Programación en ensamblador
Ejercicios resueltos

Ejercicio 1. Dado el siguiente fragmento de programa en ensamblador.

```
        .text
                            .globl main
         main:
                            ...
                            li   $a0,  5
                            jal  funcion
                            move $a0, $v0
                            li   $v0, 1
                            syscall

                            li   $v0, 10
                            syscall

       funcion:             li $t0, 10
                            bgt $a0, $t0, et1
                            li $t0, 10
                            add $v0, $t0, $a0
                            b et2
             et1:           li $t0, 8
                            add $v0, $t0, $a0
             et2:           jr $ra
```

Se pide :

a) Indicar de forma razonada el valor que se imprime por pantalla (primera llamada al sistema del código anterior).

Solución:

A la función se le pasa como argumento en el registro `$a0` el valor 5. Si este valor es mayor que (`bgt`) 10 se salta a la etiqueta `et1`. Como no es el caso, se suma 10 (`$t0`) a 5 y el resultado se almacena en `$v0`, que es el valor que se imprime, es decir, 15.

Ejercicio 2. Considere una función denominada `SumarValor`. A esta función se le pasan tres parámetros:

- El primer parámetro es la dirección de inicio de un vector de números enteros.
- El segundo parámetro es un valor entero que indica el número de componentes del vector.
- El tercer parámetro es un valor entero.

La función `SumarValor` modifica el vector, sumando el valor pasado como tercer parámetro a todas las componentes del vector. Se pide:

a) Indique en qué registros se han de pasar cada uno de los parámetros a la función.
b) Programar utilizando el ensamblador del MIPS32 el código de la función `SumarValor`.
c) Dado el siguiente fragmento de programa:

```
.data
        v:  .word    7, 8, 3, 4, 5, 6
.text
        .globl main

 main:
```

incluya en el `main` anterior, las sentencias en ensamblador necesarias para poder invocar a la función `SumarValor` implementada en el apartado b) de forma que sume a las componentes del vector `v` definido en la sección de datos, el número 5. Implemente a continuación de la llamada a la función, las sentencias en ensamblador que permitan imprimir todas las componentes del vector.

Solución:

a) La dirección de inicio se pasa en $a0, el número de elementos en `$a1` y el valor a sumar en `$a2`.

b)

```
SumarValor:       li    $t0, 0
                  move  $t1, $a0

      bucle:      bgt   $t0, $a1, fin
                  lw    $t2, ($t1)
                  add   $t2, $t2, $a2
                  sw    $t2,  ($t1)
                  addi  $t0, $t0, 1
                  addi  $t1, $t1, 4
                  b bucle
      fin:        jr $ra
```

b) El cuerpo e la función `main` es:

```
.data
       v: .word  7, 8, 3, 4, 5, 6
.text
       .globl main

 main:      sub   $sp, $sp, 24
            sw    $ra, 20($sp)
            sw    $a0, 4($sp)
            sw    $a1, 8($sp)
            sw    $a2,  12($sp)

            la    $a0, v
            li    $a1, 6
            li    $a2, 5
            jal   SumarValor


            li    $v0, 1
            li    $t0, 0
           move $t1, $a0

bucle:      bgt   $t0, $a1, fin
            lw    $a0, ($t1)
           syscall
           addi  $t0, $t0, 1
           addi  $t1, $t1, 4
           b     bucle


      fin:  lw    $ra, 20($sp)
            lw    $a0, 4($sp)
            lw    $a1, 8($sp)
            lw    $a2, 12($sp)
           addi  $sp, $sp, 20

            li   $v0, 10
           syscall
           jr    $ra
```

Ejercicio 3. Dado el siguiente fragmento de programa en ensamblador del MIPS32

```
.text
                    .globl main
        main:
                    li   $a0,  5
                    jal  f1
                    move $a0, $v0
                    li   $v0, 1
                    syscall

                    li   $v0, 10
                    syscall

       f1:          li $t0, 10
                    bgt $a0, $t0, et1

                    move $t0, $a0
                    li   $t1, 0
         b1:        beq  $t0, 0, fin
                    add  $t1, $t1, $t0
                    sub  $t0, $t0, 1
                    b    bucle
         fin:       move $v0, $t1
                    b et2

        et1:        li $v0, 0
          et2:      jr $ra
```

Indique de forma razonada el valor que devuelve la función `f1` en el registro `$v0` y que se imprime por pantalla (primera llamada al sistema del código anterior).

Solución.

La función `f1` realiza la siguiente funcionalidad:

```
if  ($a0 <= 10)
      {
            $t0 = $a0;
            $t1 = 0;
            while ($t0 > 0)
            {
                  $t1 = $t1 + $t0;
                  $t0 = $t0 – 1;
            }
            return $t1;
}
else
      return 0;
```

Como el valor de `$a0` es 5 y es menor que 10, la función realiza la suma de los valores 5, 4, 3, 2, 1 y devuelve 15.

Ejercicio 4. Dado el siguiente fragmento de programa

```
.data
            a: .word 5
            b: .word 10
.text
            li    $t0  1
            lw    $t1, a
```

```
            lw    $t2, b
label1:     bgt   $t0,  $t1,   label2
            addi  $t2, $t2, 2
            addi  $t0, $t0, 1
            b     label1
label2:     sw    $t0, a
            sw $  t2,  b
```

Indique el valor que tienen los registros `$t0`, `$t1` y `$t2` y las posiciones de memoria a y b al final de la ejecución del programa

Solución:

$t0	$t1	$t2	a	b	
-	-	-	5	10	
1	-	-	5	10	
1	5	10	5	10	
1	5	12	5	10	
2	5	12	5	10	
2	5	14	5	10	
3	5	14	5	10	
3	5	16	5	10	
4	5	16	5	10	
5	5	18	5	10	
6	5	18	5	10	fin ($t0 > $t1)

Ejercicio 5. Dado el siguiente fragmento de programa en ensamblador.

```
.text
                    .globl main
 main:
                    li   $a0,  5
                    jal  funcion
                    move $a0, $v0
                    li   $v0, 1
                    syscall

                    li   $v0, 10
                    syscall
funcion:
                    move $t0, $a0
                    li   $t1, 0
          bucle:    beq  $t0, 0, fin
                    add  $t1, $t1, $t0
                    sub, $t0, $t0, 1
                    b    bucle
           fin:     move $v0, $t1
                    jr $ra
```

Se pide:

a) Indicar de forma razonada el valor que se imprime por pantalla (primera llamada al sistema del código anterior).

b) Si en el registro `$a0`, que se utiliza para el paso de parámetros a la función, el valor que se almacena se representa en complemento a uno, ¿qué rango de números podrían pasarse a la función?

Solución:

a) La función realiza la suma de los números 5, 4, 3, 2,1 y devuelve el resultado en el registro $v0, cuyo valor por tanto es 15, que es el resultado que se imprime por pantalla.
b) Para una palabra de n bits, el rango de representación de números en complemento a 1 es $[-2^{n-1}+1, 2^{n-1}-1]$. En el caso del MIPS 32, los registros son de 32 bits, por tanto n = 32 y el rango de representación sería $[-2^{31}+1, 2^{31}-1]$.

Ejercicio 6. Escriba un programa utilizando el ensamblador del MIPS R2000, que realice la suma de los cuadrados de una serie de números introducidos por el teclado. Para ello, el programa pedirá en primer lugar la cantidad de números a leer. A continuación, leerá dichos números, realizará la suma correspondiente y por último imprimirá el resultado.

Solución:

```
.data

  msg01: .asciiz "Cantidad de números a leer: "
  msg02: .asciiz "Introducir número: "
  msg03: .asciiz "El resultado es: "


.text

  main:

    # imprimir mensaje "Cantidad de números a leer: "...
    la        $a0 msg01
    li        $v0 4
    syscall

    # leer cantidad de números a leer
    li        $v0 5
    syscall
    move      $t0 $v0

    # si cantidad de números a leer es cero, terminar.
    beqz      $t0 f01

    # bucle de lectura y cálculo.
    #     $t1: contador de números
    #     $t2: resultado parcial
    li  $t1 0
    li  $t2 0

   b01:

    # imprimir mensaje "Introducir número: "
    la  $a0 msg02
    li  $v0 4
    syscall

    # leer número
    li  $v0 5
    syscall

    # cálculo del cuadrado y suma parcial
    mul       $v0 $v0 $v0
    add       $t2 $t2 $v0

    # bucle
```

```
        add     $t1 $t1 1
        blt     $t1 $t0 b01

        # imprimir mensaje "El resultado es: "...
        la  $a0 msg03
        li  $v0 4
        syscall

        # imprimir el resultado
        move    $a0 $t2
        li      $v0 1
        syscall

f01:    li  $v0 10
        syscall
```

Ejercicio 7. Escriba una rutina de nombre `imprimirPantalla` que reciba como parámetro en el registro `$a0` la dirección de una posición en memoria que contiene un vector de enteros de 32 bits. Los elementos del vector representan la dirección inicial de una tira de caracteres (terminadas en cero), terminando el vector con el elemento de valor cero. La función ha de imprimir todas las tiras de caracteres y devolver en el registro `$v0` el número de caracteres que se ha impreso en pantalla.

Solución:

```
.data
   pan1: .asciiz “uno\n”
   pan2: .asciiz “dos y tres\n”
   pan:  .word   pan1, pan2

.text

  ImprimirPantalla:
        move    $t4 $a0
        move    $t0 $a0
        li      $t1 0

      IP_ini1: # bucle para tratar tiras
            lw    $t2 ($t0)
            beqz  $t2 IP_fin1
            move  $a0 $t2
            li     $v0 4
            syscall

      IP_ini2: # bucle para contar caracteres
            lb    $t3 ($t2)
            beqz  $t3 IP_fin2
            add   $t1 $t1 1
            add   $t2 $t2 1
            b     IP_ini2

   IP_fin2:
            add   $t0 $t0 4
            b     IP_ini1

   IP_fin1: move  $a0 $t4
            move  $v0 $t1
            jr    $ra
```

```
main:
      la       $a0 pan
      jal      ImprimirPantalla

      move     $a0 $v0
      li       $v0 1
      syscall

      li       $v0 10
      syscall
```

Ejercicio 8. Considere una función denominada `Vocales`. A esta función se le pasa como parámetro la dirección de inicio de una cadena de caracteres. La función calcula el número de veces que aparece el carácter 'a' (en minúscula) en la cadena. En caso de pasar la cadena nula la función devuelve el valor -1. En caso de que la cadena no tenga ninguna 'a', la función devuelve 0. Se pide:

a) Programar utilizando el ensamblador del MIPS32 el código de la función `Vocales`.
b) Indique en qué registro se ha de pasar el argumento a la función y en qué registro se debe recoger el resultado.
c) Dado el siguiente fragmento de programa:

```
.data
        cadena:  .asciiz    "Hola"
.text
        .globl main

 main:
```

incluya en el `main` anterior, las sentencias en ensamblador necesarias para poder invocar a la función `Vocales` implementada en el apartado a) e imprimir por pantalla el valor que devuelve la función. El objetivo es imprimir el número de veces que aparece el carácter 'a' en la cadena "Hola".

