Estructura de Computadores

Suposem un programa que s'executa sobre un processador funcionant a una freqüència de 4 GHz, el qual dissipa una potència de 90W. La següent taula mostra, per a cada tipus d'instrucció, el nombre d'instruccions executades i el CPI, referents a l'execució d'aquest programa:

Tipus d'instrucció	Nombre d'instruccions	CPI
Memòria	$5 \cdot 10^9$	8
Aritmètiques	$25 \cdot 10^9$	2
Salts	$10 \cdot 10^9$	3

a) Calcula el CPI promig de tot el programa

CPI = [ ]

b) Calcula el temps d'execució del programa, en segons

t_{exe} = [ ] s

c) Calcula l'energia total consumida durant l'execució del programa, en Joules

E = [ ] J

d) Volem millorar el rendiment del processador optimitzant la gestió de les instruccions de memòria. Quin hauria de ser el nou CPI de les instruccions de memòria per a obtenir un guany de rendiment (speed-up) de 1.2x?

CPI memòria = [ ]

Suposem un programa que s'executa sobre un processador funcionant a una freqüència de 2GHz, el qual dissipa una potència de 40W. La següent taula mostra, per a cada tipus d'instrucció, el nombre d'instruccions executades i el CPI, referents a l'execució d'aquest programa:

Tipus d'instr.	Nombre. d'instr.	CPI
Memòria	$1{,}5 \times 10^9$	3
Salts	$1{,}0 \times 10^9$	4
Resta	$2{,}5 \times 10^9$	1

a) Calcula el CPI promig de tot el programa

CPI = ☐

b) Calcula el temps d'execució del programa, en segons

t_{exe} = ☐ s

c) Calcula el consum d'energia del programa, en Joules

E = ☐ J

d) Suposem una nova versió del programa en què reduïm el nombre de Salts a la meitat, però és a costa d'augmentar el seu CPI de 4 a 6. Quin guany de rendiment (speed-up) s'ha produït?

guany = ☐

Un processador P0 disposa de 3 tipus d'instruccions: A, B i C, amb CPIs que es detallen a la taula de sota. Hem creat un disseny millorat de l'anterior, el processador P1, amb idèntic joc d'instruccions, però amb diferents CPI, voltatge i freqüència de rellotge. La taula següent resumeix, per a P0 i P1, els CPI de cada tipus d'instrucció així com els seus voltatges d'alimentació i freqüències de rellotge.

	V (Volts)	f (Ghz)	CPI_A	CPI_B	CPI_C
P0	1	1,5	1	4	5
P1	1,5	2	1	5	5

a) (1,0 pts.) Sabent que P0 dissipa una potència dinàmica de 20W, calcula la potència dinàmica dissipada (en watts) per P1, suposant que no ha variat la capacitància equivalent del circuit (C) ni el factor d'activitat (α).

Potència = [W]

b) (0,70 pts.) Calcula el guany de rendiment (speedup) de P1 respecte de P0 que s'obté en executar un programa de test que conté $80 \cdot 10^9$ instruccions de tipus A, $20 \cdot 10^9$ de tipus B i $4 \cdot 10^9$ de tipus C.

Guany = [ ]

El processador d'un telèfon mòbil disposa de 4 tipus d'instruccions (A, B, C, D), amb diferents CPI. El processador dissipa una potència dinàmica de 16W i la potència estàtica es considera nul·la (0W). La seva bateria està plena i pot emmagatzemar 500J. Executem un programa de test que triga 30 segons. Analitzant el programa sabem que el nombre d'instruccions executades de cada tipus, així com el seu CPI són els que apareixen en la següent taula.

tipus d'instrucció	nombre d'instruccions	CPI
A	$6*10^9$	1
B	$2*10^9$	2
C	$4*10^9$	5
D	$3*10^9$	4

a) Quina és la freqüència de rellotge del processador?

freq. =

b) Durant quant de temps més podrem tornar a executar el programa de test amb l'energia que queda a la bateria?

temps =

Si augmentem la freqüència del processador a 2 GHz però les instruccions de tipus B tenen un CPI=6, suposant que no ha canviat cap més paràmetre arquitectònic o electrònic, calcula els següents paràmetres durant l'execució del programa de test :

c) El temps d'execució en segons.

temps =

d) El guany de rendiment.

guany =

e) La potència dissipada en Watts.

potència =

Considera la següent funció f, que rep com paràmetres en $a0 i $a1 dos vectors de 1000 enters, i que retorna el seu producte escalar.

```
f:      addiu   $t0, $zero, 0
        addiu   $v0, $zero, 0
do:
        lw      $t1, 0($a0)
        lw      $t2, 0($a1)
        mult    $t1, $t2
        mflo    $t3
        addu    $v0, $v0, $t3
        addiu   $a0, $a0, 4
        addiu   $a1, $a1, 4
        addiu   $t0, $t0, 1
        slti    $t4, $t0, 1000
        bne     $t4, $zero, do      # salta si $t0<1000

        jr      $ra
```

