UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Estructura de Computadores
Tema 3: Traducción de Programas (cont)

Agustín Fernández

Departament d'Arquitectura de Computadors

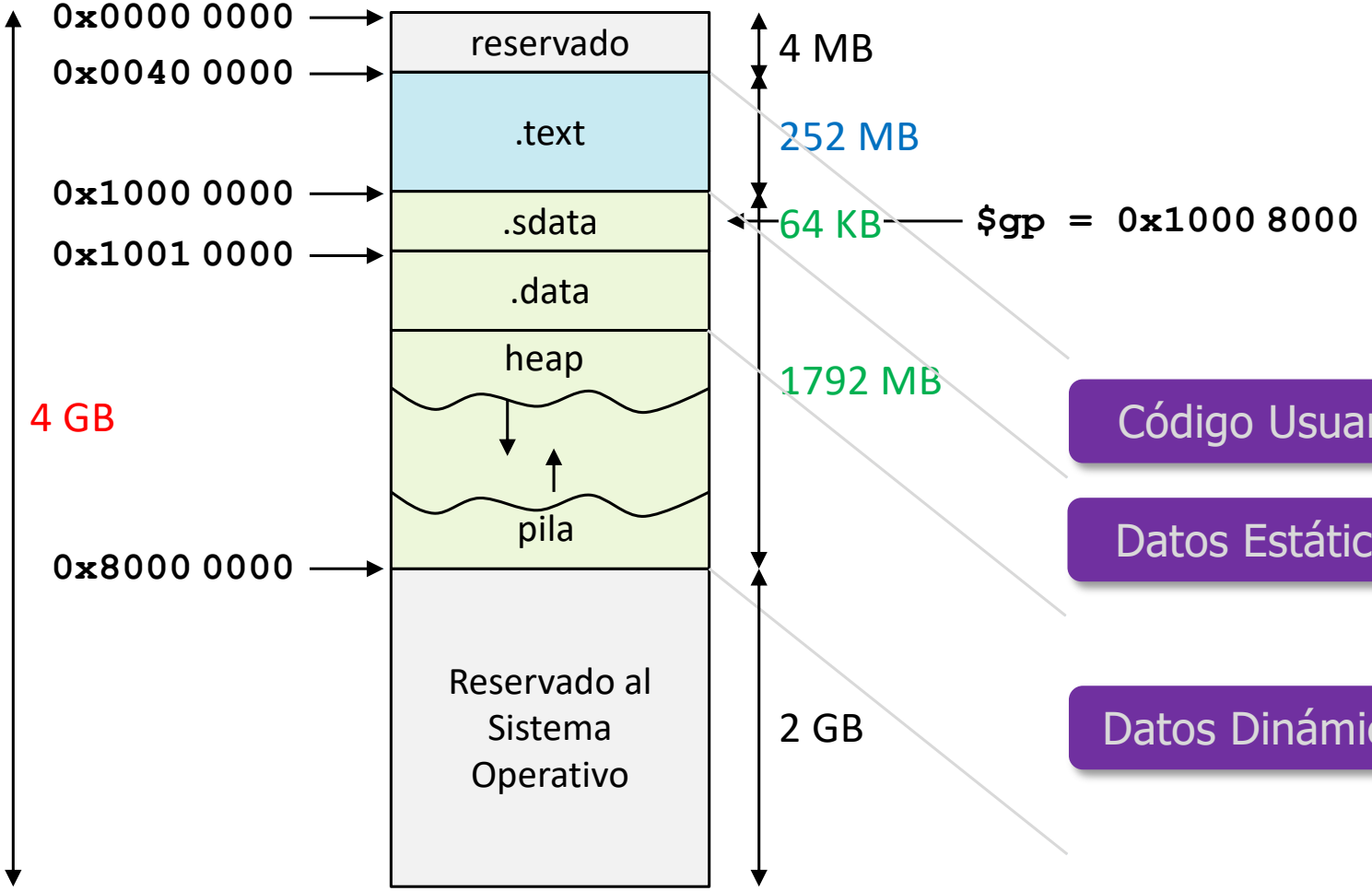
Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

Índice

- ...
- Estructura de la Memoria
- Compilación, ensamblado, enlazado y carga

Estructura de la Memoria en MIPS



Todos los programas tienen el mismo mapa de memoria.

Código Usuario

Datos Estáticos

Datos Dinámicos

Almacenamiento Estático

- La sección **.data**
  - Tamaño fijo para cada programa
  - Guarda las variables globales de C. Las definidas fuera de cualquier función.
  - Ocupan el mismo lugar durante toda la ejecución del programa.

```c
int v[5];
int a[3] = {1, 2, 3};
short b = -5;
char c = 'a';

void main() {
   ...
}
```

C

```
    .data
    .align 2
v:  .space 20
a:  .word 1, 2, 3
b:  .half -5
c:  .byte 'a'

    .text
main:
    ...
```

MIPS

Almacenamiento Estático

- Acceso a variables globales de la sección **`.data`**
  - Necesitamos cargar la dirección en un registro, usamos "**3 instrucciones**"

```
la $t0, etiquetaVarGlobal
lw $t1, 0($t0)
```

`la` es una macro que se traduce a 2 instrucciones

- La sección **`.sdata`** (small data)
  - Tamaño: 2^{16} bytes (64 KB) [Muy pequeño]
  - Se utiliza para guardar variables globales en C
  - El registro **`$gp`** (**global pointer**) apunta a una posición fija al medio de **`.sdata`**
  - Podemos acceder a las variables globales con **1 instrucción**, a offsets<2^{15} del **`$gp`**

```
lw $t1, offsetVarGlobal($gp)
```

