El stack

- Stack en Z80
  - Palabra 16 bits, “crece” hacia abajo, SP → último byte usado.
  - No inicializado en reset.
  - Prever bloque RAM asignado a stack.

```
LD SP, STACKTOPE+1
```

- Anidamiento
  - de CALL y de PUSH
  - No debe salir de bloque asignado, caso contrario
    - Sobreescribo variables o código de programa
    - “Almaceno” en ROM
    - “Almaceno” en áreas vacías del espacio de memoria

Comunicación
Prog. Principal - Subrutina

- Comunicación de:
  - Datos de entrada (parámetros o argumentos).
  - Resultados.
- Secuencia de llamada: no solo CALL
  - Eventualmente preservar registros
  - Preparación de los parámetros
  - Instrucción CALL
  - Al retorno, obtención de resultado
  - Arreglar todo

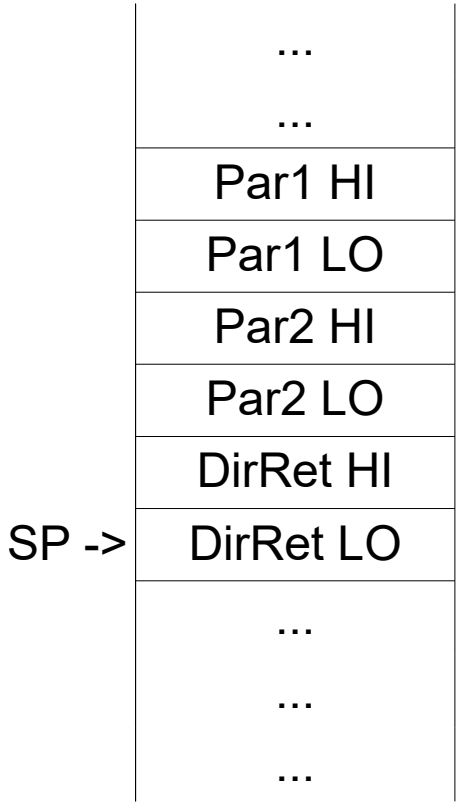
Subrutinas
Pasaje de parámetros

- Alternativas
  - Registros internos.
  - Posiciones de memoria reservadas (variable global).
  - Stack.
  - Punteros a zonas de memoria.
  - Otras.

Pasaje de parámetros en Stack

- Programa llamante:

```
    ld bc, PAR1
    push bc
    ld bc, PAR2
    push bc
    call sub1
DirRet:
    ...
```

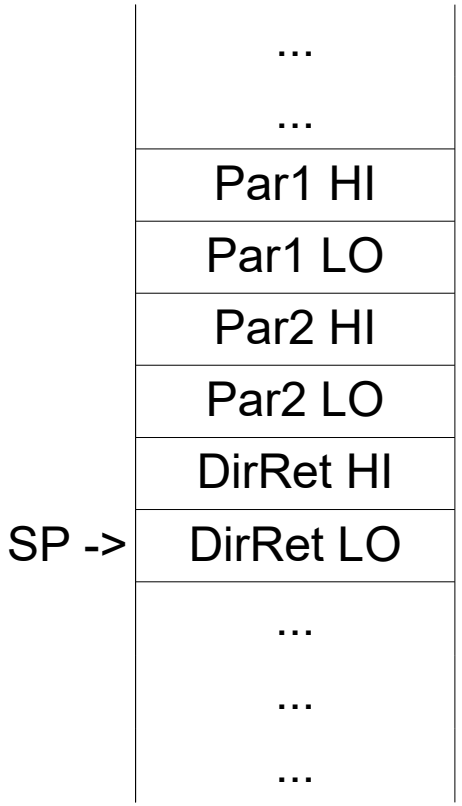


Pasaje de parámetros en Stack

- Métodos para obtener parámetros
  - Método 1: POP y PUSH
  - Método 2: Instrucción EX (SP), HL
  - Método 3: Direccionamiento al stack

Método 1: POP y PUSH

```
subrut:
    POP HL  ; dir retorno
    POP BC  ; param2
    POP DE  ; param1
    PUSH HL ; repongo dir ret
    ...
    ...
    RET
```



