Problemas Tema 3:

Programación en ensamblador

Juan Lanchares Dávila
Fernando Castro Rodríguez
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid

1) Escribe un programa en lenguaje ensamblador que implemente el siguiente bloque IF-THEN-ELSE. Usa la sección *.data* para asignar algún valor inicial a las variables.

```
if (x >= y) {
        x = x+2;
        y = y-2;
}
```

```
.global main                # Hace global la etiqueta " main "
.data                       # sección de datos iniciados
x: .word 10                 # declara una variable de 32 bits de valor 10
y: .word 5

.text                       # sección de instrucciones
main :

        la t0,x             # pseudo instrucción t0=@x
        la t1,y             # pseudo instrucción t1=@y
        lw s1,0(t0)         # s1 = 10
        lw s2,0(t1)         # s2 =5
        blt s1,s2, fin      # condición inversa s1 <s2
        addi s1,s1,2        # x=x+2
        addi s2,s2,-2       # y=y -2
        sw s1,0(t0)
        sw s2,0(t1)
fin:
        j .                 # última instrucción que se ejecuta
.end                        # directiva que indica que se acaba el fichero
```

2) Escribe un programa en lenguaje ensamblador que implemente el siguiente bloque IF-THEN-ELSE. Usa la sección *.data* para asignar algún valor inicial a las variables.

```
if (x >= y) {
        x = x+2;
        y = y+2;
}
else {
        x = x-2;
        y = y-2;
}
```

```
.global main

.data                           # sección de datos iniciados
x: .word 5
y: .word 10

.text                           # sección de instrucciones
main :
        la t0,x                 # pseudoinstrucción t0=@x
        la t1,y                 # pseudoinstrucción t1=@y
        lw s1,0(t0)             # s1 =5
        lw s2,0(t1)             # s2 =10
        blt s1,s2, else         # condición inversa s1 <s2
        addi s1,s1,2            # x=x+2
        addi s2,s2,2            # y=y+2
        j fin_if
else :
        addi s1,s1,-2           # x = x -2;
        addi s2,s2,-2           # y = y -2
fin_if :
        sw s1,0(t0)
        sw s2,0(t1)
fin:
        j .
.end
```

3) Escribe un programa en lenguaje ensamblador que implemente el siguiente código. Usa la sección *.bss* para reservar espacio en memoria para las variables.

```
a = 81;
b = 18;
do {
        a = a-b;
} while (a > 0);
```

```
.global main
.bss
a: .space 4
b: .space 4

.text
main :
        la t1,a
        la t2,b
        li s1,81
        sw s1,0(t1)       # a = 81
        li s2,18
        sw s2,0(t2)       # b = 18
do:
        sub s1,s1,s2      # a=a-b
        sw  s1,0(t1)
while :
        bgt s1,zero,do    # condición directa
end:
        j .
.end
```

4) Escribe un programa en lenguaje ensamblador que implemente el siguiente código. Usa la sección *.bss* para reservar espacio en memoria para las variables.

```
n = 5;
fprev = 1;
f = 1;
i = 2;
while (i <= n) {
        faux = f;
        f = f + fprev;
        fprev = faux;
        i = i+1;
}
```

```
.global main
.bss
n: .space 4
fprev : .space 4
f: .space 4
i: .space 4
faux : .space 4

.text
main :
        la t1,f
        li s1,1          # cargo f en s1 y la guardo
        sw s1,0(t1)
        la t0,i
        li s2,2          # cargo i en s2 y la guardo
        sw s2,0(t0)
        la t0,fprev
        li s3,1          # cargo fprev en s3 y la guardo
        sw s3,0(t0)
        la t0,n
        li s4,5          # cargo n en s4 y la guardo
        sw s4,0(t0)
```

4) Escribe un programa en lenguaje ensamblador que implemente el siguiente código. Usa la sección *.bss* para reservar espacio en memoria para las variables.

```
n = 5;
fprev = 1;
f = 1;
i = 2;
while (i <= n) {
        faux = f;
        f = f + fprev;
        fprev = faux;
        i = i+1;
}
```

```
while :
        bgt s2,s4,end
        mv t2,s1                    # t2 es faux , guardo al modificar
        la t3,faux
        sw t2,0(t3)
        add s1,s1,s3                # f = f + fprev
        sw s1,0(t1)                 # guardo f al modificarse
        mv s3,t2                    # modifico fprev , la guardo
        la t3,fprev
        sw s,0(t3)
        addi s2,s2,1                # modifico i, la guardo
        la t3,i
        sw s2,0(t3)
        j while
end:
        j .
.end
```

5) Escribe un programa en lenguaje ensamblador que implemente el siguiente código. Usa la sección *.data* para asignar algún valor inicial a las variables f y n, y la sección *.bss* para reservar espacio de memoria para la variable i.

```
for (i=2; i<=n; i++) {
        f=f+f;
}
```

```
.global main
.equ n,5
.data
f: .word 2
n: .word 5
.bss
i: .space 4

.text
main :
        la t1,f          # t1=@f
        lw s1,0(t1)      # s1 es f
        la s2,n          # s2 es n
        lw s2,0(s2)
        la t2,i
        li s3,2          # s3 es el i inicializado a 2
        sw s3,0(t2)
for:
        bgt s3,s2,end    # condición inversa
        add s1,s1,s1
        sw s1,0(t1)
        addi s3,s3,1
        sw s3,0(t2)
        j for
end:
        j .
.end
```

6) El siguiente programa calcula el máximo común divisor de dos números a y b según el algoritmo de restas de Euclides. Traducirlo a ensamblador del RISC-V. Se supone en este caso que todo el programa consta de una sola sección (*.text*) con la siguiente estructura:

- Los valores iniciales de a y b se declaran con directivas *.word*, al comienzo de la sección.
- A continuación, se reserva espacio para la variable mcd, por medio de una directiva *.space*
- Finalmente, se escriben las instrucciones del programa

```
int a=5, b=15, mcd;
while (a≠b){
        if (a>b)
                a=a-b;
        else
                b=b-a;
}
mcd=a;
```

```
.global main
.data
a: .word 5
b: .word 15
.text
mcd: .space 4
main :
        la t1,a
        lw s1,0(t1)          # s1 es a
        la t2,b
        lw s2,0(t2)          # s2 es b
while :
        beq s1,s2,fin_while
        ble s1,s2,else
        sub s1,s1,s2         # a=a-b
        j fin_if
else :
        sub s2,s2,s1         # b=b-a
fin_if :
        j while
fin_while :
        la t3,mcd            # t3 = @mcd
        sw s1,0(t3)
fin:
        j .
.end
```

