Instrucciones

(el lenguaje *assembly* RISC-V)

Arquitectura de Computadores

2021-1 Yadran Eterovic

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 localidades especiales construidas directamente en el hardware —los **registros**:

- normalmente, 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

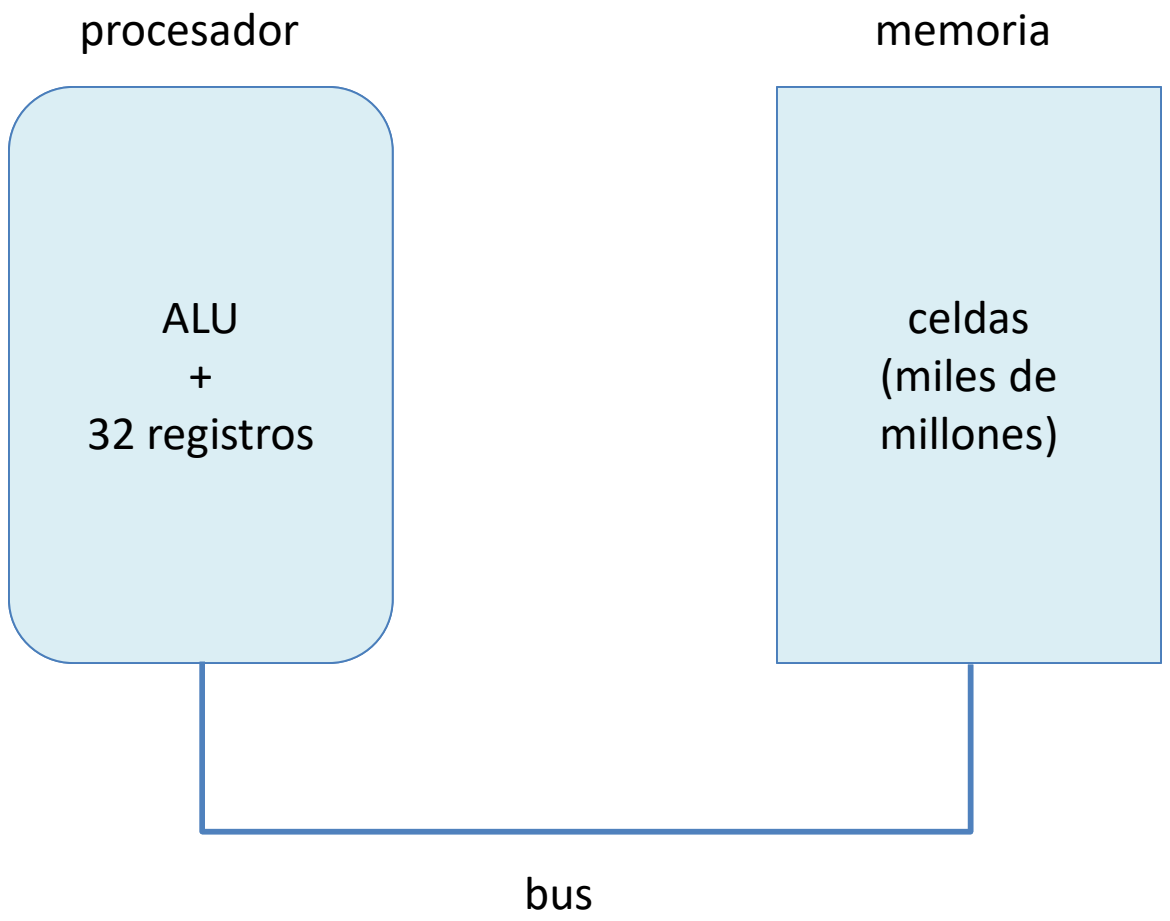
… entonces el código compilado para **`f ← (g + h) – (i + j)`** sería

```
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:

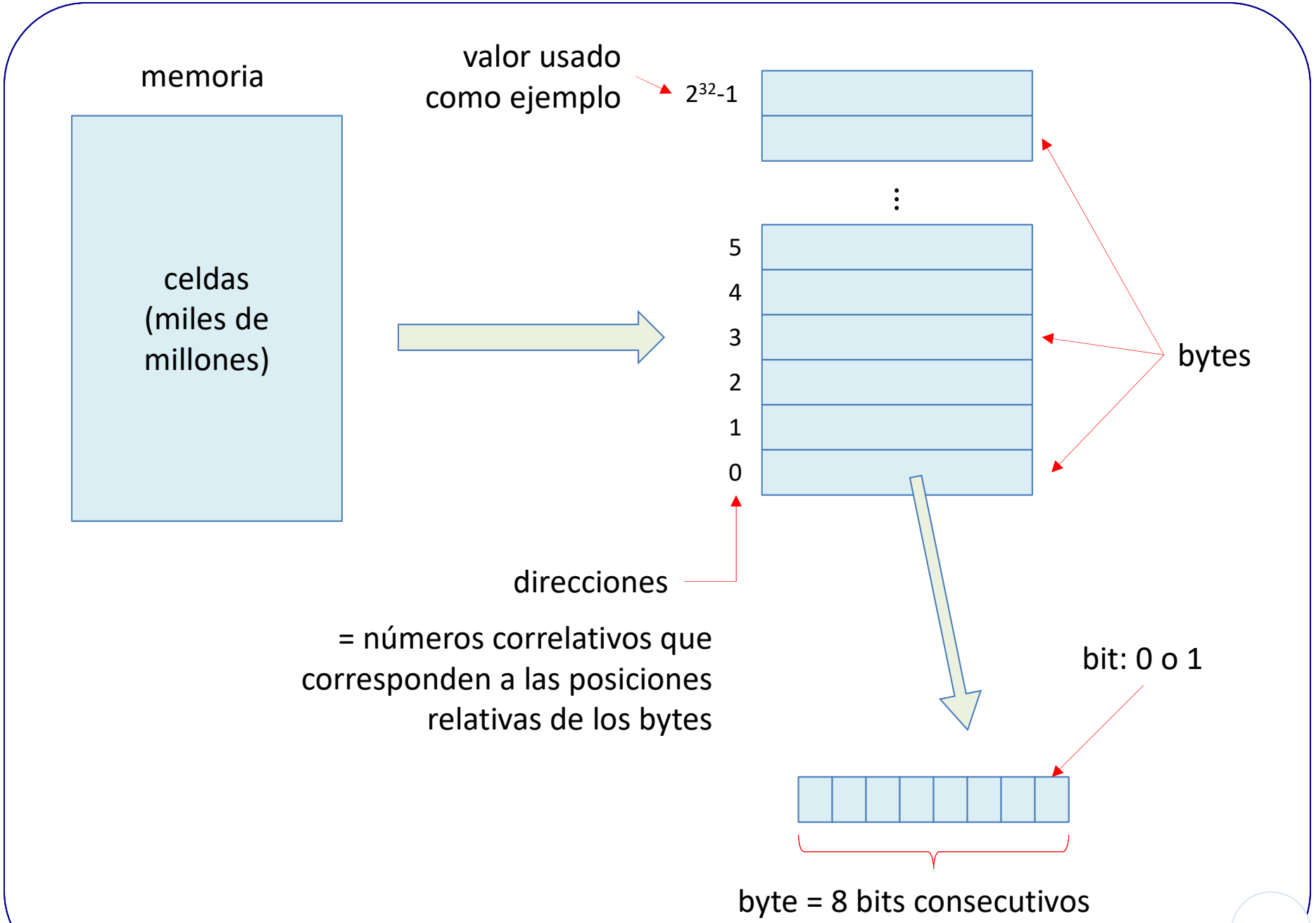
- que tiene capacidad para almacenar miles de millones de datos


