PRACTICA 5

Procesador RISC: instrucciones de Punto Flotante y pasaje de parámetros

Objetivos: Familiarizarse con las instrucciones de MIPS64 para manejar datos en formato de Punto Flotante. Familiarizarse con los conceptos que rodean a la escritura de subrutinas en una arquitectura RISC: Uso normalizado de los registros, pasaje de parámetros y retorno de resultados, generación y manejo de la pila y anidamiento de subrutinas.

1) Simular el siguiente programa de suma de números en punto flotante y analizar minuciosamente la ejecución paso a paso. Inhabilitar Delay Slot y mantener habilitado Forwarding.

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

          .code
(1)       l.d         f1, n1(r0)
(2)       l.d         f2, n2(r0)
(3)       add.d       f3, f2, f1
(4)       mul.d       f4, f2, f1
(5)       s.d         f3, res1(r0)
(6)       s.d         f4, res2(r0)
(7)       halt
```

   a) Tomar nota de la cantidad de ciclos, instrucciones y CPI luego de la ejecución del programa.
   b) ¿Cuántos atascos por dependencia de datos se generan? Observar en cada caso cuál es el dato en conflicto y las instrucciones involucradas.
   c) ¿Por qué se producen los atascos estructurales? Observar cuales son las instrucciones que los generan y en qué etapas del pipeline aparecen.
   d) Modificar el programa agregando la instrucción **`mul.d`** `f1, f2, f1` entre las instrucciones **`add.d`** y **`mul.d`**. Repetir la ejecución y observar los resultados. ¿Por qué aparece un atasco tipo WAR?
   e) Explicar por qué colocando un NOP antes de la suma, se soluciona el RAW de la instrucción ADD y como consecuencia se elimina el WAR.

2) *Es posible convertir valores enteros almacenados en alguno de los registros `r1`-`r31` a su representación equivalente en punto flotante y viceversa. Describa la funcionalidad de las instrucciones **`mtc1`**, **`cvt.l.d`**, **`cvt.d.l`** y **`mfc1`**.

3) *Escribir un programa que calcule la superficie de un triángulo rectángulo de base 5,85 cm y altura 13,47 cm. Pista: la superficie de un triángulo se calcula como:

$$\text{Superficie} = (\text{base} \times \text{altura}) / 2$$

4) El índice de masa corporal (IMC) es una medida de asociación entre el peso y la talla de un individuo. Se calcula a partir del peso (expresado en kilogramos, por ejemplo: 75,7 kg) y la estatura (expresada en metros, por ejemplo 1,73 m), usando la fórmula:

$$\text{IMC} = \text{peso} / (\text{estatura})^2$$

   De acuerdo al valor calculado con este índice, puede clasificarse el estado nutricional de una persona en: **Infrapeso** (IMC < 18,5), **Normal** (18,5 ≤ IMC < 25), **Sobrepeso** (25 ≤ IMC < 30) y **Obeso** (IMC ≥ 30).
   Escriba un programa que dado el peso y la estatura de una persona calcule su IMC y lo guarde en la dirección etiquetada `IMC`. También deberá guardar en la dirección etiquetada `estado` un valor según la siguiente tabla:

IMC	Clasificación	Valor guardado
< 18,5	Infrapeso	1
< 25	Normal	2
< 30	Sobrepeso	3
≥ 30	Obeso	4

5) El procesador MIPS64 posee 32 registros, de 64 bits cada uno, llamados r0 a r31 (también conocidos como $0 a $31). Sin embargo, resulta más conveniente para los programadores darles nombres más significativos a esos registros. La siguiente tabla muestra la convención empleada para nombrar a los 32 registros mencionados:

Registros	Nombres	¿Para que se los utiliza?	¿Preservado?
r0	$zero	es siempre 0	
r1	$at	no lo usamos,	
r2-r3	$v0-$v1	resultados	
r4-r7	$a0-$a3	valores de respuesta	
r8-r15	$t0-$t7	variables temporales	
r16-r23	$s0-$s7	preservado en llamada	si
r24-r25	$t8-$t9	variables temporales	
r26-r27	$k0-$k1	para uso del kernel	
R28	$gp	global pointer	si
R29	$sp	posicion stack pointer	si
R30	$fp	frame pointer	si
R31	$ra	posicion a salto en RET	si

Complete la tabla anterior explicando el uso que normalmente se le da cada uno de los registros nombrados. Marque en la columna "¿Preservado?" si el valor de cada grupo de registros debe ser preservado luego de realizada una llamada a una subrutina. Puede encontrar información útil en el apunte "Programando sobre MIPS64".

6) Como ya se observó anteriormente, muchas instrucciones que normalmente forman parte del repertorio de un procesador con arquitectura CISC no existen en el MIPS64. En particular, el soporte para la invocación a subrutinas es mucho más simple que el provisto en la arquitectura x86 (pero no por ello menos potente). El siguiente programa muestra un ejemplo de invocación a una subrutina.

