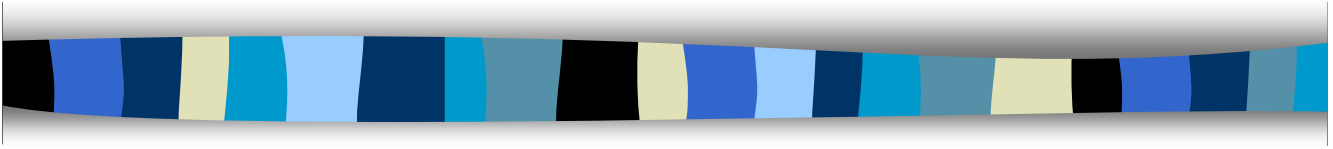


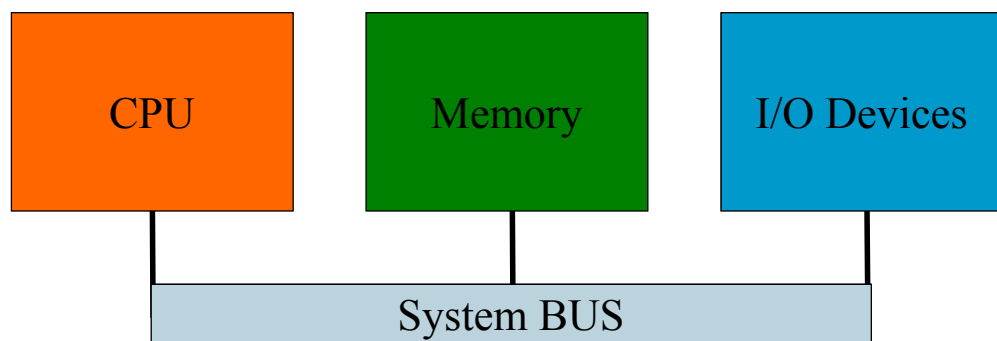
Sistemas da Computação



Linguagem Assembly do MIPS - I Introdução, Operações Aritméticas e Lógicas

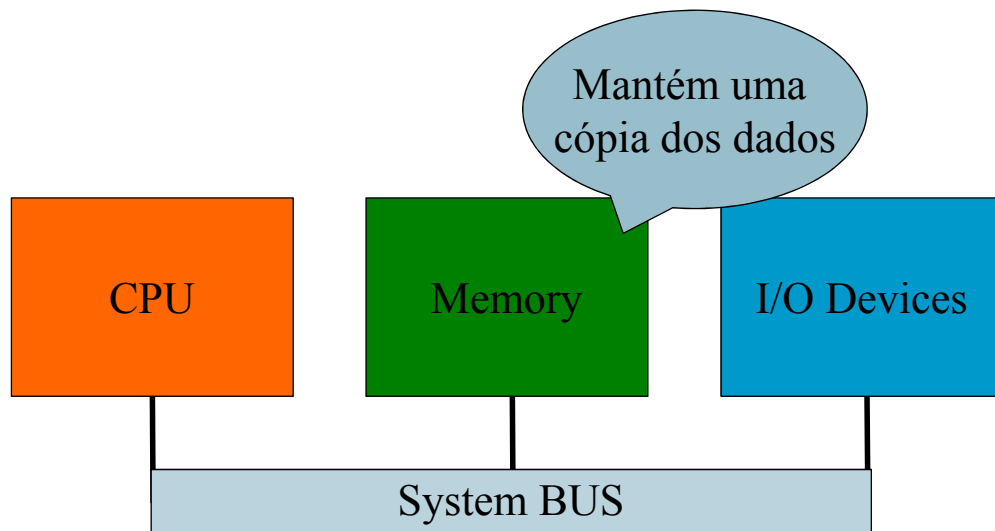
1

Arquitetura de von Neumann



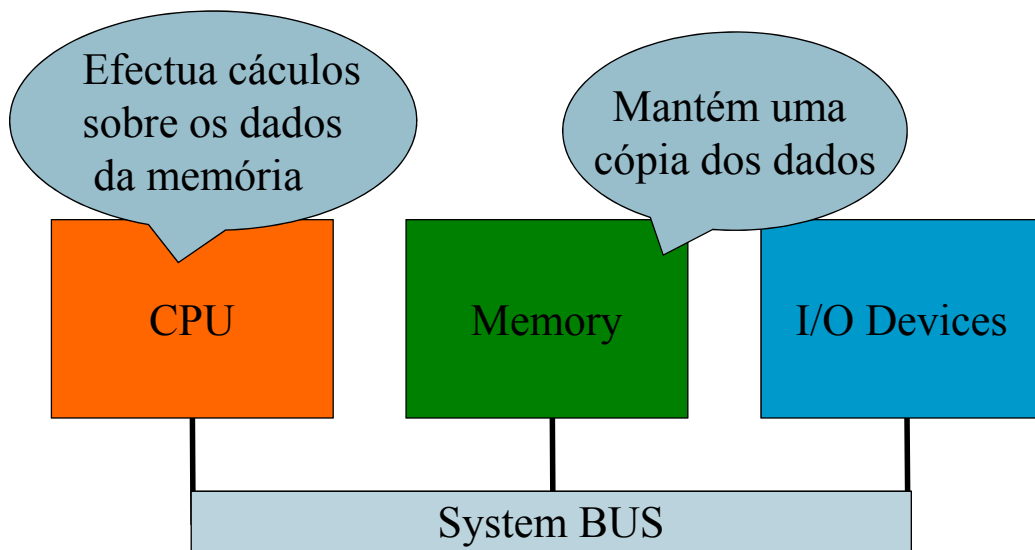
2

Arquitetura de von Neumann



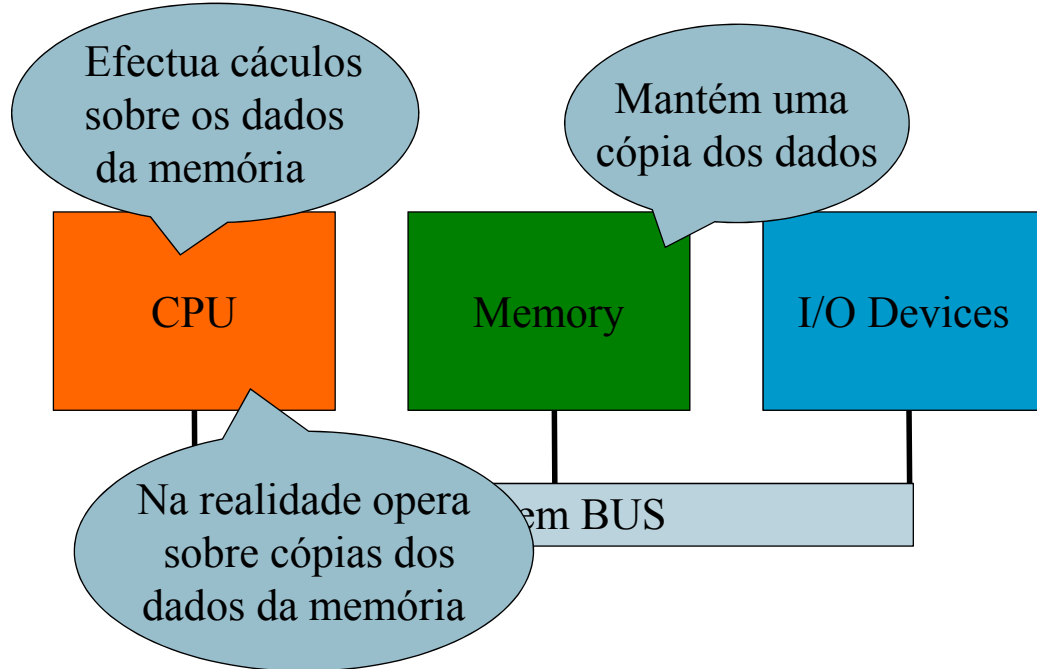
3

Arquitetura de von Neumann



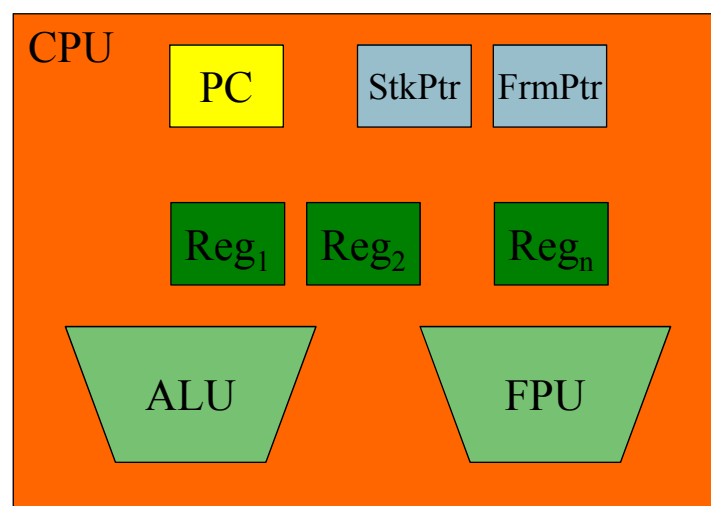
4

Arquitetura de von Neumann



