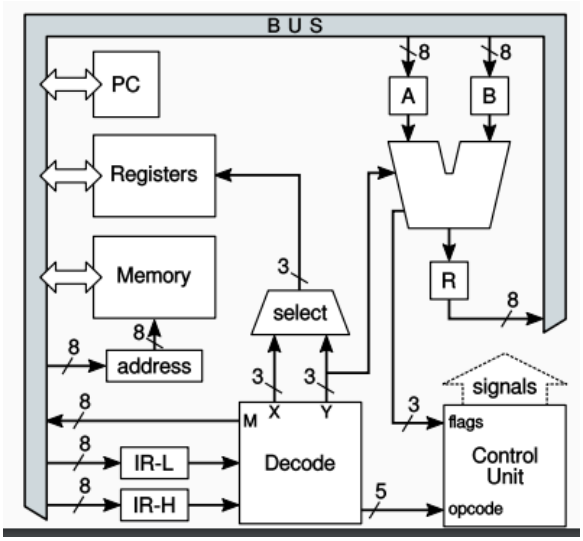
Lenguaje de Maquina

INTRODUCCION: Ciclo de Vida de un Programa

- **Programación:** Escribir un algoritmo en lenguaje ensamblador (o assembly). Esto es el código fuente.
- **Ensamblado:** Un programa llamado Ensamblador toma el código fuente y lo traduce a un código máquina.
  - Resuelve las directivas dirigidas al Ensamblador.
  - Calcula el tamaño de cada instrucciones, y resuelve las etiquetas.
  - Traduce las instrucciones a 0s y 1s.
- **Carga:** Se le indica al Ensamblador una dirección inicial, y éste copia el código máquina en la desde esa posición en adelante. Luego, carga esa misma dirección en el PC (Program Counter).
- **Ejecución:** El CPU da inicio a su ciclo de ejecución comenzando por la posición indicada en el PC.

ORGA-SMALL

OrgaSmall es un procesador diseñado e implementado sobre la herramienta Logisim. Este cuenta con las siguientes características:



- Arquitectura von Neumann, memoria de datose instrucciones compartida.
- 8 registros de propósito general, R0 a R7.
- 1 registro de propósito específico PC.
- Tamaño de palabra de 8 bits e instrucciones de 16 bits.
- Memoria de 256 palabras de 8 bits.
- Bus de 8 bits.
- Diseño microprogramado

- **PC (Program Counter):**

El **Program Counter** es un registro que almacena la dirección de la próxima instrucción a ejecutar. Se incrementa después de cada instrucción para señalar la siguiente. Su valor se envía a la memoria a través del bus para recuperar la próxima instrucción.

- **Decode**:

  Esta unidad se encarga de **decodificar** las instrucciones. Cuando se recupera una instrucción de la memoria, el **decodificador** la traduce en señales de control que se envían a distintas partes del procesador. Esto implica identificar la operación a realizar y los registros o datos involucrados.

- **Control Unit** (Unidad de Control):

  La **Unidad de Control** genera señales de control basadas en la instrucción decodificada. Estas señales activan los componentes adecuados del procesador (como registros, ALU, memoria) para ejecutar la instrucción actual. También gestiona los **flags** (banderas) que indican condiciones especiales, como desbordamientos o resultados nulos en las operaciones.

INSTRUCCIONES

Las instrucciones están codificadas en 16 bits. Los primeros 5 bits son el opcode de la instrucción, el resto de los bits indican los parámetros. Existen 4 posibles codificaciones de parámetros.

Instrucción	CodOp	Formato	Acción
ADD Rx, Ry	00001	A	Rx ← Rx + Ry
SUB Rx, Ry	00011	A	Rx ← Rx - Ry
AND Rx, Ry	00100	A	Rx ← Rx and Ry
...	...	.	...
MOV Rx, Ry	01000	A	Rx ← Ry
STR [M], Rx	10000	D	Mem[M] ← Rx
...	...	.	...
...	...	.	...
JN M	10111	C	Si flag_N=1 entonces PC ← M
...	...	.	...

