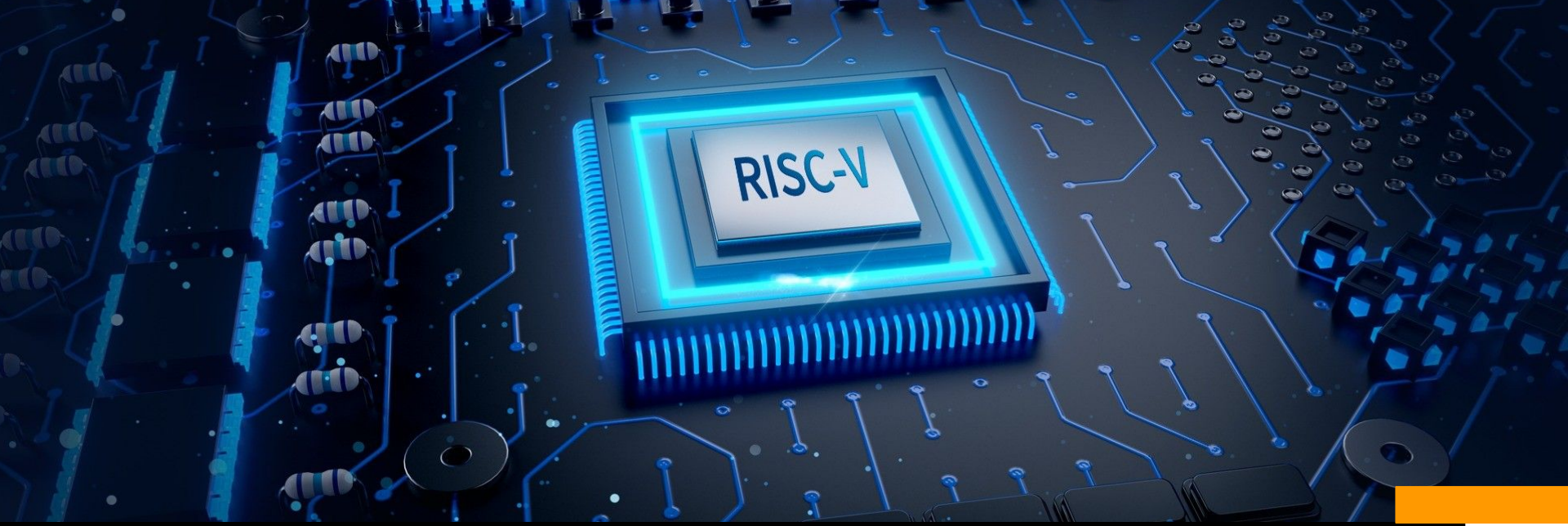


MC404AE - Organização Básica de Computadores e Ling. Montagem

Introdução à arquitetura RV32

Prof. Allan M. de Souza

UNICAMP

RISC-V

- ISA (Instruction Set Architecture) RISC moderna
  - Introduzida em 2011
- ISA aberta! (uso livre e livre de royalties)

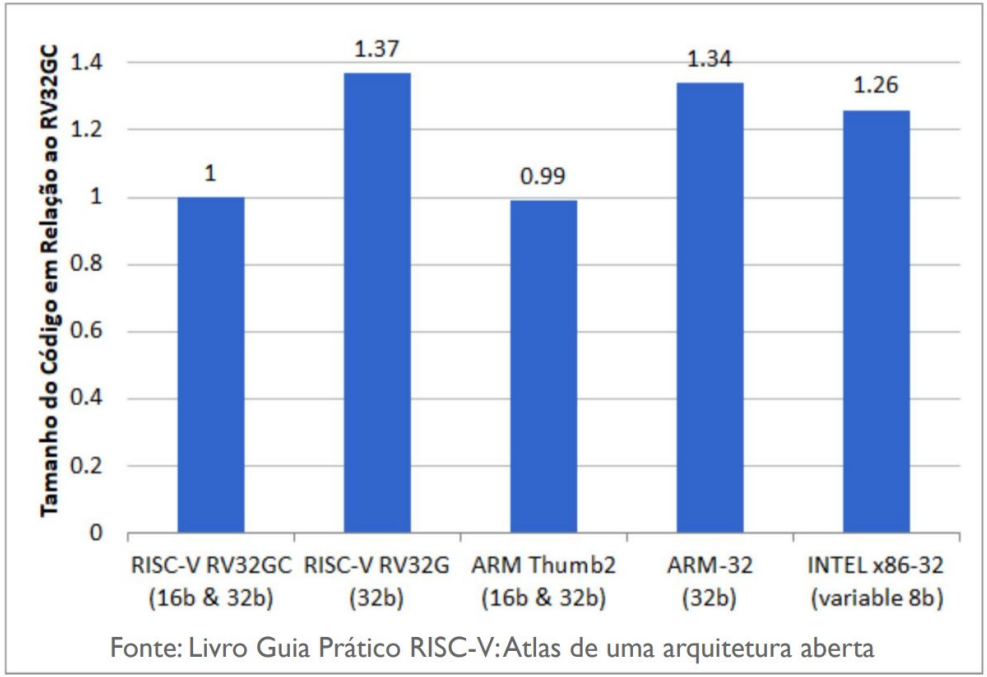
- Funcionalidades e características desenvolvidas com base nos acertos e erros de ISAs que já estão no mercado há mais de 30 anos! (x86 e ARM)
  - Mais simples do que ARM e x86
  - Veja figura 2.7 do livro “Guia Prático RISC-V: Atlas de uma arquitetura aberta”

