UNIVERSIDAD AUTONOMA DE BAJA CALIFORNIA

Facultad de Ingeniería, Arquitectura y Diseño.

Ingeniería en Software y Tecnologías Emergentes

Organización de Computadoras

Taller 12

Alejandro Palomares Ceseña

1) Importancia del carácter % en macros (NASM) — teoría y ejemplos

Resumen corto: en NASM el símbolo % introduce directivas del preprocesador y se usa en la sintaxis de macros. Las directivas y parámetros de macro se escriben con % (por ejemplo %macro, %endmacro, %define, %1, %%etiqueta para etiquetas locales, %rep, %assign, etc.). El preprocesador realiza sustitución textual antes del ensamblado, por eso % es crítico. NASM+1

a) Definir macros con parámetros (NASM)

Formato básico (multi-line macro):

```
%macro PROLOGUE 1      ; espera 1 parámetro
   push ebp
   mov ebp, esp
   sub esp, %1        ; %1 es el primer parámetro del llamado
%endmacro
```

- %macro <name> <n> define la macro <name> esperando n parámetros (o rango 1-3, ver abajo).
- Dentro de la macro, %1, %2, ... hacen referencia a los parámetros posicionales. NASM

Ejemplo de %define (macro de una línea tipo #define):

```
%define BUFSIZE 128
```

b) Llamar a una macro

```
PROLOGUE 12     ; invoca PROLOGUE con %1 = 12
```

Cuando el preprocesador expande la macro, sustituye %1 por 12.

c) Macros con parámetros opcionales / rango de parámetros

NASM permite especificar rangos:

```
%macro MYSUM 1-3
   ; %1 siempre existe
   ; si no se pasan %2 o %3, están vacíos pero puedes comprobar con %0 (número de parámetros)
%endmacro
```

- %macro name min-max permite llamadas con entre min y max parámetros.

- También se puede detectar el número de parámetros pasados con la “variable” %0 dentro de la macro (consulta del preprocesador). (Foros y doc muestran técnicas para manejar parámetros opcionales).

d) Otras utilidades importantes con %

- %label crea etiquetas locales únicas por invocación de macro (útil para temporales dentro de macro sin colisión).
- %+1 y %-1 se usan para forzar/negrar códigos de condición en sintaxis de comparaciones/exchanges (documentado en las páginas del manual).

2) Ejemplos y notas prácticas sobre macros

Macro simple para imprimir un string (NASM + Linux int 0x80)

```
%macro PRINT_STRING 1
    mov eax, 4        ; syscall write
    mov ebx, 1        ; stdout
    mov ecx, %1
    mov edx, %1_len   ; asume que existe una etiqueta con longitud
    int 0x80
%endmacro
```

Macro con etiqueta local única

```
%macro COUNTDOWN 1-2
    mov ecx, %1
%%loop_label:
    ; cuerpo
    loop %%loop_label
%endmacro
```

%loop_label evita que varias invocaciones de COUNTDOWN colisionen por la misma etiqueta.

---

3) Estructuras de datos en ensamblador — ejemplos (fecha, email, dirección, CURP)

Abajo hay ejemplos en NASM (declaro db/resb / arreglos). Incluyo ejemplos de acceso y manipulación (indexado por desplazamientos, uso de punteros y bucles con esi/edi).

```
; -------------------------
; ESTRUCTURAS: DEFINICIONES
; -------------------------
section .data
    ; Fecha dd/mm/yyyy como 10 bytes ASCII: "DD/MM/YYYY"
    sample_date    db '05/11/2025', 0    ; 10 chars + 0

    ; Correo electrónico (longitud máxima 64)
    emails         times 3 db 64 dup(0)  ; array de 3 emails (espacio fijo)
    email0         db "juan.perez@example.com",0

    ; Dirección compuesta: calle, num, colonia. Usamos offsets dentro de un bloque
    ; cada dirección ocupa 100 bytes: calle(40), numero(6), colonia(40), terminador(1), padding
    dir_pool       times 5 db 100 dup(0) ; espacio para 5 direcciones

    ; CURP tipo cadena (18 caracteres + terminador)
    curps          times 10 db 19 dup(0) ; 10 CURPs almacenables
    curp_example   db "PEMJ800101HDFRRN09", 0

section .bss
    ; Buffers de trabajo
    tmp_buffer     resb 32

; -------------------------
; EJEMPLOS DE ACCESO / MANIPULACIÓN
; -------------------------
; Idea 1: leer día/mes/año desde sample_date
;   sample_date: '0' '5' '/' '1' '1' '/' '2' '0' '2' '5' 0
; dirección: offset 0..1 -> día, 3..4 -> mes, 6..9 -> año
```

```
; PSEUDO-CÓDIGO ENSAMBLADOR para convertir día ASCII a número:
;   lea esi, [sample_date]
;   mov al, [esi]        ; '0'  (decenas día)
;   sub al, '0'
;   mov ah, [esi+1]      ; '5'
;   sub ah, '0'
;   ; número = al*10 + ah
;   movzx eax, al
;   imul eax, 10
;   add eax, ah

; Ejemplo: insertar un email en emails[1] (indexado)
;   lea edi, [emails]      ; inicio del pool
;   add edi, 64            ; apunta al segundo slot (slot 1)
;   mov esi, email0
;   ; copiar hasta terminador
; copy_loop:
;   mov al, [esi]
;   mov [edi], al
;   inc esi
;   inc edi
;   cmp al, 0
;   jne copy_loop

; Ejemplo: CURP array: acceder al CURP i
;   mov ecx, i
;   mov eax, 19
;   imul eax, ecx          ; eax = i * 19 (tamaño por curp)
;   lea esi, [curps]
```