Solución:

a) Se asume que la dirección de la cadena se pasa en el registro `$a0` y el resultado se devuelve en el registro `$v0`. El código de la función vocales es el siguiente

```
vocales:    li    $t0, -1     // contador del número de a
            move  $t1, $a0
            beqz  $t1, fin
            li    $t0, 0
            li    $t2, 'a'
   bucle:   lbu   $t3, ($t1)
            beqz  $t3, fin
            bneq  $t3, $t2, noA
            addi  $t0, $t0, 1
      noA:  addi  $t1, $t1, 1
            b bucle

      fin: move   $v0, $t0
            jr    $ra
```

c) Los argumentos se pasan en los registros `$aX` y los resultados en los registros `$vX`. En este caso la dirección de inicio de la cadena se pasa en `$a0` y el resultado se recoge en `$v0.`
d) El cuerpo de la función `main` es:

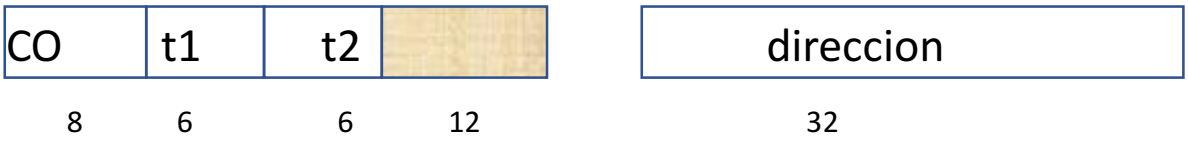
```
.data
        cadena:  .asciiz    "Hola"
```

```
.text
        .globl main

 main:      sub   $sp, $sp, 24
            sw    $ra, 20($sp)
            sw    $a0, 4($sp)

            la    $a0, cadena
            jal   vocales

            move  $a0, $v0
            li    $v0, 1
            syscall

            lw    $ra, 20($sp)
            lw    $a0, 4($sp)
            addi  $sp, $sp, 24

            li   $v0, 10
            syscall
```

Ejercicio 9. Sea un computador de 32 bits con 48 registros y 200 instrucciones máquina. Indique el formato de la instrucción hipotética `beqz $t1, $t2, dirección` donde `$t1` y `$t2` son registros y `dirección` representa una dirección de memoria.

Solución:

El número de bits necesarios para codificar 48 registros es de 6 bits. Para codificar 200 instrucciones se necesitan 8 bits. Para representar una dirección en un computador de 32 bits se necesitan 32 bits. Por tanto, son necesarias dos palabras:



Ejercicio 10. Considere una función denominada `func` que recibe tres parámetros de tipo entero y devuelve un resultado de tipo entero, y considere el siguiente fragmento del segmento de datos:

```
.data
      a: .word 5
      b: .word 7
      c: .word 9

.text
```

Indique el código necesario para poder llamar a la función anterior pasando como parámetros los valores de las posiciones de memoria a, b y c. Una vez llamada a la función deberá imprimirse el valor que devuelve la función.

Solución:

```
lw    $a0, a
lw    $a1, b
lw    $a2, c
jal   func
move  $a0, $v0
li    $a0, 1
syscall
```

Ejercicio 11. Indique una instrucción del MIPS que incluya el modo de direccionamiento relativo a registro base. ¿En qué consiste este direccionamiento?

Solución:

```
lw $t1, 20($t2)
```

El campo con este modo de direccionamiento es `20($t2)`, que representa la dirección de memoria que se obtiene de sumar 20 con la dirección almacenada en el registro `$t2`.

Ejercicio 12. Considere el siguiente fragmento en ensamblador:

```
.data
      A1: .word 5, 8, 7, 9, 2, 4, 5, 9
      A2: .word 1, 4, 3, -8, 5, 6, 5, 9
     .align 2
      A3: .space 32

.text
```

Se pide:

a) ¿Qué representa A1? ¿Cuántos bytes ocupa la estructura de datos A1 (justifique su respuesta)?
b) Se desea implementar una función cuyo prototipo en un lenguaje de alto nivel es la siguiente:

```
void Mezclar(int  a[], int b[], int c[], int N)
```

Esta función recibe 4 parámetros, los tres primeros son vectores de números enteros y el cuarto indica el número de componentes de cada uno de estos vectores. La función se encarga de almacenar en cada componente i de c, el siguiente valor: `c[i] = max (a[i], b[i])`.

Escriba, utilizando el ensamblador del MIPS32, el código correspondiente a esta función. Utilice para ello la convención de paso de parámetros que se ha descrito a lo largo del curso.

c) Escriba el código necesario para llamar a la función desarrollada en el apartado anterior, para los vectores A1, A2 y A3 definidos en la sección de datos anterior. Asuma que A3 es el vector donde se deben dejar los elementos máximos.

Solución:

a) A1 representa un vector de ocho números enteros. Si son 8 enteros, y siendo una máquina de 32 bits cada entero se representa 4 bytes, supone un total de 8*4 = 32 bytes.

b) Y c)

```
.data

 v1: .word 1, 2, 3, 4, 5
 v2: .word 5, 4, 3, 2, 1
 v3: .word 0, 0, 0, 0, 0



.text
.globl main
```

```
maximo:
         move $t0 $a0
         li   $t4 0

         move $t1 $a1
         move $t2 $a2
         move $t3 $a3

 bucle1: bge $t4 $t0 fin1
         lw $t5 ($t1)
         lw $t6 ($t2)

         bgt $t5 $t6 es5
         sw $t6 ($t3)
         b next1
    es5: sw $t5 ($t3)

  next1: add $t1 $t1 4
         add $t2 $t2 4
         add $t3 $t3 4
         add $t4 $t4 1

         b bucle1

   fin1: jr $ra

main:
        li $a0 5

        la $a1 v1
        la $a2 v2
        la $a3 v3

        jal maximo


        li $t0 0
bucle2: bge $t0 5 fin2

        lw $a0 ($a3)
        li $v0 1
        syscall

        add $a3 $a3 4
        add $t0 $t0 1
        b bucle2

fin2:   li $v0 10
        syscall
```

Ejercicio 13. Dado el siguiente fragmento de código escrito en C, escriba utilizando el ensamblador del MIPS 32 el código de la función equivalente.

```
int máximo(intA, int B)
{
      if (A > B)
            return A;
      else
            return B;
```

```
}
```

Utilizando la función en ensamblador anterior implemente el código de la siguiente función utilizando el ensamblador del MIPS 32.

```
void maximoV (int v1, int v2, int v3, int N)
{
      int i;

      for (i = 0; i < N; i++)
            v3[i] = maximo(v1[i], v2[i]);
      return;
}
```

Para el desarrollo de este ejercicio ha de seguirse la convención de paso de parámetros vista en el temario de la asignatura.

Solución:

```
maximo:       bgt   $a0, $a1, then
              move $v0, $a1
              jr    $ra

then:         move $v0, $a0
              jr    $ra


maximoV:      subu  $sp, sp, 20
              sw    $a0, ($sp)
              sw    $a1, 4($sp)
              sw    $a2, 8($sp)
              sw    $a3, 12($sp)
              sw    $ra, 16($sp)

              move  $t0, $a0
              move  $t1, $a1
              move  $t2, $a2
              li    $t3, 0
              move  $t4, $a3

bucle:        bge   $t3, $t4, finBucle
              lw    $a0, ($t0)
              lw    $a1, ($t1)
              jal   maximo
              sw    $v0, (t2)

              addi  $t0, $t0, 4
              addi  $t1, $t1, 4
              addi  $t2, $t2, 4
              addi  $t3, $t3, 1
              b     bucle

finBucle:     lw    $ra, 16($sp)
              lw    $a3, 12($sp)
              lw    $a2, 8($sp)
              lw    $a1, 4($sp)
              lw    $a0, ($sp)
              add   $sp, $sp, 20
```

```
                jr    $ra
```

Ejercicio 14. Considere la siguiente definición de función:

```
int  sustituir (String cadena, char c1)
```

La función sustituye cada ocurrencia de carácter c1 que aparece en la cadena de caracteres cadena, por el último carácter de cadena. La función devuelve también la posición del último carácter cambiado en la cadena.

Ejemplo: Si `cadena = "Hola mundo"`, y `c1 = 'a'`. La función debe modificar cadena para que su nuevo valor sea `"Holo mundo"`. La función devolvería para este caso el valor 3.

Se pide:

a) Implemente el código de la función anterior utilizando el ensamblador del MIPS 32.
b) Dado el siguiente fragmento del segmento de datos:

```
.data
      Cad: .ascizz "Esto es una cadena de prueba"
```

Indique el código necesario para invocar a la función sustituir pasando como parámetro la cadena `Cad` y el carácter `'a'`. Imprima el resultado que devuelve la función por pantalla. (Tiene que seguirse la convención en el paso de parámetros).

Solución:

```
a)
      sustituir:
                  move  $t0, $a0
                  li    $v0, 0      // posición del último caracter cambiado
                  lbu   $t1, ($t0)
                  bneqz $t1, bucle1
                  jr    $ra

        bucle1:   beqz  $t1, fin1
                  lbu   $t2, ($t0)
                  addi  $t0, 1
                  lbu   $t1, ($t0)
                  b bucle1

      // en $t2 está el ultimo caracter de la cadena

         fin1:    move  $t0, $a0
                  lbu   $t1,  ($t0)
        bucle2:   beqz  $t1, fin2
                  bneq  $t1, $t2, seguir
                  sw    $a1,  ($t0)
                  addi  $v0, $v0, 1
        seguir:   addi  $t0, 1
                  lbu   $t1, ($t0)
                  b     bucle2

         fin2:    jr    $ra

b)
      la    $a0, Cad
      li    $a1, 'a'
      jr    sustituir
      move  $a0, $v0
      li    $v0, 1
      syscall
```

Ejercicio 15. Se desea desarrollar utilizando el ensamblador del MIPS 32 el código de la siguiente función:

```
void vuelta(char[] cadena_origen, char[] cadena_final);
```

que toma como parámetros dos cadenas de texto. Esta función invierte la primera cadena y guarda el resultado en la segunda. Así, por ejemplo, la llamada a esta función con `"Hola Mundo"` almacenará en `cadena_final` la cadena `"odnuM aloH"`. Considere que la cadena final tiene al menos el mismo espacio reservado que la cadena origen. Considere además que el final de la cadena se indica con `'\0'`. Se pide:

a) Indique el algoritmo que va a utilizar para implementar la rutina anterior.
b) Desarrolle, de acuerdo al algoritmo descrito en el apartado a), el contenido de la subrutina utilizando el ensamblador del MIPS 32. Debe seguirse estrictamente el convenio de paso de parámetros descrito en la asignatura.
c) Desarrolle la función `void imprimir(char[] cadena)` en ensamblador MIPS 32. Dicha función toma como parámetro una cadena de caracteres y la imprime por pantalla. Debe seguirse estrictamente el convenio de paso de parámetros descrito en la asignatura.
d) Suponga que cuenta con el siguiente segmento de datos:
e)

```
.data
      cadena1:    .asciiz "Hola Mundo"
      .align 2
      cadena2:    .space 11
```

desarrolle, utilizando las rutinas anteriores, una rutina `main` que primero de la vuelta a cadena1 y la almacene en `cadena2`, y posteriormente imprima el contenido de `cadena2` por pantalla. Para ello, tenga en cuenta el convenio de paso de parámetros.

