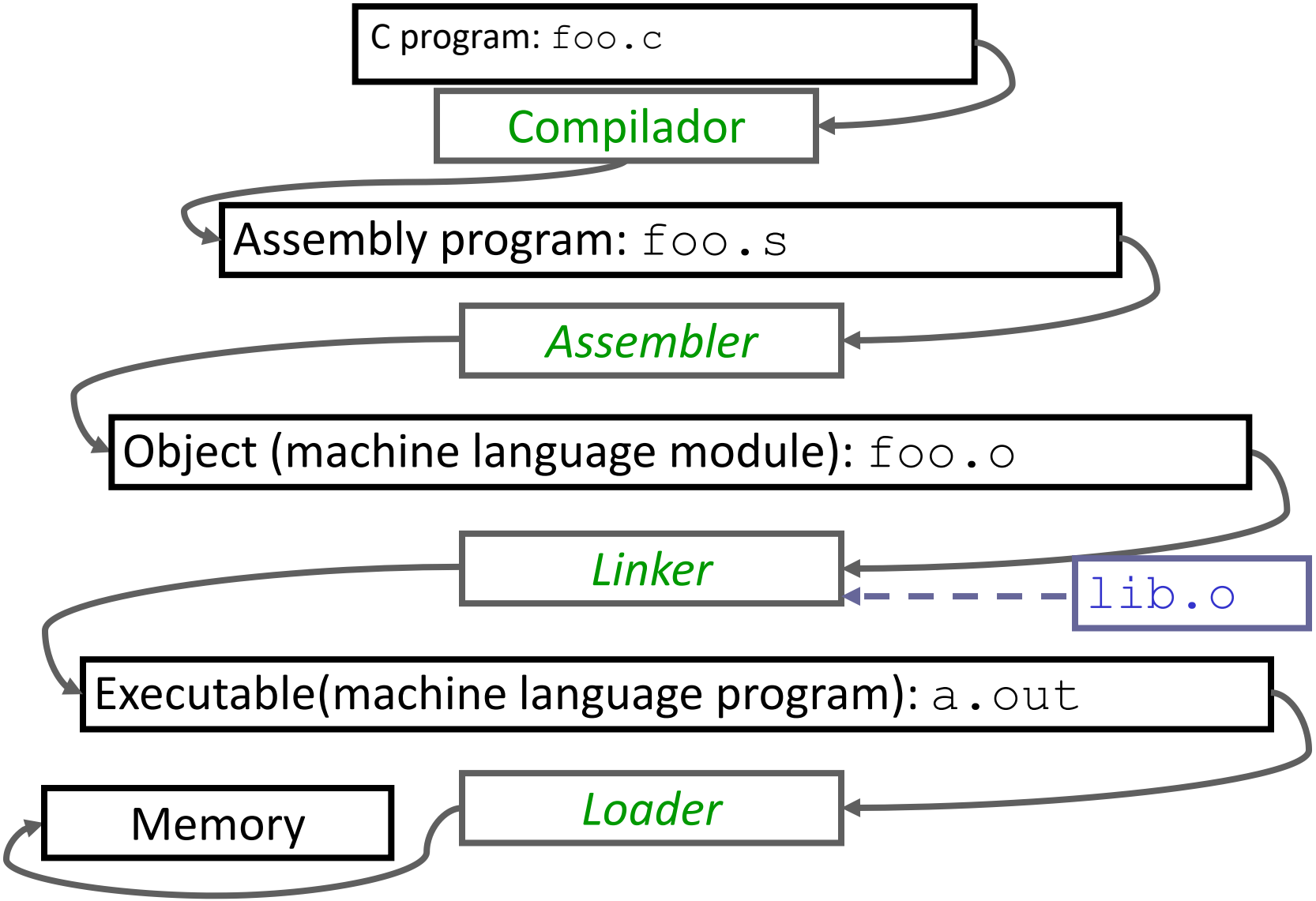
Introdução ao MIPS
- Correr um Programa -

Arquitetura de Computadores 2024/2025

Do código fonte ao executável



Compilação

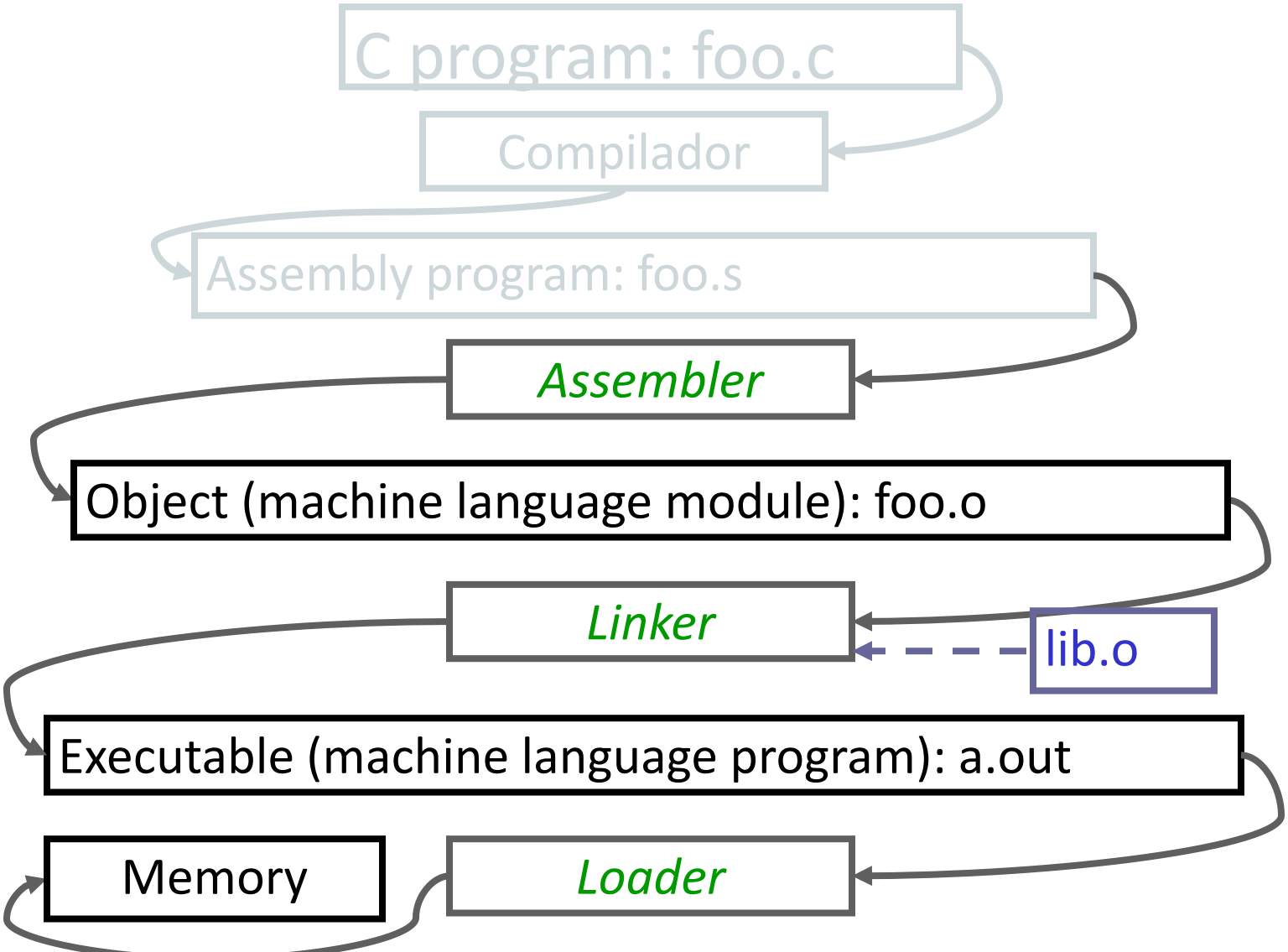
- Input: Código fonte escrito numa linguagem de alto nível
(e.g., C, Java como `foo.c`)

- Output: Código em linguagem *assembly*
(e.g., `foo.s` para o MIPS)

- Nota: O *output* pode conter pseudo-instruções

- Pseudo-instruções: instruções que o *assembler* compreende mas que não fazem parte do "*instruction set*" do processador. Por exemplo
  - `move $s1,$s2` => `add $s1,$s2,$zero`

Em que etapa estamos?



Assemblagem

- Input: Código em linguagem *assembly* (e.g., `foo.s` para o MIPS)
- Output: Código objecto, tabelas (e.g., `foo.o` para o MIPS)
- Lê e utiliza Directivas
- Substitui pseudo-instruções (MAL para TAL)
- Produz código máquina
- Cria Ficheiro de Código Objecto

Directivas do Assembler

- Dá indicações ao assembler, mas não é traduzido em instruções máquina
  - `.text`: Colocar o que vem a seguir no segmento de texto do utilizador (a ser traduzido em código máquina)
  - `.data`: Colocar o que vem a seguir no segmento de dados do utlizador
  - `.globl sym`: declarar `sym` como "label" global que pode ser referenciado a partir de outros ficheiros
  - `.asciiz str`: Armazenar a string `str` em memória terminada por null
  - `.word w1,…,wn`: Armazenar os *n* elementos de 32-bit em words sucessivas de memória

Substituição de Pseudo-Instruções

- O assembler não só considera como pseudo-instruções instruções que manifestamente não fazem parte do ISA, como rectifica variações cujo sentido é claro.

Pseudo:	Real:
`subu $sp,$sp,32`	`addiu $sp,$sp,-32`
`sd $a0, 32($sp)`	`sw $a0, 32($sp)` `sw $a1, 36($sp)`
`mul $t7,$t6,$t5`	`mult $t6,$t5` `mflo $t7`
`addu $t0,$t6,1`	`addiu $t0,$t6,1`
`ble $t0,100,loop`	`slti $at,$t0,101` `bne $at,$0,loop`
`la $a0, str`	`lui $at,left(str)` `ori $a0,$at,right(str)`

Geração de Código Máquina (1/3)

- Casos Simples
  - Instruções aritméticas e lógicas (add, sub, sll, or, etc.)
  - Toda a informação necessária está codificada na própria instrução

- E quanto aos "branches" condicionais?
  - Salto relativo ao valor do PC
  - Só podemos saber o tamanho real do salto relativo, depois de as pseudo-instruções terem sido substituídas

- No caso dos "branches" a assemblagem requer duas passagens

Geração de Código Máquina (2/3)

"*Forward Reference*" problem

- As instruções de "*branch*" podem fazer referência a "*labels*" que estão à frente no código

```
    or    $v0,$0,$0
L1: slt   $t0,$0,$a1
    beq   $t0,$0,L2
    addi  $a1,$a1,-1
    j     L1
L2: add    $t1,$a0,$a1
```

- A tradução para código máquina da instrução "beq" é feita em 2 passagens
  - A primeira passagem determina a posição do *label*
  - A segunda passagem usa a posição do *label* para fazer a tradução

Geração de Linguagem Máquina (3/3)

- E quanto aos *jumps* (`j` e `jal`)?
  - Os *jumps* funcionam em termos de endereços absolutos.
  - Só é possível gerar a instrução máquina depois de se saber a posição do *label* em memória (o salto não é relativo)
  - Isto só pode ser resolvido depois da linkagem

- E quanto às referências a dados?
  - `la` é desdobrado num `lui` e `ori`
  - Estes precisam de saber o endereço de 32 bits dos dados ... (mesmo problema que os *jumps*)

- Como isto só se sabe depois da assemblagem, precisamos de criar duas tabelas …

Tabelas

- Tabela de Símbolos
  - Lista os "itens" do "ficheiro .o" que podem ser referenciados deste ou de outros "ficheiros .o".
  - Que itens são estes?
    - *Labels*: e.g. chamada de funções
    - Dados: qualquer coisa da secção `.data`; variáveis que podem ser acedidas a partir de outros ficheiros

- Tabela de Realocação
  - Lista os"itens" que o "ficheiro .o" referencia e do qual não tem o endereço porque são externos (estão noutro ficheiro) ou serão resolvidos em "*runtime*".
    - Os "labels" usados nos `j` ou `jal`
      - internos
      - externos (includindo ficheiros .lib)
    - Dados
      - Por exemplo, a instrução `la`

Formato dos ficheiros .o (código objecto)

- **Cabeçalho:** posição e tamanho dos diferentes componentes do ficheiro objecto.

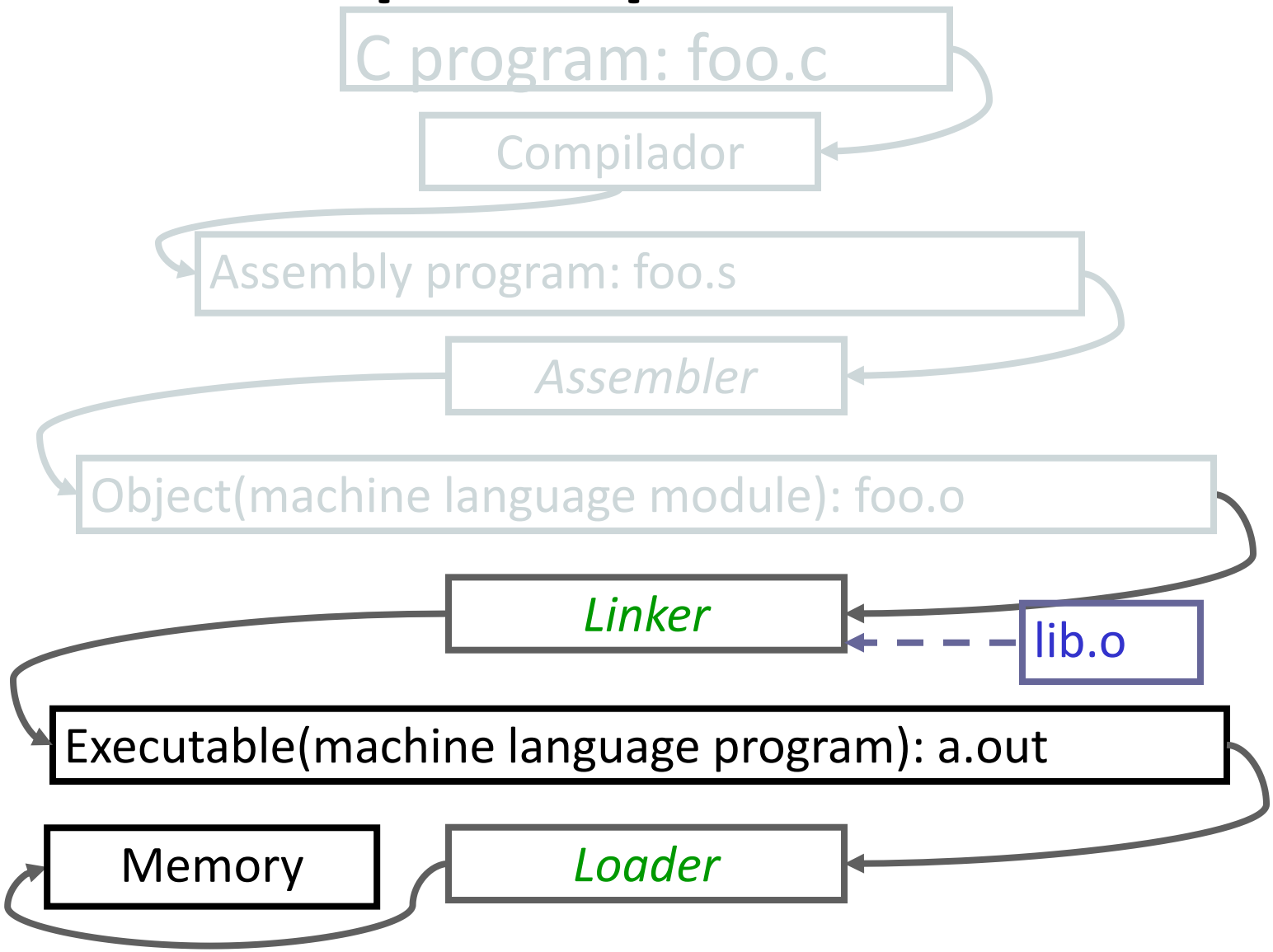
- **Segmento de texto**: código máquina
- **Segmento de dados**: representação binária dos dados e estruturas declarados no código fonte (normalmente declarações globais)

- **Tabela de realocação**: identifica as linhas de código onde há endereços a ser resolvidos
- **Tabela de símbolos**: lista de "*labels*" internos que podem ser referenciados, quer a partir do próprio ficheiro, quer a partir de ficheiros externos.

- **Informação de *debug*:** (lembre-se da *flag* –g do *gcc*)
- Um formato standard é o ELF (Executable and Linkable Format), excepto nas ferramentas Microsoft.

  http://www.skyfree.org/linux/references/ELF_Format.pdf

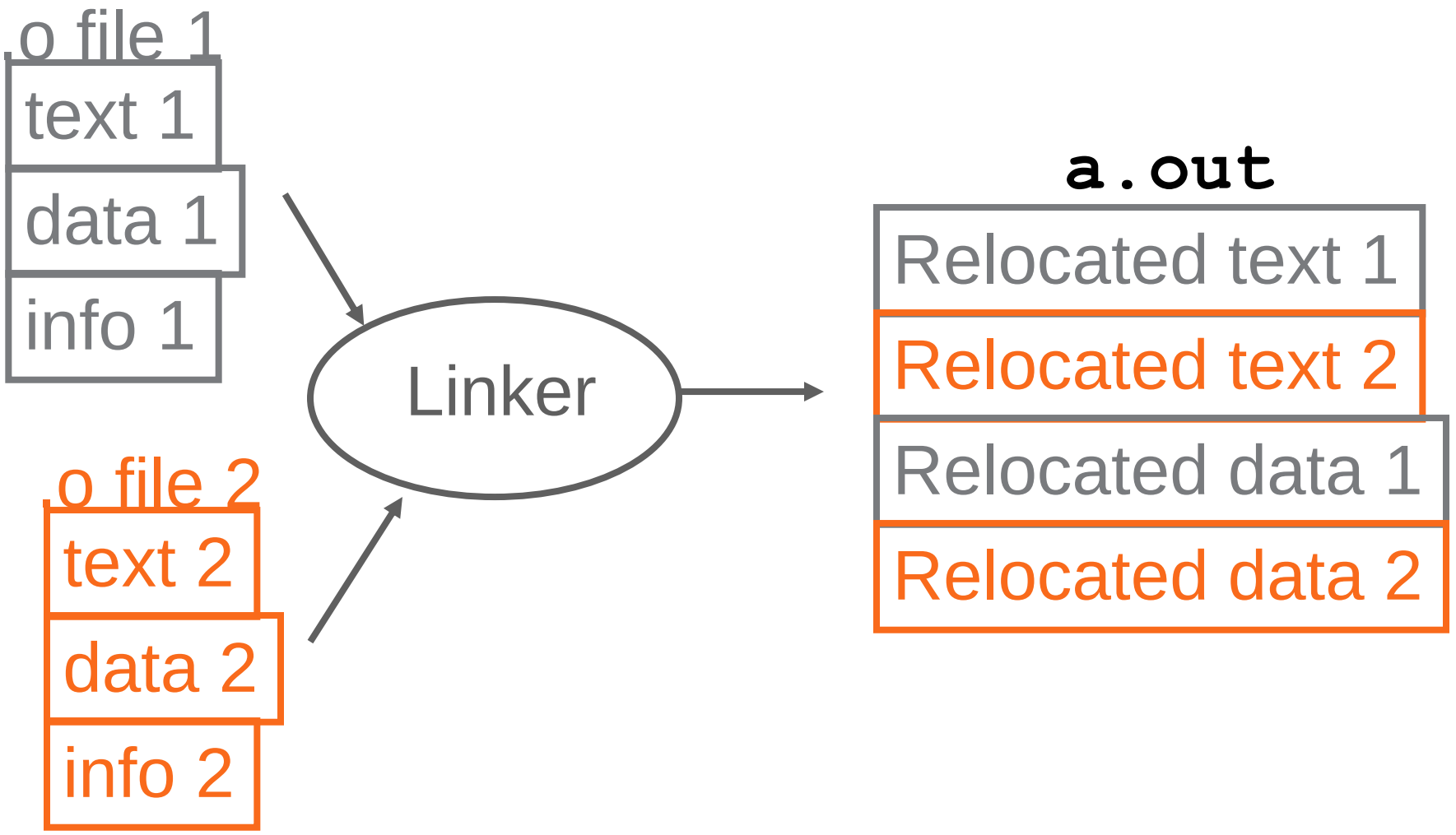
Em que etapa estamos?



Linker (1/3)

- Input: Ficheiros código objecto, tabelas (e.g., `foo.o`, `libc.o` para o MIPS)
- Output: Código executável (e.g., `a.out` para MIPS)
- Combina vários ficheiros (.o) num único executável ("linking")
- A técnica permite a compilação separada de diferentes ficheiros
  - Alterações num ficheiro fonte não requerem a recompilação de todo o programa (lembra-se do makefile?)
    - O código fonte do SO Windows mais recente tem > 50 M linhas de código!

Linker (2/3)


o file 1
text 1
data 1
info 1
o file 2
text 2
data 2
info 2
Linker
a.out
Relocated text 1
Relocated text 2
Relocated data 1
Relocated data 2

Linker (3/3)

- Passo 1: Concatenação dos segmentos de texto de cada ficheiro .o
- Passo 2: Juntar os segmentos de dados de cada ficheiro .o e concatená-los com o segmento de texto
- Passo 3: Resolver as referências
  - Ver as tabelas de realocação e resolver cada entrada
  - Definir os endereços absolutos em relação ao início do programa

Tipos de Endereçamento

- Endereçamento em relação ao PC (`beq`, `bne`): não é usada realocação
- Endereçamento absoluto (`j`, `jal`): realocação sempre
- Referências externas (normalmente `jal`): realocação sempre
- Referência a dados (normalmente `lui` e `ori`): realocação sempre

Endereçamento Absoluto no MIPS

- Quais as instruções que precisam de realocação de endereços?
  - J-format: jump, jump and link

j/jal	xxxxx

  - Loads e stores de variáveis na zona estática, referenciadas em relação ao global pointer

lw/sw	$gp	$x	address

- E quanto aos branches condicionais?

beq/bne	$rs	$rt	address

  - Como o endereçamento é feito em relação ao PC, as referências relativas mantêm-se mesmo que o código mude de sítio

Resolver Referências (1/2)

- O Linker *assume* que a primeira palavra do primeiro segmento de texto está no endereço 0x00000000.

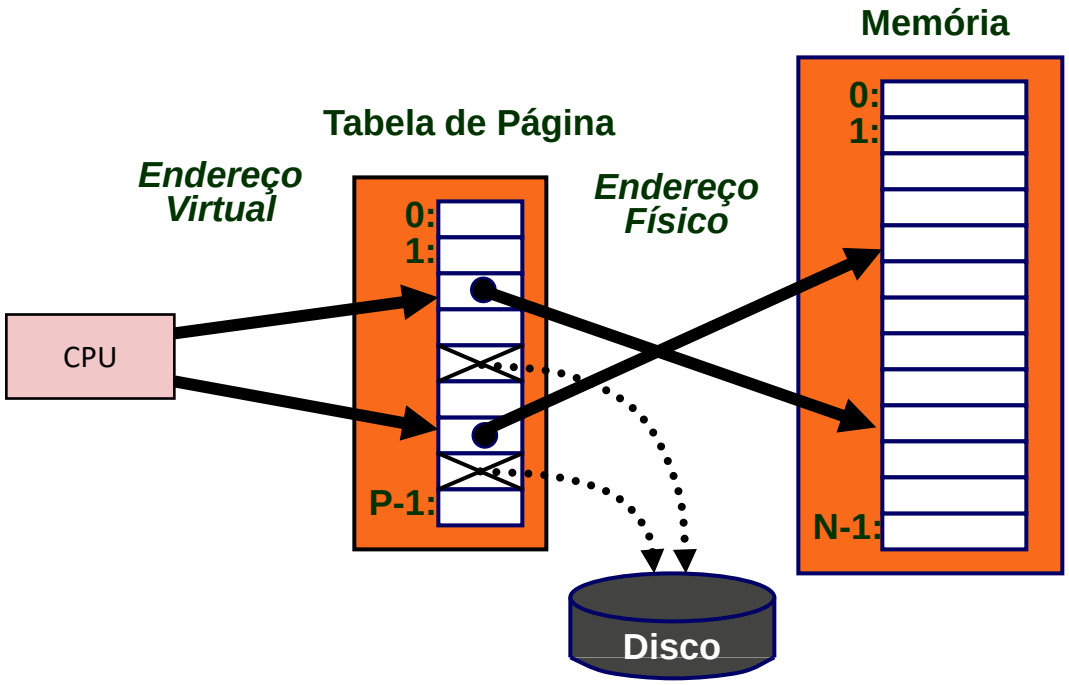
  (Quando estudarem o mecanismo de memória virtual voltarão a falar sobre isto)

- O Linker sabe:
  - O tamanho do segmento de texto e dados
  - A ordem e posição dos segmentos de texto e dados

- O Linker calcula com base nisto:
  - O endereço absoluto de cada label associado aos jumps (internos e externos) bem como cada bloco de dados que é referenciado

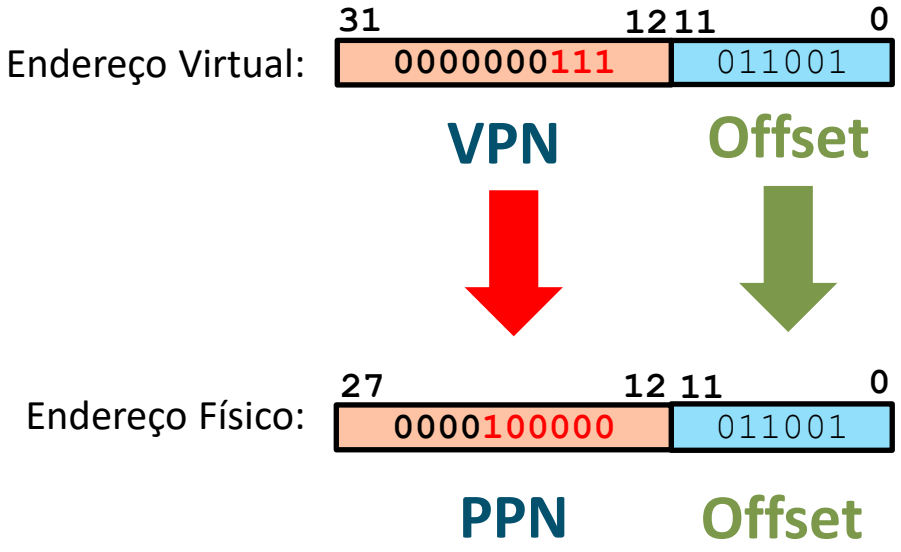
Memória Virtual (*preview* 1/2)

Tradução de Endereços: O hardware converte endereços virtuais em endereços físicos através de uma tabela de página gerida pelo Sistema Operativo



Memória Virtual (*preview* 2/2)

Tradução de Endereços: O hardware converte endereços virtuais em endereços físicos através de uma tabela de página gerida pelo Sistema Operativo



Resolver Referências (2/2)

- Para resolver as referências:
  - Procurar a referência (dados ou label) na tabela de símbolos
  - Se a referência não for encontrada, procurar nos ficheiros das bibliotecas (e.g. `printf`)
  - Assim que o endereço absoluto for encontrado, preencher o código máquina de forma apropriada
- Output do linker: ficheiro executável contendo o segmento de texto, o segmento de dados, e o cabeçalho a ser lido pelo "loader" (ver a seguir)

Bibliotecas Estáticas e Dinâmicas

- Aquilo que descrevemos é a forma tradicional de fazer "*linkagem*", normalmente conhecida por "*linkagem* estática"
  - No final a biblioteca é parte do executável. Assim, se posteriormente houver actualizações da biblioteca, o código criado não irá beneficiar das melhorias (teria que ser re-compilado a partir das fontes)
  - O executável inclui todas as bibliotecas, mesmo que só uma pequena parte tenha sido utilizada (e.g. só a função printf)
  - O executável é auto-contido.

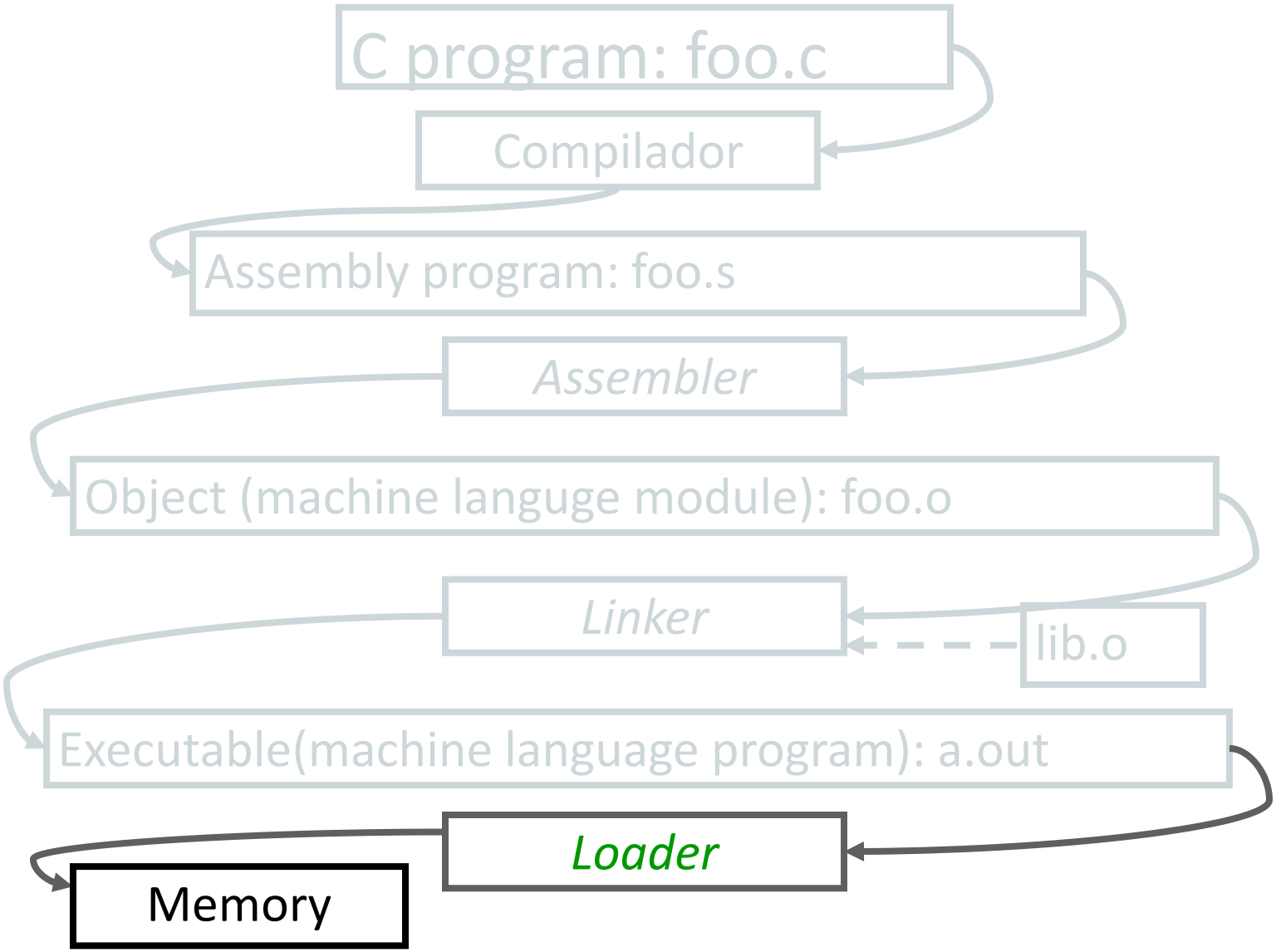
- Uma alternativa é usar "bibliotecas dinâmicas" (DLL-dynamically linked libraries), que são muito comuns em Windows & UNIX