; add esi, eax

; ; esi apunta al curp i (leer/escribir desde aquí)

Objetivos de usar estos "tipos" en ensamblador:

- Agrupar campos relacionados para facilitar lectura/escritura.
- Evitar gestión dispersa de offset/constantes.
- Permitir crear rutinas de validación/serialización (por ejemplo: validar formato DD/MM/YYYY, comprobar arroba en correo, verificar longitud de CURP).
- Poder indexar con i * size para trabajar con arrays (matrices).

4) Código del enlace (OneCompiler) — compilación y documentación

```
; ------------------------------------------------
; Programa reconstruido y comentado a partir del enlace
; - Imprime un mensaje y luego la suma de algunos valores
; - Incluye macros que el ejemplo original invocaba:
;   PRINT_STRING <label>   -> escribe la cadena terminada en 0 (usa int 0x80)
;   PRINT_SUM              -> suma un array de bytes y lo imprime
; ------------------------------------------------

%define SYS_EXIT 1
%define SYS_WRITE 4
%define STDOUT 1

section .data
    message     db "La suma de los valores es: ", 0
    message_len equ $ - message

    newline     db 10, 0

    ; Array de valores (ejemplo): 5 valores de 1 byte
    values      db 5, 10, 3, 7, 2    ; suma esperada: 27
```

```
section .bss
    buffer      resb 12      ; buffer para convertir número a ascii (hasta 10 dígitos)

section .text
    global _start

; ----------------------
; Macro: PRINT_STRING <label>
; - %1 es la etiqueta que apunta a una cadena terminada en 0
; - calculamos longitud en ensamblador usando el preprocesador no funciona así,
;   así que para simplicidad pasamos la etiqueta y usamos read-char loop.
; ----------------------
%macro PRINT_STRING 1
    ; Este macro hace una write tradicional usando int 0x80
    mov eax, SYS_WRITE
    mov ebx, STDOUT
    mov ecx, %1
    ; necesitamos la longitud: calculo en tiempo de ensamblado no trivial si no usamos otra etiqueta.
    ; Para ejemplo simple, convertimos: usaré una llamada a helper que calcule y ponga len en edx.
    push ecx
    call _strlen
    pop ecx
    mov edx, eax      ; _strlen devuelve len en eax
    mov eax, SYS_WRITE
    mov ebx, STDOUT
    ; ecx ya tiene el puntero
    int 0x80
%endmacro
```

```
; ----------------------
; Helper: _strlen (simple)
; - entrada: ecx = puntero a cadena terminada en 0
; - salida: eax = longitud (sin contar el 0)
; Clobbers: edx
; ----------------------
_strlen:
    push esi
    mov esi, ecx
    xor eax, eax
_strlen_loop:
    cmp byte [esi], 0
    je _strlen_done
    inc esi
    inc eax
    jmp _strlen_loop
_strlen_done:
    pop esi
    ret


; ----------------------
; Macro: PRINT_SUM
; - suma los bytes de 'values' y los imprime (usa buffer para convertir a ascii)
; ----------------------
%macro PRINT_SUM 0
    ; Calcular suma
    xor eax, eax        ; suma en eax
    lea esi, [values]
    mov ecx, 5          ; número de elementos
```

```
.sum_loop:
    add al, [esi]
    inc esi
    loop .sum_loop

    ; Convertir AL a decimal ASCII (AL contiene suma, máximo 255)
    ; Usamos un simple conversor que escribe en buffer (resb 12)
    movzx ebx, eax      ; ebx = suma (de 0 a 255)
    lea edi, [buffer + 11]
    mov byte [edi], 0
    dec edi
    cmp ebx, 0
    jne .conv
    mov byte [edi], '0'
    jmp .print_it

.conv:
    .conv_loop:
        mov edx, 0
        mov eax, ebx
        mov ecx, 10
        div ecx             ; eax = ebx/10, edx = ebx % 10
        add dl, '0'
        mov [edi], dl
        dec edi
        mov ebx, eax
        cmp ebx, 0
        jne .conv_loop

.print_it:
```

```
    inc edi             ; edi apunta al primer dígito
    ; imprimir string en edi
    mov eax, SYS_WRITE
    mov ebx, STDOUT
    mov ecx, edi
    ; calcular longitud simple
    push ecx
    call _strlen
    pop ecx
    mov edx, eax
    mov eax, SYS_WRITE
    mov ebx, STDOUT
    int 0x80

    ; newline
    mov eax, SYS_WRITE
    mov ebx, STDOUT
    mov ecx, newline
    mov edx, 1
    int 0x80
%endmacro

_start:
    ; Imprime el mensaje inicial
    PRINT_STRING message

    ; Imprime la suma de los valores
    PRINT_SUM

    ; salir
```

```
    mov eax, SYS_EXIT
    xor ebx, ebx
    int 0x80
```

