PROJETOS DIGITAIS E MICROPROCESSADORES
MIPS: CONJUNTO DE INSTRUÇÕES

Marco A. Zanata Alves

PRINCÍPIOS DE PROJETO EM ARQUITETURA

Princípio 1: Simplicidade favorece regularidade

Princípio 2: Menor é mais rápido (quase sempre)

Princípio 3: Um bom projeto demanda compromissos (cobertor curto)

Princípio 4: O caso comum deve ser o mais rápido

MODELO DE VON NEUMANN

"First Draft of a Report on the EDVAC" - John Von Neumann,

Moore School of Electrical Engineering, Univ. of Pennsylvania, 1945

Define um computador com programa armazenado no qual a memória é um vetor de bits e a interpretação dos bits é determinada pelo programador

Iremos começar analisando uma arquitetura simples, chamada MIPS

HISTÓRICO

1981 - J. Hennessy e sua equipe de Stanford propõem a arquitetura MIPS

1984 – J. Hennessy funda a MIPS Computer Systems

1991 - A Silicon Graphics compra a MIPS

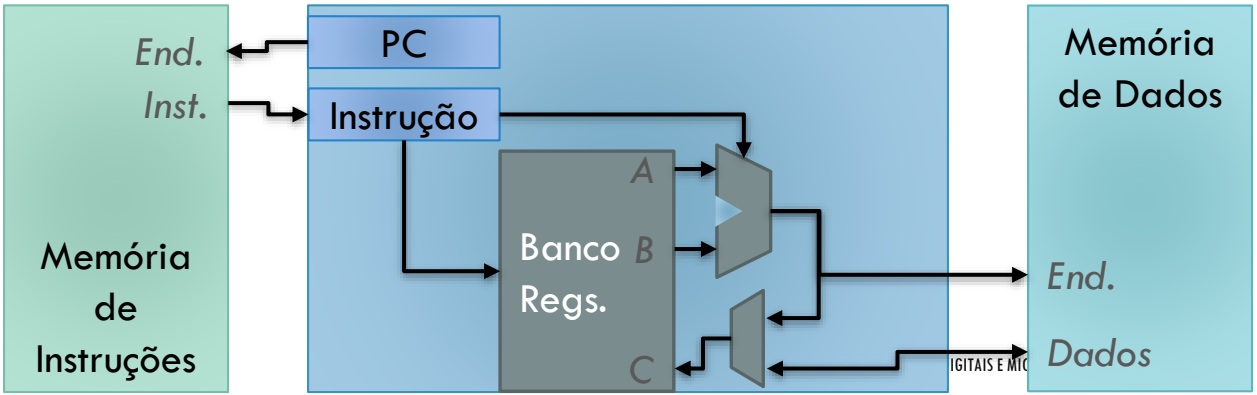
1999 - Um em cada três processadores embarcados possui arquitetura MIPS

2006 - É lançado o primeiro MIPS Multithreaded

FASES DE EXECUÇÃO DE UMA INSTRUÇÃO (I)

Processador:

- Busca na memória a instrução apontada por PC/IP — **busca**
- Decodifica instrução + Acesso aos operandos — **decodificação**
- Executa operação — **execução:** `A + B`
- Acesso à memória — **memória:** `mem[A + desl]`
- Armazena resultado da operação — **resultado:** `regs[c] ← ...`



FASES DE EXECUÇÃO DE UMA INSTRUÇÃO (II)

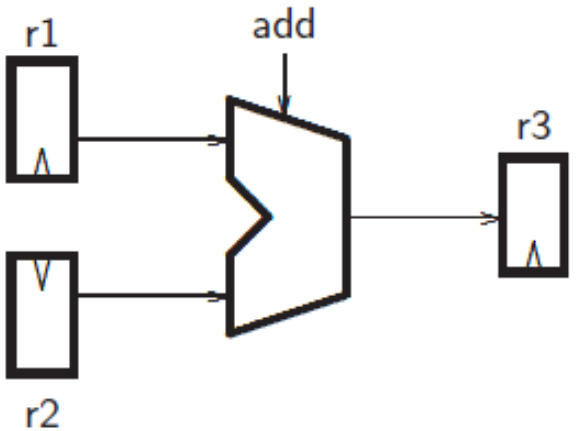
```
add r3,r1,r2 # r3←r1+r2
```

busca instrução;

decodifica, acessa regs;

executa;

grava resultado;



LINGUAGEM DE MONTAGEM (I)

Extremamente simples (programa montador em 200 linhas de C)

