# Organização Básica de computadores e linguagem de montagem

Prof. Edson
Borin

2º Semestre de 2019

# Subrotinas

# Invocando Subrotinas

"`jal a0, L`" é uma instrução de salto do RISC-V que armazena o endereço da próxima instrução (PC+4) no registrador indicado (`a0`) e depois salta para o rótulo indicado (L).

"`jal L`" é uma pseudo-instrução que é mapeada em: "`jal ra, L`".

- `ra` (*return address*) é o apelido do registrador `x1`.

# Invocando Subrotinas

- Exemplo de subrotina:

```
   ...
   LI  a0, 127
   JAL hash          # chama subrotina hash
   ADD a1, a2, 1     # continua aqui
   ...
# Retorna o valor da função hash em a0
hash:
   AND a1, a0, 63       # a1 <= a0 & 0x3F
   SRL a0, a0, 6        # a0 <= a0 >> 6
   AND a0, a1, a0       # a0 <= a1 & a0
   RET                  # jalr x0, ra, 0
```

# Invocando Subrotinas

- Exemplo de subrotina:

```
   ...
   LI   a0, 127
   JAL hash          # chama subrotina hash
   ADD a1, a2, 1     # continua aqui
   ...
# Retorna o valor da função hash em a0
hash:
   AND a1, a0, 63       #
   SRL a0, a0, 6        #
   AND a0, a1, a0       #
   RET                  # jalr x0, ra, 0
```

Grava o endereço da instrução subsequente (PC+4) no registrador `ra` e salta para hash.

Salta para o endereço armazenado em ra (endereço de retorno).

# Invocando Subrotinas

- O que acontece com o valor de `ra` se a função hash chamar outra subrotina?

```
   ...
   LI  a0, 127
   JAL hash        # chama subrotina hash
   ...
# Retorna o valor da função hash em a0
hash:
   JAL outra_rotina
   RET
```

# Invocando Subrotinas

- O que acontece com o valor de `ra` se a função hash chamar outr

```
   ...
   LI   a0,
   JAL hash
   ...
# Retorna o v                    a0
hash:
   JAL outra_rotina
   RET
```

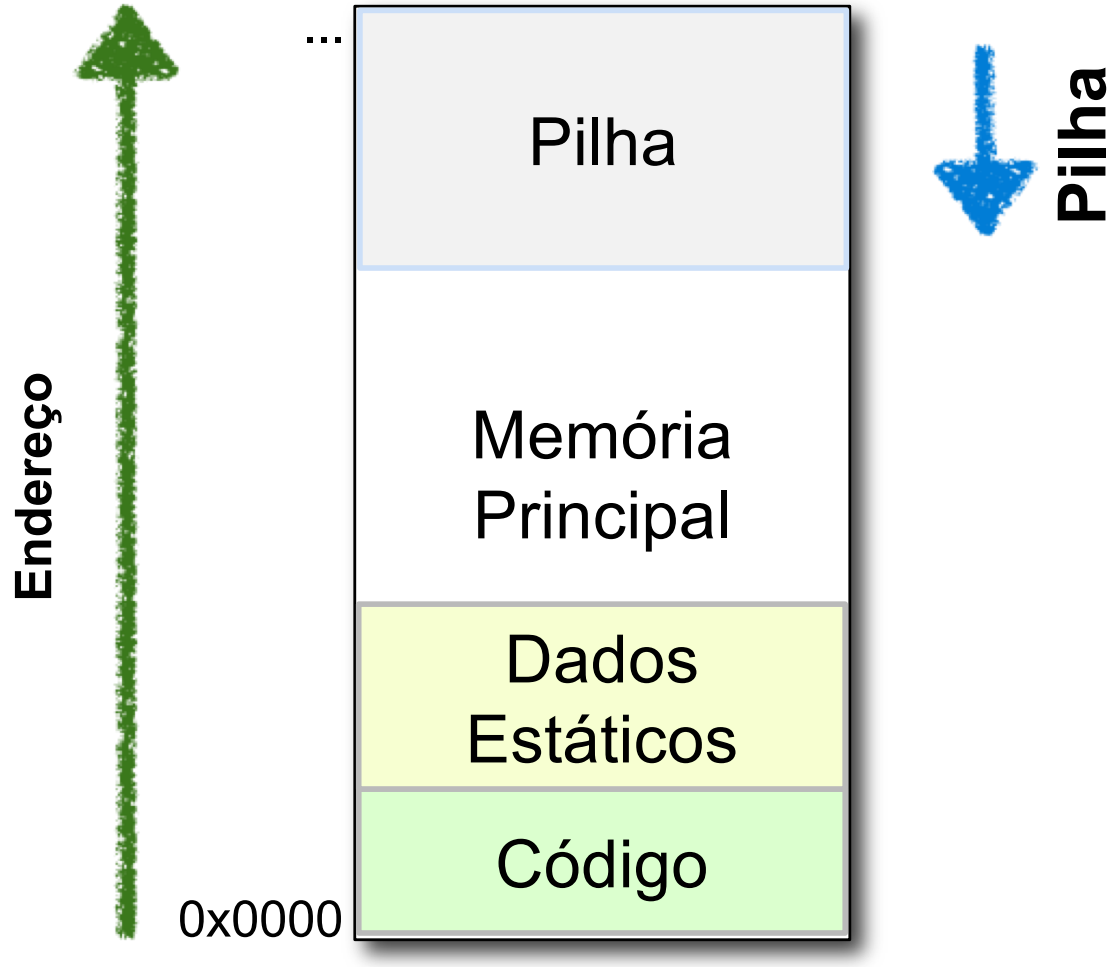
O valor de `ra` será modificado, inviabilizando o retorno para a instrução subsequente à JAL hash.