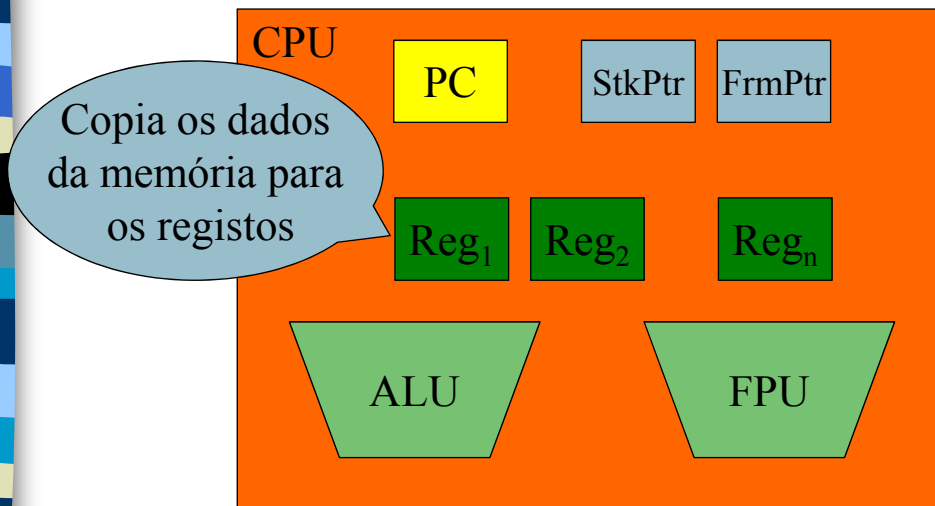
5

Arquitetura de von Neumann



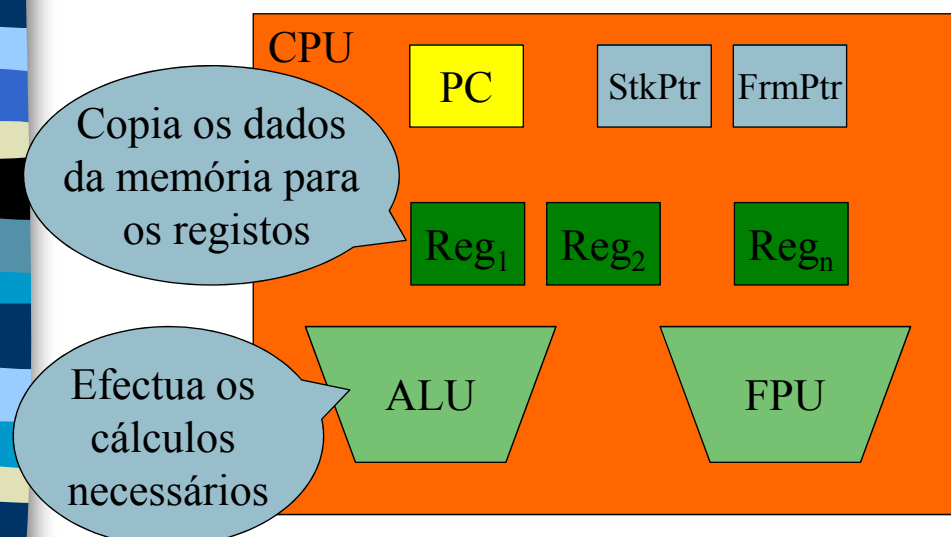
6

Arquitetura de von Neumann



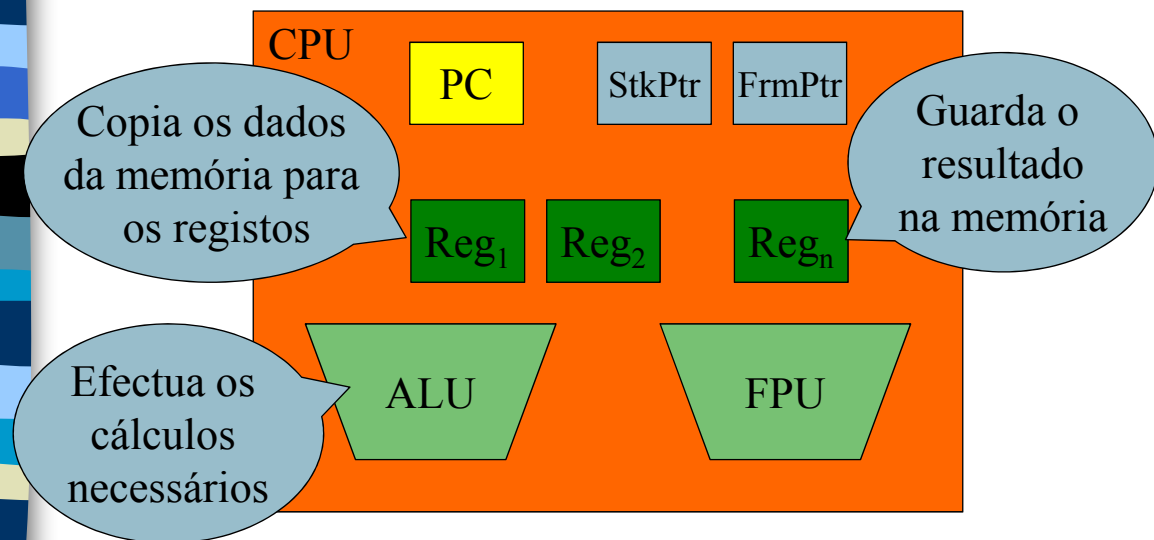
7

Arquitetura de von Neumann



8

Arquitetura de von Neumann



9

Arquitetura de von Neumann



10



Dar ordens a um CPU

- O fabricante define o conjunto de instruções do seu CPU (arquitetura)
 - Instruções e modo de uso
- A linguagem do CPU é definida
 - Conjunto de instruções (sintaxe)
 - Regras das instruções (semântica)

