Repaso primera parte

Organización del Computador II

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

1er Cuatrimestre 2018
03-04-18

Repaso Primera Parte Enteros

- **Representación y almacenamiento en memoria.**
  - little-endian, big-endian.
- **Direccionamiento y dirección efectiva.**
  - a byte, [base + indice + desplazamiento], mov/lea.
- **Instrucciones aritméticas y lógicas.**
  - ADD, SUB, INC, DEC, MUL, IMUL, DIV, IDIV, NEG, AND, OR, NOT, XOR, SHL, SHR, SAR, SAL.

Repaso Primera Parte Enteros

- **Secciones del código.**
  - data, rodata, bss, text.
- **Stack/Pila y Convención C.**
  - en 32 y 64 bits: pasaje de parámetros, preservación de registros y alineación.

Repaso Primera Parte Enteros

- **Punteros.**
- **Estructuras de datos.**
  - structs e indización.
- **Memoria.**
  - estática: global y local.
  - dinámica: malloc/free.
- **Estructuras Dinámicas y Recursivas.**
  - estructuras dentro de estructuras y autodefinidas.
  - listas, arboles, etc.

El Oráculo de Orga2

Egeo consultando el Oráculo de Orga2

El Oráculo de Orga2 dice...

- *al menos 1 de ustedes va a cometer el siguiente error:*

El Oráculo de Orga2 dice...

- *al menos 1 de ustedes va a cometer el siguiente error:*

```
mov [rax], [rbx]
```

El Oráculo de Orga2 dice...

- *al menos 1 de ustedes va a cometer el siguiente error:*

```
mov [rax], [rbx]
```

MAL: direccionamiento memoria a memoria

El Oráculo de Orga2 dice...

- *incluso, bastantes más que 1, van a hacer algo como:*

```
%define miVariable [rbp-8]
mov miVariable, [...]
```

El Oráculo de Orga2 dice...

- *incluso, bastantes más que 1, van a hacer algo como:*

```
%define miVariable [rbp-8]
mov miVariable, [...]          ; MAL: direccionamiento memoria a memoria
```

El Oráculo de Orga2 dice...

- *incluso, bastantes más que 1, van a hacer algo como:*

```
%define miVariable [rbp-8]
mov miVariable, [...]          ; MAL: direccionamiento memoria a memoria
```

- *entonces prestamos atención cuando escribimos los [ ]:*

```
mov RCX, [...]
```

El Oráculo de Orga2 dice...

- *incluso, bastantes más que 1, van a hacer algo como:*

```
%define miVariable [rbp-8]
mov miVariable, [...]          ; MAL: direccionamiento memoria a memoria
```

- *entonces prestamos atención cuando escribimos los [ ]:*

```
mov RCX, [...]    ; BIEN: mover 8 bytes
mov [...], ECX
```

El Oráculo de Orga2 dice...

- *incluso, bastantes más que 1, van a hacer algo como:*

```
%define miVariable [rbp-8]
mov miVariable, [...]          ; MAL: direccionamiento memoria a memoria
```

- *entonces prestamos atención cuando escribimos los [ ]:*

```
mov RCX, [...]    ; BIEN: mover 8 bytes
mov [...], ECX    ; BIEN: mover 4 bytes
mov CX, [...]
```

El Oráculo de Orga2 dice...

- *incluso, bastantes más que 1, van a hacer algo como:*

```
%define miVariable [rbp-8]
mov miVariable, [...]          ; MAL: direccionamiento memoria a memoria
```

- *entonces prestamos atención cuando escribimos los [ ]:*

```
mov RCX, [...]    ; BIEN: mover 8 bytes
mov [...], ECX    ; BIEN: mover 4 bytes
mov CX, [...]     ; BIEN: mover 2 bytes
mov CL, [...]
```

El Oráculo de Orga2 dice...

- *incluso, bastantes más que 1, van a hacer algo como:*

```
%define miVariable [rbp-8]
mov miVariable, [...]          ; MAL: direccionamiento memoria a memoria
```

- *entonces prestamos atención cuando escribimos los [ ]:*

```
mov RCX, [...]    ; BIEN: mover 8 bytes
mov [...], ECX    ; BIEN: mover 4 bytes
mov CX, [...]     ; BIEN: mover 2 bytes
mov CL, [...]     ; BIEN: mover 1 bytes
```

El Oráculo de Orga2 dice...

- *incluso, bastantes más que 1, van a hacer algo como:*

```
%define miVariable [rbp-8]
mov miVariable, [...]          ; MAL: direccionamiento memoria a memoria
```

- *entonces prestamos atención cuando escribimos los [ ]:*

```
mov RCX, [...]    ; BIEN: mover 8 bytes
mov [...], ECX    ; BIEN: mover 4 bytes
mov CX, [...]     ; BIEN: mover 2 bytes
mov CL, [...]     ; BIEN: mover 1 bytes
```

- *y esto?*

```
mov [...], 0xFF
```

El Oráculo de Orga2 dice...

