

Arquitectura de Ordenadores



Construcciones de Alto Nivel en Ensamblador

Abelardo Pardo

abel@it.uc3m.es



Universidad Carlos III de Madrid

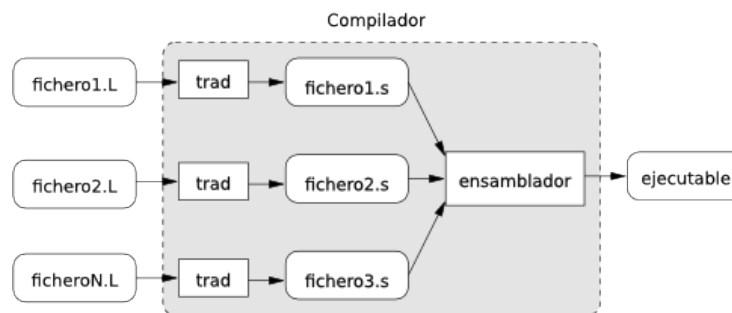
Departamento de Ingeniería Telemática

Construyendo Programas más Complejos

SUB-1

- Se ha visto cómo funciona el procesador al nivel de **lenguaje máquina**.
- ¿Cómo se construyen aplicaciones **más complejas** sobre este procesador?
- Se precisan nuevos **niveles de abstracción**.
- Los **lenguajes de programación de alto nivel** tales como Java permiten dar órdenes más complejas al procesador.

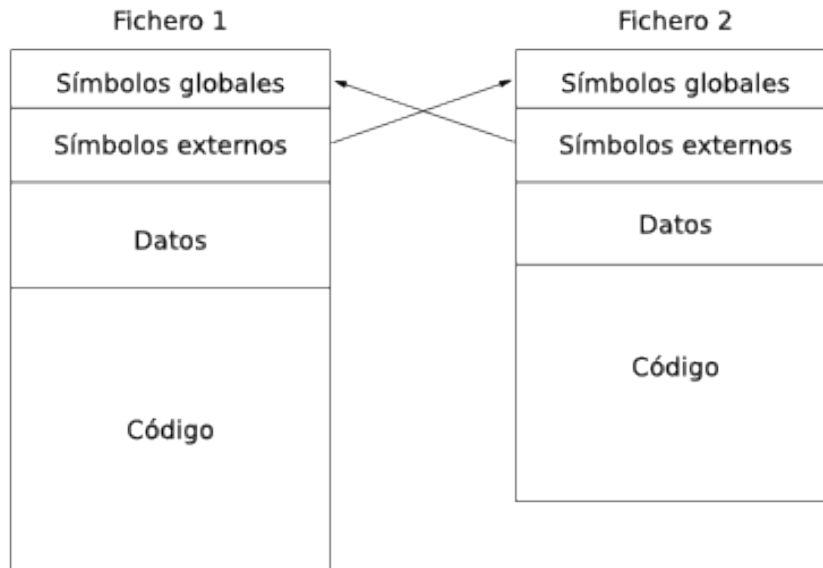
- Desde el punto de vista de la **capacidad de operación** de un procesador, el lenguaje máquina es **suficiente** para implementar **cualquier programa**.
- Construir aplicaciones complejas, aunque posible, **no es eficiente** utilizando únicamente el lenguaje ensamblador.
- El ensamblador carece de **tipos de datos**. Todo se almacena en memoria y puede accederse sin ningún tipo de restricciones.
- El mecanismo para organizar el código en subrutinas es muy **básico y delicado**.
- Necesitamos un programa que nos permita manipular nuestros datos con **cierta estructura** y que nos permita escribir código a más **alto nivel**.
- **Compilador**: Programa que traduce de un lenguaje de alto nivel a lenguaje ensamblador.



Cómo fragmentar el código en varios ficheros

- En cuanto un programa crece en complejidad se precisa **organizar el código de forma fácil de manipular**.
- El primer mecanismo a utilizar es el **distribuir el código entre varios ficheros**.
- Para que esta fragmentación sea **efectiva** se precisa:
 - Un mecanismo que en un determinado punto permita ejecutar una porción de código **con retorno**.
 - Una política de acceso a las **etiquetas** de cada uno de los ficheros.

