Instituto Superior de Ciências do Trabalho e da Empresa

Departamento de Ciências e Tecnologias da Informação

Arquitectura de Computadores (II)

Textos de apoio

– O processador MAC1 – arquitectura e programação –

Draft – Versão 0.02 – Abril de 2005 – João Baptista e Tomás Brandão.

Índice

1. Introdução

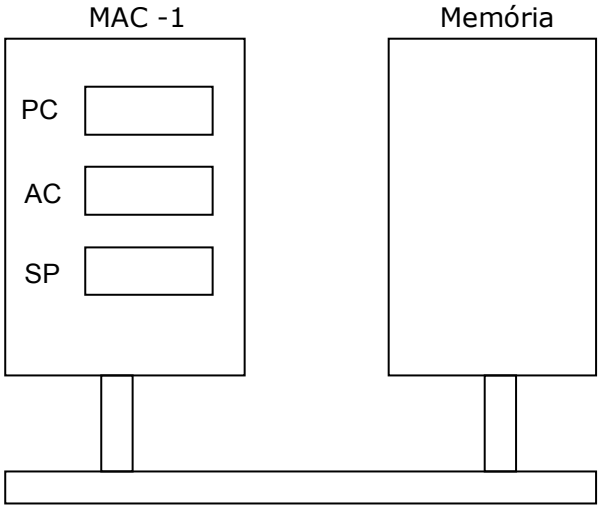
O processador MAC-1 foi desenvolvido pelo professor *Andrew Tanenbaum* para fins didácticos. Este processador é virtual (não existe fisicamente) tal como acontece, por exemplo, com a máquina virtual java (JVM). Para executar programas escritos na linguagem deste processador encontram-se disponíveis na página da disciplina alguns simuladores com funcionalidades básicas.

A arquitectura relativamente simples e o pequeno conjunto de instruções fazem com que o processador MAC-1 seja indicado para uma introdução à linguagem *assembly* e para compreender melhor conceitos relacionados com a arquitectura de um processador, execução de programas, chamadas a rotinas (ou métodos), acesso à memória e interface básica para I/O.

1.1. Arquitectura funcional MAC-1

Quando se escreve um programa em *assembly*, há que ter em conta a máquina à qual se destina o programa. Neste caso, a máquina que corre os programas é o processador MAC-1.

O processador necessita também de aceder a uma memória para ler as instruções que compõem o programa a executar, e para ler e escrever dados. De momento não é particularmente relevante, mas vamos admitir que a memória tem uma dimensão de 4K palavras, ou seja, possui endereços de 0 a 4095.



No interior do CPU existem três registos importantes para o programador:

- PC (*program counter*)
- AC (acumulador)
- SP (*stack pointer*)

Para já vamos apenas considerar o registo AC (Acumulador). O AC é um registo utilizado por grande parte das instruções *assembly* do MAC-1. Por exemplo, existem instruções que lêm um

valor da memória deixam esse valor em AC; existem outras que escreve na memória o valor que se encontra nesse momento em AC, etc.

Antes de começar a programar em *assembly*, é conveniente ter bem assentes alguns conceitos básicos que relacionam a programação em alto nível com a realidade física do hardware. Para tal, considere o seguinte pseudo-programa[1], onde são definidas três variáveis, designadas por X, Y e Z.

```
int X, Y, Z;

main()
{
        X=2;
        Y=X+1;
        Z=X+Y;
}
```

A execução deste programa muito simples consiste na realização das seguintes acções:

- colocar o valor 2 na variável X;
- incrementar X e colocar o resultado em Y;
- somar X com Y e colocar o resultado em Z;

Para o processador MAC-1, uma variável corresponde a uma posição de memória. Sendo assim, quando o programa entrar em execução, existirão três posições de memória correspondentes a cada uma das variáveis utilizadas.

A sequência de acções definidas pelo programa tem os seguinte efeitos nas posições de memória indicadas:

- X=2 – escrever o valor 2 na posição de memória correspondente a X;
- Y=X+1 – ler o valor X da memória, somar esse valor com 1 e escrever o resultado na posição de memória correspondente a Y;
- Z=X+Y – ler o valor X da memória, ler o valor Y da memória, somar os dois valores e escrever o resultado na posição de memória correspondente a Z;

Vejamos agora como escrever um programa equivalente ao anterior utilizando a linguagem *assembly* do processador MAC-1. Para fazer os programas mais elementares vamos usar o seguinte conjunto de instruções:

Instrução	Efeito
`loco x`	Escreve o valor x no acumulador (AC)
`lodd x`	Copia para AC o conteúdo da posição de memória x
`stod x`	Copia para a posição de memória x o valor dado em AC
`addd x`	Soma o conteúdo da posição de memória x ao valor de AC e guarda o resultado de volta em AC

[1] Muitas vezes vamos usar como ponto de partida pequenos programas escritos numa pseudo-linguagem de programação com sintaxe mais ou menos familiar, parecida com C++ ou com Java, e não muito diferente de outras linguagens de programação se uso comum. Estes pseudo-programas servem apenas para expressar a ideia do que se pretende programar em *assembly*.

Vamos admitir que as variáveis X, Y, Z se localizam, respectivamente, nas posições de memória 100, 101 e 102. Nestas condições, um programa em *assembly* equivalente ao anterior pode-se escrever do seguinte modo:

```
loco 2      # coloca o valor 2 em AC;
stod 100    # copia o valor de AC para a posição de memória 100 (X);

loco 1      # coloca o valor 1 em AC;
addd 100    # adiciona o valor da posição 100 (X) a AC; AC fica com 3;
stod 101    # copia o valor de AC para a posição de memória 101 (Y);

lodd 100    # copia o valor da posição 100 para o AC; AC fica com 3;
addd 101    # adiciona o valor da posição Y ao AC; AC fica com 5;
stod 102    # copia de AC para a posição 102 (Z);
```

1.2. Organização de um programa

Normalmente usaremos a organização de memória dividida em dois blocos principais: primeiro um bloco que contem as variáveis (estáticas), e depois um bloco com o programa propriamente dito. A primeira posição de memória (posição 0) é onde o programa começa, logo tem que ser ocupada por uma instrução. Usaremos para o efeito a instrução **`jump`** que tem como efeito fazer o programa "saltar" para a posição indicada.

Com esta organização o programa ficará localizado em memória da forma indicada na figura, ficando as variáveis colocadas nas posições de memória 1, 2 e 3.

```
jump 4

0           # X
0           # Y
0           # Z

loco 2      # x=2
stod 1

loco 1      # y=1+x
addd 1
stod 2

lodd 1      # z=x+y
addd 2
stod 3

halt
```

Posição	Conteúdo	
0	jump 4	
1	0	← X
2	0	← Y
3	0	← Z
4	loco 2	
5	stod 1	
6	loco 1	
7	addd 1	
8	stod 2	
9	lodd 1	
10	addd 2	
11	stod 3	
12	halt	
...	...	

1.3. Assembler

A tradução entre as instruções – mnemónicas simbólicas – que usamos para escrever os programas (`loco`, `stod`, etc.) e o respectivo código-máquina é feita por um programa que se designa por *assembler*[2].

O *assembler* que vamos usar contém alguns mecanismos que permitem aumentar a legibilidade dos programas, nomeadamente a possibilidade de definir constantes e utilizar etiquetas (*labels*).

A possibilidade de definir constantes é ilustrada no seguinte exemplo:

```
UM=1
loco UM
stod 100
```

A directiva `UM=1` define uma constante chamada UM com valor 1.

Após essa definição, sempre que se escrever UM no programa, o *assembler* traduz para o valor 1. Assim, por exemplo, onde está **`loco UM`** o *assembler* traduzirá para **`loco 1`**. UM é portanto um símbolo para o valor 1. Note que uma vez definida a constante desta maneira, o valor que representa não pode ser modificado (não se trata de uma variável...).

A utilização de etiquetas é outro mecanismo que pode ser utilizado para facilitar a elaboração e a leitura do código. Este mecanismo é ilustrado no seguinte exemplo:

```
      jump MAIN

X:    0
Y:    10
Z:    20

MAIN: loco 1
      stod X
      ...
```

X, Y, Z e MAIN são etiquetas – designações atribuídas a posições de memória. Os valores que ficam definidos são os endereços das posições de memória designadas.

Neste caso, X designa a posição de memória com endereço 1, logo X vale 1. Pelo mesmo raciocínio Y vale 2, Z vale 3 e MAIN vale 4. Note que uma coisa é o valor da posição de memória, outra coisa é o valor do seu conteúdo. No exemplo anterior, X vale 1, mas o seu conteúdo é 0, Y vale 2 mas o seu conteúdo é 10, Z vale 3 e o seu conteúdo é 20, MAIN vale 4 e o seu conteúdo será o código máquina da instrução loco 1 !

O *assembler* substitui as designações definidas pelos seus valores. Por exemplo, a instrução **`stod X`** é transformada em **`stod 1`** e, desta forma, actua sobre a posição de memória 1.

[2] Há falta de boas traduções, optou-se por se usar o termo em inglês *assembler*. Julgamos ser preferível do que uma tradução directa como *montador* ou uma pseudo-tradução como *assemblador*.

Usando etiquetas, podemos re-escrever o programa da secção anterior na seguinte forma:

```
        jump MAIN

x:      0
y:      0
z:      0

MAIN:   loco 2      # x = 2
        stod x

        loco 1      # y = 1 + x
        addd x
        stod y

        lodd x      # z = x + y
        addd y
        stod z

        halt
```

Funcionalmente este programa é idêntico programa da secção 1.2 – ambos produzirão exactamente o mesmo resultado (código máquina) quando passarem pelo *assembler*. Basta aliás substituir cada ocorrência de x, y, z e MAIN pelos seus valores para se obter o programa anterior.

