Sistemas Digitales

Alumnos de Sistemas Digitales
Facultad de Ciencias Exactas y Naturales
UBA

Choose your destiny:

(dobleclick en los ejercicio para saltar)

- Notas teóricas
- Ejercicios de la guía:

El repo en github para descargar las guías con los últimos updates.

https://github.com/nad-garraz/sistemasDigitales

La Guía 3 se actualizó por última vez: 03/12/2024 @ 18:02

Guía 3

https://github.com/nad-garraz/sistemasDigitales/blob/main/3-guia/3-sol.pdf

Si querés mandar un ejercicio o avisar de algún error, lo más fácil es por Telegram.

https://t.me/joinchat/DS9ZukGbZgIOIaHgdBlavQ

Notas teóricas:

Ejercicios de la guía:

Ejercicio 1 ¿Qué es una arquitectura? ¿Qué componentes la conforman? ¿Contiene información del funcionamiento interno de las operaciones?

Una arquitectura es lo que el programador conoce sobre la computadora. Los componentes que la conforman son:

- el conjunto de instrucciones - el lenguaje -;
- el conjunto de registros;
- la forma de acceder a memoria.

La arquitectura **no** contiene información del funcionamiento interno de las operaciones.

Ejercicio 2

1. ¿A cuántos bytes se direcciona la memoria en la arquitectura RISC-V? ¿Cuántos bytes hay en una palabra?
2. Sabiendo que la memoria se encuentra en el estado que se ve a continuación, indicar el resultado de las siguiente operaciones sabiendo que t0 = 0xAD.

Dirección	...	0xAA	0xAB	0xAC	0xAD	0xAE	0xAF	0xB0	0xB1	0xB2	...
Valor	...	0x34	0x11	0xF4	0x09	0x12	0x73	0x20	0x24	0xFF	...

(a) **lw** t1, 0(t0) (c) **lw** t1, -3(t0) (e) **lhu** t1, -1(t0) (g) **lbu** t1, 5(t0)
(b) **lw** t1, 2(t0) (d) **lh** t1, -1(t0) (f) **lb** t1, 5(t0)

1. La memoria en la arquitectura de RISC-V se direcciona a 32 bits, es decir, a unos $32 \div 8 = 4$ bytes
2. (a) t1 = 0x20731209
   (b) t1 = 0xFF242073
   (c) t1 = 0x09F41134
   (d) t1 = 0x000009F4
   (e) t1 = 0x000009F4
   (f) t1 = 0xFFFFFFFF
   (g) t1 = 0x000000FF

Ejercicio 3 Dado el siguiente arreglo de enteros de 16 bits en lenguaje Java:

```
1 int [] arreglo16b = { -1 , 170 , 255 , -255 , 0 , 32 , 10000 , 0};
```

Sabiendo que este arreglo se guarda en memoria empezando en la direccion 0xCC.

1. Dibujar el estado de la memoria
2. Si t0 = 0xCC, escribir un programa que dado un index i, devuelve **arreglo16b**[i]

1.

Dirección	...	0xCC	0xCD	0xCE	0xCF	0xD0	0xD1	0xD2	0xD3
Valor	...	0xFF	0xFF	0xAA	0x00	0xFF	0x00	0x01	0xFF

Dirección	0xD4	0xD5	0xD6	0xD7	0xD8	0xD9	0xDA	0xDB	...
Valor	0x00	0x00	0x30	0x00	0x10	0x27	0x00	0x00	...

2.

```
obtenerIndex:
# Recibe en a0 la referencia de la posicion
# Recibe en a1 la posicion a obtenerIndex
# Devuelve en a0 el valor que tenia en esa posicion el array
add a0, a0, a1
lw a0, 0(a0)
```

Ejercicio 4 Dado los siguientes dos programas y suponiendo que ambos empiezan en la direccion 0x00.

a) Escribir en que direccion se encuentra cada etiqueta

b) ¿Como se maneja el branching en RISC-V? ¿Y los saltos incondicionales? ¿Es afectado por la direccion de memoria donde comienza el programa? Para cada instruccion de salto, escribir el offset que se aplicara al PC

```
Inicio: addi a0, zero, 10
        addi a1, zero, 50
Ciclo: ble a1, zero, Fin
       sub a1, a1, a0
j Ciclo
Fin: beq a1, zero, Inicio
```

```
Inicio: li a1, 0xffffffff
        li a2, 0x1
Vuelta: beq a1, a2, Inicio
        sub a2, a2, a1
nop
j Vuelta
```