Formato	Codificación
A	00000 XXX YYY-----
B	00000 XXX --------
C	00000 --- MMMMMMMM
D	00000 XXX MMMMMMMM

XXX codifica el número del registro X (Rx), vale entre 0 y 7

YYY codifica el número del registro Y (Ry) o un inmediato, vale entre 0 y 7

MMMMMMMM Dirección de memoria o valor inmediato, número de 8 bits

EJEMPLO DE ENSAMBLADO

```
main: LOAD R1,[et1]
            LOAD R2,[et2]
            ADD R1,R2
```

DIRECCION	***OCUPA***
0x00	2 palabras
0x02	2 palabras
0x04	2 palabras

```
            STR [et1],R1
            JMP main
et1: DB 0x07
et2: DB 0x04
```

0x06	2 palabras
0x08	2 palabras
0x0A	1 palabra
0x0B	1 palabra

Tenemos que:

- Calcular los valores de las etiquetas
- Ver cuántas palabras necesita cada instrucción

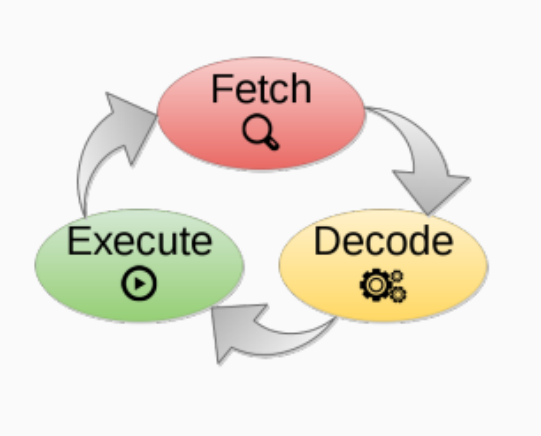
Traducidas las etiquetas lo codificamos y cargamos en memoria:

00	01	02	03	04	05	06	07	08	09	10	11
0x88	0x0A	0x89	0x0B	0x08	0x20	0x80	0x0A	0xA0	0x00	0x07	0x04

Y también cargamos la dirección inicial en el **PC**:

PC	0000

CICLO DE INSTRUCCION



- **Fetch:** La **UC (Unidad de Control)** obtiene una instrucción de la posición de a la que apunta el PC y lo incrementa *(Si es necesario: busca más palabras de la instrucción usando el PC e* incrementándolo cada vez.)
- **Decode** La UC decodifica la instrucción.
- **Execute** La UC ejecuta la instrucción

Ahora si: Lenguaje de Maquina

Arquitectura de Software o Instruction Set Architecture (ISA)

- Contiene todos los aspectos de diseño visibles para un desarrollador de software
- También llamada Arquitectura

Arquitectura de Hardware o Microarquitectura

- Refiere a una implementación específica de la ISA
- Cantidad de núcleos, frecuencia, instrucciones, etc.

- Las distintas arquitecturas de hardware para una determinada ISA se llaman familia

La arquitectura cuenta de 4 partes

- **Set de Instrucciones:** conjunto de instrucciones disponibles en el procesador y las reglas para utilizarlas
- **Organización de registros:** cantidad, tamaño y reglas para su uso
- **Organización de la memoria y direccionamiento**
- **Modos de operación:** modos de operación del procesador (modo user y modo system)

El set de instrucciones define

- **La cantidad de instrucciones disponibles**
- **Tipo de las instrucciones** (RISC / CISC)
- **Formatos:** reglas de uso
- **Ancho del datapath**
  • ancho del bus de datos
  • tamaño en bits de los registros
  • capacidad de memoria (memoria direccionable)

RISC: Utiliza un conjunto reducido de instrucciones simples que se ejecutan en un solo ciclo de reloj, lo que permite un diseño eficiente y optimiza el uso del pipelining.

- **Microprocesador (µP, uP)**
  • Propósito general
  • Rendimiento por velocidad, paralelismo, etc.
  • Sin periféricos incluidos (on-chip)
  • La potencia “no” es un problema

CISC: Emplea un conjunto más amplio y complejo de instrucciones que pueden realizar tareas sofisticadas en un solo paso, pero requieren múltiples ciclos de reloj, aumentando la complejidad del hardware.

