Universidad Nacional Autónoma de México

Facultad de Ciencias

Organización y Arquitectura de Computadoras

Tarea 05

Arriaga Santana Estela Monserrat
Castañon Maldonado Carlos Emilio
Fernández Blancas Melissa Lizbeth

1 **Menciona tres instrucciones de lenguaje máquina de manipulación de datos.**

Las instrucciones de manipulación de datos son las que establecen el contenido de un registro del procesador o realizan tareas como leer o escribir datos desde y hacia la memoria.
Algunos ejemplos de este tipo de instrucciones son: MOVE (mov), LOAD (como li) y STORE (como sw).

2 **Utilizando instrucciones aritmético-lógicas, diseña un programa que nos diga si un número es par.**

```
.data
dividendo: .word 29
impar: .asciiz "El numero es impar"
par: .asciiz "El numero es par"

.text
.globl main
main:
        li $t0, 2          # Cargamos en $t0 el numero 2
        lw $t1, dividendo  # Cargamos en $t1 el número que es par o impar
        div $t2, $t1, $t0  # Dividimos el numero entre 2
        mul $t3, $t2, $t0  # Multiplicamos el resultado por dos

        # Si el resultado multiplicado por dos no es igual al número, este es impar
        beq $t3, $t1, p

        # Imprimimos que el numero es impar
        li $v0, 4
        la $a0, impar
        syscall

        j fin

p:  # Imprimimos que el numero es par
        li $v0, 4
        la $a0, par
        syscall

fin: # Salimos del programa
        li $v0, 10
        syscall
```

En el código anterior, el dividendo es el número que queremos saber si es par o impar.
La idea principal de este programa es dividir el número dado entre dos y el resultado multiplicarlo por dos. Esto con el fin de comparar este último resultado con el número original. Si ambos son iguales, entonces el número es par y si no, es impar. Para todo esto utilizamos las operaciones aritmético-lógicas como div, mul y la comparación.

3 **Menciona los distintos tipos de instrucciones de flujo de control y menciona un ejemplo de cuándo se debería usar cada una.**

Las instrucciones de flujo de control son:

- Saltos incondicionales: Provoca un cambio incondicional en la secuencia de ejecución hacia una ubicación específica (generalmente una etiqueta). Un ejemplo es la instrucción JUMP (j).
- Saltos condicionales: Provoca un cambio en la secuencia de ejecución hacia una ubicación (normalmente una etiqueta) en específico si se cumple una condición. Un ejemplo es la instrucción Bench if equal (beq), la cual compara si el valor de dos registros es igual y si lo es, salta a una etiqueta especificada.
- Llamadas a subrutinas: Invoca a una subrutina para realizar una tarea en específico. Ésta debe devolver el control al programa que la llama, para lo cual se almacena la dirección de retorno en la pila y se pasa el control a la subrutina. Un ejemplo de este tipo de instrucciones es CALL (jal).
- Retorno: Se usan para regresar de una subrutina a la rutina que la llamó. Un ejemplo de esta es la instrucción RET(jr).

4 **Modela una solución con instrucciones de lenguaje máquina que maneja punto flotante y cómputo vectorial, que de la solución a laa suma de dos vectores** $A = (a_0, a_1)$ y $B = (b_0, b_1)$.

```
.data
vectorA: .float 3.14, 3.15
vectorB: .float 2.53, 7.66
suma: .space 8

.text
.globl main
main:
        # Cargamos los vectores A y B en $t0 y $t1 respectivamente
        la $t0, vectorA
        la $t1, vectorB

        # Cargamos las primeras entradas de los vectores en $f0 y $f1
        lwc1 $f0, ($t0)
        lwc1 $f1, ($t1)

        # Sumamos las primeras entradas de los vectores en $f2
        add.s $f2, $f0, $f1

        # Cargamos las segundas entradas de los vectores $f0 y $f1
        lwc1 $f0, 4($t0)
        lwc1 $f1, 4($t1)

        # Sumamos las segundas entradas de los vectores en $f3
        add.s $f3, $f0, $f1

        # Guardamos el resultado en la memoria para la cual reservamos espacio
        swc1 $f2, suma
        swc1 $f3, suma + 4

        # Salimos del programa
        li $v0,10
        syscall
```

