

Estructura de computadores

Programación en Ensamblador

LENGUAJE MÁQUINA

- Juego de instrucciones. Formatos
- Tipos de datos
- Modos de direccionamiento
- Tipos de instrucciones

ARQUITECTURA DEL 88110

- Banco de Registros
- Memoria principal
- Modos de direccionamiento
- Juego de instrucciones

LENGUAJE ENSAMBLADOR

- Sintaxis
- Mnemónicos y etiquetas
- Instrucciones y pseudoinstrucciones
- Macros

PROGRAMACIÓN EN ENSAMBLADOR

- Estructuras de datos: - Vectores y Matrices
 - Listas
- Subrutinas: - Paso de parámetros
 - Marco de pila
 - Recursividad

Problemas y tutorías

Programación en Ensamblador

Documentación

1. Transparencias del tema (web)
2. Descripción del emulador 88110 (web+publicaciones)
3. Subrutinas: paso de parámetros y marco de pila (web+publicaciones)
4. Enunciados de problemas (web+publicaciones) Moodle

Fundamentos de los computadores

Pedro de Miguel, Paraninfo/Thomson-2006 (capítulo 13)

Estructura de computadores: problemas y soluciones

García Clemente y otros, RAMA-2000 (capítulo 2)

Estructura de computadores: problemas resueltos

García Clemente y otros, RAMA-2006 (capítulo 3)

Solución de problemas:

<http://www.datsi.fi.upm.es/88110>

Instrucciones y direccionamientos

- Genérico:
 - Direccionamiento a byte
 - Ordenación de bytes: Little endian, big endian
 - Máquinas de 1 dir, 2 dir, 3 dir
 - Modelos de ejecución
 - IEEE
 - Tipos de Instrucciones
 - Modos de direccionamiento
- Procesador concreto: 88110 (simulador)

ARQUITECTURA 88110

Procesador

- Máquina de **3 direcciones** (0: add 1: add B 2: add A,B)

add rD, rS1, rS2
 $rD \leftarrow rS1 + rS2$

- Modelo de ejecución **registro-registro** (ALU, no jmp/ld/st)

Destino: { Registro }

Origen: { Registro + Registro
Registro + Inmediato }

- Palabra de **32 bits** (direccionable **a byte**)
- ALU opera en **complemento a 2**
- Emulador: ejecución serie / superescalar

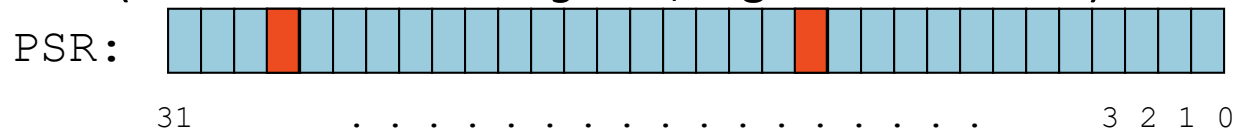
ARQUITECTURA 88110

Banco de Registros

- Banco general de **32 registros**: r0 .. r31
 - r0** cableado a 0 (siempre tiene valor 0)
 - r1** guarda dirección de retorno de subrutina
 accesibles por pares en operaciones de 64 bits

- PC**

- PSR** (*Processor Status Register*, registro de estado)



bit 12 → *overflow* en operaciones con enteros (OVF)

si OVF==1 → no se modifica rD

bit 28 → acarreo (*carry – borrow*)

- Registros que decidimos reservar para función particular:

r30 puntero de pila y **r31** puntero de marco de pila

ARQUITECTURA 88110

Memoria principal

- Almacena: Instrucciones + Datos
- Direccionable a nivel de **byte**
1 palabra \rightarrow 4 direcciones de memoria
- Bus de direcciones de **32 bits**
Máxima capacidad (teórica) $\rightarrow 2^{32}$ bytes = 4 GB
 $\left\{ \begin{array}{l} 0x00000000 \\ 0xFFFFFFFF \end{array} \right.$
- Capacidad del emulador:
 2^{18} direcciones = 2^{18} bytes = 256 KB
 $\left\{ \begin{array}{l} 0x00000000 \\ 0x0003FFFF \end{array} \right.$

ARQUITECTURA 88110

Modos de direccionamiento

- **Sí tiene:**

- Directo a registro: .Ri
- Inmediato: #aaaa
- Relativo a registro base: #desp[.Ri]
- Relativo a PC: \$xx
- Indirecto a registro: [.Ri]

- **NO tiene:**

- ~~• Absoluto: /dir~~
- ~~• Relativo a registro índice: ++, --~~
- ~~• Indirecto a memoria: [/dir]~~
- ~~• Relativo a pila: push / pop~~