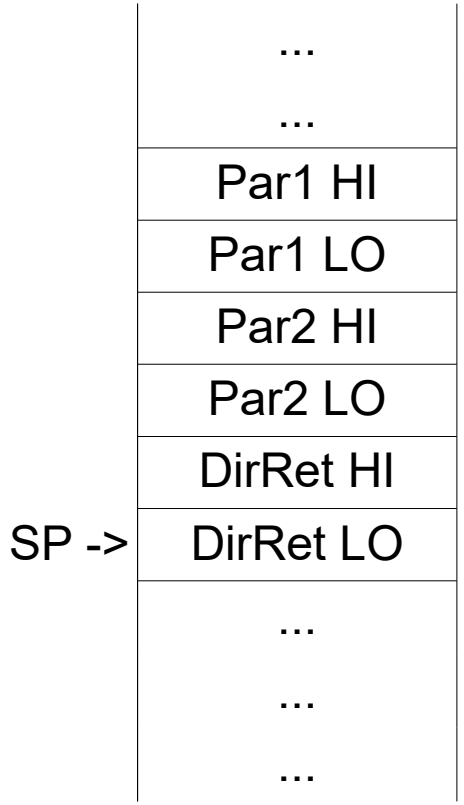
- Problema:

  no se puede preservar registros

Método 2: EX (SP), HL

- Subrutina:

```
POP HL          ; dir ret a HL

EX (SP), HL     ; sale param2
                ; entra dirret

POP IX          ; dir reta IX

EX (SP), IX     ; sale param1
                ; entra dirret

...

RET
```

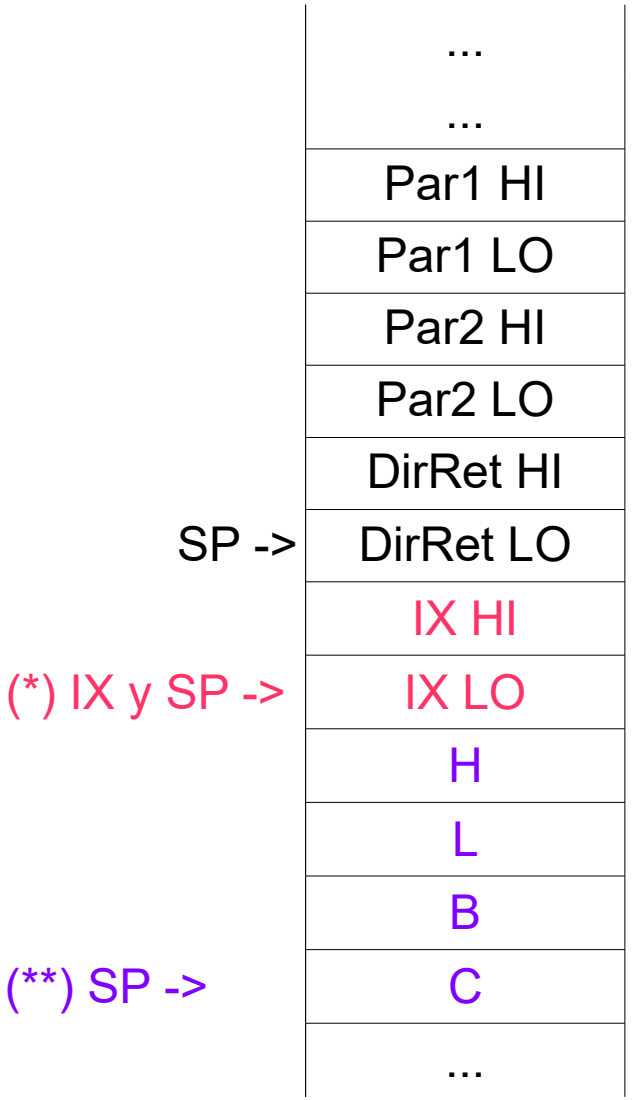


- Cómodo si necesito los parámetros de a uno por vez

Método 3: Direccionamiento indirecto al Stack

- Subrutina:

```
PUSH IX
LD IX, 0
ADD IX, SP ;  IX < - SP (*)
PUSH HL
PUSH BC    ;            (**)
LD HL, (IX+4) ; par2
...
POP BC
POP HL
POP IX
RET
```



Método 3: Direccionamiento indirecto al Stack

- Subrutina:

```
PUSH IX
LD IX, 0
ADD IX, SP ;  IX < - SP (*)
PUSH HL
PUSH BC
LD HL, (IX+4) ; par2
...
POP BC
POP HL
POP IX
RET
```

Cómo queda el stack después de RET?

- En el programa llamante se debe realinear el stack después de retornar de call

```
ld bc, PAR1
push bc
ld bc, PAR2
push bc
call sub1
pop bc
pop bc
```

Ubicación de variables en memoria

- En lenguajes como C y Pascal
- Variables Globales:
  - Existen durante toda la ejecución del programa
  - En direcciones reservadas de memoria
  - Equivalen en assembler a directivas DB con etiqueta
- Variables locales
  - Se crean en cada invocación de la función, desaparecen al retornar.
  - Área de memoria independiente para cada invocación
  - Se ubican en el stack

