

cin.ufpe.br



UNIVERSIDADE FEDERAL DE PERNAMBUCO

Infra-estrutura Hardware

Infra-estrutura de Hardware

# ARQUITETURA DO PROCESSADOR MIPS

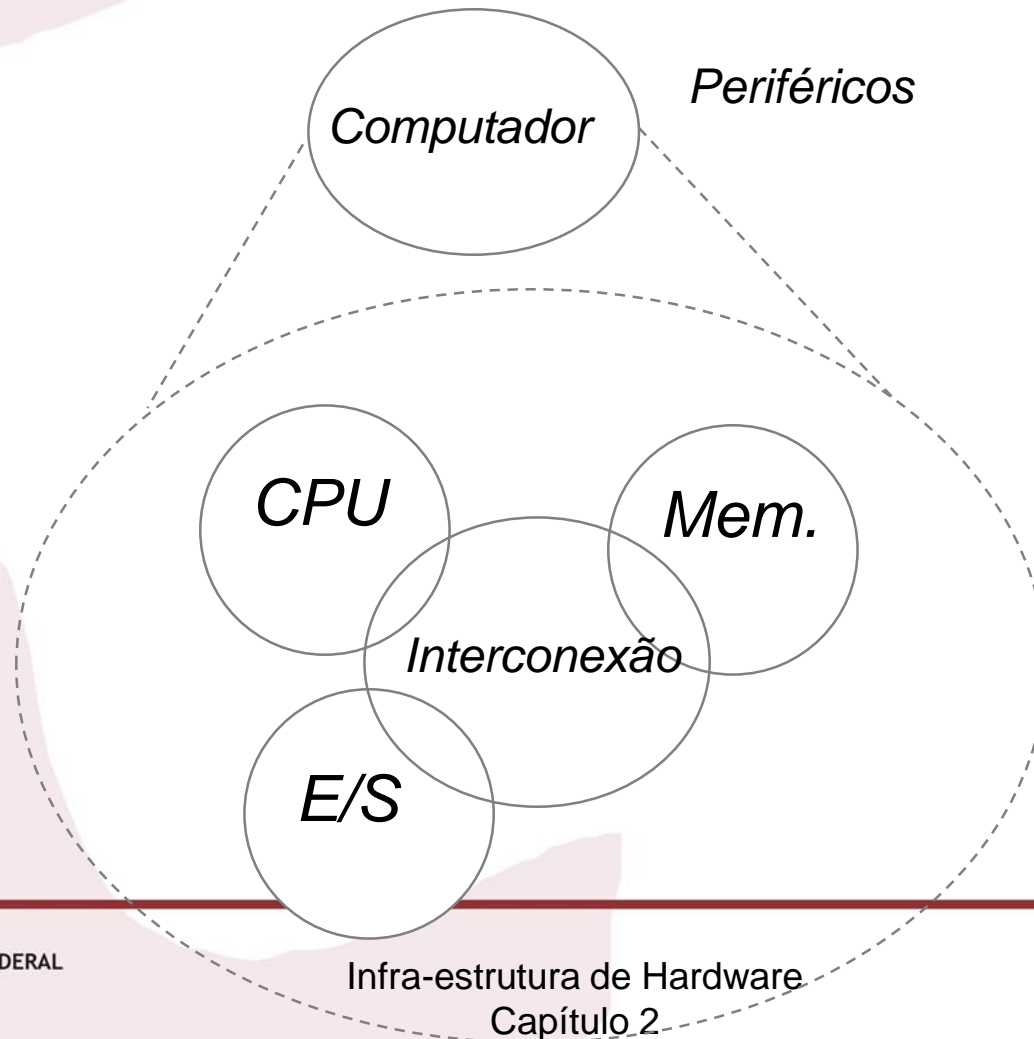
# Roteiro da Aula

---

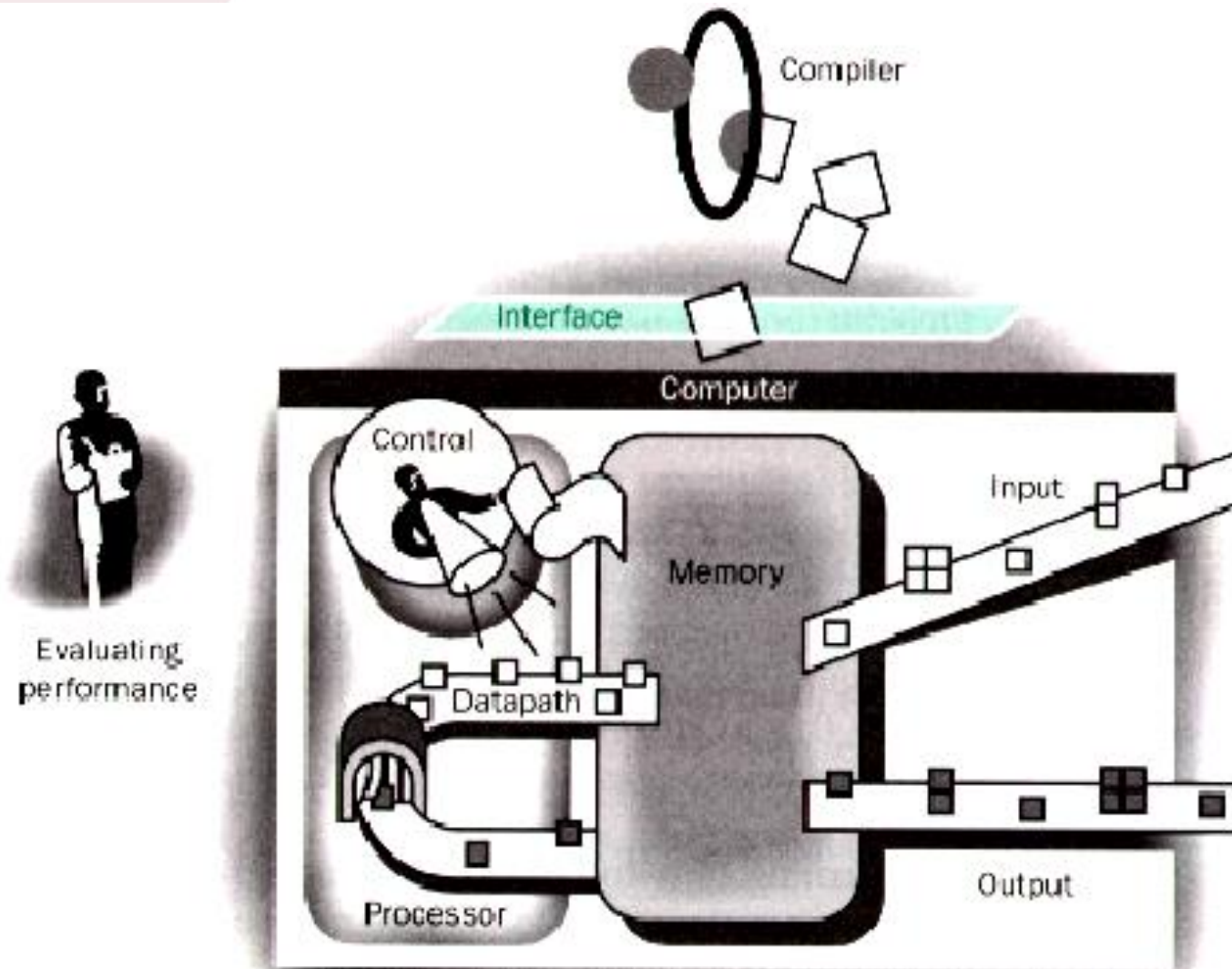


- Introdução
- Operações Aritméticas
  - Representação dos operandos
  - Uso de registradores
- Representando as instruções
- Mais operações sobre dados
- Desvios condicionais
- Funções
  - MIPS
  - Outros processadores
- Modos de Endereçamento
  - MIPS
  - Outros processadores

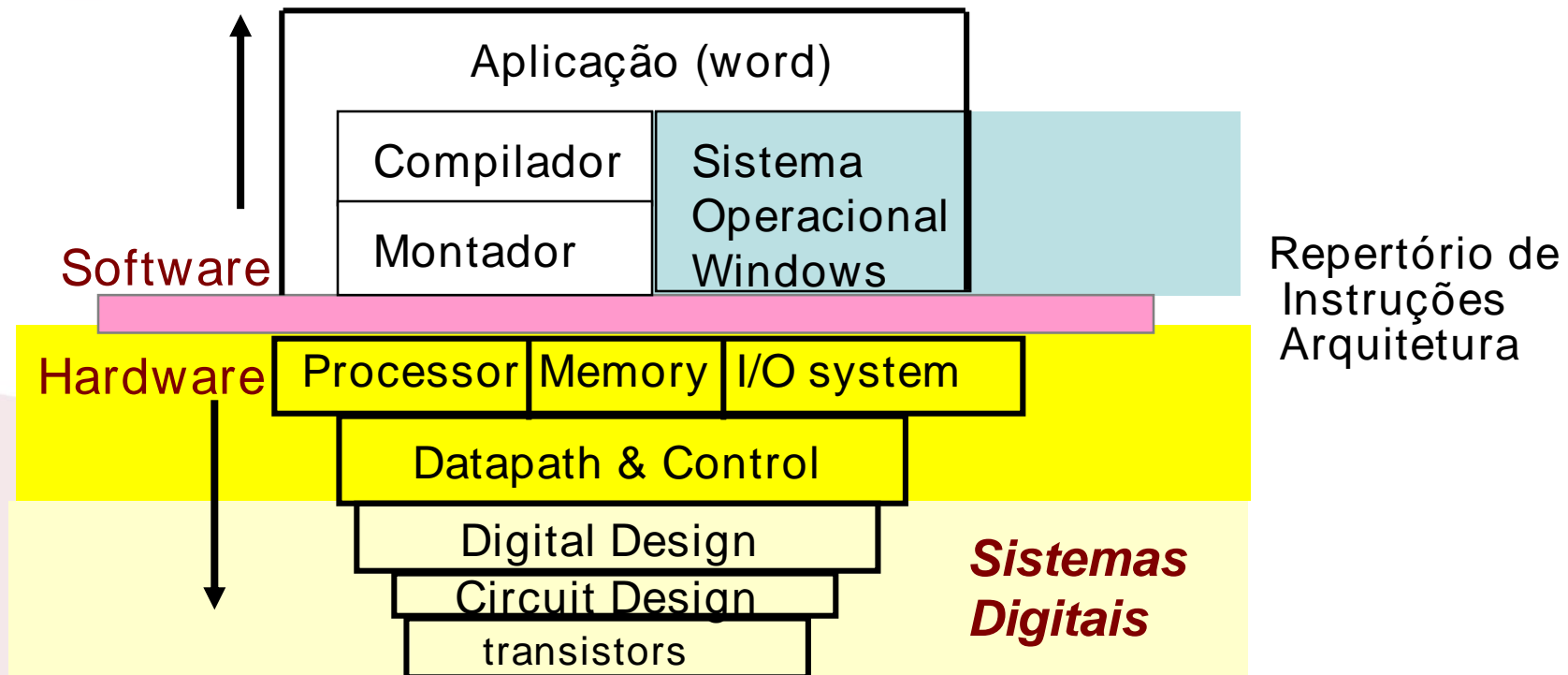
# Componentes de um Computador: Hardware



# Computador: Hardware + Software



# Computador: Hardware + Software



# Conceitos Básicos de Arquitetura de Computadores

## Capítulo 2

# Representação da Informação



Programa em  
Linguagem de alto  
nível (e.g., C)

Compilador

```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

Programa em linguagem  
assembly (e.g., MIPS)

Montador

```
lw $to,    0($2)  
lw $t1,    4($2)  
sw $t1,    0($2)  
sw $t0,    4($2)
```

Programa em  
linguagem de  
Máquina (MIPS)

Hardware

