UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

Organización De Computadores

Laboratorio N°1: Instrucciones MIPS y Simulación en MARS

Alumno: Israel Arias Panez.
Sección: L-1.

Santiago - Chile
1-2021

Parte 1: Probar las Predicciones del Pre-Laboratorio en MARS

1. Abre un archivo nuevo en MARS. Escribe el programa 1 del Pre-Laboratorio en el editor y luego guárdalo como “test1.s”. Ensambla el programa, luego ejecútalo. ¿Los valores de los registros son como esperabas? Si no, reinicia y ejecuta el programa paso a paso para ver cómo cambian los valores de los registros.

R: Si, efectivamente los valores de los registros son como esperaba $t0 tiene el valor de 4 y $t1 tiene el valor de 8.

2. Repite el procedimiento anterior para el programa 2 del Pre-Laboratorio. Nombra este programa “test2.s”.

R: Si, efectivamente los valores de los registros son como esperaba, $t1 y $t0 ambos en 0, ya que la etiqueta A no presenta una definición.

3. En el programa 2, ¿qué pasaría si se cambia la línea con beq a: beq $t0, $t0, A ?

R: El programa 2 sigue sin compilar al ejecutar ese cambio de línea, debido a que la etiqueta A sigue sin tener una definición.

4. Repite el procedimiento para el programa 3 (nombre de archivo “test3.s”).

R: Efectivamente los valores de los registros son como esperaba, $t2 y $t1 tienen un valor de 5.

$t0 tiene el valor de 0x10010004.

Y ambas direcciones de memoria 0x10010000 y 0x10010004 tienen el valor de 5.

Parte 2: Escribir programas MIPS

1. Considera el siguiente código tipo Java. Traduce este código en instrucciones MIPS, y

guárdalas en un archivo llamado “program1.s”.

int x = 20; // usar $t0 para almacenar los valores de x

int y = x+5; //usar $t1 para almacenar los valores de y

y = y | 2;

Ejecuta tu código para asegurarte que se comporta correctamente, verificando los valores de

$t1 y $t0 al finalizar el programa.

R: $t1 tiene el valor de 0x0000001b o simplemente 1b, y $t0 tiene el valor de 0x00000014 o simplemente 14 al terminar el programa.

2. Considera el siguiente código tipo Java. Traduce este código en instrucciones MIPS, y

guárdalas en un archivo llamado “program2.s”

```
int x = 1; // usar $t3 para almacenar valores de x
int y = 0; // usar $t4 para almacenar valores de y
if (x == 0) {
y++;
} else if (x == 1) {
y--;
} else {
y = 100;
}
```

Ejecuta tu código para asegurarte que se comporta correctamente para los valores iniciales x=0,

x=1 y x=12.

R:

- Para x = 0: Los registros $t3 y $t4 quedan en cero, lo cual es correcto ya que cae en la condición de restarle 1 al registro $t4, dejándolo en cero.

- Para x = 1: El registro $t3 tiene un valor de 1 y el registro $t4 tiene un valor de 0xffffffff lo que es equivalente al numero -1, por lo que se verifica su buen funcionamiento.

- Para x = 12: El registro $t3 tiene el valor de 0x0000000c que corresponde al numero 12 y el registro $t4 tiene el valor de 0x00000064, que corresponde a 100. Por lo que se verifica el buen funcionamiento del programa.

3. Considera el siguiente código tipo Java. Traduce este código en instrucciones MIPS, y
guárdalas en un archive llamado “program3.s”

```
int[] a = new int[4];
// traduzca solo el código bajo esta línea.
// Asuma que el arreglo comienza en la dirección de memoria 0x10010000
a[0] = 3;
a[3] = a[0] - 1;
```

Ejecuta tu código para asegurarte que se comporta correctamente, asegurándose que las
ubicaciones de memoria guardan los valores esperados.

R:

Efectivamente se verifica que se guardan los valores esperados en las ubicaciones de memoria. Teniendo los valores 0x0000003 en la dirección de memoria 0x10010000 y el valor 0x0000002 en la dirección de memoria 0x10010000(Value +c).

4. Considera el siguiente código tipo Java. Traduce este código en instrucciones MIPS, y

guárdalas en un archive llamado “program4.s”

```
int a = 2; // usar $t6 para almacenar valores de a
int b = 10; // usar $t7 para almacenar valores de b
int m = 0; // usar $t0 para almacenar valores de m
while (a >0) {
m += b;
a -= 1; }
```

Ejecuta tu código para asegurarte que se comporta correctamente para distintos valores

iniciales de a y b (enteros positivos).

