Princípios de Projeto em Arquitetura

Princípio 1: simplicidade favorece regularidade

Princípio 2: menor é mais rápido (quase sempre)

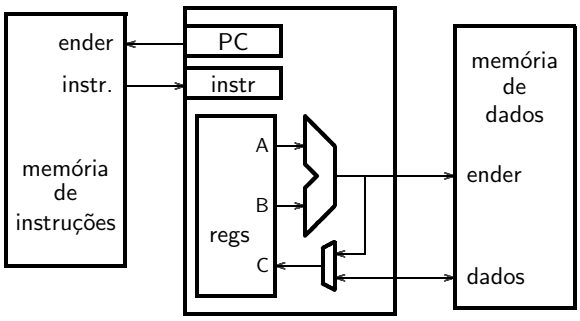
Princípio 3: um bom projeto demanda compromissos

Princípio 4: o caso comum deve ser o mais rápido

Modelo de Von Newman

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

Fases de execução de uma instrução

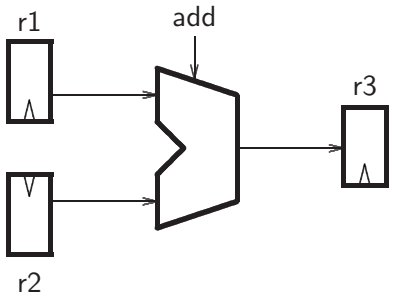


Processador:

1) busca na memória a instrução apontada por PC — busca
2) decodifica instrução — decodificação
3) executa operação — execução: $A + B$
4) acesso à memória — memória: $mem[A + desl]$
5) armazena resultado da operação — resultado: regs[c] $\leftarrow \cdots$

Fases de execução de uma instrução (cont.)

```
add r3,r1,r2 # r3 ← r1+r2
```



busca instrução;
decodifica, acessa regs;
executa;
grava resultado;

Linguagem de montagem

- Extremamente simples (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
- tipicamente, um resultado e dois operandos por instrução

```
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
```

uma instrução por linha,
label: denota endereço da linha indicada,
comentário vai do '#' ou ';' até o fim da linha.

Linguagem de montagem (cont.)

```
/* programa C */    # equivalente em assembly MIPS
a = b+c;            add a, b, c

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

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

Programa montador (*assembler*) traduz "linguagem de montagem" (*assembly*) para "linguagem de máquina" – binário que é interpretado pelo processador

Linguagem de montagem (cont.)

- 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 = |regs| = |ULA| = |vias|
- 32 registradores visíveis: $0 a $31

Usando registradores no último exemplo:

```
f = (g+h)-(i+j);  add $8, $17, $18 # f..j -> $16..$20
                  add $9, $19, $20
                  sub $16, $8, $9
```

Por convenção
  $0 contém sempre zero (fixo no hardware)
  $1 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: com/sem overflow
signed (faz detecção de overflow), patético
unsigned (ignora detecção de overflow). patético

Operações com endereços são sempre sem-overflow
(ex. `addu $1, $2, $3`) porque todos os 32 bits compõem
o endereco: `0xffff ffff` $= -\mathbf{1}_{10}$ é um endereço válido

Operações com inteiros podem ter operandos positivos/negativos,
e (talvez) 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

```
add r1, r2, r3       # r1 ←r2+r3

addi r1, r2, const   # r1 ←r2+ext(const)

addu r1, r2, r3      # sem sinal - não causa exceção
addiu r1, r2, const  # sem sinal - não causa exceção

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

Por que estender o sinal?
  constante numérica de 16 bits ⇝ número de 32 bits
  constante lógica de 16 bits ⇝ constante de 32 bits

