Interacción entre Assembler y lenguajes de alto nivel

Interacción HLL / Assembler

- ¿Para qué?
  - Acceso a instrucciones sin equivalentes en el HLL
    - Ej.: manipulación de bits, funciones directas del hardware, sets de instrucciones nuevos como MMX, SSE…
  - Control absoluto del código generado para optimizaciones

Qué nos proponemos

- Escribiremos funciones y procedimientos como subrutinas de Assembler
- Las invocaremos desde un lenguaje de alto nivel (en nuestro caso, Pascal)

Qué problemas encontramos

- ¿Cómo funciona la llamada a subrutinas (procedimientos, funciones) en los lenguajes de alto nivel?
  - ¿Dónde se pasan los parámetros?
  - ¿En qué orden?
  - Si es en la pila, ¿quién se encarga de borrar los parámetros?
- Todo esto lo determina la **calling convention** use el lenguaje

Pascal calling convention

- Los parámetros se pasan
  - En la **pila**
  - **De izquierda a derecha** (se hace PUSH de cada parámetro en orden)
- Si la subrutina devuelve un valor (función)
  - Debe escribirse en **AX**
- Se debe **preservar** el valor de los registros **BP**, **SP**, **SS** y **DS** (si se modifican, hay que restaurarlos)
- Pueden modificarse libremente los registros AX, BX, CX, DX, SI, DI, ES y Flags

Pasaje por valor y por referencia

- Para el pasaje de parámetros por **valor**, simplemente se hace PUSH de cada valor a la pila
- Cuando el pasaje es por **referencia**, se envía la **dirección completa**:

```
PUSH SEG DATO
PUSH OFFSET DATO
```

- En la subrutina, para cargar el parámetro se utiliza la instrucción:

```
LDS destino, dir_fuente
```

  - Carga:
    - destino ← [dir_fuente]
    - DS ← [dir_fuente + 2]

Ejemplo: Hola mundo

```
.model small
.data
  msg db "Hola mundo$"
.code
  ; Declaraciones
  public holaMundo

  holaMundo proc
      mov ah, 09h
      mov dx, offset msg
      int 21h
      ret
  holaMundo endp
end
```

Archivo *subs.asm*

Ej.: Hola mundo (cont.)

- Ensamblamos como de costumbre (no linkeamos, esto lo hace el programa):

```
MASM subs;
```

- Esto nos genera el archivo **.obj** que es el que vamos a referenciar desde el programa en Pascal

Ej.: Hola mundo (cont.)

Referencia a **subs.obj**

```
{$L subs}

{Declaración de las rutinas externas}
procedure holaMundo; external;

begin
   holaMundo;
end.
```

Archivo *prog.pas*

Ejemplo: suma

```
public holaMundo
public suma
...
suma proc
   push bp
   mov bp, sp

   mov ax, ss:[bp + 4]
   add ax, ss:[bp + 6]

   pop bp
   ret 4
suma endp
```

Los parámetros se reciben por **valor** en la pila, sólo hay que extraerlos

Un mismo archivo puede tener varias subrutinas

El valor de retorno se almacena en **AX**

Ejemplo: suma (cont.)

```
{$L subs}

procedure holaMundo; external;
function suma(n1, n2 : integer) : integer; external;

var x : integer;
begin
  x := suma(2, 5);
  writeln(x);
  ...
end.
```

Ejemplo: mostrarString

```
mostrarString proc
   push bp
   mov bp, sp

   mov bx, ds

   lds si, ss:[bp + 4]

   mov cl, ds:[si]
   mov ch, 0
```

```
bucle:
   inc si
   mov dl, ds:[si]
   mov ah, 02h
   int 21h
   loop bucle

   mov ds, bx

   pop bp
   ret 4
mostrarString endp
```

Ej.: mostrarString (cont.)

- Al ejecutarse la siguiente instrucción

```
LDS SI, SS:[BP + 4]
```

  - lo que está en BP+4 se carga en SI
  - lo que está en BP+6 se carga en DS
- De esa forma, DS:[SI] queda apuntando a la dirección de la variable pasada por referencia

Ej.: mostrarString (cont.)

- Los strings de Pascal se representan en memoria como arreglos de caracteres ASCII.
  - El primer elemento del arreglo (índice 0) almacena la cantidad de caracteres del string (máximo 255).
  - El string arranca desde el elemento 1.
- Para cargar la longitud en CX hacemos:

```
mov cl, ds:[si]
mov ch, 0
```

Ej.: mostrarString (cont.)

```
{$L subs}


...
procedure mostrarString(s : string); external;

begin
   mostrarString('Chau...');
   ...
end.
```

¿Este string no está pasado por valor?...