---

a) Empecemos por el código de la izquierda

Código

```
Inicio:  addi a0, zero, 10
         addi a1, zero, 50
Ciclo:   ble a1, zero, Fin
         sub a1, a1, a0
j Ciclo
Fin: beq a1, zero, Inicio
```

Posición en memoria

```
# 0x00
# 0x04
# 0x08
# 0x0C
# 0x10
# 0x14
```

Ahora vamos con el código de la derecha

Código

```
Inicio: li a1, 0xffffffff
        li a2, 0x1
```

Posición en memoria

```
# 0x00
# 0x04
```

```
Vuelta: beq a1, a2, Inicio
        sub a2, a2, a1
nop
j Vuelta
```

```
# 0x08
# 0x0C
# 0x10
# 0x14
```

Ejercicio 5 Dado el siguiente programa en lenguaje C.

```
int x = 2;
int y = 32;
x = x + y ;
```

a) Traducir a lenguaje esamblador de RISC-V. Usar los registros t0 y t1 inicializados con números de 8 bits para representar a las variables x e y respectivamente.

b) Escribir un programa que guarde en t2 un numero de 32 bits divido en sus 12 bits más significativos en t0 y el resto de 20 bits en **t1**.

c) ¿Cómo maneja RISC-V la extensión de signo en los inmediatos de 12 bits? ¿Qué resultado generaría la instrucción **andi** a0, a0, -2048 cuando a0 vale 0xFFFFFFFF? Re-escribir el código del inciso **a)** para números de 32 bits sin utilizar la instrucción **li**.

---

a)

```
li t0, 2
li t1, 32
add t0, t0, t1
```

b) 😳... hay que hacerlo! 😳

Si querés mandarlo: Telegram → , o mejor aún si querés subirlo en LaTeX→ .

c) RISC V maneja la extensión de los signos en los inmediatos, completando con el bit más significativos del inmediato. La instruccion **andi** a0, a0, -2048 si a0 vale 0xFFFFFFFF sería el valor de -2048 en 32 bits, que en hexadecimal es: $\neg$ 0x0000 0800 +1 = 0xFFFF F000

(a)

```
addi t0, zero, 2
addi t1, zero, 32
add t0, t0, t1
```

Ejercicio 6

1. ¿Cuántos bytes ocupa cada instrucción de RISC-V? ¿Cuál es la diferencia entre una instrucción y una pseudoinstrucción?

2. ¿Qué clases de instrucciones tiene la arquitectura RISC-V? ¿Qué tipo de instrucciones contiene cada clase? ¿Qué diferencia hay entre instrucciones de Registros(**R**) y de Inmediatos (**I**)?

3. Ensamblar el siguiente codigo escrito en lenguaje RISC-V

```
    addi a6, x0, 10
    add a0, a1, a6
    bltz x1, 0x0ABC
```

4. Desensamblar el siguiente programa escrito en lenguaje de maquina RISC-V.

0111 1111 1111 0000 0000 0101 0001 0011
0101 0101 0101 0000 0000 0101 1001 0011
0000 0000 1010 0101 1100 0110 0011 0011
1111 1110 0000 0110 0000 1010 1110 0011
0000 0000 0000 0000 0000 0000 0001 0011

---

1. Cada instrucción de RISC V ocupa 4 bytes.
   La diferencia entre una instruccion y una pseudoinstruccion es que una pseudoinstruccion puede ser una o varias instrucciones, y al compilarlo a lenguaje máquina es convertido, facilitando la programación, mientras que las instrucciones es lo que el procesador ejecuta en sí.

2. 😳... hay que hacerlo! 😭

   Si querés mandarlo: Telegram → , o mejor aún si querés subirlo en LaTeX→ .

```
    addi a6, x0, 10
    add a0, a1, a6
    bltz x1, 0x0ABC
```

0000 0000 1010 0000 0000 1000 0001 0011
0000 0001 0000 0101 1000 0101 0011 0011
0010 1010 0000 0000 1100 1110 1110 0011

3. 😳... hay que hacerlo! 😭

   Si querés mandarlo: Telegram → , o mejor aún si querés subirlo en LaTeX→ .

Ejercicio 7 😳... hay que hacerlo! 😭

Si querés mandarlo: Telegram → , o mejor aún si querés subirlo en LaTeX→ .

Ejercicio 8 ¿Qué es `.text` y `.data`? ¿Qué tipo de información se guarda en cada una de ellas?

Pongo un poco de contexto, para que no quede con sabor a nada la respuesta:

En el camino desde escribir el programa en *alto nivel* (C, python, Java,etc.) hasta llegar al ejecutable, se pasa por el *ensamblador*.

Las palabras `.text`, `.data`, `.word` entre otras son algunas de las `directivas del ensamblador` (*assembler directives*).

