Introdução à Arquitetura de Computadores

Aula21

Assembly 6: *Assembling & Loading*

Fases de Tradução dum Programa

Mapa de Memória
- Segmentos de Texto e de Dados

Assembler
- Diretivas
- Pseudo Instruções

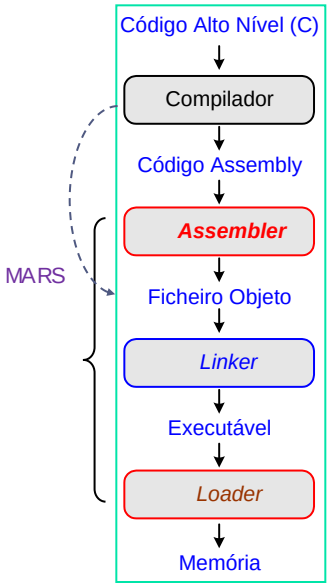
Código Executável
- *Linking e Loading*

*A. Nunes da Cruz / **DETI - UA*** *Maio / 2018*

4 - Assembling & Loading - Fases de Tradução*

O MARS implementa as funções de:
Assembler, *Linker* e de *Loader*, para além de simular o MIPS.

O *Assembler* produz código máquina e fornece informação ao *Linker* para construir um módulo executável que é carregado em memória pelo *Loader*.



4. Assembling & Loading

**Assemblar* e carregar um programa em memória.

4 - Assembling & Loading - Assembler

- O Assembler traduz o programa em código máquina.
- Produz informação suficiente para construir um *executável* com várias peças:
  - Cabeçalho (*Header*): O conteúdo do módulo *objeto*
  - Segmento de Texto (.text): Instruções traduzidas
  - Segmento de Dados estáticos (.data): Dados alocados enquanto o programa se mantém em execução
  - Tabela de Símbolos: definições globais de *Label*s de código e de variáveis.
  - Informação de Debug: Referências ao Código fonte, etc

4 - Assembling & Loading - Linker*

- Produz uma imagem executável

1. Junta os vários módulos *objeto* que constituem um programa. Por exemplo, adiciona uma biblioteca de funções gráficas.
2. Junta os vários segmentos dos módulos
3. Usa a Tabela de Símbolos para fazer os ajustes necessários ao código máquina gerado pelo *Assembler*.

* O MARS possui capacidades de *Linking* limitadas.

4 - Assembling & Loading - Loader

- Carrega o executável em memória
  1. Lê o cabeçalho (*header*) para determinar o tamanho dos segmentos.
  2. Copia o segmento de texto (.text) e o segmento de dados (.data) inicializados para a memória.
  3. Inicializa os registos (PC, incluindo $sp, $gp e $fp)
  4. Coloca os argumentos a passar ao *main* no *stack*
  5. Salta para a rotina de arranque, start-up.
     Copia os argumentos para $a0... e chama o *main*.
  6. Quando o *main* termina (jr $ra) o start-up executa a *syscall exit* (devolvendo o controlo ao SO).

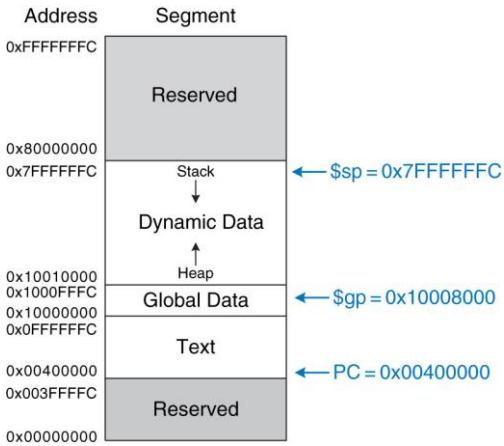
*Também pode chamar directamente o main (numa versão simplificada).

