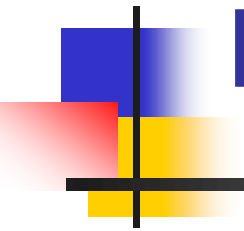
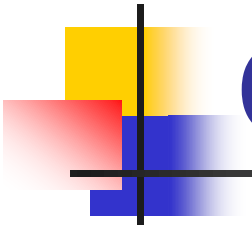


Sistemas de Procesamiento de Datos

Assembler (parte 4)

Profesor: Fabio Bruschetti
Aydte: Pedro Iriso
Ver 2019



Objetivo

- Comprender la distribución de las estructuras en la memoria y los métodos de accesos a las mismas.
- Entender el tipo abstracto de datos conocido como stack
- Entender el uso de subrutinas, el pasaje de parámetros y el uso del stack en programación assembler.

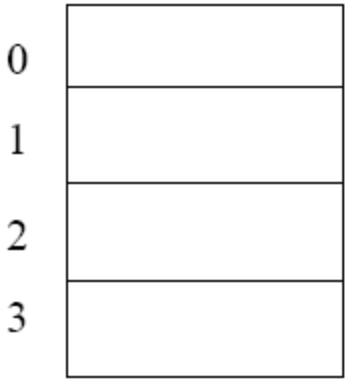
Estructuras de Datos

- El acceso a las estructuras de datos, se realiza mediante el uso del modo de direccionamiento INDIRECTO.
- Estructuras a revisar:
  - Arreglos (arrays): Una colección de elementos del mismo tipo.
    - Int arreglo[10];
    - arreglo[0] = 1; arreglo[1] = 5;
  - Registros o Estructuras: Una colección de elementos de distintos tipos.
    - Struct {
      char nombre[80];
      int numero;
      } estudiante;
    - estudiante.numero = 123456;

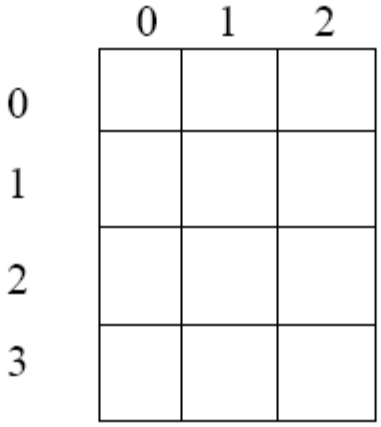
Datos - Arreglos

- Los arreglos son estructuras de datos indexadas
  - arreglo[i] = dir de comienzo + i * TamañoElemento.
  - Los offsets dependen del tamaño del elemento.
- El Primer elemento de un vector es asociado con el índice 0.

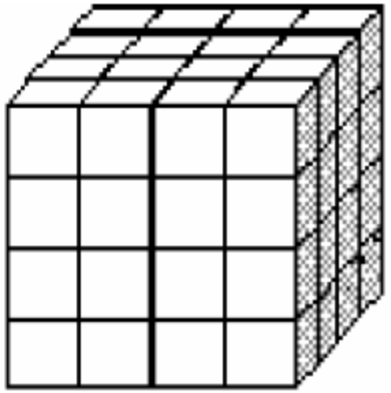
Vector



Matriz 2-D



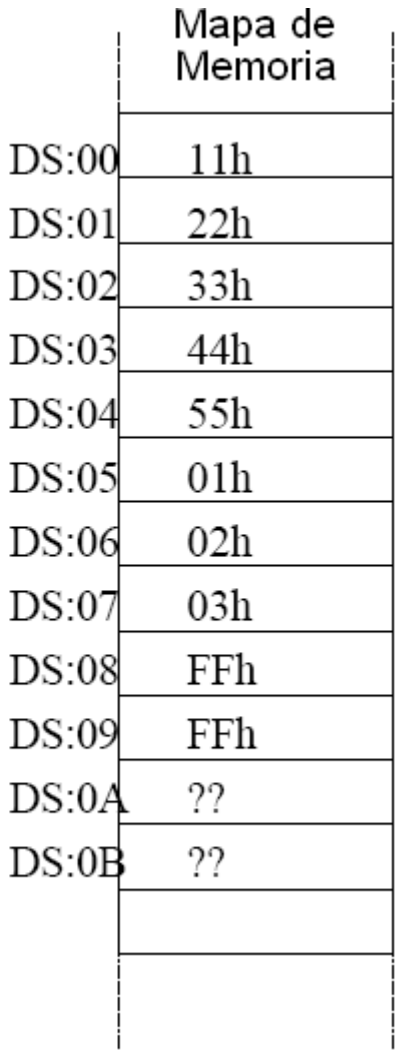
Matriz 3-D



Datos - Arreglos

```
.data
a1 db 11h, 22h, 33h, 44h, 55h
a2 db 01h
   db 02h
   db 03h
a3 db  2 dup(0FFh)
```

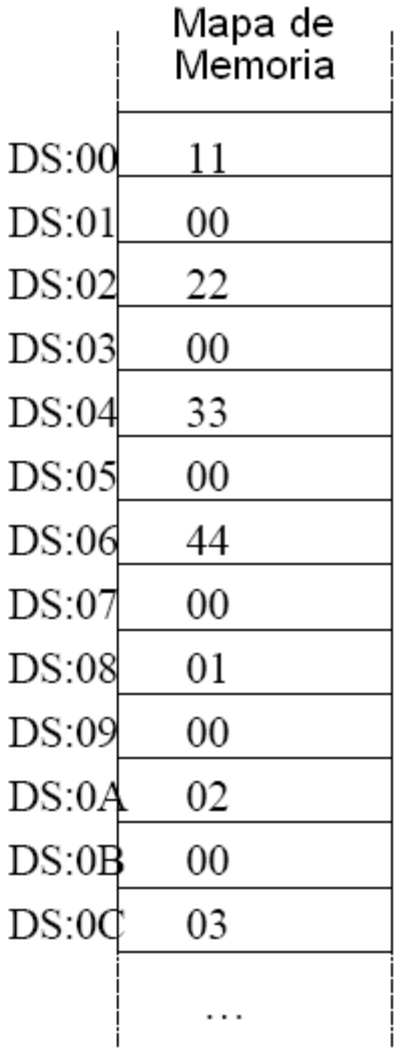
```
C :
  byte a1[5];   a1[0] = 17;
  byte a2[3];   a2[1] = 2;
  byte a3[2];   a3[1] = 255;
```



Datos - Arreglos

```
.data
a1 dw 11h, 22h, 33h, 44h
a2 dw 01h
   dw 02h
   dw 03h
a3 dw 2 dup(0FFFFh)
```

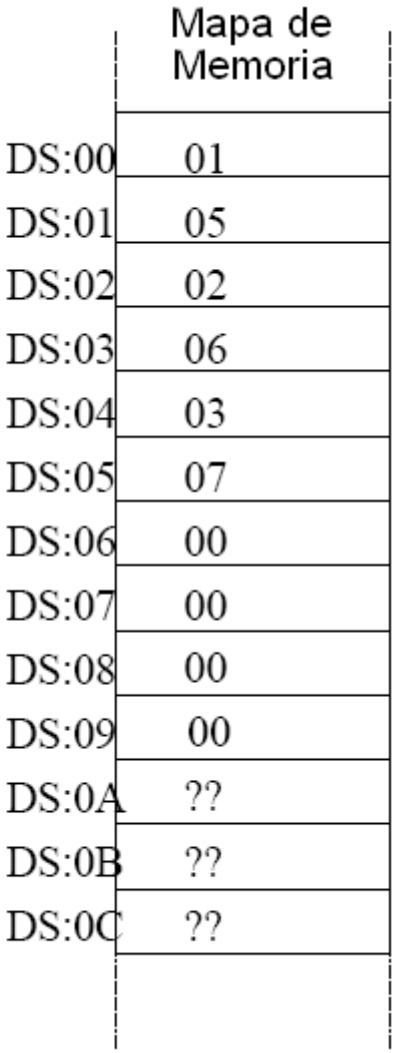
```
C:
  int a1[4];   a1[0] = 17;
  int a2[3];   a2[1] = 2;
  int a3[2];   a3[1] = 255;
```



