antidad de hojas
5

Organización del Computador 2
Primer parcial – 11/05/17

A EH

1 (35)	2 (35)	3 (30)	
25	25	30	80

Normas generales

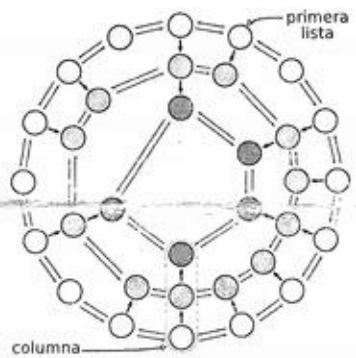
- Numere las hojas entregadas. Complete en la primera hoja la cantidad total de hojas entregadas.
- Entregue esta hoja junto al examen, la misma **no** se incluye en la cantidad total de hojas entregadas.
- Está permitido tener los manuales y los apuntes con las listas de instrucciones en el examen. Está prohibido compartir manuales o apuntes entre alumnos durante el examen.
- Cada ejercicio debe realizarse en hojas separadas y numeradas. Debe identificarse cada hoja con nombre, apellido y LU.
- La devolución de los exámenes corregidos es personal. Los pedidos de revisión se realizarán por escrito, antes de retirar el examen corregido del aula.
- Los parciales tienen tres notas: I (Insuficiente): 0 a 59 pts, A- (Aprobado condicional): 60 a 64 pts y A (Aprobado): 65 a 100 pts. No se puede aprobar con A- ambos parciales. Los recuperatorios tienen dos notas: I: 0 a 64 pts y A: 65 a 100 pts.

Ej. 1. (35 puntos)

Sea la siguiente estructura de listas doblemente enlazadas encadenadas entre si.

```
struct supernode {
supernode* abajo,
supernode* derecha,
supernode* izquierda,
int dato }
```

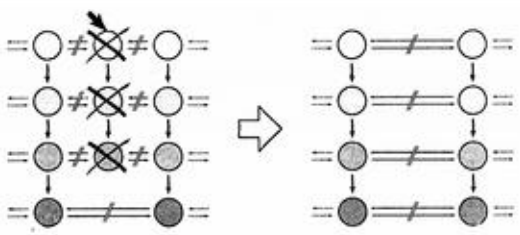
ejemplo



- todos los nodos pertenecen a una lista doblemente enlazada
- todos los nodos son referenciados desde algun otro nodo en otra lista (excepto en la primera)
- todas las listas respetan el orden de los nodos que las apuntan

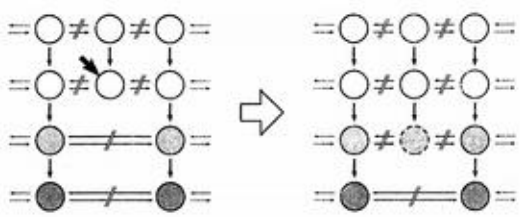
Implementar en ASM las siguientes funciones.

(20p) a. `void borrar_columna(supernode** sn)`: Dada un doble puntero a nodo dentro de la primera lista, borra una columna de nodos. Modifica el doble puntero dejando un nodo valido de la primer lista.



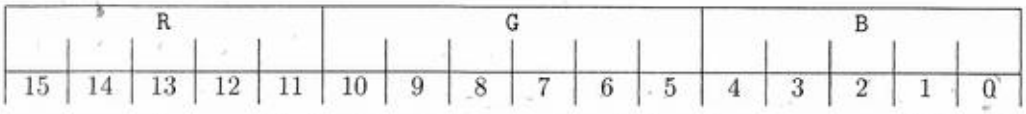
(15p) b. `void agregar_abajo(supernode** sn, int d)` Agrega un nuevo nodo a la lista inmediata inferior del nodo apuntado. Considerar que el nodo donde agregar puede no tener vecinos inmediatos en la lista inferior.

seguro tiene un nodo



Ej. 2. (35 puntos)

Un pixel es codificado como 3 componentes `RGB`.
En una codificación particular se utiliza 5 bits para `R`, 6 para `G` y 5 para `B`, almacenados en 2 bytes.



Se tiene una imagen de `m` × `n` pixeles almacenados en esta codificación, con n y m multiplos de 8.
Implementar en ASM usando instrucciones de SIMD las siguientes funciones:

(20p) a. `void to24(img16* src, int m, int n, img24** dst)`: Convierte una imagen almacenada en `src` a pixeles de 24 bits (1 byte por componente en orden `RGB`). Almacenando el resultado en `dst`. Cada componente debe ser escalado según se indica a continuación:

(15p) b. `void toBN(img16* src, int m, int n, img8** dst)`: Convierte una imagen almacenada en `src` a pixeles de una sola componente de 8 bits, cada pixel corresponde al promedio de cada componente escalada a 8bits según muestra el ejercicio anteriror, es decir $(R+G+B)/3$. Almacenando el resultado en `dst`.

