

Programación en lenguaje ensamblador

Sistemas con Microprocesadores

Mg.Ing. Esteban Volentini (evolentini@herrera.unt.edu.ar)

<http://www.microprocesadores.unt.edu.ar/procesadores>

Cronograma

Actividad	Inicio	Descripción	Completa
Presentación	24/08	Presentación y reglamento	✓
Tema 1	24/08	Estructura de las computadoras	✓
Tema 2	26/08	Proyecto con un microcontrolador	✓
Tema 3	31/08	Descripción funcional del ISA RV32IM	✓
Tema 4	07/09	Programación en lenguaje ensamblador	←
Tema 5	18/09	Diseño del camino de datos de un procesador	
Tema 6	30/09	Diseño de la unidad de control del procesador	
Tema 7	05/10	Tecnologías y sistema de memoria	
Tema 8	09/10	Dispositivos de entrada/salida	
Tema 9	19/10	Entradas/salidas digitales (GPIO)	
Tema 10	28/10	Generalidades de las interrupciones	
Tema 11	04/11	Interrupciones y excepciones en RISC-V	
Tema 12	13/11	Manejo de señales con temporizadores	
Tema 13	18/11	Transferencias de datos	
Proyectos	30/11	Seminarios de proyectos	

Temas a Tratar

- ▶ Datos simples y estructurados
- ▶ Patrones de programación
- ▶ Pilas
 - ▶ Implementación en ARM-Cortex-M4
 - ▶ Ejemplos de uso
- ▶ Variables globales y locales
- ▶ Subrutinas
 - ▶ Implementación en ARM-Cortex-M4
 - ▶ Pasaje de Parámetros

Antes de empezar: un quizz



Tipos de datos simples

- ▶ Elementos indivisibles y ordenados
- ▶ Una variable referencia a un único valor
 - ¿Ocupan una sola dirección de memoria?
- ▶ Números enteros y reales
- ▶ Caracteres y enumerados
- ▶ Booleanos
 - En la realidad se almacenan siempre tiras de bits
 - La interpretación está definida por el programa

Tipos de datos simples

- ▶ Siempre que es posible se utiliza un registros para una variable simple
- ▶ Cuando la cantidad de registros no es suficiente se almacenan en memoria
 - ▶ La variable se identifica por la dirección
 - ▶ Es constante una vez compilado el programa
 - ▶ Si la dirección es una constante grande se usa un registro como puntero a la variable

Tipos de datos simples

```
.org 0x00000100
.data
r: .space 4
```



```
.text
main:
```

```
li x1, 3           # x1 = a ← 3
li x2, -2          # x2 = b ← -2
add x3, x1, x2      # x3 = r ← 3 + (-2) = 1
sw x3, 0x100(x0)    # M[0x0000.0100] = r ← 1
```

```
void main(void) {
    int r;
    short int b = -2;
    unsigned char a = 3;

    r = a + b;
}
```

Tipos de datos simples

```
.org 0x00000100
.data
r: .space 4
b: .half -2
a: .byte 3
```



```
.text
main:
lbu x1, 0x106(x0) # x1 ← a = M[0x0000.0106] = 3
lh x2, 0x104(x0) # x2 ← b = M[0x0000.0104] = -2
add x1, x1, x2 # x1 ← 3 + (-2) = 1
sw x1, 0x100(x0) # r = M[0x0000.0106] ← x1 = 1
```

```
void main(void) {
    int r;
    short int b = -2;
    unsigned char a = 3;

    r = a + b;
}
```

Tipos de datos simples

```
.org 0x00000100
.data
r: .space 4
b: .space 2
a: .space 1
```



```
.text
```

```
main:
```

```
li x1, 3           # x1 ← 3
sb x1, 0x106(x0)    # a = M[0x0000.0106] x1 ← x1 = 3
li x2, -2          # x2 ← -2
sh x2, 0x104(x0)    # b = M[0x0000.0104] x1 ← x2 = -2
add x1, x1, x2      # x1 ← 3 + (-2) = 1
sw x1, 0x100(x0)    # r = M[0x0000.0106] ← x1 = 1
```

```
void main(void) {
    int r;
    short int b = -2;
    unsigned char a = 3;

    r = a + b;
}
```

Tipos de datos simples

```
.org 0x10000000
.data
r: .space 4
b: .half -2
a: .byte 3
```



```
.text
main:
```

```
la x1,a          # x1 ← 0x1000.0006
lbu x1,0(x1)     # x1 ← a = M[0x1000.0006] = 3
la x2,b          # x2 ← 0x1000.0004
lh x2,0(x2)      # x2 ← b = M[0x1000.0004] = -2
add x1,x1,x2     # x1 ← 3 + (-2) = 1
la x2,r          # x2 ← 0x1000.0000
sw x1,0(x2)      # r = M[0x1000.0000] ← x2 = 1
```

```
void main(void) {
    int r;
    short int b = -2;
    unsigned char a = 3;

    r = a + b;
}
```

Punteros

- ▶ Un puntero es una variable cuyo valor es una dirección de memoria
 - ▶ Igual que con los números con o sin signo, o los caracteres, esto es una interpretación
- ▶ Es habitual que el tamaño de un puntero coincida con el tamaño del dato nativo del procesador
 - ▶ En nuestro caso un entero sin signo de 32 bits
- ▶ Como las otras variables simples se puede usar un registro o almacenarla en memoria

Punteros

```
.org 0x10000000
.data
p: .space 4
r: .half 7
a: .space 2
```



```
.text
main:
```

```
la x1,p      # x1 ← 0x1000.0000
la x2,r      # x2 ← 0x1000.0004
sw x1,0(x2)  # p = M[0x1000.0000] ← 0x1000.0004

la x1,p      # x1 ← 0x1000.0000
la x1,0(x1)  # x1 ← p = M[0x1000.0000] = 0x1000.0004
lh x1,0(x1)  # x1 ← p* = M[0x1000.0004] = 7
la x2,a      # x2 ← 0x1000.0006
sw x1,0(x2)  # a = M[0x1000.0006] ← 7
```

```
void main(void) {
    short int r = 7;
    int * p;
    short int a;

    p = &r;
    a = *p;
}
```

Tipos de datos compuestos

- ▶ Elementos indivisibles y ordenados
- ▶ Una variable hace referencia a un conjunto de valores
 - ▶ Cada elemento puede ser un dato simple u otro dato compuesto
- ▶ Arreglos (vectores o matrices): varios elementos del mismo tipo
- ▶ Estructuras (o registros): varios elementos de diferentes tipos

Arreglos

- ▶ Se reserva memoria contigua para acomodar todos los elementos
- ▶ Para acceder a los elementos se utiliza un índice numérico, que puede ser constante o variable
- ▶ La dirección de inicio del elemento se puede calcular a partir de la dirección de inicio del arreglo, el índice y el tamaño los elementos

Vectores

```
.data
a: .space 20
b: .half 1, 2, 3, 4
```



```
void main(void) {
    unsigned int a[5];
    short int b[4] = {
        1, 2, 3, 4,
    };

    a[2] = b[3] + 2;
}
```

```
.text
main:
    la x1,b          # x1 ← 0x1000.0014
    lh x2,6(x1)       # x2 ← b[3] = M[0x1000.0020] = 3
    addi x2,x2,2       # x2 ← 3 + 2 = 5
    la x1,a           # x1 ← 0x1000.0000
    sw x2,8(x1)        # a[2] = M[0x1000.0000] ← x2 = 5
```

Cadenas

```
.data
c: .string "hola"
l: .space 1
```



```
void main(void) {
    char l;
    char c[5] = "hola";

    l = c[3];
}
```

```
.text
main:
    la x1,c          # x1 ← 0x1000.0000
    lb x1,3(x1)      # x1 ← M[0x1000.003] = 'a' = 0x61 = 97
    la x2,l          # x2 ← 0x1000.0005
    sb x1,0(x2)      # l = M[0x1000.0005] ← 'a' = 0x61 = 97
```

Cadenas

```
.data
p: .space 4
l: .space 1
c: .string "hola" !!!

.text
main:
    la x1,c          # x1 ← 0x1000.0005
    addi x1,x1,3      # x1 ← 0x1000.0005 + 3 = 0x1000.0008
    la x2,p          # x2 ← 0x1000.0000
    sw x1,0(x2)       # p = M[0x1000.0000] ← 0x1000.0008
    lb x1,0(x1)       # x1 ← p* = M[0x1000.0008] = 'a' = 0x61 =
97
    la x2,l          # x2 ← 0x1000.0004
    sb x1,0(x2)       # l = M[0x1000.0004] ← 'a' = 0x61 = 97
```

```
void main(void) {
    char * p;
    char l;
    char c[5] = "hola";

    p = c + 3;
    l = p*;
}
```

Matrices

```
c: .data
    .half 1, 2, 3
    .half 4, 5, 6
```



```
.text
main:
```

```
la x1,c      # x1 ← 0x1000.0000
lh x2,4(x1)   # x2 ← c[0][2] = M[0x1000.004] = 3
lh x3,8(x1)   # x3 ← c[1][1] = M[0x1000.008] = 4
addi x3,x3,x2 # x1 ← 3 + 2 = 7
sh x3,10(x1)  # c[1][2] = M[0x1000.000A] ← 7
```

```
void main(void) {
    short int c[2][3] = {
        {
            1, 2, 3,
        }, {
            4, 5, 6,
        },
    };
    c[1][2] = c[0][2] + c[1][1];
}
```

Estructuras

- ▶ Se reserva memoria contigua para acomodar todos los elementos
- ▶ Para acceder a los elementos se utiliza un identificador jerárquico
 - ▶ La parte izquierda representa la dirección de inicio de la estructura
 - ▶ La parte derecha el desplazamiento para obtener la dirección de inicio del miembro
- ▶ Si el miembro es un dato compuesto puede haber un índice o un tercer componente en el identificador
 - ▶ El proceso se repite hasta que se referencia un dato simple

Estructuras

```
.data
event:
    .space 8
```



```
.text
main:
    la x1,event    # x1 ← 0x1000.0000
    li x2,2025     # x2 ← 2025
    sh x1,4(x2)    # event.date.year = M[0x1000.0004] ← 2025
    li x2,8        # x2 ← 8
    sb x1,6(x2)    # event.date.month = M[0x1000.0006] ← 8
    li x2,25       # x2 ← 25
    sb x1,7(x2)    # event.date.day = M[0x1000.0007] ← 25
    li x2,1350     # x2 ← 1350
    sw x1,0(x2)    # event.date.seconds = M[0x1000.0000] ← 1350
```

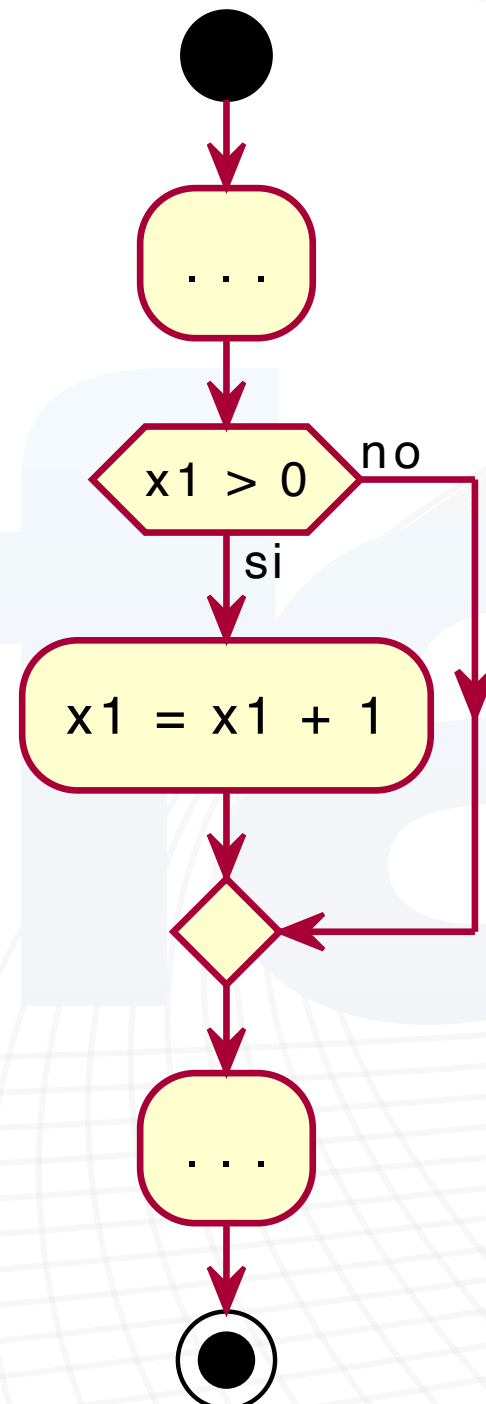
```
void main(void) {
    struct {
        unsigned int seconds; // size = 8
                                // seconds +0
        struct {
            short int year; // size = 4
                        // year = +0
            unsigned char month; // month = +2
            unsigned char day; // day = +3
        } date; // date = +4
    } event;

    event.date.year = 2025;
    event.date.month = 8;
    event.date.day = 25;
    event.seconds = 1350;
}
```

