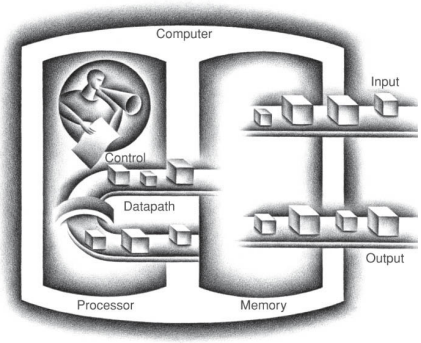
Arquitetura de Computadores I

2. Instruções executáveis por um processador: a arquitetura MIPS

António de Brito Ferrari
ferrari@ua.pt



7. O conjunto de instruções do processador

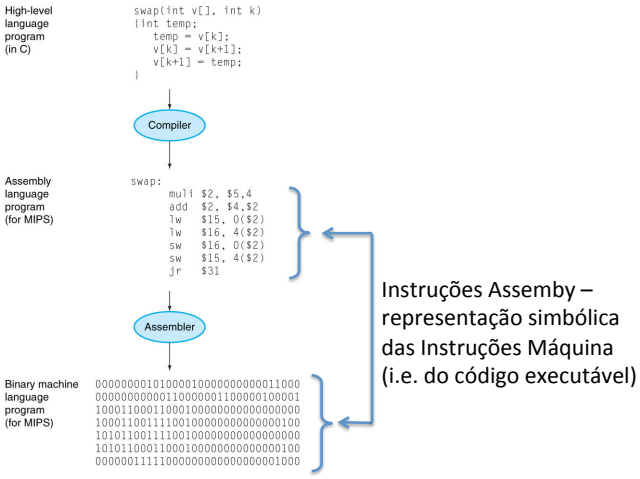
A máquina e a sua linguagem

- Princípios básicos dos computadores:
  - As instruções são representadas da mesma forma que os números
  - Os programas podem ser armazenados em memória, para serem lidos e escritos, tal como os números

    (***stored-program digital computer***)

- O conceito de *stored-program* implica que a memória contenha informações de natureza muito diversa:
  - o código fonte de um programa em C
  - um editor de texto
  - um compilador e o programa resultante da compilação
  - …

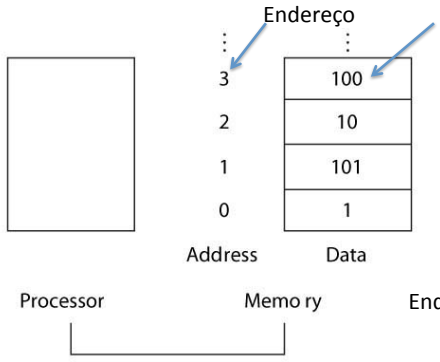
Das Linguagens de Alto Nível à linguagem do processador



Critérios de seleção de um Reportório de Instruções (**ISA**)

- *simplicidade do equipamento exigido para execução das instruções,*
- *clareza da sua aplicação aos problemas realmente importantes*
- *velocidade de resolução desses problemas*

(Burks, Goldstine e von Neumann, 1947)



Conteúdo da posição de memória 3 - M[3]

Memória: array de células que armazenam informação **(instruções e operandos)**

Espaço de Endereçamento – número de células de memória endereçáveis

Endereços de 32-bits: 2^{32} localizações endereçáveis

Duas operações possíveis:

1. **Escrita** – armazena informação na célula de memória indicada pelo endereço
   Mem[Address] = Data
2. **Leitura** – obtem a informação armazenada na célula de memória cujo endereço é fornecido
   Data = Mem[Address]

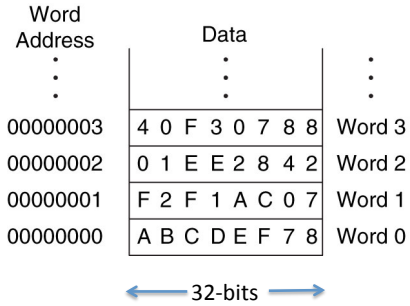
Data – quantos bits?