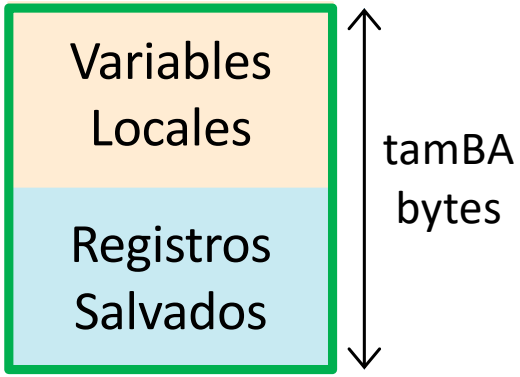
Almacenamiento Dinámico

Almacenamiento dinámico: La variable ocupa un espacio de memoria temporal

- **La sección Pila (stack)**
  - Guarda el Bloque de Activación de las subrutinas: variables locales y registros seguros.
  - Crece hacia direcciones bajas
  - Crece/decrece dinámicamente. Reservar/Liberar espacio al inicio/final de las subrutinas.

```
int func(void) {
  int v[50];
  char x[100];

  ...
}
```




```
func:
  addiu $sp,$sp,-tamBA

  ...

  addiu $sp,$sp,tamBA
  jr $ra
```

Almacenamiento Dinámico

Almacenamiento dinámico: La variable ocupa un espacio de memoria temporal

- **La sección `.heap`**
  - Guarda las variables que se crean y destruyen explícitamente.
  - Crece hacia direcciones altas
  - Reservar/Liberar espacio con llamadas al SO: **`malloc()`** y **`free()`**

```
int main(void) {
  int *ptr, numB;
  ...
  numB = 1000*sizeof(int);
  ptr = malloc(numB);
  ...
  free(ptr);
}
```

```
ptr = malloc(numB);
if (ptr == NULL)
  error&exit
```

```
free(ptr);
ptr = NULL;
```

¡CONTROL de ERRORES!

¿Dónde se almacenan las diferentes variables?

```
int gvec[100];
int *pvec;

int f() {
  int lvec[100];
  ...
  pvec = malloc(1000);
  ...
  pvec[9] = gvec[9] + lvec[9];
  ...
}
int g() {
  ...
  free(pvec);
  ...
}
```

Global, datos estáticos

Local, datos dinámicos, estarán en el BA de f en la **pila**

Global, datos dinámicos, reservamos espacio del **heap**

Liberamos espacio en la **pila**

Liberamos espacio en el **heap**

Examen final Jun/2022 P5 (1 punto)

En un programa MIPS les dades que s'utilitzen poden situar-se en 4 llocs ben diferenciats: .data, .heap, pila i registres.

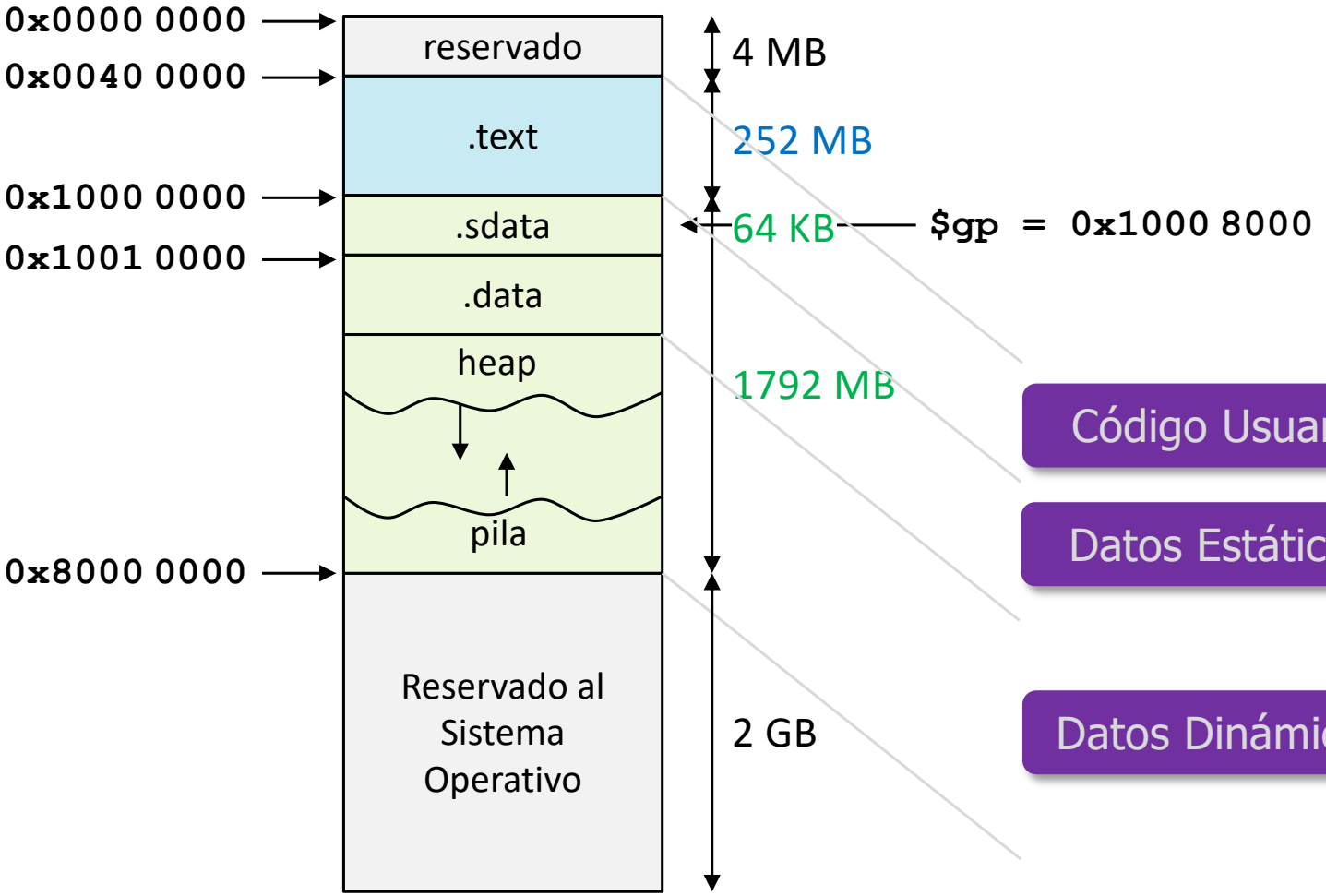
Donat el següent programa escrit en C:

```
int VG[100];
int *pG, sum;
int Test() {
  int VL[100];
  int *pL,*pD;
  ...
  pD = malloc(400);
  ...
}
```

```
... = VG[5];
... = pL;
... = *(pD+20);
... = VL[6];
... = pD;
... = sum;
... = pG;
```