```
1000 1100 0100 1000 0000 0000 0000 0000  
1000 1100 0100 1001 0000 0000 0000 0100  
1010 1100 0100 1001 0000 0000 0000 0000  
1010 1100 0100 1000 0000 0000 0000 0100
```

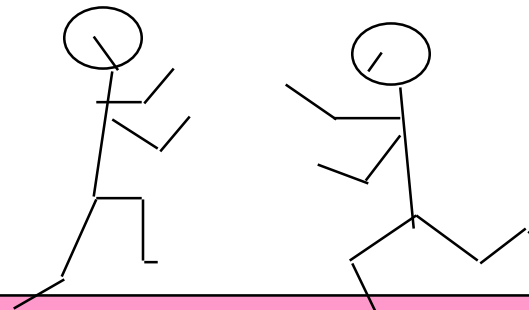




# Interface entre hw e sw: Repertório de Instruções:

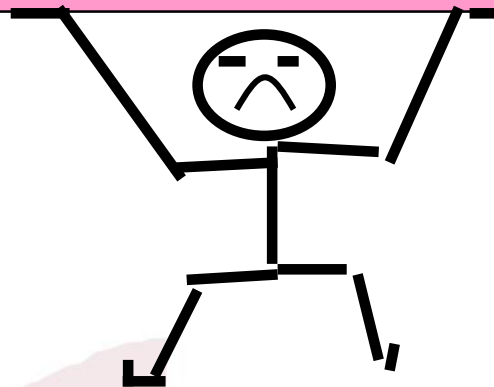


software



Repertório de Instruções

hardware

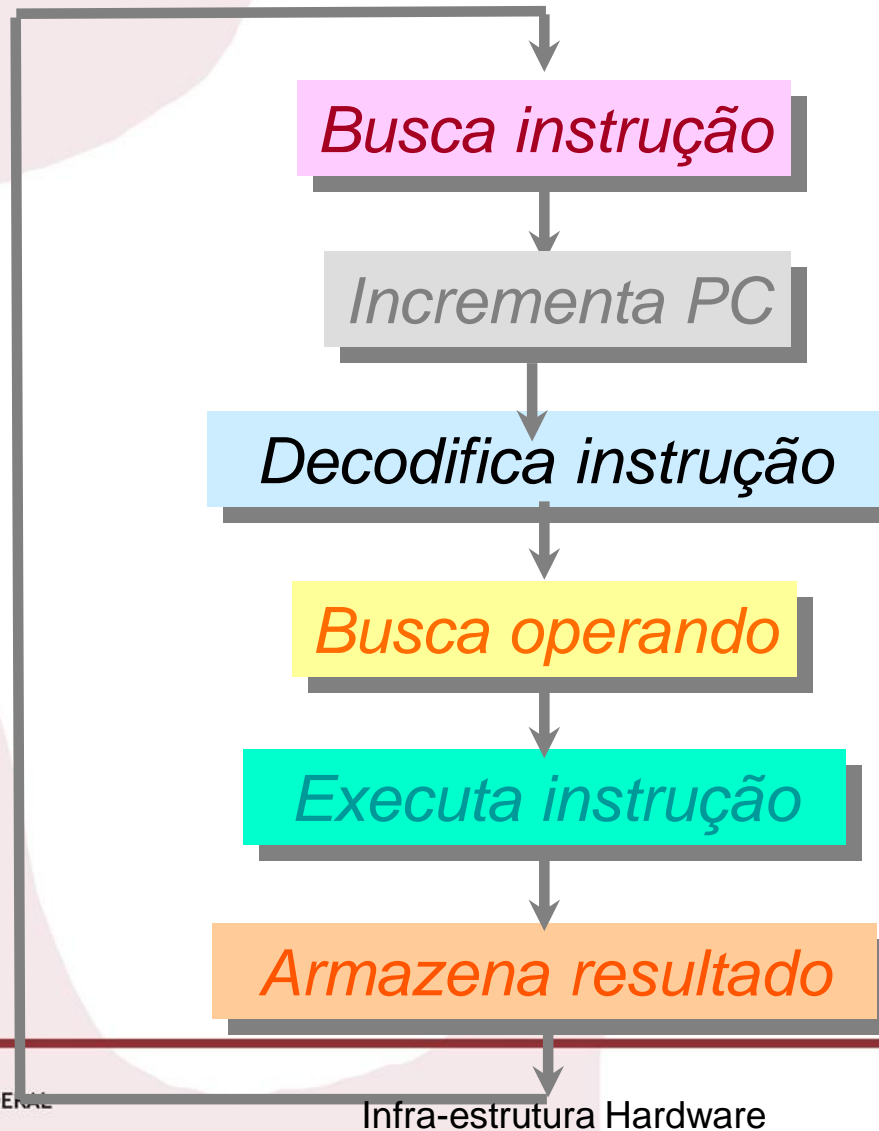


UNIVERSIDADE FEDERAL  
DE PERNAMBUCO

Infra-estrutura Hardware

cin.ufpe.br

# Executando um programa



# QUAIS INSTRUÇÕES QUE UM PROCESSADOR EXECUTA?

# Operações aritméticas



- Aritméticas

- add a,b,c

$$a = b + c$$

- $a = b + c + d + e$ ?

- sub a,b,c

$$a = b - c$$

*Todas as instruções aritméticas possuem 3 endereços:  
destino, fonte 1, fonte 2*

***A simplicidade é favorecida pela regularidade***



# Expressões Aritméticas

---



- $f = (g + h) - (i + j)$ 
  - Variáveis auxiliares:  $t_0$  e  $t_1$
  - add  $t_0, g, h$
  - add  $t_1, i, j$
  - sub  $f, t_0, t_1$



# Operandos no Hardware

---



- Para se garantir o desempenho....
- Operandos em registradores
- Vantagens:
  - leitura e escrita em registradores são muito mais rápidas que em memória

*Quanto menor mais rápido*



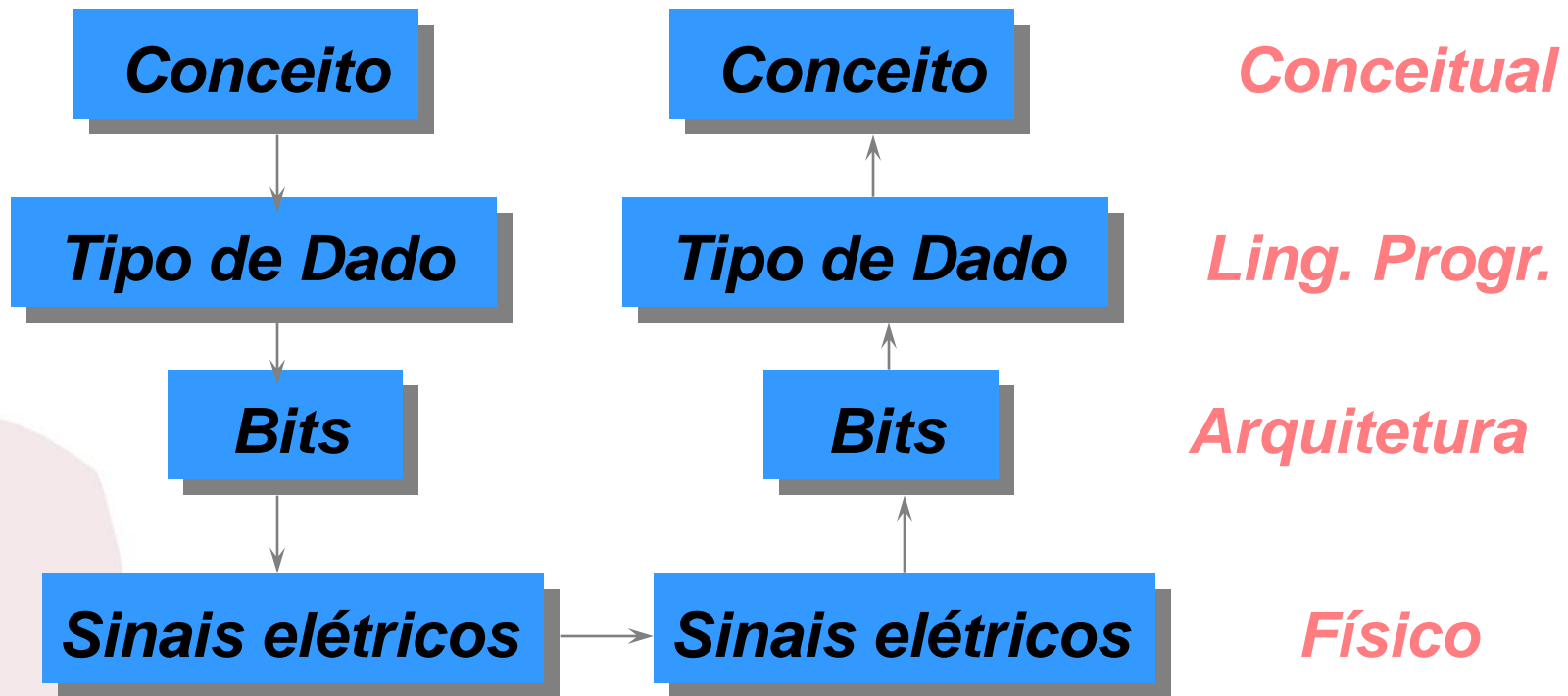
# Operandos no Hardware



- Registradores
  - \$s0, \$s1, .... : armazenam variáveis do programa
  - \$t0, \$t1, .... : armazenam variáveis temporárias
- $f = (g + h) - (i + j)$ ?
  - Variáveis g,h,i e j estão armazenadas nos registradores \$s0, \$s1, \$s2 e \$s3



# Tipos de Dados





# Tipos de Dados

---



- Escalar
  - números
    - Inteiros
    - Ponto-Flutuante (real)
  - caracteres
    - ASCII
    - EBCDIC
  - dados lógicos



# Inteiros

- Representação binária

- sinal-magnitude

00000...000001010

+ 10

10000...000001010

- 10

- complemento a 1

00000...000001010

+ 10

11111...111110101

- 10

- complemento a 2

00000...000001010

+ 10

11111...11110110

- 10

# Inteiros sem sinal



- Dado um número de n-bits

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 até  $+2^n - 1$

- Exemplo

- $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$   
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$   
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$

- Usando 32 bits

- 0 até +4,294,967,295



# Inteiros com sinal 2s-Complement



- Dado um número de n-bits

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range:  $-2^{n-1}$  até  $+2^{n-1} - 1$

- Exemplo

- $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$   
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$   
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$

- Usando 32 bits

- $-2,147,483,648$  até  $+2,147,483,647$



# Inteiros com sinal 2s-Complement



- Bit 31 – bit de sinal
  - 1 para números negativos
  - 0 para números não negativos
- $-(-2^n - 1)$  não pode ser representado
- Números positivos com única representação
- Algumas representações
  - 0: 0000 0000 ... 0000
  - -1: 1111 1111 ... 1111
  - Maior negativo: 1000 0000 ... 0000
  - Maior positivo: 0111 1111 ... 1111



# Gerando número negativo



- Complementar e adicionar 1
  - Complementar significa  $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \bar{x} = 1111 \dots 111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Exemplo: negar +2
  - $+2 = 0000 \ 0000 \ \dots \ 0010_2$
  - $-2 = 1111 \ 1111 \ \dots \ 1101_2 + 1$   
 $= 1111 \ 1111 \ \dots \ 1110_2$



# Character Data

- Caracteres Byte-encoded
  - ASCII: 128 caracteres
    - 95 gráficos, 33 controle

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

# Character Data



- Unicode: caracteres 32-bit

Latin	Malayalam	Tagbanwa	General Punctuation
Greek	Sinhala	Khmer	Spacing Modifier Letters
Cyrillic	Thai	Mongolian	Currency Symbols
Armenian	Lao	Limbu	Combining Diacritical Marks
Hebrew	Tibetan	Tai Le	Combining Marks for Symbols
Arabic	Myanmar	Kangxi Radicals	Superscripts and Subscripts
Syriac	Georgian	Hiragana	Number Forms
Thaana	Hangul Jamo	Katakana	Mathematical Operators
Devanagari	Ethiopic	Bopomofo	Mathematical Alphanumeric Symbols
Bengali	Cherokee	Kanbun	Braille Patterns
Gurmukhi	Unified Canadian Aboriginal Syllabic	Shavian	Optical Character Recognition
Gujarati	Ogham	Osmanya	Byzantine Musical Symbols
Oriya	Runic	Cypriot Syllabary	Musical Symbols
Tamil	Tagalog	Tai Xuan Jing Symbols	Arrows
Telugu	Hanunoo	Yijing Hexagram Symbols	Box Drawing
Kannada	Buhid	Aegean Numbers	Geometric Shapes



# Dados lógicos



- Representação
  - Um byte

00000000

*Verdadeiro*

00000001

*Falso*

- Um bit em uma palavra

0000....0000100000

*Falso*

*Verdadeiro*




# Ponto Flutuante



- Representação

$$3,14 = 0,314 \times 10^1 = 3,14 \times 10^0$$

$$0,000001 = 0,10 \times 10^{-5} = 1,00 \times 10^{-6}$$

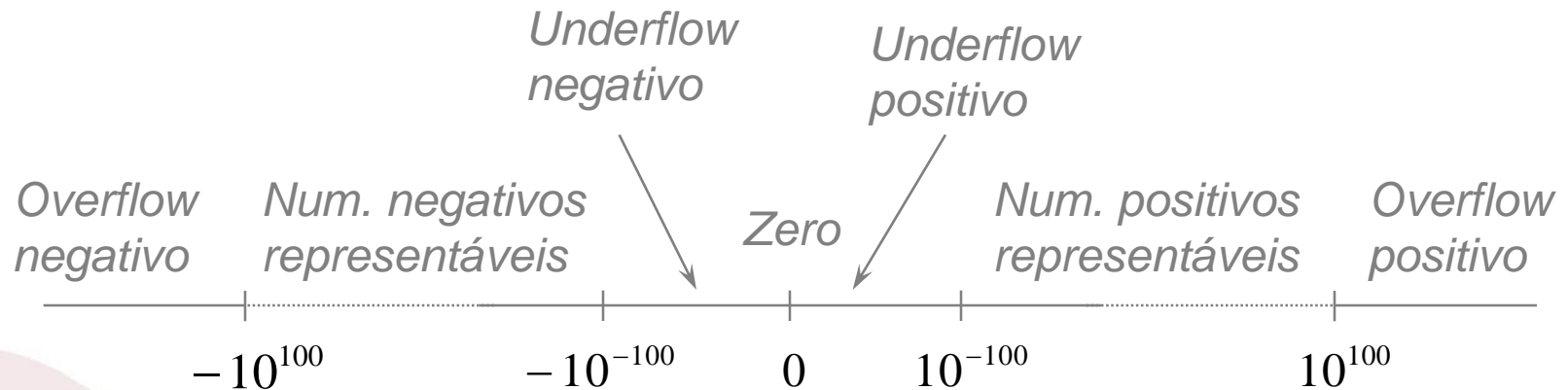

$$n = f \times 10^{e-}$$

<i>sinal</i>	<i>expoente</i>	<i>mantissa (fração)</i>
--------------	-----------------	--------------------------



# Ponto Flutuante

- Padrão IEEE



Item	Precisão simples	Precisão dupla
Sinal	1	1
Expoente	8	11
Mantissa	23	52
Total	32	64

# Tipos de Dados

---



- Escalar
  - números
    - Inteiros
    - Ponto-Flutuante (real)
  - caracteres
    - ASCII
  - dados lógicos
- Estruturas (Estático)
  - Array
  - struct
- Listas, Árvores (Dinâmico)



# Operandos na Memória



- Manipulando arrays:
  - Armazenados na memória
- Instruções que permitam transferência de informação entre memória e registrador

*Instruções de Transferência de Dados  
load word - lw*

*Array: endereço inicial de memória  
elemento a ser transferido*



- Arrays no MIPS:
  - endereço inicial:
    - registrador
  - deslocamento:
    - valor na instrução
- Instrução **L**oad **W**ord:
  - Copia conteúdo de palavra (32bits) de memória para registrador.