- Un fichero puede incluir referencias a **identificadores externos**.
- **Ejemplo:** Salto a un destino en otro fichero.
- Al ensamblar los valores de estas etiquetas **son desconocidos**.
- Sólo cuando se recolectan **todos los ficheros** que constituirán un ejecutable se pueden **resolver las referencias**.
- Cada módulo define un **conjunto de símbolos** y referencia a otro **conjunto de símbolos**.



Compilación de componentes de alto nivel

- El compilador tiene que traducir **todas las contrucciones del lenguaje** a lenguaje ensamblador.
- Respecto a los datos, el compilador sabe **su tamaño y su posición**.
- El compilador por tanto **traduce**:
 - la organización de datos en el lenguaje de alto nivel a **primitivas del ensamblador** (tales como `.asciz`, `.int`, etc).
 - las **construcciones del lenguaje** (`while`, `for`, etc) a instrucciones máquina.
 - las **llamadas a métodos o funciones**, a **llamadas a subrutinas**.
- El compilador sabe **el nombre, número y tipo** de los parámetros de todos los métodos así como **el tipo de resultado que devuelve**.

Programa en Lenguaje de alto nivel

```
if (condición) {  
    <cuerpo del then>  
} else {  
    <cuerpo del else>  
}
```

Programa en Ensamblador

```
    <Evaluación de condición>  
    <Resultado en eax>  
    CMP $0, %eax  
    JZ else  
then: <Código del Then>  
    JMP fin  
else: <Código del Else>  
fin: ...
```

Ejemplo de Traducción de un if-then-else

```
if (i == 3 && j > 4) {  
    Bloque_A  
} else {  
    Bloque_B  
}  
Bloque_C
```

- Supongamos **i** y **j** almacenados en memoria con etiquetas de ese nombre.
- La conjunción se evalúa **por partes**, y en cuanto una condición falla, se salta a la porción de código del **else**.

```
    cmp $3, i  
    jne Bloque_B  
    cmp $4, j  
    jle Bloque_B  
Bloque_A: ...  
    ...  
    jmp Bloque_C  
Bloque_B: ...  
    ...  
Bloque_C: ...  
    ...
```

Programa en Lenguaje de alto nivel

```
switch (condición) {  
  case 1:  
    <codigo caso 1>  
    break;  
  case 2:  
    <codigo caso 2>  
    break;  
  ...  
  default:  
    <codigo por defecto>  
    break;  
}
```

Programa en Ensamblador

```
cond: <Evaluación de condición>  
      <Resultado en eax>  
      CMP $1, %eax  
      JZ c1  
      CMP $2, %eax  
      JZ c2  
      ...  
def:  <código por defecto>  
      JMP fin  
c1:   <código para el caso 1>  
      JMP fin  
c2:   <código para el caso 2>  
      JMP fin  
      ...  
fin:  ...
```

¿Cómo traducir un bucle while?

Programa en Lenguaje de alto nivel

```
while (condición) {  
  <cuerpo del bucle>  
}
```

Programa en Ensamblador

```
cond: <Evaluación de condición>  
      <Resultado en eax>  
      CMP $0, %eax  
      JZ fin  
      <Código del Bucle>  
      JMP cond  
fin:  ...
```

Programa en Lenguaje de alto nivel

```
for (inicial; condición; incremento) {  
    <cuerpo del bucle>  
}
```

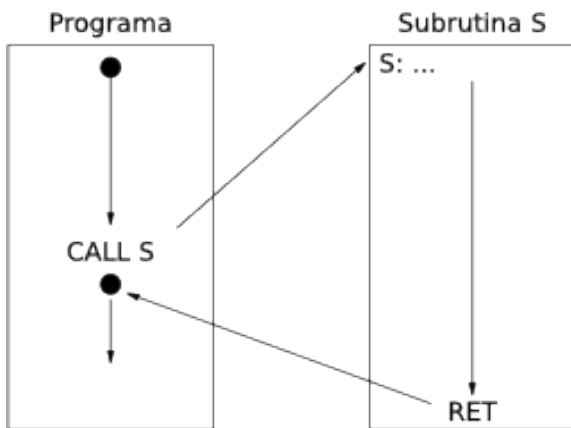
Programa en Ensamblador

```
    <Código inicial>  
cond: <Evaluación de condición>  
    <Resultado en eax>  
    CMP $0, %eax  
    JZ fin  
    <Cuerpo del Bucle>  
    <Código del incremento>  
    JMP cond  
fin:  ...
```

