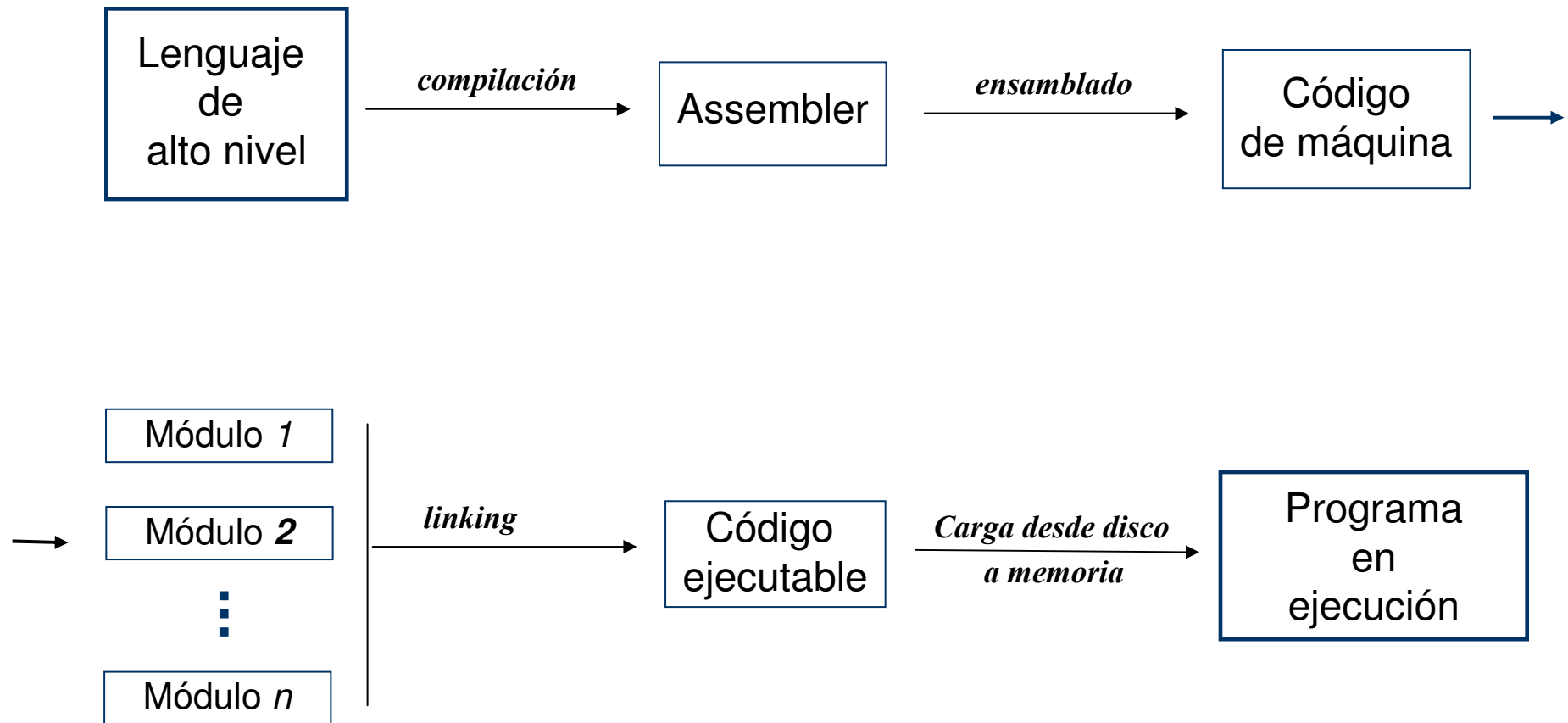


66.70 Estructura del Computador

Lenguajes y código de máquina

Desde el diseño a la ejecución



Algunos tipos de compiladores

- **De una sola pasada**

Completa el proceso en un solo recorrido del programa fuente

- **De pasadas múltiples**

- **Compilador incremental**

Genera código instrucción por instrucción (no para todo el programa) cuando el usuario pulsa una tecla. Entorno de depuración. Intérpretes vs. Compiladores.

- **Cross-compiler**

Genera código en lenguaje assembler para una máquina diferente de la que se está utilizando para compilar.

- **Descompilador**

Analiza código máquina y lo traduce a un lenguaje de alto nivel, realizando el proceso inverso a la compilación.