- *incluso, bastantes más que 1, van a hacer algo como:*

```
%define miVariable [rbp-8]
mov miVariable, [...]          ; MAL: direccionamiento memoria a memoria
```

- *entonces prestamos atención cuando escribimos los [ ]:*

```
mov RCX, [...]    ; BIEN: mover 8 bytes
mov [...], ECX    ; BIEN: mover 4 bytes
mov CX, [...]     ; BIEN: mover 2 bytes
mov CL, [...]     ; BIEN: mover 1 bytes
```

- *y esto?*

```
mov [...], 0xFF           ; MAL: operation size not specified!
mov dword [...], 0xFF
```

El Oráculo de Orga2 dice...

- *incluso, bastantes más que 1, van a hacer algo como:*

```
%define miVariable [rbp-8]
mov miVariable, [...]          ; MAL: direccionamiento memoria a memoria
```

- *entonces prestamos atención cuando escribimos los [ ]:*

```
mov RCX, [...]     ; BIEN: mover 8 bytes
mov [...], ECX     ; BIEN: mover 4 bytes
mov CX, [...]      ; BIEN: mover 2 bytes
mov CL, [...]      ; BIEN: mover 1 bytes
```

- *y esto?*

```
mov [...], 0xFF            ; MAL: operation size not specified!
mov dword [...], 0xFF      ; BIEN: mover 0xFF como 4 bytes
```

El Oráculo de Orga2 dice...

- **Consejo 1:** cada vez que escribimos [ ], nos fijamos que no estamos poniendo [ ] dos veces.
- **Consejo 2:** nos fijamos si la instrucción puede deducir el tamaño de memoria a leer/escribir, si no lo puede hacer, se lo indicamos (byte, word, dword, qword).

El Oráculo de Orga2 dice...

- *cuando se pida hacer algo como* `RAX = RAX/4`*, varios van a hacer:*

```
div 4
```

El Oráculo de Orga2 dice...

- *cuando se pida hacer algo como* `RAX = RAX/4`*, varios van a hacer:*

```
div 4    ; MAL: div toma RDX:RAX, qué tiene RDX?
```

El Oráculo de Orga2 dice...

- *cuando se pida hacer algo como `RAX = RAX/4`, varios van a hacer:*

```
div 4    ; MAL: div toma RDX:RAX, qué tiene RDX?
```

- *entonces hacemos:*

```
xor rdx, rdx
div 4          ; BIEN
```

El Oráculo de Orga2 dice...

- *cuando se pida hacer algo como `RAX = RAX/4`, varios van a hacer:*

```
div 4    ; MAL: div toma RDX:RAX, qué tiene RDX?
```

- *entonces hacemos:*

```
xor rdx, rdx
div 4          ; BIEN
```

- *¿pero tiene sentido usar `div` para dividir por un 2^n?*

El Oráculo de Orga2 dice...

- *cuando se pida hacer algo como `RAX = RAX/4`, varios van a hacer:*

```
div 4    ; MAL: div toma RDX:RAX, qué tiene RDX?
```

- *entonces hacemos:*

```
xor rdx, rdx
div 4          ; BIEN
```

- *¿pero tiene sentido usar `div` para dividir por un 2^n?*

```
shr rax, 2
```

El Oráculo de Orga2 dice...

- *cuando se pida hacer algo como `RAX = RAX/4`, varios van a hacer:*

```
div 4    ; MAL: div toma RDX:RAX, qué tiene RDX?
```

- *entonces hacemos:*

```
xor rdx, rdx
div 4            ; BIEN
```

- *¿pero tiene sentido usar `div` para dividir por un 2^n?*

```
shr rax, 2    ; BIEN: más elegante, rápido y me evito posibles errores
```

El Oráculo de Orga2 dice...

- *cuando se pida hacer algo como `RAX = RAX/4`, varios van a hacer:*

```
div 4    ; MAL: div toma RDX:RAX, qué tiene RDX?
```

- *entonces hacemos:*

```
xor rdx, rdx
div 4            ; BIEN
```

- *¿pero tiene sentido usar `div` para dividir por un 2^n?*

```
shr rax, 2    ; BIEN: más elegante, rápido y me evito posibles errores
```

- **Consejo 3:** para multiplicar/dividir por 2^n usamos *shifts*, en lugar de *mul*/*div*.

El Oráculo de Orga2 dice...

- *cuando se pida hacer algo como* `RAX = RAX/4 ;con RAX int`, *varios van a hacer:*

```
shr rax, 2
```

El Oráculo de Orga2 dice...

- *cuando se pida hacer algo como `RAX = RAX/4 ;con RAX int`, varios van a hacer:*

```
shr rax, 2    ; MAL: y el bit de signo? rompí la representación
```

El Oráculo de Orga2 dice...

- *cuando se pida hacer algo como `RAX = RAX/4 ;con RAX int`, varios van a hacer:*

```
shr rax, 2    ; MAL: y el bit de signo? rompí la representación
```

- *entonces hacemos:*

```
sar rbx, 2    ; BIEN
```

El Oráculo de Orga2 dice...

- *cuando se pida hacer algo como `RAX = RAX/4 ;con RAX int`, varios van a hacer:*

```
shr rax, 2    ; MAL: y el bit de signo? rompí la representación
```

- *entonces hacemos:*

```
sar rbx, 2    ; BIEN
```

- **Consejo 4:** para dividir enteros con signo por 2^n usamos `SAR` (aritmético), y para multiplicar `SAL`, en lugar de *imul*/*idiv*.

