

Nro. ord.	Apellido y nombre	no	#hojas
86	Gurbano Manuel	M	4

SISTEMAS DIGITALES - Segundo Parcial
Segundo Cuatrimestre 2024

EJ. 1	EJ. 2	EJ. 3	EJ. 4	Nota
1	2	2	1	6

Correctorx:

Agustín

Aclaraciones

- Anote apellido, nombre, LU y numere *todas* las hojas entregadas, entregando los distintos ejercicios en hojas separadas.
- El parcial no es a libro abierto pero pueden consultar la hoja de referencia provista por la cátedra.
- Justifique sus respuestas explicando lo que considere necesario en lenguaje natural.
- El parcial se aprueba con 6 y se deben tener ambos parciales aprobados para aprobar la materia (promoción directa).

Ejercicio 1 (2 pts.) Implemente la función que calcula el valor de un elemento en un triángulo de Pascal, definido en el siguiente bloque de código C.

```
function pascal(fila , columna) {
    if (columna>fila) return 0;
    if (columna<=1 || fila <=1) return 1;
    return pascal(fila -1, columna) + pascal(fila -1, columna-1);
}
```

El triángulo de Pascal define una secuencia triangular de números enteros que comienza con un 1 (uno) en el vértice superior y se expande hacia abajo con números calculados a partir de los números de la fila superior. Cada número en el triángulo es la suma de los dos números directamente arriba de él.

Más allá de la interpretación de la función les recomendamos concentrarse en la traducción a código RISC V que respete la convención de llamada. Deben explicar en qué registros se almacenan los valores en cada paso y cómo se aseguran que se respeta la convención.

Ejercicio 2 (2 pts.) Implemente las siguientes funciones en lenguaje ensamblador de RISC V respetando la convención de llamada presentada en la materia. Describir el comportamiento y cómo se aseguran que se respete la convención.

- int inv(int x) = -x (inverso aditivo)
- void invertirArreglo(int arr[], int largo): Dado un puntero a un arreglo de enteros de 32 bits y la cantidad de elementos, cambia cada valor del arreglo por su inverso aditivo.

Ejercicio 3 (4 pts.) Se tiene una estructura BalanceDeudor que contiene el ID del cliente como un entero sin signo de 8 bits, la suma de sus consumos como un entero en complemento a dos de 16 bits y la suma de sus pagos realizados como un entero en complemento a dos de 16 bits. Ubicación de los datos de una estructura BalanceDeudor:

Byte	0x0000	0x0001	0x0003
Nombre	ID	Consumos	Pagos

En memoria se encuentra un arreglo balanceDeudores del tipo BalanceDeudor con la forma:

Direccion	0x0000	0x0001	0x0003	0x0005	...	0x0030	0x0031	0x0033	0x0035
Valor	17	30020	1232	6	...	9	5878	300	0

Donde el final del arreglo es demarcado por un ID nulo. Se pide

- Calcular cuántos bytes ocupa en memoria la estructura BalanceDeudor y cuántos bytes un arreglo de tipo BalanceDeudor de doce elementos.
- Escribir una función contarDeudores(balanceDeudores) que dada una posición de memoria que indica el comienzo de un arreglo balanceDeudores de BalanceDeudor, devuelva la suma estructuras donde la suma de los consumos (segundo elemento de la estructura) es mayor que la suma de los pagos realizados (tercer elemento de la estructura). Ejemplo:

```
.data
balanceDeudores: .byte 17
                  .half 30020
                  .half 1232
                  .byte 6
                  .half 200
                  .half 200
                  .byte 9
                  .half 5878
                  .half 300
                  .half 0      #Declaramos el final del arreglo

.text
contarDeudores:
...
```

Para este caso `balanceDeudores` debe devolver 2 ya que el usuario con id 17 y el usuario con id 9 tienen consumos mayores a sus pagos. Recuerden que el arreglo es pasado como dirección de memoria de su primer elemento a través del primer parámetro de la función.