Suposem que s'executa en un processador amb un rellotge de 2GHz i amb els següents CPI segons el tipus d'instrucció:

TIPUS	multiplicació	salt (salta)	salt (no salta)	load/store	altres
CPI	9	5	1	5	1

Els càlculs que es demanen a continuació sobre l'execució de la funció `f` s'han d'expressar amb nombres decimals amb un màxim de 2 dígits fraccionaris de precisió (es poden obtenir fàcilment sense calculadora).

a) Calcula el temps d'execució de `f` en microsegons ($1\mu s = 10^{-6}$ s)

temps = _____ µs

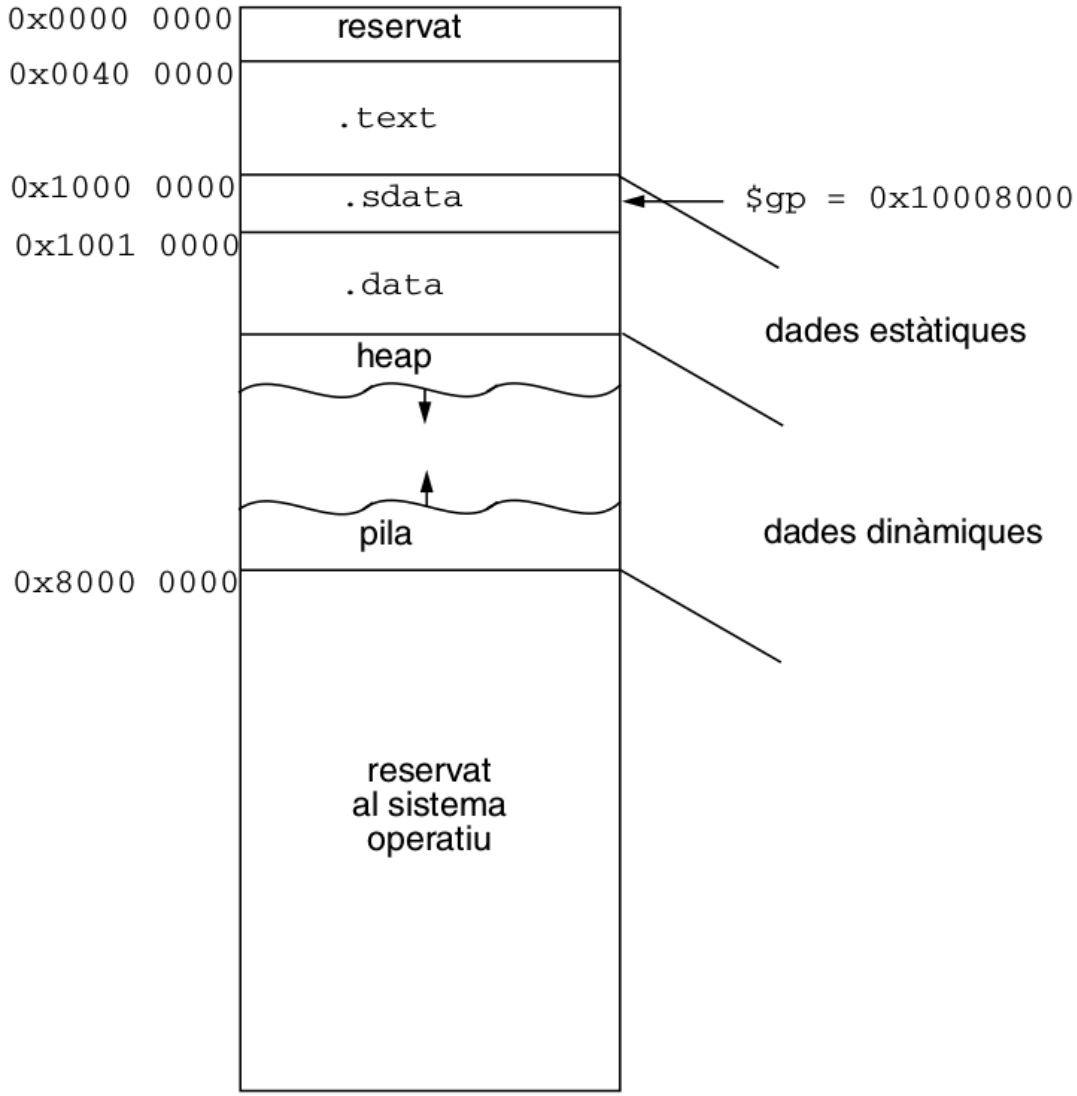
b) Calcula el CPI promig

CPI = _____

c) Calcula el guany de rendiment (speedup) que obtindríem si optimitzem el disseny de la CPU de manera que les multiplicacions tardin 4 cicles en lloc de 9.

