EVALUACION DE EXPRESIONES

El lenguaje ensamblador no dispone de estructuras de control de flujo de programa definidas, que permitan decidir entre dos (o varios) caminos de ejecución de instrucciones distintos (por ejemplo la sentencia *if* del lenguaje de programación C). Normalmente para implementar cualquier estructura de este tipo es necesario evaluar previamente una condición, simple o compuesta. El camino que seguirá la ejecución del programa dependerá del resultado de esta evaluación.

Se describe como se evalúan las condiciones en ensamblador del MIPS R2000. En primer lugar se realiza un breve repaso al conjunto de instrucciones y pseudoinstrucciones que tiene MIPS R2000 para realizar comparaciones y control del flujo de programa. A continuación, se proponen varios ejemplos que implementan fragmentos de código para evaluar diversas condiciones simples y compuestas. IMPORTANTE: estos fragmentos son una forma, no la única correcta.

Instrucciones de salto condicional del MIPS R2000

El ensamblador del MIPS incluye dos instrucciones básicas de toma de decisiones `beq` y `bne`, que permiten implementar estructuras de control muy sencillas. La sintaxis de estas instrucciones es la siguiente:

```
beq rs,rt,etiqueta

bne rs,rt,etiqueta
```

Ambas comparan el contenido de los registros `rs` y `rt` y según el resultado de esta comparación (cierta o falsa), saltan a la dirección de la instrucción que referencia `etiqueta` o no. El resultado de la evaluación es cierto si el contenido del registro `rs` es igual al del `rt` (instrucción `beq`), o distinto (instrucción `bne`), falsa en caso contrario para cada una de las instrucciones.

También dispone de instrucciones de salto condicional para realizar comparaciones con cero. Estas instrucciones son: `bgez, bgtz, blez, bltz`, y tienen la siguiente sintaxis:

```
bgez rs,etiqueta

bgtz rs,etiqueta

blez rs,etiqueta

bltz rs,etiqueta
```

Todas ellas comparan el contenido del registro rs con 0 y saltan a la dirección de la instrucción referenciada por `etiqueta` si rs>=0 (`bgez`), rs>0 (`bgtz`), rs<=0 (`blez`) y rs<0 (`bltz`).

Pseudoinstrucciones de salto condicional

El lenguaje ensamblador del MIPS R2000 dispone de un conjunto de pseudoinstrucciones de salto condicional que permiten comparar dos variables almacenadas en registros (a nivel de mayor, mayor o igual, menor, menor o igual) y, según el resultado de esa comparación, saltan o no, a la instrucción que referencia etiqueta. Estas pseudoinstrucciones son: `bge` (branch if greater or equal, saltar si mayor o igual), `bgt` (branch if greater, saltar si mayor que), `ble` (branch if less or equal, saltar si menor o igual), `blt` (branch if less, saltar si menor que). El formato es el mismo para todas ellas y coincide con el de las instrucciones de salto descritas, `beq` y `bne`:

```
bxx rs,rt,etiqueta
```

donde xx es `ge` (saltar si rs es mayor o igual que rt), `gt` (saltar si rs es mayor que rt), `le` (saltar si rs es menor o igual que), `lt` (saltar si rs es menor que).

Instrucciones de comparación

El lenguaje máquina y ensamblador del MIPS dispone de una instrucción que compara dos registros y carga un 1 ó un 0 en un tercer registro dependiendo del resultado de la comparación. Si el contenido del primer registro es menor que el segundo carga un 1, en caso contrario carga un 0. Ésta es la instrucción `slt` y tiene la siguiente sintaxis:

```
slt rd,rs,rt
```

La nomenclatura que se va a seguir a lo largo de este capítulo para describir el resultado que proporciona la ejecución de una instrucción de este tipo (evaluación de una condición) es el siguiente:

```
rd(1) ← (rs < rt),
```

indicando que el registro rd se pondrá a 1 si el contenido del registro rs es menor que rt y se pondrá a 0 en caso contrario.