Ejercicio 4 (2 pts.) Para una microarquitectura de ciclo simple para un procesador de RISC V, como la que vimos en clase, explicar que componentes y señales de control están involucrados y cómo modifican el estado del procesador al ejecutar la instrucción `or x4, x5, x6`.

Hanbel Gorbánov - Sistemas Digitales - Segundo Parcial

③ Tenemos la estructura BalanceDeudor de la forma

Byte	0x0000	0x0001	0x0010
Nombre	ID	Consumos	Pagos.
	8 bits	16 bits	16 bits

A) La estructura BalanceDeudor ocupa 40 bits en memoria, ya que som 8 del ID, 16 de los consumos y 16 de los pagos.

$$8 \text{ bits} + 16 \text{ bits} + 16 \text{ bits} = 40 \text{ bits} = 5 \text{ bytes}$$

Réspuesta:

Por otro lado, si tenemos un arreglo de tipo BalanceDeudor, no solo deberemos tener en cuenta el tamaño de los datos de cada elemento sino también el ID nulo que lo pone firm.

Luego CANTIDAD de bytes = Num elementos . 40 + 8

→ aunque sea nulo,
si que siendo una
ID por ende ocupa
8 bytes = 1 byte ✓

$$\Rightarrow \text{Cant. Bytes} = 12 \cdot 5 + 1 = 61 \text{ bytes.}$$

⇒ Un arreglo de tipo BalanceDeudor de 12 elementos ocupa 61 bytes.

Item B en otra clase.

⑧ Programar contar Deudores (balance Deudor).

→

data
balanceDeudores byte 17
half 30020
~~balanceDeudores~~
half 1232
byte 6
half 200
half 200
byte 9
half 3873
half 300
byte 0

.text

compararDeudores:

ld \$0, balanceDeudores #Cargo en \$0 (primer parámetro) dirección de comienzo
li \$t0, 0 #Contador inicializado en cero
li \$t1, 0 #índice de esta iteración (lo inicializo en cero) { los iniciales y
los finales y
el tamaño }
li \$d1, 0 #Consumo de esta iteración
li \$d2, 0 #Pagos de esta iteración
li \$d3, 0 #ID de esta iteración
li \$t2, 0 #desplazamiento

Ciclo:

X HUL \$t2, \$t2, 24*

ld \$d3, \$t2(\$d3) #Cargo ID Actual en \$d3 (desplazamiento \$t2 es la posición de inicio)

lbu beqz \$d3, Fim_programa #Si el ID es cero, salto al final

lh \$d1, 2(\$d3) #Cargo consumo de la persona actual { los desplazamientos son
lh \$d2, 4(\$d3) #Cargo monto pago de la persona actual } 1 y 3 bytes.

blt \$d1, \$d2, mo_suma #uso blt para comparar pagos y consumos

addi \$t0, \$t0, 1 #Solo se ejecuta si suma, por eso aumenta el contador

X addi \$t2, \$t2, 1 #Incremento el índice de iteración

j Ciclo #Vuelvo al ciclo

mo_suma:

addi \$t2, \$t2, 1 #Incremento el índice
j Ciclo

Fim_programma:

mv \$0, \$t0 #Aumento el contador
ret

#Funcionamiento: El ciclo carga en \$d3 el ID Actual, calculando el desplazamiento como base + tamaño. i

Luego, si el ID no es nulo, compara los consumos y pagos almacenados en sus registros correspondientes para ver si egresa o no mo_mo_suma.

Cuando encuentra el ID nulo, copia el registro del contador en \$0 para devolver allí el resultado y respetar la convención.

* Estas multiplicando el valor de los bytes que se tiene que desplazar en la dirección anterior con el tamaño de los elementos. Por ejemplo:

- $i = 0 : t_2 = 0$ numero uno antes de finalizar el ciclo
- $i = 1 : t_2 = 5 + 1 = 6$
- $i = 2 : t_2 = 30 + 1 = 31$

Para que funcione, entre tiene que estar el índice del elemento que quieras sumar

Por convención de llamada "balanceDeudores" ya está en \$0 cuando se llama a "contarDeudores".