Subrutinas

- El desarrollo de código **modular** implica que hemos de ser capaces de invocar a porciones de código en otros módulos temporalmente.
- La unidad básica en estos módulos se denomina “**subrutina**”.
- Todo programa a partir de cierto tamaño mínimo consta de un **conjunto de subrutinas** independientemente del lenguaje en el que se ha escrito.
- **Subrutina:** Conjunto de instrucciones destinadas a cumplir un determinado fin que puede ser llamado a ejecutarse desde cualquier punto de un programa, desde otra subrutina o incluso desde ella misma.

- **Instrucción de llamada:** `call nombre`.



- **Instrucción de retorno:** `ret`.

- La instrucción de retorno no especifica **dónde** volver. El valor ha de quedar apuntado en algún sitio.

- Tanto la instrucción de llamada como la de retorno modifican el contenido del registro **EIP** (Extended Instructio Pointer) que es el **contador de programa**.

- Se necesita un mecanismo para **transmitir datos** entre las distintas subrutinas y el programa.

Ventajas e Inconvenientes de las Subrutinas

- Subrutinas = **métodos, procedimientos, funciones**, etc.

- Ventajas:

1. Se reduce el espacio de memoria que se precisa para un programa. Las rutinas se **ejecutan varias veces pero sólo se escriben una vez**.
2. Facilitan el trabajo de programación. Se puede **dividir la codificación** entre varios programadores.
3. Facilita la modificación, compilación y corrección de errores. Cada uno de estos procedimientos se puede hacer en los módulos **por separado**.

- Desventaja:

1. Cada subrutina supone **código adicional** para pasar parámetros, instrucción de llamada e instrucción de retorno.

Supongamos un programa que ejecuta las siguientes operaciones:

$$C = A * B \quad F = D * E$$

$$I = G * H \quad L = J * K$$

Programa A	
Código Preparación	→ 3 instrucciones
$C = A * B$	→ 15 instrucciones
Código Preparación	→ 3 instrucciones
$F = D * E$	→ 15 instrucciones
Código Preparación	→ 3 instrucciones
$I = G * H$	→ 15 instrucciones
Código Preparación	→ 3 instrucciones
$L = J * K$	→ 15 instrucciones
Código Preparación	→ 3 instrucciones
Instrucciones Ejecutadas	75
Instrucciones Escritas	75

Programa B	
Código Preparación	→ 3 instrucciones
CALL MUL(C, A, B)	→ 1 instrucción
Código Preparación	→ 3 instrucciones
CALL MUL(F, D, E)	→ 1 instrucción
Código Preparación	→ 3 instrucciones
CALL MUL(I, G, H)	→ 1 instrucción
Código Preparación	→ 3 instrucciones
CALL MUL(L, J, K)	→ 1 instrucción
Código Preparación	→ 3 instrucciones
Procedimiento MUL	→ 16 instrucciones
Instrucciones Ejecutadas	83
Instrucciones Escritas	25