7) Traduce la siguiente sentencia en C:

f=g+h+B[4]

a ensamblador, donde f, g y h son enteros contenidos en memoria y B es un vector de 10 componentes.

```
.global main
.equ n,4
.data
g: .word 2
h: .word 3
b: .word 0,1,2,3,4,5,6,7,8,9
.bss
f: .space 4

.text
main :
        la t2,g
        lw s2,0(t2)       # g en s2
        la t3,h
        lw s3,0(t3)       # h en s3
        la t4,b           # t4 es la dirección base de B
        li t5,n           # t5 contiene el índice , en este caso 4
        slli t5,t5,2      # se multiplica el índice por 4 porque el tamaño es palabra (4B)
        add t5,t5,t4      # @= dirbase + indice *4
        lw t5,0(t5)       # t5=B[4]
        add s2,s2,s3      # g+h
        add s2,s2,t5      # g+h+B[4]
        la t1,f
        sw s2,0(t1)
fin:
        j .
.end
```

8) Tenemos un vector de 10 componentes almacenado en la posición de memoria etiquetada como V. Diseña un programa en ensamblador que sume uno a cada una de sus componentes. Ayuda: utiliza el siguiente pseudo-código de alto nivel:

```
#define N 10
int V[N]={12,1,-2,15,-8,4,-31,8,8,25};
for (i=0; i<N; i++)
            V[i] = V[i]+1;
```

```
.global main
.equ n,10
.data
v: .word 12,1,-2,15,-8,4,-31,8,8,25

.text
main :
        li s1,n          # s1=n
        mv s2,zero       # s2 es i
for:
        beq s2,s1,fin
        la t1,v          # t1= @base de v
        slli t3,s2,2     # i*4
        add t2,t1,t3     # t2= @efectiva de v[i]
        lw s3,0(t2)
        addi s3,s3,1
        sw s3,0(t2)
        addi s2,s2,1     # i=i+1
        j for
fin:
        j .
.end
```

9) Tenemos un vector de 6 componentes almacenado en la posición de memoria etiquetada como V. Diseña un programa en ensamblador que cuente el número de valores mayores que 0 que contiene. Ayuda: utiliza el siguiente pseudo-código de alto nivel:

```
#define N 6
int V[N]={14,1,-2,7,-8,4};
int count = 0;

for (i=0; i<N; i++) {
            if (V[i] > 0)
            count = count+1;
}
```

```
.global main
.equ n,6
.data
v: .word -14,1,-2,-7,-8,4
.bss
count : .space 4

.text
main :
        la t1,v          # t1 tiene la dirección base de v
        li t2,n          # t2=n
        li t3,0          # t3 es el índice
        li s2,0          # s2 = count =0
for:
        bge t3,t2,fin_for
        slli t5,t3,2     # t5=i*4
        add t5,t5,t1     # @=i*4+ @b
        lw s1,0(t5)      # @s1=v[i]
        li t6,0          # t6 =0
if:
        ble s1,t6,fin_if
        addi s2,s2,1
fin_if :
        addi t3,t3,1
        j for
fin_for :
        la t1,count
        sw s2,0(t1)
end:
        j .
.end
```

10) Implementar un programa en ensamblador que calcule la sucesión de *Fibonacci* y la almacene en un vector V de longitud arbitraria N. Esta sucesión infinita de números naturales queda definida como: V(0)=0, V(1)=1, V(i+2)=V(i+1)+V(i) (i=0,1,2,...). La dimensión del vector V se debe definir en el programa como una constante. Ayuda: utiliza el siguiente pseudo-código de alto nivel:

```
#define N 12
int V[N];
V[0] = 0;
V[1] = 1;
for (i=0; i< N-2; i++)
        V[i+2] = V[i+1] + V[i];
```

```
.global main
.equ n,12
.bss
v: .space 48
.text
main :
        la t1,v             # t1= @base de v
        mv t0,zero
        sw t0,0(t1)         # V [0] = 0
        addi t0,t0,1        # i=1
        slli t0,t0,2        # i*4
        add t0,t0,t1        # t0= @efectiva de i=1
        addi t2,zero,1      # t2 =1
        sw t2,0(t0)         # v[1]=1
        mv t0,zero          # i=0
        li t3,n
        addi t3,t3,-2       # t3=N-2, luego t3 =10
for:
        bge t0,t3,fin
        slli t2,t0,2        # i*4
        add t4,t1,t2
        lw t5,0(t4)
        addi t4,t4,4
        lw t6,0(t4)
        add t6,t6,t5
        addi t4,t4,4
        sw t6,0(t4)
        addi t0,t0,1
        j for
fin:
        j .
.end
```

11) Dado un vector A de 12 componentes se desea generar otro vector, B, tal que B sólo contiene las componentes de A que son números pares mayores que cero. Ejemplo: Si A = (0,1,2,7,-8,4,5,12,11,-2,6,3), entonces B = (2,4,12,6). Ayuda: utiliza el siguiente pseudo-código de alto nivel:

```
#define N 12
int A[N]={0,1,2,7,-8,4,5,12,11,-2,6,3};
int B[N];
int countB=0;
j=0;
for (i=0; i<N; i++) {
                  if (A[i] > 0 && A[i] es par){
                          B[j]=A[i];
                          j++;
                  }
}
countB=j;
```

```
.global main
.equ n ,12
.data
a: .word 0,1,2,7,-8,4,5,12,11,-2,6,3
.bss
b:        .space 48
countB:   .space 4
.text
main :
          mv t1,zero          # t1 es j
          mv t2,zero          # t2 es i
          la s1,a             # s1 es la @base de a
          la s2,b             # s2 es la @base de b
          li s3,n             # s3 contiene la constante n
for:
          bge t2,s3,fin_for
          slli t3,t2,2        # i*4
          add t3,t3,s1        # @efectiva de i
          lw t4,0(t3)
          blez t4,fin_if
          andi t6,t4,1        # operación AND con 1
          bnez t6,fin_if
          slli t3,t1,2
          add t3,t3,s2
          sw t4,0(t3)         # B[j]=A[i]
          addi t1,t1,1
fin_if :
          addi t2,t2,1
          j for
fin_for :
          la t3,countB
          sw t1,0(t3)         # countB =j
fin:
          j .
.end
```