Tema 3. Traducción de programas

Estructura de la memoria



Almacenamiento estático

- La variable se guarda en el mismo lugar durante toda la ejecución
- Variables globales en C
  - Variables definidas en la sección `.data` en MIPS
- La región de memoria tiene un tamaño fijo para cada programa

```
int a[2] = {8, 7};
short b = -7;

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

```
.data
a: .word 8, 7
b: .half -7

.text
main:
        ...
```

Almacenamiento estático

- El acceso a variables globales obliga a cargar la dirección en un registro

```
la $t0, etiqueta_variable_global
lw $t0, 0($t0)
```

Almacenamiento estático

- El acceso a variables globales obliga a cargar la dirección en un registro

```
la $t0, etiqueta_variable_global
lw $t0, 0($t0)
```

- Para optimizar el acceso, se puede utilizar el global pointer
  - Apunta a una dirección de memoria fija
  - Se almacena en el registro $gp ($28)
  - Acceso directo a variables cercanas al global pointer

```
lw $t0, offset_variable($gp)
```

Almacenamiento estático

- El acceso a variables globales obliga a cargar la dirección en un registro

```
la $t0, etiqueta_variable_global
lw $t0, 0($t0)
```

- Para optimizar el acceso, se puede utilizar el global pointer
  - Apunta a una dirección de memoria fija
  - Se almacena en el registro $gp ($28)
  - Acceso directo a variables cercanas al global pointer

```
lw $t0, offset_variable($gp)
```

- Sección `.sdata` (small data)
  - 2^{16} bytes accesibles desde el global pointer

Almacenamiento dinámico

- La variable se guarda en un área de memoria de manera temporal
- Pila
  - Variables locales
  - Crece dinámicamente hacia direcciones bajas
  - Se reserva memoria al inicio de la subrutina
  - Se libera memoria al final de la subrutina

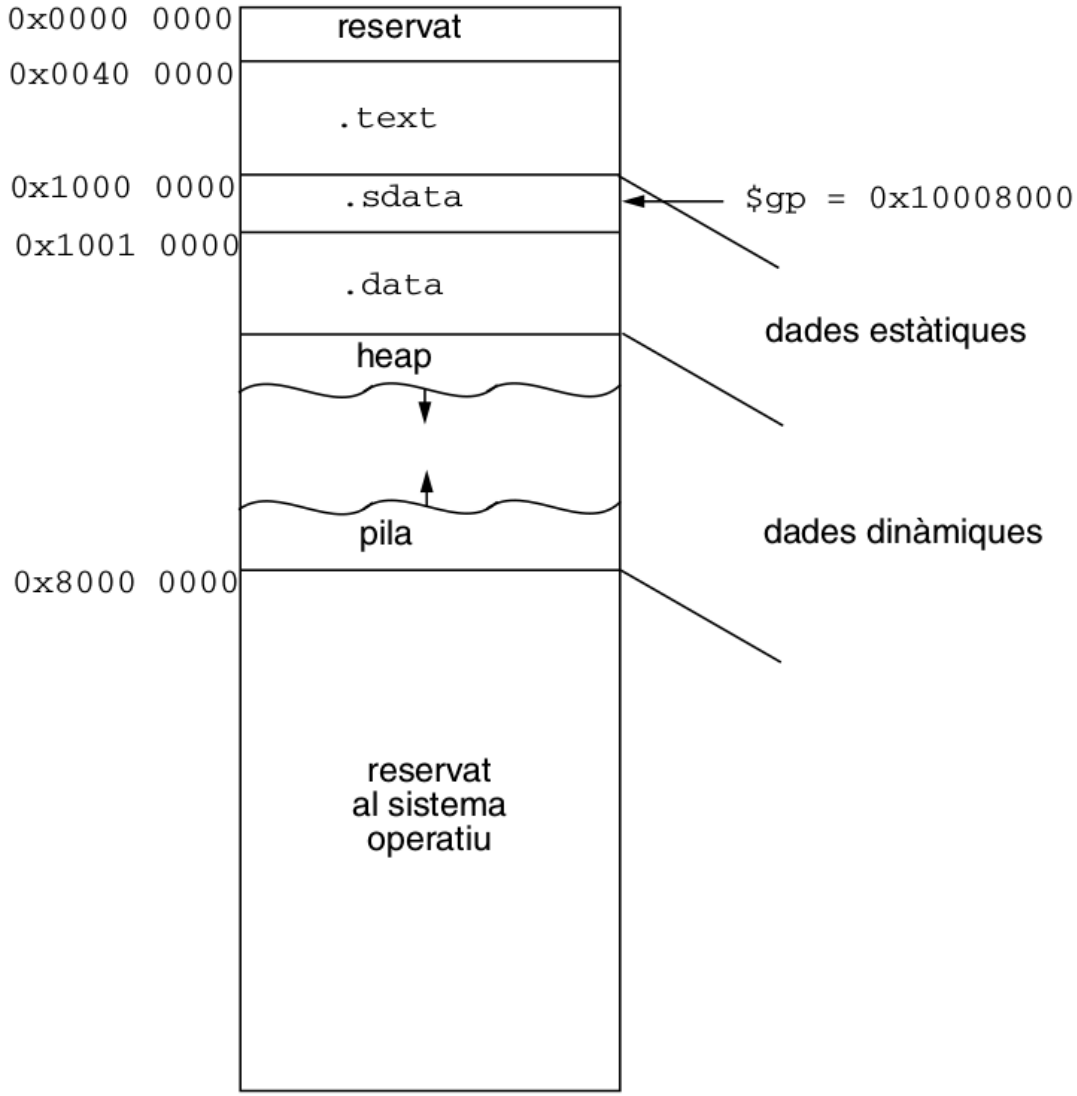
```
int f(int a, int b) {
    int v[100];
    ...
}
```

Almacenamiento dinámico

- La variable se guarda en un área de memoria de manera temporal
- Heap
  - Variables cuya memoria se reserva y libera de forma explícita
  - `malloc` / `free`
  - Gestionado por el sistema operativo
  - Crece hacia direcciones altas

