Operações de Controle de Fluxo e Acesso a Memória

Universidade Federal de Uberlândia
Faculdade de Computação
Prof. Dr. rer. nat. Daniel D. Abdala

Na Aula Anterior ...

- Instruções aritméticas em $\mathbb{Z}$;
- Formato e Codificação de Instruções;
- Overflow e underflow;
- Instruções aritméticas em $\mathbb{R}$;
- Instruções lógicas;

Nesta Aula

- Instruções de controle de fluxo;
- Codificando fluxo em Assembly:
  - If-then
  - If-then-else
  - Switch-case
- Codificando repetições em Assembly:
  - while() / do while()
  - for()
- Instruções de acesso a memória;
- Palavras Alinhadas e deselinhadas;
- Instruções de transferência de dados entre o Processador e o Coprocessador C1

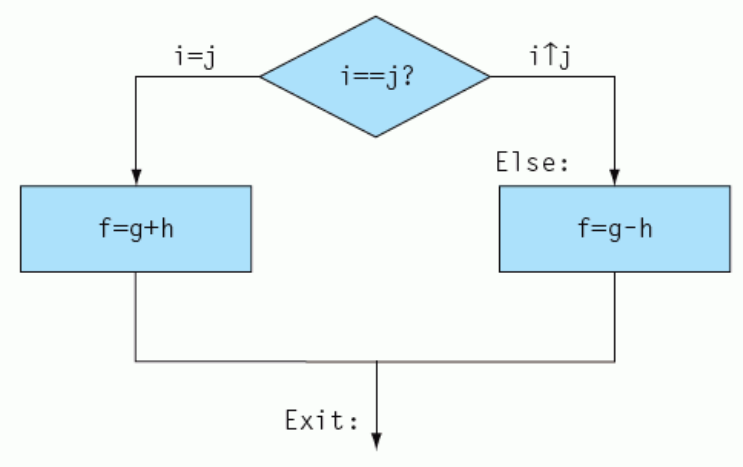
Instruções de Controle de Fluxo

OP	Descrição	Exemplo
beq	(=) Salta se igual	beq $rs, $rt, offset
bgez	(≥0)Salta se maior ou igual a zero	bgez $rs, $rt, offset
bgezal	(≥0)Salta se maior ou igual a zero (usado em subrotinas) ($ra ←PC+4)	bgezal $rs, $rt, offset
bgtz	(>0)Salta se maior que zero	bgtz $rs, $rt, offset
blez	(≤0)Salta se menor ou igual a zero	blez $rs, $rt, offset
bltz	(<0) Salta se menor que zero	bltz $rs, $rt, offset
bltzal	(<0) Salta se menor que zero ($ra ←PC+4)	bltzal $rs, $rt, offset
bne	(≠) salta se diferente	bne $rs, $rt, offset
j	Salto incondicional	j offset
jal	Salto incondicional e liga ($ra ←PC+4)	jal offset
jalr	Salto incondicional (registrador) Salva em $ra o endereço da instrução de retorno	jalr $rs
jr	Salta para endereço em registrador	jr $rs
movn	rd ← rs se rt == 0	movn $t1, $t2, $t3

if –then - else

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

```
     bne $s3,$s4,Else   # go to Else if i ≠ j
     add $s0,$s1,$s2    # f = g + h (skipped if i ≠ j)
       j Exit           # go to Exit
Else:sub $s0,$s1,$s2    # f = g - h (skipped if i = j)
Exit:
```



Switch/case

```
# include < iostream >
# include <string >

using namespace std;

int main ()
{
int resistance ; // in Ohms
string partNum ; // Part Number

cout << " Enter resistance : " << endl ;
cin >> resistance ;
switch ( resistance )
{
case 3 : partNum = " OD30GJE "; break ;
case 10 : partNum = " OD100JE "; break ;
case 22 : partNum = " OD220JE "; break ;
default : partNum = "No match "; break ;
}
cout << " Part number : " << partNum << endl ;
return 0;
}
```

- Jump address table

Switch/case (2)

```
.data
int_value: .space 20
.align 2
input:  .asciiz "Enter resistance.\n"       # declaration for string variable,
string1:    .asciiz "OD30GJE\n" # declaration for string variable,
string2:    .asciiz "OD100JE\n"
string3:    .asciiz "OD220JE\n"
string11:   .asciiz "No Match\n"
string12:   .asciiz "Enter resistance\n"
    .text
main:
li $v0, 4
la $a0, input                   # print for input
syscall

la      $t0, int_value
li  $v0, 5          # load appropriate system call code into register $v0;

syscall             # call operating system to perform operation
sw  $v0, int_value      # value read from keyboard returned in register $v0;
                        # store this in desired location
lw  $s1, 0($t0)
condition1:
slt $t1, $s1, $zero # if $s1 < 0 $t1 = 1 else $t1 = 0
beq $t1, $zero, condition2 # if $t1 = 0; InvalidEntry
bne $t1, $zero, invalid_entry

condition2:
sgt $t1, $s1, -1 # if $s1 > -1 then $t1 = 1 else $t1 = 0
beq  $t1, $zero, invalid_entry # if $t1 = 0; InvalidEntry
sgt $t1, $s1, 9 # if s1 > 9 t1 = 1 else $t1 = 0
bne $t1, $zero, condition3 # if $t1 does not equal = 0; condition3
```

