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

Dpto. Arquitectura y Tecnología de Sistemas Informáticos.
Universidad Politécnica de Madrid

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)

http://www.datsi.fi.upm.es/docencia/Estructura_09

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)

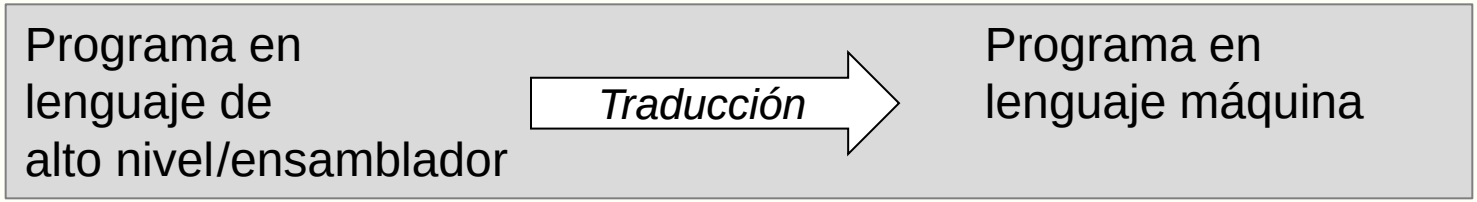
Stallings, W. *"Organización y arquitectura de computadores".* Prentice Hall, 2010. 8ª edición (capítulo 10).

Solución de problemas: http://www.datsi.fi.upm.es/88110

Introducción

- Lenguaje máquina

  Es el que es capaz de interpretar y ejecutar directamente el computador



- Características del lenguaje máquina
  - Basado en sistema de representación binario: Las instrucciones se representan en el computador como cadenas de ceros y unos
  - Particular de cada procesador

Lenguaje máquina: Ejemplos

```
int sumar (int a, int b) {
   int sum, aux1, aux2;
   aux1 = a << 2;
   aux2 = b * 3;
   sum = aux1 + aux2;
   return sum;
}
```

```
 0:  e52db004    push        {fp}
 4:  e28db000    add         fp, sp, #0
 8:  e24dd01c    sub         sp, sp, #28
 c:  e50b0018    str         r0, [fp, #-24]
10:  e50b101c    str         r1, [fp, #-28]
14:  e51b3018    ldr         r3, [fp, #-24]
18:  e1a03103    lsl         r3, r3, #2
1c:  e50b3008    str         r3, [fp, #-8]
20:  e51b201c    ldr         r2, [fp, #-28]
24:  e1a03002    mov         r3, r2
28:  e1a03083    lsl         r3, r3, #1
2c:  e0833002    add         r3, r3, r2
30:  e50b300c    str         r3, [fp, #-12]
34:  e51b2008    ldr         r2, [fp, #-8]
38:  e51b300c    ldr         r3, [fp, #-12]
3c:  e0823003    add         r3, r2, r3
40:  e50b3010    str         r3, [fp, #-16]
44:  e51b3010    ldr         r3, [fp, #-16]
48:  e1a00003    mov         r0, r3
4c:  e28bd000    add         sp, fp, #0
50:  e8bd0800    pop         {fp}
54:  e12fff1e    bx lr
```

```
40051d:   55           push   %rbp
40051e:   48 89 e5     mov    %rsp,%rbp
400521:   89 7d ec     mov    %edi,-0x14(%rbp)
400524:   89 75 e8     mov    %esi,-0x18(%rbp)
400527:   8b 45 ec     mov    -0x14(%rbp),%eax
40052a:   c1 e0 02     shl    $0x2,%eax
40052d:   89 45 fc     mov    %eax,-0x4(%rbp)
400530:   8b 55 e8     mov    -0x18(%rbp),%edx
400533:   89 d0        mov    %edx,%eax
400535:   01 c0        add    %eax,%eax
400537:   01 d0        add    %edx,%eax
400539:   89 45 f8     mov    %eax,-0x8(%rbp)
40053c:   8b 45 f8     mov    -0x8(%rbp),%eax
40053f:   8b 55 fc     mov    -0x4(%rbp),%edx
400542:   01 d0        add    %edx,%eax
400544:   89 45 f4     mov    %eax,-0xc(%rbp)
400547:   8b 45 f4     mov    -0xc(%rbp),%eax
40054a:   5d           pop    %rbp
40054b:   c3           retq
```

Lenguaje máquina

Lenguaje ensamblador

Instrucciones máquina

- Propiedades
  - Realizan una única y sencilla función (en general)
  - Operan sobre un número fijo de operandos con una representación determinada
  - Son autocontenidas: Contienen toda la información necesaria para su ejecución:
    - Operación a realizar
    - Operandos o dónde se encuentran
    - Lugar donde dejar el resultado
    - Ubicación de la siguiente instrucción a ejecutar

Instrucciones máquina

Información que contienen:

- Operación a realizar (código de operación)
- Operandos o dónde se encuentran
  - En registros del procesador
  - En memoria
  - En la propia instrucción
- Lugar donde dejar el resultado
  - En registros del procesador
  - En memoria
- Ubicación de la siguiente instrucción a ejecutar
  - De forma implícita: la siguiente instrucción
  - De forma explícita: en las instrucciones de salto

Instrucciones máquina

Información que contienen:

- Operación a realizar (código de operación)
- Operandos o dónde se encuentran
  - En registros del procesador
  - En memoria
  - En la propia instrucción
- Lugar donde dejar el resultado
  - En registros del procesador
  - En memoria
- Ubicación de la siguiente instrucción a ejecutar
  - De forma implícita: la siguiente instrucción
  - De forma explícita: en las instrucciones de salto

Ejemplo genérico de codificación:

c.o.	opdo-s1	opdo-s2	opdo-d

Ejemplos particulares:

```
e0823003    add r3, r2, r3
01 d0       add %edx,%eax
```

Juego de Instrucciones

Conjunto de instrucciones máquina que es capaz de entender y ejecutar la CPU

- El juego de instrucciones de una CPU viene definido por:
  - Las **operaciones** que realiza
    Qué es capaz de hacer el procesador
  - Los **tipos de datos** que maneja, así como su representación
    Sobre qué tipos de información
  - Los **modos de direccionamiento**
    Cómo se especifica el lugar donde están los operandos
  - El **formato** o formatos de las instrucciones
    Cómo se representan/codifican las instrucciones

Todo ello afecta a velocidad de ejecución

Tipos de instrucciones

- Operaciones básicas:
  - Transferencia de datos (*ld, st, move*)
  - Procesamiento (*add, sub, .., and, cmp, …*)
  - De control del flujo de ejecución (*br*, *bz, call, ret, ...*)

Tipos y tamaño de los datos

- Tipos de datos básicos:
  - Direcciones
  - Números (enteros con/sin signo, coma flotante)
  - Caracteres (cadenas de caracteres)
  - Datos lógicos
- Tamaños:
  - Palabra (tamaño privilegiado del computador)
  - Doble palabra
  - Media palabra
  - Byte (8 bits)

Direccionamiento de la memoria

- A nivel de palabra: direcciones de memoria consecutivas corresponden a palabras consecutivas
- **A nivel de byte**: direcciones de memoria consecutivas corresponden a bytes consecutivos
  - Dos palabras consecutivas están separadas por el tamaño en bytes de la palabra
  - **Ordenación de los bytes de una palabra en memoria**:
    - Little endian: Byte menos significativo de una palabra en la dirección menor
    - Big endian: Byte menos significativo de una palabra en la dirección mayor

Direccionamiento de la memoria

- A nivel de palabra: direcciones de memoria consecutivas corresponden a palabras consecutivas
- **A nivel de byte**: direcciones de memoria consecutivas corresponden a bytes consecutivos
  - Dos palabras consecutivas están separadas por el tamaño en bytes de la palabra
  - **Ordenación de los bytes de una palabra** en memoria:
    - Little endian: Byte menos significativo de una palabra en la dirección menor
    - Big endian: Byte menos significativo de una palabra en la dirección mayor

Ejemplo: número decimal 70.960.543 (0x043AC59F) almacenado en 32 bits en la dirección 184

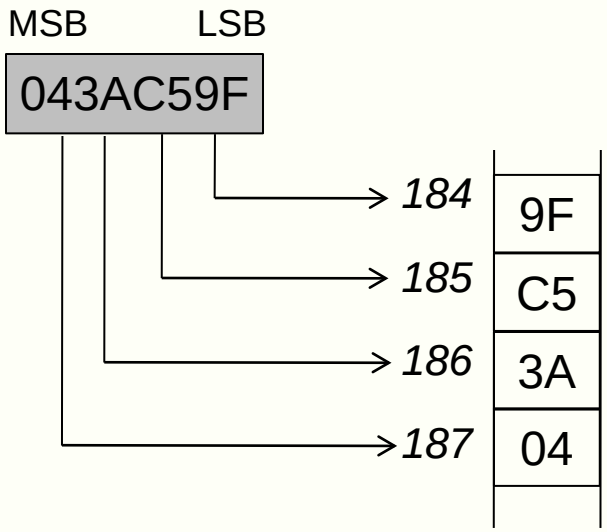
Be:

184	185	186	187
04	3A	C5	9F

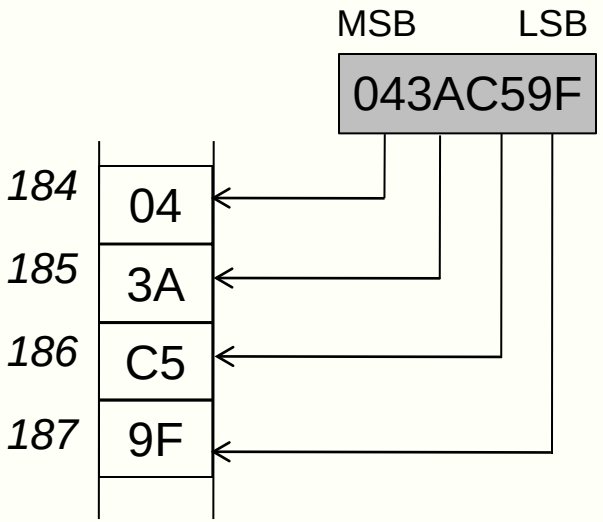
Le:

184	185	186	187
9F	C5	3A	04

Ejemplo Little endian vs Big endian



Little Endian



Big Endian

Direccionamiento de la memoria

- Alineamiento de los datos:

  Un dato que ocupa K bytes está alineado en memoria
  si está almacenado en una dirección múltiplo de K

Ejemplos:

a) Dato de 32 bits: 0x10203040 almacenado en la dirección 185 (NA)
b) Dato de 16 bits: 0x2030 almacenado en la dirección 186 (A)

184		10	20	30
188	40			

- Muchas arquitecturas sólo permiten accesos a datos alineados
- Algunas permiten el acceso a datos no alineados:
  - Los accesos a estos datos son más lentos
  - El acceso a un dato no alineado implica varios accesos a memoria

Modos de direccionamiento

- *Cómo se especifican los operandos de una instrucción* (dato, resultado o instrucción)

  ⇒ Procedimiento empleado por la CPU para acceder a ellos

- Definiciones:
  - **Objeto** del direccionamiento: Instrucción, dato o resultado que se desea direccionar. Puede estar en:
    - La propia instrucción (datos)
    - Un registro (datos o resultados)
    - Memoria (datos, resultados e instrucciones)
  - **Dirección** del objeto: Identifica el lugar en el que reside

    (identificador que especifica el registro o posición de memoria donde se encuentra)

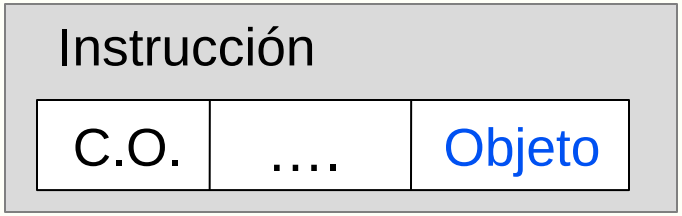
Modos de direccionamiento

- Inmediato . . . . . #valor
- Directo
  - Absoluto
    - a registro . . . . . .registro
    - a memoria . . . . . /direccion
  - Relativo
    - a registro base . . . #desp[.registro_base]
      [.registro_base, #desp]
    - a registro índice
    - a PC . . . . . $desp
- Indirecto . . . . . [ ]
- Implícito

Notación Estándar IEEE 694

Direccionamiento Inmediato

El Objeto está contenido en la propia instrucción



Ejemplos:

```
ADD .R1,#4 ; R1 ← R1+4
LD .R2, #1 ; R2 ← 1
```

- Útil para constantes
- ✓ Rápido: No requiere accesos adicionales a memoria
- ✗ Operando de rango limitado

Direccionamiento Inmediato

Posibilidad de indicar el sistema de representación mediante prefijos. En el estándar IEEE 694:

B´valor en binario
H´valor en hexadecimal
D´valor en decimal

Ejemplos:

```
ADD .R1,#4 ; R1 ← R1+4
LD .R2, #1 ; R2 ← 1
```

- Útil para constantes
- ✓ Rápido: No requiere accesos adicionales a memoria
- ✗ Operando de rango limitado

Direccionamiento Directo

En la instrucción está especificada la Dirección donde está almacenado el Objeto

- Absoluto

  La instrucción contiene la dirección completa del Objeto, que puede estar en un registro o en memoria.

- Relativo

  La instrucción contiene un desplazamiento sobre una dirección, que suele estar almacenada en un registro

  La Dirección del Objeto se calcula sumando ambos

  El Objeto está en memoria

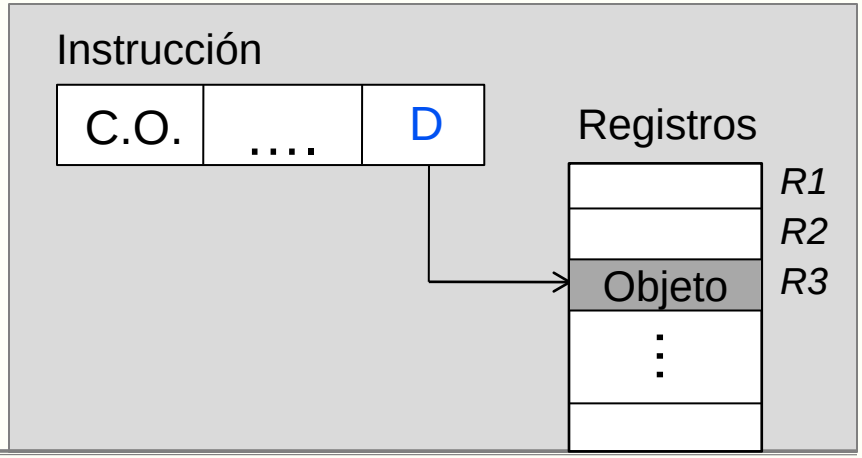
Direccionamiento Directo Absoluto

- **A registro**

  El Objeto está contenido en un registro. La instrucción contiene el *identificador* del registro donde se encuentra

Ejemplo:

```
ADD .R4,.R3 ; R4←R4+R3
```



- Útil para variables utilizadas muy frecuentemente
- ✓ El acceso al operando es rápido al estar en un registro
- ✗ Número limitado de registros
- ✓ Se necesitan pocos bits para especificar la Dirección

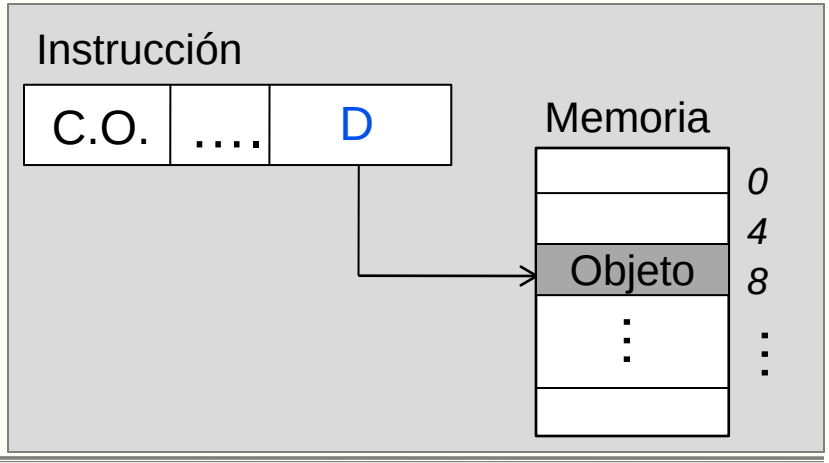
Direccionamiento Directo Absoluto

- **A memoria**

  El Objeto está contenido en memoria. La instrucción contiene la *dirección* completa de memoria donde se encuentra

Ejemplo:

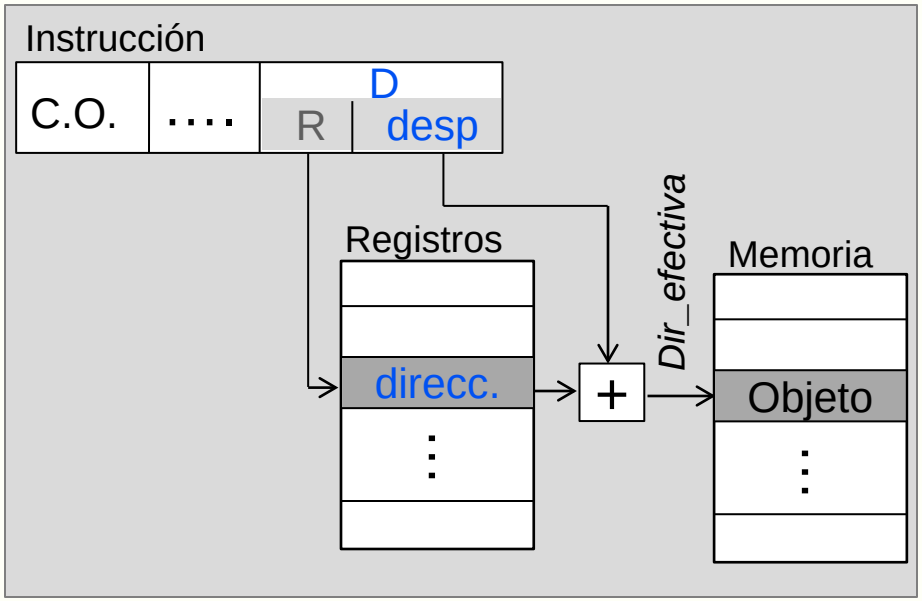
```
LD  .R4,/100 ;R4←Mem(100)
BR  /1000
```