```
int main(void) {
    int *ptr_v;
    ...
    ptr_v = malloc(400);
    ...
    free(ptr_v);
    ...
}
```

Estructura de la memoria



¿Dónde se almacenan las distintas variables?

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

int f() {
    int lvec[100];
    pvec = malloc(400);
    ...
    pvec[10] = gvec[10] + lvec[10];
}

int g() {
    ...
    free(pvec);
}
```

Compilación de programas

1. Compilación
   - El compilador traduce de C a MIPS

Compilación de programas

1. Compilación
   - El compilador traduce de C a MIPS
2. Ensamblado
   - El ensanblador traduce de MIPS a código máquina

Compilación de programas

1. Compilación
   - El compilador traduce de C a MIPS
2. Ensamblado
   - El ensanblador traduce de MIPS a código máquina
3. Enlazado
   - El enlazador (linker) combina múltiples ficheros con código máquina en un solo fichero ejecutable

Compilación de programas

1. Compilación
   - El compilador traduce de C a MIPS
2. Ensamblado
   - El ensanblador traduce de MIPS a código máquina
3. Enlazado
   - El enlazador (linker) combina múltiples ficheros con código máquina en un solo fichero ejecutable
4. Carga
   - El cargador (loader) lee el fichero ejecutable y lo carga en memoria para ser ejecutado

Compilación separada

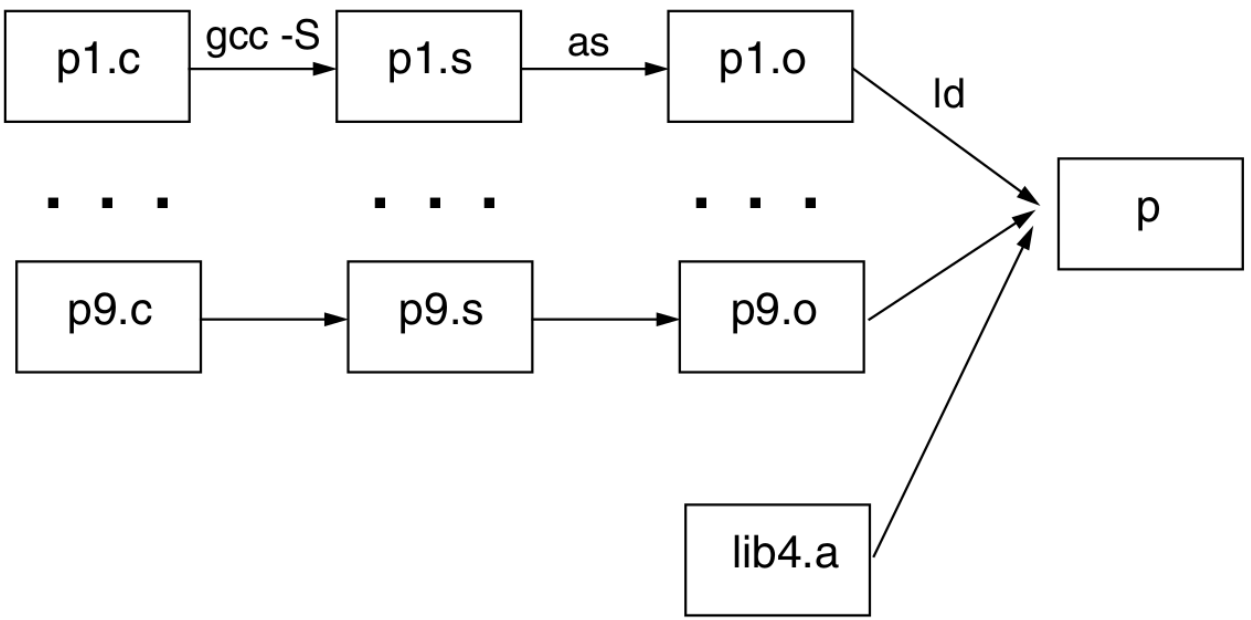
- Estructurar el código en varios módulos:
  - Facilita la gestión de proyectos complejos
  - Fomenta la reutilización de código
  - Recompilaciones parciales al realizar cambios