Poucos tipos de dados: byte, meia-palavra, palavra, float, double

Dois conjuntos de variáveis: 32 registradores e vetor de bytes (memória)

Tipicamente, consome dois operandos e gera um resultado por instrução

- Uma instrução por linha
- **Label:** denota endereço da linha indicada (opcional, note o ':')
- **Comentário** vai do '#' ou ';' até o fim da linha (opcional)
- Ex: Label: instrução # comentário

```
.L1: add r1, r2, r3    # r1 ← r2 + r3
     sub r5, r6, r7    # r5 ← r6 - r7
fim: j .L1             # salta para endereço
                       # apontado por .L1
```

LINGUAGEM DE MONTAGEM (II)

Dois formatos básicos de sintaxe: Intel ou AT&T

```
#Intel Sintaxe          instr destino, origem
mov r2, 1               # r2 ← 1
add r5, r6, r7          # r5 ← r6 - r7
```

```
#AT&T Sintaxe           instr origem, destino
mov $1, %r2             # 1 → r2
add %r6, %r7, %r5       # r6 - r7 → r5
```

LINGUAGEM DE MONTAGEM (III)

Programa montador (assembler) traduz "linguagem de montagem" (assembly language) para "linguagem de máquina" = binário que é interpretado pelo processador.

Comentários com atribuição denotada por ←

```
/* programa C */          # Equiv. em Assembly MIPS
a = b + c;                add a, b, c # a ← b + c

a = b + c + d + e;        add a, b, c # a ← b + c
                          add a, a, d # a ← a + d
                          add a, a, e # a ← a + e

f = (g+h) - (i+j);        add t0, g, h  # t0 ← g + h
                          add t1, i, j  # t1 ← i + j
                          sub f, t0, t1 # f ← t0 - t1
```

LINGUAGEM DE MONTAGEM (IV)

Instruções aritméticas/lógicas com 3 operandos RISC

- Circuito que decodifica as instruções é mais simples

Operandos SEMPRE em registradores RISC

Palavra do MIPS é de 32 bits = |ULA|+ |regs|+|vias|

32 registradores visíveis: r0 a r31 (montador usa $0 a $31)

```
f = (g+h)-(i+j);        add r8, r17, r18
                        add r9, r19, r20
                        sub r16, r8, r9
```

Por convenção

- rN identifica registrador N
- r0 contém sempre zero (fixo no hardware)
- r1 é variável temporária para montador, não deve ser usada

ARITMÉTICA COM E SEM SINAL (*SIGNED* E *UNSIGNED*)

A representação de inteiros usada no MIPS é complemento de dois

Operações aritméticas possuem dois sabores:

- signed ("com-sinal") → overflow causa exceção
- unsigned ("sem-sinal") → ignora detecção de overflow

Operações com endereços são sempre sem-sinal:

- `addu r1, r2, r3`
- Porque todos os 32 bits compõem o endereço
- `0xffff ffff` = -1_{10} é um endereço válido

Operações com inteiros podem ter operandos positivos/negativos, e (talvez) o programa deva detectar a ocorrência de overflow:

- A soma de dois números de 32 bits produz resultado de 33 bits

INSTRUÇÕES DE LÓGICA E ARITMÉTICA (I)

```
add r1, r2, r3          # r1 ← r2 + r3
addi r1, r2, const      # r1 ← r2 + ext(const = “imediato”)
addu r1, r2, r3         # sem sinal - não causa exceção
addiu r1, r2, const     # sem sinal - não causa exceção
ori r1, r2, const       # r1 ← r2 or {016 & const(15:0)}
```

Por que estender o sinal?

Para transformar constante de 16 bits em número de 32 bits:

- `0x4000 → 0x0000.4000` $4 = 0100_2$
- `0x8000 → 0xffff.8000` $8 = 1000_2$

INSTRUÇÕES DE LÓGICA E ARITMÉTICA (II)

Como obter constantes em 32 bits?

- addi e ori tem operandos de 16 bits

`%hi()` e `%lo()` são operadores do montador que extraem as partes MAIS/menos significativas dos operandos

```
ori r1, r2, const          # r1 ← r2 or {0^16 & const(15:0)}
lui r1, const              # r1 ← {const(15:0) & 0^16}
```

```
lui r5, %hi(0x0080.4000)                  # r5 ← 0x0080.0000
ori r5, r5, %lo(0x0080.4000)              # r5 ← 0x0080.4000
                          # r5 ← 0x0080.0000 or 0x0000.4000
```

```
la r5, 0x0080.4000         # la é uma pseudo instrução
                           # que substitui lui ; ori
```

VARIÁVEIS EM MEMÓRIA

Programas usam mais variáveis que os 32 registradores!