Pseudoinstrucciones de comparación

El ensamblador del MIPS permite utilizar un conjunto de pseudoinstrucciones que facilitan la evaluación de condiciones. Estas pseudoinstrucciones realizan comparaciones de dos variables a otros niveles (como mayor, mayor o igual, menor o igual, igual, distinto) contenidas en sendos registros y almacenan el resultado de la comparación en un tercer registro (poniendo un 1 si la condición es cierta, y un 0 en caso contrario). Estas pseudoinstrucciones tienen la misma sintaxis que la instrucción

`slt` descrita. Estas son: `sge` (poner 1 si mayor o igual), `sgt` (poner 1 si mayor), `sle` (poner 1 si menor o igual), `sne` (poner a 1 si distinto), `seq` (poner a 1 si igual).

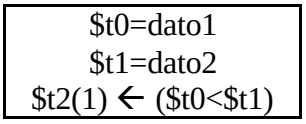
A continuación se estudia como se evalúan condiciones simples y compuestas por los operadores "and" y "or".

Evaluación de condiciones simples: menor, mayor, menor o igual, mayor o igual

Crea un fichero con el siguiente código que compara las variables `dato1` y `dato2` y deja el resultado en la variable booleana `res`:

```
          .data
dato1:    .word    30
dato2:    .word    40
res:      .space   1
          .text
main:     lw $t0,dato1($0) # cargar dato1 en t0
          lw $t1,dato2($0) # cargar dato2 en t1
          slt $t2,$t0, $t1 # poner a 1 $t2 si t0<t1
          sb $t2,res($0)   # almacenar $t2 en res
```

Borra los valores de la memoria, carga el fichero en el simulador y ejecútalo. El diagrama de flujo asociado a este fragmento de código, que puede ayudar a analizarlo es el siguiente:



Cuestión 1.1: ¿Qué valor se carga en la posición de memoria res?

Inicializa las posiciones de memoria `dato1` y `dato2` con los valores 50 y 20 respectivamente. Ejecuta de nuevo el programa

Cuestión 1.2: ¿Qué valor se carga ahora en la posición de memoria `res`?

Cuestión 1.3: ¿Qué comparación se ha evaluado entre `dato1` y `dato2`?

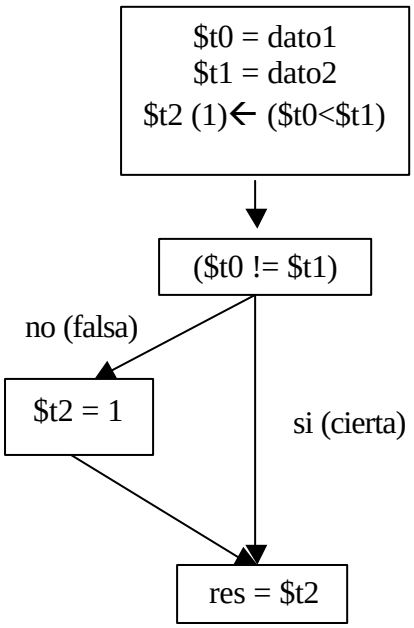
Cuestión 1.4: Modifica el código anterior para evaluar la siguiente condición `res`(1)←(`dato1 = dato2`).

Cuestión 1.5: Resuelve la cuestión anterior utilizando la pseudoinstrucción `sge`.

Crea un fichero con el siguiente código que compara las variables `dato1` y `dato2` y deja el resultado en la variable booleana `res`:

```
        .data
dato1:    .word    30
dato2:    .word    40
res:      .space   1
        .text
main:  lw $t0,dato1($0)   # cargar dato1 en t0
       lw $t1, dato2($0)  # cargar dato2 en t1
       slt $t2, $t0, $t1  # poner a 1 $t2 si t0<t1
       bne $t0,$t1,fineval # si t0<>t1 salta a fineval
       ori $t2,$0,1       # poner a 1 t3 si t0=t1
fineval: sb $t2,res($0)    # almacenar $t2 en res
```

Borra los valores de la memoria, carga el fichero en el simulador y ejecútalo. El diagrama de flujo asociado al fragmento de código anterior, que puede ayudar a analizarlo, es el que se muestra en la siguiente figura:



Cuestión 1.6: ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 1.7: Inicializa las posiciones de memoria `dato1` y `dato2` con los valores 50 y 20, respectivamente. Ejecuta de nuevo el programa, ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 1.8: Inicializa las posiciones de memoria `dato1` y `dato2` con los valores 20 y 20, respectivamente. Ejecuta de nuevo el programa, ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 1.9: ¿Qué comparación se ha evaluado entre `dato1` y `dato2`?