Solución:

a) Un posible algoritmo sería el siguiente:

```
Apuntar puntero 1 al comienzo de cadena origen
Longitud de cadena origen = 0
Mientras no se lea un 0
      Leer carácter
      Aumentar longitud en 1
      Mover puntero de cadena origen a siguiente posición
Apuntar puntero 1 al comienzo de cadena final
Retroceder una posición puntero de cadena origen
Mientras longitud sea mayor o igual a 0
      Leer carácter en cadena origen
      Escribir carácter en cadena final
      Mover puntero de cadena origen a posición anterior
      Mover puntero de cadena final a siguiente posición
      Disminuir longitud en 1
Escribir final de cadena en cadena final
```

b) El código correspondiente al algoritmo anterior sería el siguiente:

```
vuelta:
      li    $t0 0             #Contador de longitud
      li    $t1 0             #Para almacenar caracteres
      move $t2 $a0

for1:                         #Medición de la longitud
      lb    $t1 ($t2)         #Carga de un caracter
      beqz $t1 seguir         #Localizar '\0'
```

```
        add   $t0 $t0 1          #Incremento de longitud
        add   $t2 $t2 1          #Incremento de puntero de cadena1
        j     for1

    seguir:
        sub   $t2 $t2 1          #Se vuelve a la última posición de cadena1

    for2:                        #Bucle para invertir la cadena
        bgt   $a0 $t2 fin        #Terminar de leer cadena1
        lb    $t1 ($t2)          #Carga de un carácter
        sb    $t1 ($a1)          #Almacenamiento de un carácter
        sub   $t2 $t2     1      #Decremento puntero de cadena1
        add   $a1 $a1     1      #Incremento puntero de cadena2
        j     for2

    fin:                         #Se cierra la cadena2
        li    $t1 0
        sb    $t1 ($a1)

        jr    $ra
```

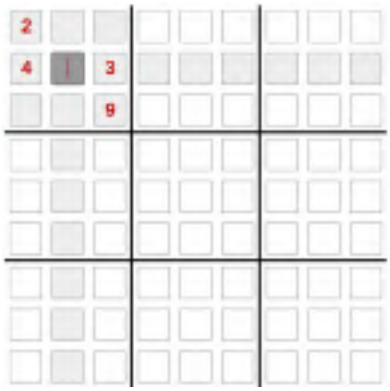
c) El código de la función `imprimir` sería:

```
    imprimir:
        li    $v0 4
        syscall

        jr    $ra
```

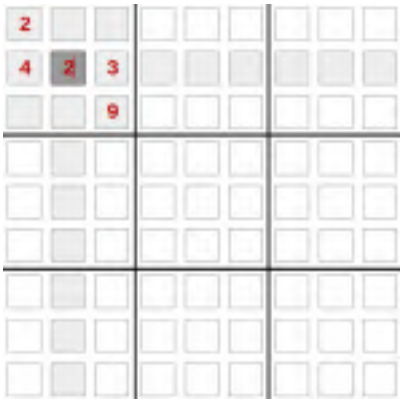
d)

```
    main:
        sw    $ra ($sp)
        sub   $sp $sp 4

        la    $a0 cadena1
        la    $a1 cadena2
        jal   vuelta

        la    $a0 cadena2
        jal   imprimir

        lw    $ra 4($sp)
        add   $sp $sp 4

        jr    $ra
```

Ejercicio 16. Como parte de un programa de resolución de sudokus, se desea desarrollar una subrutina en ensamblador del MIPS32 que compruebe si el valor de una celda cumple con la regla de los cuadrados 3x3, es decir, un número no se puede repetir dentro de un mismo cuadrado de tamaño 3x3. Un sudoku se divide en 9 cuadrados de 3 x 3. Dentro de cada uno de estos cuadrados de 3x3 no se puede repetir un número (del 1 al 9), independientemente de si están o no en la misma fila o columna. El siguiente sudoku cumple con esta regla:



Sin embargo, el siguiente sudoku no cumple la regla, ya que en el primer cuadrado 3x3 se repite el número 2.



Suponga que el sudoku se almacena como una matriz de 9 elementos por 9 elementos. Dicha matriz se almacena por filas consecutivas en la sección de datos y cada elemento de la matriz es de 1 byte. La rutina a desarrollar recibirá como parámetros:

- La dirección de inicio del sudoku
- El valor que se quiere introducir en el sudoku (un número de 0 a 9)
- El índice de la fila y columna donde se quiere introducir el valor anterior. El índice de la fila y la columna irán de 0 a 8.

La función devolverá un 0 si el contenido que se quiere introducir en el sudoku cumple con la regla y 1 si es incorrecto (no cumple con la regla)

.

Se pide:

a) Indique un posible pseudocódigo de la función anterior.
b) Desarrollo el código en ensamblador que implemente el algoritmo anterior.
c) Se desea desarrollar una rutina `main` que llame a la subrutina anterior. Suponga que cuenta con el siguiente segmento de datos:

```
.data
valor:      .word 8
fila:       .word 1
columna:    .word 2
tablero:  .byte     0, 6, 0, 1, 0, 4, 0, 5, 0
          .byte     0, 0, 8, 3, 0, 5, 6, 0, 0
          .byte     2, 0, 0, 0, 0, 0, 0, 0, 1
          .byte     8, 0, 0, 4, 0, 7, 0, 0, 6
          .byte     0, 0, 6, 0, 0, 0, 3, 0, 0
          .byte     7, 0, 0, 9, 0, 1, 0, 0, 4
          .byte     5, 0, 0, 0, 0, 0, 0, 0, 2
```

```
                .byte     0, 0, 7, 2, 0, 6, 9, 0, 0
                .byte     0, 4, 0, 5, 0, 8, 0, 7, 0
```

Solución:

a) El pseudocódigo es el siguiente:

```
Guardar cociente de fila/3 en Y
Repetir 3 veces:
      Guardar cociente de columna/3 en X
      Repetir 3 veces
            Situarse en (X, Y)
            Si la posición no es la que se quiere comprobar y el valor
            es igual
                  No se cumple la regla
                  Terminar
            Si no
                  X + 1
Y + 1
```

b) Código:

```
comprobar_cuadrado:
            div   $t0, $a1, 3       # $t0 = fila / 3
            mul   $t0, $t0, 27      # Desplazamiento de la fila
            div   $t1, $a2, 3       # $t1 = col / 3
            mul   $t1, $t1, 3       # Desplazamiento de la columna
            add   $t1, $t0, $t1     # Desplazamiento de la primera celda de
                                    # la caja
            li    $t0, 3            # Contador de filas
            li    $t3, 3            # Contador de columas
            li    $v0 1             # Carga el valor de acierto (1)

        for:
            add   $t4 $t1 $a3       # Índice de la celda
            lb    $t2, ($t4)        #Valor de la celda
            beq   $a0, $t2, incorrecto    #Número repetido en la caja
            sub   $t3, $t3, 1             #Disminuye el contador de columnas
            beqz  $t3, fin                #Condición de parada
            addi  $t1, $t1, 1             # Incrementa el desplazamiento
            j for

        incorrecto:
            li $v0, 0               # 0 en caso de fallo

        fin:
            jr $ra
```

c)

```
      main:
            lw    $a0 valor
            lw    $a1 fila
            lw    $a2 columna
            la    $a3 tablero

            sw    $ra ($sp)
            sub   $sp $sp 4
            jal   comprobar_cuadrado

            add   $sp $sp 4
```

```
        lw    $ra ($sp)
        jr    $ra
```

Ejercicio 17. Considere la rutina `Contabilizar`. Esta rutina acepta dos parámetros de entrada:

- Un vector de números de tipo *float*.
- El número de elementos del vector
- La función devuelve tres valores:
- El número de elementos con valor igual a 0.
- El número de elementos correspondientes a valores normalizados distintos de 0.
- El número de elementos correspondientes a valores no normalizados (no se incluyen los valores de tipo NaN).

Se pide:

a) Codifique correctamente la rutina `Contabilizar` anteriormente descrita. Puede hacer uso de las rutinas auxiliares que considere oportuno. Ha de seguirse estrictamente el convenio de paso de parámetros y uso de pila, aunque no es necesario hacer uso del registro de marco de pila.

b) Dada la siguiente definición de vector:

```
.data
     vector:   .float 0.0, 0.1, -0.2, 1.0, 1.1, 1.2, 2.0, 2.1, 2.2
```

Codifique el fragmento de código que permite invocar correctamente a la función `Contabilizar` e imprimir los valores que devuelve dicha función.

Solución:

a) Una posible implementación sería:

```
EsCero:     #comprueba si $f12 (argumento de tipo float) es 0.0
            mfc1  $t0, $f12
            beqz  $t0, true1
            li    $v0, 0            # 0: no es cero
            jr    $ra
true1:      li    $v0, 1            # 1: es cero
            jr    $ra

EsNormalizado:    #comprueba si $f12
            # (argumento de tipo float) es normalizado
            # es normalizado si:   0  < exponente < 255
            mfc1  $t0, $f12
            li    $t1, 0x7F800000   # se obtiene el exponente
            beqz  $t1, falso1
            li    $t2, 255
            beq   $t1, $t2, falso1
            li    $v0, 1            # 1: es normalizado
            jr    $ra
falso1:     li    $v0, 0            # 0: no es normalizado
            jr    $ra

EsNoNormalizado:
            #comprueba si $f12 (argumento de tipo float)
            # es no normalizado
            # exponente igual a 0 y mantisa distinta de 0

            mfc1  $t0, $f12
            li    $t1, 0x7F800000  # se obtiene el exponente
            beqz  $t1, comprueba
```

```
            li    $v0, 0      #0: no es no normalizado
            jr    $ra
comprueba:  li    $t1, 0x007FFFFF # se obtiene la mantisa
            beqz  $t1, falso2
            li    $v0, 1      # 1: es no normalizado
            jr    $ra
falso2:     li    $v0, 0      # 0: no es no normalizado
            jr    $ra


Contabilizar:
            addi  $sp, $sp, -20
            sw    $s0, 16($sp)
            sw    $s1, 12($sp)  # número de ceros
            sw    $s2,  8($sp)  # número de normalizados
            sw    $s3,  4($sp)  # número de no normalizados
            sw    $ra,   ($sp)

            li    $s0, 0      # índice del bucle
            li    $s1, 0      # inicializar número de ceros
            li    $s2, 0      # inicializar número de norm.
            li    $s3, 0      # inicializar número de no norm.


bucle:      bgt   $s0, $a1, fin
            l.s   $f12, ($a0)       # el siguiente elemento
                                    # del vector

            jal   EsCero
            addi  $s1, $s1, $v0

            l.s   $f12, ($a0)
            jal   EsNormalizado
            addi  $s2, $s2, $v0

            l.s   $f12, ($a0)
            jal   EsNoNormalizado
            addi  $s3, $s3, $v0

            addi  $s0, $s0, 1
            addi  $a0, $a0, 4       # preparar el siguiente
                                    # elemento del vector
            b     bucle

      fin:  move  $v0, $s1    # número de ceros
            move  $v1, $s2    # número de normalizados
            move  $t0, $s3    # número de no normalizados

            lw    $s0, 16($sp)
            lw    $s1, 12($sp)
            lw    $s2,  8($sp)
            lw    $s3,  4($sp)
            lw    $ra,   ($sp)

            # número de no normalizados en la cima de la pila
            addi  $sp, $sp, 16
            sw          $t0, ($sp)
            jr          $ra
```