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

→ 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:

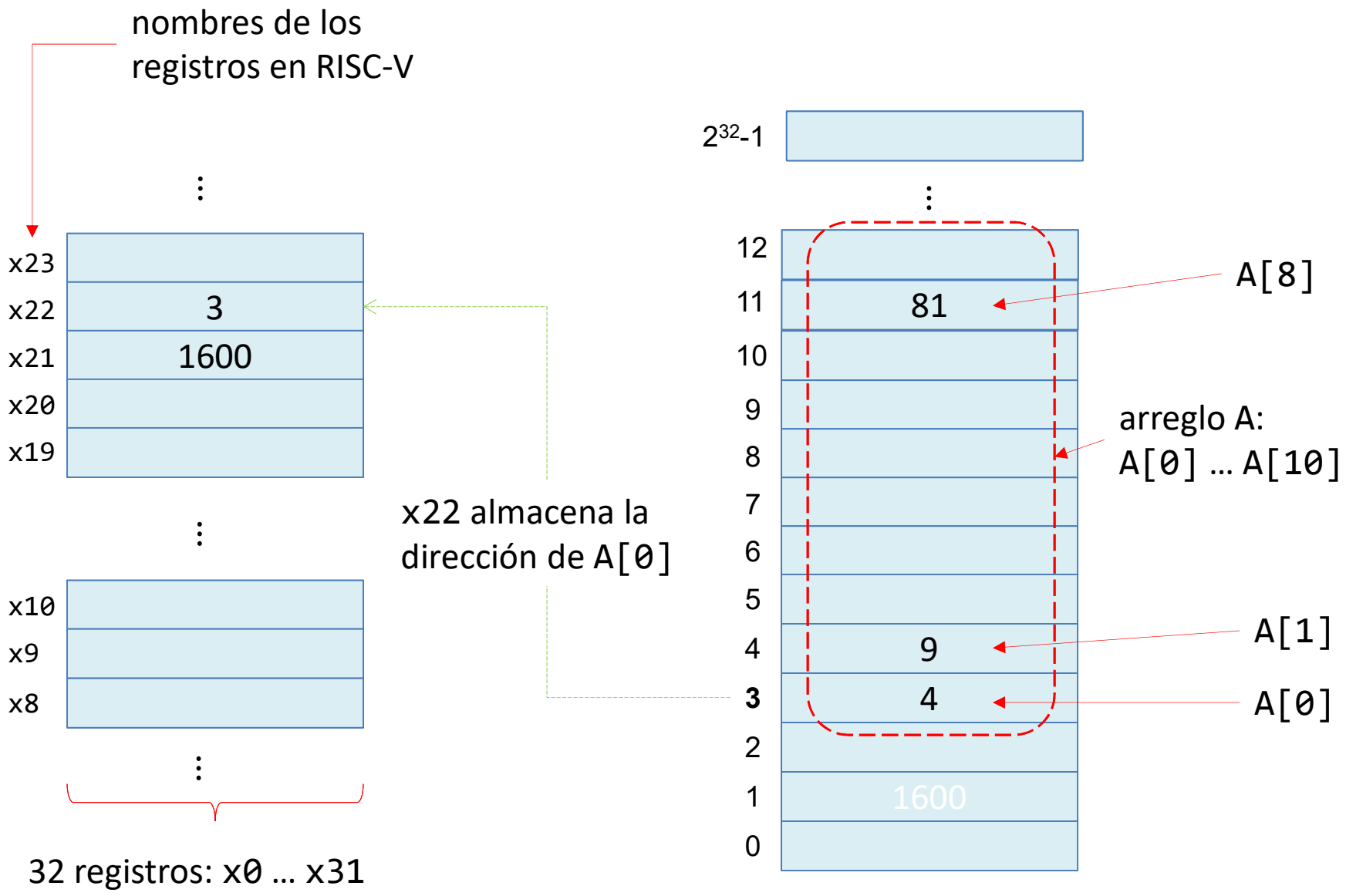
- **`ld`** y **`sd`** deben especificar la dirección de la celda involucrada
- **`ld`** y **`sd`** especifican la dirección de modo *indirecto*, por una constante y un registro → la suma de la constante y el contenido del registro


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`**


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
1600
A[8]
arreglo A:
A[0] … A[10]
A[1]