Compilación

Análisis léxico

Reducir el texto fuente a identificadores y palabras del lenguaje

Análisis sintáctico (“*parsing*”)

Identificar “ $A=B+4$ ” como una estructura

“*Identificador1 = Expresión*” con “*expresion: Identificador2 + 4*”

Análisis de nombres

Identificar “ A ” y “ B ” como variables del programa y asociadas con ubicaciones de memoria.

Análisis de tipo

Determinar el tipo de valores que guarda cada variable.

Mapeo de acciones y generación de código

Asociar cada sentencia de programa con una apropiada secuencia en lenguaje assembler.

! Sentencia de asignación

ld [B], %r0, %r1

add %r1, 4, %r2

st %r2, %r0, [A]

! guardar variable B en un registro

! calcular el valor de la expresión

! hacer la asignación del valor

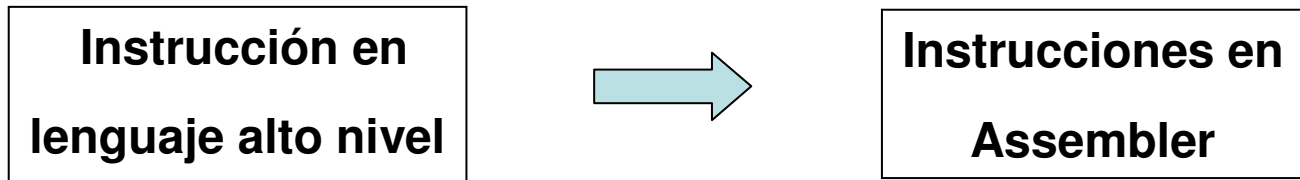
Optimización del código

Mayor eficiencia en velocidad o uso de memoria

Independiente del procesador
 (“front end”)

Dependiente del procesador
 (“back end”)

Mapeo de instrucciones



- **Movimiento de datos**
- **Operaciones aritméticas**
- **Control de flujo del programa**

Mapeo de acciones

Almacenamiento de variables en memoria

❖ Variables estáticas (globales)



=> posición en memoria conocida en tiempo de compilación

❖ Variables locales

=> aparecen y desaparecen cuando termina el procedimiento en que están declaradas => se almacenan en el stack

Mapeo de acciones

Almacenamiento de estructuras de datos

Estructura	Array en C	Array en Pascal
<pre>struct point { int x; int y; int z; }</pre> <pre>struct point pt;</pre>	<pre>int A[10];</pre> <pre>A[i]</pre>	<pre>A: array [-10..10] of integer</pre> <pre>A[i]</pre>
	<pre>ElementAddress = BASE + INDEX * SIZE</pre>	<pre>ElementAddress = BASE + (INDEX - START) * SIZE</pre>
<pre>ld [pt + 4], %r1 ! pt.y</pre>  <pre>! → %r1;</pre>	<pre>SIZE=4, INDEX in %r3</pre> <pre>sll %r3, 2, %r3 !%r3 * 4 → %r3</pre> <pre>ld [A + %r3], %r1 !valor → %r1</pre> 	<pre>SIZE=4, INDEX en %r3, START en %r4</pre> <pre>sub %r3, %r4, %r6 !INDEX-START → %r6</pre> <pre>sll %r6, 2, %r6 ! %r6 * 4 → %r6</pre> <pre>ld [A + %r6], %r1 ! valor → %r1</pre>

Puntero dentro de la estructura es conocido en tiempo de ensamblado

Puntero dentro de la estructura puede ser conocido en tiempo de ensamblado o en tiempo de ejecución

Mapeo de acciones

Operaciones aritméticas

- ❖ Modos de direccionamiento para los operadores
- ❖ En máquinas RISC: operandos siempre en registros
 - Cantidad de registros
 - Cuando es excedida => registros al stack
(“register spilling” o “desborde de registros”)
 - El compilador debe decidir cuáles registros contienen valores que ya no son necesarios. Técnica: “register coloring” o “graph-coloring”

Mapeo de acciones

Control del flujo

Goto statement	ba label
if A=B stmt1 else stmt2;	subcc %r1, %r2, %r0 ! setea flags, descarta resultado bne else1 !... código de stmt1 ba fin else1: !código de stmt2 fin: ! ...
while (r1 == r2) %r3 = %r3 + 1;	ba Test True: add %r3, 1, %r3 Test: subcc %r1, %r2, %r0 be True
Do r3 =r3 + 1 while (r1 == r2)	True: add %r3, 1, %r3 Test: subcc %r3, %r2, %r0 be True