# Invocando Subrotinas

- Solução: salvar o endereço de retorno na **pilha do programa**.

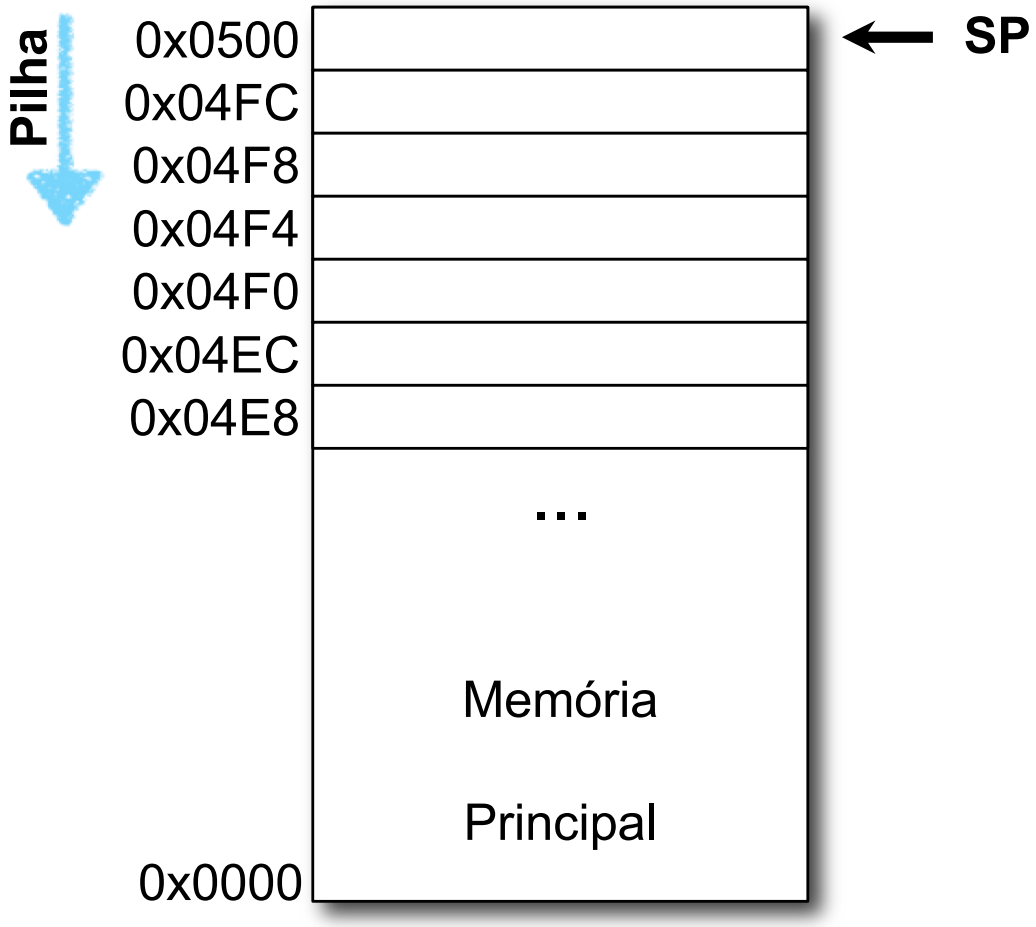
# A pilha do programa

- A pilha do programa é utilizada principalmente para guardar valores temporários.
- A pilha é armazenada na memória principal
- A pilha geralmente cresce de endereços maiores para menores. Pilha descendente.