Cuestión 1.10: Evalúa esta comparación utilizando pseudoinstrucciones.

Cuestión 1.11: Modifica el código anterior para que la condición evaluada sea `res(1) ← (dato1 >= dato2)`. No utilices pseudoinstrucciones

Cuestión 1.12: Modifica el código anterior utilizando pseudoinstrucciones.

Evaluación de condiciones compuestas por operadores lógicos "and" y "or".

Crea un fichero con el siguiente código que compara las variables `dato1` y `dato2` y deja el resultado de la comparación en la variable `res`:

```
          .data
dato1:    .word    40
dato2:    .word    -50
res       .space   1
          .text
main:     lw $t8,dato1($0)
          lw $t9,dato2($0)
          and $t0,$t0,$0
          and $t1,$t1,$0
          beq $t8,$0,igual
          ori $t0,$0,1
igual:    beq $t9,$0,fineval
          ori $t1,$0,1
fineval:  and $t0,$t0,$t1
          sb $t0,res($0)
```

Borra los valores de la memoria, carga el fichero en el simulador y ejecútalo.

Cuestión 1.13: ¿Qué valor se carga en la posición de memoria `res`? Para ayudarte en el análisis del código, dibuja el diagrama de flujo de éste.

Cuestión 1.14: Inicializa `dato1` y `dato2` con los valores 0 y 20, respectivamente. ¿Qué valor se carga en la posición de memoria res?

Cuestión 1.15: Inicializa `dato1` y `dato2` con los valores 20 y 0, respectivamente. ¿Qué valor se carga en la posición de memoria res?

Cuestión 1.16: Inicializa `dato1` y `dato2` con los valores 0 y 0, respectivamente. ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 1.17: ¿Qué comparación compuesta se ha evaluado entre `dato1` y `dato2`?

Cuestión 1.18: Modifica el código anterior para que la condición evaluada sea: `res` (1) ← ((`dato1` <> 0) and (`dato1` <> `dato2`)).

Crea el siguiente fichero con la extensión .s:

```
          .data
dato1     .word     30
dato2     .word     -50
res       .space    1
          .text
main:     lw $t8,dato1($0)
          lw $t9,dato2($0)
          and $t1,$t1,$0
          and $t0,$t0,$0
          beq $t8,$0,igual
          ori $t0,$0,1
igual:    slt $t1,$t9,$t8
fineval:  and $t0,$t0,$t1
          sb $t0,res($0)
```

Borra los valores de la memoria, carga el fichero en el simulador y ejecutalo.

Cuestión 1.19: ¿Qué valor se carga en la posición de memoria res? Si te sirve de ayuda dibuja el diagrama de flujo asociado al código que se quiere analizar.

Cuestión 1.20: Inicializa `dato1` y `dato2` con los valores 10 y 20, ejecútalo ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 1.21: Inicializa `dato1` y `dato2` con los valores 0 y -20, ejecútalo ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 1.22: ¿Qué comparación compuesta se ha evaluado?

Cuestión 1.23: Modifica el código anterior para que la condición evaluada sea `res` (1) ← ((`dato1` <> `dato2`) and (`dato1` <= `dato2`)).

Cuestión 1.24: Modifica el código anterior utilizando la pseudoinstrucción `sle`.

Crea el siguiente fichero con la extensión .s que compara las variables `dato1` y `dato2` y deja el resultado de la comparación en la variable `res`:

```
          .data
dato1:    .word    30
dato2:    .word    -20
res       .space   1
          .text
main:     lw $t8,dato1($0)
          lw $t9,dato2($0)
          and $t0,$t0,$0
          and $t1,$t1,$0
          slt $t0,$t8,$t9
          bne $t9,$0,fineval
          ori $t1,$0,1
fineval:  or $t0,$t0,$t1
          sb $t0,res($0)
```

Borra los valores de la memoria, carga el fichero en el simulador y ejecútalo.

Cuestión 1.25: ¿Qué valor se carga en la posición de memoria `res`? Si te sirve de ayuda dibuja el diagrama de flujo asociado al código que se quiere analizar.