On són (.data, heap, pila o registres) les variables següents, suposant que els accessos indicats es fan dins la rutina Test?

Estructura de la Memoria en MIPS



Todos los programas tienen el mismo mapa de memoria.

Código Usuario

Datos Estáticos

Datos Dinámicos

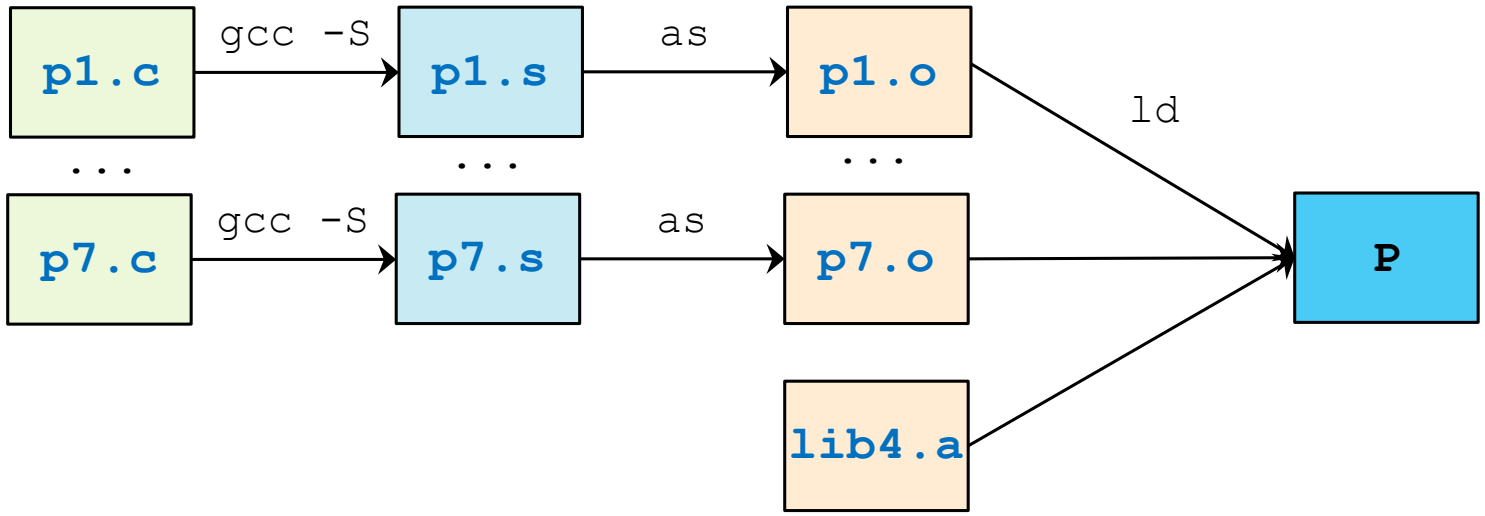
Compilación, Ensamblado, Enlazado y Carga

Pasos a seguir para ejecutar un programa:

- **Compilación**
  - Traduce un código fuente escrito en un Lenguaje de Alto Nivel a Ensamblador.
- **Ensamblado**
  - Traduce de un código escrito en Ensamblador a Lenguaje Máquina (**fichero objeto**).
- **Enlazado**
  - Normalmente en una aplicación tenemos múltiples ficheros objeto. El **enlazador** (**linker**) se encarga de generar un único **fichero ejecutable.** A veces al enlazador se le llama **montador**.
- **Carga**
  - Para **ejecutar** un programa hay que copiar en memoria el **fichero ejecutable**, de eso se encarga el **cargador** (**loader**) que es una parte del **Sistema Operativo**.

Compilación Separada

- Estructurar el código en diversos módulos:
  - Facilita la gestión de proyectos complejos y fomenta la reutilización de código (**libraries**)
  - Sólo requiere compilación parcial cuando se modifica un módulo.
- Los módulos (ficheros) se pueden compilar y ensamblar por separado.
- Los diferentes módulos se enlazan para generar el fichero ejecutable.



