

**Instituto de  
Computação**

UNIVERSIDADE ESTADUAL DE CAMPINAS



# Organização Básica de computadores e linguagem de montagem

## **Introdução à arquitetura RV32**

**Prof. Edson Borin**

<https://www.ic.unicamp.br/~edson>

Institute of Computing - UNICAMP

# RISC-V

ISA (Instruction Set Architecture) RISC moderna

- Introduzida em 2011

ISA aberta! (uso livre e livre de *royalties*)

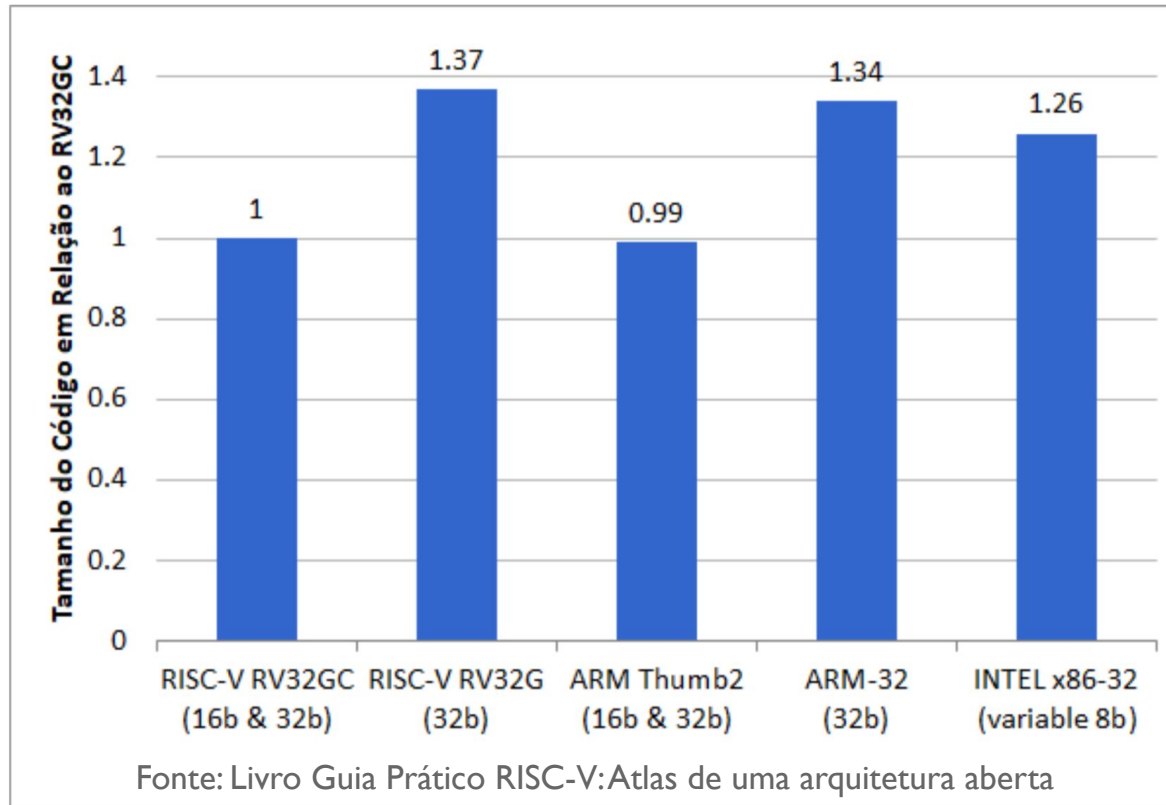
Funcionalidades e características desenvolvidas com base nos acertos e erros de ISAs que já estão no mercado há mais de 30 anos! (x86 e ARM)

- Mais simples do que ARM e x86
- Veja figura 2.7 do livro “Guia Prático RISC-V: Atlas de uma arquitetura aberta”

# RISC-V

Mais simples do que ARM e x86

Tamanho relativo de programas do *benchmark* SPEC CPU2006 compilados com o GCC.



# RISC-V

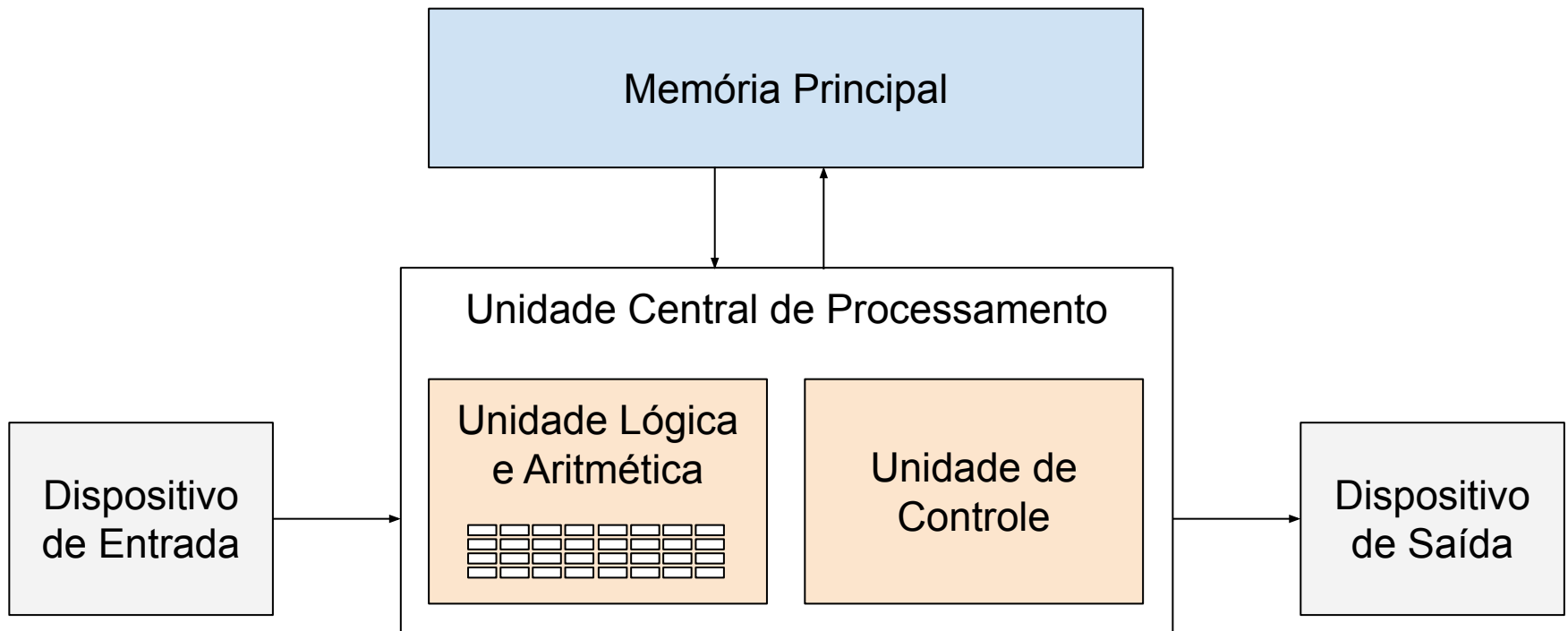
Mantida atualmente pela Fundação RISC-V

- [www.riscv.org](http://www.riscv.org)