b) El fragmento necesario para invocar a la función anterior es:

```
# se pasan los parámetros de entrada
la    $a0, matriz
li    $a1,12

jal   Contabilizar

# primer resultado en $v0, segundo en $v1 y
# el tercero en la cima de la pila
move  $a0, $v0
li    $v0, 1
syscall          # se imprime el número de ceros

move  $a0, $v1
syscall          # se imprime el número de normalizados

# se extrae el tercer resultado de la cima de la pila
lw    $a0, ($sp)
addi  $sp, $sp, -4
syscall          # se imprime el número de no normalizados
```

Ejercicio 18. Considere la rutina `sumar`. Esta rutina acepta tres parámetros de entrada:

- Un matriz cuadrada de números de tipo entero.
- Un vector de números de tipo entero
- Un número entero, que representa la dimensión de la matriz y del vector.

La función suma a cada fila de la matriz el vector pasado como segundo argumento.

a) Codifique en el ensamblador del MIP32 la rutina `sumar` anteriormente descrita. Puede hacer uso de las rutinas auxiliares que considere oportuno. Ha de seguirse estrictamente el convenio de paso de parámetros y uso de pila, aunque no es necesario hacer uso del registro de marco de pila.

b) Dada la siguiente definición de matriz:

```
.data
     matriz: .word 8, 4, 5
             .word 0, 9, 7
             .word 4, 4, 1

     vector: .word 0, 1, 3
```

Codifique el fragmento de código que permite invocar correctamente a la función `sumar` e imprimir los valores que devuelve dicha función.

Solución:

a)

```
sumar:    move    $t0, $a0
          li      $t3, 1

     B1:  move    $t1, $a1
          li      $t2, 1
     B2:  bgt     $t2  $a2, fin2
          lw      $t4  ($t0)
          lw      $t5  ($t1)
          add     $t5  $t5  $t4
          sw      $t5  ($t0)
          addi    $t0  $t0 4
          addi    $t1  $t1 4
```

```
        addi    $t2  $t2 1
        b       B2
fin2:   addi    $t3  $t3 1
        bgt     $t3  $a2, fin1
        b       B1
fin1:   lw      $v0 ($a0)
        jr      $ra
```

b)

```
        # se pasan los parámetros de entrada en $a0, $a1 y $a2
        la      $a0, matriz
        la      $a1, vector
        li      $a2, 3

        # se invoca a la subrutina
        jal     sumar

        move    $a0, $v0
        li      $v0, 1
        syscall
```

Ejercicio 19. Considere una matriz de números enteros de 32 bits con f filas y c columnas, que se almacena en memoria por filas. Se desea codificar, usando el ensamblador del MIPS, la rutina `XCH` a la que se pasa por parámetros (en este orden):

- La dirección de inicio de la matriz
- El número de filas de la matriz
- El número de columnas de la matriz
- Un número que identifica a la fila i
- Un número que identifica a la fila j

Esta función se encarga de intercambiar todos los elementos que están en las filas `i` por los de la fila `j`, de forma que en la dirección de memoria donde está el elemento k-ésimo de la fila `i` se coloca el elemento k-ésimo de la fila `j` y viceversa. La función no devuelve ningún resultado.
Ha de usar el convenio de paso de parámetros y uso de pila.

a) Indique el convenio de paso de parámetros para la rutina `XCH`, es decir, dónde y de qué forma se pasan los argumentos a esta función.
b) Codifique correctamente la rutina `XCH` anteriormente descrita. Para ello, escriba en primer lugar el pseudocódigo de la solución empleada. Considere también que no hay que hacer control de errores.
c) Dada la siguiente definición de matriz:

```
.data
     matriz: .word 00, 01, 02
             .word 10, 11, 12
             .word 20, 21, 22
             .word 30, 31, 32
```

codifique el fragmento de código que permite, utilizando la función `XCH`, intercambiar la fila 1 por la fila 3 de la matriz anteriormente definida.

Solución:

a) La función XCH recibe 5 parámetros:
   - La dirección de inicio de la matriz se pasa en $a0
   - El número de filas de la matriz en $a1
   - El número de columnas de la matriz en $a2
   - Un número que identifica a la fila i en $a3
   - Un número que identifica a la fila j en la cima de la pila.

b) Un posible pseudocódigo para la función es el siguiente:

```
void XCH(int m[][], int f, int c, int i, int j) {
   int k, aux;

   if (i == j)
         return;

         for (k = 0; k < c; k++) {
               aux = m[i][k];
               m[i][k] = m[j][k]
               m[j][k] = aux;
         }
         return;
   }
```

Un posible fragmento en ensamblador sería el que se muestra a continuación. Se va a utilizar el registro $t0 para almacenar la dirección del elemento m[i][0] , es decir, el primer elemento de la fila i y el registro $t1 para la dirección del primer elemento de la fila j, m[j][0]. En general, el elemento m[k][0] se encuentra en la dirección de memoria m + k × c × 4

```
XCH:        lw    $t4, ($sp)  # en $t4 se almacena el quinto argumento j
            bneq  $a3, $t4, no_iguales
            jr    $ra   #fila i y j iguales, finalizar

no_iguales: li    $t0, 4
            mul   $t0, $t0, $a2
            mul   $t0, $t0, $a3
            add   $t0, $t0, $a0
            # en $t0 se ha almacenado la dirección de inicio de m[i][0]

            li    $t3, 4
            mul   $t1, $a2, $t3
            mul   $t1, $t1, $t4
            add   $t1, $t1, $a0
            # en $t1 se ha almacenado la dirección de inicio de m[j][0]

            # ahora se recorre la fila i y la j y se intercambian los valores
            li    $t3, 0     # índice utilizada para recorrer las filas
  bucle:    bge   $t3, $a2, fin
            lw    $t4, ($t0)
            lw    $t5, ($t1)
            sw    $t4, ($t1)
            sw    $t5, ($t0)
            addi  $t0, $t0, 4  # dirección del siguiente elemento de la fila i
            addi  $t1, $t1, 4  # dirección del siguiente elemento de la fila j
            addi  $t3, $t3, 1
            b     bucle
            jr    $ra
```

c) Teniendo en cuenta que los parámetros se pasan de la siguiente forma:

- La dirección de inicio de la matriz se pasa en `$a0`
- El número de filas de la matriz en `$a1`
- El número de columnas de la matriz en `$a2`
- Un número que identifica a la fila `i` en `$a3`
- Un número que identifica a la fila `j` en la cima de la pila.

El fragmento necesario para invocar a la función anterior es:

```
la    $a0, matriz
li    $a1, 3
li    $a2, 3
li    $a3, 1
li    $t0, 1
addi  $sp, $sp, -4
sw    $t0, ($sp)
jal   XCH
addi  $sp, $sp, 4
```

Ejercicio 20. Considere una función denominada `SumaImpares` que recibe tres parámetros:

- El primero es la dirección de inicio de un vector de números enteros.
- El segundo un valor entero que indica el número de componentes del vector.
- El tercero es un valor entero.

La función `SumaImpares` modifica el vector, sumando el valor pasado como tercer parámetro a todas sus componentes impares. Considere que el vector no puede estar vacío.

Se pide:

a) Indique en qué registros se han de pasar a la función cada uno de los parámetros.
b) Programar el código de la función `SumaImpares` utilizando el ensamblador del MIPS32 y comentando todas las líneas.
c) Dado el siguiente fragmento de programa:

```
.data
      v: .word 3, 4, 5, 9, 6, 4, 1, 8, 2, 7
.text
.globl main
      main:
```

Incluya en el `main` las sentencias en ensamblador necesarias para poder invocar a la función `SumaImpares` de forma que sume el número 3 a las componentes impares del vector v definido en la sección de datos, e imprima después todas las componentes del vector.

Solución.

a) La dirección de inicio se pasa en `$a0`, el número de elementos en `$a1` y el valor a sumar en `$a2`.

b)

```
SumaImpares:
            li    $t0, 0
            move  $t1, $a0
bucle:      bgt   $t0, $a1, fin
            lw    $t2, ($t1)
            div   $t2, 2
            bgtz  HI siguiente
            add   $t2, $t2, $a2
            sw    $t2, ($t1)
siguiente:
            addi  $t0, $t0, 1
            addi  $t1, $t1, 4
            b     bucle
fin:
            jr $ra
```

c) El cuerpo de la función `main` es:

```
.data
      v: .word 3, 4, 5, 9, 6, 4, 1, 8, 2, 7
```

```
      .text
      .globl main

      main:
      sub   $sp, $sp, 24
      sw    $ra, 20($sp)
      sw    $a0, 4($sp)
      sw    $a1, 8($sp)
      sw    $a2, 12($sp)
      la    $a0, v
      li    $a1, 6
      li    $a2, 5
      jal   SumaImpares
      li    $v0, 1
      li    $t0, 0
      move  $t1, $a0
bucle:
      bgt   $t0, $a1, fin
      lw    $a0, ($t1)
      syscall
      addi  $t0, $t0, 1
      addi  $t1, $t1, 4
      b     bucle
fin:
      lw    $ra, 20($sp)
      lw    $a0, 4($sp)
      lw    $a1, 8($sp)
      lw    $a2, 12($sp)
      addi  $sp, $sp, 20
      li    $v0, 10
      syscall
      jr    $ra
```

Ejercicio 21. Considere la rutina `ContabilizarCeros`. Esta rutina acepta tres parámetros de entrada:

- Una matriz de números de tipo *float*.
- El número de filas de la matriz (N)
- El número de columnas de la matriz (M)

La función devuelve como valor el número de la fila (comprendido entre 0 y N-1) que tiene mayor número de valores igual a 0.