En ambas funciones se debe solicitar memoria donde guardar el resultado y retornar el puntero por `dst`.

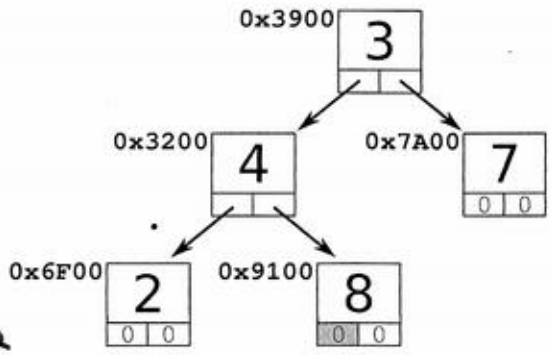
Ej. 3. (30 puntos)

Considerando el código a continuación, que realiza la sumatoria de todos los valores dentro de un arbol binario, respetando el siguiente *struct*: `struct node { node* izq, node* der, int var }`

+0 +8 +16

```
0xA000| suma: push   rbp
0xA001|       mov    rsp, rbp
0xA004|       push   rbx
0xA005|       push   r12
0xA007|       mov    rbx, rdi
0xA00a|       mov    eax, 0
0xA00f|       cmp    rdi, 0
0xA013|       je     .fin
0xA015|       mov    r12d, [rbx + 16]
0xA019|       mov    rdi, [rbx]
0xA01c|       call   suma
0xA021|       add    r12d, eax
0xA024|       mov    r12d, [rbx + 8]
0xA028|       call   suma
0xA02d|       add    eax, r12d
0xA030| .fin: pop    r12
0xA032|       pop    rbx
0xA033|       pop    rbp
0xA034|       ret
```

Caso:



Notar que cada caja es un nodo, los numeros en hexadecimal corresponden a sus direcciones en memoria.

(15p) a. Dibujar el estado de la pila en **hexadecimal** para la ejecución del algoritmo `suma` sobre el arbol de la figura. Se debe dibujar la pila hasta el momento en que el algoritmo es llamado con el puntero al calor en gris de la figura.

(15p) b. Construir una función en ASM que dado un puntero al tope de la pila, devuelve la profundida del arbol recorrido hasta el momento. Considerar que se ejecuta la función `suma` para cualquier arbol y que el puntero fue obtenido obtenido luego de crear el *stack frame*.

11/05/17

①

a) Implementar "void borrar_columna (*sqnode** sn)".

```
%define OFF_ABAJO  0
%define OFF_DER    8
%define OFF_IZQ    16
%define NULL       0

BORRAR-COLUMNA:

    CMP [RDI], NULL
    JE .FIN

    PUSH RBP
    MOV RBP, RSP

    PUSH RBX
    MOV RBX, [RDI]   ; RDi = *sn vertical actual
    SUB RSP,8        ; ALINEADA PILA
;modifico puntero
    MOV R8, [RDi]
    MOV R8, [R8+OFF-DER]
    CMP R8, [RDi]
    JE .LISTA-VACÍA
    MOV [RDi],R8
    JMP .NO-VACÍA
.LISTA-VACÍA: MOV [RDi], NULL ; se apuntaba a sí mismo
.NO-VACÍA:
```

(Idem izquierda)

```
.CICLO:
    CMP RBX, NULL
    JE .FIN-CICLO
    MOV R8, [RBX + OFF-DER]
    MOV R9, [RBX+OFF-IZQ]
    MOV [R8+OFF-IZQ], R9
    MOV [R9+OFF-DER], R8
```

Que pasa con la derecha del de la derecha cuando hay solo un hermano? ⊗

Continuation al dorsing

```
    MOV RDi, RBX                    ;prepara para liberar memoria
    MOV RBX, [RBX+OFF_ABAJO]        ; avanza el puntero
    CALL FREE
    JMP .ciclo
.FIN_CICLO
ADD RSP, 8
POP RBX
POP RBP
.FIN: ret
```

b) Implementar "void agregar_abajo (suprade** sn, int d)"