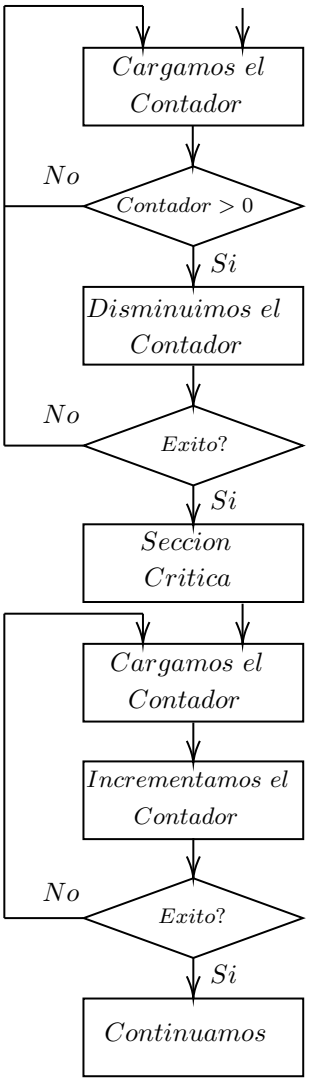
En el código anterior, tenemos dos vectores en memoria, los cuales cargamos en los registros del coprocesador 1 (pues debemos trabajar en punto flotante) y realizamos la suma de éstos, sabiendo que la suma de vectores se realiza entrada por entrada.

5 **En instrucciones de MIPS II Load Linked (LL) y Store Conditional (SC), brindan soporte de sincronización para los procesadores R4000**. Las características más importantes de las primitivas LL y SC son:

- Proporciona un mecanismo para generar todas las primitivas de sincronización comunes, incluidos test-and-set, contadores, secuenciadores, etc. sin sobrecarga adicional. Además, operan de manera que el tráfico de bus se genera sólo cuando cambia el estado de la línea de caché; las palabras bloqueadas permanecen en la caché hasta que otro procesador se hace cargo de esa línea de caché.

Implementa un Contador usando LL y SC.
Load linked (LL) y Store Conditional (SC) son instrucciones en MIPS para poder actualizar un valor de memoria en un sistema de memoria compartida con múltiples procesadores. La idea de esto es utilizar LL para cargar en un registro un valor almacenado en una dirección de memoria, trabajar con éste y después escribir el valor modificado en la misma dirección de memoria usando SC, la cual sólo escribe el nuevo valor si ningún otro procesador ha escrito en la dirección de memoria mientras se trabajaba con el registro.
Es decir, LL carga un valor que está guardado en memoria y monitorea esa dirección de memoria para saber si otro procesador escribe en ésta, mientras que SC almacena el valor en la dirección de memoria que está siendo monitoreada sólo si ningún otro procesador ha escrito en ésta y si no han ocurrido excepciones.
Un contador es una técnica de sincronización para permitir que N procesadores tengan permiso de acceder a la sección crítica (sección de datos o código compartido), usand una variable para contar el número de recursos limitados o el número de permisos disponibles. Cada procesador revisa si un recurso está disponible para usarse. Si está disponible, trata de modificar el contador y obtener permiso para trabajar con la sección crítica.
Sabiendo esto, un contador usando LL y SC se implementa de la siguiente manera:

```
Loop1:  LL r2,(r1)

        BLEZ r2,Loop1
        NOP

        SUB r3,r2,1
        SC r3,(r1)

        BEQ r3,0,Loop1
        NOP

Loop2:  LL r2,(r1)

        ADD r3,r2,1
        SC  r3,(r1)

        BEQ r3,0,Loop2
        NOP
```



6 **La siguiente función $D = (A+B)*C$ al momento de crear el código, este puede variar entre distintas arquitecturas. Visto en el libro tenemos arquitectura tipo pila, arquitectura de acumulador y arquitectura de registros de propósito general (reg-mem).**
El siguiente código *se implementó para una* **Arquitectura de Registros.**

```
Load R1, A
Add R3, R1, B
Store R3, E
Load R2, C
Mul R4, R2, E
Store R4,D
```

Da el código que modele la función anterior para arquitecturas de pila, acumulador y de registro (load-store).

