PCS-3566 / PCS-3866
LINGUAGENS E COMPILADORES

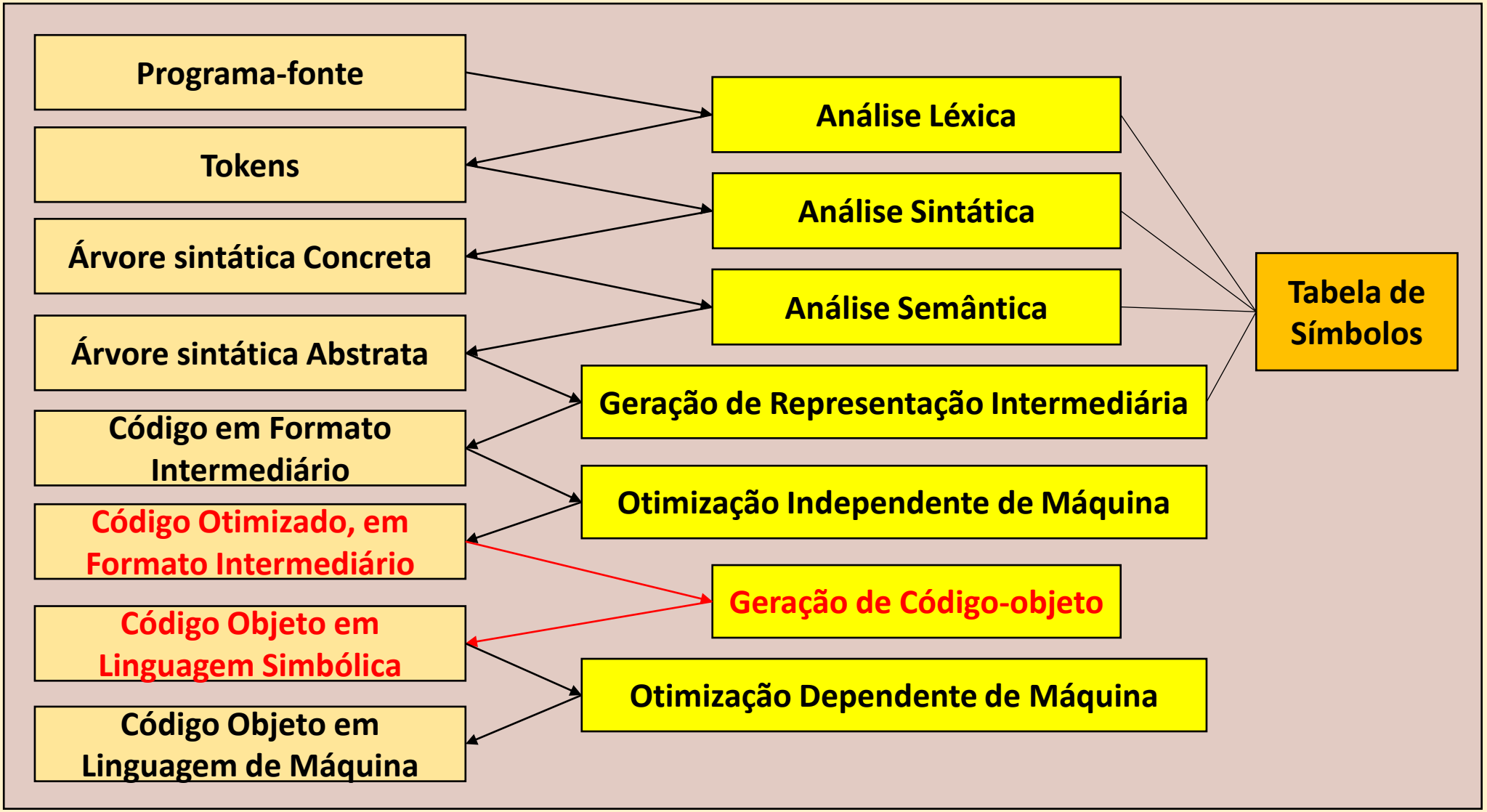
Prof. João José Neto

Aula 12 – **Ambientes de Execução**

Ambiente de Execução (*run-time environment*)

- É o ambiente computacional em que se executa o programa-objeto.
- Temas considerados nesta apresentação:
  - Alocações de memória
  - Acessos a variáveis e dados
  - Gerenciamento de memória:
    - alocação de pilha,
    - gerenciamento de *heap*,
    - *garbage collection*

Compilação: Fase de Geração de Código-objeto



Temas a serem estudados

- Organização da memória
- Códigos e dados binários
- Alocação de área para a pilha
- Dados não locais
- Gerenciamento de *Heap*
- Arquiteturas

Organização da Memória

Alocações Estática e Dinâmica de Memória

- **Estática** se refere à alocação feita em tempo de compilação
- **Dinâmica** se refere àquela que ocorre em tempo de execução
- A **pilha** contém dados locais às chamadas de procedimentos
  - O arranjo físico da memória é conhecido estaticamente
  - A pilha é gerenciada automaticamente pelo código gerado
- O ***heap*** contém dados que devem permanecer disponíveis entre uma chamada de procedimento e outra.
  - Não há como determinar a priori o arranjo físico do *heap*
  - Esse tipo de memória deve portanto ser dinamicamente gerenciada por um *garbage collector*.

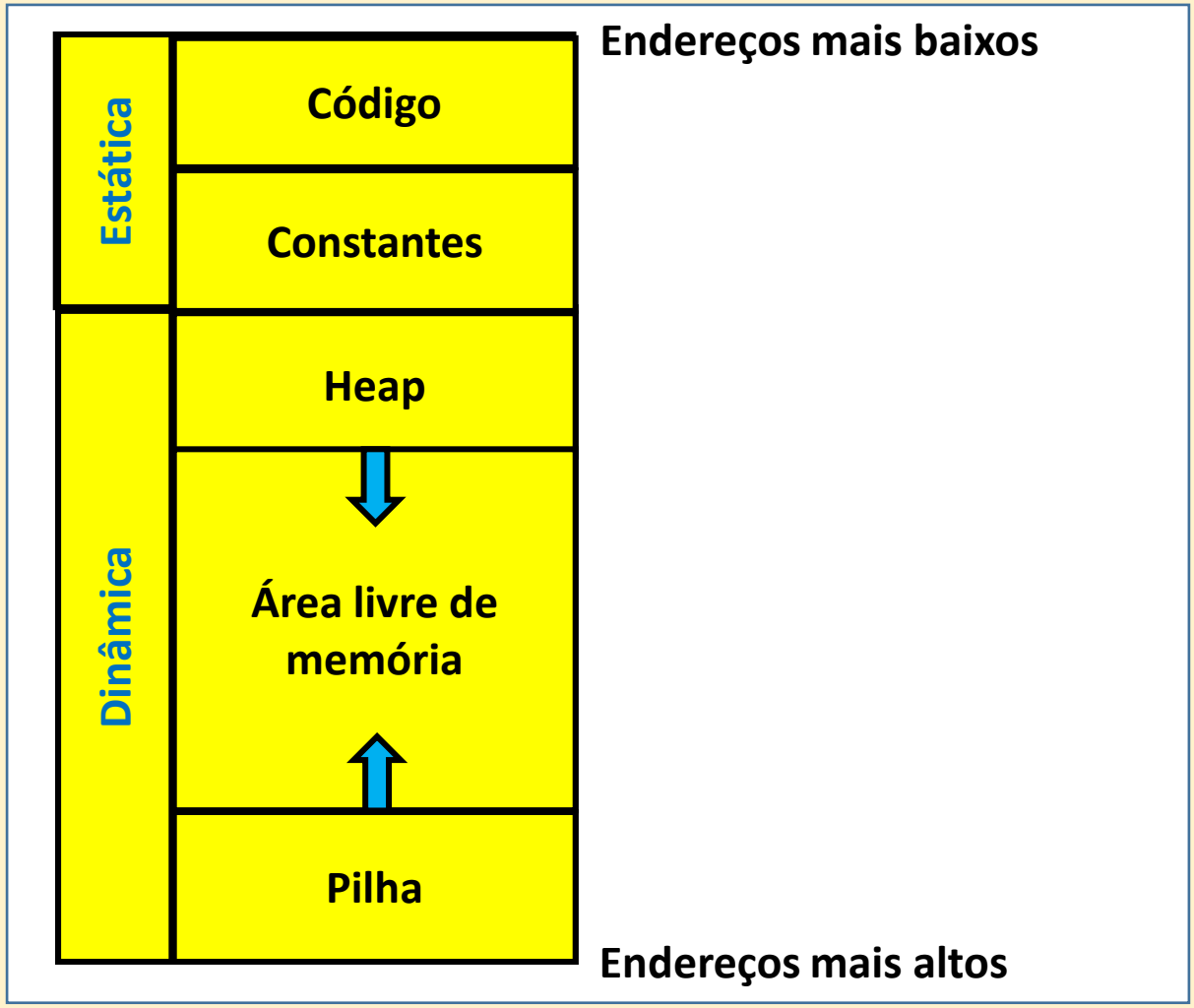
Garbage Collector

- Em ambientes com alocação dinâmica de uma memória que não seja organizada em pilha, a devolução ao sistema, em instantes arbitrários, das áreas de memória previamente alocadas, causa o fenômeno da fragmentação da memória
- Por isso, é preciso, de tempos em tempos, compactar a memória (defragmentá-la) para que ela possa ser utilizada mais eficientemente
- Propósito do *garbage collector*
  - Permitir que o ambiente de execução detecte elementos de dados inúteis, com a finalidade de reaproveitar a área de memória por eles ocupada.
  - Na maior parte das implementações de linguagens de programação modernas, esta operação costuma ser efetuada automaticamente pelo ambiente de execução da linguagem.

Organização da memória

- Representação de um programa objeto em tempo de execução:
  - **Área de código,** para instruções e ponteiros (endereços)
  - ***Heap***, onde são dinamicamente alocados:
    - Objetos, registros, vetores, matrizes, cadeias variáveis,.
  - **Pilha**, na qual são alocados estaticamente:
    - Registros de ativação e *stack frames*
    - Objetos, registros, vetores, matrizes e cadeias variáveis

Organização da memória

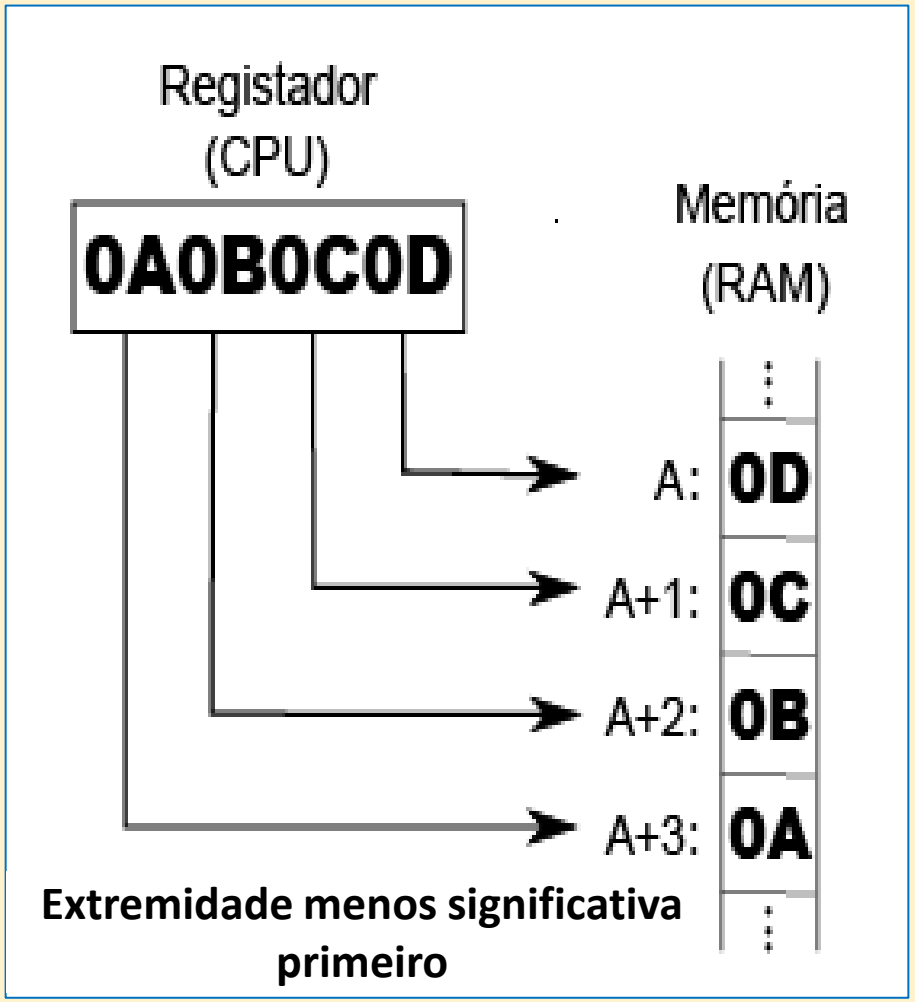


CÓDIGOS E DADOS BINÁRIOS

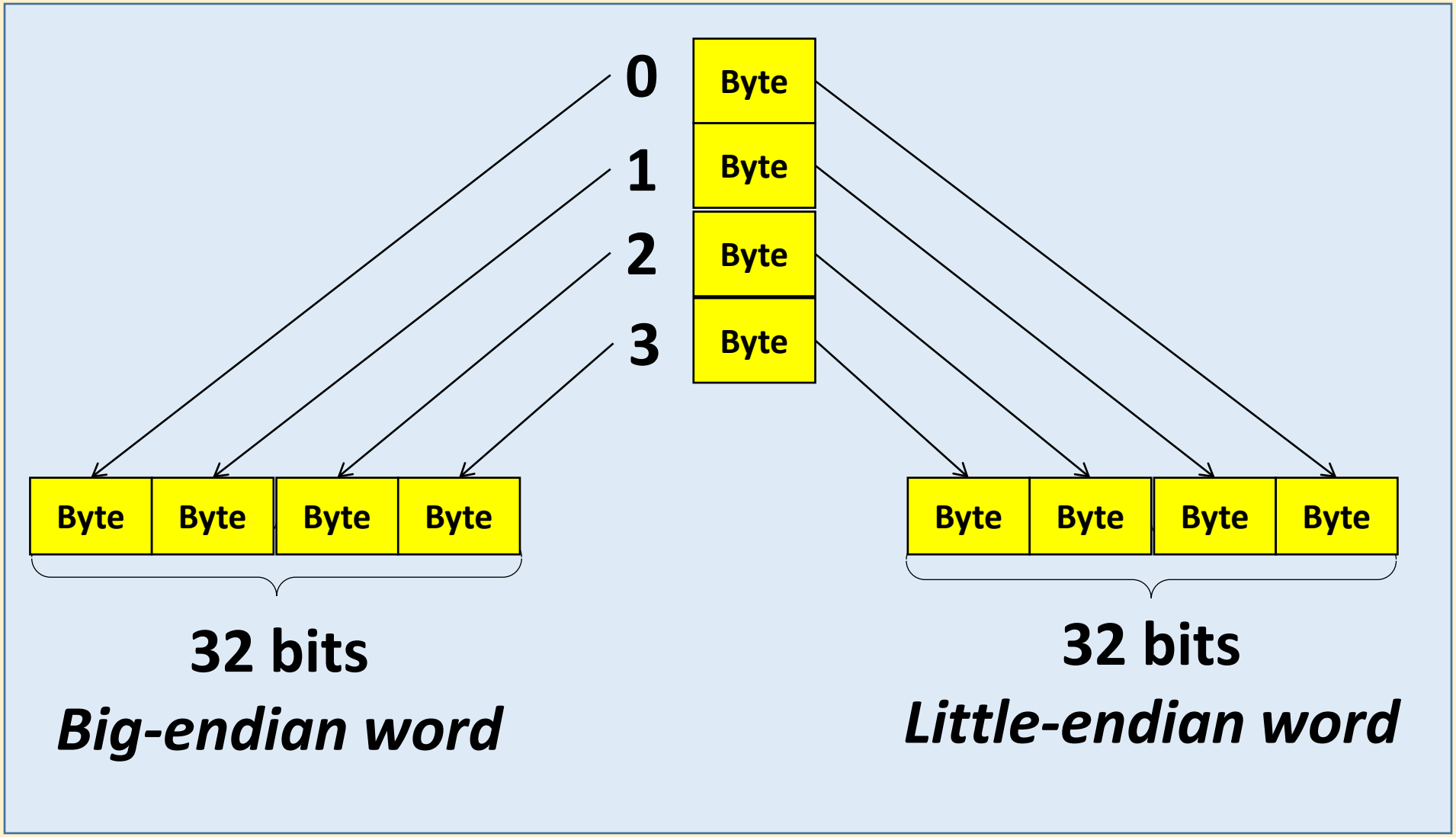
Endereços de memória

- A memória consiste de um banco de **bits**, contiguamente dispostos
- A memória se organiza em unidades formadas por grupos de **n bits**, cada qual associado a seu próprio endereço, que é único no sistema
- Computadores clássicos (até 1980) apresentavam n entre 1 e 80
- A linguagem de programação C consagrou o **byte** (n=8) como a única escolha realística
- Deseja-se em geral guardar números e instruções na memória
- Idealmente, existem, para cada byte, 2^8=256 combinações possíveis, mas deseja-se armazenar números pertencentes a domínios bem mais amplos
- Para isso, a **palavra** é um conceito mais adequado, pois constitui uma entidade formada por 4 bytes consecutivos, comportando $2^{8x4} = 2^{32}$ = 4294967296 combinações diferentes de bits.

Endianness


Registador
(CPU)
0A0B0C0D
Memória
(RAM)
A:
0D
A+1:
0C
A+2:
0B
A+3:
0A
Extremidade menos significativa primeiro

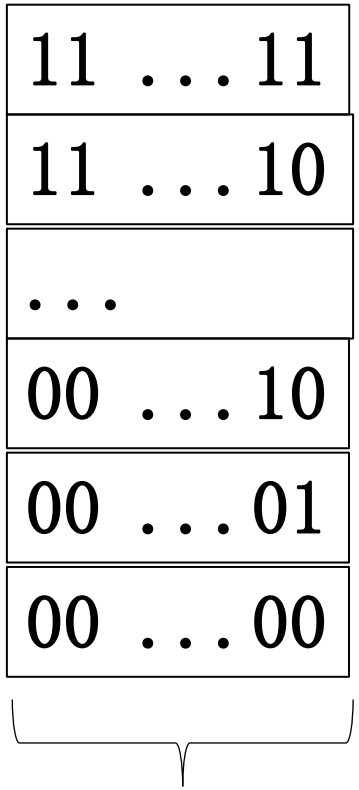
Endianness


0
1
2
3
Byte
Byte
Byte
Byte
Byte
Byte
Byte
Byte
Byte
Byte
Byte
Byte
32 bits
Big-endian word
32 bits
Little-endian word

Endianness

- ARM32 é um processador little-endian
  - O byte menos significativo fica na posição de memória de endereço mais baixo

Inteiros binários sem sinal, em representação de n bits



$1.2^{n-1} + 1.2^{n-2} + \quad + 1.2^{1} + 1.2^{0} = 2^{n}-1$

$1.2^{n-1} + 1.2^{n-2} + \quad + 1.2^{1} + 0.2^{0} = 2^{n}-2$

...

$0.2^{n-1} + 0.2^{n-2} + \quad + 1.2^{1} + 0.2^{0} = 2$

$0.2^{n-1} + 0.2^{n-2} + \quad + 0.2^{1} + 1.2^{0} = 1$

$0.2^{n-1} + 0.2^{n-2} + \quad + 0.2^{1} + 0.2^{0} = 0$

Adição, em complemento de dois

Consideremos todos os inteiros de 3 bits:

111
110
101
100
011
010
001
000

Adição, em complemento de dois

Consideremos todos os inteiros de 3 bits

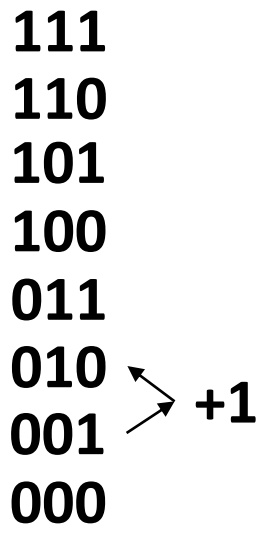
Consideremos agora a operação de somar uma unidade

111
110
101
100
011
010
001 ↘ **+1**
000 ↗

Adição, em complemento de dois

Consideremos todos os inteiros de 3 bits

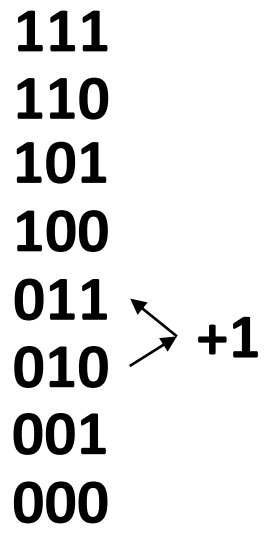
Consideremos a operação de somar mais uma unidade



Adição, em complemento de dois

Consideremos todos os inteiros de 3 bits

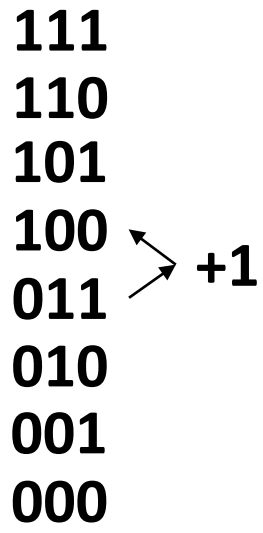
Consideremos a operação de somar outra unidade



Adição, em complemento de dois

Consideremos todos os inteiros de 3 bits

Consideremos a operação de somar mais outra unidade

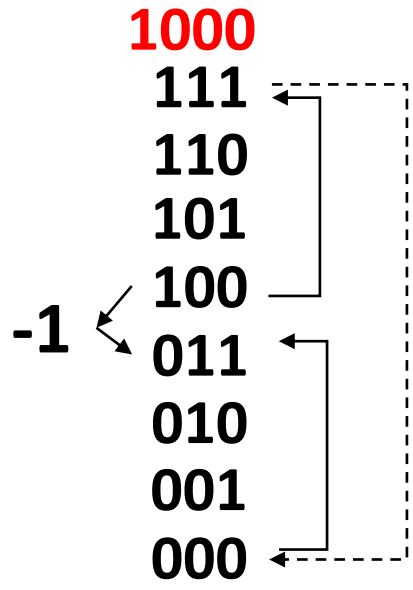


Adição, em complemento de dois

Consideremos todos os inteiros de 3 bits
Consideremos a operação de
somar mais outra unidade

Consideremos agora a operação de subtrair uma unidade

Podemos subtrair uma unidade somando 2^n-1



Adição, em complemento de dois

Consideremos todos os inteiros de 3 bits

Consideremos a operação de somar mais outra unidade

Consideremos agora a operação de subtrair uma unidade

Poderemos subtrair uma unidade somando 2^n-1

E considerando negativos todos os números maiores que 2^n-1

positivos	0111
	0110
	0101
	0100
	0011
	0010
	0001
	0000
negativos	1111
	1110
	1101
	1100
	1011
	1010
	1001
	1000

Inteiros com sinal, em complemento de dois

011 ...11	-0 .	$2^{n-1} + 1.2^{n-2} + ... + 1.2^1 + 1.2^0 = 2^{n-1}-1$
011 ...10	-0 .	$2^{n-1} + 1.2^{n-2} + ... + 1.2^1 + 0.2^0 = 2^{n-1}-2$
...		...
000 ...01	-0 .	$2^{n-1} + 0.2^{n-2} + ... + 0.2^1 + 1.2^0 = 1$
000 ...00	-0 .	$2^{n-1} + 0.2^{n-2} + ... + 0.2^1 + 0.2^0 = 0$
111 ...11	-1 .	$2^{n-1} + 1.2^{n-2} + ... + 1.2^1 + 1.2^0 = -1$
111 ...10	-1 .	$2^{n-1} + 1.2^{n-2} + ... + 1.2^1 + 0.2^0 = -2$
...		...
100 ...01	-1 .	$2^{n-1} + 0.2^{n-2} + ... + 0.2^1 + 1.2^0 = -2^{n-1}+1$
100 ...00	-1 .	$2^{n-1} + 0.2^{n-2} + ... + 0.2^1 + 0.2^0 = -2^{n-1}$

n bits

Alocação de Área na Pilha

Árvore de ativação

- Árvores de ativação são estruturas de dados que mostram (descrevem) todas as chamadas de procedimentos efetuadas em tempo de execução
- Cada chamada é representada por um nó-filho daquele nó que representa o programa chamador
- Nós-filhos vão sendo posicionados, na árvore de ativação, da esquerda para a direita, na mesma sequência em que forem ocorrendo as correspondentes chamadas em tempo de execução.

