

# LEGv8 avanzado

OdC - 2022

# Instrucciones para tomar decisiones

Podemos separar las instrucciones de saltos condicionales en dos grandes grupos:

- **CBZ / CBNZ:** El salto se efectúa o no, dependiendo del contenido de un registro que se pasa como argumento.
- **B.cond:** El salto se efectúa o no dependiendo del estado de las banderas del procesador.

# CBZ: Compare and Branch if Zero

**CBZ:** El salto se realiza si el registro es cero

**CBZ** register, L1

```
        CBZ  x0, label
        ADDI x1, x1, #8
label:  SUBI  x0, x0, #1
```

```
label:  SUBI  x0, x0, #1
        ADDI x1, x1, #8
        CBZ  x0, label
```

# CBNZ: Compare and Branch if Not Zero

CBNZ: El salto se realiza si el registro NO es cero

CBNZ register, L1

CBNZ x0, label	label: SUBI x0, x0, #1
ADDI x1, x1, #8	ADDI x1, x1, #8
label: SUBI x0, x0, #1	CBNZ x0, label

# Ejercicio 1 - b

```
MOV  X9, X0          // X9 = X0 (Se copia el valor de X0 a X9)
MOV  X0, XZR          // X0 = 0  (Se copia el valor de XZR a X0)
loop: ADD X0, X0, X9    // x0 = x0 + x9
      SUBI X9, X9, #1   // x9 = x9 - 1 (Resto 1 a x9)
      CBNZ X9, loop     // Si x9 NO es cero, salto a las instrucción add
done:
```

# Ejercicio 1 - b

```
MOV  X9, X0          // X9 = X0 (Se copia el valor de X0 a X9)
MOV  X0, XZR          // X0 = 0 (Se copia el valor de XZR a X0)*
loop: ADD X0, X0, X9   // x0 = x0 + x9
      SUBI X9, X9, #1  // x9 = x9 - 1 (Resto 1 a x9)
      CBNZ X9, loop    // Si x9 NO es cero, salto a las instrucción add
done:
```

Si inicialmente  $X0 = 10$

Iteración	0*	1	2	3	4	5	6	7	8	9
Valor de X9	10	9	8	7	6	5	4	3	2	1
Valor de X0	0	10	19	27	34	40	45	49	52	55

$$X0 = 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 55$$

$$X0 = X0 (X0 + 1) / 2$$

# B.cond: Branch if condition

B.cond *label*

¿Y donde estan los registros que estoy comparando?

1

Condiciones:

- = Igual
- $\neq$  Distinto
- > Mayor que
- $\geq$  Mayor o igual que
- < Menor que
- $\leq$  Menor o igual que

¿Como distingo si los números que estoy comparando son o no signado?

2

# 1 Comparación

- La comparación debe realizarse antes de ejecutarse la instrucción de salto
- Las únicas instrucciones que pueden utilizarse para la comparación son:
  - ADDS: ADD and set flags  $R[Rd], \text{FLAGS} = R[Rn] + R[Rm]$
  - ADDIS: ADD Immediate and set flags  $R[Rd], \text{FLAGS} = R[Rn] + \text{ALUImm}$
  - ANDS: AND and set flags
  - ANDIS: AND Immediate and set flags
  - SUBS: SUB and set flags
  - SUBIS: SUB Immediate and set flags
  - CMP: CoMPare  $\text{FLAGS} = R[Rn] - R[Rm]$
  - CMPI: CoMpare Immediate  $\text{FLAGS} = R[Rn] - \text{ALUImm}$



# 1 FLAGS

Banderas que reflejan algunas características particulares del resultado de una operación. Estas valen 1 cuando:

- **Negative (N)**: El resultado es negativo
- **Zero (Z)**: El resultado es cero
- **Carry (C)**: Existe un carry de salida o de entrada del bit más significativo.  
(Para números **no signados**)
- **Overflow (V)**: El resultado de una operación **signada** genera overflow

El resultado se almacena en el registro CPSR (Current Program Status Register)

Bit 31	Bit 30	Bit 29	Bit 28	...
Negative	Zero	Carry	Overflow	...