Se pide:

a) Codifique correctamente la rutina `ContabilizarCeros` anteriormente descrita. Puede hacer uso de las rutinas auxiliares que considere oportuno. Ha de seguirse estrictamente el convenio de paso de parámetros y uso de pila, aunque no es necesario hacer uso del registro de marco de pila.

b) Dada la siguiente definición de matriz de dimensión 3x3:

```
.data
     M: .float    0.0, 0.1, -0.2,
                  1.0, 1.1, 1.2,
                  2.0, 2.1, 2.2
```

Codifique el fragmento de código que permite invocar correctamente a la función `ContabilizarCeros` e imprimir los valores que devuelve dicha función.

Solución:

a) Una posible implementación sería:

```
ContabilizarCeros:
                    # pila (guardo $ra)
                    subu  $sp $sp 4
                    sw    $ra ($sp)

                    # v0 -> fila con más ceros
                    # t0 -> número de ceros de la fila $v0
                    li     $v0 -1
                    li     $t0 -1

                    li    $t1 0
            b1:     beq   $t $a1 fin1
                    li    $t2 0
                    li    $t4 0 # número de ceros de la fila
            b2:     beq   $t2 $a2 fin2

                    # $t3 -> dirección del elemento
                    mul   $t3 $t1 $a2
                    add   $t3 $t3 $t2
                    mul   $t3 $t3 4
                    add   $t3 $t3 $a3

                    # $t4 se incrementa si es cero
                    lw    $t3 ($t3)
                    and   $t3 $t3 0x7FFFFFFF
                    bne   $t3 $0 nocero
                    addi $t4 $t4 1
         nocero:
                    bgt   $t4 $t0 nomayor
                    mv    $t0 $t4
                    mv    $v0 $t1
        nomayor:

                    addi  $t2 $t2 1
                    b     b2
               fin2: addi  $t1 $t1 1
                    b b1
               fin1:
                    lw    $ra ($sp)
                    addu  $sp $sp 4
                    jr    $ra
```

d) El fragmento necesario para invocar a la función anterior es:

```
# se pasan los parámetros de entrada
 la    $a0, M
 li    $a1, 3
 li    $a2, 3

 jal   ContabilizarCeros

 move  $a0, $v0
 li    $v0, 1
 syscall
```

Ejercicio 22. Considere la rutina `SumaExponentes`. Esta rutina acepta cuatro parámetros de entrada:

- Una matriz A de números de tipo *float*.
- Una matriz B de números de tipo *float*.
- Una matriz C de números de tipo *int*.

- El número de filas y columnas de las tres matrices (N). Se asumen que las matrices son cuadradas.

La rutina almacena en el elemento (i,j) de la matriz C la suma de los exponentes (los exponentes reales del número, es decir, eliminando el exceso o sesgo que se introduce cuando se representa en coma flotante) de los números Aij y Bij, es decir:

```
C[i,j] = Exponente real de A[i,j] + Exponente real de B[i,j].
```

Asuma que las matrices A y B almacenan solo números normalizados o números con valor 0. Tenga en cuenta que al valor 0 le corresponde el exponente 1.

La función devuelve como valor el número de elementos de C que han tomado el valor 0.
Se pide:

a) Codifique correctamente la rutina `SumaExponentes` anteriormente descrita. Puede hacer uso de las rutinas auxiliares que considere oportuno. Ha de seguirse estrictamente el convenio de paso de parámetros y uso de pila, aunque no es necesario hacer uso del registro de marco de pila.

b) Dada las siguientes definiciones de matrices de dimensión 3x3:

```
.data
     A:     .float      0.0, 0.1, -0.2,
                        1.0, 1.1, 1.2,
                        2.0, 2.1, 2.2

     B:     .float      0.0, 0.1, -0.2,
                        4.0, 1.1, 1.2,
                        2.0, 8.1, 2.2

     A:     .word       0, 0, 0,
                        0, 0, 0,
                        0, 0, 0
```

Solución:

a) Una posible implementación sería:

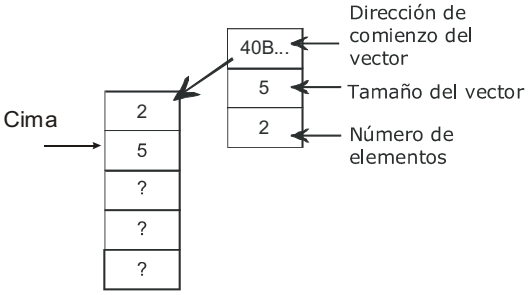
```
EsCero:     #comprueba si $f12 (argumento de tipo float) es 0.0
            mfc1        $t0, $f12
            beqz        $t0, true1
            li          $v0, 0            # 0: no es cero
            jr          $ra
true1:      li          $v0, 1            # 1: es cero
            jr          $ra


Exponente:  # devuelve el exponente de un número normalizado distinto
            # de 0
            mfc1        $t0, $f12
            li          $t1, 0x7F800000  # se obtiene el exponente
            srl         $v0, $t1, 20     # se desplaza para obtenerlo
            jr          $ra

SumaExponentes:
            # pila (guardo $ra)
            subu  $sp $sp 4
            sw    $ra ($sp)

            li    $t8 0        # v0 número de elementos de C con valor 1

            li    $t1 0        # $t1 -> i

      b1:   beq   $t1 $a3 fin1
```

```
              li    $t2 0         # $t2 -> j

        b2:   beq   $t2 $a3 fin2

              l.s   $f12, ($a1)
              jal   Escero
              bneq  $v0,  $0,  no1
              li    $t5, 1
              b       cont1
        no1:  jal   Exponente
              sub     $t5, $v0, 127
      cont1:  l.s   $f12, ($a2)
              jal   Escero
              bneq  $v0,  $0,  no2
              li    $t6, 1
              b       cont2
        no2:  jal   Exponente
              sub     $t6, $v0, 127

cont2:        add   $t7, $t5, $t6
              bneq  $t7, $0, nocero
              addi  $t8, 1
    nocero:   sw    $t7, (a2)
              addi  $a0, 4
              addi  $a1, 4
              addi  $a2, 4
              addi  $t2, 1
              b     b2
     fin2:    addi  $t1, 1
              b     b1
     fin1:    move  $v0, $t8
              lw    $ra ($sp)
              addu  $sp $sp 4
              jr    $ra
```

f) El fragmento necesario para invocar a la función anterior es:

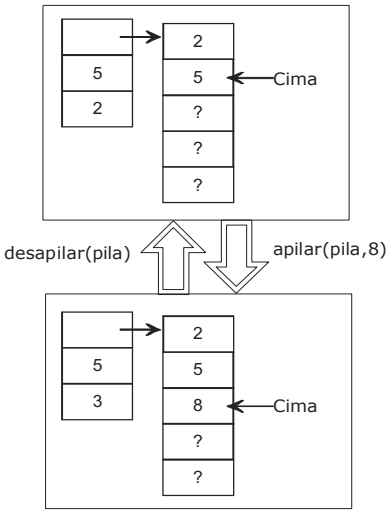
```
      # se pasan los parámetros de entrada
      la    $a0, A
      la    $a1, B
      la    $a2, C
      li    $a3, 3

      jal   SumarExponentes

      move  $a0, $v0
      li    $v0, 1
      syscall
```

Ejercicio 23. Se desea implementar una estructura de datos de tipo pila para números enteros. Para ello se ha decidido usar una serie de subrutinas en ensamblador que implementan la funcionalidad asociada a una estructura de datos de tipo pila. Estas subrutinas son: `crear`, `apilar`, `desapilar` y es_`vacia`. La disposición en memoria de la estructura de datos de tipo pila se detalla en la siguiente figura.



La subrutina crear recibe como parámetro en el registro `$a0` el máximo número de elementos que la pila puede contener. Dicha subrutina reservará memoria para tres palabras, usadas para los datos de gestión de la pila. Dichas palabras se disponen consecutivamente en memoria y contienen: la dirección de comienzo del vector usado para almacenar los elementos de la pila, el número de palabras máximo que pueden almacenarse en dicho vector y el número de elementos almacenados en la pila. También se encarga de reservar la memoria asociada al vector. Devuelve en el registro `$v0` la dirección de memoria de la estructura de control ya iniciada, que será necesaria para el resto de rutinas que gestionan la pila.

La subrutina `es_vacía` recibe en `$a0` la pila y comprueba si hay algún elemento almacenado en ella. Devuelve un 0 en el registro `$v0` si no hay ninguno o bien un 1 en el registro `$v0` si hay, al menos, un elemento.



La subrutina apilar recibe dos parámetros: en `$a0` la pila y en `$a1` el elemento a insertar. Si hay espacio en la pila, inserta dicho elemento en la cima, actualiza el número de elementos de la pila y devuelve un 1 en `$v0`. Si no hay espacio libre en la pila devuelve un 0 en `$v0`. Los elementos se insertan en direcciones crecientes de memoria (véase la figura anterior).

La subrutina `desapilar` recibe como parámetro en `$a0` la pila. Si hay elementos en la pila, extrae el elemento que ocupa la cima de la pila y lo devuelve en `$v1`. A continuación, actualiza los campos necesarios. En caso de que la pila esté vacía almacenará un 0 en `$v0`; si se puede extraer un elemento dejará en `$v0` un 1.

Implementar las subrutinas anteriormente citadas. Para ello se usará el ensamblador del MIPS R2000, siguiendo las especificaciones indicadas.