Switch/case (3)

```
condition2:
sgt $t1, $s1, -1 # if $s1 > -1 then $t1 = 1 else $t1 = 0
beq  $t1, $zero, invalid_entry # if $t1 = 0; InvalidEntry
sgt $t1, $s1, 9 # if s1 > 9 t1 = 1 else $t1 = 0
bne $t1, $zero, condition3 # if $t1 does not equal = 0; condition3

li  $v0, 4
la  $a0, string1
syscall
j exit

condition3:
sgt $t1, $s1, 9 # if $s1 > 9 then $t1 = 1 else $t1 = 0
beq  $t1, $zero, invalid_entry # if $t1 = 0; InvalidEntry
sgt $t1, $s1, 21 # if s1 > 21 t1 = 1 else $t1 = 0
bne $t1, $zero, condition3 # if $t1 does not equal = 0; condition3

li  $v0, 4
la  $a0, string2
syscall
j exit

invalid_entry:
li  $v0, 4
la  $a0, string11
syscall
j exit
exit:
li $v0, 10 # v0<- (exit)
syscall
```

while

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

```
Loop: sll  $t1,$s3,2     # Temp reg $t1 = 4 * i
      add $t1,$t1,$s6     # $t1 = address of save[i]
      lw  $t0,0($t1)      # Temp reg $t0 = save[i]
      bne  $t0,$s5, Exit  # go to Exit if save[i] ≠ k
      add  $s3,$s3,1      # i = i + 1
      j    Loop           # go to Loop
 Exit:
```

for

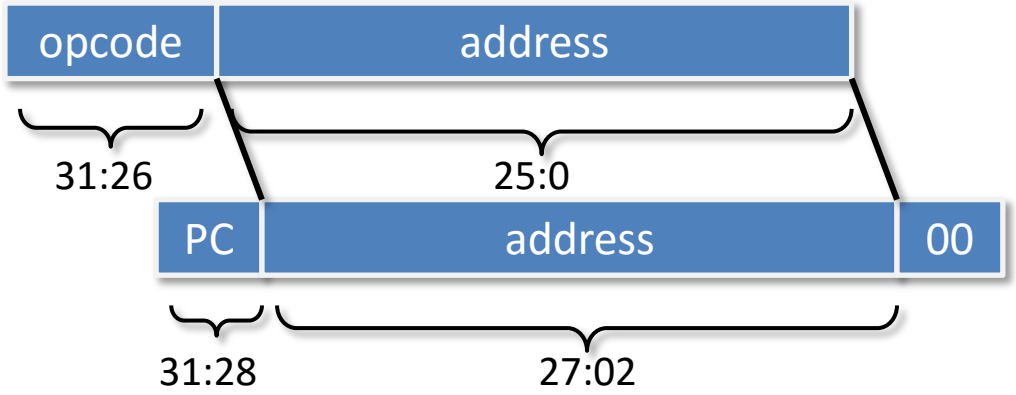
```
        ##  for(c=0; c<10; c++)
        ##      x += arr[c];

        la      $t0, arr            # carrega o end. de arr em t0
        addi    $t1, $zero, 0       # inicializa o contador c
        la      $t2, arr_sz         # carrega o end. do tam do array em t2
        lw      $t2,0($t2)          # carrega o tam do array em t2
FOR1:   beq     $t1,$t2, EXIT1      # se o contador chegou no final pula para fora do laço
        ####corpo do for
        sll     $t3,$t1,2           # t3 = c*4
        add     $t3,$t3,$t0         # t3 = c*4 + &arr
        lw      $t4,0($t3)          # t4 = arr[c]
        add     $s0,$s0,$t4         # x += arr[c]
        addi    $t1,$t1,1           # incrementa o contador c
        j       FOR1:               # retorna para executar a prox. iteração do laço
        ##################
EXIT1:
```

Jumps

- Instruções de salto permitem alterar o fluxo de execução de programas;
- A ISA do MIPS32 prevê vários tipos de instruções de salto:
  - Saltos relativos ao PC;
  - Saltos Absolutos.

Jump (2)

```
j LABEL
```



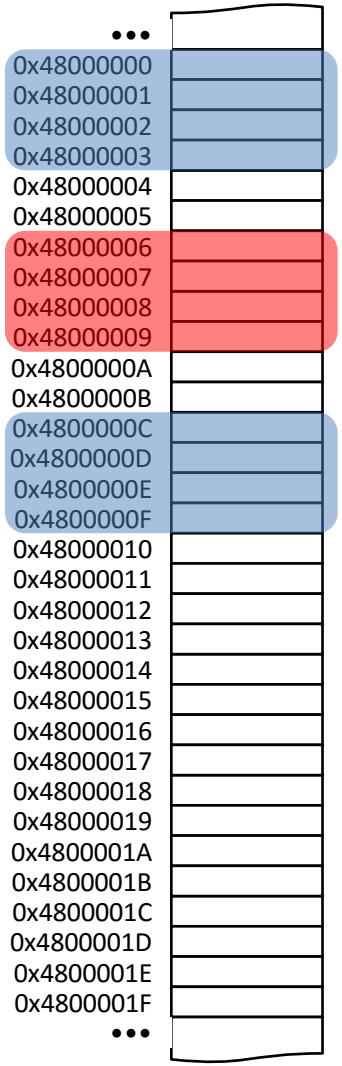
Outras Instruções de Salto

- jr → jump register
  - Usado para saltos absolutos;
  - Usado em sub-rotinas (retorno da sub-rotina);
  - Endereços de até 32 bits (capacidade do registrador);
  - Ex: `jr $s0`
- jal →jump and link
  - Usado em sub-rotinas;
  - Seta $ra para o endereço de PC+4 (prox. instrução);
  - Salta para o endereço especificado;
  - Ex: `jal LABEL`
- jalr →jump and link register
  - Usado em sub-rotinas;
  - Seta $ra para PC+4 e salta para a pos. de mem. em $s0;
  - Ex: `jalr $s0`

