PRÁCTICA 4

Introducción al WinMIPS64.
Pila, Subrutina y Convención

Objetivos: Familiarizarse con el desarrollo de programas para procesadores con sets reducidos de instrucciones (RISC). Resolver problemas y verificarlos a través de simulaciones. 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.

Parte 1: Introducción al set de instrucciones del WinMIPS64

1. Tipos de instrucciones en WinMIPS64 ☆

Para cada una de las siguientes instrucciones del WinMIPS64, indicar si son instrucciones de salto incondicional (SI), salto condicional (SC), de lectura de memoria (LMEM), escritura de memoria (EMEM), o aritmético-lógicas (AL):

and	bnez	halt	slt	ld	sb
andi	dadd	nop	slti	lb	sd
beq	daddi	or	j	lbu	sw
bne	dmul	ori	jal	lw	xor
beqz	ddiv		jr	lwu	xori

2. VonSim vs WinMIPS64 ☆

Indicar qué instrucciones se corresponden entre el simulador VonSim y el simulador WinMIPS64. Tener en cuenta que algunas de las instrucciones del VonSim pueden hacerse de distintas formas en WinMIPS64. y otras no pueden hacerse con una sola instrucción. Por ende, algunas instrucciones de la columna izquierda corresponden a varias de la derecha, y otras no corresponden a ninguna

Nota: para simplificar, usaremos los registros del WinMIPS64 (r0 a r31) como nombres de registro en las instrucciones del VonSim.

VonSim

```
mov r1, r2
mov r1, 1
mov r1, 0
add r1, r2
add r1, 1
add r1, 0
inc r1
dec r1
or r1, r2
or r1, 1
or r1, 0
mov r1, variable
mov variable, r1
add r1, variable
add variable, r1
mov r1, offset variable
jump etiqueta
call etiqueta
hlt
```

WinMIPS64

```
ld r1, variable(r0)
sd r1, variable(r0)
halt
jal etiqueta
j etiqueta
or r1, r1, r0
or r1, r1, r2
ori r1, r1, 1
dadd r1, r1, r0
daddi r1, r0, 0
daddi r1, r0, 1
dadd r1, r0, r0
dadd r1, r0, r2
daddi r1, r1, 1
dadd r1, r1, r2
daddi r1, r0, variable
daddi r1, r2, 0
daddi r1, r1, -1
```

3. Tipos de variables en WinMIPS64 ☆

Indicar qué tamaño (en bytes) ocupan cada uno de estos tipos de datos del simulador WinMIPS64. En el caso en que dos tipos de datos tengan el mismo tamaño, explicar sus diferencias:

Tipo de dato	Tamaño en bytes	Uso
space 1		
ascii		
asciiz		
byte		
word16		
word32		
word		

4. Registros del MIPS64 ☆.

El procesador MIPS64 posee 32 registros, de 64 bits cada uno, llamados r0 a r31 (también conocidos como $0 a $31). Sin embargo, al programar resulta más conveniente darles nombres más significativos a esos registros.

La siguiente tabla muestra la convención empleada para nombrar a los 32 registros mencionados:

Registros	Nombres	¿Para qué se los utiliza?	¿Preservado?
r0	$zero		
r1	$at		
r2-r3	$v0-$v1		
r4-r7	$a0-$a3		
r8-r15	$t0-$t7		
r16-r23	$s0-$s7		
r24-r25	$t8-$t9		
r26-r27	$k0-$k1		
r28	$gp		
r29	$sp		
r30	$fp		
r31	$ra		

Complete la tabla anterior explicando el uso que normalmente se le da a 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".

5. Programas simples ☆☆.

Escribir un programa en WinMIPS64 que:

a) Lee 2 números A y B de la memoria de datos a registros, y calcula S, P y D, que luego se guardan en la memoria de datos.

b) Dadas dos variables A y B de la memoria, calcula y almacena C

c) Calcula el factorial de N, y lo guarda en F

d) Calcula el logaritmo (entero) en base 2 de N (N positivo) mediante divisiones sucesivas y lo guarda en L

e) Guarda en B el valor 1 si A es impar y 0 de lo contrario

Los nombres de variables como A, B, C, etc, deben implementarse usando variables de tipo **word** a las cuales se lee y escribe de memoria con **ld** y **sd**.

a

```
S = A + B
P =  2 + (A*B)
D = A^2 / B
```

b

```
if A == 0:
     C = 0
else:
     if A > B:
          C = A * 2
     else:
          C = B
```

c

```
F = 1
for    i=1..N:
  F = F * i
```

d

```
L = 0
while N > 1:
   N = N / 2
   L = L + 1
```

e

```
if impar(A):
   B = 1
else:
   B = 0
```

6. Recorrido básico de vectores ☆

Dado un vector definido como: `V: .word 5, 2, 6`, escribir programas para:

a) Calcular la suma de los 3 valores sin utilizar un loop o lazo

Pista: Usar tres instrucciones `ld $t1, V($t2)`, donde $t2 va aumentando de a 8 bytes el desplazamiento.
b) Calcular la suma de los 3 valores utilizando un lazo con la dirección base y un registro como desplazamiento.
Pista: Idem anterior, pero ahora con una única instrucciones de lectura y $t2 se incrementa dentro de un loop
c) Calcular la suma de los 3 valores utilizando un lazo, con una dirección base de 0 y un registro como puntero.
Pista: Cargar la dirección en un registro con `daddi $t2, $zero, V` y luego cargar los valores con `ld $t1, 0($t2)`
d) ¿Qué cambios se deberían realizar al programa del inciso b) si los elementos fueran de 32 bits: `V: .word32 5,2,6`?

7. Codificación de Strings ☆

a) **Strings en memoria**: En WinMIPS los strings se almacenan como vectores de caracteres. Observar cómo se almacenan en memoria los códigos ASCII de los caracteres (código de la letra "a" es 61H). Además, los strings se suelen terminar con un carácter de fin con código 0, es decir el valor 0. Para definir un string en una variable y no tener que agregar a mano el valor 0, se puede utilizar el tipo de datos **asciiz**. Si en lugar de eso se utiliza el tipo **ascii** (sin la **z)** no se agrega el 0 de forma automática. Probar estas definiciones en el simulador y observar cómo se organizan en la memoria las variables.

```
.data
cadena:  .asciiz "ABCdef1"
cadena2: .ascii "ABCdef11"
cadena3: .asciiz "ABCdef1111111"
num:     .word 5
```

b) **Lectura de caracteres: lbu vs. lb vs. ld**: En un string, cada código ASCII ocupa 1 byte. No obstante, los registros del simulador tienen 8 bytes (64 bits) de capacidad. Por ende, al cargar un byte en un registro, se desperdician 7 bytes de espacio; esta ineficiencia es inevitable. Más allá de eso, esta diferencia nos trae otra dificultad. Cuando se carga un valor de memoria, no se puede utilizar **ld**, porque ello traería 8 caracteres (8 bytes) al registro, y sería muy difícil hacer operaciones sobre los caracteres individuales. Por ello, existen instrucciones para traer solo un byte desde la memoria: **lbu** y **lb**. ¿Cuál es la diferencia?
- **lbu** asume que el valor que se trae está codificado en BSS y entonces rellena los últimos **7 bytes** con **0**.
- **lb** asume que el valor que se trae de memoria está codificado en CA2, y entonces si el número es negativo realiza la *expansión de signo*. ¿De qué trata esto? Para que el número siga valiendo lo mismo en CA2 de 8 **bytes**, se rellenan los últimos **7 bytes** con **1**.
Para probar la diferencia entre estas 3 instrucciones, ejecutar el siguiente programa en el simulador que intenta cargar el primer valor del vector de números **datos** y observar los valores finales de $t1, $t2 y $t3.

```
.data
datos: .byte -2, 2, 2, 2, 2, 2
.code
ld $t1, datos($zero)
lb $t2, datos($zero)
lbu $t3, datos($zero)
halt
```

Responde:
a) ¿Qué registro tiene el valor "correcto" del primer valor?
b) ¿Qué instrucción deberías utilizar de las 3 para cargar un código ASCII que siempre es positivo? Tené en cuenta que, por ejemplo, el código ASCII de la **Á** es 181, que en BSS se escribe como **10110101**.

Parte 2: Pila y Subrutinas.

1. Comprendiendo la primer subrutina: potencia ☆

Muchas instrucciones que normalmente forman parte del repertorio de un procesador con arquitectura CISC como PUSH y POP no existen en el MIPS64. En particular, el soporte para la invocación a subrutinas es más simple que el provisto en la arquitectura x86, y por eso tenemos que hacer algunos pasos extra para usarlas. El siguiente programa muestra un ejemplo de invocación a una subrutina llamada **potencia**:

```
.data
base: .word 5
exponente: .word 4
result: .word 0

.code
ld    $a0, base($zero)
ld    $a1, exponente($zero)
jal   potencia
sd    $v0, result($zero)
halt
```

```
potencia: daddi  $v0, $zero, 1
    lazo: beqz   $a1, 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`? ¿Qué valores posibles puede recibir en **$a0** y **$a1** la subrutina **potencia**?
d) Supongamos que el WinMIPS no posee la instrucción **dmul** ¿Qué sucede si la subrutina `potencia` necesita invocar a otra subrutina para realizar la multiplicación en lugar de usar la instrucción `dmul`? ¿Cómo sabe cada una de las subrutinas a que dirección de memoria debe retornar?
e) Escriba un programa que utilice **potencia**. En el programa principal se solicitará el ingreso de la base y del exponente (ambos enteros) y se deberá utilizar la subrutina **potencia** para calcular el resultado pedido. Muestre el resultado numérico de la operación en pantalla.
f) Escriba un programa que lea un exponente **x** y calcule **2^x + 3^x** utilizando dos llamadas a **potencia**. Muestre en pantalla el resultado. ¿Funciona correctamente? Si no lo hace, revise su implementación del programa ¿Qué sucede cuando realiza una segunda llamada a **potencia**? **Pista**: Como caso de prueba, intente calcular **$2^3+3^3 = 8+27 = 35$.**.

2. Salvado de registros ☆

Los siguientes programas tienen errores en el uso de la convención de registros. Indicar en qué registros, cuál es el error y cómo se podría arreglar el problema en cada caso.

A)
```
.code

daddi $t0, $0, 5
daddi $t1, $0, 7
jal subrutina
sd $t2, variable ($0)
halt

subrutina: daddi $t4, $0, 2
        dmul $t0, $t0, $t4
        dmul $t1,$t1,$t4
        dadd $t2,$t1,$t0
        jr $ra
```

B)
```
.code
    daddi $a0, $0, tabla
    jal subrutina
    daddi $t0, $0, 10
    daddi $t1, $0, 0
loop: bnez $t0, fin
    ld $t2, 0($a0)
    dadd $t1, $t1, $t2
    daddi $t0, $t0, -1
    daddi $a0, $a0, 8
    j loop
fin: halt
```

C)
```
.code
daddi $a0, $0, 5
daddi $a1, $0, 7
jal subrutina
dmul $t2, $a0, $v0
sd $t2, variable ($0)
halt
```

D)
```
.code
    daddi $t0, $0, 10 # dimension
    daddi $t1, $0, 0 # contador
    daddi $t2, $0, 0 # desplazamiento
loop: bnez $t0, fin
    ld $a0, tabla ($t2)
    jal espar
    bnez $v0, seguir
    dadd1 $t1, $t1, 1
seguir: daddi $t2, $t2, 1
    daddi $t0, $t0, -1
    daddi $t2, $t2, 8
    j loop
   sd $t1, resultado($0)
fin: halt
```

3. Uso de la pila ⭐

En WinMIPS no existen las instrucciones **PUSH** y **POP**. Por ese motivo, deben implementarse utilizando otras instrucciones existentes. No solo eso, sino que el registro SP es en realidad un registro usual, r29, que con la convención se puede llamar por otro nombre, **$sp**. El siguiente programa debería intercambiar los valores de $t0 y $t1 utilizando la pila. No obstante, así como está no va a funcionar porque push y pop no son instrucciones válidas. Implementar la funcionalidad que tendrían estas operaciones utilizando instrucciones daddi, sd y ld para que el programa funcione correctamente. Recordar que los registros ocupan 8 bytes, y por ende el push y el pop deberán modificar a $sp con ese valor.

```
.code
daddi $sp, $0, 0x400
daddi $t0, $0, 5
daddi $t1, $0, 8
push $t0
push $t1
pop $t0
pop $t1
halt
```

4. Pasaje por registro y por referencia ⭐⭐

La versión anterior de **potencia** utiliza pasaje por registros y por valor. Escribir otra versión que reciba los parámetros por referencia desde el programa principal a través de registros, y devuelva el resultado a través de un registro por valor. Adaptar el programa principal de prueba de forma acorde.

5. Salvado de registros en subrutinas anidadas ⭐⭐

Las siguientes subrutinas anidadas funcionan, pero tienen errores en el uso de la convención de los registros, en especial con respecto a cuales tienen que salvarse y cuáles no, y también cuándo y en qué caso debe hacerse. Indicar los errores y corregir el código para que las subrutinas usen la convención correctamente.

A)
```
#v0: devuelve 1 si a0 es impar y 0 dlc
#a0: número entero cualquiera
esimpar: andi $v0, $a0, 1
         jr $ra

#v0: devuelve 1 si a0 es par y 0 dlc
#a0: número entero cualquiera
espar: jal esimpar
       #truco: espar = 1 - esimpar
       daddi $s0, $0, 1
       dsub $v0, $s0, $v0
       jr $ra
```

B)
```
#v0: devuelve la cantidad de bits 0 que
tiene un número de 64 bits
#a0: número entero cualquiera
cant0: daddi $t0, $0, 0
       daddi $t1, $0, 64
  loop:jal espar
       dadd $t0,$t0,$v0
       #desplazo    a    la    derecha
       #para quitar el último bit
       dsrl $a0, $a0, 1
       daddi $t1, $t1, -1
       bnez $t1, loop
       jr $ra
```

C)
```
#v0: volumen de un cubo
#a0: long del lado lado del cubo
vol: daddi $sp, $sp, -16
     sd $ra, 0($sp)
     sd $s0, 8($sp)
     dadd $s0, $0, $a0
     dmul $s0,$a0,$a0
     dmul $s0,$s0,$a0
     daddi $v0,$s0,0
     ld $ra, 0($sp)
     ld $s0, 8($sp)
     daddi $sp, $sp, 16
     jr $ra

#v0: diferencia de volumen de los cubos
#a0: long del lado del cubo más grande
#a1: long del lado del cubo más chico
diffvol: jal vol
         daddi $t0,$v0,0
         daddi $a0,$a1,0
         jal vol
         dsub $v0,$t0,$v0
         jr $ra
```

6. Elevar vector al cuadrado ☆☆

Escribir una subrutina **vector_cuadrado**, que reciba la dirección de un vector de números, por referencia, y su longitud, por valor, y eleve cada uno de ellos al cuadrado. Para ello, utilizar la subrutina **potencia** implementada previamente.

7. Pasaje por pila ☆☆

La versión anterior de **potencia** utiliza pasaje por registros y por valor, y ya escribiste una versión alternativa que recibe los parámetros por referencia. Ahora adapta la subrutina y el programa asociado para estos casos:

a) **Valor y Pila** Pasando los parámetros por valor desde el programa principal a través de la pila, y devolviendo el resultado a través de un registro por valor.

b) **Referencia y Pila** Pasando los parámetros por referencia desde el programa principal a través de la pila, y devolviendo el resultado a través de un registro por valor

Parte 3: Ejercicios de repaso o tipo parcial

1 Operaciones con strings ☆

a) **Longitud de un string** Escribir un programa que cuente la longitud de un string iterando el mismo hasta llegar al valor 0 y guarde el resultado en una variable llamada **LONGITUD.** Probarlo con el string “ArquiTectuRa de ComPutaDoras”.
b) **Contar apariciones de carácter** Escribir un programa que cuente la cantidad de veces que un determinado carácter aparece en una cadena de texto.

```
.data
cadena: .asciiz "adbdcdedfdgdhdid"  # cadena a analizar
car:    .ascii "d"                  # carácter buscado
cant:   .word 0           # cantidad de veces que se repite el carácter car en cadena.
```

c) **Contar mayúsculas** Escribir un programa que cuente la cantidad de letras mayúsculas de un string. Probarlo con el string “ArquiTectuRa de ComPutaDoras”. **Pista**: El código ASCII de la “A” es 65, y el de la “Z” es 90.
d) **Generar string** Escribir un programa que genere un string de la siguiente forma: “a**bb**ccc**dddd**eeeee….”, así hasta la letra “h”. Para ello debe utilizar un loop e ir guardando los códigos ascii en la memoria. El string debe finalizar con el valor ascii 0 para que esté bien formado (debe agregar un elemento más, que valga 0, al final del string).

2. Operaciones con vectores ☆

Definir un vector con 10 valores , y escribir programas para:
a) **Contar positivos.** Contar la cantidad de elementos positivos, y guardar la cantidad en una variable llamada POS.
b) **Calcular máximo.** Calcular el máximo elemento del vector y guardarlo en una variable llamada MAX.
c) **Modificar valores** Modificar los elementos del vector de modo que cada elemento se multiplique por 2.
d) **Impares** Generar un vector con los primeros 10 números impares.
e) **Fibonacci** Generar un vector con los primeros 10 números de la secuencia de fibonacci
f) **Vector de impares a partir de vector** Definir un vector V con 10 números cualesquiera. Escribir un programa que genere un vector W con los números impares de V.

3. Subrutinas de Strings ☆☆

Implemente subrutinas típicas para manipular strings, de modo de tener construir una pequeña librería de código reutilizable.
a) **longitud:** Recibe en $a0 la dirección de un string y retorna su longitud en $v0
b) **contiene:** Recibe en $a0 la dirección de un string y en $a1 un carácter (código ascii) y devuelve en $v0 1 si el string contiene el carácter $a1 y 0 de lo contrario.
c) **es_vocal:** Determina si un carácter es vocal o no, ya sea mayúscula o minúscula. La rutina debe recibir el carácter y debe retornar el valor 1 si es una vocal ó 0 en caso contrario. Para implementar **es_vocal**, utilizar la subrutina **contiene**, de modo que para preguntar si un carácter es una vocal, se pregunte si un string con todas las vocales posibles contiene a este carácter.
d) **cant_vocales** Usando la subrutina escrita en el ejercicio anterior, **cant_vocales** recibe una cadena terminada en cero y devuelve la cantidad de vocales que tiene esa cadena.
e) **comparar:** Recibe como parámetros las direcciones del comienzo de dos cadenas terminadas en cero y retorna la posición en la que las dos cadenas difieren. En caso de que las dos cadenas sean idénticas, debe retornar -1.
f) Usando las subrutinas anteriores, implemente un programa que defina dos strings A y B. Luego, compare los dos strings. Si son iguales, almacenar el valor -1 en la variable RES. De lo contrario, calcule la cantidad de vocales del string más largo de los dos, y almacene el valor también en esa variable. Ejemplo, si A no es igual B, pero A es más largo que B, contar la cantidad de vocales de A.

4. Subrutinas con vectores ☆☆

a) suma Escribir una subrutina que reciba como argumento en $a0 la dirección de una tabla de números enteros de 64 bits, y en $a1 la cantidad de números de la tabla. La subrutina debe devolver en $v0 la suma de los números de la tabla
b) positivos Idem a), pero la subrutina debe contar la cantidad de números positivos. Para ello, implementar y usar otra subrutina llamada **es_positivo**, que reciba un número en $a0 y devuelva en $v0 1 si es positivo y 0 de lo contrario.

5. Subrutinas anidadas ☆☆

a) **Factorial:** Implemente la subrutina **factorial**, que dado un número **N** devuelve **factorial(N) = N! = N * (N-1) * (N-2) * … * 2 * 1.** Por ejemplo, el factorial de 4 es 4! = 4*3*2*1. Recordá también que el factorial de 0 también existe, y es **factorial(0) =0! = 1**
b) **Número combinatorio:** Utilizando **factorial**, implementa la subrutina **comb** que calcula el **número combinatorio** (también llamado **coeficiente binomial**) **comb(m,n) = m! / (n! * (n-m)!)**. Asumir que **n > m**.

6. Subrutina recursiva ☆☆☆

En un ejercicio anterior se implementó la subrutina **factorial** de forma iterativa. Implementar ahora la subrutina pero de forma **recursiva.**

La definición recursiva de **factorial** es:

Caso base: **factorial**(0) = 1

Caso recursivo: **factorial**(n) = n * **factorial**(n-1)

En términos de pseudocódigo, su implementación es:

```
subrutina factorial(n):
if n = 0:
    retornar 1
else:
    m = factorial(n-1)
  retornar n * m
```

a) Implemente la subrutina `factorial` definida en forma recursiva.
Pista 1: El caso base puede codificarse directamente
Pista 2: En el caso recursivo hay una llamada a otra subrutina, con lo cual deberá preservar el registro $ra
Pista 3: En el caso recursivo deberá primero llamar a **factorial** de forma recursiva, pero si el valor **n** original está guardado en un registro temporal, se perderá ese valor.

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