No. Aunque no es evidente, todos los strings se psasan siempre por referencia, para evitar la demora de la copia

Debugging

- Nuestros programas en Pascal + Assembler pueden correrse paso a paso usando el debugger, tal como veníamos haciéndolo
- Pero nos vamos a encontrar con mucho código desconocido (es el resultado de la traducción de Pascal a Assembler)
  - Esto hace que sea difícil encontrar dónde está nuestro código

Debugging (cont.)

- Algo que nos puede ayudar es utilizar una nueva interrupción del procesador:

```
INT 3h
```

- Esta interrupción permite generar un **break point**
- Dentro del debugger, podemos hacer correr el programa (F9) y éste se detendrá al encontrar la INT 3h, permiténdonos estudiar el estado de los registros y de la memoria.
  - A partir de ahí, si lo deseamos podemos continuar paso a paso (F7).

Inline Assembler

- Algunos lenguajes (por ej. Pascal, C y C++) poseen otra forma de aprovechar las ventajas de la codificación a bajo nivel: el “Assembler embebido” (**inline Assembler**).

- Se trata de mezclar código de alto nivel y Assembler dentro del **mismo** archivo fuente. Se definen **bloques de código** dentro de los cuales se escribe Assembler.

Inline ASM: ventajas y desventajas

- Ventajas:
  - Posee las ventajas de escribir código Assembler: control total y acceso a instrucciones del procesador no disponibles mediante las construcciones de alto nivel
  - Permite usar las variables y estructuras de datos definidas en el HLL dentro de los bloques de Assembler

- Desventajas:
  - El compilador debe estar preparado para las instrucciones extendidas (SSE, etc.) si pretende usarlas
  - El código resultante suele ser más difícil de seguir

Inline ASM en Pascal

- Es tan simple como usar un bloque **asm…end**

```
var x, y : integer;
begin
    writeln('Ingrese dos números');
    readln(x);
    readln(y);

    asm
        mov ax, x
        mul y
        mov x, ax
    end;

    write('La multiplicación da: ');
    writeln(x);
end.
```

Notar que usa directamente las variables **x** e **y** dentro del bloque de Assembler

Reglas y restricciones

- Debe cumplir con las mismas reglas en cuanto a los registros:
  - Puede modificar libremente cualquier registro **excepto BP**, **SP**, **SS** y **DS** (si se modifican, hay que restaurarlos)
  - Las variables y parámetros pueden usarse directamente mediante su nombre
  - Para acceder a **arreglos y strings** debe cargase su dirección en un registro usando la instrucción **LEA** (*Load Effective Address*)
  - Las **etiquetas** internas deben comenzar por el símbolo **@** y pueden repetirse en varios bloques

Otro ejemplo

```
var s : string;
var cantA : integer;
begin
    write ('Ingrese su nombre: ');
    readln(s);
    cantA := 0;

    asm
        lea bx, s
    @bucle:
        inc bx
        cmp byte ptr ds:[bx], 'a'
        jne @fin_bucle
        inc cantA  {si encontramos una a, la contamos}
    @fin_bucle:
        loop @bucle
    end;

    write('La letra a aparece ');
    write(cantA);
    writeln(' veces en su nombre.');
end.
```

Etiquetas locales comiezan con @

Necesitamos BYTE PTR para que sepa el tamaño de los operandos

Los comentarios deben ser en formato Pascal (no con punto y coma)

Saltos hacia fuera del bloque ASM

- Si queremos saltar desde dentro del código Assembler hacia fuera, debemos declarar las etiquetas usando la palabra clave **label**:

```
label afuera;
begin
    ...
    asm
        ...
        jmp afuera
        ...
    end;
    ...
afuera:
    ...
end.
```

Procedimientos completos en ASM

- Pascal también permite definir funciones y procedimientos **íntegramente** usando inline Assembler.

- Para esto, se combina el **encabezamiento** de un procedimiento o función estándar y se usar **asm…end** en lugar de begin…end.

Ejemplo

La directiva **assembler** indica que el cuerpo de la función está directamente en ASM

```
function suma(x, y : integer) : integer; assembler;
asm
    mov ax, x
    add ax, y
end;

begin
    writeln(suma(3, 8));
    readln;
end.
```

La instrucción *ASM* genera el código:
PUSH BP
MOV BP, SP

END genera:
POP BP
RET k
Donde k depende de los parámetros

Uso para debugging

- Aún cuando no trabajemos con inline Assembler para hacer nuestras subrutinas, una buena idea puede ser usarlo para incluir **breakpoints en el código de alto nivel**.

- Basta poner, en cualquier punto del código Pascal lo siguiente:

```
asm
    int 3h
end;
```