- ✘ Número de bits necesarios para especificar la Dirección
- ✓ Acceso a un gran espacio de direcciones
- ✘ El acceso al operando requiere un acceso a memoria (más lento)
- ✘ El código no es reubicable (depende de su ubicación en Mp)

Direccionamiento Relativo

- El Objeto está en memoria. La instrucción contiene la Dirección, especificada en dos “partes”:
  - Desplazamiento:
    Valor entero con signo
  - Registro:
    Registro general o de propósito específico que contiene una dirección de memoria



- La dirección efectiva donde se encuentra el Objeto se calcula como: Dir_efectiva = Registro + Desplazamiento

Direccionamiento Relativo a Registro Base

- El registro base se carga con la dirección de memoria donde están almacenados un conjunto de datos
- El desplazamiento indica la posición relativa, respecto a la dirección de comienzo, del dato al que se accede
- El registro base no se modifica

Ejemplo:

```
LD .R1, #4[.R7]  ; R1 ← Mem(R7+4)
LD .R1, [#4,.R7]
```

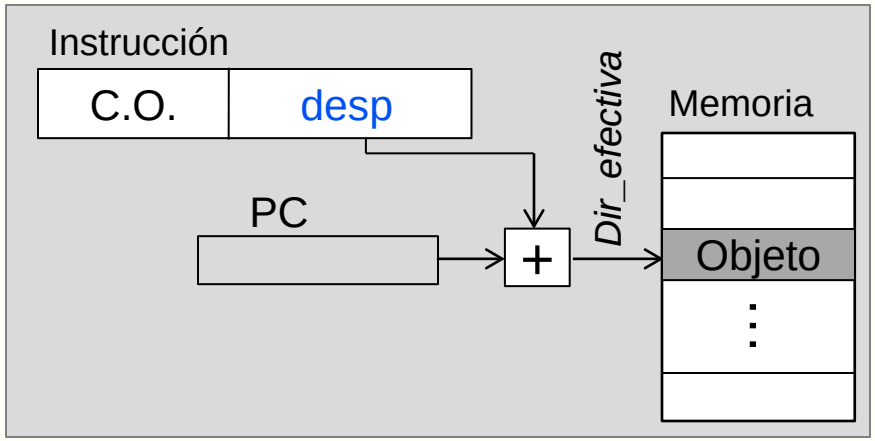
- ✗ El rango de direcciones al que se puede acceder está limitado por el tamaño del desplazamiento
- ✓ Acceso a estructuras de datos y a parámetros en pila
- ✓ Código reubicable (no depende de su ubicación en Mp)

Direccionamiento Relativo a PC

- El registro base es el PC
- Se utiliza en las instrucciones de salto: el Objeto de este direccionamiento es una instrucción

Ejemplo:

```
BR $10   ;PC ← PC+10
```



- ✓ Permite alcanzar instrucciones “cercanas” a la que se está ejecutando: ejecución de saltos “cortos”, p.e. bucles, *if then else* ..
- ✓ El código es reubicable (no depende de su ubicación en Mp)

Direccionamiento Relativo a Registro Índice

- Es un direccionamiento relativo en el que el registro base se modifica
  - Preincremento
  - Predecremento
  - Postincremento
  - Postdecremento
- El tamaño del incremento/decremento es igual al tamaño del objeto direccionado

Ejemplos:

```
LD .R1,#8[++.R7]  ; R7 ← R7+4 y R1 ← Mem(R7+8)
ST .R1,#8[--.R7]  ; R7 ← R7-4 y Mem(R7+8) ← R1
LD .R1,#8[.R7++]  ; R1 ← Mem(R7+8) y R7 ← R7+4
SUB .R1,#8[.R7--] ; R1 ← R1-Mem(R7+8) y R7 ← R7-4
```

✓ Útil para el recorrido de vectores y matrices

Ejercicios

1. Qué hacen las siguientes instrucciones y qué registros y posiciones de memoria se modifican.

```
ADD .R1, #4[.R2], #8[.R3]
ADD #0[.R1], #4[.R2], #8[.R3]
ADD #0[--.R1], #4[.R2++], #8[--.R3]
```

   Valores iniciales de registros: R1 = 100, R2 = 200, R3 = 400

2. Secuencia de instrucciones para sumar los elementos de un array almacenado a partir de la dirección contenida en R1 dejando el resultado en R2

   Inicialmente R2 = 0 y R3 = Nº de elementos

   Instrucciones a utilizar: `ADD, SUB` y `BRNZ` de 4Bytes

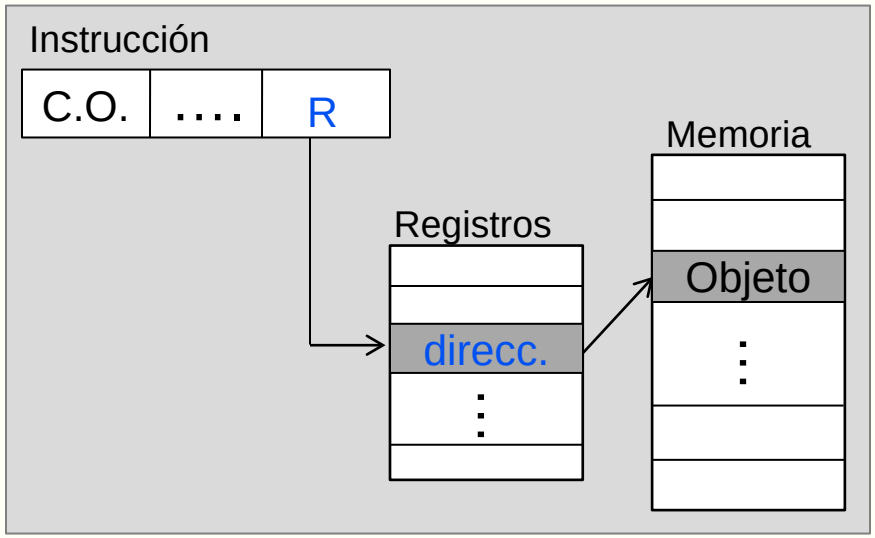
   Direccionamientos: Los vistos hasta ahora

Direccionamiento Indirecto

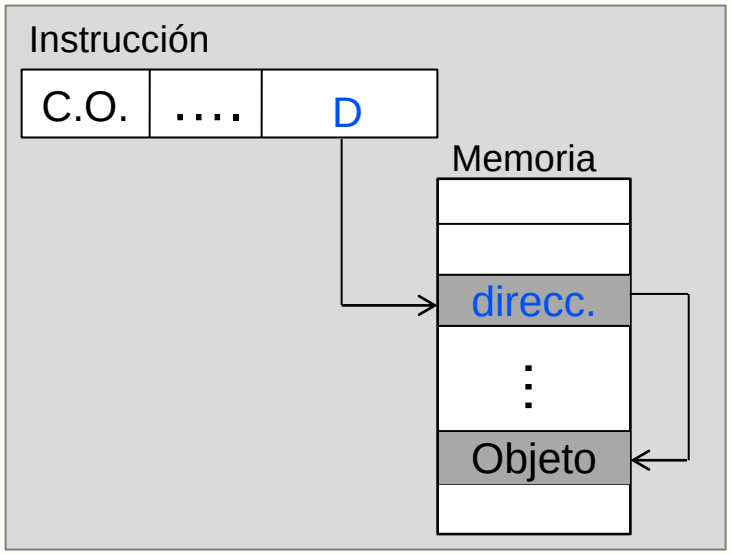
En la instrucción está especificada la dirección donde está almacenada la Dirección en la que se almacena el Objeto.

- Comienza por un direccionamiento Directo, pero la dirección obtenida no apunta al Objeto sino a su Dirección

Ejemplos:



(a) Indirecto a registro



(b) Indirecto a memoria

Direccionamiento Indirecto: Ejemplos

- A registro

```
LD .R1,[.R4]  ; R1 ← Mem(R4)
BR [.R1]      ; PC ← R1
```

- A memoria

```
LD .R1, [/1000] ; R1 ← Mem(Mem(1000))
```

- A registro base

```
LD .R1, [#8[.R4]] ; R1 ← Mem(Mem(R4+8))
```

- A registro índice

```
LD .R1, [#8[.R2++]] ; R1 ← Mem(Mem(R2+8)), R2 ← R2+4
```

- ✓ Paso de parámetros por referencia
- ✓ Uso de punteros a variables

Direccionamiento Implícito

La instrucción no contiene ni la Dirección ni el Objeto del direccionamiento

- El Operando se supone ubicado en algún lugar específico de la máquina, por ejemplo:
  - La pila

```
PUSH .R1  (operando destino implícito)
POP .R2   (operando fuente implícito)
```

  - El registro Acumulador (en máquinas de acumulador)

```
ADDA .R1 ; AC ← AC+R1
```