8 bits – ***byte***

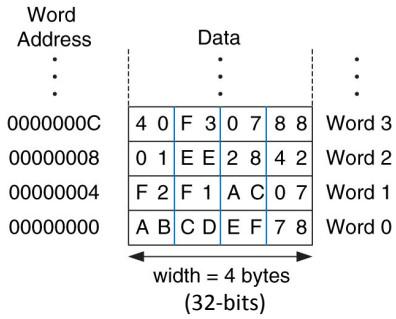
32 bits – ***word***

16 bits - ***half-word***

Word-addressable memory



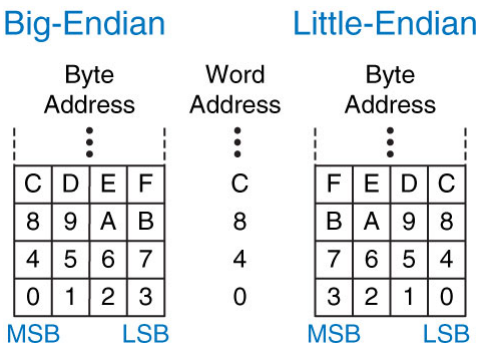
Byte-addressable memory



MIPS - *Byte-addressable memory*

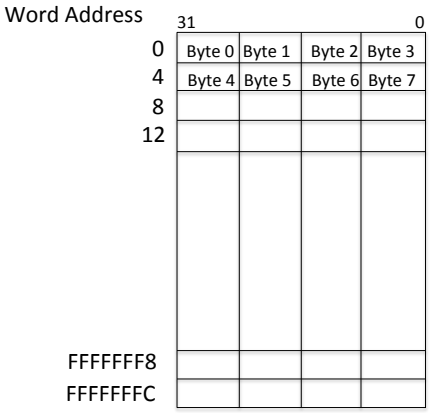
Big-endian vs. little-endian memory addressing

Qual o endereço dos bytes de uma palavra?
Duas possibilidades:



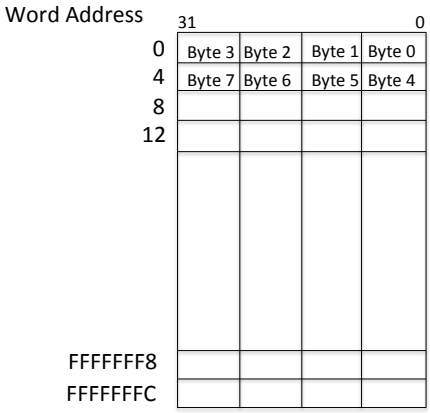
Organização da memória: Words e bytes

Big-endian



Endereço da palavra é o endereço do seu byte mais significativo (o que está no "*big end*")

Little-endian



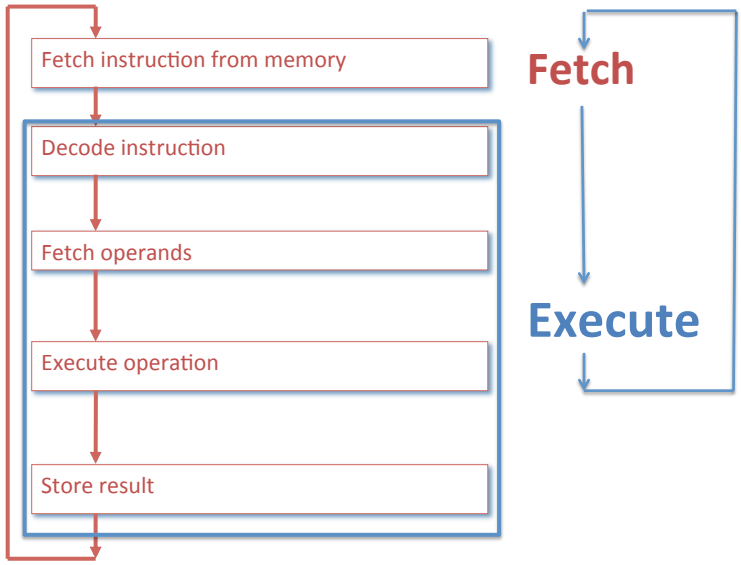
Endereço da palavra é o endereço do seu byte menos significativo (o que está no "*little end*")

Passos básicos de execução de um programa (***Fetch-Execute***)

1. **FETCH**: processador endereça a memória para obter o código da instrução seguinte a executar
   - **Program Counter (PC)** - registo onde o processador guarda o endereço da instrução seguinte a executar
   - **Instruction Register (IR)** – registo onde o processador guarda o código da instrução a executar

   IR = Mem[PC]
2. **EXECUTE:** processador executa a operação indicada no código de instrução

Processamento das Instruções



Execução de uma instrução

Onde ir buscar os operandos?

Duas alternativas:

a. À memória – de cada vez que uma operação aritmética ou lógica é executada é necessário ir buscar o operando à memória
b. Aos registos do processador – acesso mais rápido – operandos em registos

Onde armazenar o resultado da operação?