Dynamically linked libraries

en.wikipedia.org/wiki/Dynamic_linking

- Espaço em Disco / Tempo de Execução
  - + O executável requer menos espaço em disco
  - + Como o executável é mais pequeno, o seu envio/partilha é feito de forma mais rápida
  - + A execução de dois programas que partilhem a mesma biblioteca é mais rápida (ver o que é código re-entrante)
  - – Existe um "overhead" em runtime para ser feita a linkagem
- Upgrades
  - + Substituindo um ficheiro (libXYZ.so) faz o upgrade de todos os programas que usem XYZ.
  - – O executável não é auto-contido

Em que etapa estamos?



Loader (1/2)

- Input: Código Executável (e.g., `a.out` para MIPS)
- Output: (programa a correr)
- Os ficheiros executáveis estão armazenados em disco.
- Quando o executável é chamado, o "loader" tem a tarefa de o carregar em memória e iniciar a execução.
- Normalmente o "loader" é o próprio OS
  - O carregamento de programas é uma das tarefas do OS

Loader (2/2)

- O que é que o "*loader*" faz?
  - Lê o cabeçalho dos executáveis para determinar o tamanho e posição dos segmentos de texto e dados
  - Cria um espaço de endereçamento para o programa capaz de receber o texto, dados e pilha (e eventualmente "*heap*")
  - Copia os dados e instruções do executável para o espaço de endereçamento criado
  - Copia os argumentos de chamada para a pilha (lembre-se do *argc* e *argv* no C)
  - Inicializa os registos do processador
    - A maioria dos registos são colocados a 0, mas o "*stack pointer*" fica a apontar para a 1ª *frame* livre
  - Salta para a rotina de "*start-up*" (ainda OS) que copia os argumentos do programa e faz o set do PC
  - Se a rotina principal (*main*) regressar, a rotina de "*startup*" termina o programa com uma chamada a exit.

Exemplo: C => Asm => Obj => Exe => Run

Código fonte do programa em C : prog.c

```
#include <stdio.h>
int main (int argc, char *argv[]) {
    int i, sum = 0;
    for (i = 0; i <= 100; i++)
       sum = sum + i * i;
    printf ("The sum of sq from 0 .. 100 is %d\n",
      sum);
}
```

"printf" está em "libc"

Compilação: MAL

```
__.text
   .align 2
   .globl main
main:
   subu $sp,$sp,32
   sw  $ra, 20($sp)
   sd  $a0, 32($sp)
   sw  $0, 24($sp)
   sw  $0, 28($sp)
loop:
   lw  $t6, 28($sp)
   mul $t7, $t6,$t6
   lw  $t8, 24($sp)
   addu $t9,$t8,$t7
   sw  $t9, 24($sp)
   addu $t0, $t6, 1
   sw  $t0, 28($sp)
   ble $t0,100, loop
   la  $a0, str
   lw  $a1, 24($sp)
   jal printf
   move $v0, $0
   lw  $ra, 20($sp)
   addiu $sp,$sp,32
   jr $ra
   .data
   .align 0
str:
   .asciiz   "The sum of
   sq from 0 .. 100 is
   %d\n"
```

Onde estão as 7 pseudo-instrucões?

Compilação: MAL

```
___.text
   .align 2
   .globl main
main:
   subu $sp,$sp,32
   sw  $ra, 20($sp)
   sd  $a0, 32($sp)
   sw  $0, 24($sp)
   sw  $0, 28($sp)
loop:
   lw  $t6, 28($sp)
   mul $t7, $t6,$t6
   lw  $t8, 24($sp)
   addu $t9,$t8,$t7
   sw  $t9, 24($sp)
```

```
   addu $t0, $t6, 1
   sw  $t0, 28($sp)
   ble $t0,100, loop
   la  $a0, str
   lw  $a1, 24($sp)
   jal printf
   move $v0, $0
   lw  $ra, 20($sp)
   addiu $sp,$sp,32
   jr $ra
   .data
   .align 0
str:
___.asciiz   "The sum of
   sq from 0 .. 100 is
   %d\n"
```

Assemblagem: Passo 1

- Substituir Pseudo-instruções, atribuir endereços

```
00 addiu $29,$29,-32
04 sw $31,20($29)
08 sw $4, 32($29)
0c sw $5, 36($29)
10 sw    $0, 24($29)
14 sw    $0, 28($29)
18 lw   $14, 28($29)
1c multu $14, $14
20 mflo   $15
24 lw   $24, 24($29)
28 addu $25,$24,$15
2c sw   $25, 24($29)
```

```
30 addiu $8,$14, 1
34 sw $8,28($29)
38 slti  $1,$8, 101
3c bne   $1,$0, loop
40 lui   $4, l.str
44 ori   $4,$4,r.str
48 lw $5,24($29)
4c jal   printf
50 add   $2, $0, $0
54 lw    $31,20($29)
58 addiu $29,$29,32
5c jr  $31
```

Assemblagem: Passo 2

- Criar tabelas de símbolos e realocação
- Tabela de símbolos

Label	address (in module)	type
main:	0x00000000	global text
loop:	0x00000018	local text
str:	0x00000000	local data

- Tabela de realocação

Address	Instr. type	Dependency
0x00000040	lui	l.str
0x00000044	ori	r.str
0x0000004c	jal	printf

