

# **ELE 8575**



**CONJUNTO DE INSTRUÇÕES DO x86**

**Slides baseados no material do curso do  
Prof. Chen Lian Kuan**

# Programando 8088/8086

- Níveis de linguagens disponíveis para programar um microprocessador:  
**Linguagem de máquina, Linguagem de montagem e Linguagem de alto nível.**
- Linguagem de máquina: Uma sequência de códigos binários para a instrução a ser executada por microcomputadores. Cadeias de bits longas podem ser simplificados usando o formato hexadecimal. É difícil programar estando, portanto, propenso a erros. Diferentes  $\mu P$  usam diferentes códigos de máquina.

## Exemplo:

### Machine Code

Memory Address	Contents (binary)	Contents (hex)	Operation
00100H	11100100	E4	INPUT (IN)
00101H	00000101	05	Port 05H
00102H	00000100	04	ADD
00103H	00000111	07	07H
00104H	11100110	E6	OUTPUT (OUT)
00105H	00000010	02	PORT 02

# Assemblers

- A programação das instruções diretamente em código de máquina é possível, mas consome muito tempo, pois cada instrução básica pode ter um dos vários códigos de máquina diferentes, dependendo de como os dados são armazenados.
- O processo de conversão das instruções do microprocessador para o código binário da máquina pode ser executado automaticamente por um programa de computador, chamado de **ASSEMBLER**.
- A maioria dos assemblers aceita um arquivo-texto de entrada contendo linhas com uma sintaxe rigorosamente definida, dividida em quatro campos.

Label	Mnemonic/directive	Operands	comment
AQUI:	ADD	AL,0FH	; Adds 0FH to register AL

Nem todos os campos precisam estar presentes em uma linha. Por exemplo, uma linha pode ser apenas uma linha de comentário se começar com ponto e vírgula.

# Program Trapping (Tracing)

- A maioria dos microprocessadores tem um modo de operação que permite que um programa seja interrompido, após cada instrução, e a execução de outro programa (programa de depuração) para permitir a depuração do programa (examinando a operação de um programa para encontrar erros [bugs]).
- O modo de depuração é inserido ajustando a *flag* T (*trap*) no registro FLAGS do Intel 80x86
- Métodos típicos de debug
  - **Single stepping:** examina-se o conteúdo do registro e da memória após cada instrução (demorado)
  - **Program tracing:** encontra-se a sequência de instruções sendo executadas pelo programa
  - **Breakpoints:** interrompe-se um programa em um ponto decidido pelo programador para permitir o exame dos registros) - INT3
  - **Memory dump:** faz-se o dumping do conteúdo de memória

# Campos da Linguagem Assembler

<label> <Mnemonic or directive> <operands> <;comment>

Comment field contém documentação interna do programa para melhorar a legibilidade humana – recomenda-se o uso de comentários significativos

Operand field contém dados ou endereço utilizados pela instrução.

As seguintes convenções são normalmente aplicáveis (Suponha DS=A000H):

MOV AX, [10H] ; Carrega word no endereço de memória A0010H, em  
; AX

Mnemônico	Destino	Fonte (offset 10H)
-----------	---------	--------------------

# Campos da Linguagem Assembler

<label> <Mnemonic or directive> <operands> <;comment>

Mnemonic/directive field contém a abreviação de instrução do processador (por exemplo, MOV) ou uma DIRETIVA do assembler. Uma diretiva não produz código objeto, mas é usada para controlar como o assembler opera.

## Exemplo de diretiva

VALOR EQU 1000H

- define "VALOR" como o número 1000H

Label field contém uma etiqueta que recebe um valor igual ao endereço onde a etiqueta aparece.

# Assembly Language Programs

Escrever programas em Assembly levam mais tempo do que em linguagem de alto nível.

**Exemplo: (somar 2 números de 32 bits usando-se processadores de 16 bits)**

Dois números inteiros longos podem ser adicionados em linguagem C por

$$X=Y+Z;$$

Em Assembly x86 fica:

```
MOV AX, [Y]
MOV DX [Y+2]
ADD AX, [Z]
ADC DX, [Z+2]
MOV [X], AX
MOV [X+2], DX
```

# Conjunto de instruções Intel 8088/8086 - Visão geral

- O Intel 8088 tem noventa instruções básicas (ou seja, sem contar as variantes do modo de endereçamento)
  - As instruções pertencem a um dos seguintes grupos:  
transferência de dados, aritmética, lógica, manipulação de strings, transferência de controle e controle de processador.
- (data transfer, arithmetic, logic, string manipulation, control transfer and processor control.)