  - lw reg\_dst, desl(reg\_end\_inicial)
- $g = h + A[8]$  onde g e h estão nos registradores \$s1 e \$s2 e end. Inicial de A está em \$s3.
  - lw \$t0, 8(\$s3)
  - add \$s1, \$s2, \$t0

# Operandos na Memória



- MIPS
  - Inteiros com 32 bits
  - Memória endereçada por byte



$$\text{End}(A[0]) = 10$$

$$\text{End}(A[1]) = 14$$

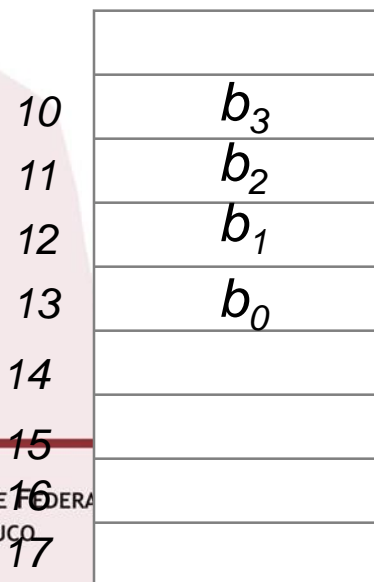
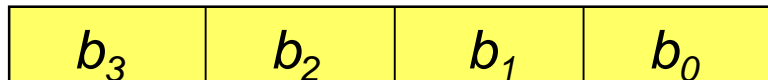
$$\text{End}(A[i]) = \text{End-inicial} + i \times 4$$



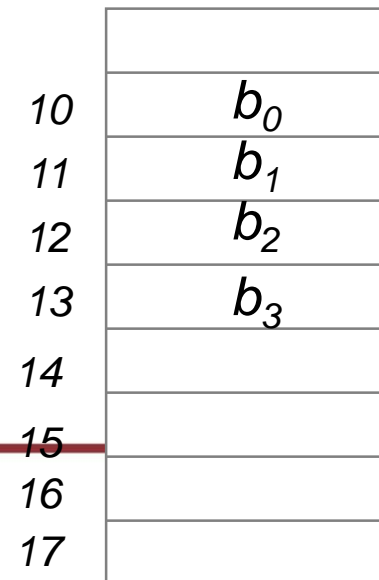
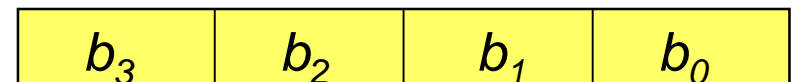
# Operandos na Memória - Endianess

- MIPS

- Inteiros com 32 bits
- Memória endereçada por byte
- Big Endian



Little Endian ( x86, RISC V)



Infra-estrutura Hardware



UNIVERSIDADE FEDERAL  
DE PERNAMBUCO

cin.ufpe.br



# Operandos na Memória



- Escrita em Memória
- Instrução **S** Store **W**ord:
  - Cópia conteúdo de palavra (32bits) de registrador para memória.
  - `sw reg_alvo, desl(reg_end_inicial)`
- $A[12] = h + A[8]$ 
  - endereço inicial de A em \$s3 e h em \$s2
  - `lw $t0, 32($s3)`
  - `add $t0, $s2, $t0`
  - `sw $t0, 48($s3)`



# Operandos na Memória



- Array com variável de indexação
- $g = h + A[i]$ 
  - endereço inicial de A em \$s3 e g, h e i estão em \$s1, \$s2 e \$s4
  - add \$t1, \$s4, \$s4
  - add \$t1, \$t1, \$t1
  - add \$t1, \$t1, \$s3
  - lw \$t0, 0(\$t1)
  - add \$s1, \$s2, \$t0



# Resumo

---



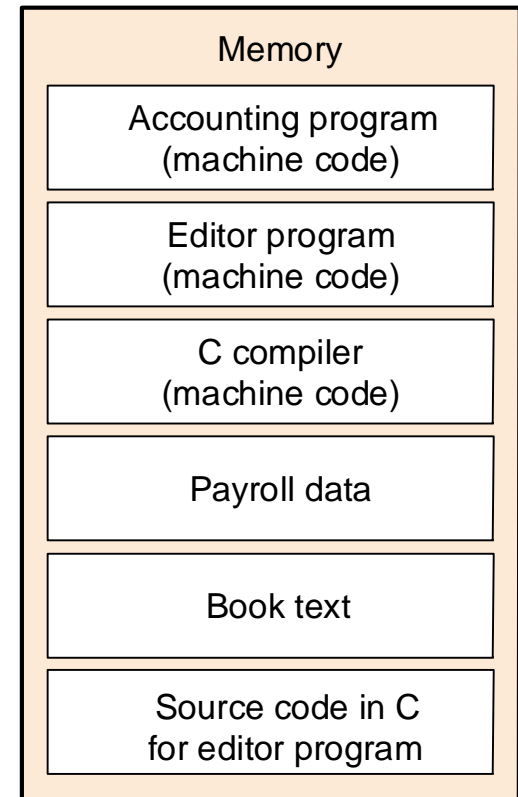
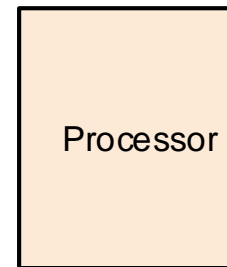
- Operandos no MIPS
  - 32 registradores
    - \$s0, \$s1,...
    - \$t0, \$t1,...
  - $2^{30}$  palavras de memória
    - palavras de 32 bits
    - endereçamento por byte



- Linguagem de montagem do MIPS
  - Aritméticas
    - add regi, regj, regk
      - $\text{regi} = \text{regj} + \text{regk}$
    - sub regi, regj, regk
      - $\text{regi} = \text{regj} - \text{regk}$
  - Acesso à memória
    - lw regi, desl(reg\_base)
      - $\text{regi} = \text{mem}(\text{reg\_base} + \text{desl})$
    - sw regi, desl(reg\_base)
      - $\text{mem}(\text{reg\_base} + \text{desl}) = \text{regi}$

# Computador de Programa Armazenado

- Instruções e Dados possuem uma representação numérica na memória



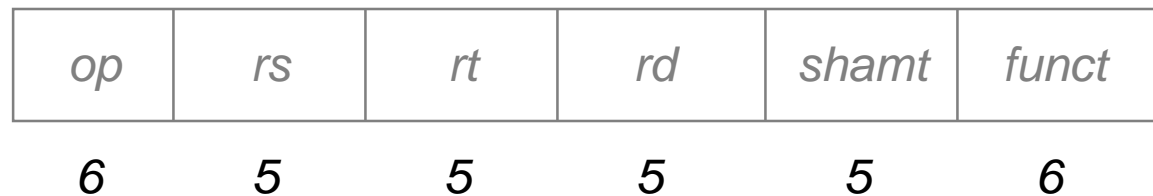
# Representação das instruções

- Informação tem uma representação numérica na base 2
  - codificação das instruções
  - mapeamento de nomes de registradores para números
    - \$s0 a \$s7 : 16 a 23
    - \$t0 a \$t7 : 8 a 15
  - add \$t0, \$s1, \$s2

00000	10001	10010	01000	00000	100000
6	5	5	5	5	6

# Representação das instruções

- Instruções do MIPS
  - Aritméticas



- Transferência de informação



# Linguagem de Montagem vs. Linguagem de Máquina



- $A[300] = h + A[300]$

lw \$t0, 1200(\$t1)

add \$t0, \$s2, \$t0

sw \$t0, 1200(\$t1)

op	rs	rt	rd	Shamt/end	funct
35	9	8		1200	
0	18	8	8	0	32
43	9	8		1200	

op	rs	rt	rd	Shamt/end	funct
100011	01001	01000		0000 0100 1011 0000	
000000	10010	01000	01000	00000	100000
101011	01001	01000		0000 0100 1011 0000	

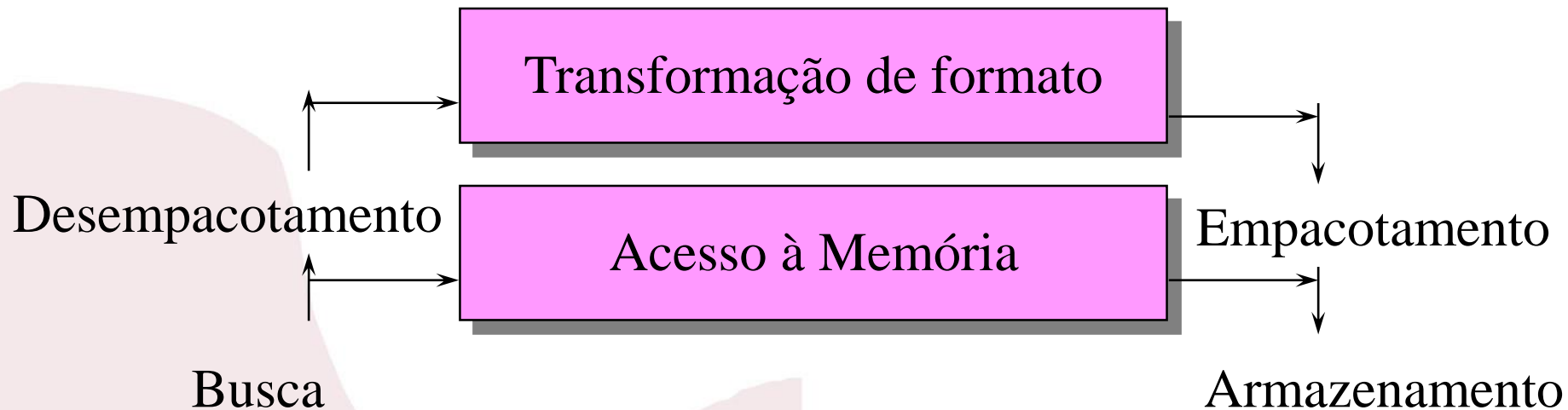




# Exemplo: uma CPU simples...

Instrução	Descrição
nop	No operation
lw reg, desl(reg_base)	reg. = mem (reg_base+desl)
sw reg, desl(reg_base)	Mem(reg_base+desl) = reg
add regi, regj, regk	Regi. <- Regj. + Regk
sub regi, regj, regk	Regi. <- Regj. - Regk

# Mais operações sobre dados



# Operações Lógicas e deslocamento



- Instruções de manipulação de bits

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise NOT	~	~	nor

- Extração de grupos de bits em palavras



# Operações deslocamento



op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- shamt: posições
- Deslocamento para esquerda lógico
  - Desloca para esquerda preenchendo com Os
  - $sll$  de  $i$  bits multiplica por  $2^i$
- Deslocamento para direita lógico
  - Desloca para direita preenchendo com Os
  - $srl$  de  $i$  bits divide por  $2^i$  (números sem sinal)



# Operações deslocamento



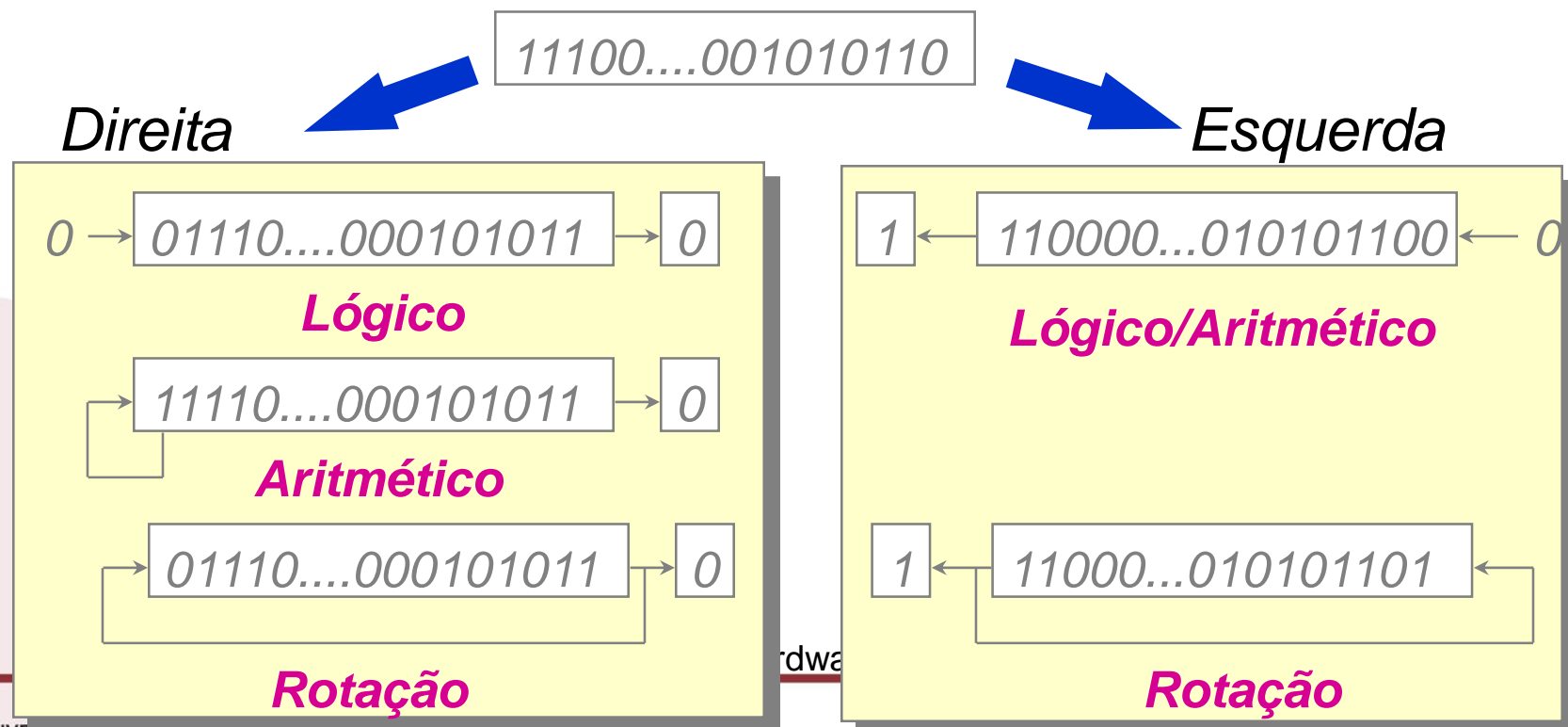
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- shamt: posições
- Deslocamento para direita aritmético
  - Desloca para direita preenchendo com bit mais significativo
  - sra de  $i$  bits divide por  $2^i$  (números sem sinal)

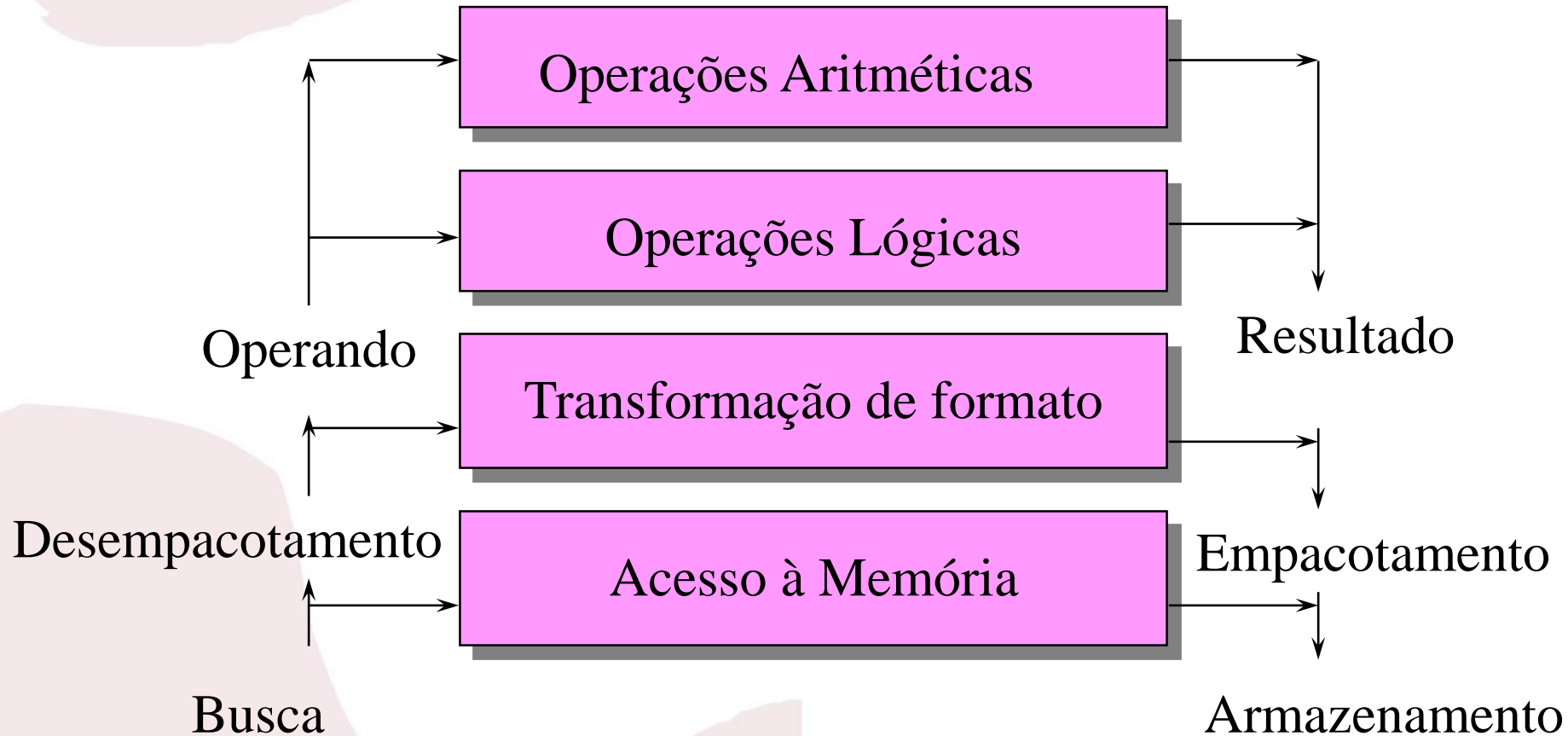


# Operações deslocamento

- Afetam a localização dos bits
- Deslocamento e rotação



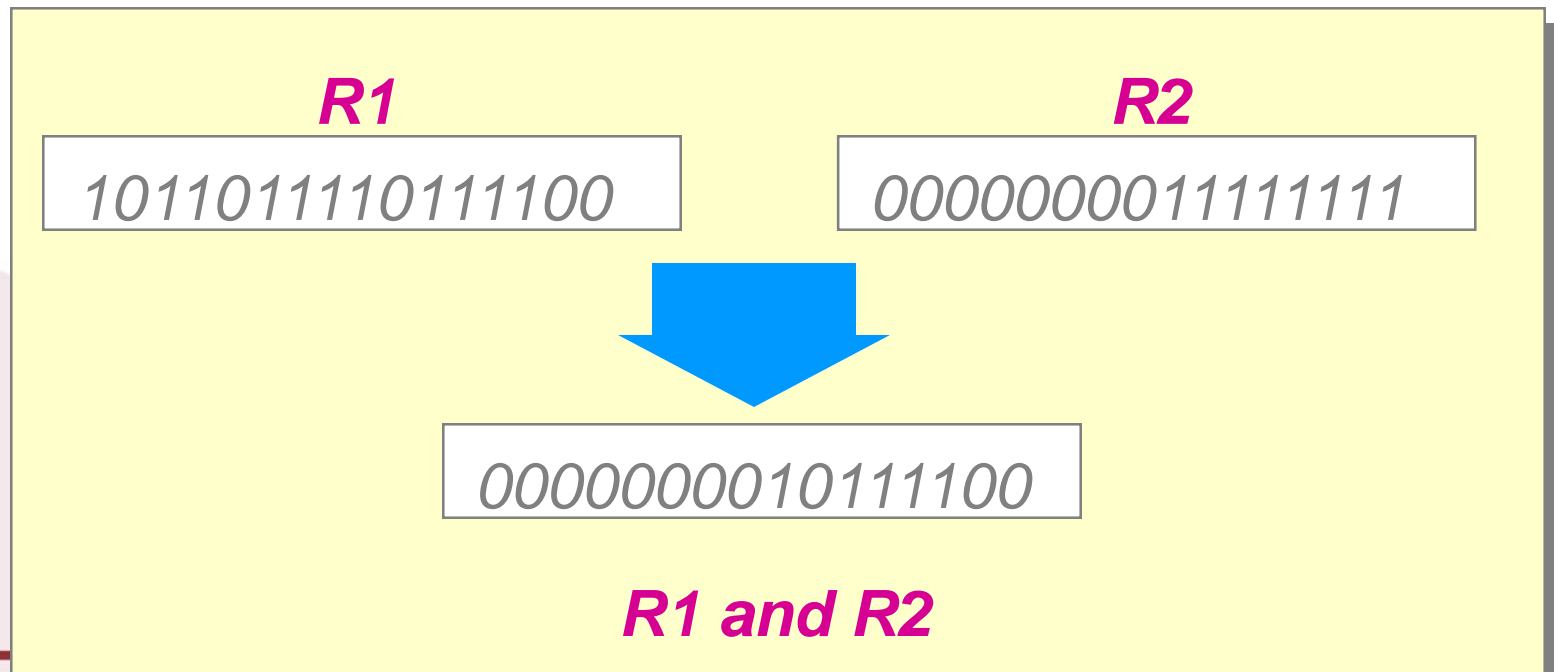
# Operações sobre Dados



# Operações Lógicas



- AND, OR, XOR, NOT
- Extração de grupos de bits





# AND Operations



and \$t0, \$t1, \$t2

\$t2 0000 0000 0000 0000 0000 1101 1100 0000

\$t1 0000 0000 0000 0000 0011 1100 0000 0000

\$t0 0000 0000 0000 0000 0000 1100 0000 0000

# OR Operations

or \$t0, \$t1, \$t2

\$t2 0000 0000 0000 0000 0000 1101 1100 0000

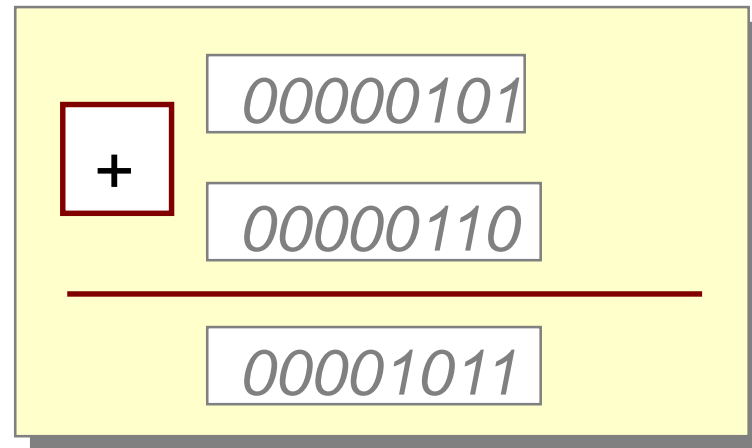
\$t1 0000 0000 0000 0000 0011 1100 0000 0000

\$t0 0000 0000 0000 0000 0011 1101 1100 0000

# Operações Aritméticas



- Soma, subtração:
  - Tipos de dados
    - inteiros
    - ponto-flutuante
  - Exemplo:



\* Pode gerar resultados não representáveis (overflow e underflow)



# Operações Aritméticas

---



- Multiplicação
  - resultado: tamanho duplo
- Divisão
  - dividendo: tamanho duplo
  - resultados:
    - quociente
    - resto



# Controle de Fluxo



- Alterar a sequência de execução das instruções:

## ***Ling. alto nível***

- *If ...then ...else*
- *case*
- *while*
- *for*

## ***Linguagem máquina***

- *Desvio incondicional*
- *Desvio condicional a comparações entre variáveis e/ou valores*



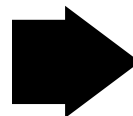
# Desvios no MIPS



- Realizado por duas instruções
  - branch if equal
    - beq reg1, reg2, label
  - jump endereço
    - j label

## ***Ling. alto nível***