Variáveis em memória

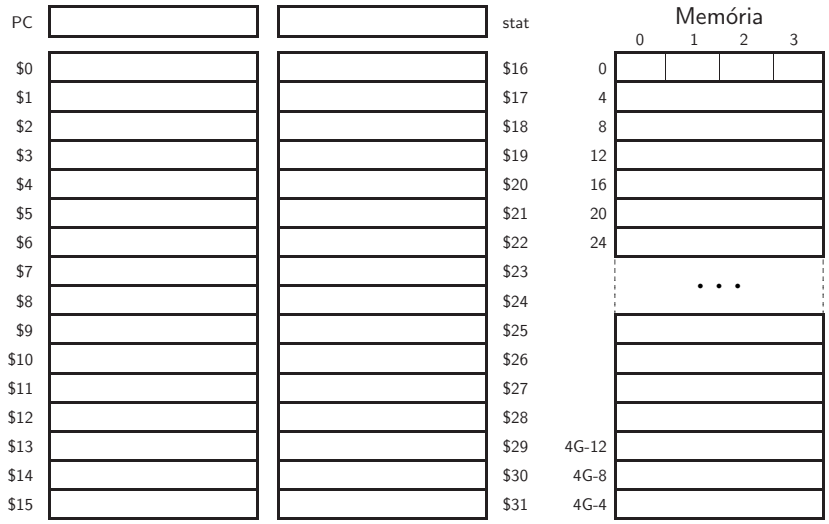
Programas usam mais variáveis que os 32 registradores!
Variáveis, vetores, etc são alocados em memória

Operações com elementos implicam na
carga dos registradores antes das operações

Memória é um vetor: M[$4 * 2^{30}$]
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 Memória

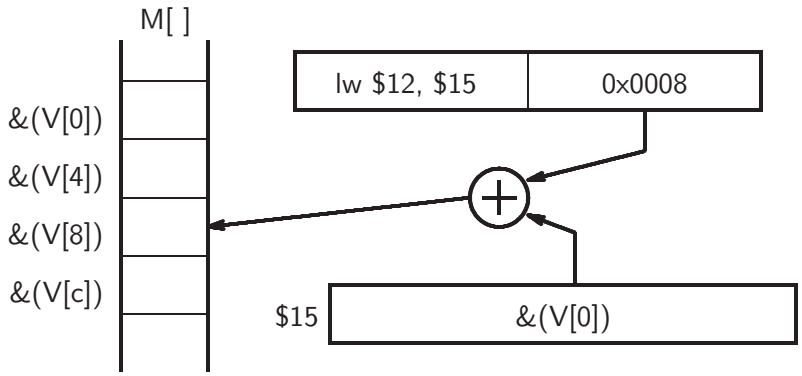


Movimentação de dados entre CPU e memória (i)

```
# LOAD WORD: end_efetivo = desloc + rIndice
  lw rd, desloc(rIndice)

# STORE WORD: end_efetivo = desloc + rIndice
  sw rd, desloc(rIndice)

  lw $8, desloc($15)     # $8 <-- M[ desloc + $15 ]

  sw $8, desloc($15)     # M[ desloc + $15 ] <-- $8
```

Programador é responsável por gerenciar o acesso a <u>todas</u> as estruturas de dados

Movimentação de dados entre CPU e memória (ii)



lw \$12, 8(\$15) # \$12 ⇐ M[\$15+8]

Movimentação de dados entre CPU e memória (iii)

Exemplo: acesso à estrutura com 4 elementos

```
typedef struct A {          ...
   int x;                   # compil aloca V em 0x800000
   int y;                   aType V[16];
   int z;                   ...
   int w;
} aType;

   # 3 elmtos * 4 pals/elmto * 4 bytes/pal
   aPtr = &(V[3]);          la $15, 0x00800030
   m = aPtr->y;             lw $8, 4($15)
   n = aPtr->w;             lw $9, 12($15)
   aPtr->x = m+n;           add $5, $8, $9
                            sw $5, 0($15)
```