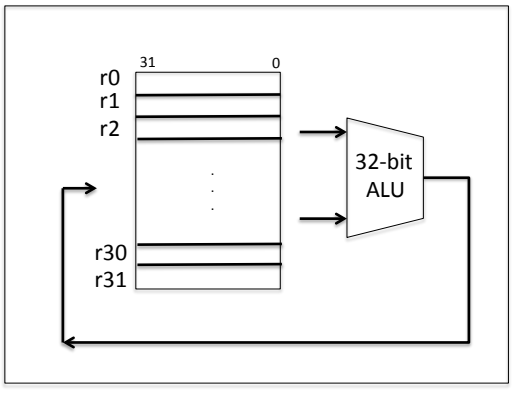
Duas alternativas:

a. Na memória
b. Num registo do processador

Tipos de Arquitetura

- **RISC** (**R**educed **I**nstruction **S**et **C**omputers) - **MIPS**, **ARM**, …
  - Operações aritméticas e lógicas só operam sobre informação armazenada nos registos do processador ou sobre constantes contidas no código de instrução (imediatos) e o resultado da operação é armazenado num registo
  - Apenas as operações de ***load*** (transferência do conteúdo de uma posição de memória para uma registo) e ***store*** (armazenamento em memória do conteúdo de um registo)comunicam com a memória de dados.
- **CISC** (**C**omplex **I**nstruction **S**et **C**omputers) - **Intel**, **IBM**, …
  - Operandos das operações aritméticas e lógicas podem estar na memória ou em registos

Processador **MIPS**: ***Datapath***



32 registos de 32 bits
32-bit ALU

Registos vs. Memória Acesso mais rápido
Menos bits para os endereçar: 5 bits vs. 32 bits

MIPS *ISA*
(Instruction Set Architecture)

1. Formato e codificação das instruções
   – Tamanho fixo (todas as instruções codificadas no mesmo número de bits) ou tamanho variável?
2. Tipo e dimensão dos dados
   – Inteiros representados em 32-bits (MIPS word)
   – Carateres (*byte*)
   – Reais (Standard IEEE para vírgula flutuante – 32 e 64 bits)
3. Que operações incluir no **IS** (*Instruction Set*)?

Tipos de instruções

- **Aritméticas e lógicas** (***add***, ***sub***, ***and***, ***or***, …)
  – Todas operam sobre o conteúdo de registos e armazenam o resultado num registo
- **Transferência de dados** de/para a memória
  - ir buscar à memória um dado transferindo-o para um registo
    load: ***lw***, ***lb***
  - armazenar na memória o conteúdo de um registo
    store: ***sw***, ***sb***
- **Instruções de controle** da sequência de execução das instruções do programa
  - *if-then, case, loops*: instruções de *branch*: ***beq***, ***bne***
  - *invocação de funções*: ***jal***, ***jr***

Localização dos operandos

MIPS e arquiteturas RISC em geral:

todas as instruções aritméticas e lógicas operam sobre o conteúdo de registos

i.e. são do tipo **Reg = Reg op Reg**

Arquitetura ***Load-Store*** - apenas estas instruções endereçam a memória

Outras arquiteturas:

as instruções aritméticas e lógicas podem ter operandos em memória, i.e. a arquitetura suporta operações do tipo

Reg = Reg op Mem; Mem = Reg op Reg; Mem = Reg op Mem;

Arquitetura ***Register-Memory***

Exemplos: Intel x86, IBM S/360, … arquiteturas **CISC** em geral

O Instruction Set do MIPS – instruções básicas

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, register $zero always equals 0, and register $at is reserved by 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 word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
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 on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional 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

Operações aritméticas

Todas as instruções aritméticas têm 2 operandos:

Exemplo: a = b + c; d = a - e

add a, b, c # a = b + c

sub d, a, e # d = a - e

- Sintaxe do *assembly*:
  - 1 instrução por linha
  - comentários são iniciados por **#** e terminam na linha

Compilação de instruções de atribuição (C ou Java)

C	**Assembly**

f = (g + h) – (i + j)	add t0, g, h
	add t1, i, j
	sub f, t0, t1

t0, t1 – variáveis temporárias

f, g, h, i, j, a, b, c, d, e – variáveis do programa

Valores das variáveis têm de estar em registos

Operandos em Registos

Convenções de uso e nomes dos registos em *assembly:*

- **`$s0,…,$s7`** (r16 a r23) – usados para armazenar (temporariamente) valores de variáveis do programa
- **`$t0,…,$t7`** (r8 a r15) - usados para armazenar (temporariamente) valores intermédios de cálculos

```
t0, t1 – variáveis temporárias
f, g, h, i, j – variáveis do programa
        .data
f:      .word     # as duas primeiras localizações da zona da memória de
g:      .word     # dados reservadas para armazenar o valor de f e g
        …
```


Compilar usando os registos

```
f = (g+h) – (i+j);
```

f mapeado em **$s0**
g mapeado em **$s1**
h mapeado em **$s2**
i mapeado em **$s3**
j mapeado em **$s4**

```
add $t0, $s1, $s2  # $t0 = g + h
add $t1, $s3, $s4  # $t1 = i + j
sub $s0, $t0, $t1  # $s0 = (g+h)-(i+j)
```