- **Microcontrolador (µC, uC)**
  • Propósitos específicos
  • Rendimiento “moderado”
  • *(posiblemente)* Una gran cantidad de periféricos incluidos
  • Eficiente en términos de potencia

- **Registros de propósito general:**
  - **EAX (acumulador):** Se utiliza comúnmente para operaciones aritméticas y de lógica.
  - **EBX (base):** A menudo se usa como un puntero base para acceder a datos en la memoria.
  - **ECX (contador):** Se usa generalmente como un contador en bucles y en operaciones de repetición.
  - **EDX (datos):** Suele ser usado para operaciones matemáticas y para expandir la capacidad del registro EAX en operaciones de multiplicación y división.

- **Registros índice:**
  - **ESI (source index) y EDI (destination index):** Se utilizan comúnmente para operaciones de manejo de memoria, como la copia de datos entre direcciones de memoria.
- **Registros de puntero:**
  - **ESP (stack pointer):** Es el puntero de pila, que apunta al tope de la pila y se usa para gestionar las llamadas a funciones y el almacenamiento temporal de datos.
  - **EBP (base pointer):** Sirve como puntero base para acceder a variables locales y parámetros dentro de una función.
- **Registros de 64 bits (nombres completos):**
  - `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `rsp`, `rbp`, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, `r15`
  - Cada uno de estos registros lee o almacena 8 bytes (64 bits).
- **Registros de 32 bits:**
  - `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `esp`, `ebp`, `r8d`, `r9d`, `r10d`, `r11d`, `r12d`, `r13d`, `r14d`, `r15d`
  - Cada uno de estos registros lee o almacena 4 bytes (32 bits).

- **Normal [R] Mem[Reg[R]]**
  • El registro R especifica la dirección de memoria
  • ¡R funciona como un puntero!

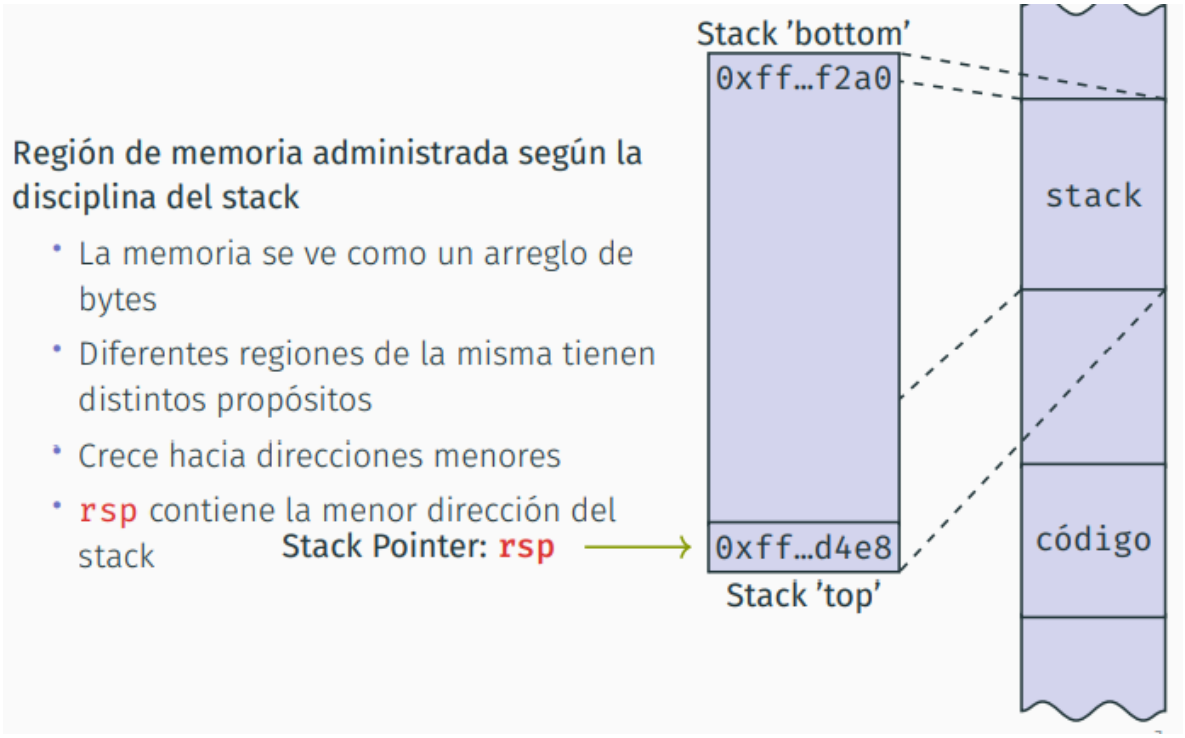
$$mov[rcx], rax$$

- **Corrimiento [R + D] Mem[Reg[R]+D]**
  • El registro R especifica el comienzo de una región de memoria
  • El corrimiento D especifica el offset