# 1 Carry (Unsigned)

Las reglas para que la bandera de carry sea uno, son:

1. La suma de dos números genera un carry de salida del bit más significativo.

$$1111 + 0001 = 0000$$

2. La resta de dos números requiere un préstamo (carry in) en los bits más significativos restados.

$$0000 - 0001 = 1111$$

En cualquier otro caso, la bandera es cero:

$$0111 + 0001 = 1000$$

$$1000 - 0001 = 0111$$

# 1 Overflow (Signed)

Las reglas para que la bandera de overflow sea uno, son:

1. La suma de dos números positivos da como resultado un número negativo  
 $0100 + 0100 = 1000$
2. La suma de dos números negativos da como resultado un número positivo  
 $1000 + 1000 = 0000$

En cualquier otro caso, la bandera es cero:

$$\begin{aligned}0100 + 0001 &= 0101 \\0110 + 1001 &= 1111 \\1000 + 0001 &= 1001 \\1100 + 1100 &= 1000\end{aligned}$$

# 1 Overflow (Signed)

Operación	Operando A	Operando B	Resultado que genera Overflow
A + B	$\geq 0$	$\geq 0$	$< 0$
A + B	$< 0$	$< 0$	$\geq 0$
A - B	$\geq 0$	$< 0$	$< 0$
A - B	$< 0$	$\geq 0$	$\geq 0$

## 2 Instrucciones de salto

	Signed numbers		Unsigned numbers	
Comparison	Instruction	CC Test	Instruction	CC Test
=	B.EQ	Z=1	B.EQ	Z=1
≠	B.NE	Z=0	B.NE	Z=0
<	B.LT	N! = V	B.LO	C=0
≤	B.LE	$\sim(Z=0 \ \& \ N=V)$	B.LS	$\sim(Z=0 \ \& \ C=1)$
>	B.GT	$(Z=0 \ \& \ N=V)$	B.HI	$(Z=0 \ \& \ C=1)$
≥	B.GE	N=V	B.HS	C=1

Signed and Unsigned numbers	
Instruction	CC Test
Branch on minus (B.MI)	N= 1
Branch on plus (B.PL)	N= 0
Branch on overflow set (B.VS)	V= 1
Branch on overflow clear (B.VC)	V= 0

# Ejercicio 1 a

```
        SUBS XZR, XZR, X0        // FLAGS = 0 - X0 (CMPI X0, 0)
        B.LT else                // Si X0 es menor que 0 salto a else
        B done                   // Salto incondicional a done
else: SUB X0, XZR, X0            // X0 = 0 - X0
done:
```

- El salto condicional se toma si el valor almacenado en X0 es menor que cero, es decir, es negativo.
- En este caso, antes de finalizar se ejecuta la instrucción SUB donde se cambia el signo de X0
- Si no se toma el salto condicional, la instrucción *branch* me lleva directamente al fin del programa.

Este programa devuelve el **módulo** del valor almacenado en X0

## Ejercicio 2

```
.data
a: .dword 0x0000000000000001
b: .dword 0x0000000000010100    //b: .dword 0x8000000000000100
.text
        SUBIS XZR, X9, #0        // FLAGS = x9 - 0
        B.GE else                // Salto a "else" X9 es mayor a cero
        B done                   // Salto incondicional a "done"
else:    ORRI X10, XZR, #2        // X10 = 0 || 2
done:
```

- El salto condicional depende del valor de X9, si este es mayor a cero se toma el salto.
- El valor de X10 depende de si el salto fue tomado o no.

# Ejercicio 4

```
loop:    ADDI X0, X0, #2    // X0 = X0 + 2
        SUBI X1, X1, #1    // X1 = X1 - 1
        CBNZ X1, loop      // Si X1 no es cero, salto a loop
```

done:

```
loop:    SUBIS X1, X1, #0    // X1 = X1 - 0 (Seteo de flags)
        B.LE done          // Si X1 es menor o igual que 0 salto a done
        SUBI X1, X1, #1     // X1 = X1 + 1
        ADDI X0, X0, #2     // X0 = X0 + 2
        B loop             // Salto incondicional a loop
```

done:

```
for (long i=N; 0<=i; --i)
    acc+=2;
```



## Ejercicio 5 - a

X10  $\leftrightarrow$  i  
X1  $\leftrightarrow$  a  
X2  $\leftrightarrow$  result  
X0  $\leftrightarrow$  &MemArray[0].

```
i = 0;
do
{
    result += MemArray[i];
    i++;
}
while (i<100)
```

```
ADDI X10, XZR, 100 // i = 0
loop: LDUR X1, [X0,#0] // a = MemArray[0]
      ADD X2, X2, X1 // result += a
      ADDI X0, X0, #8 // X0 = &MemArray[0] + 8 = &MemArray[1]
      SUBI X10, X10, #1 // i += 1
      //CMPI X10, #100 // FLAGS = X10 - 100
      cbnz loop // goto loop si X10 < 100
```