Datos - Arreglos

```
.data
a1 db 01h, 05h
   db 02h, 06h
   db 03h, 07h
a2 db 2*2 dup(00)
```

```
C:
  byte a1[3][2];   a1[0][1] = 5;
  byte a2[2][2];   a2[1][1] = 0;
```



Datos - Arreglos - Ejemplo

- Se tiene un arreglo de enteros declarado:

```
X       DW      ; 1er elemento
        DW      ; 2do elemento
        ...     ; etc.
numX    DW      ; cantidad elementos en X
```

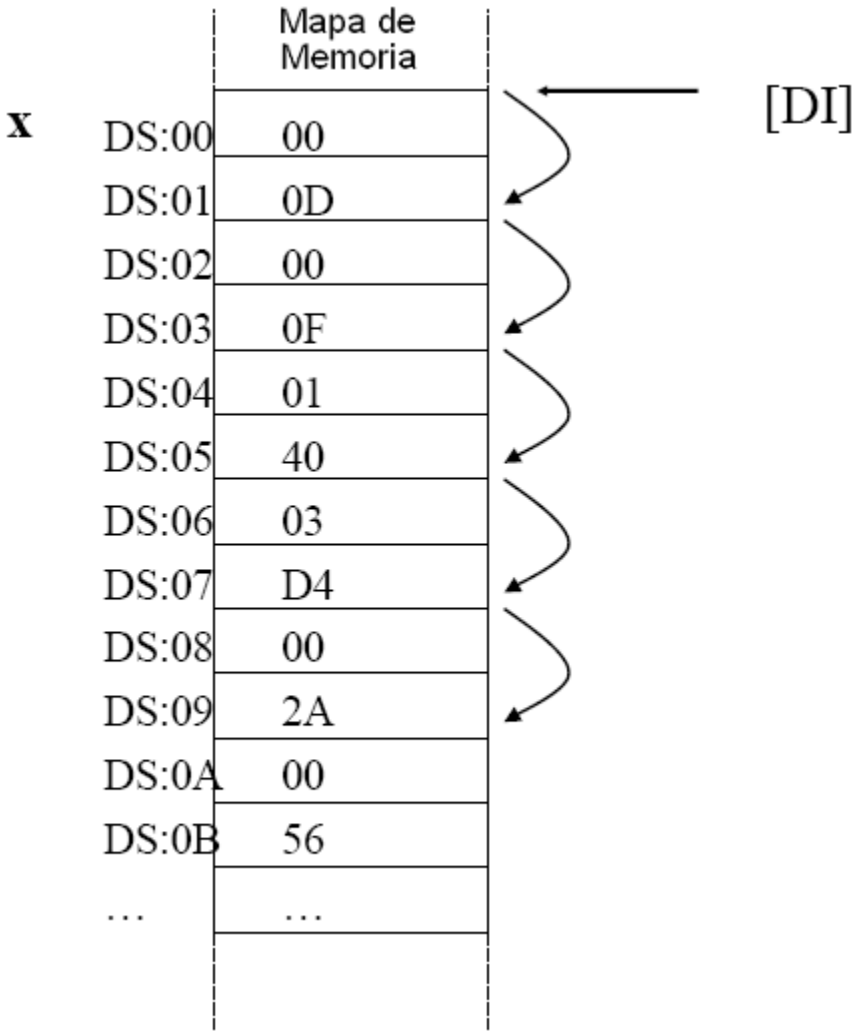
- Escriba un programa que sume todos los elementos del arreglo X en AX

```
int total = 0
for (int i=0; i <numX; i++)
  {
        total += x[i]
  }
```

- Use **DI** para contener la dirección del elemento actual, o sea juege el rol de **x[i]**

Datos - Arreglos - Ejemplo

- AX = total
- CX = contador de ciclo **i**
- DI = dirección de **x[i]**



Datos - Arreglos - Ejemplo

- Fragmento de Código

```
        MOV     AX, 0           ; inicializa sum
        LEA     DI, x           ; inicializa el índice del arreglo
        MOV     CX, numX        ; obtiene la cantidad de elementos
    Ciclo:
        CMP     CX,0            ; quedan elementos?
        JE      Listo
        ADD     AX, [DI]        ; suma el elemento i-ésimo
        ADD     DI, 2           ; ajusta el índice (offset)
        SUB     CX, 1
        JMP     Ciclo
    Listo: ...
```

Porque 2?

Acceso Registro-Indirecto

Datos - Arreglos - Ejemplo

- Fragmento de Código

```
        MOV     AX, 0           ; inicializa sum
        MOV     DI, 0           ; inicializa el índice del offset
        MOV     CX, numX        ; obtiene la cantidad de elementos
    Ciclo:
        CMP     CX,0            ; quedan elementos?
        JE      Listo

        ADD     AX, [DI + x]    ; suma el elemento i-ésimo
        ADD     DI, 2           ; ajusta el índice (offset)
        SUB     CX, 1
        JMP     Ciclo
    Listo: ...
```

Dinámico, DI contiene el offset al elemento

La dirección de comienzo de x es estática

Acceso Indirecto-Indexado

Datos - Arreglos - Ejemplo

- Fragmento de Código

```
        MOV     AX, 0               ; inicializa sum
        LEA     BX, x               ; inicializa el indice del arreglo
        MOV     SI, 0
        MOV     CX, numX            ; obtiene la cantidad de elementos
Ciclo:
        CMP     CX,0                ; quedan elementos?
        JE      Listo
        ADD     AX, [BX + SI]       ; suma el elemento i-esimo
        ADD     SI, 2               ; ajusta el indice (offset)
        SUB     CX, 1
        JMP     Ciclo
Listo: ...
```

Acceso Indirecto Base Indexado

Datos - Arreglos - Ejemplo

```
a1      db 01h, 05h
        db 02h, 06h
        db 03h, 07h

        MOV BX, offset a1
nextRow:
        MOV SI, 0
thisRow:
        MOV [BX][SI], 10
        ADD SI, 1
        CMP SI, 2*1
        JB  thisRow
        ADD BX, 2*1
        CMP BX, 3*2
        JB nextRow
```

```
C :
  byte a1[3][2];
  for (int i=0;i<3;i++)
     for (int j=0;j<2;j++)
                    a1[i][j] = 10;
```

Datos - Estructuras

- Las estructuras son un grupo de variables relacionadas que pueden ser accedidas a través de un nombre común
- Cada ítem dentro de la estructura tiene su propio tipo de datos, no necesariamente el mismo para todos los ítems
- Ejemplo:

```
struct catalog_tag {
        char        author [40];
        char        title [40];
        char        pub [40];
        unsigned int date;
        unsigned char rev;
} card;
```