ARQUITECTURA 88110

Direccionamiento directo a registro

`add r1, r2, r3 ; r1 \leftarrow r2 + r3`

equivalente en máquina de 2 direcciones, IEEE 694:

`ADD .R7, .R9 ; (add r7, r7, r9)`

ARQUITECTURA 88110

Direccionamiento inmediato

- Puede ser con/sin signo, ambos de 16 bits:

Con signo: SIMM16

Sin signo: IMM16

- Se puede expresar en decimal o hexadecimal

- Ejemplo:



add r1, r2, -13;

13 en binario: 0000 1101

-13 en binario: - 0000 1101

-13 en binario: 1111 0011

add r1, r2, 0xFFF3 { +   r2
+ FFFF FFF3 → r1

addu r1, r2, 0xFFF3 { +   r2
+ 0000 FFF3 → r1

ARQUITECTURA 88110

Direccionamiento relativo reg. base

- Ejemplo en formato del estándar IEEE:

LD .R1, #13[.R4] R1 ← MEM(R4+13)

- Ejemplo en formato de 88110 (desplazamiento inmediato):

ld r1, r4, 13 r1 ← MEM(r4+13)

st r1, r4, 13 r1 → MEM(r4+13)

- Ejemplo en formato de 88110 (desplazamiento en registro):

ld r1, r4, r10 r1 ← MEM(r4+r10)

st r1, r4, r10 r1 → MEM(r4+r10)

ARQUITECTURA 88110

Direccionamiento relativo a PC

- Ejemplo:

```
br 7 ; PC ← PC + 7*4
(desplazamiento: 26 bits / 16 bits)
```

- Ejemplo en formato del estándar IEEE:

```
ADD .7, .5
BR $desp ; PC ← et1+desp
et1: LD .1, [.7]
```

- Ejemplo en formato de 88110:

```
add r7, r7, r5
et0: br D ; PC ← et0 + 4*D
ld r1, r7, 0
```

ARQUITECTURA 88110

Direccionamiento indirecto a registro

En el 88110 solo existe en los saltos:

- Ejemplo en formato del estándar IEEE:

JMP [.R10] ; PC \leftarrow R10

- Ejemplo en formato del 88110:

jmp (r10) ; PC \leftarrow r10

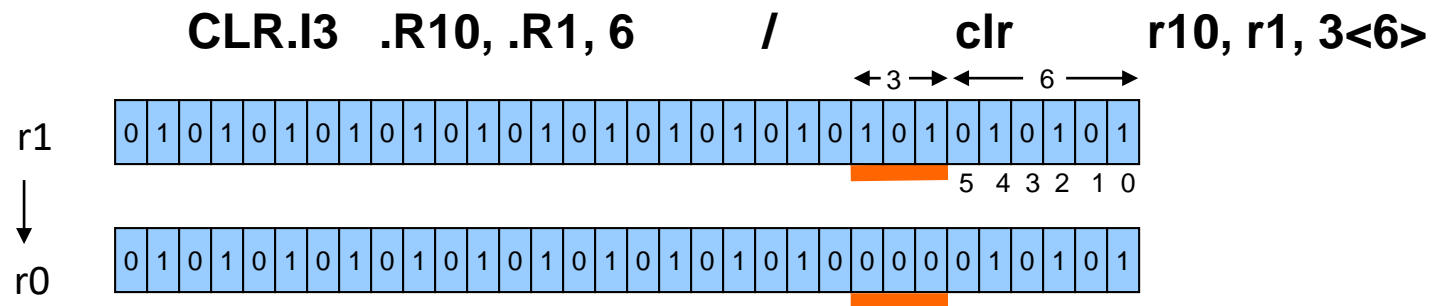
ARQUITECTURA 88110

Direccionamiento campos de bit

En ciertas instrucciones del 88110 se pueden seleccionar:

- bits individuales
- campos de bit

- Ejemplo en formato del estándar IEEE / 88110:



- Otro ejemplo (bits individuales):

bb0 3, r8, 7 ; Si (bit3 de r8) == 0 → PC ← PC + 4*7

ARQUITECTURA 88110

Juego de instrucciones

Tipos de instrucciones en el 88110:

- Lógicas (`or`, `and`, `xor`, `mask`)
- Aritméticas (`add`, `sub`, `addu`, `subu`, `muls`, `mulu`, `divs`, `divu`, `cmp`)
- Bifurcaciones (`bb0`, `bb1`, `br`, `bsr`, `jmp`, `jsr`)
- Transferencia (`ld`, `st`, `ldcr`, `stcr`, `xmem`)
- Campos de bit (`clr`, `set`, `ext`, `extu`, `mak`, `rot`)
- Coma flotante (`fadd`, `fsub`, `fmul`, `fdiv`, `fcvt`, `flt`, `int`, `fcmp`)