Estructuras de programación

- ▶ En un programa encontramos tres tipos de estructuras
 - ▶ Secuenciales: se ejecutan en orden lineal, y corresponden a transferencias o procesamiento de datos
 - ▶ Selección o condicional: permite ejecutar uno u otro bloque secuencia en función de una condición lógica (**if-then, if-then-else**)
 - ▶ Iteración o bucle: permite repetir un bloque de código hasta que se cumpla una condición o una cantidad determinada de veces (**do-while, while-do, for**)

Estructuras típicas: if-then

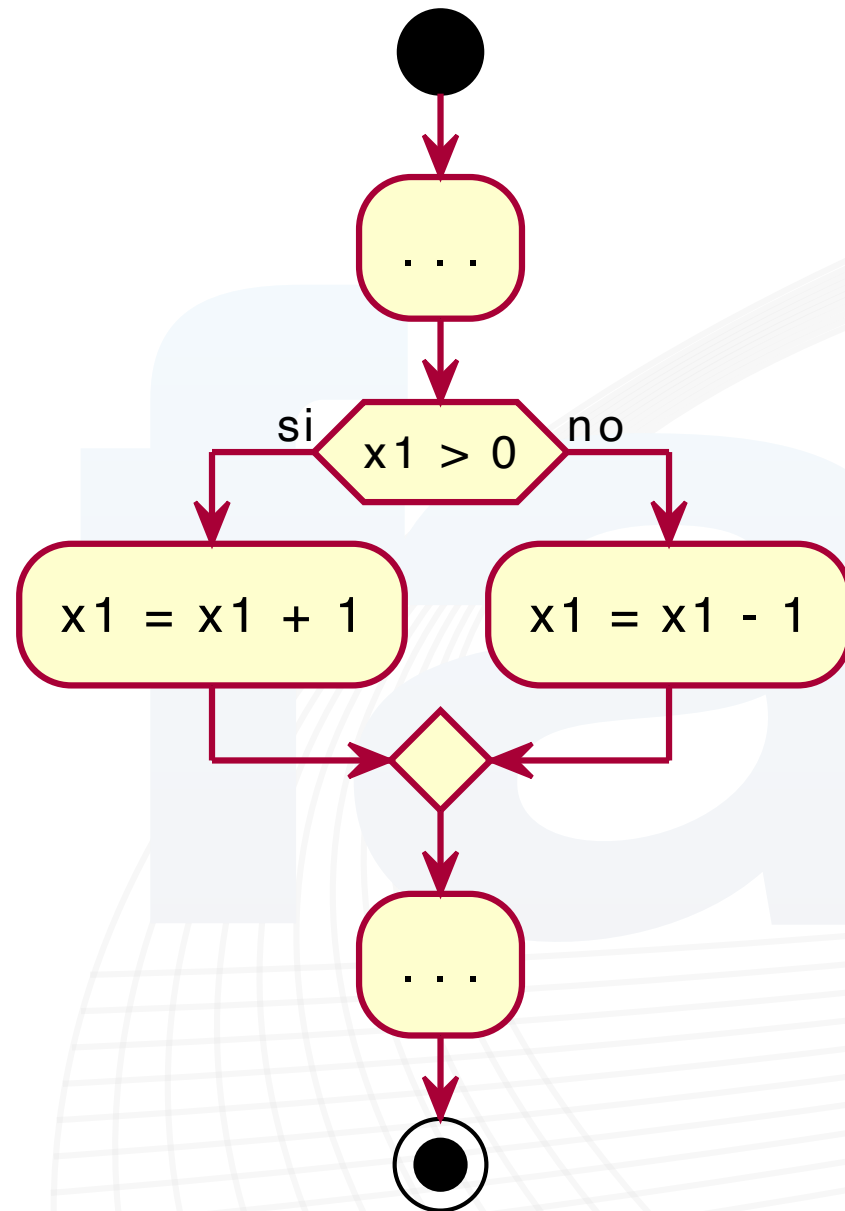


...

Salta por menor o igual
ble x1,zero,label
Bloque then
addi x1,x1,1

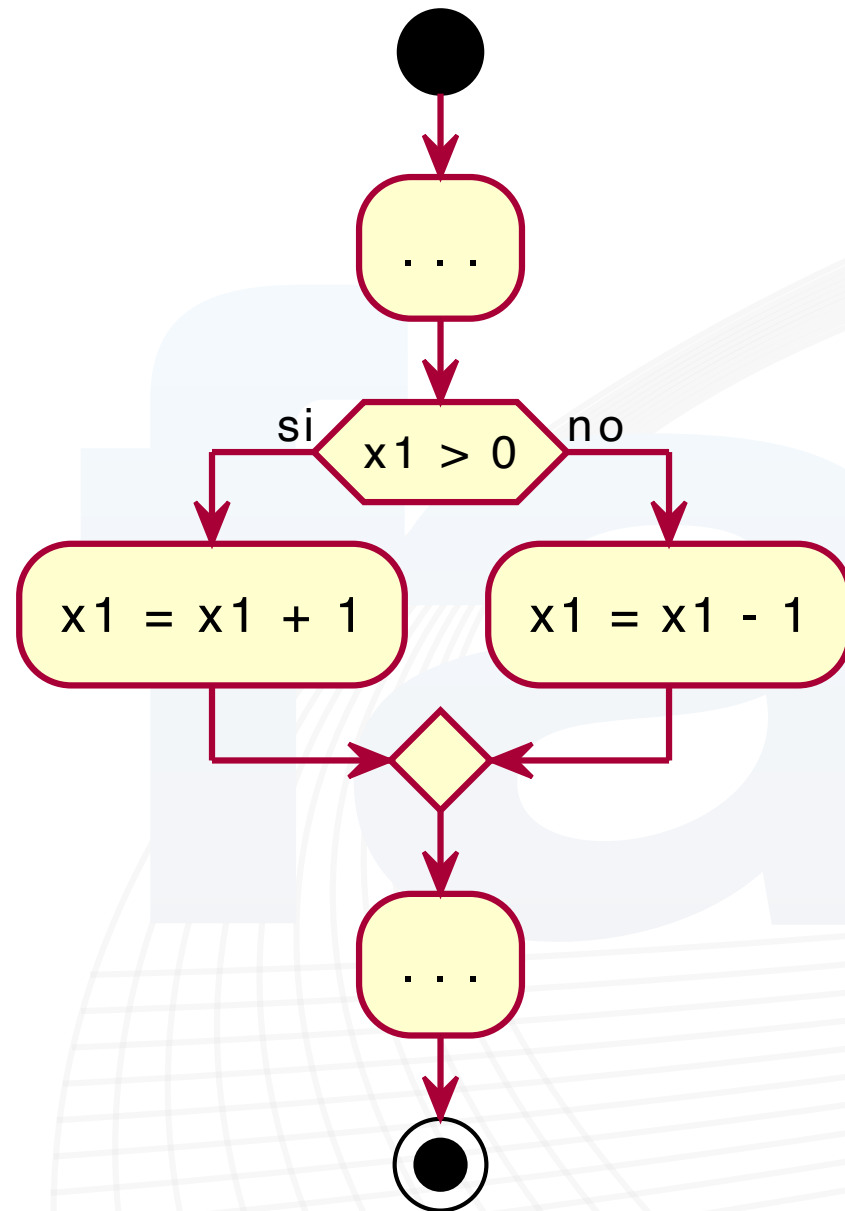
label: ...

Estructuras típicas: if-then-else



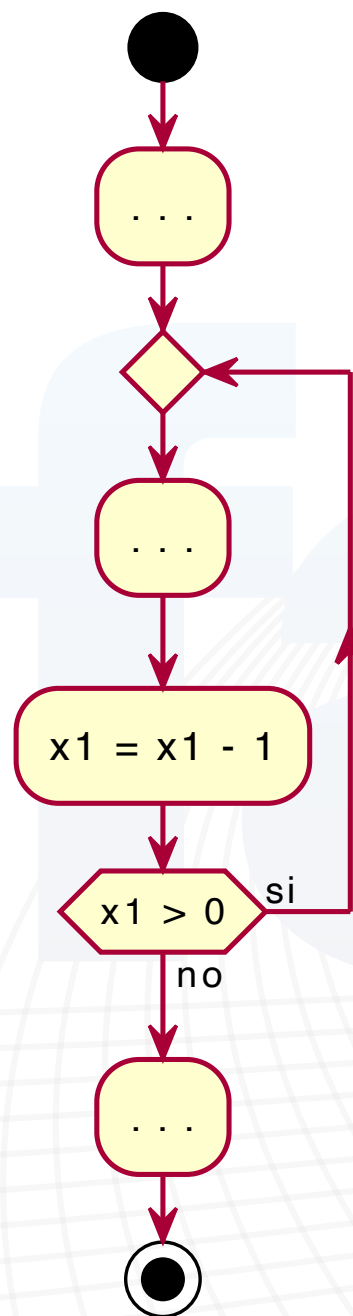
```
...  
# Salta por menor o igual  
ble x1,zero,else  
# Bloque then  
addi x1,x1,1  
# Salta al final del if  
j label  
# Bloque else  
else: addi x1,x1,-1  
label: ...
```

Estructuras típicas: if-then-else



```
...  
# Salta por menor o igual  
bgt x1,zero,then  
# Bloque else  
addi x1,x1,-1  
# Salta al final del if  
j label  
# Bloque then  
then: addi x1,x1,1  
label: ...
```

Estructuras típicas: do-while



lazo:

Cuerpo del lazo

...

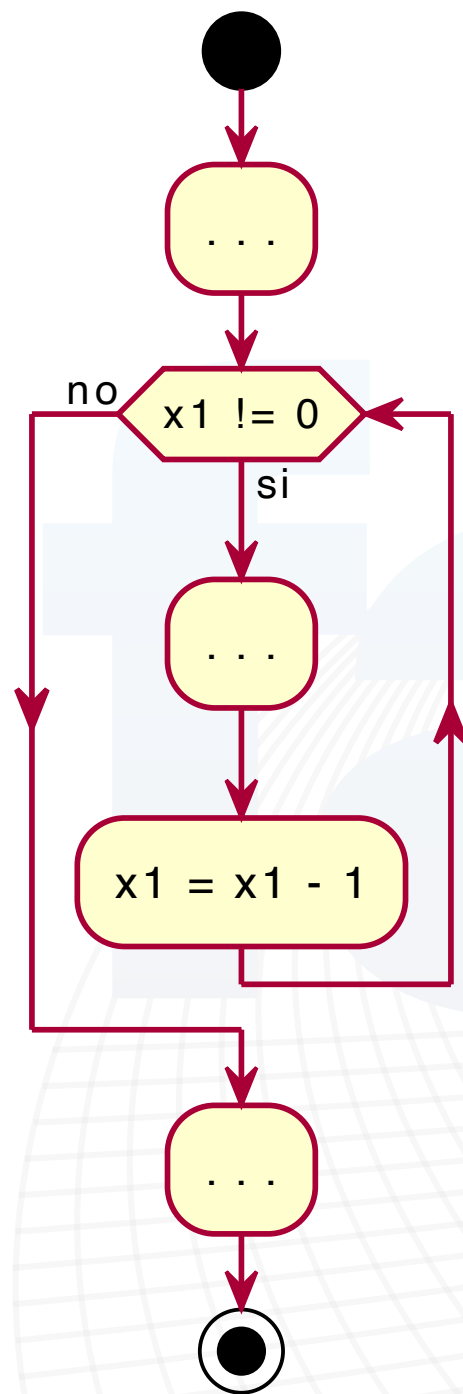
addi x1,x1,-1

Salta por mayor

bgt x1,zero,lazo

...

Estructuras típicas: while-do



lazo:

Salta por menor o igual

ble x1,zero,label

Cuerpo del lazo

...