Se accede:

card.author[0]

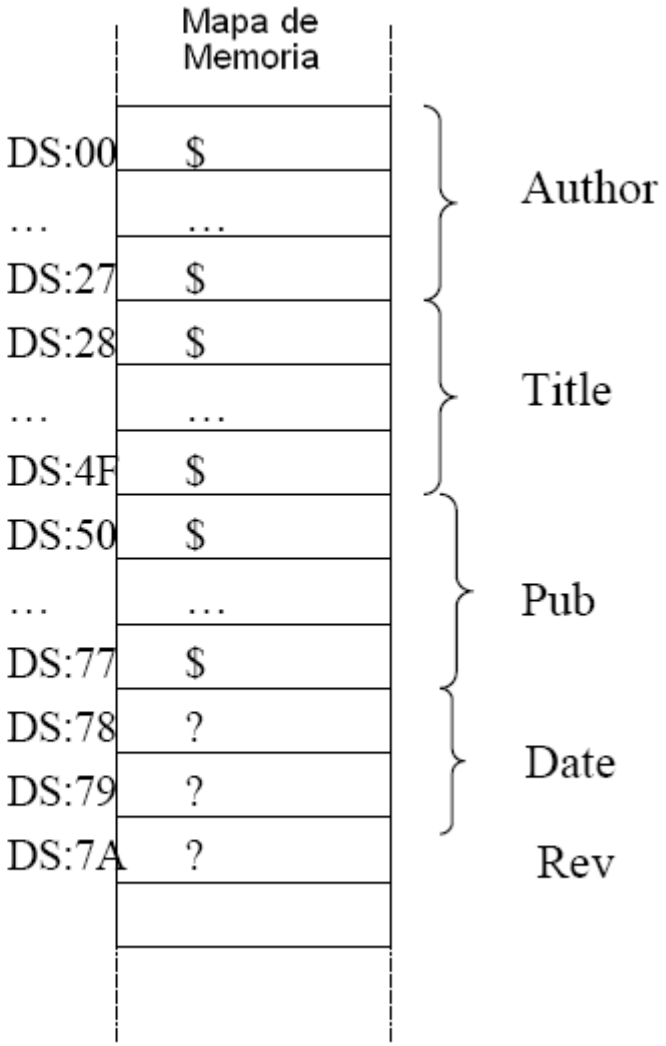
card.date

card.rev

- Donde, la variable ***card*** es del tipo ***catalog_tag***.

Datos - Estructuras

```
.data
card db 40 dup (‘$’)
     db 40 dup (‘$’)
     db 40 dup (‘$’)
     dw ?
     db  ?
```



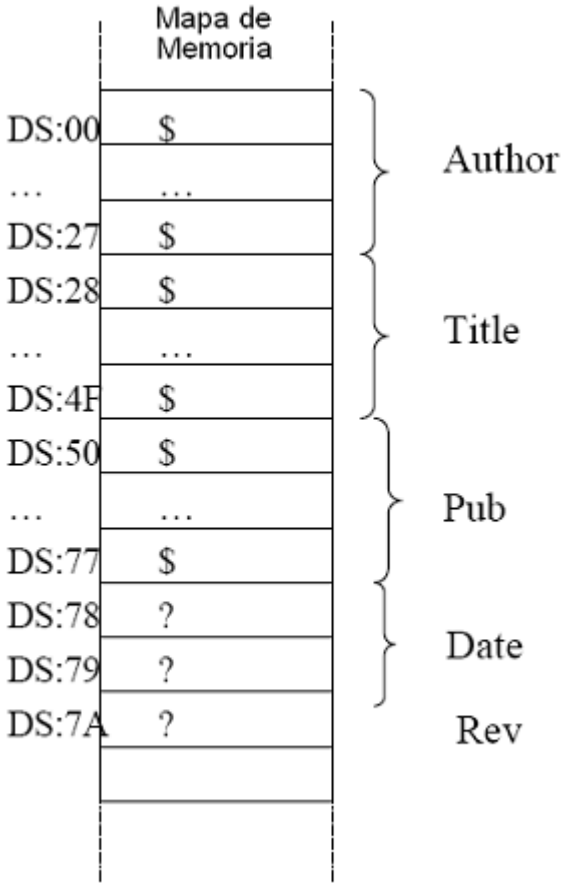
Datos - Estructuras

- Escriba un código que limpie todos los campos

```
AUTHOR      EQU 0
TITLE       EQU AUTHOR+40
PUBLISHER   EQU TITLE+40
DATE        EQU PUBLISHER+40
REVISION    EQU DATE+2
.data
card   db 40 dup(?)      ;Author
       db 40 dup(?)      ;Title
       db 40 dup(?)      ;Publisher
       dw     ?          ;Date
       db     ?          ;Revision
.code
       MOV    BX, offset card
       MOV    BYTE PTR [BX+AUTHOR], '$'
       MOV    BYTE PTR [BX+TITLE], '$'
       MOV    BYTE PTR [BX+PUBLISHER], '$'
       MOV    WORD PTR [BX+DATE], 0
       MOV    BYTE PTR [BX+REVISION], 0
```

Constante es el offset del elemento dentro de la estructura

BX constante, dirección de comienzo de la estructura



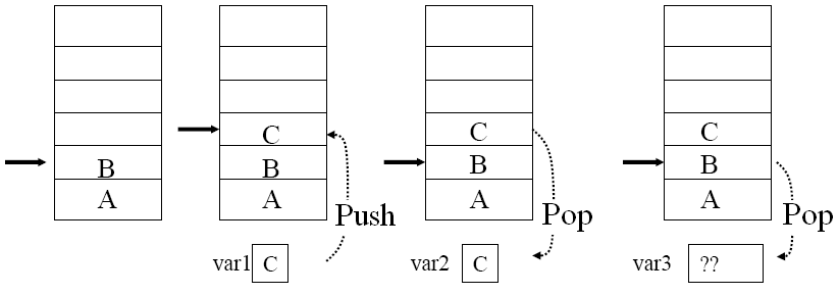
Datos - Estructuras

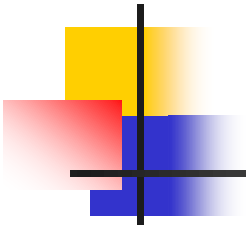
- Escriba un código que limpie todos los campos del arreglo de una estructura
  - typedef struct catalog_tag{
    char author[40];
    char title[40];
    char pub[40];
    unsigned int date;
    unsigned char rev;
    };
  - catalog_tag library[100];
- Código:

```
    MOV BX, offset library
    MOV CX, 100
proxlib:
    MOV BYTE PTR [BX+AUTHOR], ´$´
    MOV BYTE PTR [BX+TITLE], ´$´
    MOV BYTE PTR [BX+PUBLISHER], ´$´
    MOV WORD PTR [BX+DATE], 0
    MOV BYTE PTR [BX+REVISION], 0
    ADD BX,40+40+40+2+1
    LOOP proxlib
```

Datos - Stack

- Definición de Stack: es una estructura de comportamiento LIFO que se usa para mantener valores temporalmente.
  - LIFO:
    - Cada elemento es agregado en la cima (TOP), quedando el nuevo elemento en el tope.
    - Los elementos se remueven del tope y el elemento que estaba bajo este, se convierte en el nuevo tope.
    - Los elementos abajo del tope pueden ser vistos si se conoce la dirección relativa al tope.
      - Ej: Ver el segundo elemento desde el tope.





