Conjuntos de Instruções: Lidando com a Memória e Operações Lógicas

Yuri Kaszubowski Lopes

UDESC

Anotações

Revisão: Palavra de Dados e Instruções no MIPS

- Palavra de Dados (word):
  - Tamanho: 32 bits
- Instruções:
  - Tamanho: **Todas 32 bits**
  - Tipos (até o momento vimos):
    - Tipo-R
    - Tipo-I

- Ainda temos o tipo-J (aulas futuras)

Anotações

Revisão: Registradores do MIPS

Número (Decimal)	Nome Registrador	Descrição
0	$zero,$r0	Sempre contém zero
1	$at	Utilizado para o assembler (montador)
2 e 3	$v0 e $v1	Valores de retorno
4,...,7	$a0,...$a3	Argumentos de função
8,..,15	$t0,...,$t7	Para cálculos temporários (não salvos)
16,...,23	$s0,...,$s7	Registradores salvos (entre chamadas de função)
24 e 25	$t8 e $t9	Mais registradores temporários
26 e 27	$k0 e $k1	Reservados para o Kernel (S.O.)
28	$gp	Apontador de memória global
29	$sp	Ponteiro de pilha
30	$fp	Ponteiro de quadro
31	$ra	Endereço de retorno

Anotações

Comentários e Números no MIPS

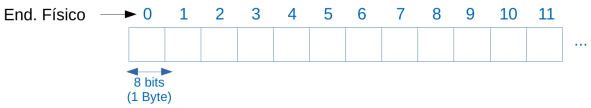
- # Inicia um comentário (resto da linha)
  - Comente bem seu código
  - Sugestão dois espaços antes e um depois do #
- No código abaixo `main` e `end` são rótulos (labels)
  - Podemos utilizar os rótulos ao invéz de endereços de memória

```
1     .text
2     .globl main
3
4 main:
5     li $t0, 011  # Número decimal 9 em octal
6     li $t1, 22  # Número decimal 22
7     li $t2, 0xFF  # Número decial 255 em hexadecimal
8 end:
9     li $v0, 10  # Código para encerrar o programa
10    syscall  # encerra o programa
```


Anotações

Acessando a memória

- A memória principal é um “vetor ou array”, onde cada posição possui um endereço físico
- As memórias são comumente endereçadas byte a byte
- Cada byte possui um endereço físico, e cada endereço suporta 1 byte



Anotações

Acessando a memória

- Considere o seguinte exemplo em C:

```
1 int x = 0x0F; //em C 0x indica um valor em hexadecimal
2 int v[2] = {0x01, 0x00FFAA1D};
```

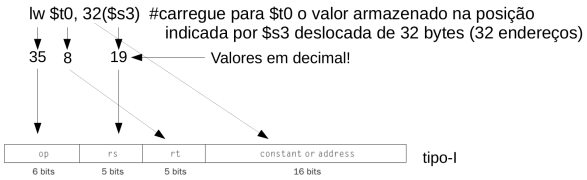
- Um vetor é algo que inicia em uma posição de memória, e cada nova posição do vetor é um deslocamento da posição inicial
- Podemos representar na memória da seguinte forma
  - Considerando big-endian, e que inteiros ocupam 32 bits
- x está no endereço 0, e ocupa 4 posições
- v começa no endereço 4 (ponteiro com endereço base aponta para 4)
- v[0] é o mesmo que v deslocado 0 endereços de 1 byte (4+0)
- v[1] é o mesmo que v deslocado 4 endereços de 1 byte (4+4)
- Deslocamos 4, pois cada inteiro ocupa 4 bytes no exemplo
  - Os deslocamentos mudariam dependendo do tipo da variável
  - e.g., 1 byte para char

Anotações

Acessando a memória

- Operamos com os valores somente nos registradores
- Sempre precisamos carregar/armazenar na memória principal
- Utilizamos **loads** e **stores**
- Instruções do **tipo-I**
- `lw $regDestino, deslocamento($regBase)`
  - `$regDestino = MEM[$regBase + deslocamento]`
- Deslocamento é um **imediato**
- `sw $regFonte, deslocamento($regBase)`
  - `MEM[$regBase + deslocamento] = $regFonte`

```
1 lw $t0, 32($s3)
```



Anotações

Acessando a memória

```
1     .data
2 vector:
3     .word 3  # v[0] = 3
4     .word 5  # v[1] = 5
5
6     .text
7     .globl main
8 main:
9     la $t0, vector
10    lw $t1, 4($t0)  # $t1 = v[1]
11    addi $t1, $t1, 6  # $t1 += 6
12    sw $t1, 8($t0)  # v[2] = $t1
13 end:
14    li $v0, 10  # Código para encerrar o programa
15    syscall  # encerra o programa
```

- Nas Linhas 1-4 estamos estabelecendo valores iniciais no espaço de dados
- `la` carrega um endereço (não os dados)

Anotações

Exercício

- Considere o trecho de código em C abaixo, traduza para assembly do MIPS

```
1 int v[4] = {3, 7, 5, 1};
2 v[2] = 87 + v[3];
```

solução:

```
1     .data
2 vector:
3     .word 3  # v[0] = 3
4     .word 7  # v[1] = 7
5     .word 5  # v[2] = 5
6     .word 1  # v[3] = 1
7
8     .text
9     .globl main
10 main:
11    la $t0, vector
12    lw $t1, 12($t0)  # $t1 = v[3]
13    addi $t1, $t1, 87  # $t1 += 87
14    sw $t1, 8($t0)  # v[2] = $t1
15 end:
16    li $v0, 10  # Código para encerrar o programa
17    syscall  # encerra o programa
```


Anotações

Operações Lógicas: Deslocamentos (shifts)

- Operações lógicas bit a bit
  - Instruções do **tipo-R**
- Deslocamentos (shifts)
  - `sll`: shift left logical (deslocamento lógico à esquerda)
    - Desloca os bits da palavra para esquerda, preenchendo as lacunas geradas com zeros
    - `sll $RegDestino, $RegFonte, deslocamento`

```
1    sll $t2, $s0, 4
```

```
0000 0000 0000 0000 0000 0000 0000 1001
0000 0000 0000 0000 0000 0000 1001 0000
```

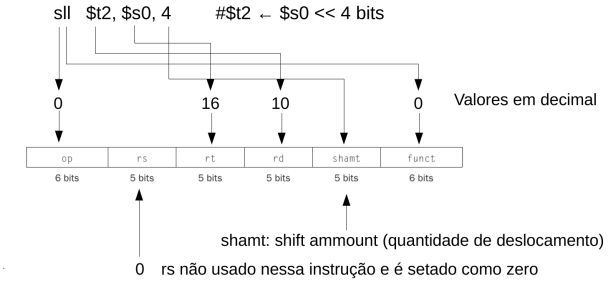
  - `srl`: shift right logical (deslocamento lógico à direita)
    - Desloca os bits da palavra para direita, preenchendo as lacunas geradas com zeros
    - `srl $RegDestino, $RegFonte, deslocamento`

```
1    srl $t2, $s0, 2
```

```
0000 0000 0000 0000 0000 1111 0000 1011
0000 0000 0000 0000 0000 0011 1100 0010
```


Anotações

Operações Lógicas: Deslocamentos (shifts)



- Qual a utilidade dos shifts?
  - Dentre outros usos, ao realizarmos um shift de n bits, estamos efetivamente multiplicando o valor por 2^n
  - Lidar com potências de 2 na máquina é muito comum
  - A unidade aritmética que faz shifts é muito simples e rápida
    - Mais rápido do que se realizássemos uma multiplicação por 2 “clássica”

Anotações

Operações Lógicas: AND e OR

- AND
  - `and`: realiza o AND lógico entre os bits dos registradores

```
1 and $t0, $t1, $t2  # $t0 = $t1 & $t2
```

```
$t1: 0000 0000 0000 0000 0000 0000 1101 0001
$t2: 0000 0000 0000 0000 0000 0000 1100 0000
$t0: 0000 0000 0000 0000 0000 0000 1100 0000
```

- OR
  - `or`: realiza o OR lógico entre os bits dos registradores

```
1 or $t0, $t1, $t2  # $t0 = $t1 | $t2
```

```
$t1: 0000 0000 0000 0000 0000 0000 1101 0001
$t2: 0000 0000 0000 0000 0000 0000 1100 0000
$t0: 0000 0000 0000 0000 0000 0000 1101 0001
```


Anotações

Operações Lógicas: NOR

- A última operação lógica que deveria existir é um not
  - Essa operação tomaria um registrador fonte e um destino
  - Não segue o padrão do tipo-R
  - Para manter o padrão, a operação NOR foi incluída no MIPS
- NOR
  - `nor`: realiza o OR-negado lógico entre os bits dos registradores

```
1 nor $t0, $t1, $t2  # $t0 = ~($t1 | $t2)
```

```
$t1: 0000 0000 0000 0000 0000 0000 1101 0001
$t2: 0000 0000 0000 0000 0000 0000 1100 0000
$t0: 1111 1111 1111 1111 1111 1111 0010 1110
```

- Como fazer um **not** com `nor`?

```
1   nor $t0, $t1, $zero  # $t0 = ~$t1
```


Anotações

AND e OR com imediatos

- As instruções `and` e `or` possuem versões imediatas

```
1   ori $t0, $t1, 0xACDC  # $t0 = $t1 | 0xACDC
2   andi $t0, $t1, 0xACDC  # $t0 = $t1 & 0xACDC
```

- É comum utilizar o ori para carregar um imediato para dentro de um registrador
- Como podemos carregar o imediato 156_{10} para o registrador `$s0` utilizando `ori`?

```
1   ori $s0, $zero, 156  # $s0 = 0 | 156 <==> $s0 = 156
```

- Poderíamos fazer a carga utilizando um `addi`?
  - Problema: o `addi` copia o bit mais alto do imediato (16-bits) para o bit mais alto do registrador (32-bits) se o somarmos com zero
  - Complemento de dois
  - Não é um problema para constantes pequenas ($\leq 0x7FFF$)
  - `addiu` pode ser uma opção válida
  - O montador pode ajudar :)

Anotações

ANDI

- Com:

```
1   addi $t0, $zero, 0x7FFF
```

- Montador gera:

```
1   addi $8, $0, 0x7FFF
```

- Com:

```
1   addi $t0, $zero, 0x8000
```

- Montador do MARS gera:

```
1   lui $1, 0x0000
2   ori $1, $1, 0x8000
3   add $8, $0, $1
```

- Dependedo do contexto e do Montador `addi` pode ser uma pseudoinstrução

Anotações

Carregando Imediatos

- `addi`
- `addiu`
- `ori`
  - Qual o maior imediato que podemos carregar com `ori`?

  - Um imediato de 16 bits, que é o tamanho do campo constant
  - E como carregamos um imediato de 32 bits?
    - Os registradores comportam 32 bits
    - Podemos ter uma **única** instrução que carrega imediatos de 32 bits?
    - Toda instrução no MIPS tem 32 bits
    - Se criarmos uma instrução para carregar esse imediato, todos os bits da instrução seriam utilizados para definir o imediato ⇒ impossível!

Anotações

Carregando Imediatos

- `addi`
- `addiu`
- `ori`
- `lui`: load upper immediate
  - Carrega o imediato para os 16 bits **mais significativos** do registrador, e **preenche o restante com zero**

    lui $t0, 255 #$255_{10}$ é o mesmo que $0000\ 0000\ 1111\ 1111_2$

    $t0 terá então

    $0000\ 0000\ 1111\ 1111\ 0000\ 0000\ 0000\ 0000_2$

    Copia para os 16 bits mais altos | Zero nos 16 bits mais baixos

Anotações

Carregando Imediatos

- `addi`, `addiu`, `ori`, `lui`
- `li`: uma **pseudoinstrução** presente nos montadores MIPS
  - Facilidade oferecida pelo montador
  - Carrega um imediato de 32 bits
  - Com:

```
1    li $t0, 0x7FFF
```

  - Montador gera:

```
1    addiu $8, $0, 0x7FFF
```

  - Com:

```
1    li $t0, 0x8000
```

  - Montador gera:

```
1    ori $8, $0, 0x8000
```

  - Com:

```
1    li $t0, 0x100AA
```

  - Montador gera:

```
1    lui $1, $0, 0x1
2    ori $8, $1, 0xaa
```


Anotações

Exercícios

1. Carregue os seguintes imediatos para o registrador `$t0`. Não utilize a pseudoinstrução **li**.
   1. 255_{10}
   2. 987342343_{10}
   3. -987342343_{10}
2. Qual o código de máquina das instruções do exercício 1? Mostre isso em binário "encaixando" os bits em suas posições corretas em instruções do tipo R ou I, dependendo das suas respostas no exercício
3. Considere as variáveis `a`, `b`, `c` e `d` de um programa, que foram carregadas para os registradores `$s0`, `$s1`, `$s2` e `$s3`, respectivamente. Como fica o seguinte código em assembly do MIPS? Considere que `x` deve ser salvo no registrador `$s4`.

```
1     x = a + b + c - d - 747;
```

4. Faça um programa em MIPS equivalente ao código C abaixo:

```
1 int v[8] = {12, 3, 10, 7, 5, 1, 0, 99};
2 v[7] = v[1] + v[2] + v[3] + 42;
```


Anotações

Sumário de algumas instruções e pseudoinstruções

Carregar/Salvar	Aritiméticas	Lógicas	Outras
`la`	`add`	`and`	`mfhi`
`lb`	`addi`	`andi`	`mflo`
`lbu`	`addiu`	`or`	`syscall`
`li`	`addu`	`ori`	`move`
`lui`	`sub`	`nor`	
`lw`	`subi`	`not`	
`sb`	`subiu`	`sll`	
`sw`	`subu`	`srl`	
	`mul`		
	`mult`		
	`div`		
	`divu`		

Anotações

Conjuntos de Instruções: Lidando com a Memória e Operações Lógicas

Yuri Kaszubowski Lopes

UDESC

Anotações