④ Para ejecutar una instrucción, lo primero que debe hacer el procesador es, con el Program Counter actual, encontrar en la MEMORIA de INSTRUCCIONES cuál es la próxima a ejecutar.

En el caso de muestra "or x4, x5, x6", vamos a tener una operación a realizar entre tres registros.

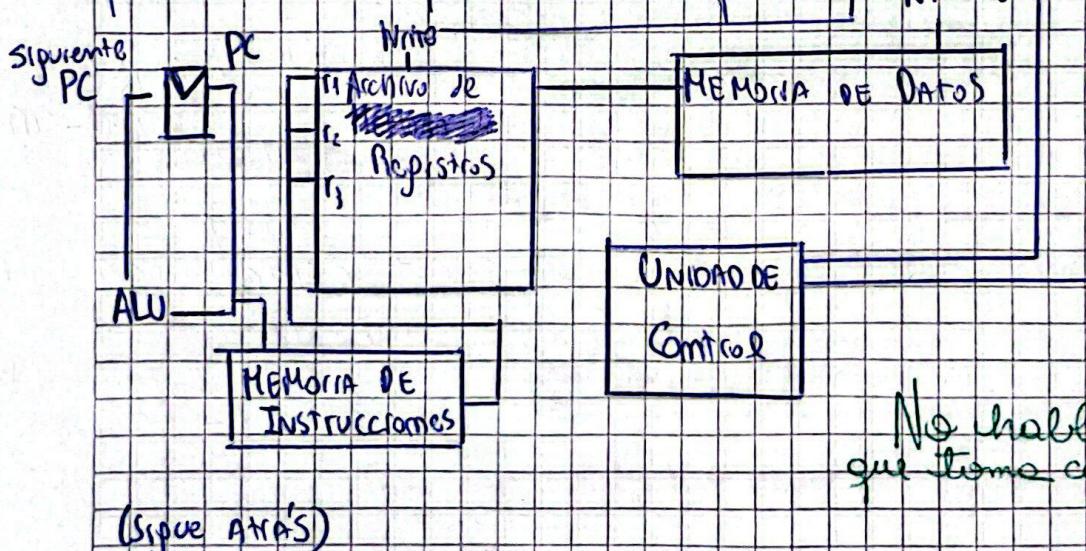
Las tiras de bits correspondientes a rd, rs1 y rs2 van a ir a la memoria de datos para evaluar su contenido.

~~memoria de datos~~

Este contenido será procesado por una ALU, que realizará la operación "or" entre el contenido de los registros rs1 y rs2, llevando el resultado a la memoria de registros para poner el valor de RD (en este caso x4) como el resultado dado por la ALU.

No fue necesario un extensor de bits, ya que no hay ningún inmediato en la instrucción de tipo R.

Como no era una instrucción de salto, la ejecución terminaría con el PC siendo PC+4, lo que se logra conectando una ALU al registro que lleva el PC que realice la operación | Write Enable



No hablo de las señales que vienen cada multiplexor.

La unidad de control debe mantener encendidas las señales de control de la siguiente manera

- El ~~W~~ Write de las MEMORIAS debe estar apagado
 - El Write de los registradores debe ESTAR ACTIVADO
 - El registro del PC debe almacenar como Próximo PC el DADO por su AW de SUMA, no por que se puede dar por una instrucción de SALTO.
- Esto se puede lograr con un multiplexor que alterne qué señal imprima al registro.