Instrucción específica del emulador: `stop`

ARQUITECTURA 88110

Instrucciones lógicas

Lógicas: `or`, `and`, `xor`, `mask`

INST	Operandos	Ext
<code>or</code>	<code>rD, rS1, rS2</code>	<code>c</code>
<code>and</code>	<code>rD, rS1, IMM16</code>	<code>u</code>
<code>xor</code>		
<code>mask</code>		

`c`: complemento a 1 de `rS2`

`u`: Opera con los 16 bits más significativos de `rS1`

(NOTA: en las operaciones lógicas con IMM16, el dato inmediato se extiende con `0x0000` para `or`, `xor` y `mask` y con `0xFFFF` para la instrucción `and`)

ARQUITECTURA 88110

Instrucciones aritméticas (I)

Aritméticas: **add**, **sub**, **addu**, **subu**, **muls**, **mulu**, **divs**, **divu**, **cmp**

INST	Operandos	Ext	} causan excep. overflow (OVF)
add	rD, rS1, rS2	ci,co,cio	
sub	rD, rS1, SIMM16		
addu	rD, rS1, rS2	ci,co,cio	
subu	rD, rS1, IMM16		

ci: opera con acarreo de entrada (bit28 de PSR)

co: actualiza el flag de acarreo (bit28 de PSR)

cio: equivale a usar ci+co

ARQUITECTURA 88110

Instrucciones aritméticas (II)

Aritméticas: **add**, **sub**, **addu**, **subu**, **muls**, **mulu**, **divs**, **divu**, **cmp**

muls **rD, rS1, rS2** ; excep. OVF

mulu { **rD, rS1, rS2**
 rD, rS1, IMM16

mulu.d **rD, rS1, rS2** ; d.p. en rD

divs { **rD, rS1, rS2** ; excep. div por 0
 rD, rS1, SIMM16

divu { **rD, rS1, rS2** ; excep. div por 0
 rD, rS1, IMM16

divu.d **rD, rS1, rS2** ; d.p. en rS1 y rD

ARQUITECTURA 88110

Instrucciones aritméticas (III)

Aritméticas: add, sub, addu, subu, muls, mulu, divs, divu, **cmp**

cmp $\begin{cases} \text{rD}, \text{rS1}, \text{rS2} \\ \text{rD}, \text{rS1}, \text{SIMM16} \end{cases}$

rD	nh	he	nb	be	hs	lo	ls	hi	ge	lt	le	gt	ne	eq	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(resto de bits a '0')

eq: 1 si y solo si $\text{rS1} = \text{rS2}$

ne: 1 si y solo si $\text{rS1} \neq \text{rS2}$

gt: 1 si y solo si $\text{rS1} > \text{rS2}$ (con signo)

.

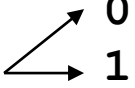
hi: 1 si y solo si $\text{rS1} > \text{rS2}$ (sin signo)

.

ARQUITECTURA 88110

Bifurcaciones/saltos

Bifurcaciones (bb0, bb1, br, bsr, jmp, jsr)

INST	Operandos	
bb0	B, rS1, D16	$PC \leftarrow PC + 4 * D16$
bb1	B, rS1, D16	si bit B de rS1 = 
br	D26	$PC \leftarrow PC + 4 * D26$
bsr	D26	$r1 \leftarrow PC + 4 ; PC \leftarrow PC + 4 * D26$
jmp	(rS1)	$PC \leftarrow rS1$ (alineado)
jsr	(rS1)	$r1 \leftarrow PC + 4 ; PC \leftarrow rS1$ (alineado)

ARQUITECTURA 88110

Transferencia (memoria)

Transferencia (**ld**, **st**, **ldcr**, **stcr**, **xmem**)

INST	Operandos	Ext. / explicación
ld	rD, rS1, SIMM16	b,bu h,hu
	rD, rS1, rS2	d
st	rD, rS1, SIMM16	b,h,d
	rD, rS1, rS2	
ldcr	rD	rD ← PSR
stcr	rS1	PSR ← rS1
xmem	rD, rS1, rS2	rD ↔ MEM(rS1+rS2)

ARQUITECTURA 88110

Campos de bit

Campos de bit (`clr`, `set`, `ext`, `extu`, `mak`, `rot`)

INST	Operandos
<code>clr</code>	
<code>set</code>	<code>rD, rS1, W5<O5></code>
<code>ext</code>	
<code>extu</code>	<code>rD, rS1, rS2</code>
<code>mak</code>	
<code>rot</code>	<code>rD, rS1, <O5></code> <code>rD, rS1, rS2</code>