Fundação aberta e sem fins lucrativos

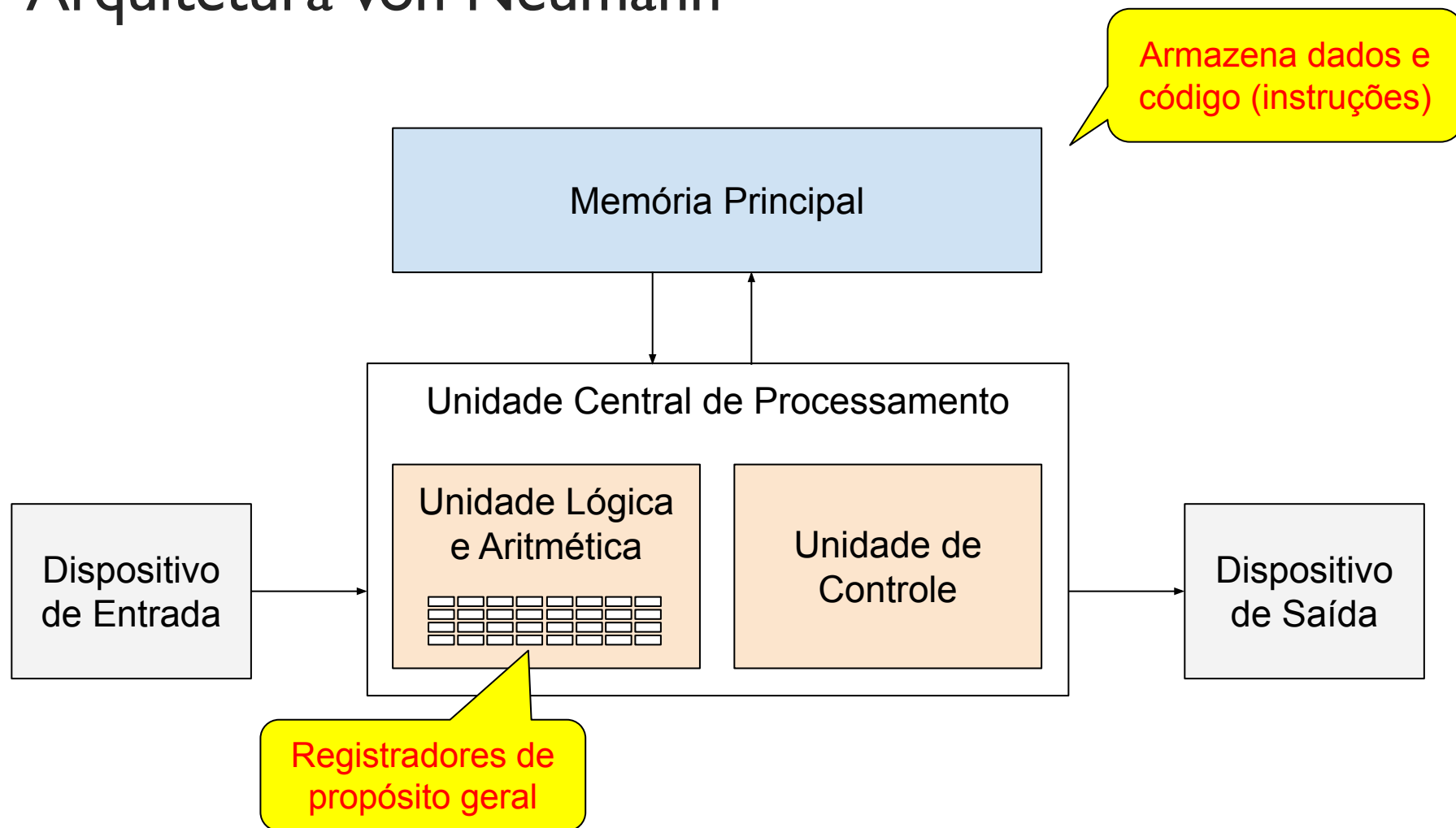
# Arquitetura do RISC-V

## Arquitetura von Neumann



# Arquitetura do RISC-V

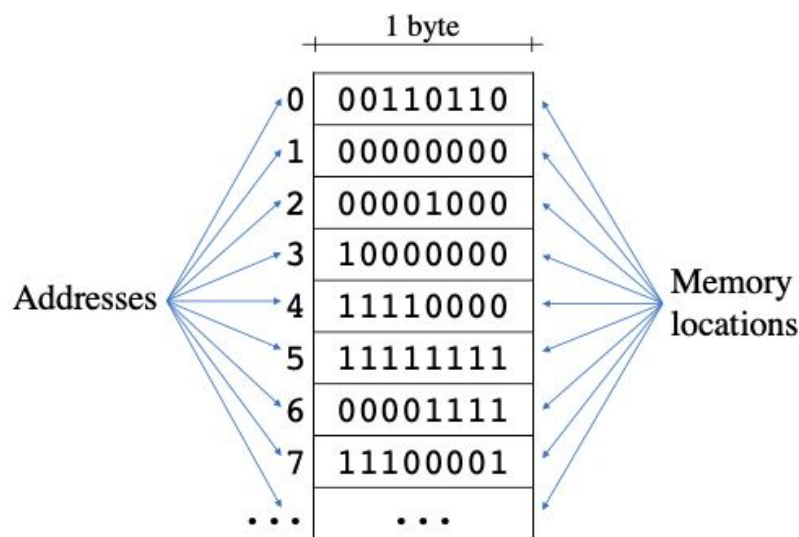
## Arquitetura von Neumann



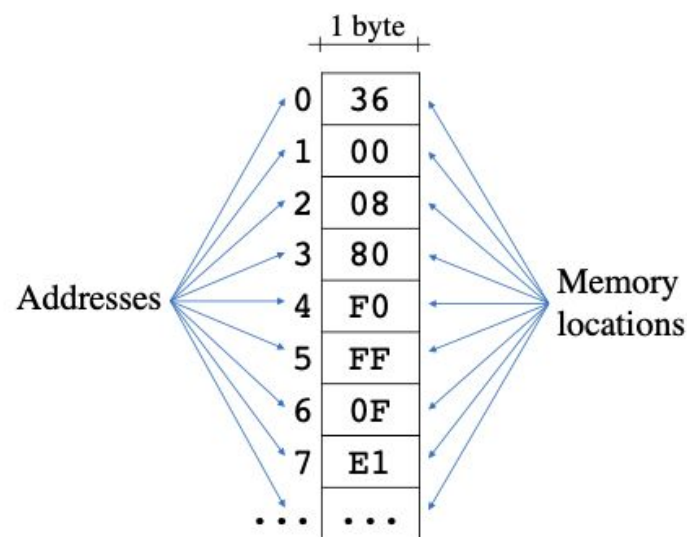
# Arquitetura do RISC-V

## Memória endereçada a *bytes*

- Cada palavra de memória armazena 1 *byte*
- Tipos de dados maiores do que 1 *byte* ocupam múltiplas palavras de memória, consecutivas.



(a)



(b)

# Arquitetura do RISC-V

Diversos conjuntos de instruções:

- **RV32I**: Conjunto base de 32 *bits* com instruções para operações com números inteiros.
- **RV32M**: Instruções de multiplicação e divisão
- **RV32F** e **RV32D**: Instruções de ponto-flutuante
- **RV32A**: Instruções atômicas
- **RV32C**: Instruções compactas, de 16 *bits*
- **RV32V**: Instruções vetoriais (SIMD)



# Arquitetura RV32

Neste curso focaremos no conjunto RV32IM

- Conjunto base de 32 *bits* + instruções para multiplicação e divisão de números inteiros
- Instruções de movimentação de dados (*load* e *store*), operações lógicas e aritméticas, comparação de valores, saltos condicionais e saltos incondicionais, chamadas de funções, ...

# Arquitetura RV32

## Tipos básicos de dados da arquitetura

- `byte`: **1 byte**
- `unsigned byte`: **1 byte (sem sinal)**
- `halfword`: **2 bytes**
- `unsigned halfword`: **2 bytes (sem sinal)**
- `word`: **4 bytes**
- `unsigned word`: **4 bytes (sem sinal)**

# Arquitetura RV32

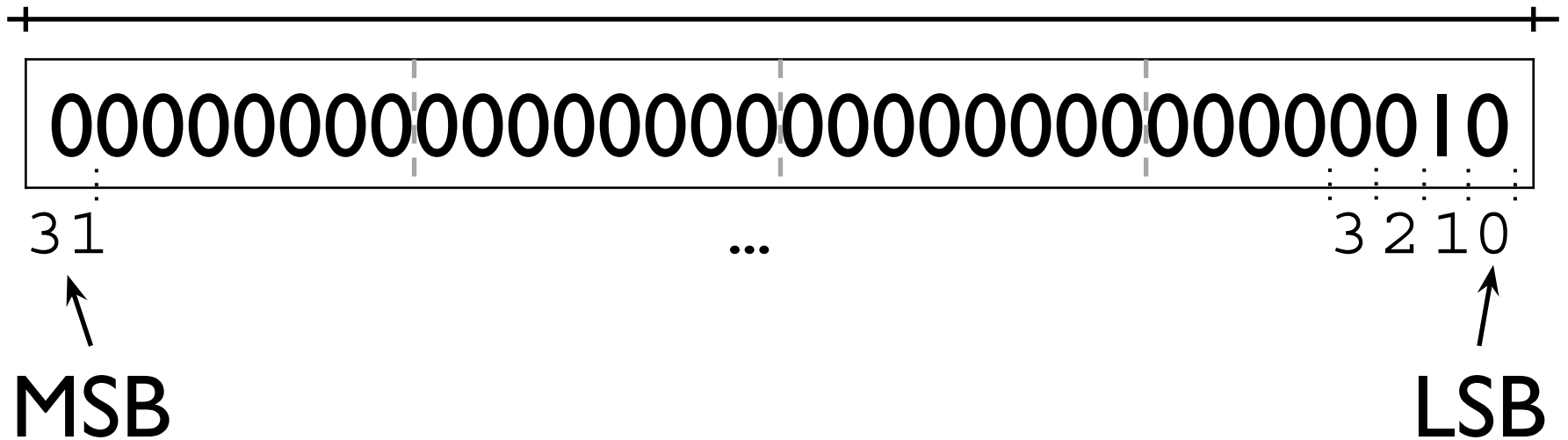
Mapeamento de tipos da linguagem `C` para tipos básicos de dados na arquitetura RV32

C datatype	RV32I native datatype	size in bytes
bool	byte	1
char	byte	1
unsigned char	unsigned byte	1
short	halfword	2
unsigned short	unsigned halfword	2
int	word	4
unsigned int	unsigned word	4
long	word	4
unsigned long	unsigned word	4
void*	unsigned word	4

# Arquitetura RV32

Registradores

32 bits



# Arquitetura RV32

## Registradores

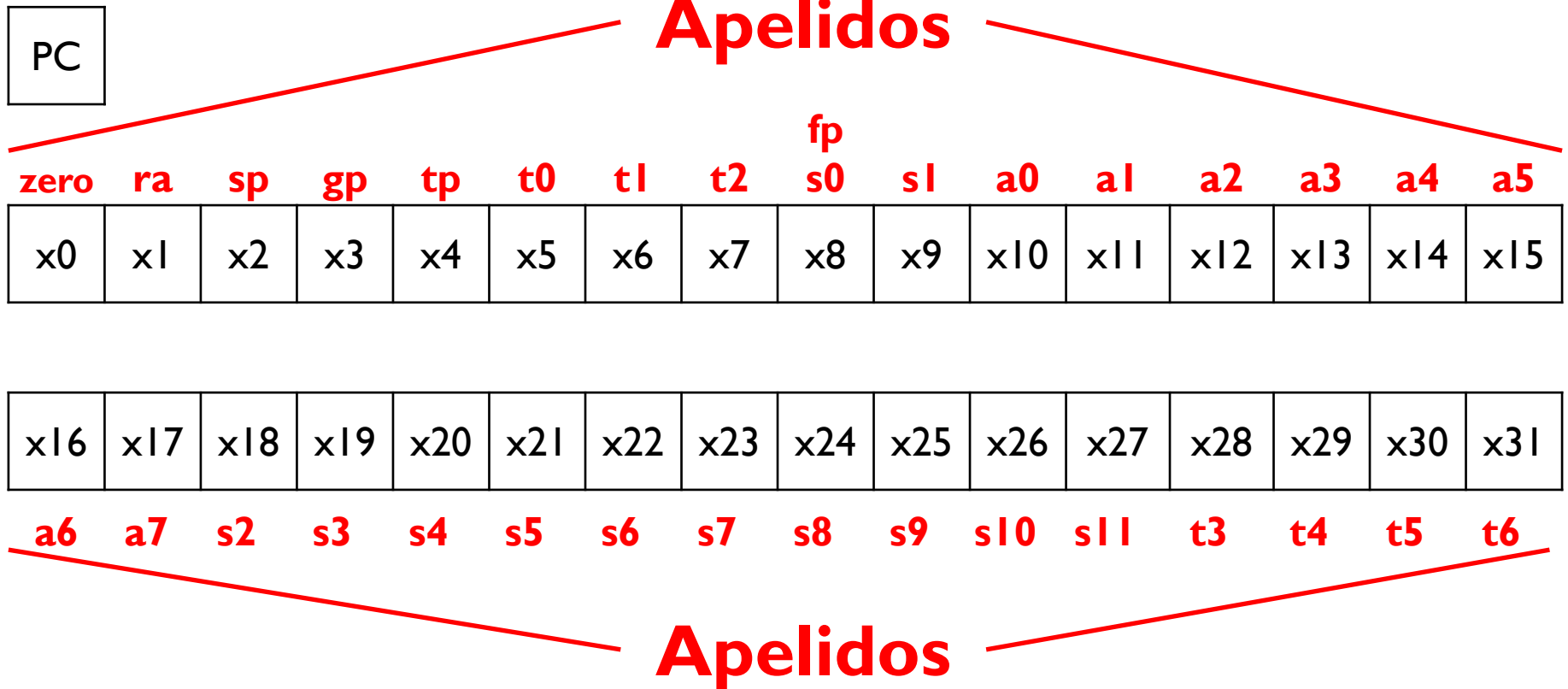
PC

x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15
----	----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----

x16	x17	x18	x19	x20	x21	x22	x23	x24	x25	x26	x27	x28	x29	x30	x31
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

# Arquitetura RV32

## Registradores



# Arquitetura RV32

## Registradores

Apelido	Significado
pc	<i>Program Counter</i> (Apontador de programa)
a0 , a1	Argumentos de função / retorno de função
a2 - a7	Argumentos de função
s0 - s11	Registrador salvo
t0 - t6	Temporário
zero	Contém sempre o valor 0 (zero)
ra	Endereço de retorno
sp	Ponteiro de pilha
gp	Ponteiro global
tp	Ponteiro de <i>thread</i>

# Arquitetura RV32

**Arquitetura Load/Store:** Os valores têm que ser carregados nos registradores antes de realizar-se operações.

- Não há instruções que operam diretamente em valores na memória!

```
lw    a5, 0(a0)    # a5 <= Mem[a0]
```

```
add   a6, a5, a5   # a6 <= a5+a5
```

```
sw    a6, 0(a0)    # Mem[a0] <= a6
```



# Arquitetura RV32

## Arquitetura Load/Store

Exemplo.



PC →

```
lw  a5, (a0)
add a6, a5, a5
sw  a6, (a1)
```

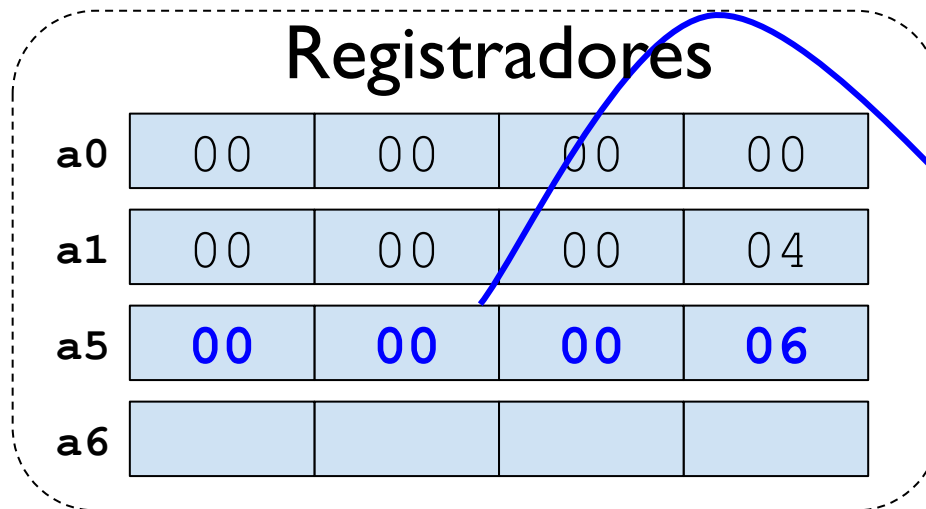
Memória

Valor	
06	0
00	1
00	2
00	3
00	4
00	5
00	6
00	7
...	...