Cuestión 1.26: Inicializa `dato1` y `dato2` con los valores -20 y 10, respectivamente, ejecuta de nuevo el código, ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 1.27: Inicializa `dato1` y `dato2` con los valores 10 y 0, respectivamente, ejecuta de nuevo el código, ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 1.28: Inicializa `dato1` y `dato2` conlos valores 20 y 10, respectivamente, ejecuta de nuevo el código, ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 1.29: ¿Qué comparación compuesta se ha realizado?

Cuestión 1.30: Modifica el código anterior para que la condición evaluada sea `res`(1)← ((`dato1` <= `dato2`) or (`dato1` <= 0)).

Cuestión 1.31: Modifica el código anterior utilizando la pseudoinstrucción `sle`.

Problemas propuestos

1. Diseña un programa en ensamblador que defina un vector de booleanos, `V`. Este se implementa a partir de un vector de bytes donde cada byte sólo puede tomar dos valores, 0 ó 1, para el valor cierto o falso, respectivamente. V se inicializará con los siguientes valores V=[0,1,1,1,0]. El programa obtendrá otro vector de booleanos, `res`, de tres elementos tal que

   `res`[1]= (V[1] and V[5]),

   `res`[2]=(V[2] or V[4]) ,

   `res`[3]=((V[1] or V[2]) and V[3]).

2. Diseña un programa en ensamblador que defina un vector de enteros, `V`, inicializado según los siguientes valores (V=[2, -4, -6]). Y obtenga un vector de booleanos, tal que cada elemento será 1 si el correspondiente elemento en el vector de enteros es mayor o igual que cero y 0 en caso contrario.

3. Diseña un programa en ensamblador que defina un vector de enteros, `V`, inicializado a los siguientes valores `V`=[1, -4, -5, 2] y obtenga como resultado una variable booleana que será 1 si todos los elementos de este vector son menores que cero.

CONTROL DE FLUJO CONDICIONAL

Una vez introducidas el conjunto de instrucciones y pseudoinstrucciones que permiten implementar cualquier estructura de control de flujo de programa, se va a describir cómo se implementan las estructuras típicas de un lenguaje de alto nivel, como *if then, if then else* o estructuras repetitivas como *while* , y *for* del lenguaje C. Estas estructuras dependen, implícita o explícitamente, de la verificación de una o varias condiciones para determinar el camino que seguirá la ejecución del código en curso. La evaluación de esta condición (o condiciones) vendrá asociada a una o varias instrucciones de salto condicional e incondicional o pseudoinstrucciones.

Estructura de control *if - then* con condición simple

Crea un fichero con el siguiente fragmento de código que implementa una estructura de control condicional *if-then* :

```
          .data
dato1:    .word    40
dato2:    .word    30
res:      .space   4
          .text
main:     lw  $t0,dato1($0)  #cargar dato1 en $t0
          lw  $t1,dato2($0)  #cargar dato2 en $t1
          and $t2,$t2,$0     #t2=0
Si:       beq $t1,$0,finsi   #si $t1 = 0 finsi
entonces: div $t0,$t1        #t0/$t1
          mflo $t2           #almacenar LO en $t2
finsi:    add $t3,$t0,$t1    #$t3=$t0+$t1
          add $t2,$t3,$t2    #$t2=$t3+$t2
          sw  $t2,res($0)    #almacenar en res $t2
```

El pseudocódigo de este programa en ensamblador se muestra a continuación. Esta descripción es muy cercana al código anterior, utilizando registros para almacenar temporalmente el contenido de las variables en memoria:

```
VARIABLES
     ENTEROS: dato1=40; dato2=30; res;
INICIO
     $t0=dato1;
     $t1=dato2;
     $t2=0;
     if ($t1!=0) $t2=$t0/$t1;
     $t3=$t0+$t1;
     $t2=$t2+$t3;
     res=$t2;
FIN
```

Esta transcripción casi directa del programa en ensamblador, donde se tienen que utilizar registros del procesador para almacenar temporalmente las variables en memoria es debida a que se trata de un procesador basado en una arquitectura de carga y almacenamiento.