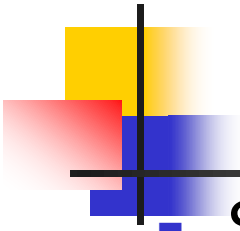
Datos - Stack

- El stack se implementa reservando:
  - Un bloque de memoria para almacenar los valores
  - Un puntero al valor del tope (TOP)
- El stack crece de direcciones altas a bajas
- Luego de la inicialización, el TOP, apunta justo fuera del bloque reservado para que en el próximo PUSH se ajuste el puntero antes de copiar el valor.
- Cuando se llena el stack (no hay mas espacio) al agregar (PUSH) ítems sucede STACK OVERFLOW.
- Cuando el stack está vacío al sacar (POP) ítems sucede STACK UNDERFLOW

Datos - Stack

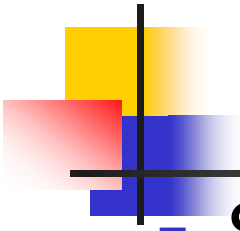
- Implementación de Stack

```
.data
top dw 102h    ;Dirección del TOP
stack db 100h dup(?)      ;Stack = arreglo de bytes
...
    MOV         BX, top              ; mueve AL al stack
    SUB         BX, 1
    MOV         [BX]. AL
    MOV         top, BX
...
    MOV         BX, top              ; lee el 7mo elemento desde el TOP
    ADD         BX, 7
    MOV         AL, [BX]
...
    MOV         BX, top              ; Saca el elemento TOP del stack
    MOV         AL, [BX]
    ADD         BX, 1
    MOV         top, BX
```



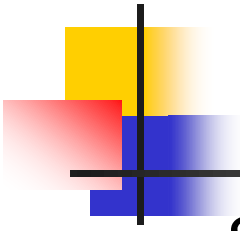
Datos - Stack

- Stack de hardware o de runtime
  - El procesador tiene un stack incorporado llamado runtime stack
  - El TOP es administrado por registro dedicados SS:SP
  - SS:SP apunta a un stack de valores de 16 bits.
  - El runtime stack crece hacia abajo en memoria (dir. altas a bajas)
  - Algunas instrucciones hace uso o alteran el runtime stack implícitamente (modificar SS:SP)
- Un programa debe inicializar el SP antes de hacer uso de las operaciones de stack
  - ***.stack size***
    - El ensamblador reserva una cantidad específica de bytes como bloque de memoria para usar el stack
    - Esta directiva es traducida en instrucciones para el loader (cargador) para inicializar SS y SP.
    - SP apunta al byte de mas arriba (dir mayor) del último byte reservado para stack



Datos - Stack

- Stack de Intel
  - PUSH operando Agrega un nuevo valor al stack
    - Se debe especificar un operando origen de 16 bits
    - El operando puede ser registro o memoria
    - El stack crece para abajo (hacia las dir mas bajas)
      - SP := SP – 2
      - mem[SP] := operando
  - POP operando Remueve un valor del stack
    - Se debe especificar un operando destino de 16 bits
    - El operando puede ser registro o memoria
      - operando := mem[SP]
      - SP := SP + 2



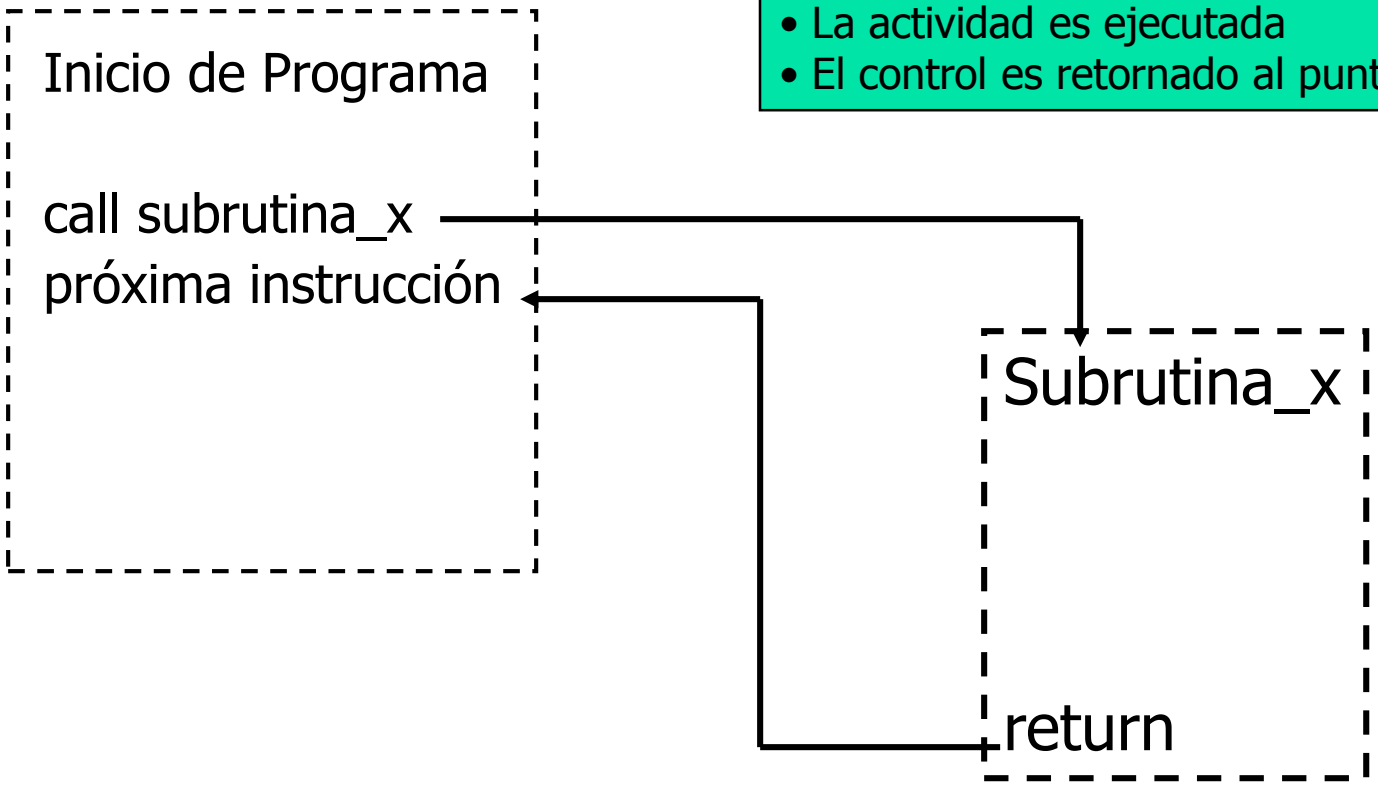
Código - Subrutinas

- Subrutinas
  - Es una secuencia de instrucciones que pueden ser llamados desde varios lugares en el programa
  - Permite realizar la misma operación pero con distintos parámetros
  - Simplifica el diseño de un programa complejo usando la aproximación divide y conquista
  - Simplifica el testeo y mantenimiento
  - Ocultamiento de información en la representación interna de variable auxiliares para resolver la operación implementada
  - En lenguajes de alto nivel se llaman FUNCIONES, PROCEDIMIENTO o METODOS
  - En lenguaje assembler se llaman SUBRUTINAS