Arquitectura de Acumulador

```
Load A
Add B
Store E
Load C
Mul E
Store D
```

Arquitectura de Pila

```
Push A
Push B
Add
Push C
Mul
Pop D
```

Arquitectura de Registros load-store

```
Load R1, A
Load R2, B
Load R3, C
Add R4, R1, R2
Mul R5, R4, R3
Store R5, D
```

7 **Dado el siguiente valor `0x4cbad39f` muestra cómo se vería almacenado en un esquema little endian y big endian.**
Notemos que `0x4cbad39f` es un numero *hexadecimal*, es por esto que cada dígito de este valor hexadecimal tiene una representación binaria de 4 bits.
➢ `4`: 0100 ➢ `c`: 1100 ➢ `b`: 1011 ➢ `a`: 1010 ➢ `d`: 1101 ➢ `3`: 0011 ➢ `9`: 1001 ➢ `f`: 1111

Por lo que la representracion en binario de nuestro numero hexadecimal sería:

0100 1100 1011 1010 1101 0011 1001 1111

Ahora, notemos que este conjunto de bytes esta representada por defecto en un esquema, en este caso es el de **big endian** ya que el byte más significativo se almacena primero y el byte menos significativo se almacena último, esto lo podremos ver mas claramente si vemos al valor anterior de la siguiente forma:

01001100 10111010 11010011 10011111

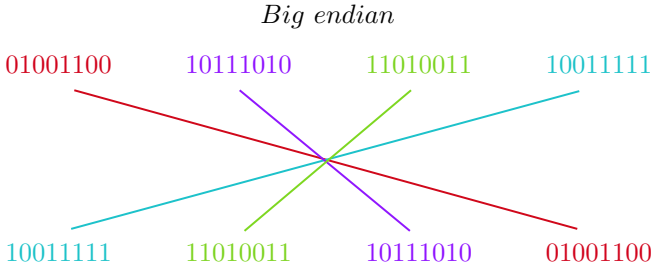
Que en hexadecimal es: `0x4cbad39f`.

Ahora solo nos falta su representación en **little endian**, en lo cual no tendremos complicaciones ya que solo tenemos que "voltear " al valor anterior para obtener esta representación, ya que si el anterior guardaba el byte más significativo primero y el byte menos significativo al ultimo, little endian debe guardar el byte menos significativo primero y el byte más significativo al ultimo.

10011111 11010011 10111010 01001100

Que en hexadecimal es: `0x9fd3ba4c`.

Una forma muy útil y visual de ver que acabamos de hacer es la siguiente:



8 **El siguiente código implementado en MIPS se utilizan instrucciones de flujo de control. ¿Cuáles son y por qué se utilizan? ¿Qué está intentando implementar el código siguiente?**

```
.text
div $t0, $t0, $t0
li  $t0, 0x7FFFFFFF
addi $t0, $t0, 1

.kdata
error : .asciiz "Ocurrio un error"
.ktext 0X80000180

mfc0 $k0, $13
andi $k0, $k0, 0x7C
beq  $k0, 0x30, over
li   $v0, 10
syscall
over :
la $a0, error
li $v0, 4
syscall
```

Notemos que las instrucciones de flujo de control y su utilidad serán las siguientes:

- **beq:** Se utiliza para comparar si el valor del registro $\$k0$ es igual a $0x30$, para entonces saltar a la etiqueta *over* si la comparación es verdadera.
- **syscall:** Aunque no se consideran estrictamente como instrucciones de flujo de control, establecen una interrupción en la ejecución del programa para que el sistema operativo ejecute una tarea específica, por ejemplo, salir del programa.

Ahora, el código proporcionado quiere darnos la impresión de que usaremos a `0X80000000` (el cual es `0x7FFFFFFF + 1`) o que usaremos a `0X80000180` para posteriormente aplicarle a alguno la operación *andi* con `0x7C` (recordemos que esto es la abreviatura de `0x0000007C`) y después verificar si el resultado concuerda con `0x30` para que salte el error.