Una descripción de más alto nivel pasaría por no mencionar los registros

```
VARIABLES
     ENTERO: dato1=40; dato2=30; res;
INICIO
     if (dato2!=0) res=dato1/dato2;
     res=res+dato1+dato2;
FIN
```

Borra los valores de la memoria y carga el fichero que contiene el programa en ensamblador en el simulador.

Cuestión 2.1: Identifica la instrucción que evalúa la condición y controla el flujo de programa. Compárala con la condición del programa descrito en pseudocódigo.

Cuestión 2.2: Identifica el conjunto de instrucciones que implementan la estructura condicional *if-then*

Cuestión 2.3: ¿Qué valor se almacena en la variable `res` después de ejecutar el programa?

Cuestión 2.4: Si `dato2` es igual 0 ¿Qué valor se almacena en la variable `res` después de ejecutar el programa? Dibuja el diagrama de flujo asociado a la estructura de control implementada en el fragmento de código anterior.

Cuestión 2.5: Implementar el siguiente programa descrito en pseudocódigo

```
VARIABLES
     ENTERO: dato1=40; dato2=30; res;
INICIO
     Si (dato2>0) res=dato1/dato2;
     res=res+dato1+dato2;
FIN
```

Estructura de control *if-then* con condición compuesta

Crea un fichero con el siguiente fragmento de código que implementa una estructura de control condicional *if then*:

```
          .data
dato1:    .word     40
dato2:    .word     30
res:      .space    4
```

```
          .text
main:     lw  $t0,dato1($0)  #cargar dato1 en t0
          lw  $t1,dato2($0)  #cargar dato2 en $t1
          and $t2,$t2,$0     #pone a 0 $t2
Si:       beq $t1,$0,finsi   #si $t1=0 saltar finsi
          beq $t0,$0,finsi #si $t0 =0 saltar finsi
entonces: div $t0,$t1        #t0/$t1
          mflo $t2           #almacenar LO en t2
finsi:    add $t3,$t0,$t1    #t3=t0+$t1
          add $t2,$t3,$t2    #$t2=t3+$t2
          sw  $t2,res($0)    #almacenar en res $t2
```

Borra los valores de la memoria y carga el fichero en el simulador.

Cuestión 2.6: Elabora el pseudocógido sin mencionar registros

Cuestión 2.7: Identifica la (las) instrucción(es) que evalúa(n) la condición y controla(n) el flujo de programa y compárala(s) con la condición del programa descrito en pseudocódigo.

Cuestión 2.8: Identifica el conjunto de instrucciones que implementan la estructura condicional *if-then* . Dibuja el diagrama de flujo asociado con esta estructura de control.

Cuestión 2.9: Al ajecutar el programa ¿Que se almacena en la variable `res`?

Cuestión 2.10: Si `dato1=0`, ¿Que valor se almacena en la variable `res` después de ejecutar el programa? Si `dato2=0` ¿Que valor se almacena en la variable `res`?

Cuestión 2.11: Implementa el siguiente programa descrito en pseudocódigo:

```
VARIABLES
     ENTERO dato1=40; dato2=30; res;
INICIO
     Si ((dato1>0) and (dato2>=0)) res=dato1/dato2;
     res=res+dato1+dato2;
FIN
```

Estructura de control *if-then-else* con condición simple.

Crea un fichero con el siguiente fragmento de código que implementa una estructura de control *if-then-else* .

```
          .data
dato1:    .word     30
dato2:    .word     40
```

```
res:      .space   4
          .text
main:     lw  $t0,dato1($0)  #cargar dato1 en $t0
          lw  $t1,dato2($0)  #cargar dato2 en $t1
Si:       bge $t0,$t1, sino  #si $t0>=$t1 ir a sino
entonces: sw  $t0,res($0)    #almacenar $t0 en res
          j finsi            #ir a finsi
sino:     sw $t1,res($0)#almacenar $t1 en res
finsi:
```

Borra los valores de la memoria, carga el fichero en el simulador y ejecútalo.

Cuestión 2.12: Describe en lenguaje algorítmico el equivalente a este programa en ensamblador.

Cuestión 2.13: ¿Qué valor se almacena en `res` después de ejecutar el programa? Si `dato1=35`, ¿qué valor se almacena en `res` después de ejecutar el programa?