4 - Conteúdo da Memória - Secções de '.text', '.data', etc

- Instruções (tb designadas por *.text*)
- Dados (.data)
  - Estáticos (.asciiz, .word, etc)
    alocados e inicializados antes do programa iniciar a execução
  - Dinâmicos (*stack* e *heap*)
    alocados pelo programa em execução (*running*)
- Qual o tamanho da memória?
  - No máximo 2^{32} = 4 Gigabytes (4 GB)
  - Gama de endereços: 0x00000000 to 0xFFFFFFFF

4 - Mapa de Memória (1) - MIPS

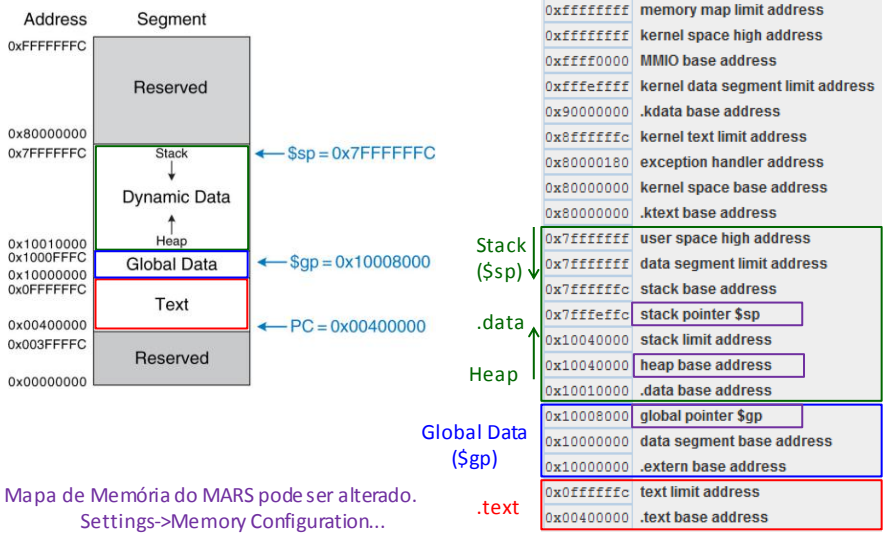
O Mapa de Memória (ie, a divisão do espaço de endereçamento de 4 GB) é específico de cada sistema



O mesmo CPU (eg, MIPS) pode usar diferentes mapas de memória.

4 - Mapa de Memória (1) - MIPS - MARS

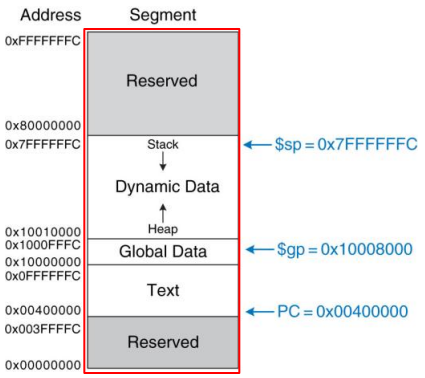
Mapa de Memória MARS (Default)



O Mapa de Memória do MARS pode ser alterado.
Settings->Memory Configuration...

4 - Mapa de Memória (2) - Espaço de Endereçamento

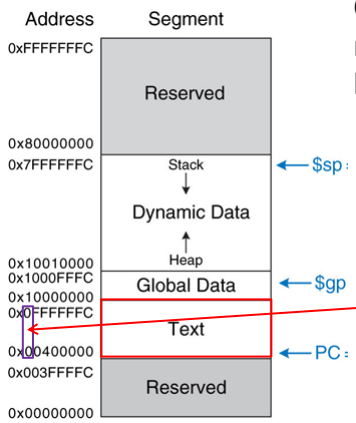
O espaço de endereçamento do MIPS é igual a 2^{32} bytes = 4 GB. Em endereços de *word* (divisiveis por 4) a gama estende-se de 0 to 0xFFFFFFFC.



Em geral, as arquiteturas MIPS dividem este espaço em 4 partes ou segmentos:

- texto
- dados dinâmicos
- dados globais
- reservado

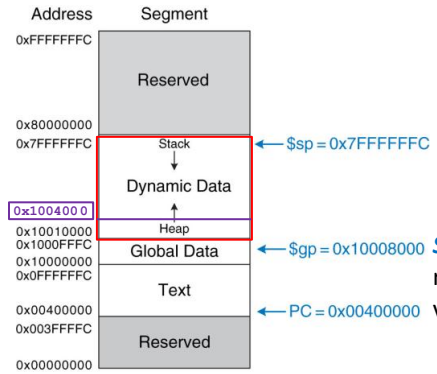
4 - Mapa de Memória (3) - Segmento de Texto



O segmento de texto armazena o código máquina do programa. É suficientemente longo para acomodar ~256MB .

Note-se que os 4 bits mais significativos do endereço do segmento de *texto* são todos 0, o que permite que a instrução j(ump) possa saltar diretamente para qualquer endereço do programa.