① #20 = FILA

#21 = Columna

(11, 10, 0) # temporal que voy a necesitar

(11, 02, 1) # constante para comparar brancheo a caso base

Tiene que
estor
abajo de
la let que

bit 20, 21, CASO-BASE-CERO

Bgt 22, 20, CASO-BASE-UNO

Bgt 22, 21, CASO-BASE-UNO

addi SP, SP, -16 # Guarda lugar en la pila

SW R0, 12(SP) # Guardo 20 actual en pila

SW R2, 8(SP) # Guardo la dirección de retorno

addi R0, R0, -1

jal PASCAL

lw R0, 8(SP) # Llamo PASCAL con FILA-1

SW R0, 4(SP) # Recupero dirección de retorno

SW R0, 4(SP) # Guardo PASCAL (FILA-1, columna)

lw R0, 12(SP)

addi R0, R0, -1 # Recupero valor de parámetro FILA

SW R2, 8(SP) # Guardo return address

addi R0, R1, R1, -1 # Guardo columna -1 en R1

jal PASCAL

lw R0, 4(SP) # En R0 quedará PASCAL (FILA-1, columna -1)

lw R0, 4(SP) # Recupero return address

lw R0, 4(SP) # Recupero PASCAL (FILA-1, columna)

add R0, R0 # Sumo en R0 PASCAL (FILA-1, columna) + PASCAL (FILA-1, C-1)

j FIN

CASO-BASE-CERO: li R0, 0

addi SP, SP, 16 # Restauro pila

ret # Si "ret" se ejecuta "caso base uno" también

CASO-BASE-UNO:

li R0, 1

addi SP, SP, 16 # " "

ret

FIN: Nop # No hace nada pues R0 tiene el resultado

(sigue atrás)

La implementación de pascal cumple la convención de Risc V
y en que siempre devuelve en \$t0 lo que necesita. Se asegura de guardar
en la pila los valores que va a modificar y los recupera al final.
Además, siempre que modifica el stack pointer respeta la congruencia
de cero módulo 16 y mantiene los casos base decrementando los
valores en el paso recursivo de forma de que en cada ejecución se
acerque a ellos.

② (A) Implementación Inverso Aditivo.

- Text

```
li T0, 0
li T1, 1
```

INV: Not T0, Z0 # Invierte bit a bit el número de entrada (Z0 por convención)

addi T0, T1, 1 # Sumo cte. 1 para tener el inverso aditivo.

MV Z0, T1 / # Devuelvo en Z0 el inverso aditivo de Z0.

(B) Implementación Inversor arreglo

OBS: No tengo que devolver nada porque en una función void, SOLAMENTE reemplazar en memoria ~~esta~~ elemento.

Por convención ya están los parámetros en Z0 y Z1

Invertir Arreglo: $\begin{cases} \text{Z0, Z1, arreglo} \\ \text{Z1, largo} \end{cases}$ # Toma los Argumentos

li T0, 0 # i=0

li T1, 0 # elemento actual arreglo[i]

li T2, 4 # largo en bits del elemento.

~~beg T0, Z1, FIN~~

~~MV T1, T2(Z0).~~

~~li T3, 0~~ # Inicializo desplazamiento en 0

ciclo: beg T0, Z1, FIN

Slli T3, T0, 2 # Desplazamiento = 4 · I

IW T1, T3(Z0) # CARGO elemento actual a T1

addi SP, SP, -16 # Hago IUPAC en SA pila

SW Z0, 12(SP) # Guarda dirección de comienzo del Arreglo

SW I2, 8(SP) # Guarda return address

MV Z0, T1 # Propio Z0 para LLAMADA

JAL iNN # Ahora Z0 es el elemento (invertido)

~~beg T0, Z1, FIN~~ ~~li T1, 0~~ (Sigue arriba)

MV T1, 20 # Guarda en T1 el actual invertido

IW 20, 12(sp) # Recupero dirección de comienzo del arreglo

IW T3(20), T1 # Guarda elemento invertido en array.

IW ra, 8(sp) # Recupero ra

addi sp, sp, 16 # Restaura la pila

j cicle

Fim: Nap

La función se compone de esta manera: MIENTRAS el index de la iteración actual sea menor al largo dado por el ~~parametro~~ de la función, se prepararán los registros para llamar a la función de invertir.

Después, se restaura la pila para no perder la dirección de comienzo del arreglo en la próxima iteración.

Cuando se llega al final, no altero el registro 20 ya que cada elemento del arreglo se está reemplazando y no necesito cambiar la dirección ya que el array seguirá siendo el mismo pero con los valores alterados.