# Conjunto de Instruções

- Os microprocessadores podem realizar uma gama de operações básicas definidas por seu conjunto de instruções.
- O conjunto de instruções é um conjunto de códigos binários conhecidos como **op-codes** que podem ser decodificados pela unidade de controle do microprocessador.
- Os **op-codes** são frequentemente combinados com algumas informações de endereço para especificar a localização dos operandos (os dados para a instrução).
- Exemplo de **op-code** e sua codificação em código de máquina

Instruction

IN AL,05h

ou

Mov dx, 05h

in al, dx

machine code

E405

BA0500

→

EC

# Conjunto de Instruções

- O número total de instruções em um conjunto de instruções é limitado pelo número de bits possíveis disponíveis para codificar o **op-code**.
- Normalmente, nem todos os bits de uma palavra são usados para o **op-code**, uma vez que alguns bits são necessários para a informação de endereçamento. Este limite pode ser contornado usando 2,3 ou mais palavras para as instruções (mais lento, pois são necessários mais ciclos de leitura da memória).
- Na arquitetura do 8088/8086 **CISC** (*Complex Instruction Set Computer*), diferentemente da arquitetura **RISC** - *Reduced Instruction Set Computer*, o número de bytes que compõe o **op-code** pode variar de 1 a 6 bytes
- O formato de uma instrução típica do 8086/8088 pode ser visto abaixo:

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
Op-code	Mod Reg R/M	LSBy deslocamento	MSBy deslocamento	LSBy dados	MSBy dados

Na figura, LSBy: *Least Significant Byte* MSBy: *Most Significant Byte*

# Intel 8088: Formato de instrução de código de máquina

- O código da máquina para uma instrução consiste em um código binário de comprimento variável (normalmente 2 bytes, mas pode ser de 1 byte a 6 bytes, dependendo da instrução). O código da máquina inclui todas as informações necessárias (op-code, endereço, dados) para uma instrução
- Os códigos de máquina para o Intel 8088 estão listados no *datasheet* do Microprocessador Intel.
- Exemplo : Instruções MOV e JMP

1	Mnemonic and Description	Instruction Code			
	<b>DATA TRANSFER</b>				
	<b>MOV = Move:</b>				
Register/Memory to/from Register		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
		1 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory		1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate to Register		1 0 1 1 w reg	data	data if w = 1	
Memory to Accumulator		1 0 1 0 0 0 0 w	addr-low	addr-high	
Accumulator to Memory		1 0 1 0 0 0 1 w	addr-low	addr-high	
Register/Memory to Segment Register		1 0 0 0 1 1 1 0	mod 0 reg r/m		
Segment Register to Register/Memory		1 0 0 0 1 1 0 0	mod 0 reg r/m		

# Intel 8088: Formato de instrução de código de máquina

## JMP = Unconditional Jump:

Direct within Segment

Direct within Segment-Short

Indirect within Segment

Direct Intersegment

Indirect Intersegment

1 1 1 0 1 0 0 1	disp-low	disp-high
1 1 1 0 1 0 1 1	disp	
1 1 1 1 1 1 1 1	mod 1 0 0 r/m	
1 1 1 0 1 0 1 0	offset-low	offset-high
	seg-low	seg-high
1 1 1 1 1 1 1 1	mod 1 0 1 r/m	

### NOTES:

AL = 8-bit accumulator

DS = Data segment

AX = 16-bit accumulator

ES = Extra segment

CX = Count register

Above/below refers to unsigned value

Greater = more positive;

Less = less positive (more negative) signed values

if d = 1 then "to" reg; if d = 0 then "from" reg

if w = 1 then word instruction; if w = 0 then byte instruction

if mod = 11 then r/m is treated as a REG field

if mod = 00 then DISP = 0\*, disp-low and disp-high are absent

if mod = 01 then DISP = disp-low sign-extended to 16 bits, disp-high is absent

if mod = 10 then DISP = disp-high; disp-low

if r/m = 000 then EA = (BX) + (SI) + DISP

if r/m = 001 then EA = (BX) + (DI) + DISP

if r/m = 010 then EA = (BP) + (SI) + DISP

if r/m = 011 then EA = (BP) + (DI) + DISP

if r/m = 100 then EA = (SI) + DISP

if r/m = 101 then EA = (DI) + DISP

if r/m = 110 then EA = (BP) + DISP\*

if r/m = 111 then EA = (BX) + DISP

DISP follows 2nd byte of instruction (before data if required)

\*except if mod = 00 and r/m = 110 then EA = disp-high; disp-low.

if s w = 01 then 16 bits of immediate data form the operand

if s w = 11 then an immediate data byte is sign extended to form the 16-bit operand

if v = 0 then "count" = 1; if v = 1 then "count" in (CL)  
x = don't care

z is used for string primitives for comparison with ZF FLAG

### SEGMENT OVERRIDE PREFIX

0 0 1 reg 1 1 0

REG is assigned according to the following table:

16-Bit (w = 1)		8-Bit (w = 0)		Segment	
000	AX	000	AL	00	ES
001	CX	001	CL	01	CS
010	DX	010	DL	10	SS
011	BX	011	BL	11	DS
100	SP	100	AH		
101	BP	101	CH		
110	SI	110	DH		
111	DI	111	BH		

Instructions which reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file:

X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)