Instruções de Acesso a Memória

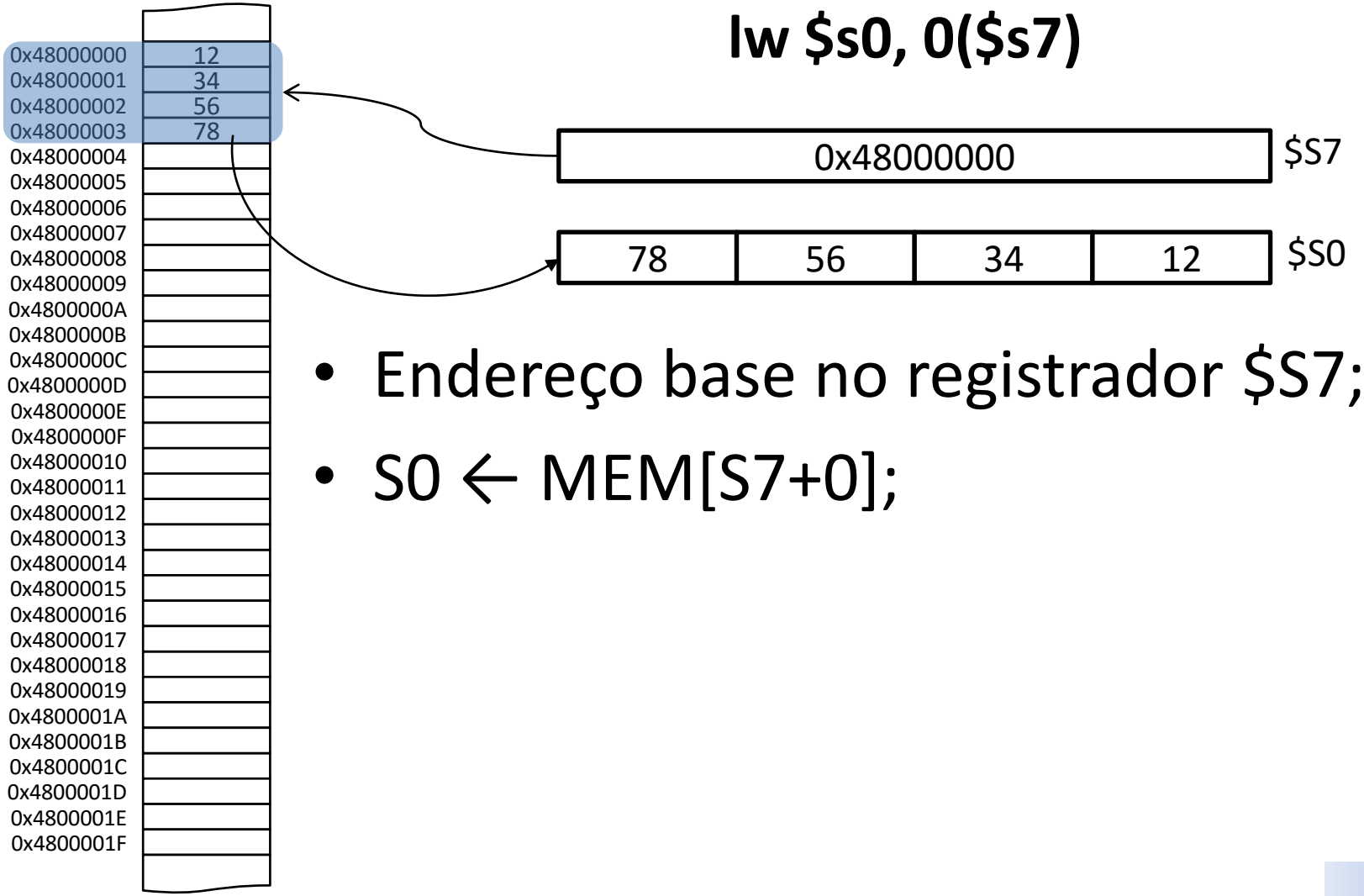
OP	Descrição	Exemplo
lb	Carrega byte da memória (rt ←MEM[rs+offset])	lb $rt, offset($rs)
lbu	Carrega byte da memória (rt ←MEM[rs+offset])	lbu $rt, offset($rs)
lh	Carrega half word da memória (rt ←MEM[rs+offset])	lh $rt, offset($rs)
lhu	Carrega half word da memória (rt ←MEM[rs+offset])	lhu $rt, offset($rs)
lui	Carrega a cte nos 16 bits mais significativos do registrador	lui $rt, cte
lw	Carrega word da memória (rt ←MEM[rs+offset])	lw $rt, offset($rs)
sb	Salva byte na memória (MEM[rs+offset] ← rt)	sb $rt, offset($rs)
sh	Salva half word na memória (MEM[rs+offset] ← rt)	sh $rt, offset($rs)
sw	Salva wordna memória (MEM[rs+offset] ← rt)	sw $rt, offset($rs)
mtc1	Move uma word de um reg. de propósito geral para o C1	mtc1 $f1, $s1
mfc1	Move uma word do C1 para um reg. de propósito geral	mfc1 $s1, $f1

Alinhamento em Memória



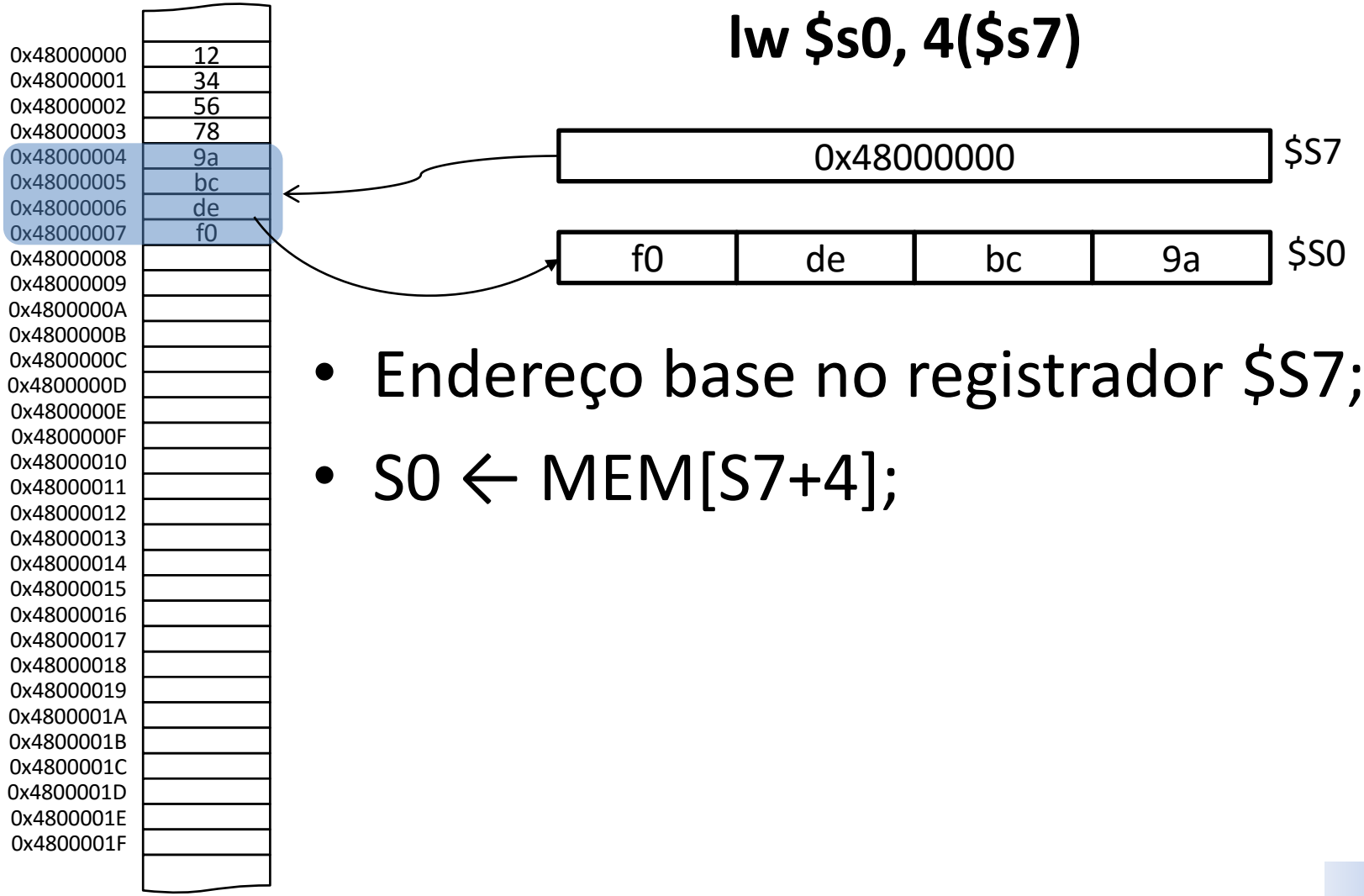
- Instruções devem ser colocadas em posições de memória múltiplas de "4";
- O mesmo se aplica a dados;
- Os endereços múltiplos de 4 (em hexadecimal) começam com:
  - 0x??0; 0x??4; 0x??8; 0x??c
- Em binário:
  - **00**00, 01**00**, 10**00**, 11**00**

LW – Load Word



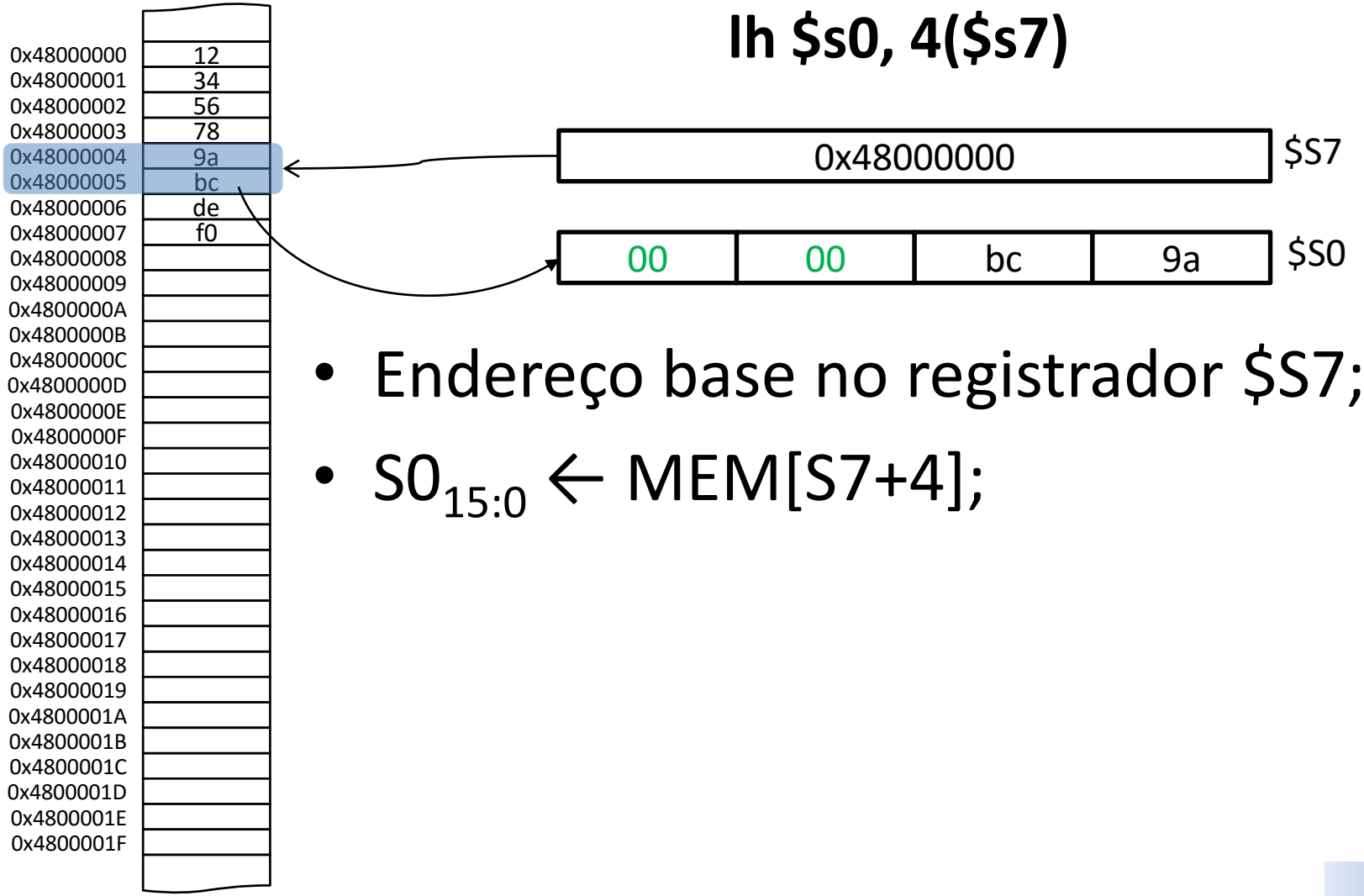
- Endereço base no registrador $S7;
- S0 ← MEM[S7+0];

LW – Load Word



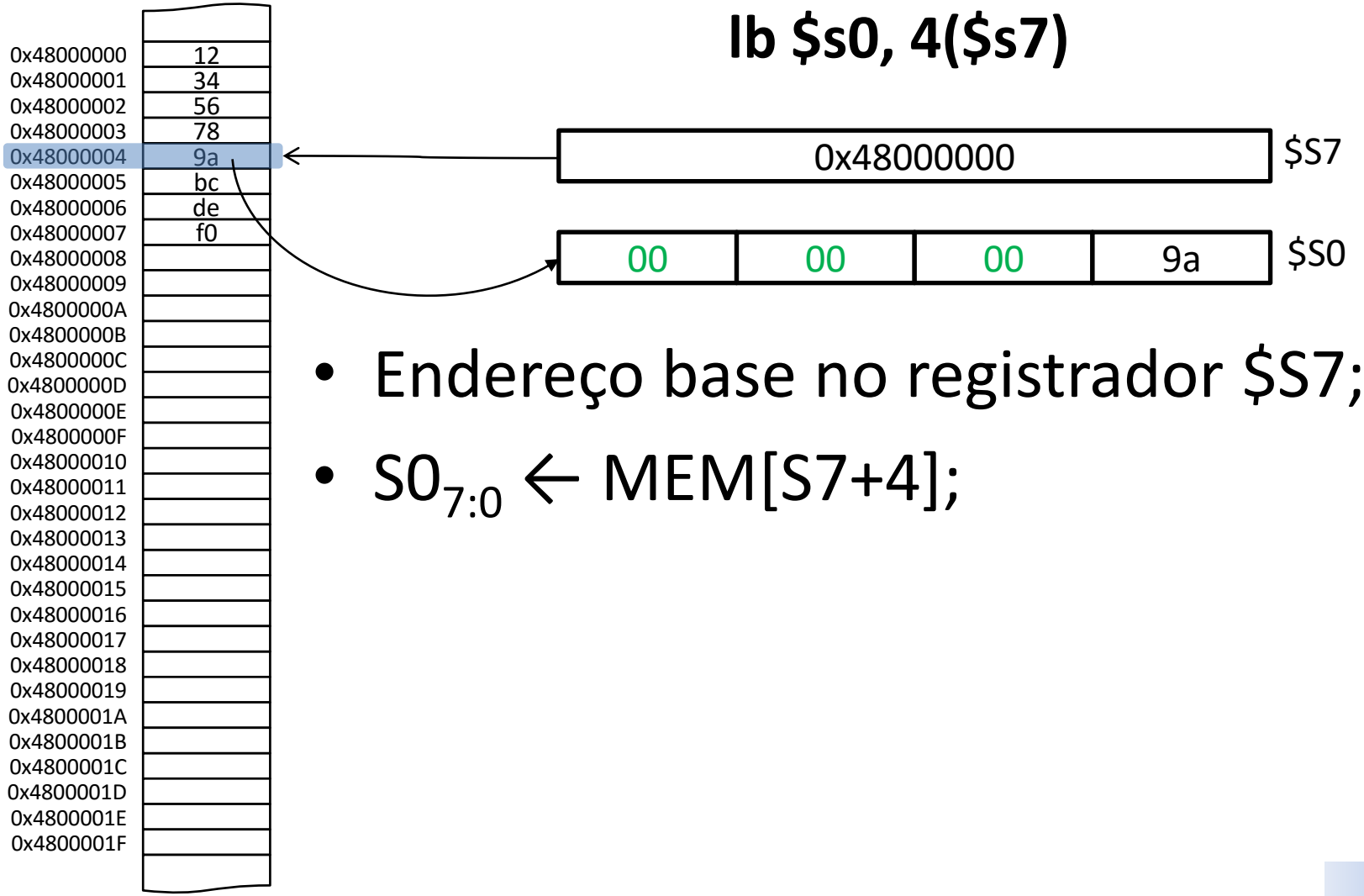
- Endereço base no registrador $S7;
- S0 ← MEM[S7+4];

LH – Load Half



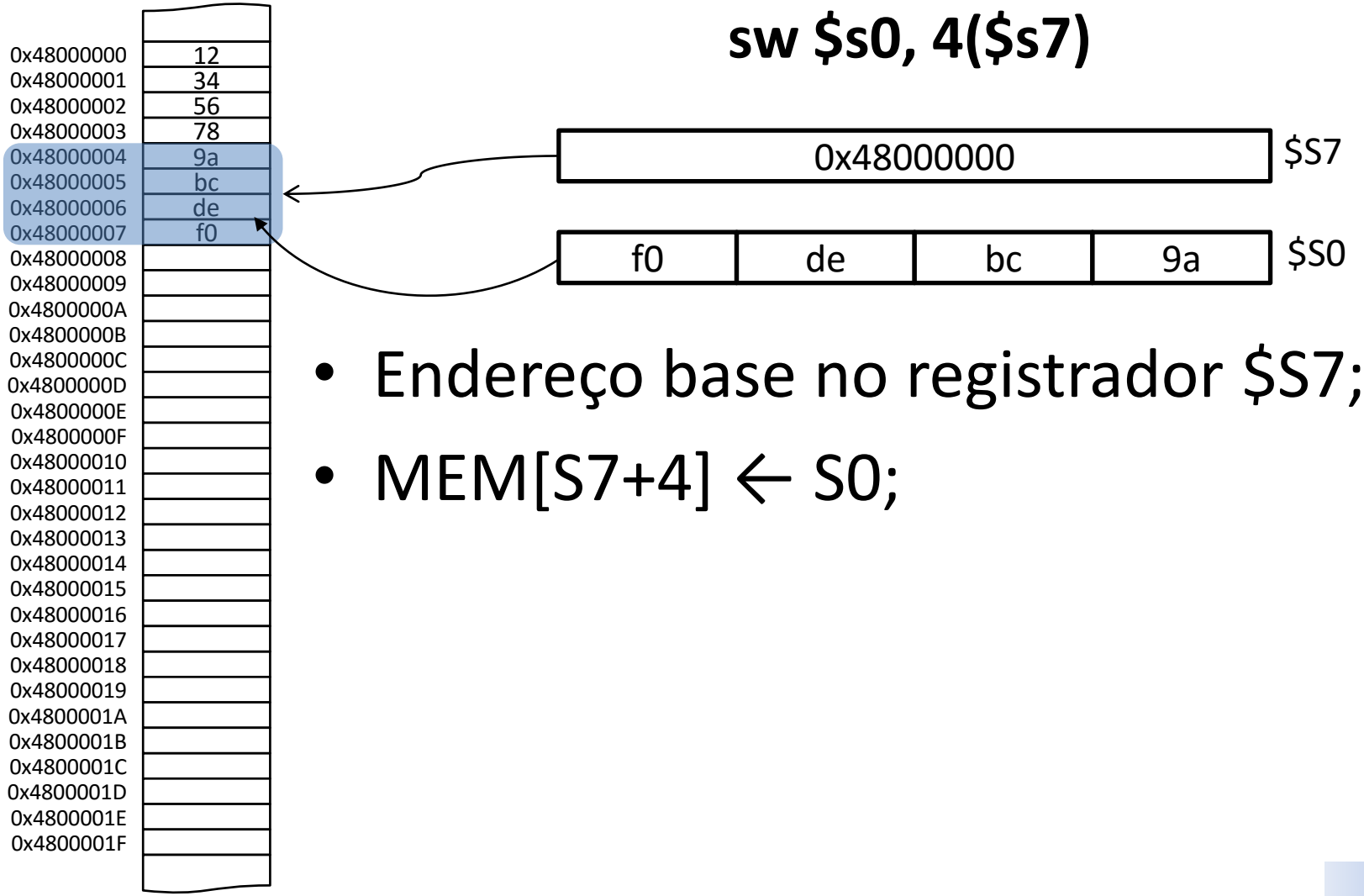
- Endereço base no registrador $S7;
- $S0_{15:0} \leftarrow MEM[S7+4];$

LB – Load Byte



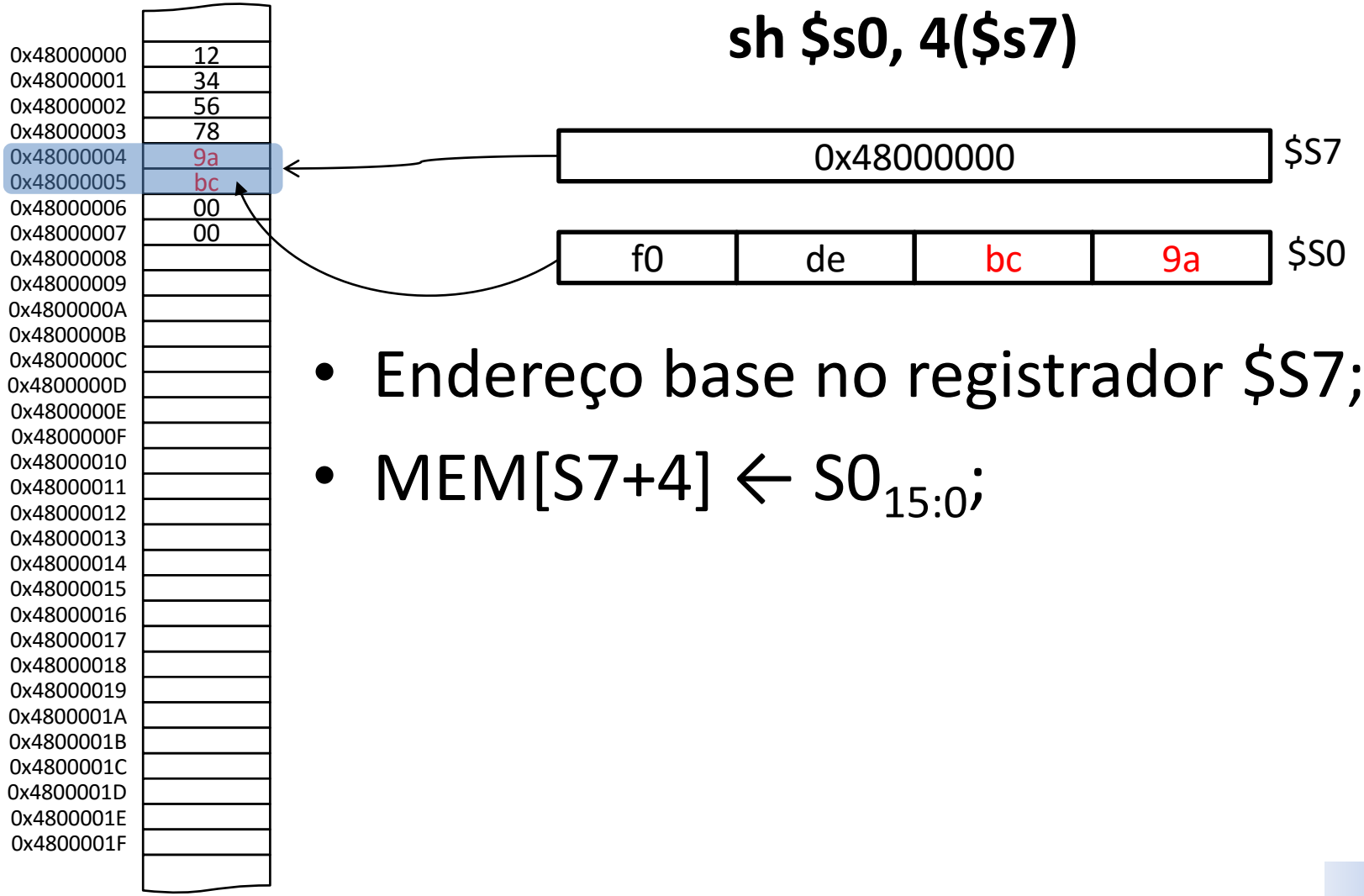
- Endereço base no registrador $S7;
- $S0_{7:0} \leftarrow MEM[S7+4];$

SW – Store Word



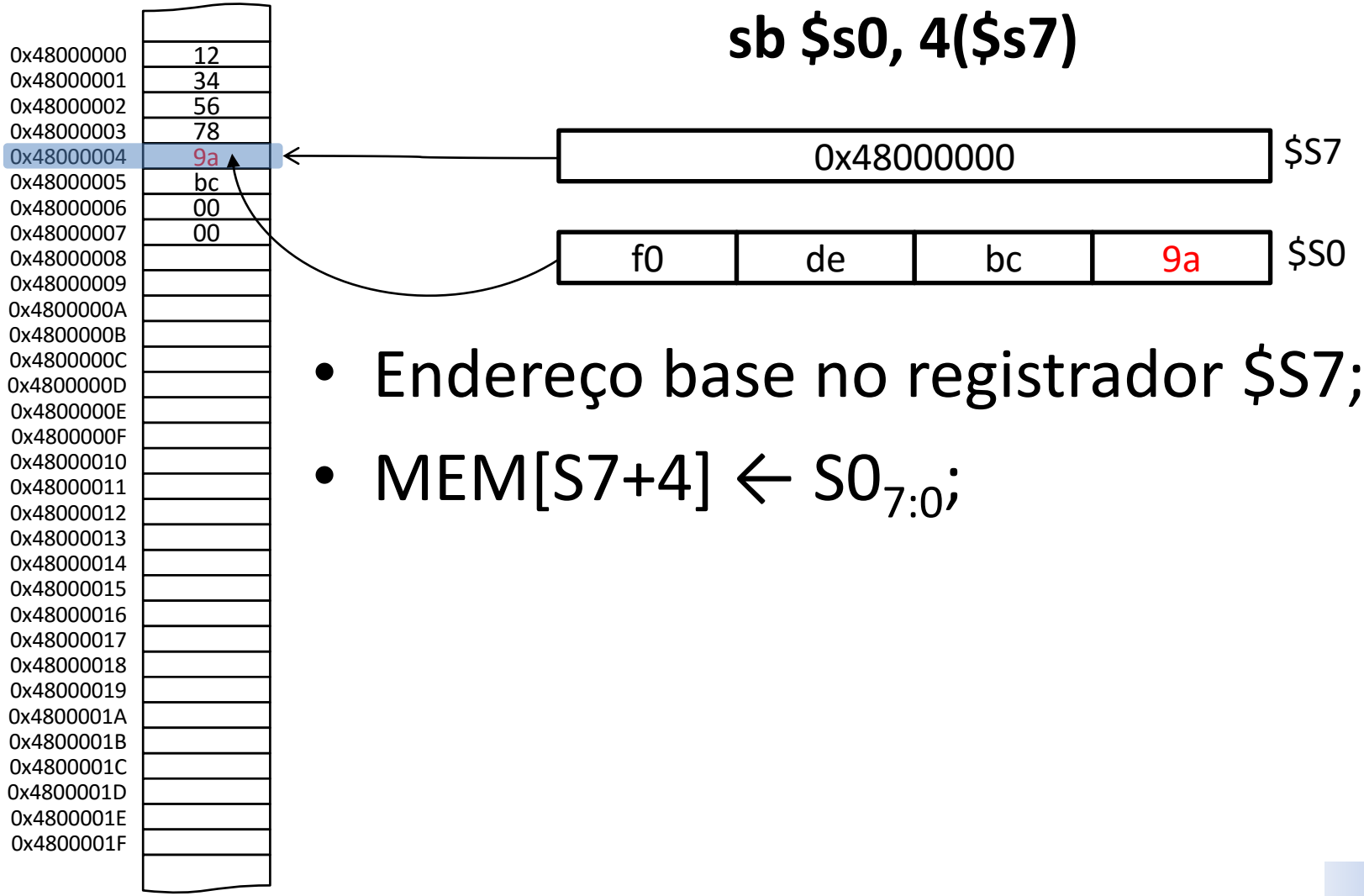
- Endereço base no registrador $S7;
- MEM[S7+4] ← S0;

SH – Store Half



- Endereço base no registrador $S7;
- MEM[S7+4] ← $S0_{15:0}$;

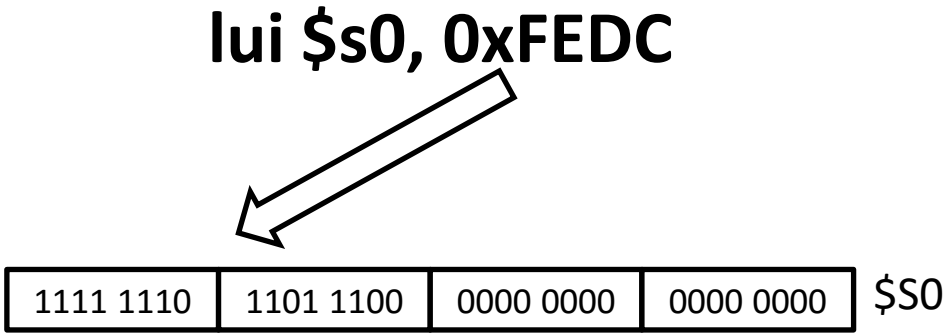
SB – Store Byte



- Endereço base no registrador $S7;
- MEM[S7+4] ← $S0_{7:0}$;

LUI – Load Upper Immediate

- Não necessariamente uma instrução de acesso à memória;
- Carrega 16 bits de uma constante nos dois bytes mais significativos de um array;