  - El PC

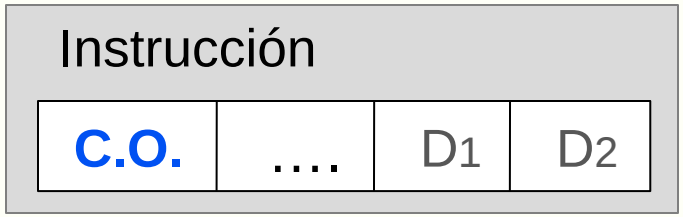
```
BR $-12   ; PC ← PC-12
```

Modos de direccionamiento más utilizados

	Estándar	Intel	Arm	MC88110
Inmediato	`#n`	`$n`	`#n`	`n`
Absoluto a registro	`.R`	`%r`	`r`	`r`
Relativo a Registro	`#n[.R]` `[.R, #n]`	`#n(%r)`	`[r, #n]`	`r, n`

Juego de instrucciones

- **Tipos de Instrucciones**
  - Transferencia
  - Aritméticas y de Comparación
  - Lógicas
  - De bit
  - Desplazamiento y rotación
  - De salto o bifurcación
  - De Entrada/Salida y misceláneas



Transferencia de datos

- Mueven información entre registros, registros y posiciones de memoria o entre posiciones de memoria
- No se modifican los biestables de estado

LD, ST y MOVE

```
LD .R2, #4[.R4]        ; R2 ← MEM(R4+4)
ST .R2, #4[.R4]        ; MEM(R4+4)← R2
MOVE .R2,.R4           ; R4 ← R2
MOVE [.R2],[.R4]       ; MEM(R4)← MEM(R2)
```

PUSH y POP

```
PUSH .R1       ; MEM(SP)← R1; SP ← SP-4
POP  .R1       ; SP ← SP + 4; R1 ← MEM(SP)
```

Transferencia de datos

Posibilidad de indicar, en algunas, el tamaño de la información mediante sufijos y si es con signo o sin el.
En el estándar IEEE 694:

.B	8 bits	**-U**	sin signo
.S	16 bits	**-S**	con signo
.L	32 bits		
.Q	64 bits		

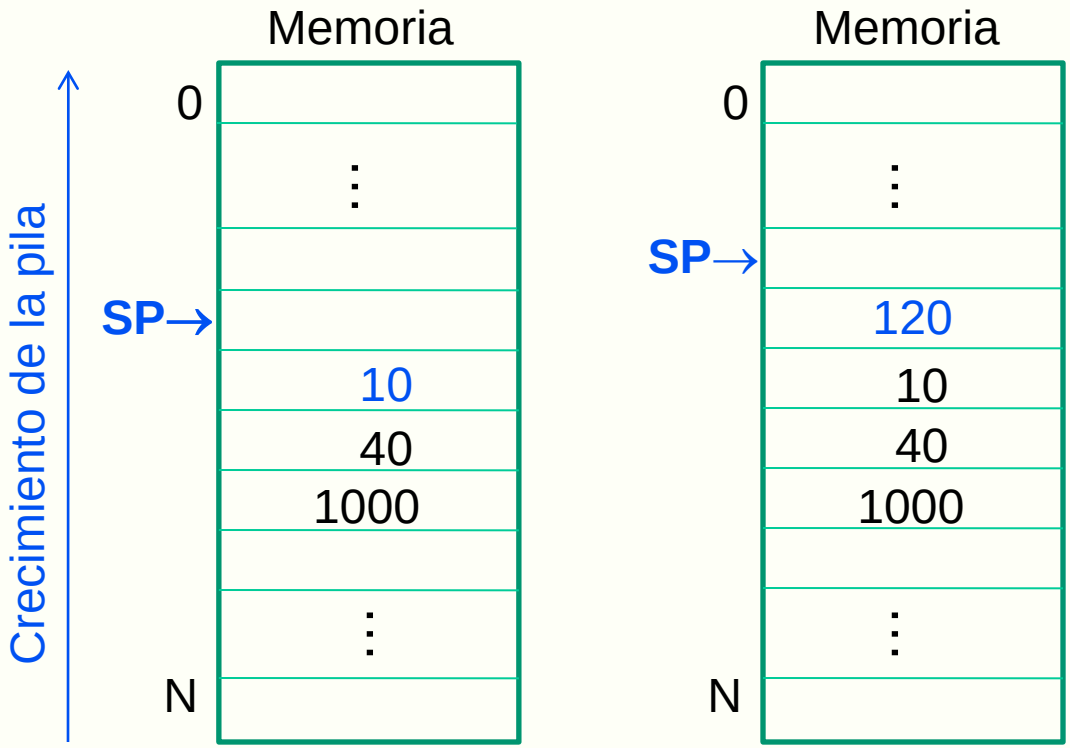
LD, ST y MOVE

```
LD .R2, #4[.R4]        ; R2 ← MEM(R4+4)
ST .R2, #4[.R4]        ; MEM(R4+4)← R2
MOVE .R2,.R4           ; R4 ← R2
MOVE [.R2],[.R4]       ; MEM(R4)← MEM(R2)
```

PUSH y POP

```
PUSH .R1       ; MEM(SP)← R1; SP ← SP-4
POP  .R1       ; SP ← SP + 4; R1 ← MEM(SP)
```

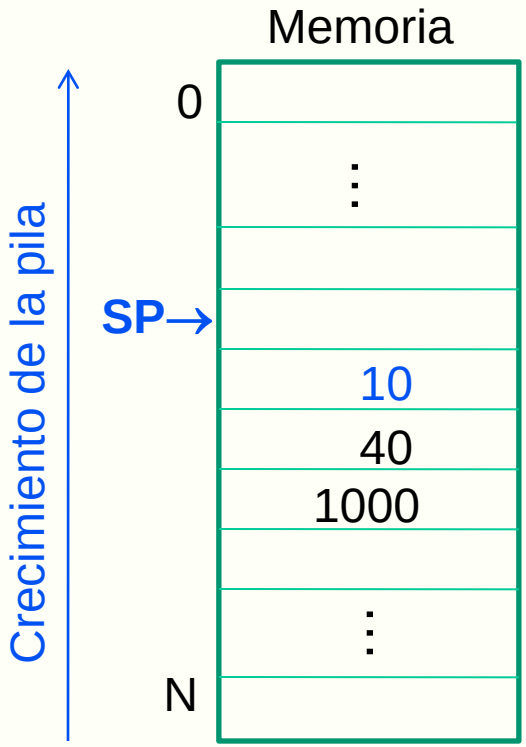
Operaciones PUSH y POP

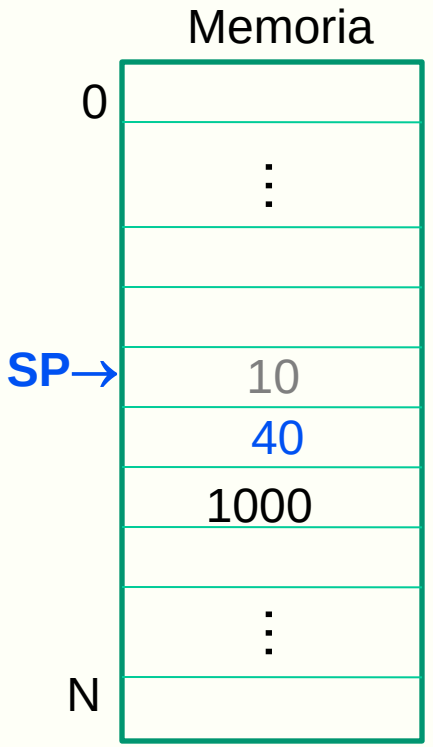


Situación inicial:
- SP apunta a la primera posición libre
- R1 contiene “120”

PUSH .R1
R1 no se modifica

Operaciones PUSH y POP





Situación inicial:
- SP apunta a la primera posición libre
- R1 contiene “120”

POP .R1
R1 contiene ahora “10”

Aritméticas y de Comparación

- Todas modifican los biestables de estado

```
ADD .R1, .R2        ; R1 ← R1 + R2
SUB .R1, .R2        ; R1 ← R1 – R2
MUL .R1, .R2        ; R1 ← R1 * R2
DIV .R1, .R2        ; R1 ← R1 / R2
ADDC .R1, .R2       ; R1 ← R1 + R2 + C
SUBC .R1, .R2       ; R1 ← R1 - R2 - C
CMP .R1, .R2        ; R1 – R2 ; No modifica R1 ni R2
INC .R1             ; R1 ← R1+1
DEC .R2             ; R2 ← R2-1
```

Aritméticas y de Comparación

Posibilidad de indicar si los datos son con signo o sin el. En el estándar IEEE 694:

- **-U** sin signo
- **-S** con signo

```
ADD .R1, .R2        ; R1 ← R1 + R2
SUB .R1, .R2        ; R1 ← R1 – R2
MUL .R1, .R2        ; R1 ← R1 * R2
DIV .R1, .R2        ; R1 ← R1 / R2
ADDC .R1, .R2       ; R1 ← R1 + R2 + C
SUBC .R1, .R2       ; R1 ← R1 - R2 - C
CMP .R1, .R2        ; R1 – R2 ; No modifica R1 ni R2
INC .R1             ; R1 ← R1+1
DEC .R2             ; R2 ← R2-1
```

Aritméticas y de Comparación

- La ubicación de los operandos depende del modelo de ejecución del procesador:
  - Registro-Registro

```
ADD .R1,.R2      ; R1 ← R1 + R2
SUB .R1,#4       ; R1 ← R1 - 4
```