A[0]

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

—*dos instrucciones en assembly:*

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

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

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

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, ….

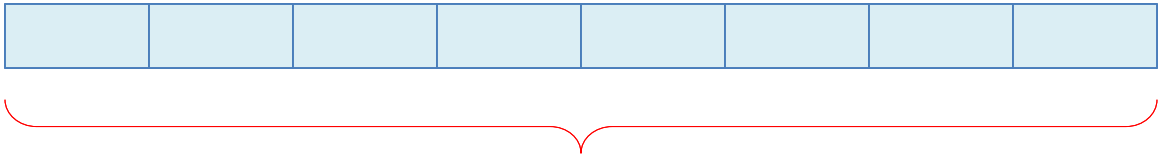
Los computadores modernos, sin embargo, operan sobre *palabras* (*words*) de 4 bytes consecutivos (32 bits)

… 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, …

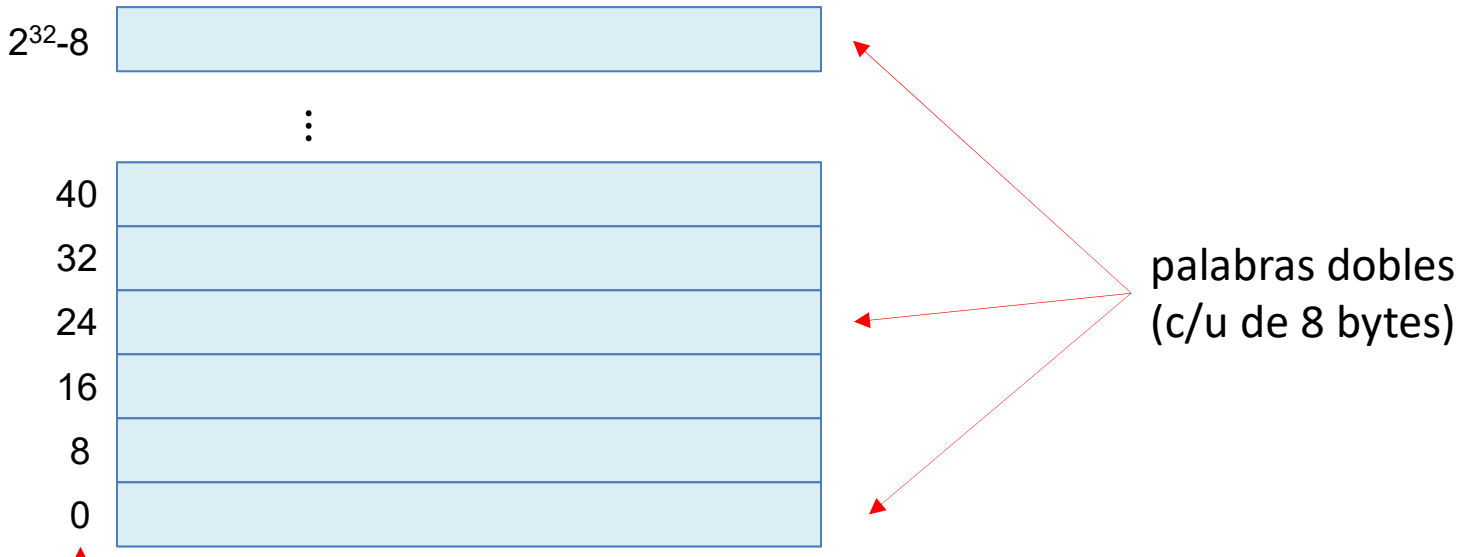
- … es decir, la dirección correspondiente al primer byte de la palabra
- … de modo que el ejemplo anterior requiere un ajuste

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 **x11** y **x10** contienen

0000 0000 0000 0000 0011 1100 0000 0000_2

… y 0000 0000 0000 0000 0000 1101 1100 0000_2

… respectivamente, entonces la ejecución de

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

… deja en **x9** el valor

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 nuestro assembly:

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