```
If a=b then  
    a:= a+c  
else  
    a:= a-c  
end if;
```



## ***Linguagem máquina***

```
lw $S0,a  
lw $S1,b  
lw $S2,c  
beq $S0, $S1, end1  
sub $t0, $S0, $S2  
j end2  
end1: add $t0, $S0, $S2  
end2: sw $t0, a
```



# Desvios condicionais no MIPS



- Instruções de comparação e desvio:
  - beq regd, regs, deslocamento  
 $PC = PC + (\text{deslocamento} * 4)$  se  $\text{regd} = \text{regs}$ ,
  - bne regd, regs, deslocamento  
 $PC = PC + (\text{deslocamento} * 4)$  se  $\text{regd} \neq \text{regs}$ ,



# Compilando If Statements

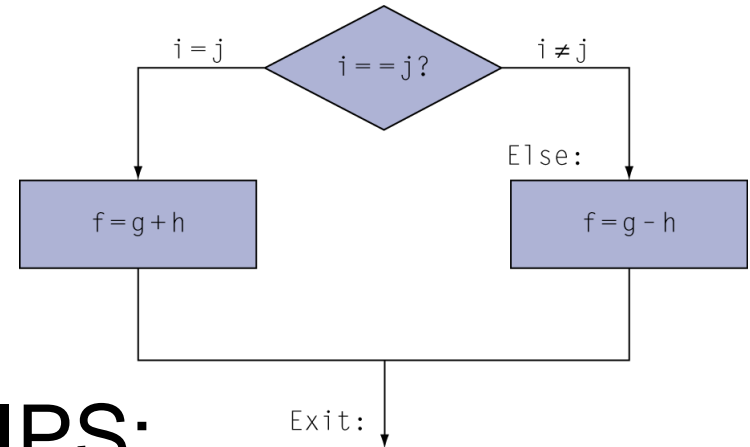
- Código C:

```
if (i==j) f = g+h;  
else f = g-h;
```

– f, g, ... em \$s0, \$s1, ...

- Código compilado para MIPS:

```
        bne $s3, $s4, Else  
        add $s0, $s1, $s2  
        j   Exit  
Else:   sub $s0, $s1, $s2  
Exit:   ...
```



Assembler calcula endereços



# Compilando Loop Statements



- Código C :