(mismas def. que antes)

```
%define SN_SIZE 28
%define OFF_DATO 24
```

⊗ 32! (no es packed)

```
AGREGAR_ABAJO:
    CMP [RDi], NULL
    JE .FIN
    PUSH RBP
    MOV RBP, RSP
    MOV RDi, [RDi]  ; desreferencio doble puntero
    ;busco nodo con puntero para abajo
    .ciclo:
        CMP [RDi+OFF_ABAJO], NULL
        JNE .FIN_CICLO
        MOV RDi, [RDi+OFF_DER]
        JMP .ciclo
    .FIN_CICLO
    PUSH RDi
    PUSH RSi  ; ALINEABA PILA
    MOV RDi, SN_SIZE
    CALL MALLOC
    POP RSi
    POP RDi
```

⊗ Se indefine si el nuevo nodo no tiene hermanos (Solo comentarlo)

→ Sigue en next página

PATOS
MENDEZ
NICOLAS

LV: 709/15

11/05/17

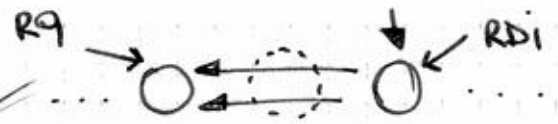
```
MOV RDi, [RDi+OFF_ABAJO]
MOV R8, [RDi + OFF_IZQ]

MOV [RAX+OFF_DER], RDi
MOV [RAX+OFF_IZQ], R8
MOV *[RAX+OFF_ABAJO], NULL ; (*qword, no me entra ahora)
MOV [RAX+OFF_DATO], esi

MOV [R8+OFF_DER], RAX
MOV [RDi+OFF_IZQ], RAX

POP RBP

.FIN: RET
```



11/05/17

(2)

a) "void to24 (img16* src, int m, int n, img24** dst)" RDI RSI RDX RCX

```
section .rodata:

LIMPIA_ARRIBA_64: DQ 0x00000000FFFFFFFF
LIMPIA_ABAJO_64: DQ 0xFFFFFFFF00000000
VERDES: DQ 0x07E007E007E007E0, 0x07E007E007E007E0
ROJOS: DQ 0xF800F800F800F800, 0xF800F800F800F800
AZULES: DQ 0x001F001F001F001F, 0x001F001F001F001F

section .text
TO24:
    PUSH RBP
    MOV RBP, RSP

    PUSH RBX

    MOV EAX, ESI
    MUL EDX
    AND RAX, [LIMPIA_ARRIBA_64]
    AND RDX, [LIMPIA_ABAJO_64]
    OR RAX, RDX

    MOV RBX, RAX
    PUSH RDI
    PUSH RCX
    MOV RDI, RAX
    ADD RDI, RDI
    ADD RDI, RDI
    SUB RSP, 8 ; ALINEADA
    CALL MALLOC
    ADD RSP, 8
    POP RCX
    POP RDI
    MOV [RCX], RAX ; copio a puntero destino

    MOV RSI, RAX ; RSI = *destino
    MOV RCX, RBX ; RCX = mxn (#pixels)
    SLR RCX, 3 ; RCX/8 (#iteraciones levantando 8 pix)
```

¿Si el numero que vas a multiplicar entra en 32 bits, podes usar regs de 64 en la mul y asegurase que el resultado entra en un reg RAX ← mxn de 64 (la parte alta va a ser 0's) (Solo comentario)

LINEAS "WINDOWS"

⊗ Esto multiplica por 4 no por 3 (la imagen dst es de 24 bits)

sigue....

• CICLO :

```
MOVDQU XMM0, [RDi]
MOVDQU XMM1, XMM0
MOVDQU XMM2, XMM0

PAND XMM2, [ROJOS]
PAND XMM1, [VERDES]
PAND XMM0, [AZULES]

PSRLW XMM1, 5  ; verdes al principio
PSRLW XMM2, 11 ; rojos al principio
```

(*) PSLLW XMM0, 2
PSLLW XMM1, 2
PSLLW XMM2, 2

Hasta acá es "con iqual" el ej. (b), ~~[illegible]~~ incluyendo (*).