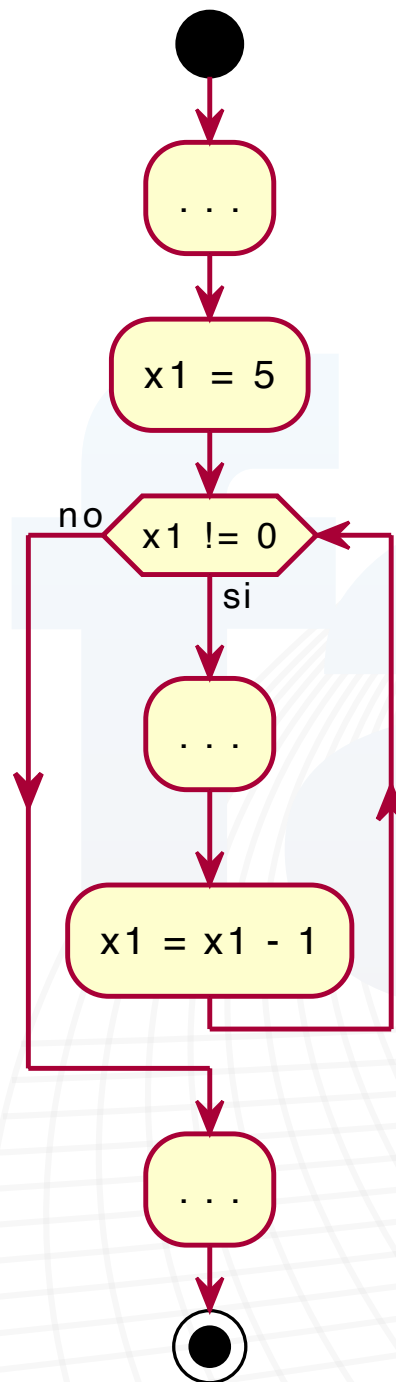
addi x1,x1,-1

Cierre del lazo

j lazo

label: ...

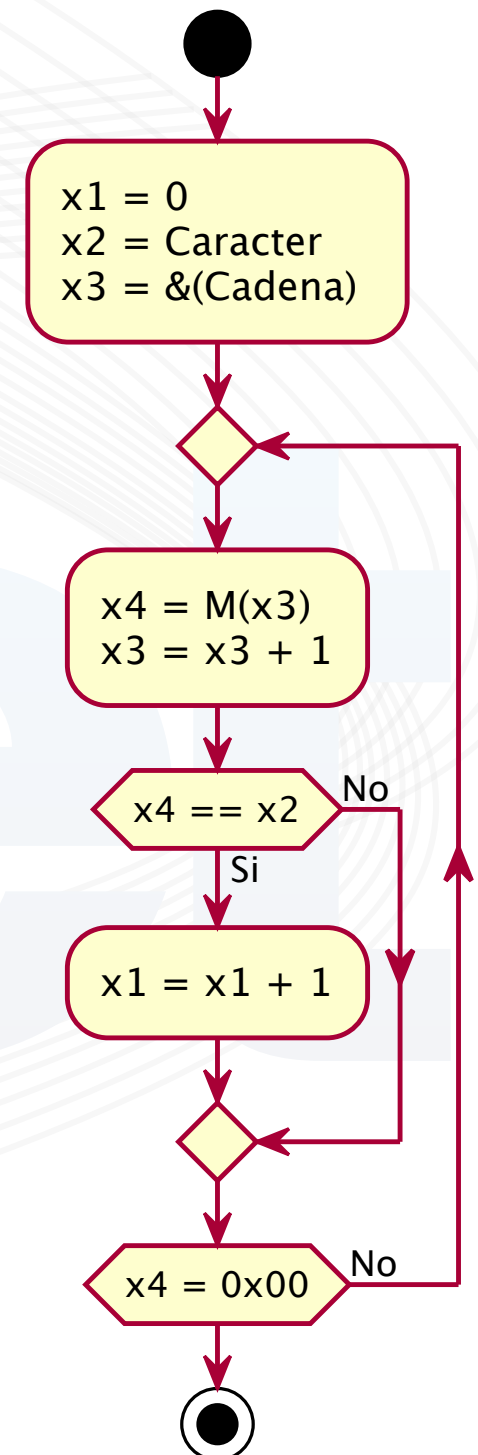
Estructuras típicas: for



Cantidad de iteraciones
li **x1,5**
Saltar por igual a cero
lazo: **bne** **x1,zero,label**
Cuerpo del lazo
...
Variable de control
addi **x1,x1,-1**
Cierre del lazo
j **lazo**
label: ...

Ocurrencias de un caracter

- ▶ Se requiere contar la cantidad de veces que un carácter aparece en una cadena.
- ▶ La cadena sigue el estándar “C”, el final está marcado por el carácter nulo **0x00**.
- ▶ La cadena esta almacenada a partir de la dirección de memoria **cadena**.
- ▶ El carácter a buscar esta almacenado en la dirección de memoria **caracter**.
- ▶ El resultado debe quedar almacenado en el registro **x1**.



Búsqueda de un caracter

```
.data
cadena:    .string "SISTEMAS CON MICROPROCESADORES"
caracter:  .byte 'S'

.text
mv    x1, zero      # Resultado es cero ocurrencias
la    x2, caracter   # Apunta al carácter
lb    x3, 0(x2)      # Carga el carácter a buscar
la    x2, cadena     # Apunta a la cadena
lazo:
lb    x4, 0(x2)      # Carga el carácter actual
addi  x2, x2, 1      # Apunta al siguiente caracter
bne   x4, x3, else   # Si los caracteres son iguales
addi  x1, x1, 1      # Entonces incrementa resultado
else:
bne   x4, x0, lazo   # Entonces repite el lazo
li    a0, 10         # Termina la ejecución sin error
ecall
```

Sentencia Switch en C

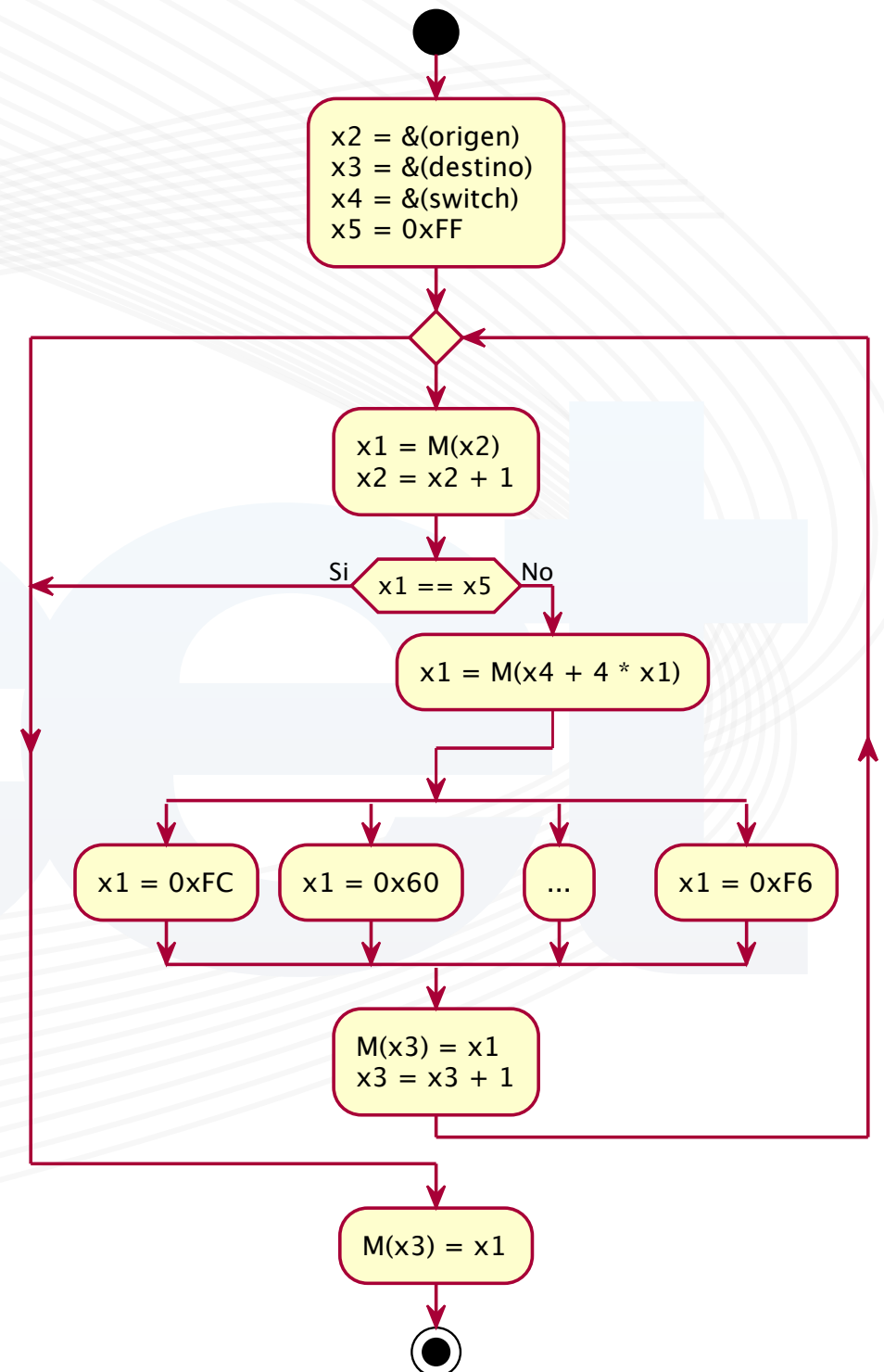
```
switch (test) { case 0: ... break; case 1: ... break}
```

```
la    x2,test    # Apunta x2 al bloque de origen
la    x3,switch  # Apunta x4 al bloque con la tabla
lazo:
lb     x1,0(x2)   # Cargar el valor de test del switch
slli  x1,x1,2     # Multiplicar el valor de test por cuatro
add   x1,x1,x3    # Apuntar al caso en la tabla de saltos
lw    x1,0(x1)    # Cargar el punto de entrada del caso
jr    x1          # Saltar al caso correspondiente
switch:
.word case0      # Punto de entrada del primer caso
.word case1      # Punto de entrada del segundo caso
```

- ▶ Se llama punto de entrada a la dirección de la primera instrucción de una secuencia o bloque de instrucciones.
- ▶ La tabla contiene direcciones de memoria, es decir palabras, y por eso se multiplica por cuatro el valor de test.

Ejemplo: Conversión de Códigos

- ▶ Se plantea resolver el mismo problema del conversor de código BCD a siete segmentos.
- ▶ No se quiere usar una tabla de conversión sino una estructura switch.
- ▶ Los datos están almacenados en los mismos lugares que antes.



Ejemplo: conversión códigos

```
switch:      .data                # Define la sección de variables
             .word case0, case1, case2, case3, case4
             .word case5, case6, case7, case8, case9

reset:       .text
             la    x2,origen      # Apunta x2 al bloque de origen
             la    x3,destino     # Apunta x3 al bloque de destino
             la    x4,switch      # Apunta x4 al bloque con la tabla
             li    x5,0xFF        # Terminador del bloque origen
lazo:        lb    x1,0(x2)        # Carga en x2 el elemento a convertir
             addi  x2,x2,1        # Apunta x2 al siguiente elemento de origen
             beq   x3,x5,final     # Terminar si es fin de conversión
             slli  x1,x1,2        # Multiplicar el caso por cuatro
             add   x1,x1,x4       # Apuntar al caso en la tabla de saltos
             lw    x1,0(x1)       # Cargar la dirección del caso switch
             jr    x1            # Saltar al caso correspondiente
case0:       li    x1,0xFC        # Cargar con la conversión del cero
             j     break         # Saltar al final del bloque switch
case1:       li    x1,0x60        # Cargar con la conversión del uno
             j     break         # Saltar al final del bloque switch
...
case9:       li    x1,0xF6        # Cargar con la conversión del nueve
             j     break         # Saltar al final del bloque switch
break:       sb    x1,0(x3)       # Guardar el elemento convertido
             addi  x3,x3,1
             j     lazo          # Repetir el lazo de conversión
final:       sb    x1,0(x3)       # Guardar el fin de conversión en destino
             li    a0,10         # Termina la ejecución sin error
             ecall
```

Uso de la memoria

- ▶ El compilador divide la memoria de datos en tres fragmentos:
 - ▶ Static: área de datos estática utilizada para variables globales
 - ▶ Heap: área de datos dinámica utilizada con las primitivas **malloc** y **free**.
 - ▶ Stack: área de datos dinámica utilizada para variables locales y llamadas a procedimientos.

Pila / Stack (repaso)

- ▶ Estructura de datos muy utilizada en general
- ▶ ¿Ejemplos de pilas en el mundo real?
- ▶ Sirve para explorar arboles y recordar el camino para retornar al origen
- ▶ ¿Cómo se implementa en software?