# Arquitetura RV32

## Arquitetura Load/Store

Exemplo.



Memória

Valor	
06	0
00	1
00	2
00	3
00	4
00	5
00	6
00	7
...	...

PC



lw a5, (a0)

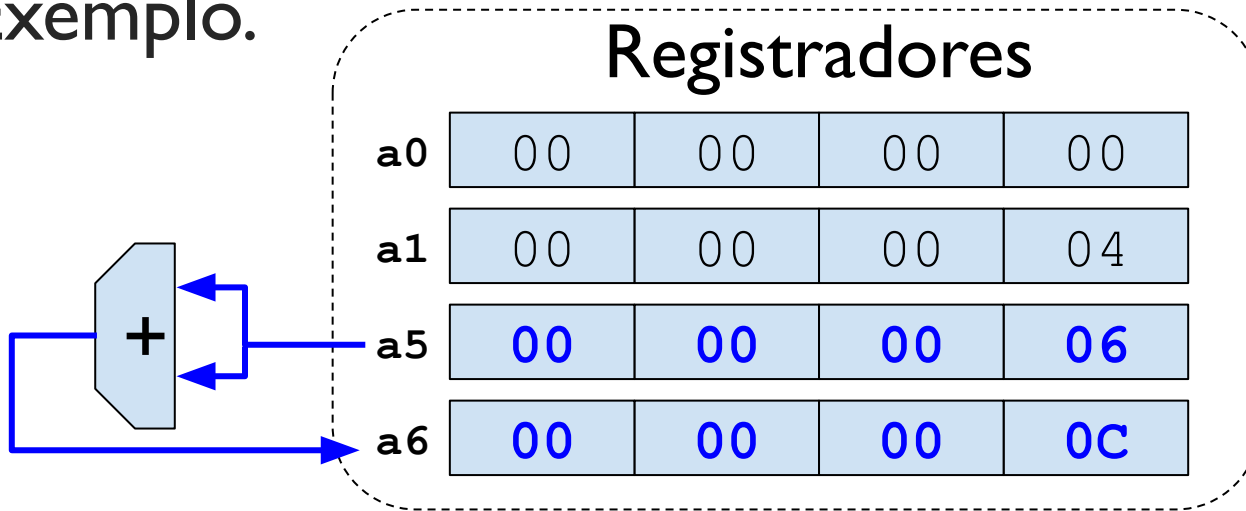
add a6, a5, a5

sw a6, (a1)

# Arquitetura RV32

## Arquitetura Load/Store

Exemplo.



Memória

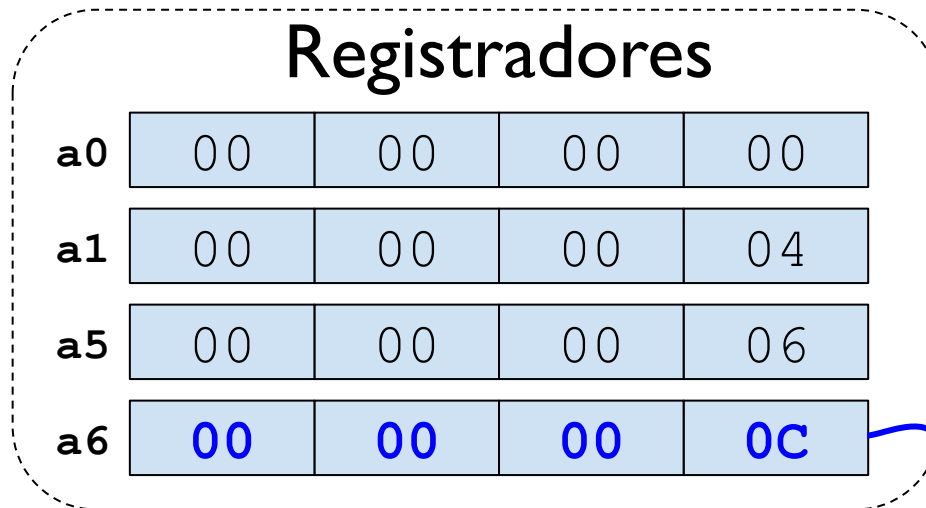
Valor	
06	0
00	1
00	2
00	3
00	4
00	5
00	6
00	7
...	...

```
PC → lw a5, (a0)
      add a6, a5, a5
      sw a6, (a1)
```

# Arquitetura RV32

## Arquitetura Load/Store

Exemplo.



```
lw  a5, (a0)
add a6, a5, a5
sw  a6, (a1)
```

PC →

Memória

Valor	
06	0
00	1
00	2
00	3
0C	4
00	5
00	6
00	7
...	...

# Instruções: Operações Lógicas

Instruções que realizam operações lógicas (e, ou, ou exclusivo)

Formato: <MNE> rd, rs1, rs2

- `and a0, a2, s2` # `a0 <= a2 & s2`
- `or a1, a3, s2` # `a1 <= a3 | s2`
- `xor a2, a2, a1` # `a2 <= a2 ^ a1`

Formato: <MNE>i rd, rs1, imm

- `andi a0, a2, 3` # `a0 <= a2 & 3`
- `ori a1, a3, 4` # `a1 <= a3 | 4`
- `xori a2, a2, 1` # `a2 <= a2 ^ 1`

# Instruções: Deslocamento de *bits*

Instruções que deslocam os *bits* dos registradores para a esquerda ou para direita.

Formato: <MNE>i rd, rs1, shamt

- `slli a0, a2, 2` # `a0 <= a2 << 2`
- `srlr a1, a3, 1` # `a1 <= a3 >> 1`
- `srai a2, a2, 1` # `a2 <= a2 >>* 1`  
# *\*aritmético*

OBS: Podem ser utilizadas para multiplicar/dividir por potências de 2.

# Instruções: Deslocamento de *bits*

Multiplicando com instruções de deslocamento de *bits*

Multiplicar um número **inteiro** com (`int`) ou sem sinal (`unsigned`) por potência de 2:

- `slli a0, a2, 2` # `a0 <= a2 * 22`
- `slli a3, a3, 4` # `a3 <= a3 * 24`

OBS: Basta deslocar os *bits* para a esquerda.

# Instruções: Deslocamento de *bits*

Dividindo com instruções de deslocamento de *bits*

Dividir um número **inteiro sem sinal** (`unsigned`) por potência de 2:

- `srli a0, a2, 2` #  $a0 \leq a2 / 2^2$

Dividir um número **inteiro com sinal** (`int`) por potência de 2:

- `srai a0, a2, 2` #  $a0 \leq a2 / 2^2$

**OBS:** Note a diferença entre a divisão de números inteiros **com e sem sinal!**



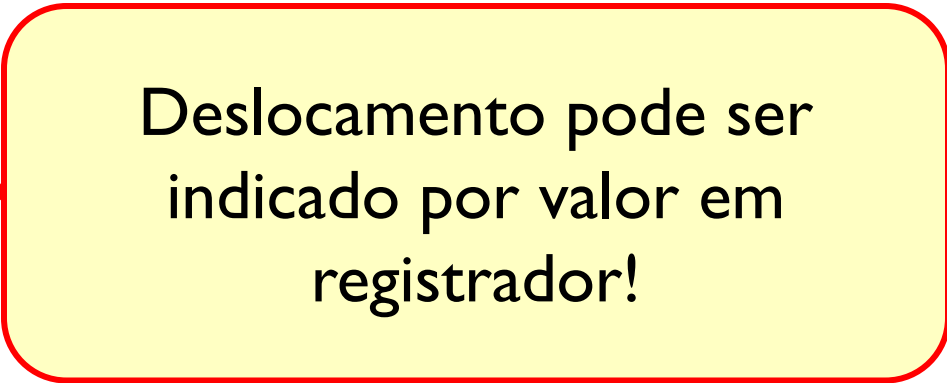
# Instruções: Deslocamento de *bits*

Formato: <MNE>i rd, rs1, shamt

- slli a0, a2, 2 # a0 <= a2 << 2
- srli a1, a3, 1 # a1 <= a3 >> 1
- srai a2, a2, 1 # a2 <= a2 >>\* 1  
# \*aritmético

Formato: <MNE> rd, rs1, **rs2**

- sll a0, a2, **s2**
- srl a1, a3, **s2**
- sra a2, a2, **a1**



Deslocamento pode ser indicado por valor em registrador!

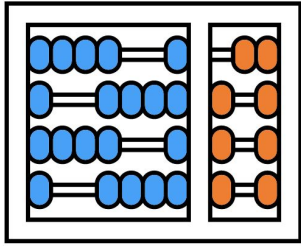
# Instruções: Operações Aritméticas

Instruções que realizam operações aritméticas (+, -, ...) com valores nos registradores

Formato: <MNE> rd, rs1, rs2

- `add a0, a2, t2`      # `a0 <= a2 + t2`
- `sub a1, t3, a0`      # `a1 <= t3 - a0`
- `mul a2, t1, a0`      # `a2 <= t1 * a0`
- `div{u} a3, t2, a1`    # `a3 <= t2 / a1`
- `rem{u} a4, t3, a2`    # `a4 <= t3 % a2`

**OBS:** Sufixo `{u}` deve ser usado para realizar operação de divisão/resto com números sem sinal (*unsigned*).



**Instituto de  
Computação**

UNIVERSIDADE ESTADUAL DE CAMPINAS



# Organização Básica de computadores e linguagem de montagem

## **Introdução à arquitetura RV32**

**Prof. Edson Borin**

<https://www.ic.unicamp.br/~edson>

Institute of Computing - UNICAMP

# Instruções: Operações Aritméticas

## Operações aritméticas com imediatos

- Imediatos: constantes codificadas diretamente na instrução

Formato: `<MNE>i rd, rs1, imm`

- `addi a0, a2, 10 # a0 <= a2 + 10`

**OBS:** Não existe a instrução `subi`, entretanto, é possível usar uma constante negativa para subtrair valores. Ex:

- `addi a0, a2, -10 # a0 <= a2 - 10`

# Instruções: Movimentação de dados

Instruções para copiar valores da memória p/ registradores.

Formato: <MNE> rd, imm(rs1)

- lw a0, imm(a2) # a0 <= Mem [a2+imm]
- lh a0, imm(a2) # a0 <= Mem [a2+imm]
- lhu a0, imm(a2) # a0 <= Mem [a2+imm]
- lb a0, imm(a2) # a0 <= Mem [a2+imm]
- lbu a0, imm(a2) # a0 <= Mem [a2+imm]

Instruções de load (l) carregam dados da memória para registradores. O sufixo (w, h, hu, b, bu) indica o tipo de dado!

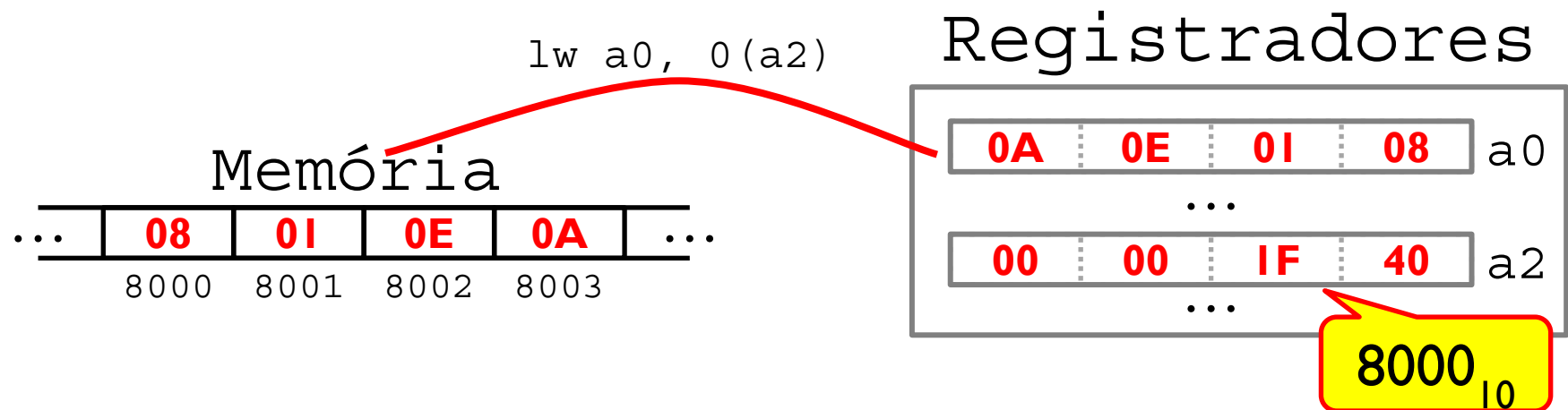
# Instruções: Movimentação de dados

## Load word

- `lw a0, imm(a2) # a0 <= Mem[a2+imm]`

Carrega um número de 32 *bits* (4 *bytes*) da memória.