$$mov[rbp + 8], rdx$$

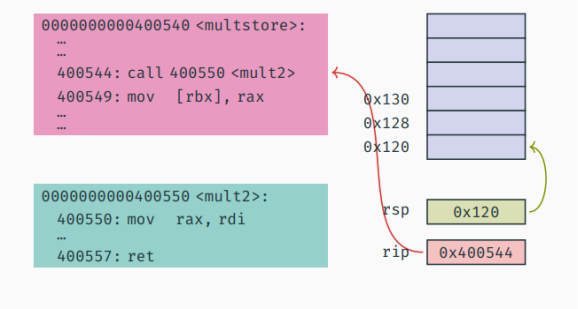
Procedimientos (ABI)

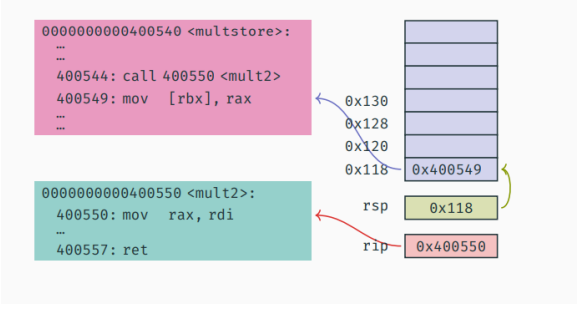
Pila (Stack) x86-64

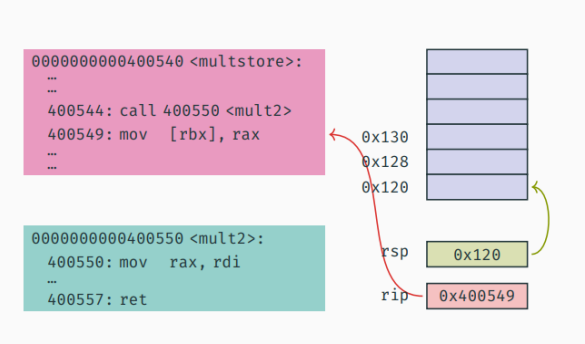


Transferencia de control

- Usa el stack para dar soporte a las llamadas y retornos de procedimientos
- **Llamada a procedimientos/funciones: call etiqueta**
  - Hacer un `push` de la dirección de retorno
  - "Saltar" a la etiqueta
- **Dirección de retorno**
  - Dirección de la instrucción siguiente (inmediata) a la instrucción call
- **Retorno de procedimientos/funciones: ret**
  - Hacer un `pop` de la dirección de retorno
  - "Saltar" a dicha dirección



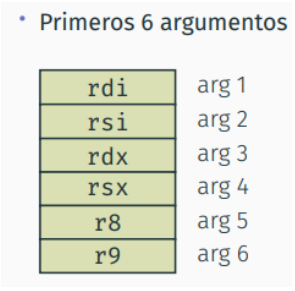


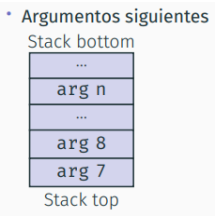


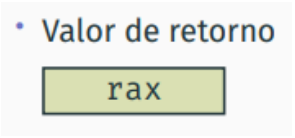
Pasaje de Datos

Los argumentos se pasan por registros o usando el stack

- en x86 (32 bits) únicamente usando la pila

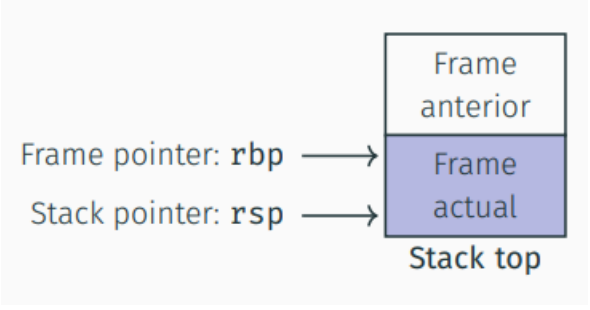






Stack frames

- **Contiene:**
  - Información de retorno
  - Almacenamiento local
  - Espacio temporal