¿Qué puede haber en un file.s?

file.s

```
.data
...

.text
...
```

```
.data
    .globl b
a:  .word -1
b:  .space 400
c:  .half 3,4,5
```

```
.text
.globl f1,f3
f1: ...
    jr $ra
f2: ...
    la $a0,b
    jal f1
    ...
    jr $ra
f3: ...
    la $a0,d
    jal g1
    ...
    jr $ra
```

- En `file.s` se puede:
  - Acceder a: a, b, c
  - Llamar a las subrutinas: f1, f2, f3
- La etiqueta b, es global, accesible desde otros módulos.
- Las subrutinas f1, f3 son globales, se pueden llamar desde otros módulos.
- En `file.s` usamos la etiqueta d, y llamamos a g1 (no están definidas en `file.s`), han de estar declaradas como globales en otros módulos.

Ensamblado (en 2 pasadas)

Primera Pasada

- Expandir Macros
- Generar una tabla de símbolos con la dirección de cada etiqueta
  - No pueden ser direcciones definitivas, son relativas (desplazamientos, offsets) a la sección dónde está definida la etiqueta.
- Generar una **tabla de símbolos globales** para las etiquetas declaradas con la directiva `.globl`

Ensamblado (en 2 pasadas)

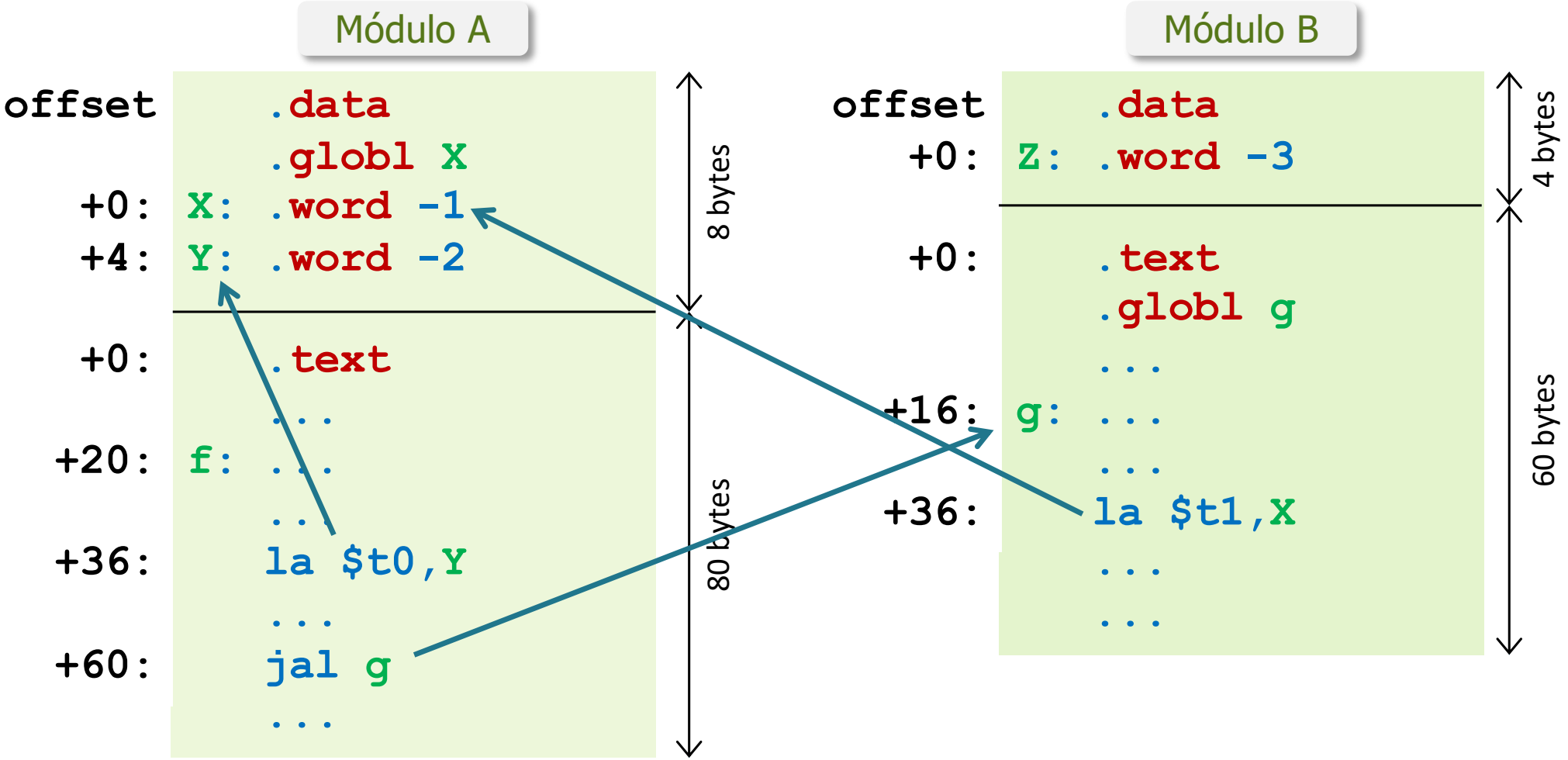
Segunda Pasada, codificar instrucciones en Lenguaje Máquina