ARQUITECTURA 88110

Instrucciones de coma flotante

Coma flotante (**fadd**, **fsub**, **fmul**, **fdiv**, **fcvt**, **flt**, **int**, **fcmp**)

INST	Operandos	explicación
fadd.xxx fsub.xxx fmul.xxx fdiv.xxx	rD, rS1, rS2	x=s \bar{x}=d x=d \bar{x}=s
fcvt.x\bar{x} flt.xs int.sx	rD, rS2	
fcmp.sxx	rD, rS1, rS2	

Pueden generar excepciones: Overflow/Underflow/NaN/Div0

ARQUITECTURA 88110

Ensamblador/Cargador

- **Ensamblador:** Programa que se encarga de “traducir” un programa escrito en lenguaje ensamblador a lenguaje máquina.

etiqueta: instruccion_ensamblador ; Comentarios

- **Instrucción_ensamblador:** Puede ser una instrucción-máquina o una pseudoinstrucción.
- Pseudoinstrucción:
 - Instrucción para el programa ensamblador.
 - No se traduce en una instrucción en memoria.
 - Indica al ensamblador cómo debe generar el código-máquina.

ARQUITECTURA 88110

Pseudoinstrucciones

- **Org n:** Indica que el código que le sigue se almacene en la posición de memoria n.
- **Res n:** Indica que se reserven n bytes en memoria. N debe estar alineado a palabra.
- **Data a, b, c,:** Inicializa las posiciones de memoria con los valores a, b y c.
- **Data “texto”:** Inicializa las posiciones de memoria con la cadena de bytes texto. Asegura que la siguiente palabra en memoria está alineada (véase ejemplo).
- **Low(etiqueta o inmediato):** Devuelve los 16 bits menos significativos de la dirección asociada a la etiqueta o dato inmediato.
- **High(etiqueta o inmediato):** Devuelve los 16 bits más significativos de la dirección asociada a la etiqueta o dato inmediato.

ARQUITECTURA 88110

Ejemplo “data” (1)

```
INI:      ld      r3, r0, 400          ; "data.ens"
          or      r2,r0,low(numeros)
          or.u    r2,r2,high(numeros)
          stop
          org     400
          data    0x01020304

          org     412
          res     4
          data    "SSOO"
          data    "1234567890abcdefgh\n\0\t"
numeros:  data    15, 0x7AF, -5
```

```
practica@avellano% 88110e -o data.bin data.ens
88110.ens-INFO: Compilando data.ens ...
88110.ens-INFO: Compiladas 12 lineas
88110.ens-INFO: GenerandoCodigo...
88110.ens-INFO: Programa generado correctamente
practica@avellano%
```

ARQUITECTURA 88110

Ejemplo “data” (2)

```
practica@avellano% mc88110 data.bin
```

```
PC=0          ld          r03,r00,400          Tot. Inst: 0      ; ; Ciclo : 1
FL=1 FE=1 FC=0 FV=0 FR=0
R01 = 00000000 h R02 = 00000000 h R03 = 00000000 h R04 = 00000000 h
R05 = 00000000 h R06 = 00000000 h R07 = 00000000 h R08 = 00000000 h
R09 = 00000000 h R10 = 00000000 h R11 = 00000000 h R12 = 00000000 h
R13 = 00000000 h R14 = 00000000 h R15 = 00000000 h R16 = 00000000 h
R17 = 00000000 h R18 = 00000000 h R19 = 00000000 h R20 = 00000000 h
R21 = 00000000 h R22 = 00000000 h R23 = 00000000 h R24 = 00000000 h
R25 = 00000000 h R26 = 00000000 h R27 = 00000000 h R28 = 00000000 h
R29 = 00000000 h R30 = 00000000 h R31 = 00000000 h
88110>
```

ARQUITECTURA 88110

Ejemplo “data” (3)

88110> e

Fin ejecución

PC=16 instrucción incorrecta Tot. Inst: 4 ¡¡ Ciclo : 62

FL=1 FE=1 FC=0 FV=0 FR=0

R01 = 00000000 h R02 = 000001BC h R03 = 01020304 h R04 = 00000000 h
 R05 = 00000000 h R06 = 00000000 h R07 = 00000000 h R08 = 00000000 h
 R09 = 00000000 h R10 = 00000000 h R11 = 00000000 h R12 = 00000000 h
 R13 = 00000000 h R14 = 00000000 h R15 = 00000000 h R16 = 00000000 h
 R17 = 00000000 h R18 = 00000000 h R19 = 00000000 h R20 = 00000000 h
 R21 = 00000000 h R22 = 00000000 h R23 = 00000000 h R24 = 00000000 h
 R25 = 00000000 h R26 = 00000000 h R27 = 00000000 h R28 = 00000000 h
 R29 = 00000000 h R30 = 00000000 h R31 = 00000000 h