Implementación en RISC-V

- ▶ La memoria RAM hace de tabla.
- ▶ Se emplea un registro del CPU ($sp = x2$) como puntero para señalar el último lugar ocupado en la pila.
- ▶ No existen instrucciones para operar con la pila, se implementan como secuencias de dos instrucciones

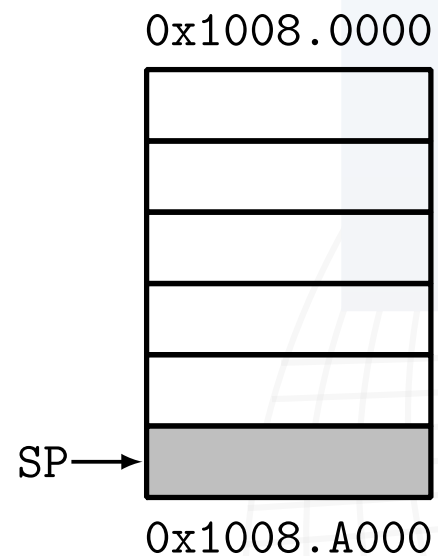
```
# push x1  
addi sp, sp, -4  
sw    x1, 0(sp)
```

```
# pop x1  
lw    x1, 0(sp)  
addi sp, sp, 4
```

- ▶ La pila crece en el sentido de las direcciones decrecientes

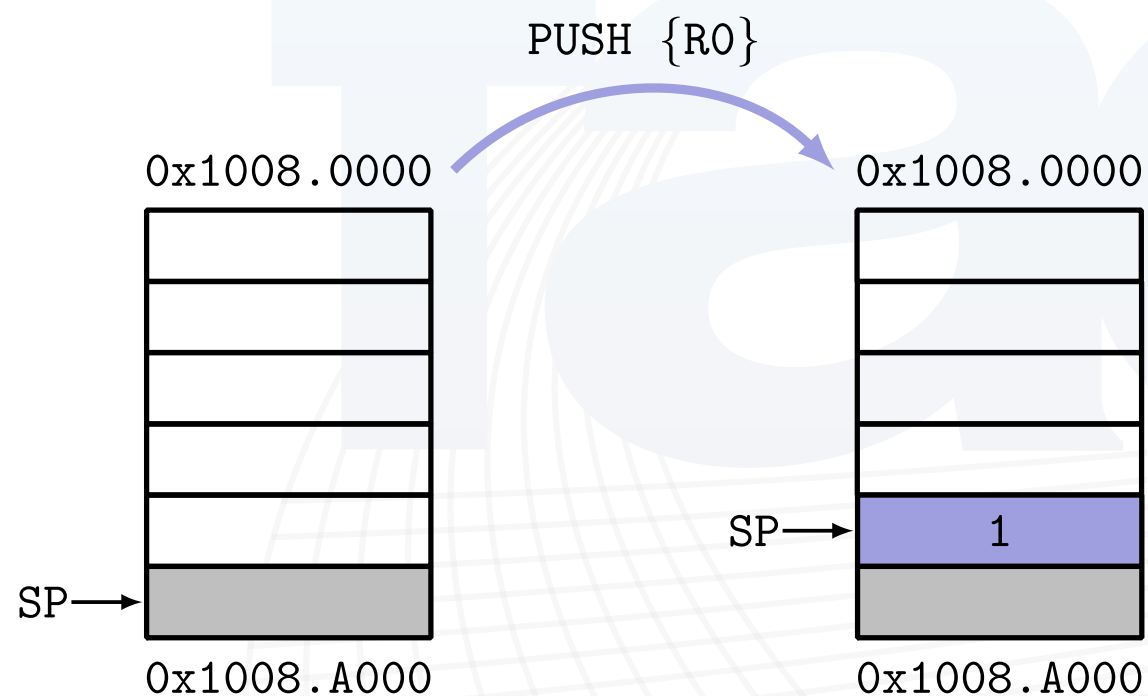
Ejemplo de uso del stack

- Suponiendo que $x3=1$, $x4=2$, $x5=3$



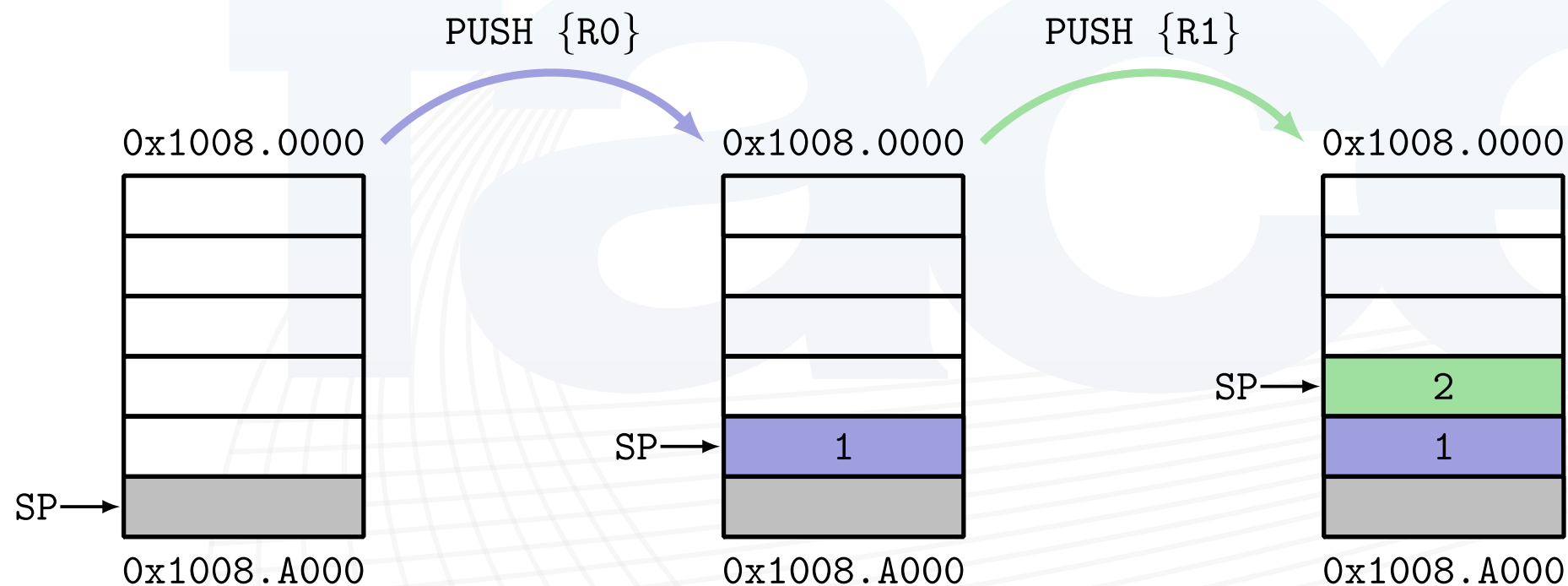
Ejemplo de uso del stack

- Suponiendo que $x3=1$, $x4=2$, $x5=3$
- $sp=sp-4$: $M(sp)=x3$; # push x3



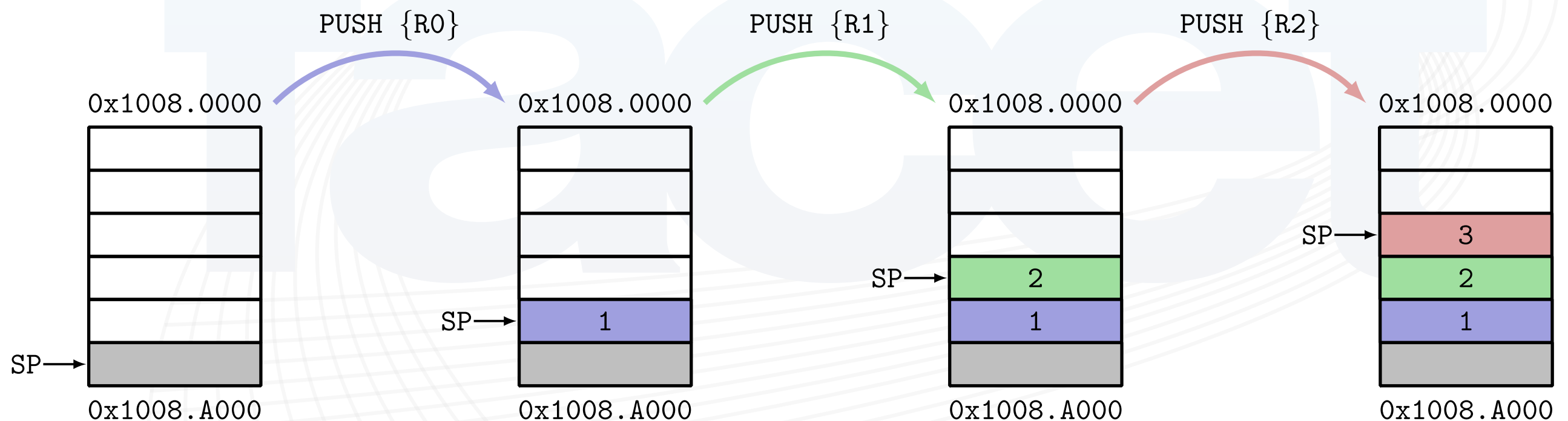
Ejemplo de uso del stack

- Suponiendo que $x3=1$, $x4=2$, $x5=3$
- $sp=sp-4$: $M(sp)=x3$; # push x3
- $sp=sp-4$: $M(sp)=x4$; # push x4



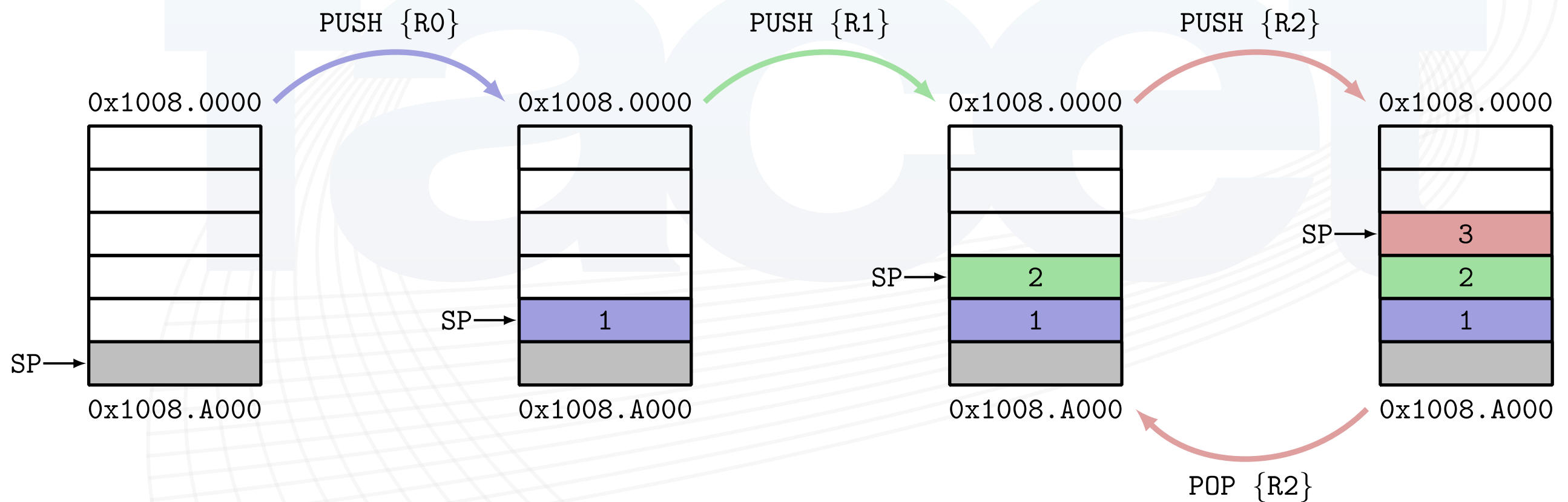
Ejemplo de uso del stack

- Suponiendo que $x3=1$, $x4=2$, $x5=3$
- $sp=sp-4$: $M(sp)=x3$; # push x3
- $sp=sp-4$: $M(sp)=x4$; # push x4
- $sp=sp-4$: $M(sp)=x5$; # push x5