4 - Mapa de Memória (4) - Segmento de Dados Dinâmico



O segmento de dados dinâmico contém o *stack* e o *heap*.

Estes dados são alocados durante a execução do programa (e não antecipadamente). É o maior segmento do programa, ocupando quase 2GB (~256MB são para Texto).

Stack (*Pilha*): É usado para salvaguardar registos usados pelas funções e ainda por variáveis locais.

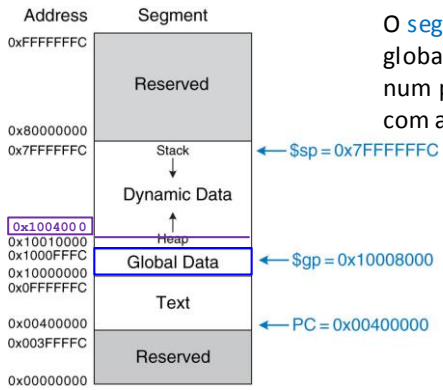
Valor Inicial $sp = 0x7FFFFFFC.

Heap (*Monte*): É usado para alocar dinamicamente (**sbrk** syscall) memória para as variáveis do programa. Ao contrário do stack, o modelo de acesso ao *heap* é aleatório (e não rígido do tipo LIFO). Valor inicial heap = 0x10040000.

.data-estática: No MARS, a faixa inicial do segmento de .data é usada para alocar memória para as variáveis estáticas locais ao programa.

Gama: 0x10010000 - 0x10040000 (192kB)

4 - Mapa de Memória (5) - Segmento de Dados Global



O segmento de dados global é para as variáveis globais, as quais são visíveis a todas as funções num programa, ao contrário do que acontece com as variáveis locais das funções.

A utilização deste segmento depende do Assembler usado. O MARS tem um Assembler mais simples, por isso é de esperar algumas discrepâncias com o que vem no livro! Mais a seguir...

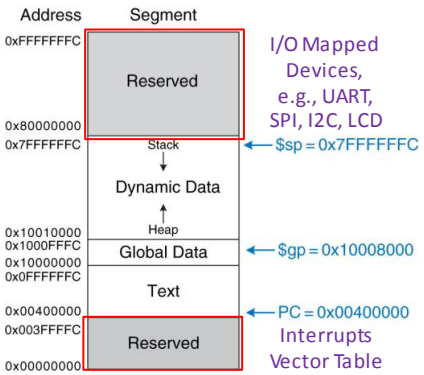
As variáveis globais são acedidas através do ponteiro global ($gp). O ponteiro $gp, ao invés do stack pointer ($sp), não se altera durante a execução do programa.

Qualquer váriável global pode ser acedida através dum *offset* de 16-bits relativo ao $gp, o qual é usado como Endereço-Base. O *offset* é determinado durante a fase de 'assemblagem' (mas depende do Assembler como veremos).

*O Segmento de Dados Global não é usado nas aulas TP de IAC!

4 - Mapa de Memória (6) - Segmentos Reservados

Os segmentos reservados são usados pelo Sistema Operativo e não estão acessíveis diretamente ao programa do utilizador (*user mode*).



Últimas aulas de IAC!?

Parte desta memória é usada na gestão de interrupções (Tabela de Vectores) e ainda para mapear periféricos em *janelas-de-memória* (*Memory-mapped I/O*).

5 - O Assembler (1) - Tarefas

- Seguir *Diretivas*
  <Instruções> para o Assembler
- Traduzir *Pseudo-Instruções*
  Em instruções nativas
- Construir a *Tabela de Símbolos*
  Lista de *Labels* (*.text* e *.data*) e endereços
- Gerar código máquina
  A partir das instruções nativas
- Criar módulo/ficheiro *Objeto*

Em geral, compete a um *Linker* criar o módulo/ficheiro *Executável*, a partir de vários objetos. Porém, estas funções: *Assembler*, *Linker* e *Loader* estão todas integradas no MARS.

5 - O Assembler (2) - Diretivas

'Instruções' para o *Assembler* (não geram código máquina!)

.align n	Align the next datum on a 2^n byte boundary
.text	Subsequent items put in user text segment
.data	Subsequent items put in user data segment
.globl sym	sym can be referenced from other files
.asciiz str	Store the string str in memory
.space n	Reserve space for n successive bytes
.word w1,...,wn	Store the *n* 32-bit quantities in successive memory words
.byte b1,...,bn	Store n 8-bit values in successive bytes of memory
.float f1,...,fn	Store n floating-point numbers in successive memory words

5 - O Assembler (3) - Pseudo-Instruções

As pseudo-instruções proporcionam ao programador uma máquina virtual mais fácil (e mais poderosa), tornando o código ASM mais legível. Isto é comum a todos *Assemblers* de CPUs RISC!