88110> v 400

400	04030201	00000000	00000000	00000000
416	53534F4F	31323334	35363738	39306162
432	63646566	67680A00	09000000	0F000000
448	AF070000	FBFFFFFF	00000000	00000000
464	00000000	00000000	00000000	00000000

88110>

ARQUITECTURA 88110

Macroinstrucciones (“macros”)

- Conjunto de sentencias a las que se le asigna un nombre y se les pasa un conjunto de argumentos.
- Cuando aparece la invocación de la macro **se sustituye en fase de ensamblado** la macro por el conjunto de sentencias declarado en la macro cambiando los parámetros declarados por los que se pasan en la invocación.

```
Nombre_de_macro: MACRO(arg1, arg2, ..., argn)
    Conjunto de instrucciones
    Que componen la macro
ENDMACRO
```

```
swap: MACRO(ra,rb)
    or r1,ra,ra
    or ra,rb,rb
    or rb,r1,r1
ENDMACRO
```

ARQUITECTURA 88110

Macroinstrucciones (“macros”) II

- Una macro debe haberse definido **previamente**.
- Se permiten macros **anidadas**.
- **No** se permite la definición de **etiquetas** dentro de una macro.
- Se utilizan para **encapsular** pequeños fragmentos de código para los que no merece la pena construir una subrutina.

ESTRUCTURAS DE DATOS

Vectores: Ej. 1. Suma de los elementos
de un vector

ESTRUCTURAS DE DATOS

Matrices: Ej. 2. Suma de los elementos
cada columna de la matriz

ESTRUCTURAS DE DATOS

Listas: Ej. 3. Inserción de un nuevo elemento en una lista compacta y ordenada

ESTRUCTURAS DE DATOS

Listas: Ej. 4. Inserción de un nuevo elemento en una lista encadenada y ordenada

Subrutinas

- Parte de **código cerrado**, con especificación bien definida, que se puede utilizar desde varios puntos de un programa o diferentes programas.
- Una vez ejecutado el código de la subrutina se debe **retornar** “al lugar desde el que se llamó”.
- **Activación** de la subrutina:
 - Paso de parámetros.
 - Salvaguarda de registros. Habitualmente lo realiza el programa llamante.
 - Salvaguarda del PC
 - Bifurcación.

Subrutinas

- Tipos de variables que utiliza una subrutina:
 - Variables **globales**: Se crean cuando arranca el programa y tienen validez durante toda la vida del mismo. Cualquier subrutina puede acceder a estas variables.
 - Variables **locales** a una subrutina: Se crean cada vez que se activa la subrutina y se destruyen cuando se finaliza cada ejecución. Solo la subrutina que las crea puede acceder a ellas.
- Parámetro: Dato de entrada/salida de una subrutina que es necesario para su operación:
 - Por **valor**: Se pasa el dato necesario para su operación.
 - Por **dirección**: Se pasa la dirección de memoria en la que está contenido el dato/resultado necesario para la operación.

Subrutinas: Paso de parámetros

En registros

El llamante y la subrutina “acuerdan” un conjunto de registros de propósito general para intercambiar los datos y resultados.

- Rápido
- Limitado en el tipo y el número de parámetros.

– Llamante

```
ld r2, r20, 0
```

```
ld r3, r20, 4
```

```
bsr suma
```

Subrutina

```
suma: add r2, r2, r3
```

```
      jmp (r1)
```

Subrutinas: Paso de parámetros

En variables globales

Llamante

```
num1:    res 4
num2:    res 4
bsr suma
```

Subrutina

```
suma:    or r20, r0, low(num2)
          or.u r20,r20,high(num2)
          ld r5,r20,0
          or r20, r0, low(num1)
          or.u r20,r20,high(num1)
          ld r6,r20,0
          add r5, r5, r6
          st r5, r20, r0
          jmp(r1)
```

Subrutinas: Paso de parámetros

En variables globales

- El llamante y la subrutina “acuerdan” un conjunto de variables globales, puesto que son visibles desde todas las subrutinas, para intercambiar los datos y resultados.
 - Sencillo.
 - Limitado en subrutinas de librería y genera problemas de reentrancia

Subrutinas: Paso de parámetros

En la pila

- Resuelve las limitaciones que tienen los sistemas anteriores.
- El llamante introduce los parámetros mediante **PUSH**
- Ejecuta la llamada a subrutina (**CALL o bsr**)
- La subrutina recoge los parámetros **accediendo a la pila utilizando direccionamiento relativo a registro base.**
- Si la **dirección de retorno** se almacena en la pila, se debe asegurar que al realizar el retorno, está **en la cima de la pila.**