Solución:

```
.data
```

```
      msg_ec: .asciiz "Estructura de Computadores (UC3M)\n"

.text
.globl main

      crear:
            # estructura
            move  $t0 $a0
            li    $a0 12
            li    $v0 9
            syscall
            move  $t1 $v0

            # vector
            move  $a0 $t0
            li    $v0 9
            syscall

            # enlazar
            sw    $v0 0($t1)
            sw    $t0 4($t1)
            sw    $0  8($t1)

            # fin
            move  $v0 $t1
            j     $ra

      es_vacia:
            # comprobar si vacia
            li    $v0, 1
            lw    $t0 8($a0)
            beqz  $t0 fin_1
            li    $v0 0
      fin_1: j    $ra

      apilar:
            # comprobar si llena
            li    $v0 0
            lw    $t0 4($a0)
            lw    $t1 8($a0)
            beq   $t0 $t1 fin_2

            # apilar
            mul   $t2, $t1, 4
            lw    $t3 0($a0)
            add   $t3 $t3 $t2
            sw    $a1  0($t3)

            # menos libres
            add   $t1 $t1 1
            sw    $t1 8($a0)

            # fin
            li    $v0 1
      fin_2: j    $ra

      desapilar:
            # comprobar si vacia
            li    $v0 0 #!!!
```

```
        lw    $t0 4($a0)
        lw    $t1 8($a0)
        beq   $t0 $t1 fin_3

        # más libres
        sub   $t1 $t1 1
        sw    $t1 8($a0)

        # desapilar
        mul   $t1 $t1 4
        lw    $t2 0($a0)
        add   $t2 $t2 $t1
        lw    $v0 0($t2)

        # fin
fin_3:  j     $ra
```

Ejercicio 24. Escriba en ensamblador del MIPS R2000 una función que realice la factorización de un número entero. Esta función recibe en `$a0` el número entero a factorizar e imprime por pantalla todos los factores primos del número pasado como argumento. La factorización de un número supone encontrar la descomposición del mismo como producto de números primos. Por ejemplo:

Factorizar(15) = 5 * 3 * 1
Factorizar(30) = 5 * 3 * 2 * 1
Factorizar(72) = 3 * 3 * 2 * 2 * 2 * 1

Solución:

Para la resolución de este ejercicio se van a proponer dos alternativas: una solución iterativa y otra recursiva.

a) Solución iterativa:

```
Factorizar:


    # Empezamos a buscar factores por el 2, ya que el 1 siempre está
    # El número a factorizar se va dividiendo por los factores encontrados
    # hasta que valga 1
    # El número que vamos factorizando lo vamos a llevar en t0, luego
    # hacemos una copia
    # El número que se está probando lo vamos a llevar en t2,
    # inicialmente a 2

    move  $t0, $a0

    li    $t1, 1      # caso de parada
    li    $t2, 2

    siguiente_factor:
          beq   $t0, $t1, fin
          #Vemos si el numero es un factor del numero a factorizar
          rem   $t3, $t0,$t2
          beqz  $t3, es_divisor

          # Si t2 no es un factor habrá que ver el siguiente numero
          add   $t2, $t2, 1
          b     siguiente_factor

    es_divisor:
```

```
            # Si es un divisor lo imprimimos
            move  $a0, $t2
            li    $v0, 1
            syscall

            # y dividimos
            div   $t0, $t0, $t2

            # Buscamos el siguiente factor
            # No se incrementa, ya que el factor actual puede aparecer
            #varias veces
            b     siguiente_factor


      fin:
            # El uno siempre debe aparecer, luego lo imprimimos
            li    $v0, 1
            li    $a0, 1
            syscall


      jr $ra
```

b) Solución recursiva

```
      Factorizar:
      # Guardar en pila registros que se vayan a sobreescribir
      # Ya que se van a imprimir números al menos habrá que guardar a0 y v0
      sub   $sp, $sp, 4
      sw    $ra, ($sp)


      li    $t0, 1
      beq   $a0, $t0, cond_parada   # El 1 no se puede factorizar,
                                    # condición de parada
                                    # del algoritmo recursivo

      # Empezamos a  ver si un número es divisor de a0,
      # en cuanto encuentre uno
      # lo imprimo y llamo recursivamente. Empiezo en el 2.

      li    $t0, 2
      # Vemos si es divisor (si el modulo el número es 0,
# entonces es divisor)

      es_divisor:
            rem   $t1, $a0, $t0
            beqz  $t1, caso_recursivo
            # No es divisor, pruebo el siguiente número
            add   $t0, $t0, 1
            b     es_divisor

      caso_recursivo:
            # El numero encontrado es divisor, imprimo el divisor,
            # divido y factorizo el resto
            # Guardo a0, ya que vamos a imprimir y lo necesito
            # para hacer la llamada al sistema
            move  $t1, $a0
```

```
        #Imprimo el factor
        li    $v0, 1
        move  $a0, $t0
        syscall

        # Divido por el factor
        div   $a0, $t1, $t0

        # Sigo factorizando
        jal   Factorizar
        b     fin


    cond_parada:
        # Esto es el fin, el unico factor del 1 es el 1
        li    $v0, 1
        li    $a0, 1
        syscall

    fin:

    lw    $ra, ($sp)
    add   $sp, $sp, 4

    jr    $ra
```

Ejercicio 25. Escriba una función en ensamblador (utilizando el juego de instrucciones MIPS R2000) que ordene los elementos de un vector. Suponga que el registro `$a0` se pasa como parámetro la dirección de memoria donde empieza el vector y que en `$a1` se pasa la longitud del vector (número de palabras que ocupa). La función no devolverá ningún valor, el vector ordenado deberá almacenarse en la misma zona de memoria que el original, es decir, `$a0` debe seguir apuntando al comienzo del nuevo vector. Se permite la utilización de vectores auxiliares, pero recuerde que en ese caso debe reservar memoria para tal fin.

Solución:

```
# Ordena un vector de menor a mayor, no hay valores repetidos
# Implementa el algoritmo de insercción directa, que aunque
# no es el más sencillo permite ver varios ejemplos de bucles
# Para buscar el menor y mayor elemento, para copiar vectores
# y ademas muestra como reservar memoria.
# IN: a0 -> dir. de inicio del vector
#     a1 -> num. de elementos del vector

ordenar:
      ## Guardamos en pila aquellos registros que vamos a
      ## modificar y debe
      sub   $sp, $sp, 4
      sw    $a0, ($sp)
      sub   $sp, $sp, 4
      sw    $a1, ($sp)

      # t0 apunta al vector inicial
      move  $t0, $a0

      # Reservo memoria (vector auxiliar)
      li    $v0, 9
      move  $a0, $a1    # Calculo el numero de bytes a reservar
      sll   $a0, $a0, 2 # En a0 guardo el num de bytes (palabras * 4)
      syscall           # Los reservo
```

```
move  $t1, $v0    # Guardo el vector auxiliar en t1


# Busco el menor
li    $t3, 0            # Uso t3 como indice en el vector viejo
lw    $t6, ($t0)        # Guardo el resultado el t6

busca_menor:
      addu  $t3, $t3, 4 # Incremento el indice de bytes
      addu  $t2, $t0, $t3     # Calculo posicion respecto a la base
      lw    $t7, ($t2)        # Leo el siguiente valor

      bgt   $t7, $t6, busca_siguiente_menor
      move  $t6, $t7

busca_siguiente_menor:

      bne   $t3, $a0, busca_menor


# Busco el mayor
li    $t3, 0
lw    $t9, ($t0)
busca_mayor:
      addu  $t3, $t3, 4
      addu  $t2, $t0, $t3
      lw    $t7, ($t2)

      blt   $t7, $t9, busca_siguiente_mayor
      move  $t9, $t7

busca_siguiente_mayor:
      bne   $t3, $a0, busca_mayor


# En t6 tengo el menor
# Lo meto en la lista
# Bucle que busca el menor mayor que el ultimo menor
li    $t4, 0
move  $t8, $t6          # Inicio para guardar el menor
move  $t5, $t1          # que ya encontre
inserta_en_aux:         # Bucle que va insertando en vector aux
      sw    $t8, ($t5)        # Lo guardo
      addu  $t4, $t4, 4
      move  $t6, $t9
      li    $t3, 0

busca_mayor_2:    # Busco el menor que me quede
      addu  $t2, $t0, $t3
      lw    $t7, ($t2)

      ble   $t7, $t8, busca_siguiente_2
      bgt   $t7, $t6, busca_siguiente_2
      move  $t6, $t7

busca_siguiente_2:

      addu  $t3, $t3, 4
      bne   $t3, $a0, busca_mayor_2
      move  $t8, $t6
```

```
        # Almaceno el valor encontrado
        addu  $t5, $t1, $t4
        bne   $t4, $a0, inserta_en_aux


    # Copio la lista auxiliar a la lista vieja
    # Ya que dicen que el vector ordenado debe de
    # estar donde apunta a0 (a0 no se puede modificar)
    li    $t4, 0
    bucle_copia:
        add   $t2, $t1, $t4
        lw    $t2, ($t2)
        add   $t3, $t0, $t4
        sw    $t2, ($t3)
        add   $t4, $t4, 4
        blt   $t4, $a0, bucle_copia


    # Saco de pila los registros que he modificado y
    # quien llamo a la funcion puede estar utilizando
    lw    $a1, ($sp)
    add   $sp, $sp, 4
    lw    $a0, ($sp)
    add   $sp, $sp, 4

    # Retorno de la funcion
    jr    $ra
```

Ejercicio 26. Disponemos de un computador cuyo bus de datos tiene un tamaño de 16 bits, el tamaño de sus registros también es de 16 bits y que maneja una instrucción que dispone de tres campos: un código de operación de 5 bits, un campo para un registro de 3 bits y un campo para una dirección de 8 bits.

A la vista de esta información, se pide (Razone todas las respuestas):

a) ¿Cuál es el número máximo de instrucciones que puede tener este computador en su juego de instrucciones?
b) Número de registros de uso general que posee la máquina.
c) Si utilizamos direccionamiento directo, que cantidad de memoria es posible direccionar con este formato de instrucciones. Explique en que consiste el direccionamiento directo
d) Si utilizamos direccionamiento indirecto, que cantidad de memoria es posible direccionar con este formato de instrucciones. Explique en que consiste el direccionamiento indirecto.
e) ¿Cuál es el rango de direccionamiento con desplazamiento relativo a registro base para este formato de instrucción?

Solución:

a) Puesto que el código de operación que maneja este computador es de 5 bits, el número máximo de instrucciones que puede tener el juego de instrucciones del computador es: $2^5 = 32$.
b) El campo que posee la instrucción en el enunciado tiene tres bits para referirse a los registros de la máquina, luego podemos referenciar $2^3 = 8$ registros.
c) El campo que podemos utilizar para escribir una dirección de memoria tiene 8 bits, en el direccionamiento directo la dirección de memoria a la que queremos acceder ha de estar escrita en la propia instrucción, luego podemos acceder desde la dirección 0 a la dirección $(2^8 – 1) = 255$.
d) En el direccionamiento indirecto, el campo de la instrucción que contiene la dirección en realidad tiene la dirección de la posición de memoria donde se encuentra la dirección real del dato, luego cuando leamos en memoria el dato que se encuentra en el campo de dirección de la instrucción vamos a leer 16 bits que me indican realmente la dirección de memoria donde se encuentra el dato, por tanto en este caso es posible direccionar $2^{16} = 65536$ direcciones.
e) Como los registros son de 16 bits, con ellos se puede referenciar desde la posición 0 hasta 65535 (2^{16}-1), y como el campo de dirección es de 8 bits, se puede representar desde la posición 0 hasta 255 (2^8-1). Por tanto,

el rango de direccionamiento con desplazamiento con registro base es desde 0 (0+0) hasta 65790 (65535 + 255).

Ejercicio 27. Sea un computador de 32 bits y 16 registros, con una instrucción que tiene el siguiente formato:

- Código de operación de 6 bits.
- Un campo de indirección de 1 bit.
- Dos campos de 4 bits para representar un registro.
- Un campo de 17 bits para representar una dirección.

