Organização de Computadores I
DCC006

Aula 3 – Linguagem de Máquina (Instruções)

Prof. Omar Paranaiba Vilela Neto

Instruções:

- **Linguagem de Máquina**
- **Mais *primitiva* que linguagens de alto nível**
  i.e., controle de fluxo não sofisticado
- **Muito restritiva**
  ex. MIPS Instruções Aritméticas

- **Nós trabalharemos com a arquitetura do conjunto de instruções do MIPS**
  - **similar a outras arquiteturas desenvolvidas após 1980's**
  - **Mais de 100 milhões de processadores MIPS fabricados em 2009**
  - **usado pela NEC, Nintendo, Silicon Graphics, Sony**

objetivos do projeto: maximizar o desempenho e minimizar custo e tempo de projeto

MIPS - Aritmética

- Todas instruções tem 3 operandos
- A ordem dos operandos é fixa (destino primeiro)

Exemplo:

Código C: `A = B + C`

Código MIPS: `add $s0, $s1, $s2`

(associação com variáveis pelo compilador)

MIPS - Aritmética

- **Principio de projeto:** simplicidade favorece a regularidade.

- **Código C:**
```
A = B + C + D;
E = F - A;
```

- **Código MIPS:**
```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

- Operandos devem ser registradores, só existem 32 registradores
  - Tamanho dos registradores: 32 bit. (MIPS-32)

- **Principio de Projeto:** Menor é mais rápido.
  - Razão para poucos registradores

MIPS - Aritmética

- **Operandos de instruções Aritméticas devem ser registradores,**
  - **— só 32 registradores existem**
- **Compilador associa variáveis com registradores**

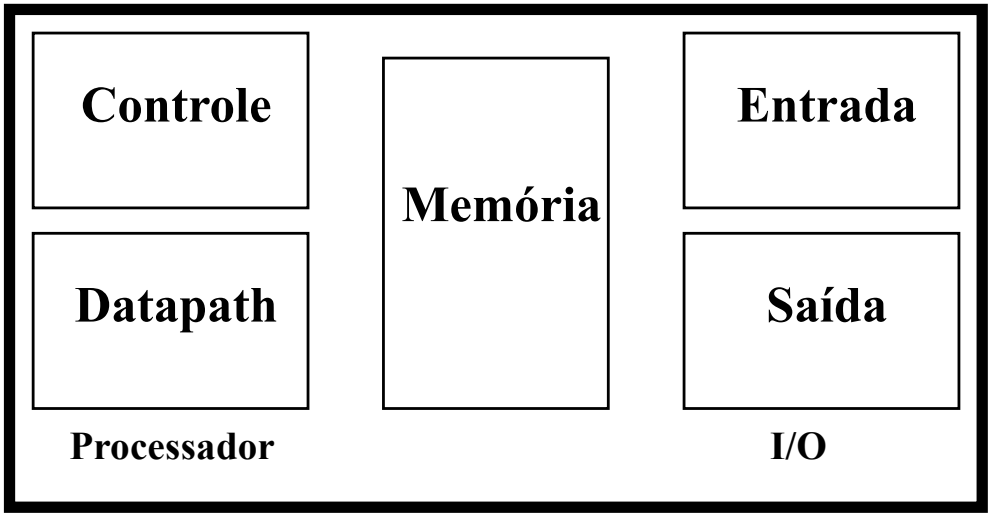
Registradores

Name	Register number	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Como fazer quando um programa tem muitas variáveis?

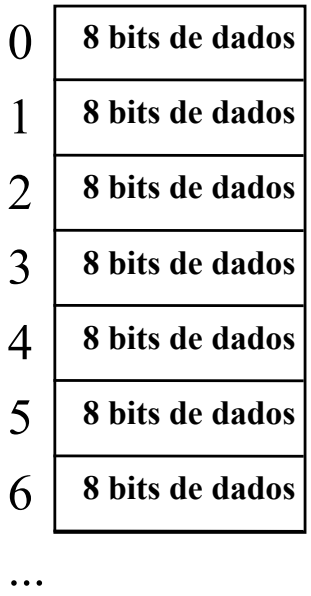
Registradores X Memória

Por quê não usamos a Memória?



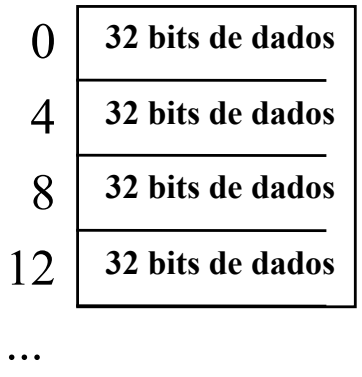
Organização da Memória

- Visto como uma matriz unidimensional, com um endereço.
- Um endereço de memória é um índice em uma matriz
- endereçamento de Byte aponta para um byte da memória.



Organização da Memória

- Bytes são pequenos, mas vários dados usam "words" (palavra)
- Para o MIPS, uma palavra tem 32 bits ou 4 bytes.



Registradores retém 32 bits de dados

- 2^{32} bytes com endereço de byte de 0 to 2^{32}-1
- 2^{30} palavras com endereço de byte 0, 4, 8, ... 2^{32}-4

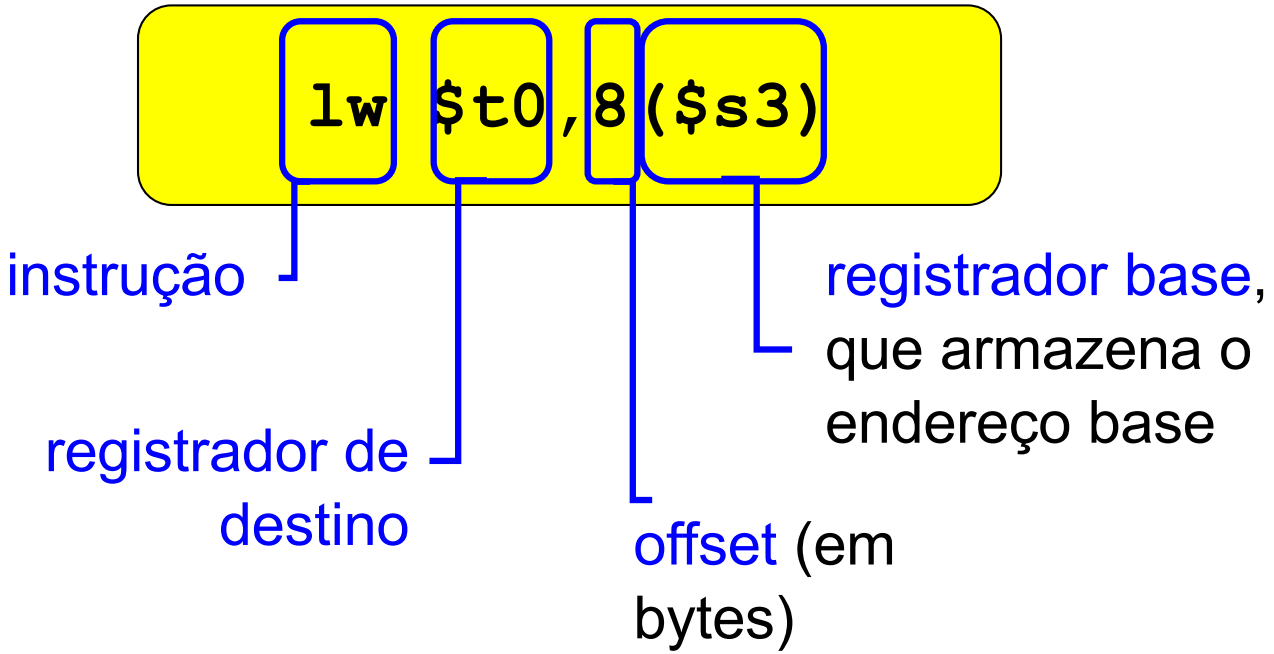
Instruções – Acesso a Memória

- **Instruções load e store**
- **Load – Carrega conteúdo da memória para o registrador**
  - **lw $t0, 32($s3)**
- **Store – Copia conteúdo do registrador na memória**
  - **sw $t0, 32($s3)**

Instruções – Acesso a Memória

Copiar dados de → para	Instrução
Memória → Registrador	load word (lw)
Registrador → Memória	store word (sw)

Formato:



Instruções

- Instruções load e store
- Exemplo:

Código C: `A[8] = h + A[8];`

Código MIPS:

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

- Store tem destino por último
- Relembre operandos aritméticos são registradores, não memória!

Continuamos a Conhecer o MIPS:

- **MIPS**
  - **— carrega palavras mas endereça bytes**
  - **— aritmética somente em registradores**

- **Instrução** **Resultado**

```
add $s1, $s2, $s3       $s1 = $s2 + $s3
sub $s1, $s2, $s3       $s1 = $s2 - $s3
lw $s1, 100($s2)        $s1 = Memória[$s2+100]
sw $s1, 100($s2)        Memória[$s2+100] = $s1
```

O Primeiro Exemplo

- **Análise o código gerado.**

```
swap(int v[], int k);
{ int temp;
temp = v[k]
v[k] = v[k+1];
v[k+1] = temp;
}
```

```
swap:
muli $2, $5, 4
add $2, $4, $2
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
jr $31
```

Principio de Projeto: **Agilize os casos mais comuns.**

(Lembre a Lei de Amdahl)

Linguagem de Máquina

- **Instruções, como registradores e palavras, são de 32 bits**
  - **Exemplo:** `add $t0, $s1, $s2`
  - **registradores tem números,** `$t0=9, $s1=17, $s2=18`

- **Formato de Instrução:**

op	rs	rt	rd	shamt	funct

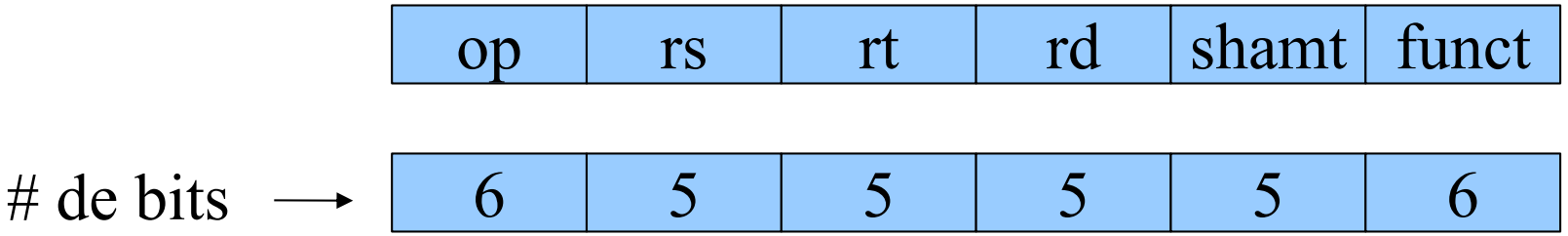
  - **op:** opcode - Operação básica da instrução
  - **rs:** registrador do primeiro operando de origem
  - **rt:** registrador do segundo operando de origem
  - **rd:** registrador do operando de destino
  - **shamt:** quantidade de deslocamento
  - **funct:** função. Seleciona a variante de op.

Linguagem de Máquina

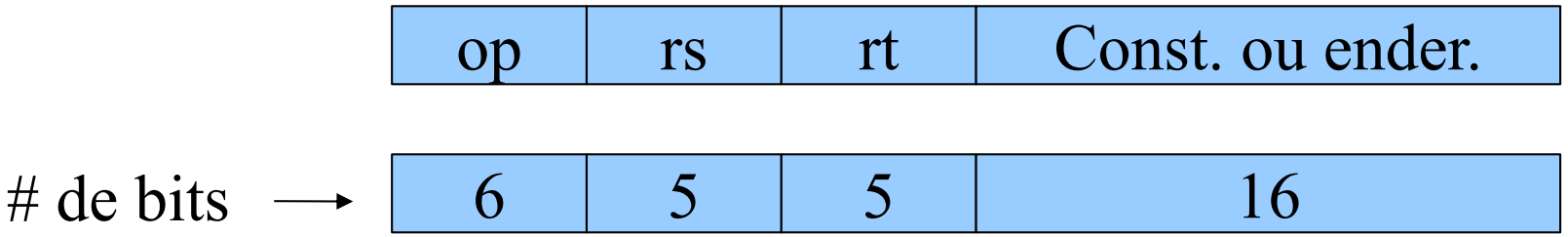
Principio de Projeto: **bom projeto exige bons compromissos**

- Instruções devem ter o mesmo tamanho.

- Tipo R



- Tipo I



Linguagem de Máquina

- Instruções – Código de máquina

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

Linguagem de Máquina

- Exemplo

```
A[300] = h + A[300];
```

is compiled into

```
lw   $t0,1200($t1) # Temporary reg $t0 gets A[300]
add  $t0,$s2,$t0   # Temporary reg $t0 gets h + A[300]
sw   $t0,1200($t1) # Stores h + A[300] back into A[300]
```

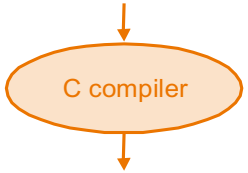
op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Linguagem de Máquina

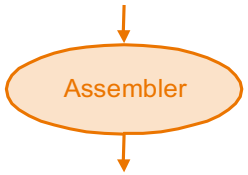
High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



Assembly language program (for MIPS)

```
swap:
     muli $2, $5,4
     add  $2, $4,$2
     lw   $15, 0($2)
     lw   $16, 4($2)
     sw   $16, 0($2)
     sw   $15, 4($2)
     jr   $31
```



Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Agora vocês entendem!

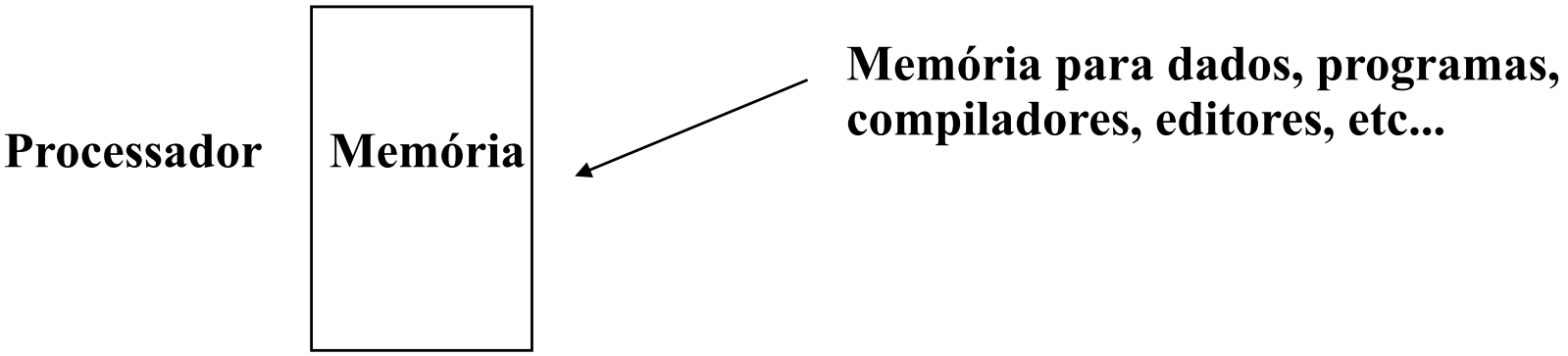
Lógica

- Operações Lógicas

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	$s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori	$s1,$s2,100	$s1 = $s2 | 100	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

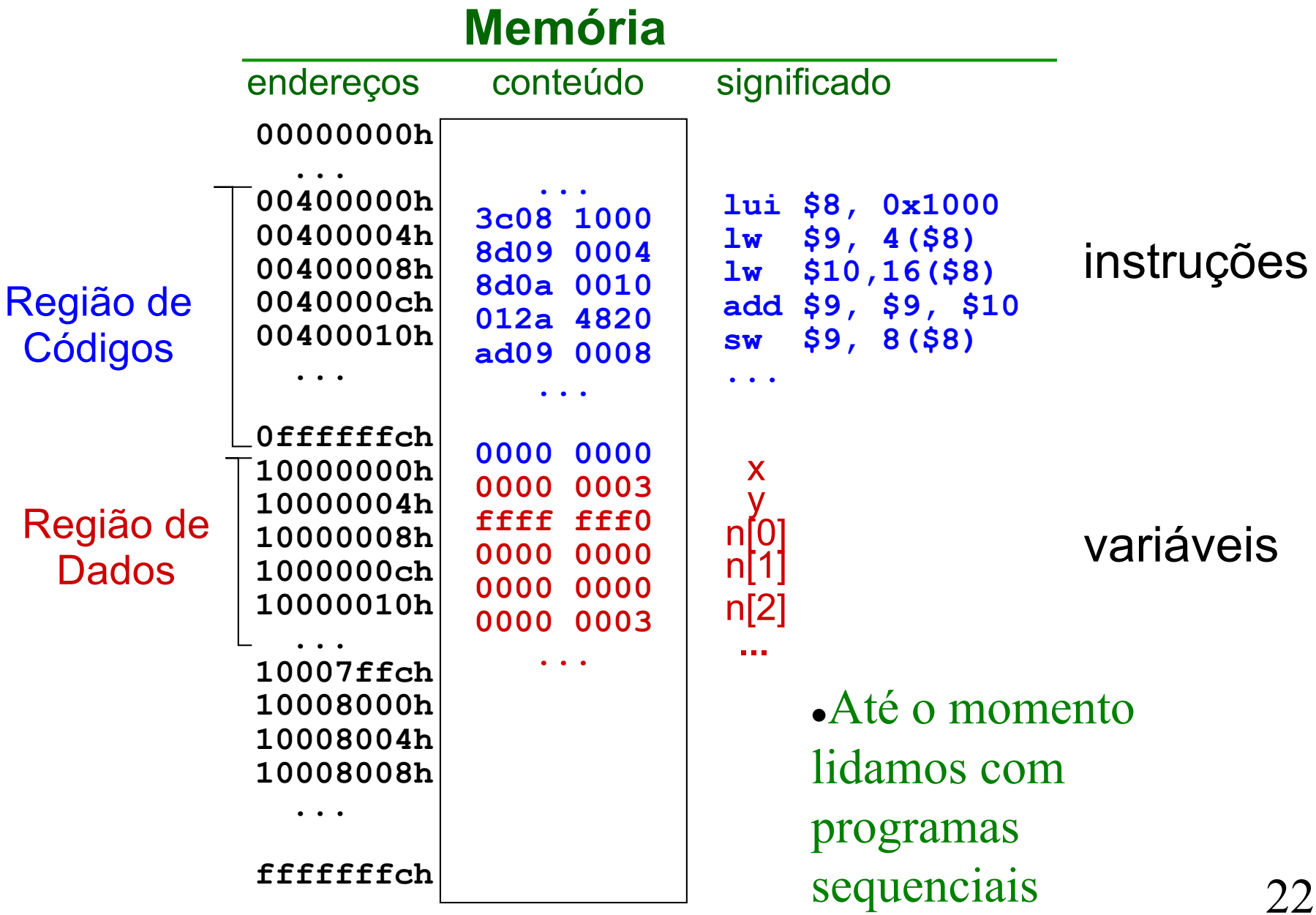
Programa armazenado - Conceito

- Instruções são bits
- Programas são armazenados em memória
  — lido ou escritos da mesma forma que dados



- Fetch (Busca) & Execute Cycle (Ciclo de Execução)
  1. Instruções são buscadas e colocadas em um registrador especial
  2. Registrador de controle indica as ações subsequentes
  3. Executa a instrução e endereço da próxima instrução calculado

Programas Sequenciais



- Até o momento lidamos com programas sequenciais

Instruções de Controle

- Instruções para tomada de decisão
  - Alteram o fluxo de controle do programa
  - Alteram a “próxima” instrução a ser executada

- Instruções de controle:
  - Salto condicional
  - Salto incondicional

Instruções de Controle

- Instruções MIPS para salto condicional:
  - Branch on equal (**`beq`**)
  - Branch on not equal (**`bne`**)
  - Set on less than (**`slt`**)
  - Set on less than immediate (**`slti`**)

- Instruções MIPS para salto incondicional:
  - jump (**`j`**)

Instruções de Controle

- **Branch on not equal (`bne`)**
  - Desvia o programa para <label1> se $t0 != $t1

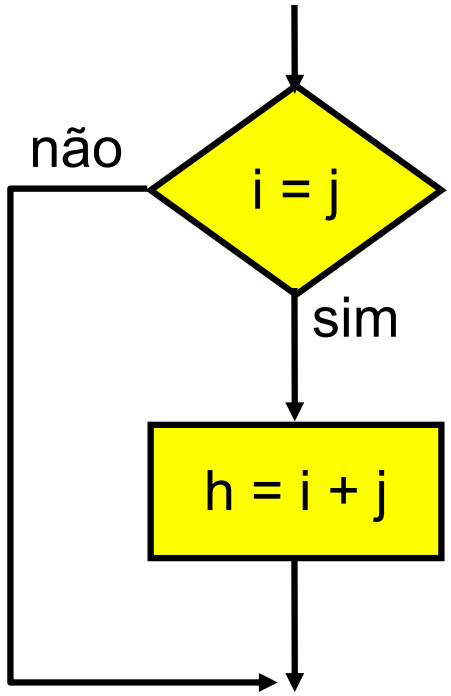
```
bne  $t0, $t1, label1  #if ($t0 != $t1) goto label1
```

- **Branch on equal (`beq`)**
  - Desvia o programa para <label2> se $t0 == $t1

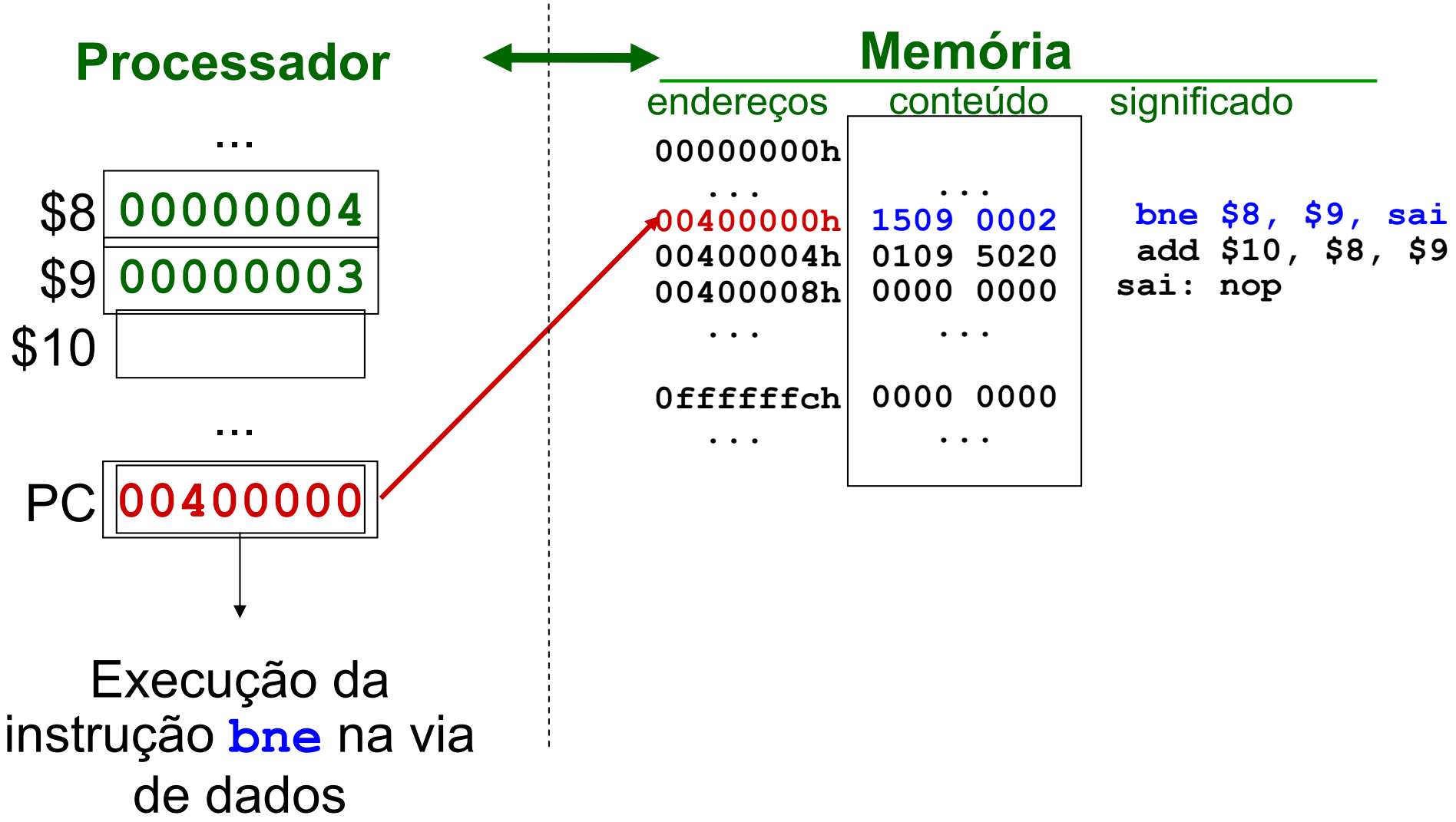
```
beq  $t0, $t1, label2  #if ($t0 == $t1) goto label2
```

Instruções de Controle

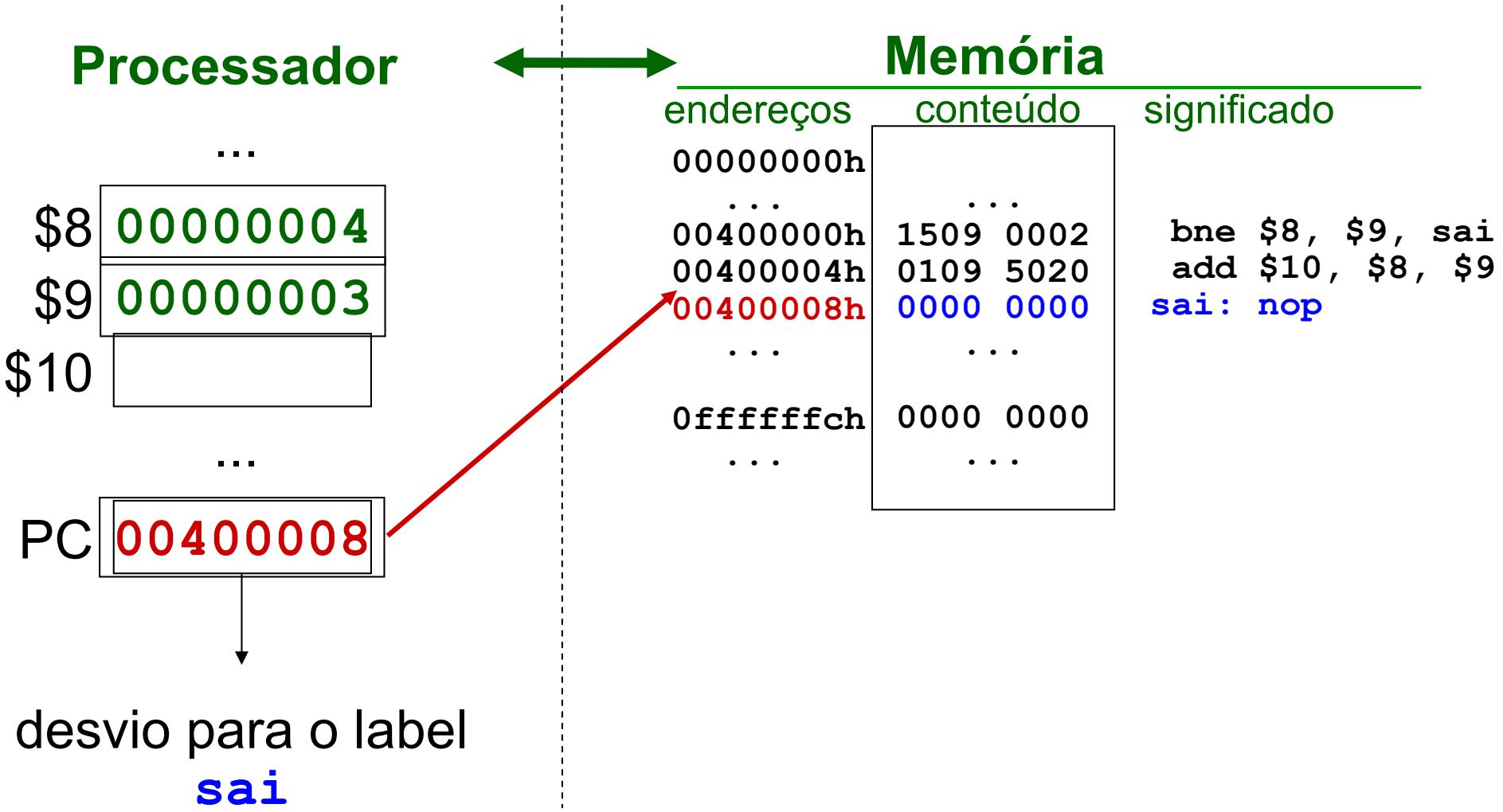
```
      bne $8, $9, sai
      add $10, $8, $9
sai:    nop
```



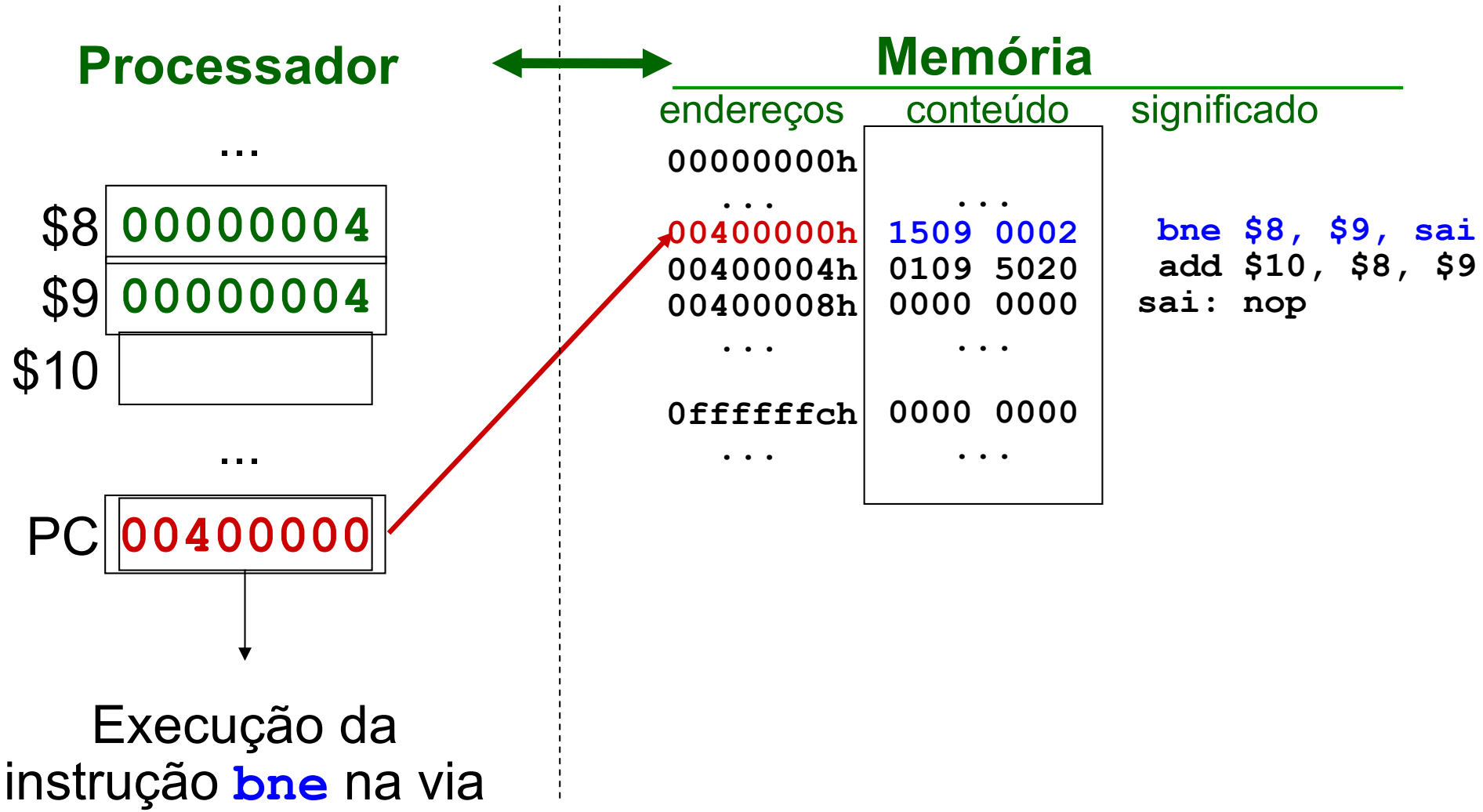
Instruções de Controle



Instruções de Controle

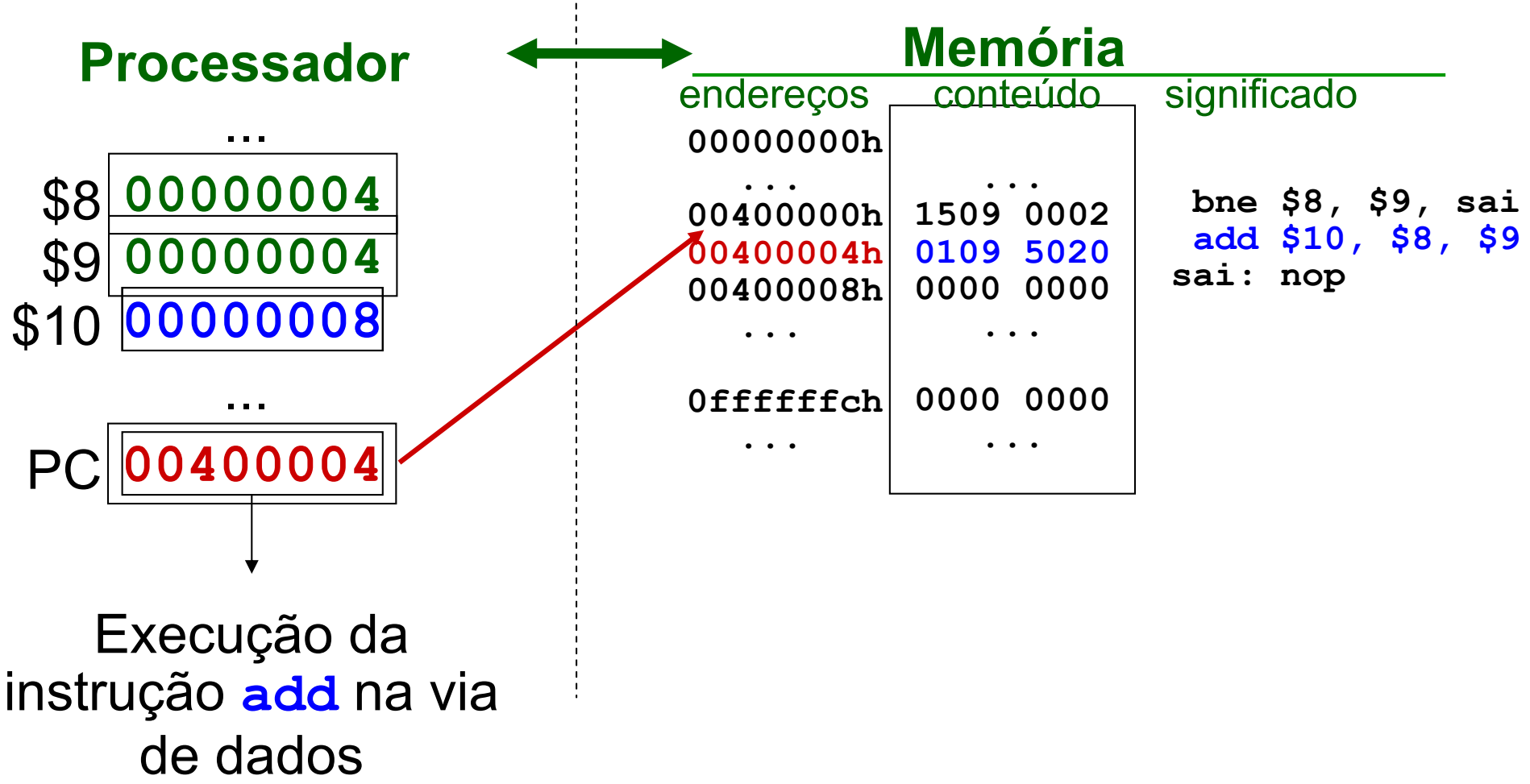


Instruções de Controle



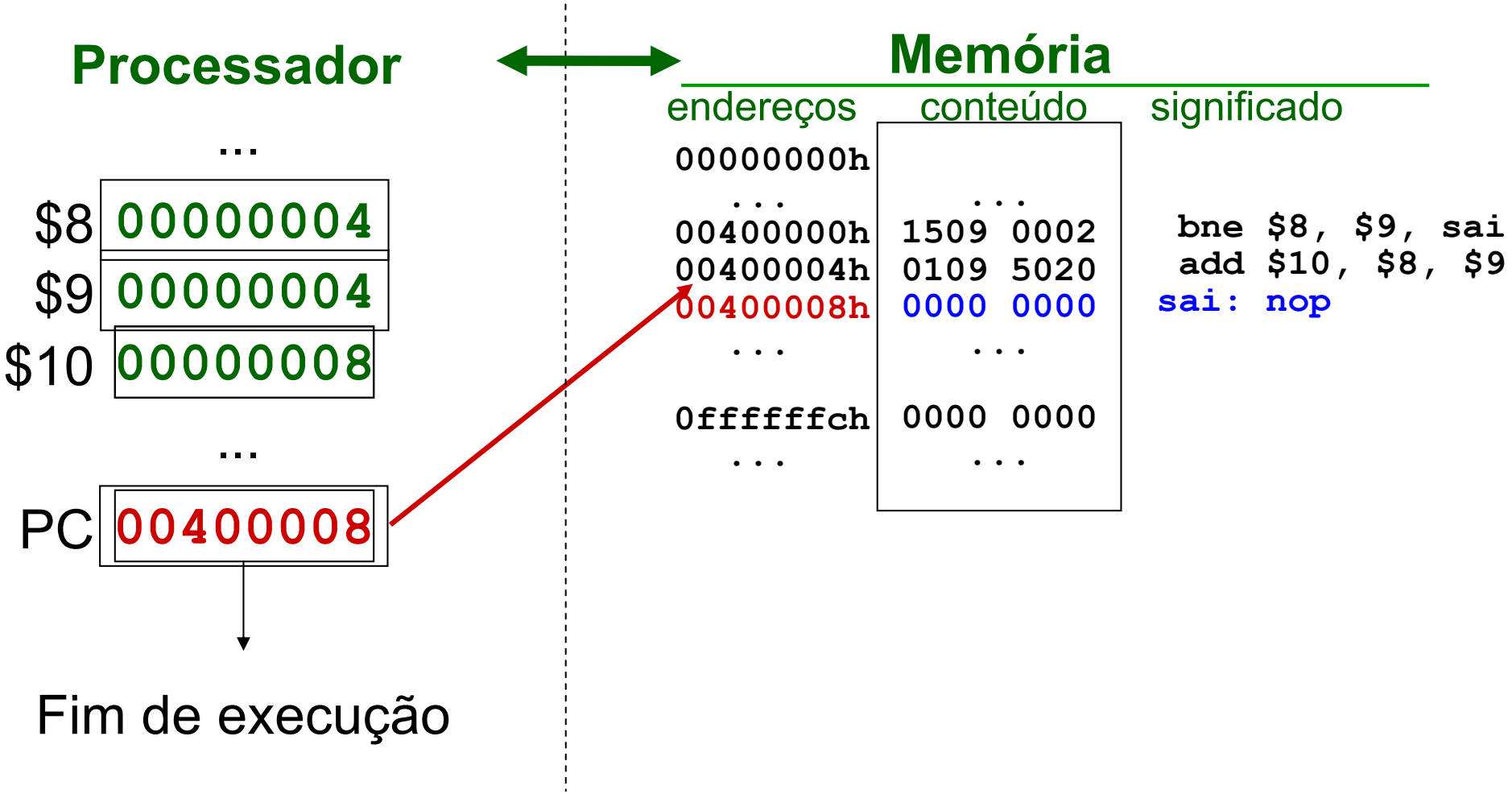
Execução da instrução **bne** na via de dados.

Instruções de Controle



Execução da instrução **add** na via de dados

Instruções de Controle



Instruções de Controle

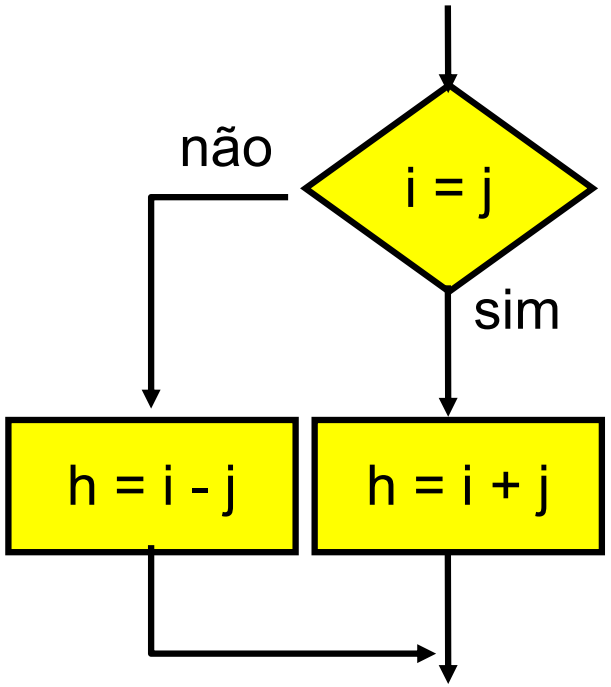
- **Jump (`j`)**
  - Desvio incondicional para um endereço de memória apontado por um label

```
j   label
```

  - Instruções do tipo branch indicam desvio da sequência do programa mediante análise de uma condição
  - Instruções do tipo jump indicam desvio incondicional da sequência do programa

Instruções de Controle

```
      bne  $8, $9, else
      add  $10, $8, $9
      j    sai
else: sub  $10, $8, $9
sai:  nop
```

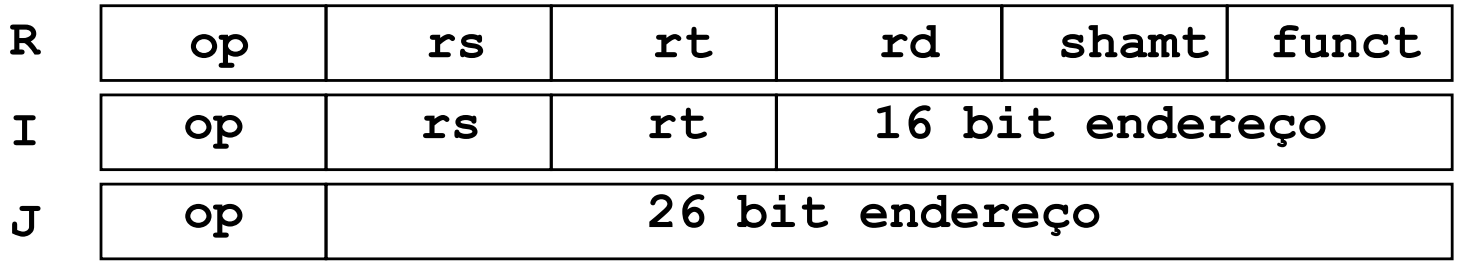


Então:

- **Instrução** **Resultado**

```
add $s1,$s2,$s3   $s1 = $s2 + $s3
sub $s1,$s2,$s3   $s1 = $s2 - $s3
lw $s1,100($s2)   $s1 = Memória[$s2+100]
sw $s1,100($s2)   Memória[$s2+100] = $s1
bne $s4,$s5,L     próxima instr. é Label se $s4 ≠ $s5
beq $s4,$s5,L     próxima instr. é Label se $s4 = $s5
j Label           próxima instr. é Label
```

- **Formatos:**



Controle de Fluxo

- **Nós Temos: beq, bne, como produzir Branch-if-less-than?**
- **Nova Instrução:**

```
if  $s1 < $s2 then
   $t0 = 1
else
   $t0 = 0

slt $t0, $s1, $s2
```

- **Pode usar esta instrução para integrar "`blt $s1, $s2, Label`"**
  - **— Podemos fazer uma instrução geral de controle**
- **Note que o montador precisa de registradores para isso,**
  - **— existe uma política convencionada para uso de registradores**

Controle de Fluxo

- **Set on less than (`slt`)**
  - Compara dois registradores

```
slt   $s1, $s2, $s3    #if ($s2 < $s3) $s1 = 1
                       #else $s1 = 0
```

- **Set on less than immediate (`slti`)**
  - Compara um registrador e uma constante

```
slti  $s1, $s2, 100    #if ($s2 < 100) $s1 = 1
                       #else $s1 = 0
```

Constantes

- **Pequenas constantes são usadas frequentemente (50% dos operandos)**

  A = A + 5;
  B = B + 1;
  C = C - 18;

- **Soluções?**
  - **Colocar constantes típicas em memória e carrega-las.**
  - **criar hard-wired registros (como $zero) para constantes como um**

- **MIPS - Instruções:**

```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```

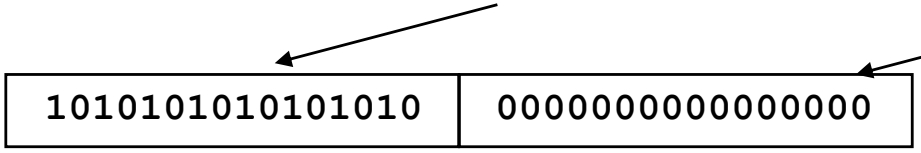
- **Como nós fazemos este trabalho?**

Qual o tamanho das constantes?