Compilación separada

- Estructurar el código en varios módulos:
  - Facilita la gestión de proyectos complejos
  - Fomenta la reutilización de código
  - Recompilaciones parciales al realizar cambios
- Los módulos (ficheros) se compilan y ensamblan por separado
- Los distintos módulos se enlazan para generar un ejecutable

Compilación separada

- Estructurar el código en varios módulos:
  - Facilita la gestión de proyectos complejos
  - Fomenta la reutilización de código
  - Recompilaciones parciales al realizar cambios
- Los módulos (ficheros) se compilan y ensamblan por separado
- Los distintos módulos se enlazan para generar un ejecutable



Ensamblado

- Expansión de macros (pseudoinstrucciones)

Ensamblado

- Expansión de macros (pseudoinstrucciones)
- Direcciones de las etiquetas
  - En MIPS, las etiquetas se pueden usar antes de ser definidas
  - El ensamblador realiza dos pasadas sobre el código MIPS
    - La primera pasada genera una tabla de símbolos con la dirección de cada etiqueta
    - La segunda pasada codifica las instrucciones

Ensamblado

- Expansión de macros (pseudoinstrucciones)
- Direcciones de las etiquetas
  - En MIPS, las etiquetas se pueden usar antes de ser definidas
  - El ensamblador realiza dos pasadas sobre el código MIPS
    - La primera pasada genera una tabla de símbolos con la dirección de cada etiqueta
    - La segunda pasada codifica las instrucciones
- Las direcciones de etiquetas que asigna el ensamblador NO son las direcciones de memoria definitivas
  - El ensamblador no tiene acceso al código y los datos definidos en otros módulos

Resolución de etiquetas

- Direcciones relativas al PC (`beq`/`bne`)
  - Las resuelve el **ensamblador**, no dependen del contenido de otros módulos

Resolución de etiquetas

- Direcciones relativas al PC (`beq`/`bne`)
  - Las resuelve el **ensamblador**, no dependen del contenido de otros módulos
- Direcciones absolutas (`j`/`jal`/`la`)
  - Las resuelve el **enlazador** (reubicación)
  - Referencias a variables definidas en otros módulos (`.globl`)
  - El ensamblador elabora una lista de instrucciones que requieren reubicación

Resultado del ensamblado

- Fichero con código máquina con los siguientes componentes:
  - Cabecera con información del resto de componentes del fichero

Resultado del ensamblado

- Fichero con código máquina con los siguientes componentes:
  - Cabecera con información del resto de componentes del fichero
  - Sección .text con el código máquina de las subrutinas del módulo

Resultado del ensamblado

- Fichero con código máquina con los siguientes componentes:
  - Cabecera con información del resto de componentes del fichero
  - Sección .text con el código máquina de las subrutinas del módulo
  - Sección de datos estáticos (variables globales)