La instrucción de llamada a subrutina

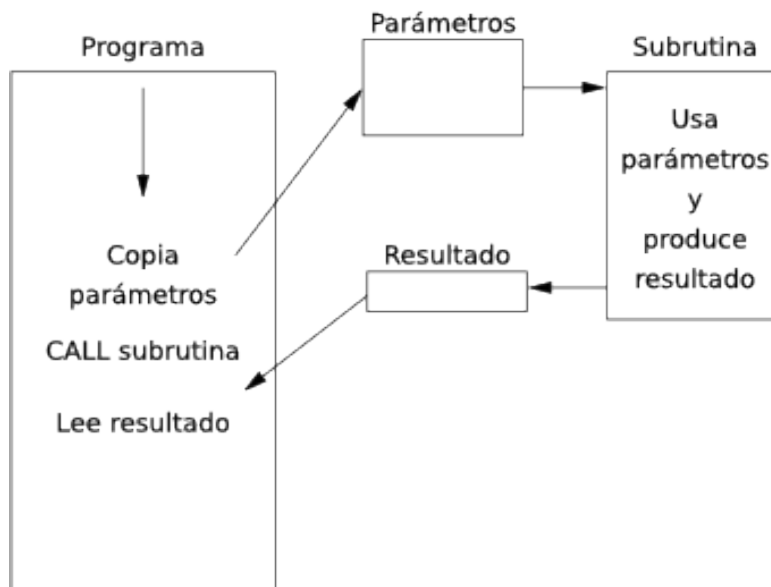
- En la arquitectura Pentium la instrucción de llamada a subrutina es **CALL destino**
- Donde destino es **una etiqueta del propio fichero** o **una etiqueta definida como global un fichero remoto**.
- La instrucción call **contiene operandos implícitos**.
- La dirección de retorno de la subrutina, que corresponde con el valor del contador de programa, **se almacenada automáticamente por el procesador en la cima de la pila**.
- Modifica el registro **%esp** de manera pertinente (igual que un push).
- El contador de programa se modifica de igual forma que **un salto incondicional**.

- La instrucción de retorno es **ret**
- Carece de operandos. Su efecto es **complementario** a la de llamada.
- El procesador **carga en el contador de programa el valor en la cima de la pila**
- Modifica el registro **%esp** de manera pertinente (igual que un pop).

Los parámetros o argumentos de las subrutinas

- Las subrutinas se utilizan **generalmente** para escribir código que se tiene que ejecutar **varias veces pero con diferentes valores en los datos**.
- **Ejemplo:** Código para imprimir un array de enteros. Si se permite cambiar la dirección de comienzo del array y su tamaño, el mismo código sirve para imprimir **cualquier array de cualquier tamaño**.
- Los **parámetros** son valores que la rutina recibe del programa que la invoca, y que ésta manipula en su código.
- Mientras se ejecuta la rutina **ésta conserva una copia de esos valores**.
- Por tanto, antes de realizar una llamada a subrutina **se deben depositar los parámetros necesarios** para su ejecución.

- En general una subrutina no sólo recibe unos datos, sino que **es posible** que tenga que devolver un resultado.
- Este resultado debe ser depositado en un lugar tal que pueda ser accedido **por el programa que ha invocado la rutina.**



Paso de parámetros a subrutinas

- Hay dos formas posibles de pasar los parámetros a una subrutina.
 1. **Por Valor:** Se envía a la subrutina los valores de los datos involucrados en la ejecución.
 2. **Por Referencia:** Se envían **las direcciones** donde se encuentran almacenados los datos.
- La **devolución de resultados** se puede también llevar a cabo de las dos maneras diferentes.
- Al margen del tipo de mecanismo para pasar o devolver datos, debemos seleccionar el **lugar** en el que se lleva a cabo esta comunicación. Para ello hay tres formas posibles:
 1. **A través de registros.**
 2. **A través de posiciones fijas de memoria.**
 3. **A través de la pila.**

- **Ventaja:** La **velocidad**. Los registros requieren un tiempo de acceso muy pequeño.
- **Inconveniente:** Hay un número limitado y generalmente reducido de registros. Incorrecto cuando se permiten **llamadas recursivas** (una subrutina se llama a sí misma).
- **Inconveniente:** La rutina no puede trabajar con todos los registros del procesador.
- Más común para **devolver resultados**, pues generalmente sólo hay un único resultado.

Programa		
MOV	p1, %eax	Primer parámetro
MOV	p2, %ebx	Segundo parámetro
MOV	p3, %ecx	Tercer parámetro
CALL	subrutina	
...		

Subrutina	
MOV	%eax, ?
MOV	%ebx, ?
MOV	%ecx, ?
...	
RET	

Paso de parámetros a través de posiciones fijas de memoria

- **Ventajas:** Los parámetros están siempre en el mismo sitio.
- **Inconveniente:** Tiene que ser accesible tanto por el programa que llama como por la subrutina.
- **Inconveniente:** Incorrecto cuando se permiten **llamadas recursivas** (una subrutina se llama a sí misma).

