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

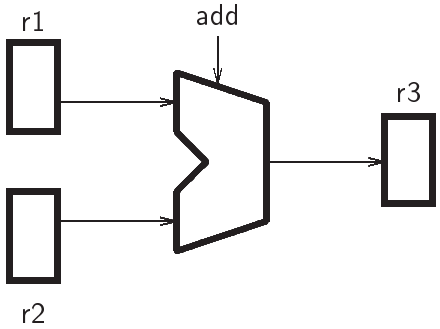
Princípio 4: o caso comum deve ser o mais rápido

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

Ciclo de operação de um processador

`add r3,r1,r2 # r3 ← r1+r2`



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

Linguagem de máquina

- Extremamente simples (simplória?)
- poucos tipos de dados: byte, meia-palavra, palavra, float, double
- dois conjuntos de variáveis: 32 registradores e vetor de bytes

```
/* 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
```


Linguagem de máquina

- 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:
signed (com-sinal)
unsigned (ignora detecção de overflow).

Operações com endereços são sempre sem-sinal
(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 de 16 bits $\rightsquigarrow$ número de 32 bits

Qual a diferença entre núm de operandos no MIPS e no Mico?

Variáveis em memória

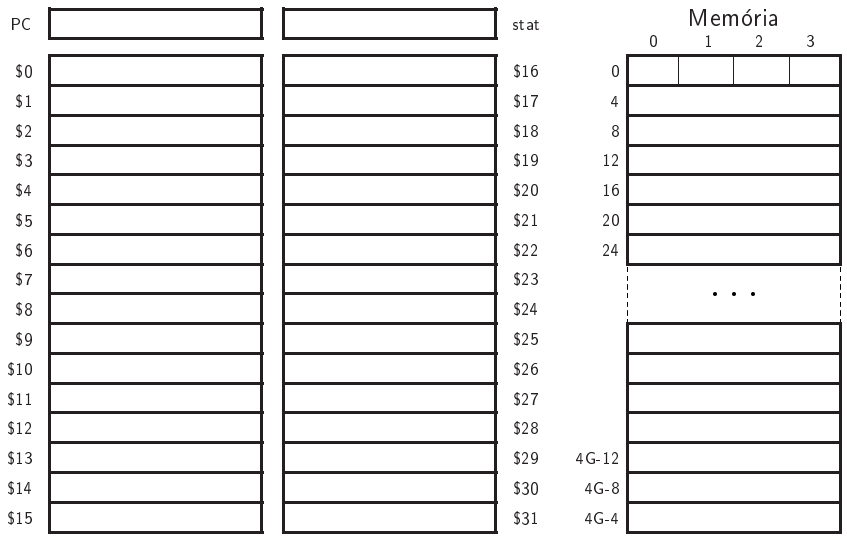
Programas usam mais variáveis que os 32 registradores!
Variáveis, vetores, etc são alocados em memória

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)

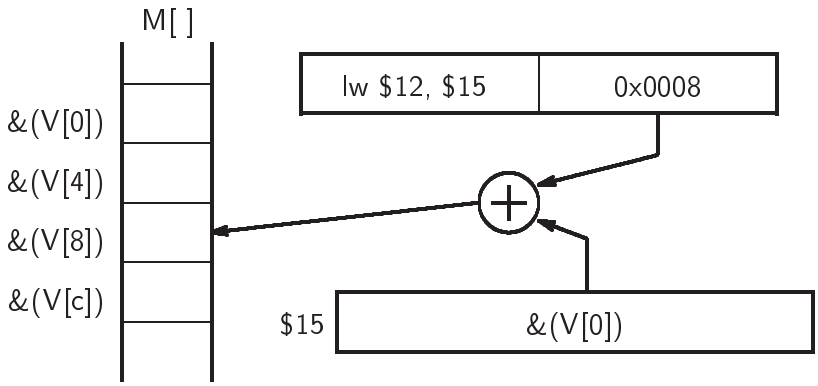
```
# LOAD WORD: end_efetivo = desloc + rIndice
  lw rd, desloc(rIndice)

# 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
```


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;
   int y;
   int z;
   int w;
} aType;
```

```
...
# compil aloca V em 0x800000
aType V[16];
...
```

```
   # 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)
```

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)}
```