Assemblagem: Passo 3

- Resolução de *labels* locais relativos a PC

```
00 addiu $29,$29,-32
04 sw    $31,20($29)
08 sw    $4, 32($29)
0c sw    $5, 36($29)
10 sw    $0, 24($29)
14 sw    $0, 28($29)
18 lw    $14,28($29)
1c multu $14,$14
20 mflo  $15
24 lw    $24,24($29)
28 addu  $25,$24,$15
2c sw    $25, 24($29)
```

```
30 addiu $8,$14, 1
34 sw    $8,28($29)
38 slti  $1,$8, 101
3c bne   $1,$0, -10
40 lui   $4,l.str
44 ori   $4,$4,r.str
48 lw    $5,24($29)
4c jal   printf
50 add   $2, $0, $0
54 lw    $31,20($29)
58 addiu $29,$29,32
5c jr    $31
```

- Gerar ficheiro código objecto (.o):
  - Representação binária
    - Segmento de texto (instruções),
    - Segmento de dados,
    - Tabelas de símbolos e realocação.
  - Utiliza endereços “dummy” para referências não resolvidas (endereços absolutos e itens externos).

Segmento de Texto no ficheiro .o

```
0x000000    00100111101111011111111111100000
0x000004    10101111101111110000000000010100
0x000008    10101111101001000000000000100000
0x00000c    10101111101001010000000000100100
0x000010    10101111101000000000000000011000
0x000014    10101111101000000000000000011100
0x000018    10001111101011100000000000011100
0x00001c    10001111101110000000000000011000
0x000020    00000001110011100000000000011001
0x000024    00100101110010000000000000000001
0x000028    00101001000000010000000001100101
0x00002c    10101111101010000000000000011100
0x000030    00000000000000001111000000010010
0x000034    00000011000011111110010000100001
0x000038    00010100001000001111111111110111
0x00003c    10101111101110010000000000011000
0x000040    00111100000001000000000000000000
0x000044    10001111101001010000000000000000
0x000048    00001100000100000000000011101100
0x00004c    001001000000000000000000000000000
0x000050    10001111101111110000000000010100
0x000054    00100111101111010000000000100000
0x000058    00000011111000000000000000001000
0x00005c    00000000000000000001000000100001
```

Entradas na Tabela de realocação

Link passo 1: combina `prog.o`, `libc.o`

- Junta os segmentos de texto/dados
- Cria endereços absolutos de memória (o início do programa é 0x00000000)
- Modifica e concatena as tabelas de símbolos e realocação
- Tabela de símbolos
  - Label Address

```
main:     0x00000000
loop:     0x00000018
str:      0x10000430
printf:   0x000003b0   …
```

- Informação de realocação
  - Address Instr. Type Dependency

```
0x00000040   lui    l.str
0x00000044   ori    r.str
0x0000004c   jal    printf   …
```

Link passo 2:

- Edita endereços da tabela de realocação
  - (*mostrado em TAL por razões de clareza, mas feito em binário*)

```
00 addiu $29,$29,-32
04 sw    $31,20($29)
08 sw    $4, 32($29)
0c sw    $5, 36($29)
10 sw    $0, 24($29)
14 sw    $0, 28($29)
18 lw    $14, 28($29)
1c multu $14, $14
20 mflo  $15
24 lw    $24, 24($29)
28 addu  $25,$24,$15
2c sw    $25, 24($29)
```

```
30 addiu $8,$14, 1
34 sw    $8,28($29)
38 slti  $1,$8, 101
3c bne   $1,$0, -10
40 lui   $4, 4096
44 ori   $4,$4,1072
48 lw    $5,24($29)
4c jal   940
50 add   $2, $0, $0
54 lw    $31,20($29)
58 addiu $29,$29,32
5c jr    $31
```

Link passo 3:

- Executável:
  - Um único segmento de texto
  - Um único segmento de dados
  - Cabeçalho com informação da posição e tamanho de cada segmento (informação para o *loader*)

Exercício 1

Considere a codificação da instrução ‘beq $s0, $s1, init’ **que se encontra no excerto de código abaixo indicado. Sabendo que o “*label*”** salto **indica o endereço** *0xF2000*, **qual é o valor que se encontra no campo “*immediate*”?**

a) 0xFFFC
b) 0xFFFB
c) 0xFFFE
d) 0xF200

```
init: la    $s0, salto
      addi $s0, $s0, 12
      sra   $s0, $s0, 1
      beq  $s0, $s1, init
```

opcode	rs	rt	imediato

Exercício 2

Assumindo que a "label" *array* se refere a uma tabela de inteiros armazenada no endereço de memória *0x10000434*, e que as "labels" *func1* e *func2* são referências externas ao ficheiro, indique quantas entradas na tabela de realocação gerará o seguinte código Assembly do MIPS?

a) 2 entradas na tabela de realocação
b) 3 entradas na tabela de realocação.
c) 4 entradas na tabela de realocação.
d) 5 entradas na tabela de realocação.

```
loop:
    lw      $t0, 20($sp)
    lw      $t1, 24($sp)
    addu    $t2,$t1,$t0
    sw      $t2, 28($sp)
    blt     $t0,100, loop
    la      $a0, array
    lw      $t3, 24($a0)
    jal     func1
    move    $v0, $0
    lw      $ra, 16($sp)
    addiu $sp,$sp,32
    jal     func2
    ...
```

Para saber mais ...

- P&H - Capítulo 2.10, 2.12