  - Registro-Memoria

```
CMP .R1,[.R2++] ; R1 - Mem(R2) ; R2 ← R2+4
```

  - Memoria-Memoria

```
MUL [.R1],#8[.R2] ; Mem(R1) ← Mem(R1)* Mem(R2+8)
```

- Definen el número de direcciones del procesador

Número de direcciones: Ejemplos

- Tres direcciones

```
ADD .R3,.R1,.R2    ; R3 ← R1 + R2
```

- Dos direcciones

```
ADD .R1,.R2        ; R1 ← R1 + R2
```

- Una dirección (máquina de acumulador).

```
ADD .R1            ; AC ← AC + R1
```

- Cero direcciones (máquina de pila)

```
ADD
```

Lógicas

- Realizan la operación indicada, bit a bit
- Modifican los biestables de estado

```
AND .R1,.R2   ; R1 ← R1 AND R2
XOR .R1, #4   ; R1 ← R1 XOR 4
OR .R1,[.R2]  ; R1 ← R1 OR Mem(R2)
NOT #8[.R1]   ; R1 ← NOT Mem(R1+8)
```

- ✓ Útiles para trabajar con máscaras, p.e:

```
AND .R1, #1          ; Si biestable Z = 1 el número es par
AND .R1, #H'000000FF ; Se extrae el byte de menor peso
```

De bit

- Realizan operaciones sobre un bit

```
CLR.I #3,.R1    ; Pone a cero el bit 3 de R1
SET.I #3,.R1    ; Pone a uno el bit 3 de R1
TEST.I #3,.R1   ; biestable Z ← NOT(bit3(R1))
                ; si bit3(R1)=0 biestable Z ← 1
                ; si no, biestable Z ← 0
```

De bit

- Realizan operaciones sobre un bit

```
CLR.I #3,.R1    ; Pone a cero el bit 3 de R1
SET.I #3,.R1    ; Pone a uno el bit 3 de R1
TEST.I #3,.R1   ; si bit3(R1)=0 biestable Z ← 1
                ; si no biestable Z ← 0
```

Ejercicio:

CLR.I #3,.R1

¿Utilizando la instrucción AND .R1, #inmediato?

SET.I #3,.R1

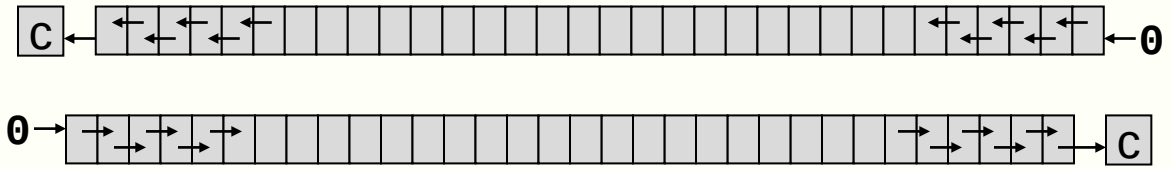
¿Utilizando la instrucción OR .R1, #inmediato?

Desplazamiento

- Realizan desplazamientos de bits a izquierda/derecha (equivalen a multiplicar/dividir por 2)
- Tipos de desplazamiento:
  - **Lógico**

    SHL .R1

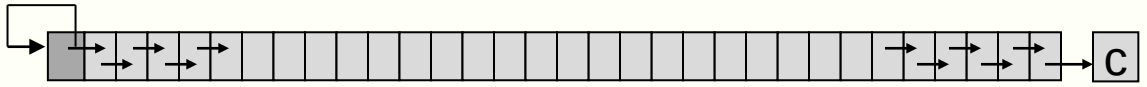
    SHR .R2

  - **Aritmético**: Para enteros con signo

    SHLA .R1

    SHRA .R2

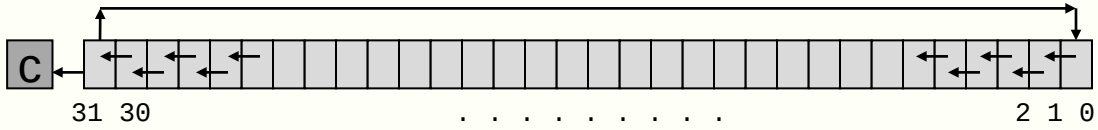
    Comportamiento si se utiliza c.a 1 o c.a 2

- Pueden especificar el nº de posiciones a desplazar, p.e. **SHL .R1, #4**

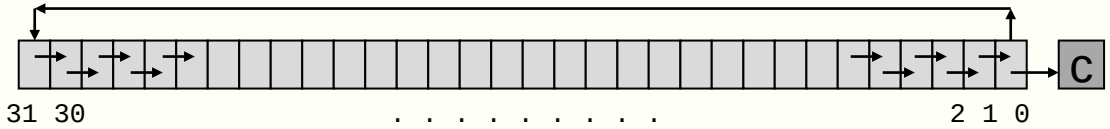
Rotación

- Realizan rotaciones de bits a izquierda/derecha incluyendo o no el biestable de Acarreo
  - Rotación

ROL .R1

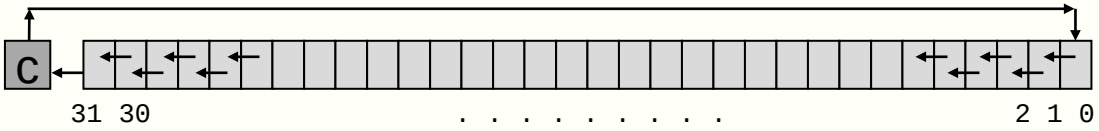


ROR .R1

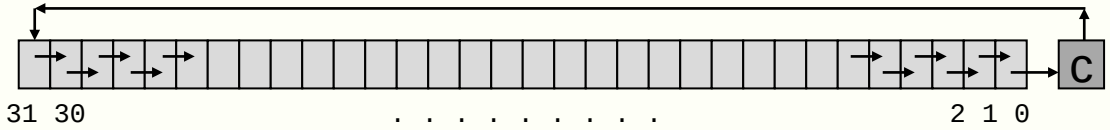


  - Rotación a través del biestable de Acarreo

ROLC .R1

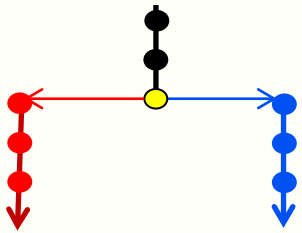


RORC .R1

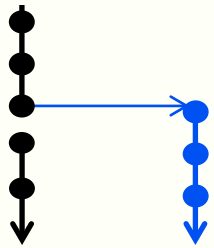


Salto o Bifurcación

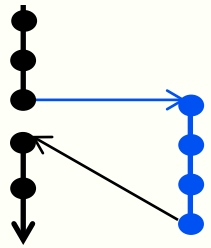
- Modifican la secuencia habitual de ejecución (secuencial)
- No modifican los biestables de estado
- Tipos:
  - Condicionales/Incondicionales
  - Con/Sin Retorno



```
if (cond)
        código-1
else
        código-2
```



goto etiqueta



```
x = sumar(a)
y = x*2
```

Saltos Incondicionales

- El salto se realiza siempre
- Tienen como operando la dirección de memoria donde está la siguiente instrucción a ejecutar

```
BR /1000      ; PC ← 1000
BR #4[.R3]    ; PC ← R3 + 4
BR $10        ; PC ← PC + 10
              ; ¡¡El PC apunta a la dirección de
              ; la siguiente instrucción!!
```

Saltos Condicionales

- El salto se realiza solo si se cumple la condición especificada en la propia instrucción:

  B*cc* Direccion

  cc = Z, C, V, P, N, EQ, GT, GE, LT, LE, NZ, NC, NV, NEQ, …

```
BZ /1000      ; si biestable(Z)=1   PC ← 1000
              ; si no PC ← PC+N
BNC #4[.R3]   ; si biestable(C)=0   PC ← R3 + 4
              ; si no PC ← PC+N
BP $10        ; si biestable(S)=0  PC ← PC + 10
              ; si no PC ← PC+N
```

Saltos Con retorno

- Se utilizan para implementar saltos a subrutinas o llamadas a funciones.
- Una vez ejecutado el código de la subrutina/función se debe retornar a la instrucción siguiente al salto.

```
CALL direccion y RET
```

- Es necesario guardar la dirección de retorno:
  - En un registro de propósito general

```
CALL /1000    ; r1 ← PC , PC ← 1000
RET           ; PC ← R1
```

No permite llamadas anidadas

  - En la pila

Saltos Con retorno

- Guardando la dirección de retorno en la pila

```
CALL /1000   ; Mem(SP) ← PC, SP ← SP-4,
             ; PC ← 1000
RET          ; SP ← SP+4, PC ← Mem(SP)
```

Permite llamadas anidadas

- Si en la subrutina se trabaja con la pila, hay que tener en cuenta que la dirección de retorno ha quedado en la cima de la pila.