Subrutinas: Paso de parámetros

En la pila

Llamante

```
RESULT: RES 4
parámetro
...
PUSH .R1
PUSH .R2
LD .R1, #RESULT
PUSH .R1
CALL /SUMA
```

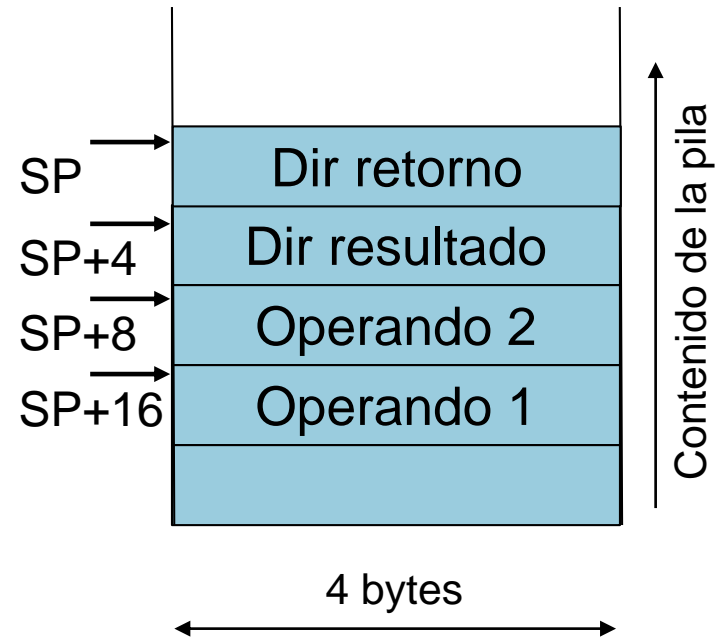
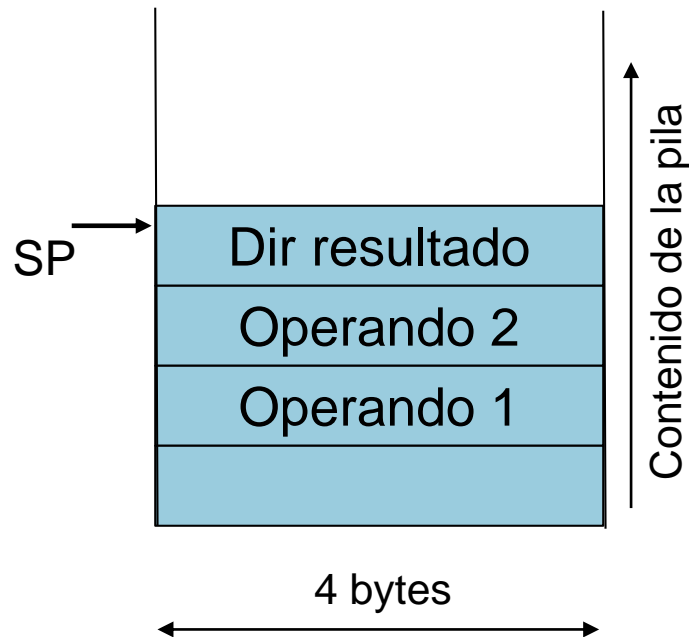
Subrutina

```
SUMA: LD .R5, #8[.SP] ;

LD .R6, #12[.SP] ; parámetro
ADD .R5, .R6
LD .R6, #4[.SP]
ST .R5, [.R6] ; resultado
RET
```


Subrutinas: Paso de parámetros

En la pila



88110: Gestión de la pila

- 88110 tiene **limitaciones**:
 - No tiene puntero de pila, por tanto **no tiene PUSH ni POP**: el número y tipos de parámetros es limitado.
 - Almacena la dirección de retorno en **r1**: el número de llamadas anidadas es limitado.
- **Soluciones**:
 - Asignar registro de propósito general como **puntero de pila: r30**.
 - Crear dos macros: **PUSH** y **POP**
- El **protocolo** que se muestra en la siguiente transparencia es obligatorio **si hay llamadas anidadas**.

