

# INTRODUÇÃO AO CURSO

Anotações do material suplementar (apresentações PPT) ao Livro do Hennesy e Patterson e do material do Prof. Celso Alberto Saibel Santos (DI/CT).

# PROGRAMA ARMAZENADO LINGUAGEM DE MONTAGEM INSTRUÇÕES MIPS

High-level language program (in C)	<pre> swap(int v[], int k) {int temp;   temp = v[k];   v[k] = v[k+1];   v[k+1] = temp; } </pre>
---	---

Compiler

Assembly language program (for MIPS)	<pre> swap:     multi \$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 </pre>
---	---

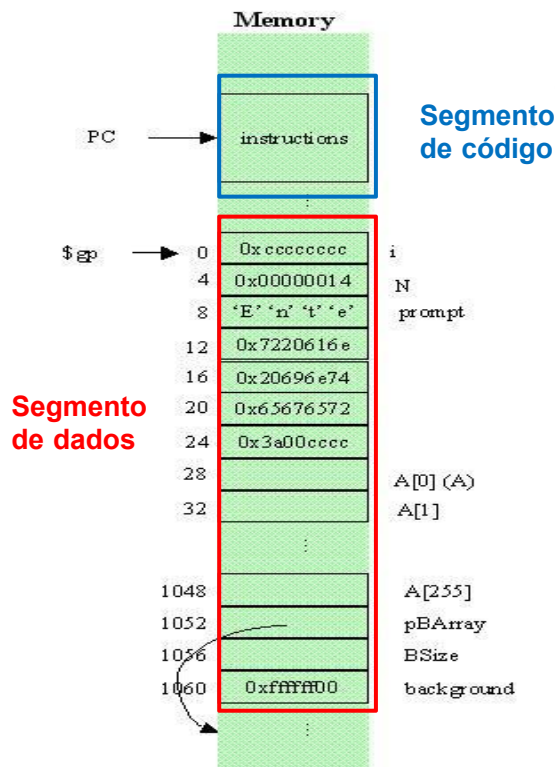
Assembler

Binary machine language program (for MIPS)	<pre> 000000001010001000000000100011000 00000000100000100001000000100001 10001101111000100000000000000000 10001110000100100000000000000100 10101110000100100000000000000000 10101101111000100000000000000100 0000001111100000000000000001000 </pre>
---	---

# Considere o programa em C

```
// none of these allocate any storage
#define MAX_SIZE 256
#define IF(a)    if (a) {
#define ENDIF    }
typedef struct {
    unsigned char red;        // 'unsigned char' is an unsigned, 8-bit int
    unsigned char green;
    unsigned char blue;
    unsigned char alpha;
} RGBa;
// these allocate storage
int    i;
int    N = 20;
char    prompt[] = "Enter an integer:";
int    A[MAX_SIZE];
int*    pBArray;
int    BSize;
RGBa    background = {0xff, 0xff, 0xff, 0x0};
```

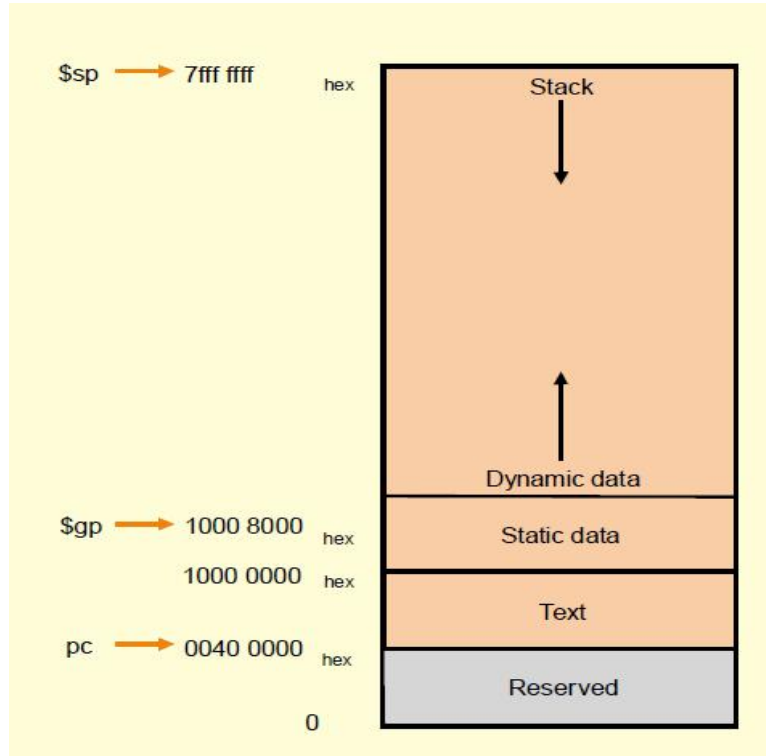
# Layout da memória



## Notes:

- The OS sets `$gp` before starting our program(s)
- Labels on the left are offsets relative to `$gp`
- Variable `i` is assumed uninitialized, so the memory contents there are undefined (random)
- The machine is assumed to be operating in “big endian” mode, so bytes within a word are ordered left-to-right.
- Array `A` must be word aligned (so there is some padding inserted before it)
- We assume `pBArray` has been initialized somehow...
- Note that while this layout follows the order of declarations, there is no particular reason why we had to do that – most any layout is valid. Different C compilers will do different things.