El Oráculo de Orga2 dice...

- *cuando tengan que direccionar a memoria muchos van a hacer:*

```
mov rax, [rdi*4 - esi*16 * rcx]
```

El Oráculo de Orga2 dice...

- *cuando tengan que direccionar a memoria muchos van a hacer:*

```
mov rax, [rdi*4 - esi*16 * rcx]    ; MAL: invalid effective address /
                                   ; impossible combination of address sizes
```

El Oráculo de Orga2 dice...

- *cuando tengan que direccionar a memoria muchos van a hacer:*

```
mov rax, [rdi*4 - esi*16 * rcx]    ; MAL: invalid effective address /
                                   ; impossible combination of address sizes
```

- *¿entonces?*

```
mov RAX, [Base + Indice*i +/- desplazamiento]
; BIEN: hay un formato:
    ; Base = algún registro
    ; Indice = algún registro (igual tamaño que Base)
    ; i = 1, 2, 4 u 8
    ; desplazamiento = inmediato de 32 bits
```

El Oráculo de Orga2 dice...

- *cuando tengan que direccionar a memoria muchos van a hacer:*

```
mov rax, [rdi*4 - esi*16 * rcx]    ; MAL: invalid effective address /
                                   ; impossible combination of address sizes
```

- *¿entonces?*

```
mov RAX, [Base + Indice*i +/- desplazamiento]
; BIEN: hay un formato:
    ; Base = algún registro
    ; Indice = algún registro (igual tamaño que Base)
    ; i = 1, 2, 4 u 8
    ; desplazamiento = inmediato de 32 bits
```

- **Consejo 5:** para indizar usamos el formato correcto. Si necesitamos hacer algo raro en la indización, entonces calculamos los valores que necesitemos, los ponemos en registros antes de indizar, y luego usamos el formato correcto nuevamente.

El Oráculo de Orga2 dice...