## Ejercicio 5 - b

X10  $\leftrightarrow$  i  
X1  $\leftrightarrow$  a  
X2  $\leftrightarrow$  result  
X0  $\leftrightarrow$  &MemArray[0].

```
i = 50;
do
{
    result += MemArray[100 - i*2];
    result += MemArray[100 - i*2 + 1];
    i--;
}
while (i!=0)
```

```

                ADDI X10, XZR, #50 // i = 50
loop:          LDUR X1, [X0,#0]    // a = MemArray[0]
                ADD X2, X2, X1     // result += a
                LDUR X1, [X0,#8]   // a = MemArray[1]
                ADD X2, X2, X1     // result += a
                ADDI X0, X0, #16   // x0 = x0 + 16 = &MemArray[2]
                SUBI X10, X10, #1  // i -= 1
                CBNZ X10, loop     // salta si x10 != 0
```

# Ejercicio 3: Enunciado

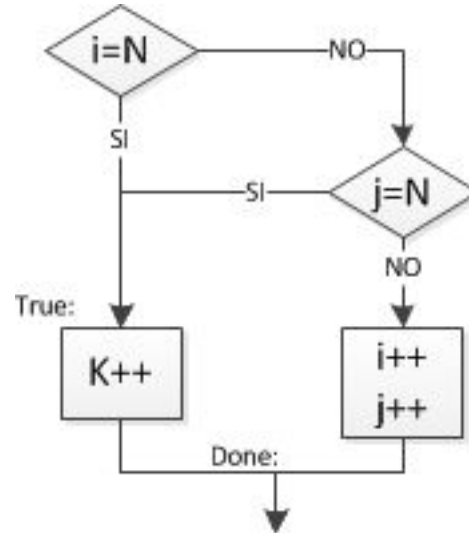
Dado el siguiente programa “C” y la asignación  $i, j, k, N \leftrightarrow X0, X1, X2, X9$ , escribir el programa LEGv8 que lo implementa. Notar que como se usa el operador `||` la evaluación es por cortocircuito. **Opcional:** hacerlo con el operador `|` que no está cortocircuitado.

```
long long int i,j,k,N;

if (i==N || j==N) {
    ++k;
} else {
    ++i; ++j;
}
```

# Ejercicio 3: Cortocircuitado

```
long long int i,j,k,N;  
  
if (i==N || j==N) {  
    ++k;  
} else {  
    ++i; ++j;  
}
```



# Ejercicio 3: Cortocircuitado

```
long long int i,j,k,N;
```

```
if (i==N || j==N) {
```

```
    ++k;
```

```
} else {
```

```
    ++i; ++j;
```

```
}
```

x0 ↔ i

x1 ↔ j

x2 ↔ k

x3 ↔ N

True:

end:

```
sub x9, x0, x3    // comparo i con N y lo guardo en x9
cbz x9, True      // Si son iguales salto a True
sub x10, x1, x3   // comparo j con N y lo guardo en x10
cbz x9, True      // Si son iguales salto a True
add x0, x0, #1    // ++i
add x1, x1, #1    // ++j
b end
addi x2, x2, #1   // ++k
```

# Ejercicio 3: No cortocircuitado

```
long long int i,j,k,N;
```

```
if (i==N | j==N) {
```

```
    k = 2;
```

```
} else {
```

```
    ++k;
```

```
}
```

```
return(k);
```

```
sub    sp, sp, #32
```

```
str    x0, [sp, 24]
```

```
str    x1, [sp, 16]
```

```
str    x2, [sp, 8]
```

```
str    x3, [sp]
```

```
ldr    x1, [sp, 24]
```

```
ldr    x0, [sp]
```

```
cmp    x1, x0
```

```
cset   w0, eq
```

```
and    w1, w0, 255
```

```
ldr    x2, [sp, 16]
```

```
ldr    x0, [sp]
```

```
cmp    x2, x0
```

```
cset   w0, eq
```

```
and    w0, w0, 255
```

```
orr    w0, w1, w0
```

```
and    w0, w0, 255
```

```
cmp    w0, 0
```

```
beq    .L2
```

```
mov    x0, 2
```

```
str    x0, [sp, 8]
```

```
b      .L3
```

```
//i
```

```
//j
```

```
//k
```

```
//N
```

```
//X1 = i
```

```
//X0 = N
```

```
//if(i==N){
```

```
//    w0 = 1}
```

```
//w1 = w0 and 0xff
```

```
//x2 = j
```

```
//X0 = N
```

```
//if(j==N){
```

```
//    w0 = 1}
```

```
//w0 = w0 and 0xff
```

```
//w0 = w0 or w1
```

```
//w0 = w0 and 0xff
```

```
//if(w0 == 0) {
```

```
//    goto L2}
```