Comentarios de funcionamiento:

- PRINT_STRING es un macro que llama a _strlen para obtener la longitud y luego invoca write (int 0x80).
- _strlen recorre la cadena hasta encontrar el 0 y devuelve la longitud en eax.
- PRINT_SUM suma los bytes en values, convierte la suma a ASCII en buffer (un convertidor decimal simple) y lo imprime.
- Usé %% donde era útil para labels locales (ejemplo no mostrado por simplicidad) y %macro para las macros.
- Si quieres, en el taller puedes comentar línea a línea explicando registros usados (eax para syscall y operaciones, esi para punteros, edi para buffer, etc.).

5) Programa: estructura x/x/x y operación sobre cada X (usa macro)

```
; ------------------------------------------------------
; Programa: parsea "num/num/num" y suma los números
; usa un macro PARSE_NEXT que toma:
;   - %1 = esi (puntero a la cadena actual)
; salida del macro:
;   - eax = valor entero del segmento (por ejemplo 12)
;   - esi apuntará al siguiente carácter después del separador '/' (o al 0)
; ------------------------------------------------------

section .data
    nums db "12/3/4", 0

section .bss
    ; no usado
section .text
    global _start
```

```
; Macro: PARSE_NEXT <ptr_reg>
; Uso: PARSE_NEXT esi
; - itera desde [esi] hasta '/' o 0 y construye valor decimal en eax
%macro PARSE_NEXT 1
    xor eax, eax        ; valor resultante
    xor ebx, ebx
.parse_loop_%=:
    mov bl, byte [%1]   ; leer char
    cmp bl, '0'
    jb .end_parse_%=
    cmp bl, '9'
    ja .end_parse_%=
    ; es dígito
    sub bl, '0'
    imul eax, eax, 10   ; eax *= 10
    add eax, ebx        ; add digit
    inc dword %1        ; avanzar puntero (%1 es un registro, el preprocesador sustituye
literal)
    jmp .parse_loop_%=
.end_parse_%=:
    ; si char es '/', avanzar una posición para la próxima llamada
    mov bl, byte [%1]
    cmp bl, '/'
    jne .parse_done_%=
    inc dword %1
.parse_done_%=:
%endmacro

_start:
```

```
lea esi, [nums]
xor edi, edi        ; sumador total en edi

; Parsear primero
PARSE_NEXT esi      ; produce eax = 12 ; esi avanza
add edi, eax

; Parsear segundo
PARSE_NEXT esi      ; produce eax = 3
add edi, eax

; Parsear tercero
PARSE_NEXT esi      ; produce eax = 4
add edi, eax

; Ahora edi contiene la suma total (19)
; (podrías imprimirlo usando una rutina de conversión similar al ejemplo anterior)
; Para terminar:
mov eax, 1
xor ebx, ebx
int 0x80
```

Notas:

- En el macro uso %= para generar etiquetas locales únicas a cada expansión del macro (esto evita colisiones en múltiples invocaciones).
- El macro está pensado para tomar un registro (ej. esi) por nombre; el preprocesador sustituye %1 por esi. En una implementación real hay que asegurarse que la sintaxis inc dword %1 sea válida para el registro usado (algunos ensambladores prefieren inc esi) — ajusta según el ensamblador/convenio. La idea principal (y lo que debes documentar en el taller) es la separación de responsabilidades: el macro parsea un bloque, el main suma resultados.