12) Dados dos vectores A y B de 10 componentes cada uno, se desea construir otro vector, C, tal que:

$$C(i) = |A(i) + B(9-i)|, \quad i = 0,..,9.$$

Escribe un programa en lenguaje de alto nivel que construya el vector C. Traduce el programa al lenguaje ensamblador del RISC-V. Ayuda:

```
#define N 10

for (i=0; i<N; i++) {
        aux = A[i]+B[N-1-i];
        /* if aux is negative, change the sign */
        if (aux < 0)
                aux = 0-aux;
        C[i] = aux;
}
```

```
.global main
.equ n,10
.data
A: .word 1,2,3,4,5,6,7,8,9,10
B: .word 10,9,8,7,6,5,4,3,2,1
.bss
C: .space 40 #2,4,6,8,10,12,14,16,18,20
.text
main :
        li t1,n             # t1=n =10
        mv t2, zero         # t2 es i
for:
        bge t2,t1, fin_for
        la t3,A             # t3 = @base A
        slli t4,t2,2        # i*4
        add t4,t4,t3        # @efectiva de i
        lw s1,0(t4)         # s1=A[i]
        la t3,B             # t3 = @base de B
        li t4,9             # t4 =9
        sub t4,t4,t2        # t4 =9-i
        slli t4,t4,2
        add t4,t4,t3
        lw s2,0(t4)
        add s1,s1,s2
        bge s1,zero,store
        sub s1,zero,s1      # negativo
store :
        la t3,C
        slli t4,t2,2        # i*4
        add t4,t4,t3        # @efectiva de i
        sw s1,0(t4)
        addi t2,t2,1
        j for
fin_for :
        j .
.end
```

13) Traduce el siguiente programa escrito en un lenguaje de alto nivel a lenguaje ensamblador. La orden swap(a, b) intercambia los valores de las variables a y b.

```
int a=13, b=16;

while (a>10){
                a=a-1;
                b=b+2;
}
if (a<b)
                swap (a, b);
else
                b= a-1;
```

```
.global main
.data
a: .word 13
b: .word 16

.text
main :
        la t1,a
        lw s1,0(t1)
        la t2,b
        lw s2,0(t2)
        li t3,10
while :
        ble s1,t3,if
        addi s1,s1,-1
        addi s2,s2,2
        j while
if:
        bge s1,s2,else
        sw s1,0(t2)
        sw s2,0(t1)
        j fin
else :
        addi s2,s1,-1
        sw s1,0(t1)
        sw s2,0(t2)
fin:
        j .
.end
```

14) Escribe un programa para el ensamblador del RISC-V que llame a una subrutina, swap(int *a, int *b), encargada de intercambiar el contenido de dos posiciones de memoria. La subrutina recibirá como parámetros de entrada las posiciones de memoria correspondiente a *a* y *b* y deberá preservar el contenido de todos los registros que obligue el estándar de llamadas estudiado en clase.

Cuestión: ¿Qué registros debemos utilizar dentro de la función para evitar tener que salvar y restaurar registros durante el proceso?

Bastaría solo con utilizar registros temporales durante la subrutina para no tener que guardar nada

```
.global main
.extern _stack # La @ de inicio de pila está en el archivo de enlazado

.data
a: .word 10
b: .word 15

.text
main :
        la sp,_stack      # iniciación de la pila
        la a0,a           # paso de parámetros
        la a1,b
        jal swap          # jal=call , salto a subrutina
fin:
        j .
swap :
        # prólogo
        addi sp,sp,-8     # es sr hoja , no hay que guardar el
                          # registro de retorno ra
        # cuerpo
        sw s1,0(sp)
        sw s2,4(sp)
        lw s1,0(a0)
        lw s2,0(a1)
        sw s1,0(a1)
        sw s2,0(a0)
        # epílogo
        lw s1,0(sp)
        lw s2,4(sp)
        addi sp,sp,8
        jr ra              # también ret
.end
```

15) Escribe un programa para el ensamblador del RISC-V que cuente el número de 0's de un vector de longitud arbitraria. Emplea para ello una subrutina llamada cuenta0s que reciba como parámetros de entrada toda la información necesaria para llevar a cabo la tarea.

```
.global main
.extern _stack
.equ n ,10
.data
V: .word 1,0,2,0,3,0,4,0,5,0
.bss
numceros : .space 4

.text
main :
        la sp,_stack      # inicia la pila
        la a0,V           # paso de parámetros a la subrutina
        li a1,n
        call cuenta0s     # llamada a subrutina
        la t1,numceros
        sw a0,0(t1)
fin:
        j .

cuenta0s :
        #prólogo
        addi sp,sp,-24
        sw s1,0(sp)
        sw s2,4(sp)
        sw s3,8(sp)
        sw s4,12(sp)
        sw s5,16(sp)
        sw s6,20(sp)
```

15) Escribe un programa para el ensamblador del RISC-V que cuente el número de 0's de un vector de longitud arbitraria. Emplea para ello una subrutina llamada cuenta0s que reciba como parámetros de entrada toda la información necesaria para llevar a cabo la tarea.

```
        # cuerpo
        mv s1,zero          # s1=i
        mv s2,a1            # s2=n
        mv s3,a0            # s3= @base v
        mv s6,zero          # s6 cuenta ceros
for:
        bge s1,s2,fin_for
        slli s4,s1,2
        add s4,s4,s3
        lw s5,0(s4)
        bnez s5,fin_if
        addi s6,s6,1
fin_if :
        addi s1,s1,1
        j for
fin_for :
        mv a0,s6            # devolución del valor

        #epílogo
        lw s1,0(sp)
        lw s2,4(sp)
        lw s3,8(sp)
        lw s4,12(sp)
        lw s5,16(sp)
        lw s6,20(sp)
        addi sp,sp,24
        ret
.end
```

16) Implementar el algoritmo de ordenación de la burbuja o *bubble sort* en ensamblador. Este sencillo algoritmo ordena los elementos de un vector de menor a mayor por medio de un procedimiento muy sencillo: recorre repetidas veces el vector, intercambiado posiciones sucesivas si V(i)>V(i+1), hasta que no se realiza ningún cambio. Se proporciona como ayuda el siguiente código de alto nivel:

```
do
      swapped=false
      for i from 0 to N-2 do:
               if V[i] > V[i+1] then
                        swap( V[i], V[i+1] )
                        swapped = true
               end if
      end for
while swapped
```