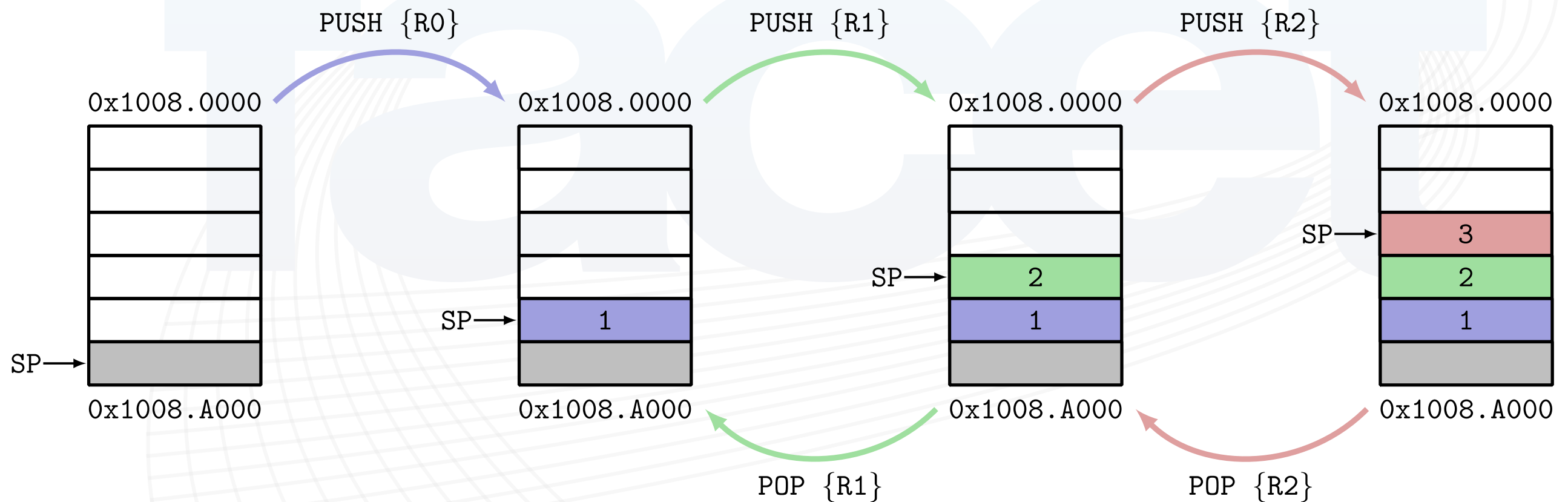
Ejemplo de uso del stack

- Suponiendo que $x3=1$, $x4=2$, $x5=3$
- $M(sp)=x5$: $sp=sp+4$; # pop $x5$



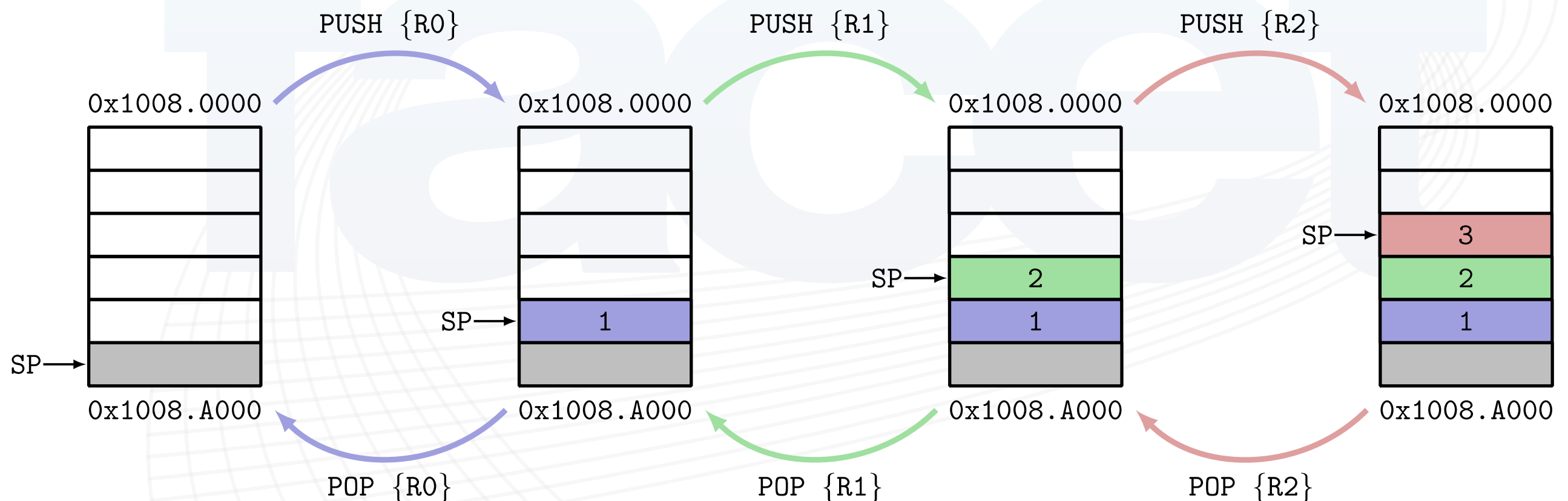
Ejemplo de uso del stack

- Suponiendo que $x3=1$, $x4=2$, $x5=3$
- $M(sp)=x5$: $sp=sp+4$; # pop $x5$
- $M(sp)=x4$: $sp=sp+4$; # pop $x4$



Ejemplo de uso del stack

- Suponiendo que $x3=1$, $x4=2$, $x5=3$
 - $M(sp)=x5$: $sp=sp+4$; # pop x5
 - $M(sp)=x4$: $sp=sp+4$; # pop x4
 - $M(sp)=x3$: $sp=sp+4$; # pop x3



Optimización del uso de la pila

- ▶ Se pueden optimizar las secuencias para varios registros moviendo el puntero de pila una sola vez
- ▶ Se debe tener cuidado con el orden al guardar y recuperar los registros

push x3,x4,x5

addi sp,sp,-12

sw x3,0(sp)

sw x4,4(sp)

sw x5,8(sp)

pop x5,x3,x5

lw x5,8(sp)

lw x4,4(sp)

lw x3,0(sp)

addi sp,sp,12

Subrutinas, procedimientos o funciones

- ▶ Generalmente los programas contienen bloques de código que se repiten.
- ▶ En estos bloques de código pueden variar algunos de los operandos (parámetros).
- ▶ Se puede ahorrar memoria (y tiempo de desarrollo) si estos bloques se escriben una vez y se ejecutan cada vez que se los requiere.
- ▶ Además estos bloques se podrían utilizar en otros programas.

Ventajas de las subrutinas

- ▶ **Diseño Top-Down**
 - ▶ Se divide el sistema en módulos
 - ▶ Cada módulo se divide nuevamente en sub-módulos
 - ▶ Hasta llegar a componentes elementales
 - ▶ Se arma (programa) de arriba hacia abajo
 - ▶ Se prueba al revés
 - ▶ La cantidad de módulos debe ser manejable (máx < 7)
- ▶ **Trabajo en Equipo**
- ▶ **Reutilización (Bibliotecas de Rutinas)**
 - ▶ Requiere estandarización y transparencia

Requerimientos

- ▶ Se las convoca de distintos lugares
- ▶ Deben saber de dónde fueron llamadas para retornar a la instrucción siguiente al llamado
 - Evidentemente un simple salto no es suficiente...
- ▶ Debe haber un mecanismo para pasar y devolver parámetros

Implementación en RISC-V

- ▶ Instrucción de llamada: jump and link (jal)

jal rd,etiqueta

$R[rd] \leftarrow pc + 4; pc \leftarrow etiqueta;$

- ▶ Guarda $pc+4$ en el registro rd
 - ▶ El salto es relativo al valor actual del pc
 - ▶ El alcance del destino es $\pm 1\text{MiB}$
- ▶ Tambien existe una versión indexada

jalr rd,rs,desplazamiento

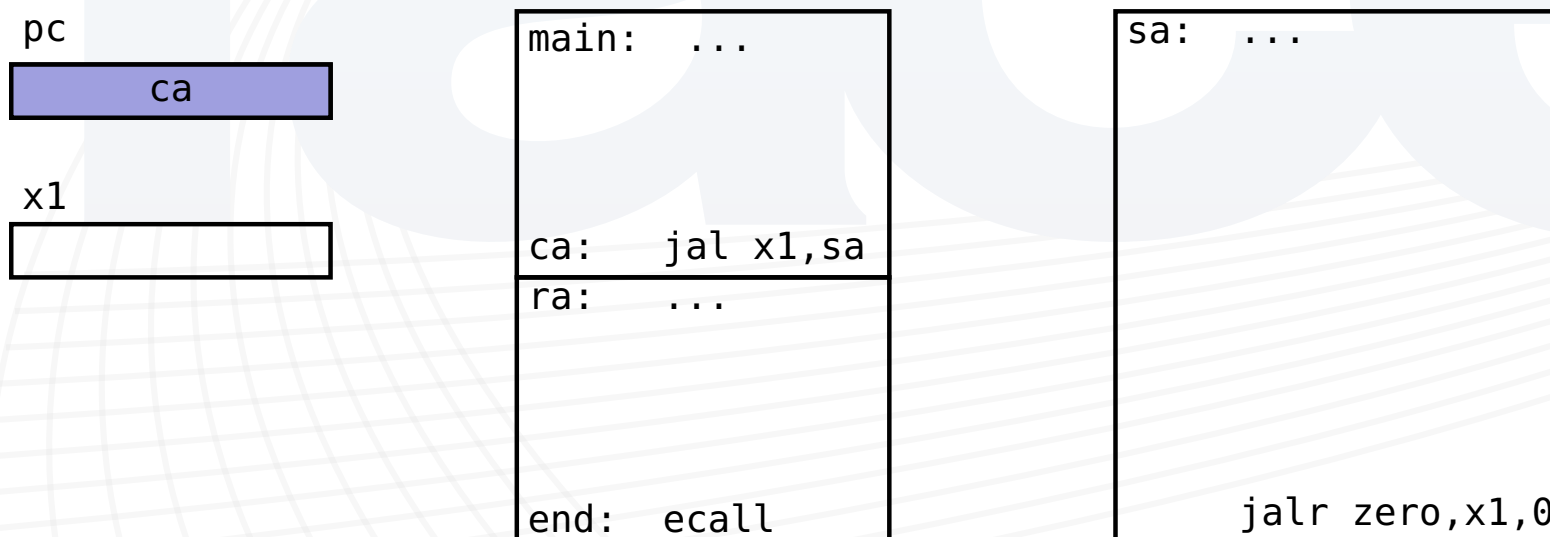
$R[rd] \leftarrow pc + 4; pc \leftarrow R[rs] + desplazamiento;$

- ▶ Instrucción de retorno

jalr zero,rs,0 # $pc \leftarrow R[rs]$

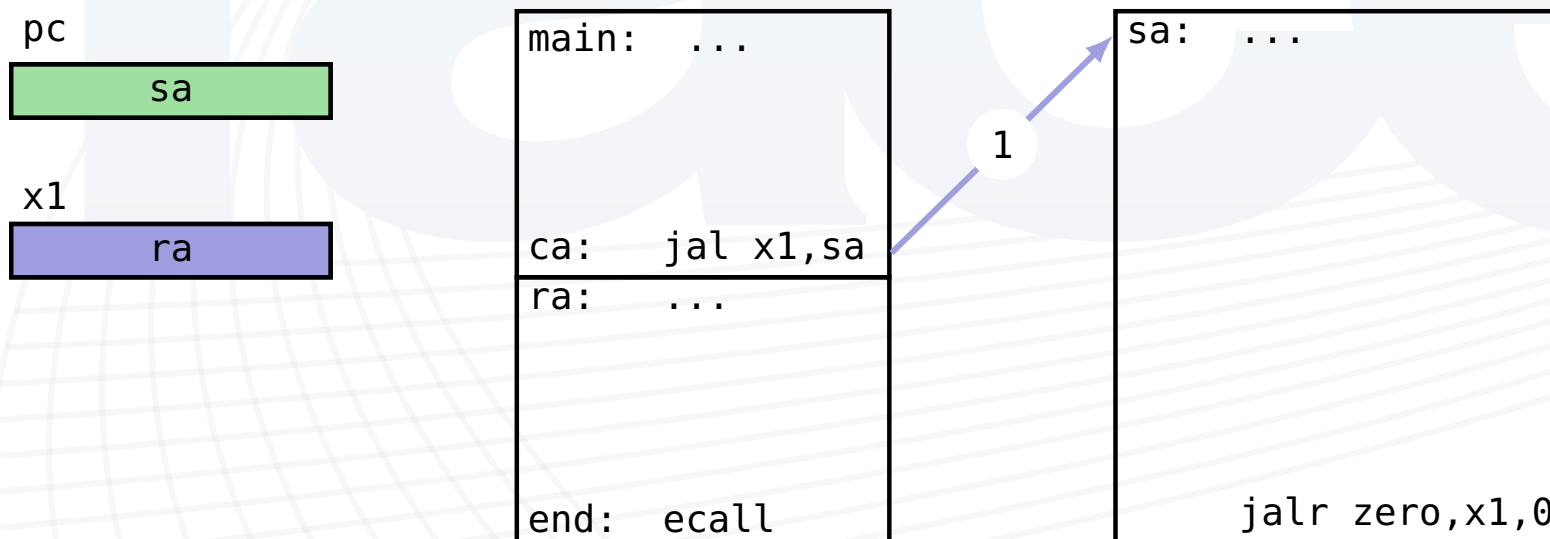
Ejemplo de llamada a subrutinas

- ¿Esta implementación permite a un programa convocar a otra subrutina para hacer parte del trabajo?