Ensamblador

- ❖ Relación 1 a 1 entre código Assembler y código de máquina
- ❖ Ofrece al programador
 - Representación simbólica para direcciones y constantes
 - Definir la ubicación de las variables en memoria
 - Variables inicializadas antes de ejecución
 - Provee cierto grado de aritmética en tiempo de ensamblado
 - Utilizar variables declaradas en otros módulos
 - Utilizar macros (dar nombre a fragmentos de texto)

Ensamblador

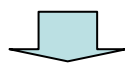
Assembler a código de máquina

! This program adds two numbers

```
.begin
.org 2048
main: ld    [x], %r1      ! Load x into %r1
      ld    [y], %r2      ! Load y into %r2
      addcc %r1, %r2, %r3 ! %r3 ← %r1 + %r2
      st    %r3, [z]      ! Store %r3 into z
      jmp1  %r15 + 4, %r0 ! Return

x:    15
y:    9
z:    0
.end
```

```
ld    [x], %r1
ld    [y], %r2
```



```
11 00001 000000 00000 1 0100000010100
op  rd  op3  rs1  i  simm13

11 00010 000000 00000 1 0100000011000
op  rd  op3  rs1  i  simm13
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1 1		rd				op3				rs1				0 0 0 0 0 0 0 0 0				rs2													
1 1		rd				op3				rs1				1				simml3													

Formato de acceso a memoria: $op=11$

$op3$	direcc. de memoria (ld o st)	
000000 ld rd=reg.destino	$= rs1 + rs2$	si i = 0
000100 st rd=reg.origen	$= rs1 + \text{simm13 (constante)}$	si i = 1

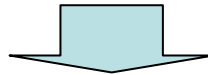
Ensamblador

Assembler a código de máquina (cont.)

```
! This program adds two numbers

    .begin
    .org 2048
main: ld     [x], %r1          ! Load x into %r1
      ld     [y], %r2          ! Load y into %r2
      addcc  %r1, %r2, %r3     ! %r3 ← %r1 + %r2
      st     %r3, [z]          ! Store %r3 into z
      jmp1   %r15 + 4, %r0     ! Return

x:    15
y:    9
z:    0
      .end
```

[illegible]

Ensamblador

❖ Primer pasada

- Detecta identificadores y les asigna una posición de memoria
- Crea la tabla de símbolos

❖ Segunda pasada

- Genera programa objeto y el listado de assembler
- Cada línea es procesada completamente antes de avanzar a la siguiente
- Cada instrucción es convertida a código de máquina
- Cada identificador es reemplazado por su ubicación en memoria según indica la tabla de símbolos

❖ La doble pasada permite la referencia adelantada o “*forward referencing*”

```
      .  
      .  
      .  
      call sub_r      ! Subroutine is invoked here  
      .  
      .  
sub_r:  .  
      st    %r1, [w]    ! Subroutine is defined here  
      .  
      .
```

Crear la tabla de símbolos

(Primer pasada del assembler)

```
! This program sums LENGTH numbers
! Register usage:      %r1 - Length of array a
!                      %r2 - Starting address of array a
!                      %r3 - The partial sum
!                      %r4 - Pointer into array a
!                      %r5 - Holds an element of a

        .begin          ! Start assembling
        .org 2048        ! Start program at 2048
a_start .equ 3000        ! Address of array a
        ld [length], %r1 ! %r1 ← length of array a
        ld [address], %r2 ! %r2 ← address of a
        andcc %r3, %r0, %r3 ! %r3 ← 0
loop:   andcc %r1, %r1, %r0 ! Test # remaining elements
        be done          ! Finished when length=0
        addcc %r1, -4, %r1 ! Decrement array length
        addcc %r1, %r2, %r4 ! Address of next element
        ld %r4, %r5       ! %r5 ← Memory[%r4]
        addcc %r3, %r5, %r3 ! Sum new element into r3
        ba loop          ! Repeat loop.

done:   jmp1 %r15 + 4, %r0 ! Return to calling routine

length:      20          ! 5 numbers (20 bytes) in a
address:     a_start
        .org a_start    ! Start of array a
a:          25          ! length/4 values follow
           -10
           33
           -5
           7

