Cui prodest?

Actio?

repeat?

quo vadis?

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

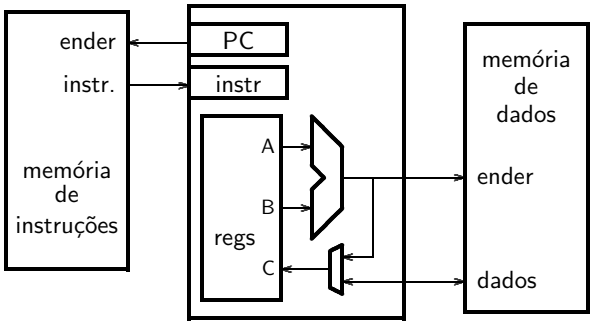
Princípio 4: o caso comum deve ser o mais rápido

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

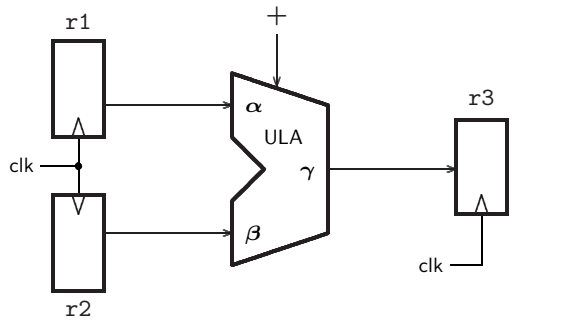


Processador:

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] $\leftarrow \ldots$

Fases de execução de uma instrução (cont.)

```
add r3,r1,r2 # r3 ← r1+r2
```



decodifica instrução
acessa registradores
com operandos

executa

grava resultado

Linguagem de montagem

- Extremamente simples (progr. montador em $\approx$ 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

Linguagem de montagem – sintaxe

Uma instrução por linha,

label: denota endereço da linha indicada (opcional, note o '**:**'),

comentário vai do '#' ou ';' até o fim da linha (opcional).

```
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
                           #  ←  denota atribuição
```

Linguagem de montagem (cont.)

```
/* programa C */     # equivalente 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
```

Programa montador (*assembler*) traduz "linguagem de montagem" (*assembly language*) 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: r0 a r31 (montador chama $0 a $31)

Último exemplo com registradores: rN identifica registrador N

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

Por convenção

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") $\rightsquigarrow$ *overflow* causa exceção
unsigned ("sem-sinal") $\rightsquigarrow$ ignora detecção de *overflow*

Operações com endereços são sempre sem-sinal:
`addu r1, r2, r3` porque todos os 32 bits compõem o endereco: `0xffff ffff` $= -1_{10}$ 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, cnst   # r1 ← r2 + extSinal(cnst)

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

ori r1, r2, cnst    # r1 ← r2 or {0^16 & cnst(15..0)}
                    # constantes lógicas não têm sinal
```

Por que estender o sinal?
Para transformar constante de 16 bits em número de 32 bits:
0x4000 $\rightsquigarrow$ 0x0000.4000 $4 = 0100_2$
0x8000 $\rightsquigarrow$ 0xffff.8000 $8 = 1000_2$

Instruções de Lógica e Aritmética (cont.)

Como obter constantes em 32 bits? & = concat do VHDL
`addi` e `ori` tem operandos de 16 bits

```
lui r1, cnst        # r1 ← {cnst(15..0) & x0000}
ori r2, r1, cnst    # r2 ← r1 or {x0000 & cnst(15..0)}

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 pseudoinstrução
                    #   que substitui lui ; ori
```

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

Variáveis em memória

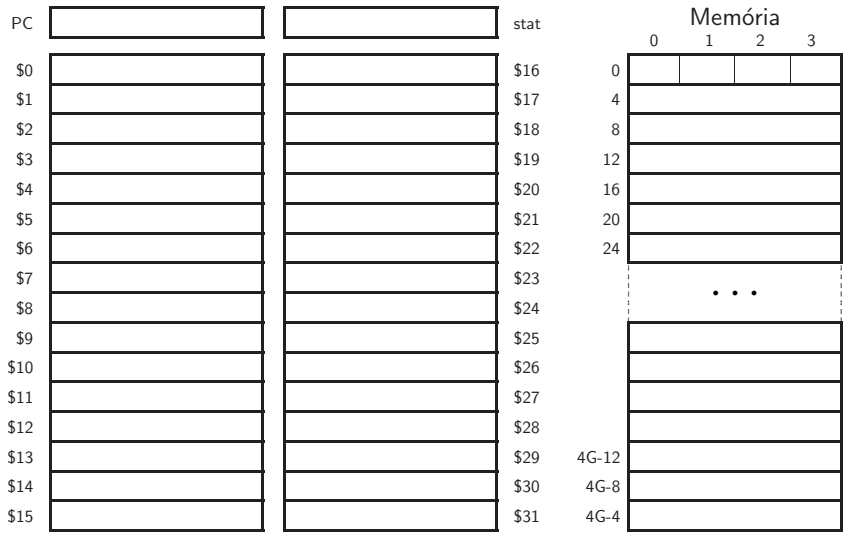
Programas usam mais variáveis que os 32 registradores!
Variáveis, vetores, etc são alocados em memória

Operações com elementos necessitam da
carga dos registradores antes das operações

Memória é um vetor: M[2^{32}] bytes
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

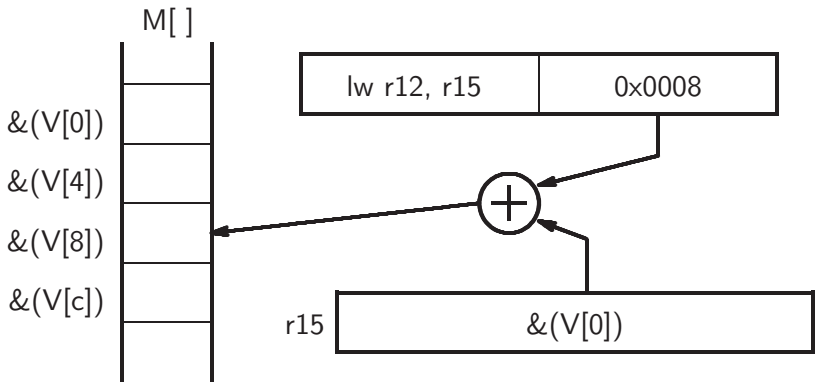


Movimentação de dados entre CPU e memória (i)

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

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

  lw r8, 8(r15)          # r8 ← M[ 8 + r15 ]

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

Programador é responsável por gerenciar o acesso a <u>todas</u> 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

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

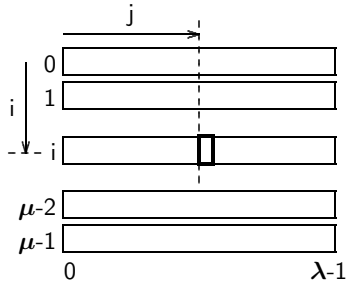
ender	20	21	22	23	24	25	26	27
char	c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
short	s[0]	s[0]	s[1]	s[1]	s[2]	s[2]	s[3]	s[3]
int	i[0]	i[0]	i[0]	i[0]	i[1]	i[1]	i[1]	i[1]

Matrizes em C

uma matriz é alocada em memória como vetor de vetores

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

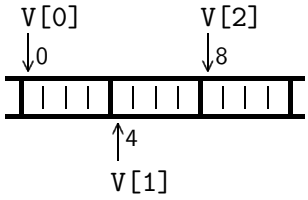
para elementos de tipo τ, linhas com λ colunas e μ linhas



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;
```

Estruturas de Dados em C – structs

Registros (structs) agregam informação relacionada:

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

Componente do registro é selecionado com o operador '.'

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

Estruturas de Dados em C (cont.)

Registros podem ser usados para definir novos tipos:

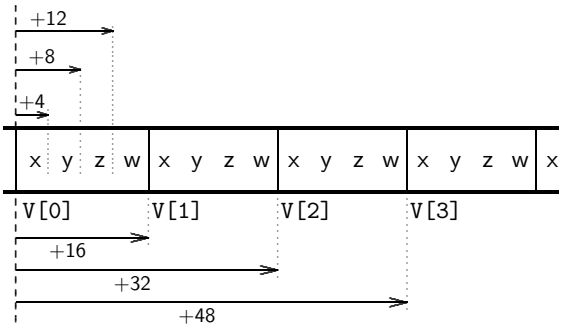
o typedef declara o registro aluno como sendo o novo tipo alunoType, e este tipo pode ser usado na declaração do vetor ufpr[60000].

```
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 (v)

Exemplo: acesso à estrutura com 4 elementos

```
typedef struct A {
   int x;
   int y;
   int z;
   int w;
} aType;
```

```
...
// compil aloca V em 0x0080.0000
aType V[16];
...
```

endereços podem ser calculados em tempo de compilação

```
   # 3 elmtos * 4 pals/elmto * 4 bytes/pal = 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)
```

Movimentação de dados entre CPU e memória (vi)

Exemplo: acesso à estrutura com 4 elementos

```
typedef struct A {
   int x; int y; int z; int w;
} aType;
```

```
# i elmtos * 4 pals/elmto * 4 bytes/pal → i*16
                  la r1, 0x00800030
                  endereços DEVEM ser computados
                  em tempo de execução
m = V[i].y;       sll t0, ri, 4    # (i * 16)
                  add t1, t0, r1   # V + i*16
                  lw r8,  4(t1)    # r8 ← V[i].y
n = V[i].w;       lw r9, 12(t1)    # r9 ← V[i].w
V[i].x = m+n;     add r5, r8, r9
                  sw r5,  0(t1)    # V[i].x ← r5
```

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)
```

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];
```