88110: Gestión de la pila

```
PUSH: MACRO (ra)
    subu r30, r30, 4
    st    ra, r30, 0
ENDMACRO
```

```
result: res 4
    ...
PUSH(r1)
PUSH(r2)
or r1,r0,low(result)
or.u r1,r1,high(result)
PUSH(r1)
bsr suma
```

```
POP: MACRO (ra)
    ld    ra, r30, 0
    addu r30, r30, 4
ENDMACRO
```

```
suma: PUSH(r1)
    ld r5, r30, 8
    ld r6, r30, 12
    add r5, r5, r6
    ld r6, r30, 4
    st r5, r6, 0
    POP(r1)
    jmp(r1)
```

Marco de pila

- **Marco de pila:** Conjunto de datos privados a una subrutina que incluye, parámetros, dirección de retorno y variables locales.
- Es necesario conocer cómo se organiza la información en la pila en un lenguaje de alto nivel.
- En los ejemplos anteriores, si es necesario almacenar información en la pila el puntero de pila varía a lo largo de la ejecución de la subrutina
- Por ejemplo, el desplazamiento para acceder a un parámetro sería diferente en distintos puntos de una subrutina.

Marco de pila

- Se dedica un registro que apunta a una posición conocida del marco de pila: **puntero de marco de pila** (*frame pointer* o FP). En 88110 es r31.
- Si hay llamadas anidadas, cada una de las llamadas tendrá su propio FP.
- Al entrar en la subrutina hay que asegurar que el marco de pila de la subrutina llamante no se destruye.
- Se debe crear el espacio necesario para variables locales.
- Se deben realizar las inicializaciones de las variables locales.

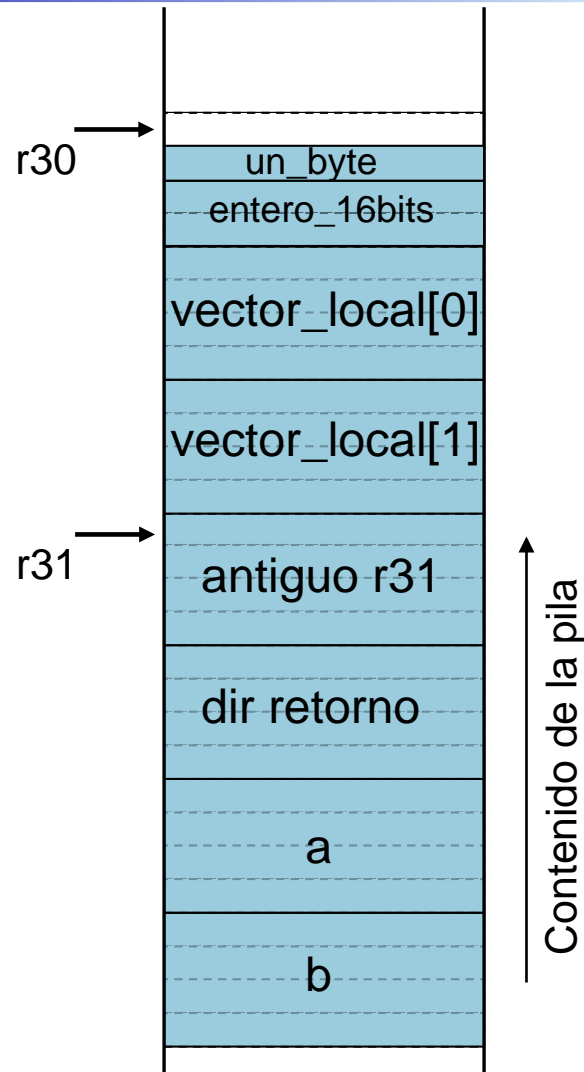
Marco de pila

```
void vector (int a[], int b [])
{
    int vector_local[2];          /* Vector de enteros
                                   de una palabra (4 bytes) */
    short entero_16bits = 0; //Entero de dos bytes
    char un_byte;                // Variable de un byte
    ... ..
```

Marco de pila: creación

```
vector: PUSH (r1)           ; Guarda la dir. de retorno
      PUSH(r31)             ; Guarda el puntero al marco
                              ; de pila del llamante
      or r31, r30, r0        ; Crea el nuevo marco de pila
                              ; a partir de SP (r30)
      subu r30, r30, 12      ; Reserva 12 bytes para
                              ; variables locales
      st.h r0, r31, -10      ; Inicializ. de entero_16bits
      ; Inicio del código de la subrutina
      ld r6, r31, 12         ; Acceso la dirección de
                              ; comienzo del vector b
      ... ..
```