- Las instrucciones que utilizan direcciones relativas al PC (`bne`, `beq`)
  - Codificar con el offset que figura en la tabla de símbolos
- Las instrucciones que utilizan direcciones absolutas (la, j, jal)
  - Codificar provisionalmente con un cero, lo resolverá el enlazador (**reubicación**).
  - Si la etiqueta ESTÁ en la tabla de símbolos:
    - ✓ Añadir la instrucción a una **Lista de Reubicación**, especificando posición, tipo, dirección provisional
  - Si la etiqueta NO ESTÁ (p.e. porque está en la tabla de símbolos globales de otro módulo):
    - ✓ Añadir la instrucción a una **Lista de Referencias Externas no Resueltas**

Ensamblado (en 2 pasadas)

- Al acabar el ensamblado el fichero objeto resultante (**`xxx.o`**) en UNIX contiene:
  - Cabecera (una especie de índice del fichero)
  - Código Máquina (sección **`.text`**)
  - Datos Estáticos Globales (sección **`.data`**)
  - **Lista de Reubicación**
    - ✓ Posición de la instrucción, tipo y dirección provisional
  - **Tabla de Símbolos** Globales
  - **Lista de Referencias no Resueltas**
  - Información de depuración (debugging): p.e. números de línea del código fuente

Ensamblado. Ejemplo con 2 módulos



Ensamblado. Ejemplo con 2 módulos

Módulo A

```
    .data
    .globl X
X:  .word -1
Y:  .word -2

    .text
    ...
f:  ...
    ...
    la $t0,Y
    ...
    jal g
    ...
```

Módulo B

```
    .data
Z:  .word -3

    .text
    .globl g
    ...
g:  ...
    ...
    la $t1,X
    ...
    ...
```

Símbolos globales
Símbolos etiquetados con `.globl`, son visibles [utilizables] desde todos los módulos.
En el ejemplo: `X` y `g`

Ensamblado. Ejemplo con 2 módulos

Módulo A

```
    .data
    .globl X
X:  .word -1
Y:  .word -2
```

```
    .text
    ...
f:  ...
    ...
    la $t0,Y
    ...
    jal g
    ...
```

Módulo B

```
    .data
Z:  .word -3
```

```
    .text
    .globl g
    ...
g:  ...
    ...
    la $t1,X
    ...
    ...
```

Referencias que NO se pueden resolver

- En el módulo A, la llamada a `g`
- En el módulo B, el acceso a `x`

No se pueden resolver porque están definidas en otro módulo (referencias cruzadas) y no tenemos ni idea de dónde están.

Ensamblado. Ejemplo con 2 módulos

Módulo A

```
    .data
    .globl X
X:  .word -1
Y:  .word -2
```

```
    .text
    ...
f:  ...
    ...
    la $t0,Y
    ...
    jal g
    ...
```

Módulo B

```
    .data
Z:  .word -3
```

```
    .text
    .globl g
    ...
g:  ...
    ...
    la $t1,X
    ...
    ...
```

Reubicación

El acceso a `Y` en el módulo A es a una variable conocida. Pero, la dirección definitiva de `Y` no se sabrá hasta que se genere el ejecutable. La dirección de `Y` hay que reubicarla.

Ensamblado Módulo A

Módulo A

```
offset
                .data
                .globl X
   +0:  X:      .word -1
   +4:  Y:      .word -2

   +0:          .text
                ...
  +20:  f:      ...
                ...
  +36:          la $t0,Y
                ...
  +60:          jal g
                ...
```

8 bytes

80 bytes

Lista de Reubicación:
- posición +36 (text), tipo la, offset +4 (data)

Símbolos Globales
- posición +0 (data), label="X"

Referencias NO resueltas
- posición +60 (text), tipo jal, label="g"

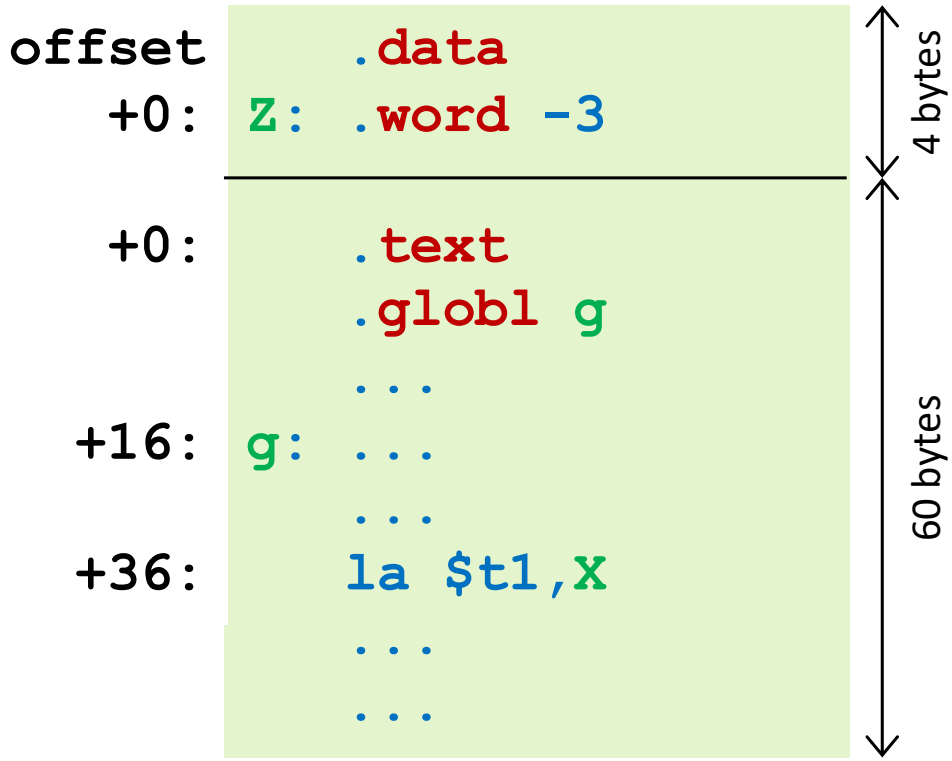
Ensamblado Módulo B

Lista de Reubicación:
- ...