```
while (save[i] == k) i += 1;
```

– i em \$s3, k em \$s5, endereço de save em \$s6

- Código compilado para MIPS :

```
Loop:  slt    $t1, $s3, 2
       add   $t1, $t1, $s6
       lw    $t0, 0($t1)
       bne   $t0, $s5, Exit
       addi  $s3, $s3, 1
       j     Loop
Exit:  ...
```



# Desvios condicionais no MIPS

- Instruções de Comparação:
  - set less than
  - slt regd, regs1, regs2
    - Regd = 1 se regs1 menor que regs2, caso contrário regd = 0
- Registrador com constante zero
- Desvio se maior ou igual???
  - slt \$t0, \$s0, \$s1
  - beq \$t0, \$0, end

# Desvios no MIPS

---



- Instruções de desvio:
  - j endereço  
PC = endereço
- Instruções de desvio indireto:
  - jr reg  
PC = (reg)



# MIPS

U · F · P · E

Instrução	Descrição
nop	No operation
lw regi, desl(reg_base)	regi. = mem (regi_base+desl)
sw regi,desl(reg_base)	Mem(reg_base+desl) = regi
add regi, regj,regk	Regi. <- Regj. + Regk
sub regi, regj, regk	Regi. <- Regj. – Regk
and regi, regj,regk	Regi. <- Regj. and Regk
srl regd, regs, n	Desloca regs para direita logico n vezes e armazena em regd
sra regd, regs, n	Desloca regs para dir. aritm. N vezes e armazena em regd
sll regd, regs, n	Desloca regs para esquerda n vezes
ror regd, regs, n	Rotaciona regs para direita n vezes
rol regd, regs, n	Rotaciona regs para esquerda n vezes
beq regi, regj , desl	PC=PC+desl*4 se regi = regj
bne regi, regj, desl	PC=PC+desl*4 se regi <> regj
slt regi, regj, regk	Regi =1 se regj < regk senão regi=0
j end	Desvio para end
jr regd	Desvio para endereço em regd



# Procedimentos e Funções



- Implementação de procedimentos e funções

## *Ling. alto nível*

*Programa principal*

*Var i,j,k: integer*

*Procedure A (var x: integer);*

*...*

*Begin*

*...(corpo do procedimento)*

*End;*

*Begin*

*...*

*A(k); (chamada do procedimento)*

*....*

*End;*

*O endereço de  
retorno deve ser salvo...*

*mas onde?*

*Retorno após  
a chamada*



UNIVERSIDADE FEDERAL  
DE PERNAMBUCO

[cin.ufpe.br](http://cin.ufpe.br)

# Onde salvar o endereço de retorno?



- registradores
  - só permite chamadas seriais
- pilha
  - permite aninhamento e recursividade