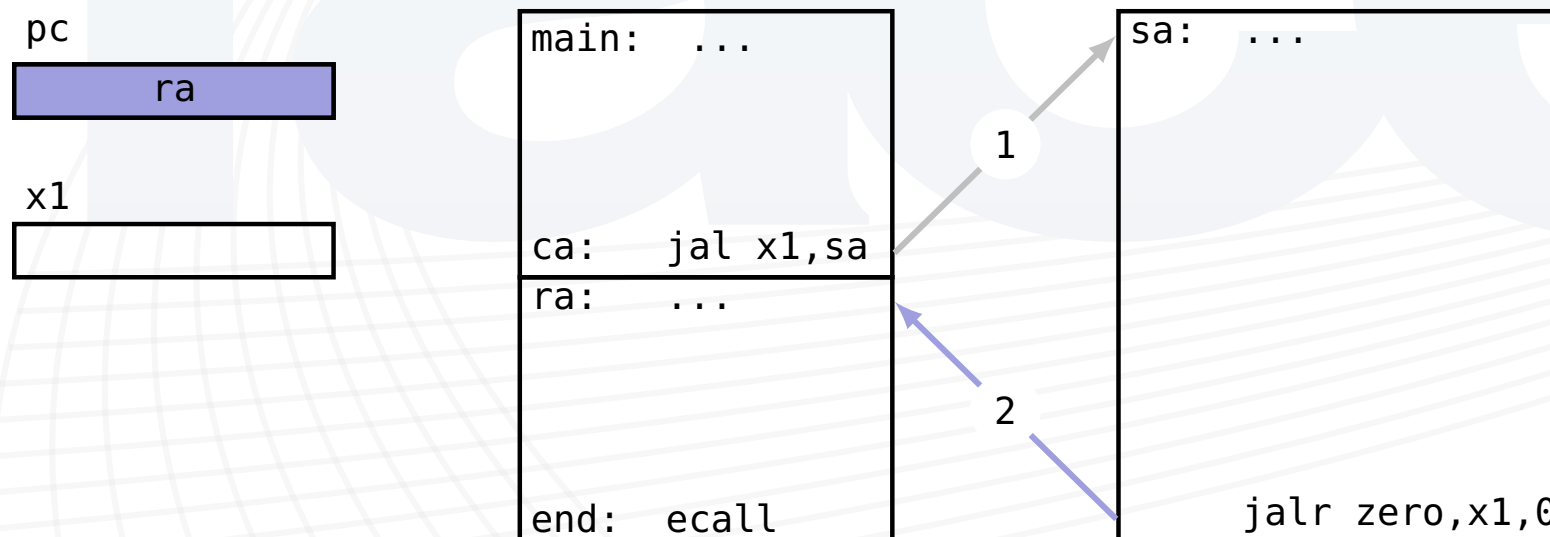
Ejemplo de llamada a subrutinas

- ▶ ¿Esta implementación permite a un programa convocar a otra subrutina para hacer parte del trabajo?
- ▶ La ejecución de **jal** **x1**, **sa** almacena **pc**, que vale **ra**, en **x1** y asigna a **pc** el valor **sa**



Ejemplo de llamada a subrutinas

- ▶ ¿Esta implementación permite a un programa convocar a otra subrutina para hacer parte del trabajo?
- ▶ La ejecución de **jalr zero,0(x1)** descarta el valor actual del **pc** y le asigna el valor **ra** almacenado en el registro **x1**

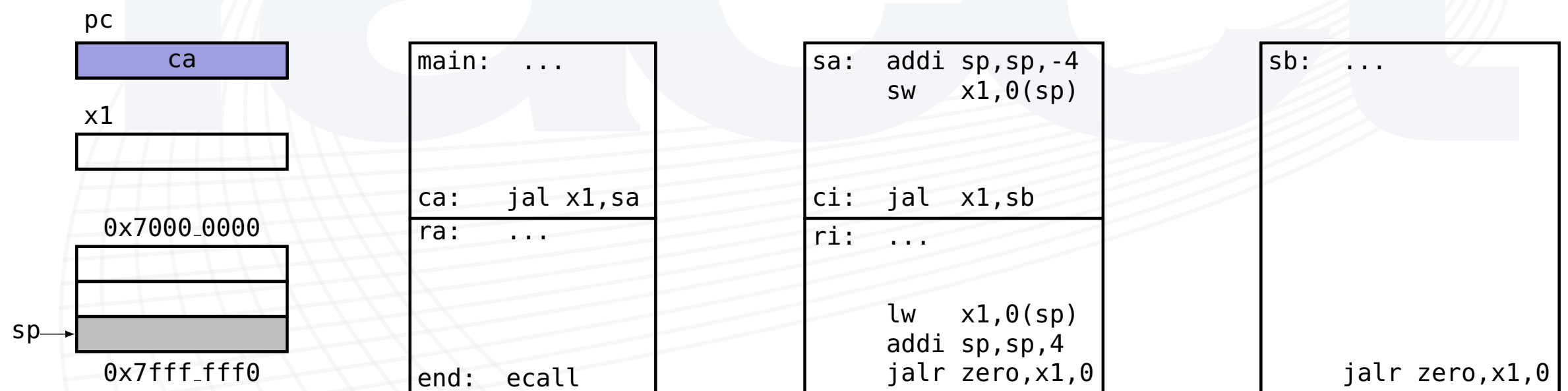


Subrutinas anidadas

- ▶ ¿Esta implementación permite a una subrutina convocar a otra subrutina para hacer parte del trabajo?
 - No porque se pierde el contenido del registro en la segunda llamada
- ▶ Para resolverlo se podría utilizar otro registro para guardar la dirección de retorno, pero esto no es escalable
 - Una mejor alternativa es guardar la dirección de retorno en la pila

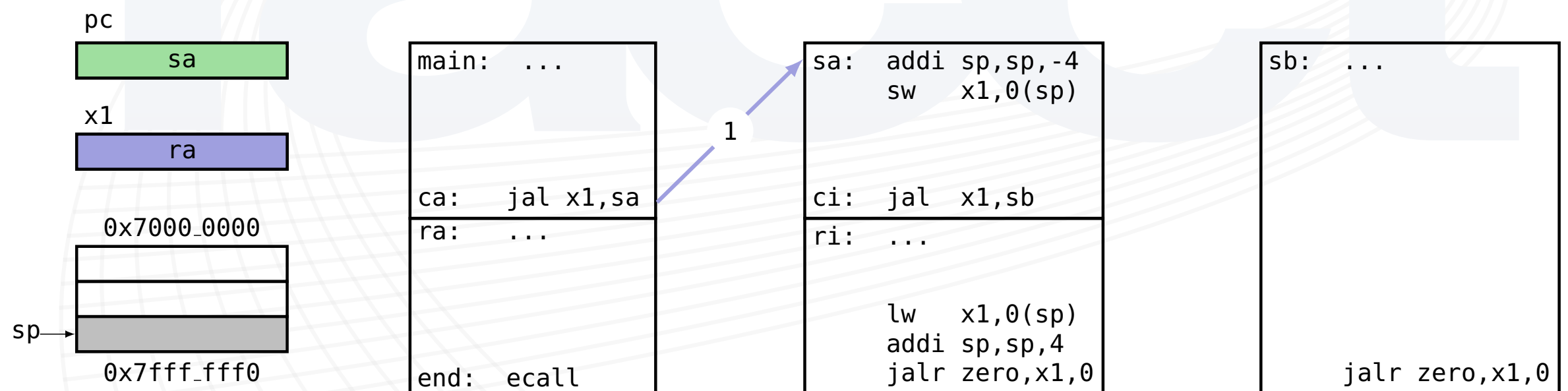
Ejemplo de subrutinas anidadas

- ¿Puede una subrutina convocar a otra subrutina para hacer parte del trabajo?



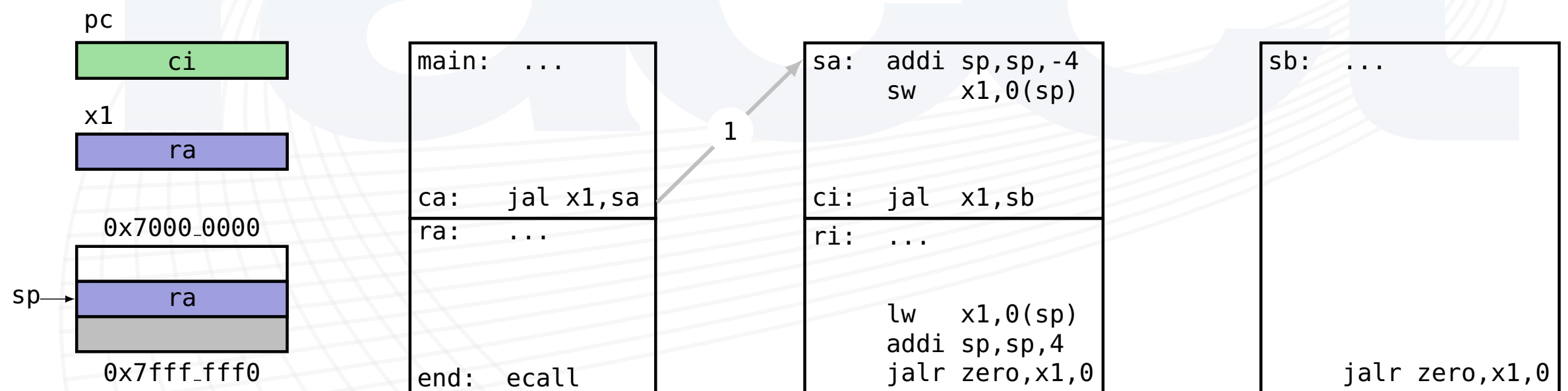
Ejemplo de subrutinas anidadas

- ▶ ¿Puede una subrutina convocar a otra subrutina para hacer parte del trabajo?
- ▶ La ejecución de **jal** **x1**, **sa** almacena **pc**, que vale **ra**, en **x1** y le asigna el valor **sa**



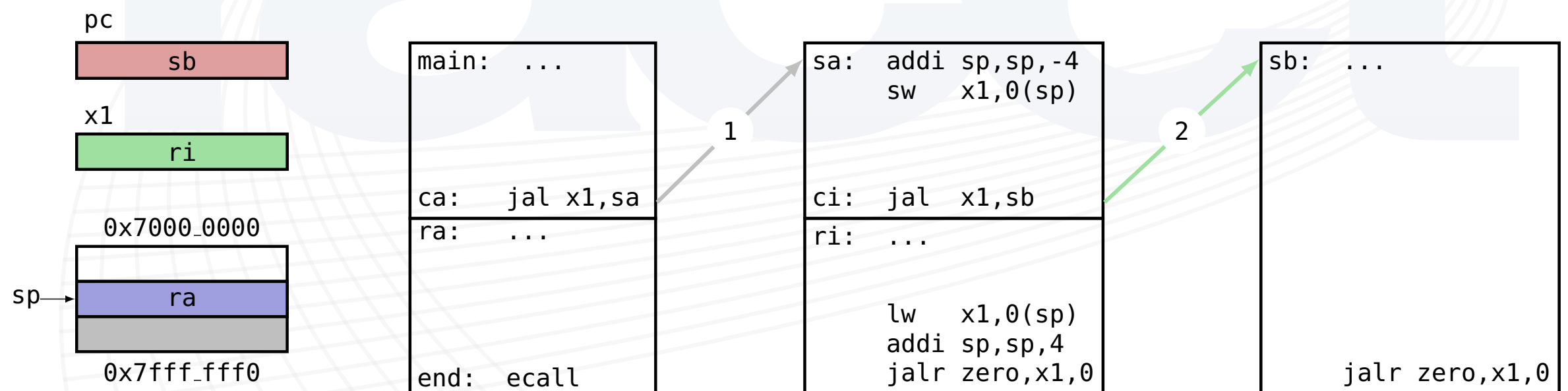
Ejemplo de subrutinas anidadas

- ▶ ¿Puede una subrutina convocar a otra subrutina para hacer parte del trabajo?
- ▶ La subrutina guarda en la pila el valor de retorno **ra** almacenado en el registro **x1**