Resultado del ensamblado

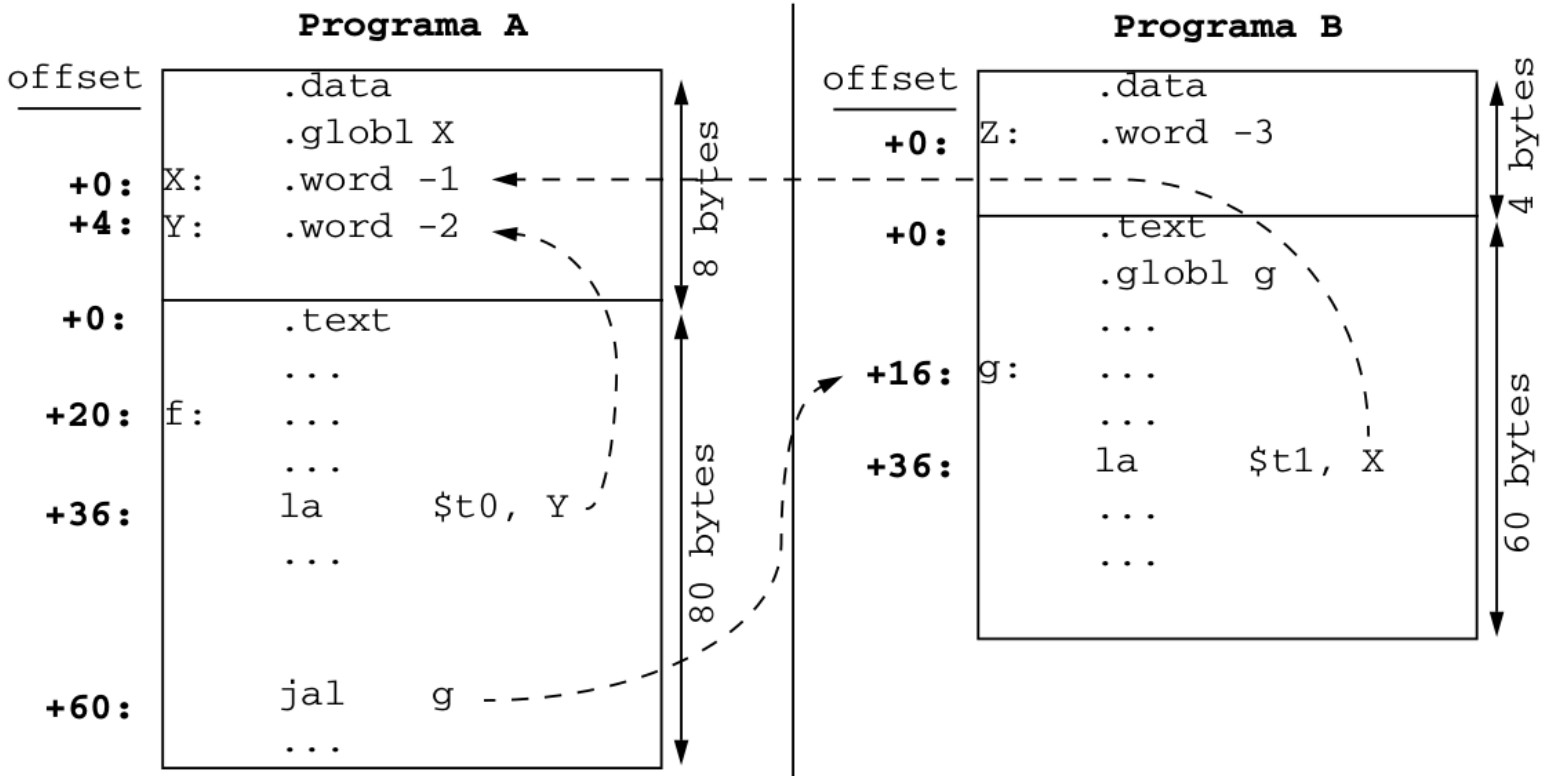
- Fichero con código máquina con los siguientes componentes:
  - Cabecera con información del resto de componentes del fichero
  - Sección .text con el código máquina de las subrutinas del módulo
  - Sección de datos estáticos (variables globales)
  - Lista de reubicación (posición de la instrucción, tipo y dirección provisional)

Resultado del ensamblado

- Fichero con código máquina con los siguientes componentes:
  - Cabecera con información del resto de componentes del fichero
  - Sección .text con el código máquina de las subrutinas del módulo
  - Sección de datos estáticos (variables globales)
  - Lista de reubicación (posición de la instrucción, tipo y dirección provisional)
  - Tabla de símbolos globales y listado de referencias no resueltas

Resultado del ensamblado

- Fichero con código máquina con los siguientes componentes:
  - Cabecera con información del resto de componentes del fichero
  - Sección .text con el código máquina de las subrutinas del módulo
  - Sección de datos estáticos (variables globales)
  - Lista de reubicación (posición de la instrucción, tipo y dirección provisional)
  - Tabla de símbolos globales y listado de referencias no resueltas
  - Información de depuración



Reubicar:
- posició +36 (text), tipus la, offset +4 (data)

Símbols globals:
- posició +0 (data), label="X"

Referències no-resoltes:
- posició +60 (text), tipus jal, label="g"

Reubicar:
-

Símbols globals:
- posició +16 (text), label="g"