RISC-V

Mais simples do que ARM e x86

- Mantida atualmente pela Fundação RISC-V
- Fundação aberta e sem fins lucrativos

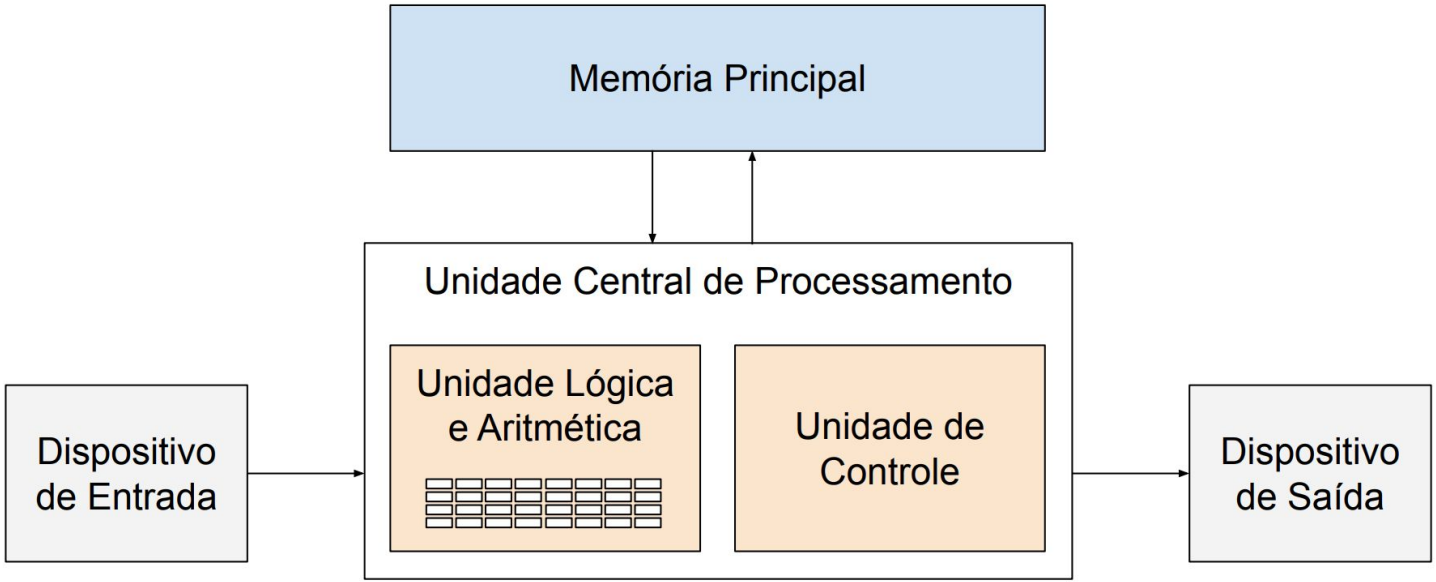
Tamanho relativo de programas do *benchmark* SPEC CPU2006 compilados com o GCC.