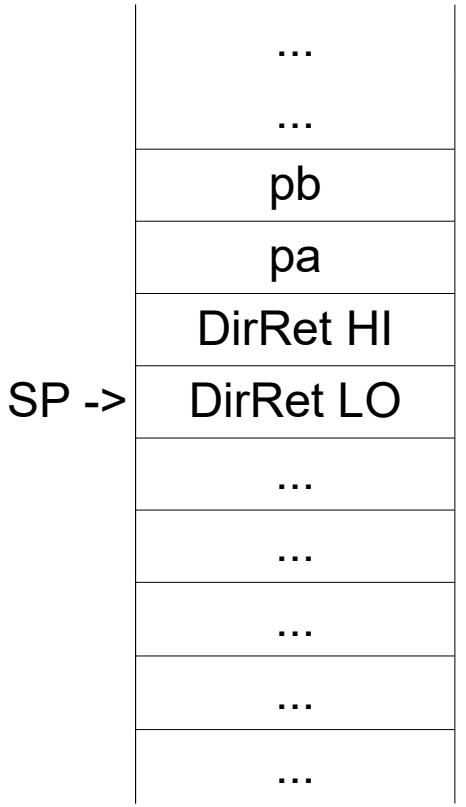
- Programa en C:

```
char mifuncion(char pa, char
pb){
    char lvector[10];
    lvector[0] = pa;
    lvector[9] = pb;

    return( pa + pb );
}

int gvector[10];

main(){
 gvector[0] = 0x5555;
 mifuncion(2,5);
 return( gvector[0] );
}
```

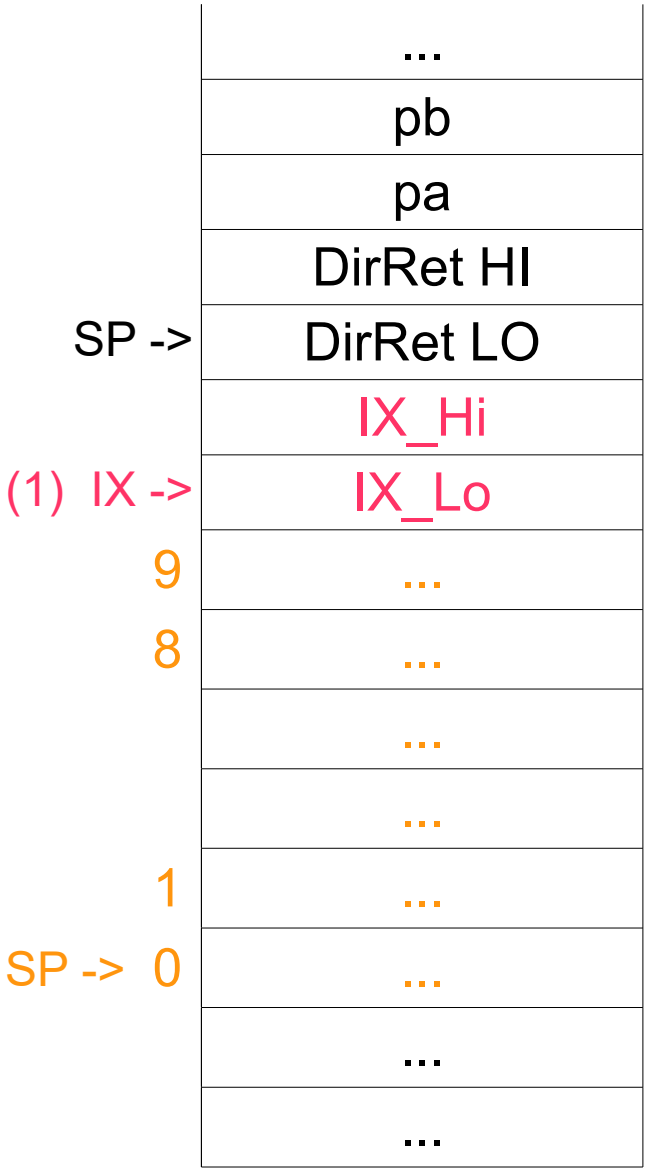
- Programa compilado

```
    .area _DATA
_gvector::
    .ds 20
    .area _CODE
;sdcc_local.c:1: char
; mifuncion(char pa, char pb){
_mifuncion:
    push  ix
    ld ix,#0
    …
    …
;sdcc_local.c:11: main(){
_main:
;sdcc_local.c:12:
;     gvector[0] = 0x5555;
    ld hl,#_gvector
    ld (hl),#0x55
    …
    ret
```