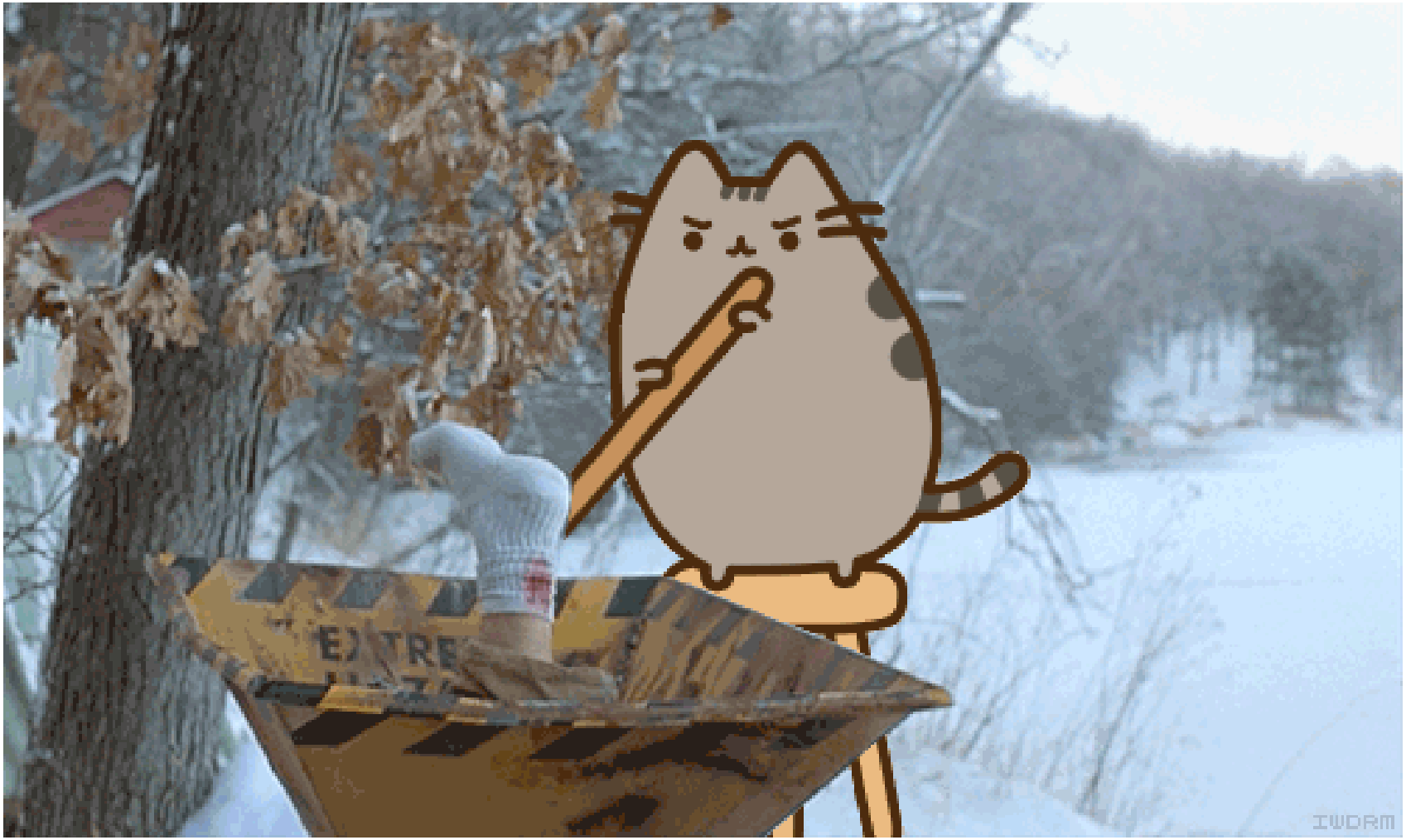
- *cuando armen el stack frame, usen la Convención C o simplemente "pusheen" registros, muchos van a hacer:*

```
f:
        push rbp
        mov rbp, rsp
        push rbx
        push r12
        push r13
        push r14
        push r15

        mov rdi, 8
        call malloc

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


IWORM


IWORM

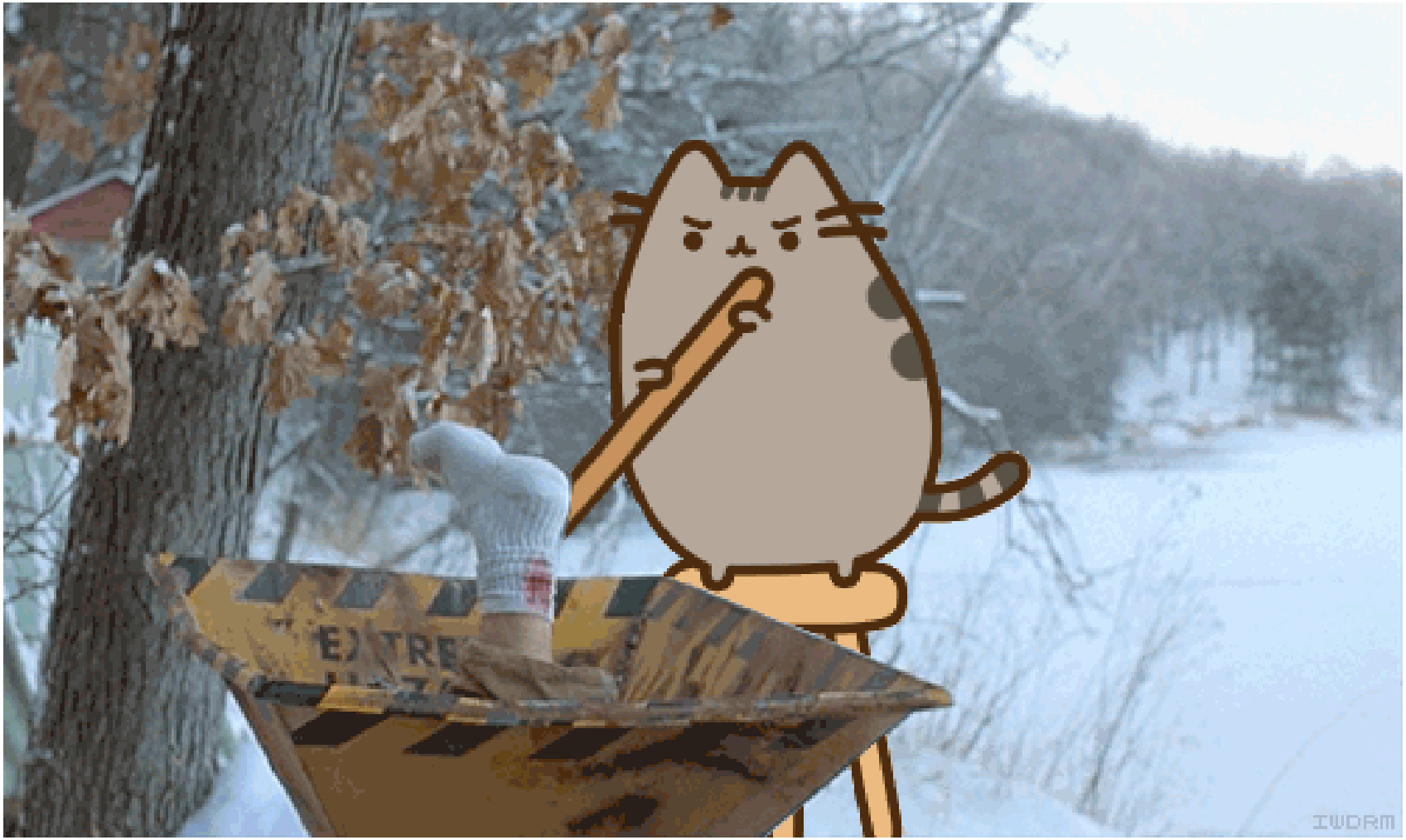

IWORM


IWORM


IWORM


IWORM

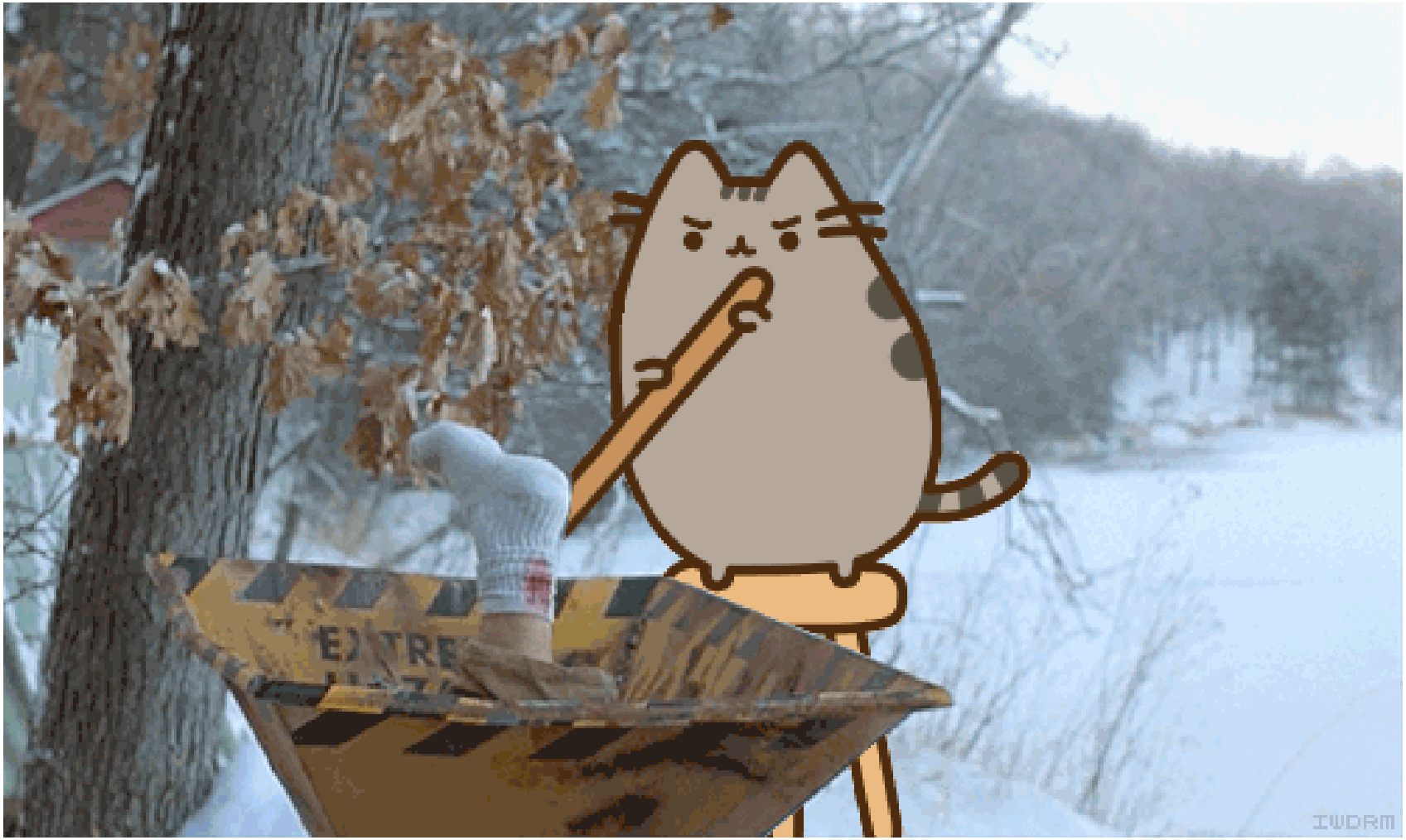

IWORM


IWORM


IWORM


IWORM


IWORM


IWORM

El Oráculo de Orga2 dice...

- *cuando armen el stack frame, usen la Convención C o simplemente "pusheen" registros, muchos van a hacer:*