Estruturas de Dados em C

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

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

Vetores e Matrizes em C

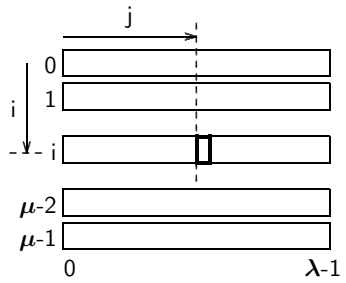
Vetores em C

ender	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

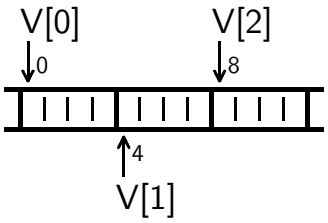
$\&(M[i][j]) =$
$\&(M[0][0]) + |\tau|(\lambda \cdot i + j)$
para elementos de tipo τ, linhas com λ colunas e μ linhas



Movimentação de dados entre CPU e memória (iii)

Exemplo: acesso à vetor

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

la r1, 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

# Re-escreva o código para:
V[i] = V[j] + V[k]*16;
```



Instr de moviment 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 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
lhu r1, desl(r2)   # r1 ←{0^16, M[x](15:0)}

lbu r1, desl(r2)   # r1 ←{0^24, M[x](7:0)}
```

fim da primeira aula

Controle de fluxo de execução (i)

Instruções para efetuar Desvios if( ){ } while( ){ }

```
beq r1, r2, ender  # branchEqual desvia se r1 == r2
bne r1, r2, ender  # branchNotEq desvia se r1 != r2
```

Instruções para efetuar Saltos goto

```
j ender            # jump (salto incondicional)
jr rt              # jump register
                   # rt contem endereço de destino
```


Controle de fluxo de execução (i)

Instruções para efetuar Desvios if( ){ } while( ){ }

```
beq r1, r2, ender    # branchEqual desvia se r1 == r2
bne r1, r2, ender    # branchNotEq desvia se r1 != r2
slt rd, r1, r2       # setOnLessThan rd←1 se r1 < r2
                     # em C: rd = ((r1 < r2) ? 1 : 0);
```

sequência equivalente a `blt` (branch on less than)

```
slt r1, r2, r3       # r1 <-- 1 se (r2 < r3)
bne r1, r0, ender    # salta se (r2 < r3)

slt rd, r1, r2       # rd←1 se ( r1 < r2 )
slti rd, r1, const   # rd←1 se ( r2 < ext(const) )
sltu rd, r1, r2      # subtração não gera exceção
sltiu rd, r1, const  # subtração não gera exceção
```

Desvios e Saltos (i)

```
if (i == j) goto L1;          beq $i, $j, L1
  f = g + h;                  add $f, $g, $h
L1:                     L1:   sub $f, $f, $i
f = f - i;
```

```
if (i == j)                   bne $i, $j, Else
  f = g + h;                  add $f, $g, $h
else                          j Exit    # salta else
  f = g - h;            Else: sub $f, $g, $h
                        Exit:
```


Desvios e Saltos (ii)

```
while (save[i] == k)
  i = i + j;

# i,j,k <-> $19,$20,$21, $7 = &(save[0])
Loop:  muli $9, $19, 4      # $9←i*4
       add $9, $7, $9       # $9←&(save[i])
       lw $8, 0($9)         # $8←save[i]
       bne $8, $21, Exit
       add $19, $19, $20
       j Loop
Exit:
```


Modos de Endereçamento

Modos de endereçamento já vistos:

- **a registrador** – instrução especifica registradores que contém operandos e destino
  `add $4, $3, $2`
- **base-deslocamento** – endereço_efetivo é conteúdo_de_registrador + deslocamento_16_bits
  `lw $4, 32($5)`