Cuestión 2.14: Identifica en el lenguaje máquina generado por el simulador el conjunto de instrucciones que implementan la pseudoinstrucción `bge`.

Cuestión 2.15: Implementa en ensamblador el siguiente programa descrito en lenguaje algorítmico:

```
VARIABLES
     ENTERO: dato1=30; dato2=40; res;
INICIO
     Si ((dato1>=dato2)) entonces
          res=dato1-dato2;
     Sino
          res=dato2-dato1;
     FinSi
FIN
```

Estructura de control *if-then-else* con condición compuesta

Crea un fichero con el siguiente fragmento de código que implementa una estructura de control *if-then-else* .

```
          .data
dato1:    .word    30
dato2:    .word     40
dato3:    .word     -1
res:      .space   4
          .text


main:     lw $t1,dato1($0)   #cargar dato1 en $t1
          lw $t2,dato2($0)   #cargar dato2 en $t2
          lw $t3,dato3($0)   #cargar dato3 en $t3


Si:       blt $t3,$t1, entonces #si $t3<$t1 ir entonces
          ble $t3,$t2, sino       #si $t3<=$t2 ir a sino
entonces: addi $t4,$0,1           #$t4=1
          j finsi                 #ir a finsi
sino:     and  $t4,$0,$0          #$t4=0
finsi:    sw   $t4,res($0)        #almacenar res
```

Borra los valores de la memoria, carga el fichero en el simulador.

Cuestión 2.16: Describe en lenguaje algorítmico el equivalente a este programa en ensamblador.

Cuestión 2.17: ¿Qué valor se almacena en `res` después de ejecutar el programa? Si `dato1=40` y `dato2=30`, ¿qué valor se almacena en `res` después de ejecutar el programa?

Cuestión 2.18: Implementa en ensamblador el siguiente programa descrito en lenguaje algorítmico:

```
VARIABLES
     ENTERO: dato1=30; dato2=40; dato3=-1; res;
INICIO
     Si ((dato3>=dato1) AND (dato3<=dato2)) entonces
          res=1;
     Sino
          res=0;
     FinSi
FIN
```

Estructura de control repetitiva *while*

Crea un fichero con el siguiente fragmento de código que implementa una estructura de control repetitiva *while* :

```
          .data
cadena:   .asciiz  "hola"
          .align   2
n:        .space   4
          .text
main:     la $t0,cadena #carga dir. cadena en $t0
          andi $t2,$t2, 0    #$t2=0
mientras: lb   $t1,0($t0)    #almacenar byte en $t1
          beq  $t1,$0,finmientras #si $t1=0 saltar a
                                  #finmientras
          addi $t2,$t2, 1    #$t2=$t2+1
          addi $t0,$t0, 1    #$t0=$t0+1
          j    mientras      #saltar a mientras
finmientras:  sw $t2,n($0)       #almacenar $t2 en n
```

La descripción algorítmica de este programa en ensamblador se muestra a continuación:

```
VARIABLES
     VECTOR DE CARACTERES: cadena(4)='hola';
     ENTERO: n;
INICIO
     i=0;
     n=0;
     Mientras (cadena[i]!=0) hacer
          n=n+1;
          i=i+1;
     FinMientras
FIN
```

Borra los valores de la memoria y carga el fichero que contiene el programa en ensamblador en el simulador.

Cuestión 2.19: Ejecuta paso a paso el programa anterior y comprueba detenidamente la función de cada una de las instrucciones que constituyen el programa ensamblador.

Cuestión 2.20: ¿Qué valor se almacena en n después de ejecutar el programa?

Cuestión 2.21: Implementa en ensamblador el siguiente programa descrito en lenguaje algorítmico

```
VARIABLES
     VECTOR DE CARACTERES:
          tira1(4)="hola"; tira2(5)="adios";
     ENTERO: n;
INICIO
     i=0;
     n=0;
Mientras ((tira1[i])!=0) and (tira2[i]!=0)) hacer
          n=n+1;
          i=i+1;
     FinMientras
FIN
```