Pseudo-Instrução	Instrução MIPS
`li $s0, 0x1234AA77`	`lui $s0, 0x1234` `ori $s0, 0xAA77`
`move $s1, $s2`	`add $s2, $s1, $0`
`nop`	`sll $0, $0, 0`

Compete ao *Assembler* traduzir as pseudo-instruções (ou instruções virtuais) em instruções nativas.

6 - Loading (1) - Programa em C - Exemplo

Fases de geração do Código Executável

1. *C* para ASM e
2. ASM para código máquina e dados (.text + .data),
3. Criação do executável a carregar em memória.

```
int f, g, y; // globals
int sum(int a, int b);

int main(void) {
    f = 2;
    g = 3;
    y = sum(f, g);

    return y;
}
int sum(int a, int b) {
    return (a + b);
}
```


6 - Loading (2) - Programa em Assembly - MARS

```
int f, g, y;      // globals
int sum(int a, int b);

int main(void) {
    f = 2;
    g = 3;
    y = sum(f, g);

    return y;
}

int sum(int a, int b){
    return (a + b);
}
```

O Assembler do MARS não permite obter o código exacta/ como vem no livro. Para aceder às variáveis globais através do $gp temos de calcular os **offsets** no segmento global **manualmente**!

```
# variables f,g,y in global
# data segment at 0x10000000
      .extern f, 4    # 0x10008000-0x8000 = 0x10000000
      .extern g, 4    # 0x10008000-0x7FFC = 0x10000004
      .extern y, 4    # 0x10008000-0x7FF8 = 0x10000008
# offsets to $gp = 0x10008000
      .eqv, f_off, -0x8000     # manually! ☺
      .eqv, g_off, -0x7FFC
      .eqv, y_off, -0x7FF8
#
      .text
      .globl main
main: addi    $sp, $sp, -4     # stack frame
      sw      $ra, 0($sp)      # store $ra
      addi    $a0, $0, 2       # $a0 = 2
#     sw      $a0, f           # f = 2, NOT in MARS
      sw      $a0, f_off($gp)  # f = 2
      addi    $a1, $0, 3       # $a1 = 3
#     sw      $a1, g           # g = 3, NOT in MARS
      sw      $a1, g_off($gp)  # g = 3
      jal     sum              # call sum
#     sw      $v0, y           # y = sum, NOT in MARS
      sw      $v0, y_off($gp)  # y = sum(f,g)
      lw      $ra, 0($sp)      # restore $ra
      addi    $sp, $sp, 4      # restore $sp
      jr      $ra              # return to OS
      #
sum:  add     $v0, $a0, $a1    # $v0 = a + b
      jr      $ra              # return
```

Outros Assemblers calculam estes offsets automatica/.

Diferente do Livro!

6 - Loading (3) - Geração do Código Executável

O *Assembler** percorre o programa duas vezes:

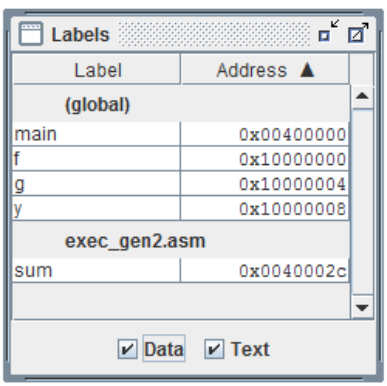
Fase 1 - Constroi a Tabela de Símbolos
Tabela de Endereços dos Labels (.data + .text)

Fase 2 - Gera o Código máquina
Usa a Tabela de Símbolos e a
Tabela do Código das Instruções do MIPS

* O MARS para além de *Assembler*, implementa tb as funções de *Linker* e de *Loader*.