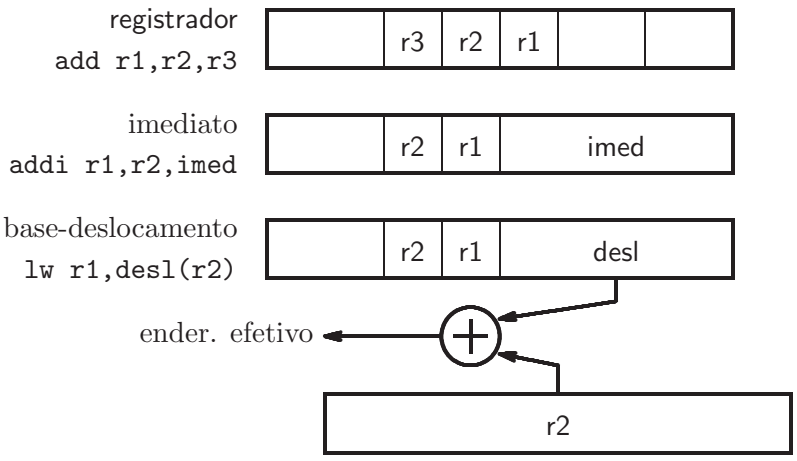
Endereçamento com Imediatos

Motivação:
no gcc, 52% das operações aritméticas involvem uma constante;
no simulador de circuitos Spice são 69%.

Exemplos:

```
addi $29,$29,4  # add-immediate:  $29 = $29+4
slti $8,$18,10  # set-on-less-than-immeditate:
                # $8 ←($18 < 10);
lui  $8,252     # load-upper-immediate: operando nos
                # 16 bits mais signif do registrador
```

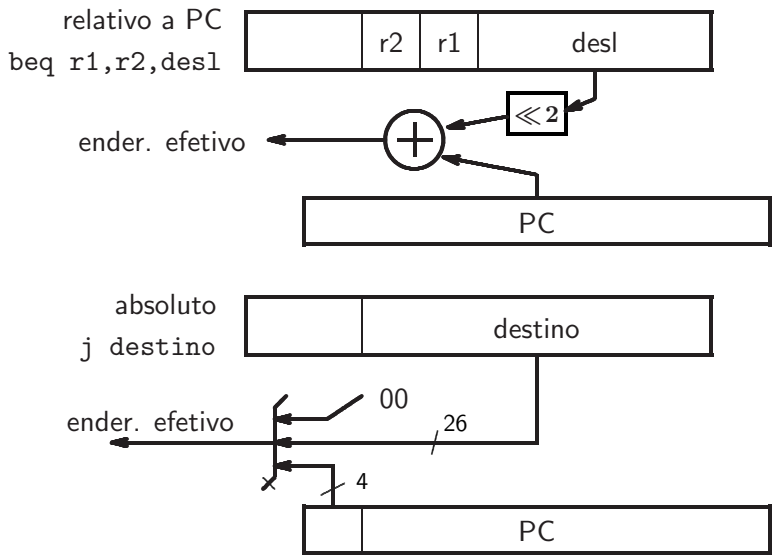

MdE: registrador, imediato, base-deslocamento



Endereçamento em Saltos e Desvios

- Em geral, desvios são para endereços próximos
- por ser rápido e eficiente, desvios são relativos ao PC
- o PC contém o endereço da próxima instrução a ser executada

Endereçamento em Saltos e Desvios



Endereçamento em Saltos e Desvios

Relativo à PC – endereço efetivo = (PC+4) + deslocamento

Na imensa maioria dos casos, uma distância de ± 32K palavras (16 bits) é suficiente para cobrir `if()`'s, `for()`'s, etc...

Se o destino de um desvio está além das 32K palavras, a seguinte transformação é efetuada automaticamente pelo montador:

```
beq  $18, $19, L1       # | L1 - PC | > 32K palavras
```

é transformada em (pela inversão do teste)

```
bne  $18, $19, L2       # | L2 - PC | < 32K palavras
j    L1                 # | L1 | <= 2**26
L2:
```