R:

En efecto se comporta correctamente, el registro $t0 que corresponde a m tiene un valor de 0x00000014 lo que decimalmente corresponde a 20, lo que confirma que el ciclo se ejecuto dos veces

5. Para integrar todo, considera el siguiente código tipo Java. Traduce este código en

instrucciones MIPS, y guárdalas en un archivo llamado “program5.s”

```
int[] arr = {11, 22, 33, 44};
arrlen = arr.length; // traducción de lo de arriba está dada
// complete la traducción de lo siguiente...
int evensum = 0;          // usar $t0 para valores de evensum
for (int i=0; i<arrlen; i++) {
   if ((arr[i] & 1) == 0) {  // ¿Qué significa esta condición?
     evensum += arr[i];

} }
```

Tu código MIPS debería comenzar con algo así:

```
.data
arr: .word 10 22 15 40
end:
.text
la $s0, arr    # esta instrucción pone la dirección base de arr en $s0
la $s1, end
subu $s1, $s1, $s0
srl $s1, $s1, 2  # ahora $s1 = num elementos en arreglo. ¿Cómo?
```

Todo bajo .data hasta .text es el segmento de datos, donde podemos declarar datos estáticos como arreglos. Todo debajo .text es el segmento de texto o código, donde escribimos las instrucciones del programa. Cuando ni .data ni .text están presentes en el archivo, se asume que el archivo completo contiene un segmento de texto. Asegúrate de probar tu programa con arreglos de distinto contenido y largos.

R:

El programa cumple la función de sumar todos los números pares del arreglo y almacenarlos en la variable llamada **evensum** (Se almacena hexadecimalmente en el registro $t0)**.**

En primer lugar, la condición `if ((arr[i] & 1) == 0)` significa que verifica si el número del arreglo de la posición i es par. Esto debido a que se ejecuta una operación binaria llamada "bitwise AND" entre el numero del arreglo y 1, cuyo resultado siempre dará cero cuando el numero sea par, debido a que, por ejemplo: si el primer numero del arreglo es 7 binariamente es 111, entonces se ejecuta la operación AND entre 111 y 1, la cual se expresa de la siguiente manera:

```
   111
AND 001
   ___
   001
```

Como se puede observar la operación bitwise AND se realiza con números binarios de la misma longitud y se puede lograr multiplicando los números binarios bit por bit, por ende, si hay un cero presente, inmediatamente se escribe un cero y solo se escribe un uno cuando se da el caso de 1*1 como en la operación recién realizada.

Entonces, en el caso contrario si por ejemplo se efectuara la operación AND entre un numero par y 1, por ejemplo, el numero 14 binariamente es representado "1110":

```
   1110
AND 0001
   ____
   0000
```

El resultado es cero, por lo que se confirma que la condición chequea si un numero del arreglo es par o no.

El programa es capaz de obtener el largo del arreglo siguiendo el siguiente procedimiento:

Se obtiene el número de elementos del arreglo usando otro "arreglo" vacío auxiliar para conseguir la dirección de memoria, se realiza una resta entre esta dirección de memoria junto a la dirección de memoria del elemento 0 del arreglo, lo que da como resultado la "posición" de la última dirección de memoria usada (+0, +4, +8, +c, +10, etc). Entonces como se sabe que cada vez que se suma 4 a una dirección de memoria se avanza a una siguiente dirección para almacenar cosas, basta dividir por 4 para conseguir la cantidad exacta de direcciones de memoria usadas.

Por ejemplo, para un arreglo de 4 elementos:

1) "Carga" la primera posición de la dirección de memoria de **arr** al registro $s0, en este caso la dirección de memoria 0x10010000.

2) vuelve a cargar la primera posición, pero del arreglo llamado **end,** cuya primera posición será en la dirección de memoria que continúe inmediatamente a la usada por el arreglo, por ende si el arreglo tiene 4 elementos significa que fueron usadas las direcciones: 0x10010000, 0x10010004, 0x10010008 y 0x1001000c. Por lo que la instrucción la(load address) cargara la dirección de memoria que le sigue a 0x1001000c, o sea 0x10010010, esta se almacena en el registro $s1.

3) Se ejecuta una resta entre la dirección de memoria almacenada en el registro $s1 y la almacenada en $s0, la cual se guarda en $s1. Entonces en el caso presentado de 4 elementos en el arreglo se ejecuta la operación 0x10010010 - 0x10010000, lo que da 0x00000010.

4) Cabe destacar que 0x00000010 es el numero 16 decimal, por lo que se realiza la operación srl $s1, $s1, 2. Lo que significa srl(Shift right logical) con 2 bits lo que se encuentre en el registro $s1, lo que se traduce en dividir por 4 el numero 16, lo que da el total de 4.

Se probo el programa para distintas entradas y distintos valores y se verifico que cumple su funcionalidad para todos los casos testeados.