} True

```
.L2:
```

```
ldr    x0, [sp, 8]
```

```
add    x0, x0, 1
```

```
str    x0, [sp, 8]
```

```
.L3:
```

```
ldr    x0, [sp, 8]
```

```
add    sp, sp, 32
```

```
ret
```

} False

x0 ↔ i ↔ [sp, 24]

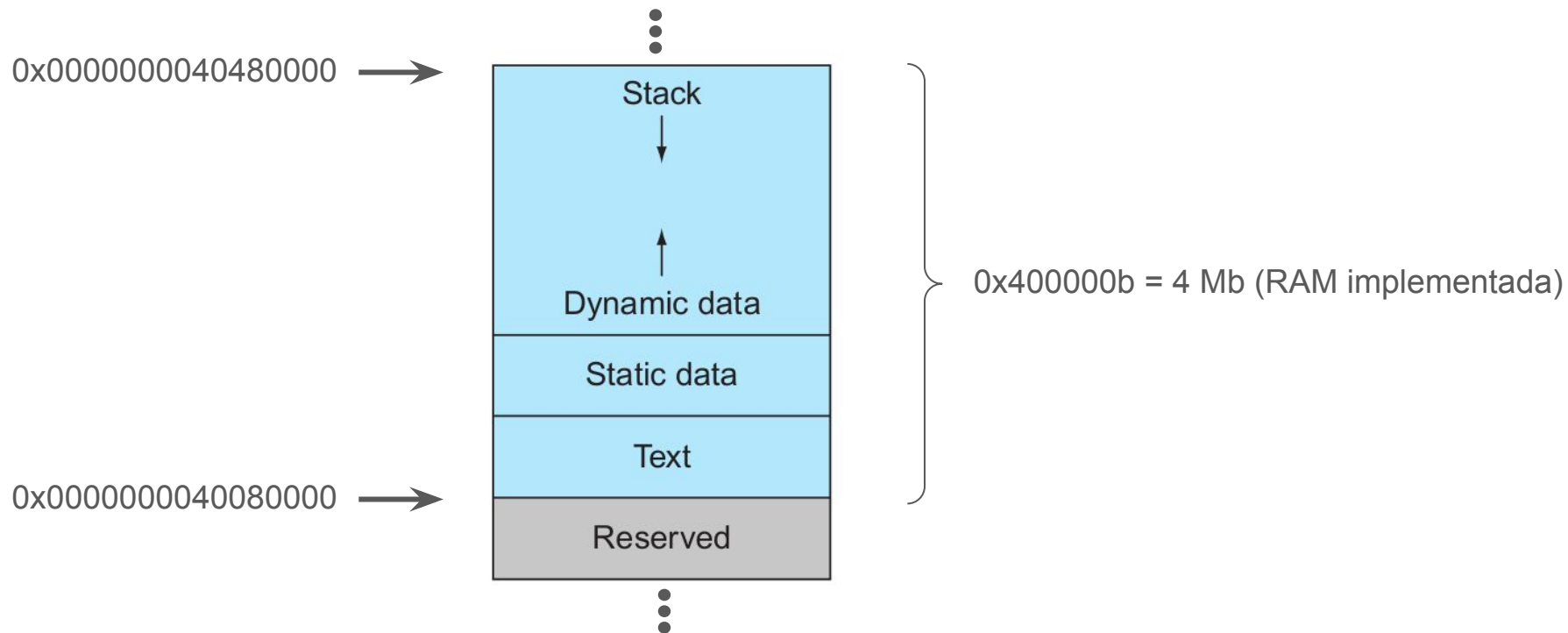
x1 ↔ j ↔ [sp, 16]