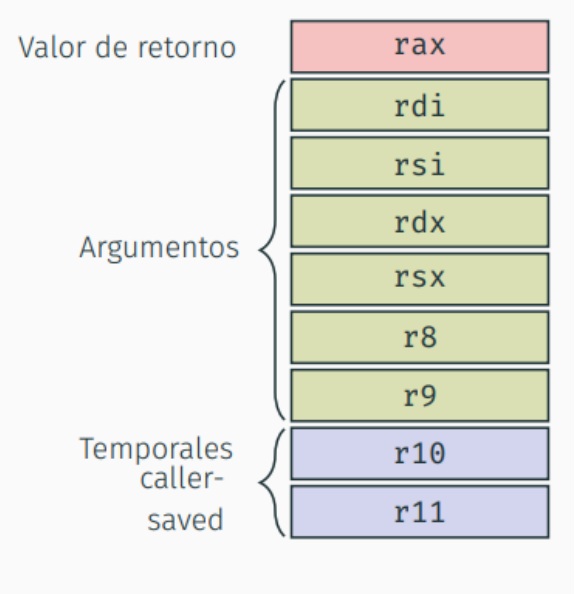
- **Administración:**
  - **El espacio se reserva al entrar**
    - Requiere código de inicialización
    - Incluye el push de la instrucción call
  - **El espacio se retorna al salir**
    - Requiere código de finalización
    - Incluye el pop de la instrucción ret

Las llamadas de `rbp` apunta a la anterior `rbp`
Y la ultima posicion del frame anterior habra un call que mete en la pila
`rip`

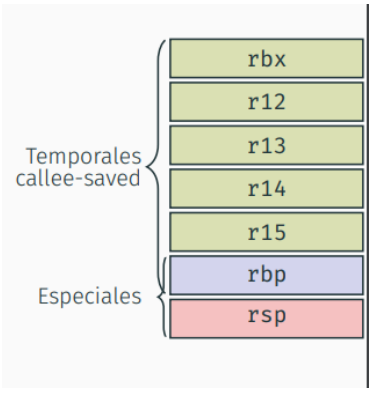
Registros volátiles y no volátiles

Volátiles

- **rax**
  - valor de retorno
  - caller-saved
  - un procedimiento puede modificarlo
- **rdi, ..., r9**
  - argumentos
  - caller-saved
  - un procedimiento puede modificarlos
- **r10, r11**
  - caller-saved
  - un procedimiento puede modificarlos



No-Volátiles

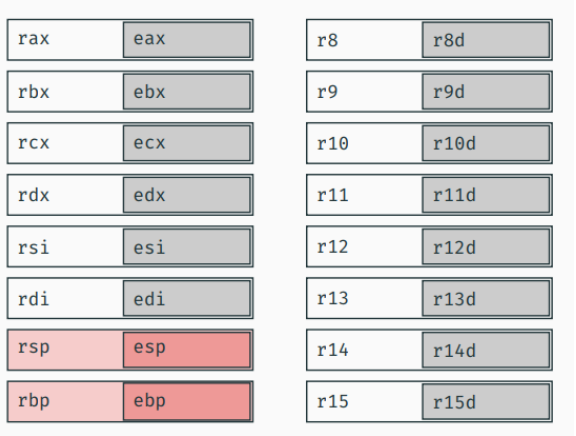


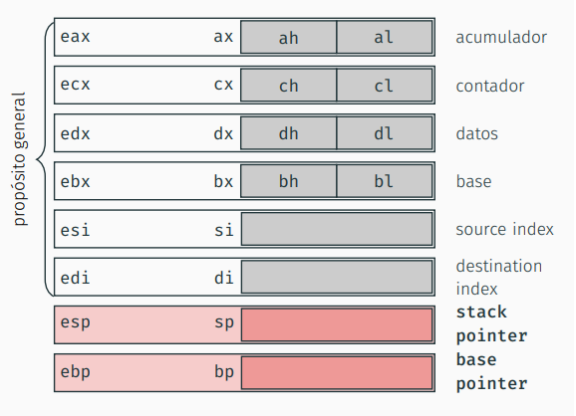
- **rbx, r12, r13, r14, r15**
  - callee-saved
  - la función callee debe guardarlos y restaurarlos
- **rbp**
  - callee-saved
  - la función callee debe guardarlos y restaurarlos
  - opcionalmente puede usarse como frame pointer
- **rsp**
  - callee-saved especial
  - se restaura a su valor original al retornar del procedimiento