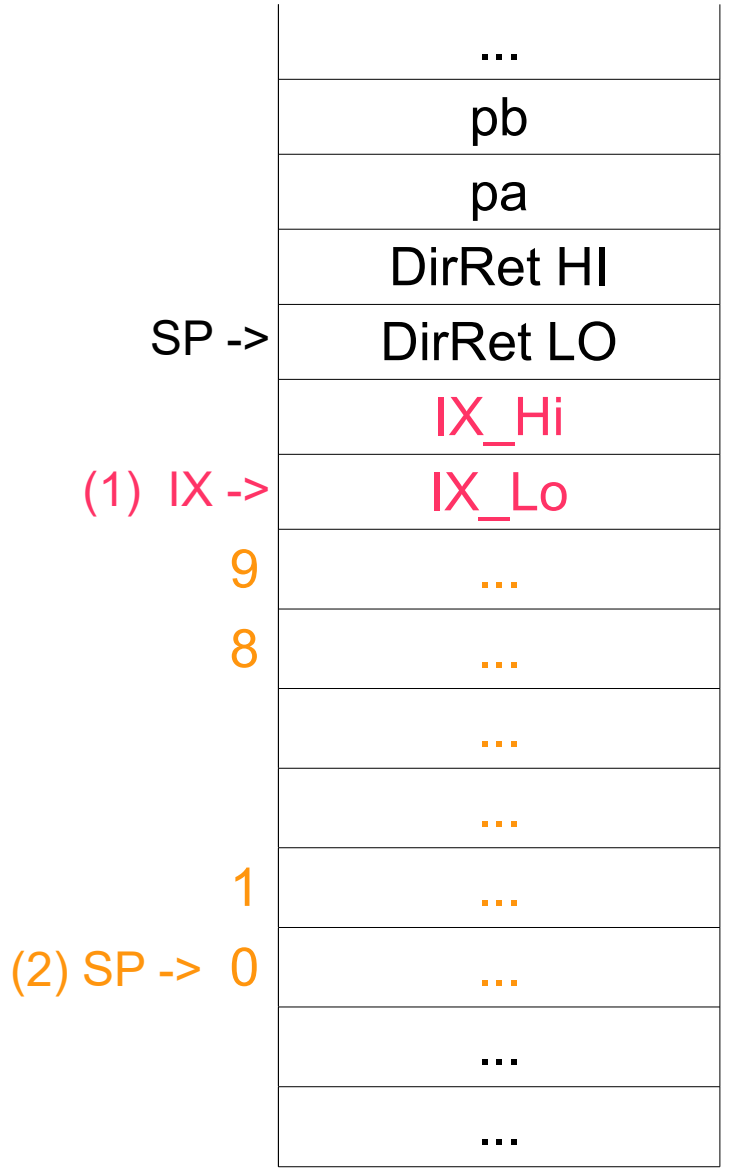
```
_main:
;sdcc_local.c:12: gvector[0] = 0x5555;
   ld hl,#_gvector
   ld (hl),#0x55
   inc  hl
   ld (hl),#0x55
;sdcc_local.c:13: mifuncion(2,5);
   ld hl,#0x0502
   push  hl
   call  _mifuncion
   pop   af
;sdcc_local.c:14: return(gvector[0]);
   ld hl,#_gvector
   ld c,(hl)
   inc  hl
   ld b,(hl)
;   reg = bc
   ld l,c
   ld h,b
   ret
```



```
_mifuncion:
   push  ix
   ld ix,#0
   add   ix,sp    ; (1)
   ld hl,#-10
   add   hl,sp
   ld sp,hl       ; (2)
    . . .
    . . .
;sdcc_local.c:6: return(pa+pb);
   ld a,4(ix)     ; ld a, (ix+4)
   add   a,5(ix)  ; add a, (ix+5)
   ld l,a
;  genRet
   ld sp,ix
   pop   ix
   ret
```



```
_mifuncion:
   . . .
;sdcc_local.c:3: lvector[0] = pa;
   ld hl,#0x0000
   add  hl,sp
   ld c,l
   ld b,h
;  genAssign (pointer)
   ld a,4(ix)
   ld (bc),a
;sdcc_local.c:4: lvector[9] = pb;
   ld a,c
   add  a,#0x09
   ld c,a
   ld a,b
   adc  a,#0x00
   ld b,a
;  genAssign (pointer)
   ld a,5(ix)
   ld (bc),a
;sdcc_local.c:6: return( pa + pb );
   . . .
```



Reentrancia

- Reentrancia:
  - Nueva invocación antes de retornar de invocación anterior.
  - A propósito para algoritmos recursivos
  - No buscado cuando hay procesos concurrentes (p. ej. interrupciones)
- Problema:
  - Uso de memoria reservada para almacenamiento de parámetros o resultados intermedios
    - La segunda invocación destruye datos de la primera
    - Estructuras de datos con contenido incoherente
- Soluciones
  - Área de memoria independiente para cada invocación
  - Deshabilitar interrupciones

Reentrancia

- Área de memoria independiente...
  - Opción 1 (no siempre posible)
    - Dos copias de la subrutina (y de sus variables), una invocada desde programa principal y otra desde interrupciones.
  - Opción 2
    - Área de memoria local para cada invocación
    - En el stack
    - Es lo que hace un compilador C o Pascal para variables locales de una función.
  - Deshabilitar interrupciones