Formato *little-endian*: O *byte* menos significativo é carregado do endereço  $a2+imm$  enquanto que o *byte* mais significativo é carregado do endereço  $a2+imm+3$ .



# Instruções: Movimentação de dados

## Load word

- `lw a0, imm(a2) # a0 <= Mem[a2+imm]`

Endereço  
de memória

Carre

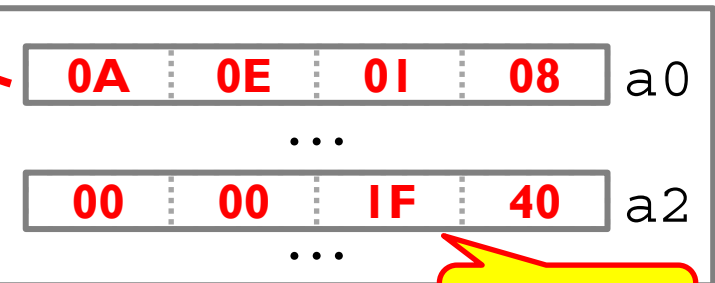
Forma

carreg

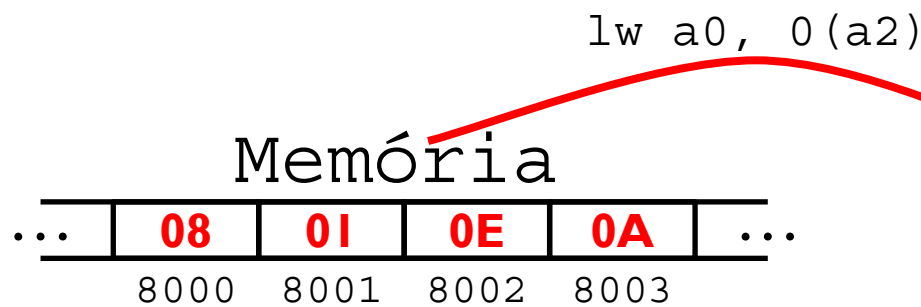
mais s

**lw deve ser usada quando carregarmos dados do tipo “int” ou “unsigned int” da memória!**

## Registradores



**8000<sub>10</sub>**

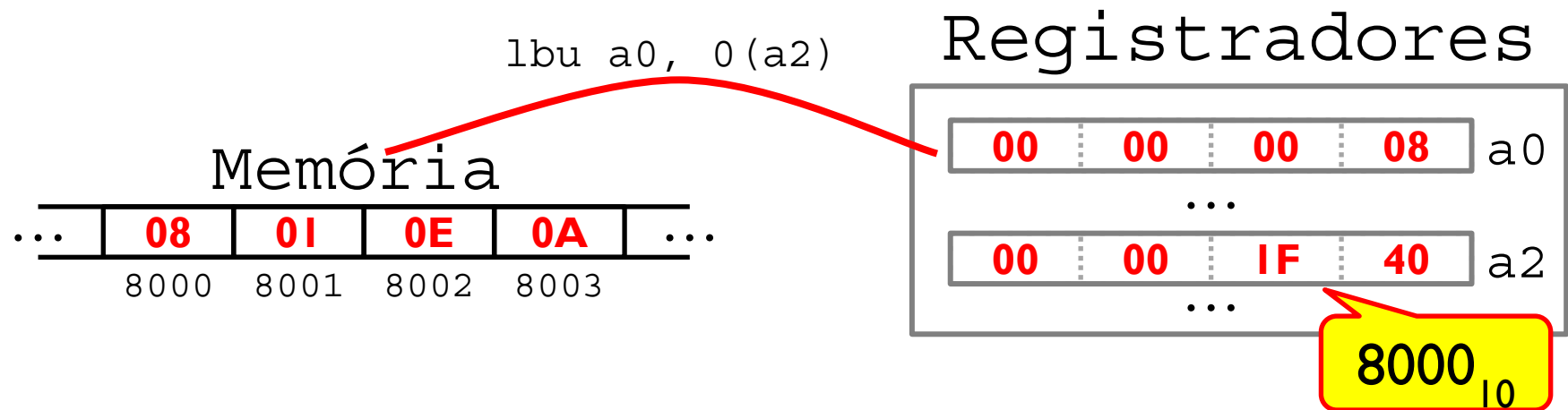


# Instruções: Movimentação de dados

## Load byte unsigned

- `lbu a0, imm(a2) # a0 <= Mem[a2+imm]`

Carrega um número de 8 *bits sem sinal* (1 *byte*) da memória. Como o registrador tem 32 *bits*, o restante é preenchido com zeros.





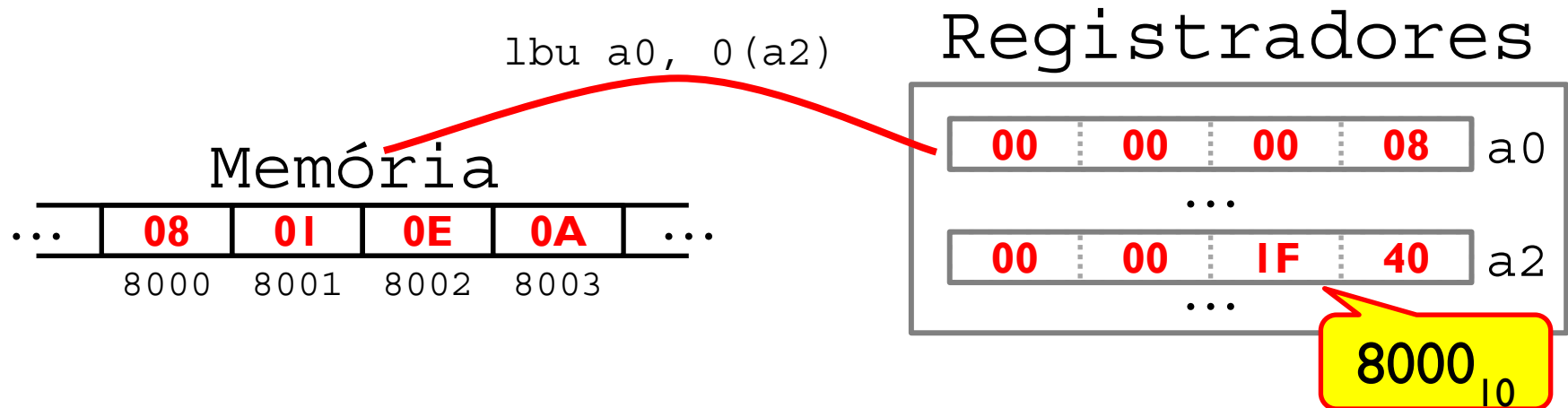
# Instruções: Movimentação de dados

## Load byte unsigned

- `lbu a0, imm(a2) # a0 <= Mem[a2+imm]`

Carrega um byte da memória.  
Como o byte é carregado sem sinal, ele é preenchido com zeros.

**lbu deve ser usada quando carregarmos dados do tipo “unsigned char” da memória!**



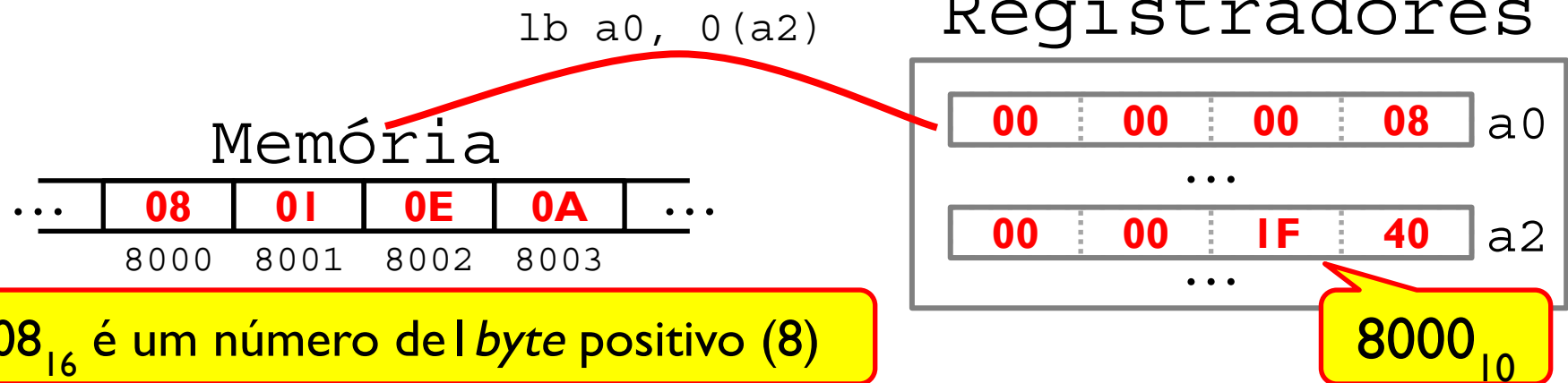
# Instruções: Movimentação de dados

## Load byte

- `lb a0, imm(a2) # a0 <= Mem[a2+imm]`

Carrega um número de 8 *bits com sinal* (1 *byte*) da memória.

Como o registrador tem 32 *bits*, o restante é preenchido com 0's caso o número seja positivo ou com 1's (caso seja negativo).



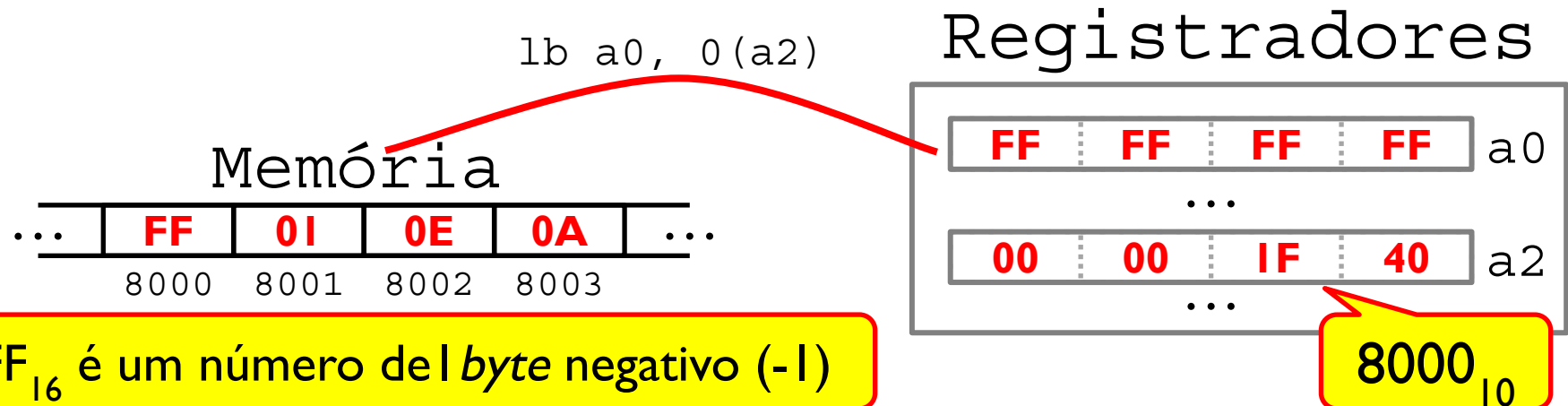
# Instruções: Movimentação de dados

## Load byte

- $1b \ a0, \ imm(a2) \ \# \ a0 \ \leq \ Mem[a2+imm]$

Carrega um número de 8 *bits com sinal* (1 *byte*) da memória.