Sin embargo, como el registro $\$t0$ tiene al valor 0, al realizar la instrucción *div* $\$t0, \$t0, \$t0$ se genera a la par el valor `0X00000024` en el registro \$13 del `Coproc 0`, mismo al que vamos a acceder después para guardar este valor en el registro $\$k0$ mediante la instrucción $mfc0$ $\$k0, \$13$.

Como podemos darnos cuenta, entonces el valor con el que se aplicara *andi* será `0X00000024` con `0x7C`, a lo que tendremos:

AND

$1 \ \& \ 1 = 1$

$1 \ \& \ 0 = 0$

$0 \ \& \ 1 = 0$

$0 \ \& \ 0 = 0$

$$\begin{array}{rl} & 00000000\ 00000000\ 00000000\ 00100100 = 0x24 \\ \& & 00000000\ 00000000\ 00000000\ 01111100 = 0x7C \\ \hline & 00000000\ 00000000\ 00000000\ 00100100 = 0x24 \end{array}$$

Como podemos observar, este resultado es diferente al de `0X30` (y de hecho es igual al de `0X24` por las propiedades de AND), es por esto que el programa termina sin que salte la etiqueta de `error`.

9 **A partir de código del ejercicio anterior implementado en Mips ¿Cuál es el correspondiente código en lenguaje C?**

```c
#include <stdio.h>
int main() {
    // El registro $t0
    int t0 = 0;
    t0 = t0 / t0;
    // Debido a magia negra de MARS
    // Genera esto derivado de lo anterior
    // Y lo guarda en el Coproc0 $13
    int coproc13;
    coproc13 = 0x00000024;

    t0 = 0x7FFFFFFF;
    t0 = t0 + 1;

    char* error = "Ocurrio un error";
    int ktext = 0X80000180;

    // El registro $k0
    int k0;
    k0 = coproc13;
    k0 = k0 & 0x7C;

    if (k0 == 0x30) {
        printf("%s\n", error);
    } else {
        return 0;
    }
}
```

10 **¿Qué tipo de datos inmediatos puede manejar la arquitectura DLX? ¿Cuáles expresiones simbólicas pueden ser usadas para especificar direcciones de memoria.**

La arquitectura DLX puede manejar los siguientes tipos de datos inmediatos:

- Enteros de 16 bits con signo
- Enteros de 16 bits sin signo
- Números de punto flotante de precisión simple
- Números de punto flotante de precisión doble
- Bytes
- Cadenas de caracteres que no terminan con el byte null
- Cadenas de caracteres que terminan con el byte null
- Palabras (son de 32 bits)

La forma más simple de especificar direcciones de memoria es introduciendo un número, que se interpreta como la dirección de memoria. Sin embargo, las expresiones de direcciones de memoria pueden formarse por números, símbolos definidos en el fichero ensamblador como $*, /, +, -, \%, <<, >>, \&\&, |, \wedge$.
Además se usan paréntesis para agrupar.

11 **¿Para qué sirven las siguientes instrucciones movfp2i, 1f, ld, sf, sd, bnez y jalr? ¿En qué tipo de instrucciones de DLX se encontraría cada una?**

- movfp21: Se usa para mover un valor de un registro de punto flotante a un registro entero. Con esta instrucción puede haber pérdida de información debido a que en punto flotante hay números de valores decimales y en enteros no. Esta instrucción es una instrucción de transferencia de datos.
- lf: Se usa para cargar un número de punto flotante de simple precisión en un registro de punto flotante. Esta instrucción es una instrucción de transferencia de datos.
- ld: Se usa para cargar un número en punto flotante de doble precisión en un registro. Esta instrucción es de transferencia de datos.
- sf: Se usa para almacenar en memoria un número en punto flotante de simple precisión. Esta es una instrucción de transferencia de datos.
- sd: Se usa para almacenar en memoria un número en punto flotante de doble precisión. Esta es una instrucción de transferencia de datos.
- bnez: Se hace una bifurcación si el valor que está en el registro es distinto de cero (desplazamiento de 16 bits desde PC+4). Esta es una instrucción de control de flujo.
- jalr: Salta y enlaza. Guarda el PC+4 en R31 y el destino se proporciona en un registro. Esta es una instrucción de flujo de control.