6 - Loading (4) - Tabela de Símbolos

Para o Exemplo anterior:



Símbolo	Endereço
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

Disponível no MARS: Labels (sub)window: segmentos .text + .data ambos selecionados

6 - Loading (5) - Programa em Assembly - MARS

Address	Code	Basic				Source
0x00400000	0x23bdfffc	addi $29,$29,0xfffffffc	18: main:	addi	$sp, $sp, -4	# stack frame
0x00400004	0xafbf0000	sw $31,0x00000000($29)	19:	sw	$ra, 0($sp)	# store $ra
0x00400008	0x20040002	addi $4,$0,0x00000002	20:	addi	$a0, $0, 2	# $a0 = 2
0x0040000c	0xaf848000	sw $4,0xffff8000($28)	22:	sw	$a0, -0x8000($gp)	# f = 2
0x00400010	0x20050003	addi $5,$0,0x00000003	23:	addi	$a1, $0, 3	# $a1 = 3
0x00400014	0xaf858004	sw $5,0xffff8004($28)	25:	sw	$a1, -0x7FFC($gp)	# g = 3
0x00400018	0x0c10000b	jal 0x0040002c	26:	jal	sum	# call sum
0x0040001c	0xaf828008	sw $2,0xffff8008($28)	28:	sw	$v0, -0x7FF8($gp)	# y = sum(f,g)
0x00400020	0x8fbf0000	lw $31,0x00000000($29)	29:	lw	$ra, 0($sp)	# restore $ra
0x00400024	0x23bd0004	addi $29,$29,0x00000004	30:	addi	$sp, $sp, 4	# restore $sp
0x00400028	0x03e00008	jr $31	31:	jr	$ra	# return to OS
0x0040002c	0x00851020	add $2,$4,$5	33: sum:	add	$v0, $a0, $a1	# $v0 = a + b
0x00400030	0x03e00008	jr $31	34:	jr	$ra	# return

```
                # f,g,y in global data segment ($gp)
                .text
main:   addi    $sp, $sp, -4        # stack frame
        sw      $ra, 0($sp)         # store $ra
        addi    $a0, $0, 2          # $a0 = 2
        sw      $a0, -0x8000($gp)   # f = 2
        addi    $a1, $0, 3          # $a1 = 3
        sw      $a1, -0x7FFC($gp)   # g = 3
        jal     sum                 # call sum
        sw      $v0, -0x7FF8($gp)   # y = sum(f,g)
        lw      $ra, 0($sp)         # restore $ra
        addi    $sp, $sp, 4         # restore $sp
        jr      $ra                 # return to OS
        #
sum:    add     $v0, $a0, $a1       # $v0 = a + b
        jr      $ra                 # return
```

Não contem pseudo-instruções, para simplificar!

*$sp : addi ->addiu

6 - Loading (6) - Ficheiro* Executável

Executable File Header

Text Size	Data Size
0x34 (52 bytes)	0xC (12 bytes)

Text Segment

Address	Instruction
0x00400000	0x23BDFFFC
0x00400004	0xAFBF0000
0x00400008	0x20040002
0x0040000C	0xAF848000
0x00400010	0x20050003
0x00400014	0xAF858004
0x00400018	0x0C10000B
0x0040001C	0xAF828008
0x00400020	0x8FBF0000
0x00400024	0x23BD0004
0x00400028	0x03E00008
0x0040002C	0x00851020
0x00400030	0x03E00008

```
addi    $sp, $sp, -4
sw      $ra, 0($sp)
addi    $a0, $0, 2
sw      $a0, -0x8000($gp)
addi    $a1, $0, 3
sw      $a1, -0x7FFC($gp)
jal     0x0040002C
sw      $v0, -0x7FF8($gp)
lw      $ra, 0($sp)
addi    $sp, $sp, -4
jr      $ra
add     $v0, $a0, $a1
jr      $ra
```

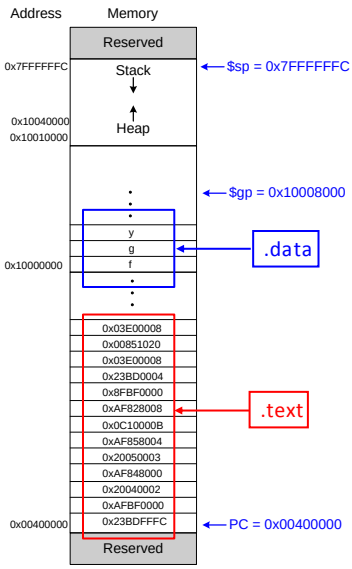
Data Segment

Address	Data
0x10000000	f
0x10000004	g
0x10000008	y

*Não no MARS!

*$sp : addi ->addiu

6 - Loading (7) - Executável Em Memória



XX - NEXT DataPath Single-Cycle