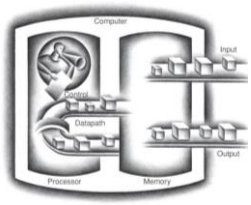
# Arquitetura de Computadores I

## Instruções executáveis por um processador - a arquitetura MIPS

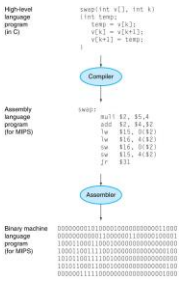
António de Brito Ferrari
ferrari@ua.pt






7. O conjunto de instruções do processador

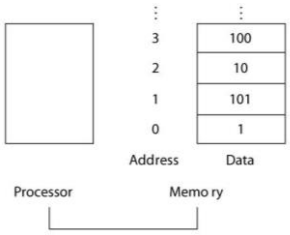
## Das Linguagens de Alto Nível à linguagem do processador



## 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 e a velocidade de resolução desses problemas*

  Burks, Goldstine e von Neumann, 1947




**Memória: array de células que armazenam informação**
**Espaço de Endereçamento – número de células de memória endereçáveis**

Duas operações possíveis

1. Escrita – armazena informação na célula de memória indicada pelo endereço

   **Mem[Address] = Data**

1. Leitura – obtem a informação armazenada na célula de memória cujo endereço é fornecido

   **Data = Mem[Address]**

Espaço de Endereçamento do MIPS: $\mathbf{2^{32}}$ (4,294,967,296 bytes = 4 GB)

## Passos básicos de execução de um programa

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

## Tipos de instruções

- Que operações incluir no **IS** (Instruction Set)?
  - Aritméticas e lógicas
  - Transferência de dados de/para a memória – ir buscar à memória os dados sobre que operar (*load)* e armazenar na memória o resultado das operações efetuadas (*store*)
  - Instruções de controle da sequência de execução das instruções do programa (*if-then-else, case, invocação de funções*)

## O Instruction Set do MIPS – instruções básicas

**MIPS operands**

| Name | Example | Comments |
|---|---|---|
| 32 registers | $s0, $s1, ..., $s7, $t0, $t1, ..., $t9, $zero, $a0, $a1, $a2, $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 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 conditional 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 |

## Execução de uma instrução

**2. EXECUTE:** processador executa a operação indicada no código de instrução

**Onde estão colocados os operandos?**

Duas alternativas:

a. Na memória – de cada vez que uma operação aritmética ou lógica é executada é necessário ir buscar o operando à memória

***von Neumann bottleneck***

b. Em registos do processador – acesso mais rápido – operandos em registos e resultado da operação colocado tambem num registo

**MIPS e todas as outras arquiteturas tipo RISC**

MIPS: 32 registos - r0 a r31

## Operações aritméticas

Todas as instruções aritméticas têm 2 operandos:

**add a, b, c** # a = b + c
**add a, a, d** # a = a + d (= b + c + d)
**add a, a, e** # a = a + e (= b + c + d + e)

- 3 instruções para somar 4 variáveis
- 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** |
|---|---|
| a = b + c; | add a, b, c |
| d = a - e; | sub d, a, e |
| f = (g + h) – (i + j) | add t0, g, h<br>add t1, i, j<br>sub f, t0, t1 |
| | t0, t1 – variáveis temporárias<br>f, g, h, i, j, a, b, c, d, e – variáveis |

## Operandos em Registos

t0, t1 – variáveis temporárias
f, g, h, i, j – variáveis

- MIPS – 32 Registos, r0 .. r31 de 32-bits "***arquitetura de 32-bits***"
  - **$zero** (r0) – contem a constante **0**
- Convenções de uso e nomes dos registos em ***assembly***:
  - **`$s0`**, … , **`$s7`** (r16 a r23) - correspondem a variáveis em C
  - **`$t0`**, … , **`$t7`** (r8 a r15) - correspondem a variáveis temporárias

Nota: em C os operandos (variáveis) são declarados, em a *Assembly* os operandos (registos) são fixos e não são declarados

## 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
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Sintaxe do *assembly*:

1) Nome da operação
2) Operando onde é colocado o resultado
3) 1º operando da operação
4) 2º operando da operação

## 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**

```
# transferir operandos da memória para registos
add a, b, c
# colocar resultado na memória
```

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

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

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

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

## 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):

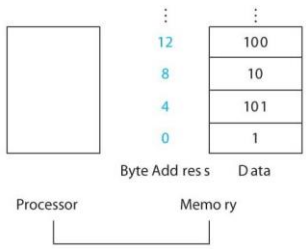
  **`sw $t0, 8($s3)`** # Mem[($s3) + 8] = ($t0)

**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




Exemplo:

A[12] = h + A[8]

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

Endereço do 1º elemento do array

## Constantes e Operandos Imediatos

- Muitas operações aritméticas envolvem constantes
  - Exemplo: endereçar elementos sucessivos de um array
  - Mais eficiente incluir a constante 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

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

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

## Inteiros com e sem sinal

- Revisão – ver slides

"**Representação de inteiros e aritmética binária**"