Quicksort – o programa e uma possível execução

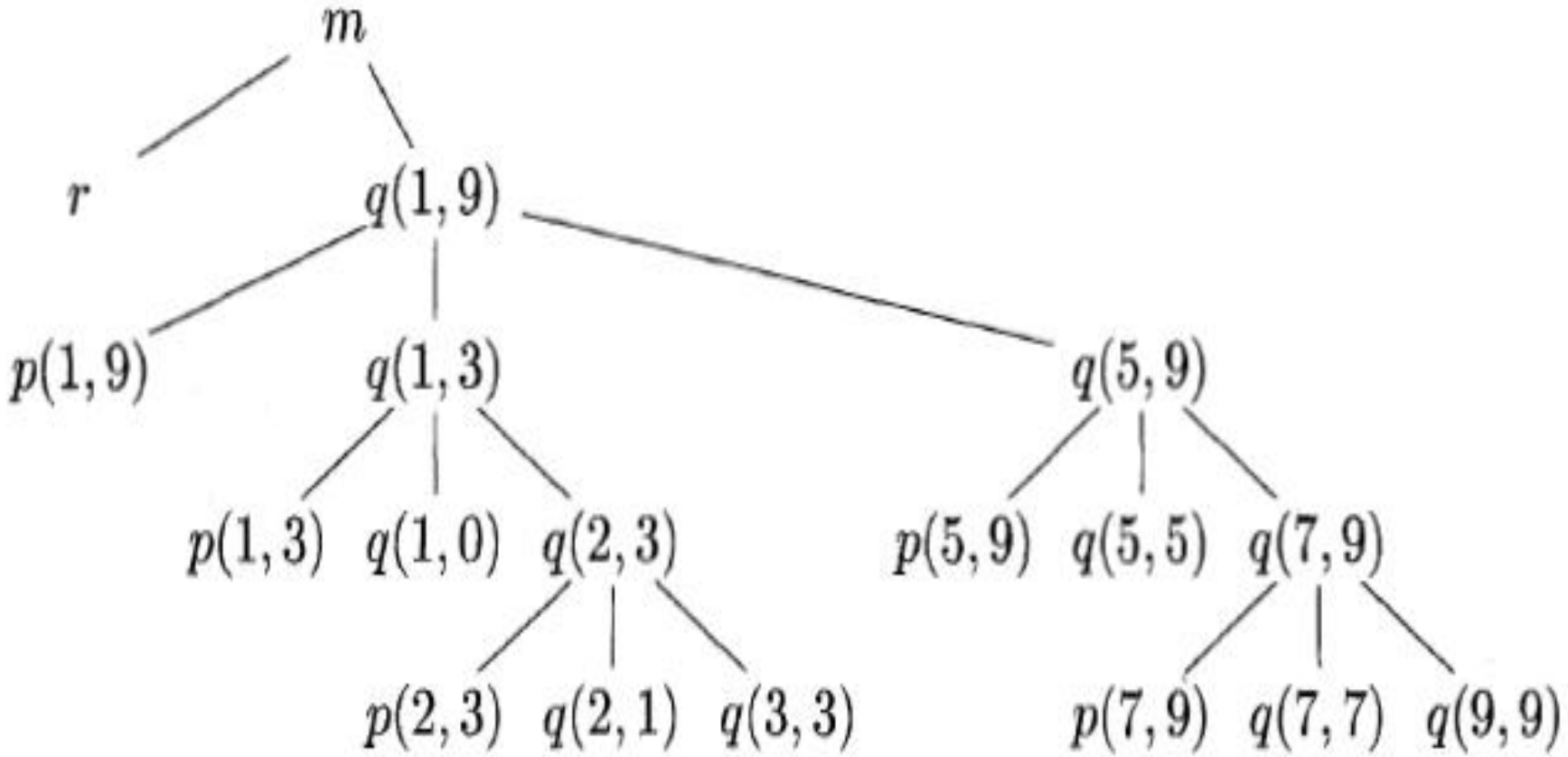
```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Esboço de um programa *quicksort*

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Ativações possíveis para o programa ao lado

Árvore de ativação para essa execução do Quicksort



Árvore de ativação representando as chamadas durante uma execução particular do algoritmo *quicksort*

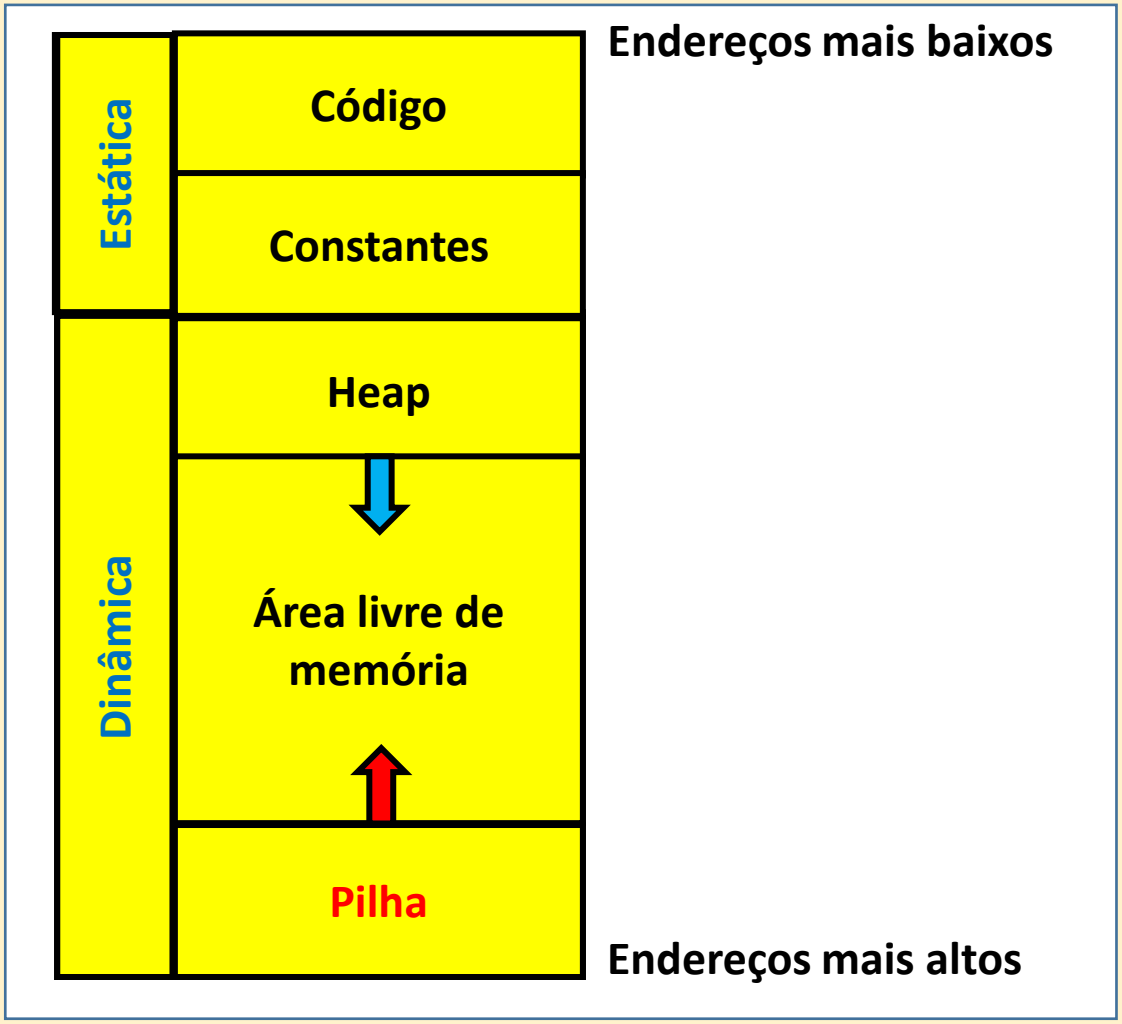
A Pilha de Controle

- **Chamadas e retornos de procedimento** são gerenciados tipicamente por estrutura de dados organizada como uma pilha (*last in, first out*), em tempo de execução.
- Esta estrutura de dados é chamada **pilha de controle** (*control stack*).

Registros de Ativação / *Frames* da Pilha

- **Cada chamada** na árvore de ativação corresponde à criação de **um registro de ativação** na pilha (de controle).
- Notar que **a sequência de chamadas** se associa em **pré-ordem** com o percurso da árvore de ativação.
- Por outro lado, **a ordem dos sucessivos empilhamentos e desempilhamentos** é dada pelo percurso da árvore de ativação em **pós-ordem** .

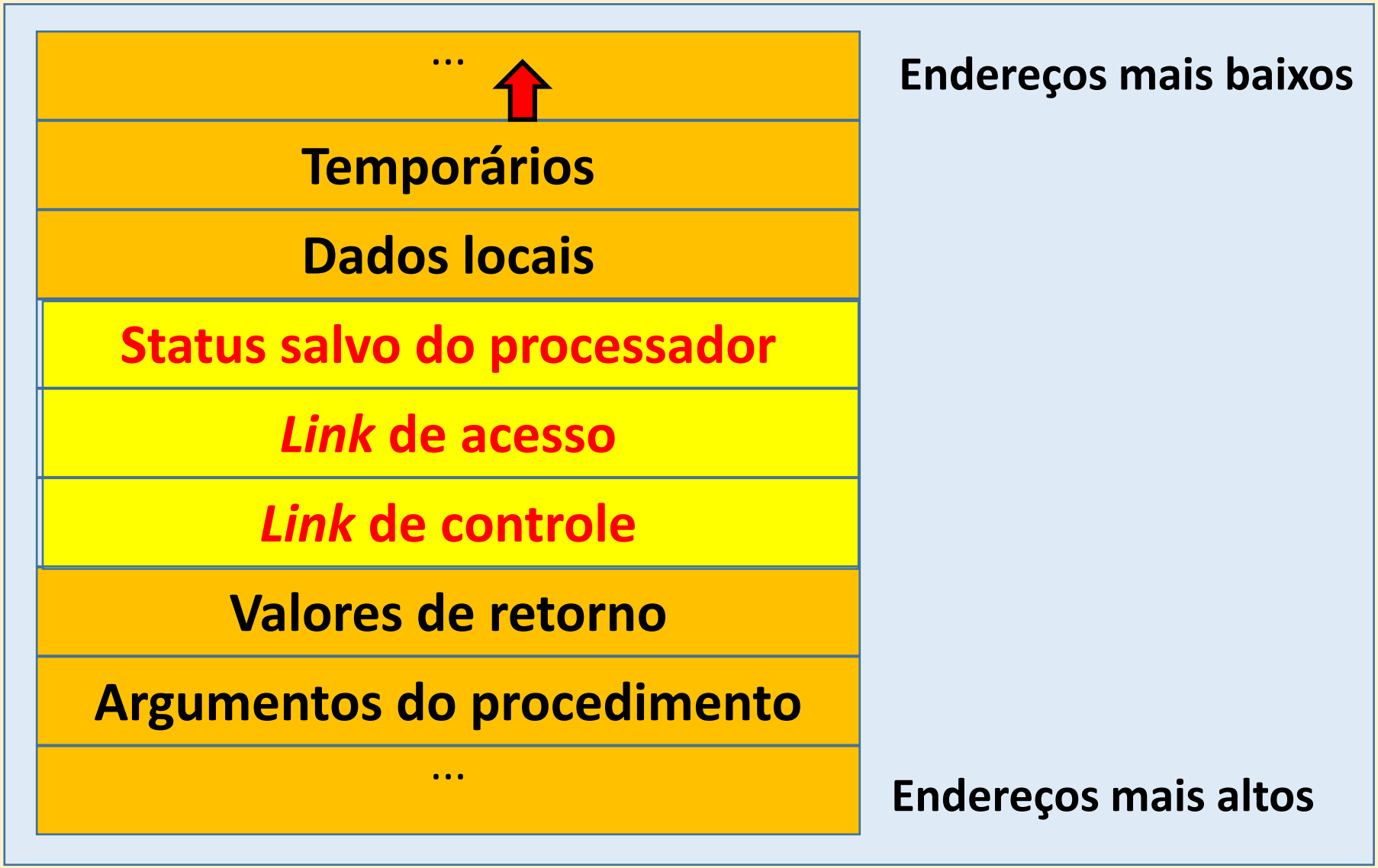
A Pilha de Controle



Conteúdo do Registro de Ativação

1. **Argumentos** passados ao procedimento (alternativamente, pré-colocados em registradores pelo chamador)
2. **Área para o valor de retorno** do procedimento
3. **Link de controle** (link dinâmico para o registro de ativação do programa chamador)
4. **Link de acesso** (link estático)
5. Cópia do **estado** da máquina (principalmente o endereço de retorno)
6. **Dados locais** do procedimento chamado
7. Valores **temporários** não compreendidos nos registradores

Registro de Ativação ou *Frame*



Um instantâneo da pilha de execução do Quicksort

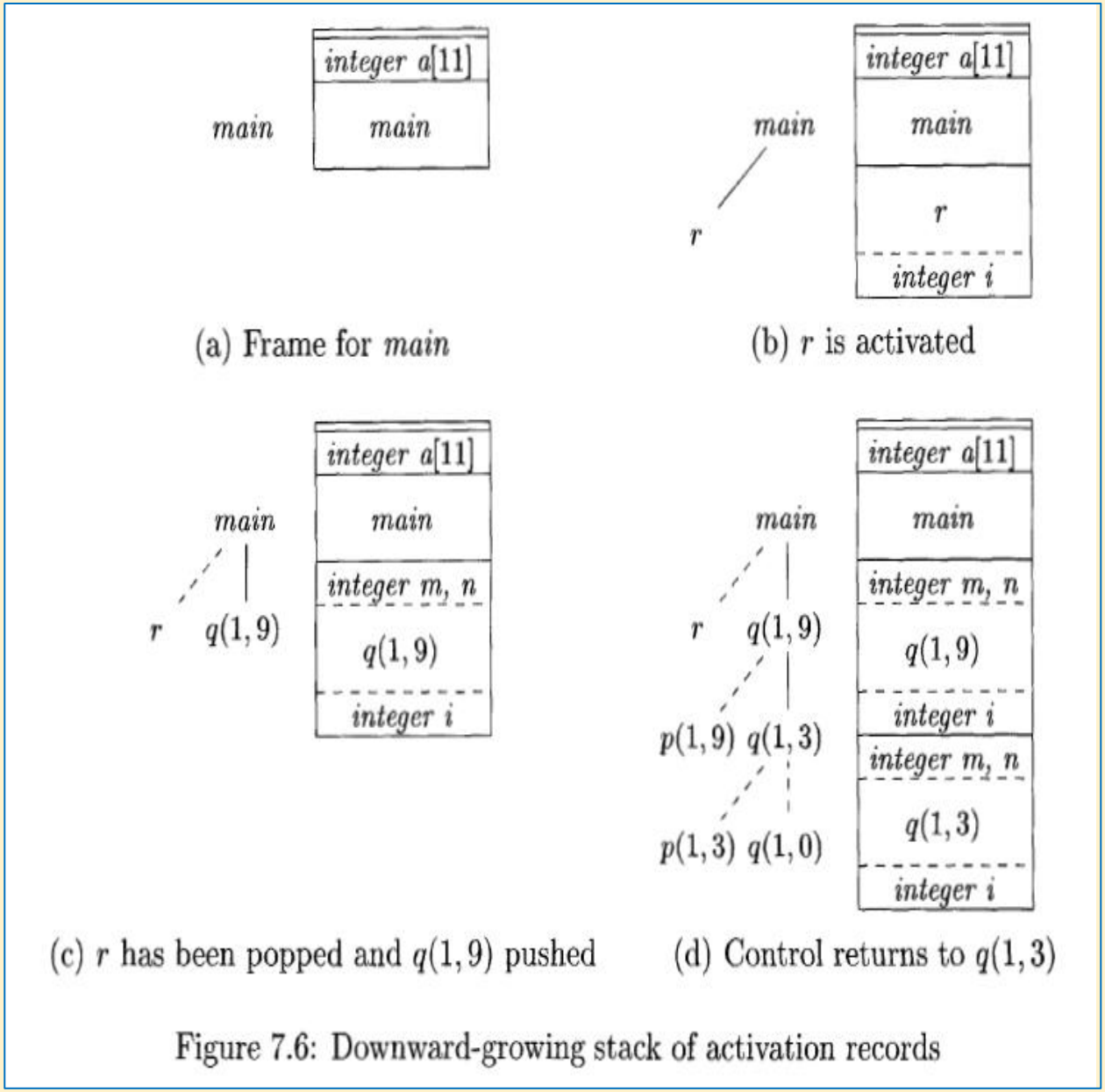
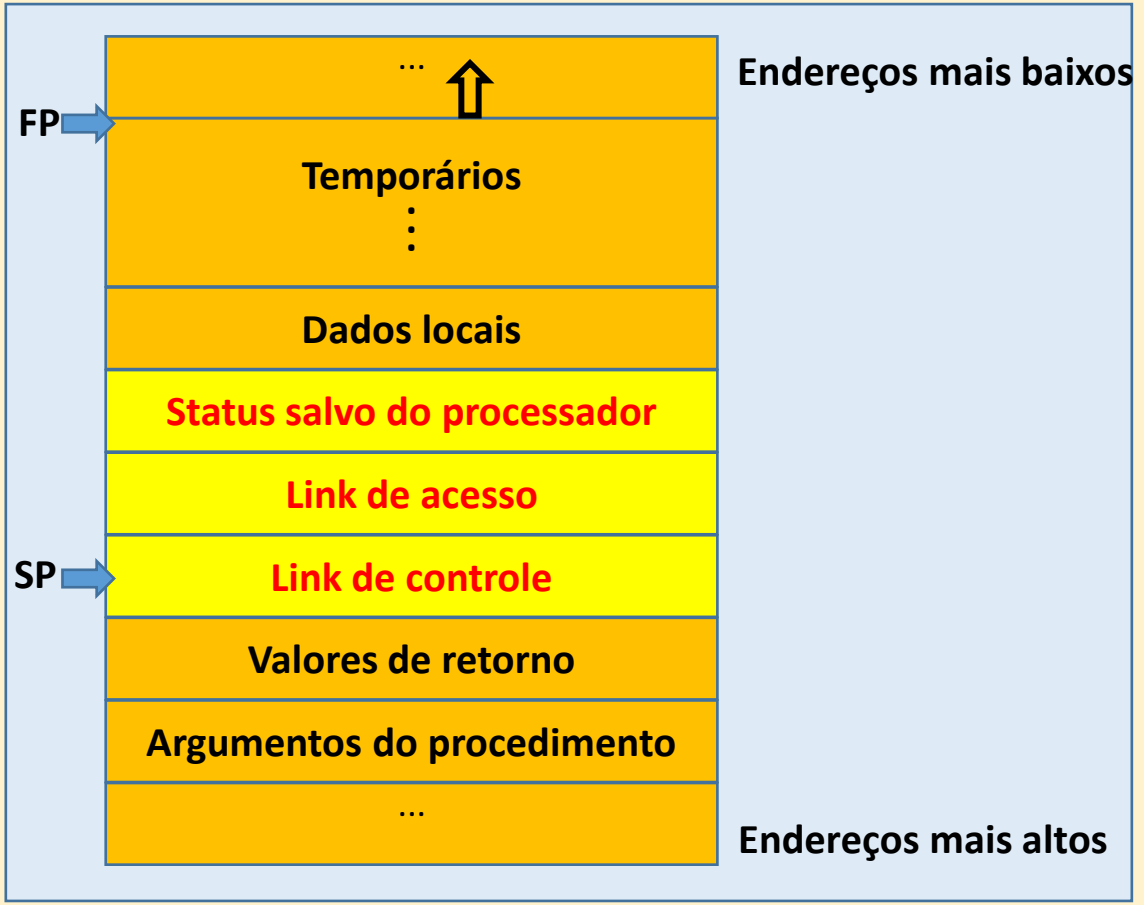


Figure 7.6: Downward-growing stack of activation records

Onde fica o registro de ativação?

- O ponteiro para o topo da pilha (**FP – *frame pointer***) é mantido em um registrador especial.
- Outro registrador permite monitorar a área ocupada pela pilha (***local stack size* - SP**)
  - Os dados da pilha apresentam um comprimento variável.
  - SP mantém atualizada a informação da área total ocupada pelos dados depositados na pilha



Sequência de chamada

- Uma **sequência de chamada** é uma cadeia de instruções (código: **chamador** e **chamado**) para a qual, na pilha:
  1. O **chamador** aloca espaço para os parâmetros e guarda nele os argumentos
  2. O **chamador** aloca espaço para o valor de retorno, a ser gerado pelo **chamado**
  3. O **chamador** empilha o link dinâmico de controle (FP)
  4. O **chamado** aloca espaço para aí armazenar o link estático de acesso