```
...  
Procedure A  
begin  
  Procedure B  
  begin  
    Procedure C  
    begin  
      ...  
      C  
      ...  
    end  
  end  
end  
...
```

topo da  
pilha



end. ret. de C
end. ret. de C
end. ret. de B
end. ret. de A

# Chamada de função - MIPS



- Instruções:
  - jal
    - guarda endereço de retorno em \$ra (reg. 31)
    - muda fluxo de controle
  - jr - jump register
    - recupera endereço de retorno de \$ra



# Suporte Função - MIPS



100 jal A

104 sub....

(300) A: add \$t0, \$t0, \$s2

.....

jr \$31

PC

\$31





# Suporte Funções Aninhadas - MIPS



100 jal A

104 sub....

(500) B: sub \$t0, \$t0, \$s2

(300) A: add \$t0, \$t0, \$s2

.....

.....

PC

(340) jal B

jr \$31

\$31

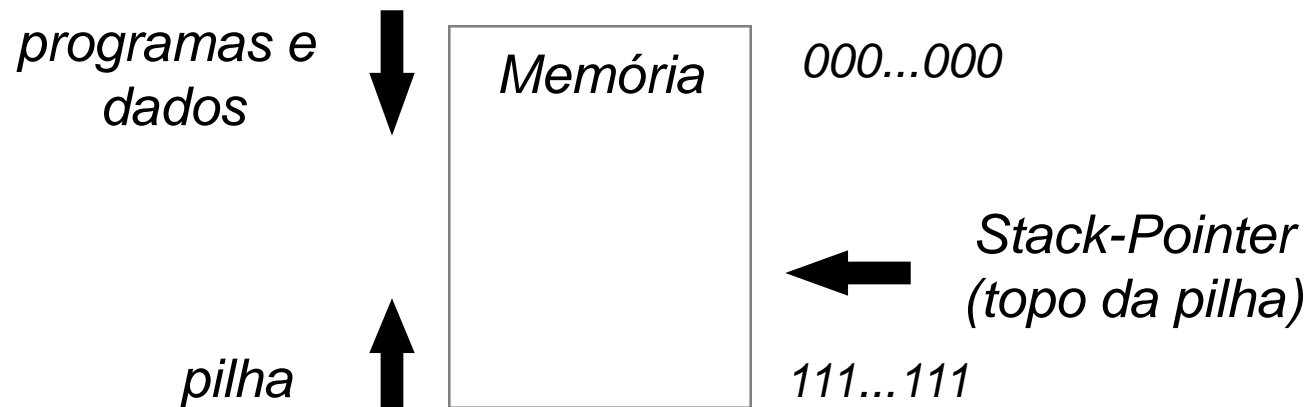
jr \$31



# Implementando a pilha



- Utiliza parte da memória como pilha



- MIPS: stack-pointer (Reg 29)



# Chamade de Procedimento

---



- Passos
  1. Armazena parametros em registradores
  2. Transfere controle para o procedimento
  3. Adquire armazenamento para procedimento (registradores)
  4. Realiza atribuições do procedimento
  5. Armazena resultados nos registradores
  6. Retorna



# Uso Registradores



- \$a0 – \$a3: argumentos (reg's 4 – 7)
- \$v0, \$v1: resultados (reg's 2 e 3)
- \$t0 – \$t9: temporários
  - Podem ser sobreescritos
- \$s0 – \$s7: salvos
  - Devem ser salvos
- \$gp: global pointer – dado estático (reg 28)
- \$sp: stack pointer (reg 29)
- \$fp: frame pointer (reg 30)
- \$ra: return address (reg 31)



# Exemplo Função



- Código C:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Argumentos g, ..., j em \$a0, ..., \$a3
- f em \$s0 (então necessita salvar \$s0 na pilha)
- Resultado \$v0

# Exemplo Função

- MIPS code:

leaf\_example:

addi \$sp, \$sp, -4

sw \$s0, 0(\$sp)

Salva \$s0 na pilha

add \$t0, \$a0, \$a1

add \$t1, \$a2, \$a3

Corpo da função

sub \$s0, \$t0, \$t1

add \$v0, \$s0, \$zero

Resultado

lw \$s0, 0(\$sp)

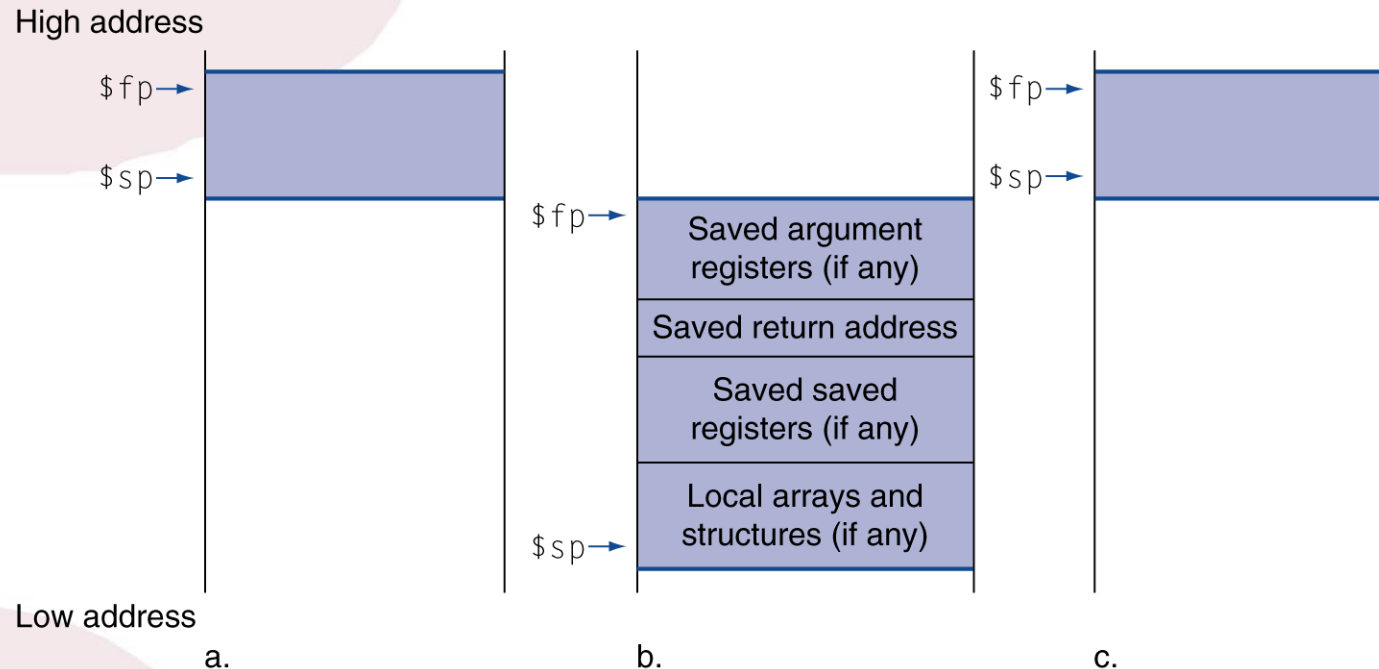
Restaura \$s0

addi \$sp, \$sp, 4

jr \$ra

Retorna

# Dado Local na Pilha



- Dado local armazenado por quem chama
  - e.g., C variáveis automáticas

# Layout da Memória

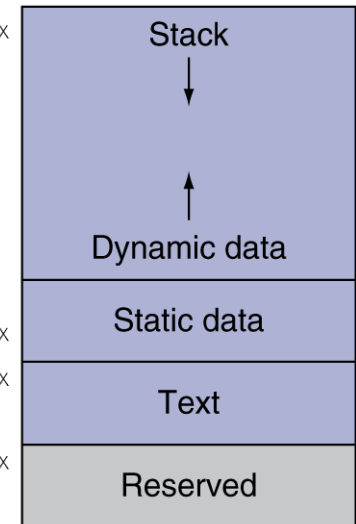


- Texto: código do programa
- Dados estáticos: var. globais
  - e.g., C: static variable, constant arrays e strings
  - \$gp inicializado para endereçar a partir de  $\pm$ offsets
- Dados dinâmicos: : heap
  - E.g., malloc em C, new em Java
- Stack: armazenamento automático

\$sp → 7fff fffc<sub>hex</sub>

\$gp → 1000 8000<sub>hex</sub>  
1000 0000<sub>hex</sub>

pc → 0040 0000<sub>hex</sub>  
0





# Chamada de função em outros processadores

---



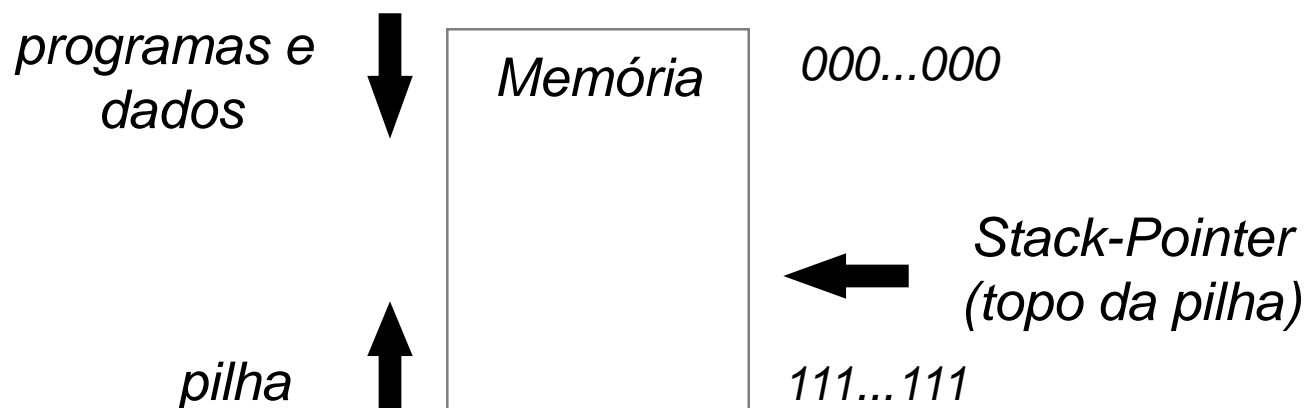
- Instruções:
  - call
    - empilha endereço de retorno
    - muda fluxo de controle
  - ret
    - recupera endereço de retorno
- Outras instruções de suporte...
  - Salvar todos registradores na pilha
  - alocar parte da pilha para armazenar variáveis locais e parâmetros



# Usando a pilha em outros processadores



- Utiliza parte da memória como pilha



- SP: Registrador adicional
- Instruções adicionais:
  - push reg:  $\text{mem}(\text{SP}) \leftarrow \text{reg}$  ; decrementa SP
  - pop reg: incrementa SP,  $\text{reg} \leftarrow \text{mem}(\text{SP})$ ;

# MIPS vs. Outros processadores



- Endereço de retorno:
  - MIPS: registrador
  - Outras: Memória
  - Melhor desempenho
- Acesso à Pilha:
  - MIPS: instruções lw e sw
  - Outras: instruções adicionais
  - Menor complexidade na implementação
  - Compilador mais complexo



# MIPS vs. Outros processadores



- Chamadas aninhadas ou recursivas
  - MIPS: implementada pelo compilador
  - Outras: suporte direto da máquina
  - Compilador mais complexo



# MIPS

Instrução	Descrição
nop	No operation
lw reg, desl(reg_base)	reg. = mem (reg_base+desl)
sw reg, desl(reg_base)	Mem(reg_base+desl) = reg
add regi, regj, regk	Regi. <- Regj. + Regk
sub regi, regj, regk	Regi. <- Regj. - Regk
and regi, regj, regk	Regi. <- Regj. and Regk
xor regi, regj, regk	Regi = regj xor regk
srl regd, regs, n	Desloca regs para direita n vezes sem preservar sinal, armazena valor deslocado em regd
sra regd, regs, n	Desloca regs para dir. n vezes preservando o sinal, armazena valor deslocado em regd.
sll regd, regs, n	Desloca regs para esquerda n vezes, armazena valor deslocado em regd.
ror regd, regs, n	Rotaciona regs para direita n vezes, armazena valor deslocado em regd.
rol regd, regs, n	Rotaciona regs para esquerda n vezes, armazena valor deslocado em regd.
beq regi, regj, desl	PC = PC + desl*4 se regi = regj
bne regi, regj, desl	PC = PC + desl *4 se regi <> regj
slt regi, regj, regk	Regi =1 se regj < regk senão regi=0
j end	Desvio para end
jr reg	Pc = reg
jal end	Reg31 = pc, pc = end
break	Para a execução do programa

# Constantes

- Instruções com constantes:

`addi $reg_d, $reg_f, constante`

`andi $reg_d, $reg_f, constante`

`ori $reg_d, $reg_f, constante`

`slti $reg_d, $reg_f, constante`

- Constantes de 32 bits

`lui $s0, 61`

0000 0000 0111 1101	0000 0000 0000 0000
---------------------	---------------------

`ori $s0, $s0, 2304`

0000 0000 0111 1101	0000 1001 0000 0000
---------------------	---------------------

# Constantes

---



- Muitas constantes são pequenas
  - 16-bit é suficiente
  - Armazenando valor constante