Símbolos Globales
- posición +16 (text), label="g"

Referencias NO resueltas
- posición +36 (text), tipo la, label="X"

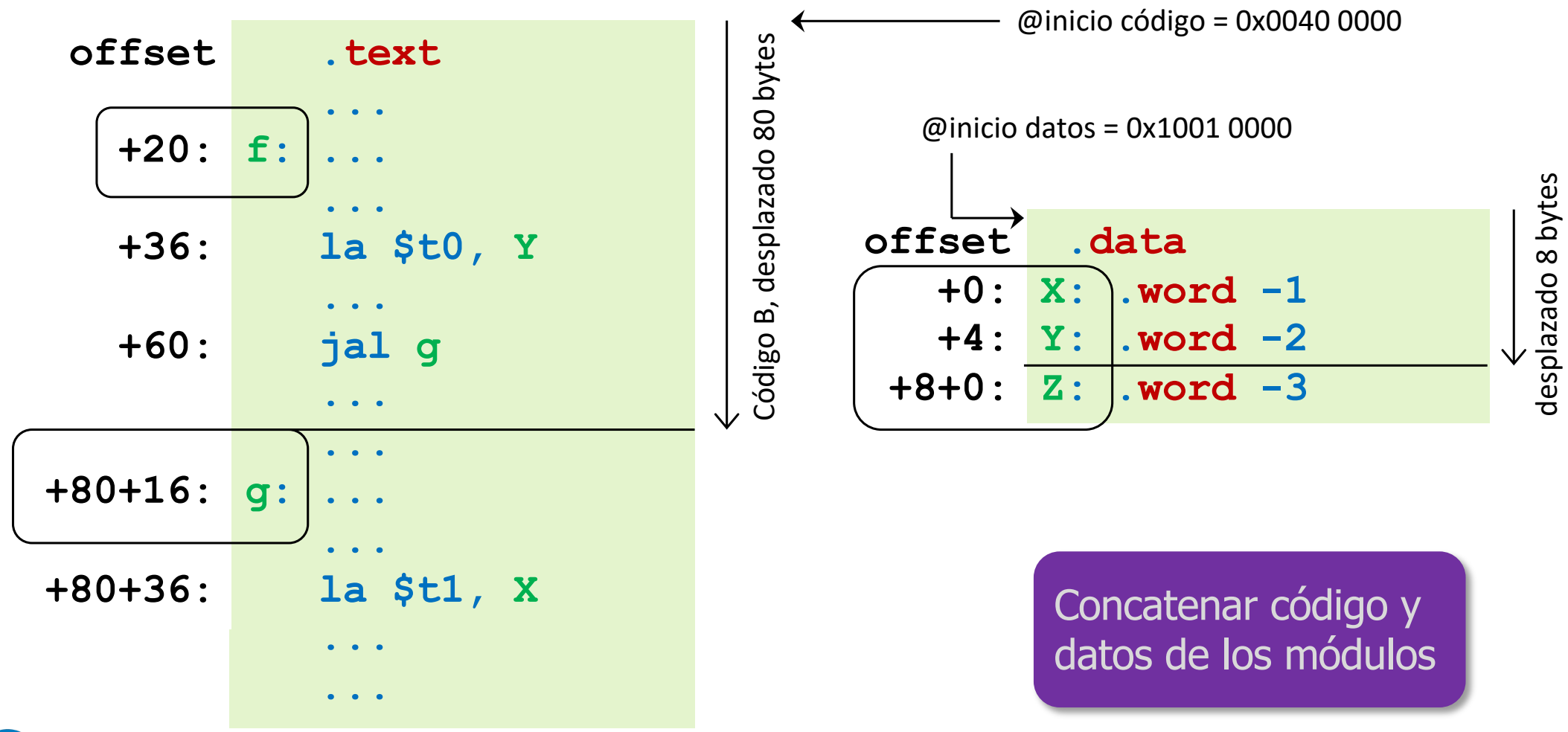
Módulo B



Enlazado (o Montaje o “Linkado”)

1. Buscar, en cascada, los ficheros necesarios ya sean del programa, o del sistema (**libraries**)
2. Concatenar el código y los datos de los módulos
   - Anotando el desplazamiento de cada sección
   - Asignando direcciones definitivas a las etiquetas en las tablas de símbolos