A utilização de etiquetas e de constantes simbólicas são um mecanismo muito simples: é a mera utilização de um nome em vez de um valor numérico. Mas a sua utilização facilita a elaboração de um programa em *assembly* e aumenta bastante a sua legibilidade. Escrever:

`stod X` em vez de `stod 1`,

aproxima-nos mais da noção de abstracta de variável com "nome", em vez da noção mais próxima da realidade física, que é uma variável como sendo uma "posição de memória".

2. Programas sequenciais simples

2.1. Endereçamento directo e imediato

Para desenvolver programas simples como os dos exemplos anteriores vamos usar as seguintes instruções do MAC-1:

Instrução	Descrição	Especificação
`loco X`	escreve o valor X no registo AC	AC ← X
`lodd P`	copia da posição de memória P para AC	AC ← M[P]
`stod P`	copia o valor de AC para a posição de memória P	M[P] ← AC
`addd P`	soma o conteúdo da posição de memória P a AC	AC ← AC + M[P]
`subd P`	subtrai o conteúdo da posição de memória P a AC	AC ← AC – M[P]

A instrução **`loco X`** coloca o valor X em AC. Esta forma de atribuir valores a um registo designa-se por ***endereçamento imediato***. É dada esta designação sempre que o valor a guardar num registo é dado directamente pela própria instrução, sem que para obter esse valor o processador tenha que aceder a dados em memória.

As restantes quatro instruções envolvem o acesso a uma posição de memória. Como a posição de memória a aceder é dada explicitamente (directamente) pela instrução, esta forma de aceder à memória designa-se por ***endereçamento directo***.

Por exemplo, a instrução **`stod P`** causa um acesso à memória para escrever um valor na posição **P**; as restantes instruções acedem a uma posição de memória para ler o valor nela contido.

Quanto à instrução **`jump`**, que também tem aparecido nos exemplos, de momento vamos utilizá-la apenas como a primeira instrução do programa, de modo a fazer com que o programa salte as posições de memória usadas para variáveis e constantes. A instrução jump pertence a conjunto de instruções designado por ***instruções de salto***, que serão analisadas mais adiante.

Instrução	Descrição	Especificação
`Jump P`	O programa salta para a instrução situada na posição de memória P.	PC ← P

Estas instruções permitem desenvolver programas sequenciais muito simples, envolvendo variáveis estáticas, atribuições de valores e operações de adição e subtracção.

Exemplo

Considere o seguinte pseudo-programa:

```
int x=0, y=7, z=1;

main ()
{
    x = -y - z;
    z = x - z + 1;
}
```

Uma tradução deste programa para assembly poderá ser a seguinte:

```
        jump main

x:      0           # inicialização
y:      7
z:      1

main:   loco 0
        subd y
        subd z
        stod x      # x = 0 - y - z

        loco 1
        addd x
        subd z
        stod z      # z = 1 + x - z

        halt
```

2.2. Alguns erros comuns

A única instrução que permite manipular constantes de uma forma imediata é a instrução **`loco`**. Este facto obriga a que operações comuns, tal como o incremento de uma variável (e.g., X+1), se façam começando por se colocar em primeiro lugar a constante em AC, ou seja:

```
loco 1
addd X
```

Repare que se faz "1+X" em vez de "X+1". O contrário seria impossível:

```
lodd X      # copiar X para AC
????        # então e agora para somar 1 ?
```

Nestas condições pode haver a tentação de fazer um erro crasso:

```
lodd X      # copiar X para AC
addd 1      # somar a AC o conteúdo da posição de memória 1 (!?)
```

A questão é que **`addd 1`** não soma 1 ao valor em AC; soma é o conteúdo da posição da memória 1. Ou seja, com esta última sequência, em vez de "X+1" estaríamos a fazer "X+M[1]".

No entanto, é considerada uma boa prática (no contexto do processador MAC-1) definir algumas constantes em memória para poderem ser usadas na programação. Sendo assim seria bastante razoável a seguinte versão:

```
        ...
UM:     1
        ...
        lodd X      # copiar X para AC
        addd UM     # somar o conteúdo da posição UM
        ...
```

Outro erro comum que envolve estas instruções (até certo ponto semelhante ao anterior) é uma certa confusão existente entre a instrução `loco` e a instrução `lodd`. Para ilustrar esta situação, considere o seguinte programa em que se encontram definidas posições de memória x, y e z, com conteúdos inicializados a 0, 0 e 1, respectivamente:

```
        jump main

x:      0
y:      0
z:      1

main:   loco z      # será que carrega 1 no acumulador ????
        stod x

        lodd z      # então e aqui, o que carrega no acumulador ?
        stod y

        halt
```

Como já foi dito anteriormente na secção 1.3, a designação *z* vai ser substituída pelo valor 3 (porque se refere à posição de memória 3). Sendo assim, ao ser executada a instrução `loco z`, vai ser carregado no acumulador o valor 3. Ao ser feito `lodd z` vai ser carregado no acumulador o conteúdo da posição de memória 3, que é o valor 1.

Desta feita, quando o programa terminar, o conteúdo da posição de memória *x* vai ser o valor 3, ao passo que o conteúdo da posição *y* vai ser o valor 1.

No fundo a diferença está em fazer AC = z (`loco`) ou AC = M[z] (`lodd`) . Não esquecer é que para o programa a designação *z* é o mesmo que 3. É importante clarificar esta ideia para evitar problemas no futuro.

3. Instruções de salto

3.1. Condições

Considere o seguinte fragmento de código que calcula o valor absoluto de x.

```
if ( x < 0 )
      x = -x;
```

O programa contém uma instrução de controlo – *if* – que condiciona a execução da instrução seguinte. Assim sendo, se a condição `x<0` for verdadeira a instrução `x=-x` é executada; caso contrário não o é.

A mesma ideia pode-se expressar de outro modo:

```
      se ( x >= 0 ) saltar para P
      x = -x;
P:    ...
```

Esta expressão pretende ilustrar uma ordem para o programa seguir (ou "saltar") para outra posição, neste caso abstendo-se de fazer a instrução `x=-x`, por cima da qual salta.

Neste exemplo o salto diz-se ***condicional***, pois só se realiza se a condição `x>=0` for verdadeira. Assim, se `x>=0`, a instrução `x=-x` não é executada porque ocorre o salto; se `x<0`, então o salto não se processa e a instrução `x=-x` é de facto executada.

A mesma ideia de salto condicionado a uma condição é expressa pela instrução assembly **`jpos P`**. Quando executa esta instrução, o processador olha para o valor que estiver em AC; se esse valor for maior ou igual a zero, salta para a posição P; caso contrário o programa segue o curso normal, sendo executada a instrução que vem a seguir à de salto.

Usando esta instrução podemos escrever o seguinte programa em assembly:

```
      lodd X
      jpos P      # se X >= 0 saltamos para P
      loco 0      # se ainda estamos aqui é porque X < 0
      subd X
P:    ...
```

Existem outras instruções de salto condicional. Na seguinte tabela encontram-se sintetizadas todas as instruções possíveis de utilizar no MAC-1:

Instrução	Descrição	Especificação
`jpos P`	salta para P se AC ≥ 0	if AC ≥ 0 then PC ← P
`jneg P`	salta para P se AC < 0	if AC < 0 then PC ← P
`jzer P`	salta para P se AC == 0	if AC == 0 then PC ← P
`jnze P`	salta para P se AC != 0	if AC != 0 then PC ← P

Em qualquer um dos casos o efeito é o mesmo: se a condição se verificar o programa segue a partir da posição de memória P, ou seja, "salta" para a posição P. Se a condição não se verificar, o programa executa a instrução seguinte, sem que ocorra o salto.

Em todos os casos a condição depende apenas do valor que, no momento, estiver no AC:

- `jpos` faz com haja "salto" se esse valor for maior ou igual a 0;
- `jneg` faz com haja "salto" se esse valor for menor do que 0;
- etc.

A instrução `jump P` que tem também aparecido nos exemplos (e.g., `jump main`) faz com que o programa salte inevitavelmente para a posição P. Neste caso o salto é ***incondicional***, pois não depende de nenhuma condição, isto é, salta-se sempre.

Exemplo
Trocar o conteúdo de duas variáveis x e y, apenas no caso de x ser maior do que y.

Em pseudo-código poderíamos escrever:

```
...
if ( x > y ) {
    aux = x;
    x = y;
    y = aux
}
...
```

Ou, utilizando a notação de saltos:

```
     ...
     se ( x-y < 0 ) saltar para P
     aux = x;
     x = y;
     y = aux;
P:   ...
```

Neste caso a condição original é `x > y`. Na linguagem *assembly* do MAC-1 não é possível utilizar directamente esta condição pois só existem instruções de salto relativas a comparações com 0, i.e., ver se um número é positivo ou negativo, ver se é igual ou diferente de zero...

Para contornar esta limitação, em vez de condições do género `x > y`, teremos que usar condições do género `x-y > 0`, que são equivalentes, mas comparam com 0. Posto isto, podemos escrever o programa em *assembly* da seguinte forma:

```
     ...
     lodd x
     subd y
     jneg P            # se x < y saltar para P

     lodd x
     stod aux          # aux = x
     lodd y
     stod x            # x = y
     lodd aux
     stod y            # y = aux

P:   ...
```

Exemplo
Guardar numa variável *m* o maior de dois números *x* e *y*.

Em pseudo-código pode-se escrever:

```
...
if ( x >= y )
      m = x;
else
      m = y;
...
```

Neste caso temos uma estrutura *if-else*. Repare que uma estrutura deste tipo envolve dois saltos:

- se a condição se verificar, é executado o bloco do *if* e a seguir salta-se por cima do bloco do *else*;
- se a condição não se verificar, salta-se directamente para o bloco do *else*

Ou seja,

```
      se ( x-y < 0) saltar para ELSE
      m = x;
      saltar para P;
ELSE: m = y;
P:    ...
```

Em *assembly*:

```
      lodd x
      subd y
      jneg ELSE

      lodd x      # se está aqui é porque x >= y
      stod m
      jump P

ELSE: lodd y      # se está aqui é porque x < y
      stod m

P:    ...
```

Exemplo
Código que guarda numa variável *z* o valor de um número *x* caso este esteja entre 0 e 20.

```
if ( x >= 0 && x <= 20 )
      z = x;
...
```

```
      se x < 0 saltar para P;
      se 20 - x < 0 saltar para P;
      z = x;
P:    ...
```

Em *assembly*:

```
        lodd x      # se x < 0 saltar para P
        jneg P

        loco 20
        subd x
        jneg P      # se x > 20 (ou seja, 20-x < 0) saltar para P

        lodd x
        stod Z
P:      ...
```

3.2. Ciclos

Vejamos agora os programas com ciclos. Por exemplo: considere o seguinte programa que calcula o produto 10×a, utilizando para esse efeito um **ciclo *do...while*** :

```
a = 3;
m = 0;
i = 0;

do {
      m = m + a;
      i = i + 1;

} while {i <= 10)
```

Tal como acontece na generalidade dos ciclos, num ciclo do tipo *do...while*, as instruções contidas no "interior" do ciclo são repetidas enquanto se verificar determinada condição. A verificação da condição é feita no final de cada iteração do ciclo.

A mesma ideia do programa anterior mas posta noutra forma, é ilustrada no seguinte pseudo-código:

```
        a = 3;
        m = 0;
        i = 0;

DO:     m = m + a;
        i = i + 1;
        se (10-i >= 0) saltar para DO;
        ...
```

Repare que um ciclo *do...while* pode então ser construído da seguinte maneira:

1. executar as instruções interiores ao ciclo;
2. testar a condição (se for verdadeira saltar para o ponto 1)

Vejamos agora como ficaria o código *assembly* correspondente ao exemplo anterior:

```
          jump CICLO

a:        3        # a = 3, por exemplo
DEZ:      10
m:        0
i:        0

CICLO:    lodd m
          addd a
          stod m   # m = m + a

          loco 1
          addd i
          stod i   # i = i + 1

          lodd DEZ
          subd i
          jpos CICLO    # se (i <= 10) volta a repetir

          halt
```

Exemplo

Pretende-se escrever um programa que calcula a soma dos *n* primeiros números naturais.

Em pseudo-código:

```
int n = 5, i = 1, S = 0;

do {
       S = S + i;
       i = i + 1;
} while (i <= n)
```

Em *assembly*:

```
       jump CICLO

n:     5          # n = 5, por exemplo
i:     1
S:     0

CICLO: lodd S
       addd i
       stod S     # S = S + i

       loco 1
       addd i
       stod i     # i = i + 1

       lodd n
       subd i
       jpos CICLO  # se n-i >= 0 continua em ciclo

       halt
```

Vejamos agora um programa equivalente ao mostrado anteriormente, para calcular 10×a, mas utilizando agora um **ciclo *for***:

```
int a = 3, m = 0;

for (int i=0; i!=10; ++i)
      m = m + a;
```

Um pseudo-código mais próximo do assembly poderia ser o seguinte:

```
      a = 3;
      m = 0;
      i = 0;
C:    se ( i - 10 == 0 ) saltar para P;
            m = m + a;
            i = i + 1;
            saltar para C;
P: ...
```

Repare que o ciclo *for* envolve a utilização de dois saltos: um salto condicional, no início de cada iteração, para se saber se o ciclo chegou ao fim; e um salto incondicional no fim de cada iteração, para se voltar ao início do ciclo. Deste modo, a execução do ciclo consiste no seguinte algoritmo:

1. avaliar a condição e saltar para fim do ciclo se for caso disso ;
2. executar as instruções situadas no interior do ciclo;
3. voltar ao ponto 1

O programa *assembly* correspondente pode-se então escrever do seguinte modo:

```
      jump CICLO

a:    3           # a = 3, por exemplo
m:    0
i:    0
DEZ:  10

CICLO:
      lodd i      # testar o fim do ciclo: se i==10 termina
      subd DEZ
      jzer FIM

      lodd m      # m = m + a
      addd a
      stod m

      loco 1      # i = i + 1
      addd i
      stod i

      jump CICLO

FIM:  halt
```

Se o ciclo tivesse sido escrito utilizando um **ciclo *while***, o programa equivalente em *assembly* seria o mesmo que o desenvolvido para o **ciclo *for***. Um pseudo-código para o mesmo programa, mas escrito com um ciclo *while*, podia ser escrito da seguinte forma:

```
a = 3;
m = 0;
i = 0;

while ( i != 10 ) {
    m = m + a;
    i = i + 1;
}
```

Traduzindo este programa para *assembly* chegaríamos ao mesmo programa do exemplo anterior.

4. A pilha (*stack*)

4.1. Organização.

Como se tem tornado evidente ao longo deste texto, a organização da memória é uma preocupação importante e constante quando se fazem programas em *assembly*.

A arquitectura do processador MAC-1 favorece a organização de memória ilustrada na figura. Nas primeiras posições de memória existe um bloco com as variáveis estáticas, depois um bloco com programa [3], como já foi visto nas secções anteriores.

Quer a zona de variáveis estáticas quer o programa são blocos de dimensão fixa. Em todos os programas que fizemos até agora o conjunto das variáveis utilizadas é conhecido à partida - fica determinado quando se escreve o programa. Quando o programa é carregado na memória para ser executado, cada variável passa a ocupar uma posição de memória que permanece a mesma até o programa acabar. Assim sendo, neste tipo de programas, o número de variáveis e a dimensão da memória que ocupam, permanecem constantes desde o início até ao fim da execução do programa.

Daqui para a frente vamos começar a fazer programas com outras exigências que não são compatíveis com este modelo de memória tão simples, nomeadamente programas que envolvem chamadas a funções que utilizam variáveis locais próprias. A implementação destes mecanismos implica a utilização de um bloco de memória "especial" designado por pilha (ou *stack*).

A pilha é uma zona da memória de dimensão variável. Quando o programa começa, a pilha está vazia (ocupa 0 posições de memória). Ao longo da execução do programa, a pilha pode crescer, ocupando posições de memória não utilizadas, que se situam abaixo da zona de topo ocupada pelas instruções do programa.

A pilha responde às necessidades de memória dinâmica do programa em tempo de execução: cresce quando o programa precisa de mais memória para guardar dados; decresce quando deixa de precisar desses dados. Cresce, por exemplo, quando o programa cria novas variáveis ou quando é chamada uma função; decresce quando o programa retorna de uma função ou quando "destroi" variáveis.

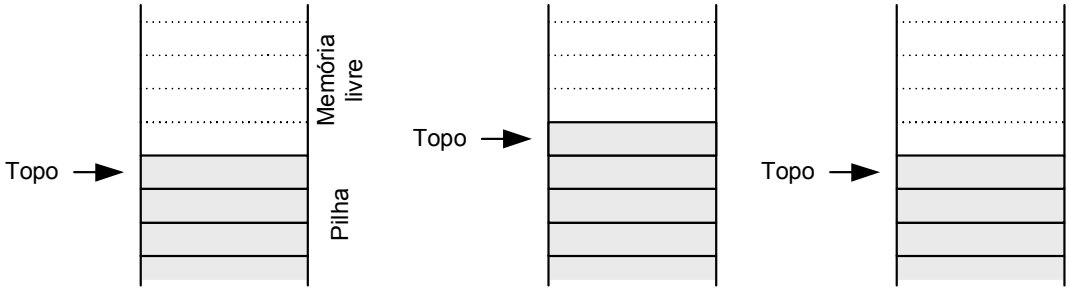
4.2. Instruções PUSH e POP

Como se disse na secção anterior, a pilha localiza-se na base da memória e cresce na direcção contrária aos endereços. O SP (*stack pointer*) indica a posição de memória em que corresponde ao topo da pilha; a pilha situa-se dessa posição para baixo.

[3] E na primeira posição de memória colocamos uma instrução para saltar para o início do programa.

Uma pilha é uma estrutura de dados básica onde se realizam duas operações elementares: acrescentar um elemento no topo (**`push`**) e retirar um elemento do topo (**`pop`**). A operação `push` faz com que a pilha cresça, juntando-lhe uma nova posição de memória. `pop` faz com que a dimensão da pilha diminua, retirando-lhe uma posição.

As duas operações são ilustradas nos diagramas seguintes:



(a) Estado inicial **(b) Após `PUSH`** **(c) Seguido de `POP`**

A realização de operações sucessivas de `push` e `pop` fazem com que a pilha varie de dimensão como se fosse um harmónio (cresce e decresce).

Como vimos no início o MAC-1 contém três registos importantes para o programador: PC, AC e SP. Relembre que o SP (*Stack Pointer*) contém, em cada momento, a posição de memória correspondente ao topo do *stack*.

