Software Básico

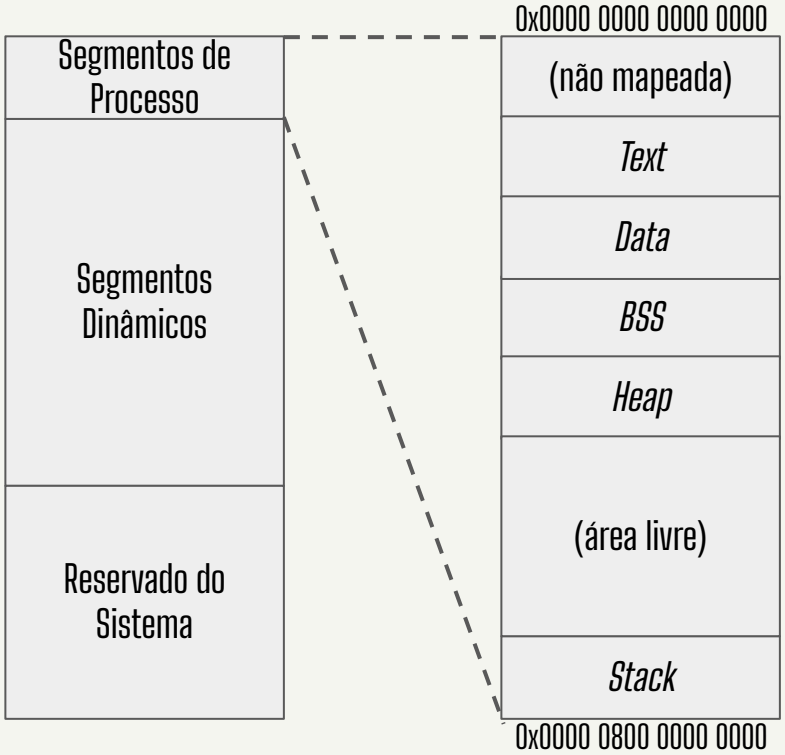
Aula #06

Pilha: Procedimentos Sem Parâmetros e Variáveis Locais

O que é exatamente a seção pilha? Como criar procedimentos simples em *assembly*? Como operacionalizar os procedimentos através da pilha?

Ciência da Computação - BCC2 - 2023/02
Prof. Vinícius Fülber Garcia

Relembrando



Seções do segmento de processo:

- *Text*: código
- *Data*: dados globais inicializados
- *BSS*: dados não inicializados
- *Heap*: variáveis dinâmicas
- *Stack*: pilha de execução

Pilha: Escopo

O foco desta aula, e das próximas também, será explorar a seção de pilha (*stack*).

- Procedimentos
- Registros de ativação
- Parâmetros
- Variáveis locais

Tradução de programas em C para *assembly* (AMD64), gerando aplicações com funcionalidade equivalente.

Um Pouco de História

Houve uma era em que não existiam procedimentos e variáveis locais!

- Assembly
- Fortran (~1954)
- Cobol (~1959)

Nesse tempo, haviam diversos desafios a serem considerados:

- Repetição de código
- Programas grandes e complexos
- Ausência de comandos estruturados (tudo na base do *goto*)

Um Pouco de História

E quando isso mudou?

Na proposta da linguagem de programação ALGOL!

Essa linguagem foi conceitualizada em 1958 e lançada, em primeira versão, em 1960. A linguagem ALGOL foi revolucionária, lançando a era das linguagens estruturadas: agora as linguagens de programação não eram mais tão parecidas com *assembly*.

E nesse contexto é que a seção pilha se tornou fundamental!

Abstraindo os Conceitos

Vamos começar a entender a seção pilha através de um modelo que abstrai e concretiza a mesma!

MODELO DE DIAGRAMAS DE EXECUÇÃO
(Tomasz Kowaltowski)

Nesse modelo, folhas de papel armazenam informações de um procedimento, e cada procedimento é representado por uma folha de papel.
As folhas de papel são empilhadas, representando a execução de um programa.

Abstraindo os Conceitos

```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```

As regras são:

- Uma função em execução utiliza uma folha;
- Funções que não retornaram ficam na pilha;
- A função em execução está no topo da pilha;
- A pilha armazena o valor de todas as variáveis usadas.

Abstraindo os Conceitos

```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```

Nova função, nova folha!

Escreve o valor das variáveis!

a = 10
b = 5
retorno = ?

MAIN

Abstraindo os Conceitos

```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```

Chama uma nova função.

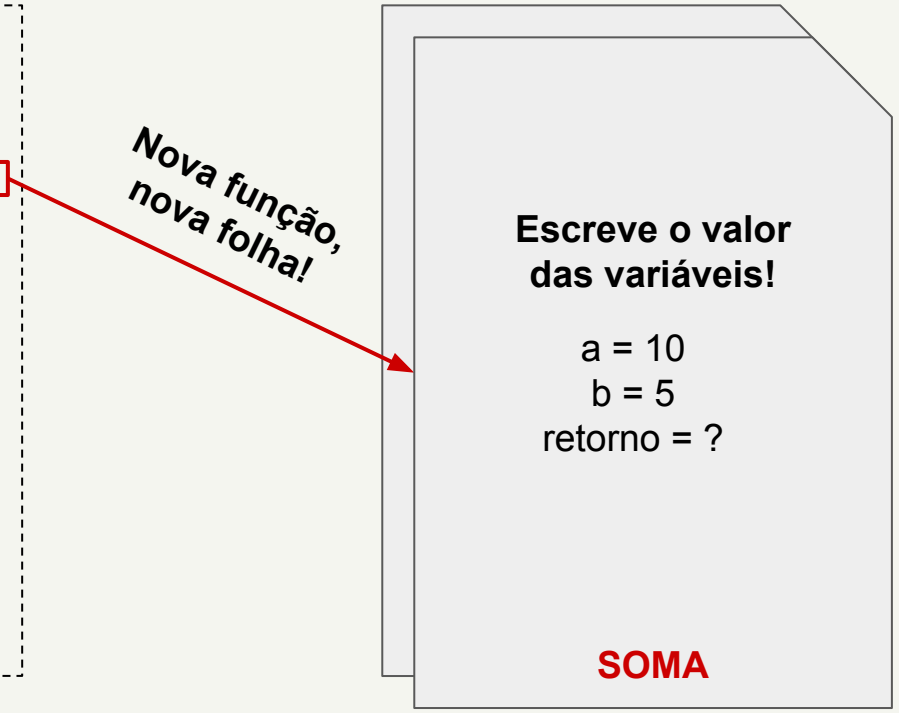
Mas a *main* ainda não acabou!

a = 10
b = 5
retorno = ?

MAIN

Abstraindo os Conceitos

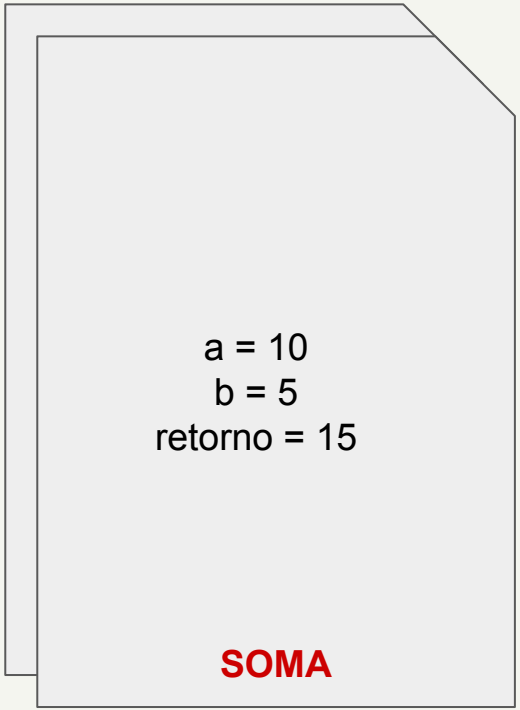
```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```