```
f:
        push rbp
        mov rbp, rsp
        push rbx        ; MAL: lo pusheo de más y no lo uso
        push r12        ; MAL: lo pusheo de más y no lo uso
        push r13        ; MAL: lo pusheo de más y no lo uso
        push r14        ; MAL: lo pusheo de más y no lo uso
        push r15        ; MAL: lo pusheo de más y no lo uso

        mov rdi, 8
        call malloc     ; MAL: pila desalineada

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

El Oráculo de Orga2 dice...

- *¿entonces?*

El Oráculo de Orga2 dice...

- *¿entonces?*

```
f:
                        ; desalineada
    push rbp            ; BIEN: ya queda alineada!
    mov rbp, rsp

    mov rdi, 8
    call malloc         ; BIEN: alineada

    pop rbp
    ret
```

El Oráculo de Orga2 dice...

- *¿y si tengo que resguardar alguno?*

El Oráculo de Orga2 dice...

- *¿y si tengo que resguardar alguno?*

```
f:
                          ; desalineada
    push rbp              ; alineada
    mov rbp, rsp
    push rbx              ; desalineada

    mov rbx, [...]

    sub rsp, 8            ; BIEN: alineo
    mov rdi, 8
    call malloc           ; BIEN: alineada
    add rsp, 8            ; BIEN: deshago

    pop rbx
    pop rbp
    ret
```

El Oráculo de Orga2 dice...

- **Consejo 6:** ojo con estar *pusheando* registros de más/menos, sólo hacemos *push* de lo que necesitamos.

- **Consejo 7:** nunca perdemos el control de la alineación de la pila, los registros que tenemos que resguardar y el espacio reservado para variables. Esto nos ayuda muchísimo a que todo esté como debería antes de hacer un `CALL`.

El Oráculo de Orga2 dice...