Referències no-resoltes:
- posició +36 (text), tipus la, label="X"

Enlazado

- Se comprueba si hay etiquetas sin definir

Enlazado

- Se comprueba si hay etiquetas sin definir
- Se junta el código y los datos de los distintos módulos
  - Se asignan direcciones definitivas a los símbolos globales

Enlazado

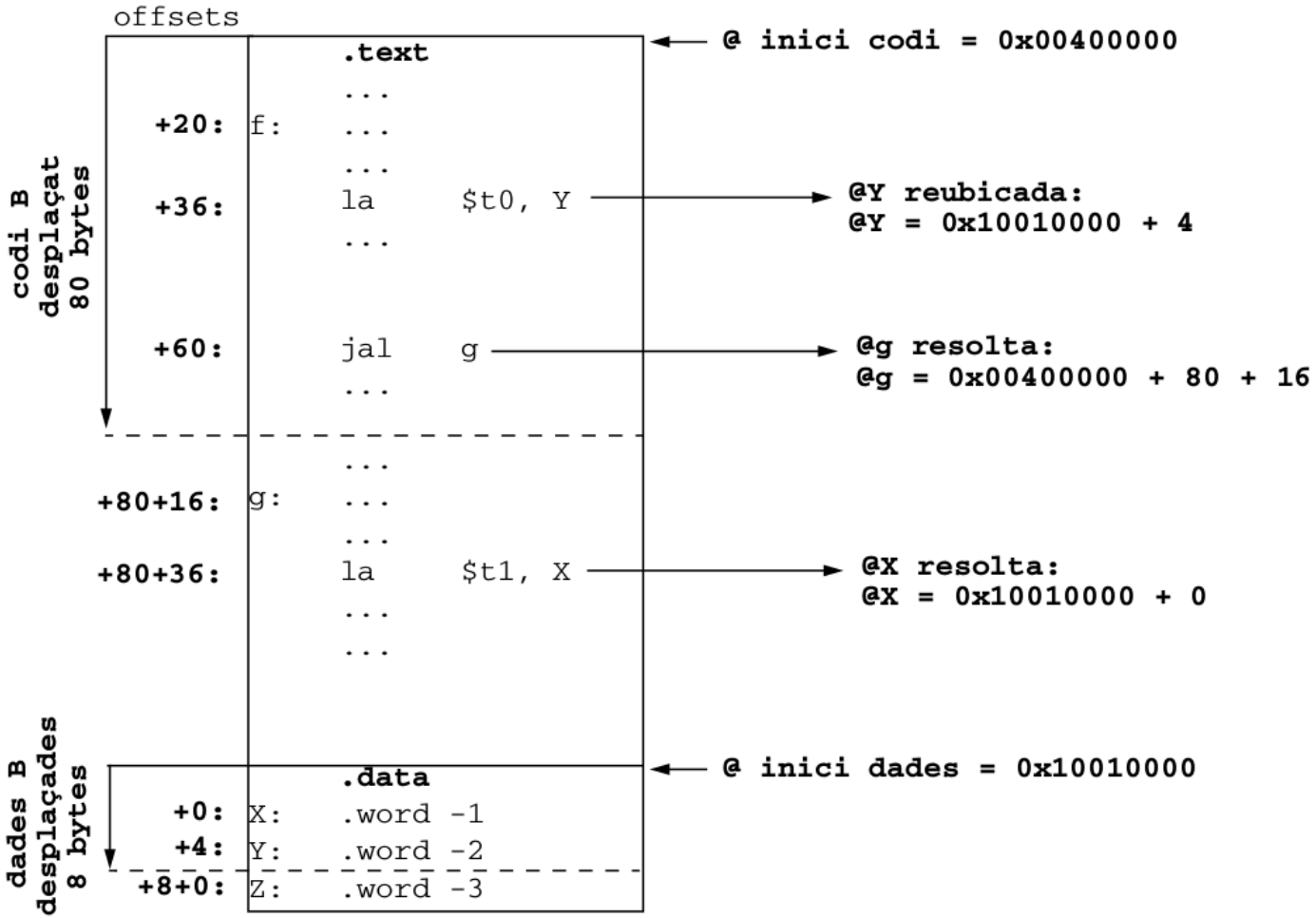
- Se comprueba si hay etiquetas sin definir
- Se junta el código y los datos de los distintos módulos
  - Se asignan direcciones definitivas a los símbolos globales
- Se realiza la reubicación de las instrucciones con direcciones absolutas
  - Se utilizan las listas de reubicación y los offsets generados por el ensamblador