As duas instruções elementares relacionadas com a manipulação da pilha são:

Instrução	Descrição	Especificação
`push`	acrescenta uma posição ao topo da pilha; preenche essa posição com o conteúdo de AC	SP←SP-1; M[SP] ←AC
`pop`	retira a posição do topo da pilha; copia o respectivo conteúdo para AC;	AC←M[SP]; SP←SP+1

Note que no caso do processador MAC-1 a pilha cresce no sentido das posições de memória com endereços menores. Sendo assim, quando se faz PUSH, o topo da pilha passa a ser a posição com endereço anterior, ou seja, o *stack pointer* é decrementado. Do mesmo modo, quando se faz POP, o topo desce para a posição com endereço seguinte, ou seja, o *stack pointer* é incrementado.

Exemplo

Suponha que SP se encontra inicialmente com o valor 4015. Isso quer dizer que as posições actuais da pilha seriam de 4015 para baixo: 4015, 4016, 4017, etc. Ao fazer o PUSH acrescenta-se a posição 4014 à pilha; em correspondência, o SP passará a ter o valor 4015-1 = 4014. Posteriormente, o POP virá a retirar a posição 4014 passando o topo da pilha de novo para 4015; em correspondência o SP passará para 4014 + 1 = 4015.

4.3. Variáveis temporárias

A pilha pode (e deve !) ser utilizada para criar variáveis temporárias necessárias no decurso da execução do programa. Este conceito é ilustrado no seguinte pseudo-programa:

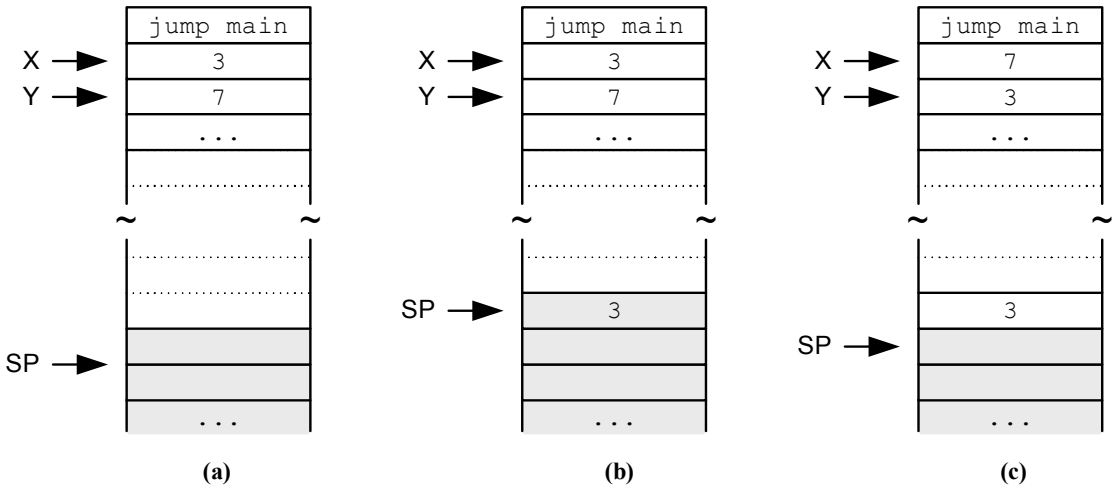
```
int x=3, y=7;
...
// trocar os valores de variáveis
{
      int tmp = x;      // criação da variável tmp
      x = y;
      y = tmp;
}                       // no fim do bloco a variável é destruída
```

O objectivo é trocar o valor de duas variáveis *x* e *y*. Para apoiar a troca é criada uma terceira variável temporária chamada *tmp*. Esta é criada quando é feita a sua declaração e é "destruída" no final do bloco entre chavetas.

O programa correspondente em assembly é o seguinte:

```
            jump MAIN
      x: 3
      y: 7

      MAIN: ...
            lodd x      # (a) AC = 3
            push        # (b) cria uma nova variável na pilha,
                        #     inicializada com o valor de AC
            lodd y
            stod x

            pop         # (c) elimina a variável criada na pilha,
                        #     passando o seu valor para AC
            stod y
            ...
```

As figuras seguintes ilustram a colocação na memória das variáveis X, Y e da variável T (enquanto existe) nos pontos (a), (b) e (c) assinalados no programa.



É importante perceber a questão do tempo de vida da variável *tmp*. Contrariamente às variáveis X e Y que existem (têm espaço na memória) desde que o programa é carregado na memória até terminar, a variável *tmp* só existe a partir do momento em que a instrução de criação da variável é executada (neste caso a instrução `push`). As variáveis X e Y ficam localizadas na zona de memória estática ao passo que a variável *tmp*, durante o período em que existe, fica localizada na pilha.

4.4. Endereçamento local.

As instruções **`push`** e **`pop`** permitem a forma mais básica de manipulação de variáveis criadas na pilha. Existem outras instruções que permite uma manipulação mais ampla.

Assim, a instrução **`desp n`** permite decrementar em `n` unidades o valor de SP, acrescentando desta forma um espaço na pilha para `n` novas variáveis (recorde que ao diminuir o valor de SP, o tamanho da pilha aumenta). As variáveis criadas desta maneira não são inicializadas (a instrução `push` cria uma só variável que é inicializada com o valor de AC).

A instrução **`insp n`** permite aumentar em `n` unidades o valor de SP, desta forma descartando as `n` variáveis colocadas no topo da pilha (ao aumentar o valor do SP diminui o tamanho da pilha). O valor das variáveis descartadas é perdido (a instrução `pop` descarta só uma variável do topo da pilha mas salvaguarda o respectivo valor em AC).

As instruções `desp` e `insp` que permitem criar e eliminar variáveis do pilha conjugam-se com as instruções **`stol`**, **`lodl`**, **`addl`** e **`subl`** que permitem manipular essas variáveis.

Estas novas instruções são semelhantes às suas congéneres terminadas em D: `stod`, `lodd`, `addd` e `subd`. A diferença entre os dois grupos de instruções é apenas o modo de endereçamento ou seja, a forma como mencionam a posição de memória envolvida na instrução.

Recorde que as instruções terminadas em D dizem-se de *endereçamento directo*: a instrução menciona explicitamente ("directamente") a posição de memória em jogo. Por exemplo, ao escrever

```
stod 10      # M[10] <- AC
```

estamos a dizer que a instrução envolve a posição de memória com endereço 10. O que a instrução faz é copiar o conteúdo de AC para a posição de memória 10.

As novas instruções, terminadas em L, dizem-se de ***endereçamento local***. Para estas instruções, a posição de memória é indicada dando a sua localização relativamente ao topo da pilha. Por exemplo, a instrução

```
stol 2       # M[SP + 2] <- AC
```

menciona a posição de memória que se situa duas posições abaixo do topo (ou seja, a posição SP+2, uma vez que o topo é a posição SP). Assim, o efeito da instrução é copiar o conteúdo de AC para a posição de memória SP+2.

O endereço indicado nestas instruções é um endereço relativo, desde o topo da pilha até à posição pretendida. O valor dado deverá ser maior ou igual a 0. Em particular, o deslocamento 0 refere a posição de topo da pilha. Por exemplo:

```
stol 0      # copia o valor de AC para o topo do stack
addl 0      # soma a AC o valor contido na posição do topo do stack
```

Note também que estas instruções de endereçamento local não modificam o valor de SP, ou seja, o topo da pilha mantém-se na mesma posição após a execução de qualquer uma das 4 instruções.

Em resumo as instruções directamente relacionadas com a pilha, para além de `push` e `pop`, são:

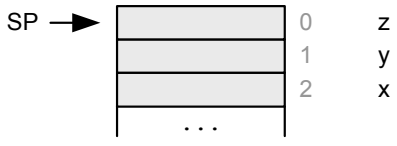
Instrução	Descrição	Especificação
`desp n`	acrescenta n posições no topo da pilha	SP ← SP-n
`insp n`	retira n posições do topo da pilha	SP ← SP+n
`lodl n`	copia para AC o conteúdo da posição n da pilha	AC ← M[SP+n]
`stol n`	copia o valor de AC para a posição n da pilha	M[SP+n] ← AC
`addl n`	adiciona o conteúdo da posição n da pilha a AC	AC ← AC + M[SP+n]
`subl n`	subtrai a AC o conteúdo da posição n da pilha	AC ← AC – M[SP+n]

Nesta descrição, a expressão *posição n da pilha* significa a posição de memória que se situa n posições abaixo da posição do topo da pilha ou seja, a posição de memória dada por SP+n.

Considere o seguinte programa ilustrativo. O programa usa as variáveis x, y e z que, por serem variáveis locais da função *main*, deverão ser criadas na pilha.

```
main() {
      int x, y, z;

      x = 4; y = 3;
      z = x + y;
}
```

Para criar as variáveis locais vamos usar a instrução `desp 3`, acrescentando três posições na pilha. Feito isso a pilha ficará como se indica na figura.



A variável *x* ficará localizada na posição 2 a partir do topo da pilha; a variável *y* na posição 1 e a variável *z* na posição 0. Assim, por exemplo, para obter a variável *x* faríamos `lodl 2`, para *y* faríamos `lodl 1` e para obter *z* faríamos `lodl 0`.

Nestas condições o programa *assembly* será o seguinte:

```
MAIN: desp 3            # int x, y, z;

      loco 4
      stol 2            # x = 4

      loco 3
      stol 1            # y = 3

      lodl 2
      addl 1
      stol 0            # z = x + y

      insp 3            # "destruir" as variáveis

      halt
```

Note ainda que, tratando-se de variáveis locais, criadas na pilha, deverão ser explicitamente destruídas quando a função terminar; é esse o propósito da instrução `insp 3` (ainda que neste caso isso não faça qualquer diferença, pois o programa termina imediatamente a seguir).