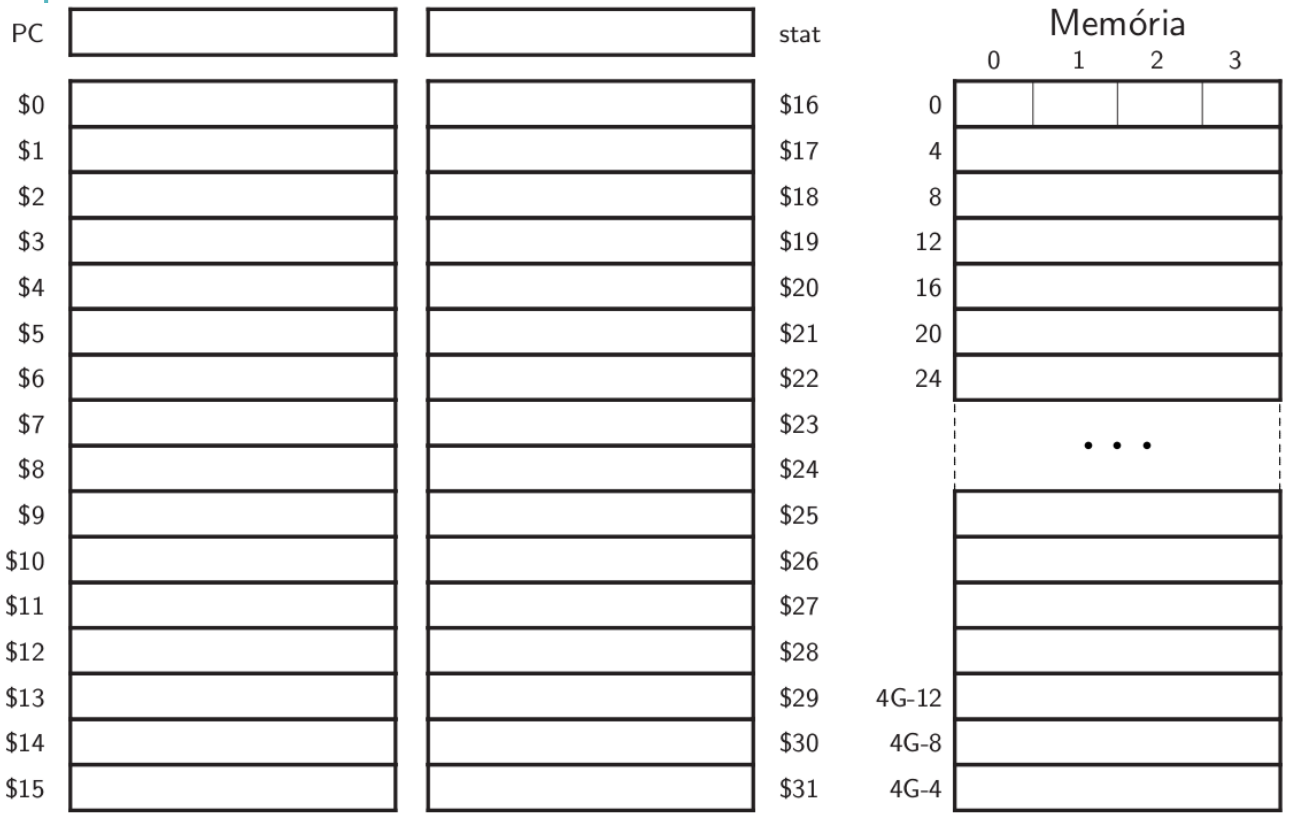
- Variáveis, vetores, matrizes, etc. são alocados em memória
- Operações com elementos necessitam da carga dos registradores antes das operações

Memória é um vetor: M[2^{32}] bytes, ou seja 4GB

- Endereço em memória é o índice *i* do vetor M[*i*]
- Bytes são armazenados em endereços consecutivos
- Palavras armazenadas em endereços múltiplos de 4 (2^{30} palavras)

```
bytes              end % 1 = ?
meia-palavras      end % 2 = 0 alinhado!!
palavras           end % 4 = 0 alinhado!!
double-words       end % 8 = 0 alinhado!!
```