```
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*

… y permite al programador usar simplemente etiquetas (*labels*)

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
```

```
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`**

```
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**:

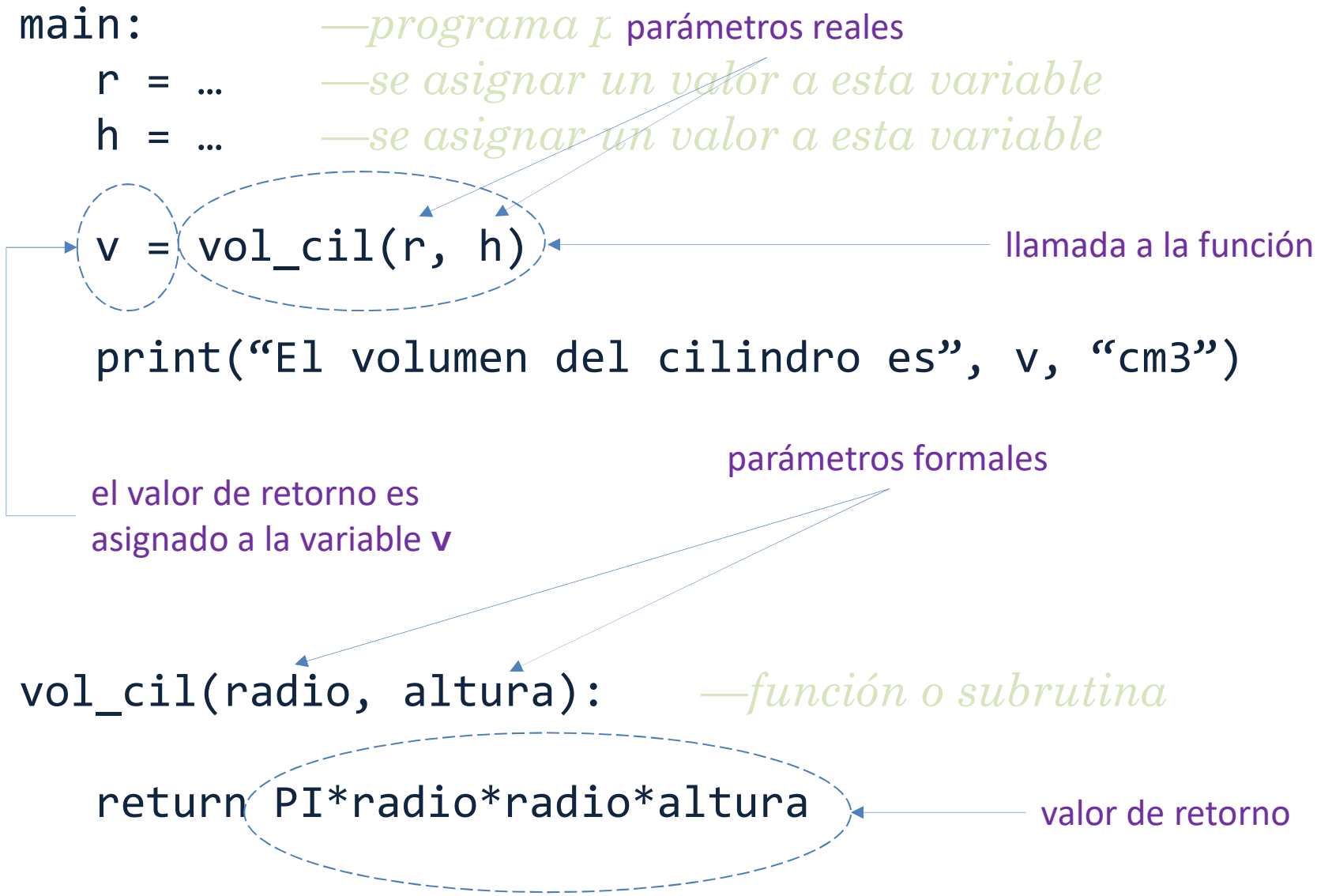
- saltan si el valor en el primer registro es al menos tan grande como el valor en el segundo registro (*branch if greater than or equal*)

  … para números con signo y sin signo, respectivamente

Subrutinas (funciones o métodos)

```
main:           —programa principal
   r = …        —se asigna un valor a esta variable
   h = …        —se asigna un valor a esta variable
   v = vol_cil(r, h)
   print(“El volumen del cilindro es”, v, “cm3”)