Enlazado. Ejemplo con 2 módulos



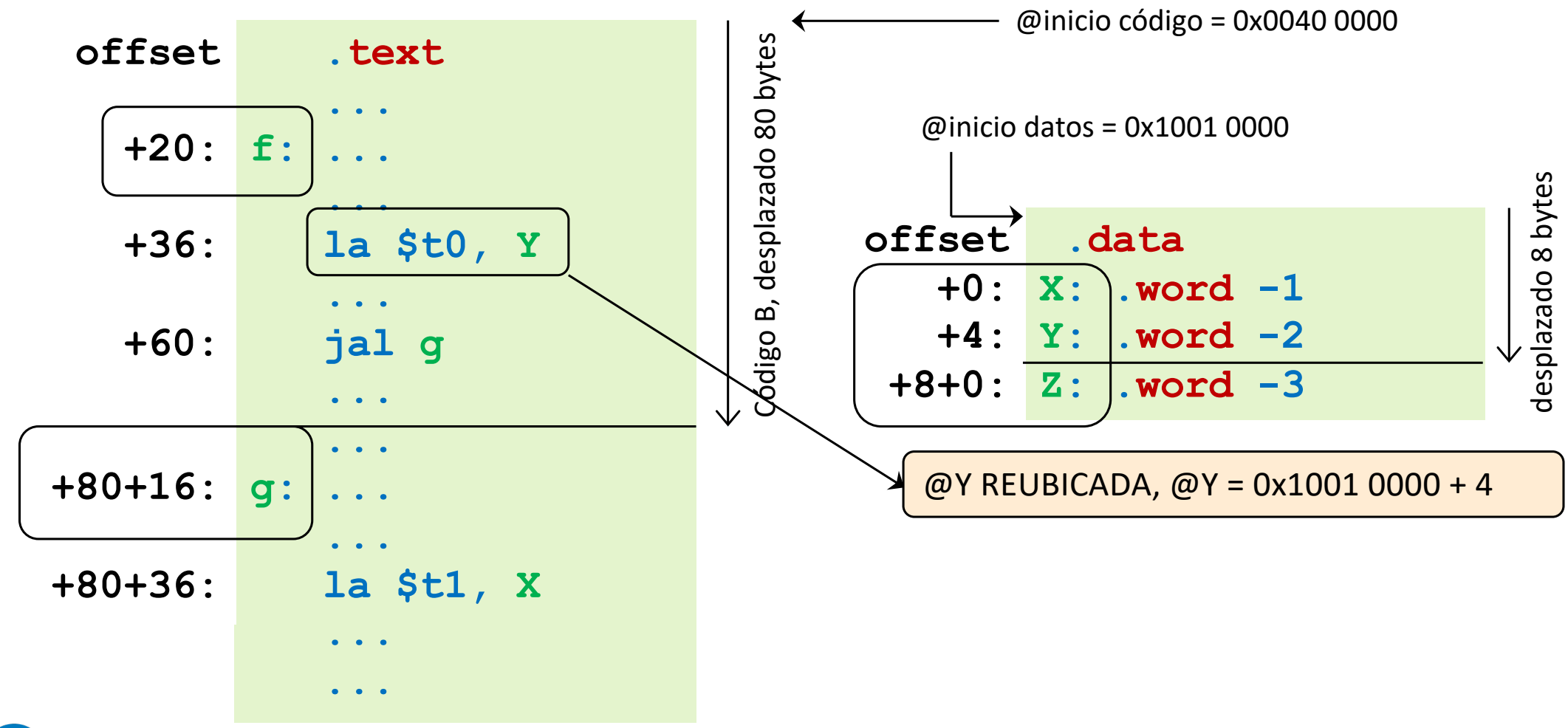
Concatenar código y datos de los módulos

Enlazado (o Montaje o “Linkado”)

1. **Buscar, en cascada, los ficheros necesarios ya sean del programa, o del sistema (libraries)**
2. **Concatenar el código y los datos de los módulos**
   - Anotando el desplazamiento de cada sección
   - Asignando direcciones definitivas a las etiquetas en las tablas de símbolos
3. **Reubicar** instrucciones con direcciones absolutas
   - Dirección definitiva = Dirección inicial (0x00400000 [código] o 0x10010000 [datos]) +
     + Desplazamiento de la Sección +
     + Offset provisional dentro de su sección

Lista de Reubicación:
- posición +36 (text), tipo la, offset +4 (data)