11



Dar ordens a um CPU

- O fabricante define o conjunto de instruções do seu CPU (arquitetura)
 - Instruções
- A linguagem do CPU é definida
 - Conjunto de instruções (sintaxe)
 - Regras das instruções (semântica)

Todas as linguagens
são definidas
deste modo

12



Dar ordens a um CPU

- O fabricante define o conjunto de instruções (dependendo da arquitetura)
 - Instruções
- A linguagem é definida (dependendo da arquitetura)
 - Conjunto de instruções (sintaxe)
 - Regras das instruções (semântica)

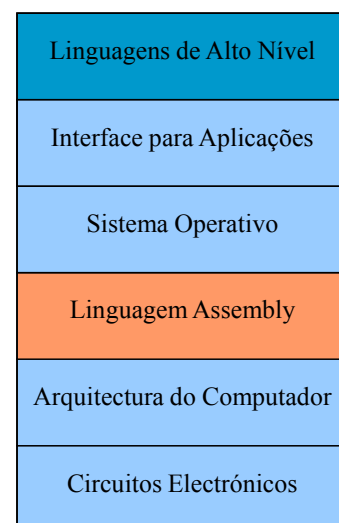
Linguagem
Assembly

13



Níveis de Abstração

- Uma instrução em linguagem assembly (pura) corresponde a uma operação básica do processador (ex: somar dois inteiros, comparar dois números, etc)
- As linguagens assembly suportam normalmente pseudo-instruções que correspondem a mais do que uma instrução de linguagem assembly



14



Exemplo Assembly

Exemplo

```
                .data
item:           .word      1
                .text
                .globl __start
__start:        lw $t0, item
```

15



Directivas de Assemblagem

Segmentos e símbolos

.text <addr> - Os próximos itens (instruções ou palavras) são montados no segmento de texto.

- Se estiver presente o argumento *addr*, a montagem é feita a partir do endereço *addr*.

.data <addr> - Os próximos itens são montados no segmento de dados.

- Se estiver presente o argumento *addr*, a montagem é feita a partir do endereço *addr*.

.globl sym - Declara *sym* como um símbolo geral (global) que também pode ser referenciado noutros ficheiros.

16



Directivas de Assemblagem

Tipos de Dados

.byte b1, b2, ... bn - guarda os *n bytes* sequencialmente na memória.

.half h1, h2, ... hn - guarda as *n half-words* sequencialmente na memória.

.word w1, w2, ... wn - guarda as *n words* sequencialmente na memória.

.ascii str - guarda a cadeia de caracteres *str* sequencialmente na memória.

17



Directivas de Assemblagem

Exemplo

```
item:      .data
           .word      1
           .text
           .globl __start
__start:   lw $t0, item
```

18



Sintaxe de montagem

Comentários: iniciam-se com “#”

Identificadores: cadeia de caracteres alfanuméricos (incluindo “_” e “.”) não iniciada por um algarismo

Instruções: símbolos reservados que não podem ser usados como identificadores

Etiquetas: identificadores declarados no início de uma linha e terminados por “:”

Cadeia de caracteres: definida entre aspas seguindo a convenção “\n” para nova linha, “\t” para tabulação e “\” para aspas.

19



MIPS

- Processador de 32-bits
 - 32 registos de 32-bits
 - Versões mais recentes de 64-bits
- Arquitetura RISC
 - Reduced instruction set computer
- Cache
 - 32 KB dados e 63KB instruções
- ...

20

Registos

Nome	Número	Utilização
zero	0	Constante 0
at	1	Reservado ao <i>assembler</i>
v0 .. v1	2 .. 3	Resultado de uma função/procedimento
a0 .. a3	4 .. 7	Argumentos 1, 2, 3 e 4
t0 .. t7	8 .. 15	Temporários (não preservados entre chamadas)
s0 .. s7	16 .. 23	Persistentes (preservados entre chamadas)
t8 a t9	24 .. 25	Temporários (não preservados entre chamadas)
k0 .. k1	26 .. 27	Reservados ao <i>kernel</i> do S.O.
gp	28	Ponteiro para a área global (dados estáticos)
sp	29	Ponteiro da <i>stack</i>
fp	30	Ponteiro da <i>frame</i>
ra	31	Endereço de retorno (usado pela chamada de uma função)

21

Memória

- Endereços de 32 bits (4 bytes)
- MIPS faz endereçamento ao byte (little endian)
- little endian: bit menos significativo está no endereço menor e uma word é endereçada pelo endereço do byte menos significativo

11	10	9	8
7	6	5	4
3	2	1	0

Endereço word

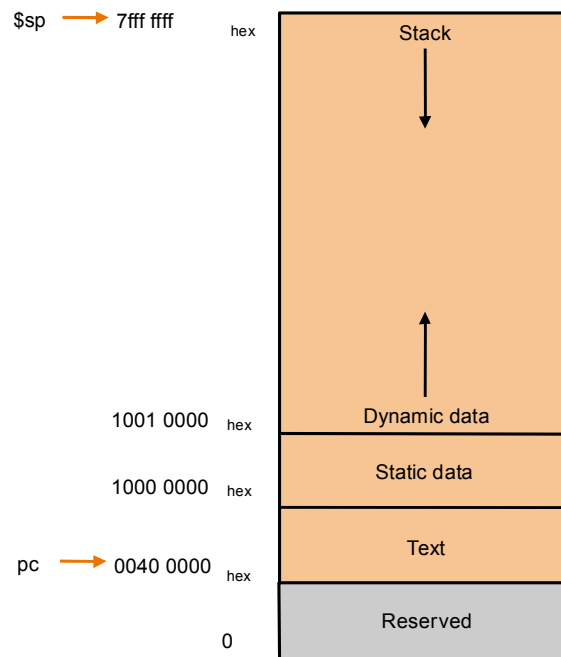
8

4

0

22

Memória



23

Instruções no MIPS

load (carregar dados da memória para um registo)

`lw` (word), `lh` (half-word) e `lb` (byte)

`l? reg, offset(registo base)`

carrega os dados na posição de memória obtida a partir da soma do ponteiro guardado no `registo base` com o `offset` (deslocamento) e guarda-os no registo `reg`.

Exemplo: `lw $t0, 8($s3)`

24

Instruções no MIPS

load (carregar dados da memória para um registo)

Registo de destino

, lh (half-word) e lb (byte)

Endereço de memória de origem

l? reg, offset(registo base)

carrega os dados na posição de memória obtida a partir da soma do ponteiro guardado no registo base com o offset (deslocamento) e guarda

MEM[\$s3+8]

Exemplo: lw \$t0, 8(\$s3)

25

Instruções no MIPS

store (guarda dados de um registo na memória)

sw (word), sh (half-word) e sb (byte)

s? reg, offset(registo base)

guarda os dados do registo reg na posição de memória obtida a partir da soma do ponteiro guardado no registo base com o offset (deslocamento).

