Instrucciones

(el lenguaje *assembly* RISC-V)

Arquitectura de Computadores

Todo computador debe tener instrucciones para ejecutar las operaciones aritméticas básicas; p.ej.:

```
add a,b,c
```
—suma las variables **b** *y* **c** *y pone la suma en* **a**

La notación (a este nivel) es rígida:

- cada instrucción aritmética ejecuta sólo una operación y siempre debe tener exactamente tres variables

P.ej., para sumar cuatro variables —**b**, **c**, **d** y **e**— y poner la suma en **a** se necesitan tres instrucciones, cada una en una línea por sí sola:

```
add a,b,c
add a,a,d
add a,a,e
```

P.ej., ¿cómo traduciría el compilador el siguiente segmento de un programa en C o Python a instrucciones del lenguaje assembly?

```
a ← b + c
d ← a – e
```

Respuesta:

```
add a,b,c
sub d,a,e
```

P.ej., ¿qué haría el compilador en el caso de la siguiente sentencia?

```
f ← (g+h) – (i+j)
```

Respuesta:

```
add t0,g,h
add t1,i,j
sub f,t0,t1
```

`t0` y **`t1`** son variables auxiliares para almacenar temporalmente los resultados de **`g+h`** e **`i+j`** antes de hacer la resta

A diferencia de un programa en C o Java, los tres operandos de las instrucciones aritméticas deben corresponder a **registros**:

- hay un número limitado de registros, p.ej., 32
- en RISC-V, son de 64 bits de largo c/u

Para los nombres de los registros, RISC-V usa una **x** seguida del número del registro (un número entre 0 y 31)

P.ej., si las variables **f**, **g**, **h**, **i** y **j** están asignadas a los registros **x19**, **x20**, **x21**, **x22** y **x23**

… y usamos los registros **x5** y **x6** para almacenar los resultados intermedios

… entonces el código compilado para **f ← (g + h) – (i + j)** sería

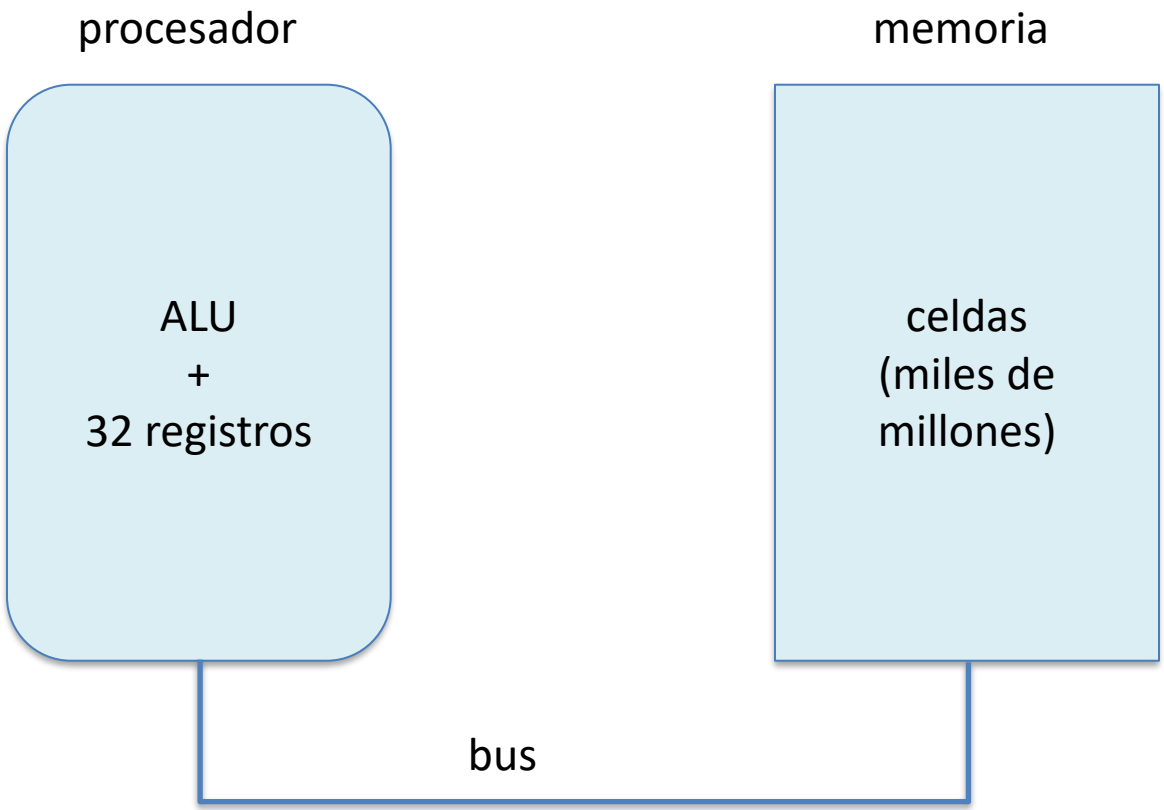
```
add x5,x20,x21
add x6,x22,x23
sub x19,x5,x6
```

Los lenguajes de programación también tienen arreglos (y otras estructuras), que pueden contener muchos más datos que la cantidad de registros del computador

Estas estructuras, que simplemente no caben en 32 registros, se almacenan en la memoria del computador:

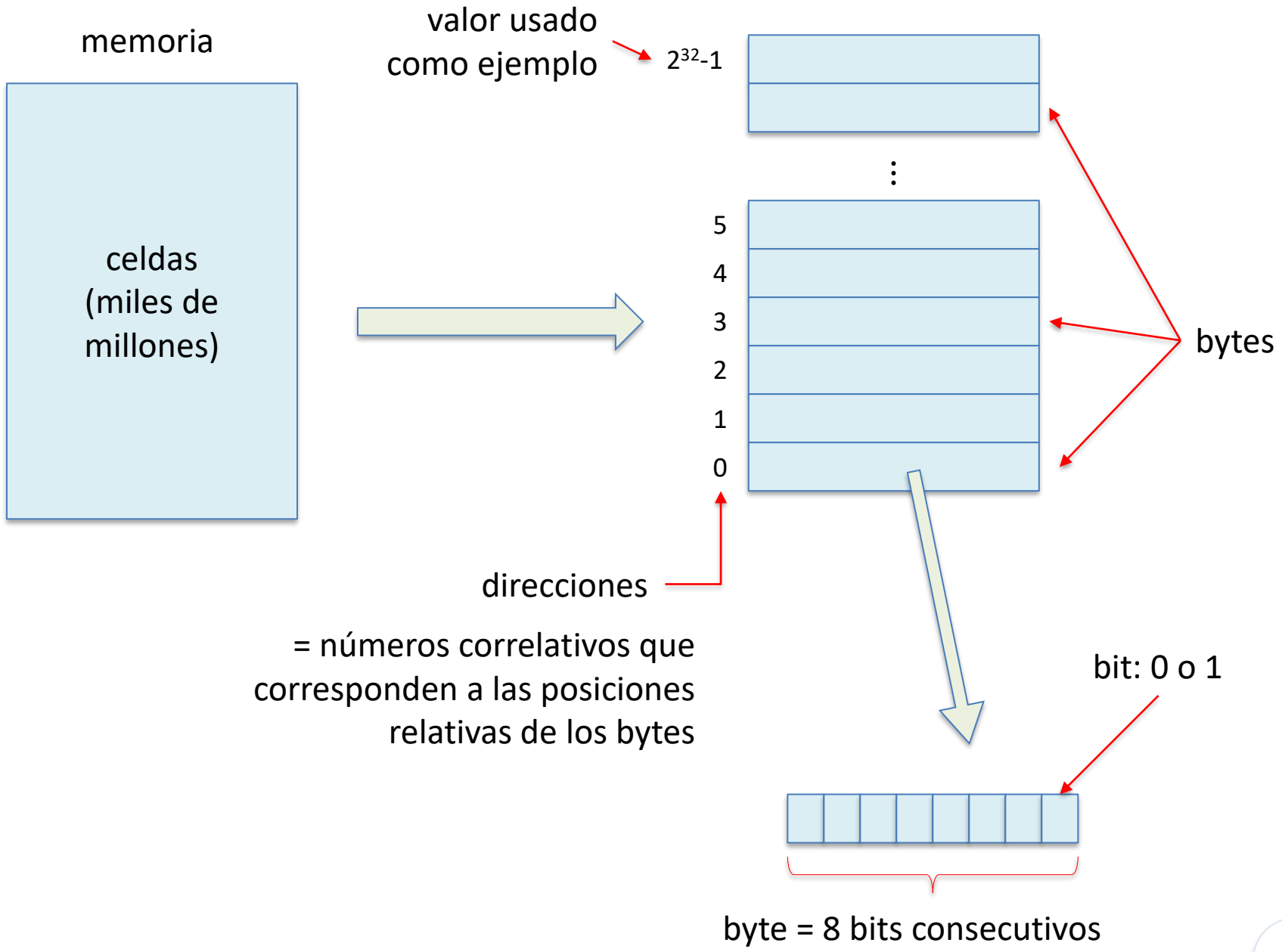
- que tiene capacidad para almacenar miles de millones de datos

… → el lenguaje assembly debe tener instrucciones para *transferir datos entre la memoria y los registros*: **`ld`** y **`sd`**



La memoria es como un arreglo unidimensional muy grande de celdas o casillas, en que la *dirección* (de memoria) actúa como índice:

- **`ld`** y **`sd`** deben especificar la dirección de la celda involucrada
- **`ld`** y **`sd`** especifican la dirección de modo *indirecto*, por una constante y un registro → la suma de la constante y el contenido del registro


memoria
celdas
(miles de
millones)
valor usado
como ejemplo
$2^{32}-1$
5
4
3
2
1
0
bytes
direcciones
= números correlativos que
corresponden a las posiciones
relativas de los bytes
bit: 0 o 1
byte = 8 bits consecutivos

`ld` : copia datos desde la memoria a un registro (*load doubleword*)

P.ej., ¿cómo traduce el compilador la sentencia **g ← h + A[8]** en C?