Enlazado. Ejemplo con 2 módulos



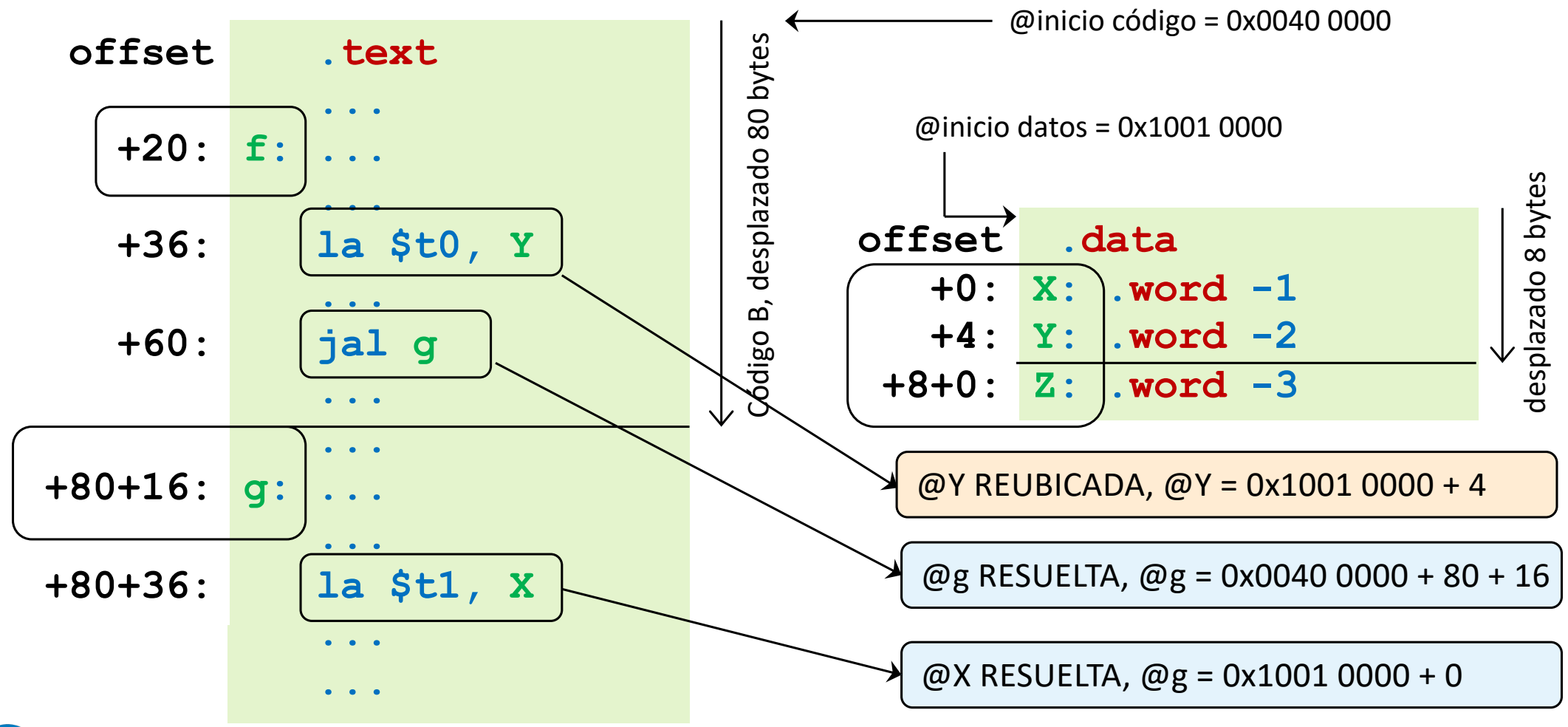
Enlazado (o Montaje o “Linkado”)

1. Buscar, en cascada, los ficheros necesarios ya sean del programa, o del sistema (**libraries**)
2. Concatenar el código y los datos de los módulos
   - Anotando el desplazamiento de cada sección
   - Asignando direcciones definitivas a las etiquetas en las tablas de símbolos
3. **Reubicar** instrucciones con direcciones absolutas
   - Dirección definitiva = Dirección inicial (0x0040 0000 o 0x1001 0000) +
     + Desplazamiento de la Sección +
     + Offset provisional dentro de su sección
4. Resolver **Referencias NO Resueltas** (cruzadas)
   - Consultando las direcciones definitivas en la **Tabla de Símbolos**

Módulo A
Referencias NO resueltas
- posición +60 (text), tipo jal, label=“g”

Módulo B
Referencias NO resueltas
- posición +36 (text), tipo la, label=“X”

Enlazado. Ejemplo con 2 módulos



Enlazado (o Montaje o “Linkado”)

1. Buscar, en cascada, los ficheros necesarios ya sean del programa, o del sistema (**libraries**)
2. Concatenar el código y los datos de los módulos
   - Anotando el desplazamiento de cada sección
   - Asignando direcciones definitivas a las etiquetas en las tablas de símbolos
3. **Reubicar** instrucciones con direcciones absolutas
   - Dirección definitiva = Dirección inicial (0x0040 0000 o 0x1001 0000) +
     + Desplazamiento de la Sección +
     + Offset provisional dentro de su sección
4. Resolver **Referencias NO Resueltas** (cruzadas)
   - Consultando las direcciones definitivas en la **Tabla de Símbolos**
5. Escribir el fichero ejecutable en disco
   - Estructura similar al fichero objeto

Carga en Memoria

- Comando de invocación del programa

```
> filtrador fileIN.bmp fileOUT.bmp
```

  - `filtrador`, es el fichero ejecutable.
  - `fileIN.bmp fileOUT.bmp`, son los dos parámetros de entrada al programa.
- ¿Qué le llega al ejecutable?, 2 variables:
  - `int argc`, en este caso valdría 3.

```
argc: .word 3
```

  - `char *argv[]`, es un vector de punteros a char, en este caso un vector con 3 posiciones y la siguiente información:

```
x1:   .asciiz "filtrador"
x2:   .asciiz "fileIN.bmp"
x3:   .asciiz "fileOUT.bmp"
argv: .word x1, x2, x3
```

```
void main(int argc, char *argv[] {
...
}
```
C

Carga en Memoria

- El **loader del SO** es el encargado de cargar el programa en Memoria
  1. Lee la cabecera del ejecutable para determinar el tamaño de las secciones.
  2. Reserva espacio en Memoria Principal
  3. Copia el código y los datos del fichero en Memoria (en el tema 7, veremos que no es necesario copiarlo todo).
  4. Copia en la pila los parámetros del **`main()`**
  5. Inicializa los registros, dejando el **`$sp`** apuntando a la cima de la pila.
  6. Salta a la rutina **`startup()`** (el montador la incluye en el ejecutable).
- ¿Qué hace **`startup()`**?
  1. Pasa los parámetros al **`main()`** (en los registros **`$a0`** y **`$a1`**) [argc y argv].
  2. Llama a la rutina **`main()`**
  3. Cuando **`main()`** retorna, **`startup()`** invoca la rutina **`exit()`** del SO para liberar los recursos asignados al programa.

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Estructura de Computadores
Tema 3: Traducción de Programas (cont)

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya