Juan Menduiña

Trabajo Práctico N° 5:
Procesador RISC (Instrucciones de Punto Flotante y Pasaje de Parámetros).

Ejercicio 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
            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
```

(a) *Tomar nota de la cantidad de ciclos, instrucciones y CPI luego de la ejecución del programa.*

Ciclos: 16.
Instrucciones: 7.
CPI: 2,286.

(b) *¿Cuántos atascos por dependencia de datos se generan? Observar, en cada caso, cuál es el dato en conflicto y las instrucciones involucradas.*

Se generan 4 atascos por dependencia de datos RAW. El dato en conflicto y las instrucciones involucradas son:

- 1 RAW: *add.d f3, f2, f1* (en su etapa A0, debe esperar que la instrucción *l.d f2, N2(r0)*) finalice su etapa MEM.
- 2 RAW: *s.d f3, RES1(r0)* (en su etapa EX, debe esperar que la instrucción *add.d f3, f2, f1* finalice su etapa A3)
- 1 RAW: *s.d f4, RES2(r0)* (en su etapa EX, debe esperar que la instrucción *mul.d f4, f2, f1* finalice su etapa M6).

(c) *¿Por qué se producen los atascos estructurales? Observar cuáles son las instrucciones que los generan y en qué etapas del pipeline aparecen.*

Los atascos estructurales se producen por conflictos por los recursos. Las instrucciones que las generan y las etapas del *pipeline* que aparecen son:

- 1 Atasco Estructural: *s.d f3, RES1(r0)* (la etapa MEM de esta instrucción se encuentra, al mismo tiempo, con la etapa MEM de la instrucción *add.d f3, f2, f1*).
- 1 Atasco Estructural: *s.d f4, RES2(r0)* (la etapa MEM de esta instrucción se encuentra, al mismo tiempo, con la etapa MEM de la instrucción *mul.d f4, f2, f1*).

(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?*

```
            .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 f1, f2, f1
            mul.d f4, f2, f1
            s.d f3, RES1(r0)
            s.d f4, RES2(r0)
            halt
```

Aparece un atasco tipo WAR porque la instrucción *mul.d f1, f2, f1* (en su etapa ID) necesita escribir el registro F1 que aún la instrucción *add.d f3, f2, f1* (en su etapa A0) no leyó.

(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.*

```
            .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 f1, f2, f1
mul.d f4, f2, f1
s.d f3, RES1(r0)
s.d f4, RES2(r0)
halt
```

Colocando un NOP antes de la suma se soluciona el RAW de la instrucción ADD y, como consecuencia, se elimina el WAR porque, ahora, cuando la instrucción *mul.d f1, f2, f1* se encuentran en su etapa ID, la instrucción *add.d f3, f2, f1* finaliza su etapa A0.

Ejercicio 2.

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

La funcionalidad de las siguientes instrucciones es:

- *mtc1* r_f, f_d: Copia los 64 bits del registro entero r_f al registro de punto flotante f_d.
- *cvt.d.l* f_d, f_f: Convierte a punto flotante el valor entero copiado al registro f_f, dejándolo en f_d.
- *cvt.l.d* f_d, f_f: Convierte a entero el valor en punto flotante contenido en el registro f_f, dejándolo en f_d.
- *mfc1* r_d, f_f: Copia los 64 bits del registro de punto flotante f_f al registro entero r_d.

Ejercicio 3.

Escribir un programa que calcule la superficie de un triángulo rectángulo de base 5,85 cm y altura 13,47 cm. La superficie de un triángulo se calcula como: Superficie= $\frac{base*altura}{2}$.

```
            .data
BASE:       .double 5.85
ALTURA:     .double 13.47
MEDIO:      .double 0.5
RES:        .double 0.0

            .code
            l.d f1, BASE(r0)
            l.d f2, ALTURA(r0)
            l.d f4, MEDIO(r0)
            mul.d f3, f1, f2
            mul.d f5, f3, f4
            s.d f5, RES(r0)
            halt
```

Ejercicio 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: IMC= $\frac{peso}{altura^2}$*. De acuerdo al valor calculado con este índice, puede clasificarse el estado nutricional de una persona en: Infrapeso (IMC* $< 18,5$*), Normal (*$18,5 \leq IMC < 25$*), Sobrepeso (*$25 \leq IMC < 30$*) y Obeso (IMC* ≥ 30*). Escribir 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*	*5*

```
                .data
ESTATURA:       .double 1.65
PESO:           .double 83.0
INFRAPESO:      .double 18.5
NORMAL:         .double 25.0
SOBREPESO:      .double 30.0
IMC:            .double 0.0
ESTADO:         .word 0

                .code
                l.d f1, ESTATURA(r0)
                mul.d f6, f1, f1
                l.d f2, PESO(r0)
                l.d f3, INFRAPESO(r0)
                l.d f4, NORMAL(r0)
                l.d f5, SOBREPESO(r0)
                div.d f7, f2, f6
                c.lt.d f7, f3
                bc1t INFRA
                c.lt.d f7, f4
                bc1t NORM
                c.lt.d f7, f5
                bc1t SOBRE
                daddi r1, r0, 4
                j FIN
INFRA:          daddi r1, r0, 1
                j FIN
NORM:           daddi r1, r0, 2
                j FIN
SOBRE:          daddi r1, r0, 3
```

```
FIN:    s.d f7, IMC(r0)
        sd r1, ESTADO(r0)
        halt
```

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

Registro	Nombre	¿Para qué se lo utiliza?	¿Preservado?
r0	$zero	Siempre tiene valor 0 y no se puede cambiar	
r1	$at	*Assembler Temporary* - Reservado para ser usado por el ensamblador	
r2-r3	$v0-$v1	Valores de retorno de la subrutina llamada	
r4-r7	$a0-$a3	Argumentos pasados a la subrutina llamada	
r8-r15	$t0-$t7	Registros temporarios. No son conservados en el llamado a subrutinas	
r16-r23	$s0-$s7	Registros salvados durante el llamado a subrutinas	x
r24-r25	$t8-$t9	Registros temporarios. No son conservados en el llamado a subrutinas	
r26-r27	$k0-$k1	Para uso del kernel del sistema operativo	
r28	$gp	*Global Pointer* - Puntero a la zona de la memoria estática del programa	x

r29	$sp	*Stack Pointer* - Puntero al tope de la pila	x
r30	$fp	*Frame Pointer* - Puntero al marco actual de la pila	x
r31	$ra	*Return Address* - Dirección de retorno en un llamado a una subrutina	x

Ejercicio 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?*

El programa calcula $16^4 = 65.536$ y almacena el resultado en la variable RESULT. En la variable VALOR1, guarda la base de la potencia y, en el VALOR2, guarda el exponente. Luego, carga estos valores en los registros y salta a una subrutina que se encarga de calcular la potencia y guarda el resultado en el registro $v0.

(b) *¿Qué acciones produce la instrucción jal? ¿Y la instrucción jr?*

La instrucción *jal* salta a la dirección de memoria de la subrutina A_LA_POTENCIA y copia en *$ra* la dirección de retorno. Y la instrucción *jr* salta a la dirección contenida en *$ra*.

(c) *¿Qué valor se almacena en el registro $ra? ¿Qué función cumplen los registros $a0 y $a1? ¿Y el registro $v0?*

El valor que se almacena en el registro *$ra* es la dirección de memoria de la instrucción siguiente al llamado de la subrutina A_LA_POTENCIA. La función que cumplen los

registros *$a0* y *$a1* son de argumentos/parámetros pasados a la subrutina llamada (en este caso, la base y el exponente). Y el registro *$v0* contiene el valor de retorno de la subrutina llamada (en este caso, el resultado de la potencia).

(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?*

Si la subrutina A_LA_POTENCIA necesitara invocar a otra subrutina para la realizar la multiplicación, esta subrutina volvería a la dirección de retorno incorrecta (la de la subrutina interna). Para que cada una de las subrutinas sepa a qué dirección de memoria deben retornar se debe guardar el *$ra* de la primera subrutina usando la pila (*push $ra*) y, una vez que se retorna de la segunda subrutina, se debe recuperar el *$ra* de la primera subrutina usando la pila (*pop $ra*).

Ejercicio 7.

Escribir 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.

```
                .data
TABLA:          .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
M:              .word 5
TAM:            .word 10
RES:            .word 0

                .code
                daddi $sp, $0, 0x400
                ld $a0, M($0)
                ld $a1, TAM($0)
                daddi $a2, $0, TABLA
                daddi $a0, $a0, 1
                jal EMPEZAR
                sd $v0, RES($0)
                halt

EMPEZAR:        daddi $sp, $sp, -16
                sd $ra, 0($sp)
                sd $s2, 8($sp)
                dadd $t0, $a1, $0
                dadd $s2, $a2, $0
                dadd $v0, $0, $0
LAZO:           ld $t1, 0($s2)
                slt $t2, $t1, $a0
                bnez $t2, MENOR
                daddi $v0, $v0, 1
MENOR:          daddi $t0, $t0, -1
                daddi $s2, $s2, 8
                bnez $t0, LAZO
                ld $ra, 0($sp)
                ld $s2, 8($sp)
                daddi $sp, $sp, 16
                jr $ra
```

Ejercicio 8.

Escribir 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.

```
            .data
CAD1:       .asciiz  "abcde"
CAD2:       .asciiz  "abcd"
RES:        .word 0

            .code
            daddi $sp, $0, 0x400
            daddi $a0, $0, CAD1
            daddi $a1, $0, CAD2
            jal COMP
            sd $v0, RES($0)
            halt

COMP:       daddi $sp, $sp, -24
            sd $ra, 0($sp)
            sd $s0, 8($sp)
            sd $s1, 16($sp)
            dadd $s0, $a0, $0
            dadd $s1, $a1, $0
            dadd $v0, $0, $0
LAZO:       lbu $t0, 0($s0)
            lbu $t1, 0($s1)
            beqz $t0, FIN1
            beqz $t1, FIN2
            bne $t0, $t1, FIN2
            daddi $v0, $v0, 1
            daddi $s0, $s0, 1
            daddi $s1, $s1, 1
            j LAZO
FIN1:       bnez $t1, FIN2
            daddi $v0, $v0, -1
FIN2:       ld $ra, 0($sp)
            ld $s0, 8($sp)
            ld $s1, 16($sp)
            daddi $sp, $sp, 24
            jr $ra
```

Ejercicio 9.

Escribir 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 o 0 en caso contrario.

```
                .data
VOCALES:        .asciiz "AEIOUaeiou"
CHAR:           .ascii "a"
RES:            .word 0

                .code
                daddi $sp, $0, 0x400
                lbu $a0, CHAR($0)
                daddi $a1, $0, VOCALES
                jal ES_VOCAL
                sd $v0, RES($0)
                halt

ES_VOCAL:       daddi $sp, $sp, -16
                sd $ra, 0($sp)
                sd $s1, 8($sp)
                dadd $v0, $0, $0
                dadd $s1, $a1, $0
LAZO:           lbu $t1, 0($s1)
                beqz $t1, FIN
                beq $a0, $t1, VOCAL
                daddi $s1, $s1, 1
                j LAZO
VOCAL:          daddi $v0, $v0, 1
FIN:            ld $ra, 0($sp)
                ld $s1, 8($sp)
                jr $ra
```

Ejercicio 10.

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

```
                .data
VOCALES:        .asciiz "AEIOUaeiou"
CADENA:         .ascii "AbCdE"
RES:            .word 0

                .code
                daddi $sp, $0, 0x400
                daddi $a0, $0, CADENA
                jal CONTAR_VOC
                sd $v1, RES($0)
                halt

CONTAR_VOC:     daddi $sp, $sp, -16
                sd $ra, 0($sp)
                sd $s0, 8($sp)
                daddi $a1, $0, VOCALES
                dadd $v1, $0, $0
                dadd $s0, $a0, $0
LAZO1:          lbu $a0, 0($s0)
                beqz $a0, FIN1
                jal ES_VOCAL
                dadd $v1, $v1, $v0
                daddi $s0, $s0, 1
                j LAZO1
FIN1:           ld $ra, 0($sp)
                ld $s0, 8($sp)
                jr $ra

ES_VOCAL:       daddi $sp, $sp, -8
                sd $s1, 0($sp)
                dadd $v0, $0, $0
                dadd $s1, $a1, $0
LAZO2:          lbu $t1, 0($s1)
                beqz $t1, FIN2
                beq $a0, $t1, VOCAL
                daddi $s1, $s1, 1
                j LAZO2
VOCAL:          daddi $v0, $v0, 1
FIN2:           ld $s1, 0($sp)
                daddi $sp, $sp, 8
                jr $ra
```

Ejercicio 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. Esta 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.

```
                .data
TABLA:          .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0
RES:            .word 0

                .code
                daddi $sp, $0, 0x400
                daddi $a0, $0, TABLA
                jal CANT_IMP
                sd $v1, RES($0)
                halt

CANT_IMP:       daddi $sp, $sp, -16
                sd $ra, 0($sp)
                sd $s0, 8($sp)
                dadd $v1, $0, $0
                dadd $s0, $a0, $0
LAZO:           ld $a0, 0($s0)
                beqz $a0, FIN1
                jal ES_IMPAR
                dadd $v1, $v1, $v0
                daddi $s0, $s0, 8
                j LAZO
FIN1:           ld $ra, 0($sp)
                ld $s0, 8($sp)
                daddi $sp, $sp, 16
                jr $ra

ES_IMPAR:       dadd $v0, $0, $0
                andi $t0, $a0, 1
                beqz $t0, FIN2
                daddi $v0, $v0, 1
FIN2:           jr $ra
```

Ejercicio 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
                ld $a0, VALOR($zero)
                jal FACTORIAL
                sd $v0, RESULT($zero)
                halt
FACTORIAL:      ...
                ...
                ...
```

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

```
                .data
VALOR:          .word 10
RESULT:         .word 0

                .code
                daddi $sp, $0, 0x400
                ld $a0, VALOR($0)
                jal FACTORIAL
                sd $v0, RESULT($0)
                halt

FACTORIAL:      daddi $sp, $sp, -16
                sd $ra, 0($sp)
                sd $s0, 8($sp)
                beqz $a0, FIN1
                dadd $s0, $a0, $0
                daddi $a0, $a0, -1
                jal FACTORIAL
                dmul $v0, $v0, $s0
                j FIN2
FIN1:           daddi $v0, $0, 1
FIN2:           ld $ra, 0($sp)
                ld $s0, 8($sp)
                daddi $sp, $sp, 16
                jr $ra
```

(b) *¿Es posible escribir la subrutina factorial sin utilizar una pila? Justificar.*

No, no es posible escribir la subrutina FACTORIAL sin utilizar una pila, ya que es lo que permite guardar los distintos *$ra* y *$s0* de la recursión.