Nome da operação

Operando onde é armazenado o resultado

1º operando

2º operando

Sintaxe do *assembly*

Operandos em memória

C

Declarar uma variável numa linguagem de alto nível é reservar uma posição de memória para armazenar o respetivo valor

```
int a, b, c;
…
a = b + c;
…
```

Assembly

\# 1. transferir operandos da memória para registos **lw …**

\# 2. efetuar operação; resultado colocado num registo **add …**

\# 3. armazenar resultado na memória **sw …**

load – transfere o conteúdo da posição de memória cujo endereço é indicado no código de operação para um registo – **Memory Read**

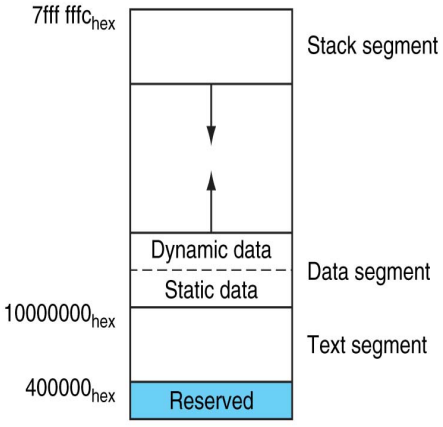
(Registo) = **Mem[Address]**

store – transfere para a memória o conteúdo de um registo – **Memory Write**

Mem[Address] = (Registo)

Mapa de memória

Text: código do programa
Static data: variáveis globais
e.g., *static variables* em C,
constant arrays e strings
acesso via $gp ±offsets
Dynamic data: ***heap***
e.g., ***malloc*** em C, ***new*** em Java
Stack: *automatic storage*
variáveis locais



Instruções de transferência de dados

- *Load* e *Store* precisam de indicar o endereço de memória:

  Mem.Addr. = Base Register + Offset

  (único modo de endereçamento do MIPS)

- **`lw`** (load word):

  `lw $t0, 8($s3)` # ($t0) = Mem[($s3) + 8]

  "offset" "base register"

- **sw** (store word):

  `sw $t0, 8($s3)` # Mem[($s3) + 8] = ($t0)

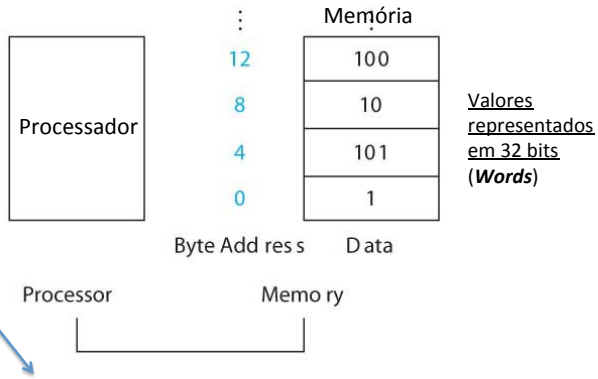
Organização da memória

MIPS – Memória endereçável byte a byte – a bytes sucessivos correspondem endereços sucessivos (***Byte-addressable memory***)

Byte – 8 bits;

Word – 32 bits (4 bytes)

➢Endereço de duas palavras sucessivas de memória difere de 4



Valores representados em 32 bits (***Words***)

Exemplo:

A[12] = h + A[8]

Endereços das palavras em memória alinhados (múltiplos de 4): ***Word Address*** = XXX…XX**00**

```
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 48($s3)
```

Endereço do elemento índice 8 do array (32 = 8*4)

Endereço do 1º elemento do array

Endereço do elemento índice 12 do array (48 = 12*4)

Armazenamento de arrays em memória

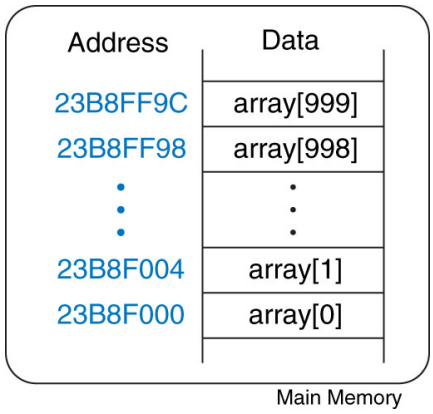


Figure 6.21 Memory holding array[1000] starting at base address 0x23B8F000

Constantes e Operandos Imediatos

- Muitas operações aritméticas envolvem constantes
  - Exemplo: endereçar elementos sucessivos de um array
  - Mais eficiente incluir a constante 4 no código de instrução em lugar de a armazenar em memória e ter de a transferir para um registo sempre que se queira utilizá-la

Imediato (constante) – indicado no código da instrução

addi $s3, $s3, 4 # $s3 = $s3 + 4

add immediate

$s3 "fica a apontar" para o elemento seguinte do array