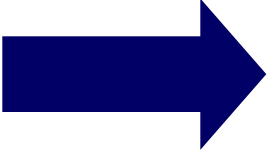
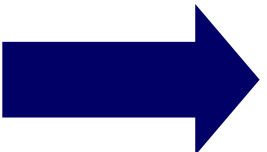
Almacenarlos en la **memoria principal**  **implementando una pila**

MIPS64 – Convención

Convención para el uso de la pila

Como **no hay instrucciones específicas** para manipular y mantener la pila, por convención todas las subrutinas usarán el registro **\$sp** (stack pointer) como puntero al tope de la pila.

Las operaciones **deberán ser implementadas**.

push \$t1		daddi \$sp, \$sp, -8 sd \$t1, 0(\$sp)
pop \$t1		ld \$t1, 0(\$sp) daddi \$sp, \$sp, 8

MIPS64 – Convención

Toda subrutina puede ser dividida 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

MIPS64 – Ejemplo

El siguiente ejemplo convierte los caracteres de una cadena dada a caracteres en mayúsculas

.data

cadena: .asciiz "Caza"

.text

; La pila comienza en el tope de la memoria de datos

DADDI \$sp, \$zero, 0x400

; Guarda como primer argumento para upcaseStr

DADDI \$a0, \$zero, cadena

JAL *upcaseStr*

HALT

MIPS64 – Ejemplo

; Parámetros:

\$a0 → inicio de cadena

; 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:

DADDI \$sp, \$sp, -16

SD \$ra, 0(\$sp)

SD \$s0, 8(\$sp)

DADD \$s0, \$a0, \$zero

upcaseStrLOOP:

LBU \$a0, 0(\$s0)

BEQ \$a0, \$zero, ***upcaseStrFIN***

JAL ***upcase***

SB \$v0, 0(\$s0)

DADDI \$s0, \$s0, 1

J ***upcaseStrLOOP***

upcaseStrFIN:

LD \$ra, 0(\$sp)

LD \$s0, 8(\$sp)

DADDI \$sp, \$sp, 16

JR \$ra

; Reserva lugar en la pila -> 2 x 8

; copia la dirección de inicio de la cadena

; recupera el car actual como argumento para upcase

; Si es el fin de la cadena, termina

; Guarda el carácter procesado en la cadena

; avanza al siguiente caracter

MIPS64 – Ejemplo

; La subrutina upcase tiene como función pasar un carácter a mayúscula

; Parámetros:

; \$a0 → carácter de entrada

; \$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 car sig a 'z' minúscula (z=7AH)

BEQZ \$t0, salir

; no es un carácter en minúscula

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

; pasa el carácter a mayúscula

salir: JR \$ra

; retorna al programa principal