Marco de pila



Marco de pila: destrucción

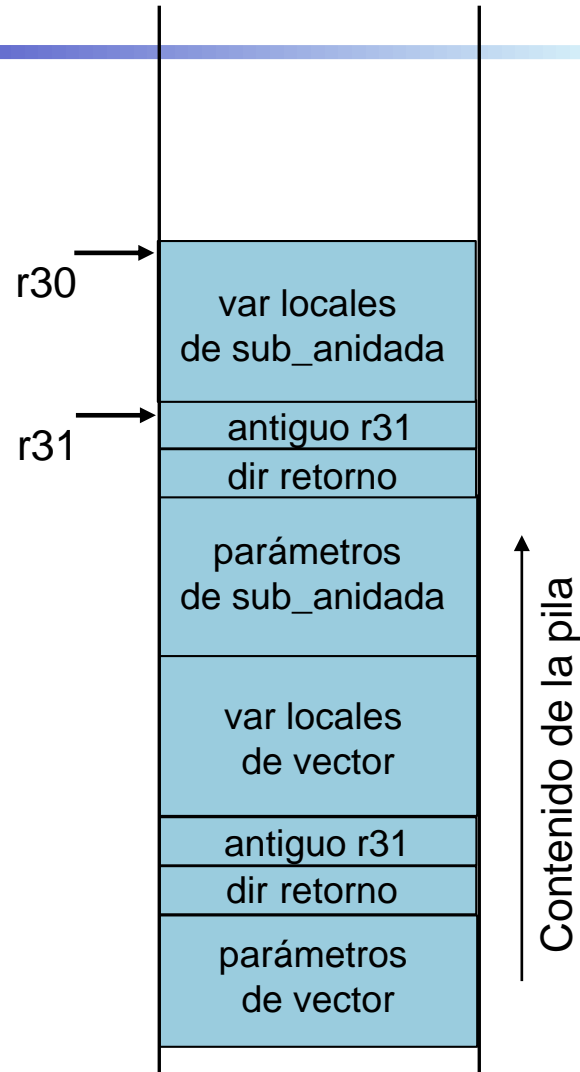
```
or r30, r31, r0      ; Restaura el puntero de pila
                     ; (r30) al valor del puntero
                     ; de marco (r31)

POP(r31)              ; Recupera el puntero de
                     ; marco del llamante

POP(r1)               ; Recupera la dir. de retorno

jmp(r1)               ; Retorno de subrutina
```

Marco de pila



Marco de pila

- CREACIÓN

- Almacenar DR
- Salvar FP llamante
- Activar FP
- Reservar espacio para var locales
- Inicializar var locales

- DESTRUCCIÓN

- Eliminar var locales
- Recuperar FP llamante
- Retornar

subrutinal

```
PUSH (r1)
PUSH (r31)
or r31, r30, r0
sub r30, r30, k

st n, r31, x
. . .
or r30, r31, r0
POP (r31)
POP (r1)
jmp (r1)
```

Tamaño (bytes) de
variables locales

88110: Gestión de la pila

- LLAMADA

1. Salvar registros con variables importantes en variables local (llamante)

```
st r20, r30, 0
```

2. Poner en pila parámetros

```
PUSH (r2)
```

3. Salto a subrutina

```
bsr subrutina1
```

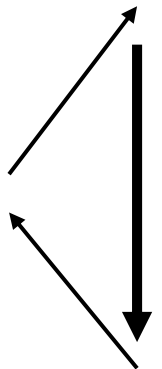
- TRAS RETORNO

4. Limpiar parámetros de la pila

```
POP (r2)
```

5. Recuperar registros con variables importantes de variables locales (llamante)

```
ld r20, r30, 0
```



Parámetros: otras consideraciones

- El orden en el paso de parámetros es un **acuerdo** entre la subrutina llamante y la llamada.
- En el ejemplo anterior el parámetro **a** está en la **cima de la pila** al realizar la llamada a subrutina, tal y como lo realizan los lenguajes de programación de alto nivel.
- El parámetro que está en la cima siempre ocupa la misma posición en la pila independientemente del número de parámetros. Utilizando este se puede acceder al resto.
- Este acuerdo es útil para subrutinas con número variable de parámetros: printf de la librería de C.

Parámetros: funciones

- **Funciones:** subrutinas que tienen un único valor de retorno
 - Indica el estado de ejecución de la función.
 - Es una función matemática y es su resultado.
- Estos valores de retorno se utilizan inmediatamente:
 - Comprobar el estado.
 - Introducir el resultado en una expresión.
- Por estas razones se utilizan **registros** y, habitualmente, el valor de retorno no suele ser un tipo complejo de datos (estructura).

Recursividad

- Una **subrutina recursiva** es la que en su ejecución tiene llamadas a sí misma.
- La recursividad se utiliza para resolver
 - funciones matemáticas cuya definición es recursiva (por ejemplo el factorial).
 - Problemas que requieren almacenamiento en estructuras de datos recursivas (árboles, sistemas de ficheros basados en directorios, etc.)
- Hay que tener en cuenta:

fact (n)

 - **Caso general:** incluye la llamada a la propia función con diferentes parámetros a los que se recibieron.

fact (n) = n * fact (n-1)
 - **Caso particular:** no se realiza la llamada recursiva.

fact (0) = 1