Código - Subrutinas

- Ejecución de Subrutinas

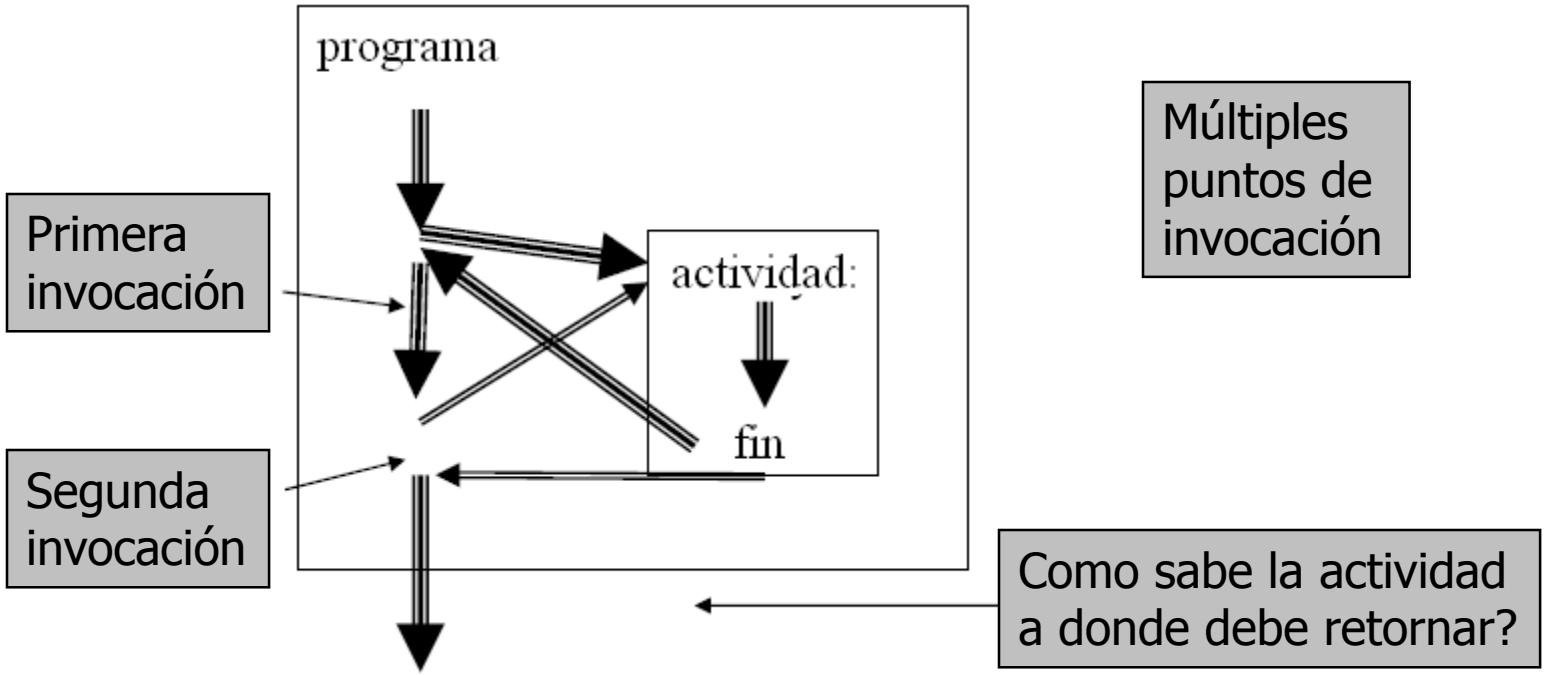
Las subrutinas son una forma de control de flujo
- El control es pasado a la actividad
- La actividad es ejecutada
- El control es retornado al punto de invocación



Flujo de Programa durante una llamada a subrutina

Código - Subrutinas

- Ejecución de Subrutinas



Durante la invocación se debe salvar el punto de invocación
Durante el retorno, el punto de invocación debe restablecerse

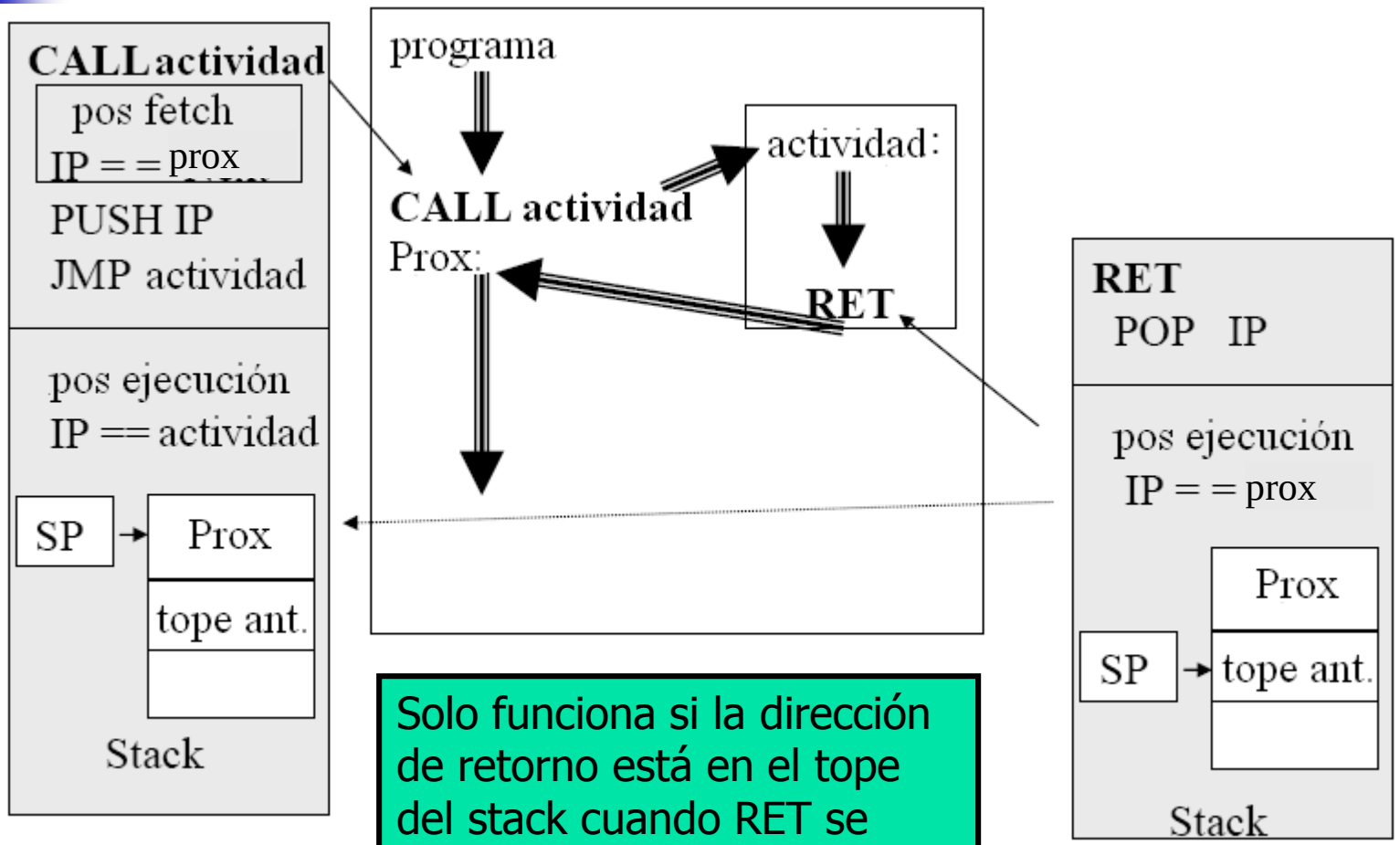
Código - Subrutinas

- Implementación de Subrutinas a nivel de maquina
  - CALL destino ; Invoca la rutina destino
    - Semántica de la Ejecución
      - Salvar la dirección de retorno en el STACK de ejecución (runtime)
        - PUSH IP
      - Transferir el control a destino
        - JMP destino