REGISTRADORES VISÍVEIS E MEMÓRIAS



REGISTRADORES - CONVENÇÃO

#	Nome	Função	Preservado
0	$zero	Constante	-
1	$at	Reservado para Assembly	Não
2	$v0	Retorno da Função	Não
3	$v1		
4	$a0	Argumentos da Função	Não
5	$a1		
6	$a2		
7	$a3		
8	$t0	Temporários	Não
...			
15	$t7		

#	Nome	Função	Preservado
16	$s0	Temporários Salvos	Sim
...			
23	$s7		
24	$t8	Temporários	Não
25	$t9		
26	$k0	Reservado para SO/Kernel	-
27	$k1		
28	$gp	Ptr. Área Global	Sim
29	$sp	Ptr. Pilha	Sim
30	$fp	Ptr. Frame	Sim
31	$ra	End. Retorno	-

MOVIMENTAÇÃO DE DADOS ENTRE CPU E MEMÓRIA (I)

LOAD WORD: End. efetivo = Deslocamento + Reg. Índice

- `lw rd, desloc(regIndice)`

STORE WORD: End. efetivo = Deslocamento + Reg. Índice

- `sw rd, desloc(regIndice)`

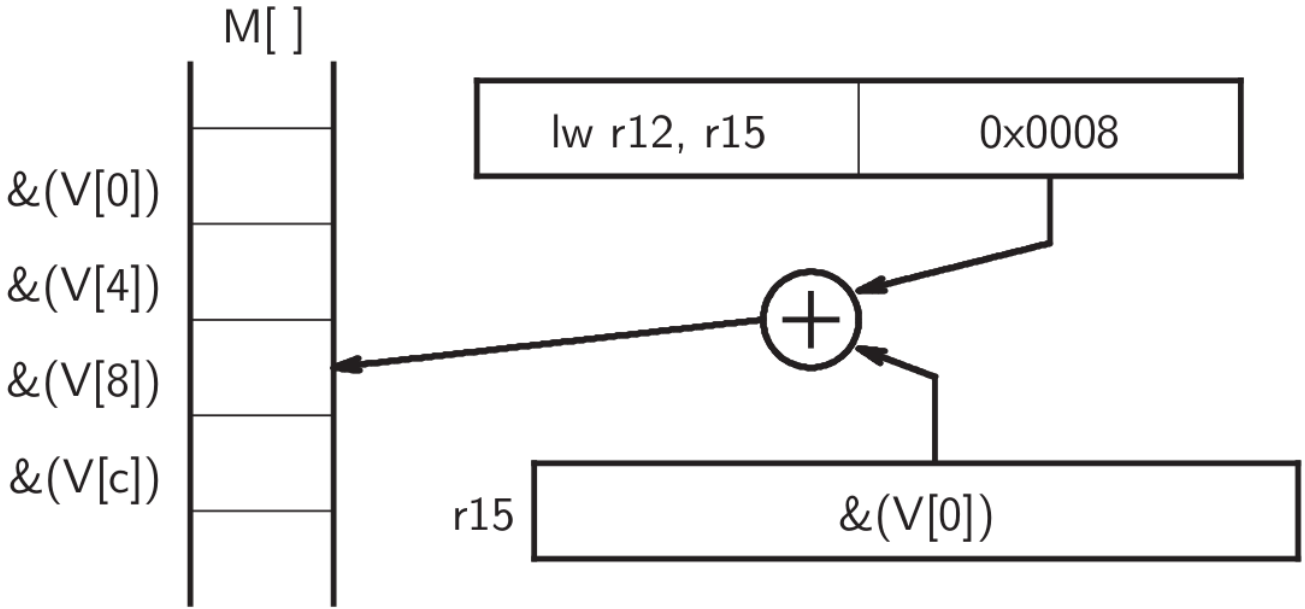
```
lw r8, 64(r15)      # r8 ← M[ 64 + r15 ]
sw r8, -16(r15)     # M[ -16 + r15 ] ← r8
```

O MIPS é chamado de **arquitetura load/store** pois apenas essas duas instruções podem acessar a memória.

Programador é responsável por gerenciar o acesso a todas as estruturas de dados; palavras devem ser acessadas de 4 em 4 bytes

MOVIMENTAÇÃO DE DADOS ENTRE CPU E MEMÓRIA (II)

```
lw r12, 8(r15)          # r12 ← M[ r15 + 8 ]
```



ESTRUTURAS DE DADOS EM C

A função `sizeof(x)` retorna o número de bytes necessários para representar `x`

Elementos de vetores são alocados em endereços contíguos: `V[i+1]` é alocado no endereço seguinte a `V[i]`.