(*) MARCA

```
MOVDQU XMM3, XMM0
MOVDQU XMM4, XMM1
MOVDQU XMM5, XMM2

PSHUFB XMM0, [ORDENA_AZULES]
PSHUFB XMM1, [ORDENA_4VERDES]
PSHUFB XMM2, [ORDENA_4ROJOS]

POR XMM0, XMM1
POR XMM0, XMM2

PSRLDQ XMM3, 4
PSRLDQ XMM4, 4
PSRLDQ XMM5, 4

PSHUFB XMM3, [ORDENA_4AZULES]
PSHUFB XMM4, [ORDENA_4VERDES]
PSHUFB XMM5, [ORDENA_4ROJOS]

POR XMM3, XMM4
POR XMM3, XMM5

MOVDQU [RSi], XMM0
ADD RSi, 12 ; avanzo 12 bytes ≡ 4 pixels (destino)

MOVDQU [RSi], XMM3
ADD RSi, 12 ; 12 bytes ≡ 4 pix

ADD RDi, 16 ; avanzo 16 bytes destino ≡ 8 pixels

LOOP .ciclo

POP RBX
POP RBP
RET
```

DI MARCA DI MARCA

⊗ Si quiero hacer los siguientes 4 pixeles son 4*2=8 bytes

Nota: no llegué a tiempo a hacer los mezcados los shuffles de bytes lo que hacen es dejar ~~[illegible]~~ la parte baja de los primeros 4 words (4 bytes) en su posición para armar el canal a 24 bits.

OK

11/05/17

b)

El código es igual al "a)" hasta (MARCA), pero pidiendo mxn a malloc (es decir, sin las "LINEAS windows").

```
... (MARCA)                                   ; con α_i = R_i + G_i + B_i
PADDUSW XMM0, XMM1
PADDUSW XMM0, XMM2   ; XMM0: [α7|α6|α5|α4|α3|α2|α1|α0]

MOV? [ilegible]
MOVQU XMM1, XMM0     ; XMM1: [ " ]
PXOR XMM7, XMM7      ; XMM7: [ 0 . . . . 0 ]

PUNPCKLWD XMM0, XMM7 ; XMM0: [ α3 | α2 | α1 | α0 ]
PUNPCKHWD XMM1, XMM7 ; XMM1: [ α7 | α6 | α5 | α4 ]

MOV XMM6, [TRES]

CVTPI2PS XMM6
CVTPI2PS XMM1
CVTPI2PS XMM0

DIVPS XMM0, XMM6     ; XMM0: [ α3/3 | α2/3 | α1/3 | α0/3 ]
DIVPS XMM1, XMM6     ; XMM1: [ α7/3 | α6/3 | α5/3 | α4/3 ]

CVTPS2PI XMM1
CVTPS2PI XMM0

                                              ; β_i = α_i / 3
PACKUSDW XMM0, XMM1  ; XMM0: [β3|β2|β5|β4|β3|β2|β1|β0]
PACKUSWB XMM0, XMM0  ; XMM0: [β7 ... β0 | β7 ... β0]

[ilegible]
MOVQU [RSI], XMM0

ADD RSi, 8
ADD RDi, 16
LOOP, CICLO
POP RBX
POP RBP RET
```

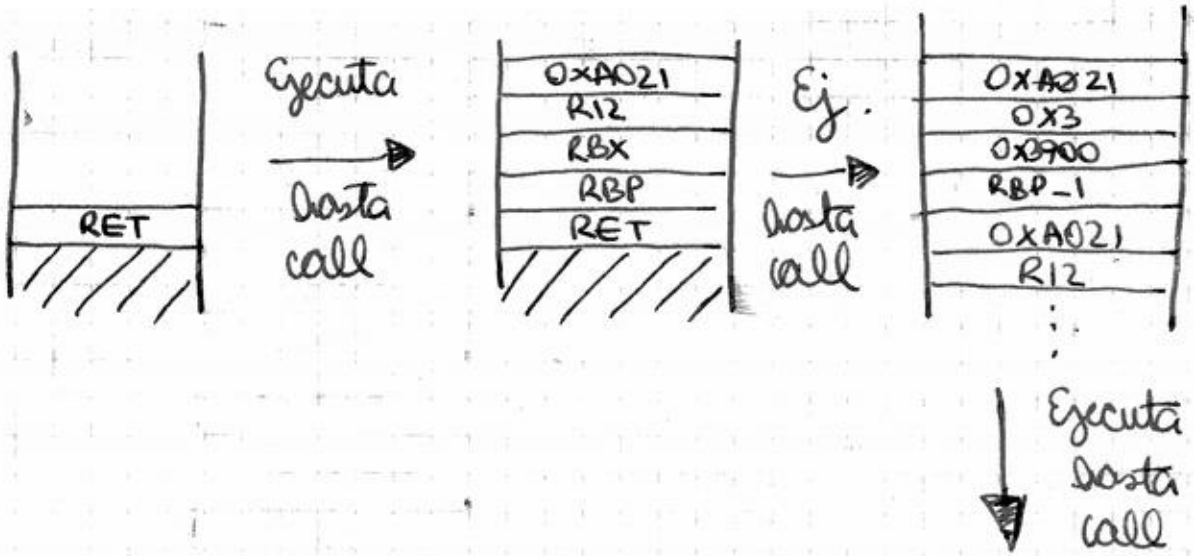
Máscara producto:

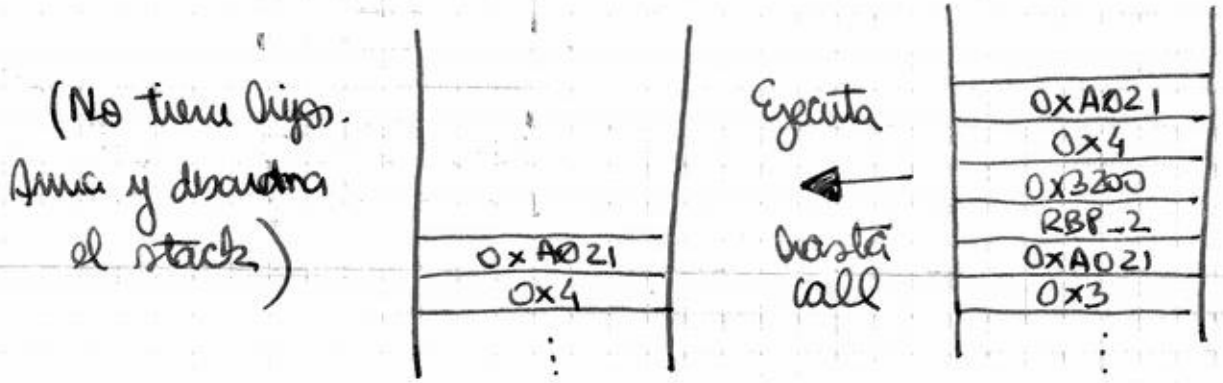
TRES: DD 0x3, 0x3, 0x3, 0x3

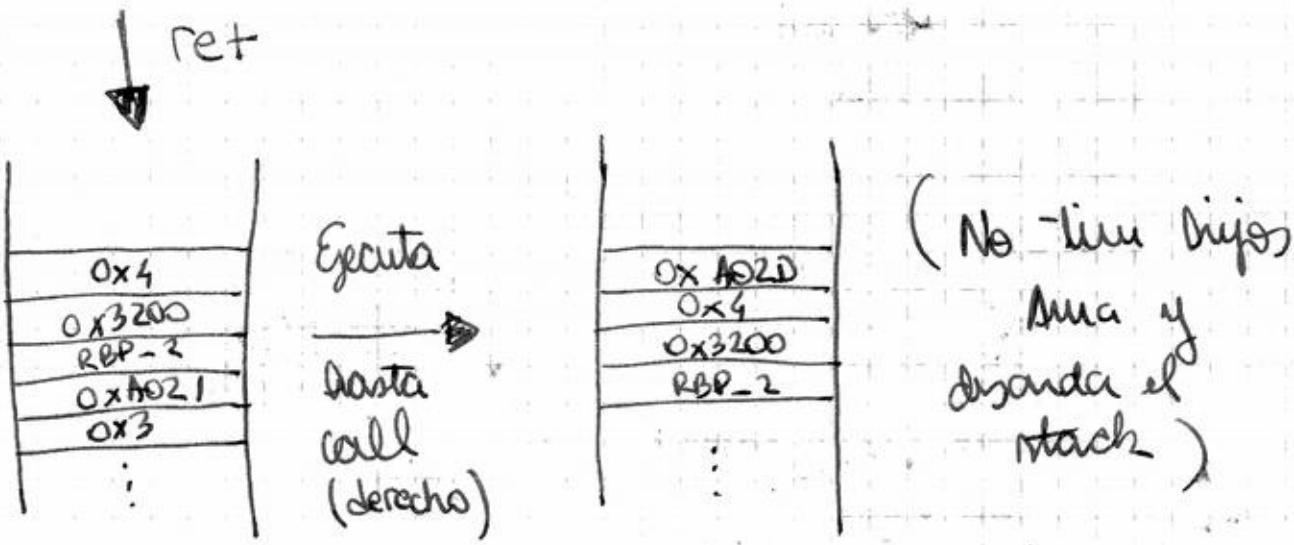
11/05/17

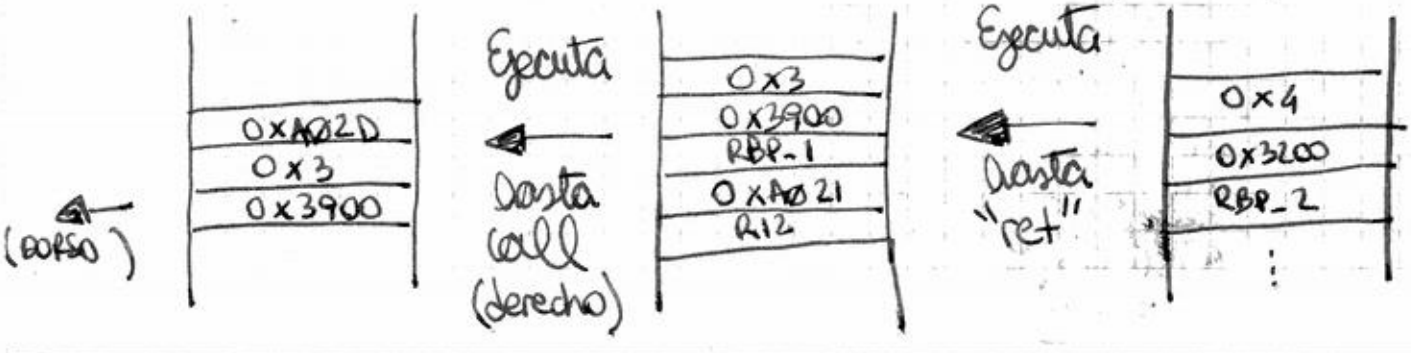
(3)

a)

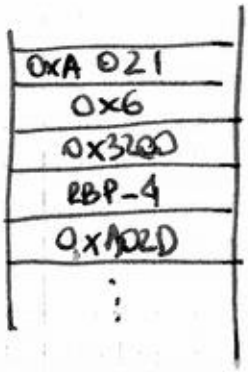

RET
Ejecuta
hasta call
0xA021
R12
RBX
RBP
RET
Ej.
hasta call
0xA021
0x3
0x3900
RBP_1
0xA021
R12
Ejecuta hasta call

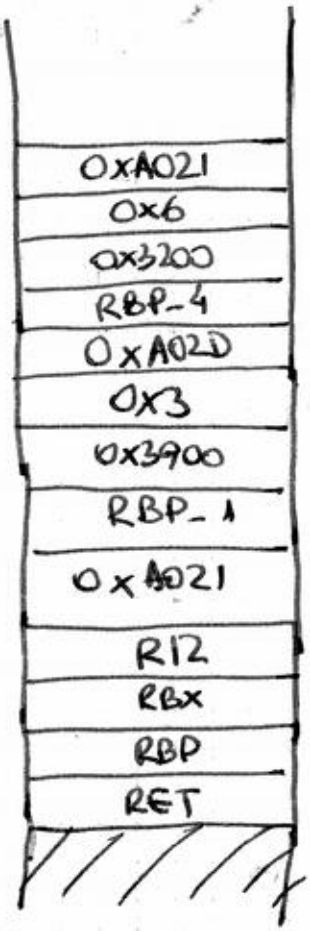

(No tiene hijos. Anula y desarma el stack)
0xA021
0x4
Ejecuta
hasta call
0xA021
0x4
0x3200
RBP_2
0xA021
0x3


ret
0x4
0x3200
RBP_2
0xA021
0x3
Ejecuta
hasta call (derecho)
0xA02D
0x4
0x3200
RBP_2
(No tiene hijos. Anula y desarma el stack)
ret


(FONDO)
0xA02D
0x3
0x3900
Ejecuta
hasta call (derecho)
0x3
0x3900
RBP_1
0xA021
R12
Ejecuta
hasta "ret"
0x4
0x3200
RBP_2

Ejecuta → hasta call (izquierdo)





Aquí, con este stack, comienza a ejecutar la llamada a "suma" con el puntero en rdi.

b)

```
MOV R9, RDi
XOR RAX, RAX
MOV R8, 0
.ciclo:
    CMP R8, 4
    JE .FIN
    CMP [R9], 0xA021
    JE .UNO
    CMP [R9], 0xA02D
    JE .UNO
    JMP .NINGUNO
.UNO:
    INC RAX
    XOR R8, R8
    ADD R9, 8
    JMP ciclo
.NINGUNO
    INC R8
    ADD R9, 8
    JMP .ciclo
.FIN
```