Valor de IP posterior al fetch del CALL

  - RET ; Retorna de la subrutina
    - Semántica de la Ejecución
      - Retorna el control a la dirección salvada en el tope del stack
        - POP IP

Código - Subrutinas



Solo funciona si la dirección de retorno está en el tope del stack cuando RET se ejecute : ***Responsabilidad de la subrutina***

Assembler - Subrutinas

- Informalmente: una subrutina es una secuencia de instrucciones nombrada que termina con una instrucción RETURN
- El assembler de Intel tiene directivas adicionales que proveen una encapsulamiento de la subrutina

```
main    PROC                    subr PROC
   ….                              …
   MOV   AX, 4C00                  RET
   INT   21h                    subr ENDP
main ENDP
END main
```

C - Subrutinas y parámetros

```
unsigned int displayAddress;
int main ()
{   int number = 5, number2 = 6;
    display ( number2, 0);
    …
}
void display (word number, byte base)
{   int divisor, digit;
    if (base == 0) divisor = 2
    else divisor = 2;
    digit = number / divisor;
    …
    displayAddress++;
}
```

Variable Global

number2 es un PARAMETRO

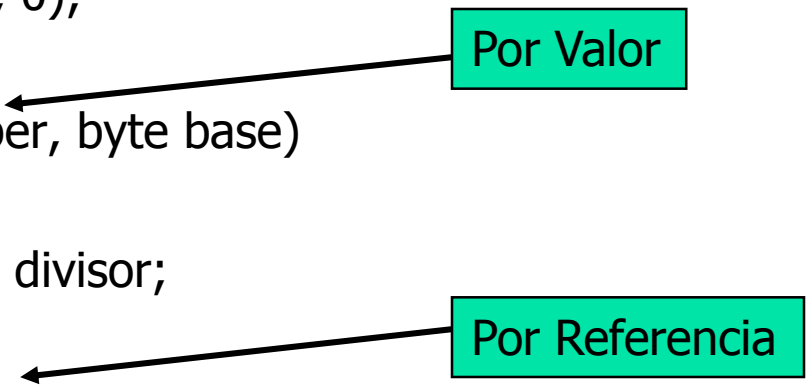
Variable Local

number es un ARGUMENTO

C - Subrutinas y parámetros

- **Por Valor vs Por Referencia**

```
int main ()
{       int number = 5, number2 = 6;
        display1 ( number2, 0);
        display2 ( &number, 0);
}
void display1 (word number, byte base)
{       …
        number = number / divisor;
}
void display2 (word &number, byte base)
{       …
        number = number / divisor;
}
```



Assembler - Pasaje Parámetros

- Los parámetros pueden ser pasados en varias formas:
  - Variables globales
  - Registros
  - Stack

- Por **Variables Globales**
  - El parámetro es una variable compartida estáticamente en memoria (alcanzable por el **invocante** y el **invocado**)
  - El pasaje de parámetros sucede cuando:
    - El **invocante** pone el valor en la variable
    - El **invocado** lee el valor de la variable

Assembler - Pasaje Parámetros

- Ejemplo: Prototipo en C: void actividad(word Valor)

```
                    Valor:   DW
Invocante:                   MOV Valor, 245
                             CALL Actividad

Invocado:           actividad PROC
                             MOV AX,Valor
                             RET
                    actividad ENDP
```

- El Pasaje de Parámetros vía variables globales no es muy usado en la práctica
  - Considere las subrutinas anidadas (o autoinvocantes)
  - Considere un programa con muchas rutinas con muchos parámetros
  - Aunque, hay veces que es la única forma de pasaje de parámetros por ejemplo en las interrupciones.

Assembler - Pasaje Parámetros

- Los Parámetros pueden ser pasados por **Registros**
  - Cada parámetro es asignado a un registro en particular
  - El **invocante** carga los registros con los valores apropiados
  - El **invocado** lee los registros para obtener el valor.
- Parámetros por Registros son usados por DOS
  - MOV AH, 9 ;AH= Función del S.O (9=Imprimir)
  - MOV DX, OFFSET mensaje ;DX=Dirección del mensaje
  - INT 21h ;"llamada" a la función de DOS
- Ventaja:
  - Poco overhead ya que los valores están en los registros
- Desventaja:
  - Existe un número finito de registros
  - ¿Que se hace si hay mas parámetros que registros?

Assembler - Pasaje Parámetros

- Los Parámetros pueden ser pasados por **Stack**
  - El **invocante** pone (push) los valores en el stack
  - El **invocado** indexa dentro del stack para obtener los valores
- Ejemplo: Prototipo en C: void actividad(word Valor)

```
Invocante:      MOV DX, 245
                PUSH DX
                CALL Actividad

Invocado:       actividad PROC
                MOV BP,SP
                MOV AX, [BP + ?]   <-- Direccionamiento Indirecto
                RET
                actividad ENDP
```

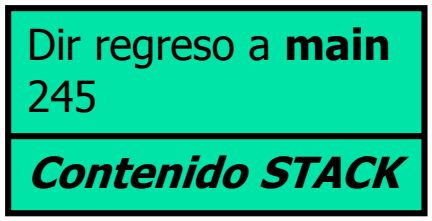
C - Pasaje Parámetros por Stack

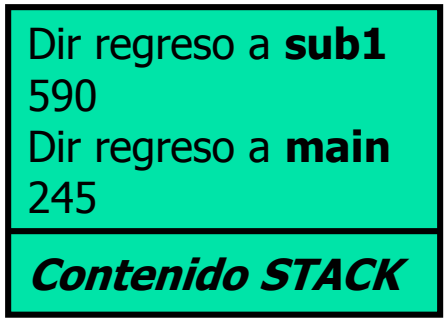
- Llamados a Subrutinas anidadas

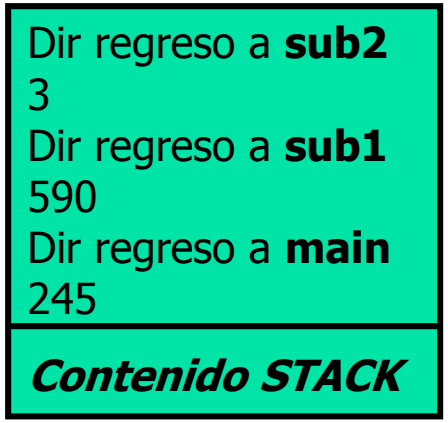
```
void main()
{
        ...
        sub1(245);
}
```

```
void sub1(word valor)
{
        ...
        sub2(valor *2);
}
```

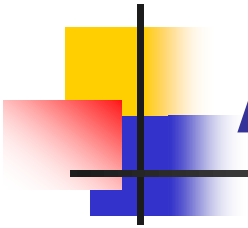
```
void sub2(word valor)
{
        ...
        sub1(3);
}
```







Assembler - Subrutinas



- Posible Problema: Si las rutinas necesitan usar registros, que son utilizados por el invocante luego del return?
- Solución: Asignarle responsabilidades al invocante o al invocado
  - El invocante tiene la responsabilidad de salvar todos los valores útiles antes de la llamada a la subrutina
    - El invocado está liberado de usar cualquier registro
    - Luego del regreso, el invocante restaura los valores útiles
  - El invocado (subrutina) debe salvar los registros antes de usarlos y restaurarlos a su valor original antes del regreso
    - El invocante tiene la garantía de que sus registros contiene los mismos datos antes y después del llamado a la subrutina
    - Mas eficiente dado que la subrutina sabe qué registros usa.
- Solución: La subrutina debe salvar los registros antes de usarlos y restaurarlos a su valor original antes del regreso, excepto del registro utilizado para retornar el resultado de la función.