vol_cil(radio, altura):     —función o subrutina
   return PI*radio*radio*altura
```


main:
—programa p
r = …
—se asignar un valor a esta variable
h = …
—se asignar un valor a esta variable
parámetros reales
v = vol_cil(r, h)
llamada a la función
print(“El volumen del cilindro es”, v, “cm3”)
parámetros formales
el valor de retorno es
asignado a la variable v
vol_cil(radio, altura):
—función o subrutina
return PI*radio*radio*altura
valor de retorno

1) Al producirse la llamada a la función —la evaluación de la expresión **`vol_cil(r, h)`**— el computador debe empezar a ejecutar las instrucciones de la función:

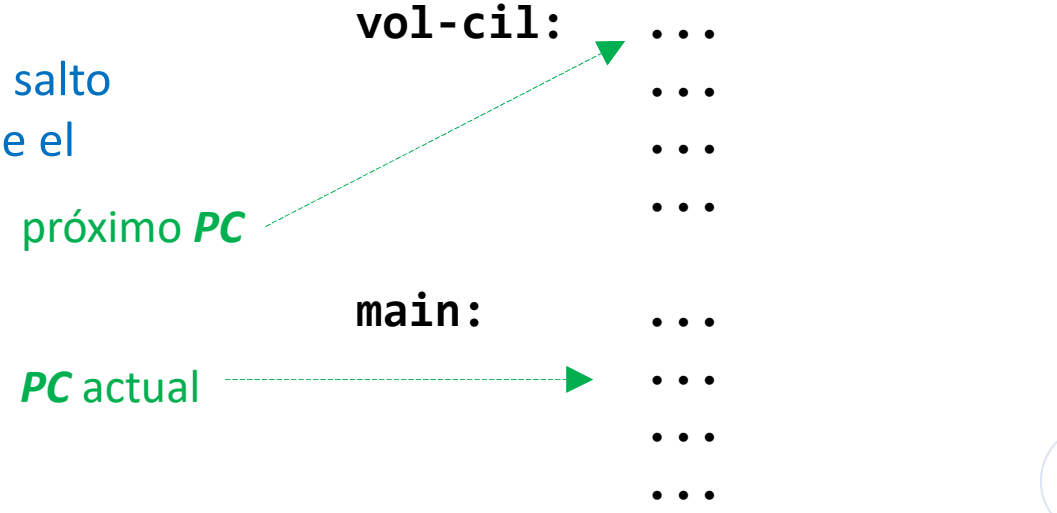
- … mediante una instrucción —que hay que agregar al **`main`**— equivalente a un salto incondicional, que cambie el valor del registro *PC*

`DATA:` —*en Data Memory*

```
vol-cil:
    radio   ...
    altura  ...


main:
    r       2
    h       5
    v       ...
```

`CODE:` —*en Instruction Memory*



2) Sólo que antes, es necesario "pasarle" a la función **`vol_cil`** los valores que deben tomar los parámetros formales **`radio`** y **`altura`**

… es decir, los valores que en ese momento tienen las variables **`r`** y **`h`** (los parámetros reales):

- → hay que almacenar los valores de los parámetros reales en algún lugar de la *Data Memory* al que la función tenga acceso

  … mediante instrucciones adicionales en el **`main`**

`DATA:` —*en Data Memory*

```
vol-cil:
    radio   2
    altura  5


main:
    r       2
    h       5
    v       ...
```

a través de los registros

`CODE:` —*en Instruction Memory*

```
vol-cil:    ...
            ...
            ...
            ...

main:       ...
            ...
            ...
            ...
```

3) Finalmente, al terminar la ejecución de la función, es necesario "pasar de vuelta", o "retornar", el valor calculado por la función:

- usando nuevamente la *Data Memory*

… y reanudar la ejecución del programa **`main`** en el punto en que fue suspendida:

- retomando el valor original del registro *PC* **más 1**
- → este valor debió haber quedado guardado en alguna parte antes de que se empezara a ejecutar la función

`DATA:` —*en Data Memory*

```
vol-cil:
    radio   2
    altura  5
    retval  63

main:
    r       2
    h       5
    v       ...
```

a través de los registros

`CODE:` —*en Instruction Memory*

```
vol-cil:    ...
            ...
            ...
            ...

main:       ...
            ...
            ...
            ...
```

PC actual

próximo ***PC***

En las siguientes diapositivas, vamos a construir de a poco una solución al problema de llamar funciones con parámetros, cuando hay llamadas anidadas

- veremos primero el caso de una única llamada a una función
- luego, el caso de una llamada a una función que a su vez llama a otra función
- finalmente, cómo manejar los registros *A* y *B* cuando sus valores antes de la llamada a una función deben seguir disponibles al volver de la llamada

Primero, vamos a construir la solución en el caso de nuestro computador básico

… y luego, vamos a ver cómo se resuelve en RISC-V