Formato de las instrucciones

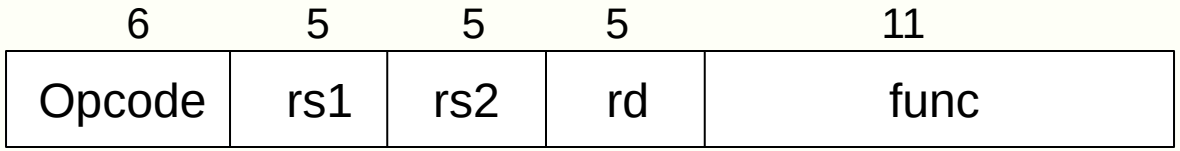
- Modo en que se disponen los bits que identifican cada uno de los elementos que componen una instrucción: campos
  - Código de operación → Nº de operaciones distintas
  - Operando(s) → Número de direcciones explícitas y modos de direccionamiento
- Se suele emplear más de un formato

  Cada instrucción encaja en un formato predeterminado
- Aspectos de diseño:
  - Longitud fija o variable
  - Número de formatos distintos
  - Codificación regular

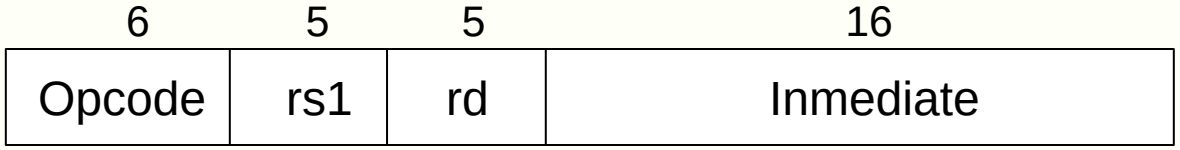
Impacto en la velocidad y sencillez de CPU

Formatos de instrucción: Ejemplo

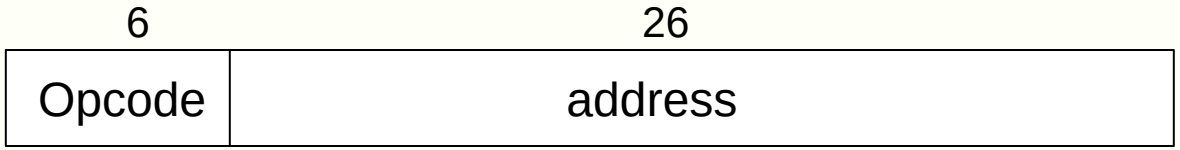
DLX



R-Type



I-Type



J-Type

Computadores CISC y RISC

- Dos paradigmas de Arquitectura de un computador
- Responden a la evolución tecnológica en el diseño de procesadores
- CISC (*Complex Instruction Set Computer*)

  Observación:

  Alrededor del 20% de las instrucciones ocupa el 80% del t.ejecución de un programa

- RISC (*Reduced Instruction Set Computer*) - 1980

  Idea básica:

  Hacer más rápidos los casos más frecuentes

CISC vs RISC

- Modelo R-R, R-M y M-M
- Muchas instrucciones y complejas (↑t.ejecución)
- Modos de direccionamiento complejos
- Muchos formatos de instrucción
- Instrucciones de tamaño variable
- U.Control más compleja
- Menor nº de instrucciones por programa
- Ejs: ix86, Amd

- Modelo R-R (solo load y store hacen referencia a memoria)
- Instrucciones sencillas y ortogonales (↓t. ejecución)
- Modos de direccionamiento sencillos
- Pocos formatos de instrucción y codificación uniforme
- Instrucciones de tamaño fijo
- U. Control más sencilla
- Mayor nº de instrucciones por programa
- Ejs: Sparc, Arm, PowerPC

Formatos de instrucción de un CISC

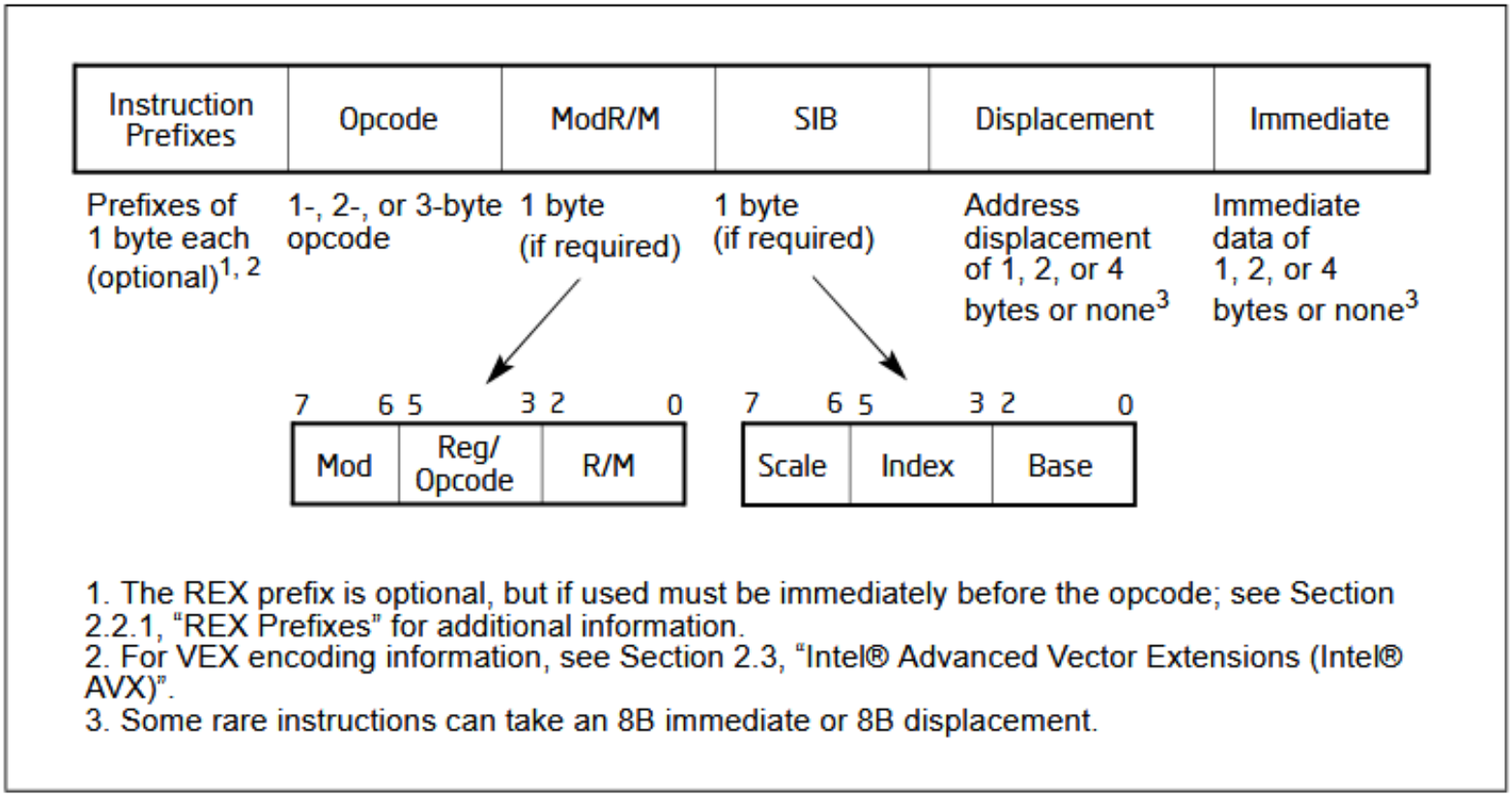
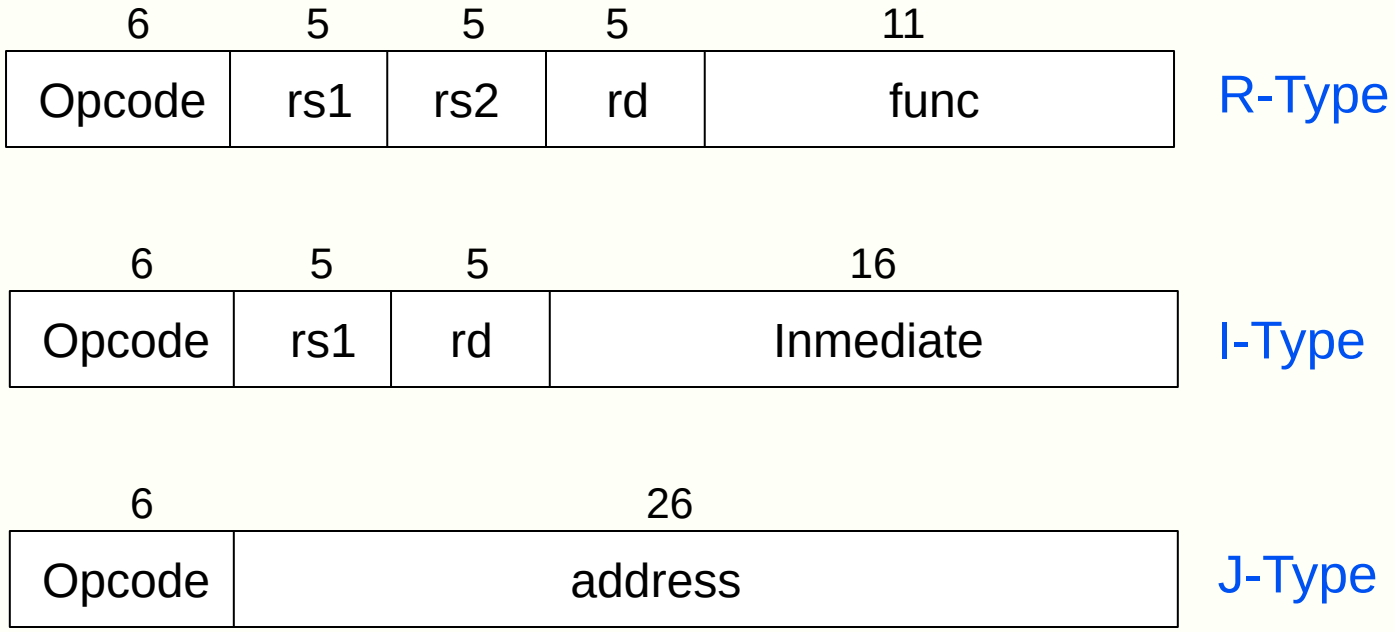