Cuando el programa se carga en la memoria (para luego leerse) se genera un *mapa de la memoria* el cual tiene una estructura particular dividida en *segmentos*, donde se cargan cosas en cierto orden.

Entre esos segmentos están el `Text Segment` y el `Global Data Segment`, segmentos a los que se accede con el `PC` (program counter) y el `GP` (global pointer) respectivamente.

`The Text Segment`: En esta parte de la memoria se guarda el código que escribimos en las instrucciones, *la sustancia lógica que programamos en el Ripes o emulador favorito.*
`The Global Data Segment`: En esta parte de la memoria se guardan variables globales. Éstas se pueden accedere desde cualquier parte del código, desde adentro de alguna función por ejemplo. Esto sucede antes de que comience el programa.

Entonces cuando se usan las palabras `.text` y `.data`, lo hacemos para decirle al ensamblador en cual segmento de la memoria meter el código que estamos escribiendo en el Ripes o emulador en cuestión.

`Culpables de que esto haya sucedido:`
Nad Garraz

Ejercicio 9 Se cuenta con cuatro datos sin signo de 1-byte cada uno almacenados en el registro `t0` y queremos sumar el valor de los cuatro datos. Escribir un programa en lenguaje ensamblador RISC-V que realice esta operación y almacene el resultado en el registro `t0`.
Ejemplo:

t0	0x90	0x1A	0x00	0x02

Con este dato el registro debería valer `0x000000AC`.

```
# Se usa la directiva de ensamblador .data
# se carga la palabra antes de comenzar
# el programa

.data
    palabra: .byte 0x90
             .byte 0x1a
             .byte 0x00
             .byte 0x02

# Ahora se usa la directiva donde se usaran variable
# locales y ademas desde donde se podra acceder a
# la informacion del segmento de memoria .data

.text
main:
    la s0 palabra    # s0 <- direcci n del primer elemento de palabra
    addi sp sp -4
    sw s0 0(sp)

    li a2 4 # contador inicializado en 4
    li t0 0 # acumulo la suma en t0

```

```
24      for:
25          lbu t1 0(s0) # t1 <-- valor del byte
26          addi s0 s0 1 # Siguiente byte
27
28          add t0 t0 t1 # suma
29
30          addi a2 a2 -1 # actualizo el contador
31          beqz a2 done
32      j for
33
34  done:
35      lw s0 4(sp)
36      addi sp sp 4
37      j fin
38
39  fin: j fin
```

Ejercicio 10 😳... hay que hacerlo! 😳

Si querés mandarlo: Telegram → , o mejor aún si querés subirlo en LaTeX→ .

Ejercicio 11 😳... hay que hacerlo! 😳

Si querés mandarlo: Telegram → , o mejor aún si querés subirlo en LaTeX→ .

Ejercicio 12 😳... hay que hacerlo! 😳

Si querés mandarlo: Telegram → , o mejor aún si querés subirlo en LaTeX→ .

Ejercicio 13 😳... hay que hacerlo! 😳

Si querés mandarlo: Telegram → , o mejor aún si querés subirlo en LaTeX→ .

Ejercicio 14 😳... hay que hacerlo! 😳

Si querés mandarlo: Telegram → , o mejor aún si querés subirlo en LaTeX→ .

Ejercicio 15 😳... hay que hacerlo! 😳

Si querés mandarlo: Telegram → , o mejor aún si querés subirlo en LaTeX→ .

Ejercicio 16 😳... hay que hacerlo! 😳

Si querés mandarlo: Telegram → , o mejor aún si querés subirlo en LaTeX→ .

Ejercicio 17 ... hay que hacerlo!
Si querés mandarlo: Telegram →, o mejor aún si querés subirlo en LaTeX→.

Ejercicio 18 ... hay que hacerlo!
Si querés mandarlo: Telegram →, o mejor aún si querés subirlo en LaTeX→.

Ejercicio 19 ... hay que hacerlo!
Si querés mandarlo: Telegram →, o mejor aún si querés subirlo en LaTeX→.

Ejercicio 20 ... hay que hacerlo!
Si querés mandarlo: Telegram →, o mejor aún si querés subirlo en LaTeX→.

Ejercicio 21 ... hay que hacerlo!
Si querés mandarlo: Telegram →, o mejor aún si querés subirlo en LaTeX→.

Ejercicio 22 ... hay que hacerlo!
Si querés mandarlo: Telegram →, o mejor aún si querés subirlo en LaTeX→.

Ejercicio 23 ... hay que hacerlo!
Si querés mandarlo: Telegram →, o mejor aún si querés subirlo en LaTeX→.