Repare que face a este grupo de instruções mais flexível, as instruções `push` e `pop` são redundantes. A instrução `push` cria uma nova variável na pilha, copiando o valor de AC para essa nova variável; o efeito é o mesmo que

```
desp 1      # acrescentar uma posição ao stack
stol 0      # copiar AC para a posição SP+0, ou seja o topo do stack
```

Por outro lado, a instrução `pop` descarta uma posição da pilha, depois de ter salvaguardado o respectivo valor no AC: o efeito é o mesmo que

```
lodl 0      # copiar a posição do topo do stack para o AC
insp 1      # retirar uma posição ao stack
```

Por exemplo, o código para trocar o conteúdo de *x* com o de *y* (apresentado na secção anterior) poderia ser escrito também na seguinte forma:

```
...
desp 1      # acrescentar uma posição ao stack - a variável tmp

lodd X
stol 0      # tmp = x

lodd y
stod x      # x = y

lodl 0
stod y      # y = tmp

insp 1      # retirar uma posição ao stack – destruir tmp
...
```

5. Rotinas

5.1. Instruções CALL e RETN

Considere o seguinte pseudo-programa que ilustra a chamada a uma função:

```
int x;

main() {
      x=1;
      func();     // (a) chamada da função func
      x=3;        // (c) retorno ao ponto de chamada
}

int func(){
      x=2;        // (b) execução da função func
}
```

O mecanismo de chamada ou invocação de funções já é por demais conhecido: no ponto (a) é chamada a função *func*. "Chamar a função" significa que o programa prossegue executando as instruções que constituem essa função - neste caso a instrução no ponto (b). Quando a função termina, o programa retorna ao ponto que se segue onde a função foi chamada – esse ponto designa-se por ***ponto ou endereço de retorno*** (c).

O programa correspondente em assembly é o seguinte:

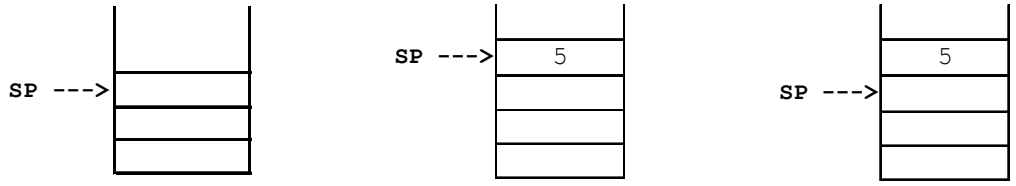
```
      jump MAIN

x:    0

MAIN: loco 1
      stod x            # x = 1

                        # chamar a função...
      call FUNC

      loco 3            # ...ponto de retorno da função
      stod x            # x = 3

      halt

FUNC: loco 2
      stod x            # x = 2
      retn              # retorno da função
```

As novas instruções *assembly* presentes neste exemplo são `call` e `retn`. A instrução `call` chama a função, ou seja faz com que o programa prossiga a execução a partir do ponto FUNC. A partir desse ponto executam-se as instruções da função FUNC, que termina com `retn`. Esta instrução `retn` faz com que o programa retorne ao ponto que se segue à chamada da função.

Há neste mecanismo uma questão crucial que é a memorização do ponto de retorno. Isto é feito com recurso à pilha: a instrução `call`, ao ser executada, memoriza o ponto de retorno na pilha;

em correspondência a instrução `retn` faz com que o programa retome ao ponto em que a função foi chamada.

A situação é ilustrada no seguinte diagrama que mostra a pilha antes, durante e depois da chamada à função:



Situação inicial | Após a chamada | Após o retorno

Quando a função é chamada memoriza-se o ponto de retorno – o processador faz automaticamente um PUSH ao conteúdo do *program counter* (incrementado) para o topo da pilha, guardando o endereço da posição onde o programa deve prosseguir após a função retornar. Esse ponto é justamente a posição a seguir à instrução `call FUNC` - neste caso a posição de memória 5, onde se encontra a instrução `loco 3`.

Por sua vez, a instrução `retn` extrai esse mesmo valor do topo da pilha (faz um pop) que usa para fazer o programa prosseguir, neste caso, a partir da posição de memória 5.

Note que, no conjunto, as instruções `call` e `retn` deixam o topo na mesma posição: a primeira aumenta uma posição; a segunda diminui uma posição. È também evidente que quando a instrução `retn` for executada deve encontrar a pilha na mesma posição onde o `call` o deixou; de outro modo, o `retn` não encontraria o endereço de retorno correcto e o programa teria um erro fatal, pois iria interpretar o valor lido da pilha como se fosse um endereço de retorno, quando de facto este não o era.

Em resumo as duas instruções envolvidas na chamada a uma função são:

Instrução	Descrição	Especificação
`call P`	chama a função colocada na memória a partir da posição P	SP ← SP-1; M[SP] ← PC+1; PC ← P
`retn`	retorna ao ponto de chamada da função	PC←M[SP]; SP←SP+1

É importante salientar o papel do registo PC nesta especificação. Relembre que PC é o registo que indica a posição da próxima instrução a ser executada. Ao ser feito `call`, na pilha é guardada a posição de retorno. Como a posição da instrução em curso (a instrução `call` em execução) é dada por PC, o ponto de retorno será a instrução seguinte, ou seja, o valor de PC+1; é este o valor que fica memorizado na pilha.

Quando o `retn` for executado esse valor será reposto no PC, fazendo com que o programa prossiga a partir desse endereço memorizado.

5.2. Colocação na memória

Uma função ou rotina é simplesmente um conjunto de instruções colocadas na memória. O que lhe dá o carácter de função é o facto da execução dessas instruções ser iniciada através de um `call` e terminar com um `retn`.

É importante distinguir a colocação do programa na memória e a sequência de execução. A função fica colocada na memória pela ordem de escrita, que é de alguma forma arbitrária. É indiferente que a função fique antes ou depois do "programa".

Por exemplo: o mesmo programa do ponto anterior poderia ser escrito:

```
      jump MAIN

x:    0

FUNC: loco 2
      stod x      # x = 2
      retn        # retorno da função

MAIN: loco 1
      stod x      # x = 1
      call FUNC
      loco 3
      stod x      # x = 3
      halt
```

ficando, neste caso, a função FUNC colocada na memória antes do resto do programa.

Embora situada em posições de memória diferentes, a ordem de execução pode-se ler praticamente da mesma forma em ambos os casos: o programa começa na primeira instrução; aí há um salto que o faz seguir para a posição MAIN; quando encontrar a instrução `call FUNC` prossegue a partir da posição FUNC (memorizando o ponto de retorno); depois disso quando encontrar a instrução `retn` volta ao ponto de retorno memorizado.

5.3. Variáveis locais, argumentos e valor devolvido

As funções podem usar variáveis locais, como se ilustra no seguinte exemplo:

```
int x=1;

main() {
      int y;
      y = x+1;    // (1)
      func();
      y = y+1;    // (3)
}

int func() {
      int z;
      z = x+3;    // (2)
}
```

O pseudo programa anterior ilustra os dois tipos de variáveis com que vamos trabalhar: ***variáveis estáticas*** (externas ou globais), como é o caso da variável x e ***variáveis locais*** como as chamadas y e z.

Como já foi visto anteriormente, as variáveis estáticas existem (i.e., têm local definido na memória) logo que o programa é carregado para execução, ficando colocadas na zona de variáveis estáticas, num bloco inicial de memória.

Ao contrário as variáveis locais são criadas, na pilha, durante a execução do programa. Mais especificamente, são criadas quando a função se inicia e descartadas quando a função termina - ou seja, só existem enquanto a função a que pertencem está a ser executada.

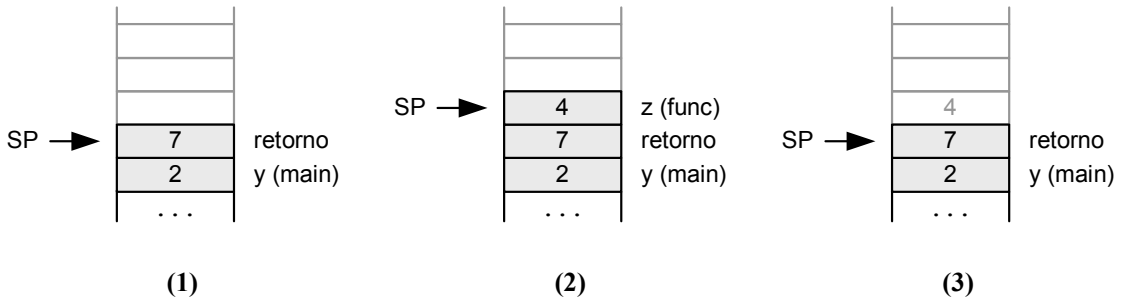
No exemplo apresentado temos a seguinte situação:

- no ponto (1) existem duas variáveis: a variável estática x e a variável local y;
- no ponto (2) existe, adicionalmente, a variável z, local da função func;
- no ponto (3) voltamos a ter as duas variáveis x e y.

O programa correspondente em *assembly* é o seguinte:

```
        jump MAIN

x:      1

MAIN:   desp 1              # int y (criar y);
        loco 1
        addd x
        stol 0              # y = x + 1
        call FUNC
        loco 1
        addl 0
        stol 0              # y = y + 1
        insp 1              # destruir a variável y
        halt

FUNC:                                                   <--(1)
        desp 1              # int z (criar z)
        loco 3
        addd x
        stol 0                                          <--(2)
        insp 1              # destruir a variável z
        retn                                            <--(3)
```

As variáveis foram criadas na pilha utilizando a instrução `desp` e eliminadas usando `insp`. As figuras seguintes ilustram o conteúdo da pilha nos pontos (1), (2) e (3).