ASM

Instrucciones:

- **MOV**: Copia datos de una ubicación a otra. Ejemplo: `MOV AX, 5` copia el valor 5 al registro `AX`.
- **ADD, SUB**: Suma y resta valores. Ejemplo: `ADD AX, BX` suma `AX` y `BX`.
- **MUL, DIV**: Multiplicación y división.
- **CMP**: Compara dos valores y afecta las banderas para usar en instrucciones condicionales.
- **JMP**: Salto incondicional a otra línea de código.
- **JE, JNE, JG, JL**: Saltos condicionales (según las banderas establecidas por `CMP`).
- `malloc` **Recibe**: El tamaño en bytes que se desea asignar ,**Devuelve**: Un puntero a la memoria asignada
- `free` **Recibe**: Un puntero a la memoria previamente asignada. No devuelve nada





Para que tengan a mano en la **primera parte de la materia (64 bits)**:

No volátiles:	RBX, RBP, R12, R13, R14 y R15
Valor de retorno:	RAX enteros/punteros,XMM0 flotantes
Entero,puntero:	RDI, RSI, RDX, RCX, R8, R9(izq. a der.)
Flotantes:	XMM0, XMM1, ..., XMM7(izq. a der.)
¿No hay registros?	PUSH a la pila(der. a izq.)
Inv. de pila:	Todo PUSH/SUB debe tener su POP/ADD
Llamada func. C:	pila alineada a 16 bytes (SUB RSP, X)

Para que tengan a mano en la **segunda parte de la materia (32 bits)**:

No volátiles:	EBX, EBP, ESI y EDI
Valor de retorno:	EAX
Parámetros:	PUSH a la pila(der. a izq.)
Inv. de pila:	Todo PUSH/SUB debe tener su POP/ADD
Llamada func. C:	pila alineada a 16 bytes (SUB RSP, X)

Tamaño de datos en Bytes

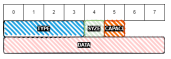
- **int8_t / uint8_t:** 1 byte (8 bits)
- **int16_t / uint16_t:** 2 bytes (16 bits)
- **int32_t / uint32_t:** 4 bytes (32 bits)
- **int64_t / uint64_t:** 8 bytes (64 bits)
- **char:** 1 byte (8 bits) – Usado para caracteres individuales
- **float:** 4 bytes (32 bits) – Precisión simple
- **double:** 8 bytes (64 bits) – Precisión doble
- **Punteros:** 8 bytes (64 bits)

¿Como manejar structs?

Para manejar structs en ASM, debemos considerar el tamaño de los datos en bytes. Esto es crucial no solo para determinar el espacio de memoria necesario para almacenar el struct, sino también para acceder correctamente a cada uno de sus campos desde el código `ASM`.

```
typedef struct s_array {
    type_t  type; // 4 bytes ; off = 0
    uint8_t size; // 1 byte  ; off = 4
    uint8_t capacity; // 1 byte ; off = 5
    void** data; // 8 bytes ;off = 8, no entra en 2byte
} array_t;
```

Observemos que los bytes se acomodan en los offsets según el tamaño de cada tipo de dato.