Ponteiros (char *, int *) são **endereços e tem sempre o mesmo tamanho,** que é de 4 bytes no MIPS

Tipo de Dado	Sizeof
char	1
short	2
int	4
long long	8
float	4
double	8
char[12]	12
short[6]	12
int[3]	12
char *	4
short *	4
int *	4

VETORES E MATRIZES EM C

Vetores em C

End.	20	21	22	23	24	25	26	27
char	V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]
short	V[0]		V[1]		V[2]		V[3]	
int	V[0]				V[1]			

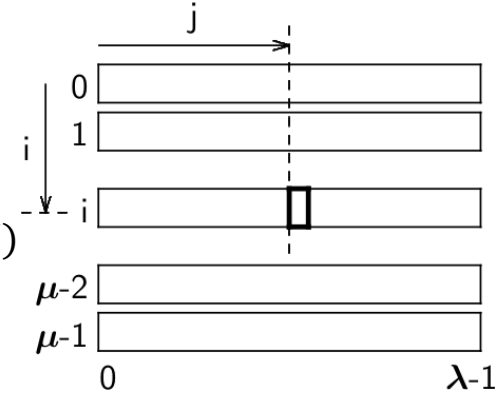
Matrizes em C

Uma matriz é alocada em memória como vetor de vetores

$$\&(M[i][j]) = \&(M[0][0]) + |\tau|((\lambda \cdot i) + j)$$

Para elementos de tipo τ ,

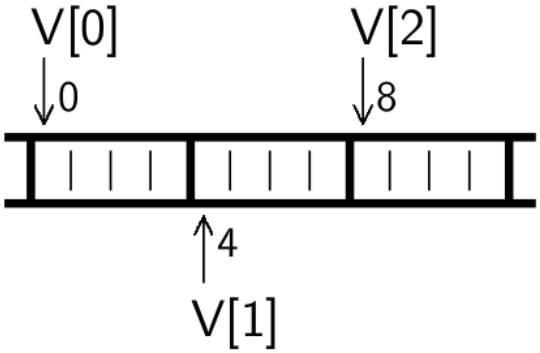
linhas com λ colunas e μ linhas



MOVIMENTAÇÃO DE DADOS ENTRE CPU E MEMÓRIA (III)

Exemplo: acesso à vetor

```
int V[NNN];
...
V[0] = V[1] + V[2]*16;
```




```
la r1, End_V            # r1 ← &V[0]
lw r4, 4(r1)            # r4 ← M[r1+1*4]
lw r6, 8(r1)            # r6 ← M[r1+2*4]
sll r6, r6, 4           # r6 ← r6*16 = r6<<4
add r7, r4, r6          # r7 ← V[1] + V[2]*16
sw r7, 0(r1)            # M[r1+0*4] ← r4+r6
```

MOVIMENTAÇÃO DE DADOS ENTRE CPU E MEMÓRIA (III)

```
int V[NNN];
...
V[0] = V[1] + V[2]*16;
```

Reescreva o código para:
V[i] = V[j] + V[k]*16;

```
la r1, V                    # r1 ← &V[0]
lw r4, 4(r1)                # r4 ← M[r1+1*4]
lw r6, 8(r1)                # r6 ← M[r1+2*4]
sll r6, r6, 4               # r6*16 = r6<<4
add r7, r4, r6
sw r7, 0(r1)                # M[r1+0*4] ← r4+r6
```

ESTRUTURAS DE DADOS EM C – STRUCTS

Registros (structs) agregam informação relacionadas

Componente do registro é selecionado com o operador '.'

```
for (i=0; i<100; i++){
 aluno.nome[i]=candidato[i];
}
```

```
struct aluno {
  char nome[100];
  int GRR;
  short anoIngresso;
  float IRA;
}
...
aluno.GRR = 12345;
aluno.anoIngresso = 2010;
aluno.IRA = 0.666;
```

ESTRUTURAS DE DADOS EM C (CONT.)

Registros podem ser usados para definir novos tipos:

O typedef declara o registro aluno como sendo o novo tipo alunoType.