Exemplo: sb \$s1, 3(\$s2)

26

Instruções no MIPS

store (guarda dados de um registo na memória)

Registo de
origem

, sh (half-word) e sb (byte)

Endereço de
memória de
destino

s? reg, `offset(registo base)`

guarda os dados do registo `reg` na posição de memória obtida a partir da soma do ponteiro guarda o registo base com o offset (deslocamen

`MEM[$s2+3]`

Exemplo: `sb $s1, 3($s2)`

27

Instruções no MIPS

add

`add reg1, reg2, reg3`

adiciona `reg2` a `reg3` e coloca resultado em `reg1`

Exemplo: `add $t0, $t1, $s0`

Adicionar mais que dois valores ($D = A + B + C$):

`add $t0, $s0, $s1`

`add $s3, $t0, $s2`

28

Instruções no MIPS

add

reg1 = reg2 + reg3

add reg1, reg2, reg3

adiciona reg2 a reg3 e coloca resultado em reg1

Exemplo: add \$t0, \$t1, \$s0

Adicionar mais que dois valores ($D = A + B + C$):

add \$t0, \$s0, \$s1

add \$s3, \$t0, \$s2

29

Instruções no MIPS

sub

sub reg1, reg2, reg3

subtrai reg3 a reg2 e coloca resultado em reg1

Exemplo: sub \$t3, \$s1, \$s2

30

Instruções no MIPS

sub

reg1 = reg2 - reg3

sub reg1, reg2, reg3

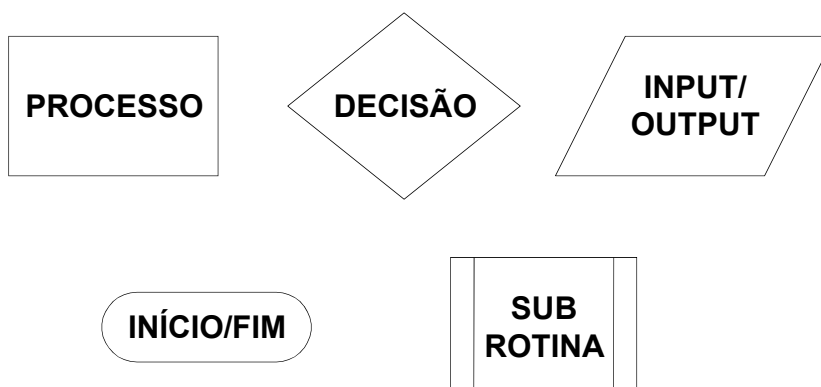
subtrai reg3 a reg2 e coloca resultado em reg1

Exemplo: sub \$t3, \$s1, \$s2

31

Fluxogramas

- Representação do fluxo de um programa
- Simbologia:



32

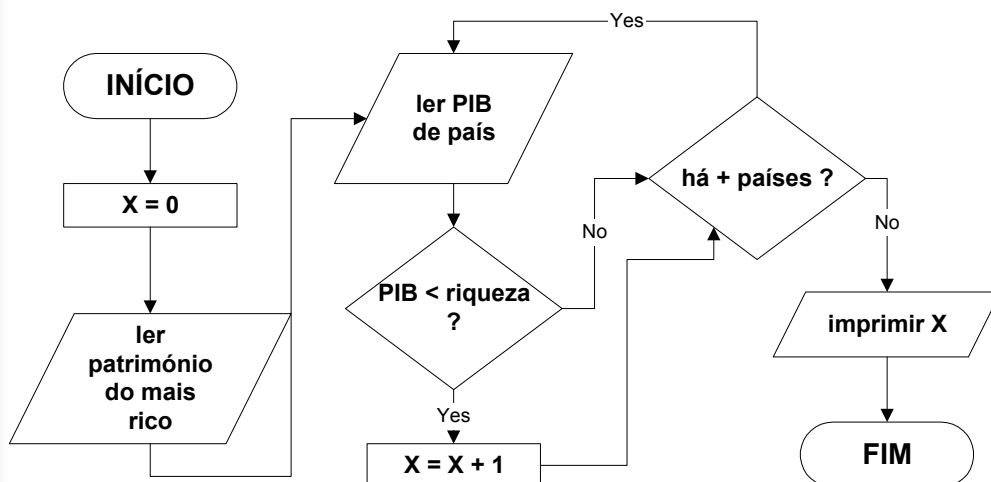
Fluxogramas

- **Exemplo:** saber quantos países têm o PIB menor que o património da pessoa mais rica do mundo

33

Fluxogramas

- **Exemplo:** saber quantos países têm o PIB menor que o património da pessoa mais rica do mundo



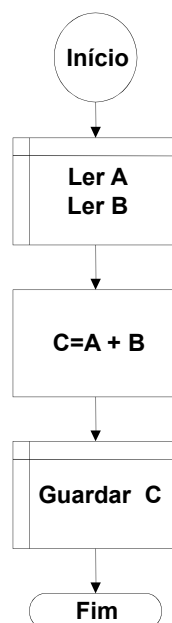
34

Problema 1

- Dados dois valores inteiros positivos em memória, armazene o resultado da sua soma na posição de memória seguinte
- Considere os valores iniciais entre 0 e 127

35

Diagrama de fluxo



36

Programa Assembler

```
.data
A: .byte 7
B: .byte 4
C: .byte 0
.text
.globl __start
__start:
    lb    $s0, A
    lb    $s1, B
    add   $s2, $s1, $s0
    sb    $s2, C
```

37

...do SPIM

Text Segment

```
=====
[0x00400000] 0x3c011001 lui $1, 4097 ; 8: lb $s0, A
[0x00400004] 0x80300000 lb $16, 0($1)
[0x00400008] 0x3c011001 lui $1, 4097 ; 9: lb $s1, B
[0x0040000c] 0x80310001 lb $17, 1($1)
[0x00400010] 0x02309020 add $18, $17, $16 ; 10: add $s2,$s1,$s0
[0x00400014] 0x3c011001 lui $1, 4097 ; 11: sb $s2,C
[0x00400018] 0xa0320002 sb $18, 2($1)
```

Data Segment

```
=====

DATA
[0x10000000]...[0x1000fffc] 0x00000000
[0x1000fffc] 0x00000000
[0x10010000] 0x00000407 0x00000000 0x00000000 0x00000000
[0x10010010]...[0x10040000] 0x00000000

STACK
[0x7ffefffc] 0x00000000
```

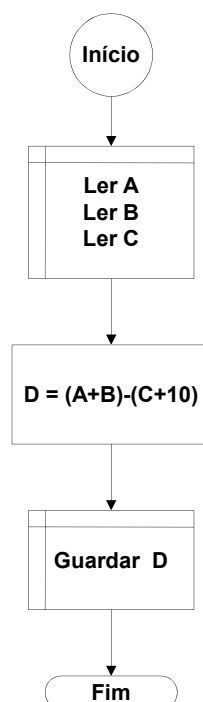
38

Problema 2

- Dados três valores inteiros positivos (A, B e C) em memória, armazene o resultado da operação $(A+B)-(C+10)$ na posição de memória seguinte
- Considere os valores iniciais entre 0 e 32767 (16 bits)

39

Diagrama de fluxo



40

Programa Assembly

```
.data
varA: .half 3200
varB: .half 127
varC: .half 1024
varD: .half 0
.text
.globl __start
__start:
    lh      $s0, varA
    lh      $s1, varB
    lh      $s2, varC
    add     $t0, $s0, $s1
    addi    $t1, $s2, 10
    sub     $s3, $t0, $t1
    sh      $s3, varD
```