C - Subrutinas - Variables Locales

- Las subrutinas suelen utilizar variables locales que existen solo durante la ejecución de la subrutina
- Ejemplo

```
double average (double array[], int numero)
{
   double total= 0;
   for (int i=0; i< numero; i++ )
   {
            total += array[I];
   }
   double resultado = total / numero;
   return resultado;
}
```

- Las variables locales se almacenan como variables registros o usando el stack como un buffer temporario de almacenamiento.

Assembler - Pasaje de Parametros

- Los parámetros al ser pasados por el STACK
  - PUSHeados de Izquierda a Derecha (estilo "PASCAL")
  - PUSHeados de Derecha a Izquierda (estilo "C")
  - El invocante debe colocarlos en el STACK antes de la invocación de la subrutina y retirarlos del STACK luego del retorno de la misma.
- Ejemplo:
  - void display (word numero, byte base)
  - Invocante:

```
MOV AL, 0
PUSH AX ;base
PUSH [BX+SI]      ;numero
CALL display
ADD SP, 4         ;retira los parámetros
```

- Los parámetros tipo byte, son pasados en la parte baja de la word puesta en el stack
- Retirar los parámetros se puede hacer usando POP o incrementando SP la cantidad de bytes puestos

Assembler - Pasaje de Parametros

- El invocado debe indexar dentro del stack para hacer uso de los valores del parámetro, usando un STACK FRAME
- Un STACK FRAME es una vista consistente del stack justo antes de comenzar con el código ejecutable de la subrutina
  - Provee un método uniforme de acceso a los parámetros pasados en el STACK usando un direccionamiento BASE-INDIRECTO sobre BP mas allá del número de parámetros o/y del número de registros salvados/restoreados por la rutina.

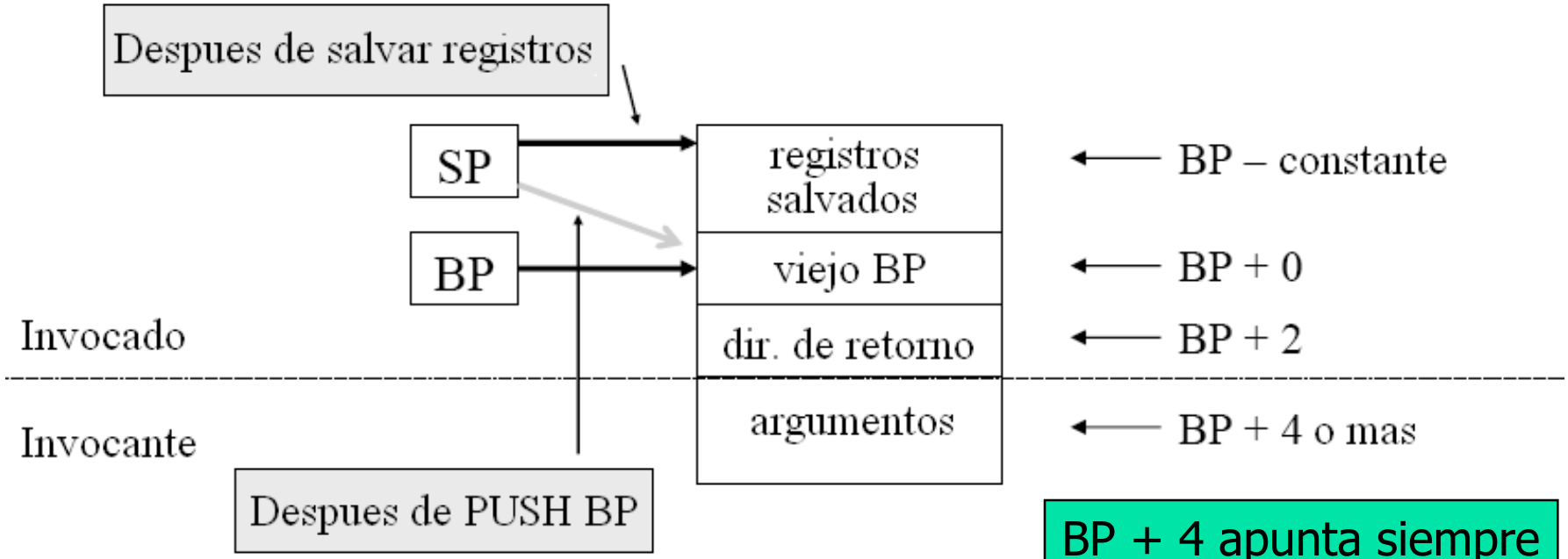
```
anySub proc
    PUSH BP
    MOV BP, SP
    ; PUSH los registros a utilizar
        ; Código ejecutable de la rutina donde se realiza el trabajo
    ; POP todos los registros salvados (en orden inverso!)
    POP BP
    RET
anySub endp
```

Código de Entrada estándar

Código de Salida estándar

Assembler - Pasaje de Parametros

- Esqueleto del Stack Frame asociado con la subrutina

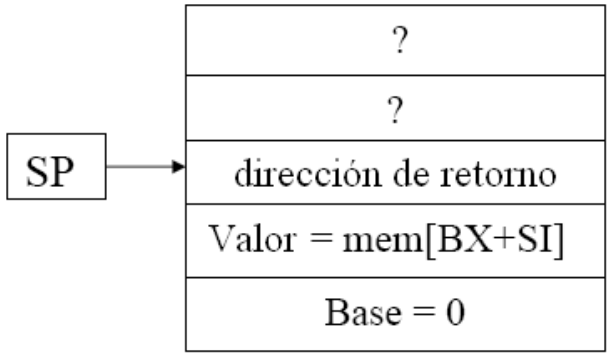


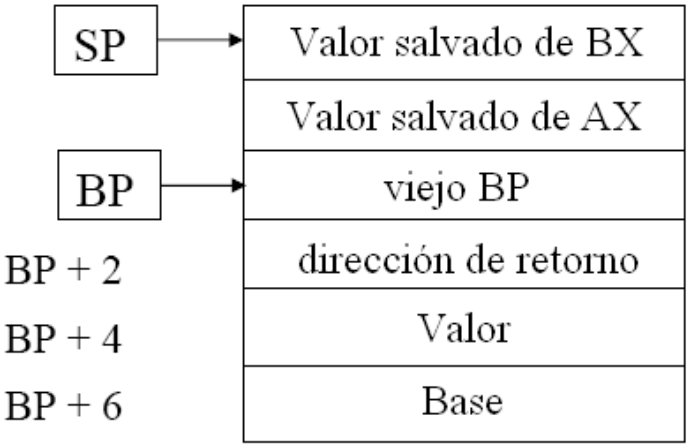
BP + 4 apunta siempre al argumento de mas a la izquierda en estilo "C"