Abstraindo os Conceitos

```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```

Realiza a soma e insere o resultado no reg. de retorno.



Abstraindo os Conceitos

```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```

Retorna!

Com o retorno, esta folha é removida!

a = 10
b = 5
retorno = 15

SOMA

Abstraindo os Conceitos

```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```

Move o valor de retorno para a!

a = 15
b = 5
retorno = 15

MAIN

Abstraindo os Conceitos

```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```

Chama uma nova função.

Mas a *main* ainda não acabou!

a = 15
b = 5
retorno = 15

MAIN

Abstraindo os Conceitos

```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```

Nova função, nova folha!

Escreve o valor das variáveis!

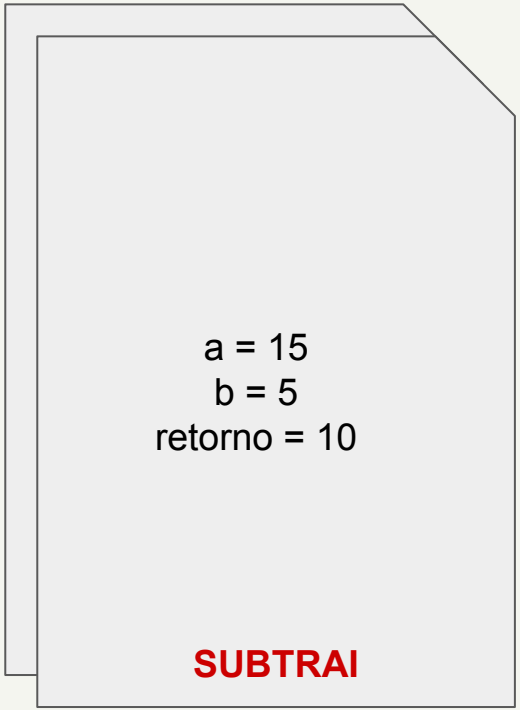
a = 15
b = 5
retorno = 15

SUBTRAI

Abstraindo os Conceitos

```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```

Realiza a subtração e insere o resultado no reg. de retorno.



Abstraindo os Conceitos

```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```

Retorna!

Com o retorno, esta folha é removida!

a = 15
b = 5
retorno = 10

SUBTRAI

Abstraindo os Conceitos

```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```

Move o valor de retorno para b!

a = 15
b = 10
retorno = 10

MAIN

Abstraindo os Conceitos

```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```

Chama uma nova função.

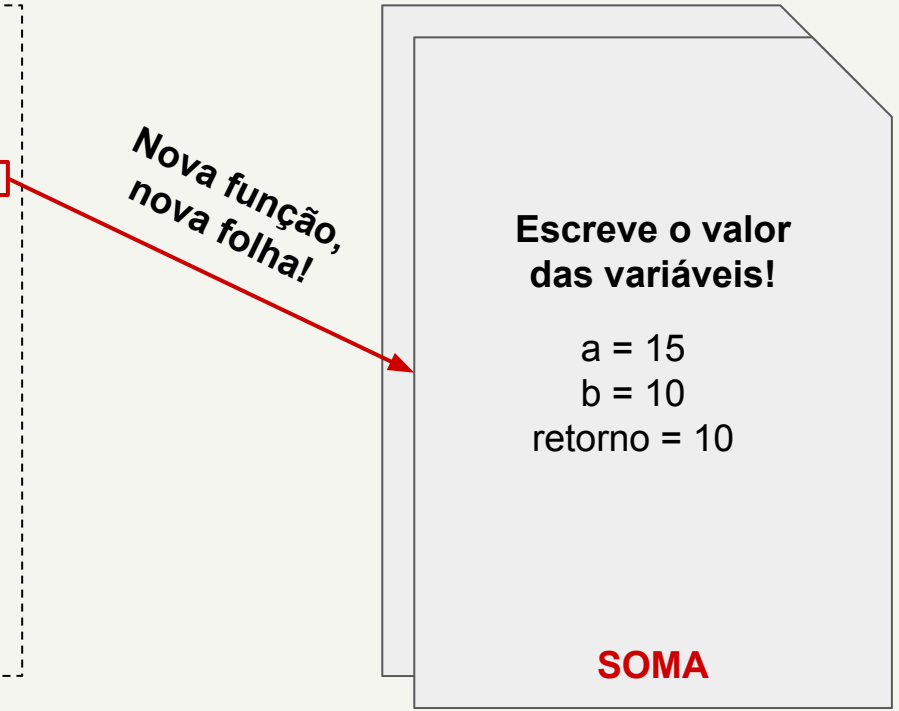
Mas a *main* ainda não acabou!

a = 15
b = 10
retorno = 10

MAIN

Abstraindo os Conceitos

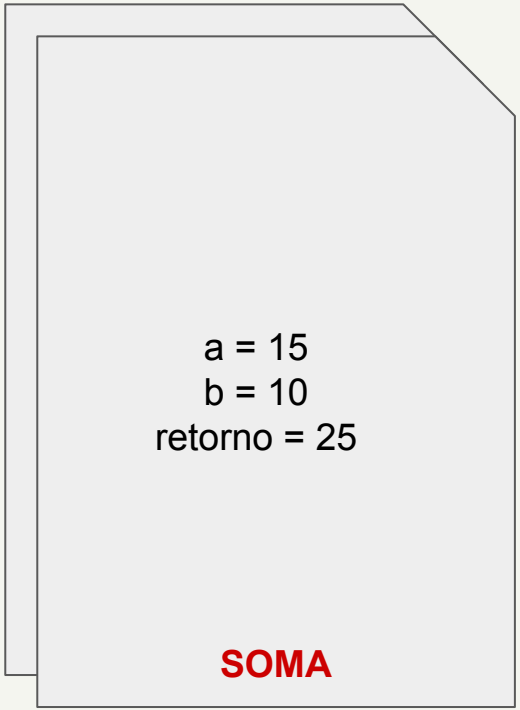
```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```



Abstraindo os Conceitos

```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```

Realiza a soma e insere o resultado no reg. de retorno.



Abstraindo os Conceitos

```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```

Retorna!