Estructura de control repetitiva *for*

Crea un fichero con el siguiente fragmento de código que implementa una estructura de control repetitiva *for* . Ésta es una estructura de control donde el bucle se repite un número de veces conocido a priori. Para implementarla se necesita pues un contador (inicializado a 0 ó al máximo número de veces menos uno que queremos que se repita el bluce) que llevará la cuenta de las veces que éste se repite. Este contador se deberá incrementar o decrementar dentro del bucle. La condición de salida del bucle siempre será comprobar si este contador alcanza la cota superior o inferior (0) (según el contador cuente de forma ascendente o descendente).

```
        .data
vector    .word     6,7,8,9,10,1
res       .space    4
        .text
main:  la $t2,vector      #$t2=dirección de vector
       and $t3,$0,$t3     #$t3=0
       li $t0,0           #$t0=0
       li $t1,6           #$t1=5
para:  bgt $t0,$t1,finpara #si $t0>$t1 saltar finpara
       lw $t4,0($t2) #carga elemento vector en $t4
       add $t3,$t4, $t3 #suma los elementos del vector
       addi $t2,$t2, 4    #$t2=$t2+4
       addi $t0,$t0, 1    #$t0=$t0+1
       j para             #saltar a bucle
finpara:  sw $t3, res($0) #almacenar $t3 en res
```

La descripción algorítmica de este programa en ensamblador se puede transcribir como un bucle repetitivo while como se muestra a continuación:

```
VARIABLES
     VECTOR DE ENTEROS: v(6)=(6,7,8,9,10,1);
     ENTERO: res;
INICIO
     res=0;
     i=0;
          Mientras (i<=5) hacer
               res=res+v[i];
               i=i+1;
```

```
          FinMientras
FIN
```

Una descripción algorítmica equivalente es utilizar una estructura de control que incluyen la mayoría de los lenguajes de alto nivel, por ejemplo *for* de C etc. Así utilizando esta estructura la descripción algorítmica del programa ensamblador anterior resulta más sencilla:

```
PROGRAMA PARA-1
VARIABLES
     VECTOR DE ENTEROS: v(6)=(6,7,8,9,10,1);
     ENTERO: res;
INICIO
     res=0;
     Para i=0 hasta 5 hacer
          res=res+v[i];
     FinPara
FIN
```

Borra los valores de la memoria, carga el fichero que contiene el programa en ensamblador en el simulador.

Cuestión 2.22: Ejecuta paso a paso el programa y comprueba detenidamente la función que tiene cada una de las instrucciones que componen el programa en ensamblador.

Cuestión 2.23: ¿Qué valor se almacena en `res` después de ejecutar el código anterior?

Cuestión 2.24: Implementa en ensamblador el siguiente programa descrito en pseudocódigo:

```
VARIABLES
     VECTOR DE ENTEROS: v1(8)=(6,7,8,9,10,-1,34,23);
                        v2(8);
INICIO
     Para i=0 hasta 7 hacer
          v2[i]=v1[i]+1;
     FinPara
FIN
```

Problemas propuestos

4. Diseña un programa en ensamblador que almacene en memoria los 5 enteros siguientes (dato1=2, dato2=10, dato3=50, dato4=70, dato5=34) y que reserve 1 palabra para almacenar el resultado, (variable res). Implementa en ensamblador del R2000 un programa que almacene en la variable res un 1 si dato5 está en alguno de los intervalos formados por dato1 y dato2 o dato3 y dato4. Se almacenará un cero en caso contrario.

5. Diseña un programa en ensamblador que dado un vector de enteros, obtenga como resultado cuántos elementos son iguales a cero. Este resultado se debe almacenar sobre la variable "total". El programa deberá inicializar los elementos del vector en memoria, así como una variable que almacenará el número de elementos que tiene el vector y reservará espacio para la variable resultado.

6. Diseña un programa en ensamblador que dado un vector de enteros "V" obtenga cuántos elementos de este vector están dentro del rango determinado por dos variables "rango1" y "rango2". El programa deberá inicializar los elementos del vector en memoria, una variable que almacenará el número de elementos que tiene ese vector y dos variables donde se almacenarán los rangos. También deberá reservar espacio para la variable resultante.

7. Diseña un programa en ensamblador que dado un vector de caracteres, contabilice cuántas veces se repite un determinado carácter en el mismo. El programa deberá inicializar la cadena en memoria, y ésta deberá finalizar con el carácter nulo. También deberá reservar espacio para la variable resultado.