Nota. Usa una constante, N, para definir la longitud del vector.

```
.global main
.extern _stack
.equ n, 10
.data
V: .word 2,5,6,0,9,4,6,5,-10,-1
.text
main :
          la sp,_stack
          li s4,n              # s1 =n
          addi s4,s4,-1
do:
          mv s3,zero           # s3= swapped = false
          mv s5,zero           # t1=i
for:
          bge s5,s4, fin_for
          la t2,V              # t2= @base v
          slli t3,s5,2         # i*4
          add a0,t3,t2         # @i
          lw s1,0(a0)          # V[i]
          addi a1,a0,4         # @i +1
          lw s2,0(a1)          # V[i +1]
if:
          ble s1,s2,fin_if
          call swap
          li s3,1              # swapped = true
fin_if :
          addi s5,s5,1
          j for
fin_for :
          li t4,1
          beq s3,t4,do
fin:
j .
```

16) Implementar el algoritmo de ordenación de la burbuja o *bubble sort* en ensamblador. Este sencillo algoritmo ordena los elementos de un vector de menor a mayor por medio de un procedimiento muy sencillo: recorre repetidas veces el vector, intercambiado posiciones sucesivas si V(i)>V(i+1), hasta que no se realiza ningún cambio. Se proporciona como ayuda el siguiente código de alto nivel:

```
do
      swapped=false
      for i from 0 to N-2 do:
               if V[i] > V[i+1] then
                        swap( V[i], V[i+1] )
                        swapped = true
               end if
      end for
while swapped
```

Nota. Usa una constante, N, para definir la longitud del vector.

```
#vamos a utilizar el código de la subrutina
#swap del problema 15 modificado:
#los parámetros de llamada son las direcciones
#de memoria de los valores a intercambiar.

swap :
        # prologo
        addi sp,sp,-8
        sw s1,0(sp)
        sw s2,4(sp)

        # cuerpo
        lw s1,0(a0)
        lw s2,0(a1)
        sw s1,0(a1)
        sw s2,0(a0)

        # epílogo
        lw s1,0(sp)
        lw s2,4(sp)
        addi sp,sp,8
        jr ra               # también ret
.end
```

17) Escribe un programa en lenguaje de alto nivel que llame a una función *fact* que calcule el factorial de un número no negativo, n, por medio de un bucle que realiza una secuencia iterativa de multiplicaciones. Traduce el programa al lenguaje ensamblador del RISC-V.

```
int fact(int n);
int n=6, rFact;

void main(){
        rFact = fact(n);
        while(1);
}
// La función fact devuelve el
// factorial del entero
// que recibe como argumento
int fact(int num){
        int i,resu;
        if (num > 1){
                resu=num;
                for (i=num-1;i>1;i--)
                        resu = resu*i;
                }
        else
                resu=1;
        return(resu);
}
```

```
.global main
.extern _stack
.equ n,6
.bss
res: .space 4

.text
main :
        la sp,_stack
        li a0,n
        call fact
        la t1,res
        sw a0,0(t1)
fin:
        j .
```

17) Escribe un programa en lenguaje de alto nivel que llame a una función *fact* que calcule el factorial de un número no negativo, n, por medio de un bucle que realiza una secuencia iterativa de multiplicaciones. Traduce el programa al lenguaje ensamblador del RISC-V.

```
int fact(int n);
int n=6, rFact;

void main(){
        rFact = fact(n);
        while(1);
}
// La función fact devuelve el
// factorial del entero
// que recibe como argumento
int fact(int num){
        int i,resu;
        if (num > 1){
                resu=num;
                for (i=num-1;i>1;i--)
                        resu = resu*i;
                }
        else
                resu=1;
        return(resu);
}
```

```
fact :
        # prólogo
        addi sp,sp,-12    # es subrutina hoja ,
        sw s1,0(sp)
        sw s2,4(sp)
        sw s3,8(sp)

        # cuerpo
        li s1,1           # s1=resu
        mv s2,a0          # s2=i
        li s3,1
for:
        ble s2,s3,fin_for
        mul s1,s1,s2
        addi s2,s2,-1
        j for
fin_for :
        mv a0,s1

        # epílogo
        lw s1,0(sp)
        lw s2,4(sp)
        lw s3,8(sp)
        addi sp,sp,12
        jr ra             # también ret
.end
```

18) (Septiembre 2015) Dados dos puntos P1(x1,y1) y P2(x2,y2), la distancia de Chebyshev entre ellos puede calcularse con el siguiente algoritmo:

```
int chebyshev(x1, y1, x2, y2)
{
  int d1, d2;
  d1 = abs(x1-x2)
  d2 = abs(y1-y2)
  if (d2 > d1)
       d1 = d2;
  return d1;
}
```

a) Codificar en ensamblador la subrutina chebyshev(x1, x2, y1, y2), que recibe cuatro parámetros enteros con signo que se corresponden con las coordenadas *(x1, x2)* e *(y1, y2)* de dos puntos P1 y P2 y devuelva la distancia de Chebyshev que separa P1 y P2. En su cuerpo, invocará a la subrutina absoluto. Téngase en cuenta que la subrutina no es una subrutina hoja.

b) Codificar en ensamblador un programa que almacene en un vector D la distancia de Chebyshev desde un punto P a todos los puntos de un vector V de N puntos, donde P, V y D serían variables globales. El vector V tendrá 2N enteros de forma que el i-ésimo punto tendrá coordenadas (x, y) = (V[2*i], V[2*i + 1]). Un código C equivalente sería:

```
#define N, ...

int Px, Py; //coordenadas x e y del punto P
int V[2N];  //Vector con N puntos V=[x0,y0,x1,y1,...]
int D[N];   //Vector de N distancias

void main(void)
{
      int i;
      for (i=0; i < N; i++)
           D[i] = chebyshev(Px, Py, V[2*i], V[2*i + 1]);
}
```

```
int chebyshev(x1, y1, x2, y2)
{
  int d1, d2;
  d1 = abs(x1-x2)
  d2 = abs(y1-y2)
  if (d2 > d1)
       d1 = d2;
  return d1;
}
```

```
#define N, ...

int Px, Py; //coordenadas x e y del punto P
int V[2N];  //Vector con N puntos V=[x0,y0,x1,y1,...]
int D[N];   //Vector de N distancias

void main(void)
{
      int i;
      for (i=0; i < N; i++)
           D[i] = chevishev(Px, Py, V[2*i], V[2*i + 1]);
}
```

```
.global main
.extern _stack
.equ n,5  #nº de puntos a testear (2*n componentes)

.data
P: .word 4,5 # coordenadas x e y del punto P
V: .word 1,2,-3,4,5,9,17,-15,20,12 # Vector de N
puntos V=[x0,y0,x1,y1,...]
.bss
sol : .space n*4

.text
main :
          la sp,_stack
          mv fp,zero
          mv s1,zero
          li s2,n
          la s3,V

for:      bge s1,s2,fin_for
          la s6,P
          lw a0,0(s6)
          lw a1,4(s6)
          slli s4,s1,1
          slli s4,s4,2
          add s4,s4,s3
          lw a2,0(s4)
          lw a3,4(s4)
          call chebyshev
          la s5,sol
          slli s4,s1,2
          add s4,s4,s5
          sw a0,0(s4)
          addi s1,s1,1
          j for
fin_for: j .
```

```
int chebyshev(x1, y1, x2, y2)
{
  int d1, d2;
  d1 = abs(x1-x2)
  d2 = abs(y1-y2)
  if (d2 > d1)
       d1 = d2;
  return d1;
}
```

```
#define N, ...

int Px, Py; //coordenadas x e y del punto P
int V[2N];  //Vector con N puntos V=[x0,y0,x1,y1,...]
int D[N];   //Vector de N distancias

void main(void)
{
      int i;
      for (i=0; i < N; i++)
           D[i] = chevishev(Px, Py, V[2*i], V[2*i + 1]);
}
```

```
chebyshev :
          #prólogo
          addi sp,sp,-12
          sw s1,0(sp)
          sw s2,4(sp)
          sw ra,8(sp)

          # cuerpo
d1:
          sub s1,a0,a2 #x1 -x2
          mv a0,s1
          call abs
          mv s1,a0
d2:
          sub s2,a1,a3 #y1 -y2
          mv a0,s2
          call abs
          mv s2,a0
if:
          ble s2,s1,fin_call
          mv s1,s2
fin_call :
          mv a0,s1

# epílogo
          lw s1,0(sp)
          lw s2,4(sp)
          lw ra,8(sp)
          addi sp,sp,12
          ret
abs:      bgez a0,pos
          sub a0,zero,a0
pos:      ret
.end
```

19) (Junio 2016) Dado un vector, A, de 3*N componentes, se desea obtener un nuevo vector, B, de N componentes donde cada componente de B es la suma módulo 32 de una tripleta de elementos consecutivos de A. Es decir:

B[0] = (A[0]+A[1]+A[2]) mod 32, B[1] = (A[3]+A[4]+A[5]) mod 32, etc

Se pide:

a) Escribir un programa en lenguaje ensamblador del RISC-V que implemente el cálculo descrito de acuerdo con el siguiente código C equivalente:

```
#define N 4
int A[3*N] = {una lista de 3*N valores},
int B[N];
int i, j=0;
void main (void)
{
          for (i=0; i<N; i++){
                    B[i] = sum_mod_32(A, j, 3);
                    j=j+3
          }
}
```

donde la subrutina sum_mod_32(V,p,m) devuelve como resultado la suma módulo 32 de m elementos consecutivos del vector V tomados a partir de la posición p.

b) Escribir el código ensamblador de la subrutina sum_mod_32, de acuerdo al siguiente código C equivalente:

```
sum_mod_32 (int A[ ], int j, int len)
          {
                    int i, sum=0;
                    for (i=0; i<len; i++)
                              sum = sum + A[j+i];
                    sum=mod_power_of_2(sum,5);
                    return sum;
          }
```

donde la subrutina mod_power_of_2(num, exp), siendo *num* un entero positivo y *exp* un entero mayor que 0 y menor que 32, devuelve como resultado el valor de (num mod 2^exp). Por ejemplo, si invocamos a la función como: mod_power_of_2(34, 5), la función devolvería como salida: ((34) mod (2^5))=(34 mod 32)=2. Nota.- En todos los apartados se debe respetar el estándar de llamadas a subrutinas estudiado en clase, y las variables pueden ubicarse en registros o en memoria (global o pila según corresponda).

a)

```
#define N 4
int A[3*N] = {una lista de 3*N valores},
int B[N];
int i, j=0;
void main (void)
{
          for (i=0; i<N; i++){
                    B[i] = sum_mod_32(A, j, 3);
                    j=j+3
          }
}
```

```
.global main
.extern _stack
.equ n,4
.data
A: .word 10,11,12,10,11,12,10,11,12,10,11,12
.bss
B: .space 16 # también se puede .zero

.text
main :
          la sp,_stack
          mv fp,zero
          la s1,A           # s1 tiene la @base de A
          la s2,B           # s2 @base de B
          mv s3,zero        # s3 es i
          mv s4,zero        # s4 es j
          li s5,n           # s5=n
for:
          bge s3,s5,fin_for
          la a0,A           # pasa @base A como argumento
          mv a1,s4          # pasa j como argumento
          li a2,3           # pasa 3 como argumento
          call sum_mod      # llama a la subrutina
          la s6,B           # s6 @base de B
          slli t0,s3,2      # t0=s3 *4
          add t0,t0,s6      # @efectiva
          sw a0,0(t0)
          addi s4,s4,3      # j=j+3
          addi s3,s3,1      # i=i+1
          j for
fin_for :
          j .
```

b)

```
sum_mod_32 (int A[ ], int j, int len)
        {
                int i, sum=0;
                for (i=0; i<len; i++)
                        sum = sum + A[j+i];
                sum=mod_power_of_2(sum,5);
                return sum;
        }
```

```
sum_mod : #prólogo
        addi sp,sp,-20
        sw s0,0(sp)
        sw s1,4(sp)
        sw s2,8(sp)
        sw s3,12(sp)
        sw s4,16(sp)

        # cuerpo; a0 es @base de A; a1 es j; a2 es 3
        mv s0,zero        # s0 es el índice del for2
        mv s4,zero        # inicializa el acumulador
for2 :
        bge s0,a2,fin_for2 # compara con a2 =3
        slli s1,a1,2      # a1 es j=>j*4
        add s2,s1,a0      # s2 = @efectiva =j*4+ @base
        lw s3,0(s2)
        add s4,s4,s3      # s4 acumulador
        addi s0,s0,1      # indice +1
        addi a1,a1,1      # j+1
        j for2
fin_for2 :
        andi s4,s4,31
        mv a0,s4
        #epílogo
        lw s0,0(sp)
        lw s1,4(sp)
        lw s2,8(sp)
        lw s3,12(sp)
        lw s4,16(sp)
        addi sp,sp,20
        ret
.end
```

20) (Septiembre 2016) Dado un vector **A** de **N** enteros de 32bits sin signo, se desea calcular su Código de Redundancia Cíclico o **CRC** siguiendo el algoritmo de Fletcher. Dicho algoritmo genera como salida dos enteros sin signo que servirán para comprobar la integridad de los datos. El algoritmo de Fletcher de 64bits se puede implementar a partir del siguiente código:

```
void Fletcher64( unsigned int data[], int length, unsigned int crc[] ) {
    unsigned int sum1 = 0;
    unsigned int sum2 = 0;
    int index;

    for ( index = 0; index < length; index++ ) {
        sum1 = sum_mod64(sum1, data[index]);
        sum2 = sum_mod64(sum1 , sum2);
    }
    crc[0] = sum1;
    crc[1] = sum2;
}
```

donde el primer argumento es el vector de datos, el segundo es la longitud del vector y el tercero es el vector que contiene el resultado (es decir, los dos enteros sin signo que componen el CRC). Se supone que la rutina `unsigned int sum_mod64(unsigned int A, unsigned int B)` está ya implementada.

a) Escribe el código ensamblador de la rutina `Fletcher64`.

b) Escribe un programa en lenguaje ensamblador del RISC-V que, invocando a la rutina `Fletcher64`, calcule el CRC de los siguientes vectores:

V={0x12340000, 0x00005678}, y

W={0xAB000000, 0x00CD0000, 0x0000EF00, 0x00000011}

Debes llamar a la rutina `Fletcher64` dos veces. La primera llamada retornará el CRC de V y la segunda el CRC de W.

Nota.- En todos los apartados se debe respetar el estándar de llamadas a subrutinas estudiado en clase, y las variables pueden ubicarse en registros o en memoria (global o pila según corresponda).

a)

```
void Fletcher64( unsigned int
data[], int length, unsigned
int crc[] ) {
    unsigned int sum1 = 0;
    unsigned int sum2 = 0;
    int index;

    for ( index = 0; index <
length; index++ ) {
        sum1 = sum_mod64(sum1,
data[index]);
        sum2 = sum_mod64(sum1
, sum2);
    }
    crc[0] = sum1;
    crc[1] = sum2;
}
```

```
Fletcher64 :
        #prólogo
        addi sp,sp,-36
        sw s0,0(sp)
        sw s1,4(sp)
        sw s2,8(sp)
        sw s3,12(sp)
        sw s4,16(sp)
        sw s5,20(sp)
        sw s6,24(sp)
        sw s7,28(sp)
        sw ra,32(sp)

        # cuerpo
        mv s1,zero        # s1 guarda sum1
        mv s2,zero        # s2 guarda sum2
        mv s0,a0          # en s0 la dirección del vector
        mv s3,a1          # en s3 la longitud del vector
        mv s4,a2          # en s4 la dirección del vector resultado
        mv s5,zero        # s5 es el índice i del vector A
        mv s6,zero        # s6 para dirección de acceso a A
```

a)

```
void Fletcher64( unsigned int
data[], int length, unsigned
int crc[] ) {
    unsigned int sum1 = 0;
    unsigned int sum2 = 0;
    int index;

    for ( index = 0; index <
length; index++ ) {
        sum1 = sum_mod64(sum1,
data[index]);
        sum2 = sum_mod64(sum1
, sum2);
    }
    crc[0] = sum1;
    crc[1] = sum2;
}
```

```
for:
        bge s5,s3,fin_for
        slli s6,s5,2        # s6=s5*4
        add s6,s6,s0        # @efectiva
        lw s7,0(s6)         # pasa elemento de A como argumento
        mv a0,s1
        mv a1,s7
        call sum_mod64      # llama a la subrutina
        mv s1,a0
        mv a1,s2
        call sum_mod64
        mv s2,a0
        addi s5,s5,1        # i=i+1
        j for
fin_for : mv a0,s1
         mv a1,s2
         sw a0,0(s4)
         sw a1,4(s4)

        #epílogo
        lw s0,0(sp)
        lw s1,4(sp)
        lw s2,8(sp)
        lw s3,12(sp)
        lw s4,16(sp)
        lw s5,20(sp)
        lw s6,24(sp)
        lw s7,28(sp)
        lw ra,32(sp)
        addi sp,sp,36
        ret
.end
```

b)

```
void Fletcher64( unsigned int data[], int
length, unsigned int crc[] ) {
    unsigned int sum1 = 0;
    unsigned int sum2 = 0;
    int index;

    for ( index = 0; index < length;
index++ ) {
        sum1 = sum_mod64(sum1,
data[index]);
        sum2 = sum_mod64(sum1 , sum2);
    }
    crc[0] = sum1;
    crc[1] = sum2;
}
```

```
.global main
.extern _stack
.equ Nv,64
.equ Nw,128

.data
V: .word 0x1234000, 0x00005678
W: .word 0xAB00000, 0x000CD000, 0x00000EF00, 0x00000011

.bss
crc_v: .space 2*4
crc_w: .space 2*4

.text
main :
        la sp,_stack
        mv fp,zero
        la a0,V          # en a0 1er argumento
        li a1,Nv         # en a1 segundo argumento
        la a2,crc_v      # en a2 tercer argumento
        call Fletcher64

        la a0,W          # en a0 1er argumento
        li a1,Nw         # en a1 segundo argumento
        la a2,crc_w      # en a2 tercer argumento
        call Fletcher64

        j .
```

21) (Junio 2013). Supongamos que definimos que un número natural es “bonito” si es menor que cien mil y además su valor puede obtenerse como una suma de números naturales de la forma 1+2+3+4+5+…
Se pide:

a) Escribir un programa en lenguaje ensamblador del RISC-V tal que dado un número natural N decida si es o no bonito. El programa escribirá en la variable B un 1 si el número es bonito y un 0 en caso contrario.

b) Convertir el código anterior en una subrutina que reciba como entrada un número natural N y devuelva como salida un 1 si el número N es bonito y un 0 si no lo es. Escribir un programa en lenguaje ensamblador del RISC-V que llame a la subrutina, y tal que dado un vector A de M números naturales sea capaz de hallar cuántos números bonitos hay en el vector. El programa debe almacenar la cantidad de números bonitos hallada en la variable “cuenta_bonitos”.

Nota: Se debe respetar el convenio del RISC-V visto en clase para llamadas a subrutinas. Además, en ambos apartados se deben incluir las directivas para reservar memoria y declarar las secciones (.data, .bss y .text) correspondientes.

a) Escribir un programa en lenguaje ensamblador del RISC-V tal que dado un número natural N decida si es o no bonito. El programa escribirá en la variable B un 1 si el número es bonito y un 0 en caso contrario.

```
.global main
.equ N,21
.bss
bonito : .space 4

.text
main :
        mv s0,zero        # bonito = false
        li s1,N
        li s2,100000
        mv s3,zero        # s3 es el acumulador
        mv s4,zero        # s4 es el generador de nºs naturales
do:
        add s3,s3,s4
        addi s4,s4,1
        bge s3,s2,fin     # acumulador >=100000
        bgt s3,s1,fin     # acumulador >n
        beq s3,s1,bontrue
        j do

bontrue:
        addi s0,zero,1 # bonito = true

fin:    la t2,bonito
        sw s0,0(t2)
        j .
.end
```

b) Convertir el código anterior en una subrutina que reciba como entrada un número natural N y devuelva como salida un 1 si el número N es bonito y un 0 si no lo es. Escribir un programa en lenguaje ensamblador del RISC-V que llame a la subrutina, y tal que dado un vector A de M números naturales sea capaz de hallar cuántos números bonitos hay en el vector. El programa debe almacenar la cantidad de números bonitos hallada en la variable "cuenta_bonitos".

```
.global main
.extern _stack
.equ N,5
.data
A: .word 3,5,6,15,13          # array de muestra
M: .word 5                    # longitud del array de muestra
.bss
cuenta_bonitos: .space 4
.text
main :
        la sp,_stack
        la s1,M
        lw s1,0(s1)           # s1 =M
        la s2,A               # s2=@A
        mv s3,zero            # s3 es el índice i del vector A
        mv s4,zero            # s4 es la cuenta de números bonitos
        li s5,1               # s5 guarda un uno para comparaciones
        mv s6,zero            # s6 para construir la dirección de acceso a A
for:
        bge s3,s1,fin_for
        slli s6,s3,2          # s6=s3 *4
        add s6,s6,s2          # @efectiva
        lw a0,0(s6)           # pasa elemento de A como argumento
        call bonito           # llama a la subrutina
        addi s3,s3,1          # i=i+1
        beq a0,s5,exito
        j for
exito:
        addi s4,s4,1          # aumentamos la cuenta de números bonitos
        j for
fin_for :
        la s1,cuenta_bonitos
        sw s4,0(s1)
        j .
```

b) Convertir el código anterior en una subrutina que reciba como entrada un número natural N y devuelva como salida un 1 si el número N es bonito y un 0 si no lo es. Escribir un programa en lenguaje ensamblador del RISC-V que llame a la subrutina, y tal que dado un vector A de M números naturales sea capaz de hallar cuántos números bonitos hay en el vector. El programa debe almacenar la cantidad de números bonitos hallada en la variable “cuenta_bonitos”.

```
bonito : #prólogo
         addi sp,sp,-20
         sw s0,0(sp)
         sw s1,4(sp)
         sw s2,8(sp)
         sw s3,12(sp)
         sw s4,16(sp)

         # cuerpo;
         mv s0,zero          # bonito = false
         li s2,100000
         mv s3,zero          # s3 es el acumulador
         mv s4,zero          # s4 es el generador de nºs naturales
do:
         add s3,s3,s4
         addi s4,s4,1
         bge s3,s2,fin       # acumulador >=100000
         bgt s3,a0,fin       # acumulador >n
         beq s3,a0,bontrue
         j do
bontrue:
         addi s0,zero,1 # bonito = true
fin:
         mv a0,s0
         #epílogo
         lw s0,0(sp)
         lw s1,4(sp)
         lw s2,8(sp)
         lw s3,12(sp)
         lw s4,16(sp)
         addi sp,sp,20
         ret
.end
```

22) (Junio 2014) Dado un vector V de N componentes se dice que es Melchoriforme si posee al menos un elemento Rubio. Un elemento V[i] es Rubio si satisface la siguiente expresión:

$$\sum_{j=0}^{N-1} \mathrm{v[j]} = 2 * \mathrm{v[i]}$$

Se pide:

a) Una subrutina SumaVector(V,N) que sume los N elementos del vector V, respetando el convenio de llamadas a subrutinas visto en clase.

b) Un programa que dado un vector V y su dimensión N decida si es Melchoriforme, utilizando la subrutina SumaVector.

a)

```
sumavector :
        #prólogo
        addi sp,sp,-24
        sw s0,0(sp)
        sw s1,4(sp)
        sw s2,8(sp)
        sw s3,12(sp)
        sw s4,16(sp)
        sw s5,20(sp)
        # cuerpo
        mv s0,a0 #s0=@V
        mv s1,a1 #s1=n
        mv s2,zero          #s2=i
        mv s5,zero          # acumulador
for:
        bge s2,s1,fin_for
        slli s3,s2,2
        add s3,s3,s0
        lw s4,0(s3)
        add s5,s5,s4
        addi s2,s2,1
        j for
fin_for :
        mv a0,s5
        #epílogo
        lw s0,0(sp)
        lw s1,4(sp)
        lw s2,8(sp)
        lw s3,12(sp)
        lw s4,16(sp)
        lw s5,20(sp)
        addi sp,sp,24
        ret
.end
```

22) (Junio 2014) Dado un vector V de N componentes se dice que es Melchoriforme si posee al menos un elemento Rubio. Un elemento V[i] es Rubio si satisface la siguiente expresión:

$$\sum_{j=0}^{N-1} v[j] = 2 * v[i]$$

Se pide:

a) Una subrutina SumaVector(V,N) que sume los N elementos del vector V, respetando el convenio de llamadas a subrutinas visto en clase.