x2 ↔ k ↔ [sp, 8]

x3 ↔ N ↔ [sp, 0]

aarch64-linux-gnu-gcc -O0 -S -o- -xc ej3.c > main.s

# QEMU - Memory map



# Ejercicio 6: Enunciado

Traducir el siguiente programa en “C” a ensamblador LEGv8 dada la asignación de variables a registros X0, X1, X2, X9  $\leftrightarrow$  str, found, i, N. El número 48 se corresponde con el carácter ‘0’ en ASCII, por lo tanto el programa cuenta la cantidad de ‘0’s que aparecen en una cadena de caracteres de longitud N.

```
#define N (1<<10)
char *str;
long found, i;
for (found=0, i=0; i!=N; ++i)
    found += (str[i]==48);
```



# Ejercicio 6: Resolución

```
#define N (1<<10)
char *str;
long found, i;
for (found=0, i=0; i!=N; ++i)
    found += (str[i]==48);
```

X0 ↔ str
X1 ↔ found
X2 ↔ i
X9 ↔ N

```
.data
str: .dword 0x754d30616c6f4830, 0x00000000306f646e
N: .dword 15
offset: .dword 0x40080000
.text
ldr X0, =str
ldr X10, offset
ldr X9, N                // N = 15
add X0, X0, X10           // X0 = &str[0]
add X1, XZR, XZR          // found = 0
add X2, XZR, XZR          // i = 0
for: cmp X2, X9            // comparo i y N
    b.eq end              // Salto si son iguales
    add X11, X0, X2        // X11 = &str[0] + i
    ldurb W12, [X11, #0]   // X12 = str[i]
    cmp W12, #48           // Verifico si el byte que traje es un 0
    b.ne skip             // Si son distintos no lo cuento
    add X1, X1, #1         // Si es un cero, found +=1
skip: add X2, X2, #1       // i = i + 1
    B for
end:
```

# Ejercicio 7: Enunciado

Traducir el siguiente programa “C” a LEGv8. La asignación de variables a registros X0, X1, X2, X3, X9  $\leftrightarrow$  A, s, i, j, N. Notar que en “C” los arreglos bidimensionales se representan en memoria usando un **orden por filas**, es decir  $\&A[i][j] = A + 8*(i*N+j)$ .

```
#define N (1<<10)
long A[N][N], s, i, j;
s=0;
for (i=0; i<N; ++i)
    for (j=0; j<N; ++j)
        s += A[i][j];
```

X0 $\leftrightarrow$ &A[0]
X1 $\leftrightarrow$ s
X2 $\leftrightarrow$ i
X3 $\leftrightarrow$ j
X9 $\leftrightarrow$ N

A[2][3] =

1	7	2
44	3	21

A ->


1	7	2	44	3	21
---	---	---	----	---	----

# Ejercicio 7.0: Resuelto

```
.data
    N: .dword 3
    dirBase: .dword 0x0000000040080000
    A: .dword 1,7,2,44,3,21,1,2,3 // A[N][N]

.text
    ldr X0, =A           // x0 =&A[0][0] (relativo)
    ldr X9, N             // N=3
    ldr X10, dirBase
    add X0, X0, X10       // x0 =&A[0][0] (absoluto)
    add X1, XZR, XZR      // s=0
    add X2, XZR, XZR      // i=0
    add X3, XZR, XZR      // j=0
```

X0	↔	&A[0]
X1	↔	s
X2	↔	i
X3	↔	j
X9	↔	N



```
oLoop:    cmp X2,X9      // if(i==N)
          b.eq oEnd      // goto oEnd;
          add X3, XZR, XZR // j=0
iLoop:    cmp X3,X9      // if(j==N)
          b.eq iEnd      // goto iEnd;
          mul X12, X2, X9 // X12 = i * N
          add X12, X12, X3 // X12 = (i * N) + j
          lsl x12, x12, #3 // X12 = ((i * N) + j) * 8
          add X12, X12, X0 // X12 = &A[0][0] + ((i * N) + j) * 8
          ldur X11, [X12,#0] // X11=A[i][j]
          add X1, X1, X11    // s+=A[i][j]
          addi X3, X3, #1    // j++;
          b iLoop
iEnd:     addi X2, X2, #1    // i++;
oEnd:
```

# Ejercicio 7.1: Enunciado

Traducir el siguiente programa “C” a LEV8. La asignación de variables a registros  $X0, X1, X2, X3, X9 \leftrightarrow A, s, i, j, N$ . Notar que en “C” los arreglos bidimensionales se representan en memoria usando un **orden por filas**, es decir  $\&A[i][j] = A + 8*(i*N+j)$ .