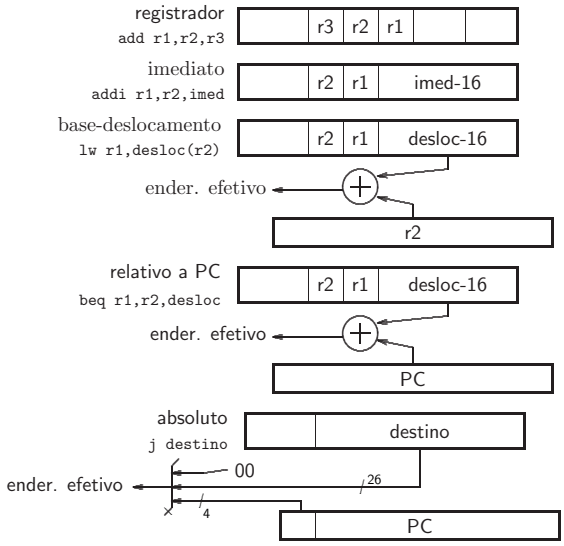

Modos de Endereçamento

- **a registrador:** operandos e destino em registradores
- **imediato:** constante é parte da instrução
- **base-deslocamento:** end_efetivo = reg + deslocamento
- **relativo a PC:** end_efetivo = PC + deslocamento
- **(pseudo)absoluto:** end_efetivo é parte da instrução

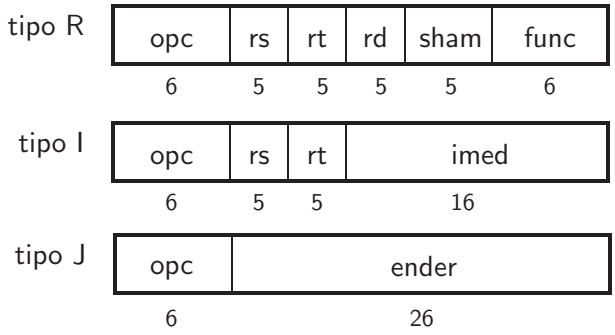
∗ **Princípio 1**: simplicidade favorece regularidade
∗ **Princípio 3**: um bom projeto demanda compromissos
∗ **Princípio 4**: o caso comum deve ser o mais rápido

Quais são os casos comuns?
Quais são os compromissos?

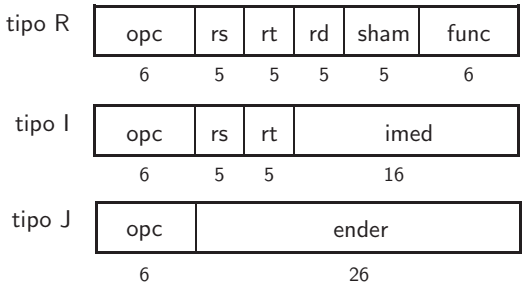
Modos de Endereçamento



Codificação das instruções



Codificação das instruções

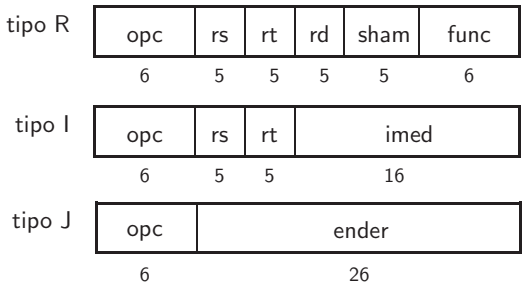


- **Princípio 1**: simplicidade favorece regularidade
- **Princípio 3**: um bom projeto demanda compromissos
- **Princípio 4**: o caso comum deve ser o mais rápido

Quais são os casos comuns?
Quais são os compromissos?

Modos de Endereçamento vs Codificação

- **a registrador:** operandos e destino em registradores
- **imediato:** constante é parte da instrução
- **base-deslocamento:** end_efetivo = reg + deslocamento
- **relativo a PC:** end_efetivo = PC + deslocamento
- **(pseudo)absoluto:** end_efetivo é parte da instrução



Qual a relação entre codificação e modos de endereçamento?

Pseudoinstruções

Montador sintetiza instruções mais complexas a partir de instruções simples do conjunto de instruções original do MIPS

```
li   $a0, 4          # load-immediate
é
ori  $a0, $0, 4      # or-immediate com $0 (zero)
#-----------------------------------------------
move $a1, $v0        # move conteúdo de $v0 para $a1
é
ori  $a1, $0, $v0
#-----------------------------------------------
blt  $19, $20, end   branch-on-less-than
é
slt  $1, $19, $20    # $1 ←1 se ($19 < $20)
bne  $1, $0, end     # salta se ($19 < $20)
```