Enlazado

- Se comprueba si hay etiquetas sin definir
- Se junta el código y los datos de los distintos módulos
  - Se asignan direcciones definitivas a los símbolos globales
- Se realiza la reubicación de las instrucciones con direcciones absolutas
  - Se utilizan las listas de reubicación y los offsets generados por el ensamblador
- Se resuelven las referencias cruzadas
  - Referencias a etiquetas externas definidas en otros módulos

Enlazado

- Se comprueba si hay etiquetas sin definir
- Se junta el código y los datos de los distintos módulos
  - Se asignan direcciones definitivas a los símbolos globales
- Se realiza la reubicación de las instrucciones con direcciones absolutas
  - Se utilizan las listas de reubicación y los offsets generados por el ensamblador
- Se resuelven las referencias cruzadas
  - Referencias a etiquetas externas definidas en otros módulos
- Se escribe el fichero ejecutable en disco


offsets
.text
...
+20: f: ...
...
+36: la $t0, Y
...
+60: jal g
...
+80+16: g: ...
...
+80+36: la $t1, X
...
...
.data
+0: X: .word -1
+4: Y: .word -2
+8+0: Z: .word -3
codi B desplaçat 80 bytes
dades B desplaçades 8 bytes
@ inici codi = 0x00400000
@Y reubicada:
@Y = 0x10010000 + 4
@g resolta:
@g = 0x00400000 + 80 + 16
@X resolta:
@X = 0x10010000 + 0
@ inici dades = 0x10010000

Carga en memoria

- El "cargador" del sistema operativo lee el fichero ejecutable de disco y lo almacena en la memoria del computador:
  1. Leer la cabecera del fichero ejecutable (tamaño del código y los datos)
  2. Reservar memoria prinicipal para el código y los datos
  3. Copiar las instrucciones y los datos del fichero a memoria
  4. Copiar en la pila los parámetros del programa principal (línea de comandos)
  5. Inicializar los registros del procesador (`$sp`)
  6. Llamar a la función main

¿Verdadero o falso?

- En un programa escrit en assemblador MIPS, que consta de 2 mòduls A i B que es compilen per separat, i on el mòdul A invoca una funció `func` que està declarada al mòdul B, l'assemblatge del mòdul A fallarà sense generar cap fitxer objecte.

MÒDUL 1

```
1         .data
2         .globl V
3   V:    .word 1, 2, 3, 4, 5
4   Y:    .word 0

5         .text
6         .globl main
7   main:addiu  $sp, $sp, -4
8         sw    $ra, 0($sp)

9         la    $t0, X
10        lw    $t1, 0($t0) # $t1 <- X
11        la    $t2, V
12        sll   $t3, $t1, 2
13        addu  $t4, $t2, $t3
14        lw    $a0, 0($t4) # $a0 <- V[X]
15        jal   f

16        lw    $ra, 0($sp)
17        addiu $sp, $sp, 4
18        jr    $ra
```

MÒDUL 2

```
1         .data
2         .globl  X
3   X:    .word   3

4         .text
5         .globl  f
6   f:    la      $t0, Y
7         sw      $a0, 0($t0) # Y <- $a0
8         jr      $ra
```

a) Quan hem intentat enllaçar els dos fitxers objecte generats per l'assemblador, l'enllaçador ha detectat un error. Com caldrà corregir el codi MIPS perquè no torni a fallar?

mòdul:

codi corregit:

MÒDUL 1

```
1        .data
2        .globl V
3   V:   .word 1, 2, 3, 4, 5
4   Y:   .word 0

5        .text
6        .globl main
7   main:addiu  $sp, $sp, -4
8        sw     $ra, 0($sp)

9        la     $t0, X
10       lw     $t1, 0($t0) # $t1 <- X
11       la     $t2, V
12       sll    $t3, $t1, 2
13       addu   $t4, $t2, $t3
14       lw     $a0, 0($t4) # $a0 <- V[X]
15       jal    f

16       lw     $ra, 0($sp)
17       addiu  $sp, $sp, 4
18       jr     $ra
```

MÒDUL 2

```
1        .data
2        .globl  X
3   X:   .word   3

4        .text
5        .globl  f
6   f:   la      $t0, Y
7        sw      $a0, 0($t0) # Y <- $a0
8        jr      $ra
```

b) Contesta les següents 3 preguntes suposant que s'ha corregit l'error anterior i que conservem la numeració de línies original:

Pregunta	MÒDUL 1	MÒDUL 2
Quines etiquetes conté la Taula de Símbols Globals de cada fitxer objecte?		
Quines línies de codi en cada fitxer objecte (sols el número) contenen referències **no-resoltes** (referències creuades)?		
Quines línies de codi en cada fitxer objecte (sols el número) contenen **adreces absolutes** (i necessiten ser reubicades)?		