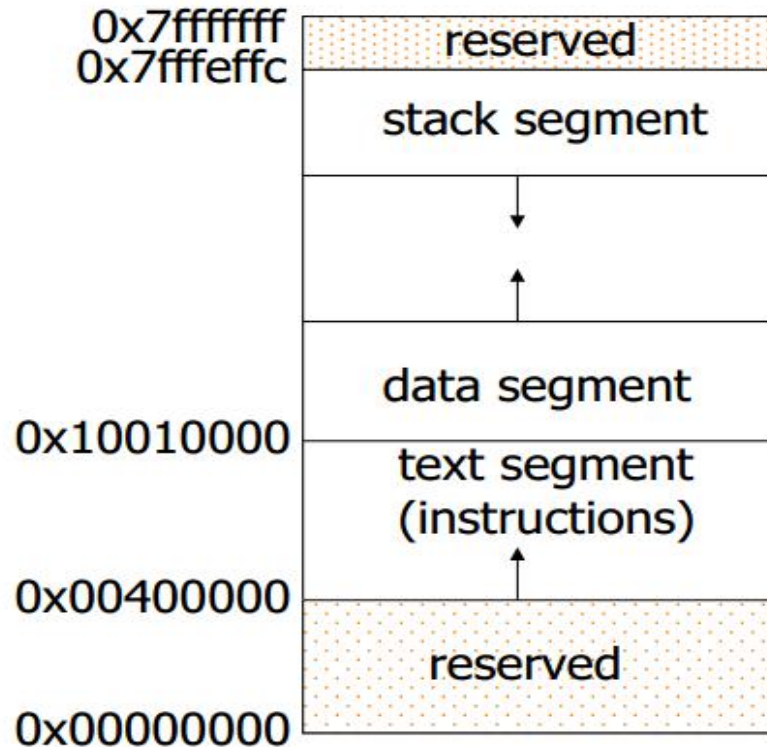
# Carregando um programa executável



Para carregar um executável, o SO faz o seguinte:

1. Lê o cabeçalho do arquivo executável e determina o tamanho dos segmentos de texto (código) e dados
2. Cria um espaço de endereçamento grande o suficiente para texto (código) e dados
3. Copia instruções e dados do arquivo executável para a memória
4. Copia os parâmetros (seu houver) do programa principal para a pilha
5. Inicializa os registradores e 'seta' o ponteiro da pilha para a primeira alocação livre
6. Salta para a 'rotina de start-up' que copia os parâmetros nos registradores de argumentos e chama a rotina principal do programa

# No simulador MIPS MARS



# Ao final do processo...

- ❑ O programa foi carregado na memória
- ❑ Dados e instruções (MIPS, em nosso caso) estarão na memória
- ❑ CPU indica endereços de instruções ou de dados a serem lidos
- ❑ Algumas questões intrigantes:

Qual a instrução ou dado busco na memória a cada instante ?

Qual tipo de instrução estou lendo da memória ?

Qual tipo de dado estou lendo/escrevendo na memória ?

Fluxo de execução e busca de instruções ?

Como faço com as operações de entrada/saída, por exemplo, ler uma entrada do teclado e imprimir na tela ?



# EXECUÇÃO DO PROGRAMA ARMAZENADO EM LINGUAGEM DE MÁQUINA MIPS



# Ciclo de Busca & Execução

- Uma CPU deve:
  1. Buscar instruções
  2. Decodificar instruções
  3. Buscar operandos/dados usados nas instruções
  4. Executar instruções (processar dados/operandos)
  5. Escrever o resultado da execução
- Esses passos básicos definem o **ciclo de busca e execução** da arquitetura!
- Outros passos poderiam ainda ser incluídos neste ciclo para cálculos de endereços de operandos, seja na fase de busca, seja na de escrita.

# Código em linguagem C

```
#include <stdio.h>

int
main (int argc, char *argv[])
{
    int i;
    int sum = 0;

    for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

**Trecho de Código para computar e imprimir a soma dos quadrados dos inteiros entre 0 e 100.**

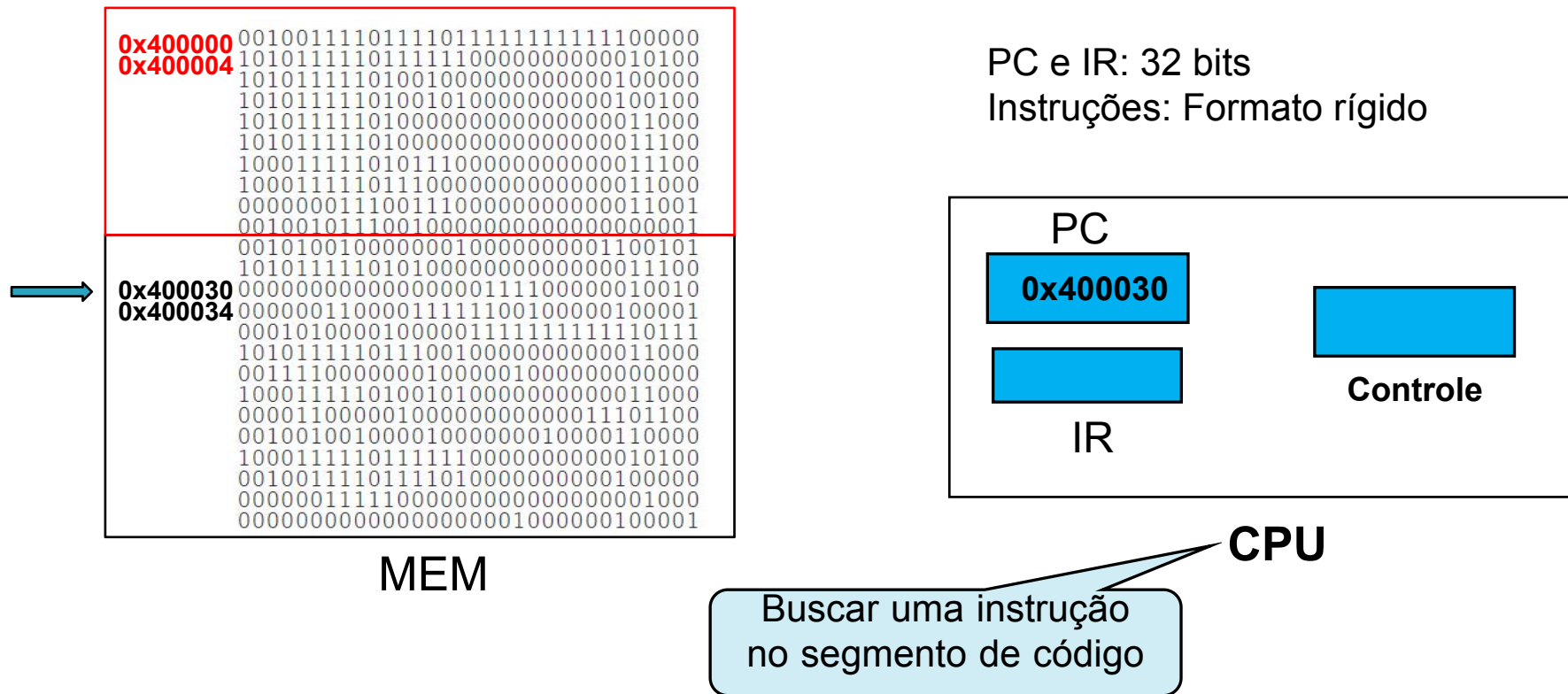
# Código Linguagem de Máquina MIPS

```
0010011110111101111111111111100000
1010111110111111100000000000010100
1010111110100100000000000001000000
101011111010010100000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
000000000000000000011110000010010
00000011000011111100100000100001
0001010000100000111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
000011000001000000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
001001111011110100000000000100000
0000001111100000000000000001000
000000000000000000001000000100001
```

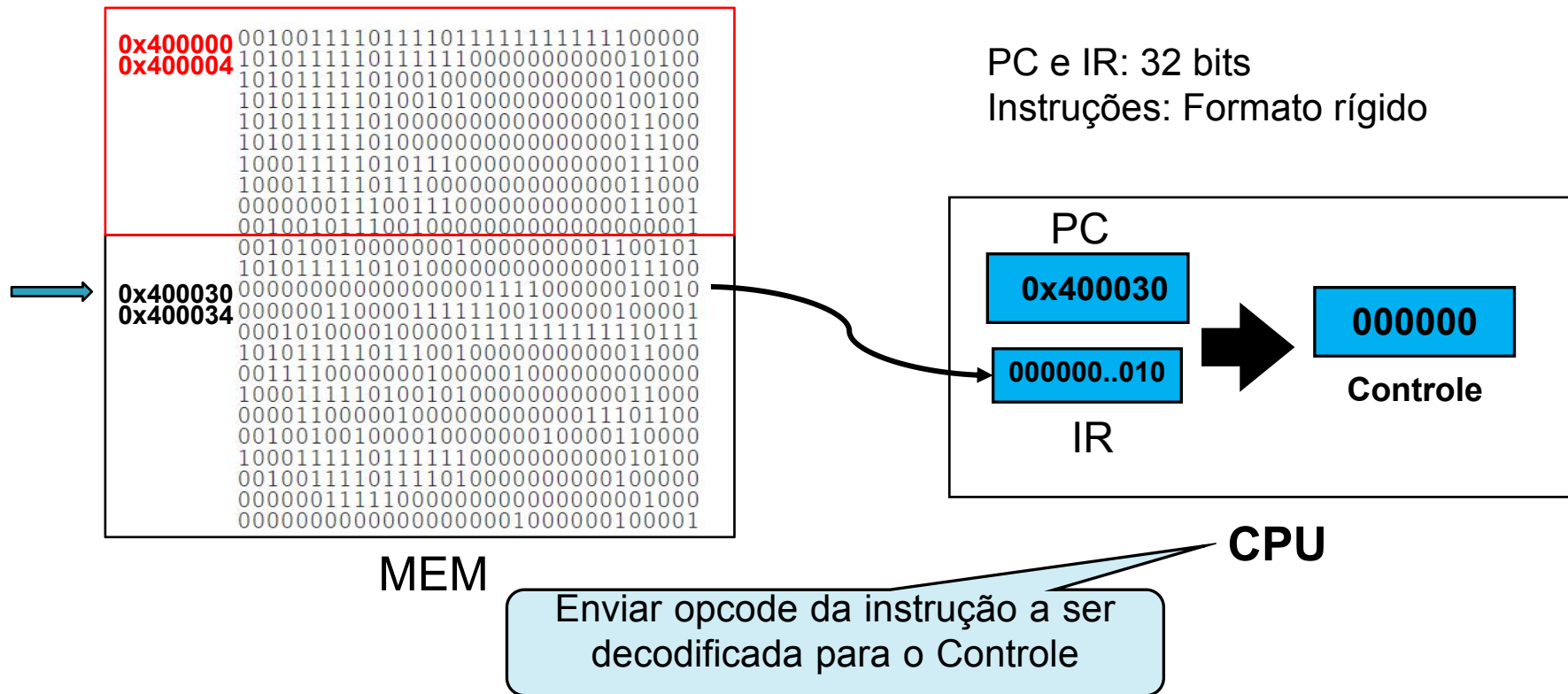
bne  
sw

\$1, \$0, -9  
\$25, 24(\$29)

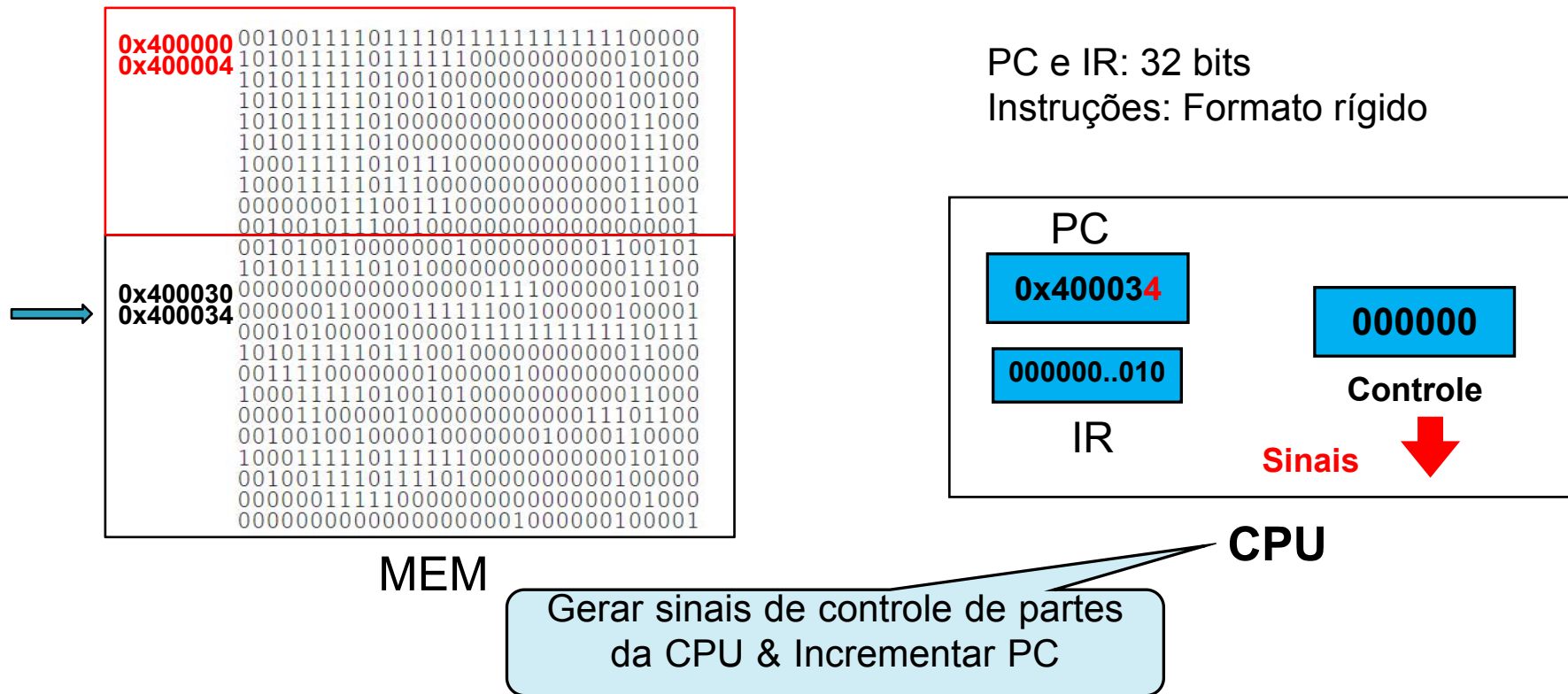
# Ciclo de Busca-Execução



# Ciclo de Busca-Execução

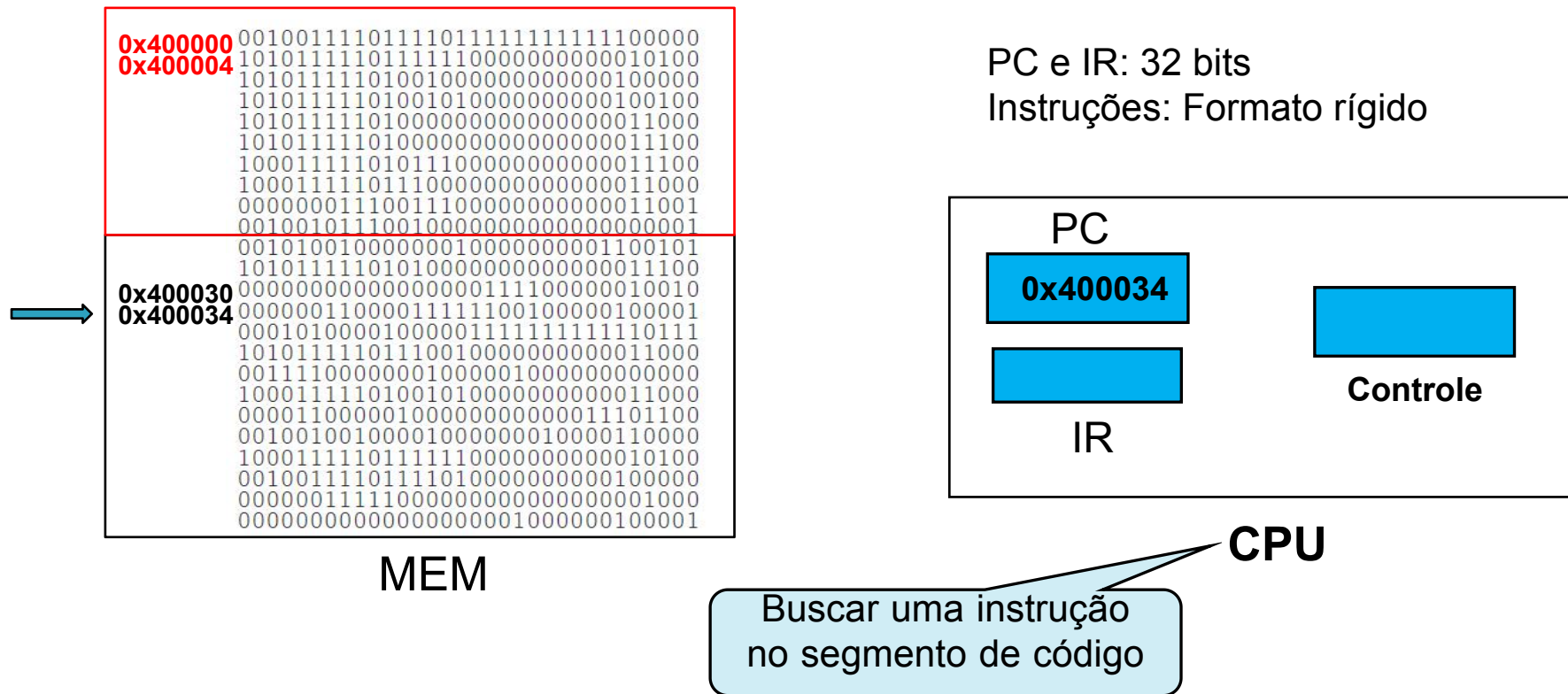


# Ciclo de Busca-Execução





# Ciclo de Busca-Execução





# CÓDIGO DE MÁQUINA: AS INSTRUÇÕES DA LINGUAGEM MIPS

# Instruções

- **Instruções = palavras**      **conjunto de instruções = vocabulário**
- **Programa armazenado = dados + instruções armazenados (binários) na memória**
- Instruções estão em Linguagem de Máquina (*assembly*) e dados seguem padrões
- Nós iremos trabalhar com o instruções **MIPS**

***Projeto MIPS: Maximizar Desempenho e Minimizar Custo, reduzir tempo de design!***

MIPS em processadores reais ?

<http://stackoverflow.com/questions/2635086/mips-processors-are-they-still-in-use-which-other-architecture-should-i-learn>

# Assumiremos que...

- Dados e instruções (0s e 1s) estão na memória principal
- Foram gerados a partir de um processo de compilação e carga na memória
- Alocação de espaço feita durante o processo de carga:
  - Compilador indica quantidade em Bytes para as variáveis
  - Sequência de instruções (dependente do algoritmo)
  - Endereço (relativo ao segmento do código) de variáveis, labels, rotinas, etc.
- Uso frequente de instruções do tipo *LOAD* (*STORE*) para trazer (levar) dados da (para) memória para (dos) os registradores

# Organização da Memória

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data

...

- Memória = enorme vetor unidimensional acessado por endereços
  - Endereço de memória = índice para o vetor
- “Endereçar um palavra”** = achar um índice que aponta para uma palavra da memória de maneira única.

# Acesso à Memória MIPS

- Acesso de “palavras” por seus endereços: Múltiplos de 4 (32 bits = 4 Bytes) no MIPS
- Endereços alinhados:

Depende da arquitetura da CPU (32 ou 64 bits) e do SO

Nem todas as arquiteturas exigem alinhamento:

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data

...

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data
16	32 bits of data
20	32 bits of data
24	32 bits of data

...

→ *E ainda, a ordem dos bytes nas palavras:*

***Little Endian no MARS***

***MIPS -- big ou little endian***

# Organização da Memória MIPS

- O acesso a Bytes pode ser interessante, mas acessar “palavras” de Bytes é mais usual.

- No MIPS, uma palavra possui 32 bits (4 bytes)

Isso traz diversas implicações para a arquitetura

- Registradores armazenam 32 bits
- Instruções ocupam 32 bits
- Operandos ocupam 32 bits

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data

...

- $2^{32}$  bytes (4G Bytes) com endereços byte de 0 a  $2^{32}-1$
- $2^{30}$  palavras (1G words) com endereços 0, 4, 8, ...  $2^{32}-4$

# Os operandos MIPS

23

## MIPS operands

Name	Example	Comments
32 registers	<code>\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at</code>	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register <code>\$zero</code> always equals 0, and register <code>\$at</code> is reserved by the assembler to handle large constants.
$2^{30}$ memory words	<code>Memory[0], Memory[4], ..., Memory[4294967292]</code>	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

# Representação de Valores

24

- Representação binária de valores (dados) inteiros no MIPS

$1011_{\text{two}}$   
represents

$$\begin{aligned} & (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{\text{ten}} \\ = & (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{\text{ten}} \\ = & 8 + 0 + 2 + 1_{\text{ten}} \\ = & 11_{\text{ten}} \end{aligned}$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

(32 bits wide)



# OS 32 REGISTRADORES

Nome	No.Reg.	Uso	Preservado (call)?
\$zero	0	valor constante 0	n.a.
\$v0-\$v1	2-3	valores para resultados e avaliação de expressões	no
\$a0-\$a3	4-7	argumentos (procedimentos e funções)	yes
\$t0-\$t7	8-15	temporários	no
\$s0-\$s7	16-23	salvos/armazenados (variáveis estáticas)	yes
\$t8-\$t9	24-25	mais temporários	no
\$gp	28	ponteiro global para área de memória	yes
\$sp	29	ponteiro de pilha (stack pointer)	yes
\$fp	30	ponteiro de quadro (frame pointer)	yes
\$ra	31	endereço de retorno	yes

**OBS:**

- 1- Registrador 1, chamado **at**, é reservado para o **montador**.
- 2- Registradores 26 e 27, chamados **k0** e **k1**, são reservados para o **SO**

# Compilando códigos C em MIPS (1)

```
a = b + c;  
d = a - e;
```

compilador



```
add $s1, $s2, $s3  
sub $s4, $s1, $s5
```

1. Compilador gera a(s) instrução(ões) correspondentes – *assembly* MIPS
2. Formato bastante rígido: ex: *add <soma, parcela, parcela>*
3. Operandos sempre em registradores
4. Registradores temporários

OBS: A representação binária para cada instrução é única, i.e. não é possível que duas instruções com o mesmo código de máquina sejam decodificadas e executadas de forma diferente da prevista no projeto da CPU.

# Compilando códigos C em MIPS (2)

- ❑ Registradores armazenam as variáveis do programa em C
- ❑ Associação variável  $\leftrightarrow$  registrador feita pelo compilador
- ❑ E se as variáveis `f, g, h, i, j` forem associadas aos registradores `$s0, $s1, $s2, $s3, $s4` como ficaria o código?

f = (g+h) - (i+j);      compilador C

add \$t0, \$s1, \$s2  
add \$t1, \$s3, \$s4  
sub \$s0, \$t0, \$t1

1. Registradores de tamanho de 1 palavra: 32 bits
2. Número limitado: 32 (numerados de 0 a 31 – 5 bits)
3. Registradores temporários são as vezes utilizados

# Compilando códigos C em MIPS (3)

- Instruções para transferir dados Memória ↔ Registrador (CPU)  
Para se acessar um dado, é necessário um endereço
- Dois movimentos básicos:  
`load` (traz da ...) e `store` (leva para a memória)
- No MIPS, acesso somente a blocos (palavras) de 32 bits (4 Bytes)
- Estruturas mais complexas tipo vetores e matrizes?  
Endereço de base (início do vetor) e deslocamento (posição/índice)

# Compilando códigos C em MIPS (4)

- Seja  $A$  um vetor de 100 valores inteiros e as variáveis  $g$  e  $h$  associadas aos registradores  $\$s1$  e  $\$s2$ .
- O endereço de base de  $A$  está armazenado no registrador  $\$s3$ .

**Problema: Como encontrar valor armazenado em  $A[8]$  ?**

$g = h + A[8];$	<b>compilador</b>	<code>lw <math>\\$t0</math>, 32(\$s3)</code>
		<code>add <math>\\$s1</math>, <math>\\$s2</math>, <math>\\$t0</math></code>

# Compilando códigos C em MIPS (5)

- Mesmo vetor  $A$  de 100 valores inteiros e as variáveis  $h$  e  $i$  associadas aos registradores  $\$s2$  e  $\$s3$ .
- O endereço base de  $A$  está armazenado no registrador  $\$s4$ .

$A[4] = h + i;$

compilador C

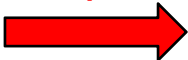


```
add $t0, $s2, $s3  
sw  $t0, 16($s4)
```

Como encontrar valor  
armazenado em  $A[4]$  ?

# Compilando códigos C em MIPS (6)

- Mesmo vetor  $A$  preenchido com 100 valores inteiros e a variável  $h$  associada ao registrador  $\$s2$ . O endereço de base de  $A$  está armazenado no registrador  $\$s3$

$A[12] = h + A[8];$  compilador C 

```
lw    $t0, 32($s3)
add   $t0, $s2, $t0
sw    $t0, 48($s3)
```

1. Uso do registrador temporário para armazenar resultado parcial
2. Acesso aos elementos do vetor: *base e deslocamento*
3. A ordem dos operandos não é alterada para `lw` e `sw`

# Compilando códigos C em MIPS (7)

- Operandos constantes e imediatos são muito frequentes...
- Seja *a* uma variável (um contador em um programa qualquer) associada ao registrador `$s1`.

`a = a + 1;`

ou `a++;`

compilador C



`addi $s1, $s1, 1`

Faz sentido ter uma instrução do tipo `subi`?

**Não:** Uma instrução a mais para uma situação que está resolvida!

*Então, que instruções preciso no meu ISA?*



# INSTRUÇÕES MIPS:

## LÓGICAS & ARITMÉTICAS, TRANSFERÊNCIA DE DADOS E DESVIOS

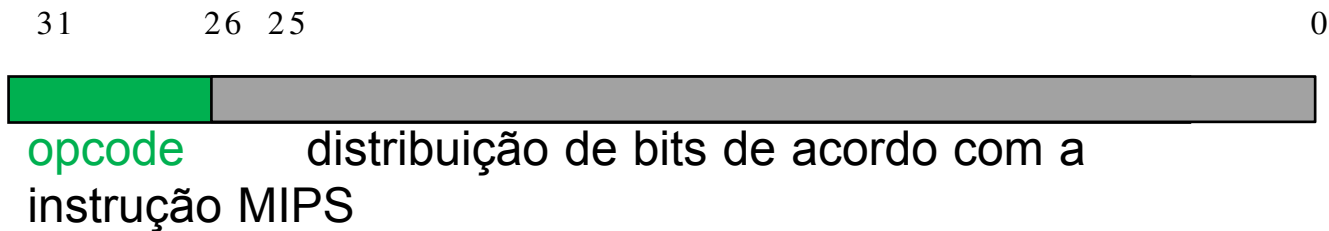


# Possibilidades de Instruções

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, and, subtract, or
Data transfer	Loads-stores (move instructions on machines with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel operations, compression/decompression operations

\_\_\_\_\_

- 
- Diagram illustrating the distribution of bits in a MIPS instruction format:
- Bits 31 to 26 (6 bits) are designated for the **opcode** (highlighted in green).
  - Bits 25 to 0 (26 bits) are distributed according to the instruction type (highlighted in gray).



# Exemplo da Aritmética MIPS

- Princípio de *design*: A simplicidade favorece a regularidade!
- Obviamente, isto complica algumas coisas...

C code:             $A = B + C + D;$   
                      $E = F - A;$

MIPS code:        `add $t0, $s1, $s2`  
                     `add $s0, $t0, $s3`  
                     `sub $s4, $s5, $s0`

- Operandos devem ser registradores, e só existem 32 deles na máquina
- Princípio de *design* : Quanto mais simples, menor e mais rápido!

# Linguagem de máquina (binária)

## MIPS

### Instruções Lógicas e Aritméticas

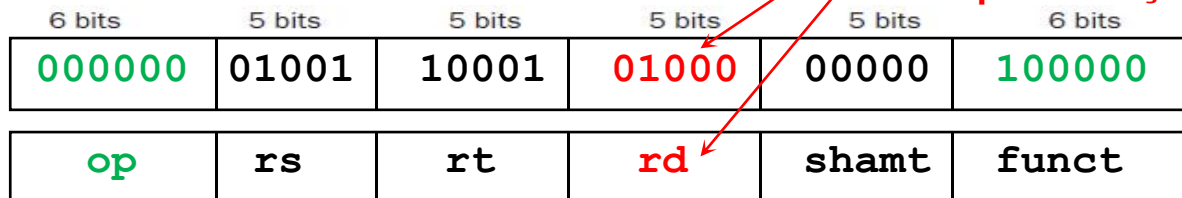
# Aritmética em Linguagem MIPS

- Instruções, assim como registradores de palavras de dados, tem tamanho de 32 bits

Exemplo: `add $t0, $t1, $s1` (assembly MIPS)

registradores são enumerados, `$t0=8`, `$t1=9`, `$s1=17`

- Formato de uma Instrução R típico:



Ordem diferente dos operandos  
no assembly MIPS e na  
implementação !

Machine code

- Você pode tentar imaginar o que esses nomes (siglas) indicam ?

*op* – código de operação      *rs, rt* – registradores fonte      *rd* – registrador destino

*shamt* – (shift amount) é usado em instruções de deslocamento

*funct* – código de função da ULA (opcode para qualquer operação da ULA=00000)

# Lógica em Linguagem MIPS

- Operação de ULA, similar ao add: 32 bits, operandos em registradores

Ex: `or $t0, $t1, $t2`

registradores são enumerados, ~~`$t0=8`~~, `$t1=9`, `$t2=10`

**Ordem diferente dos operandos no assembly MIPS e na implementação!**

- Formato de uma Instrução lógica típica:

000000	01001	01010	01000	00000	100101
op	rs	rt	rd	shamt	funct

Machine code

- Similar ao que foi visto anteriormente (também é uma operação de ULA):  
op – código de operação    rs, rt – registradores fonte    rd – registrador destino  
shamt – (*shift amount*) é usado em instruções de deslocamento  
funct – *opcode* = 00000 para operações da ULA como o OR

# Linguagem de máquina (binária)

## MIPS

### Instruções de Transferência de Dados



# Movimento de Dados em MIPS

- Considere as instruções *load-word* e *store-word*:  
O que o princípio da regularidade nos traz?  
Novo princípio: Bons projetos envolve um compromisso...
- Introdução de um novo tipo de formato de Instrução  
Instruções tipo-I para transferência de dados (*lw* e *sw*)  
Formato de instrução diferente do tipo-R (registradores)
- Exemplo: `lw $t1, 40($s2)`      # `$t1 = MEM[$s2+40]`

100011	10010	01001	0000000000101000
op	rb	rt	16 bit number (immediate)

*rb* – registrador-base para o cálculo do endereço de memória  
*rt* – registrador-destino (para *lw*) ou registrador-fonte (para *sw*)

**Ordem diferente dos  
operandos no assembly MIPS  
e na implementação em  
linguagem binária!**

# Movimento de Dados

- Instruções *store-word* ( $sw$ ):

Observe o que o princípio da regularidade proporciona...

Regularidade  $\rightarrow$  Simplicidade  $\rightarrow$  Velocidade

- Formato da Instrução *store* ( $sw$ )

Também é do tipo-I

Operando imediato: deslocamento

- Exemplo:  $sw \ \$t1, 44(\$s2)$        $\# \text{ MEM}[\$s2+44] = \$t1$

101011	10010	01001	0000000000101100
op	rb	rt	16 bit number (offset)

*rb* – registrador-base para o cálculo do endereço de memória

*rt* – registrador-destino (para *lw*) ou registrador-fonte (para *sw*)

**Ordem diferente dos  
operandos no assembly MIPS  
e na implementação em  
linguagem binária!**

# Modos de endereçamento

- Apesar de ser uma máquina “RISC load-store”, usando os registradores e operandos imediatos, diferentes modos de endereçamento de palavras (de dados) podem ser implementados:

## 1. Endereçamento de operandos

imediato – valor já faz parte da instrução

registrador – registrador usado para dados

# Modos de endereçamento

## 2. Referência à Memória (instruções *load* e *store*)

label – endereço fixo que faz parte da instrução

Indireto – registrador contém um endereço

Endereçamento de Base – campo de um registrador

Endereçamento indexado – elementos de um vetor

□ E as Instruções ? Como buscá-las na memória ?

# Linguagem de máquina (binária)

## MIPS

### Instruções de Controle de Fluxo de Execução (Desvios)

# Controle de Fluxo

- Pontos de decisão num programa criam situações que:  
alteram o fluxo de controle, i.e., mudam a **próxima instrução** a ser executada
- Decisão tomada em tempo de execução de acordo com o que foi codificado para definir **a próxima instrução do programa (valor de PC)**
- Endereço de “destino” é **sempre especificado**: O desvio é **RELATIVO ao PC**.
- Instruções MIPS de desvio condicional:

```
bne $t0, $t1, Label    # ($s<>$t) → PC=PC+(endif1<<2) else PC=PC+4
```

```
beq $t0, $t1, Label    # ($s==$t) → PC=PC+(endif1<<2) else PC=PC+4
```

**Em C:** `if (i!=j) h=i+j;`      **Em MIPS:** `bne $s0, $s1, Label`

```
...  
Label: add $s3, $s0, $s1  
...
```

# Desvios Condicionais em MIPS

- Considere as instruções *bne* e *beq*:  
Novamente, obedecem ao princípio da regularidade... Operando sobre registradores Base para construções do tipo *if-then-else*
- Ex: `beq $s2, $t1, end`
- `if1 # ($s2==$t1) → PC=PC+(endif1<<2) else PC=PC+4;`

000100	10010	01001	0000010000111000
--------	-------	-------	------------------

op

rs

rt

16 bit number (target)

*rs* – registrador-base para a comparação

*rt* – registrador-teste a ser comparado

*label* – endereço-alvo do desvio (usado no cálculo do endereço a ser escrito em PC em caso do desvio se realizar)

# Controle Fluxo: Desvios Incondicionais

- Instruções MIPS de desvio incondicional (*jumps*):

j    Label

opcode

26 bits (endereço alvo)

000010	10001 10010 01000 00000 101010
--------	--------------------------------

- Instruções usadas para a construção de loops, como em:

```
if (i!=j)                beq $s4, $s5, Lab1
    h=i+j;               add $s3, $s4, $s5
else                      j Lab2
    h=i-j;               Lab1:  sub $s3, $s4, $s5
                        Lab2:  ...
```

- *É o mais puro e simples GOTO!*
- *Poderíamos escrever a mesma lógica em MIPS com outros comandos?*
- *Você poderia pensar numa construção simples de um loop em MIPS?*



# Controle do Fluxo

- Temos: `beq`, `bne`. E para uma nova instrução tipo *branch-if-less-than*?

`slt $t0, $s1, $s2`



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

- Usando `beq`, `bne` e `$t0` pode se implementar a “lógica requerida”

000000	10001	10010	01000	00000	101010
--------	-------	-------	-------	-------	--------

- A Instrução pode ser usada para construir ‘`blt $s1, $s2, Label`’  
— é possível se construir estruturas de controle genéricas
- Note que o montador necessita de um registrador para fazer isso  
— existem políticas de uso específicas para os registradores
- Instrução `slt` tem mesmo formato das aritméticas:

# Linguagem de Máquina (binária) MIPS

Uso de Constantes em Operações  
Instruções Lógicas e Artiméticas

# Usando Constantes no MIPS

- Constantes pequenas são usadas frequentemente...

Ex:       $A = A + 5;$

$B = B - 1;$

- Soluções ?

Colocar “constantes típicas” na memória e carregá-las.

Criar registradores *hardwired* para certas constantes: Ex. `$zero`

- Instruções MIPS com constantes: tipo-I (1 operando imediato)