b) Un programa que dado un vector V y su dimensión N decida si es Melchoriforme, utilizando la subrutina SumaVector.

b)

```
.global main
.extern _stack
.equ n,6
.data
V: .word 2,1,10,2,2,3
.bss
melcho : .zero 4
.text
main :
        la sp,_stack
        mv fp,zero
        la a0,V
        li a1,n
        call sumavector
        mv s1,a0 #s1 contiene el sumatorio del vector
        la t1,V          #s1=@V
        mv t2,zero       #t2=i
        li t3,n
do:
        slli t3,t2,2
        add t3,t3,t1
        lw s2,0(t3)
        slli s2,s2,1
        beq s2,s1,meltrue
        addi t2,t2,1 #i++
        bge t2,t3,fin
        j do
meltrue :
        addi s1,zero,1
        la t1,melcho
        sw s1,0(t1)
fin:
        j .
```

23) (Septiembre 2014) Un vector V de N números naturales es noeliano si es una secuencia monótona creciente y sus elementos suman en total 45. Por ejemplo: 0-1-2-3-4-5-6-7-8-9 es noeliano porque $0 \leq 1 \leq 2 \leq 3 \leq 4 \leq 5 \leq 6 \leq 7 \leq 8 \leq 9$ y $1+2+3+4+5+6+7+8+9=45$. También: 3-5-5-7-10-15 es noeliano, ya que $3 \leq 5 \leq 5 \leq 7 \leq 10 \leq 15$ y $3+5+5+7+10+15=45$. Se pide:

a) Escribir en ensamblador de RISC-V una subrutina Sum45(A, N) que reciba la dirección de comienzo de un vector A como primer parámetro, el número N de elementos del vector como segundo parámetro y devuelva 1 si su suma es 45 y 0 en otro caso. La subrutina debe programarse de acuerdo con el estándar de llamadas a subrutinas que hemos estudiado en clase.

b) Escribir un programa RISC-V que, utilizando la subrutina anterior, determine si un vector de entrada es noeliano o no.

a)

```
Sum45 :
        #prólogo
        addi sp,sp,-20
        sw s0,0(sp)
        sw s1,4(sp)
        sw s3,8(sp)
        sw s4,12(sp)
        sw s5,16(sp)

        # cuerpo
        mv s0,a0          # s0=@v
        mv s1,a1          # s1=n
        mv s3,zero        # s3 es i
        mv s5,zero        # s5 es acumulador
        # suma del vector
for:
        bge s3,s1,fin_for
        slli s4,s3,2
        add s4,s4,s0
        lw s4,0(s4)
        add s5,s5,s4
        addi s3,s3,1
        j for
```

23) (Septiembre 2014) Un vector V de N números naturales es noeliano si es una secuencia monótona creciente y sus elementos suman en total 45. Por ejemplo: 0-1-2-3-4-5-6-7-8-9 es noeliano porque $0 \leq 1 \leq 2 \leq 3 \leq 4 \leq 5 \leq 6 \leq 7 \leq 8 \leq 9$ y $1+2+3+4+5+6+7+8+9=45$. También: 3-5-5-7-10-15 es noeliano, ya que $3 \leq 5 \leq 5 \leq 7 \leq 10 \leq 15$ y $3+5+5+7+10+15=45$. Se pide:

a) Escribir en ensamblador de RISC-V una subrutina Sum45(A, N) que reciba la dirección de comienzo de un vector A como primer parámetro, el número N de elementos del vector como segundo parámetro y devuelva 1 si su suma es 45 y 0 en otro caso. La subrutina debe programarse de acuerdo con el estándar de llamadas a subrutinas que hemos estudiado en clase.

b) Escribir un programa RISC-V que, utilizando la subrutina anterior, determine si un vector de entrada es noeliano o no.

a)

```
# comparación de la suma con 45
fin_for :
        addi s3,zero,45
        bne s5,s3,else
        addi s4,zero,1
        mv a0,s4          # a0 =1 suma es 45
        j fin_sr
else :
        mv a0,zero

        #epílogo
fin_sr :
        lw s0,0(sp)
        lw s1,4(sp)
        lw s3,8(sp)
        lw s4,12(sp)
        lw s5,16(sp)
        addi sp,sp,20
        ret
.end
```

23) (Septiembre 2014) Un vector V de N números naturales es noeliano si es una secuencia monótona creciente y sus elementos suman en total 45. Por ejemplo: 0-1-2-3-4-5-6-7-8-9 es noeliano porque $0 \leq 1 \leq 2 \leq 3 \leq 4 \leq 5 \leq 6 \leq 7 \leq 8 \leq 9$ y $1+2+3+4+5+6+7+8+9=45$. También: 3-5-5-7-10-15 es noeliano, ya que $3 \leq 5 \leq 5 \leq 7 \leq 10 \leq 15$ y $3+5+5+7+10+15=45$. Se pide:

a) Escribir en ensamblador de RISC-V una subrutina Sum45(A, N) que reciba la dirección de comienzo de un vector A como primer parámetro, el número N de elementos del vector como segundo parámetro y devuelva 1 si su suma es 45 y 0 en otro caso. La subrutina debe programarse de acuerdo con el estándar de llamadas a subrutinas que hemos estudiado en clase.

b) Escribir un programa RISC-V que, utilizando la subrutina anterior, determine si un vector de entrada es noeliano o no.

b)

```
.global main
.extern _stack
.equ n,6
.data
V: .word 3,5,5,7,10,15
.bss
noeliano : .space 4
.text
main :
        la sp,_stack
        la a0,V
        li a1,n
        call Sum45
        mv s0,a0  # s0 resultado de la suma45
        addi s3,zero,1      # s3 = 1
        bne s0,s3,falso # no cumple la condición suma =45
        la t0,V             # t0 =@V
        li t3,n-1           # t3=n-1 (n-1 comparaciones creciente)
        mv t1,zero          # t1=i=0
do:
        slli t2,t1,2        # i *4
        add t2,t2,t0        # t2 = @i
        lw s4,0(t2)
        lw s5,4(t2)
        bgt s4,s5,falso # no cumple y salta
        addi s3,zero,1      # noeliano = true
        la t4,noeliano
        sw s3,0(t4)
        addi t1,t1,1        # i++
        blt t1,t3,do        # i<n
        j fin
falso :
        la t1,noeliano
        sw zero,0(t1)
fin:
        j .
```