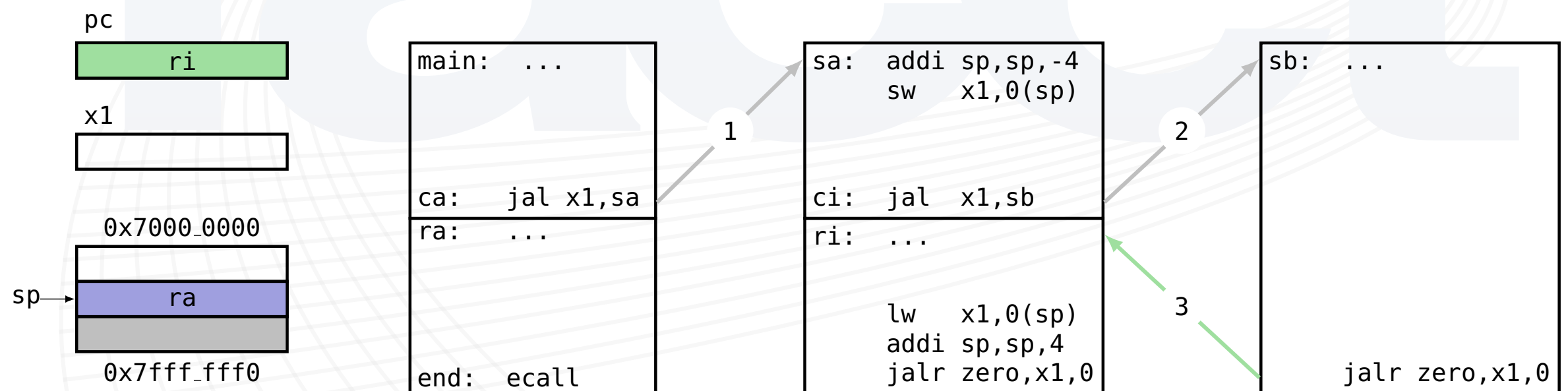
Ejemplo de subrutinas anidadas

- ▶ ¿Puede una subrutina convocar a otra subrutina para hacer parte del trabajo?
- ▶ La ejecución de **jal** **x1**, **sb** almacena **pc**, que vale **ri**, en **x1** y le asigna a **pc** el valor **sb**



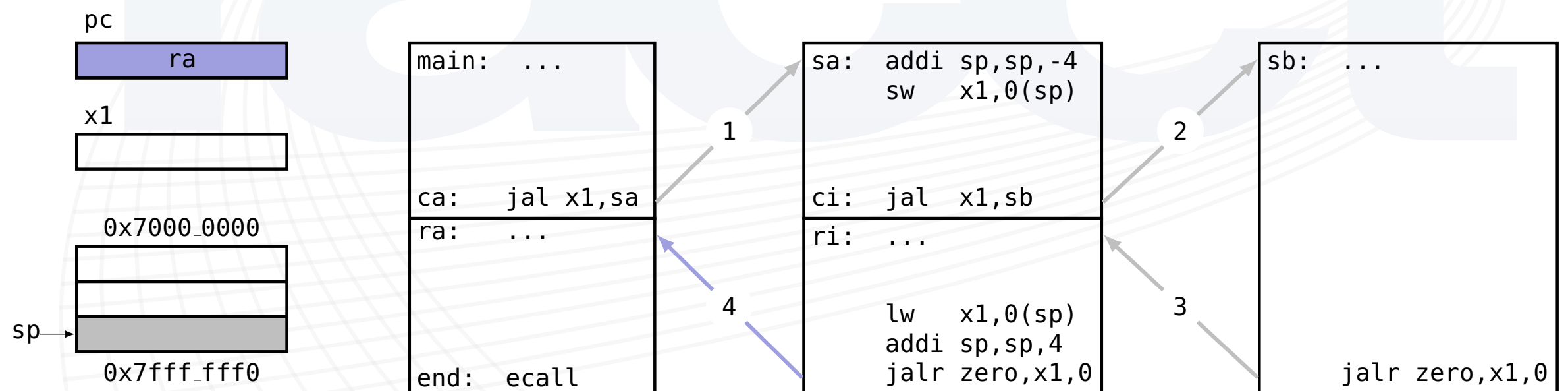
Ejemplo de subrutinas anidadas

- ▶ ¿Puede una subrutina convocar a otra subrutina para hacer parte del trabajo?
- ▶ La ejecución de **jalr zero,0(x1)** descarta el valor actual del **pc** y le asigna el valor **ri** almacenado en el registro **x1**



Ejemplo de subrutinas anidadas

- ▶ ¿Puede una subrutina convocar a otra subrutina para hacer parte del trabajo?
- ▶ La subrutina recupera en **x1** el valor de retorno **ra** almacenado en la pila y retorna con la instrucción **jalr zero,0(x1)**



Antes de empezar: un quizz



Reglas para el uso del stack

- ▶ Se debe hacer uso balanceado del stack, es decir debe haber el mismo número de push y pop
- ▶ Los accesos push o pop no deben realizarse fuera del área asignada de antemano por el diseñador.
- ▶ Ese área debe estar en memoria RAM
 - ▶ No superponerse con otros datos
 - ▶ No superponerse con FLASH
 - ▶ No superponerse con partes no ocupadas

Paso de parámetros

- ▶ Los parámetros son los datos de entrada y los resultados de salida de la subrutina
- ▶ Cuatro formas para el paso de parámetros:
 - ▶ Por registros
 - ▶ En memoria estática
 - ▶ En la pila
 - ▶ Por referencias a memoria
- ▶ Cada caso posee ventajas y desventajas, hay que saber cuándo conviene utilizar cada uno

Paso de parámetros por registros

- ▶ Los parámetros son almacenados en los registros de propósito general antes de la llamada a la subrutina
- ▶ Es la forma más sencilla y rápida para pasar parámetros

Paso de parámetros por registros

- ▶ Si las variables ya están en los registros correctos
- ▶ Si se opera con constantes

```
.text                                # Define la sección de código
reset: li x4,57                      # Carga el primer argumento
      li x5,10                      # Carga el segundo argumento
      jal x1,suma                   # Invoca a la subrutina
final: li a0,10                     # Termina la ejecución sin error
      ecall

suma:  add x3,x4,x5                  # Realiza la suma
      jalr zero,x1,0                # Retorna al programa principal
```

Paso de parámetros por registros

- Si las variables están en memoria

```
.data
num_a: .word 57
num_b: .word 10

.text
reset: la x4,num_a
      lw x4,0(x4)
      la x5,num_b
      lw x5,0(x5)
      jal x1,suma
final: li a0,10
      ecall

suma:  add x3,x4,x5
      jalr zero,x1,0
```

Define la sección de variables
Variable con el primer argumento
Variable con el segundo argumento

Define la sección de código
Apunta a la primera variable
Carga el primer argumento
Apunta a la primera variable
Carga el segundo argumento
Invoca a la subrutina
Termina la ejecución sin error

Realiza la suma
Retorna al programa principal

Paso de parámetros por registros

- ▶ **Ventajas:**
 - ▶ Rapidez y comodidad, sobre todo si la variable ya está en un registro
- ▶ **Desventajas:**
 - ▶ Pocos registros
 - ▶ Imposibilidad de pasar estructuras grandes como parámetro a un subrutina

Paso de parámetros en memoria

- ▶ Los parámetros son almacenados en una zona de memoria conocida por el programa principal y la subrutina
- ▶ Es el equivalente a utilizar variables globales

Paso de parámetros en memoria

```
num_a: .data          # Define la sección de variables
num_b: .space 4        # Reserva espacio para una palabra
resul: .space 4        # Reserva espacio para una palabra

reset: .text          # Define la sección de código
li x3,57              # Carga el primer argumento
la x4,num_a           # Apunta al primer parámetro
sw x3,0(x4)           # Prepara el primer parámetro
li x3,10              # Carga el segundo argumento
la x4,num_b           # Apunta al segundo parámetro
sw x3,0(x4)           # Prepara el segundo parámetro
jal x1,suma           # Invoca a la subrutina
la x3,resul           # Apunta al parámetro de resultado
lw x3,0(x3)           # Recupera el resultado
final: li a0,10        # Termina la ejecución sin error
ecall

suma: la x3,num_a      # Apunta al primer parámetro
lw x3,0(x3)           # Carga el primer argumento
la x4,num_b           # Apunta al segundo parámetro
lw x4,0(x4)           # Carga el segundo argumento
add x3,x3,x4          # Realiza la suma
la x4,resul           # Apunta al parámetro de resultado
sw x3,0(x4)           # Almacena el resultado
jalr zero,x1,0        # Retorna al programa principal
```

Paso de parámetros en memoria

- ▶ Ventajas:
 - ▶ No hay límite de parámetros a pasar.
 - ▶ Bueno para variables globales (la pueden acceder todos los procedimientos)
- ▶ Desventaja: referencia fija a memoria.
 - ▶ Si varias subrutinas comparten un mismo bloque de memoria, entonces esas subrutinas no se pueden llamar entre si
 - ▶ Si hay n rutinas distintas que se puedan llamar entre sí...
¿cuántas áreas independientes debo tener?
 - ▶ Se pueden mitigar usando bloques en lugar de variable individuales

Paso de parámetros en memoria

```
base:    .data                # Define la sección de variables
        .space 12            # Reserva un espacio de tres palabras

        .text                # Define la sección de código
reset:   la    x3,base         # Apunta al bloque de parámetros
        li    x4,57           # Carga el primer argumento
        sw    x4,0(x3)        # Prepara el primer parámetro
        li    x4,10           # Carga el segundo argumento
        sw    x4,4(x3)        # Prepara el segundo parámetro
        jal   x1,suma         # Invoca a la subrutina
        la    x3,base         # Apunta al bloque de parámetros
        lw    x3,8(x3)        # Recupera el resultado del bloque
final:   li    a0,10          # Termina la ejecución sin error
        ecall

suma:    la    x3,base         # Apunta al bloque de parámetros
        lw    x4,0(x3)        # Carga el primer argumento del bloque
        lw    x5,4(x3)        # Carga el segundo argumento del bloque
        add   x4,x4,x5        # Realiza la suma
        sw    x4,8(x3)        # Almacena el resultado en el bloque
        jalr  zero,x1,0       # Retorna al programa principal
```

Paso de parámetros en la pila

- ▶ Los parámetros son almacenados en la pila antes de llamar a la subrutina
- ▶ Los resultados producidos por la subrutina tambien se colocan en la pila, en un espacio reservado por el programa principal al efectuar el llamado

Paso de parámetros en la pila

```
.text                                # Define la sección de código
reset: addi sp,sp,-12                # Reserva espacio para tres palabras
      li    x4,57                    # Carga el primer argumento
      sw    x4,8(sp)                # Prepara el parámetro en la pila
      li    x4,10                   # Carga el segundo argumento
      sw    x4,4(sp)                # Prepara el parámetro en la pila
      jal   x1,suma                 # Invoca a la subrutina
      lw    x4,0(sp)                # Recupera el resultado de la pila
      addi  sp,sp,12                # Libera el espacio reservado
final: li    a0,10                  # Termina la ejecución sin error
      ecall

suma:  lw    x4,8(sp)                # Carga el primer argumento de la pila
      lw    x5,4(sp)                # Carga el segundo argumento de la pila
      add   x4,x4,x5                # Realiza la suma
      sw    x4,0(sp)                # Almacena el resultado en la pila
      jalr  zero,x1,0               # Retorna al programa principal
```

Paso de parámetros en la pila

▶ Ventajas

- ▶ Número ilimitado de parámetros
- ▶ No hace falta m áreas de memoria
- ▶ El programador no se preocupa por “cuidar áreas”
- ▶ Rutinas anidadas no presentan problemas

▶ Desventajas

- ▶ Copiar los datos en la pila demora tiempo
 - ▶ Puede ser demasiado tiempo y lugar si son muchos datos
- ▶ A veces se accede a datos que no están en el tope de la pila

Pasaje por referencia a memoria

- ▶ Los parámetros están en un bloque de memoria tal como los utiliza el programa principal
- ▶ La dirección inicial, y la dimensión del bloque si hace falta, se ponen en registros o en la pila
- ▶ Si la rutina entrega un bloque como resultado, también pasa la dirección inicial y la dimensión de este bloque
 - ▶ Tambien en la pila o en registros

Paso por referencia en registros

```
base: .data # Define la sección de variables
      .space 12 # Reserva un espacio de tres palabras

reset: .text # Define la sección de código
      la x3, base # Apunta al bloque de parámetros
      li x4, 57 # Carga el primer argumento
      sw x4, 0(x3) # Prepara el primer parámetro
      li x4, 10 # Carga el segundo argumento
      sw x4, 4(x3) # Prepara el segundo parámetro
      jal x1, suma # Invoca a la subrutina
      lw x3, 8(x3) # Recupera el resultado del bloque
final: li a0, 10 # Termina la ejecución sin error
      ecall

suma: lw x4, 0(x3) # Carga el primer argumento del bloque
      lw x5, 4(x3) # Carga el segundo argumento del bloque
      add x4, x4, x5 # Realiza la suma
      sw x4, 8(x3) # Almacena el resultado en el bloque
      jalr zero, x1, 0 # Retorna al programa principal
```