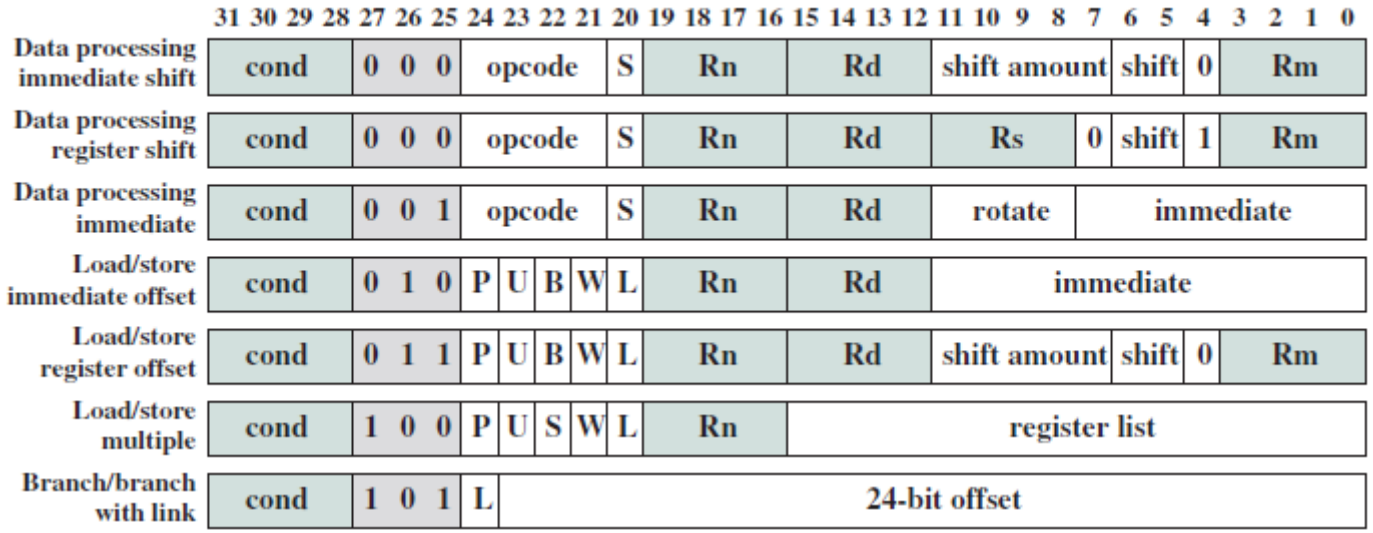
Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

Formatos de instrucción de un RISC

DLX



Formatos de instrucción de un RISC



S = For data processing instructions, signifies that the instruction updates the condition codes

S = For load/store multiple instructions, signifies whether instruction execution is restricted to supervisor mode

P, U, W = bits that distinguish among different types of addressing mode

B = Distinguishes between an unsigned byte (B==1) and a word (B==0) access

L = For load/store instructions, distinguishes between a Load (L==1) and a Store (L==0)

L = For branch instructions, determines whether a return address is stored in the link register

Figure 13.10 ARM Instruction Formats

W. Stallings

Lenguaje máquina: Ejemplos

```
int sumar (int a, int b) {
   int sum, aux1, aux2;
   aux1 = a << 2;
   aux2 = b * 3;
   sum = aux1 + aux2;
   return sum;
}
```

```
 0:  e52db004    push        {fp}
 4:  e28db000    add         fp, sp, #0
 8:  e24dd01c    sub         sp, sp, #28
 c:  e50b0018    str         r0, [fp, #-24]
10:  e50b101c    str         r1, [fp, #-28]
14:  e51b3018    ldr         r3, [fp, #-24]
18:  e1a03103    lsl         r3, r3, #2
1c:  e50b3008    str         r3, [fp, #-8]
20:  e51b201c    ldr         r2, [fp, #-28]
24:  e1a03002    mov         r3, r2
28:  e1a03083    lsl         r3, r3, #1
2c:  e0833002    add         r3, r3, r2
30:  e50b300c    str         r3, [fp, #-12]
34:  e51b2008    ldr         r2, [fp, #-8]
38:  e51b300c    ldr         r3, [fp, #-12]
3c:  e0823003    add         r3, r2, r3
40:  e50b3010    str         r3, [fp, #-16]
44:  e51b3010    ldr         r3, [fp, #-16]
48:  e1a00003    mov         r0, r3
4c:  e28bd000    add         sp, fp, #0
50:  e8bd0800    pop         {fp}
54:  e12fff1e    bx lr
```

```
40051d:   55           push   %rbp
40051e:   48 89 e5     mov    %rsp,%rbp
400521:   89 7d ec     mov    %edi,-0x14(%rbp)
400524:   89 75 e8     mov    %esi,-0x18(%rbp)
400527:   8b 45 ec     mov    -0x14(%rbp),%eax
40052a:   c1 e0 02     shl    $0x2,%eax
40052d:   89 45 fc     mov    %eax,-0x4(%rbp)
400530:   8b 55 e8     mov    -0x18(%rbp),%edx
400533:   89 d0        mov    %edx,%eax
400535:   01 c0        add    %eax,%eax
400537:   01 d0        add    %edx,%eax
400539:   89 45 f8     mov    %eax,-0x8(%rbp)
40053c:   8b 45 f8     mov    -0x8(%rbp),%eax
40053f:   8b 55 fc     mov    -0x4(%rbp),%edx
400542:   01 d0        add    %edx,%eax
400544:   89 45 f4     mov    %eax,-0xc(%rbp)
400547:   8b 45 f4     mov    -0xc(%rbp),%eax
40054a:   5d           pop    %rbp
40054b:   c3           retq
```

Resumen: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 ← rS1 + rS2
```

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

Destino: { Registro }

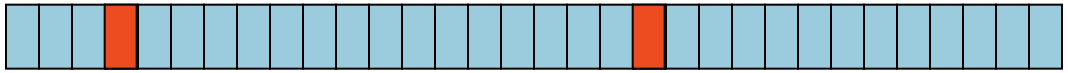
Origen: { Registro + Registro ; Registro + Inmediato }

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

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)

PSR:



31 . . . . . 3 2 1 0

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 → 4 direcciones de memoria
- Bus de direcciones de **32 bits**
  Máxima capacidad (teórica) → 2^{32} bytes = 4 GB
  { 0x00000000
  { 0xFFFFFFFF
- Capacidad del emulador:
  2^{18} direcciones = 2^{18} bytes = 256 KB
  { 0x00000000
  { 0x0003FFFF

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

Direccionamiento directo a registro

```
add   r1, r2, r3      ; r1 ← 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:

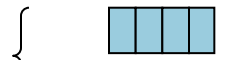
```
Con signo:  SIMM16
Sin signo:   IMM16
```

- Se puede expresar en decimal o hexadecimal
- Ejemplo:

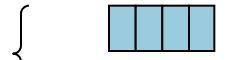
```
add r1, r2, -13;
```

13 en binario: **0000 1101**
-13 en binario: **- 0000 1101**
-13 en binario: **1111 0011**

add r1, r2, 0xFFF3



addu r1, r2, 0xFFF3



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
```

Direccionamiento indirecto a registro

En el 88110 solo existe en los saltos:

- Ejemplo en formato del estándar IEEE:

```
JMP     [.R10]          ; PC ← R10
```

- Ejemplo en formato del 88110:

```
jmp     (r10)           ; PC ← r10
```

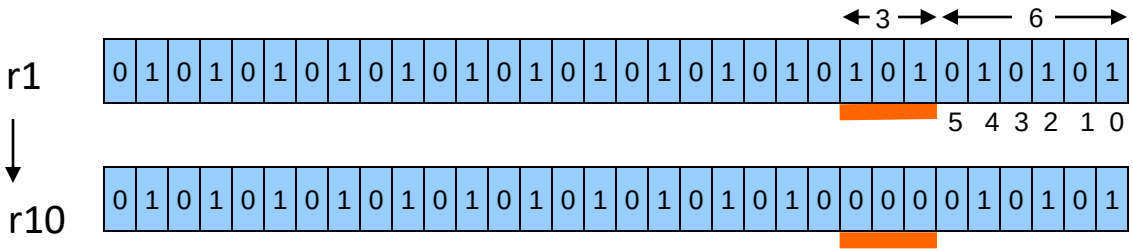
Direccionamiento campos de bit

En ciertas instrucciones del 88110 se pueden seleccionar:

- bits individuales
- campos de bit

- Ejemplo en formato del estándar IEEE / 88110:

CLR.I3 .R10, .R1, 6 / clr r10, r1, 3<6>



- 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`**

Instrucciones lógicas

Lógicas: **or, and, xor, mask**

INST	Operandos	Ext
or and	rD, rS1, rS2	c
xor	rD, rS1, IMM16	u
mask	rD, rS1, IMM16	u

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	
add	rD, rS1, rS2	ci,co,cio	causan excep. overflow (OVF)
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 { rD, rS1, rS2
    { rD, rS1, SIMM16
```

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 rS1 = rS2
ne: 1 si y solo si rS1 ≠ rS2
gt: 1 si y solo si rS1 > rS2 (con signo)
. . . . . .

hi: 1 si y solo si rS1 > rS2 (sin signo)
. . . . . .

ARQUITECTURA 88110

Bifurcaciones/saltos

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

INST	Operandos	
bb0 bb1	B,rS1,D16 B,rS1,D16	PC ← PC+4*D16 si bit B de rS1= 0 / 1
br	D26	PC ← PC+4*D26
bsr	D26	r1←PC+4;PC←PC+4*D26
jmp	(rS1)	PC←rS1 (alineado)
jsr	(rS1)	r1←PC+4;PC←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 rD, rS1, rS2	b,h,d
ldcr	rD	rD ← PSR
stcr	rS1	PSR ← rS1
xmem	rD, rS1, rS2	rD ←→ MEM(rS1+rS2)

Campos de bit

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

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

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 x̄=d x=d x̄=s
fcvt.xx̄ 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: Generando Codigo...
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>
```

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
```

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.

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

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

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

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

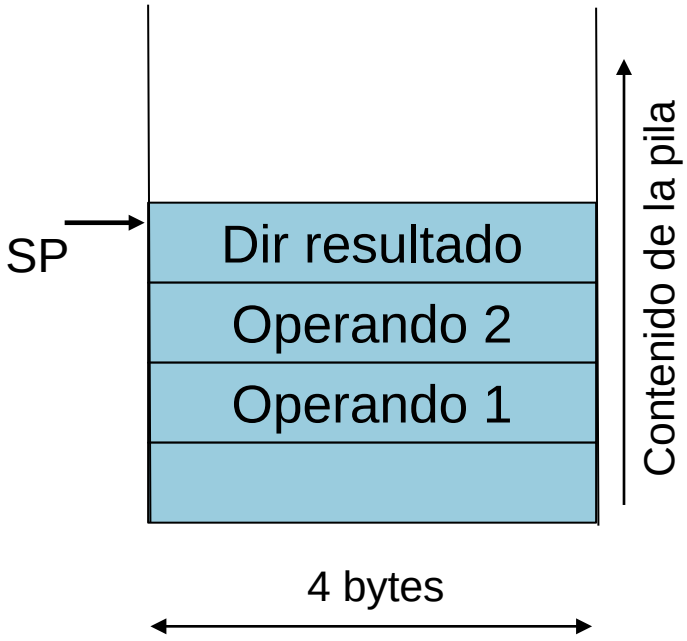
```
RESULT: RES 4
parámetro
   …
PUSH .R1
PUSH .R2
LD .R1,#RESULT
PUSH .R1
CALL /SUMA
```

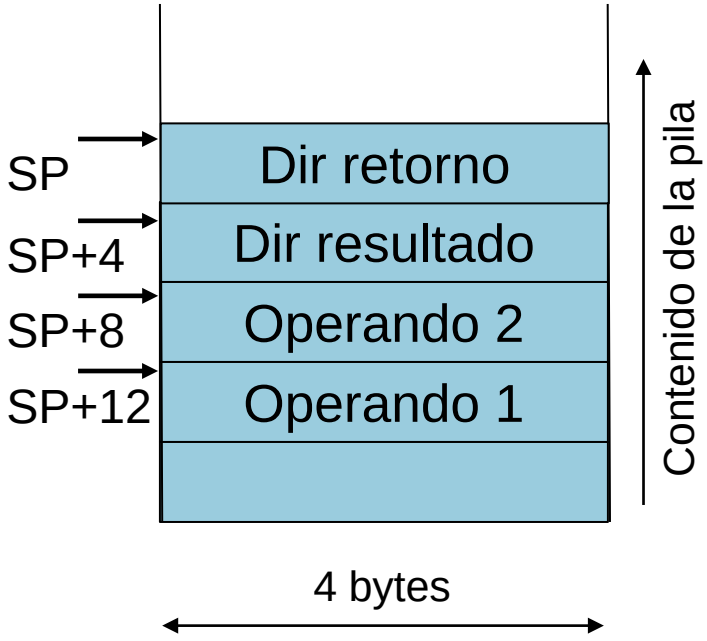
Subrutina

```
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.

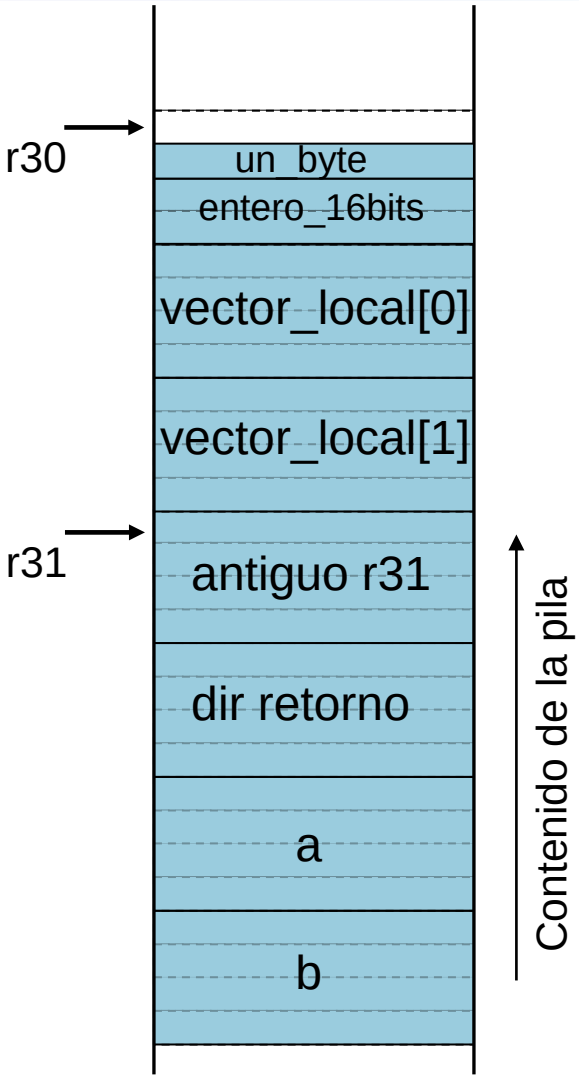
Marco de pila

```
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
        … … …
```

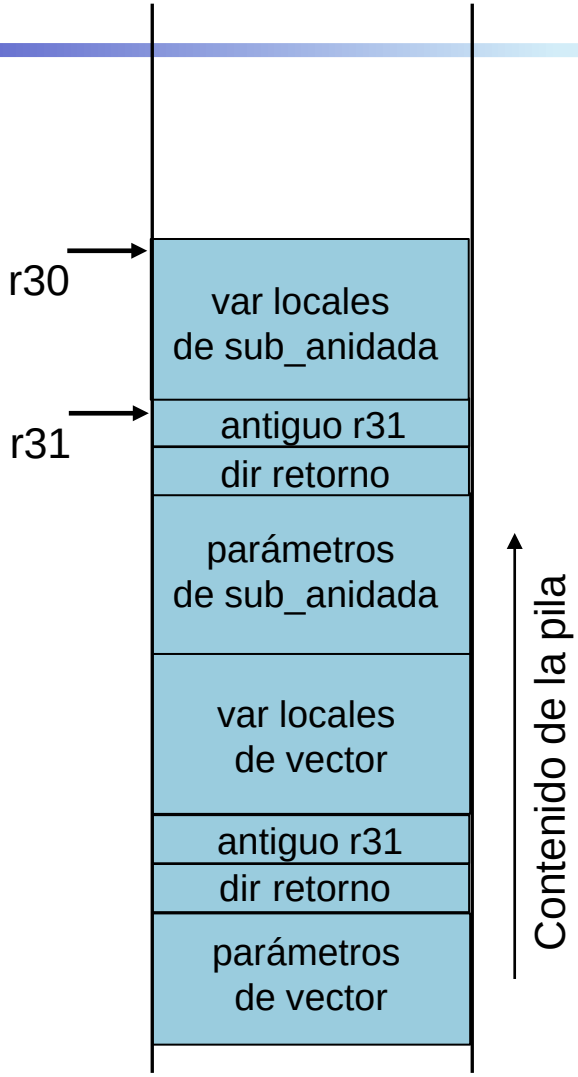
Marco de pila



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

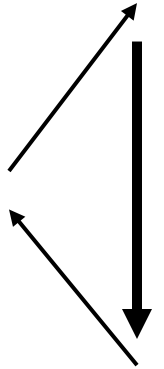
```
subrutina1
    PUSH (r1)
    PUSH (r31)
    or r31, r30, r0
    subu r30, r30, k

    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)
  2. Poner en pila parámetros
  3. Salto a subrutina
- TRAS RETORNO
  4. Limpiar parámetros de la pila
  5. Recuperar registros con variables importantes de variables locales (llamante)

```
st r20, r31, x
PUSH (r2)
bsr subrutina1
POP (r2) /addu r30, r30, k
ld r20, r31, x
```



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`