- *cuando "alguien" les pida que pasen el siguiente código en C a ASM:*

```
double Pi = 3.14;
long long f( int x , long long y){
    char A = 'a';
    return (x+y);
{
```

El Oráculo de Orga2 dice...

- *cuando "alguien" les pida que pasen el siguiente código en C a ASM:*

```
double Pi = 3.14;
long long f( int x , long long y){
    char A = 'a';
    return (x+y);
{
```

- *muchos van a hacer:*

```
Pi: dq 3.14
A: db 'a'
f:
    mov rax, rdi
    add rax, rsi
    ret
```

El Oráculo de Orga2 dice...

- *cuando "alguien" les pida que pasen el siguiente código en C a ASM:*

```
double Pi = 3.14;
long long f( int x , long long y){
    char A = 'a';
    return (x+y);
{
```

- *muchos van a hacer:*

```
Pi: dq 3.14          ; MAL: va en section .data o .rodata
A: db 'a'            ; MAL: era local
f:
    mov rax, rdi     ; MAL: int x (es 32 no 64)
    add rax, rsi
    ret
```

El Oráculo de Orga2 dice...

- *entonces:*

```
section .data                          ; MEJOR: las globales van en .data
Pi: dq 3.14

section .text
%define OFFSET_A -8                    ; BIEN: defino etiqueta a var local
f:
    push rbp
    mov rbp, rsp
    sub rsp, 8                         ; BIEN: reservo espacio para var local

    mov byte [rbp+OFFSET_A], 'a'       ; BIEN: inicializo A = 'a'
    xor rax, rax                       ; OK: limpio parte superior de RAX
    mov eax, edi                       ; BIEN: muevo a parte inferior de RAX
    add rax, rsi

    add rsp, 8
    pop rbp
    ret
```

El Oráculo de Orga2 dice...

- **Consejo 8:** las variables locales van en la pila. La sección `.data` no es la pila!

- **Consejo 9:** cuando bajo un parámetro a un registro, me detengo a ver qué tamaño tiene. No puedo asumir que el registro de 64 bits está relleno con 0s. Lo bajo a un registro con el tamaño correcto *limpiando* (`XOR`, `MOVZX`) o extendiendo el signo de su parte superior (`MOVSX`, `CBW`, `CWDE`, `CDQE`).

El Oráculo de Orga2 dice...

- *cuando "alguien" les pida que pasen el siguiente código en C a ASM:*

```
extern uInt f2( uLongLong );
uLongLong f( uInt* V1, uInt* V2, uInt* V3, uInt n ){
    int i;
    uLongLong sum = 0;
    for( i=0; i<n; i++)
        sum = V1[i] + V2[i] + V3[i];
    return ( sum + f2(sum/2) );
    }
```

El Oráculo de Orga2 dice...

- *cuando "alguien" les pida que pasen el siguiente código en C a ASM:*

```
extern uInt f2( uLongLong );
uLongLong f( uInt* V1, uInt* V2, uInt* V3, uInt n ){
    int i;
    uLongLong sum = 0;
    for( i=0; i<n; i++)
        sum = V1[i] + V2[i] + V3[i];
    return ( sum + f2(sum/2) );
    }
```

- *muchos van a hacer:*

```
f:  ; rdi(V1) rsi(V2) rdx(V3) ecx(n)
    xor r8, r8
    xor r9, r9
    .ciclo:
        add r9, [rdi + r8*4]
        add r9, [rsi + r8*4]
        add r9, [rdx + r8*4]
        inc r8
        loop .ciclo
    mov rdi, r9
    shr rdi, 1
    sub rsp, 8
    call f2
    add rsp, 8
    add rax, r9
    ret
```

El Oráculo de Orga2 dice...

- *cuando "alguien" les pida que pasen el siguiente código en C a ASM:*

```
extern uInt f2( uLongLong );
uLongLong f( uInt* V1, uInt* V2, uInt* V3, uInt n ){
    int i;
    uLongLong sum = 0;
    for( i=0; i<n; i++)
        sum = V1[i] + V2[i] + V3[i];
    return ( sum + f2(sum/2) );
    }
```

- *muchos van a hacer:*

```
f:  ; rdi(V1) rsi(V2) rdx(V3) ecx(n)    ; BIEN ecx: pero ojo con la parte superior
    xor r8, r8
    xor r9, r9
    .ciclo:
        add r9, [rdi + r8*4]
        add r9, [rsi + r8*4]
        add r9, [rdx + r8*4]
        inc r8
        loop .ciclo                     ; MAL: y la parte alta de RCX?
    mov rdi, r9
    shr rdi, 1
    sub rsp, 8
    call f2                             ; MAL: perdí contenido de R9
    add rsp, 8
    add rax, r9                         ; MAL: f2 devuelve en eax (es int)
    ret
```

El Oráculo de Orga2 dice...