(1) **(2)** **(3)**

O ponto (1) indica a situação à entrada (ou seja imediatamente a seguir à chamada) da função *func* – a chamada acrescentou o endereço de retorno na pilha, por cima da posição previamente reservada para a variável local y da função *main*.

No ponto (2) foi criada a variável local z – esta variável local fica localizada na pilha depois (i.e., por cima) do endereço de retorno.

No ponto (3) a variável local z já foi destruída – ou seja, a pilha está na mesma posição que estava no ponto de entrada da função (1).

O exemplo ilustra as regras gerais em matéria das variáveis locais de uma função:

- na entrada de uma função o topo da pilha contém o endereço de retorno;
- as variáveis locais são criadas na pilha depois (em cima) do endereço de retorno;
- todas as variáveis assim criadas devem ser destruídas antes do fim da função, de forma que ao executar a instrução `retn` se encontre de novo, no topo da pilha, o endereço de retorno;

As funções caracterizam-se ainda por mais dois aspectos relevantes: os seus ***argumentos*** e o ***valor devolvido***. Estes aspectos são ilustrados no pseudo-programa seguinte:

```
int x = 2;

main() {
      int s;
      s = soma( 3, x);
      ...
}

int soma ( int a, int b ) {
      int z;
      z = a + b;
      return z;
}
```

No exemplo apresentado, a função *soma* recebe dois argumentos e devolve como resultado a soma desses mesmos argumentos.

A regra relativamente aos argumentos é a seguinte: antes de se chamar a função colocam-se os argumentos na pilha. Neste caso são colocados na pilha os valores 3 (uma constante) e 2 (valor da variável estática *x*).

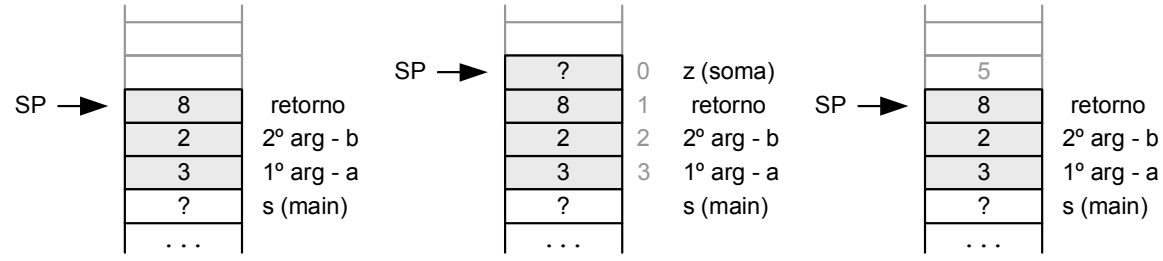
Por sua vez, a função recebe estes argumentos, que usa como se fossem variáveis locais comuns. Neste caso usa os argumentos para calcular a soma que devolve quando ao fazer `return`.

O programa correspondente em *assembly* é o seguinte:

```
      jump MAIN

x:    2

MAIN: desp 1      # criação da variável s

      loco 3      # preparar os argumentos
      push        # 1º argumento
      lodd x
      push        # 2º argumento
      call SOMA
      insp 2      # destruir os argumentos

      stol 0      # guardar o valor devolvido em s
      ...
      insp 1      # destruir s

      halt

SOMA:                                          <--(1)
      desp 1      # criar a variável z          <--(2)

      lodl 3      # obter a
      addl 2      # somar b
      stol 0      # z = a + b

      insp 1      # destruir z
      retn                                     <--(3)
```

O primeiro argumento é preparado pela sequência: `loco 3` seguido de `push`, que coloca o valor 3 na pilha. O segundo argumento é preparado pela sequência `lodd x` e `push` que copia o valor da variável x para a pilha.

As seguintes figuras ilustram a pilha nos pontos notáveis (1), (2) e (3).



(1) **(2)** **(3)**

O ponto (1) é o ponto de entrada na função. Como sempre, no ponto de entrada de uma função tem-se o endereço de retorno no topo da pilha. Os argumentos estão localizados atrás do endereço de retorno.

De seguida, são criadas as variáveis locais da função. Neste caso é criada a variável z, levando à situação ilustrada no ponto (2) – relembre que as variáveis locais ficam à frente do endereço de retorno.

Tanto os argumentos como as variáveis locais da função são acedidas através de endereçamento local. É evidente que os seus endereços relativos têm que se escrever tendo em conta a situação da pilha nesse momento. Neste caso, para escrever o corpo da função interessa a situação ilustrada no ponto (2) onde :

- o argumento a tem o endereço local 3
- o argumento b tem o endereço local 2
- a variável local z tem o endereço local 0

Face a esta situação temos que, por exemplo:

`lodl 3` refere-se ao argumento a
`addl 2` refere-se ao argumento b
`stol 0` refere-se à variável z

Antes de ser feito o retorno, deve ser reposta a situação da entrada na função, eliminando a variável local criada. Assim, na altura em que vai ser executada a instrução `retn` está-se na situação (3) em que temos, de novo, o endereço de retorno no topo da pilha.

Voltando ao ponto de chamada, após o `call`, permanecem na pilha os argumentos preparados. Esses argumentos são descartados pela instrução seguinte `insp 2`. Depois disto a pilha fica tal e qual como estava antes de se começar o processo de chamada à função.

Estes mecanismos traduzem um princípio universal: no final de uma função a pilha tem que estar igual ao que estava no ponto de entrada. Vimos que é assim relativamente à função soma; deve também ser assim relativamente ao corpo do programa: imediatamente antes do fim a pilha deve estar como estava no princípio, ou seja, vazia.

Resta ver o valor devolvido pela função: neste aspecto vamos convencionar que a função devolve o valor no AC. Ou seja: antes de terminar a função deve-se escrever em AC o valor que pretende devolver.

Os exemplos deste ponto ilustram as convenções relativas a argumentos e valor de retorno de uma função. Em resumo temos:

- Do lado que invoca a função:
  - colocar os argumentos na pilha, antes de se chamar a função;
  - chamar a função, fazendo o call;
  - descartar da pilha os argumentos que foram passados à função (no ponto de chamada);

- "Dentro" da função:
  - criar as variáveis locais;
  - escrever o código;
  - colocar em AC o resultado a devolver;
  - eliminar as variáveis locais;
  - retornar, fazendo retn;

Convém respeitar sempre estas "regras"...

Exemplos

Neste exemplo usa-se a função *soma* para somar 3 números 3+7+13.

```
MAIN: loco 3      # preparar os argumentos
      push        # 1º argumento
      loco 7
      push        # 2º argumento
      call SOMA
      insp 2      # destruir os argumentos
      push        # 1º argumento(o resultado da chamada anterior
                  # que ainda está no AC)
      loco 13
      push
      call SOMA   # AC fica com 3+7+13
      insp 2
      halt
```

Codificar a seguinte função que recebe dois argumentos e devolve o maior

```
int maior( int x, int y ) {
    int z = x;
    if ( y > x )
        z = y;
    return z;
}
```

```
MAIOR:
      desp 1
      lodl 3
      stol 0      # criação e inicialização de z com o valor de x

      subl 2
      jpos RET    # se x-y >= 0 saltar para R

      lodl 2
      stol 0      # z = y;

RET:  lodl 0      # valor de retorno AC = z
      insp 1      # destruir z
      retn
```

Outra versão equivalente, mas que utiliza `push` e `pop` para criar e destruir z:

```
MAIOR:
      lodl 2      # atenção: aqui x ainda está na posição 2 !
      push        # criar z - a partir daqui o stack altera:
                  # x passa a estar na posição 3 e y na posição 2 !

      subl 2
      jpos R

      lodl 2
      stol 0

R:    pop         # descartar a variável e pôr o valor de retorno em AC
      retn
```

Multiplicação de dois números *a* e *b* positivos:

```
int mult ( int a, int b) {
      int m = 0, i = 0;

      for (i=0; i!=b; ++i)
            m = m + a;
}

MULT: loco 0      # int m, i = 0
      push
      push

CICLO:
      lodl 0
      subl 3
      jzer RET    # se i==b o ciclo termina - salta para RET

      lodl 1
      addl 4
      stol 1      # m = m + a


      loco 1
      addl 0
      stol 0      # i = i + 1

      jump CICLO

RET:  lodl 1      # resultado (m) em AC
      insp 2
      retn
```

5.4. Ainda sobre os argumentos

Os argumentos de uma função são tratados como variáveis locais. Esta é aliás uma situação normal na generalidade das linguagens – no entanto existe uma particularidade que é a inicialização dos argumentos do lado que chama a função.

Acontece que a uma função pode modificar os valores dos seus argumentos. E, uma vez que os argumentos ainda estão na pilha depois da função retornar, essas modificações podem ser recuperadas no ponto de chamada da função. Ou seja: no MAC-1 os argumentos de uma função são, por construção, de leitura/escrita (modificáveis pela função).

Esta particularidade pode ser útil, por exemplo, em situações que se pretendam devolver múltiplos valores – para tal bastaria que o lado que chama a função reservasse na pilha um conjunto de posições que seriam modificados pela função, e que poderiam depois ser utilizados como se fossem valores devolvidos.