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)`

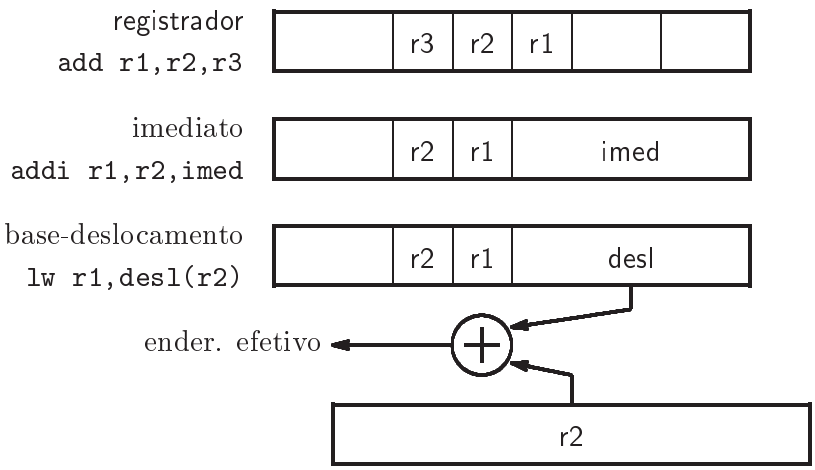
Endereçamento com Imediatos

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
```

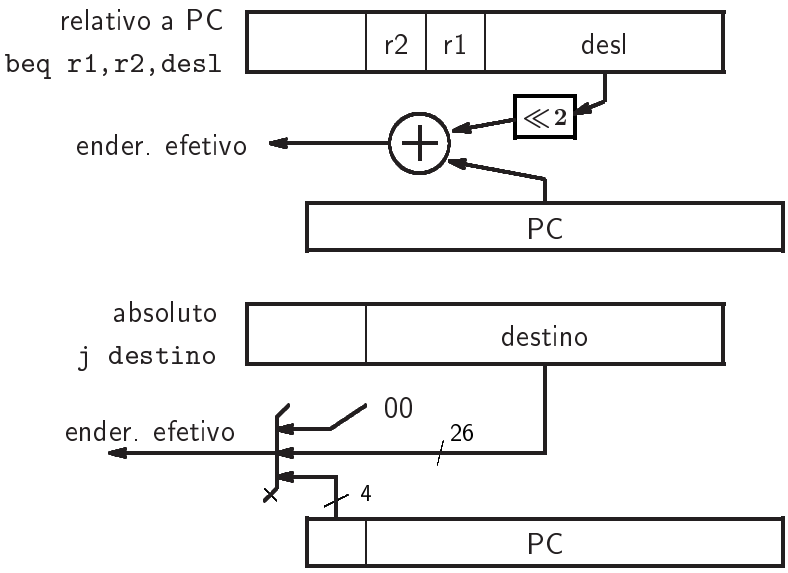

MdE: registrador, imediato, base-deslocamento



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:
```

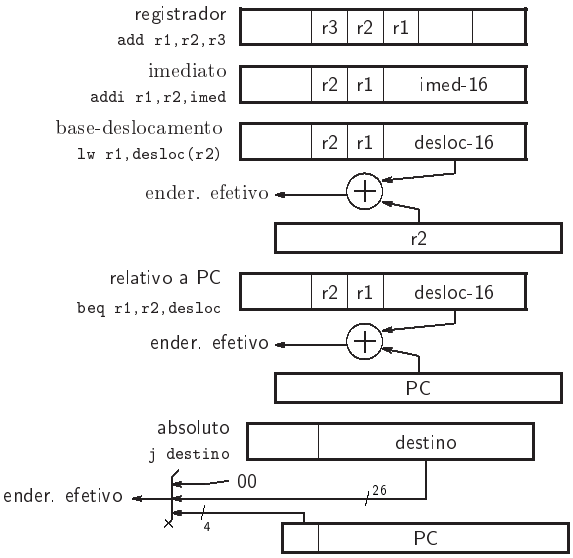

Modos de Endereçamento

- **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

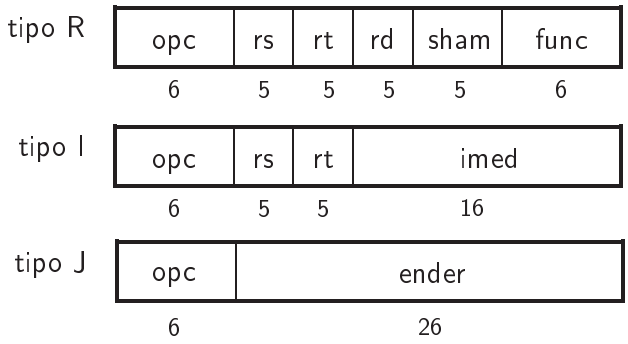
* **Princípio 1**: simplicidade favorece regularidade
* **Princípio 3**: um bom projeto demanda compromissos
* **Princípio 4**: o caso comum deve ser o mais rápido

Quais são os casos comuns?
Quais são os compromissos?

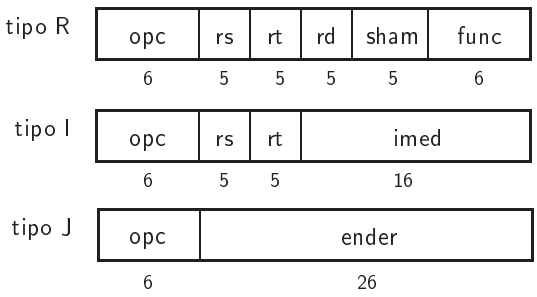
Modos de Endereçamento



Codificação das instruções



Codificação das instruções

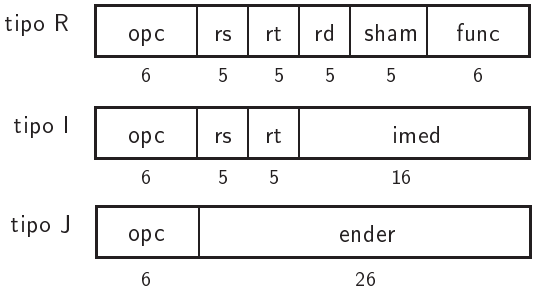


- **Princípio 1**: simplicidade favorece regularidade
- **Princípio 3**: um bom projeto demanda compromissos
- **Princípio 4**: o caso comum deve ser o mais rápido

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)
```