41

...do SPIM

Text Segment

```
=====
[0x00400000]      0x3c011001 lui $1, 4097          ; 9: lh      $s0, varA
[0x00400004]      0x8c300000 lh $16, 0($1)
[0x00400008]      0x3c011001 lui $1, 4097          ; 10: lh     $s1, varB
[0x0040000c]      0x8c310004 lh $17, 2($1)
[0x00400010]      0x3c011001 lui $1, 4097          ; 11: lh     $s2, varC
[0x00400014]      0x8c320008 lh $18, 4($1)
[0x00400018]      0x02114020 add $8, $16, $17      ; 12: add    $t0, $s0, $s1
[0x0040001c]      0x2249000a addi $9, $18, 10      ; 13: addi   $t1, $s2, 10
[0x00400020]      0x01099822 sub $19, $8, $9       ; 14: sub    $s3, $t0, $t1
[0x00400024]      0x3c011001 lui $1, 4097          ; 15: sh     $s3, varD
[0x00400028]      0xac33000c sh $19, 6($1)
```

Data Segment

```
=====
DATA
[0x10000000]...[0x1000fffc] 0x00000000
[0x1000fffc]      0x00000000
[0x10010000]      0x007f0c80 0x00000400 0x00000000 0x00000000
[0x10010010]...[0x10040000] 0x00000000
```

STACK

```
[0x7ffefffc]      0x00000000
```

42



Instruções no MIPS

Operações Aritméticas

mult, mflo, mfhi

`mult reg1, reg2`

multiplica `reg1` por `reg2` e coloca resultado em `hi/lo`

Exemplo: `mult $t0, $t1`

`mflo reg1`

move o conteúdo de `lo` para `reg1`

`mfhi reg1`

move o conteúdo de `hi` para `reg1`

43



Instruções no MIPS

Operações Aritméticas

div, mflo, mfhi

`div reg1, reg2`

divide `reg1` por `reg2` e coloca resultado em `lo` e o resto da divisão em `hi`

Exemplo: `div $t0, $t1`

`mflo reg1`

move o conteúdo de `lo` para `reg1`

`mfhi reg1`

move o conteúdo de `hi` para `reg1`

44



Instruções no MIPS

Operações Aritméticas

Exercício:

Escreva um programa, em linguagem Assembly do MIPS, que calcula a média entre dois números inteiros de 32 bits. Repita o mesmo exercício usando a pseudo-instrução `div` (`div rdest, reg1, reg2`)

45



Estruturas de controlo

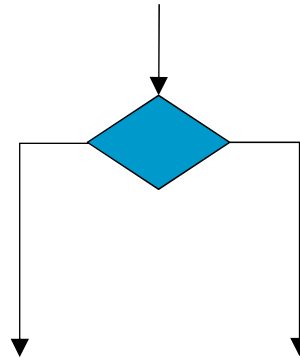
46

Instruções no MIPS

Estruturas de Controlo

Permitem criar fluxos alternativos de execução

se (condição) então x
senão y



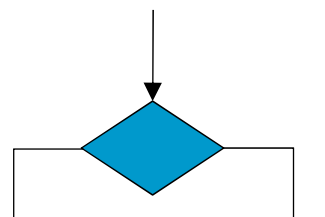
47

Instruções no MIPS

Estruturas de Controlo

Permitem criar fluxos alternativos de execução

se (condição) então x
senão y



Mas o código é sequencial, como
criamos caminhos distintos de execução?

48



Instruções no MIPS

Estruturas de Controlo

Saltos (jump)
j destino

Não estão associados
a uma condição

Saltos condicionais (branch)
b? reg1, reg2, destino

Estão associados
a uma condição

49



Instruções no MIPS

Estruturas de Controlo

Saltos (jump)
j

Não estão associados

Mas o código é sequencial, como
criamos caminhos distintos de execução?

Saltando por cima do que não interessa 😊

Saltos condicionais (branch)

Estão associados
a uma condição

b? reg1, reg2, destino

50



Instruções no MIPS

Estruturas de Controlo

Salto (sintaxe)

j destino

51



Instruções no MIPS

Estruturas de Controlo

Salto (sintaxe)

j destino

```
main:  li $s0, 10
        li $s1, 20

        j xpto
        add $s2, $s0, $s1

xpto:  sub $s2, $s0, $s1
```

52



Instruções no MIPS

Estruturas de Controlo

Salto (sintaxe)

j destino

```
main: li $s0, 10
      li $s1, 20

      j xpto
      add $s2, $s0, $s1

      xpto sub $s2, $s0, $s1
```

53



Instruções no MIPS

Estruturas de Controlo

Salto (sintaxe)

j destino

**Alguma vez
executamos esta
instrução?**

```
main: li $s0, 10
      li $s1, 20

      j xpto
      add $s2, $s0, $s1

      xpto sub $s2, $s0, $s1
```

54



Instruções no MIPS

Estruturas de Controlo

Salto condicionais (sintaxe)

b? reg1, reg2, destino

b destino → semelhante a j destino

bgt reg1, reg2, destino → se $\text{reg1} > \text{reg2}$ salta

bge reg1, reg2, destino → se $\text{reg1} \geq \text{reg2}$ salta

blt reg1, reg2, destino → se $\text{reg1} < \text{reg2}$ salta

ble reg1, reg2, destino → se $\text{reg1} \leq \text{reg2}$ salta

beq reg1, reg2, destino → se $\text{reg1} = \text{reg2}$ salta

bne reg1, reg2, destino → se $\text{reg1} \neq \text{reg2}$ salta

55



Instruções no MIPS

Estruturas de Controlo

Salto condicionais (sintaxe)

Exemplo:

Se o valor no registo s0 for superior que o valor do registo s1 então somamos os dois valores. Se não for subtraímos.

56



Instruções no MIPS

Estruturas de Controle

Saltos condicionais (sintaxe)

Exemplo:

```
main:  li $s0, 20
        li $s1, 10

        bgt $s0, $s1, xpto
        add $s2, $s0, $s1

xpto:   sub $s2, $s0, $s1
```

57



Instruções no MIPS

Estruturas de Controle

Saltos condicionais (sintaxe)

Exemplo:

```
main:  li $s0, 20
        li $s1, 10

        bgt $s0, $s1, xpto
        add $s2, $s0, $s1

xpto:   sub $s2, $s0, $s1
```

58

Instruções no MIPS

Estruturas de Controlo

Saltos condicionais (sintaxe)

Exemplo:

**E se os trocarmos
os valores?**

```
main:  li $s0, 10  
       li $s1, 20  
  
       bgt $s0, $s1, xpto  
       add $s2, $s0, $s1  
  
xpto:  sub $s2, $s0, $s1
```

59

Instruções no MIPS

Estruturas de Controlo

Saltos condicionais (sintaxe)

Exemplo:

**A condição é
falsa, logo não
saltamos para 'xpto'**

```
main:  li $s0, 10  
       li $s1, 20  
  
       bgt $s0, $s1, xpto  
       add $s2, $s0, $s1  
  
xpto:  sub $s2, $s0, $s1
```

60



Instruções no MIPS

Estruturas de Controlo

Saltos condicionais (sintaxe)

Exemplo:

**E executamos ambas
as instruções seguintes**

```
main:  li $s0, 10
        li $s1, 20

        bgt $s0, $s1, xpto
        add $s2, $s0, $s1

xpto:  sub $s2, $s0, $s1
```

61



Instruções no MIPS

Estruturas de Controlo

Saltos condicionais (sintaxe)

Exemplo:

```
main:  li $s0, 10
        li $s1, 20

        bgt $s0, $s1, xpto
        add $s2, $s0, $s1
        j end

xpto:  sub $s2, $s0, $s1
end:
```

62



Instruções no MIPS

Estruturas de Controlo

Salto condicionais (sintaxe)

Exemplo:

É necessário um salto para evitar o código referente à condição

```
main:  li $s0, 10
        li $s1, 20

        bgt $s0, $s1, xpto
        add $s2, $s0, $s1
        j end
xpto:   sub $s2, $s0, $s1
end:
```

63



Instruções no MIPS

Estruturas de Controlo

Exercício:

Dado dois valores inteiros, guardados em duas posições contíguas de memória, determinar qual o menor dos valores e guarda-lo na próximo endereço de memória. Caso sejam iguais, o resultado deverá ser -1.

64



Instruções no MIPS

Estruturas de Controlo

Nada nos impede de fazer os nossos programas saltar para uma instrução previamente executada

É assim que se criam ciclos

Que faz este programa?

```
main:  li $s0, 1
        li $s1, 20

abc:    bgt $s0, $s1, xpto
        addi $s0, $s0, 1
        j abc

xpto:
```

65



Instruções no MIPS

Operações Lógicas

and, or

`and rdest, reg1, reg2`

and faz o *e lógico bit a bit* de `reg1` e `reg2` e coloca resultado em `rdest`

`or rdest, reg1, reg2`

or faz o *ou lógico bit a bit* de `reg1` e `reg2` e coloca resultado em `rdest`

66



Instruções no MIPS

Operações Lógicas

sll, slr

`sll rdest, reg1, reg2`

`sll` afasta para a esquerda todos os bits de `reg1`, o número de bits contido em `reg2` e coloca o resultado em `rdest`

`slr rdest, reg1, reg2`

`slr` afasta para a direita todos os bits de `reg1`, o número de bits contido em `reg2` e coloca o resultado em `rdest`

67



Instruções no MIPS

Operações Lógicas

Exercício:

Utilizando apenas operações lógicas escreva um programa Assembly do MIPS que converte quatro dígitos decimais representados em ASCII para o código *packet BCD*. Suponha que os quatro dígitos em ASCII estão guardados em 4 posições contíguas da memória. O resultado da conversão deverá ser armazenado na posição de memória seguinte.

68



Arrays

69



Arrays (vectors)

Permitem manter, em memória, conjuntos de valores normalmente relacionados

Ex: notas de alunos, temperaturas, etc.

Representados por um conjunto de endereços de memória contíguos

i.e., posições de memória seguidas

Acedidos por indexação

i.e., valor na posição x , $x+1$, $x+2$, etc.

70



Arrays (vectors)

A dimensão de um array define o número de elementos que suporta

Bem como o número de posições de memória que ocupa

Um array de dimensão n , contem valores nas posições 0 a $n-1$

71



Arrays (vectors)

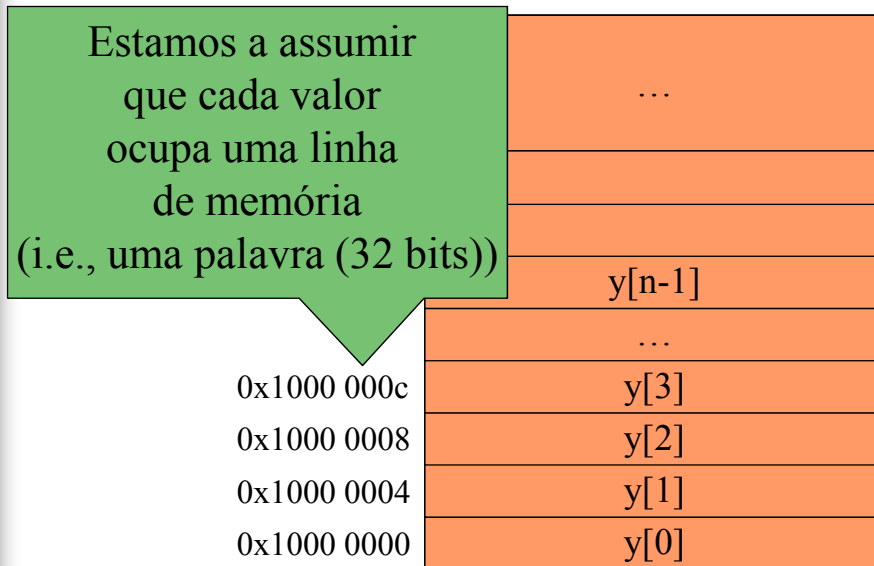
Ex: Array y de dimensão n

	...
	$y[n-1]$
	...
0x1000 000c	$y[3]$
0x1000 0008	$y[2]$
0x1000 0004	$y[1]$
0x1000 0000	$y[0]$

72

Arrays (vectores)

Ex: Array y de dimensão n



73

Arrays em Assembly

```
.data  
array_t: .word 10,12,7,5,9
```

74

Arrays em Assembly

Declara um array
com 5 inteiros de 32 bits

```
.data  
array_t: .word 10,12,7,5,9
```

75

Arrays em Assembly

Declara um array
com 5 inteiros de 32 bits

```
.data  
array_t: .word 10,12,7,5,9  
  
array_x: .space 40
```

76

Arrays em Assembly

Declara um array
com 5 inteiros de 32 bits

```
.data  
array_t: .word 10,12,7,5,9  
  
array_x: .space 40
```

Declara um array
vazio com 40 bytes

77

Arrays em Assembly

```
.data  
array_t: .word 10,12,7,5,9  
array_x: .space 40  
  
.text  
main:  
    la $t0, array_t
```

78

Arrays em Assembly

```
.data
array_t: .word 10,12,7,5,9
array_x: .space 40
.text
main:
```

Carrega para o registo \$t0
o endereço de memória
do array t

la \$t0, array_t

79

Arrays em Assembly

```
.data
array_t: .word 10,12,7,5,9
array_x: .space 40
.text
main:
```

Carrega para o registo \$t0
o endereço de memória
do array t

la \$t0, array_t

Endereço onde
este começa
(endereço base)

80

Arrays em Assembly

Como se acede aos elementos do array?

Carrega para o registo \$t0 o endereço de memória do array t

Endereço onde este começa (endereço base)

```
array_x: .space 10
        .text
main:
        la $t0, array_t
```

81

Arrays em Assembly

```
        .data
array_t: .word 10,12,7,5,9
array_x: .space 40
        .text
main:
        la $t0, array_t
        lw $s0, 0($t0)
```

82

Arrays em Assembly

```
.data
array_t: .word 10,12,7,5,9
array_x: .space 40
.text
main:
    la $t0, array_t
    lw $s0, 0($t0)
```

Endereçamento indirecto

83

Arrays em Assembly

```
.data
array_t: .word 10,12,7,5,9
array_x: .space 40
.text
main:
    la $t0, array_t
    lw $s0, 0($t0)
```

O valor contido em \$t0
é um endereço de memória

84

Arrays em Assembly

O objectivo é carregar,
para \$s0, o conteúdo
da posição cujo
endereço está em \$t0

```
.data
array_t: .word 10,12,7,5,9
array_x: .space 40
.text
main:
    la $t0, array_t
    lw $s0, 0($t0)
```

85

Arrays em Assembly

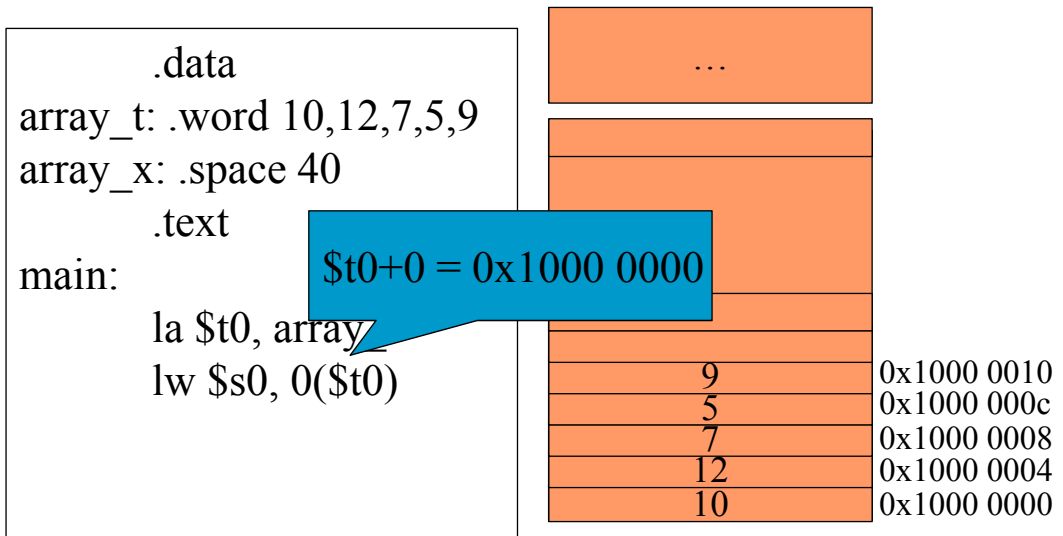
```
.data
array_t: .word 10,12,7,5,9
array_x: .space 40
.text
main:
    la $t0, array_t
```

\$t0 = 0x1000 0000

...	
9	0x1000 0010
5	0x1000 000c
7	0x1000 0008
12	0x1000 0004
10	0x1000 0000

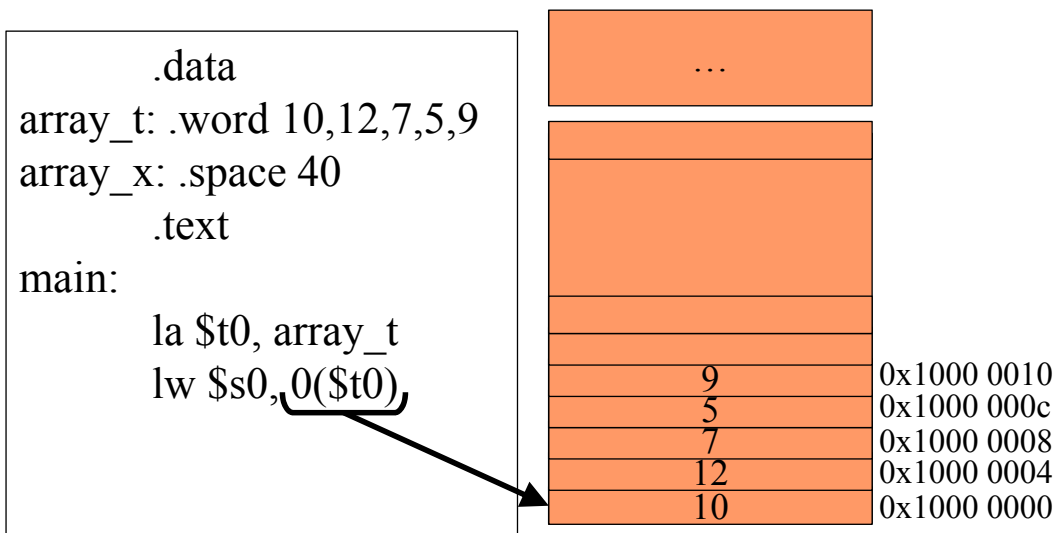
86

Arrays em Assembly



87

Arrays em Assembly



88



Arrays em Assembly

Carrega para $\$s0$ a **palavra**
contida na **posição** de
memória **cujo endereço é**
 $\$t0+0$
i.e, 1ª posição do array

```
.data
array_t: .word 10,12,7,5,9
array_x: .space 40
.text
main:
    la $t0, array_t
    lw $s0, 0($t0)
```

89



Arrays em Assembly

```
.data
array_t: .word 10,12,7,5,9
array_x: .space 40
.text
main:
    la $t0, array_t
    lw $s0, 0($t0)
    lw $s1, 4($t0)
```

90

Arrays em Assembly

```
.data
array_t: .word 10,12,7,5,9
array_x: .space 40
.text
main:
    la $t0, array_t
    lw $s0, 0($t0)
    lw $s1, 4($t0)
```

Carrega a palavra
contida no endereço
\$t0+4
i.e, 2ª posição do array

91

Arrays em Assembly

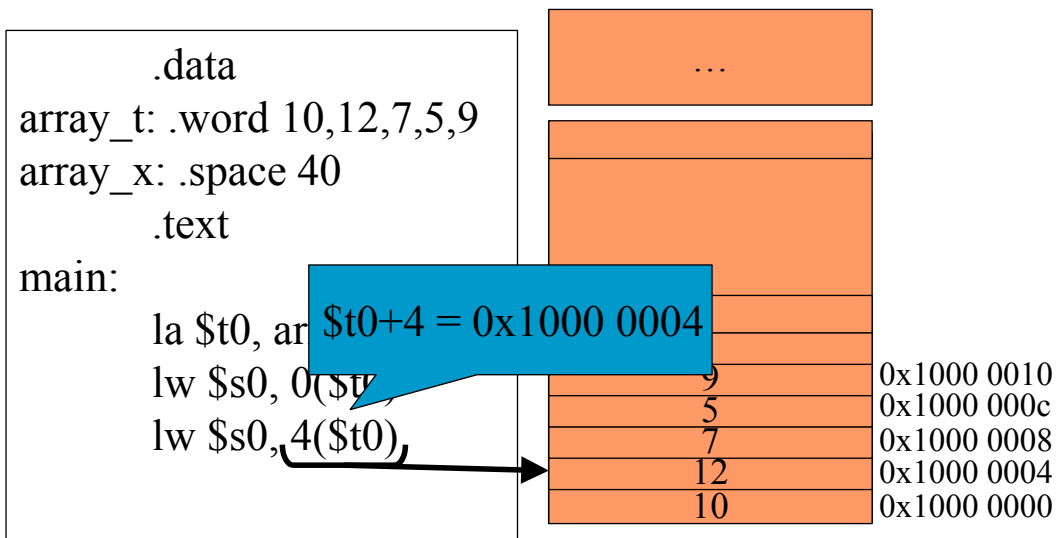
```
.data
array_t: .word 10,12,7,5,9
array_x: .space 40
.text
main:
    la $t0, array_t
    lw $s0, 0($t0)
    lw $s1, 4($t0)
```

Porquê 4?

Carrega a palavra
contida no endereço
\$t0+4
i.e, 2ª posição do array

92

Arrays em Assembly



93

Arrays em Assembly

```
.data
array_t: .word 10,12,7,5,9
array_x: .space 40
.text
main:
    la $t0, array_t
    lw $s0, ($t0)
    addi $t0, $t0, 4
    lw $s1, ($t0)
```

Alternativa: avançar
o endereço da posição
de memória...

94

Arrays em Assembly

```
.data
array_t: .word 10,12,7,5,9
array_x: .space 40
.text
main:
    la $t0, array_t
    lw $s0, ($t0)
    addi $t0, $t0, 4
    lw $s1, ($t0)
```

... e “reutilizar” o novo endereço

95

Arrays em Assembly

```
.data
t: .word 10,12,7,5,9
x: .space 20
.text
main:
    la $s3, t
    la $s4, x
    li $s0, 4
    addi $t1, $s3, 16
init:
    lw $t0, ($s3)
    sw $t0, ($s4)
    add $s3, $s3, $s0
    add $s4, $s4, $s0
    bgt $s3, $t1, end
    j init
end:
```

Que faz este programa?

96

Arrays em Assembly

```
.data
t: .word 10,12,7,5,9
x: .space 20
.text
main:
    la $s3, t
    la $s4, x
    li $s0, 4
    addi $t1, $s3, 16
init: lw $t0, ($s3)
      sw $t0, ($s4)
      add $s3, $s3, $s0
      add $s4, $s4, $s0
      bgt $s3, $t1, end
      j init
end:
```

Declara dois arrays, *x* e *t*.
t tem 5 valores e *x* tem 20 bytes

97

Arrays em Assembly

```
.data
t: .word 10,12,7,5,9
x: .space 20
.text
main:
    la $s3, t
    la $s4, x
    li $s0, 4
    addi $t1, $s3, 16
init: lw $t0, ($s3)
      sw $t0, ($s4)
      add $s3, $s3, $s0
      add $s4, $s4, $s0
      bgt $s3, $t1, end
      j init
end:
```

Declara dois arrays, *x* e *t*.
t tem 5 valores e *x* tem 20 bytes

Qual a dimensão de cada array?

98

Arrays em Assembly

```
.data
t: .word 10,12,7,5,9
x: .space 20
.text
main:
    la $s3, t
    la $s4, x
    li $s0, 4
    addi $t1, $s3, 16
init:
    lw $t0, ($s3)
    sw $t0, ($s4)
    add $s3, $s3, $s0
    add $s4, $s4, $s0
    bgt $s3, $t1, end
    j init
end:
```

Carrega para \$s3 e \$s4
os **endereços** de *t* e *x*, e
para \$s0 o **valor** 4

99

Arrays em Assembly

```
.data
t: .word 10,12,7,5,9
x: .space 20
.text
main:
    la $s3, t
    la $s4, x
    li $s0, 4
    addi $t1, $s3, 16
init:
    lw $t0, ($s3)
    sw $t0, ($s4)
    add $s3, $s3, $s0
    add $s4, $s4, $s0
    bgt $s3, $t1, end
    j init
end:
```

Guarda em \$t1 o resultado
da soma do **endereço**
guardado em \$s3 com 16

100

Arrays em Assembly

```
.data
t: .word 10,12,7,5,9
x: .space 20
.text
main:
    la $s3, t
    la $s4, x
    li $s0, 4
    addi $t1, $s3, 16
init:
    lw $t0, ($s3)
    sw $t0, ($s4)
    add $s3, $s3, $s0
    add $s4, $s4, $s0
    bgt $s3, $t1, end
    j init
end:
```

Guarda em \$t1 o resultado da soma do **endereço** guardado em \$s3 com 16

Qual o significado desse valor?

101

Arrays em Assembly

```
.data
t: .word 10,12,7,5,9
x: .space 20
.text
main:
    la $s3, t
    la $s4, x
    li $s0, 4
    addi $t1, $s3, 16
init:
    lw $t0, ($s3)
    sw $t0, ($s4)
    add $s3, $s3, $s0
    add $s4, $s4, $s0
    bgt $s3, $t1, end
    j init
end:
```

Carrega para \$t0, o valor apontado por \$s3

102

Arrays em Assembly

```
.data
t: .word 10,12,7,5,9
x: .space 20
.text
main:
    la $s3, t
    la $s4, x
    li $s0, 4
    addi $t1, $s3, 16
init:
    lw $t0, ($s3)
    sw $t0, ($s4)
    add $s3, $s3, $s0
    add $s4, $s4, $s0
    bgt $s3, $t1, end
    j init
end:
```

Guarda no endereço
apontado por \$s4, o valor
de \$t0

103

Arrays em Assembly

```
.data
t: .word 10,12,7,5,9
x: .space 20
.text
main:
    la $s3, t
    la $s4, x
    li $s0, 4
    addi $t1, $s3, 16
init:
    lw $t0, ($s3)
    sw $t0, ($s4)
    add $s3, $s3, $s0
    add $s4, $s4, $s0
    bgt $s3, $t1, end
    j init
end:
```

Guarda no endereço
apontado por \$s4, o valor
de \$t0

Qual o valor na posição de
memória apontada por \$s4?

104

Arrays em Assembly

```
.data
t: .word 10,12,7,5,9
x: .space 20
.text
main:
    la $s3, t
    la $s4, x
    li $s0, 4
    addi $t1, $s3, 16
init:
    lw $t0, ($s3)
    sw $t0, ($s4)
    add $s3, $s3, $s0
    add $s4, $s4, $s0
    bgt $s3, $t1, end
    j init
end:
```

Adiciona aos endereços em
\$s3 e \$s4 o valor de \$s0

105

Arrays em Assembly

```
.data
t: .word 10,12,7,5,9
x: .space 20
.text
main:
    la $s3, t
    la $s4, x
    li $s0, 4
    addi $t1, $s3, 16
init:
    lw $t0, ($s3)
    sw $t0, ($s4)
    add $s3, $s3, $s0
    add $s4, $s4, $s0
    bgt $s3, $t1, end
    j init
end:
```

Qual o novo valor
de \$s3 e \$s4?

Adiciona aos endereços em
\$s3 e \$s4 o valor de \$s0

106

Arrays em Assembly

```
.data
t: .word 10,12,7,5,9
x: .space 20
.text
main:
    la $s3, t
    la $s4, x
    li $s0, 4
    addi $t1, $s3, 16
init:
    lw $t0, ($s3)
    sw $t0, ($s4)
    add $s3, $s3, $s0
    add $s4, $s4, $s0
    bgt $s3, $t1, end
    j init
end:
```

Para onde apontam
agora \$s3 e \$s4?

Adiciona aos endereços em
\$s3 e \$s4 o valor de \$s0

107

Arrays em Assembly

```
.data
t: .word 10,12,7,5,9
x: .space 20
.text
main:
    la $s3, t
    la $s4, x
    li $s0, 4
    addi $t1, $s3, 16
init:
    lw $t0, ($s3)
    sw $t0, ($s4)
    add $s3, $s3, $s0
    add $s4, $s4, $s0
    bgt $s3, $t1, end
    j init
end:
```

Salta?

Se o valor em \$s3 for
maior que o valor em
\$t1 salta para *end*

108

Arrays em Assembly

```
.data
t: .word 10,12,7,5,9
x: .space 20
.text
main:
    la $s3, t
    la $s4, x
    li $s0, 4
    addi $t1, $s3, 16
init: lw $t0, ($s3)
      sw $t0, ($s4)
      add $s3, $s3, $s0
      add $s4, $s4, $s0
      bgt $s3, $t1, end
      j init
end:
```

Salta para *init* e repete novamente...

109

Arrays em Assembly

```
.data
t: .word 10,12,7,5,9
x: .space 20
.text
main:
    la $s3, t
    la $s4, x
    li $s0, 4
    addi $t1, $s3, 16
init: lw $t0, ($s3)
      sw $t0, ($s4)
      add $s3, $s3, $s0
      add $s4, $s4, $s0
      bgt $s3, $t1, end
      j init
end:
```

Repete quantas vezes?

Salta para *init* e repete novamente...

110



Arrays

Exercício:

Escreva um programa que, dado um array x com 4 valores inteiros de 32 bits, escreva num novo array y os sucessores de cada um dos elementos de x .