Paso por referencia en la pila

```
base: .data # Define la sección de variables
      .space 12 # Reserva un espacio de tres palabras

reset: .text # Define la sección de código
      la x3, base # Apunta x3 al bloque de parámetros
      li x4, 57 # Carga en x4 el primer argumento
      sw x4, 4(x3) # Salva el primer argumento en el bloque
      li x4, 10 # Carga en x4 el segundo argumento
      sw x4, 8(x3) # Salva el segundo argumento en el bloque
      addi sp, sp, -4 # Reserva espacio para una palabra
      sw x4, 4(x3) # Prepara el puntero al bloque en la pila
      jal x1, suma # Invoca a la subrutina
      lw x3, 0(sp) # Recupera el puntero al bloque de la pila
      addi sp, sp, 4 # Libera el espacio reservado en la pila
      lw x3, 0(x3) # Recupera el resultado del bloque
final: li a0, 10 # Termina la ejecución sin error
      ecall

suma: la x3, 0(sp) # Carga el puntero al bloque de parámetros
      lw x4, 0(x3) # Carga el primer argumento del bloque
      lw x5, 4(x3) # Carga el segundo argumento del bloque
      add x4, x4, x5 # Realiza la suma
      sw x4, 8(x3) # Almacena el resultado en el bloque
      jalr zero, x1, 0 # Retorna al programa principal
```

Pasaje por referencia a memoria

- ▶ **Ventajas:**
 - ▶ No hay límite en cantidad de parámetros
 - ▶ No hay problemas de anidamientos si uso la pila
 - ▶ El programa principal no gasta tiempo moviendo datos de un lugar a otro
- ▶ **Desventajas:**
 - ▶ Complicado para pocos datos, y lento en ese caso
 - ▶ Nadie dice que sólo se debe emplear un único método
 - ▶ Se lo puede combinar con otros métodos, vistos antes

Variables globales

- ▶ Son comunes a todas las subrutinas o procedimientos
- ▶ Antes de comentar la ejecución del programa se asigna un espacio suficiente en memoria
- ▶ Este espacio de memoria permanece ocupado durante toda la ejecución del programa
- ▶ Cualquier parte del programa puede acceder para leer o modificar el valor almacenado en la variable

Variables locales

- ▶ Solo existen en un ámbito determinado, generalmente un subprograma
- ▶ La asignación de memoria se realiza durante la ejecución del programa al ingresar en el ámbito de la variable
- ▶ Solo el fragmento de programa que se encuentra en el mismo ámbito puede acceder a la variable
- ▶ Al completar la ejecución del ámbito el espacio asignado es liberado y la variable deja de existir

Uso de la pila para variables locales

- ▶ Al ingresar al ámbito de la variable se reserva espacio en la pila moviendo el puntero de pila tantos bytes como sea necesario
- ▶ Al salir del ámbito se retorna el espacio reservado devolviendo al puntero de pila al valor original
- ▶ El valor del puntero de pila antes de la reserva es un puntero a la variable local

Ejemplo de variables locales

```
reset: .text
       addi sp,sp,-12
       mv   x3,sp
       li   x4,57
       sw   x4,0(x3)
       li   x4,10
       sw   x4,4(x3)
       jal  x1,suma
       lw   x3,8(x3)
       addi sp,sp,-12
final: li   a0,10
       ecall

suma:  addi sp,sp,-8
       sw   x4,0(sp)
       sw   x5,4(sp)
       lw   x4,0(x3)
       lw   x5,4(x3)
       add  x4,x4,x5
       sw   x4,8(x3)
       lw   x5,4(sp)
       lw   x4,0(sp)
       addi sp,sp,8
       jalr zero,x1,0
```

Define la sección de código
Reserva espacio para tres palabras
Obtiene un puntero al bloque de parámetros
Carga el primer argumento
Prepara el primer parámetro
Carga el segundo argumento
Prepara el segundo parámetro
Invoca a la subrutina
Recupera el resultado del bloque
Libera el espacio reservado en la pila
Termina la ejecución sin error

Reserva espacio para dos palabras
Conserva el valor del primer registro
Conserva el valor del segundo registro
Carga el primer argumento del bloque
Carga el segundo argumento del bloque
Realiza la suma
Almacena el resultado en el bloque
Conserva el valor del primer registro
Conserva el valor del segundo registro
Libera el espacio reservado en la pila
Retorna al programa principal

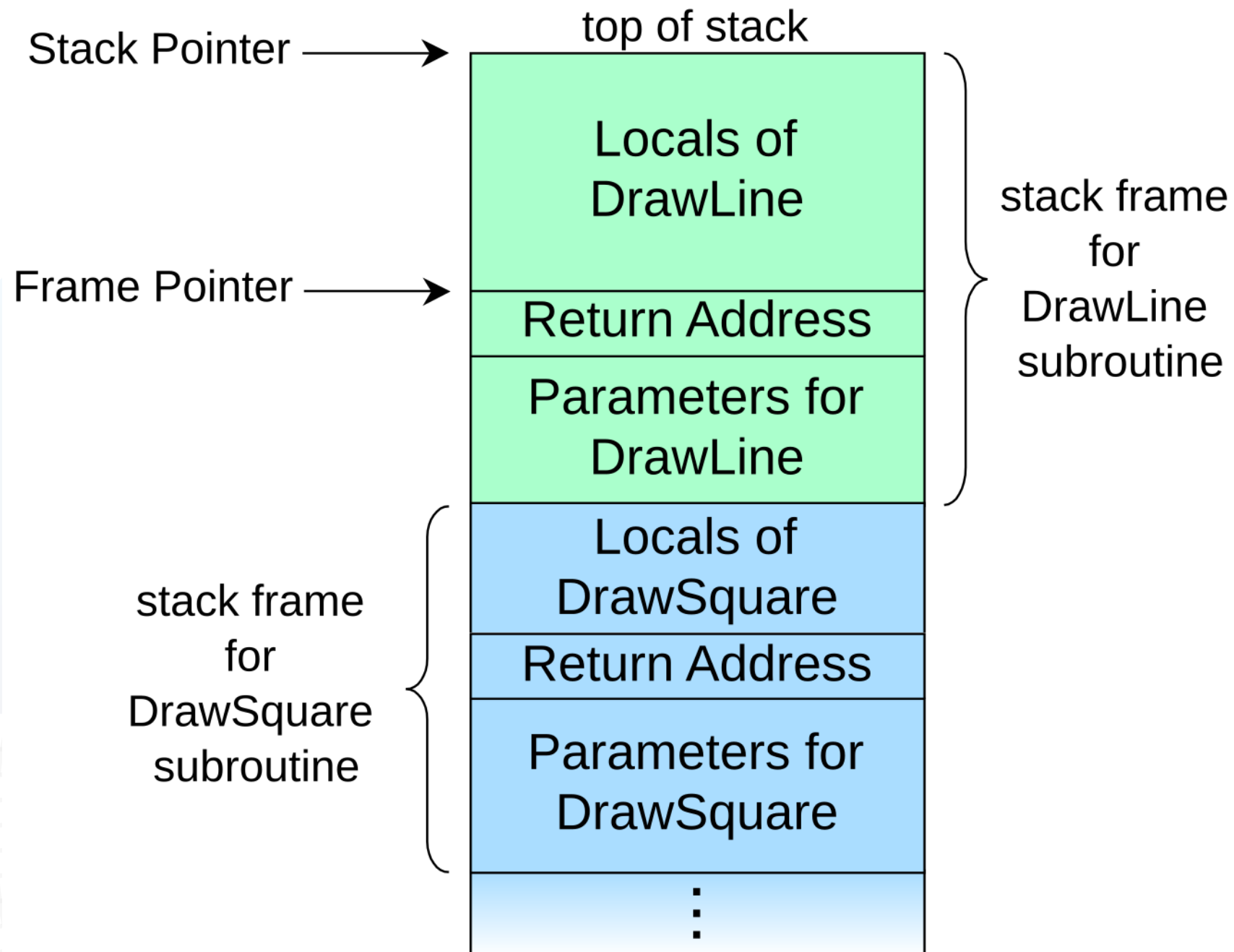
Uso de la pila para variables locales

- ▶ Ocupan lugar solo cuando se usan
 - Dynamic Allocation/Release
- ▶ Protección de datos: solo son visibles mientras se ejecuta el procedimiento
- ▶ Código reentrante: se puede ejecutar una segunda copia antes de terminar la primera
- ▶ Código relocizable: se puede mover el código en memoria sin recompilar

Puntero de marco (Stack frame)

- ▶ El puntero de pila puede variar y esto dificulta acceder a las variable locales
- ▶ El puntero al marco de pila sirve para resolver ese problema:
 - ▶ Conserva el valor del puntero de pila antes de reservar el espacio para las variables locales
 - ▶ Permite el acceso a las variables locales en forma indexada respecto al puntero de marco de pila que es constante

Puntero de marco (Stack frame)



Paso de parámetros (standard)

- ▶ Varias formas de pasar parámetros
- ▶ Tanto registros como stack tienen ventajas, pero se pueden combinar:
 - ▶ Algunas entradas podrían venir en registros
 - ▶ Otras en la pila
 - ▶ Resultado en registros o la pila
- ▶ Convención para llamados a procedimientos.
 - ▶ Application Binary Interface (ABI) – compatible con C
 - ▶ Permite que una rutina puede llamarse desde otro lenguaje
 - ▶ Permite poder usar bibliotecas de rutinas de terceras partes

Convención de llamada (ILP32)

- ▶ Convención ABI para todas las arquitecturas RISC-V
- ▶ Se definen cuatro variantes: para 32 y 64 bits, general y con implementación de punto flotante por software
- ▶ Se asignan nombres y funciones específicas a los registros
- ▶ Se definen los responsables de conservar los valores almacenados en cada registro

Convención en el uso de registros

Registro	Nombre ABI	Descripción	Conservado
x0	zero	Hard-wired zero	No aplica
x1	ra	Return address	No
x2	sp	Stack pointer	Si
x3	gp	Global pointer	No aplica
x4	tp	Thread pointer	No aplica
x5 a x7	t0 a t2	Temporaries	No
x8	s0 o fp	Saved register / frame pointer	Si
x9	s1	Saved register	Si
x10 a x11	a0 a a1	Function arguments / return values	No
x12 a x17	a2 a a7	Function arguments	No
x18 a x27	s2 a s11	Saved registers	Si
x28 a x31	t3 a t6	Temporaries	No

Convención de llamada (ILP32)

- ▶ Los parámetros se pasan por registros siempre que sea posible usando los registros **a0** a **a7**
- ▶ Para los tipos de datos simples con tamaño menor a 32 bits se usan los bits menos significativos de un registro
- ▶ Para tipos de datos de 64 bits se utilizan dos registros, empezando siempre en un registro par, por ejemplo **a2:a3**
- ▶ Los parámetros que no se pueden pasar en registros se pasan en la pila, con **sp** apuntando al primer parámetro que no se pasa por registro
- ▶ Los bloques de datos se manejan directamente en memoria utilizan un registro como puntero
- ▶ Los resultados se devuelven en los registros **a0** y **a1**

Abstracción

- ▶ Tiene por objetivo independizar las subrutinas del programa principal
- ▶ Una persona puede emplear la rutina, sabiendo cuál es su función y sin saber cómo funciona internamente
- ▶ El usuario se **abstrae** de los detalles de implementación
- ▶ Cada subrutina debe respetar la convención ABI
- ▶ Se pueden integrar las subrutinas en librerías y reutilizarlas en futuros proyectos

Controlador de dispositivos

- ▶ La abstracción nos permite modularizar nuestro código y brinda la opción de exponer al usuario lo que queremos que vea y esconder lo que no deseamos que toque
- ▶ Un controlador de dispositivo (device driver) es un buen ejemplo del uso de abstracción para exponer rutinas públicas que deseamos que el usuario del controlador emplee
- ▶ Se usan rutinas privadas para esconder detalles internos del controlador

Subrutinas recursivas

- ▶ Rutinas que se llaman a sí mismas
 - De manera directa o indirecta
- ▶ Existen problemas de naturaleza recursiva
- ▶ Dos condiciones para que funcione bien:
 - Código puro: el programa no se modifica a si mismo
 - Areas de datos independientes por cada llamado: la pila cumple esta condición sin problema
- ▶ Código reentrante: se ejecutan mas de una instancia del mismo código simultáneamente.