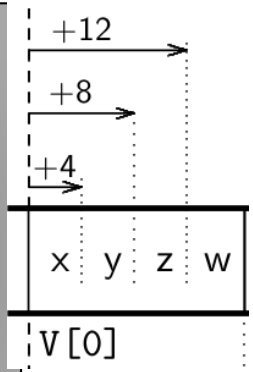
Este novo tipo pode ser usado na declaração do vetor ufpr[60000].

```
typedef struct aluno {
  char nome[100];
  int GRR;
  short anoIngresso;
  float IRA;
} alunoType;
alunoType ufpr[60000];
...
for (i=0; i<60000; i++) {
  ufpr[i].GRR = 0;
  ufpr[i].IRA = 0.0;
}
```

MOVIMENTAÇÃO DE DADOS ENTRE CPU E MEMÓRIA (IV)

Exemplo: acesso à estrutura com 4 elementos

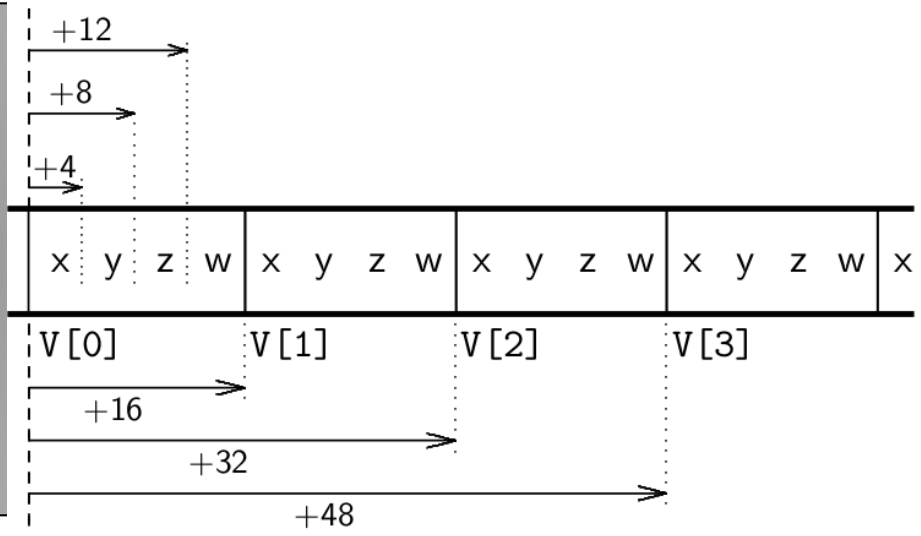
```
typedef struct A {
  int x;
  int y;
  int z;
  int w;
} aType;
aType V[16];
```



MOVIMENTAÇÃO DE DADOS ENTRE CPU E MEMÓRIA (IV)

Exemplo: acesso à estrutura com 4 elementos

```
typedef struct A {
  int x;
  int y;
  int z;
  int w;
} aType;
aType V[16];
```



MOVIMENTAÇÃO DE DADOS ENTRE CPU E MEMÓRIA (V)

Exemplo: acesso à estrutura com 4 elementos

```
typedef struct A {
  int x;                    ...
  int y;                    // compilador aloca V
  int z;                    // em 0x0080.0000
  int w;                    aType V[16];
} aType;                    ...
```

```
# 3 elementos * 4 palavras/elemento * 4 bytes/palavra = 0x30 = 48
                            la r15, 0x00800030
m = V[3].y;                 lw r8, 4(r15)
n = V[3].w;                 lw r9, 12(r15)
V[3].x = m+n;               add r5, r8, r9
                            sw r5, 0(r15)
```

INSTRUÇÕES DE MOVIMENTAÇÃO DE DADOS ENTRE CPU E MEMÓRIA

```
lw r1, desl(r2)        # r1 ← M[ r2 + ext(desl) ]
sw r1, desl(r2)        # M[ r2 + ext(desl) ] ← r1
```

load-half and load-byte -- expande sinal do deslocamento para 32 bits

```
                       # x = r2+ext(desl)
lh r1, desl(r2)        # r1 ← M[x](15)^16 & M[x](14:0)
lb r1, desl(r2)        # r1 ← M[x](7)^24 & M[x](6:0)
```

load-half and load-byte **unsigned** -- preenche com zeros o deslocamento

```
lhu r1, desl(r2)       # r1 ← 0^16 & M[x](15:0) (16 bits)
lbu r1, desl(r2)       # r1 ← 0^24 & M[x](7:0)  (8 bits)
```

EXERCÍCIOS

Traduza para assembly do MIPS os seguintes comandos em C:

```
int P[NN];
int Q[MM];
int x, y, z, i, j, k;

i = P[4];
j = P[9];
k = i - j;
y = Q[i];
z = Q[i*4];
P[ P[k] ] = y + z + Q[i+j];
```