- *entonces:*

```
f:  ; rdi(V1) rsi(V2) rdx(V3) ecx(n)    ; BIEN ecx: pero ojo con la parte superior
    push rbx                            ; BIEN: lo uso entonces lo guardo antes
    mov ecx, ecx                        ; BIEN: limpia la parte superior de rcx
    xor r8, r8
    xor r9, r9
    .ciclo:
        add r9, [rdi + r8*4]
        add r9, [rsi + r8*4]
        add r9, [rdx + r8*4]
        inc r8
        loop .ciclo                     ; BIEN: porque rcx ya es consistente
    mov rdi, r9
    shr rdi, 1
    sub rsp, 8
    mov rbx, r9                         ; BIEN: hice backup de r9 en rbx
    call f2                             ; BIEN: el resultado parcial lo tengo en rbx
    add rsp, 8
    mov eax, eax                        ; BIEN: limpié la parte superior de eax que devuelve f2
    add rax, rbx
    pop rbx                             ; BIEN: lo restauro
    ret
```

El Oráculo de Orga2 dice...

- **Consejo 10:** antes de hacer un `CALL` me fijo si no pierdo datos parciales en registros por la Convención C.
- **Consejo 11:** tampoco pierdo de vista el tamaño de datos del retorno de las funciones, ni de los registros en las operaciones.
- **Consejo 11 bis:** recordar que la mayoría de las operaciones aritmético/lógicas están limitadas a inmediatos de 32 bits (salvo MOV). i.e. no se puede hacer

~~`and rax, 0x00000000FFFFFFFF`~~

El Oráculo de Orga2 dice...

- *cuando tengan que hacer un ciclo muchos van a hacer:*

```
%define miVariable [rbp-8]
.ciclo:
    ...
    add rdi, miVariable
    loop .ciclo
```

- *o también:*

```
%define primerElementoArreglo [rdi]
mov r8, rdi
mov r9, rsi
.ciclo:
    mov r10, primerElementoArreglo
    add r10, [r8]
    mov [r9], r10
    add r8, 8
    add r9, 8
    loop .ciclo
```

El Oráculo de Orga2 dice...

- *cuando tengan que hacer un ciclo muchos van a hacer:*

```
%define miVariable [rbp-8]          ; DESACONSEJADO: [ ] en defines => errores sutiles
.ciclo:
    ...
    add rdi, miVariable              ; MAL: acceso memoria innecesario dentro del ciclo
    loop .ciclo
```

- *o también:*

```
%define primerElementoArreglo [rdi]
mov r8, rdi
mov r9, rsi
.ciclo:
    mov r10, primerElementoArreglo   ; MAL: levantar el mismo valor innecesario dentro del ciclo
    add r10, [r8]
    mov [r9], r10
    add r8, 8
    add r9, 8
    loop .ciclo
```

El Oráculo de Orga2 dice...

- *entonces:*

```
%define primerElementoArreglo rdi
mov r8, rdi
mov r9, rsi
mov r11, [primerElementoArreglo]
.ciclo
    mov r10, r11                    ; BIEN: está en un registro porque no cambia
    add r10, [r8]
    mov [r9], r10
    add r8, 8
    add r9, 8
    loop .ciclo
```

El Oráculo de Orga2 dice...

- **Consejo 12:** cuando tenemos que usar un valor dentro de un ciclo, y el valor no cambia, usamos un registro, no memoria. Ojo con poner una etiqueta que simboliza un acceso a memoria, porque es lo mismo!

- **Consejo 13:** cuando tenemos que procesar una estructura y para el proceso necesitamos un valor determinado de esta estructura en un ciclo, si éste no va a cambiar, lo ponemos en un registro.

- **Consejo 14:** los consejos 12 y 13 se parecen mucho, son dos formas de lo mismo...

El Oráculo de Orga2 dice...

- *cuando accedan a una estructura como la siguiente:*

```
struct alumno{
    char* nombre;
    char comision;
    int dni;
}
```

- *muy pocos van a hacer algo como:*

```
%define offset_nombre 0
%define offset_comision 8
%define offset_dni 12
%define size_alumno 8+4+4

mov rbx, [rsi+offset_nombre]
mov cl, [rsi+offset_comision]
mov edx, [rsi+offset_dni]
mov rdi, size_alumno
call malloc
```

- *por lo que al tener que acceder varias veces durante el código, van a estar haciendo cuentas sobre los punteros, calculando tamaños en el momento, y lo más probable es que se equivoquen.*

El Oráculo de Orga2 dice...

- *lo mismo puede pasarles con:*

```
struct alumno{
    char* nombre;
    char comision;
    int dni;
}__attribute__((packed))
```

- *donde también muy pocos van a hacer algo como:*

```
%define offset_nombre 0
%define offset_comision 8
%define offset_dni 9
%define size_alumno 8+1+4

mov rbx, [rsi+offset_nombre]
mov cl, [rsi+offset_comision]
mov edx, [rsi+offset_dni]
mov rdi, size_alumno
call malloc
```

- *notar que una vez definidas las etiquetas correctamente, el código no cambia, podemos olvidarnos de los tamaños de las estructuras, los tamaños de cada campo y si están empaquetados o no.*

El Oráculo de Orga2 dice...

- **Consejo 15:** cuando tengo una estructura que voy a acceder, calculo una sola vez al principio las cuentas de acceso a cada campo (*offsets*) y los defino (`%define`) con etiquetas al principio del código. Luego me quedo tranquilo que en adelante no me voy a equivocar con eso y me concentro en el ejercicio.