        .end            ! Stop assembling
```

(1)

Símbolo	Valor
a_start	3000
length	-
address	-
loop	2060
done	-

1er pasada del assembler

(2)

Símbolo	Valor
a_start	3000
length	2092
address	2096
loop	2060
done	2088

❖ Contador de posición (análogo a un *program counter* en tiempo de ensamblado)

- Inicializado a cero al inicio de la primer pasada y por cada directiva .org
- Incrementado por cada instrucción según su tamaño (en ARC 4 bytes)

Creación del código objeto y listado

(Segunda pasada del assembler)

Location counter	Instruction	Object code
	.begin	
	.org 2048	
	a_start .equ 3000	
2048	ld [length],%r1	11000010 00000000 00101000 00101100
2052	ld [address],%r2	11000100 00000000 00101000 00110000
2056	andcc %r3,%r0,%r3	10000110 10001000 11000000 00000000
2060 loop:	andcc %r1,%r1,%r0	10000000 10001000 01000000 00000001
2064	be done	00000010 10000000 00000000 00000110
2068	addcc %r1,-4,%r1	10000010 10000000 01111111 11111100
2072	addcc %r1,%r2,%r4	10001000 10000000 01000000 00000010
2076	ld %r4,%r5	11001010 00000001 00000000 00000000
2080	ba loop	00010000 10111111 11111111 11111011
2084	addcc %r3,%r5,%r3	10000110 10000000 11000000 00000101
2088 done:	jmp1 %r15+4,%r0	10000001 11000011 11100000 00000100
2092 length:	20	00000000 00000000 00000000 00010100
2096 address:	a_start	00000000 00000000 00001011 10111000
	.org a_start	
3000 a:	25	00000000 00000000 00000000 00011001
3004	-10	11111111 11111111 11111111 11110110
3008	33	00000000 00000000 00000000 00100001
3012	-5	11111111 11111111 11111111 11111011
3016	7	00000000 00000000 00000000 00000111
	.end	

Listado

Símbolo	Valor
a_start	3000
length	2092
address	2096
loop	2060
done	2088

Tabla de símbolos

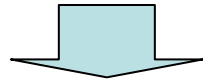
Información adicional que el ensamblador incluye en el módulo objeto

- Dirección de la primer instrucción a ejecutar (si corresponde): *main()*
- Símbolos declarados en otros módulo: *externos*
- Símbolos globales: *accesibles desde otros módulos*
- *Librerías* externas que son utilizadas por el módulo
- Información sobre la *relocalización* del código

Localización del programa en memoria

- En general no se sabe donde va a ser cargado el programa
=> ~~“.org 2048”~~
- Si varios módulos son vinculados no se sabe donde va a ser cargado el programa

~~=> “.org 2048”~~



Código relocizable

- El ensamblador es responsable de marcar direcciones relocizables y direcciones absolutas
- Esta información es necesaria por el linker

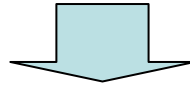
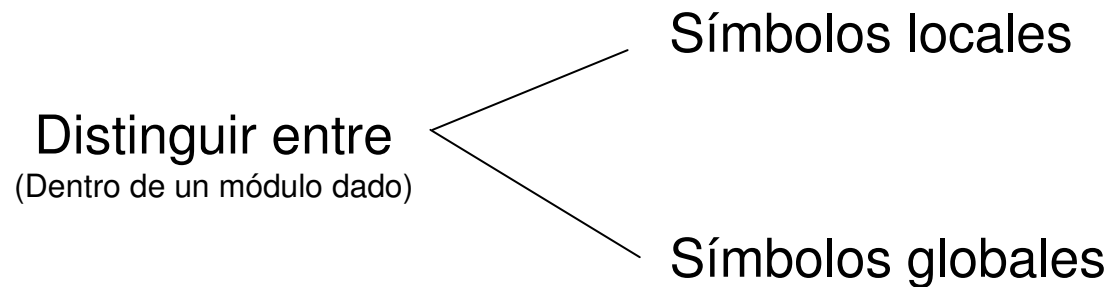
EL LINKER

Combina dos o más módulos que fueron ensamblados separadamente

Para ello

- Resuelve referencias de memoria externa al módulo
- Relocaliza los módulos combinándolos y reasignando las direcciones internas a cada uno para reflejar su nueva localización
- Define en el módulo a cargar la dirección de la primer instrucción a ser ejecutada (“main”)

Referencias externas



Directivas al ensamblador

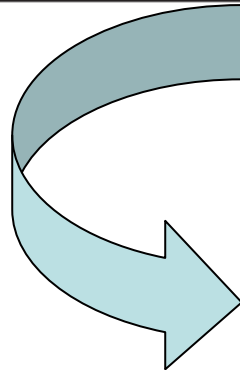
<i>.global</i>	→	declara un símbolo global
<i>.extern</i>	→	utiliza símbolo declarado en otro módulo

- Declarar un ".equ" como global no tiene sentido ¿Porqué?

Referencias externas

```
! Main program
    .begin
    .org 2048
    .extern sub
main: ld    [x], %r2
      ld    [y], %r3
      call  sub
      jmp1  %r15 + 4, %r0
x: 105
y: 92
    .end
```

