Trabajo Práctico Nº 6
ARQUITECTURA DEL SET DE INSTRUCCIONES

1.- Una palabra en una computadora “little endían” tiene el valor numérico 3. Si se trasmite a una computadora “big endían” byte por byte y se almacena allí, con el byte 0 en el byte 0 y así sucesivamente, cuál es el valor numérico en la máquina “big endían”?

2.- Dados los valores de memoria indicados más abajo y una máquina de una dirección con un acumulador, qué valores almacenan las siguientes instrucciones de LOAD en el acumulador?

Palabra 20 contiene 40
Palabra 30 contiene 50
Palabra 40 contiene 60
Palabra 50 contiene 70

a) LOAD INMEDIATO 20
b) LOAD DIRECTO 20
c) LOAD INDIRECTO 20
d) LOAD INMEDIATO 30
e) LOAD DIRECTO 30
f) LOAD INDIRECTO 30

3.- A continuación se presentan dos secuencias de código escrito en lenguaje C que hacen lo mismo: calcular la sumatoria de i desde 0 hasta n. El de la izquierda sigue un método recursivo (sucesivas llamadas anidadas a una función) mientras que el de la derecha se ha realizado siguiendo un procedimiento iterativo (lazo controlado por un contador).

algoritmo recursivo	**algoritmo iterativo**

`int sumatorio (int n, int acumulado)`
`{`
`  if (n>0) return sumatorio (n - 1, acumulado + n);`
`  else return acumulado;`
`}`
	`int sumatorio (int n)`
	`{`
	`  int i, acumulado = 0;`
	`  for (i = n; i > 0; i--)`
	`  {`
	`  acumulado = acumulado + n;`
	`  }`
	`  return acumulado;`
	`}`

$\sum_{i=0}^{n} i = \sum_{i=0}^{n-1} i + n$	$\sum_{i=0}^{n} i = 0+1+2+\cdots+n$

Aunque ambas secuencias funcionan correctamente y generan el mismo resultado, la que se usa es la iterativa ya que tiene un mejor rendimiento. ¿Por qué? Describa cómo quedan las instrucciones en Assembler y cuál es su comportamiento en tiempo de ejecución.

4.- Comparar las cuatro arquitecturas presentadas en la siguiente tabla

ARQUITECTURAS					
Formato	0 direcciones	1 dirección	2 direcciones	3 direcciones	
Modo de ejecución	pila	acumulador	registro-registro	memoria-memoria	registro-registro (carga/almacenamiento)

Juego de instrucciones	PUSH M
	POP M
	ADD
	SUB
	MUL
	DIV
		LOAD M
		STORE M
		ADD M
		SUB M
		MUL M
		DIV M
			LOAD X, M
			STORE M, X
			MOVE X, Y
			ADD X, Y
			SUB X, Y
			MUL X, Y
			DIV X, Y
				ADD M1, M2, M3
				SUB M1, M2, M3
				MUL M1, M2, M3
				DIV M1, M2, M3
					LOAD X, M
					STORE M, X
					MOVE X, Y
					ADD X, Y, Z
					SUB X, Y, Z
					MUL X, Y, Z
					DIV X, Y, Z

escribiendo un programa que calcule

$$X = (A + B \times C) / (D - E \times F)$$

En cada caso aplicar el conjunto de instrucciones referido en la tabla

Tener en cuenta que:

Máquina de 0-direcciones: Las operaciones toman sus operandos de la cima de la pila y los eliminan sustituyéndolos por el resultado. Las operaciones de transferencia PUSH y POP son las únicas que realizan accesos a memoria.
Máquina de 1-dirección: Las operaciones se realizan con un operando de memoria y otro implícito que es siempre el acumulador. El set de instrucciones incluye dos instrucciones de transferencia: cargar el registro acumulador (LOAD M) o almacenar su contenido en memoria (STORE M).
Máquina de 2-direcciones: Las operaciones se realizan entre dos operandos residentes en alguno de los 16 registros con que cuenta. Las instrucciones de transferencia son las encargadas de mover información entre los registros y memoria (LOAD, STORE) o entre los propios registros.
Máquina de 3-direcciones: Se especifican tres operandos por instrucción. En la tabla se ilustran dos casos extremos: uno de ellos memoria-memoria y otro registro-registro.

