# Arquitetura de Computadores I

**Prof. Ricardo Santos**

ricr.santos@gmail.com

**(Cap 2)**

# Fluxo de controle

- **Vimos até agora: beq, bne**
- **Uma nova instrução:**

```
                  if  $s1 < $s2 then
                     $t0 = 1
slt $t0, $s1, $s2      else
                     $t0 = 0
```

- **slt = set if less than=configurar se for menor que**
- **Podemos usar essa instrução para construir outras mais poderosas como: "`blt $s1, $s2, Label`"**
  - **— assim, podemos construir instruções mais completas**
- **Note que o montador (assembler) necessita de um registrador extra para executar essa instrução blt,**
  - **— existe uma "política" de convenção de uso de registradores**

# Formatos e opcodes das instruções

| Mnemonico | Formato | Opcode | Function |
|---|---|---|---|
| Add | R | 0 | 32 |
| Sub | R | 0 | 43 |
| Lw | I | 35 | - |
| Sw | I | 43 | - |
| And | R | 0 | 36 |
| Or | R | 0 | 37 |
| Nor | R | 0 | 38 |
| Andi | I | 12 | - |
| Ori | I | 13 | - |
| Sll | R | 0 | 0 |
| Srl | R | 0 | 2 |
| Beq | I | 4 | - |
| Bne | I | 5 | - |
| Slt | R | 0 | 42 |
| J | J | 2 | - |
| Jr | R | 0 | 8 |
| Jal | J | 3 | - |

# Nomenclatura dos registradores no processador MIPS

| Name | Register number | Usage |
|---|---|---|
| `$zero` | 0 | the constant value 0 |
| `$v0-$v1` | 2-3 | values for results and expression evaluation |
| `$a0-$a3` | 4-7 | arguments |
| `$t0-$t7` | 8-15 | temporaries |
| `$s0-$s7` | 16-23 | saved |
| `$t8-$t9` | 24-25 | more temporaries |
| `$gp` | 28 | global pointer |
| `$sp` | 29 | stack pointer |
| `$fp` | 30 | frame pointer |
| `$ra` | 31 | return address |

Register 1 ($at) é reservado para o montador, 26-27 são reservados para o SO

# Constantes

- **Constantes "pequenas" são usadas frequentemente (50% dos operandos)**

  **e.g., A = A + 5;**
  **B = B + 1;**
  **C = C - 18;**

- **Soluções**
  - **Colocar constantes na memória e carrega-las**
  - **Criar registradores hard-wired (como $zero) para algumas constantes.**

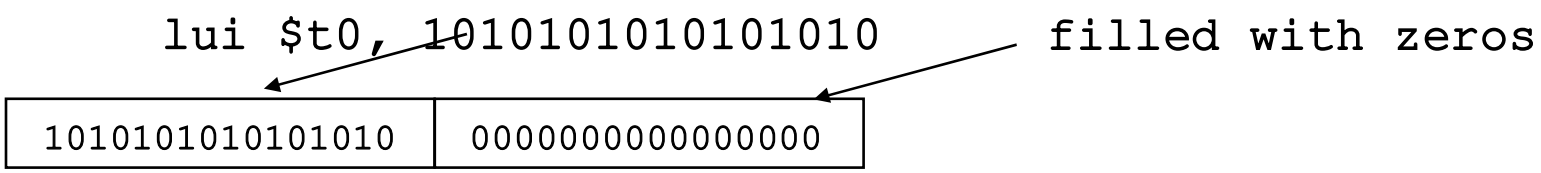
- **Instruções MIPS:**

```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```

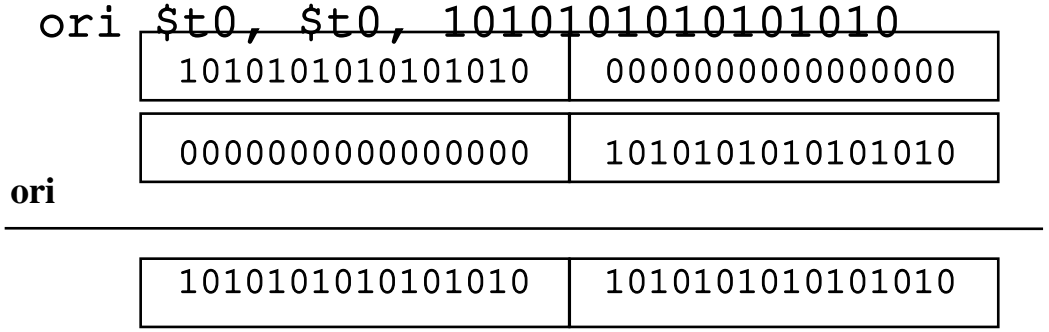
- **Princípio de projeto: Tornar o caso comum rápido**

# E quanto as constantes ‘grandes’?

- **Como é possível carregar uma constante de 32 bits em um registrador?**
- **Utilizaremos duas novas instruções:**
  - **Instrução lui="load upper immediate"**




- **Depois disso, obtemos os bits de “baixa” ordem da direita:**

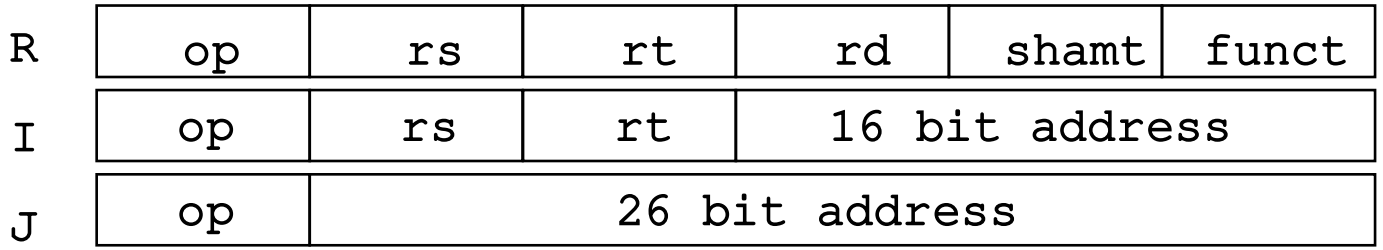


# Linguagem assembly vs. Linguagem de máquina

- Assembly fornece uma representação simbólica conveniente
  - Muito mais fácil que escrever os números (em binário ou hexa) diretamente
- Linguagem de máquina representa como os dados serão "entendidos" pela máquina
- Assembly permite a criação de 'pseudoinstruções'
  - e.g., "move $t0, $t1" existe apenas em assembly
  - Poderia também ser implementado usando "add $t0,$t1,$zero"
- Ao considerar o desempenho, procura utilizar instruções 'reais'

# Resumo do MIPS

- **Instruções simples e todas com 32 bits de largura**
- **Conjunto de instruções bem estruturado**
- **Apenas três formatos de instruções**




- **‘confia’ no compilador para alcançar melhor desempenho**

- **‘ajuda’ o compilador onde é possível**

# Entendendo Endereços em desvios e saltos

- **Instruções:**

  `bne $t4,$t5,Label` **Next instruction is at Label if** `$t4≠$t5`

  `beq $t4,$t5,Label` **Next instruction is at Label if** `$t4 = $t5`

  `j Label` **Next instruction is at Label**

- **Formatos:**


- **Endereços não são de 32 bits**
  **— Como nós manipulamos isso com instruções de load e store?**

# Entendendo endereços em desvios

- **Instruções:**

```
bne $t4,$t5,Label   Next instruction is at Label if $t4≠$t5
beq $t4,$t5,Label   Next instruction is at Label if $t4=$t5
```

- **Formato:**


- **Especificar um registrador (como em lw e sw) e adicionar o endereço (de desvio) para ele**
  - **Usar o registrador de endereço da instrução (PC = program counter)**
  - **Muitos desvios são locais (princípio da localidade)**
- **Instruções de jump (salto) utilizam apenas os bits mais significativos do PC**
  - **Faixa de endereços é de 256 MB**

# Resumindo...

**MIPS operands**

| Name | Example | Comments |
|---|---|---|
| 32 registers | `$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at` | Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants |
| $2^{30}$ memory words | Memory[0], Memory[4], Memory[4294967292] | Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls |

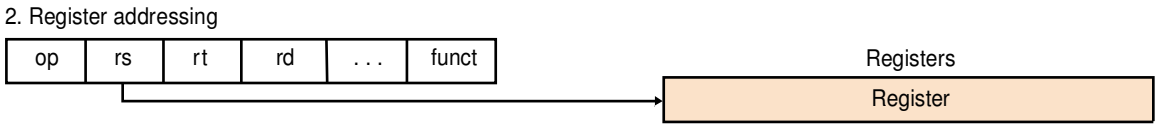
**MIPS assembly language**

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add | `add $s1, $s2, $s3` | `$s1 = $s2 + $s3` | Three operands; data in registers |
| | subtract | `sub $s1, $s2, $s3` | `$s1 = $s2 - $s3` | Three operands; data in registers |
| | add immediate | `addi $s1, $s2, 100` | `$s1 = $s2 + 100` | Used to add constants |
| Data transfer | load word | `lw $s1, 100($s2)` | `$s1 = Memory[$s2 + 100]` | Word from memory to register |
| | store word | `sw $s1, 100($s2)` | `Memory[$s2 + 100] = $s1` | Word from register to memory |
| | load byte | `lb $s1, 100($s2)` | `$s1 = Memory[$s2 + 100]` | Byte from memory to register |
| | store byte | `sb $s1, 100($s2)` | `Memory[$s2 + 100] = $s1` | Byte from register to memory |
| | load upper immediate | `lui $s1, 100` | $s1 = 100 * $2^{16}$ | Loads constant in upper 16 bits |
| Conditional branch | branch on equal | `beq $s1, $s2, 25` | if (`$s1 == $s2`) go to PC + 4 + 100 | Equal test; PC-relative branch |
| | branch on not equal | `bne $s1, $s2, 25` | if (`$s1 != $s2`) go to PC + 4 + 100 | Not equal test; PC-relative |
| | set on less than | `slt $s1, $s2, $s3` | if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0 | Compare less than; for beq, bne |
| | set less than immediate | `slti $s1, $s2, 100` | if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0 | Compare less than constant |
| Uncondi-tional jump | jump | `j 2500` | go to 10000 | Jump to target address |
| | jump register | `jr $ra` | go to `$ra` | For switch, procedure return |
| | jump and link | `jal 2500` | `$ra` = PC + 4; go to 10000 | For procedure call |

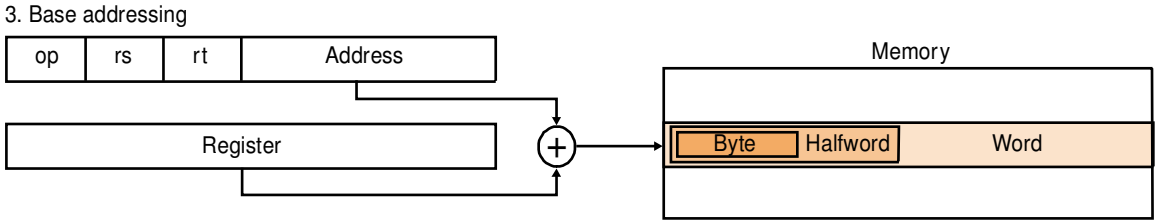
# Modos de endereçamento

Exemplos: addi, subu

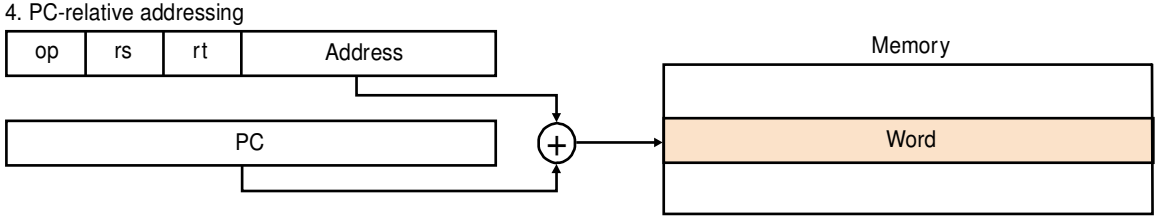

Exemplos: add, sub



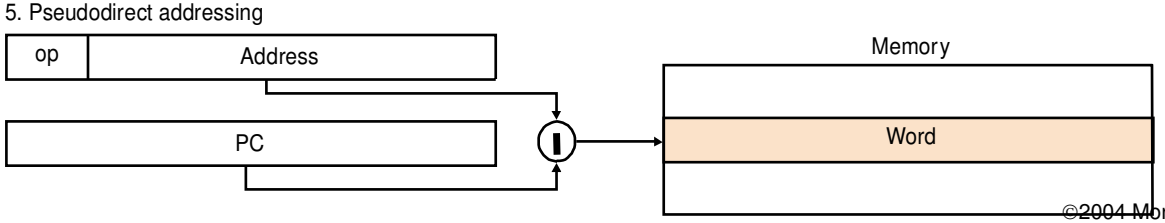

Exemplos: lw, sw




Exemplos: beq, bne




Exemplos: j, jal



# Exercícios

- **1) Traduza a instrução a seguir para o assembly MIPS utilizando apenas as instruções do processador (não utilize pseudoinstruções). Considere que os valores de A, B e C estão nas posições de memória a, b e c, respectivamente. Utilize a menor quantidade de registradores possível.**

```
if (A>=0)
  B=C/A;
else
  B=A+10
```

- **2) Utilizando as mesmas restrições do exercício 1), traduza o código a seguir. Considere que o endereço base do vetor V está em $s1.**

```
for (i=0; i<10; i++)
   V[i]=V[i]+1;
```

# Uma solução para o exercício 1

- **Uma solução para o exercício 1 utiliza apenas os registradores $s0, $s1 e $zero. Observe que os endereços de memória são carregados apenas quando os valores são necessários. Além disso, note que, ao final, o resultado é armazenado na memória (endereço de B).**

```
    lw $s1, a($zero) #carrega o valor de A em $s1
    slt $s0, $s1, $zero
    bne $s0, $zero, soma
    lw $s0, c($zero)
    div $s1, $s0, $s1
    j saida
soma:
    addi $s1, $s1, 10
saida:
    sw $s1, b($zero)
```

# Arquiteturas alternativas

- **Alternativa de projeto:**
  - **Operações mais robustas**
  - **Objetivo é reduzir o número de instruções executadas**
  - **O ‘perigo’ está em diminuir o tempo de ciclo e/ou aumentar a CPI**

- **Nos próximos slides, temos o conjunto de instruções IA-32**

# IA - 32

- 1978: Intel 8086 é anunciado (arquitetura de 16 bits)
- 1980: Co-processador de ponto-flutuante 8087 é adicionado
- 1982: 80286 aumenta o espaço de endereçamento para 24 bits
- 1985: O 80386 extende para 32 bits e adiciona novos modos de endereçamento
- 1989-1995: O 80486, Pentium, Pentium Pro adicionam algumas instruções (projetadas para alto desempenho)
- 1997: 57 novas instruções "MMX" são adicionadas
- 1999: O Pentium III adicionou outras 70 instruções (SSE)
- 2001: Outro conjunto de 144 instruções é lançado (SSE2)
- 2003: AMD extende a arquitetura para suportar espaço de endereço de 64 bits, aumenta todos os registradores para 64 bits (AMD64)
- 2004: Intel abrange todas as inovações do AMD64 (chama o projeto de EM64T) e adiciona mais instruções multimedia

- "This history illustrates the impact of the "golden handcuffs" of compatibility

  "adding new features as someone might add clothing to a packed bag"

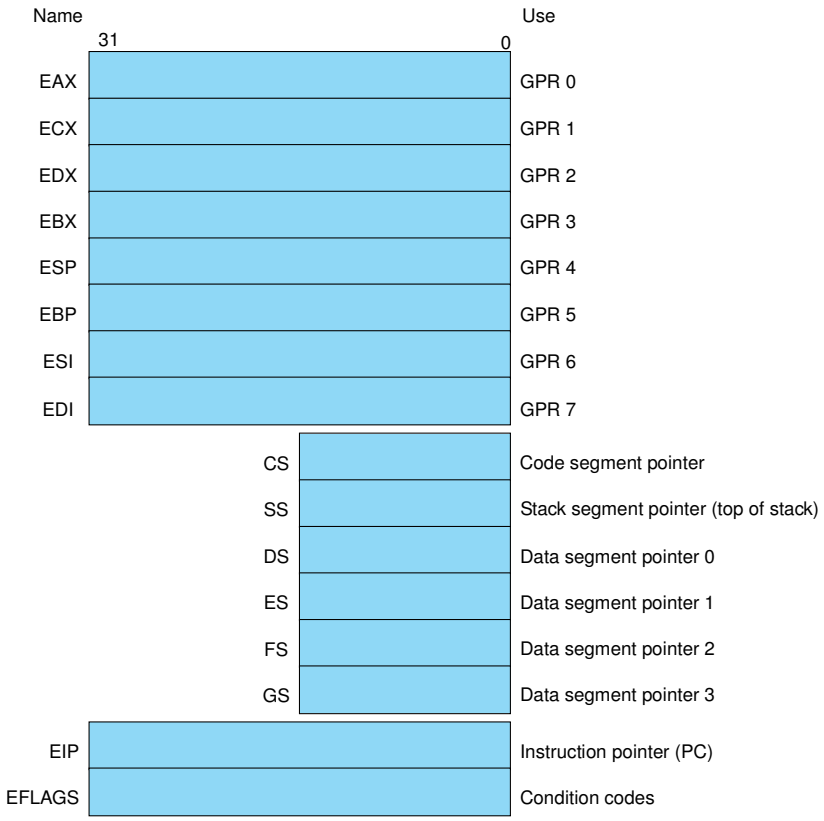
  "an architecture that is difficult to explain and impossible to love"

# IA-32 Overview

- **Complexidade:**
  - **Instruções são de 1 até 17 bytes de tamanho**
  - **Um operando funciona como fonte e destino**
  - **Um operando pode ser uma referência a memória**
  - **Modos de endereçamento complexos**
    - e.g., “índice ou base com deslocamento de 8 ou 32 bits”
  - **Instruções mais freqüentes não são difíceis de construir**
  - **Compiladores evitam as porções da arquitetura que são mais lentas**

# Registradores IA-32 e Endereçamento de Dados

- **Registradores de 32-bit que originaram com o 80386**



# Restrições aos Registradores IA-32

- **Nem todos os registradores são de propósito geral**

| Mode | Description | Register restrictions | MIPS equivalent |
|---|---|---|---|
| Register Indirect | Address is in a register. | not ESP or EBP | `lw $s0,0($s1)` |
| Based mode with 8- or 32-bit displacement | Address is contents of base register plus displacement. | not ESP or EBP | `lw $s0,100($s1) # ≤16-bit # displacement` |
| Base plus scaled index | The address is Base + ($2^{Scale}$ x Index) where Scale has the value 0, 1, 2, or 3. | Base: any GPR<br>Index: not ESP | `mul $t0,$s2,4`<br>`add $t0,$t0,$s1`<br>`lw $s0,0($t0)` |
| Base plus scaled index with 8- or 32-bit displacement | The address is Base + ($2^{Scale}$ x Index) + displacement where Scale has the value 0, 1, 2, or 3. | Base: any GPR<br>Index: not ESP | `mul $t0,$s2,4`<br>`add $t0,$t0,$s1`<br>`lw $s0,100($t0) # ≤16-bit # displacement` |

**FIGURE 2.42 IA-32 32-bit addressing modes with register restrictions and the equivalent MIPS code.** The Base plus Scaled Index addressing mode, not found in MIPS or the PowerPC, is included to avoid the multiplies by four (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.34 and 2.36). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. Scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register `$s1`. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

# Instruções IA-32

- **Quatro maiores tipos de instruções de inteiros**
  - **Movimento de dados incluindo move, push, pop**
  - **Aritméticas e lógicas (registrador de destino ou memória)**
  - **Fluxo de controle (uso de códigos de condições / flags )**
  - **Instruções de strings, incluindo comparação e transporte de strings**

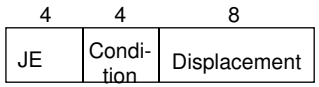
| Instruction | Function |
|---|---|
| JE name | if equal(condition code) {EIP=name};<br>EIP-128 ≤ name < EIP+128 |
| JMP name | EIP=name |
| CALL name | SP=SP-4; M[SP]=EIP+5; EIP=name; |
| MOVW EBX,[EDI+45] | EBX=M[EDI+45] |
| PUSH ESI | SP=SP-4; M[SP]=ESI |
| POP EDI | EDI=M[SP]; SP=SP+4 |
| ADD EAX,#6765 | EAX= EAX+6765 |
| TEST EDX,#42 | Set condition code (flags) with EDX and 42 |
| MOVSL | M[EDI]=M[ESI];<br>EDI=EDI+4; ESI=ESI+4 |

**FIGURE 2.43 Some typical IA-32 instructions and their functions.** A list of frequent operations appears in Figure 2.44. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

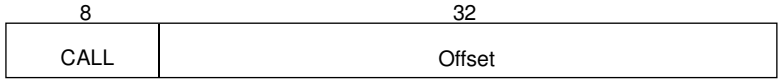
# Formatos de instruções IA-32

- **Formatos típicos: (note as diferenças de tamanhos)**

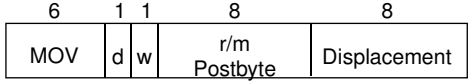
a. JE EIP + displacement



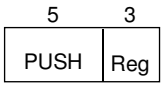

b. CALL



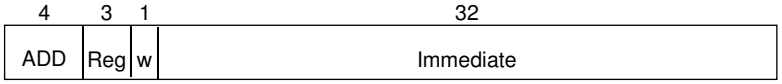

c. MOV EBX, [EDI + 45]



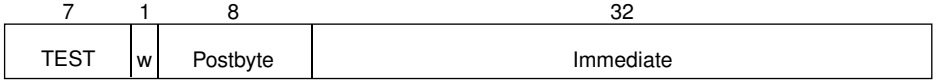

d. PUSH ESI




e. ADD EAX, #6765




f. TEST EDX, #42



# Resumo

- Complexidade das instruções é uma variável
  - Baixo número de instruções vs. maior CPI / baixa taxa de clock
- Principios de projeto:
  - Simplicidade favorece a regularidade
  - Menor é mais rápido
  - Bom projeto requer compromisso
  - Tornar o ‘caso-comum’ mais rápido possível
- Conjunto de Instruções da Arquitetura
  - De fato, uma abstração muito importante!