  5. O **chamado** aloca espaço para aí armazenar o status salvo do processador
  6. O **chamado** aloca espaço para os seus dados locais
  7. O **chamado** entra em execução, e pode usar áreas de armazenamento temporário enquanto ele vai sendo executado
  8. O **chamado** faz uma limpeza em suas áreas locais de trabalho (desempilhando dados locais e temporários)
  9. O **chamado** restaura o status de máquina do programa **chamador** (que deve incluir a informação do endereço para onde deve ser feito o retorno ao **chamador**)
  10. O **chamador** extrai o valor de retorno e recupera a área de memória antes ocupada pelos argumentos previamente passados ao **chamado**.

Dados não locais

Fácil: linguagem C

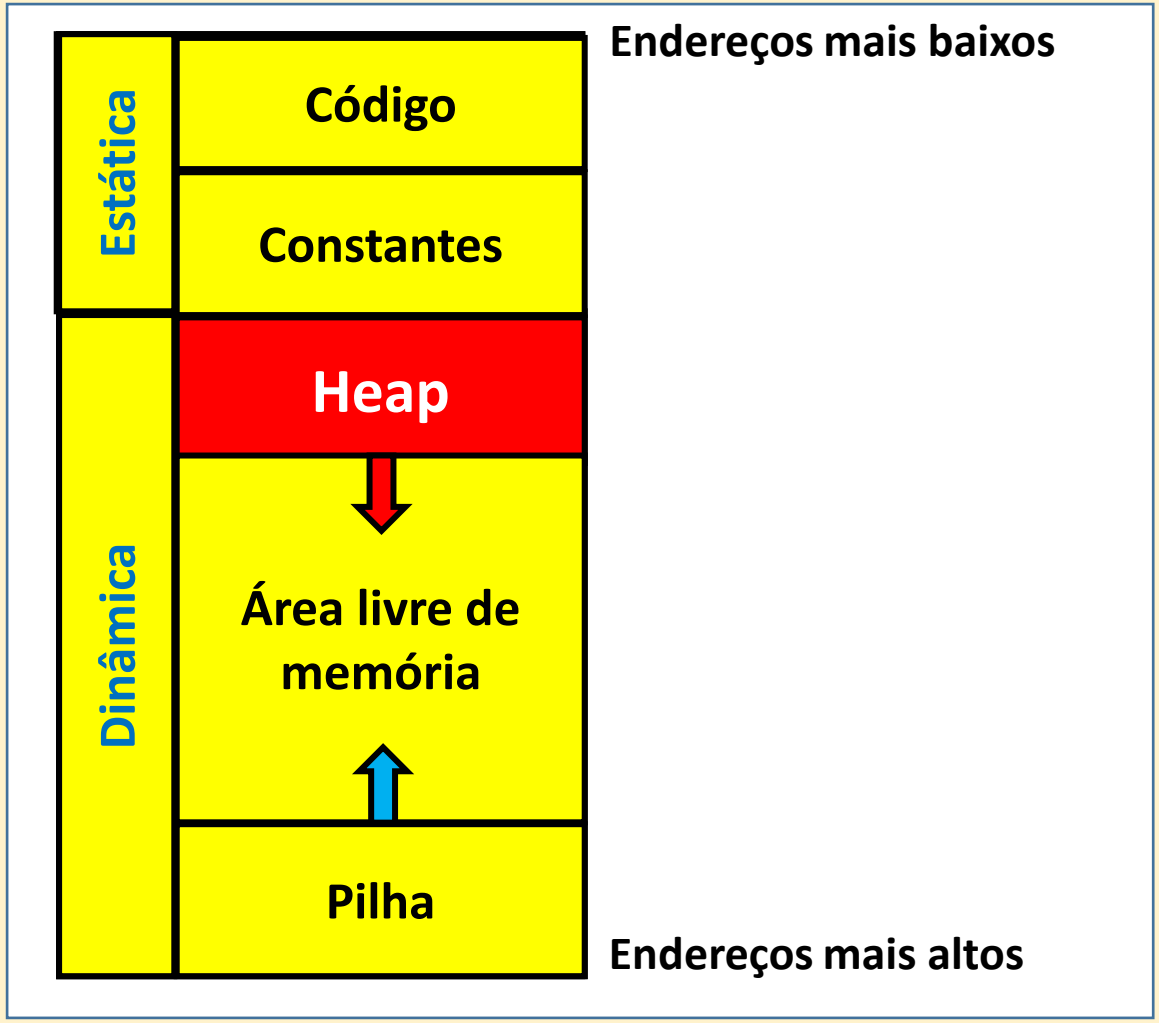
- A linguagem C não apresenta procedimentos aninhados, e todos os seus escopos são estáticos
- **Dados estáticos e variáveis globais** são alocados em uma **região de memória estática** (conhecida em tempo de compilação)
- Todo o **restante**, por default, é alocado na **pilha**
- Para a utilização do ***Heap***, devem ser executados explicitamente os procedimentos malloc() e free() na linguagem de alto nível
- A rigor, sbrk() costuma ser utilizado no nível inferior de abstração, para ampliar ou reduzir a área correntemente alocada de memória

Mais complexo: linguagens C++, C#, Java, Scala, Clojure, ML, Haskell,...

- Essas e outras linguagens em uso são estrutural e funcionalmente mais complexas que a linguagem C
- Uma das fontes adicionais de complexidade é que elas permitem a declaração e o uso de procedimentos aninhados
- Usam o *link* estático de acesso (ou *display*)

Gerenciamento do *Heap*

O *Heap*


Endereços mais baixos
Estática
Código
Constantes
Dinâmica
Heap
Área livre de memória
Pilha
Endereços mais altos

O *Heap*

- Objetos do *Heap* podem ter uma duração indefinida
- O gerenciamento desse tempo de vida pode ser feito automaticamente ou então com a ajuda explícita de um programa gerenciador de memória.

	Java	C	C++
Alocação	**new**	**malloc**	**new**
Dealocação	**garbage collection**	**free**	**delete**

Gerenciador de memória – função e propriedades

- A operação de **alocação** - obtém para os programas **áreas contíguas** do espaço de endereçamento (eventualmente virtual) do computador
- A operação de **dealocação** - libera para **reutilização** áreas de memória previamente alocadas, mas **não mais necessárias**
- **Metas do gerenciador de memória**
  - **Uso eficiente do espaço** de endereçamento - procura fazer bom uso da memória disponível, **evitando a fragmentação**
  - **Eficiência dos programas** – Faz com que seja **rápido o acesso** à memória alocada (tirando proveito de propriedades associadas à **localidade**)
  - **Baixo overhead** – **através da redução do custo administrativo** da alocação