- el compilador coloca el arreglo **A** en memoria → sabe cuál es la dirección de memoria de **`A[0]`** (el primer elemento de A) y coloca esta dirección en un registro, llamado el *registro base*; supongamos que el registro base es **`x22`**
- el compilador asocia variables con registros → supongamos que las variables **g** y **h** están asociadas a los registros **`x20`** y **`x21`**

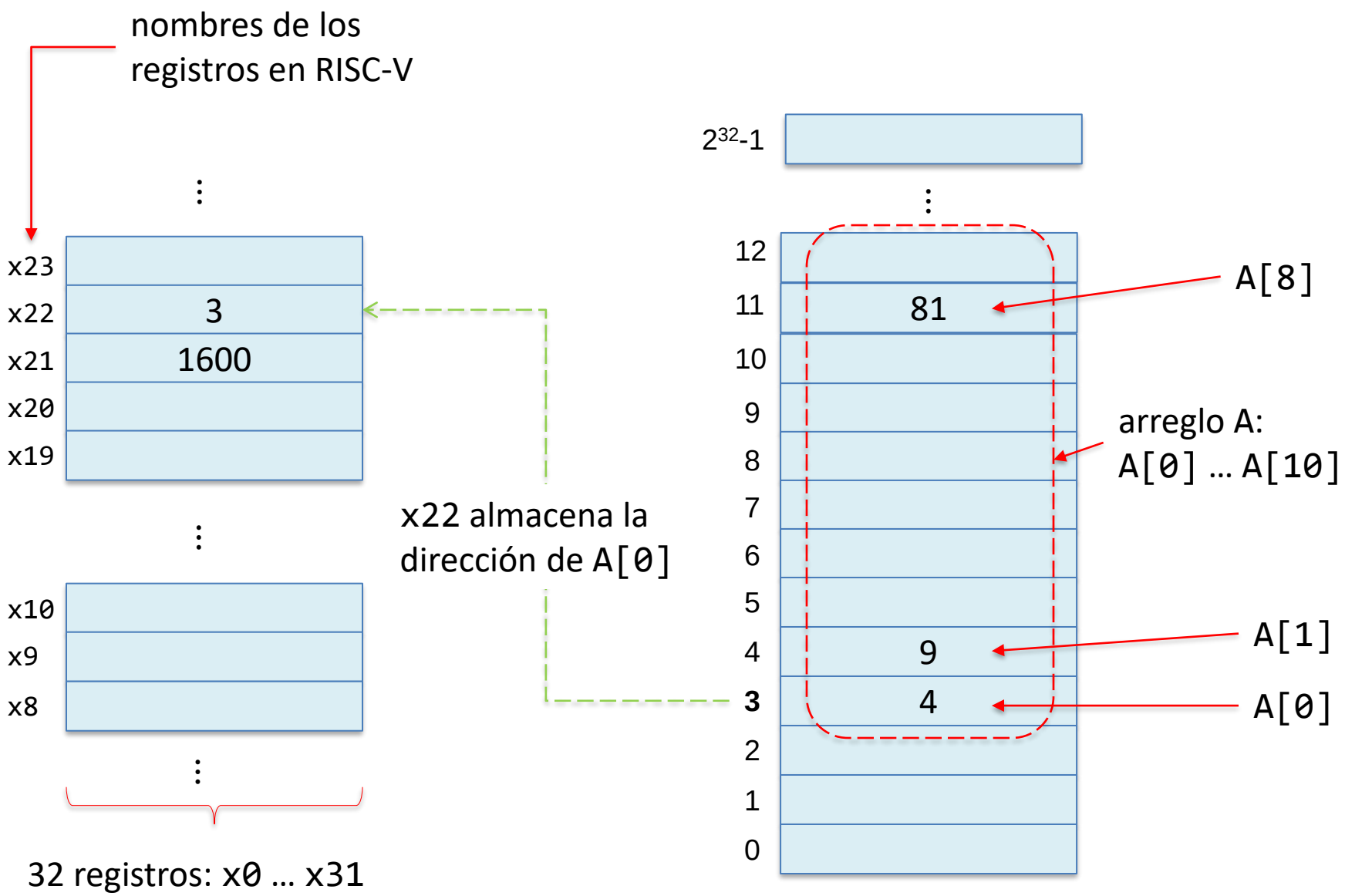
Entonces **g ← h + A[8]** es traducida por el compilador así:

```
ld   x9,8(x22)
add  x20,x21,x9
```

—*dos instrucciones en assembly:*

—… *primero, copia* `A[8]` *al registro* `x9`

—… *luego, suma y pone el resultado en* g


nombres de los registros en RISC-V
x23
x22
x21
x20
x19
3
1600
x10
x9
x8
32 registros: x0 … x31
x22 almacena la dirección de A[0]
2^32-1
12
11
10
9
8
7
6
5
4
3
2
1
0
81
9
4
A[8]
arreglo A:
A[0] … A[10]
A[1]
A[0]

La memoria, en la práctica en todos los computadores, está organizada en *bytes* (8 bits consecutivos)

… en que cada byte tiene una dirección: 0, 1, 2, ….