- **Consejo 16:** para acceder a cada campo y saber el tamaño de la estructura, tengo cuidado con el orden en que están los campos y recuerdo cómo es que se alinean en memoria las estructuras y sus campos. Si me agarran dudas, reviso la *Clase Práctica: Memoria Dinámica - Malloc y Free*.

- **Consejo 17:** tengo cuidado con los *malloc/free*. Nunca termino una función si no hice free por cada malloc, salvo que quiera que esa memoria dinámica persista. *Nunca se me ocurriría hacer `free` de una porción de memoria que no es dinámica!*

El Oráculo de Orga2 dice...

- *cuando vayan a procesar una estrucutra como la siguiente:*

```
struct guardaConEsto{
    void *entradas[256];
    void *otras_entradas;
}__attribute__((packed))
```

- *y tengan que acceder al i-ésimo elemento de "guardaConEsto.entradas" muchos la van a indizar como:*

```
; rdi = puntero a guardaConEsto
mov r8, [rdi]
mov r9, i
mov rax, [r8 + r9]
```

El Oráculo de Orga2 dice...

- *cuando vayan a procesar una estrucutra como la siguiente:*

```
struct guardaConEsto{
    void *entradas[256];
    void *otras_entradas;
}__attribute__((packed))
```

- *y tengan que acceder al i-ésimo elemento de "guardaConEsto.entradas" muchos la van a indizar como:*

```
; rdi = puntero a guardaConEsto
mov r8, [rdi]                ; MAL: no es un puntero a estructura, es arreglo de punteros
mov r9, i
mov rax, [r8 + r9]           ; MAL: no tiene en cuenta tamaño de datos al indizar
```

El Oráculo de Orga2 dice...

- *cuando vayan a procesar una estrucutra como la siguiente:*

```
struct guardaConEsto{
    void *entradas[256];
    void *otras_entradas;
}__attribute__((packed))
```

- *y tengan que acceder al i-ésimo elemento de "guardaConEsto.entradas" muchos la van a indizar como:*

```
; rdi = puntero a guardaConEsto
mov r8, [rdi]                    ; MAL: no es un puntero a estructura, es arreglo de punteros
mov r9, i
mov rax, [r8 + r9]               ; MAL: no tiene en cuenta tamaño de datos al indizar
```

- *en lugar de:*

```
; rdi = puntero a guardaConEsto
mov r9, i
mov rax, [rdi + r9*PTR_SIZE + OFFSET_ENTRADAS]    ; BIEN: el arreglo empieza en rdi y
                                                  ; me muevo de a 1 puntero
```

El Oráculo de Orga2 dice...

- **Consejo 18:** miramos bien lo que hay dentro de la estructura y no nos dejamos llevar con sólo ver un *.
- **Consejo 19:** siempre tenemos en cuenta el tamaño de los datos cuando indizamos memoria para escalar el índice.
- **Consejo 20:** para pedir memoria para una estructura, trato de no confundirme con ver un * y pienso detenidamente cuál es el tamaño correcto teniendo en cuenta el tamaño de cada dato.
- **Consejo 21:** si voy a acceder a una estructura (`guardaConEsto` u `otras_entradas`) a través de su *, chequeo antes que no sea `NULL`, salvo que me indiquen que puedo asumir que no es `NULL`.

El Oráculo de Orga2 dice...

- **Consejo 22:** si necesito crear estructuras, me fijo que antes de terminar todo quede en un estado consistente (invariante de representación). Por ejemplo, inicializar estructuras, enlazar estructuras correctamente, etc.

- **Consejo 23:** si necesito borrar (`free`) una estructura recursiva, no me conformo con borrar sólo el primer elemento de la estructura a donde tengo el *, pienso si no es también mi responsabilidad borrar cada uno de los elemento de la estrucutra recursiva.

El Oráculo de Orga2 dice...

- *cuando "alguien" les pida que resuelvan un algoritmo determinado y que primero escriban el pseudocódigo del algoritmo y luego el código en ASM, muchos van a hacer:*

El Oráculo de Orga2 dice...

- *cuando "alguien" les pida que resuelvan un algoritmo determinado y que primero escriban el pseudocódigo del algoritmo y luego el código en ASM, muchos van a hacer:*
  - escribir primero el codigo en ASM !!
  - escribir segundo el pseudocódigo y tratar de encajarlo al código ASM

El Oráculo de Orga2 dice...

- *cuando "alguien" les pida que resuelvan un algoritmo determinado y que primero escriban el pseudocódigo del algoritmo y luego el código en ASM, muchos van a hacer:*
  - escribir primero el codigo en ASM !!
  - escribir segundo el pseudocódigo y tratar de encajarlo al código ASM
- *en lugar de:*
  - pensar tranquilamente un pseudocódigo que lidie correctamente con la estructura recursiva, haciendo dibujitos cuando haga falta que me convenzan que anda y, luego, escribir el código ASM, casi sin pensar el problema algorítmico porque me guío por el pseudocódigo, y sólo me concentro en cómo utilizar la arquitectura para hacer lo que dice el pseudocódigo

El Oráculo de Orga2 dice...

- **Consejo 24:** cuando tengo que resolver un problema donde me piden el pseudocódigo del algoritmo y luego el código en ASM, trato de hacerlo en ese orden. Así consigo un codigo ASM que coincide con el pseudocódigo, me sale más fácil y rápido.