Programa		
MOV	%eax, p1	Primer parámetro
MOV	%ebx, p2	Segundo parámetro
MOV	%ecx, p3	Tercer parámetro
CALL	subrutina	
...		

Subrutina	
MOV	p1, ?
MOV	p2, ?
MOV	p3, ?
...	
RET	

- Esta técnica trata de **solucionar los inconvenientes** de los dos métodos anteriores y **enfatar en las ventajas**.
- **Ventaja:** La pila es accesible tanto para el programa que llama como para la subrutina.
- **Ventaja:** Aunque la pila tiene tamaño limitado, se pueden almacenar más datos que en los registros.
- **Ventaja:** La pila ofrece el tipo de datos idóneo para el **anidamiento de llamadas**.
- **Inconveniente:** Se precisan unas reglas muy precisas a respetar por el programa que llama a la subrutina y por la subrutina acerca de la localización de los parámetros y resultado.
- Este es el método **más utilizado** por la mayoría de procesadores, el que se utiliza en las prácticas y el que se exige en el examen de la asignatura.

Cómo acceder a los parámetros almacenados en la pila

- Una vez almacenados los parámetros en la pila y efectuada la llamada, la subrutina se encuentra **con la dirección de retorno almacenada encima de los parámetros en la pila**.
- ¿Cómo acceder a los **parámetros**?
- El único registro a utilizar es el propio **puntero de pila** o **%esp**
- El valor de este puntero **puede fluctuar** debido a que en la pila se depositan datos temporalmente.
- Para garantizar que el acceso a los parámetros se realiza **siempre igual** se utiliza un registro como referencia fija a una porción de la pila.
- Generalmente, la primera instrucción de una subrutina es asignar al registro **%ebp** el valor actual del puntero de pila.
- Como el propio **%ebp** debe restaurarse a su valor inicial, previo a este paso se almacena él mismo en la pila.
- Utilizando el modo de direccionamiento Base + Desplazamiento y el registro **%ebx** los parámetros **se acceden de forma idéntica** en todo el código de la subrutina.

Por parte del programa que **llama** a la subrutina:

1. Guardar **opcionalmente** espacio en la pila para almacenar los resultados.
2. Cargar los parámetros en **cierto orden** en la pila.
3. Efectuar la **llamada** a subrutina.
4. **Descargar** los parámetros de la pila.
5. **Descargar** opcionalmente el resultado e la pila.

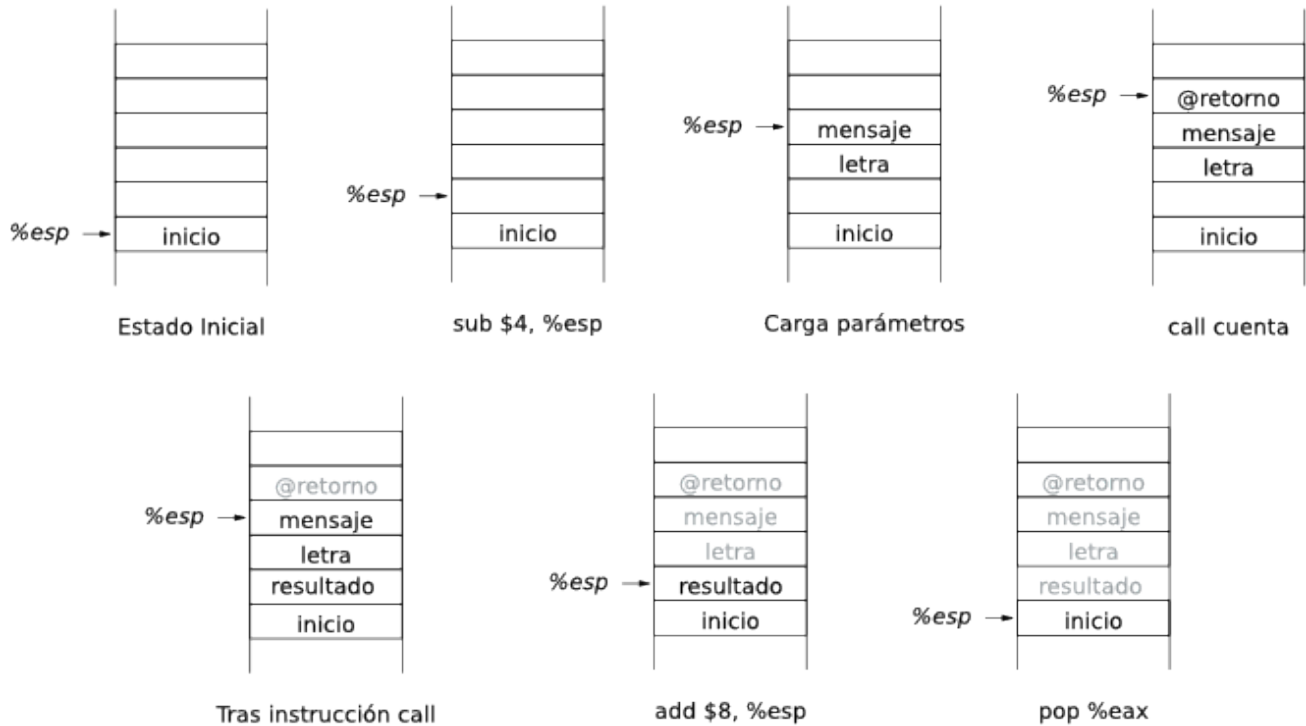
Por parte de la **subrutina**:

1. Opcionalmente, salvar el **puntero a la pila** para poder referenciar a los parámetros.
2. Reservar espacio en la pila para **datos locales**.
3. Opcionalmente, salvar en la pila los **valores de los registros** que se vayan a utilizar.
4. Ejecutar el **código** de la subrutina.
5. **Depositar resultado** en el espacio reservado a tal efecto (ya sea en la pila o en registros).
6. **Restaurar** los valores de los registros si se han salvado.
7. Dejar la pila tal y como estaba, es decir, descargar los datos locales y restaurar el valor del puntero de pila.
8. Efectuar el **retorno** de subrutina.

Llamada desde el programa

- El programa va a recibir el resultado a través de **la pila**.
- La rutina recibe **tres** parámetros.
- Los parámetros **no se reutilizan** al acabar de ejecutar la rutina.
- El resultado se **descarga** sobre el registro `%eax`.

	Programa
<code>main: SUB \$4, %esp</code>	Espacio para almacenar el resultado
<code>PUSH \$1</code>	Primer parámetro
<code>PUSH %eax</code>	Segundo parámetro
<code>PUSH \$mensaje</code>	Segundo parámetro
<code>CALL sbrt</code>	Llamada a subrutina
<code>ADD \$12, %esp</code>	Restaurar la pila
<code>POP %eax</code>	Obtener resultado



Llamada a la subrutina

- La **dirección de retorno** está en la cima de la pila.

Subrutina

sbtrt: PUSH	<code>%ebp</code>	Salvamos <code>%ebp</code>
MOV	<code>%esp, %ebp</code>	Puntero a parámetros
SUB	<code>\$8, %esp</code>	Para datos locales
PUSH	<code>%eax</code>	Salvar registros
PUSH	<code>%ebx</code>	
...		Código adicional
MOV	<code>8(%ebp), ?</code>	Mover 1 ^{er} parámetro
MOV	<code>12(%ebp), ?</code>	Mover 2 ^o parámetro
MOV	<code>16(%ebp), ?</code>	Mover 3 ^{er} parámetro
...		Código adicional
MOV	<code>?, 20(%ebp)</code>	Escritura de resultado
POP	<code>%ebx</code>	Restaurar registros
POP	<code>%eax</code>	
MOV	<code>%ebp, %esp</code>	Restauramos la pila
POP	<code>%ebp</code>	Restauramos <code>ebp</code>
RET		Retorno

- Durante la ejecución, el valor de `%ESP` **fluctúa**.

- Se utiliza `%ebp` como **registro base** para acceder a los parámetros.

- Al final de la rutina la pila quedar con la **dirección de retorno** en la cima.