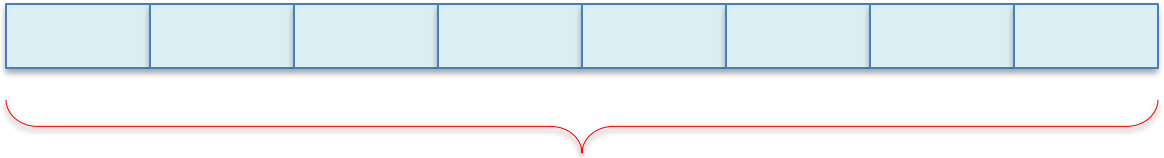
Los computadores modernos, sin embargo, operan sobre *palabras* (*words*) de 4 bytes consecutivos (32 bits),

… o, como en el caso de RISC-V, incluso sobre *palabras dobles* (*double-words*) de 8 bytes consecutivos (64 bits)

… por lo que las direcciones de estas palabras dobles son 0, 8, 16, …

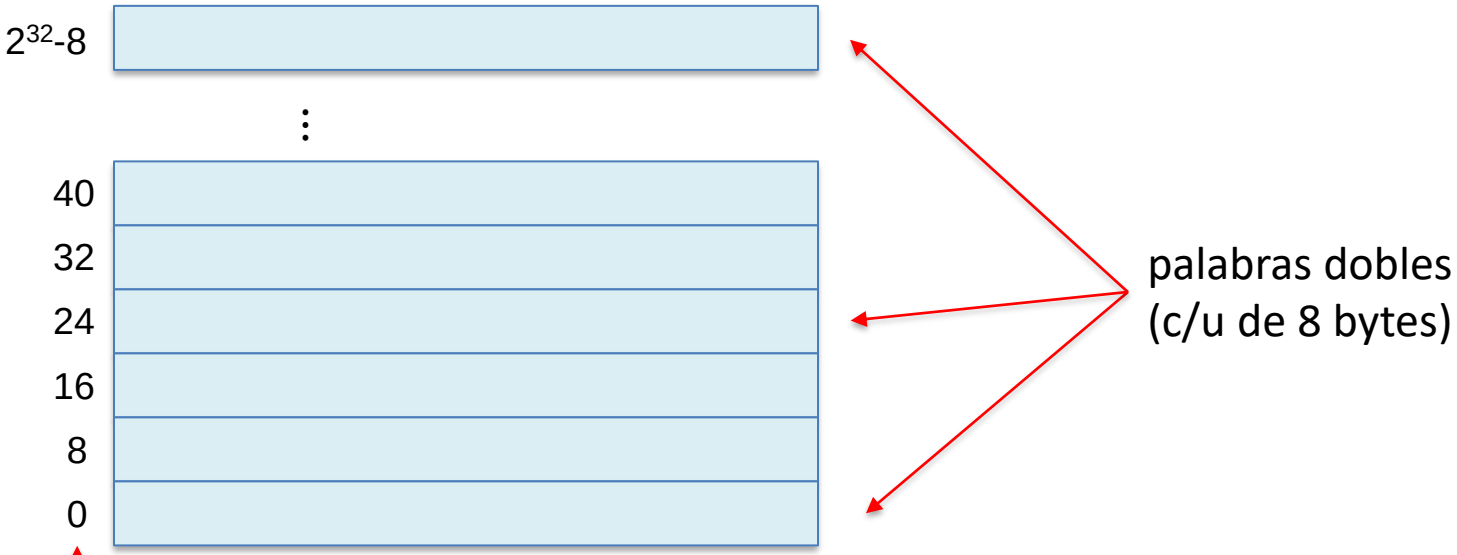
- … es decir, la dirección correspondiente al primer byte de la palabra
- … de modo que el ejemplo anterior requiere un ajuste

Veámoslo con la siguiente instrucción



palabra doble
= 8 bytes consecutivos (64 bits)

memoria RISC-V



direcciones (de las palabras dobles): corresponden
a la dirección de uno de los 8 bytes de la palabra
—el byte con la dirección numéricamente menor

sd : copia datos desde un registro a la memoria (*store doubleword*):

`A[12]` ← `h + A[8]` —*sentencia en C, Java, etc.*

De nuevo, **x22** contiene la dirección de **A[0]** y **x21** está asociado a **h**:

```
                                —tres instrucciones en assembly:
ld   x9,64(x22)                     —… copia (load) A[8] al registro x9
add  x9,x21,x9                      —… suma h a A[8]
sd   x9,96(x22)                     —… copia (store) el registro x9 a A[12]
```

En muchas operaciones, aparecen operandos constantes, o *inmediatos*: el valor numérico propiamente tal

RISC-V tiene versiones de las instrucciones aritméticas en que uno de los operandos es una constante

P.ej., la instrucción *add-immediate*, or **`addi`**:

`addi x22,x22,4` —*corresponde a la sentencia* `x22 ← x22 + 4`

Es útil poder operar sobre conjuntos de bits dentro de una palabra o incluso sobre bits individuales —operaciones lógicas, típicamente:

AND, OR, XOR, NOT, *shift left*, *shift right* y *shift right arithmetic*

Los *shifts* mueven todos los bits de una palabra a la izquierda o a la derecha, llenando los bits que quedan "vacíos" con 0s:

P.ej., si **x19** contiene

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_2$ (= 9_{10})

… y ejecutamos la instrucción

`slli x11,x19,4` —*shift left logical immediate*

… → *shift left* de 4 posiciones y dejamos el resultado en **x11**:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_2$ (= 144_{10})

El lenguaje assembly también tiene la instrucción *shift right logical immediate* (**`srli`**)

… y las versiones de estas instrucciones en que la cantidad de bits desplazados es especificada en un registro y no como una constante:

- *shift left logical* (**`sll`**) y *shift right logical* (**`srl`**)

En el caso de *shift right*, el lenguaje tiene además las instrucciones en que los bits que quedan “vacíos” a la izquierda se llenan con copias del bit de signo original (y no necesariamente con 0s):

- *shift right arithmetic immediate* (**`srai`**) y *shift right arithmetic* (**`sra`**)

and y **or** : operaciones bit a bit entre los contenidos de dos registros

P.ej., si

x10 = 0000 0000 0000 0000 0011 1100 0000 0000_2

x11 = 0000 0000 0000 0000 0000 1101 1100 0000_2

… entonces la ejecución de

`and x9,x10,x11` —*la sentencia en C podría ser* `r ← s & t`

… deja

x9 = 0000 0000 0000 0000 0000 1100 0000 0000_2

Todo lenguaje de programación

… incluyendo los lenguajes de máquina y los lenguajes *assembly*

… deben tener instrucciones para controlar la ejecución condicional de otras instrucciones:

- dependiendo de los datos de entrada y de los valores creados durante la ejecución del programa,

  … se ejecutan unas instrucciones u otras

En el assembly RISC-V:

beq *registro1*, *registro2*, *L1*

… significa “ir a la instrucción etiquetada *L1* si el valor en *registro1* **es igual** al valor en *registro2*” —*branch if equal*

bne *registro1*, *registro2*, *L1*

… significa “ir a la instrucción etiquetada *L1* si el valor en *registro1* **no es igual** al valor en *registro2*” —*branch if not equal*

P.ej., compilemos la siguiente sentencia a RISC-V:

```
if (i = j):
    f ← g + h
else:
    f ← g – h
```

Si las variables **`f`**, **`g`**, **`h`**, **`i`** y **`j`** corresponden a los registros **`x19`** a **`x23`**:

```
        bne x22,x23,Else
        add x19,x20,x21
        beq x0,x0,Exit
Else:   sub x19,x20,x21
Exit:
```

`beq x0,x0,Exit` es en la práctica un *branch incondicional* (ya que **`x0`** es siempre igual a **`x0`**) a la instrucción etiquetada **`Exit`**

El *assembler* (ensamblador) libera al compilador y al programador de tener que calcular las direcciones (de memoria) para los *branches*

Las decisiones también son importantes para repetir la ejecución de una computación —*loops*

Las mismas instrucciones condicionales sirven para estos casos

P.ej., compilemos el siguiente loop:

```
while (save[i] = k):
    i ← i+1
```

Supongamos que **`i`** y **`k`** corresponden a los registros **`x22`** y **`x24`**, **`save[0]`** está en **`x25`**, y queremos guardar **`save[i]`** en **`x9`**

Primero, necesitamos la dirección de **`save[i]`**: sumamos **`i`** a la base de **`save`**, por lo que antes hay que multiplicar **`i`** por **`8`**, lo que en este caso hacemos empleando *shift left logical immediate*:

```
Loop:  slli   x10,x22,3       —multiplicación i*8
       add    x10,x10,x25     —suma i + save[0]
       ld     x9,0(x10)       —carga save[i] en x9
       bne    x9,x24,Exit     —el test del loop
       addi   x22,x22,1       —incremento de i
       beq    x0,x0,Loop      —volvemos al principio
Exit:
```

Además de chequear igualdad (**beq**) o desigualdad (**bne**), a veces es útil chequear si una variable es menor que otra

La instrucción

```
blt   registro1,registro2,L1
```

… salta a la instrucción etiquetada *L1* si *registro1* < *registro2*, cuando los valores son tratados como números con signo:

- **`bltu`** salta si *registro1* < *registro2*, cuando los valores son tratados como números sin signo (*branch if less than, unsigned*)

También tenemos las instrucciones **bge** y **bgeu**:

- saltan si el valor en el primer registro es al menos tan grande como el valor en el segundo registro (*branch if greater than or equal*)

  … para números con signo y sin signo, respectivamente

Una ***función*** (*procedimiento* o *subrutina*) es una herramienta que los programadores usamos para estructurar un programa:

- los programas son más fáciles de entender
- el código puede ser reusado

Los *parámetros* de la función actúan como una interfaz entre la función y el resto del programa y datos:

- pasan valores
- devuelven resultados

Al ejecutar la función, el programa debe seguir estos seis pasos:

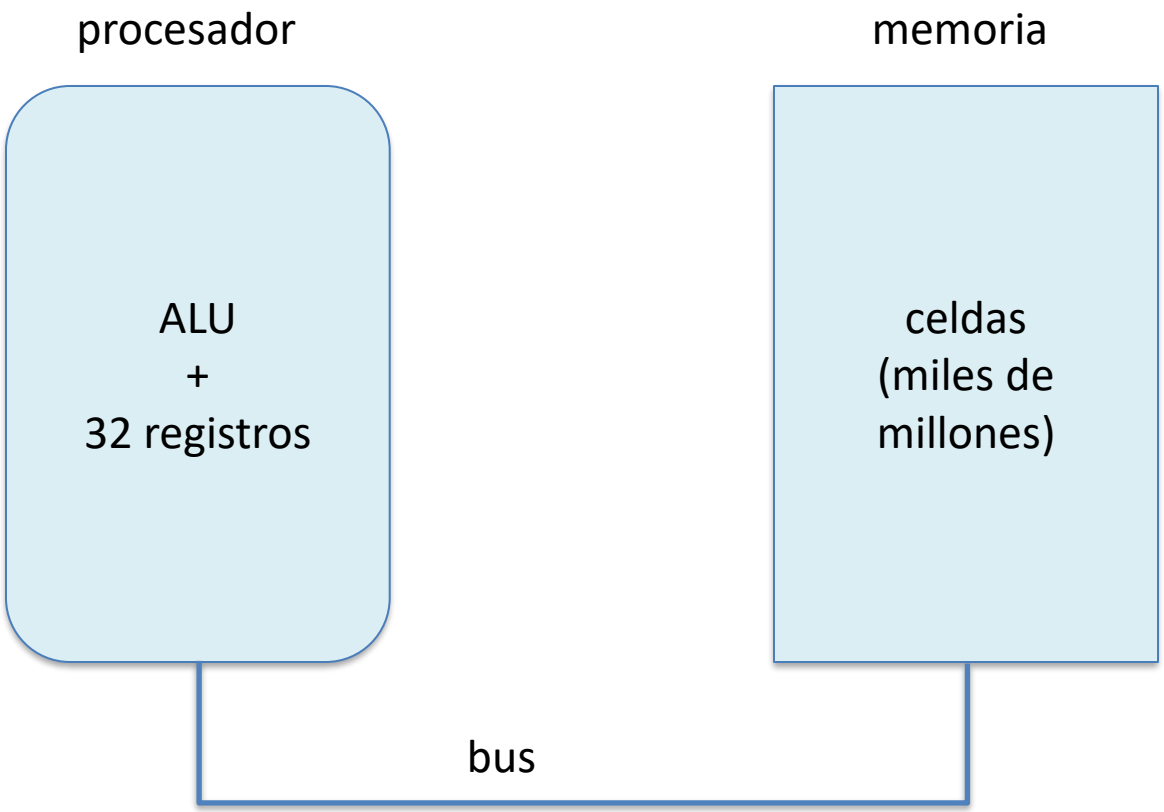
- poner los parámetros en un lugar donde la función tenga acceso a ellos
- transferir el control a la función
- obtener la memoria necesaria para la ejecución de la función
- ejecutar la tarea correspondiente
- poner el resultado en un lugar donde el programa que hizo la llamada tenga acceso a él
- devolver el control al punto de origen, ya que una función puede ser llamada desde varios puntos en un programa

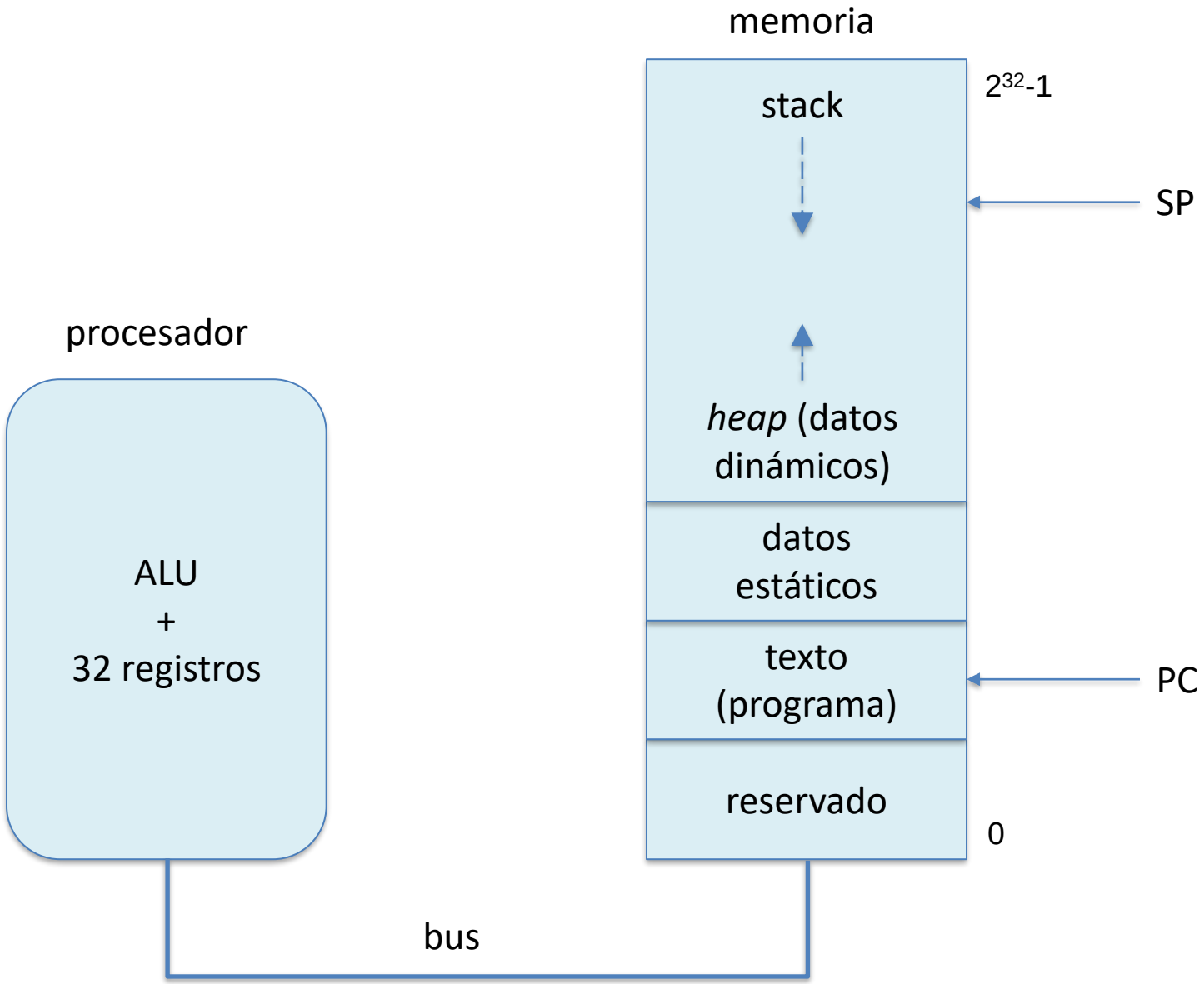
Los registros son el lugar de más rápido acceso para tener datos en un computador

En RISC-V hay 32 registros

… al llamar a una función, los usamos así:

- **`x10`** – **`x17`**: ocho registros para pasar parámetros o retornar valores
- **`x1`**: un registro para la dirección de retorno para volver al punto de origen de la llamada


procesador
memoria
ALU
+
32 registros
celdas
(miles de
millones)
bus


memoria
stack
2^{32}-1
SP
procesador
heap (datos dinámicos)
datos estáticos
ALU
+
32 registros
texto (programa)
PC
reservado
0
bus

RISC-V tiene además la instrucción *jump&link* sólo para funciones:

- **`jal x1,`** *direcciónde la Función*

  … salta a la dirección *direcciónde la Función*

  … y simultáneamente guarda la dirección de la siguiente instrucción —la **dirección de retorno**— en el registro **`x1`**

  (la *dirección de retorno* es necesaria porque la misma función podría ser llamada desde varias partes del programa)

… y la instrucción *jump&link register* para volver desde una función:

- **`jalr x0,0(x1)`**

  … salta a la dirección almacenada en **`x1`** (**`x0`** está *hard-wired* con el valor 0)

El programa que hace la llamada primero pone los valores de los parámetros en los registros **`x10`** – **`x17`**

… y luego ejecuta **`jal x1,`*`F`*** para saltar (al código correspondiente) a la función ***F*** y guardar la dirección de retorno en **`x1`**:

- por supuesto, el registro **PC** —*program counter*— tiene la dirección de la instrucción que está siendo ejecutada en este momento

  … por lo que **`jal`** pone realmente **PC** + 4 en **`x1`**

La función ***F*** entonces hace sus cálculos (o lo que sea)

… pone los resultados en los mismos registros **`x10`** – **`x17`**

… y finalmente devuelve el control al programa que hizo la llamada ejecutando **`jalr x0,0(x1)`**

Si al llamar a una función el compilador necesita más registros que los ocho registros **x10** – **x17**, entonces

… todos los registros adicionales que sean usados deben ser finalmente restaurados a los valores que tenían antes de que se produjera la llamada —recuperar sus valores originales:

- → los valores originales de esos registros deben ser guardados en la memoria mientras dura la llamada a la función

La estructura de datos para esto es un ***stack*** —una cola del tipo "el último en entrar es el primero en salir", o LIFO

… y un puntero —el *stack pointer* o **sp**, almacenado en el registro **x2**— a la dirección más recientemente reservada en el stack para estos registros adicionales

P.ej., ¿cuál es el código assembly compilado de la siguiente función?

```
int xmpl(int g, int h, int i, int j):
    int f
    f ← (g+h) - (i+j)    —ver diap. 4
    return f
```

Los parámetros **g**, **h**, **i** y **j** corresponden a los registros **x10**, **x11**, **x12** y **x13**

La variable **f** corresponde al registro **x20** (este es un registro adicional)

Además, para calcular el valor de **f**, usamos dos registros temporales: **x5** y **x6** (también adicionales)

Así, primero guardamos en el stack los valores de los registros **x5**, **x6** y **x20** —tres palabras dobles (24 bytes)— para poder usarlos durante la ejecución de la función:

```
addi    sp, sp, -24     —ajustamos el stack
sd      x5, 16(sp)      —guardamos el contenido de x5
sd      x6, 8(sp)       —guardamos el contenido de x6
sd      x20, 0(sp)      —guardamos el contenido de x20
```