O seguinte exemplo em *assembly* ilustra esse conceito. O programa chama uma função *f* que aceita dois argumentos. A função faz alterações aos argumentos, alterações essas que são recuperadas pela função *main* que as usa para atribuir novos valores a *x* e a *y*. No fim do programa tem-se que x=-4 e y=10.

```
      jump MAIN

x: 3
y: 7

MAIN: lodd x
      push        # 1° argumento a = x
      lodd y
      push        # 2° argumento b = y
      call F
      pop
      stod y      # y = b
      pop
      stod x      # x = a
      halt

F:    lodl 2
      subl 1
      push        # int tmp = a - b

      lodl 2
      addl 3
      stol 2      # b = b + a

      pop
      stol 2      # a = tmp
      retn
```

6. Matrizes

6.1. Noções básicas

O último mecanismo basilar de programação que vamos abordar são as matrizes (ou, se preferir, vectores ou *arrays*). O seguinte exemplo ilustra a utilização de uma matriz:

```
int i=0, A[5];

main() {
    for ( i = 0; i < 5; i ++ )
        A[i] = i+1;
}
```

Em termos de espaço ocupado na memória, uma matriz não é mais do que um conjunto de variáveis colocadas contiguamente na memória. Neste caso podemos dizer que o programa tem 6 variáveis: a variável chamada *i* mais as 5 variáveis associadas à matriz *A*.

A colocação em memória dessas variáveis resulta no seguinte:

```
      jump MAIN

i:    0                 # variável i

A:    0                 # primeiro elemento da
      0                 # segundo elemento do array
      0
      0
      0

MAIN: ...               # programa...
```

A variável *i* fica colocada na posição de memória 1. A matriz *A* fica colocada nas posições de memória 2 a 6, inclusive.

6.2. Indexação

O interesse particular das matrizes é a possibilidade de indexar os seus elementos. Usando esta possibilidade podemos designar as 5 variáveis que compõem a matriz por A[0], A[1] e assim sucessivamente até A[4]. Desta forma podemos aplicar à matriz procedimentos iterativos – como o que se vê no exemplo em que se percorrem as 5 variáveis que constituem a matriz através de um ciclo for, usando a variável de contagem i para indexar as posições sucessivas.

É importante perceber que a expressão A[i] não é muito mais do que uma conta para obter uma posição na memória. Efectivamente, sendo A a posição de memória onde a matriz começa (isto é, onde se situa a primeira das suas variáveis) dizer A[i] não é muito mais do que dizer "a posição de memória A+i". Isto é verdade na medida em que as 5 variáveis que constituem a matriz se situam em posições contíguas na memória; sendo assim, se a primeira está na posição A, a segunda está na posição A+1, a terceira na posição A+2, etc.

Para aceder a um elemento da matriz, seja para ler seja para escrever, é necessário em primeiro lugar calcular a posição de memória (endereço) onde esse elemento se encontra.

O ponto de partida este efeito é obter o endereço do primeiro elemento, neste caso o endereço 2 (ou A). Para colocar este endereço no AC podemos fazer:

```
loco 2      # coloca o endereço do elemento A[0] no AC
```

ou

```
loco A      # coloca o endereço do elemento A[0] no AC
```

A partir daqui, para calcular o endereço da posição de memória onde se encontra o elemento *i* basta somar *i*. Ou seja, para obter o endereço de A[i] basta fazer:

```
loco A
addd i      # coloca o endereço do elemento A[i] no AC
```

6.3. Instruções PSHI e POPI

Vamos chamar "ler" e "escrever" às operações que se fazem sobre um elemento de uma matriz: ler é obter o valor que se encontra nessa posição; escrever é alterar o valor que se encontra nessa posição. O primeiro caso corresponde a operações do género:

$$x = A[i] \text{ ou } x = x + A[i]$$

o segundo caso corresponde a operações do género

$$A[i] = 1 \text{ ou } A[i] = x+1.$$

Se tivermos algo semelhante a:

$$A[i] = A[i] + 1$$

estão envolvidas as duas operações: primeiro ler o valor que está na posição *i* da matriz A, incrementar o valor lido e depois escrever o resultado nessa mesma posição.

Posto isto, a manobra para ler um elemento de uma matriz é:

1. colocar em AC o endereço desse elemento
2. executar a instrução **`pshi`**
3. extrair o valor lido do topo da pilha

A instrução **`pshi`** parte do princípio que AC contem o endereço da posição de memória onde se encontra o elemento da matriz. O que a instrução faz é um "push" do valor que se encontra nessa posição para o topo da pilha (o valor de A[i] é copiado para o *stack*). A partir daí o programa pode prosseguir extraindo o valor pretendido do topo da pilha, por exemplo com `pop`.

Em suma, para implementar uma operação do género x=A[i] faz-se

```
loco A
addd i      # guardar em AC o endereço do elemento A[i]

pshi        # ler o respectivo valor para o stack

pop         # extrair do stack e ...

stod x      # ... guardar nalgum lado, ou fazer alguma coisa com ele
```

Por outro lado, a manobra para escrever um valor na posição *i* de uma matriz é:
1. colocar no topo da pilha o valor que se pretende escrever na matriz;
2. colocar em AC o endereço da posição onde se pretende escrever o valor
3. executar a instrução **`popi`**

A mecânica é ilustrada pela seguinte sequência que implementa A[i] = x.

```
lodd x      # no topo do stack, preparar o valor a transferir
push        # para a matriz

loco A
addd i      # colocar em AC o endereço de A[i]

popi        # transferir o valor do topo do stack para
            # a posição de memória indicada em AC
...
```

Exemplos

Considere o seguinte exemplo, que soma o valor dos elementos de uma matriz inicializada:

```
int i, A[5] = { 2, 3, 7, 1, 2}, S=0;

main()
{
    for ( i = 0; i < 5; i ++ )
        S = S + A[i];
}
```

O programa *assembly* pode ser o seguinte:

```
      jump MAIN

i:    0

A:    2
      3
      7
      1
      2

S:    0
```

```
MAIN: loco A
      addd i
      pshi        # tmp = a[i]

      pop
      addd S
      stod S      # S = S + tmp

      loco 1
      addd i
      stod i      # i = 1 + i

      loco 5
      subd i
      jnze MAIN   # se i != 5 salta para MAIN

      halt
```

6.4. Utilização de strings

O *assembler* que vamos usar permite definir matrizes de caracteres (ou *strings*) de uma forma mais compacta usando a directiva .string do seguinte modo:

```
A:    .string "aaaxaaa"
```

Neste caso, a directiva *.string* permite a inicialização de uma matriz A com 7 posições preenchidas com o código ASCII dos caracteres indicados.

Exemplo

O seguinte programa localiza a posição onde se encontra o caracter 'x'. A ideia corresponde ao seguinte programa:

```
      char A[7] = "aaaxaaa";
      int n = -1

      main ()
      {
            for (int i = 0; i < 7; ++i)
                  if ( A[i] == 'x' )
                        n = i;
      }
```

que se pode traduzir para *assembly* do seguinte modo:

```
      jump MAIN

A:    .string "aaaxaaa"

CHR_x:'x'
n:    -1
```

```
MAIN: loco 0
      push              # int i = 0

CICLO:
      loco 7
      subl 0
      jzer FIM          #se i == 7 é porque chegou ao fim da matriz

      loco A
      addl 0
      pshi
      pop               # ler A[i] e carregar no acumulador

      subd CHR_X
      jzer MATCH        # se A[i] == 'x' salta para MATCH

      loco 1
      addl 0
      stol 0            # i = 1 + i

      jump CICLO

MATCH:
      lodl 0
      stod n            # n = i

FIM:  halt
```

Para comparar com o caracter 'x' usamos outra das possibilidades do *assembler* que é poder escrever o caracter na notação usual. Como é evidente, isso equivale a escrever o respectivo código ASCII, mas de forma muito mais legível.

7. I/O

7.1. Output – saída para o ecrã

O processador MAC não tem instruções específicas de entrada e saída. Entretanto o simulador implementa para este efeito um mecanismo que se designa por *memory-mapped IO*. Este mecanismo faz corresponder um dispositivo de I/O a um endereço de memória comum. Assim, por exemplo, ao escrever para determinado endereço estamos a realidade escrever num dispositivo de saída.

É isto que acontece, no simulador que vamos usar, quando se escreve para a posição de memória 4094. O seguinte exemplo ilustra a escrita de uma palavra para o ecrã:

```
STDOUT=4094
...
loco 65
stod STDOUT
...
```

Na saída do simulador o valor escrito para a memória é interpretado como um caracter ASCII. Deste modo o que se vê aparecer como resultado no ecrã é a letra A (maiúscula), cujo código ASCII é o valor 65.

Para simplificar a programação, o *assembler* utilizado permite uma especificação mais simples de códigos ASCII, colocando-se para tal o caracter pretendido entre pelicas. Por exemplo, caso o valor do código ASCII do caracter A, uma alternativa a escrever 65 seria escrever 'A'. O seguinte exemplo pretende ilustrar esta possibilidade:

```
...
loco 'A'          # carrega o código ASCII de A para o acumulador
stod STDOUT       # aparece a letra A no ecrã
...
```

Seguindo a mesma linha de raciocínio, para aparecer um número no ecrã é preciso enviar o respectivo caracter ASCII. Por exemplo, para escrever o número 7 é pode-se escrever:

```
loco '7'
stod STDOUT
```

De forma mais geral, para escrever o número *i* (entre 0 e 9) podemos fazer:

```
STDOUT=4094
...
loco '0'
addd i
stod STDOUT
...
```

uma vez que '0'+i é o valor ASCII do algarismo i.

Também se de pode aplicar o mesmo raciocínio para escrever a i-ésima letra a partir de A (ordem alfabética). Por exemplo:

```
...
loco 'A'
addd i
stod STDOUT
...
```

Por exemplo, se *i* contem o valor 5, aparece no ecrã a letra F.

