“Um mouse é um dispositivo que contém um, dois, ou três botões, dependendo da estimativa que os projetistas dão para a capacidade intelectual de seus usuários” (Tanenbaum, Bos; Sistemas Operacionais Modernos; 2016).

Conjuntos de Instruções

Lidando com a Memória e Operações Lógicas

Paulo Ricardo Lisboa de Almeida

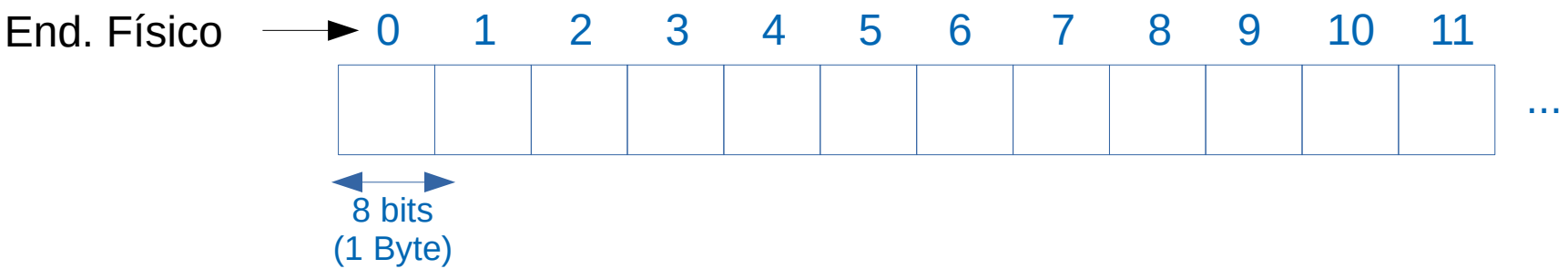
UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

Antes de continuarmos

- # isso é um comentário em Assembly do MIPS
  - **Sempre inclua o máximo possível de comentários em seus programas**
- 123456 # isso é uma constante em **decimal**
- 0xAABBCCDD #isso é uma constante em **hexadecimal**
- 0b1100110011 #isso é uma constante em **binário**

Acessando a memória

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



Acessando a memória

- Considere o seguinte exemplo em C

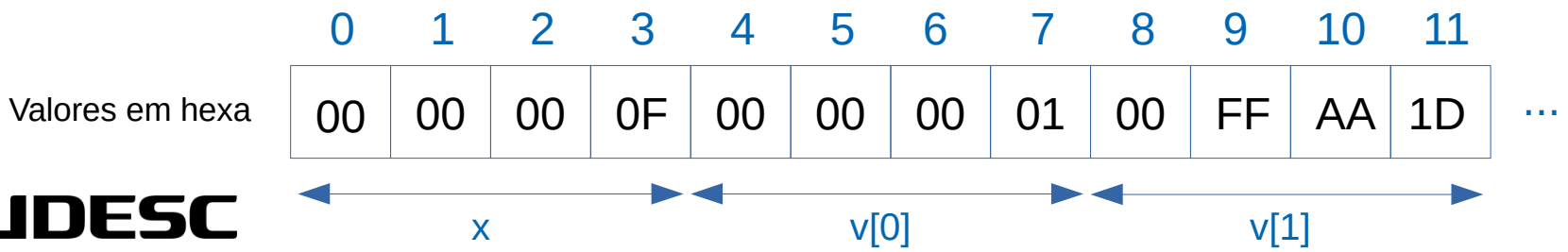
```
int x = 0x0F; //em C 0x indica um valor em hexadecimal
int v[2] = {0x01,0x00FFAA1D};
```

Acessando a memória

- Considere o seguinte exemplo em C

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

- Lembre-se que 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
    - Assumindo big-endian, e que inteiros ocupam 32 bits

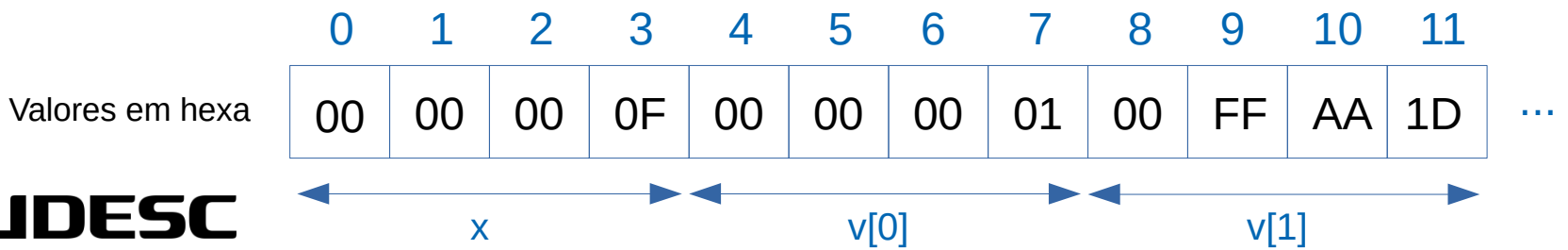


Acessando a memória

- Considere o seguinte exemplo em C

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

- x está no endereço 0, e ocupa 4 posições
- v começa no endereço 4 (Seu 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
    - Exemplo: Deslocamentos de 1 byte para chars

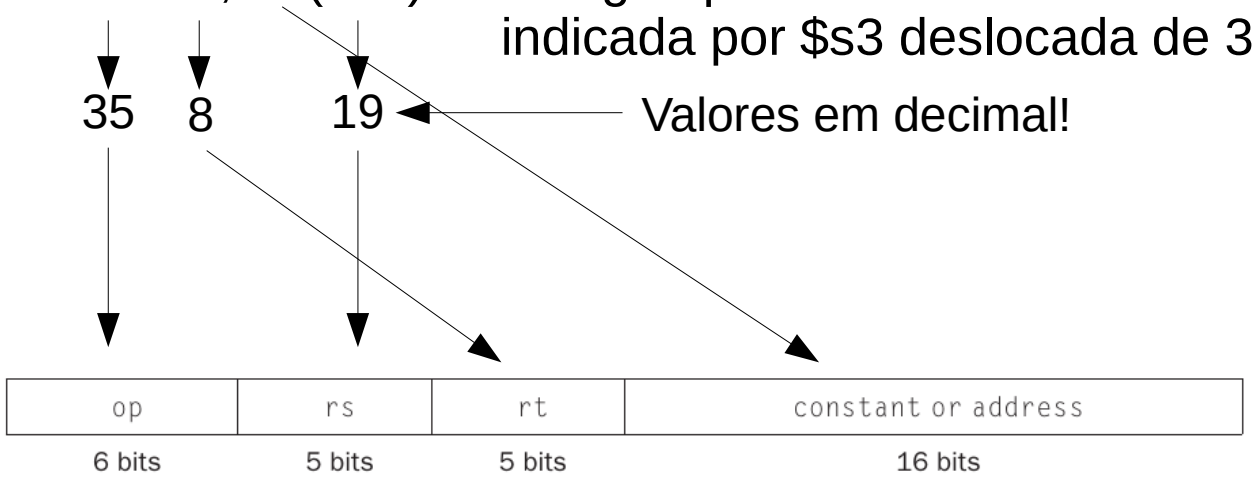


Acessando a memória

- 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) #load word (carregar palavra)
    - $regDestino = MEM[$regBase + deslocamento]
  - Deslocamento é um **imediato**
    - **Pode ser positivo ou negativo!**
  - sw $regFonte, deslocamento($regBase) #store word (armazenar palavra)
    - MEM[$regBase + deslocamento] = $regFonte

Acessando a memória

- Exemplo
  - lw $t0, 32($s3) #carregue para $t0 o valor armazenado na posição indicada por $s3 deslocada de 32 bytes (32 endereços)



tipo-I

- A ideia do sw é similar!

Exercício

- Considere o seguinte array em C

  ```
  int a[15];
  ```

- Suponha que o endereço base de **a** está no registrador $s3, e que um inteiro ocupa 4 bytes.

- Traduza a seguinte instrução C para assembly do MIPS

  ```
  a[12] = 87+a[8];
  ```

Exercício

- Considere o seguinte array em C

  int a[15];

- Suponha que o endereço base de **a** está no registrador $s3, e que um inteiro ocupa 4 bytes.

- Traduza a seguinte instrução C para assembly do MIPS

  a[12] = 87+a[8];

- Resposta

```
lw $t0,32($s3) #registrador temporário $t0 recebe a[8]
addi $t0, $t0, 87 #$t0 = $t0 + 87
sw $t0,48($s3) #armazena $t0 em a[12]
```

Operações Lógicas

- Operações lógicas bit a bit
  - Instruções do **tipo-R**
- Shifts (deslocamentos)
  - sll → shift left logical (deslocamento lógico à esquerda)
  - Desloca os bits da palavra para esquerda, preenchendo as lacunas geradas com zeros
  - Exemplo:

    $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_2$

    Quando deslocado 4 bits à esquerda se torna

    $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_2$
  - Formato

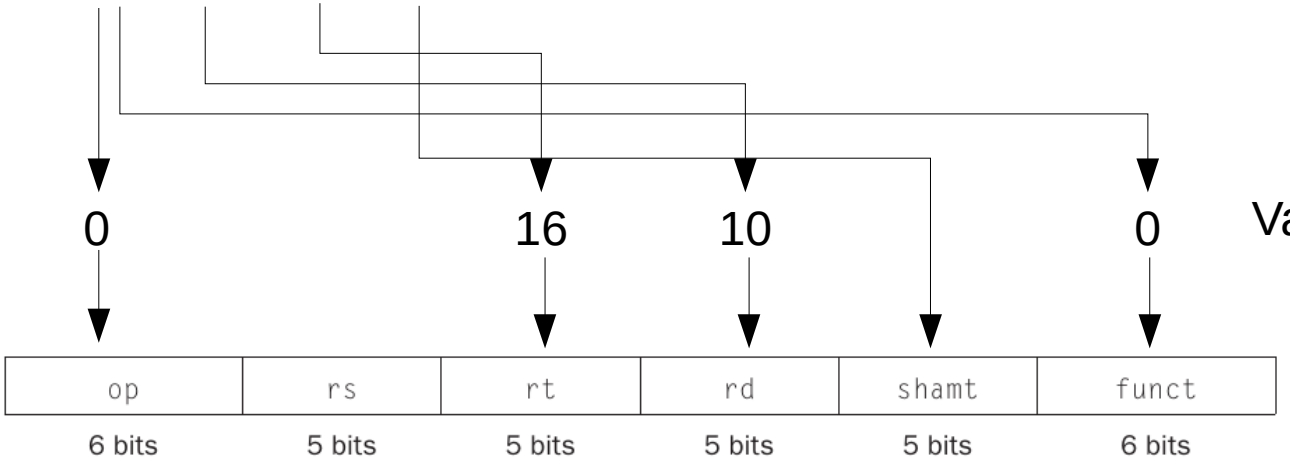
    sll $RegDestino, $RegFonte, deslocamento

    #$RegDestino ← $RegFonte deslocado “deslocamento” bits à esquerda
  - Dual de sll é o srl
  - srl → shift right logical (deslocamento lógico à direita)
    - Tem o mesmo formato do sll, mas desloca na direção oposta (para direita) ao sll

Operações Lógicas

- Exemplo concreto

sll $t2, $s0, 4 #$t2 ← $s0 << 4 bits



Valores em decimal

shamt: shift ammount (quantidade de deslocamento)

0 rs não usado nessa instrução e é setado como zero

UDESC

Shifts

- Qual a utilidade dos shifts?

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”

AND

- and → realiza o AND lógico entre os bits dos registradores
- Considere o seguinte
  - $t1 contém
    0000 0000 0000 0000 0000 0000 1101 0001_2
  - $t2 contém
    0000 0000 0000 0000 0000 0000 1100 0000_2
  -
- A operação
  and $t0, $t1, $t2 #$t0 ← $t1 AND $t2
- Resulta no seguinte em $t0
  - 0000 0000 0000 0000 0000 0000 1100 0000_2

OR

- or → realiza o OR lógico entre os bits dos registradores
- Considere o seguinte
  - $t1 contém
    0000 0000 0000 0000 0000 0000 1101 0001_2
  - $t2 contém
    0000 0000 0000 0000 0000 0000 1100 0000_2
- A operação
  or $t0, $t1, $t2 #$t0 ← $t1 OR $t2
- Resulta no seguinte em $t0
  - 0000 0000 0000 0000 0000 0000 1101 0001_2

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
- Exemplo

  nor $t0,$t1,$t2 #$t0 ← $t1 nor $t2
- Como podemos utilizar um nor como um not?

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
- Exemplo

  nor $t0,$t1,$t2 #$t0 ← $t1 nor $t2
- Como podemos utilizar um nor como um not?
  - Por exemplo, inverter $t1 e armazenar em $t0
  - nor $t0,$t1,$zero #$t0 ← not $t1
  - De acordo com a álgebra de Boole: $\overline{(A + 0)} = \overline{A}$

AND e OR Imediatos

- As instruções **and** e **or** possuem versões imediatas
  - **andi** e **ori**
- Exemplo

  ori $s0, $s1, 0xACDC
- É comum utilizarmos o ori para carregarmos um imediato para dentro de um registrador
- Exemplo
  - Como podemos carregar o imediato 156_{10} para o registrador $s0 utilizando ori?

Carregando Imediatos

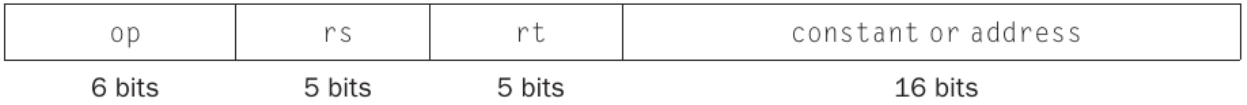
- Como podemos carregar o imediato 156_{10} para o registrador $s0 utilizando ori?

  ori $s0,$zero,156

- Poderíamos carga utilizando um addi
  - Problema: o addi copia o bit mais alto do imediato para os 16 primeiros bits do registradores se o somarmos com zero
    - Complemento a dois
  - Veja a documentação
  - Isso não é um problema para constantes pequenas, mas nos próximos slides veremos que essa solução não vai funcionar para constantes grandes
- Obs.: addiu pode ser uma opção válida

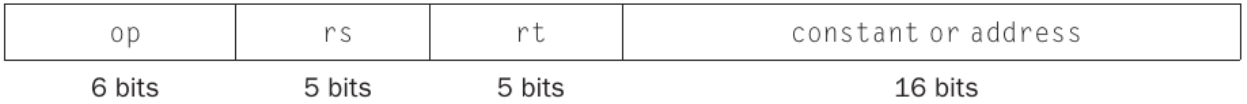
Carregando Imediatos

- Qual o maior imediato que podemos carregar com ori?



Carregando Imediatos

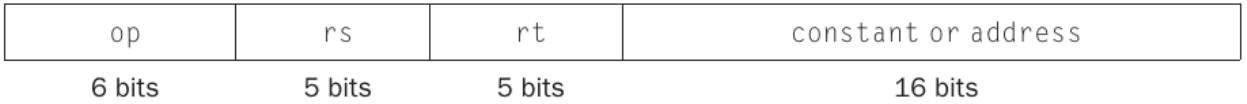
- Qual o maior imediato que podemos carregar com ori?



- 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 instrução que carrega imediatos de 32 bits?**

Carregando Imediatos

- 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 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

Carregando Imediatos

- lui → load upper immediate
  - Carrega o imediato para os 16 bits **mais significativos** do registrador, e **preenche o restante com zero**
- Exemplo

  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

Exercício

- Utilizando lui e ori, carregue o imediato 1048992_{10} para $t0

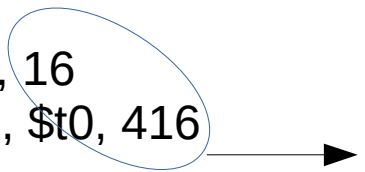
Exercício

- Utilizando lui e ori, carregue o imediato 1048992_{10} para $t0
  - 1048992_{10} é o mesmo que $001001A0_{16}$, então

```
lui $t0, 0x0010
ori $t0, $t0, 0x01A0
```

  - Poderíamos fazer em decimal, mas as contas se tornam tediosas e confusas

```
lui $t0, 16
ori $t0, $t0, 416
```



Pare de brigar com a máquina e fale a língua dela. A programação em baixo nível se tornará mais simples!

Equivalências em linguagens de alto nível

Operação	C	Java	MIPS
Shift à esquerda	<<	<<	sll
Shift à direita	>>	>>>	srl
and bit a bit	&	&	and, andi
or bit a bit	|	|	or, ori
not bit a bit	~	~	nor (utilizando $zero)

li

- li é uma **pseudoinstrução** presente nos montadores MIPS
  - Facilidade oferecida pelo montador
  - Carrega um imediato de 32 bits
- Formato

  li $s0, imediato # $s0 ← imediato
- Ao transformar para linguagem de máquina, o montador transforma automaticamente o li em um liu e ori
- Para isso ele precisa de um registrador temporário que garantidamente não está sendo utilizado
- Esse é um dos usos do registrador $at, que deve ser reservado para o montador
  - Se você colocar um valor em $at, é possível que o montador o apague

Exercícios

1. Carregue os seguintes imediatos para o registrador $t0. Não utilize a pseudoinstrução li.
   a) 255_{10}
   b) 987342343_{10}
   c) -987342343_{10} ← Utilize complemento de dois

2. 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 a seguinte instrução em assembly do MIPS? Considere que x deve ser salvo no registrador $s4.
   x = a + b + c - d - 747;

3. Considere um vetor de inteiros vet que começa no endereço $100080AA_{16}$. Realize a seguinte operação em assembly do MIPS (considere que inteiros ocupam 4 bytes):
   vet[7] = vet[1]+vet[2]+vet[3]+65

Referências

- D. Patterson; J. Henessy. **Organização e Projeto de Computadores: A Interface Hardware/Software**. 5a Edição. Elsevier Brasil, 2017.
- Andrew S. Tanenbaum. **Organização estruturada de computadores**. 5. ed. São Paulo: Pearson, 2007.
- Ronald Tocci, Neal Widmer, Greg Moss. **Digital Systems**. 12 ed. Pearson Education. 2016.
- James Bignell, Robert Donovan. **Eletrônica digital**. Cengage Do Brasil, 2010.
- MELO, M. **Eletrônica Digital**. Makron Books.2003.