El campo Indirección de un bit indica si la Dirección presente en la instrucción es una dirección absoluta(0) o relativa(1). Se pide:

a) Indicar cuál es el número máximo de instrucciones que podrá tener dicho computador en su juego de instrucciones.
b) ¿Es posible utilizar modo de direccionamiento directo absoluto de memoria con este formato? Justificar la respuesta. Si lo es, indicar a cuál es el tamaño de la memoria direccionable con este formato e indicar qué campos de la instrucción estarían rellenos y los posibles valores.
c) ¿Se puede utilizar modo de direccionamiento directo relativo a registro? Justificar la respuesta. Si lo es, indicar cuál es el tamaño de la memoria direccionable con este formato e indicar qué campos de la instrucción estarían rellenos y los posibles valores.
d) ¿Se puede utilizar modo de direccionamiento indirecto? Justificar la respuesta. Si lo es, indicar cuál es el tamaño de la memoria direccionable con este formato e indicar qué campos de la instrucción estarían rellenos y los posibles valores.

Solución:

a) Puesto que el campo Código de Operación tiene 6 bits sería posible tener un juego de instrucciones de 2^6 instrucciones = 64.
b) Sí. En el direccionamiento directo absoluto de memoria la dirección a la que queremos acceder se encuentra en la propia instrucción, en este caso tenemos un campo de dirección con 17 bits, luego podríamos acceder con este tipo de direccionamiento a las direcciones de [0 , 2^{17}-1], los campos con valor serían: C.Op. con el código de Operación correspondiente, Bit de Indirección a 0, y campo de dirección con la dirección donde se encuentra el dato que queremos utilizar.
c) Sí. En el direccionamiento directo relativo a registro la dirección a la que queremos acceder se encuentra dentro del registro indicado en la propia instrucción, en este caso tenemos un campo de registro y por lo tanto podemos utilizar este direccionamiento. El número de direcciones a las que podremos acceder será todo el contenido de memoria, ya que la dirección a la que queremos acceder se encuentra dentro del registro, es decir, podemos acceder a las direcciones de [0 , 2^{32}-1], los campos con valor serían: C.Op. con el código de Operación correspondiente, Bit de Indirección a 0, y Reg. 1 con el número de registro que contiene la dirección a la que queremos acceder.
d) Sí. En el direccionamiento indirecto, la dirección a la que queremos acceder se encuentra dentro de memoria en la dirección indicada en la propia instrucción, luego tendríamos que acceder a memoria a la dirección indicada, allí obtendríamos la dirección donde se encuentra el dato y tendríamos que volver a acceder a memoria para acceder al dato. El número de direcciones a las que podremos acceder será todo el contenido de memoria, ya que la dirección a la que queremos acceder se encuentra dentro de una dirección de memoria que tendrá 32 bits, es decir, podemos acceder a las direcciones de [0 , 2^{32}-1]. Los campos con valor serían: C.Op. con el código de Operación correspondiente, Bit de Indirección a 1, y Dirección con la dirección donde se encuentra la dirección del dato que queremos obtener.

Ejercicio 28. La matriz `buscaminas_6` es capaz de representar el tablero 6x6 utilizado en el juego del buscaminas. Para ello, utiliza los dos primeros bytes de almacenamiento para representar la dimensión de la matriz (el primer byte indica el número de filas y el segundo byte indica el número de columnas). A continuación, utiliza tantas palabras de memoria de 32 bits como elementos contiene la matriz. Inicialmente, los elementos de la matriz `buscaminas_6` pueden contener el valor 0 (indicando que no existe mina en esa posición) o el valor 42 (código ASCII del carácter "*", indicando que existe una mina en esa posición). Un ejemplo de matriz buscaminas_6 es el siguiente:

```
.data
      buscaminas_6: .align 2
.byte 6,6
```

```
.word 0   0   0  42  42   0
.word 0   0   0   0   0   0
.word 42 42  42   0   0   0
.word 0   0   0  42  42  42
.word 0   0   0   0   0  42
.word 42 42  42  42  42  42
```

Se pide:

a) Declare en memoria los siguientes datos:
- Un array `minas_buscaminas6_filas` capaz de representar el número de minas en cada fila de la matriz `buscaminas_6`. Inicialmente, todos los elementos del array deberán contener el valor 0.
- Un número entero `total_minas` capaz de representar el número de minas existentes en la matriz `buscaminas_6`.

b) Implemente la función `obtenerMinasPorFilas`, que acepta dosos argumentos de entrada:
- La dirección de memoria de la matriz buscaminas_6.
- La dirección de memoria del array minas_buscaminas6_filas.

La función `obtenerMinasPorFilas` deberá almacenar en el array `minas_buscaminas6_filas`, cuya dirección es recibida como segundo argumento, el número de minas encontradas en cada fila de la matriz `buscaminas_6`, cuya dirección es recibida como primer argumento. Esta función devolverá como resultado el número de minas totales presentes en la matriz `buscaminas_6`.

c) Escriba el código necesario para invocar a la función anterior.

d) La función `apilarFilaColumna` recorre la matriz `buscaminas_6` en busca de minas. Cuando encuentra una mina, apila la fila y columna correspondiente a la posición donde se encontró la mina en la cima de la pila. Se pretende utilizar la función `apilarFilaColumna` para construir una copia exacta de la matriz `buscaminas_6`, etiquetada como `copia_6` (considere que ambas matrices se declaran en memoria de idéntica forma). Para ello, la función `construirMatrizBuscaminas` acepta los siguientes argumentos de entrada:
- La dirección de memoria de la matriz `copia_6`.
- El número de minas existentes en la matriz `buscaminas_6`.

La función `construirMatrizBuscaminas` desapila los datos apilados por la función `apilarFilaColumna` para insertar las minas en la posición análoga de la matriz destino `copia_6`, de tal manera que se obtenga una matriz idéntica a `buscaminas_6`. Esta función no devuelve ningún valor. Implemente el código de la función `construirMatrizBuscaminas`.

Solución:

a)

```
.data
minas_buscaminas6_filas:      .byte 0 0 0 0 0 0
            total_minas:      .byte 0
```

b)

```
obtenerMinasPorFilas:

      move  $s0 $a0     #$s0=$a0
      move  $s1 $a1     #$s1=$a1

      lb    $t1 ($s0)   #$t1=6,

      li    $t0 0       #$t0=0
      li    $v0 0       #$v0=0, Inicialmente numero de minas es 0

      addu  $s0 $s0 4
```

```
bucle_obtenerMinasPorFilas:
      beq   $t0 $t1 fin_bucle_obtenerMinasPorFilas

      li    $t2 0 #$t2=contador de columnas
      li    $t3 0 #$t3=contador de minas por filas

columnas:
      beq   $t2 $t1 fin_columnas    #for (j=0;i<filas;i++)

      lw    $t4 ($s0)   # Cargamos el valor en i,j

      beq   $t4 42      incrementar_minas
      b     no_incrementar_minas

incrementar_minas:
      addu  $t3 $t3 1         #$t3++

no_incrementar_minas:
      addu  $s0 $s0 4         #Siguiente elemento

      addu  $t2 $t2 1               #j++
      b     columnas

fin_columnas:
      sb    $t3 ($s1)
      addu  $s1 $s1 1               #$s1++
      addu  $v0 $v0 $t3

      addu  $t0 $t0 1               #i++
      b     bucle_obtenerMinasPorFilas

fin_bucle_obtenerMinasPorFilas:
      jr    $ra
```

c)

```
la    $a0 buscaminas_6
la    $a1 minas_buscaminas6_filas

jal   obtenerMinasPorFilas

sb    $v0 total_minas
```

d)

```
construirMatrizBuscaminas:
      li    $t0 0       #$t0=0, contador de iteraciones
      li    $t1 6       #$t1=6

      sb    $t1 ($a0)
      sb    $t1 1($a0)

      li    $t4 42            #$t4=42

Bs_construirMatrizBuscaminas:
      beq   $t0 $a1 Fin_Bs_construirMatrizBuscaminas

Bs_DesApilar:
      lw    $t2 4($sp)        #$t2=fila
      lw    $t3 8($sp)        #$t3=columna
```

```
            addu  $sp $sp 8

            #Obtener Dirección
            mul   $t2 $t2 6
            addu  $t2 $t2 $t3
            addu  $t2 $t2 1
            mul   $t2 $t2 4
            addu  $s0 $a0 $t2

            #Almacenar Mina
            sw    $t4 ($s0)

            #Incrementar número de iteraciones
            addu  $t0 $t0 1                          #j++

            b     Bs_construirMatrizBuscaminas

Fin_Bs_construirMatrizBuscaminas:
            jr    $ra
```

Ejercicio 29. Considere un computador de 32 bits con un formato de instrucciones como el siguiente:

Indicar de forma razonada:

a) ¿Cuántos registros tiene este computador?
b) ¿Cuántas instrucciones máquina es capaz de ejecutar?
c) Asumiendo que los valores inmediatos se expresan en complemento a 2, ¿Cuál es el rango de los números que se pueden expresar en las instrucciones de tipo 2?
d) Suponiendo que el computador es capaz de ejecutar instrucciones similares a las del MIPS, indique en qué tipo instrucción de las anteriores se podría codificar las siguientes instrucciones, mostrando en qué campo se almacena cada uno de los elementos de estas instrucciones.

```
lw  $t1, 80($t2)
jal 90000
li  $t1, 80
```

e) Indique los modos de direccionamiento presentes en las instrucciones anteriores.

Solución:

a) 2^6 = 64 registros.
b) 2^8 = 256 instrucciones.
c) $[-2^{11}..2^{11}-1]$
d) La instrucción `lw` en la instrucción de tipo 2; la instrucción `jal` en una instrucción de tipo 4; la instrucción `li` en una instrucción de tipo 2
e) En la instrucción `lw`: modo de direccionamiento de registro y direccionamiento relativo a registro base o registro índice; en la instrucción `jal` dirección directo; y en `li` dirección de registro y direccionamiento inmediato.

Ejercicio 30. Considere un computador de 64 bits que direcciona la memoria por bytes, con un banco de 256 registros y que dispone de un repertorio de instrucciones compuesto por 190 instrucciones. Dada las tres siguientes instrucciones:

- `LOAD Reg, direccion`, que almacena el contenido de una posición de memoria en el registro `Reg`.
- `STORE Reg1, (Reg2)`, que almacena en el registro `Reg1` el dato almacenado en la posición de memoria almacenada en `Reg2`.
- `BEQZ Reg1, direccion`, instrucción de salto condicional. Si el contenido del registro `Reg1` es cero, la siguiente instrucción a ejecutar será la almacenada en `direccion`.

Se pide:

a) Indique el modo o modos de direccionamiento presentes en cada una de las instrucciones anteriores.
b) Indique un posible formato para cada una de las instrucciones anteriores, asumiendo que en este computador todas las instrucciones ocupan una palabra.
c) De acuerdo al formato indicado en el apartado anterior, ¿cuál es el valor máximo de la dirección que se puede especificar en la instrucción `LOAD` y `BEQZ`?