5.- Cuáles son las direcciones más bajas y más altas de una memoria de 220 bytes que se accede por palabras de 4-bytes

6.- Busque en Internet especificaciones de tres procesadores de distintas marcas e indique para cada uno: a) los distintos tamaños de las instrucciones, b) la cantidad de registros generales, c) el nombre de todos los registros especiales, d) los modos de direccionamientos con sus nombres y efectos

7.- Un programa compilado para una ISA SPARC escribe el entero sin signo de 32 bits 0xABCDEF01 en un archivo, y lo recupera correctamente luego de una operación de lectura. El mismo programa compilado para una ISA Pentium también funciona correctamente. Sin embargo, cuando se transfiere el archivo entre máquinas, el programa lee el entero incorrectamente desde el archivo como 0x01EFCDAB. ¿Qué está ocurriendo?

8.- Una sección de código assembly de ARC se muestra a continuación. Qué tarea realiza? Exprese su respuesta en término de las acciones que se producen con cada instrucción. Suma números, o pone en cero algo? Simula un lazo for, un lazo while, u otra estructura?
Suponga que *a* y *b* son ubicaciones de memoria que ya están definidas en alguna otra sección del código.

```
Y:    ld [k], %r1
      addcc %r1, -4, %r1
      st %r1, [k]
      bneg X
      ld [a], %r1, %r2
      ld [b], %r1, %r3
      addcc %r2, %r3, %r4
      st %r4, %r1, [c]
      ba Y
X:    jmpl %r15 + 4, %r0
k:    40
```

9.- Escriba una subrutina en ARC que intercambie el contenido de dos variables de 32 bits guardadas en memoria principal. Utilizar el mínimo número de registros que le sea posible.

10.- Indique cuál es el tipo de pasaje de parámetros (registros, stack, variables en memoria) que se utiliza en el siguiente programa ARC.

```
! Este programa suma "length" números

! Uso de registros:    %r1 – Largo del array a
!                      %r2 – Dirección de inicio del array a
!                      %r3 – Suma parcial
!                      %r4 – Puntero dentro del array a
!                      %r5 – Guarda un elemento de a
```

```
            .begin                  ! Iniciar ensamblado
            .org 20 4 8             ! Dirección de inicio del programa en 2048
a_start     .equ 3000               ! Dirección del array a
            ld [length], %r1        ! %r1   largo del array a
            ld [address],%r2        ! %r2  dirección de inicio del array a
            andcc %r3, %r0, %r3     ! %r3
loop:       andcc %r1, %r1, %r0     ! Chequea número de elementos restantes
            be done                 ! Termina cuando length=0
            addcc %r1, -4, %r1      ! Decrementa el largo del array
            addcc %r1, %r2, %r4     ! Dirección del próximo elemento
            ld %r4, %r5             ! %r5  Memoria[%r4]
            addcc %r3, %r5, %r3     ! Suma el nuevo elemento a r3
            ba loop                 ! Repite el loop.
done:       jmpl %r15 + 4, %r0      ! Vuelve a la rutina invocante
length:     20                      ! 5 numeros (20 bytes) in a
address:    a_start
            .org   a_start          ! Inicio del array a
a:            25                    ! A continuación valores length/4
            –10
            33
            –5
            7
            .end                    ! Terminar ensamblado
```

11.- Un programa en ARC, según se muestra continuación, invoca la subrutina *foo* pasándole tres argumentos: “a”, “b” y “c”. Esta rutina no devuelve parámetros y tiene dos variables locales: “m” and “n”. Los argumentos son pasados en el stack. Mostrar la posición del stack pointer y el contenido relevante del stack en los puntos del programa que se especifican a continuación:

(a) Justo antes de ejecutar el *call* en la etiqueta *x*

(b) Cuando se completa el pasaje de parámetros para *foo*

(c) Justo antes de ejecutar *ld* en la etiqueta *z* (esto es, cuando reasume el control la rutina que invocó a *foo*.

```
                                    ! Pasa al stack los argumentos a, b, and c
x:          call foo
z:          ld %r1, %r2
.
.
.
foo:                                ! Inicio de la subrutina
.
.
.
y:          jmpl %r15 + 4, %r0
```

12.- La instrucción SETHI se aplica sobre los 22 bits más significativos de un registro. Sin embargo, sería útil que SETHI permitiera asignar un valor sobre los 32 bits del registro. Justifique porqué esto último no resulta posible.

13.- Un programa fue bajado de disco a memoria principal de una computadora ARC. En la tabla siguiente se muestra el contenido de un segmento de memoria dentro del rango ocupado por ese programa.

Dirección	Contenido
00000810	CA006BCC
00000814	CA206BB8
00000818	82206004
0000081C	02800003
00000820	80A06000
00000824	10BFFFFB

Indicar el código Assembler de ese segmento.