Assembler - Pasaje de Parametros

- Ejemplo: **void display(word Valor, byte Base)**
- Seteo de la llamada (por el invocante)
  - MOV AL, 0 ; Base = binario
  - PUSH AX
  - PUSH [BX+SI] ; Valor a mostrar
  - CALL Display16
  - ADD SP, 4
- Implementación de la subrutina

```
display PROC
        PUSH BP
        MOV BP, SP
        PUSH AX
        PUSH BX
        MOV AX, [BP + 4]
        MOV BL, [BP + 6]
        CMP BL,0
        ...
        POP BX
        POP AX
        POP BP
        RET
display ENDP
```



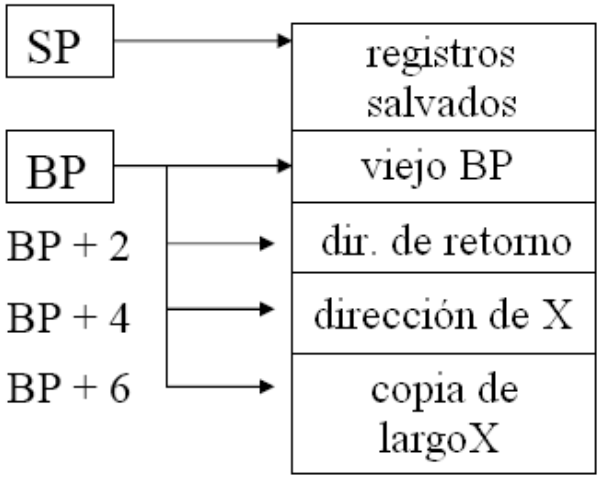


Assembler - por valor o referencia

- Definición: Parámetro POR VALOR
  - El argumento es una copia del valor de interés
  - En lenguje de alto nivel como C++, pasaje POR VALOR, es la forma de default de pasar variables simples (tipos primitivos: int, char, float)
  - Los parámetros pasados POR VALOR, dentro de la subrutina, pueden ser tratados coma variables locales
    - El contenido del stack puede ser leído y modificado
    - La variables es local y solo existe durante la ejecución de la subrutina
    - Los cambios en estos argumentos no son persistentes y no son vistos por el invocante.
- Definición: Parámetro POR REFERENCIA
  - El argumento es la dirección de la variable en memoria
  - Usado cuando se necesita acceder a las variables del invocante:
    - El propósito de la subrutina es modificar las variables del invocante
    - El pasaje de estructuras grandes requerirán mucho tiempo y espacio en el stack si se pasan POR VALOR.
  - En lenguajes de alto nivel como C, el pasaje por referencia hace necesario usar el operador &: (int & porferencia) (int porvalor)

Assembler - Por Referencia

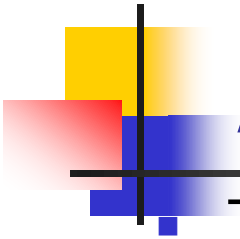
- Ejemplo: void SortArray(int &ordenar[], int largo)
  - ;declaración de Arreglo
  - X DW
    DW
    ...
    largoX DW



- Invocante:
  - PUSH largoX
    MOV AX, OFFSET X
    PUSH AX ; pasa la dir. de arreglo X
    CALL SortArray
    ADD SP, 4
- Invocado:
  - MOV BX, [BP + 4] ; obtiene la dir. del arreglo
    MOV SI, 0 ; índice del arreglo
    ...
    MOV AX, [BX + SI] ; accede al elemento del arreglo

Assembler - Valor de Retorno

- Las rutinas pueden retornar información al invocante de dos modos:
  - Retornar valores en las variables pasadas por referencia
  - Retornar un valor via el tipo de retorno de la subrutina
    - Ejemplo:
      boolean ValorAbs(int &X, int Y);
    - donde **boolean** es normalmente un byte con 0=false o 1=true
- Retornar el valor del tipo de retorno al invocante puede ser hecho por una de las tres formas usadas para pasar parámetros
  - Variables Globales (mismos problemas que antes)
  - En el stack
    - Por ejemplo, después de pasar los parámetros, el invocante puede alocar una palabra extra antes del CALL
      - SUB SP, 2
    - El invocado podría retronar el valor allí
  - Vía Registros
    - Hay un solo valor de retorno, por lo tanto necesita un solo registro
      - retornar 8 bits en AL
      - retornar 16 bits en AX
      - retornar 32 bits en DX:AX (igual que los 32 bits utilizados por DIV)



Assembler - Subrutinas

- Todos los ejemplos de subrutinas vistos usan control de flujo intra-segmento
  - Solo IP es salvado/cambiado/restablecido
  - A estas subrutinas se las llama NEAR
- En programas largos y en bibliotecas, las subrutinas pueden ser ubicadas en distintos segmentos de código
  - Requiere un control de flujo inter-segmento
  - CS e IP son salvados/cambiados/restablecidos
  - A esas subrutinas se las llama FAR
- La directiva PROC usa un modificador opcional para denotar el tipo de control de flujo
  - PROC NEAR o PROC FAR
  - Por default, sin algún modificador, una subrutina es NEAR

Assembler - Subrutina

```
0000                    .code
0000                    main PROC
0000  E8 0004            CALL  nearsub
0003  0E E8 0001          CALL  farsub
0007                    main ENDP

                        ; NEAR subroutine
0007                    nearsub PROC NEAR
0007  C3                 RET
0008                    nearsub ENDP

                        ; FAR subroutine
0008                    farsub PROC FAR
0008  CB                 RET
0009                    farsub ENDP
                        END main
```

E8 → CALL

0E → PUSH CS

C3 → RET (NEAR)

Intra-segment

CB→ RET (FAR)

Inter-segment

C y Assembler

- Cómo empezar y terminar una función en assembler

```
funcion:
    push ebp                ; salva la base de la pila anterior
    mov ebp, esp            ; reposiciona la nueva base en el tope de la pila
    sub esp, 8              ; reserva 2 dword de espacio para variables locales
    ...
    pop ebp                 ; recupera la base anterior
    ret
```

- Cómo usar desde C una función escrita en assembler
  - *Desde C:* extern <prototipo de la función>;
    - **Ejemplo:** extern void funcionAsm(int n);
  - *Desde Assembler:* global <defínción del símbolo>
    - **Ejemplo:**

```
global funcionAsm
...
funcionAsm:
                ; aquí va el código de la función en asm
ret
```

- Cómo usar una función C desde assembler
  - *Desde Assembler:* extern <funcionC>;
    - **Ejemplo:**

```
extern funcionC
...
call funcionC:
```

C y Assembler

- Ejemplo:

Archivo .c

```
extern int funcionAsm(char* param1, int param2);

int main(void){
        return  funcionAsm("Orga",2);
}
```

Archivo .asm

```
global funcionAsm
extern printf

section .data
formato db '%s %d',10,0

section .text
funcionAsm:
        push ebp
        mov ebp,esp
        push esi
        push edi
        push ebx
        push dword [ebp+12]  ;param2
        push dword [ebp+8]   ;param1
        push dword formato
        call printf
        add esp,12
        xor eax,eax          ;devuelve 0
        pop ebx
        pop edi
        pop esi
        pop ebp
        ret
```