Hierarquia de Memória (ordem de grandeza)

- A tabela abaixo dá uma boa ideia das ordens de grandeza do **tamanho** e da **velocidade** dos elementos de armazenamento utilizados em arquiteturas com memórias organizadas hierarquicamente:

Registradores	10^3 bytes	@ 0,2 ns
Cache L1	10^5 bytes	@ 5 ns
Cache L2	10^7 bytes	@ 20 ns
Memória física	10^{10} bytes	@ 100ns
Memória flash	10^{12} bytes	@ 1ms
Disco	10^{14} bytes	@ 10ms

Localidade

- A memória é copiada para o ***cache*** em ***chunks*** (grupos contíguos do espaço de endereçamento)
- Acessos múltiplos a um mesmo *chunk* melhora a resposta do *cache*
- A simples reorganização (ou, melhor ainda, um projeto com prévio planejamento criterioso) dos dados de um programa pode melhorar dramaticamente o seu desempenho
- O **alocador** e o ***Garbage collector*** podem auxiliar muito nessa tarefa.

Passagem de Parâmetros

MECANISMOS DE PASSAGEM DE PARÂMETROS

- Em uma chamada de procedimento, os parâmetros correspondem às posições do registro de ativação que o chamador deverá preencher com os argumentos, antes de desviar para o código do procedimento chamado, que deve efetuar o processamento desses argumentos.
- O processo de construção desses valores às vezes é conhecido como **amarração** dos parâmetros aos respectivos argumentos (*binding*).
- A forma como o valor dos argumentos é interpretado pelo código do procedimento depende do particular mecanismo de passagem de parâmetros adotado pela linguagem-fonte.
  - Fortran 77 amarra os parâmetros a endereços e não a valores
  - A linguagem C considera todos os argumentos como valores
  - C++, Pascal e Ada oferecem ao programador a possibilidade de escolher o mecanismo desejado de passagem de parâmetros

- Os mecanismos mais comuns de passagem de parâmetros são:
  - Por valor
  - Por referência
  - Por valor-resultado
  - Por nome (também chamado avaliação tardia – *late binding*)
- Outros temas não considerados pelo mecanismo de passagem de parâmetros propriamente dito:
  - A ordem na qual são avaliados os argumentos
  - Muitas linguagens permitem que os argumentos das chamadas causem efeitos colaterais
    - Exemplo: F(++x,x);
      - Compiladores C avaliam seus argumentos da direita para a esquerda

PASSAGEM POR VALOR

- Na passagem por valor, os argumentos dos procedimentos são tratados como expressões, a serem avaliadas **na ocasião da chamada** do procedimento
- Durante toda a execução do procedimento, os valores dos argumentos são mantidos intactos, e interpretados como valores dos parâmetros correspondentes.
  - Este é o único mecanismo de passagem de parâmetros disponível em C
  - É também o mecanismo *default* em Pascal e Ada.
- A seguinte função inc2, escrita em C, não produz o efeito intuitivamente desejado de substituir o valor x do argumento por x+2:

```
Void inc2 (int x)
/* incorreto! */
{ ++x; ++x;}
```

- Em C, é possível permitir a alteração de variáveis não-locais passando ao procedimento chamado seu endereço no lugar de seu valor:

```
Void inc2 (int *x)
/* now ok */
{++(*x);++(*x);}
```

- O mecanismo de passagem por valor não exige qualquer atitude especial por parte do compilador
- É de fácil implementação, bastando para isso escolher a forma mais trivial de interpretação, tanto para a avaliação do argumento como para a construção do registro de ativação

PASSAGEM POR REFERÊNCIA

- A passagem por referência passa ao procedimento chamado o endereço da variável, de modo que o parâmetro é considerado um ponteiro para o argumento
  - É o único mecanismo de passagem de parâmetros em Fortran 77
  - Em Pascal, a passagem por referência é denotada com a palavra-chave var.
  - Em C++ usa-se para isso “&” na declaração do parâmetro.

```
Void inc2( int &x)
  /* passagem de parâmetro por referência em C++ */
  {++x;++x}
```

- A passagem por referência requer que os compiladores calculem o endereço do argumento para guardá-lo no registro de ativação local
- O compilador deve também converter em modos indiretos de endereçamento os acessos locais a parâmetros passados por referência.

- Para chamadas como p(2+3), em Fortran 77 o compilador deve escolher um endereço para a expressão 2+3, computar o valor, armazená-lo nesse endereço, e então passar tal endereço para o procedimento chamado
- Características da passagem por referência:
  - Não se requer que uma cópia seja feita do valor anterior. Para uma estrutura grande, a economia de memória assim obtida pode ser significativa.
  - Passar um argumento por referência impede que modificações sejam feitas no valor do argumento. Tal opção é possível na linguagem C++.

    Void f (const muchdata & X)

    onde muchdata é um tipo de dado com uma estrutura grande.
  - O compilador deverá efetuar uma verificação estática para garantir que x nunca apareça do lado esquerdo (como variável-destino) de uma atribuição.

PASSAGEM POR VALOR-RESULTADO

- Este mecanismo obtém resultados similares à passagem por referência, exceto que não se estabelece nenhum ponteiro para o argumento.
- É conhecido como ***copy-in/copy-out*** ou então ***copy-restore***.
- A passagem por valor-resultado só é distinguível da passagem por referência no caso da ocorrência de "aliasing" (ponteiros diferentes referenciando a mesma posição de memória).
- Por exemplo, no código C a seguir:

```
Void p(int x, int y)
   { ++x;
    ++y;}
main( )
   { int a=1;
     p(a, a);
     return 0;}
```

  - Se a passagem por referência for usada, a variável **a** terá o valor 3 após p ser executada, enquanto
  - Se for usada a passagem por valor-resultado, após a execução de p o valor da variável **a** será 2.

- Nesse mecanismo, não são especificados :
  - A **ordem em que resultados retornam** nos argumentos
  - Os **endereços dos argumentos** são calculados apenas na entrada ou recalculados na saída?
- Passagem por valor-resultado requer a modificação da estrutura básica da pilha de tempo de execução e a sequência de chamada:
  - O registro de ativação **não pode ser liberado** pelo procedimento chamado
  - O procedimento chamador deve **empilhar o endereço do argumentos** na pilha
    - antes de construir o novo registro de ativação, ou
    - antes de recomputar tais endereços de retorno.

- Este é o mais complexo dos mecanismos de passagem de parâmetros, e é também conhecido como **avaliação tardia**.
- A ideia é que o argumento não seja avaliado enquanto não acontecer seu real uso no programa chamado
- Como exemplo, no código

```
Void p (int x)
{ ++x;}
```

o efeito de p(a[i]) é avaliar ++(a[i]).

Então, se I devesse modificar-se antes do uso de x dentro de P, então o resultado seria diferente tanto da passagem por referência como da passagem por valor-resultado.

PASSAGEM POR NOME

- Por exemplo, no código em linguagem C abaixo:

```
int i;
int a[10];
void p(int x)
  {  ++i;
     ++x;  }
main()
  {  i=1;
     a[1]=1;
     a[2]=2;
     p(a[i]);
     return 0;  }
```

- O resultado da chamada p(a[i]) é que a[2] recebe 3 e a[1] permanece inalterado.

- A interpretação do **passagem por nome** é a seguinte:
  - O texto de um argumento no ponto de chamada é tratado como se fosse uma função em seu próprio escopo de declaração
  - Os **argumentos são avaliados todas as vezes que o nome do parâmetro é referenciado** no procedimento
  - Os **argumentos são avaliados no ambiente do procedimento chamador** enquanto o **procedimento é executado em seu ambiente original** de definição.

- O mecanismo de passagem por nome foi usado na importante e histórica linguagem Algol 60, e acabou sendo considerado inadequado por diversas razões:
  - Dá **resultados imprevisíveis** e não-intuitivos no caso de ocorrerem **efeitos colaterais**
  - É **difícil de implementar**
  - É **oneroso e ineficiente**

- Uma variação desse mecanismo, denominado **avaliação preguiçosa** (***lazy evaluation***) tornou-se recentemente popular em **linguagens de programação puramente funcionais.**

ARQUITETURAS

Arquiteturas-alvo

- Arquiteturas de alto padrão (onerosas)
  - **ARM** – Tablets, Telefones celulares, Ultrabooks, Raspberry Pi.
  - **amd64** – PCs e servidores Intel (e AMD)
  - **SPARC** – Servidores Oracle
  - **Power PC** – Servidores IBM
- Arquiteturas de baixo padrão (mais baratas)
  - **6505** – arquitetura de 8 bits
  - **Increasingly** – chips de baixo padrão ARM32...

A pilha

- Em geral, as **convenções de chamada** costumam ser assim definidas:
  - *Um "contrato estático de alocação de memória" entre chamador e chamado, que se garante valer em tempo de execução.*
- As convenções de chamada estabelecem regras para a indicação de
  - Argumentos e valores retornados (p/ex, baseados em informações de tipo estático)
  - Dimensões dessas entidades
  - Localização em que se encontram essas entidades (em registradores ou na pilha)

Convenções de chamada para o ARM

Registrador	Na chamada	No retorno
R0-3	Parâmetro ou não utilizado	Valor retornado ou não utilizado
R4-11	Preservado	O mesmo que na chamada
R12	Indefinido	Indefinido
R13 ‘sp’	Ponteiro para a pilha	O mesmo que na chamada
R10 ‘lr’	Endereço de retorno	(sem vínculos)
R15 ‘pc’	Contador de instruções	Endereço de retorno

Para obter informações mais detalhadas, recomenda-se consultar o manual de referência do ARM

Convenções de chamada para o ARM (modificadas ‘fp’)

Registrador	Na chamada	No retorno
R0-3	Parâmetro ou não utilizado	Valor retornado ou não utilizado
R4-11	Preservado	O mesmo que na chamada
R12 ‘fp’	Indefinido	Indefinido
R13 ‘sp’	Ponteiro para a pilha	O mesmo que na chamada
R10 ‘lr’	Endereço de retorno	(sem vínculos)
R15 ‘pc’	Contador de instruções	Endereço de retorno

Instruções ARM (panorama superficial)

Mnemônico	Instrução	Função executada
MOV B,A	*Move*	Copia em B o conteúdo de A
BL label	*Branch and Link*	Guarda em lr o endereço da próxima instrução e desvia para *label*
STMFD spl,{regs}	*Store Multiple/ Fully Descending*	Empilha todos os registradores indicados na pilha (esta cresce de cima para baixo)
LDMFD spl, {regs}	*Load Multiple/ Fully Descending*	Desempilha todos os registradores indicados da pilha (de cima para baixo)

Códigos de Chamada e Retorno no ARM

```
Chamador:
MOV r0, <param1>
MOV r1, <param2>
MOV r2, <param3>
MOV r3, <param4>
STMFD sp!,{r12} // r12 contém FP
BL Chamado
```

```
Chamado:
MOV r12, sp
STMFD sp!,{r4-r11,lr}
Corpo com parâmetros em r0-r3...
MOV r0, <valor de retorno>
LDMFD sp!,{r4-r11,pc}
```

```
Chamador:
LDMFD sp!, {r12} // restaura o FP previamente salvo em r12
Usa o valor de retorno, em r0
```

Código de chamada e retorno do AMD64 (Intel)

Chamador:

coloca os valores dos argumentos na pilha ou registradores

call Chamado

```
Chamado:
push %rbp
move %rbp, %rsp
sub %rsp, <tamanho da frame>
Corpo e parâmetros em [%rbp-…]…
move %rax, <valor de retorno>
move %rsp, %rbp
pop %rbp
Ret
```

```
Remove da pilha os argumentos
Usa o valor de retorno, em %rax
```

Código de chamada e retorno do AMD64 (Intel)

```
Chamador:
MOV r0, #1
MOV r1, #1
STMFD sp!,{r12} // r12 contém FP
BL Chamado
```

```
Chamado:
MOV r12, sp
STMFD sp!,{r4-r11,lr}
ADD r0,r0,r1
LDMFD sp!,{r4-r11,pc}
```

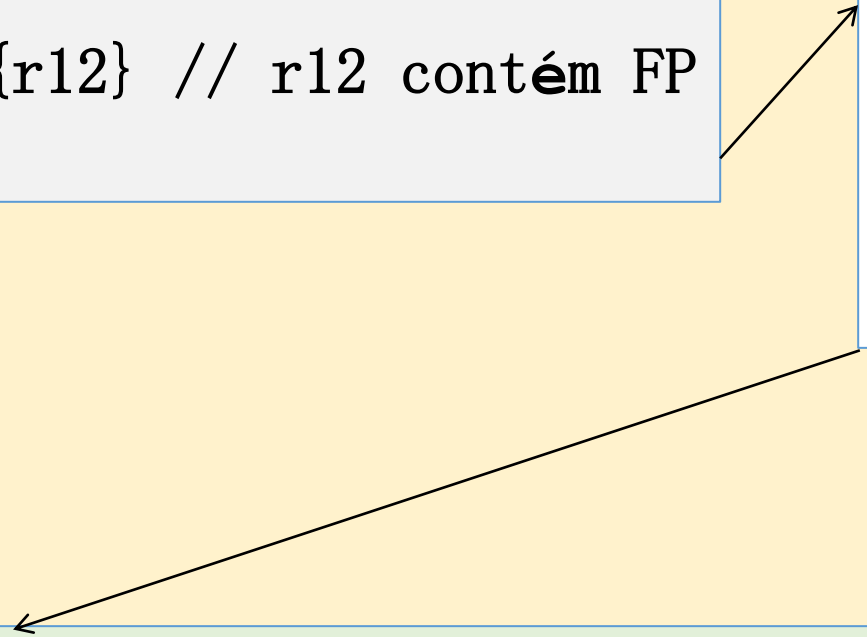
```
Chamador:
LDMFD sp!, {r12} // restaura o FP previamente salvo em r12
```

Continua...

Código de chamada e retorno do AMD64 (Intel)

```
Chamador:
MOV r0, #1
MOV r1, #1
STMFD sp!,{r12} // r12 contém FP
BL Chamado
```

```
Chamado:
MOV r12, sp
STMFD sp!,{lr}
ADD r0,r0,r1
LDMFD sp!,{pc}
```



```
Chamador:
LDMFD sp!, {r12} // restaura o FP previamente salvo em r12
```

Continua...

Código de chamada e retorno do AMD64 (Intel)

```
Chamador:
MOV r0, #1
MOV r1, #1
STMFD sp!,{r12} // r12 contém FP
BL Chamado
```

```
Chamado:
MOV r12, sp

ADD r0,r0,r1
MOV pc, lr
```

```
Chamador:
LDMFD sp!, {r12} // restaura o FP previamente salvo em r12
```

Créditos

Apresentação baseada em notas de aula do curso Construção de Compiladores NYU Courant Institute – Primavera de 2018
cs.nyu.edu/courses/spring18/CSCI-GA.2130-001/lectures/lecture-8.pdf

O tópico sobre passagem de parâmetros foi extraído do livro

Kenneth C. Louden

COMPILER CONSTRUCTION - Principles and Practice, cap. 7