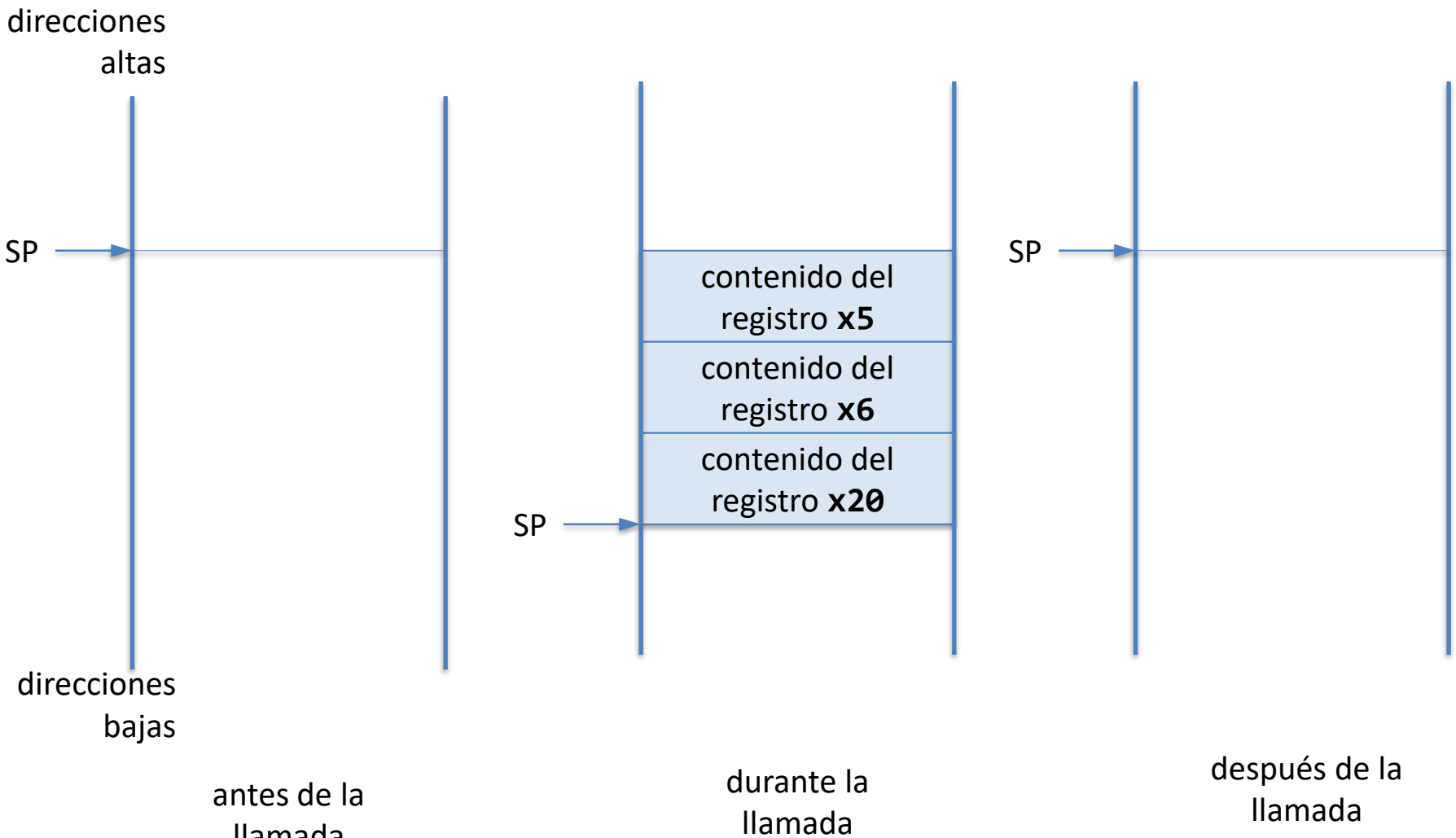
Luego, calculamos el valor de la expresión y los guardamos en **x20**:

```
add     x5, x10, x11
add     x6, x12, x13
sub     x20, x5, x6
```

… y para retornarlo lo guardamos en **x10** —un registro de parámetro:

```
addi    x10, x20, 0
```

El stack, en la memoria



Antes de volver al punto donde se produjo la llamada, restauramos los valores de los tres registros guardados en el stack:

```
ld      x20, 0(sp)
ld      x6, 8(sp)
ld      x5, 16(sp)
addi    sp, sp, 24
```

… y finalmente retornamos, ejecutando

```
jalr    x0, 0(x1)
```

En realidad, no siempre es necesario guardar en el stack los valores de los registros adicionales

… p.ej., cuando no representan valores de variables del programa, sino resultados intermedios:

- en RISC-V, los registros **x5** a **x7** y **x28** a **x31** son registros temporales que no son preservados durante una llamada a una función
- … mientras que los registros **x8**, **x9**, y **x18** a **x27** son registros que, si son usados, deben ser preservados durante una llamada a una función

Esto significa que en el ej. anterior podemos ahorrarnos los dos *loads* y los dos *stores* de **x5** y **x6**

¿Qué pasa en el siguiente caso?

```
int factorial(int n):
    if n < 1:
        return 1
    else:
        return n*factorial(n-1)
```

Como la función es recursiva, ahora también es necesario guardar en el stack los valores de los registros **x10** (el parámetro) y **x1** (la dirección de retorno), para poder recuperarlos al volver a la llamada anterior:

```
factorial:
    addi   sp, sp, -16
    sw     x1, 8(sp)
    sw     x10, 0(sp)
```

Luego, comparamos **n** con 1 (en realidad, **n**–1 con 0), y vamos al *label* **L1** si **n** ≥ 1:

```
    addi   x5, x10, -1
    bge    x5, x0, L1
```

Si **n** < 1, entonces **factorial** devuelve 1, a través del registro **x10**:

```
    addi   x10, x0, 1
    addi   sp, sp, 16
    jalr   x0, 0(x1)
```

Si **n** ≥ 1, **n** es decrementado y **factorial** es llamado recursivamente:

```
L1:
    addi   x10, x10, -1
    jal    x1, factorial
```

De ahí, empezamos a “salir” de (esta llamada de) la función:

```
addi   x6, x10, 0
ld     x10, 0(sp)
ld     x1, 8(sp)
addi   sp, sp, 16
```

… ponemos en **`x10`** el producto de **`x10`** y **`x6`**:

```
mul    x10, x10, x6
```

… y finalmente, **`factorial`** salta de nuevo a la dirección de retorno:

```
jalr   x0, 0(x1)
```

El stack, en la memoria