7.2. Input – Leitura de caracteres do teclado

(secção por fazer)

8. Recursividade

8.1. Introdução à recursividade

Considere o seguinte pseudo-programa:

```
int n = 0, k=0;

main()
{
      f(4);
}

void f( int i )
{
      n = n + 1;
      if ( i > 0 )
            f( i-1 );
      k = k + 1;
      return;
}
```

Dizemos que uma função deste género é recursiva porque se chama a si própria.

É importante realçar que *fazer uma chamada à função f dentro da função f* é semelhante a fazer uma chamada a outra função qualquer. Para nos entendermos, vamos considerar individualmente cada uma das chamadas à função *f* distinguindo essas mesmas chamadas pelo valor do argumento *i*. Assim, primeiro é chamada f(4), depois f(3) e assim sucessivamente até ser chamada f(0).

A sequência da execução do programa é ilustrada no seguinte traçado:

```
f(4)
      n=1
      f(3)
            n=2
            f(2)
                  n=3
                  f(1)
                        n=4
                        f(0)
                              n=5
                              k=1
                              return
                        k=2
                        return
                  k=3
                  return
            k=4
            return
      k=5
      return
fim
```

A partir do main é chamada a função f(4). Esta por sua vez chama a função f(3). Evidentemente quando f(3) terminar volta ao ponto de chamada ou seja f(4).

Na sequência completa: f(4) chama f(3) que chama f(2) e assim sucessivamente até f(0). Até este ponto nenhuma das funções fez ainda return. Depois f(0) retorna para f(1) que recomeça a execução e termina com return para f(2), que prossegue a sua execução e termina com return para f(3) e assim sucessivamente até que f(4) vai finalmente retornar para main.

Esta chamada encadeada faz com que as instruções "n=n+1" sejam todas feitas antes das instruções k=k+1. Efectivamente, n=n+1 é executada antes de se chamar a função seguinte, e k=k+1 já só é executada depois de todas as funções encadeadas terem sido executadas.

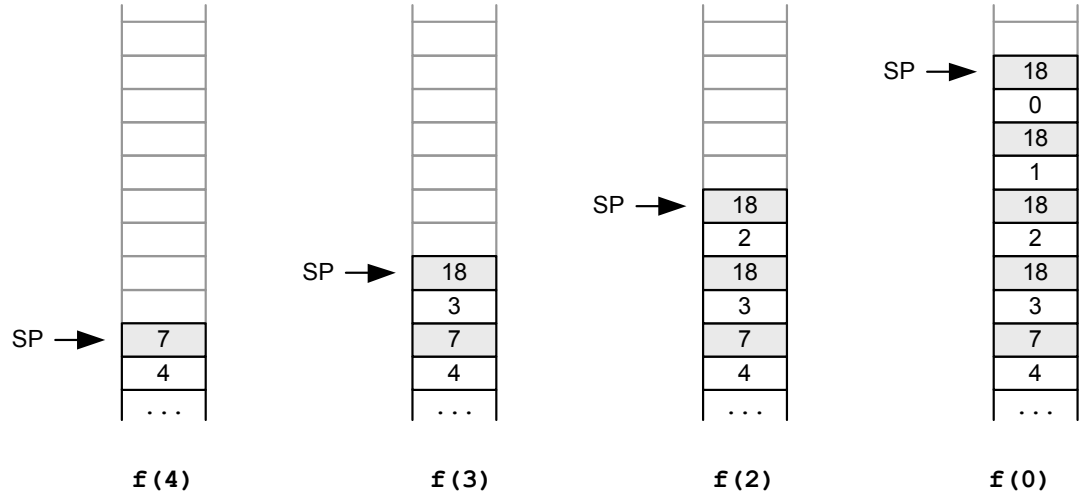
A tradução do programa anterior para *assembly* pode ser a seguinte:

```
      jump MAIN

UM:   1
N:    0
K:    0

MAIN: loco 4
      push
      call F            # f(4)
      insp 1
      halt

##################

F:    lodd N            # n = n + 1
      addd UM
      stod N

      lodl 1            # se i <= 0 salta para SKIP
      jzer SKIP
      jneg SKIP

      subd UM
      push
      call F            # f(i-1)
      insp 1

SKIP: lodd K
      addd UM           # k = k + 1
      stod K

FIM:  retn

##################
```

Há medida que a função *f* vai sendo sucessivamente chamada, a pilha vai crescendo para ir guardando o endereço de retorno e o valor do argumento *i*.

As figuras seguintes representam a pilha após o programa chamar f(4), f(3), f(2) e f(0). Em todos os casos, excepto para a primeira chamada – f(4) –, o endereço de retorno é o mesmo (vale 18), mas o valor do argumento é obviamente diferente. A pilha atinge a sua dimensão máxima quando é chamada f(0). Nesta altura temos na pilha os endereços de retorno e os argumentos relativos a todas as chamadas encadeadas feitas até ao momento. A partir daí e há

medida que começam a ser feitos os retornos (executada a instrução `retn`), o tamanho da pilha começa a diminuir.



8.2. Desenvolvimento de funções recursivas

A implementação de funções recursivas corresponde a uma forma de raciocínio "natural" para resolução de determinados problemas. Exemplos clássicos de problemas deste género são a "soma dos N primeiros números naturais" ou o "factorial de N".

Consideremos a soma dos primeiros n números. A versão iterativa corresponde ao seguinte raciocínio:

```
int soma(int n)
{
      int s=0;
      for ( i=0; i<=n ; i++)
            s = s + i;
      return s;
}
```

Por outro lado, também se poderia definir a soma – S – dos n primeiros números do seguinte modo :

$$S(n) = \begin{cases} n + S(n-1) & \text{se } n > 0 \\ 0 & \text{se } n = 0 \end{cases}$$

Esta expressão diz-nos, por exemplo, que a soma dos primeiros 4 números é dada por:

$$\begin{aligned} S(4) &= 4 + S(3) \\ &= 4 + 3 + S(2) \\ &= 4 + 3 + 2 + S(1) \\ &= 4 + 3 + 2 + 1 + S(0) \\ &= 4 + 3 + 2 + 1 + 0 \end{aligned}$$

Como vê, é o resultado que se pretende, mas calculado de forma recursiva – para calcular S(4) é necessário calcular primeiro S(3) , que por sua vez exige S(2) e assim sucessivamente até se chegar a S(0) que vale 0.

Traduzindo este raciocínio para um pseudo-programa, pode-se escrever o seguinte código:

```
int soma( int n )
{
      if ( n == 0)
            return 0;
      return n + soma(n-1);
}
```

Em *assembly* chega-se então à seguinte função *soma* recursiva:

```
...
UM:   1
...

SOMA: lodl 1
      jzer RET    # se n = 0 retorna devolvendo 0...

      subd UM     # ...caso contrário, chama soma(n-1)
      push
      call SOMA
      insp 1      # no acumulador está o resultado de soma(n-1)

      addl 1      # adiciona n ...

RET:  retn        # ... e retorna, devolvendo o n + soma(n-1) em AC
```

9. Lista de revisões

Versão	Autor	Data	Comentários
0.01	João Baptista	Dez/2004	Versão *draft* inicial; Publicação antecipada para apoio ao trabalho 2004/2005.
0.01a	João Baptista	Jan/2005	Correcção de gralhas no texto.
0.02	Tomás Brandão	Abr/2005	Revisão e correcção de gralhas; Formatação e re-estruturação do texto; Introdução de texto complementar..
0.02a	Tomás Brandão	Jul/2005	Correcção de gralhas no texto.

Para elaborar a curto prazo:

- Texto mais aprofundado para a arquitectura do processador MAC-1, salientando:
  - Datapath (simplificado e completo)
  - Unidade de controlo
  - Micro-instruções necessárias para executar algumas instruções

- Fazer a secção 7.2 (input de dados a partir do teclado)

- Melhorar o texto e acrescentar mais exemplos de funções recursivas recursivas.

Anexo A – Lista de instruções

	Instrução	Descrição	Significado
Endereçamento directo	`lodd x`	AC = M[x]	load direct
	`stod x`	M[x] = AC	store direct
	`addd x`	AC = AC + M[x]	add direct
	`subd x`	AC = AC – M[x]	subtract direct
Endereçamento imediato	`loco x`	AC = x	load constant
Saltos	`jump x`	PC = x	inconditional jump
	`jzer x`	if (AC==0) PC = x	jump if zero
	`jnze x`	if (AC!=0) PC = x	jump if nonzero
	`jpos x`	if (AC>=0) PC = x	jump if positive
	`jneg x`	if (AC<0) PC = x	jump if negative
Endereçamento local	`lodl n`	AC = M[SP + n]	load local
	`stol n`	M[SP + n] = AC	store local
	`addl n`	AC = AC + M[SP + n]	add local
	`subl n`	AC = AC – M[SP + n]	subtract local
Procedimentos	`call x`	SP = SP – 1; M[SP] = PC; PC = x	call procedure
	`retn`	PC = M[SP]; SP = SP + 1	return
Pilha	`push`	SP = SP – 1; M[SP] = AC	push onto stack
	`pop`	AC = M[SP]; SP = SP + 1	pop from stack
	`desp n`	SP = SP – n	decrement SP
	`insp n`	SP = SP + n	increment SP
Endereçamento indirecto	`pshi`	SP = SP – 1; M[SP] = M[AC]	push indirect
	`popi`	M[AC] = M[SP]; SP = SP + 1	pop indirect
Terminar	`halt`	---	stop execution