Como o registrador tem 32 *bits*, o restante é preenchido com 0's caso o número seja positivo ou com 1's (caso seja negativo).



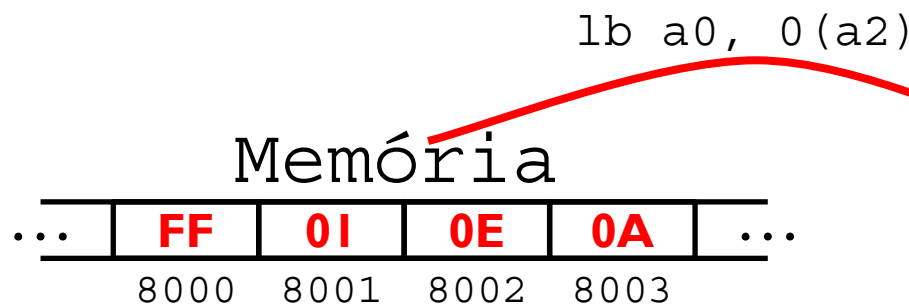
# Instruções: Movimentação de dados

## Load byte

- `lb a0, imm(a2) # a0 <= Mem[a2+imm]`

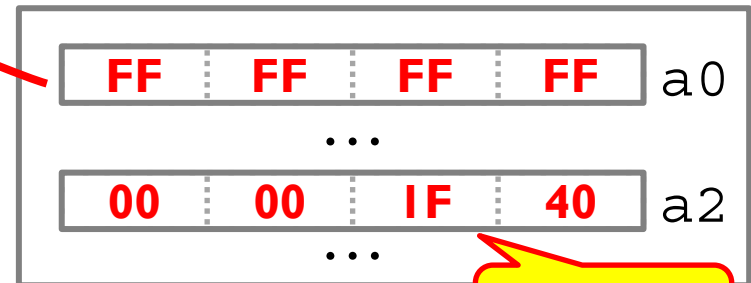
Carrega um byte da memória.  
Como o registro a0 é inicializado com zeros caso o número de deslocamento (imm) seja negativo).

**lb deve ser usada quando carregarmos dados do tipo “char” da memória!**



**FF<sub>16</sub> é um número de 1 byte negativo (-1)**

## Registradores



**8000<sub>10</sub>**

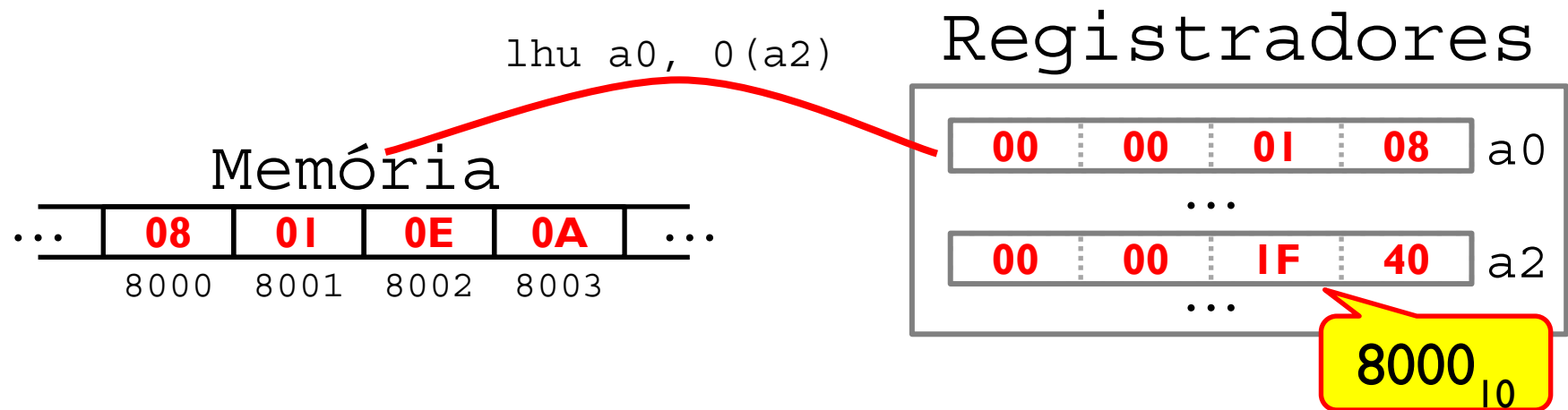
# Instruções: Movimentação de dados

## Load halfword unsigned

- `lhu a0, imm(a2) # a0 <= Mem[a2+imm]`

Carrega um número de 16 *bits sem sinal* (2 *bytes*) da memória.

Como o registrador tem 32 *bits*, o restante é preenchido com zeros.



# Instruções: Movimentação de dados

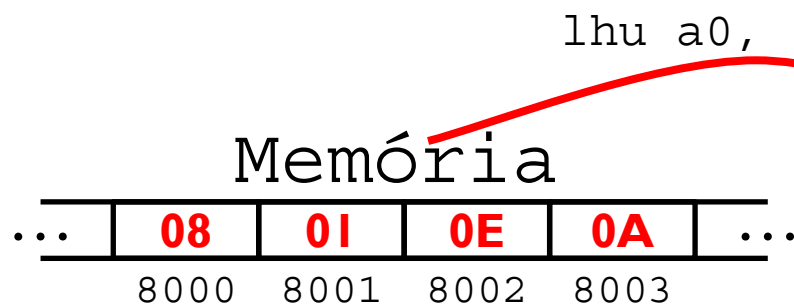
## Load halfword unsigned

- `lhu a0, imm(a2) # a0 <= Mem[a2+imm]`

Carrega um  
memória.

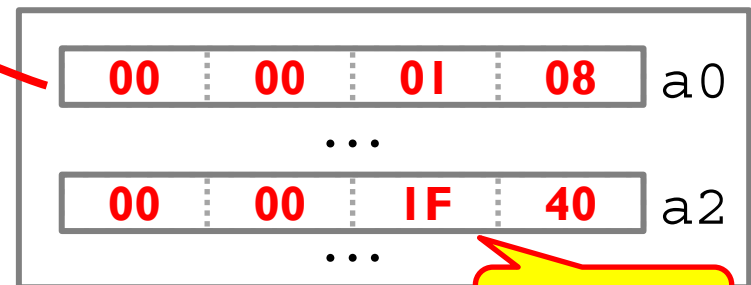
Como o r  
zeros.

**lhu deve ser usada quando carregarmos dados do tipo “unsigned short” da memória!**



`lhu a0, 0(a2)`

## Registradores



8000<sub>10</sub>

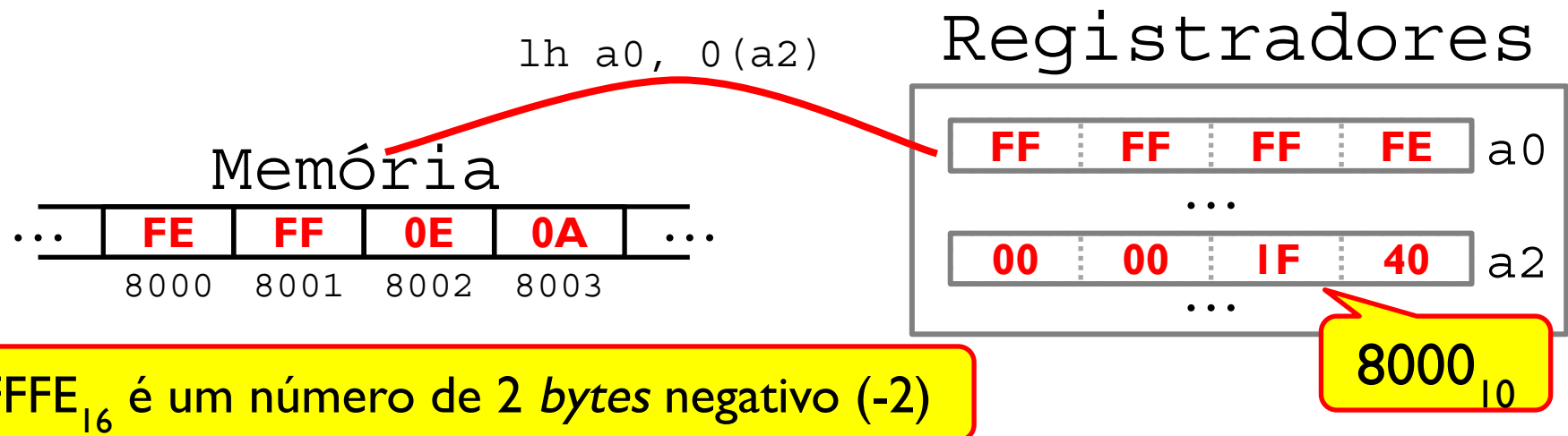
# Instruções: Movimentação de dados

## Load halfword

- `lh a0, imm(a2) # a0 <= Mem[a2+imm]`

Carrega um número de 16 *bits com sinal* (2 *bytes*) da memória.

Como o registrador tem 32 *bits*, o restante é preenchido com 0's caso o número seja positivo ou com 1's (caso seja negativo).



# Instruções: Movimentação de dados

## Load halfword

- `lh a0, imm(a2) # a0 <= Mem[a2+imm]`

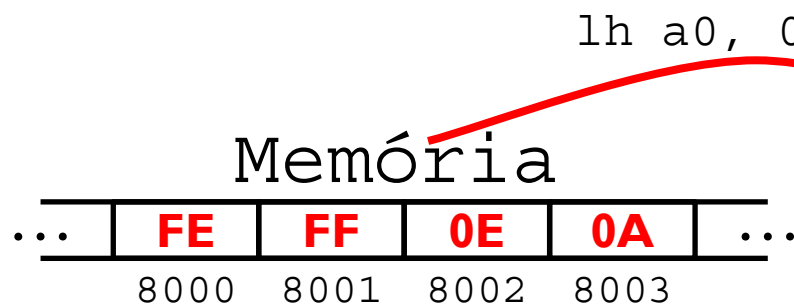
Carrega um  
memória.

Como o re  
0's caso o n

`lh` deve ser usada quando carregarmos dados do tipo “short” da memória!

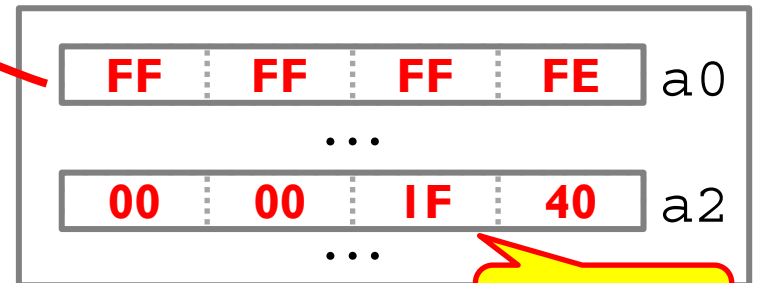
Endereço  
de memória

) da  
chido com  
(a negativo).



`lh a0, 0(a2)`

## Registradores



$FFFE_{16}$  é um número de 2 bytes negativo (-2)

$8000_{10}$



# Instruções: Movimentação de dados

Instruções para copiar valores de registradores p/ memória.

Formato: <MNE> rs1, imm(rs2) Endereço de memória  
= rs2 + imm

- sw a0, imm(a2) # Mem[a2+imm] <= a0
- sh a0, imm(a2) # Mem[a2+imm] <= a0
- sb a0, imm(a2) # Mem[a2+imm] <= a0

Instruções de store (l) armazenam dados de registradores na memória. O sufixo (w, h, b) indica o tipo de dado!