Com o retorno, esta folha é removida!

a = 15
b = 10
retorno = 25

SOMA

Abstraindo os Conceitos

```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```

Move o valor de retorno para %rdi e retorna!

Com o retorno, esta folha é removida!

a = 15
b = 10
retorno = 25

MAIN

Abstraindo os Conceitos

```
1)   int a = 10;
2)   int b = 5;
3)
4)   int soma() {
5)     return (a + b);
6)   }
7)   int subtrai() {
8)     return (a - b);
9)   }
10)
11)  int main (){
12)    a = soma();
13)    b = subtrai();
14)    return soma();
15)  }
```

Note que, mais para frente, vamos estudar **parâmetros e variáveis** locais.

Esses dados, diferentes das variáveis globais e dos registradores, **irão existir somente enquanto a "folha" existir!**

Implementação da Pilha

Mas o que são essas “folhas” e o que elas armazenam realmente?

A primeira coisa que precisamos compreender é que a chamada de procedimento envolve um desvio no fluxo de execução do programa. Então, precisamos:

- Armazenar o endereço de retorno da chamada;
- Executar um desvio incondicional para o início do procedimento;
- Restaurar o endereço armazenado no final do procedimento.

Implementação da Pilha

Note que, para existirem chamadas aninhadas de procedimento, existe uma estrutura de pilha relacionada. **Veja que o último procedimento chamado é o primeiro a ser concluído.**

As “folhas” que são armazenadas nessa estrutura de pilha, na prática, são chamadas de **REGISTROS DE ATIVAÇÃO**, que armazenam:

- Endereço de retorno
- Parâmetros
- Variáveis locais
- Encadeamento de folhas

Implementação da Pilha

VALE RESSALTAR!

Diferentes tipos de arquitetura, têm diferentes formatos de execução de procedimentos.

- Registradores usados para a manipular a pilha
- Acesso a variáveis locais
- Leitura de argumentos (manipulação dos parâmetros)
- etc...

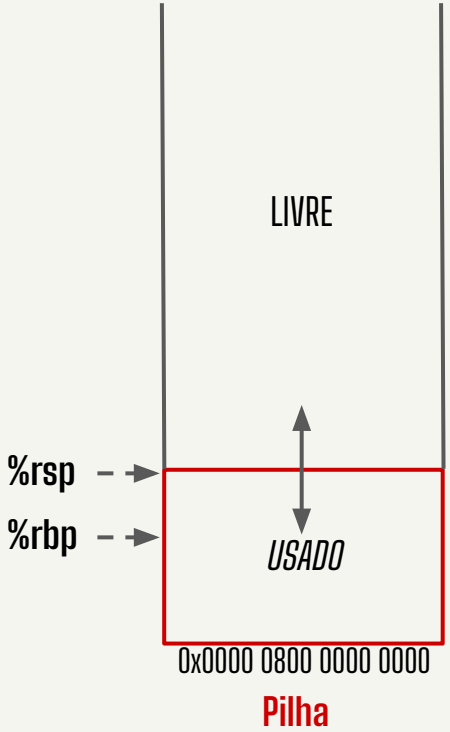
Implementação da Pilha

Nós vamos considerar o documento **ABI64**, que indica o modelo de implementação de chamadas de procedimentos em **Linux com arquitetura AMD64**. Neste modelo:

- %rsp: registrador (exclusivo) que armazena o topo da pilha
  - Isto é, o último endereço usado na pilha
- %rbp: registrador (exclusivo) que aponta o registro de ativação corrente na pilha
  - Do procedimento executado em um dado momento

A ABI64 é usada por todas as aplicações GNU!

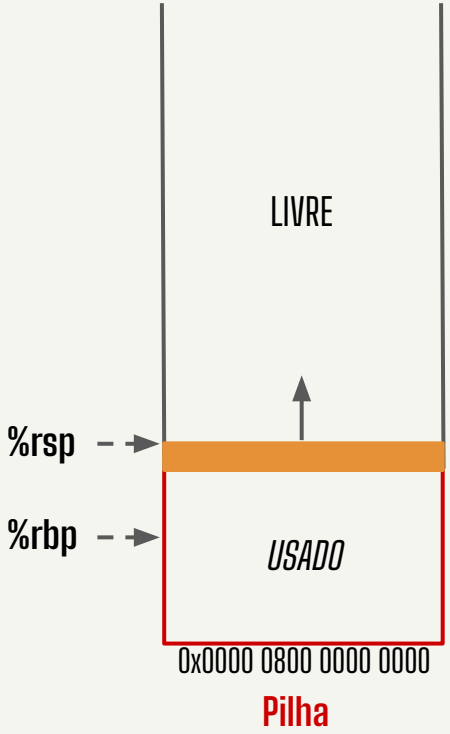
Implementação da Pilha



Também vamos precisar de novas instruções para chamarmos procedimentos e manipularmos a pilha:

- *push*: empilha um dado (manipula %rsp)
- *pop*: desempilha um dado (manipula %rsp)
- *call*: chama um procedimento (manipula %rsp, %rbi e %rip)
- *ret*: retorna de um procedimento (manipula %rsp, %rbi e %rip)

Instrução *Push*



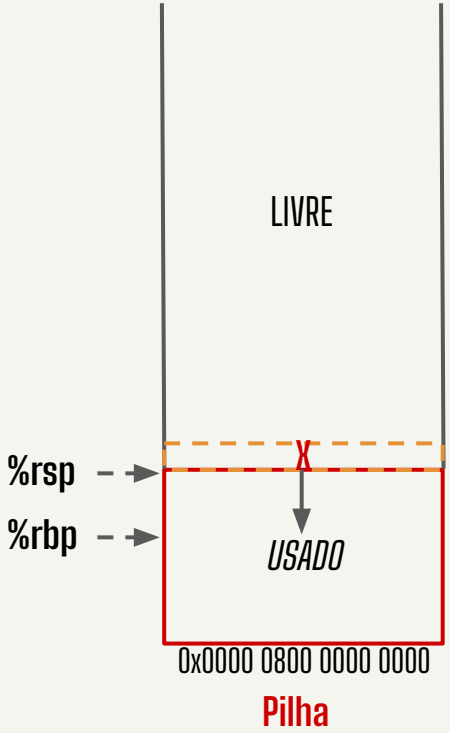
A instrução *push* adiciona um dado na pilha. Ela recebe como parâmetro este dado, que pode ser uma constante, um valor em registrador ou em memória:

```
pushq parâmetro
```

Essa instrução equivale (funcionalmente) a:

```
subq $8, %rsp
movq parâmetro, (%rsp)
```

Instrução *Pop*



A instrução *pop* remove um dado na pilha. Ela recebe como parâmetro o local onde deve ser armazenado o valor removido, sendo esse local um registrador ou posição de memória:

```
popq parâmetro
```

Essa instrução equivale (funcionalmente) a:

```
movq (%rsp), parâmetro
addq $8, %rsp
```

Push/*Pop* na Prática

```
1)    .section .data
2)    A: .quad 5
3)    .section .text
4)    .global _start
5)    _start:
6)    movq $10, %rax
7)    pushq $0
8)    pushq $A
9)    pushq A
10)   pushq %rax
11)   popq A
12)   popq %rax
```

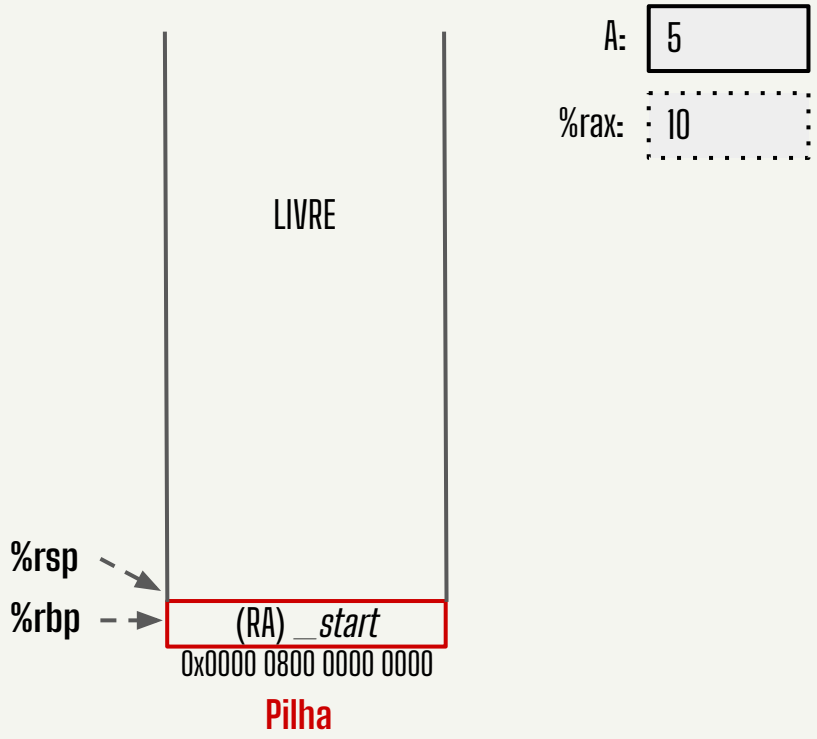
Observe o código ao lado. Nesse código, fazemos uso de várias instruções ***push*** e ***pop***.

Vamos considerar a execução do programa até a linha quatro (4):

- Qual é o valor em A?
- Qual é o valor em %rax?

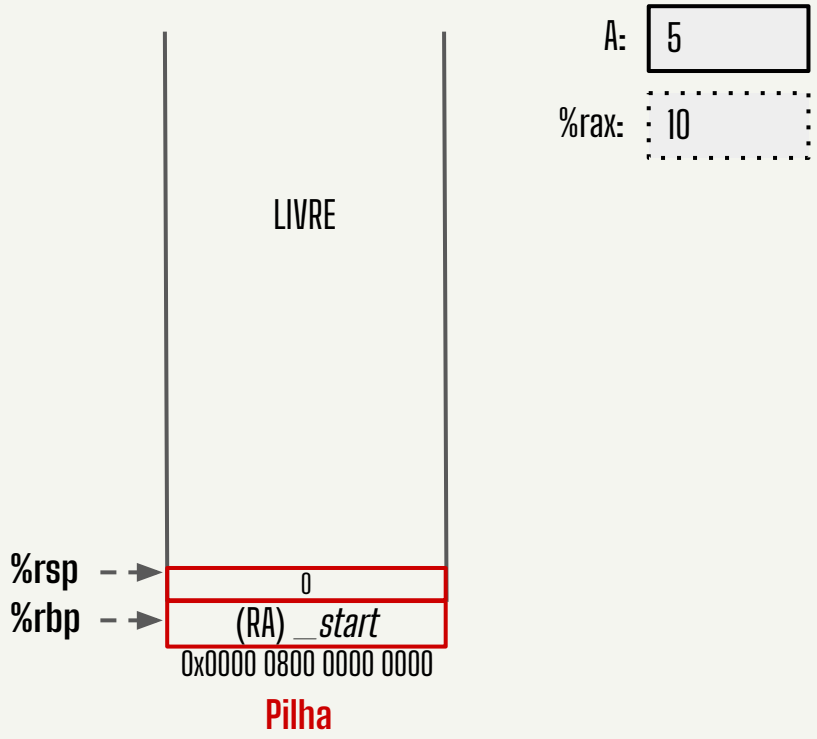
Push/ Pop na Prática

```
1)    .section .data
2)    A: .quad 5
3)    .section .text
4)    .global _start
5)    _start:
6)    movq $10, %rax
7)    pushq $0
8)    pushq $A
9)    pushq A
10)   pushq %rax
11)   popq A
12)   popq %rax
                    ...
```



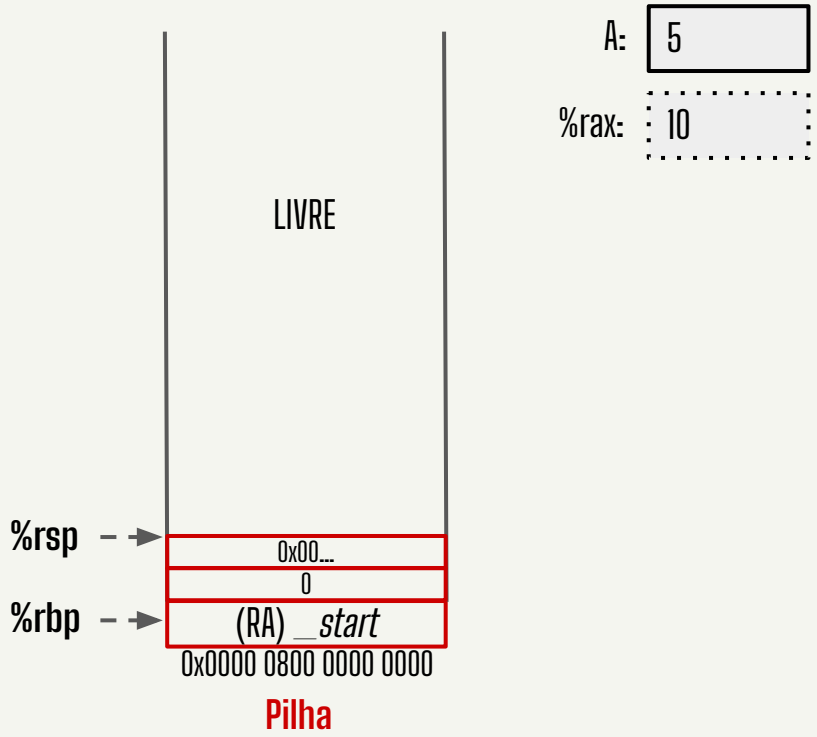
Push/ Pop na Prática

```
1)    .section .data
2)    A: .quad 5
3)    .section .text
4)    .global _start
5)    _start:
6)    movq $10, %rax
7)    pushq $0
8)    pushq $A
9)    pushq A
10)   pushq %rax
11)   popq A
12)   popq %rax
                    ...
```



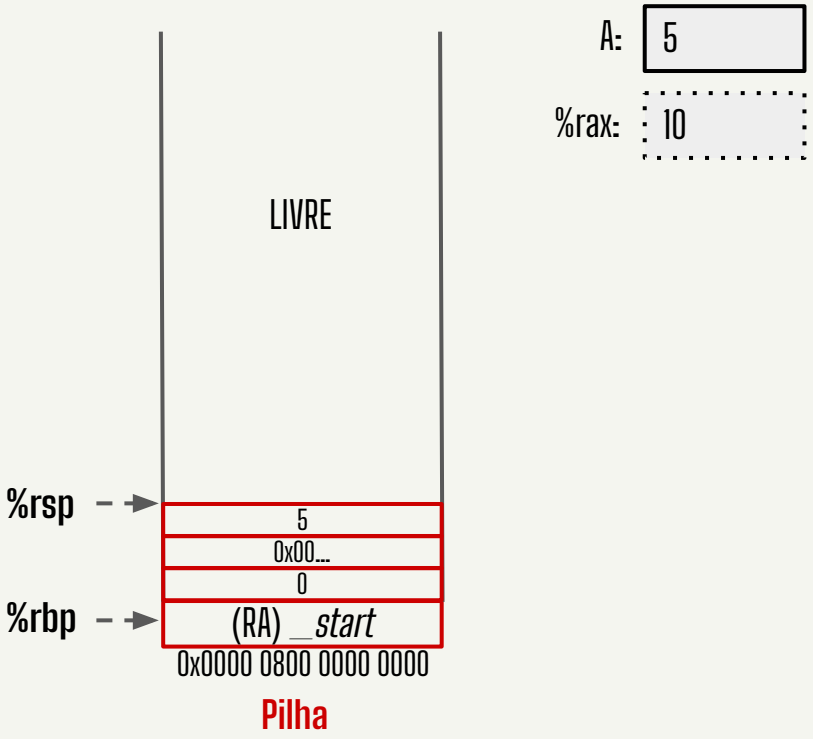
Push/ Pop na Prática

```
1)    .section .data
2)    A: .quad 5
3)    .section .text
4)    .global _start
5)    _start:
6)    movq $10, %rax
7)    pushq $0
8)    pushq $A
9)    pushq A
10)   pushq %rax
11)   popq A
12)   popq %rax
                    ...
```



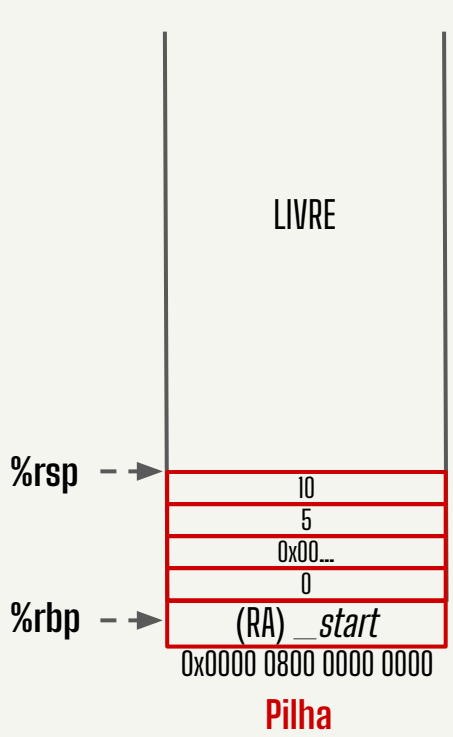
Push/ Pop na Prática

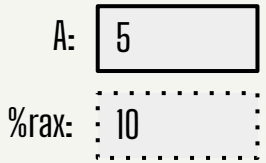
```
1)    .section .data
2)    A: .quad 5
3)    .section .text
4)    .global _start
5)    _start:
6)    movq $10, %rax
7)    pushq $0
8)    pushq $A
9)    pushq A
10)   pushq %rax
11)   popq A
12)   popq %rax
                    ...
```



Push/ Pop na Prática

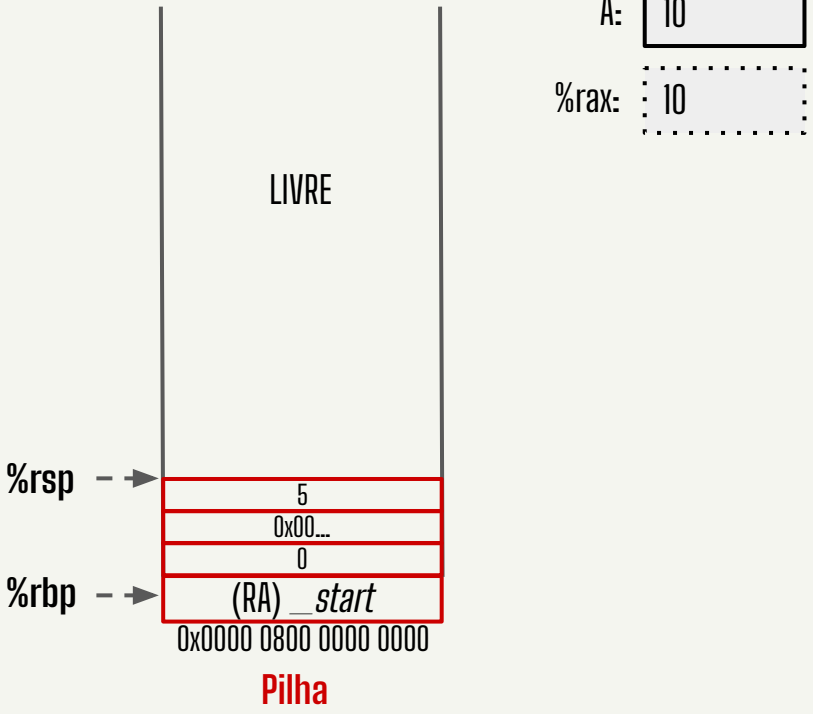
```
1)    .section .data
2)    A: .quad 5
3)    .section .text
4)    .global _start
5)    _start:
6)    movq $10, %rax
7)    pushq $0
8)    pushq $A
9)    pushq A
10)   pushq %rax
11)   popq A
12)   popq %rax
                    ...
```





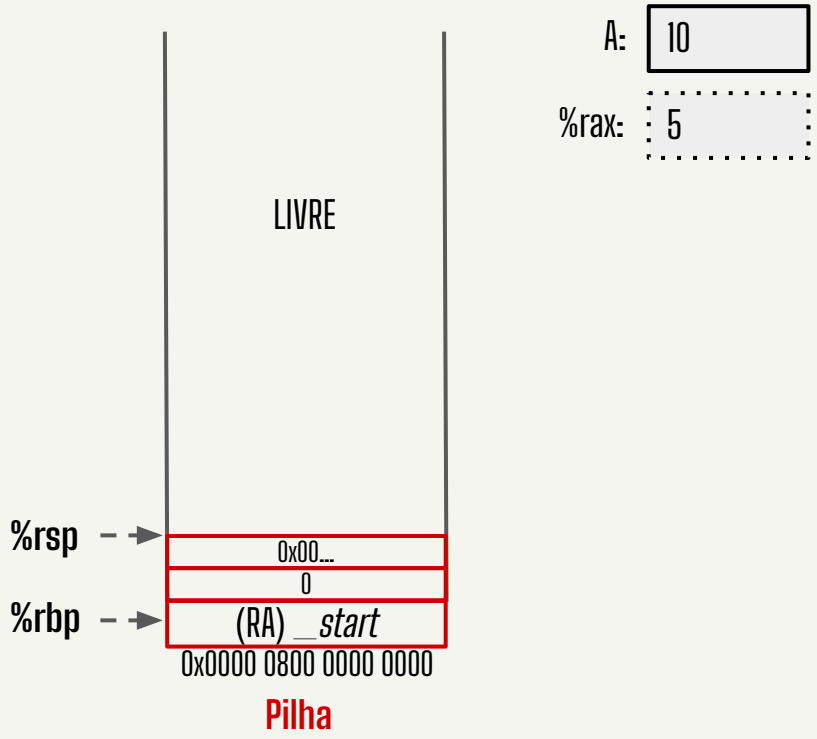
Push/Pop na Prática

