

# Organização Básica de computadores e linguagem de montagem

Prof. Edson Borin

2º Semestre de 2016

# Subrotinas

# *Branch and Link*

- BL é uma instrução de salto do ARM que armazena o endereço da próxima instrução no registrador R14, ou LR (link register), antes de saltar.

# *Branch and Link*

- Exemplo de subrotina:

...

MOV R0, #127

BL hash @ chama subrotina hash

ADD R1, R2, #1 @ continua aqui

...

@ Retorna o valor da função hash em R0

hash:

AND R1, R0, #63

AND R0, R1, R0, LSR 6

MOV PC, LR

# *Branch and Link*

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

...

```
MOV R0, #127
```

```
BL hash          @ chama subrotina hash
```

...

@ Retorna o valor da função hash em R0

hash:

```
BL outra_rotina
```

```
MOV PC, LR
```

# Branch and Link

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

```
...  
MOV R0, ...  
BL hash  
...  
@ Retorna o valor em R0  
hash:  
    BL outra_rotina  
    MOV PC, LR
```

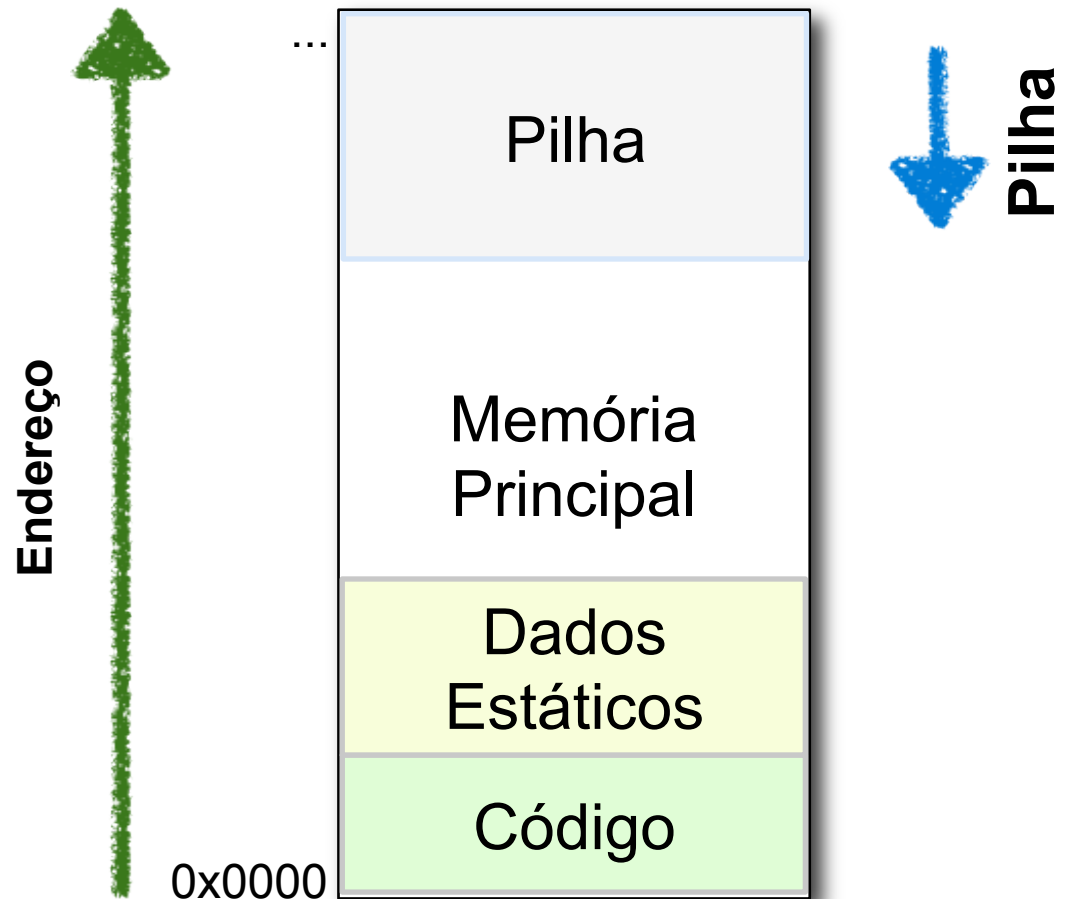
O valor de LR será  
modificado, inviabilizando o  
retorno para a instrução  
subsequente à BL hash.

# *Branch and Link*

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

# A pilha

- A pilha 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

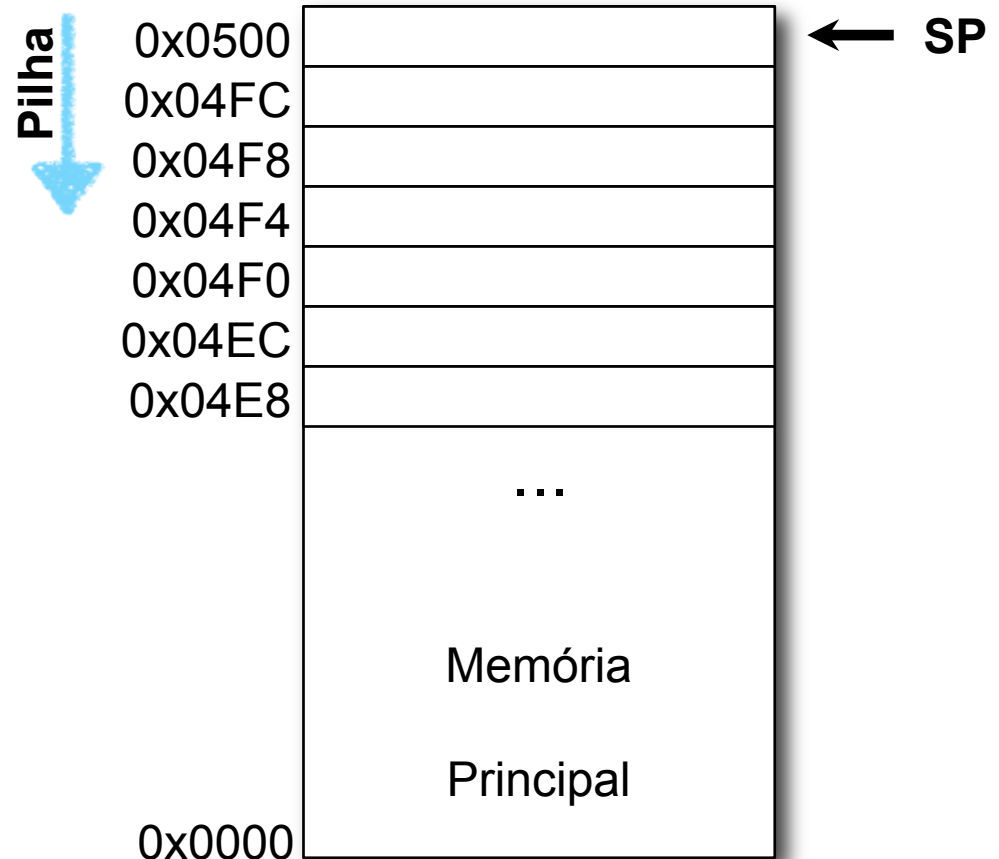
- O registrador R13, ou SP (*stack pointer*), aponta para o topo da pilha. Ele deve ser inicializado antes de utilizarmos a pilha.

- Empilhar Rn

STMDB SP!, {Rn}

- Desempilhar Rn

LDMIA SP!, {Rn}



# A pilha

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

# A Pilha Decrescente

- **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 Decrescente-Cheia

- A pilha padrão do ARM é **decrescente-cheia**, ou *Full-Descendant*.
- Empilhar:
  - STR Rn, [SP, #-4]!
  - STMDB SP!, {Rn}
- Desempilhar:
  - LDR Rn, [SP], #4
  - LDMIA SP!, {Rn}
- Em vez de usar STMDB e LDMIA, podemos usar os apelidos: STM**FD** e LDM**FD**.

# Exercício

```
mov R0, #0
```

```
mov R1, #1
```

```
mov R2, #2
```

```
STMFD SP!, {R0}
```

```
STMFD SP!, {R1}
```

```
STMFD SP!, {R2}
```

```
EOR R0, R0, R0
```

```
EOR R1, R1, R1
```

```
EOR R2, R2, R2
```

```
LDMFD SP!, {R0}
```

```
LDMFD SP!, {R1}
```

```
LDMFD SP!, {R2}
```

@ Qual o valor de R0, R1 e R2 após a execução da  
@ última instrução?

# Como implementar recursão?

- Usamos a Pilha!

funcao\_rec:

STMFD SP!, {LR} @ Salva LR no início da função

...

BL funcao\_rec @ Chamada recursiva

...

LDMFD SP!, {LR} @ Recupera LR antes de retornar

MOV PC, LR @ Retorna

# Como implementar recursão?

- Usamos a Pilha!

```
funcao_rec:
```

```
    STMFD SP!, {LR}    @ Salva LR
    SUBS R0, R0, #1
    BLGE funcao_rec    @ Chamada recursiva
    LDMFD SP!, {LR}    @ Recupera LR
    MOV PC, LR        @ Retorna
```

```
@ Quantas chamadas à funcao_rec
```

```
@ ocorrerá no código abaixo?
```

```
MOV R0, #18
```

```
BL funcao_rec
```

# Passagem de Parâmetros

- Por Registrador
  - Especifica-se quais registradores devem ser usados
  - Podem ser especificados parâmetros de entrada ou de saída
- Pela Pilha
  - Parâmetros são colocados na pilha do programa
- ARM EABI
  - 4 primeiros parâmetros vão em: r0, r1, r2 e r3
  - Parâmetros restantes vão na pilha, empilhados de trás para frente.



# 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 chamada, o espaço alocado para os parâmetros na pilha é desalocado.

# Passagem de Parâmetros

- Exemplo

```
int soma(int a, int b, int c,  
         int d, int e, int f);
```

@ Chamando soma

```
soma(10,20,30,40,50,60);
```

# Passagem de Parâmetros

- Exemplo

```
int soma(int a, int b, int c,  
        int d, int e, int f);
```

```
@ Chamando soma
```

```
mov r0, #10
```

```
mov r1, #20
```

```
mov r2, #30
```

```
mov r3, #40
```

```
mov r4, #60
```

```
stmfd sp!, {r4} @ Empilha 60
```

```
mov r4, #50
```

```
stmfd sp!, {r4} @ Empilha 50
```

```
bl soma
```

```
add sp, sp, #8 @ Desempilha parâmetros
```

# Passagem de Parâmetros

- Exemplo

```
int soma(int a, int b, int c,  
        int d, int e, int f);
```

soma:

```
ldr r4, [sp]      @ load e into r4  
ldr r5, [sp, #4]  @ load f into r5  
...  
mov pc, lr
```

# 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 ARM especifica que:
  - Registradores R0 a R3 são *caller-save*. Devem ser salvos pelo código que chamou a rotina.
  - Registradores R4 a R11 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

```
int soma(int a, int b, int c,  
        int d, int e, int f);
```

soma:

```
stmfd sp!, {r4-r11, lr} @ Salva regs
```

```
ldr r4, [sp, #36] @ carrega e em r4
```

```
ldr r5, [sp, #40] @ carrega f em r5
```

```
... @ corpo da funcao
```

```
ldmfd sp!, {r4-r11, lr} @ Recupera regs
```

```
mov pc, lr
```

# Política de uso dos registradores.

- Exemplo – salvando os registradores

```
int soma(int a, int b, int c,  
        int d, int e, int f);
```

soma:

```
stmfd sp!, {r4-r11, lr} @ Salva regs  
ldr r4, [sp, #36] @ carrega e em r4  
ldr r5, [sp, #40] @ carrega f em r5  
... @ corpo da funcao  
ldmfd sp!, {r4-r11, pc} @ Recupera regs
```

Recupera o valor de LR diretamente em PC



# Retorno de valor em funções

- ARM ABI

- Se for um valor de até 32 *bits* retornamos o valor em r0.
- Se o valor tiver de 32 a 64 *bits*, retornamos o valor no par r1:r0.

# Organização Básica de computadores e linguagem de montagem

Prof. Edson Borin

2º Semestre de 2016

# 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 r0

@ saída: número de *bits* 1 em r0

@ destrói: r1

ContaUm:

mov r1, r0 @ move o parâmetro para r1

mov r0, #0 @ inicia o contador de *bits* com 0

loop:

movs r1, r1, LSR #1 @ Desloca r1 e atualiza a *carry*

addcs r0, r0, #1 @ incrementa contador se *carry set*

cmp r1, #0 @ continua enquanto

bne loop @ r1 não for zero

mov pc, lr @ retorna da função

# Passagem de parâmetros por Valor

- `y = ContaUm(x);`

...

```
y: .word 10
```

```
x: .word 0
```

...

```
ldr r0, =x      @ carrega o
```

```
ldr r0, [r0]    @ valor de x
```

```
bl ContaUm
```

```
ldr r1, =y      @ armazena o resultado
```

```
str r0, [r1]    @ 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`

...

`ldr r0, =x`

`ldr r1, =y`

`bl 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
@ destrói: r2, r3
troca:
    ldr r2, [r0]
    ldr r3, [r1]
    str r2, [r1]
    str r3, [r0]
    mov pc, lr
```

# Variáveis Locais

- Quando um procedimento utiliza muitas variáveis locais, e o número de registradores não é suficiente para aloca-las, a pilha também é utilizada para armazenar estas variáveis locais.
- 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

- Utilizamos outro registrador para facilitar acesso a parâmetros e variáveis locais.
- *Frame Pointer* (FP): apontador de quadro
- No ARM, FP é sinônimo de r11
- FP marca a posição de SP na entrada do procedimento
  - Estabelece um **ponto fixo** de acesso

# Variáveis Locais

- Exemplo:

```
int soma(int a, int b, int c, int d, int e, int f);
```

```
soma:
```

```
stmfd sp!, {r4-r11, lr} @ Salva regs
```

```
add fp, sp, #32 @ seta fp
```

```
sub sp, sp, #12 @ aloca 3 palavras
```

```
ldr r4, [fp, #4] @ carrega param. e em r4
```

```
ldr r5, [fp, #8] @ carrega param. f em r5
```

```
... @ corpo da funcao
```

```
add sp, sp, #12 @ ???
```

```
ldmfd sp!, {r4-r11, lr} @ Recupera regs
```

```
mov pc, lr
```

# 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 = NULL;
```

# 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

```
ldr r0, =fd    @ r0: Valor do file descriptor
```

```
ldr r0, [r0]
```

```
ldr r1, =msg    @ r1: Apontador para o buffer
```

```
mov r2, #5      @ r2: 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

```
ldr r0, =fd      @ r0: Valor do file descriptor
```

```
ldr r0, [r0]
```

```
ldr r1, =msg     @ r1: Apontador para o buffer
```

```
mov r2, #5       @ r2: Número de bytes a serem escritos
```

@ Chamar a *syscall write*

```
mov r7, #4       @ Código da syscall: 4 == write
```

```
svc #0           @ Invoca o sistema operacional
```

# Chamada de sistema

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

@ Entrada:

@ r0: descritor do arquivo (fd)

@ r1: apontador para o buffer

@ r2: número de bytes a ser escrito

@ Saída:

@ r0: número de bytes escrito pela write.

*write*:

```
stmfd sp!, {r7, lr} @ Salva regs
```

```
mov r7, #4 @ Código da syscall: 4 == write
```

```
svc #0 @ Invoca o sistema operacional
```

```
ldmfd sp!, {r7, pc}
```

# Chamada de sistema

- Chamada de sistema *read*

@ Entrada:

@ r0: descritor do arquivo (fd)

@ r1: apontador para o buffer

@ r2: número de bytes a ser lido

@ Saída:

@ r0: número de bytes lido pela read.

read:

```
stmfd sp!, {r7, lr} @ Salva regs
```

```
mov r7, #3 @ Código da syscall: 3 == read
```

```
svc #0 @ Invoca o sistema operacional
```

```
ldmfd sp!, {r7, pc}
```