12 **El ensamblador soporta numerosas directivas, las cuales afectan a la forma en que se carga la memoria, el código ensamblado. Estas directivas deben ser introducidas en el lugar en el que normalmente se introducen las instrucciones y sus argumentos.**
Da 5 directivas aceptadas por el ensamblador del DLX y explica en qué situación se usa cada una.

- `.data`: Esta directiva se utiliza para declarar y reservar espacio en la sección de datos para almacenar variables y constantes. Las variables declaradas con esta directiva se asignan en memoria durante la fase de enlace. Por ejemplo:

```
.data
var1: .word 10    ; Declara una variable llamada var1 con el valor 10
const1: .word 5   ; Declara una constante llamada const1 con el valor 5
```

- `.text`: Esta directiva se utiliza para indicar el comienzo de la sección de código del programa. Las instrucciones de programa se colocan después de esta directiva. Por ejemplo:

```
.text
main:
add R1, R2, R3   ; Instrucción de suma
sub R4, R1, R5   ; Instrucción de resta
```

- `.align`: Esta directiva se utiliza para alinear el siguiente dato o instrucción en una dirección de memoria específica. Es útil para garantizar un acceso eficiente a los datos y mejorar el rendimiento. Por ejemplo:

```
.data
.align 4
array: .word 1, 2, 3, 4    ; Crea un arreglo de 4 palabras alineado
                            ; en una dirección múltiplo de 4
```

- `.asciiz`: Esta directiva se utiliza para declarar una cadena de caracteres en la sección de datos. La cadena se almacena como una secuencia de caracteres ASCII seguidos de un byte nulo (cero) para indicar el final de la cadena. Por ejemplo:

```
.data
message: .asciiz "Hola, mundo!"   ; Declara una cadena de caracteres
                                  ; con la cadena "Hola, mundo!"
```

- `.globl`: Esta directiva se utiliza para declarar que una etiqueta (como una función) es visible y accesible desde otros módulos o archivos enlazados. Permite la creación de símbolos globales en el programa. Por ejemplo:

```
.text
.globl main   ; Declara que la etiqueta "main" es global y
              ; puede ser accedida desde otros archivos
 main:
```

REFERENCIAS

1. Galaviz, J. (2015). El conjunto de instrucciones. Capitulo 5. Recuperado 8 de mayo de 2023, de https://drive.google.com/file/d/1Vtrq7iCw7nquy5NGsWwTGwuTFOxkPImG/view
2. GeeksforGeeks. (2021). Data Manipulation Instructions in Computer Organization. GeeksforGeeks. https://www.geeksforgeeks.org/data-manipulation-instructions-in-computer-organization/
3. GeeksforGeeks. (2021b). Types of Program Control Instructions. GeeksforGeeks. https://www.geeksforgeeks.org/types-of-program-control-instructions/
4. https://www.ibm.com/docs/en/aix/7.2?topic=adapters-ascii-decimal-hexadecimal-octal-binary-conversion-table
5. Llombart, V. (s.f.). La arquitectura DLX. Arquitectura de computadoras. Recuperado 8 de mayo de 2023, de https://www.uv.es/varnau/ManualDLX.pdf
6. (1993, abril). MIPS R4000 Synchronization Primitives. mips.com. Recuperado 14 de mayo de 2023, de https://www.cs.auckland.ac.nz/courses/compsci313s2c/resources/MIPSLLSC.pdf
7. In MIPS, what are load linked and store conditional instructions? (s.f.). Quora. https://www.quora.com/In-MIPS-what-are-load-linked-and-store-conditional-instructions
8. Comprender las instrucciones ll (carga vinculada) y sc (almacenamiento condicional) en el conjunto de instrucciones MIPS. (2017). CSDN. Recuperado 14 de mayo de 2023, de https://blog.csdn.net/gzxhjwddf/article/details/70098521