```
addi $29, $29, 4
```

```
slti $8, $18, 10
```

```
andi $29, $29, 6
```

```
ori $29, $29, 4
```

# Aritmética com Constantes

- Constantes são usadas frequentemente em programas...
- Constantes funcionam como operandos imediatos:  
Observe novamente o princípio da regularidade...  
Operações *load* e *store* também usam um imediato
- Note que o formato da instrução é exatamente como *lw* e *sw* (tipo-I)
- Ex: `addi $t1,$s2,15` ou `addi $9,$18,15` => # \$t1= \$s2+15

001000	10010	01001	00000000000001111
op	rs	rt	16 bit number (immediate)

*rs* – registrador-base como operando fonte

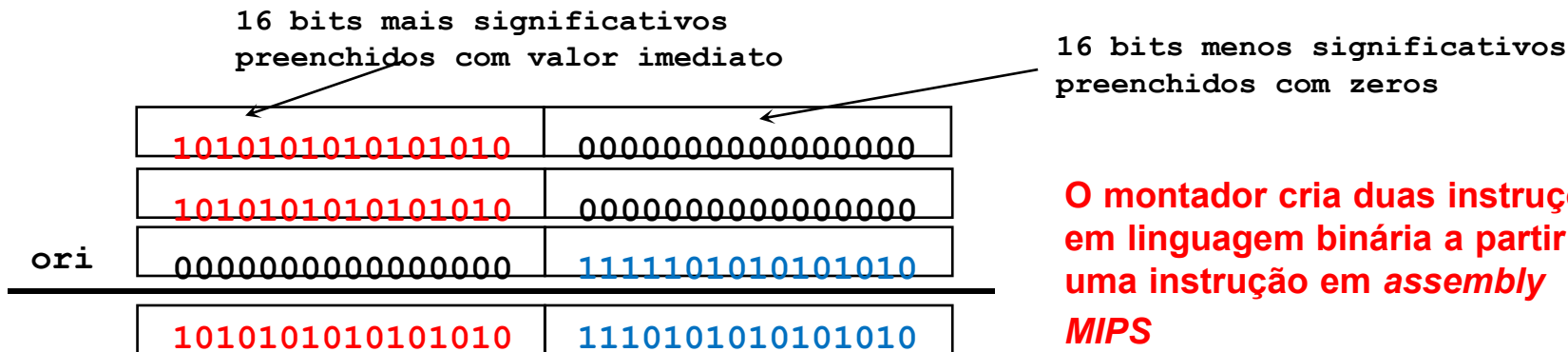
*rt* – registrador-destino para o resultado da operação

# O que fazer com “grandes” constantes ?

- “Constantes grandes”? Maior valor possível como imediato ?
- É possível carregar constantes de até 32 bits em um registrador

Devem ser usadas 2 instruções para implementar isso:

1. Nova Instrução “*load upper immediate*”: `lui $t0, "1010101010101010"`
2. Depois, completar os bits “inferiores” à direita: `ori $t0, $t0, "1111101010101010"`

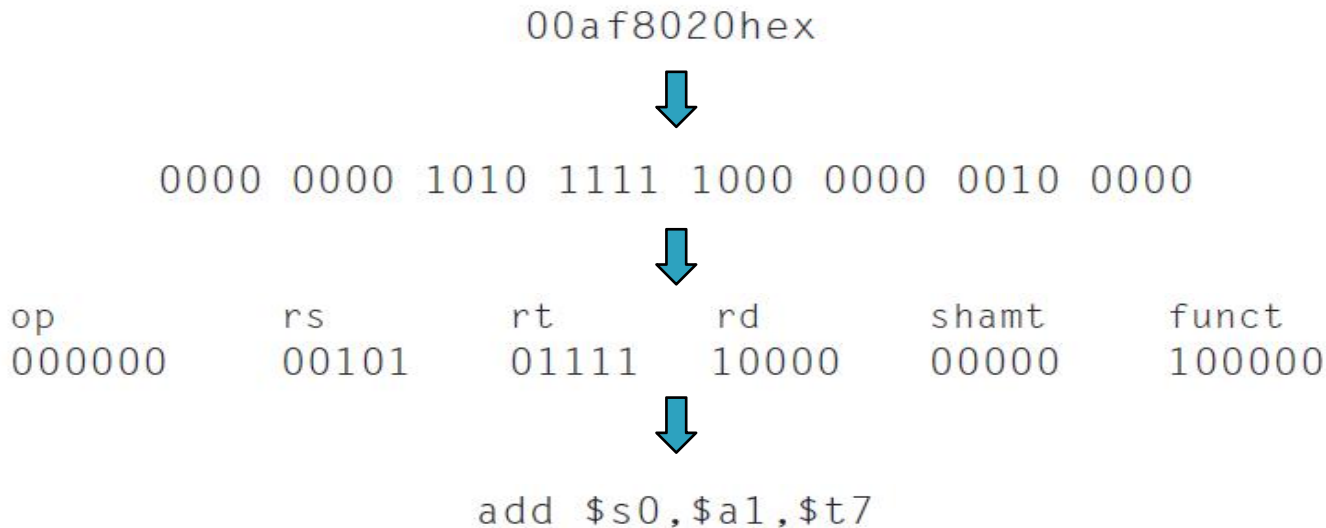


# Linguagem de Máquina X Linguagem Montagem MIPS

# Assembly x Linguagem de Máquina

- ❑ *Assembly* proporciona uma representação simbólica conveniente  
MUITO MAIS FÁCIL que escrever dígitos binários  
Ex: registrador destino é sempre o primeiro
- ❑ Linguagem de Máquina está em um nível de abstração mais baixo  
Ex: Registrador destino não é o primeiro na instrução tipo-R em linguagem de máquina MIPS, diferente do assembly MIPS
- ❑ *Assembly* disponibiliza algumas “pseudoinstruções”  
Ex: “`move $t0, $t1`” existe somente em *Assembly*  
Poderia ser implementado usando “`add $t0, $t1, $zero`”
- ❑ Para avaliar desempenho, apenas a contagem das “instruções reais”, em linguagem de máquina binária, deve ser feita!

# Decodificando a Linguagem de Máquina



**Próximo passo: Construir uma CPU para decodificar MIPS**



# Assembly MIPS x Linguagem Máquina MIPS

```
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		



10011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

# Template Programa MIPS

#Nome: Celso

#Data: 27/08/15

#Descrição: Exemplo de estruturação de um programa MIPS

**.data**

Segmento de dados:

Definição de constantes e variáveis

**.text**

Segmento de texto (código)

Instruções assembly MIPS

# Componentes de um programa MIPS

---

Comment	<code># do the thing</code>
Assembler directive	<code>.data, .asciiz, .global</code>
Operation mnemonic	<code>add, addi, lw, bne</code>
Register name	<code>\$10, \$t2</code>
Address label (decl)	<code>hello:, length:, loop:</code>
Address label (use)	<code>hello, length, loop</code>
Integer constant	<code>16, -8, 0xA4</code>
String constant	<code>"Hello, world!\n"</code>
Character constant	<code>'H', '?', '\n'</code>

# Convenções: uso dos registradores

Nome	No.	Uso	Preservados
\$zero	0	valor constante 0x00000000	-
\$at	1	assembly temporary	Não
\$v0-\$v1	2-3	valores retorno de funções	Não
\$a0-\$a3	4-7	argumentos (procedimentos e funções)	Não
\$t0-\$t7	8-15	temporários	Não
\$s0-\$s7	16-23	temporários salvos (variáveis estáticas)	Sim
\$t8-\$t9	24-25	mais temporários	Não
\$k0-\$k1	26-27	reservado para o Kernel do SO	-
\$gp	28	ponteiro global (global pointer)	Sim
\$sp	29	ponteiro de pilha (stack pointer)	Sim
\$fp	30	ponteiro de quadro (frame pointer)	Sim
\$ra	31	endereço de retorno	Sim