Uma Palavra Sobre Arranjos

- Arrays são muito grandes para serem armazenados diretamente em registradores;
- Eles são mantidos em memória;
- Embora sejam simples de utilizar em qualquer linguagem de programação de alto nível, em assembly não é tão simples;
- A melhor forma de entender é pensar em manipulação de arrays via ponteiros tal como usualmente é feito em C;

Uma Palavra Sobre Arranjos

```
...
int arr[10];
int i, *pa;
pa  = &arr[0];
for ( i = 0; i < 10; i++){
  *pa = i * i;
  pa++;
}
...
```

```
.data

#arr: .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
arr: .space 40 # 4 x 10 words (dado em bytes)
sz:  .word 10

i:   .word
pa:  .word

.text
        la      $t0, arr # t0 <- &arr[0]
        la      $t7, sz
        lw      $t7, 0($t7)

        add     $s0, $zero, $zero   # i = 0
        add     $t6, $t0, $zero     #copia de &arr[0]
        addi    $t7, $t7, -1        # tam array [0,9]
FOR1:   beq     $s0, $t7, SAI1

        mul     $t1, $s0, $s0       # i^2
        sw      $t1, 0($t6)

        addi    $t6, $t6, 4         #pa++
        addi    $s0, $s0, 1
SAI1:   li      $v0, 10
        syscall
```

Uma Palavra Sobre Arranjos

RPG – Registrador de Propósito Geral

- Endereço base em um RPG;
- Deslocamento `<<constante>>`;
- Infelizmente o deslocamento constante não pode ser alterado dinâmicamente no programa;
- SOLUÇÃO: Utilizar outro RPG auxiliar para calcular endereço `<<base + deslocamento>>`;

Uma Palavras Sobre <STRUCTS>

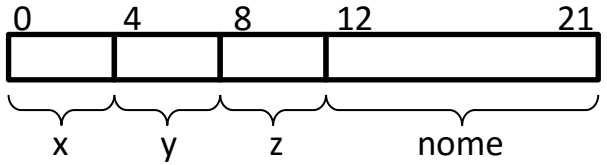
- Estruturas de dados estáticas são uma conveniência ofertada por linguagens de programação de alto nível;
- Não há suporte a structs em assembly;
- Então como um struct é traduzido em assembly?

Uma Palavras Sobre <STRUCTS>

```
typedef struct P3D{
        int x;
        int y;
        int z;
        char nome[10];
    } Ponto3D;
```

32 bits
32 bits
32 bits
80 bits

176 bits = 22 bytes → 24 bytes




```
int main()
{
    Ponto3D p1, p2;

    p1.x = 31;
    p1.y = 42;
    p1.z = 53;

    p2.x = p1.x + p1.y;
    p2.y = p1.y + p1.z;
    p2.z = p1.x + p1.z;

    return 0;
}
```

22 não é múltiplo de 4,
Consequentemente os dados não estarão alinhados!

```
.data
p1: .space 24  #22 efetivamente usados
p2: .space 24
.text
        #$t0 aponta para o primeiro byte da struct p1
        la      $t0, p1
        #$t1 aponta para o primeiro byte da struct p2
        la      $t1, p2

        #####p1.x = 31
        addi    $t7, $zero, 31
        sw      $t7, 0($t0)
        #####p1.y = 42
        addi    $t7, $zero, 42
        #calcula onde está p1.y na memória
        add     $t6, $t0, 4
        sw      $t7, 0($t6)
        #####p1.z = 53
        addi    $t7, $zero, 53
        #calcula onde está p1.y na memória
        add     $t6, $t0, 8
        sw      $t7, 0($t6)
        #p2.x = p1.x + p1.y;
        lw      $t2, 0($t0)   #p1.x
```

```
addi    $t6, $t0, 4
lw      $t3, 0($t6)   #p1.y
add     $t2, $t2, $t3
sw      $t2, 0($t1)   #p2.x = p1.x + p1.y

#p2.y = p1.y + p1.z;
addi    $t6, $t0, 4
lw      $t3, 0($t6)   #p1.y
addi    $t6, $t0, 8
lw      $t2, 0($t6)   #p1.z
add     $t2, $t2, $t3
addi    $t6, $t1, 4
sw      $t2, 0($t6)   #p2.y = p1.y + p1.z

#p2.z = p1.x + p1.z;
lw      $t3, 0($t0)   #p1.x
addi    $t6, $t0, 8
lw      $t2, 0($t6)   #p1.z
add     $t2, $t2, $t3
addi    $t6, $t1, 8
sw      $t2, 0($t6)   #p2.z = p1.x + p1.z

li $v0, 10
syscall
```

Bibliografia Comentada

- **PATTERSON, D. A. e HENNESSY, J. L. 2014.** *Organização e Projeto de Computadores – A Interface Hardware/Software.* Elsevier/ Campus 4ª edição.

- **HENNESSY, J. L. e PATTERSON, D. A. 2012.** *Arquitetura de Computadores – Uma Abordagem Quantitativa.* Elsevier/ Campus 5ª edição.

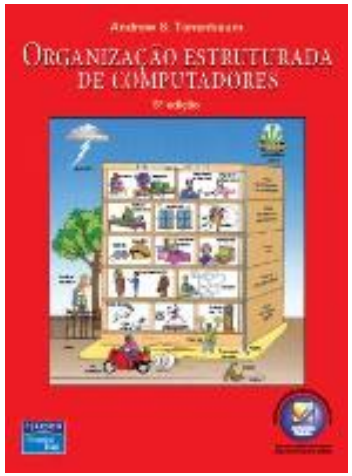
Bibliografia Comentada

- **MONTEIRO, M. A. 2001.** *Introdução à Organização de Computadores.* s.l. : LTC, 2001.

- **MURDOCCA, M. J. e HEURING, V. P. 2000.** *Introdução à Introdução de Computadores.* 2000. 85-352-0684-1.

Bibliografia Comentada

- **STALLINGS, W. 2002.** *Arquitetura e Organização de Computadores.* 2002.



- **TANENBAUM, A. S. 2007.** *Organização Estruturada de Computadores.* 2007.