```
1)    .section .data
2)    A: .quad 5
3)    .section .text
4)    .global _start
5)    _start:
6)    movq $10, %rax
7)    pushq $0
8)    pushq $A
9)    pushq A
10)   pushq %rax
11)   popq A
12)   popq %rax
                ...
```



Push/ Pop na Prática

```
1)    .section .data
2)    A: .quad 5
3)    .section .text
4)    .global _start
5)    _start:
6)    movq $10, %rax
7)    pushq $0
8)    pushq $A
9)    pushq A
10)   pushq %rax
11)   popq A
12)   popq %rax
                                  ...
```

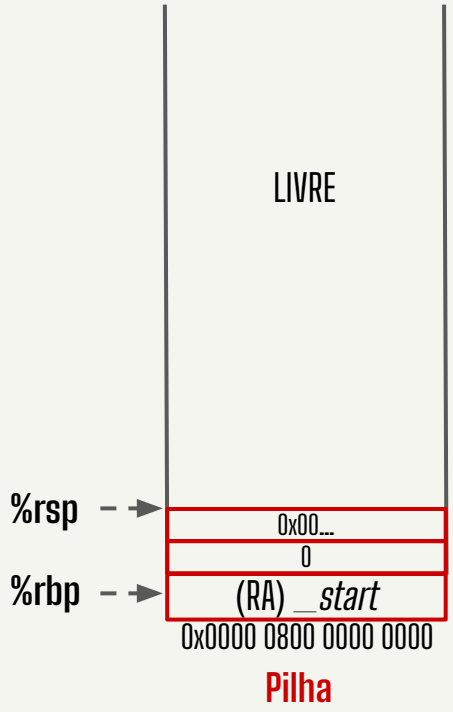


Push/ Pop na Prática

```
1)    .section .data
2)    A: .quad 5
3)    .section .text
4)    .global _start
5)    _start:
6)    movq $10, %rax
7)    pushq $0
8)    pushq $A
9)    pushq A
10)   pushq %rax
11)   popq A
12)   popq %rax
13)   popq $A
```

Qual é o resultado?

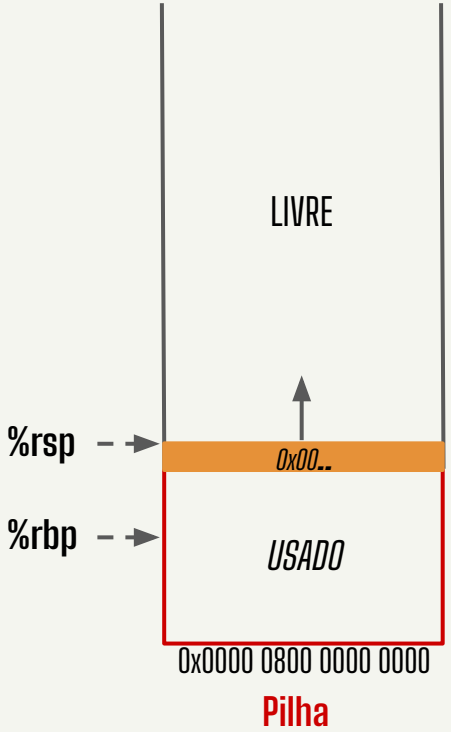
...



A: 10

%rax: 5

Instrução *Call*



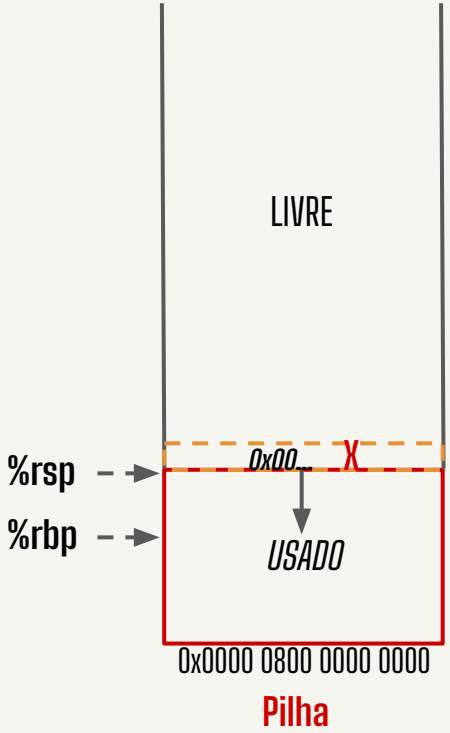
A instrução *call* adiciona o valor atual de %rip na pilha e realiza um desvio incondicional para a posição de memória (ou rótulo) recebido por parâmetro.

```
call parâmetro
```

Essa instrução equivale (funcionalmente) a:

```
pushq %rip
jmp parâmetro
```

Instrução *Ret*



A instrução *ret* adiciona o valor no topo da pilha no registrador %rip. Essa instrução não define nenhum parâmetro.

```
ret
```

Essa instrução equivale (funcionalmente) a:

```
popq %rip
```

Implementando Procedimentos

Conhecendo as instruções call e ret, a pergunta que surge é:
COMO IMPLEMENTAR PROCEDIMENTOS A PARTIR DESSAS INSTRUÇÕES?

Existem algumas coisas que precisamos saber antes de colocar procedimentos em prática:

- O procedimento é identificado por um rótulo
- Ao iniciar o procedimento, é necessário colocar o %rbp na pilha
  - Além de mover o valor de %rsp para %rbp
- Antes de realizar o retorno, o valor empilha é recolocado em %rbp

Implementando Procedimentos

E o valor de retorno de um procedimento? Como fazer?

Por padrão, utilizamos o registrador **%rax** para retornar valores. Portanto, devemos mover o retorno para esse registrador antes de executarmos a instrução *ret*.

Implementando Procedimentos

```
1)    int a = 10;
2)    int b = 5;
3)    int soma() {
4)      return (a + b);
5)    }
6)    int main (){
7)      return soma();
8)    }
```

```
1)    .section .data
2)    A: .quad 10
3)    B: .quad 5
```

Implementando Procedimentos

```
1)    int a = 10;
2)    int b = 5;
3)    int soma() {
4)      return (a + b);
5)    }
6)    int main (){
7)      return soma();
8)    }
```

```
1)    .section .data
2)    A: .quad 10
3)    B: .quad 5
4)    .section .text
5)    .global _start
```

Implementando Procedimentos

```
1)   int a = 10;
2)   int b = 5;
3)   int soma() {
4)     return (a + b);
5)   }
6)   int main (){
7)     return soma();
8)   }
```

```
1)   .section .data
2)   A: .quad 10
3)   B: .quad 5
4)   .section .text
5)   .global _start
6)   soma:
7)   pushq %rbp
8)   movq %rsp, %rbp
```

Implementando Procedimentos

```
1)   int a = 10;
2)   int b = 5;
3)   int soma() {
4)     return (a + b);
5)   }
6)   int main (){
7)     return soma();
8)   }
```

```
1)   .section .data
2)   A: .quad 10
3)   B: .quad 5
4)   .section .text
5)   .global _start
6)   soma:
7)   pushq %rbp
8)   movq %rsp, %rbp
9)   movq A, %rax
10)  addq B, %rax
```

Implementando Procedimentos

```
1)   int a = 10;
2)   int b = 5;
3)   int soma() {
4)     return (a + b);
5)   }
6)   int main (){
7)     return soma();
8)   }
```

```
1)   .section .data
2)   A: .quad 10
3)   B: .quad 5
4)   .section .text
5)   .global _start
6)   soma:
7)   pushq %rbp
8)   movq %rsp, %rbp
9)   movq A, %rax
10)  addq B, %rax
11)  popq %rbp
12)  ret
```

Implementando Procedimentos

```
1)    int a = 10;
2)    int b = 5;
3)    int soma() {
4)      return (a + b);
5)    }
6)    int main (){
7)      return soma();
8)    }
```

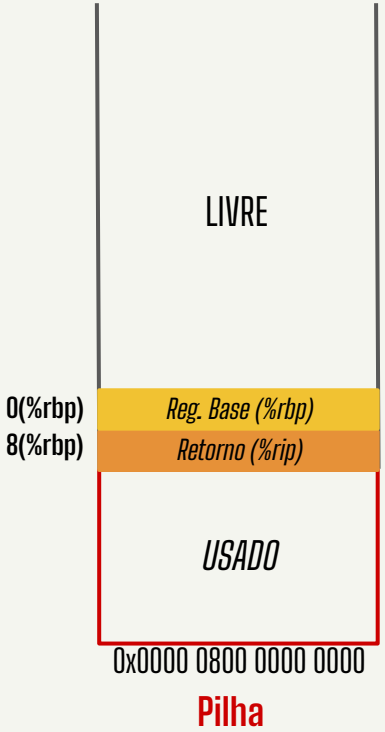
```
1)    .section .data
2)    A: .quad 10
3)    B: .quad 5
4)    .section .text
5)    .global _start
6)    soma:
7)    pushq %rbp
8)    movq %rsp, %rbp
9)    movq A, %rax
10)   addq B, %rax
11)   popq %rbp
12)   ret
13)   _start:
```

Implementando Procedimentos

```
1)   int a = 10;
2)   int b = 5;
3)   int soma() {
4)     return (a + b);
5)   }
6)   int main (){
7)     return soma();
8)   }
```

```
1)   .section .data
2)   A: .quad 10
3)   B: .quad 5
4)   .section .text
5)   .global _start
6)   soma:
7)   pushq %rbp
8)   movq %rsp, %rbp
9)   movq A, %rax
10)  addq B, %rax
11)  popq %rbp
12)  ret
13)  _start:
14)  call soma
```

Registro de Ativação



O registro de ativação é construído em dois momentos:

- Na execução da chamada do procedimento (gravação do endereço de retorno)
- Na gravação do registrador base no início do procedimento

Os parâmetros (argumentos) e variáveis locais também serão acessados a partir de %rbp. Ou seja, nas próximas aulas ele terá maior utilidade.

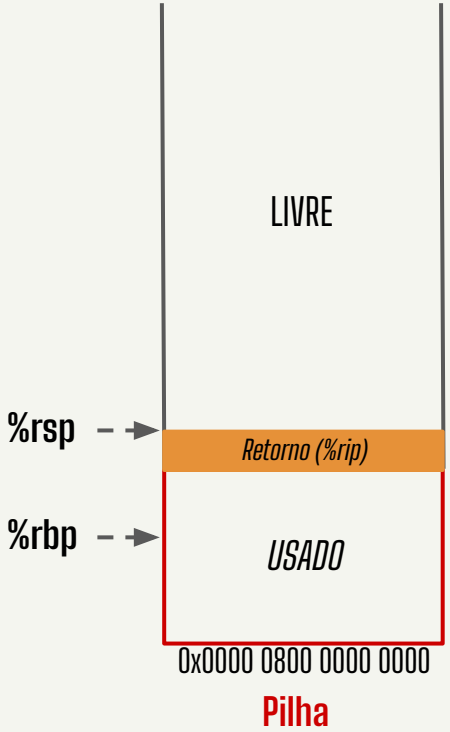
Executando o Procedimento

Mas vamos **entender um pouco mais a parte nova!**

O que acontece na pilha quando chamamos um procedimento?

```
1)   .section .data
2)   A: .quad 10
3)   B: .quad 5
4)   .section .text
5)   .global _start
6)   soma:
7)   pushq %rbp
8)   movq %rsp, %rbp
9)   movq A, %rax
10)  addq B, %rax
11)  popq %rbp
12)  ret
13)  _start:
14)  call soma
15)  movq %rax, %rdi
16)  movq %rax, %rdi
17)  syscall
```

Executando o Procedimento

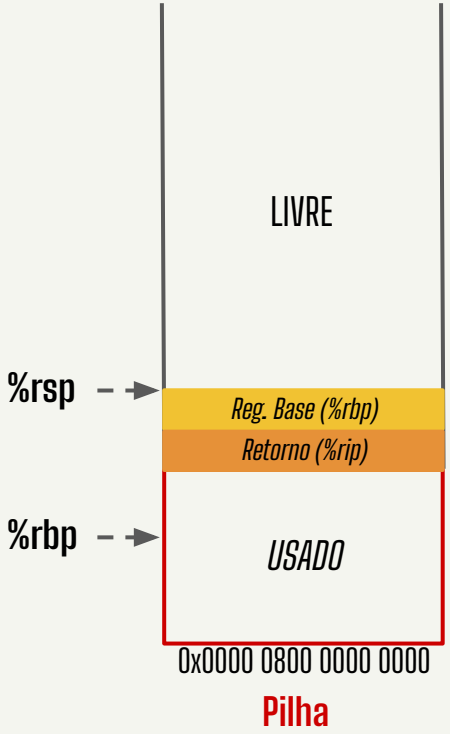


A instrução *call* é executada!

```
call soma
```

- O registrador %rsp é atualizado, “liberando” um espaço de memória de 64 bits;
- O conteúdo atual de %rip é armazenado no topo da pilha (nesses 64 bits “liberados”);
- Um desvio incondicional para o procedimento soma acontece.

Executando o Procedimento

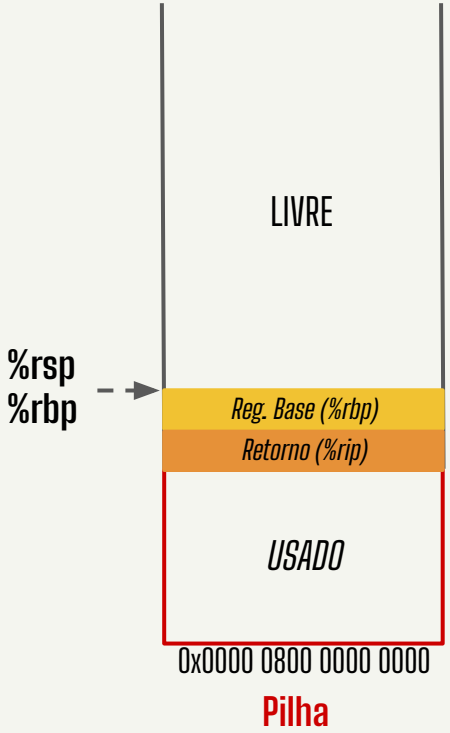


O registrador base é salvo na pilha!

```
pushq %rbp
```

- O registrador base atual (%rbp) é salvo na pilha através de uma instrução *push*.

Executando o Procedimento

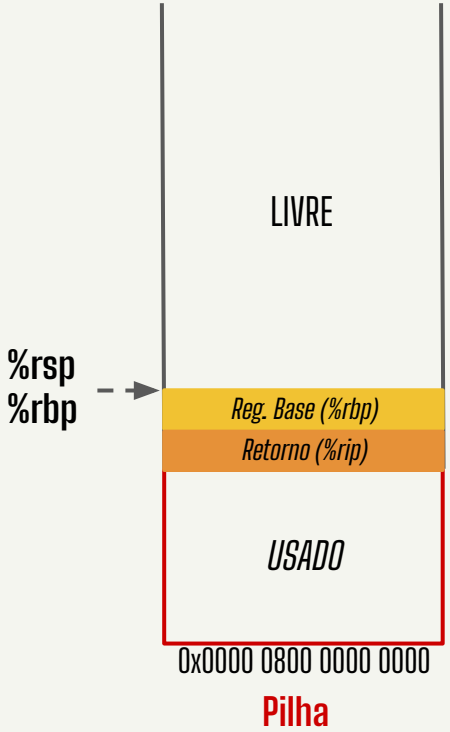


O registrador base é atualizado!

```
movq %rsp, %rbp
```

- O registrador base atual é atualizado para o topo da pilha atual (onde foi salvo o valor anterior de %rbp).

Executando o Procedimento

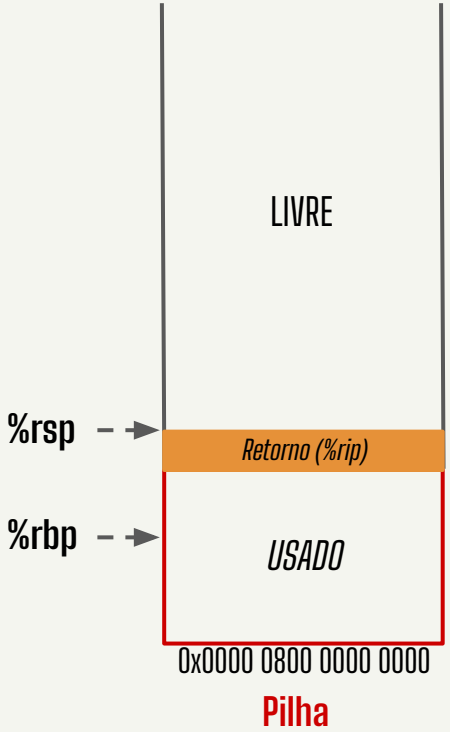


As operações são realizadas!

```
movq A, %rax
addq B, %rax
```

- Não há modificações na pilha neste momento;
- Porém, note que as operações já são realizadas com os resultados salvos no registrador de retorno (%rax).

Executando o Procedimento

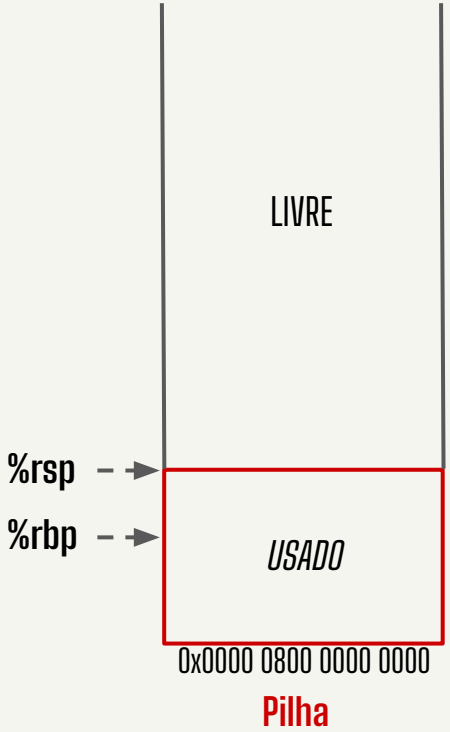


O registrador base é atualizado para retorno!

```
popq %rbp
```

- Tendo as operações concluídas, o registrador base é atualizado para o valor salvo na pilha;
- Este é também removido da pilha através da execução da instrução *pop*.

Executando o Procedimento



As operações são realizadas!

`ret`

- O %rip é atualizado com o valor contido no topo da pilha;
- Também, como consequência da instrução *ret*, este valor é removido da pilha.

Exercício #05

```
1)    int a = 10;
2)    int b = 5;
3)    int soma() {
4)      return (a + b);
5)    }
6)    int subtrai() {
7)      return (a - b);
8)    }
9)    int main (){
10)     a = soma();
11)     b = subtrai();
12)     return soma();
13)   }
```

Considere o algoritmo ao lado. Tal algoritmo foi usado como exemplo na primeira parte da aula e foi parcialmente traduzido na segunda parte da aula.

Conclua a tradução do algoritmo em C para código *assembly* AMD64.

Obrigado!

Vinícius Fülber Garcia
inf.ufpr.br/vinicius/
viniciusfulber@ufpr.br