# A pilha do programa

- O registrador `x2`, ou `SP` (*stack pointer*), aponta para o topo da pilha. Ele deve ser inicializado antes de utilizarmos a pilha.




- Empilhar `a0`

```
addi sp, sp, -4
sw   a0, 0(sp)
```

- Desempilhar `a0`

```
lw   a0, 0(sp)
addi sp, sp, 4
```

# A pilha do programa

- **Pilha descendente**: A pilha cresce do maior endereço para o menor.

- **Pilha ascendente**: A pilha cresce do menor endereço para o maior.

# A pilha descendente

- **Pilha cheia**: O SP aponta para um dado que está no topo pilha.
  - Empilhar: Decrementa SP e depois armazena o dado
  - Desempilhar: Lê o dado e depois incrementa SP.

- **Pilha vazia**: O SP aponta para a posição subsequente à do dado que está no topo da pilha (esta posição está vazia).
  - Empilhar: Escreve o dado e depois decrementa SP
  - Desempilhar: Incrementa SP e depois lê o dado

# A pilha descendente-cheia

- A pilha padrão do RISC-V é **descendente-cheia**, ou *Full-Descendant*.

- Empilhar:

```
addi sp, sp, -4
sw   a0, 0(sp)
```

Aloca espaço na pilha

Salva valor de a0

- Desempilhar:

```
lw   a0, 0(sp)
addi sp, sp, 4
```

Recupera valor de a0

Desaloca espaço da pilha

# (Des)empilhando múltiplos valores

- Empilhando múltiplos valores

```
addi sp, sp, -12
sw   a0, 8(sp)
sw   a1, 4(sp)
sw   a2, 0(sp)
```

Aloca espaço na pilha

Salva valores de a0, a1 e a2 na pilha.

- Desempilhando múltiplos valores

```
lw   a2, 0(sp)
lw   a1, 4(sp)
lw   a0, 8(sp)
addi sp, sp, 12
```

Recupera valores de a2, a1 e a0 da pilha.

Desaloca espaço da pilha

# Exercício

```
li   a1, 1
li   a2, 2
addi sp, sp, -4
sw   a1, 0(sp)
addi sp, sp, -4
sw   a2, 0(sp)
li   a1, 0
li   a2, 0
lw   a1, 0(sp)
addi sp, sp, 4
lw   a2, 0(sp)
addi sp, sp, 4
@ Qual o valor de a1 e a2 após a execução da
@ última instrução?
```

Inicializa a1 e a2

Empilha a1

Empilha a2

Desempilha a1

Desempilha a2

# Como implementar recursão?

- Usamos a Pilha!

```
funcao_rec:
  ADDI sp, sp, -4
  SW   ra, 0(sp)   # Salva RA no início da função
  ...
  JAL  funcao_rec  # Chamada recursiva
  ...
  LW   ra, 0(sp)   # Recupera RA antes de retornar
  ADDI sp, sp,  4
  RET              # Retorna
```

# Como implementar recursão?

- Usamos a Pilha!

```
funcao_rec:
  ADDI sp, sp, -4              # Salva endereço de
  SW   ra, 0(sp)               # retorno na pilha
  ADDI a0, a0, -1
  BLT  a0, zero, fr_retorna
  JAL  funcao_rec              # Chamada recursiva
fr_retorna:
  LW   ra, 0(sp)               # Recupera endereço
  ADDI sp, sp, 4               # de retorno da pilha
  RET                          # Retorna
```

# Como implementar recursão?

- Usamos a Pilha!

```
funcao_rec:
  ADDI sp, sp, -4
  SW   ra, 0(sp)
  ADDI a0, a0, -1
  BLT  a0, zero, fr_retorna
  JAL  funcao_rec              # Chamada recursiva
fr_retorna:
  LW   ra, 0(sp)               # Recupera endereço
  ADDI sp, sp, 4               # de retorno da pilha
  RET                          # Retorna
```

Quantas chamadas à funcao_rec ocorrerá no código abaixo?

```
  LI  a0, 18
  JAL funcao_rec
```

# Passagem de Parâmetros

- Por Registrador
  - –Parâmetros são colocados em registradores
  - –Podem ser especificados parâmetros de entrada ou de saída (retorno de valores)
- Pela Pilha
  - –Parâmetros são colocados na pilha do programa
- RISC-V EABI
  - –8 primeiros parâmetros vão em: `a0`, `a1`, ... `a7`
  - –Parâmetros restantes vão na pilha, empilhados de trás para frente.

# Organização Básica de computadores e linguagem de montagem

Prof. Edson
Borin

2º Semestre de 2019

# Uso da pilha para parâmetros

- Antes da chamada, os parâmetros são empilhados

- Dentro do procedimento, os parâmetros são lidos com o auxílio do registrador de pilha, o SP (*stack pointer*).

- Após a instrução de chamada, o espaço alocado para os parâmetros na pilha é desalocado (Pela mesma rotina que empilhou os parâmetros)

# Passagem de Parâmetros

- Exemplo

```
int soma(int a, int b, int c, int d, int e,
         int f, int g, int h, int i, int j);
# Chamando soma
soma(10,20,30,40,50,60,70,80,90,100);
```

# Passagem de Parâmetros

- Exemplo

```
# Chamando soma
soma(10,20,30,40,50,60,70,80,90,100);
```

```
LI a0, 10          # Parâmetro 1
LI a1, 20          # Parâmetro 2
LI a2, 30          # Parâmetro 3
LI a3, 40          # Parâmetro 4
LI a4, 50          # Parâmetro 5
LI a5, 60          # Parâmetro 6
LI a6, 70          # Parâmetro 7
LI a7, 80          # Parâmetro 8
ADDI sp, sp, -8    # Aloca espaço na pilha
LI t1, 100         # Empilha parâmetro 10
SW t1, 4(sp)
LI t1, 90          # Empilha parâmetro 9
SW t1, 0(sp)
JAL soma           # Chama “soma”
ADDI sp, sp, 8     # Desaloca espaço na pilha
```

# Passagem de Parâmetros

- Exemplo

```
# implementação de soma

soma:
 lw  t1, 0(sp)   # Carrega parâmetro 9 em t1
 lw  t2, 4(sp)   # Carrega parâmetro 10 em t2
 add a0, a0, a1  # Soma parâmetros
 add a0, a0, a2
 add a0, a0, a3
 add a0, a0, a4
 add a0, a0, a5
 add a0, a0, a6
 add a0, a0, a7
 add a0, a0, t1
 add a0, a0, t2  # Retorna valor da soma em a0
 ret             # Retorna
```

# Política de uso dos registradores

- O procedimento pode utilizar muitos registradores.
- Ao chamarmos uma função, gostaríamos de garantir que alguns valores armazenados nos registradores não sejam sobrescritos.
  - Devemos salvar estes valores na memória.
  - Quem deve salvar? A sub-rotina sendo chamada ou a rotina que está invocando a sub-rotina?

# Política de uso dos registradores

- Quem deve salvar? A sub-rotina sendo chamada ou a rotina que está chamando?
- A ABI do RISC-V especifica que:
  - Registradores `t0` a `t6`, `ra` e `a0` a `a7` são *caller-save*. Devem ser salvos pelo código que chamou a rotina.
  - Registradores `s0` a `s11` são *callee-save*. Devem ser salvos pela rotina sendo chamada.
- Os registradores são salvos na PILHA.

# Política de uso dos registradores

- Exemplo – salvando os registradores *callee-save*

```
foo:
 addi sp, sp, -8   # Aloca espaço da pilha
 sw   s1, 4(sp)    # Salva s1 na pilha
 sw   s2, 0(sp)    # Salva s2 na pilha
 ...
 add s1, a0, a1    # Usando o s1
 add s2, a2, a3    # Usando o s2
 ...
 lw   s2, 0(sp)    # Recupera s2 da pilha
 lw   s1, 4(sp)    # Recupera s1 da pilha
 addi sp, sp,  8   # Desaloca espaço da pilha
 ret               # Retorna
```

# Retorno de valor em funções

- RISC-V ABI
  - Se for um valor de até 32 *bits* retornamos o valor em `a0`.
  - Se o valor tiver de 32 a 64 *bits*, retornamos o valor no par `a1:a0`.

# Passagem de parâmetros por Valor

- Suponha a função C

```
int ContaUm(int v)
{
  ...
}
```

- Exemplo de chamada

```
int x,y;
...
y = ContaUm(x);
```

# Passagem de parâmetros por Valor

- Suponha a função C

```
# Conta o número de bits 1
# entrada: palavra de 32 bits em a0
# saida: número de bits 1 em a0
# destrói: conteúdo de a1 e t1
ContaUm:
   MV a1, a0          # Move o parâmetro para a1
   LI a0, 0           # Inicia o contador de bits com 0
loop:
   ANDI t1, a1, 1     # Seta t1 com 1 se o LSB de a1 for 1
   ADD  a0, a0, t1    # Incrementa a0 se o LSB de a1 for 1
   SRLI a1, a1, 1     # Desloca bits de a1 para a direita
   BNEZ a1, loop      # Continua enquanto a1 != 0
   RET                # Retorna
```

# Passagem de parâmetros por Valor

- y = ContaUm(x);

```
...
y: .word 123
x: .word 0
...
   LW  a0, x     # Carrega o valor de x em a0
   JAL ContaUm   # Invoca a função ContaUm
   SW  a0, y     # Grava o retorno em y
```

# Passagem de parâmetros por Referência

- Considere o procedimento:

```
void troca(int *a, int* b)
{
      int x;
      x = *a;
      *a = *b;
      *b = x;
}
```

- Exemplo de chamada:

```
int x, y;
...
troca(&x, &y);
```

# Passagem de parâmetros por Referência

- troca(&x, &y);

```
...
y: .word 9
x: .word 10
...
   LA a0, x  # Carrega o endereço de x em a0
   LA a1, y  # Carrega o endereço de y em a1
   JAL troca # Invoca a função troca
...
```

# Implementação de Troca

```
void troca(int *a, int* b)
{
  int x;
  x = *a;
  *a = *b;
  *b = x;
}
```

# Implementação de Troca

```
void troca(int *a, int* b)
{
   int x;
   x = *a;
   *a = *b;
   *b = x;
}
```

```
# troca: troca dois valores
# entrada: endereços na pilha
# suja: t1, t2
troca:
    LW  t1, 0(a0) # Carrega *a em t1
    LW  t2, 0(a1) # Carrega *b em t2
    SW  t2, 0(a0) # Grava t2 em *a
    SW  t1, 0(a1) # Grava t1 em *b
    RET
```

# Variáveis Locais

Idealmente, as variáveis locais de um procedimento ou função devem ser alocadas em registradores. Entretanto, em alguns casos, as variáveis locais precisam ser alocadas na memória. Por exemplo:

- Quando um procedimento utiliza muitas variáveis locais, e o número de registradores não é suficiente para alocá-las;
- Quando a variável local é um registro ou vetor;
- Quando a rotina faz uso do endereço da variável local;

# Variáveis locais na memória

Variáveis locais que precisam ser alocadas na memória são alocadas na pilha!

- O espaço para as variáveis locais é reservado na entrada do procedimento, e desalocado ao final do procedimento
- Mas isso altera o valor de SP, precisamos ter cuidado ao acessar os parâmetros

# Variáveis locais na memória

- Utilizamos outro registrador para facilitar acesso a parâmetros e variáveis locais.
- *Frame Pointer* (FP): apontador de quadro
- No RISC-V, FP é sinônimo de s0 ou x8
- FP marca a posição de SP na entrada do procedimento
  - Estabelece um **ponto fixo** de acesso

# Variáveis Locais

- Exemplo:

```
int func(int a, int b, int c, int d, int e,
         int f, int g, int h, int i, int j);
func:
 addi sp, sp, -32 # Cria espaço na pilha
 sw   ra, 28(sp)  # Salva o endereço de retorno
 sw   fp, 24(sp)  # Salva o Frame Pointer anterior
 addi fp, sp, 32  # Define o Frame Pointer atual
 lw   t1, 0(fp)   # Carrega parâmetro 9 em t1
 lw   t2, 4(fp)   # Carrega parâmetro 10 em t2
 [... corpo da função ...]
 sw   t0, -12(fp) # Salva t0 em variável local
 [... corpo da função ...]
 sw   ra, 28(sp)
 sw   fp, 24(sp)
 addi sp, sp, 32
 ret
```

# Considerações da ABI padrão do RISCV

- A pilha é descendente-cheia e o apontador de pilha **(SP) deve sempre apontar para um endereço múltiplo de 16 *bytes***.

```
# implementação de soma
```

# Representação de Registros na Memória

```
struct id {
      int cpf;
      char nome[256];
      int idade;
};
```

- Como representar?
- Como acessar os campos?

# Representação de Registros na Memória

```
struct no {
     struct no* prox;
     int valor;
};

struct no x;
struct no y;
...
x.valor = 5;
x.prox  = &y;
y.valor = 3;
y.prox  = 0;
```

# Representação de Registros na Memória

```
struct no {
     struct no* prox;
     int valor;
};

struct no x;
struct no y;
...
x.valor = 5;
x.prox  = &y;
y.valor = 3;
y.prox  = 0;
```

```
x: .skip 8
y: .skip 8

...

  LA  a3, x        # a3 <= &x
  LI  a1, 5        # a1 <= 5
  SW  a1, 4(a3)    # x.valor <= a1
  LA  a4, y        # a4 <= &y
  SW  a4, 0(a3)    # x.prox <= a4
  LI  a1, 3        # a1 <= 3
  SW  a1, 4(a4)    # y.valor = a1
  SW  zero, 0(a4) # y.prox = 0
```

# Chamada de sistema

- Programas de usuário geralmente operaram com dados armazenados na memória e nos registradores.
- Entrada e saída de dados são realizadas com o auxílio de dispositivos de entrada e saída:
  - Teclado, monitor, impressora, rede, etc...
  - Estes dispositivos são gerenciados pelo sistema operacional e a entrada e saída é feita através de requisições ao sistema operacional.
- Requisições ao sistema operacional => Chamadas de sistemas (*system call* ou *syscall*)

# Chamada de sistema

- Exemplo: escrita em arquivo

```
char* msg = "My message”;
char* filename = "my_file.txt”;
...
/* Set the flags. */
int flags = O_WRONLY | O_CREAT | O_TRUNC;
/* Open the file. */
int fd = open (filename,flags);
/* Write the first 5 bytes pointed by msg into the file. */
write(fd, msg, 5);
/* Close the file. */
close(fd);
```

# Chamada de sistema

- Exemplo: chamando a *syscall write*

```
ssize_t write(int fildes, const void *buf, size_t nbyte);

# Ajustar os parâmetros
li a0, fd     # a0: Valor do file descriptor
la a1, msg    # a1: Apontador para o buffer
li a2, 5      # a2: Número de bytes a serem escritos
# Chamar a função write
```

# Chamada de sistema

- Exemplo: chamando a *syscall write*

```
ssize_t write(int fildes, const void *buf, size_t nbyte);

# Ajustar os parâmetros
li a0, fd     # a0: Valor do file descriptor
la a1, msg    # a1: Apontador para o buffer
li a2, 5      # a2: Número de bytes a serem escritos
# Chamar a função write
li a7, 64     # Código da syscall: 64 == write
ecall         # Invoca o sistema operacional
```

# Chamada de sistema

- Podemos implementar uma função *write* que encapsula a chamada à *syscall*:

```
# Entrada: a0: descritor do arquivo (fd)
#          a1: apontador para o buffer
#          a2: número de bytes a ser escrito
# Saída:a0: número de bytes escrito pela write.
write:
   addi sp, sp, -16
   sw ra, 0(sp)          # Salva ra
   li  a7, 64            # Código da syscall: 64 == write
   ecall                 # Invoca o sistema operacional
   lw ra, 0(sp)          # Restaura ra
   addi sp, sp, 16
   ret
```

# Chamada de sistema

- Chamada de sistema *read*

```
# Entrada: a0: descritor do arquivo (fd)
#          a1: apontador para o buffer
#          a2: número de bytes a ser lido
# Saída:   a0: número de bytes lidos pela read.
read:
   addi sp, sp, -16
   sw ra, 0(sp)          # Salva ra
   li  a7, 63            # Código da syscall: 63 == read
   ecall                 # Invoca o sistema operacional
   lw ra, 0(sp)          # Restaura ra
   addi sp, sp, 16
   ret
```