- `lui rt, constant`
  - Copia constante de 16-bit para os 16 bits mais significativos de `rt`
  - Zera os 16 bits menos significativos de `rt`

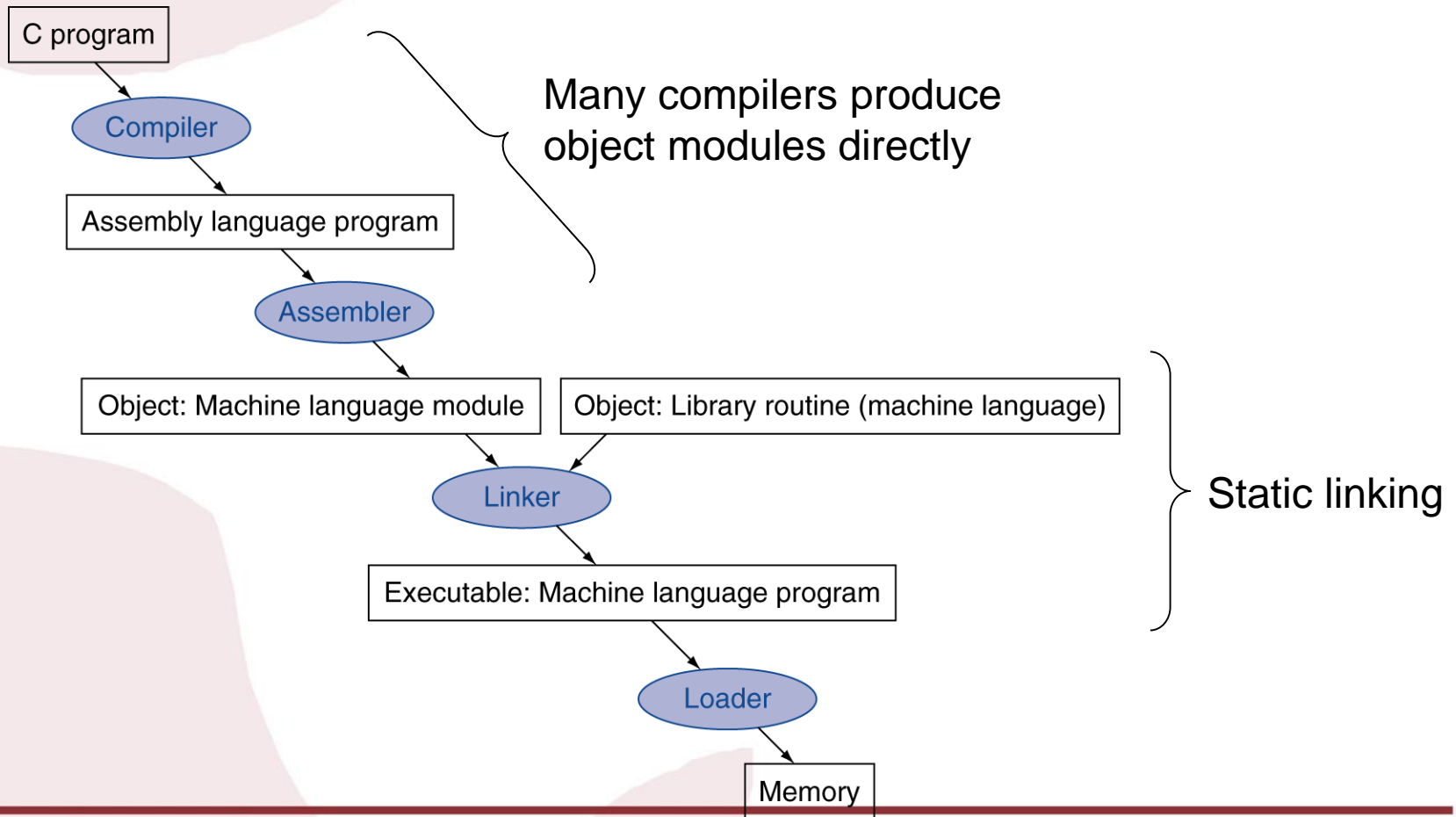


# MIPS

Instrução	Descrição
nop	No operation
lw reg, desl(reg_base)	reg. = mem (reg_base+desl)
sw reg, desl(reg_base)	Mem(reg_base+desl) = reg
lui reg, constante	reg(31..16) = constante
add regi, regj, regk	Regi. <- Regj. + Regk
addi regi, regj, cte	Regi = regj + cte
sub regi, regj, regk	Regi. <- Regj. – Regk
and regi, regj, regk	Regi. <- Regj. and Regk
andi regi, regj, cte	Regi = regj and cte
shrl regd, regs, n	Desloca regs para direita n vezes (Lógico) e armazena valor deslocado em regd .
shra regd, regs, n	Desloca regs para dir. n vezes (aritmético), armazena valor deslocado em regd.
shll regd, regs, n	Desloca regs para esquerda n vezes, armazena valor deslocado em regd.
rotr regd, regs, n	Rotaciona regs para direita n vezes, armazena valor deslocado em regd.
rotl regd, regs, n	Rotaciona regs para esquerda n vezes, armazena valor deslocado em regd.
beq regi, regj, end	Desvia para end. se regi = regj
bne regi, regj, end	Desvia para end se regi <> regj
slt regi, regj, regk	Regi = 1 se regj < regk senão regi=0
slti regi, regj, cte	Regi = 1 se regj < cte senão regi=0
j end	Desvio para end
jr regi	PC = (regi)
jal end	R31 = PC; PC = end
break	Para a execução do programa



# Compilação



# Assembler Pseudo-instruções

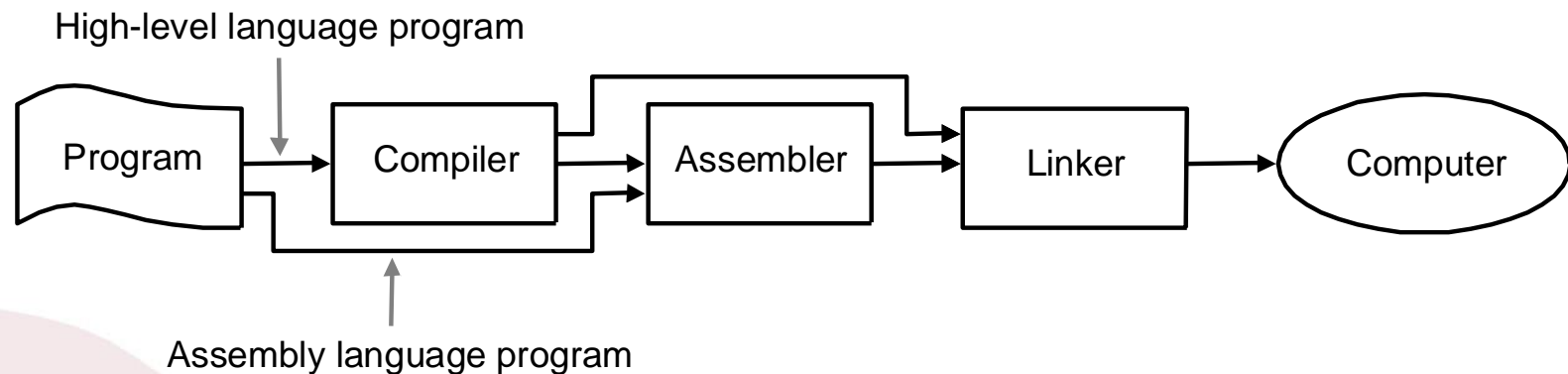
---



- Maioria dos montadores instruções de máquina
- Pseudo-instruções: grupos de instruções de suporte a programação
- `move $t0, $t1`       $\rightarrow$  `add $t0, $zero, $t1`  
`blt $t0, $t1, L`       $\rightarrow$  `slt $at, $t0, $t1`  
   `bne $at, $zero, L`
  - \$at (registrador 1): registrador temporário



# Executando um programa



# Usando o Simulador MIPSIT



- Simulador MIPSIT
  - Local: P:\\cin04\\apps\\
  - Executar mipsit.exe
  - Criando um projeto:
    - File/new -> criar project
  - Editando um programa
    - File/new -> criar arquivo e editar texto



# Exemplo de Programa em Linguagem de Montagem



.data

a: .word 5

b: .word 10

c: .word 5

.text

.globl start

.ent start

start: lw \$8, a

lw \$9, b

lw \$10,c

add \$11, \$9, \$8

sub \$11, \$11, \$10

sw \$11,a

.end start

# Usando o Simulador MIPSIT



- Simulador MIPSIT
  - Compilando um programa
    - build -> build
    - Códigos objetos são armazenados na pasta object
  - Carregando código executável para o simulador
    - Executar o simulador mips.exe
    - Carregar usando mipsit:
      - Build/upload -> to simulator



# Usando o Simulador MIPSIT

- Simulando um programa
  - O que é visível:
    - Registradores (CPU)
      - Conteúdo em hexadecimal
      - Associação nome e número
    - Memória (RAM)
      - Quatro colunas:
        - » Endereço
        - » Conteúdo (hexa)
        - » Rótulos
        - » Instrução de máquina

# Usando o Simulador MIPSIT

- Simulando um programa
  - Iniciando a execução:
    - Indo para rótulo inicial
      - Jump To Symbol -> start
  - Executando passo a passo
    - CPU -> step
    - Botao: seta para quadrado azul
- Para usar nomes simbólicos:
  - Inclua no programa em linguagem de montagem: `#include ,iregdef.h>`



# Outros Modos de Endereçamento

# Operandos no MIPS

---

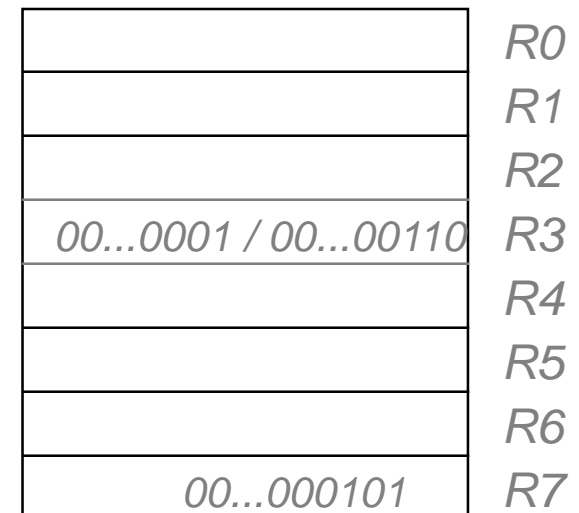
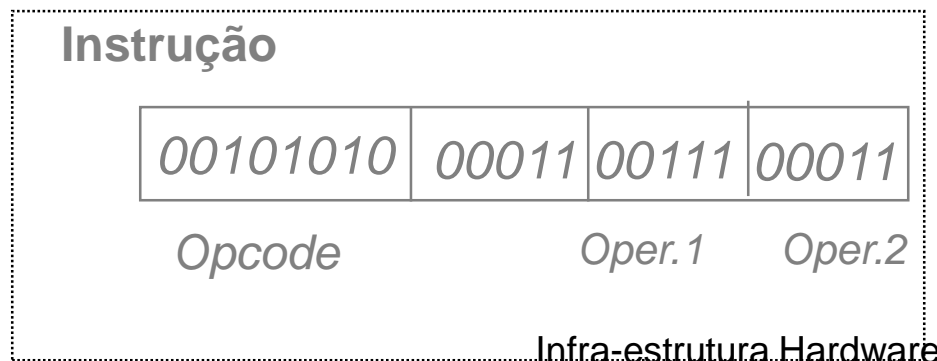


- Aritméticas
  - Registradores
- Load, store
  - Memória



# Endereçamento de registrador

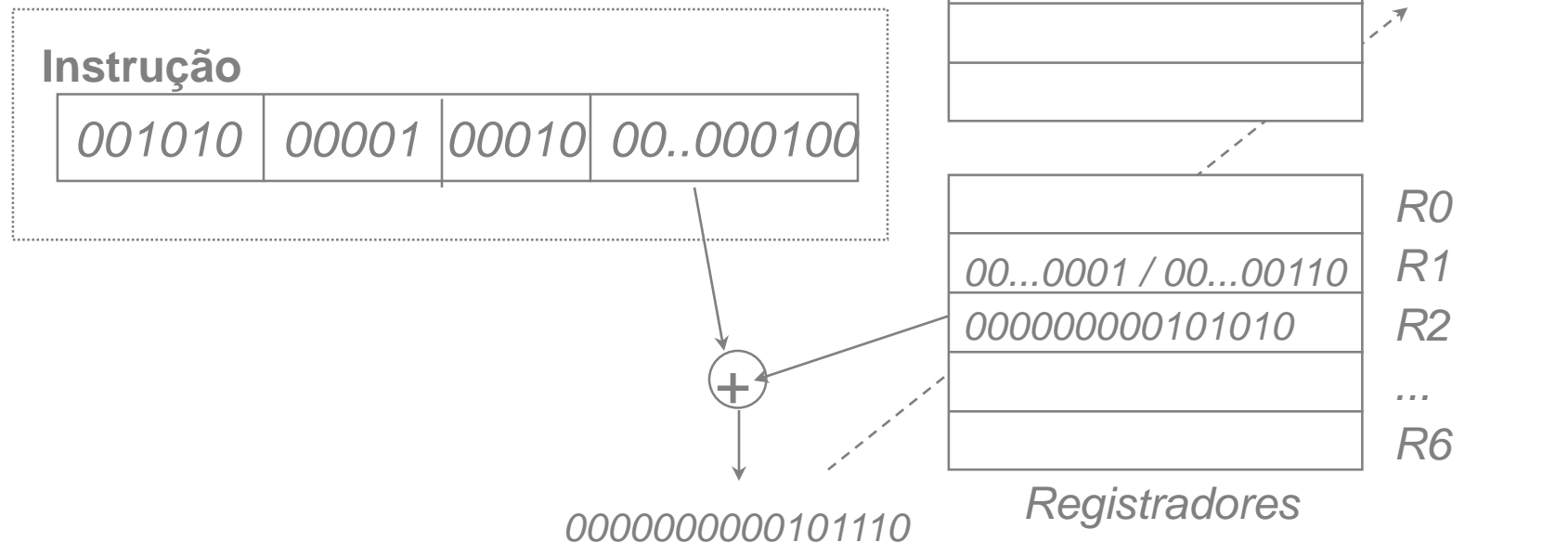
- Operações aritméticas:
  - O operando está em um registrador e a instrução contem o número do registrador
    - ADD R3, R3, R7
      - $R3 \leftarrow R3 + R7$



*Registradores*

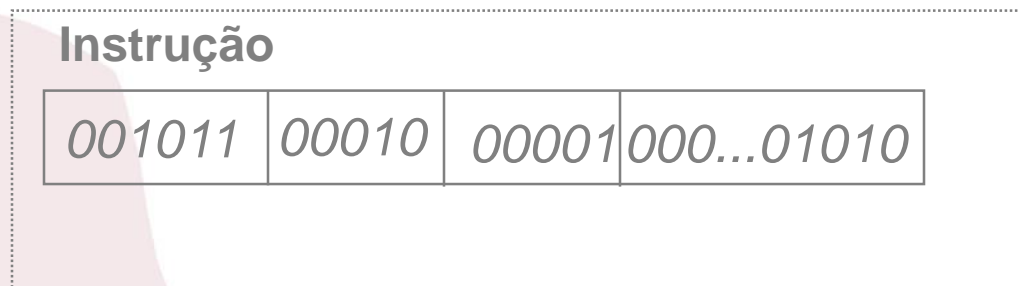
# Endereçamento base

- Instruções de acesso à memória:
  - Instrução: deslocamento
  - Registrador de base: end- inicial
    - Lw R1, desl(R2)



# Endereçamento imediato

- Operações aritméticas e de comparação:
  - O operando é especificado na instrução
    - ADDI R1, R2, #A



	R0
00...0000 / 00...01010	R1
	R2
	R3
	R4
	R5
	R6
	R7

- Modo bastante frequente
- Operando constante

# Endereços no MIPS

---



- Endereço na Instrução
  - J endereço
  - PC = endereço
- Endereço em Registrador
  - Jr reg
  - PC = (reg)
- Endereço relativo a registrador
  - Beq deslocamento
  - PC = PC + deslocamento\*4



# Endereçamento (Pseudo)Direto

- Instrução de Desvio:
  - o endereço da próxima instrução é especificado na instrução
  - J end1
  - PC  $\leftarrow$  end1

## Instrução

001010	000000000000000000000000101010
--------	--------------------------------

## Memória

Add R1, R1, R3
J 000000.....101010

PC=00..101010

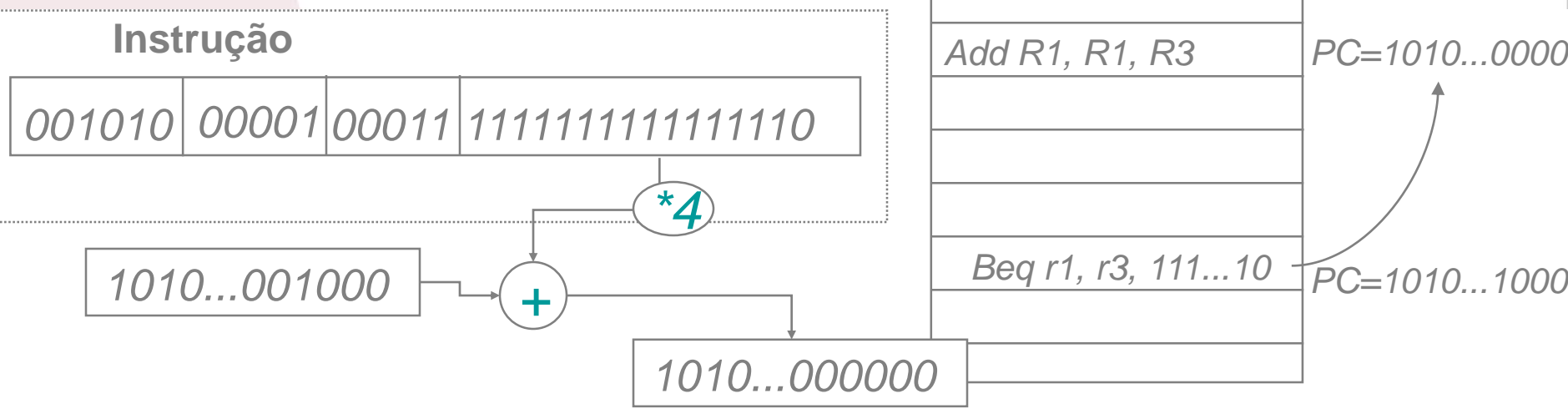
PC=0..101110

Infra-estrutura Hardware

# Endereçamento Relativo a PC



- Instrução de Branch:
  - o número de instruções a serem puladas a partir da instrução é especificado na instrução
  - Beq R1, R3, desl1
  - $PC \leftarrow PC + desl1 * 4$





# Endereçamento de Registrador

- Instrução de Desvio:
  - o endereço do operando na memória é especificado em um registrador
  - Jr R1
    - $PC \leftarrow (R1)$

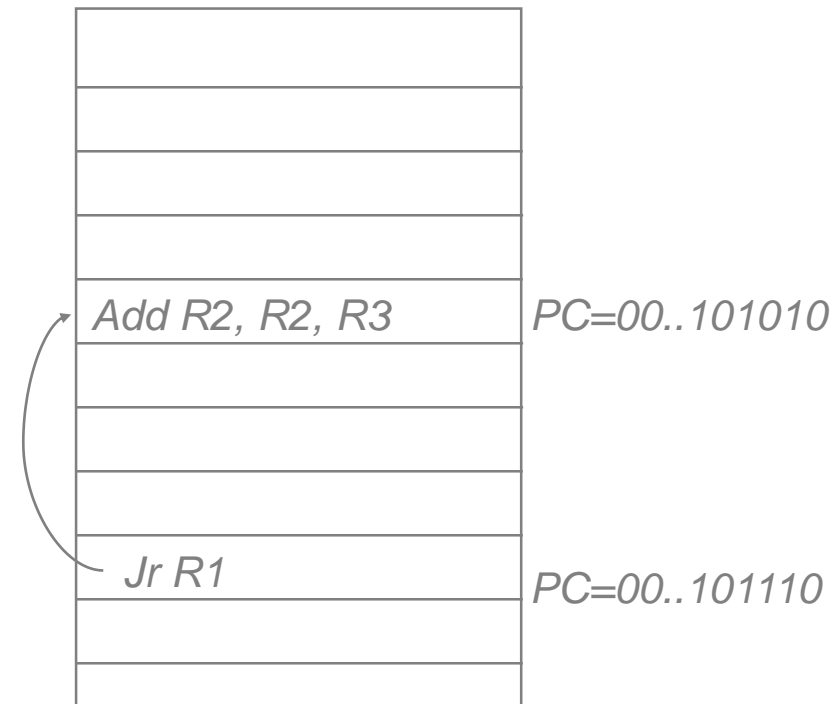
## Instrução

001010	00001
--------	-------

**R1**

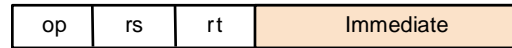
00000..000000101010

## Memória

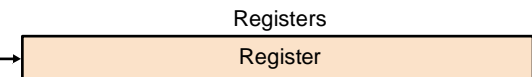
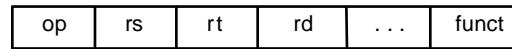


# Modos do MIPS

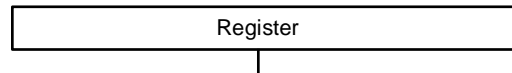
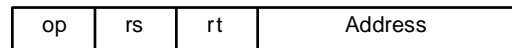
## 1. Immediate addressing



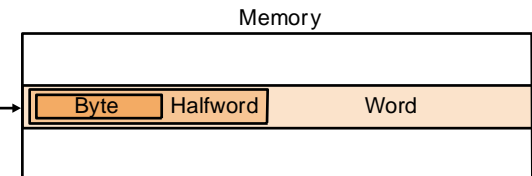
## 2. Register addressing



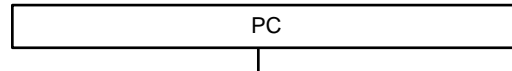
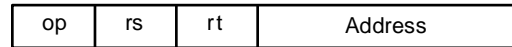
## 3. Base addressing



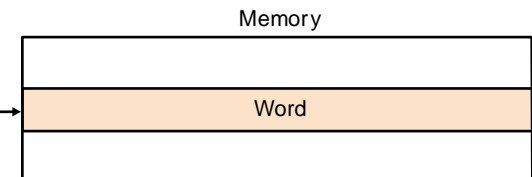
+



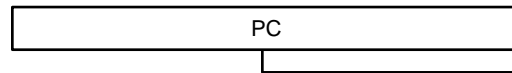
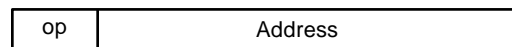
## 4. PC-relative addressing



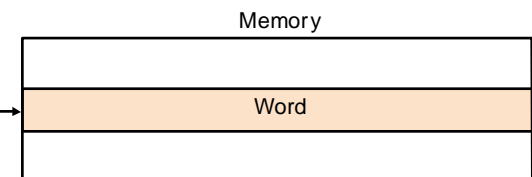
+



## 5. Pseudodirect addressing



!



# Em geral...

- Aritméticas:
  - Operandos em registradores
  - Operandos em memórias
  - ...
- Vários modos de endereçamento

***Como especificar na instrução onde está o operando e como este pode ser acessado ?***



***Campo na Instrução***

# Endereçamento em outras arquiteturas

---



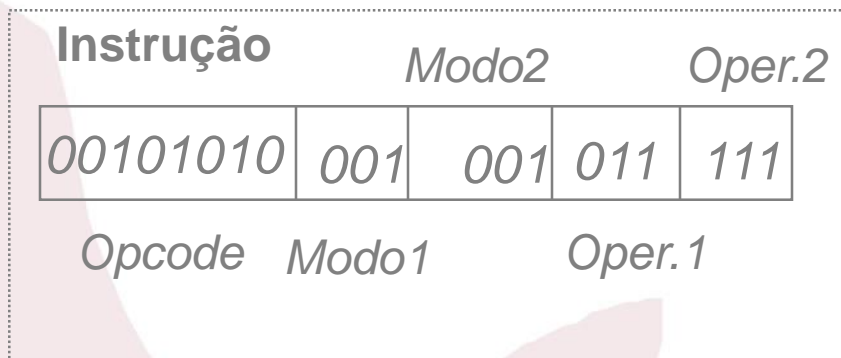
- Aritméticas
  - Registrador-Registrador
  - Registrador - Memória
  - Memória - registrador
- Exemplo: Adição



# Endereçamento de registrador



- O operando está em um registrador e a instrução contém o número do registrador
  - ADD R3, R3
    - $R3 \leftarrow R3 + R3$



	R0
	R1
	R2
00...0001 / 00...00110	R3
	R4
	R5
	R6
00...000101	R7

Registradores  
[cin.ufpe.br](http://cin.ufpe.br)



# Endereçamento Direto



- O endereço do operando na memória é especificado na instrução
  - ADD R1, end2
    - $R1 \leftarrow R1 + [\text{end2}]$

## Registradores

	R0
00...0001 / 00...00010	R1
	R2
	R3
	R4
	R5
	R6
	R7

## Memória

00000...0000001	00..101010

## Instrução

	Modo1	Modo2	
00101010	001	010	001
0000000000000101010			

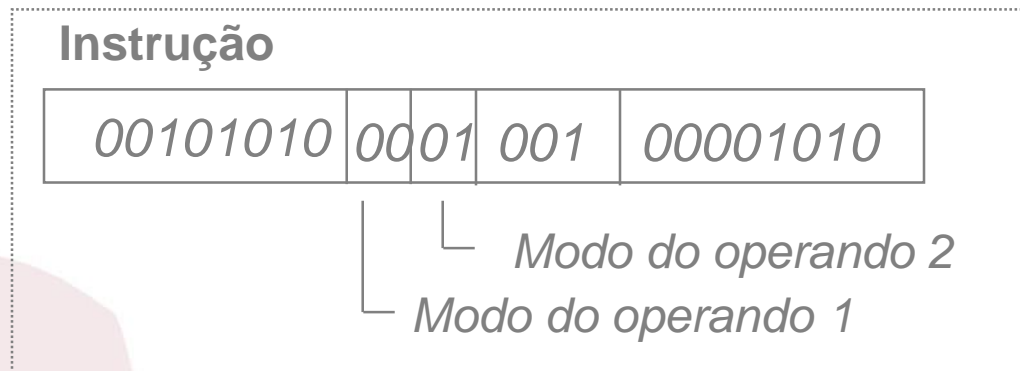
– instruções maiores



# Endereçamento imediato



- O operando é especificado na instrução
  - ADD R1, #A



	R0
00...0000 / 00...01010	R1
	R2
	R3
	R4
	R5
	R6
	R7

*Registradores*

- Modo bastante frequente
- Operando constante



# Endereçamento indireto

- O endereço (reg. ou memória) contem o endereço do operando

– ADD R1,(R2)

- $R1 \leftarrow R1 + \text{mem}(R2)$

## Instrução

00101010	00	11	001	010
----------	----	----	-----	-----

└─ Modo do operando 2

└─ Modo do operando 1

– Endereçamento variável

- ponteiros

## Memória

00000000000000101

00..101010

00...0001 / 00...00110
00000000000101010

R0

R1

R2

R3

...

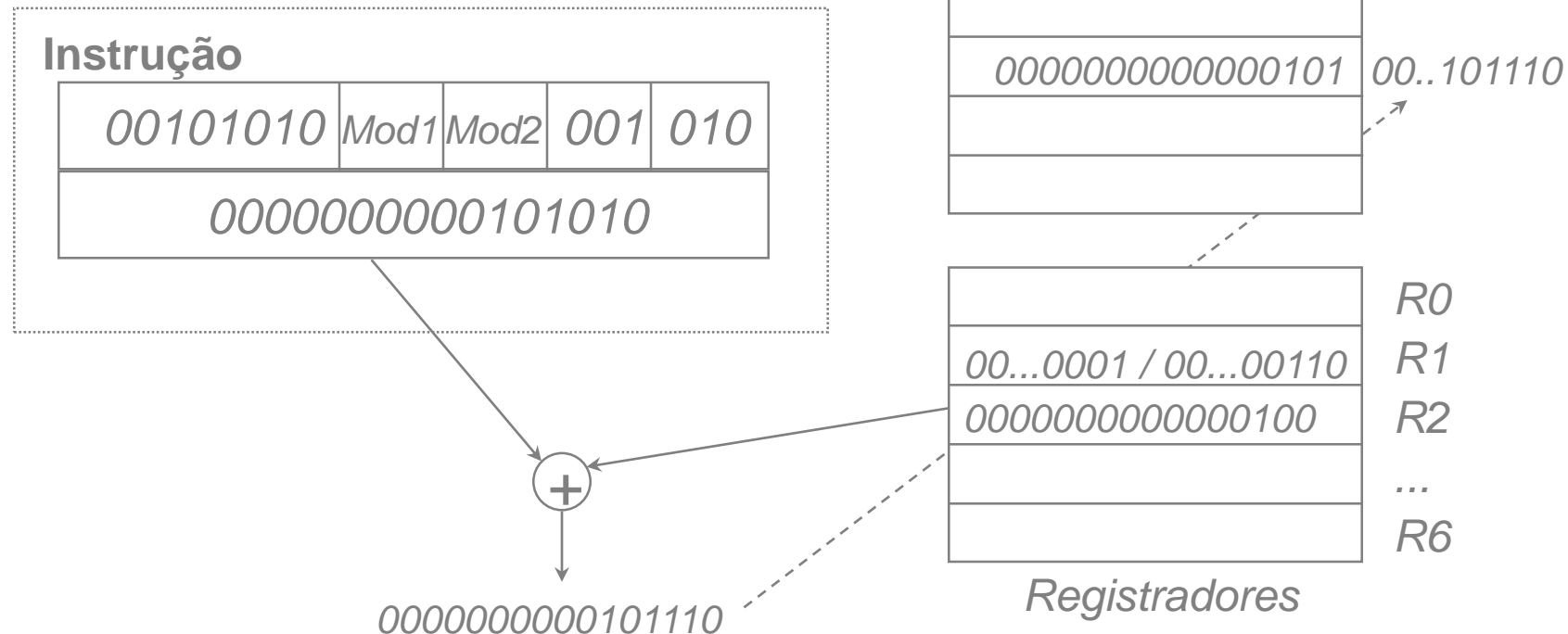
## Registradores



# Endereçamento indexado



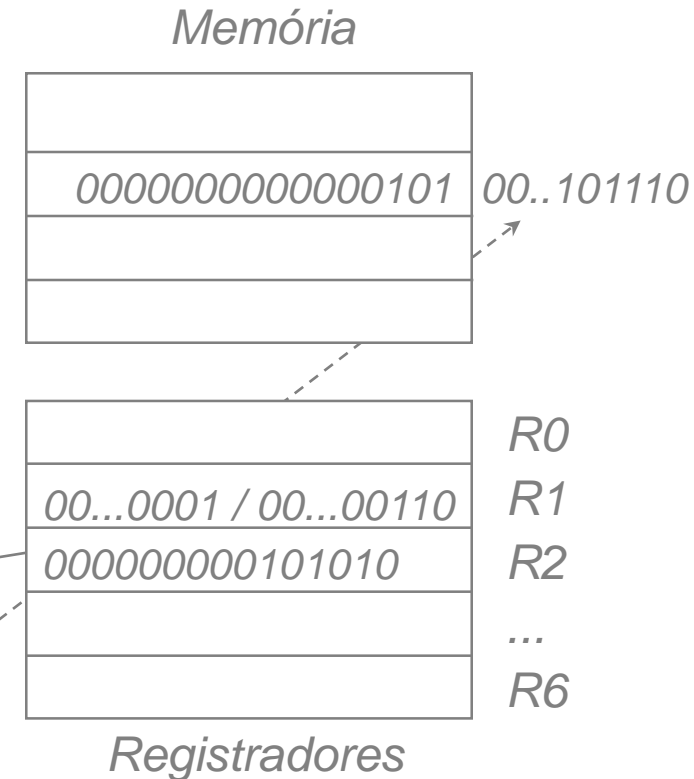
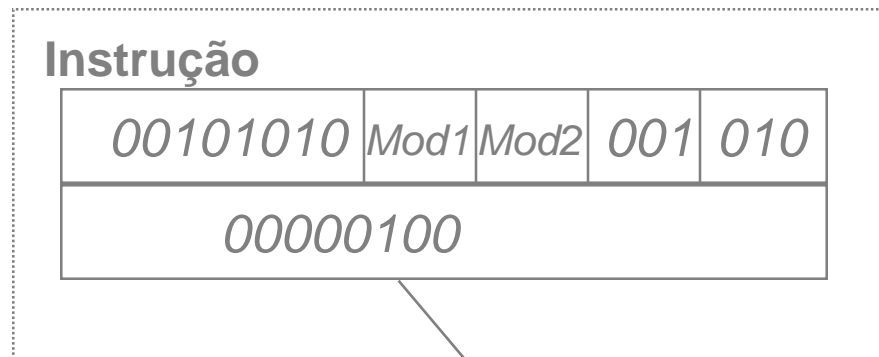
- Instrução: endereço inicial do array
- Registrador de índice: deslocamento
  - ADD R1, [R2]end



# Endereçamento base



- Instrução: deslocamento
- Registrador de base: end- inicial
  - ADD R1, desl(R2)



+



0000000000101110

# Modos de Endereçamento

Addressing mode	Example	Meaning
Register	Add R4,R3	$R4 \leftarrow R4 + R3$
Immediate	Add R4,#3	$R4 \leftarrow R4 + 3$
Displacement	Add R4,100(R1)	$R4 \leftarrow R4 + \text{Mem}[100 + R1]$
Register indirect	Add R4,(R1)	$R4 \leftarrow R4 + \text{Mem}[R1]$
Indexed / Base	Add R3,(R1+R2)	$R3 \leftarrow R3 + \text{Mem}[R1 + R2]$
Direct or absolute	Add R1,(1001)	$R1 \leftarrow R1 + \text{Mem}[1001]$
Memory indirect	Add R1,@(R3)	$R1 \leftarrow R1 + \text{Mem}[\text{Mem}[R3]]$
Auto-increment	Add R1,(R2)+	$R1 \leftarrow R1 + \text{Mem}[R2]; R2 \leftarrow R2 + d$
Auto-decrement	Add R1,-(R2)	$R2 \leftarrow R2 - d; R1 \leftarrow R1 + \text{Mem}[R2]$
Scaled	Add R1,100(R2)[R3]	$R1 \leftarrow R1 + \text{Mem}[100 + R2 + R3 * d]$

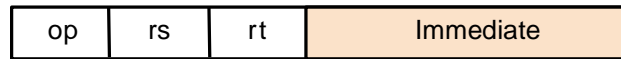
*Why Auto-increment/decrement? Scaled?*

# Endereçamento de desvio

- Especificação do endereço de desvio:
  - Absoluto:
    - ✕ **PC**  **Endereço de Desvio**
    - restrito a algumas funções do S.O.
    - implícito: vetor de interrupções
  - Relativo (PC = endereço base):
    - ✕ **PC**  **PC + Deslocamento (instrução)**
    - permite relocação
    - codificação econômica (poucos bits para o deslocamento)

# Modos do MIPS

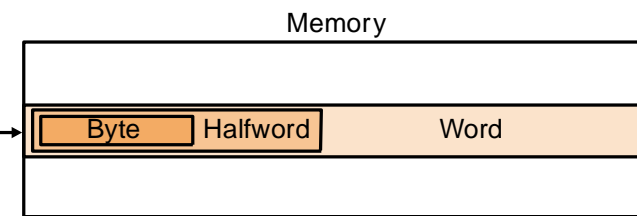
## 1. Immediate addressing



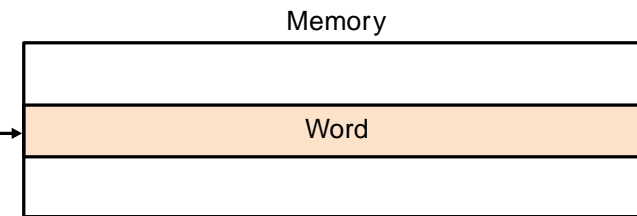
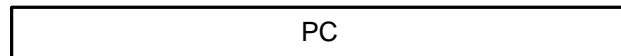
## 2. Register addressing



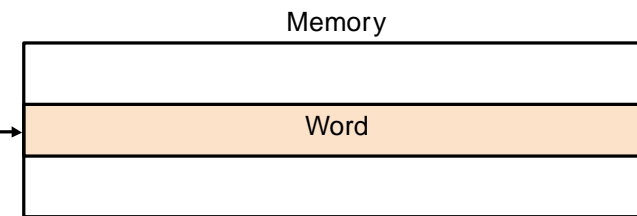
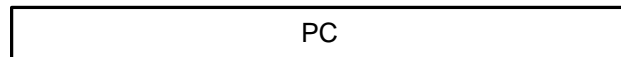
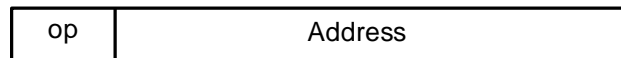
## 3. Base addressing



## 4. PC-relative addressing



## 5. Pseudodirect addressing



# Resumindo

---



- Organização de um computador
- Definição de arquitetura
  - Tipos de Dados
    - Inteiros
    - Booleanos
    - Ponto-Flutuante
  - Formato das instruções
  - Conjunto de registradores



# Resumindo

---



- ...Definição de Arquitetura
  - Repertório de instruções
    - sobre o dado
      - movimentação
      - transformação
      - codificação
      - aritméticas
      - lógicas
    - alteração do fluxo de execução
      - desvios condicionais
      - desvios incondicionais
      - Subrotinas



# Resumindo

---



- ...Definição de Arquitetura
  - Modos de Endereçamento: Dados
    - Operações Aritméticas e de Comparação:
      - Registrador
      - Imediato
    - Load/Store:
      - Base
  - Modos de Endereçamento: Instruções
    - Desvio Incondicional
      - (pseudo) direto
      - Indireto de registrador
    - Desvio Condicional
      - Relativo ao PC
    - Subrotina
      - (pseudo) direto