7.1) Hacer lineal el acceso al arreglo y recorrerlo con un solo lazo.

```
#define N (1<<10)
long A[N][N], s, i, j;
s=0;
newN = N * N
for (i=0; i<newN; ++i)
    s += A[i];
```

$X0 \leftrightarrow \&A[0]$
$X1 \leftrightarrow s$
$X2 \leftrightarrow i$
<del><math>X3 \leftrightarrow j</math></del>
$X9 \leftrightarrow N$

$A[2][3] =$

1	7	2
44	3	21

$A \rightarrow$

1	7	2	44	3	21
---	---	---	----	---	----

# Ejercicio 7.1: Resuelto


```
.data
N: .dword 3
dirBase: .dword 0x0000000040080000
A: .dword 1,7,2,44,3,21,1,2,3 // A[N][N]

.text
ldr X0, =A           // x0 =&A[0][0] (relativo)
ldr X9, N             // N=3
ldr X10, dirBase
add X0, X0, X10       // x0 =&A[0][0] (absoluto)
add X1, XZR, XZR      // s=0
add X2, XZR, XZR      // i=0
mul X9, X9, X9        // newN = N * N
```

X0 ↔ &A[0]  
X1 ↔ s  
X2 ↔ i  
~~X3 ↔ j~~  
X9 ↔ N



oLoop:



```
cmp X2,X9             // if(i==N)
b.eq oEnd             // goto oEnd;
add X12, XZR, X2      // X12 = i
lsl x12, x12, #3       // X12 = i * 8
add X12, X12, X0       // X12 = &A[0] + i * 8
ldur X11, [X12,#0]     // X11=A[i]
add X1, X1, X11        // s+=A[i]
addi X2, X2, #1        // i++
b oLoop
```

oEnd:

## Ejercicio 7.2: Enunciado

Traducir el siguiente programa “C” a LEV8. La asignación de variables a registros X0, X1, X2, X3, X9  $\leftrightarrow$  A, s, i, j, N. Notar que en “C” los arreglos bidimensionales se representan en memoria usando un **orden por filas**, es decir  $\&A[i][j] = A + 8*(i*N+j)$ .

7.2) Se puede hacer lo mismo sin usar ninguna variable índice i, j.

```
#define N (1<<10)
long A[N][N], s, i, j;
s=0;
finalAddress = &A[0][0] + (N * N)
for (i=0; i!=finalAddress; ++i)
    s += A[i];
```

X0 $\leftrightarrow$ &A[0]
X1 $\leftrightarrow$ s
<del>X2 <math>\leftrightarrow</math> i</del>
<del>X3 <math>\leftrightarrow</math> j</del>
X9 $\leftrightarrow$ N

A[2][3] =

1	7	2
44	3	21

A ->

1	7	2	44	3	21
---	---	---	----	---	----

# Ejercicio 7.2: Resuelto

```
.data
    N: .dword 3
    dirBase: .dword 0x0000000040080000
    A: .dword 1,7,2,44,3,21,1,2,3 // A[N][N]

.text
    ldr X0, =A           // x0 =&A[0][0] (relativo)
    ldr X9, N             // N=3
    ldr X10, dirBase
    add X0, X0, X10       // x0 =&A[0][0] (absoluto)
    add X1, XZR, XZR      // s=0
    mul X9, X9, X9        // X9= N * N
    lsl x9, x9, 3         // X9= N * N * 8
    add X9, x9, X0        // finalAddr = &A[0][0] + (N*N*8)
```

X0 ↔ &A[0]  
X1 ↔ s  
~~X2 ↔ i~~  
~~X3 ↔ j~~  
X9 ↔ N



oLoop:



```
cmp X0,X9           // if(i==finalAddr)
b.eq oEnd           // goto oEnd;
ldur X11, [X0,#0]    // X11=A[i]
add X1, X1, X11      // s+=A[i]
addi X0, X0, #8       // i++
b oLoop
```

oEnd: