

Aula 3

- Instruções e classes de instruções
- Principios básicos de projecto de uma Arquitectura
- Aspectos chave da arquitectura do MIPS
- Instruções aritméticas
- Codificação de instruções no MIPS:
 - Instruções do tipo R

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira e Silva

Introdução: A máquina e a sua linguagem

Princípios básicos dos computadores actuais:

- As instruções são representadas da mesma forma que
- Os programas são armazenados em memória, ~~palavras~~ e escritos, tal como os números

Estes princípios formam os fundamentos do conceito de arquitectura **stored-program**

- O conceito **stored-program** pressupõe que na memória co-residam informações de natureza tão variada como o código fonte de um programa em C, um editor de texto, um compilador, e o próprio programa resultante da compilação

ISA: Instruções e classes de instruções

It is easy to see by formal-logical methods that there exist certain instruction sets that are in abstract adequate to control and cause the execution of any sequence of operations... The really decisive considerations from the present point of view, in selecting an instruction set, are more of a practical nature: simplicity of the equipment demanded by the instruction set, and the clarity of its application to the actually important problems together with the speed of its handling of those problems.

Burks, Goldstine and von Neumann, 1947

Instruções e classes de instruções

Será, portanto, possível considerar a existência de um grupo limitado de classes de instruções que possam ser usadas comuns à generalidade das arquiteturas?

There must certainly be instructions for performing the fundamental arithmetic operations

Burks, Goldstine and von Neumann, 1947

Quais serão então essas classes?

Classes de instruções

• Instruções de processamento (aritméticas e lógicas)

• Instruções de transferência de informação

• Instruções de controlo de fluxo de execução

• No projecto de um processador a definição de *instruction set* exige um delicado compromisso entre múltiplos aspectos, nomeadamente:

• as facilidades oferecidas aos programadores (por ex. instruções de manipulação de *strings*)

• a complexidade de *hardware* envolvido na sua implementação

• Quatro princípios básicos estão subjacentes ao *design* do *hardware*:

1. A regularidade favorece a simplicidade
2. Quanto mais pequeno mais rápido
3. O que é mais comum deve ser mais rápido
4. Um bom *design* implica compromissos adequados

1. A regularidade favorece a simplicidade

Ex1: Todas as instruções do *instruction set* são codificadas com o mesmo número de bits

Ex2: instruções aritméticas operam sobre registos internos e depositam o resultado também num registo interno

2. Quanto mais pequeno mais rápido

3. O que mais comum deve ser mais rápido

Ex: quando o operando é uma constante esta deve fazer parte da instrução (é vulgar que mais de 50% das instruções que envolvem a ALU num programa utilizem constantes)

4. Um bom *design* implica compromissos adequados

Ex: o compromisso que resulta entre a possibilidade de se poder codificar constantes de maior dimensão nas instruções e a manutenção da dimensão fixa nas instruções

Relembrando - A arquitectura do *set* de instruções (ISA) define:

Formato e codificação das instruções

como são decodificadas?

Operandos das instruções e resultados

onde podem residir?

quantos operandos explícitos?

como localizar?

quais podem residir na memória externa?

Tipo e dimensão dos dados

Operações

quais são suportadas?

Instruções auxiliares

jumps, conditions, branches

• Quanto ao formato e codificação das instruções

• Tamanho variável

• Código mais pequeno

• Maior flexibilidade

• *Instruction fetch* em vários passos

• Tamanho fixo

• *Instruction fetch* e *decode* mais simples

• Mais simples de implementar *pipeline*

• Quanto ao número de registos internos do CPU

• Vantagens de um número pequeno de registos

• Menos hardware

• Acesso mais rápido

• Menos bits para identificação do registo

• Mudança de contexto mais rápida

• Vantagens de um número elevado de registos

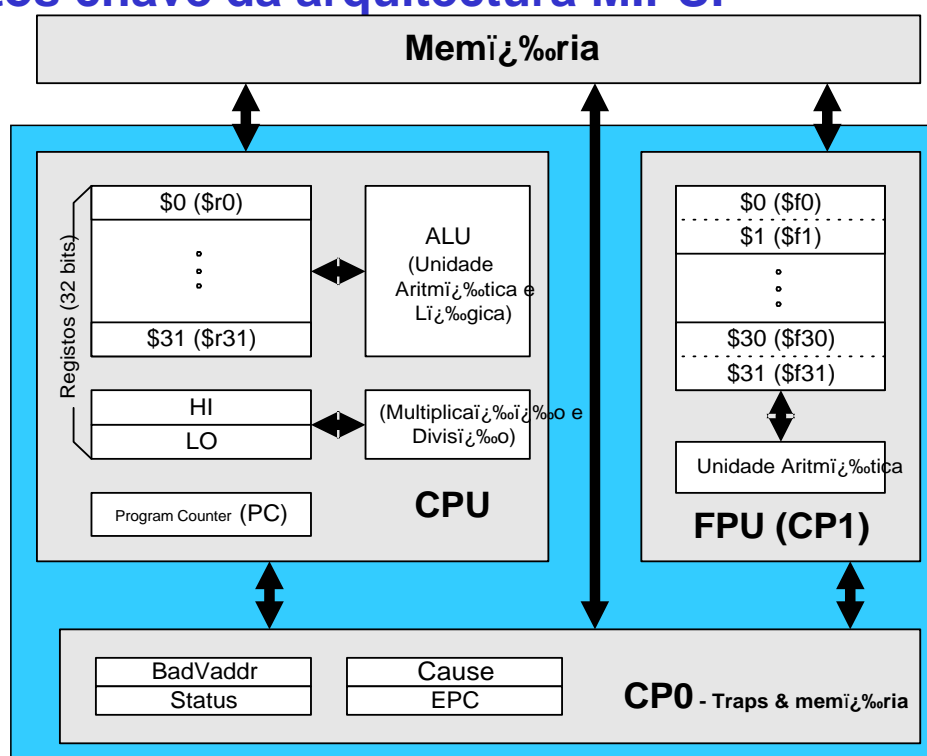
• Menos acessos à memória

• Vários registos

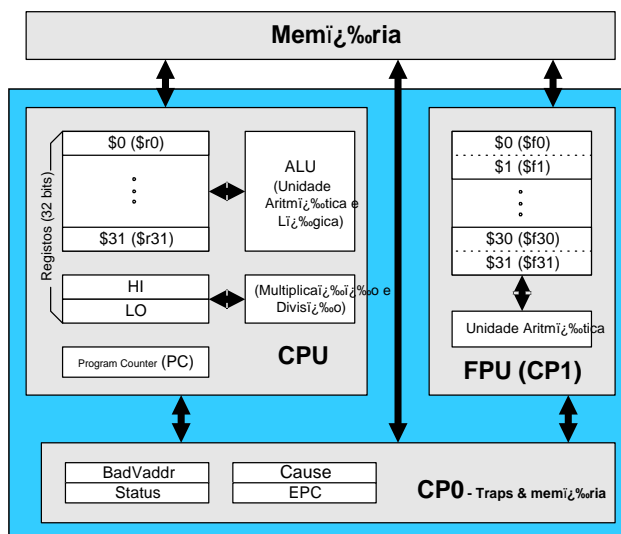
• Certos registos podem ter restrições de utilização

- Quanto localiza dos operandos das instruções
- Arquitecturas baseadas em **acumulador**
 - Resultado das operações é armazenado num registo es designado de acumulador
- Arquitecturas baseadas em **Stack**
 - Operandos e resultado armazenados numa **stack** (pilha) de registos
- Arquitecturas **Register-Memory**
 - Operandos residem em registos ou em memória
- Arquitecturas **Load-store**
 - Operandos residem em registos de uso geral (mas não na memória).

Aspectos chave da arquitectura MIPS:



Aspectos chave da arquitectura MIPS:



• 32 Registos de uso geral, de 32 bits cada (1 word = 32 bits)

• ISA baseado em instruções de dimensão fixa (32 bits)

• Memória organizada em bytes (memória byte addressable)

• Espaço de endereçamento de 32 bits (2^{32} endereços possíveis, i.e. máximo de 4 GB de memória)

• Barramento de dados externo de 32 bits

• Arquitectura **load-store** (register-register operation)

Instruções para efectuar operações aritméticas

Operações binárias: soma e subtracção

Soma:

Formato da instrução **Assembly** do Mips que efectua a operação de soma:

add a, b, c # Soma **b** com **c** e armazena o resultado em **a**

Exemplo: Uma adição do tipo $z = a + b + c + d$

tem de ser decomposta em:

```
add    z, a, b    # Soma a com b e armazena o resultado em z
add    z, z, c    # Soma z com c e armazena o resultado em z
add    z, z, d    # Soma z com d e armazena o resultado em z
```

Instruções para efectuar operações aritméticas

Subtração:

Formato da instrução Assembly do Mips que efectua a subtração:

sub a, b, c # Subtrai **c** a **b** e armazena o resultado em **a**

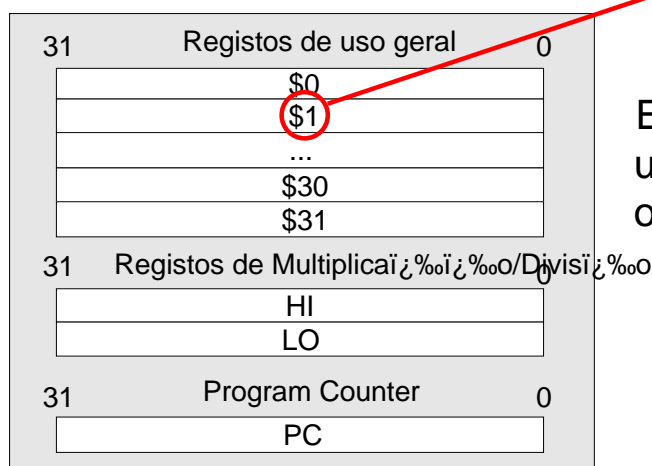
Exemplo: A operação $z = (a + b) - (c + d)$

tem de ser decomposta em:

```
add t1, a, b    # Soma a com b e armazena o resultado em t1
add t2, c, d    # Soma c com d e armazena o resultado em t2
sub  z, t1, t2   # Subtrai t2 a t1 e armazena o resultado em z
```

Os registos internos do MIPS:

Endereço do registo



Em *assembly* são, normalmente, usados nomes alternativos para os registos (nomes virtuais):

$\$zero$ (\$0)
 $\$v0 - \$v1$
 $\$a0 - \$a3$
 $\$t0 - \$t9$
 $\$s0 - \$s7$
 $\$sp$
 $\$ra$ (\$31)

Porquê 32 registos ?

Exemplo anterior revisitado ($z = (a + b) \% (c + d)$)

```
add    $8, $17, $18    # Soma $17 com $18 e armazena o resultado em $8
add    $9, $19, $20    # Soma $19 com $20 e armazena o resultado em $9
sub    $16, $8, $9      # Subtrai $9 a $8 e armazena o resultado em $16
```

O equivalente em C:

```
// a, b, c, d, z residem, respectivamente, em:
// $17, $18, $19, $20 e $16
// $8 e $9 representam variáveis temporárias utilizadas em C
```

```
int a, b, c, d, z;

z = (a + b) % (c + d);
```

Como comentar adequadamente um programa em *Assembly*?

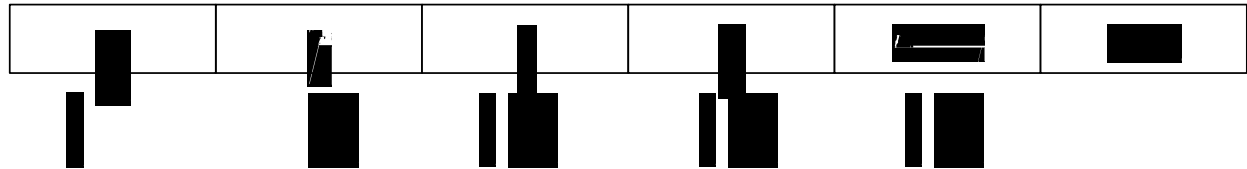
```
# a > $17, b > $18
# c > $19, d > $20, z > $16

%0
add    $8, $17, $18    #
add    $9, $19, $20    #
sub    $16, $8, $9      # z = (a + b) % (c + d);
%0
```

A linguagem C é uma ótima forma de comentar programas em *Assembly* uma vez que permite uma interpretação directa e mais simples do(s) algoritmo(s) implementado(s).

Codificação de instruções no MIPS

1. Instruções do Tipo R



Campos da instrução:

op: opcode (3 bits **nas instruções tipo R**)

rs: Endereço do registo que contém o 1º operando fonte

rt: Endereço do registo que contém o 2º operando fonte

rd: Endereço do registo onde o resultado vai ser armazenado

shamt: *shift amount* (util apenas em instruções de deslocamento)

funct: código da operação a realizar

Instruções lógicas (bit a bit *):

~~and~~ Rdst, Rsrc1, Rsrc2 # Rdst = Rsrc1 & Rsrc2;
~~or~~ Rdst, Rsrc1, Rsrc2 # Rdst = Rsrc1 | Rsrc2;
~~nor~~ Rdst, Rsrc1, Rsrc2 # Rdst = ~(Rsrc1 | Rsrc2);
~~xor~~ Rdst, Rsrc1, Rsrc2 # Rdst = (Rsrc1 ^ Rsrc2);
~~sl~~ Rdst, Rsrc, k # Rdst = Rsrc << k; // shift left logical
~~sr~~ Rdst, Rsrc, k # Rdst = Rsrc >> k; // shift right logical
~~sra~~ Rdst, Rsrc, k # Rdst = Rsrc >> k; // shift right arithmetic

Operadores lógicos bitwise em C:

& - AND

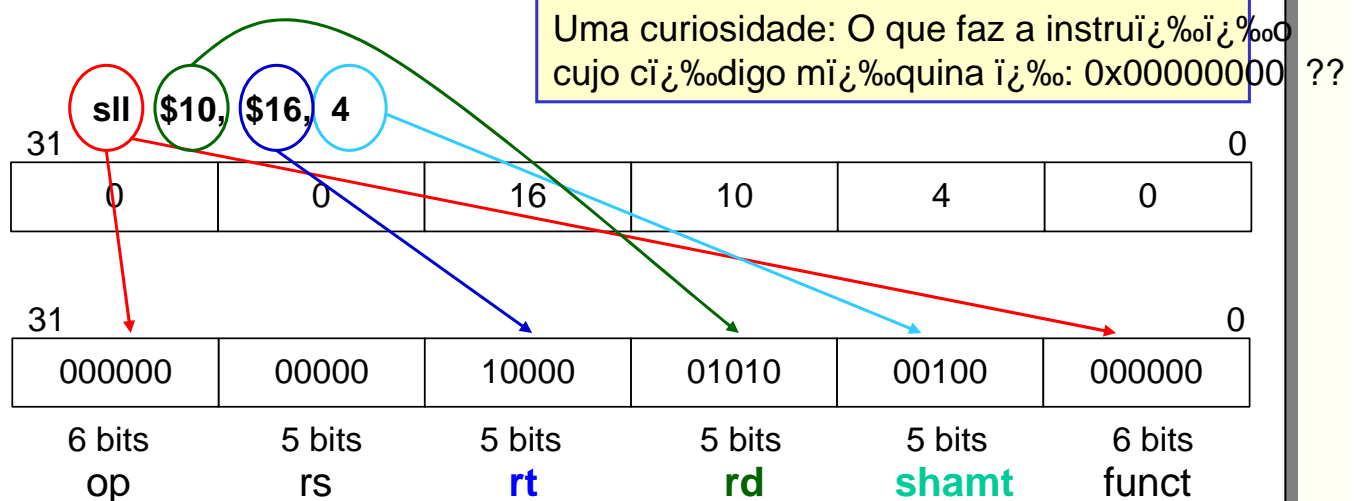
| - OR

^ - XOR

~ - NOT

*bit a bit *bitwise*

Codificação de instruções no MIPS (tipo R)



sll rd, rt, shamt

Código máquina

da instrução **00000000001000001010001000000000₂ = 0x00105100**

Instruções de transferência de informação entre registos internos

Transferência entre registos internos: $R_{dst} = R_{src}$

Registo \$0 do MIPS tem sempre o valor 0x00000000 (apenas pode ser lido)

Utilizando o registo \$0 e a instrução `OR` para realizar uma operação de transferência entre registos internos

`OR Rdst, $0, Rsrc` # $R_{dst} = 0 + R_{src} = R_{src}$

Exemplo: **`OR $t1, $0, $t2`** # $\$t1 = \$t2$

Para esta operação é habitualmente usada uma instrução **virtual** que melhora a legibilidade dos programas - "**move**". No processo de geração do código máquina, substitui essa instrução na instrução nativa anterior:

`move Rdst, Rsrc` # $R_{dst} = R_{src}$

Exemplo: **`move $t1, $t2`** # $\$t1 = \$t2$