(Aclaración: type es una estructura de tipo enum, por lo tanto ocupa 4 bytes)

Códigos de ejemplo para recordar

▼ **ArrayRemove**

```
arrayRemove:
    push rbp
    mov rbp, rsp
    push r12
    push r13
    push rbx
    push r14
    push r15
    sub rsp, 8
```

```
    mov     r12, rdi ; r12 es array
    mov     r13 , rsi ; r13 es i
    cmp     r13b ,  [r12+ARRAY_OFF_SIZE]
    jge .fuera_rango

    mov     r14 , [r12 + ARRAY_OFF_DATA] ; r14 es void**data
    mov     rbx , [r14 + r13*8] ; rbx es el data a eliminar

    ;clono el data
    mov     rdi , [r12 +ARRAY_OFF_TYPE]
    call getCloneFunction
    mov     rdi , rbx
    call rax
    ;rax es el puntero clonado
    mov r15, rax

    mov     rdi , [r12 +ARRAY_OFF_TYPE]
    call getDeleteFunction
    ;rax es la funcion
    mov     rdi , rbx
    call rax
    ;rax en teoria no es nada

    inc r13 ;sera mi contador [6, 3, Nada, 8] => elijo 8
    cmp r13b , [r12 + ARRAY_OFF_SIZE]
    jg .fin ;caso de que me vaya de rango

    .mover_elementos:
        mov     rbx , [r14 + r13*8] ;el prox elem a mover
        mov     [r14 + (r13-1)*8] , rbx ;lo pongo en su posicion ar
        inc     r13 ; busco el proximo
        cmp     r13b , [r12 + ARRAY_OFF_SIZE]
        jg .fin_mover
        jmp .mover_elementos

    .fin_mover
        dec  byte[r12 + ARRAY_OFF_SIZE]
        mov rax , r15
        jmp .fin ; voy a fin

    .fuera_rango:
        mov rax, 0

    .fin:
```

```
        add rsp,8
        pop r15
        pop r14
        pop rbx
        pop r13
        pop r12
        pop rbp
ret
```

▼ **ArrayAddLast**

```
arrayAddLast:
    push rbp
    mov rbp, rsp
    push r12
    push r13
    push rbx
    push r15
    push r14
    sub rsp , 8

    mov r12 , rdi ; r12 es el array
    mov r13 , rsi ; r13 es data

    mov     bl, byte [r12 + ARRAY_OFF_SIZE] ; bl es size
    mov     cl, byte [r12 + ARRAY_OFF_CAPACITY] ; cl es capacidad
    cmp     bl, cl
    je      .fin

    mov     rdi , [r12+ARRAY_OFF_TYPE]
    call getCloneFunction ;consigo mi funcion clonar con el tipo de
    mov     rdi , r13
    call    rax

    mov     r14, rax ; r14 es el nuevo dato
    xor     rbx, rbx ;pongo en 0 a rbx
    mov     bl, [r12 + ARRAY_OFF_SIZE] ; rbx es size
    mov     r15 , [r12 + ARRAY_OFF_DATA] ; r15 sera data del array
    mov     [r15 + rbx*8], r14
    inc     byte [r12 + ARRAY_OFF_SIZE]

    .fin:
        add rsp, 8
```

```
        pop r14
        pop r15
        pop rbx
        pop r13
        pop r12
        pop rbp
ret
```

▼ **StrLen**

```
strLen:
    push rbp
    mov rbp, rsp
    push r12
    sub rsp , 8
    mov r12, 0
    .while:
        mov dl, [rdi + r12]
        inc r12
        cmp dl, 0
        jne .while
    dec r12
    .fin:
    mov rax, r12
    add rsp , 8
    pop r12
    pop rbp
ret
```

▼ **StrCmp**

```
strCmp:
    push    rbp
    mov     rbp, rsp
    push    r12
    push    r13
    push    r14
    push    r15

    mov     r12, rdi
    mov     r14, rsi

    call    strLen
```

```
    mov     r13, rax
    mov     rdi, r14
    call    strLen
    mov     r15, rax

    cmp     r13, r15
    jg      .b_mas_grande
    jl      .a_mas_grande

    mov     rdi, r12
    mov     rsi, r14
    xor     r12, r12

    .while:
        mov     al, [rdi + r12]
        mov     bl, [rsi + r12]
        cmp     al, bl
        jne     .a_mas_grande
        cmp     al, 0
        je      .son_iguales
        inc     r12
        jmp     .while

    .son_iguales:
        xor     eax, eax
        jmp     .fin

    .a_mas_grande:
        mov     eax, 1
        jmp     .fin
    .b_mas_grande:
        mov     eax, -1
        jmp     .fin

    .fin:
        pop     r13
        pop     r14
        pop     r15
        pop     r12
        pop     rbp
ret
```