```
! Subroutine library
    .begin
ONE .equ 1
    .org 2048
    .global sub
sub: ornc  %r3, %r0, %r3
      addcc %r3, ONE, %r3
      jmp1  %r15 + 4, %r0
    .end
```



Symbol	Value	Global/ External
sub	-	External
main	2048	No
x	2064	No
y	2068	No

Main Program

Symbol	Value	Global/ External
ONE	1	No
sub	2048	Global

Subroutine Library

INCOMPLETAS
(sigue en próx transparencia)

Símbolos relocizables

TIEMPO DE ENSAMBLADO

<pre>! Main program .begin .org 2048 .extern sub main: ld [x], %r2 ld [y], %r3 call sub jmp1 %r15 + 4, %r0 x: 105 y: 92 .end</pre>	<pre>! Subroutine library .begin ONE .equ 1 .org 2048 .global sub sub: orncc %r3, %r0, %r3 addcc %r3, ONE, %r3 jmp1 %r15 + 4, %r0 .end</pre>
---	--

TIEMPO DE LINKING

~~Addr(Main) = Addr(Sub) = 2048~~



Direcciones relocizables

- **Ensamblador**

Marca direcciones como relocizables o no relocizables (absolutas)

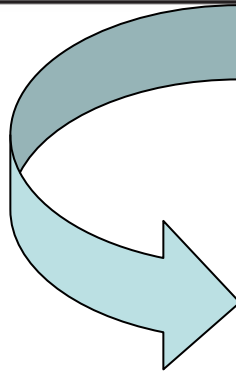
- **Linker**

Redefine las direcciones relocizables a partir de la nueva dirección de origen

Símbolos relocalizables

```
! Main program
    .begin
    .org 2048
    .extern sub
main: ld    [x], %r2
      ld    [y], %r3
      call sub
      jmp1  %r15 + 4, %r0
x: 105
y: 92
    .end
```

```
! Subroutine library
    .begin
ONE .equ 1
    .org 2048
    .global sub
sub: ornc  %r3, %r0, %r3
      addcc %r3, ONE, %r3
      jmp1  %r15 + 4, %r0
    .end
```



Symbol	Value	Global/ External	Reloc- atable
sub	-	External	-
main	2048	No	Yes
x	2064	No	Yes
y	2068	No	Yes

Main Program

Symbol	Value	Global/ External	Reloc- atable
ONE	1	No	No
sub	2048	Global	Yes

Subroutine Library

Símbolos relocizables

EL ENSAMBLADOR

- ✓ Determina qué símbolos son relocizables y cuáles no
- ✓ Los marca en el módulo ensamblado
- ✓ Identifica código que debe ser modificado como resultado de la relocización

> No tiene sentido marcar como relocizables símbolos declarados en otros módulos

No son relocizables:

- ✓ constantes declaradas por medio de un .equ
- ✓ constantes indicadas como contenido en memoria (p.e.: x=**105**, y=**92**)
- ✓ *Direcciones de entrada/salida*
- ✓ *Rutinas del sistema*

Sí son relocizables:

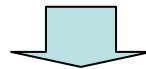
- ✓ Posiciones de memoria relativas a un .org (p.e.: x o y) (**no** su contenido)

Carga del programa en memoria

El loader toma el programa de disco y lo carga en memoria principal, para ello:

- *Carga diferentes segmentos de memoria con los valores apropiados*
- *Inicializa el stack pointer a su valor inicial*
- *Inicializa el program counter a su valor inicial (=salta a la primera instrucción)*

- En ambientes multitarea este modelo no funciona, ya que:
 - El assembler no puede saber donde puede ser cargado
 - El linker no puede saber donde puede ser cargado



El loader debe relocalizar todos los símbolos relocalizables
=> No existe una división clara entre el linker y el loader

- *Loaders relocalizadores* (direcciones absolutas vs. Dir. referidas a segmento base)
- *Linkers dinámicos (loaders)* (librerías estáticas vs. librerías dinámicas)

Archivos objeto

- ✓ Archivo objeto *relocateable* : código binario y datos en un formato que permite combinarlo con otros archivos objeto relocateables (linking)
 - ✓ Archivo objeto *ejecutable*: código binario y datos en un formato que permite cargarlo directamente a memoria y ejecutarlo
 - ✓ Archivo objeto *compartido*: tipo especial de archivo objeto relocateable que puede ser cargado en memoria y vinculado dinámicamente
-
- El formato de los archivos objeto es dependiente del sistema donde van a ejecutarse