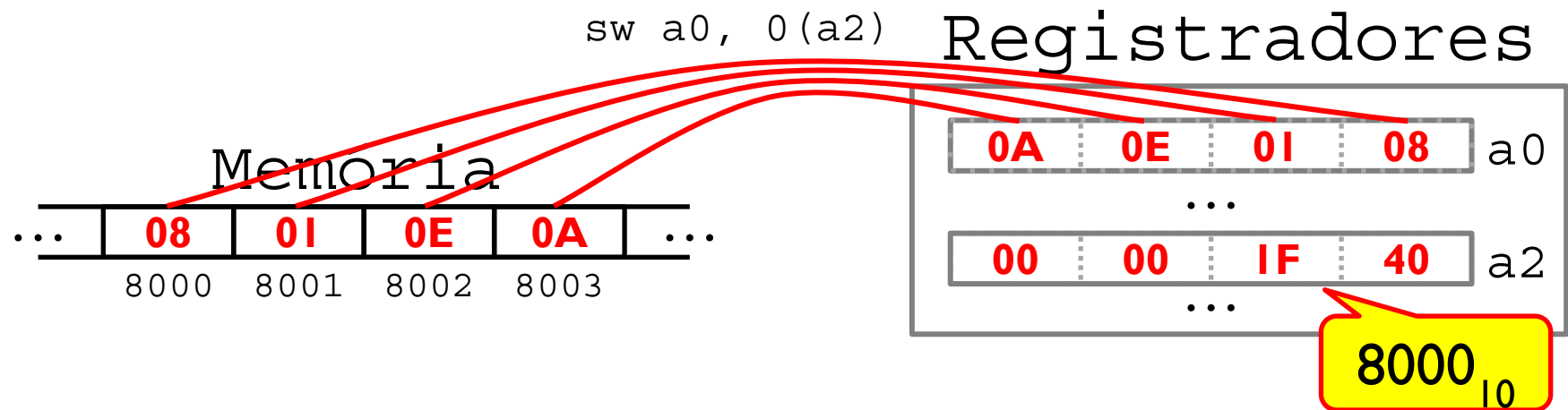
# Instruções: Movimentação de dados

## Store word

- `sw a0, imm(a2) # Mem[a2+imm] <= a0`

Grava um número de 32 *bits* (4 *bytes*) na memória.

Formato *little-endian*: O *byte* menos significativo é gravado no endereço  $a2+imm$  enquanto que o *byte* mais significativo é gravado no endereço  $a2+imm+3$ .



# Instruções: Movimentação de dados

## Store word

- `sw a0, imm(a2) # Mem[a2+imm] <= a0`

Grava

Form

ende

grava

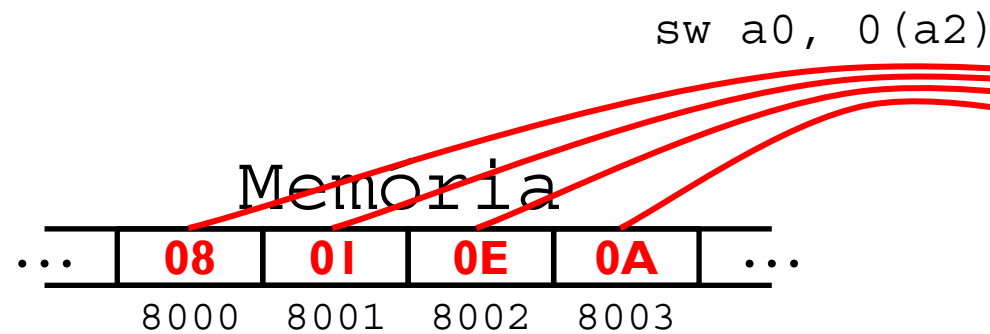
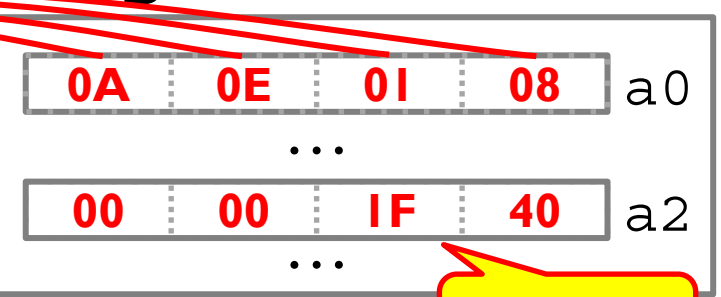
**sw deve ser usada quando gravarmos dados do tipo “int” ou “unsigned int” na memória!**

Endereço de memória



# Mem[a2+imm] <= a0

Registradores



8000<sub>10</sub>

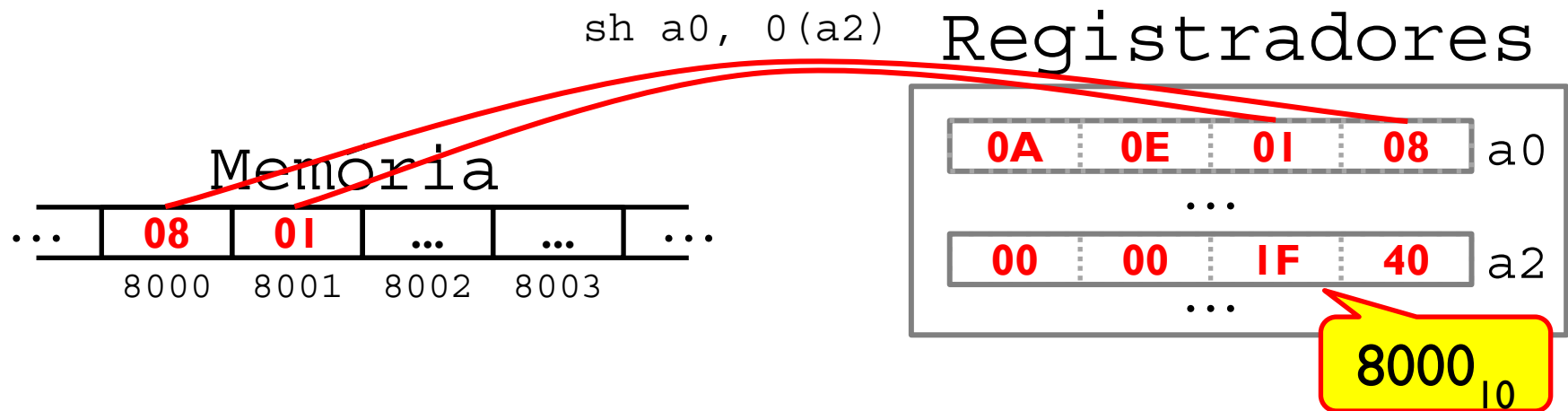
# Instruções: Movimentação de dados

## Store half word

- `sh a0, imm(a2) # Mem[a2+imm] <= a0`

Grava um número de 16 *bits* (2 *bytes*) na memória.

Formato *little-endian*: O *byte* menos significativo é gravado no endereço  $a2+imm$  enquanto que o *byte* mais significativo é gravado no endereço  $a2+imm+1$ .



# Instruções: Movimentação de dados

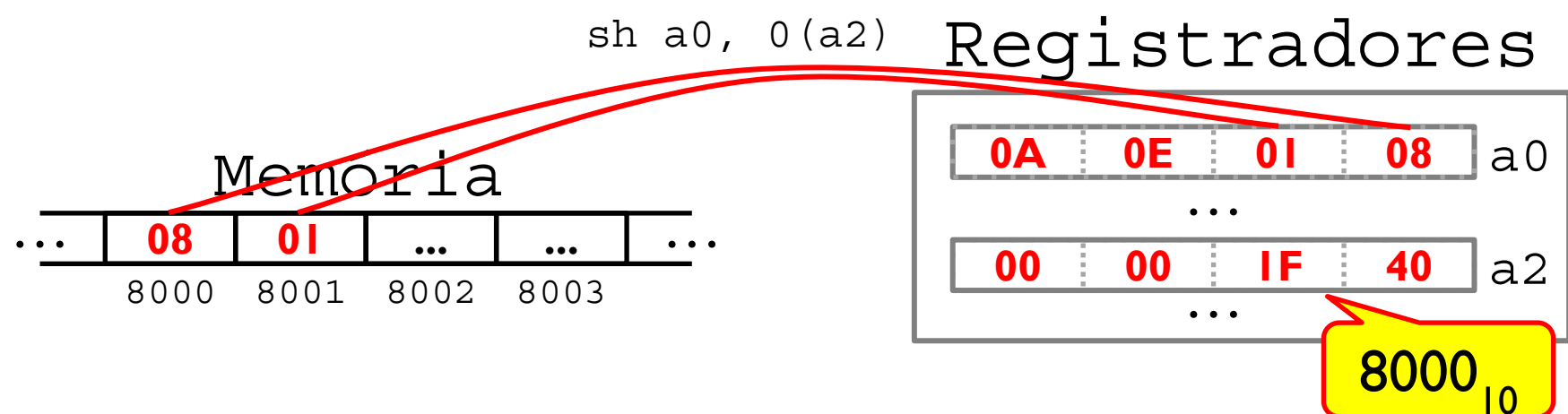
## Store half word

- `sh a0, imm(a2) # Mem[a2+imm] <= a0`

Grava  
Formato  
endere  
gravado

sh deve ser usada quando gravarmos dados do tipo “short” ou “unsigned short” na memória!

Endereço de memória



# Instruções: Movimentação de dados

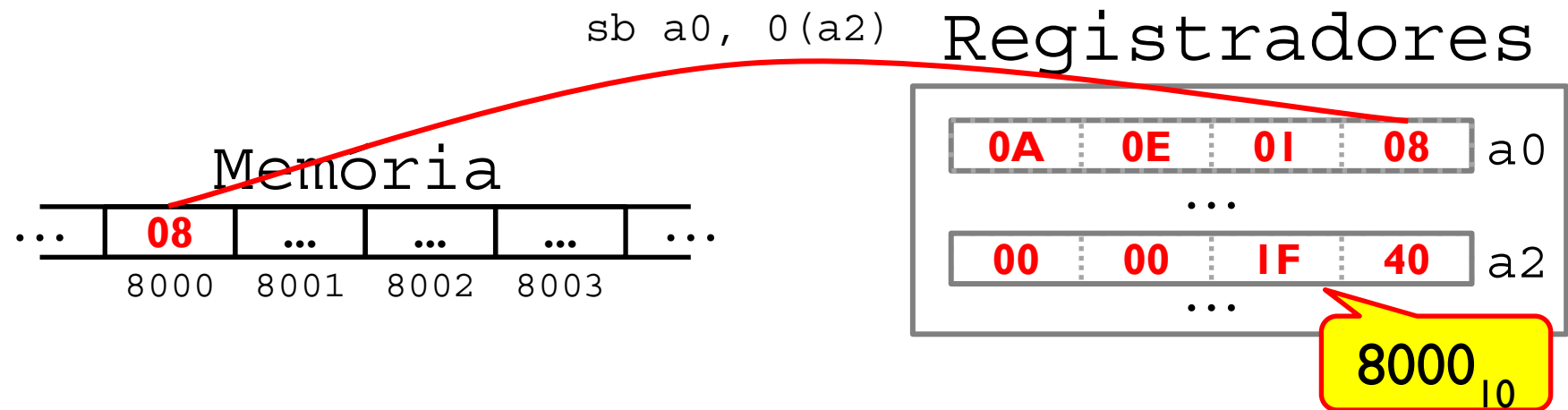
## Store byte

- `sb a0, imm(a2) # Mem[a2+imm] <= a0`

Grava um número de 8 *bits* (1 *byte*) na memória.

Formato *little-endian*: O *byte* menos significativo é gravado no endereço  $a2+imm$ .

Endereço  
de memória



# Instruções: Movimentação de dados

## Store byte

- `sb a0, imm(a2) # Mem[a2+imm] <= a0`

Grava

Forma  
endere

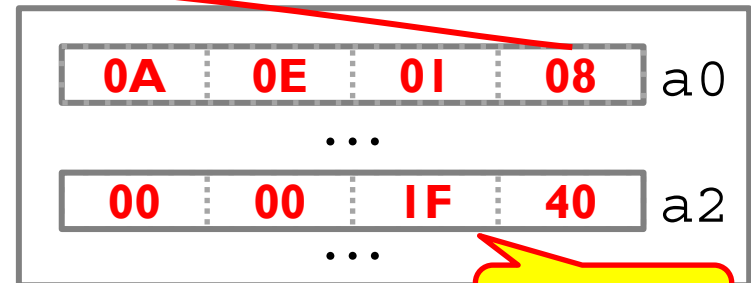
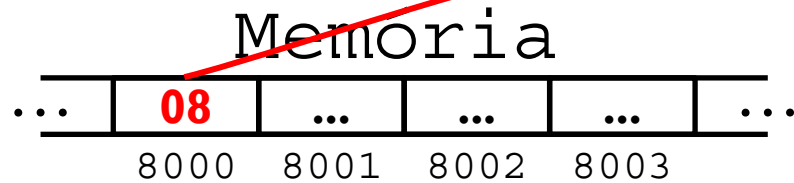
**sb deve ser usada quando gravarmos dados do tipo “char” ou “unsigned char” na memória!**

Endereço  
de memória



`sb a0, 0(a2)`

Registradores



8000<sub>10</sub>

# Instruções: Controle de fluxo

O fluxo normal de execução consiste em executar instruções uma após a outra na mesma ordem em que elas são organizadas na memória.

**Instruções de controle de fluxo são instruções capazes de mudar o fluxo normal de execução.**

Exemplo:

```
beq a0, a2, next_item
add a0, a0, a1
next_item:
addi a0, s1, -1
```



# Instruções: Controle de fluxo

**Instruções de controle de fluxo condicionais** são instruções que mudam o fluxo normal de execução apenas sob certas condições.

Exemplo:

```
beq a0, a2, next_item
add a0, a0, a1
next_item:
addi a0, s1, -1
```

No caso acima, a instrução *branch equal* (`beq`) só "salta" (**desvia o fluxo de execução**) para o rótulo `next_item` se o valor em `a0` for igual ao valor em `a2`.

# Instruções: Controle de fluxo

## Instruções de controle de fluxo condicionais

Formato: <MNE> rs1, rs2, rot

- beq a0, a2, L # Salta se a0 = a2
- bne a0, a2, L # Salta se a0 != a2
- blt a0, a2, L # Salta se a0 < a2
- bge a0, a2, L # Salta se a0 >= a2
- bltu a0, a2, L # Salta se a0 <\* a2
- bgeu a0, a2, L # Salta se a0 >=\* a2

\* Comparação sem sinal!

# Instruções: Controle de fluxo

**Instruções de controle de fluxo incondicionais** são instruções que sempre mudam o fluxo normal de execução.

Exemplo:

```
jal foo
add a0, a0, a1
...
foo:
sub a0, s1, 1
```

No caso acima, a instrução *jump and link* (`jal`) salta para o rótulo `foo` e a próxima instrução a ser executada é a instrução `sub`.

# Instruções: Controle de fluxo

## Instruções de controle de fluxo incondicionais

Formato: <MNE> rd, rot

`jal a0, L # "Faz o link" e salta p/ L`

- Grava  $PC+4$  em `a0` e depois salta p/ o rótulo `L`
- $PC+4$  é o endereço da instrução subsequente à instrução sendo executada (`jal`)
- `jal` é utilizada para invocar rotinas.  $PC+4$  é o local para onde o fluxo deve retornar após a execução da rotina

# Instruções: Controle de fluxo

## Instruções de controle de fluxo incondicionais

Invocando rotinas com `jal`

```
    ...  
8000    jal ra, foo      # Invoca foo  
8004    sub a3, a1, a0  
    ...  
    ...  
9000    foo:           # Função foo  
9000    add a0, a0, a1  
9004    jalr x0, ra, 0  # Retorna de foo
```

# Instruções: Controle de fluxo

## Instruções de controle de fluxo incondicionais

Invocando rotinas com `jal`

Grava 8004 (PC+4) no registrador `ra` e salta para `foo` (9000)

```
    ...  
8000    jal ra, foo        # Invoca foo  
8004    sub a3, a1, a0  
    ...  
    ...  
9000    foo:              # Função foo  
9000        add a0, a0, a1  
9004    jalr x0, ra, 0    # Retorna de foo
```

# Instruções: Controle de fluxo

## Instruções de controle de fluxo incondicionais

Formato: <MNE> rd, rs1, imm

`jalr a0, a1, 0 # "Faz o link" e salta p/ a1+0`

- Grava PC+4 em a0 e depois salta p/ a1+0
- PC+4 é o endereço da instrução subsequente à instrução sendo executada (`jalr`)
- `jalr` é geralmente utilizada para retornar de rotinas saltando para o endereço de retorno que foi armazenado em um registrador pela instrução `jal`.

# Instruções: Controle de fluxo

## Instruções de controle de fluxo incondicionais

Retornando de rotinas com `jalr`

```
...  
8000    jal ra, foo      # Invoca foo  
8004    sub a3, a1, a0  
...  
...  
9000    foo:            # Função foo  
9000    add a0, a0, a1  
9004    jalr x0, ra, 0  # Retorna de foo
```

`jal` grava 8004 (PC+4)  
no registrador `ra` e salta  
para `foo` (9000)

Salta para o endereço 8004 (armazenado em `ra`)



# Instruções: Controle de fluxo

## Instruções de controle de fluxo incondicionais

Retornando de rotinas com `jalr`

```
...  
8000    jal ra, foo      # Invoca foo  
8004    sub a3, a1, a0  
...  
...  
9000    foo:  
9000    add a0, a0  
9004    jalr x0, ra, 0  # Retorna de foo
```

`jal` grava 8004 (PC+4) no registrador `ra` e salta para `foo` (9000)

Endereço 9008 (PC+4) é descartado (escrita no reg. `x0`)

Salta para o endereço 8004 (armazenado em `ra`)

# Instruções: Controle de fluxo

## Instruções de salto direto vs salto indireto

**Salto direto:** o endereço alvo está codificado na própria instrução.

- Ex: `jal ra, foo #Salta p/ foo`

**Salto indireto:** o endereço alvo é computado a partir de um valor que está em um registrador de propósito geral.

- Ex: `jalr x0, ra, 0 # Salta p/ ra+0`

# Instruções: Controle de fluxo

## Invocar o sistema operacional

`ecall` # Invoca o sistema operacional

Exemplo - Chamando a chamada de sistema (*syscall*) *write*:

```
.data
msg: .asciz "Assembly rocks" # String

.text
_start:
li a0, 1      # a0: File descriptor = 1 (stdout)
la a1, msg    # a1: endereço do buffer msg
li a2, 14     # a2: tamanho do buffer msg (14 bytes)
li a7, 64     # Código da chamada (write = 64)
ecall        # Invocar o SO
```

# Instruções: Comparação de Valores

Instruções que realizam comparações de valores e gravam o resultado em um registrador.

Formato: <MNE> rd, rs1, rs2

- `slt a0, a2, t2 # a0 = (a2 < t2) ? 1 : 0`
- `sltu a1, t3, a0 # a1 = (t3 < a0) ? 1 : 0`

Formato: <MNE>i rd, rs1, imm

- `slti a0, a2, 10 # a0 = (a2 < 10) ? 1 : 0`
- `sltui a1, t3, 25 # a1 = (t3 < 25) ? 1 : 0`

OBS: Sufixo `u` e `ui` indicam comparação sem sinal (unsigned)

# Codificação das instruções RV32

Cada instrução é codificada com 32 *bits*, em um dos seguintes formatos

	31	27	26	25	24	20	19	15	14	12	11	7	6	0	
<b>R</b>	funct7				rs2			rs1		funct3		rd		opcode	
<b>I</b>	imm[11:0]						rs1		funct3		rd		opcode		
<b>S</b>	imm[11:5]				rs2			rs1		funct3		imm[4:0]		opcode	
<b>B</b>	imm[12 10:5]				rs2			rs1		funct3		imm[4:1 11]		opcode	
<b>U</b>	imm[31:12]										rd		opcode		
<b>J</b>	imm[20 10:1 11 19:12]										rd		opcode		

**R:** sll, srl, sra, add, sub, xor, or, and, slt, sltu

**I:** slli, srli, srai, addi, xori, ori, andi, slti, sltiu, jalr, lw, lh, lb

**U:** lui, auipc

**B:** beq, bne, blt, bge, bltu, bgeu

**J:** jal

**S:** sw, sh, sb

# Codificação das instruções RV32

Cada instrução é codificada com 32 *bits*, em um dos seguintes formatos

	31	27	26	25	24	20	19	15	14	12	11	7	6	0		
<b>R</b>	funct7						rs2			rs1		funct3		rd		opcode
<b>I</b>	imm[11:0]						rs1			funct3		rd		opcode		
<b>S</b>	imm[11:5]			rs2			rs1		funct3		imm[4:0]		opcode			
<b>B</b>	imm[12 10:5]			rs2			rs1		funct3		imm[4:1 11]		opcode			
<b>U</b>	imm[31:12]										rd		opcode			
<b>J</b>	imm[20 10:1 11 19:12]										rd		opcode			

**add** rd, rs1, rs2  $x[rd] = x[rs1] + x[rs2]$

*Add.* R-type, RV32I and RV64I.

Adiciona o registrador  $x[rs2]$  ao registrador  $x[rs1]$  e grava o resultado em  $x[rd]$ . O overflow aritmético é ignorado.

*Formas comprimidas:* **c.add** rd, rs2; **c.mv** rd, rs2

31	25	24	20	19	15	14	12	11	7	6	0	
0000000			rs2			rs1		000		rd		0110011

# Codificação das instruções RV32

Cada instrução é codificada com 32 *bits*, em um dos seguintes formatos

	31	27	26	25	24	20	19	15	14	12	11	7	6	0	
<b>R</b>	funct7					rs2		rs1		funct3		rd		opcode	
<b>I</b>	imm[11:0]					rs1		funct3		rd		opcode			
<b>S</b>	imm[11:5]					rs2		rs1		funct3		imm[4:0]		opcode	
<b>B</b>	imm[12 10:5]					rs2		rs1		funct3		imm[4:1 11]		opcode	
<b>U</b>	imm[31:12]										rd		opcode		
<b>J</b>	imm[20 10:1 11 19:12]										rd		opcode		

**addi** rd, rs1, immediate                       $x[rd] = x[rs1] + \text{sext}(\text{immediate})$

*Add Immediate.* I-type, RV32I and RV64I.

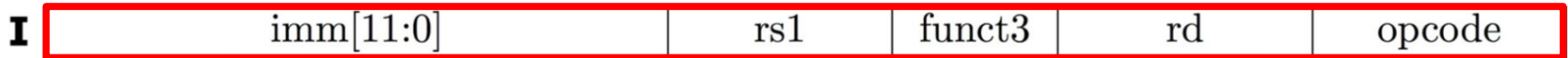
Adiciona o *valor imediato* de sinal estendido ao registrador  $x[rs1]$  e escreve o resultado em  $x[rd]$ . O overflow aritmético é ignorado.

*Formas comprimidas:* **c.li** rd, imm; **c.addi** rd, imm; **c.addi16sp** imm; **c.addi4spn** rd, imm

31	20	19	15	14	12	11	7	6	0			
immediate[11:0]					rs1		000		rd		0010011	

# Limitações dos operandos imediatos

Formato I: `slli, srli, srai, addi, xori, ori, andi, slti, sltiu, jalr`



Campo de imediato (`imm`) é codificado na instrução com apenas 12 *bits*.

Valores válidos: `-2048 : 2047`

Formatos U e J: `imm` tem 20 *bits*



# Limitações dos operandos imediatos

Ao tentar montar um programa que contenha imediatos que não podem ser codificados, o montador reclama.

Ex: Programa prog.s com as seguintes instruções

```
addi a0, a5, 2048  
addi a0, a5, 10000  
addi a0, a5, -3000
```

} prog.s

```
$ as prog.s -o prog.o  
prog.s: Assembler messages:  
prog.s:1: Error: illegal operands `addi a0,a5,2048'  
prog.s:2: Error: illegal operands `addi a0,a5,10000'  
prog.s:3: Error: illegal operands `addi a0,a5,-3000'
```

# Pseudo-instruções

Pseudo-instruções são instruções que existem na linguagem de montagem mas não existem na arquitetura do conjunto de instruções do processador.

O montador mapeia pseudo-instruções em instruções do processador. Ex:

```
nop
```

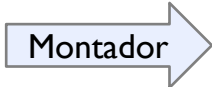
É uma pseudo-instrução mapeada em:

```
addi x0, x0, 0
```

# Pseudo-instruções: Movimentação de dados

Pseudo-instrução: `mv rd, rs`

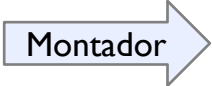
- Exemplo: `mv a0, a1`
- Copia o valor do registrador fonte (`rs`) para o registrador destino (`rd`)

`mv a0, a1`  `addi a0, a1, 0`

# Pseudo-instruções: Movimentação de dados

Pseudo-instrução:  $l\{w|h|hu|b|bu\} rd, rotulo$

- Exemplo: `lw a0, var_x`
- Carrega um valor da memória usando como endereço um rótulo.
- Rótulos representam endereços de 32 *bits*, que não podem ser codificados em um campo de uma instrução de 32 *bits*. Esta pseudo-instrução é expandida pelo montador em 2 instruções. Ex:

`lw a0, var_x`  `auipc a0, var_x[31:12]`  
`lw a0, var_x[11:0](a0)`

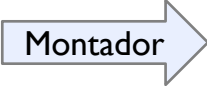
20 bits mais significativos de var\_x

12 bits menos significativos de var\_x

# Pseudo-instruções: Movimentação de dados

Pseudo-instrução: `s{w|h|b} rd, rotulo, rs`

- Exemplo: `sw a0, var_x, t1`
- Grava o valor de `a0` na memória usando como endereço um rótulo (o segundo registrador é usado como temporário).
- Esta pseudo-instrução é expandida pelo montador em 2 pseudo-instruções. Ex:

`sw a0, var_x, t1`  `auipc t1, var_x[31:12]`  
`sw a0, var_x[11:0](t1)`

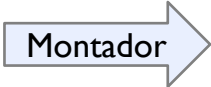
*20 bits mais significativos de var\_x*

*12 bits menos significativos de var\_x*

# Pseudo-instruções: Movimentação de dados

Pseudo-instrução: `la rd, rótulo`

- Exemplo: `la a0, var_x`
- Grava no registrador o endereço do rótulo.
- Rótulos representam endereços de 32 *bits*, que não podem ser codificados em um campo de uma instrução de 32 *bits*. Esta pseudo-instrução é expandida pelo montador em 2 pseudo-instruções. Ex:

`la a0, var_x`  `auipc a0, var_x[31:12]`  
`addi a0, a0, var_x[11:0]`

20 bits mais significativos de var\_x

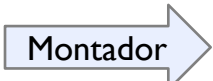
12 bits menos significativos de var\_x

# Pseudo-instruções: Movimentação de dados

Pseudo-instrução: `li rd, imediato`

- Exemplo: `li a0, 1969`
- Carrega um valor de até 32 *bits* no registrador `rd`.
- Valores pequenos (que podem ser representados com poucos *bits*) podem ser carregados com uma única instrução (`addi`) enquanto que valores grandes (que precisam ser codificados com muitos *bits*) podem exigir 2 instruções.

`li a0, 1000`  `addi a0, x0, 1000`

`li a0, 10000`  `lui a0, 0x2` }  $0x2 = 10[31:12] \ll 12$   
`addi a0, a0, 1808`

\*  $10000_{10} == 0000000000000000000010011100010000_2$

$1808 == 011100010000[11:0]$

# Pseudo-instruções: Controle de fluxo

Pseudo-instrução: `ret`

- Retorna de função

```
...
8000    jal ra, foo      # Invoca foo
8004    sub a3, a1, a0
...
9000    foo:            # Função foo
9000    add a0, a0, a1
9004    ret           # Retorna de foo
```

`jal` grava 8004 (PC+4) no registrador `ra` e salta para `foo` (9000)

“`ret`” é uma pseudo-instrução para “`jalr x0, x1, 0`”

“`jalr x0, x1, 0`” é equivalente a “`jalr zero, ra, 0`”



# Pseudo-instruções: Outras

## Outras Pseudo-instruções do RISC-V

<code>nop</code>	<code>addi x0, x0, 0</code>	Operação No
<code>neg rd, rs</code>	<code>sub rd, x0, rs</code>	Complemento de dois
<code>negw rd, rs</code>	<code>subw rd, x0, rs</code>	Palavra em complemento de dois
<code>snez rd, rs</code>	<code>sltu rd, x0, rs</code>	"Seta" se $\neq$ zero
<code>sltz rd, rs</code>	<code>slt rd, rs, x0</code>	"Seta" se $<$ zero
<code>sgtz rd, rs</code>	<code>slt rd, x0, rs</code>	"Seta" se $>$ zero
<code>beqz rs, offset</code>	<code>beq rs, x0, offset</code>	Desvia se $=$ zero
<code>bnez rs, offset</code>	<code>bne rs, x0, offset</code>	Desvia se $\neq$ zero
<code>blez rs, offset</code>	<code>bge x0, rs, offset</code>	Desvia se $\leq$ zero
<code>bgez rs, offset</code>	<code>bge rs, x0, offset</code>	Desvia se $\geq$ zero
<code>bltz rs, offset</code>	<code>blt rs, x0, offset</code>	Desvia se $<$ zero
<code>bgtz rs, offset</code>	<code>blt x0, rs, offset</code>	Desvia se $>$ zero
<code>j offset</code>	<code>jal x0, offset</code>	Pula
<code>jr rs</code>	<code>jalr x0, rs, 0</code>	Registrador de pulo
<code>ret</code>	<code>jalr x0, x1, 0</code>	Retorna da sub-rotina

# Pseudo-instruções: Outras

## Outras Pseudo-instruções do RISC-V

Pseudo-Instrução	Instrução (ões) Base	Significado
li rd, immediate	<i>Miríades de sequências</i>	Load valor imediato
mv rd, rs	addi rd, rs, 0	Copia registrador
not rd, rs	xori rd, rs, -1	Complemento de um
sxt.w rd, rs	addiw rd, rs, 0	Estende o sinal da palavra
seqz rd, rs	sltiu rd, rs, 1	"Seta" se = zero
bgt rs, rt, offset	blt rt, rs, offset	Desvia se >
ble rs, rt, offset	bge rt, rs, offset	Desvia se ≤
bgtu rs, rt, offset	bltu rt, rs, offset	Desvia se >, sem sinal
bleu rs, rt, offset	bgeu rt, rs, offset	Desvia se ≤, sem sinal
jal offset	jal x1, offset	Pula e linka
jalr rs	jalr x1, rs, 0	Jump e linka o registrador

# Detecção de *overflow*

Detecção de *overflow* em somas de valores na representação **sem sinal**.

- Saltar para rótulo se houver *overflow*:

```
add  a0, a1, a2      # somamos os valores
bltu a0, a1, trata_ov # salta para trata_ov se
                        # houve overflow

...
trata_ov:            # Tratamento de overflow
...
```

- Indicar *overflow* em registrador

```
add  a0, a1, a2 # somamos os valores
sltu t1, a0, a1 # t1 <= 1 se (a1+a2) < a1 (Overflow)
                # do contrário, t1 <= 0
```

# Detecção de *overflow*

Detecção de *overflow* em somas de valores na representação **com sinal**.

- Saltar para rótulo se houver *overflow*:

```
add  a0, a1, a2      # somamos os valores
slti t1, a2, 0       # t1 = (a2 < 0)
slt  t2, a0, a1      # t2 = (a1+a2 < a1)
bne  t1, t2, trata_ov # overflow se (a2<0) && (a1+a2>=a1)
                          # ou se (a2>=0) && (a1+a2<a1)
...
trata_ov:           # Tratamento de overflow
...
```

# Soma de valores multi-palavras

A soma de valores de 64 *bits* (long long)

- Utilizaremos a notação  $a1 : a0$  para indicar que o par de registradores  $a1$  e  $a0$  armazena um número de 64 *bits* sendo que os 32 *bits* menos (mais) significativos estão em  $a0$  ( $a1$ ).

Desejamos somar dois números de 64 *bits* armazenados em  $a1 : a0$  e  $a3 : a2$  e armazenar o resultado em  $a5 : a4$ .

- Podemos somar desta forma?

```
add a4, a0, a2 # somamos a parte menos significativa
add a5, a1, a3 # somamos a parte mais significativa
```

# Soma de valores multi-palavras

A soma de valores de 64 *bits* (long long)

- Utilizaremos a notação de registradores  $a1$  e  $a2$  sendo que os 32 *bits* estão em  $a0$  ( $a1$ ).

Não! Desta forma o Código não leva em consideração a propagação de “vai um” entre o *bit 31* e o *bit 32*.

Desejamos somar dois números armazenados em  $a1 : a0$  e  $a3 : a2$  e armazenar o resultado em  $a5 : a4$ .

- Podemos somar desta forma

```
add a4, a0, a2 # somamos a parte menos significativa
add a5, a1, a3 # somamos a parte mais significativa
```

# Soma de valores multi-palavras

Ao somarmos a parte menos significativa, verificamos se houve *overflow* considerando a representação sem sinal. Caso positivo, adicionamos 1 à soma da parte mais significativa.

Desejamos somar dois valores de 32 bits armazenados em a1:a0 e a3:a2. Queremos armazenar o resultado em a5:a4.

- Podemos somar desta forma?

```
add  a4, a0, a2 # somamos a parte menos significativa
sltu t1, a4, a2 # t1 <= 1 se (a0+a2) < a2 (Overflow)
                # do contrário, t1 <= 0
add  a5, a1, a3 # somamos a parte mais significativa
add  a5, t1, a5 # somamos o "vai um"
```

# Soma de valores multi-palavras

Exercício 1: Mostre o código para somar dois valores de 64 *bits* armazenados na memória, identificados pelos rótulos *x* e *y*, e armazenar o resultado no rótulo *z*

- Dica 1: Você pode carregar os valores da memória com a pseudo-instrução “`lw rd, rotulo`”
- Dica 2: Você pode carregar o valor armazenado na posição de memória `rotulo+4` com a pseudo-instrução “`lw rd, rotulo+4`”

○ Exemplo:

```
x:  
...  
lw a0, x  
lw a1, x+4
```



# Soma de valores multi-palavras

Exercício 1: Mostre o código para somar dois valores de 64 *bits* armazenados na memória, identificados pelos rótulos *x* e *y*, e armazenar o resultado no rótulo *z*

```
lw  a0, x          # Carrega a parte menos sig. de x
lw  a1, y          # Carrega a parte menos sig. de y

add a1, a0, a1     # Soma partes menos significativas
sltu t1, a1, a0    # computa o "vai um"
sw  a1, z, a0      # Armazena resultado parcial em z

lw  a0, x+4        # Carrega a parte mais sig. de x (x+4)
lw  a1, y+4        # Carrega a parte mais sig. de y (x+4)

add a1, a0, a1     # Soma partes mais significativas
add a1, a1, t1     # Adiciona o "vai um"
sw  a1, z+4, a0    # Armazena resultado parcial em z+4
```

# Soma de valores multi-palavras

Exercício 2: Mostre o código para somar dois valores de **128** *bits* armazenados na memória, identificados pelos rótulos *x* e *y*, e armazenar o resultado no rótulo *z*

- Dica 1: Agora o número é composto por quatro palavras de 32 *bits*.
- Dica 2: Lembre-se de levar em consideração o "vai um" das palavras menos significativas para as mais significativas.