- Nós gostaríamos de ser capazes de carregar uma constante de 32 bit em um registrador
- Deve ser usado duas novas instruções "load upper immediate"

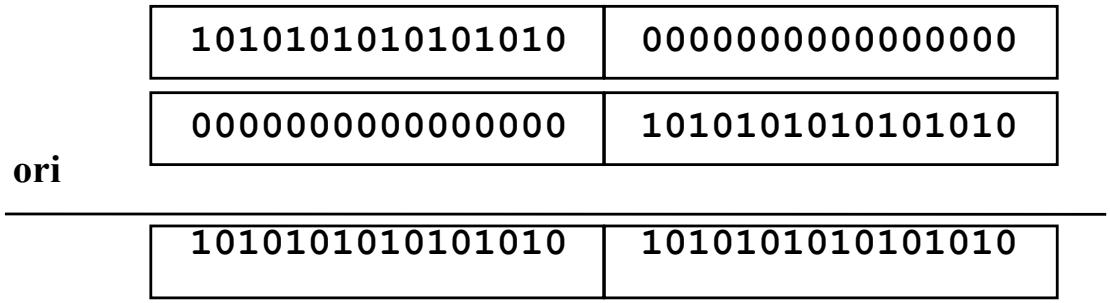
```
lui $t0, 1010101010101010
```

Finalizado com zeros



- Então buscamos os bits de mais baixa ordem da direita.

```
ori $t0, $t0, 1010101010101010
```



Instruções de Ponto Flutuante no MIPS

MIPS floating-point operands

Name	Example	Comments
32 floating-point registers	`$f0, $f1, $f2, . . . , $f31`	MIPS floating-point registers are used in pairs for double precision numbers.
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, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	`add.s $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (single precision)
	FP subtract single	`sub.s $f2,$f4,$f6`	`$f2 = $f4 - $f6`	FP sub (single precision)
	FP multiply single	`mul.s $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP multiply (single precision)
	FP divide single	`div.s $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (single precision)
	FP add double	`add.d $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (double precision)
	FP subtract double	`sub.d $f2,$f4,$f6`	`$f2 = $f4 - $f6`	FP sub (double precision)
	FP multiply double	`mul.d $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP multiply (double precision)
	FP divide double	`div.d $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (double precision)
Data transfer	load word copr. 1	`lwc1 $f1,100($s2)`	`$f1 = Memory[$s2 + 100]`	32-bit data to FP register
	store word copr. 1	`swc1 $f1,100($s2)`	`Memory[$s2 + 100] = $f1`	32-bit data to memory
Condi-tional branch	branch on FP true	`bc1t 25`	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	`bc1f 25`	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	`c.lt.s $f2,$f4`	if (`$f2 < $f4`) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	`c.lt.d $f2,$f4`	if (`$f2 < $f4`) cond = 1; else cond = 0	FP compare less than double precision

Exemplo

```
Float f2c (float fahr)
    {
       return((5.0/9.0) * (fahr - 32.0))
    }
```

```
f2c:
    lwc1 $f16, const5($gp)       # $f16 = 5.0
    lwc1 $f18, const9($gp)       # $f18 = 9.0
    div.s $f16, $f16, $f18
    lwc1 $f18, const32($gp)      # $f18 = 32.0
    sub.s $f18, $f12, $f18
    mul.s $f0, $f16, $f18
    jr $ra
```

Linguagem Assembly X Linguagem de Máquina

- **Assembly provê representação simbólica conveniente**
  - **mais fácil que escrever números**

- **Linguagem de Máquina é a mais real**

- **Assembly pode prover 'pseudoinstruções'**
  - **e.g., “move $t0, $t1” existe somente em Assembly**
  - **Pode ser implementado usando “add $t0,$t1,$zero”**
- **Quando consideramos desempenho você poderia contar como instrução real**

Outras miscelâneas

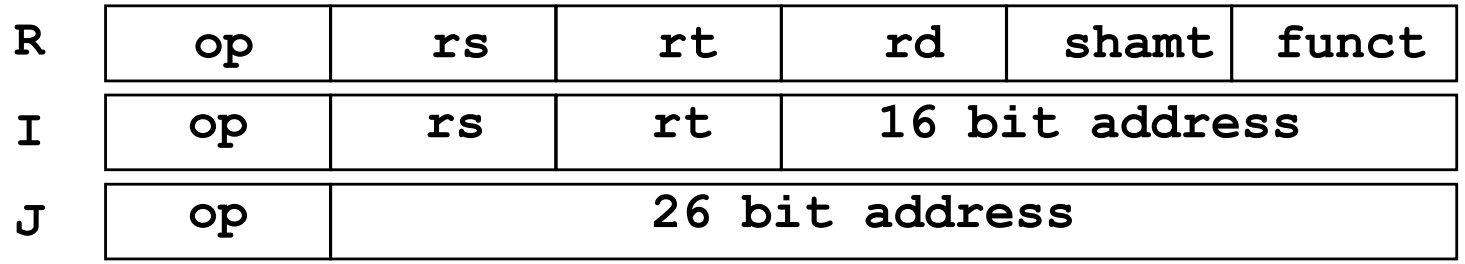
- **Coisas que não foram cobertas ainda**
  - **suporte para procedimentos**
  - **linkers, loaders, layout de memória**
  - **stacks, frames, recursividade**
  - **manipulação de strings e pointers**
  - **interrupções e exceções**
  - **chamadas do sistema e convenções**

- **Vários destes itens abordaremos mais tarde**

- **Nós focaremos na arquitetura**
  - **linguagem assembly do MIPS e código de máquina**
  - **como integramos um processador para executar estas instruções.**

Overview do MIPS

- Instruções simples todas com 32 bits
- estruturada,
- somente três formatos de instrução



- facilitar o compilador para bom desempenho
  — Quais são os objetivos dos compiladores?
- Ajudar o compilador onde nós podemos

Resumindo:

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$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 for 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 words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
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 less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
Uncondi-tional 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