Fonte: Livro Guia Prático RISC-V: Atlas de uma arquitetura aberta

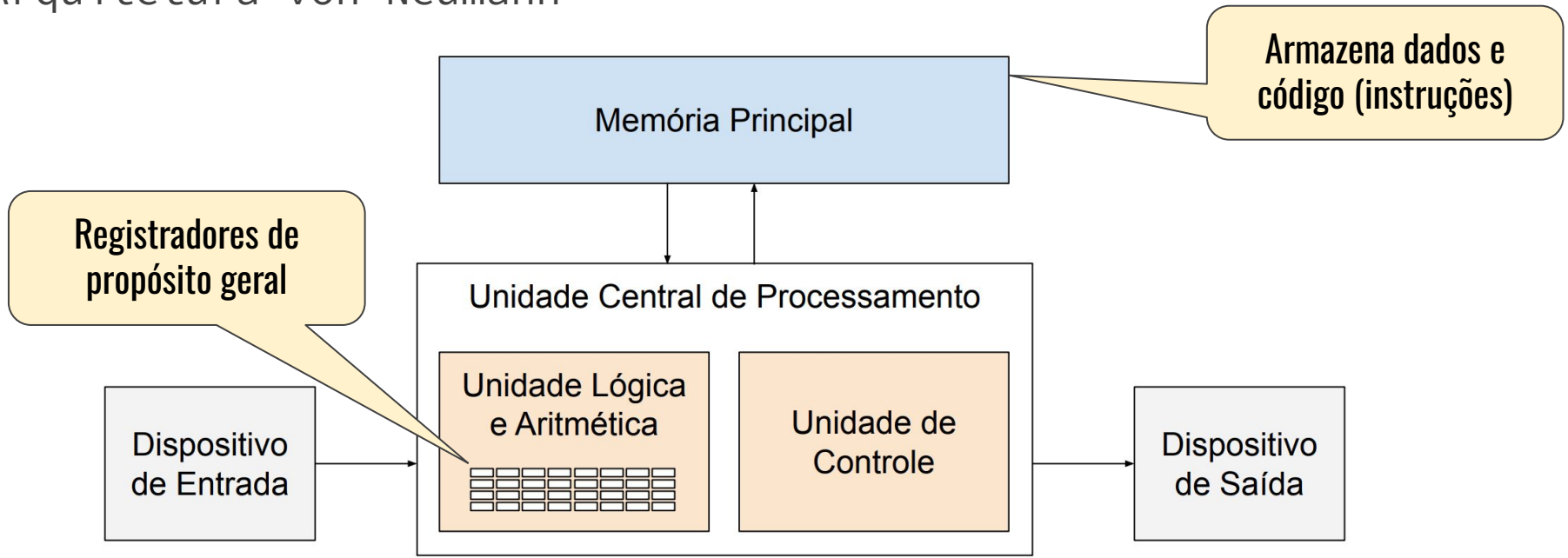
Arquitetura do RISC-V

Arquitetura von Neumann



Arquitetura do RISC-V

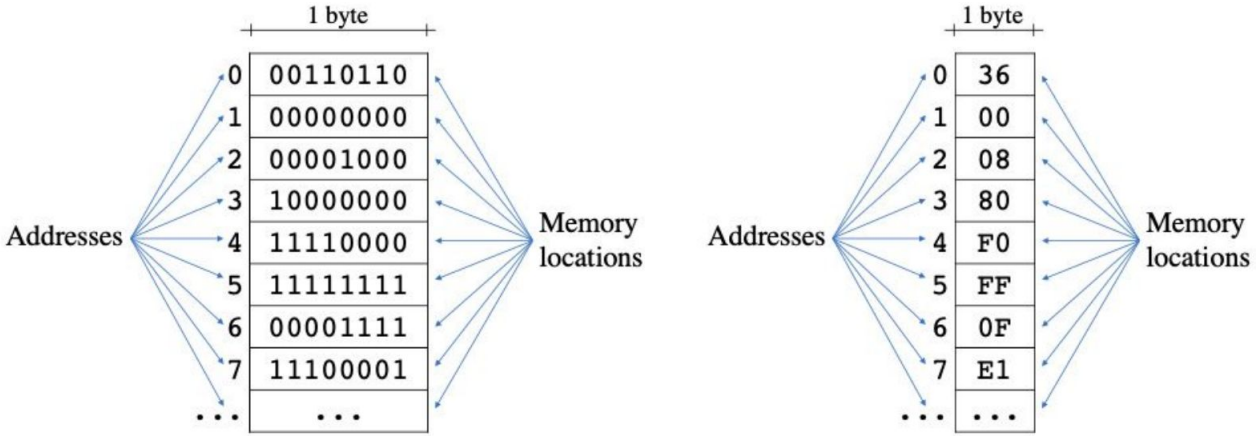
Arquitetura von Neumann



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.



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)

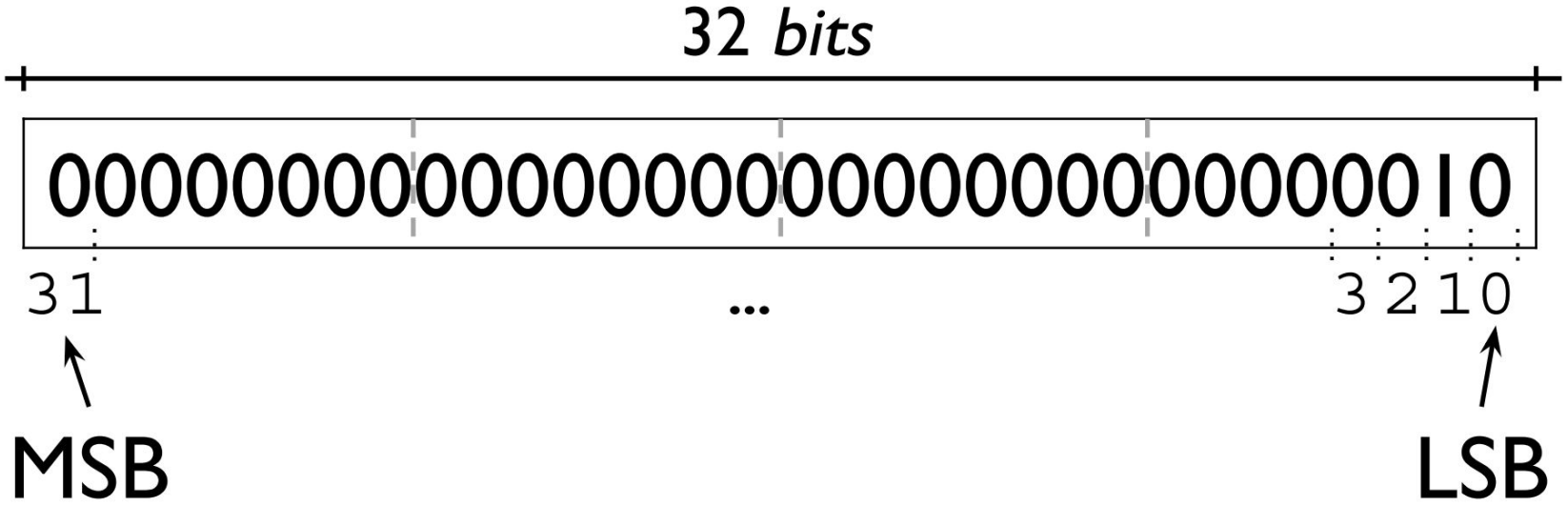
Arquitetura RV32

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

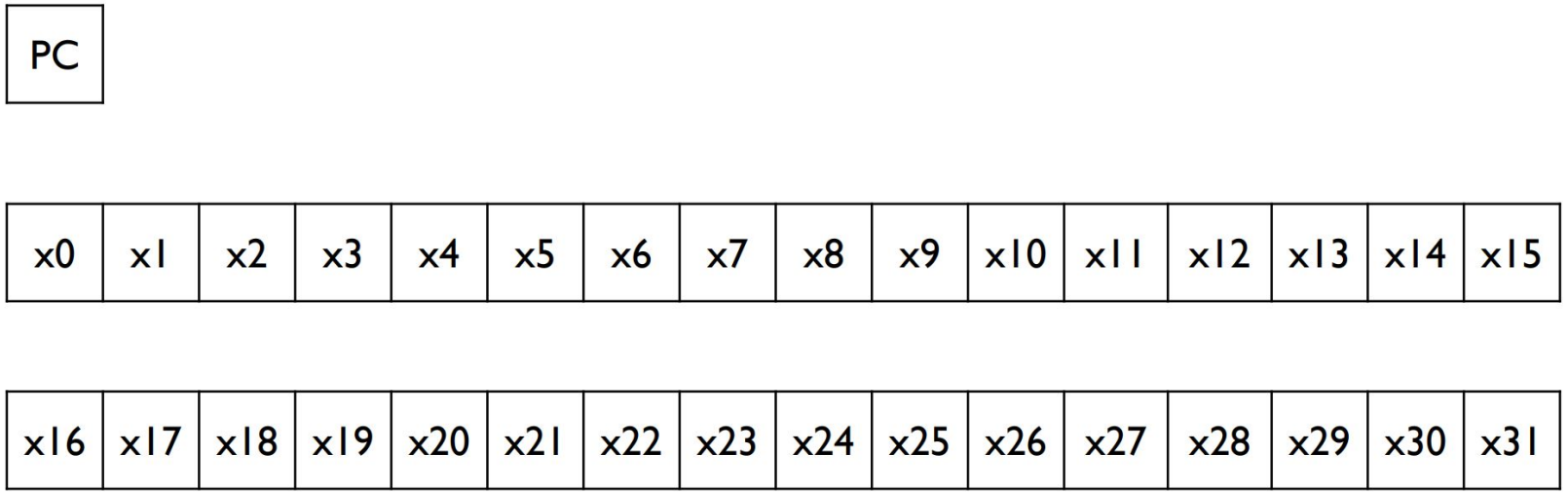
Arquitetura RV32

Registradores



Arquitetura RV32

Registradores



Arquitetura RV32

Registradores

PC

Apelidos

zero	ra	sp	gp	tp	t0	t1	t2	s0	s1	a0	a1	a2	a3	a4	a5
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
a6	a7	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	t3	t4	t5	t6

Apelidos

Arquitetura RV32

Registradores

Apelido	Significado
`pc`	*Program Counter* (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 *thread*

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

Exemplo

Registradores

a0	00	00	00	00
a1	00	00	00	04
a5				
a6				

PC →

```
lw a5, 0(a0)    # a5 <= Mem[a0]
add a6, a5, a5  # a6 <= a5+a5
sw a6, 0(a1)    # Mem[a1] <= a6
```

Memória

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

Arquitetura RV32

Exemplo

Registradores

a0	00	00	00	00
a1	00	00	00	04
a5	00	00	00	06
a6				

Memória

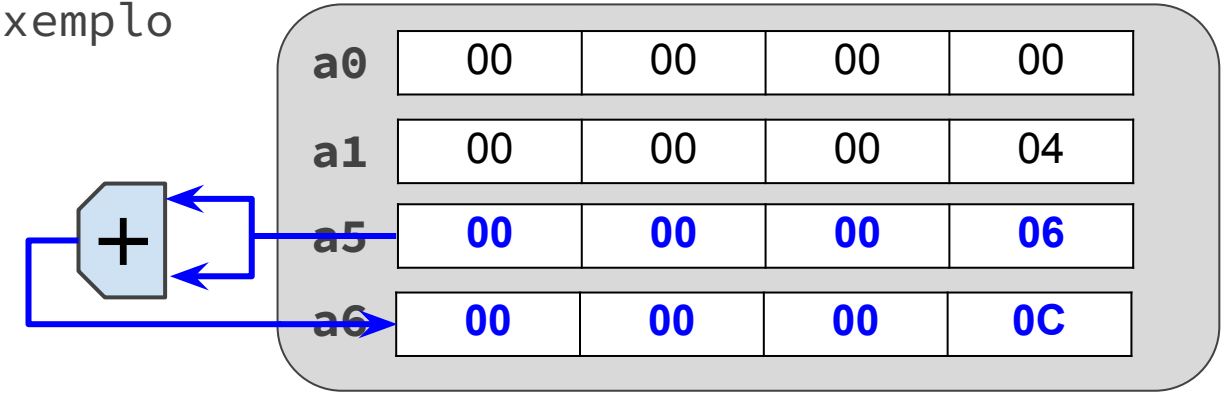
Valor	
06	0
00	1
00	2
00	3
00	4
00	5
00	6
00	7
...	...

PC →

```
lw a5, 0(a0)     # a5 <= Mem[a0]
add a6, a5, a5   # a6 <= a5+a5
sw a6, 0(a1)     # Mem[a1] <= a6
```

Arquitetura RV32

Exemplo



PC →

```
lw a5, 0(a0)     # a5 <= Mem[a0]
add a6, a5, a5   # a6 <= a5+a5
sw a6, 0(a1)     # Mem[a1] <= a6
```

Memória

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

Arquitetura RV32

Exemplo

Registradores

a0	00	00	00	00
a1	00	00	00	04
a5	00	00	00	06
a6	**00**	**00**	**00**	**0C**

```
lw a5, 0(a0)    # a5 <= Mem[a0]
add a6, a5, a5  # a6 <= a5+a5
sw a6, 0(a1)    # Mem[a1] <= a6
```

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`
- `srli a1, a3, 1 # a1 <= a3 >> 1`
- `srai a2, a2, 1 # a2 <= a2 >>* 1`
  `# *aritmético`

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 * 2^2`
- `slli a3, a3, 4 # a3 <= a3 * 2^4`

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 <= a2 / 2^2`

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

- `srai a0, a2, 2 # a0 <= a2 / 2^2`

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**

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`

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

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

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

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

Instruções: Movimentação de dados

Instruções para copiar valores da memória p/ registradores.

Formato: <MNE> rd, imm(rs1)

Endereço de memória

- `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!**

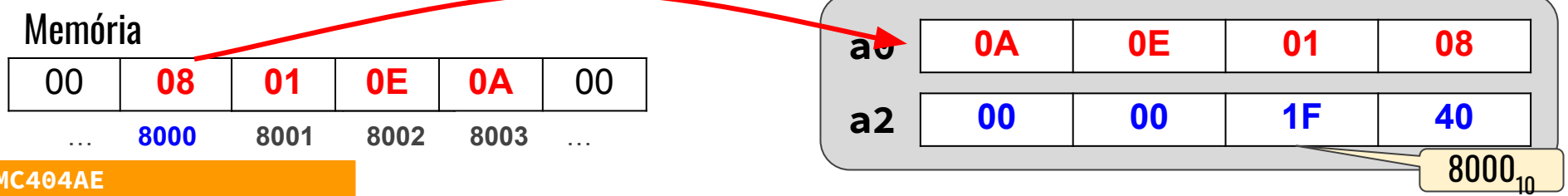
Instruções: Movimentação de dados

Load word

Endereço de memória

- `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**.

`lw a0, 0(a2)`



Instruções: Movimentação de dados

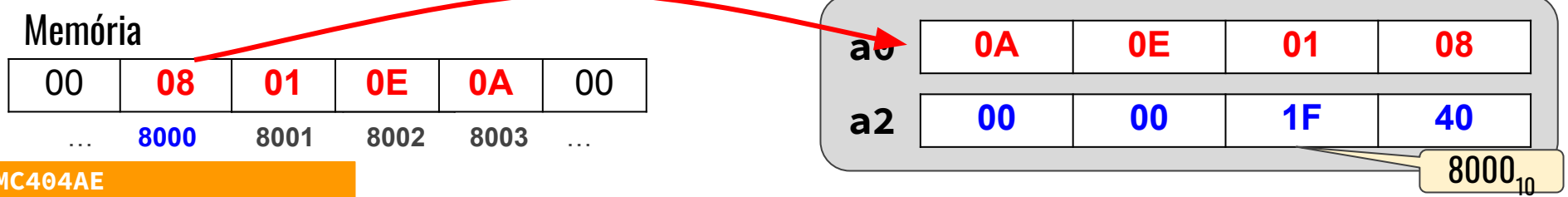
Load word

Endereço de memória

- `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**.

lw deve ser usado quando carregarmos dados tipo **int** ou **unsigned int** da memória

`lw a0, 0(a2)`



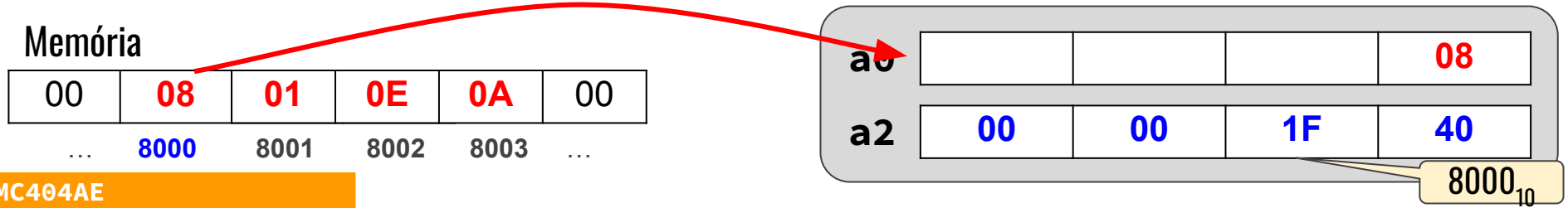
Instruções: Movimentação de dados

Load byte unsigned

Endereço de memória

- `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.

`lbu a0, 0(a2)`



Instruções: Movimentação de dados

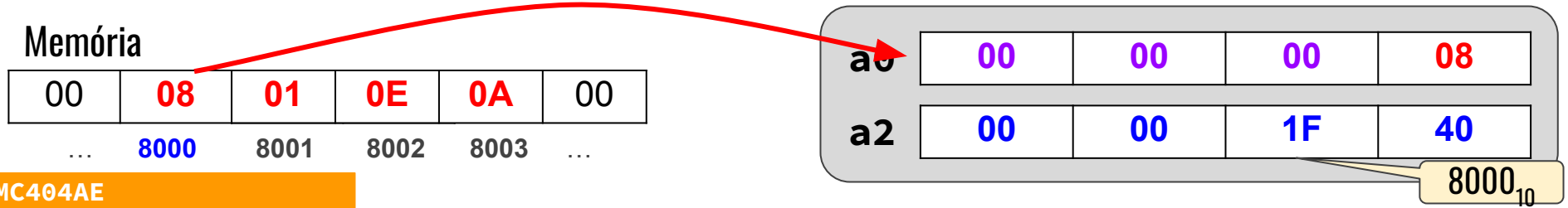
Load byte unsigned

Endereço de memória

- `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.

lbu deve ser usado quando carregarmos dados tipo **unsigned char** da memória

`lbu a0, 0(a2)`



Instruções: Movimentação de dados

Load byte

Endereço de memória

- `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).

`lb a0, 0(a2)`

08_{16} é um número de 1 byte positivo (8)

Memória

00	08	01	0E	0A	00
…	8000	8001	8002	8003	…

a0	00	00	00	08
a2	00	00	1F	40

8000_{10}

Instruções: Movimentação de dados

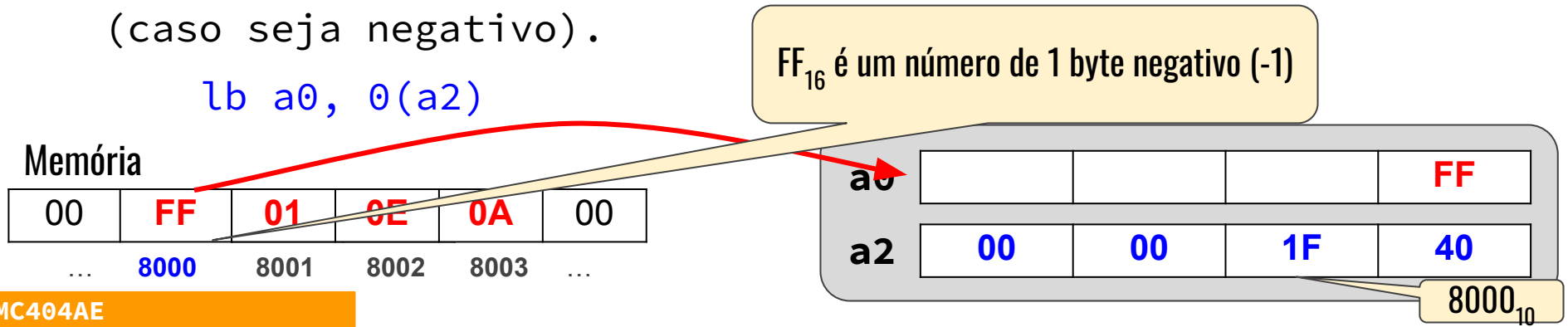
Load byte

Endereço de memória

- `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).

`lb a0, 0(a2)`

FF_{16} é um número de 1 byte negativo (-1)



Instruções: Movimentação de dados

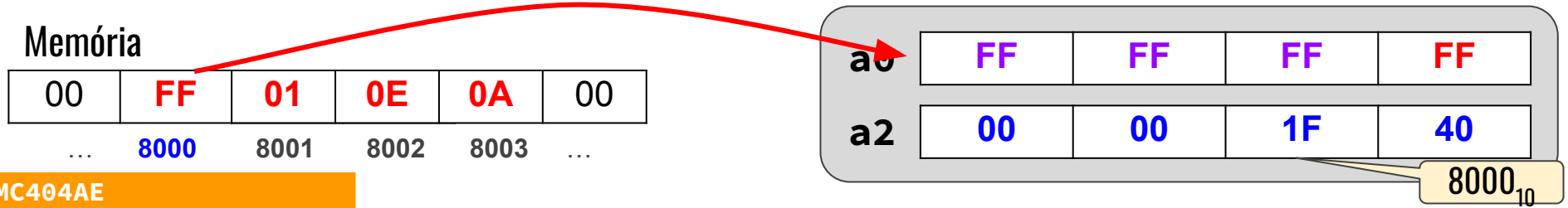
lb deve ser usado quando carregarmos dados tipo **char** da memória

Load byte

Endereço de memória

- `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).

`lb a0, 0(a2)`



Instruções: Movimentação de dados

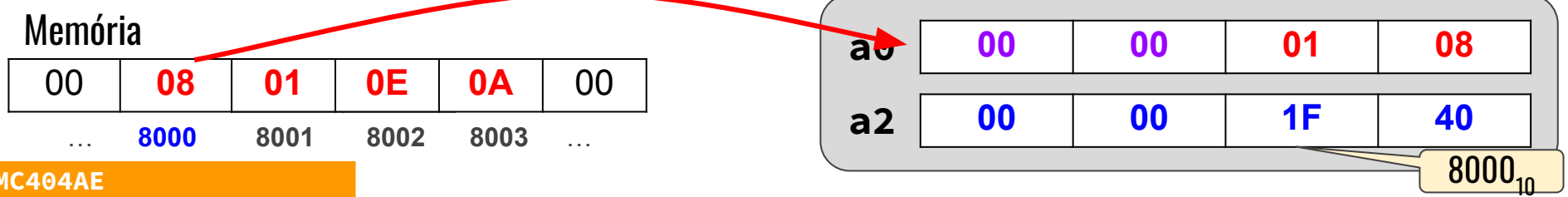
lhu deve ser usado quando carregarmos dados tipo **unsigned short** da memória

Load halfword unsigned

Endereço de memória

- `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.

`lhu a0, 0(a2)`



Instruções: Movimentação de dados

lh deve ser usado quando carregarmos dados tipo **short** da memória

Load halfword

Endereço de memória

- `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).

```
lhu a0, 0(a2)
```

Memória

00	FE	FF	0E	0A	00
…	8000	8001	8002	8003	…

a0	FF	FF	FF	FE
a2	00	00	1F	40

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** (**s**) carregam dados da memória para registradores.
O sufixo (**w**, **h**, **b**) indica o **tipo de dado!**

Instruções: Movimentação de dados

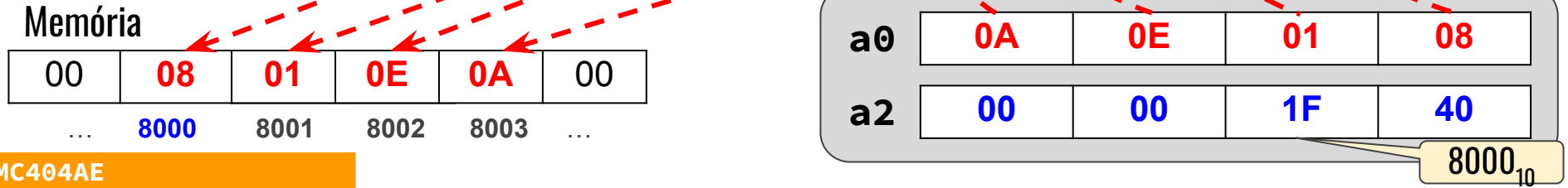
sw deve ser usado quando gravamos dados tipo **int** ou **unsigned int** na memória

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.

`sw a0, 0(a2)`



Instruções: Movimentação de dados

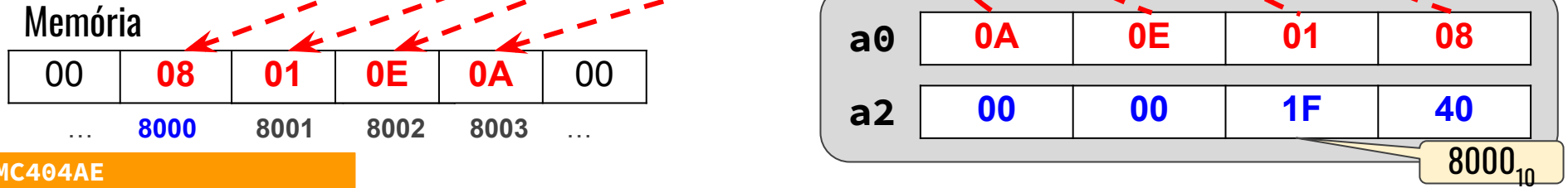
sw deve ser usado quando gravamos dados tipo **int** ou **unsigned int** na memória

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.

`sw a0, 0(a2)`



Instruções: Movimentação de dados

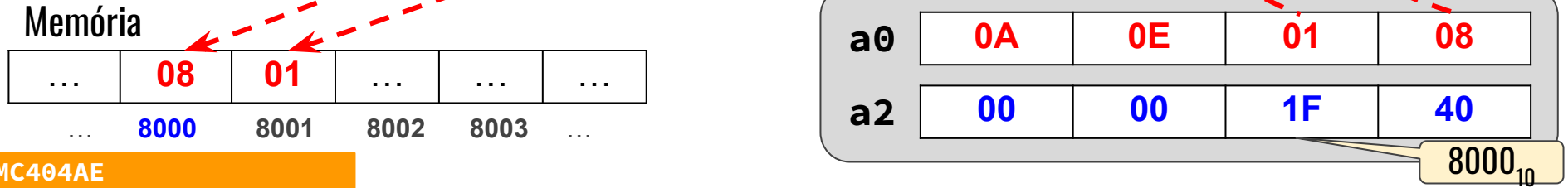
sw deve ser usado quando gravamos dados tipo **short** ou **unsigned short** na memória

Store half word

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

Grava um número de 16 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+1.

`sh a0, 0(a2)`



Instruções: Movimentação de dados

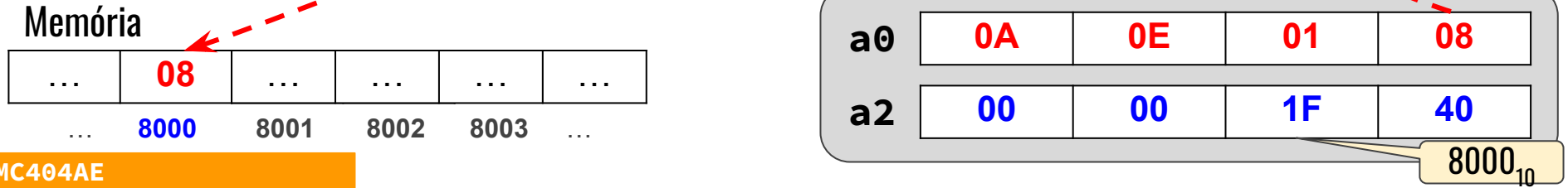
sw deve ser usado quando gravamos dados tipo **char** ou **unsigned char** na memória

Store byte

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

Grava um número de 16 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.

`sb a0, 0(a2)`



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.

```
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.

```
beq a0, a2, next_item
add a0, a0, a1

next_item:
   addi a0, s1, -1
```

no caso apresentado, a instrução **branch equal (beq)** só "salta" (**desvio de fluxo**) para no 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  (sem sinal)`
- `bgeu a0, a2, L # Salta se a0 >=* a2 (sem sinal)`

Instruções: Controle de fluxo

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

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

no caso apresentado, a instrução **jump and link (jal)** salta para no rótulo **foo** e a próxima instrução a ser executada é a **sub**

Instruções: Controle de fluxo

Instruções de controle de fluxo incondicionais

Formato: <MNE> rd, rot

- jal a0, L # “Faz o link” e salta para L

- Grava PC+4 em a0 e depois salta para 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

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

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

Invocando rotinas com **jal**

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

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

Salta para o endereço **8004** (armazenado em ra) Endereço **9008** (**PC+4**) é descartado (**escrita no reg. x0**)

Instruções: Controle de fluxo

Instruções de salto direto vs salto indireto

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

- 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.

- 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 "Hello World!" # String

.text
_start:
  li a0, 1     # a0: File descriptor = 1 (stdout)
  la a1, msg   # a1: endereço do buffer msg
  li a2, 12    # 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: rd, rs1, rs2

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

Formato: i rd, rs1, imm

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

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
R	funct7		rs2	rs1	funct3	rd	opcode
I	imm[11:0]			rs1	funct3	rd	opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
J	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
R	funct7		rs2	rs1	funct3	rd	opcode
I	imm[11:0]			rs1	funct3	rd	opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
J	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
R	funct7		rs2	rs1	funct3	rd	opcode
I	imm[11:0]			rs1	funct3	rd	opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
J	imm[20|10:1|11|19:12]					rd	opcode

addi rd, rs1, immediate `x[rd] = x[rs1] + sext(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.

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)

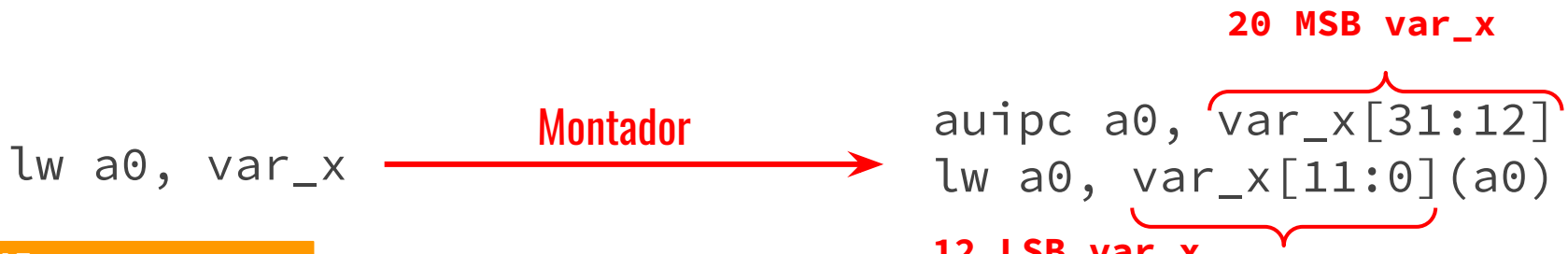
mv a0, a1 —Montador→ addi a0, a1, 0

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.



Pseudo-instruções: Movimentação de dados

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

Montador →

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

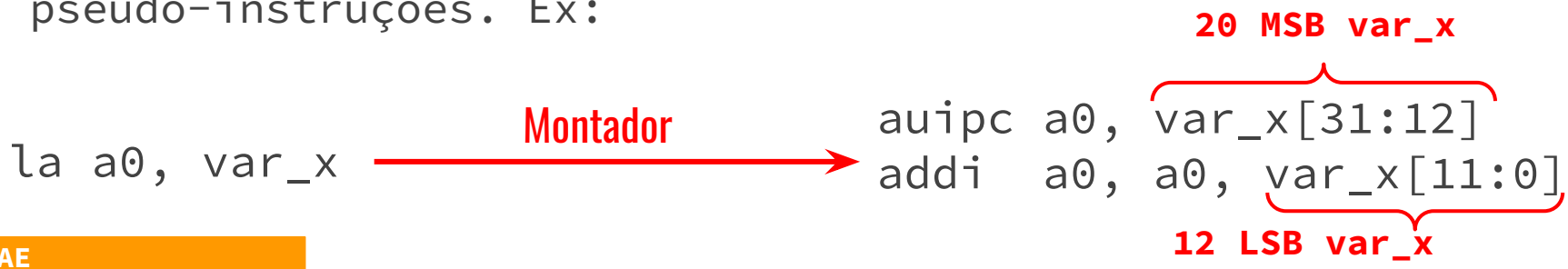
20 MSB var_x

12 LSB var_x

Pseudo-instruções: Movimentação de dados

Pseudo-instrução: la rd, rotulo

- 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:



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 grandes (que precisam ser codificados com muitos bits) podem exigir 2 instruções

```
li a0, 1000    --Montador-->  addi a0, x0, 1000

li a0, 10000   --Montador-->  lui  a0, 0x2
                              addi a0, a0, 1808
```

0x2 = 10[31:12] << 12

10000 = 00000000000000010 011100010000

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:
9000 add a0, a0, a1
9004 ret                  # Retorna de foo
```

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

"**ret**" é uma pseudo-instrução para "**jalr x0, x1, 0**"