Solución:

a)

- Instrucción `LOAD Reg, dirección`:
  - `Reg`: modo de direccionamiento de registro
  - `Direccion`: modo de direccionamiento directo
- Instrucción `STORE Reg1, (Reg2)`
  - `Reg1`: modo de direccionamiento de registro
  - `Reg2`: modo de direccionamiento indirecto de registro
- Instrucción `BEQZ Reg1, direccion`
  - `Reg1`: modo de direccionamiento de registro
  - `Direccion`: modo de direccionamiento directo

b)

Para la instrucción `LOAD` y `BEQZ`, el formato ocuparía 64 bits con tres campos:

- Código de operación de 8 bits.
- Campo para un registro de 8 bits.
- Campo para la dirección de 48 bits.

Para la instrucción `STORE`, el formato sería una palabra de 64 bits con los siguientes campos:

- Código de operación de 8 bits.
- Dos campos para los registros de 8 bits cada uno
- Un campo de 40 bits si utilizar.

c) Como se utilizan 48 bits para la dirección, el máximo valor que se podrá direccional es 2^{48} -1.

Ejercicio 31. Considere un computador de 32 bits que direcciona la memoria por bytes, con un banco de 32 registros y que dispone de un repertorio de instrucciones compuesto por 158 instrucciones. Esta máquina utiliza las siguientes convenciones sintácticas:

- Los valores inmediatos se representan indicando directamente su valor en la instrucción.
- Las direcciones absolutas a memoria se identifican así: `/dirección`
  Ejemplo: `LOAD R1,/300`, guarda el contenido de la dirección de memoria 300 en el registro `R1`.
- El direccionamiento relativo a registro se indica como: `numero(registro)`
  Ejemplo: `STORE R1, 100(R2)`, guarda el contenido de la dirección de memoria apuntada por `R2+100` en la posición de memoria apuntada por `R1`.
- El direccionamiento indirecto se indica `@RX`

Ejemplo: `BEQZ R1, @R2`, si el contenido del registro 1 es 0, se salta a la dirección de memoria que está guardada en la dirección de memoria que contiene el registro 2.

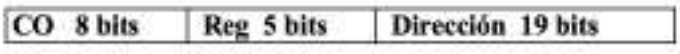
Se pide:

a) Indique un posible formato para la instrucción LOAD Reg1,/3000, asumiendo que en este computador todas las instrucciones ocupan una palabra.
b) De acuerdo al formato indicado en el apartado anterior, ¿cuál es el valor máximo de la dirección que se puede especificar en la instrucción LOAD?
c) Suponiendo que el contenido inicial de los registros es 0 y que la memoria del computador está inicialmente vacía, después de ejecutarse la siguiente secuencia de instrucciones indique el contenido de los registros que se hayan modificado y el contenido de las posiciones de memoria que hayan cambiado.

```
MOVE  R1,   1000
STORE R1,   10
LOAD  R2,   1000(R3)
STORE R2,   50
MOVE  R4,   R2
ADD   R4,   R1
STORE R4,   /1000
LOAD  R5,   @R4
```

Solución:

a) El formato de la instrucción sería:



b) Como se utilizan 19 bits para la dirección, el máximo valor que se podrá direccionar es $2^{19} - 1$.

c) El contenido final de los registros será:

```
R1:   1000
R2:   10
R3:   0
R4:   1010
R5:   50
```

Las direcciones de memoria que se modifican y sus contenidos serán:

```
10:   50
1000: 10
1010: 10
```

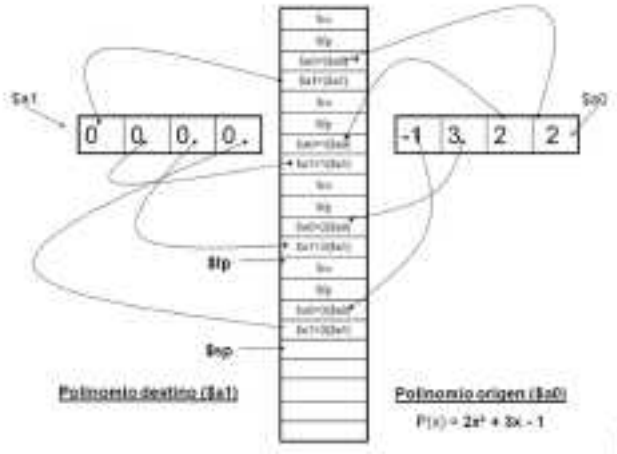
Ejercicio 32. La función recursiva `plxCopiaPila` toma los siguientes tres argumentos de entrada:

- En el primer argumento se pasará la dirección de memoria del polinomio origen que se desea copiar.
- En el segundo argumento se pasará la dirección de la memoria reservada estáticamente para almacenar la copia del polinomio origen.
- En el tercer argumento se pasará el grado del polinomio origen a copiar.

Suponga que todos los elementos del polinomio son de tipo byte.Para realizar esta copia se llamará recursivamente a la función `plxCopiaPila`, utilizando para ello la pila de la siguiente manera:

- La función `plxCopiaPila` deberá copiar completamente el polinomio recibido como primer argumento en la dirección de memoria recibida como segundo argumento, desde el último elemento (término independiente) hasta el primero (grado del polinomio) utilizando llamadas recursivas.
- La función `plxCopiaPila` invocada desde el programa principal recibe los argumentos en los registros `$a0`, `$a1` y `$a2`, como es habitual.

La siguiente figura muestra un ejemplo del estado de la pila inmediatamente antes de proceder a realizar el primer *backtracking*, siendo el polinomio origen a copiar 2x2 + 3x – 1.



Se pide:

a) Definición y declaración en memoria de los polinomios origen y destino de la función, así como el grado del polinomio origen.
b) Implementar usando el ensamblador de MIPS R2000 la sección de código que desde la función principal realiza el paso de argumentos e invocación a la función `plxCopiaPila`.
c) Implementar usando el ensamblador de MIPS R2000 la función recursiva `plxCopiaPila`.

Solución:

a)

```
.data
      polinomio_origen:       .byte 4            #Grado del polinomio
                              .byte 4 -3 2 -1 0  #4x4 – 3x3 + 2x2 –x
      polinomio_destino:      .space 20

      grado:                  .byte 4
```

b)

```
.text
      .globl main

main:
      la $a0 polinomio_origen
      la $a1 polinomio_destino
      lb $a2 grado

      jal plxCopiaPila
```

c)

```
plxCopiaPila:
      #PRIMER PASO: Criterio de parada
      blt   $a2 -1 fin_recursividad

      #SEGUNDO PASO: Marco de pila
      subu  $sp $sp 16
      sw    $ra 16($sp)
      sw    $fp 12($sp)
      sw    $a0 8($sp)
      sw    $a1 4($sp)
      addu  $fp $sp 16

      #TERCER PASO: Actualizar argumentos
recursividad:
```

```
        addu  $a1 $a1 1
        addu  $a0 $a0 1
        addi  $a2 $a2 -1        #$a2=$a2-1

        #CUARTO PASO: Llamada recursiva
        jal    plxCopiaPila

fin_recursividad:  #Backracking
        #QUINTO PASO: Restaurar valores $
        lw    $a0 8($sp)  #$t0<--($fp)
        lw    $a1 4($sp)  #Almacenar el elemento en $a1
        lw    $ra 16($sp)
        lw    $fp 12($sp)
        lb    $t1 0($a0)
        sb    $t1 0($a1)
        addu  $sp $sp 16  #$sp=$sp+16 ()

        jr    $ra
```

Ejercicio 33. Implementar la rutina `strcut` que elimina los caracteres que se encuentran entre la posición inicial y final (ambas inclusive) de una cadena.

* En el registro `$a0` se pasará la dirección de memoria de la cadena origen.
* En el registro `$a1` se pasará la posición inicial.
* En el registro `$a2` se pasará la posición final.

La cadena resultado se devuelve en `$v0` y debe apuntar a la misma dirección que la cadena origen. Si la posición final es menor que la posición inicial, la función no hace nada. Si la dirección de la cadena es puntero nulo o la cadena es vacía, la función devuelve puntero nulo o cadena vacía, respectivamente. Si la posición final es mayor que la posición del último carácter se deberá hacer el corte hasta el fin de la cadena. Si la posición inicial o final fueran números negativos no se realizará acción alguna.
Ejemplos de llamadas a `strcut ($a0,$a1,$a2) =  $v0`

```
1-strcut (0,x,y) = 0  (0 se refiere a puntero nulo, x e y pueden tener cualquier
valor)
2-strcut ("",x,y) = "" (x e y pueden tener cualquier valor)
3-strcut ("cadena",3,5) = "cad"
4-strcut ("cadena",0,10) = ""
5-strcut ("cadena",3,3) = "cadna"
6-strcut ("cadena",5,5) = "caden"
7-strcut ("cadena",5,10) = "caden"
```

Solución:

```
strcut:
      #casos base

      beqz  $a0 end_cut #puntero nulo
      move  $t0 $a0           #cadena vacía
      lb    $t4 ($a0)
      beqz  $t4 end_cut
      blt   $a2 $a1 end_cut   #posición final < posición inicial, no cortar
      bltz  $a0 end_cut #posición inicial o posición final negativos
      bltz  $a1 end_cut

      move  $t1 $a1     #punto donde se empieza a cortar
      move  $t2 $a2     #punto donde terminar de cortar (y empezar a copiar)

      #situar $t0 en la posición donde comenzar a cortar
```

```
      #NOTA: Se pasa de posición en posición de la cadena hasta encontrar
      #la posición inicial
      #No se puede saltar directamente a la dirección pues no conocemos
      #la longitud de la cadena  (y no hay control de los índices)
      #y se podría terminar referenciando una posición que no es de la
      #cadena que se pasa como parámetro

first:
      blez  $t1 end
      addi  $t0 $t0 1
      lb    $t4 ($t0)
      beqz  $t4 end_cut
      addi  $t1 $t1 -1
      addi  $t2 $t2 -1
      b     first

end:  move  $t5 $t0
      #situar $t5 en la posición donde terminar de cortar y empezar a copiar

endi: blez  $t2 cut1
      addi  $t5 $t5 1
      lb    $t4 ($t5)
      beqz  $t4 cut2
      addi  $t2 $t2 -1
      b     endi

      #NOTA: Se ha de diferenciar entre que se haya llegado a final de
      #cadena (saltar a cut2) o no se haya llegado (saltar a cut1)
      #debido a que no hay control de índices y el índice final puede
      #ser superior a la longitud de la cadena

cut1: addi  $t5 $t5 1   #sumar  1 porque empezamos a copiar el
                        #siguiente carácter (si existe, si no, se habrá
                        #saltado a cut2)
      #copiar contenido de $t5 a $t0 hasta llegar $t5 a fin de cadena
cut2: lb    $t7 ($t5)   #mover datos de la posición anterior a la nueva
      sb    $t7 ($t0)
      sb    $0 ($t5)    #borrar el contenido que vamos copiando a su nueva
                        #posición (paso opcional)
      addi  $t0 $t0 1         #incrementar contadores
      addi  $t5 $t5 1
      beqz  $t7 end_cut
      b     cut2

end_cut:
      move  $v0 $a0
      jr    $ra
```