```
                .data
valor1:         .word 16
valor2:         .word 4
result:         .word 0

                .text
                ld    $a0, valor1($zero)
                ld    $a1, valor2($zero)
                jal   a_la_potencia
                sd    $v0, result($zero)
                halt

a_la_potencia:  daddi $v0, $zero, 1
         lazo:  slt   $t1, $a1, $zero
                bnez  $t1, terminar
                daddi $a1, $a1, -1
                dmul  $v0, $v0, $a0
                j     lazo
     terminar:  jr    $ra
```

a) ¿Qué hace el programa? ¿Cómo está estructurado el código del mismo?
b) ¿Qué acciones produce la instrucción **`jal`**? ¿Y la instrucción **`jr`**?
c) ¿Qué valor se almacena en el registro `$ra`? ¿Qué función cumplen los registros `$a0` y `$a1`? ¿Y el registro `$v0`?
d) ¿Qué sucedería si la subrutina `a_la_potencia` necesitara invocar a otra subrutina para realizar la multiplicación, por ejemplo, en lugar de usar la instrucción **`dmul`**? ¿Cómo sabría cada una de las subrutinas a que dirección de memoria deben retornar?

7) Escriba una subrutina que reciba como parámetros un número positivo M de 64 bits, la dirección del comienzo de una tabla que contenga valores numéricos de 64 bits sin signo y la cantidad de valores almacenados en dicha tabla. La subrutina debe retornar la cantidad de valores mayores que M contenidos en la tabla.

8) *Escriba una subrutina que reciba como parámetros las direcciones del comienzo de dos cadenas terminadas en cero y retorne la posición en la que las dos cadenas difieren. En caso de que las dos cadenas sean idénticas, debe retornar -1.

9) *Escriba la subrutina ES_VOCAL que determina si un caracter es vocal o no, ya sea mayúscula o minúscula. La rutina debe recibir el caracter y debe retornar el valor 1 si es una vocal ó 0 en caso contrario.

10) Usando la subrutina escrita en el ejercicio anterior, escriir la subrutina CONTAR_VOC, que recibe una cadena terminada en cero y devuelve la cantidad de vocales que tiene esa cadena.

11) Escribir una subrutina que reciba como argumento una tabla de números terminada en 0. La subrutina debe contar la cantidad de números que son impares en la tabla. Ésta condición se debe verificar usando la subrutina ES_IMPAR. La subrutina ES_IMPAR debe devolver 1 si el número es impar y 0 si no lo es.

12) El siguiente programa espera usar una subrutina que calcule en forma recursiva el factorial de un número entero:

```
                .data
valor:          .word 10
result:         .word 0

                .text
                daddi $sp, $zero, 0x400  ; Inicializa el puntero al tope de la pila (1)
                ld    $a0, valor($zero)
                jal   factorial
                sd    $v0, result($zero)
                halt

factorial:      ...
                ...
                ...
```

(1) La configuración inicial de la arquitectura del WinMIPS64 establece que el procesador posee un bus de direcciones de 10 bits para la memoria de datos. Por lo tanto, la mayor dirección dentro de la memoria de datos será de $2^{10} = 1024 = 400_{16}$.

a) *Implemente la subrutina `factorial` definida en forma recursiva. Tenga presente que el factorial de un número entero n se calcula como el producto de los números enteros entre 1 y n inclusive:

$$\text{factorial}(n) = n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1$$

b) ¿Es posible escribir la subrutina `factorial` sin utilizar una pila? Justifique.