Ex: B4 06 =? → 1011 0100 0000110

MOV AH, 6

# Conjunto de Instruções



Disponível em

[http://www.emu8086.com/assembly\\_language\\_tutorial\\_assembler\\_reference/8086\\_instruction\\_set.html](http://www.emu8086.com/assembly_language_tutorial_assembler_reference/8086_instruction_set.html)

# Conjunto de Instruções Intel 8088/8086

## Visão Geral

### Data Transfer (14)

MOV, PUSH, POP, XCHG, IN, OUT, XLAT, LEA, LDS, LES, LAHF, SAHF, PUSHF, POPF

### Arithmetic (20)

ADD, ADC, INC, AAA, DAA, SUB, SSB, DEC, NEG, CMP, AAS, DAS, MUL, IMUL, AAM, DIV, IDIV, AAD, CBW, CWD

### Logic (12)

NOT, SHL/SAL, SHR, SAR, ROL, ROR, RCL, RCR, AND, TEST, OR, XOR

### String Manipulation (6)

REP, MOVS, CMPS, SCAS, LODS, STOS

### Control Transfer (26)

CALL, JMP, RET, JE/JZ, JL/JNGE, JLE/JNG, JB/JNAE, JBE/JNA, JP/JPE, JO, JS, JNE/JNZ, JNL/JGE, JNLE/JG, JNB/JAE, JNBE/JA, JNP/JPO, JNO, JNS, LOOP, LOOPZ/LOOPE, LOOPNZ/LOOPNE, JCXZ, INT, INTO, IRETR

### Processor Control (12)

CLC, CMC, STC, CLD, STD, CLI, STI, HLT, WAIT, ESC, LOCK, NOP

# Instruções de movimentação de dados (14)

(d=destination, s=source)

## Instruções gerais de movimentação de dados

- MOV d,s** - move um byte ou palavra; instrução mais comumente usada
- PUSH s** - armazena uma palavra (registro ou memória) na pilha
- POP d** - remove uma palavra da pilha
- XCHG d,s** - troca de dados, entre regs ou de memória para registro
- XLATB** - traduz um byte utilizando uma tabela de pesquisa (sem operandos)
  
- IN d,s** - move os dados (byte ou palavra) da porta de E/S para AX ou AL
- OUT d,s** - move dados (byte ou palavra) de AX ou AL para a porta de E/S
  
- LEA d,s** - carrega endereço efetivo (não os dados) no registro
- LDS d,s** - carrega 4 bytes (começando em s) para o ponteiro (d) e DS
- LES d,s** - carrega 4 bytes (começando em s) até o ponteiro (d) e ES

# Instruções de movimentação de dados MOV



- LAHF** - carrega o byte de baixa ordem do registro da FLAGS para a AH
- SAHF** - armazena AH no byte de baixa ordem da FLAGS
- PUSHF** - copia o registro da FLAGS para a pilha apontada pelo par SS:SP
- POPF** - copia uma palavra da pilha para o registro da FL

## Instruções para mover strings

As instruções de string são repetidas quando aparecer o prefixo REP (CX contém a contagem de repetição)

- MOVS** **d,s** - (MOVSB, MOVSW) transferência de dados de mem. p/ mem.
- LODS** **s** - (LODSB e LODSW) copia os dados para AX ou AH
- STOS** **d** - (STOSB, STOSW) armazena dados da AH ou AX



# Instruções de movimentação de dados MOV

## MOV d,s

**d**=destination (registrador ou endereço efetivo de memória),

**s**=source (dato imediato, registrador ou endereço de memória)

**MOV** pode transferir dados de :

qualquer registro para qualquer registro (exceto registro CS)

memória para qualquer registro (exceto CS)

operação imediata a qualquer registro (exceto CS)

qualquer registro em um local de memória

operação imediata à memória

**MOV** não realiza transferências de memória para memória (deve-se usar um registrador como um armazenamento intermediário).

**MOV** move uma word ou byte dependendo do comprimento de bits do operando; os operandos de origem e destino devem ter o mesmo número de bits

**MOV** não pode ser usado para transferir dados diretamente para o registro CS.

Típico número de **clock cycles** da instrução MOV

2 clocks for register-register

4 clocks for immediate-register

8+ea clocks for memory-register (8086)

12+ea clocks for memory-register (8088)

**clock cycle:** ciclos de clock

**ea:** um valor depende do endereço efetivo

# Número de ciclos de relógio para calcular o endereço efetivo

- O número de ciclos de relógio necessários para uma instrução depende do modo de endereçamento.
- Para sistemas 8088/8086, o cálculo do endereço efetivo (16 bits) do operando pode levar vários ciclos de relógio (estes ciclos de relógio extras não são necessários para os microprocessadores posteriores):
- Exemplos Ciclos de relógio extras necessários para o cálculo do endereço efetivo:

Addressing mode	Example	Clock
register indirect	MOV CL,[DI]	5
Direct	MOV CX,FRED	3
Based index	MOV BL,[BP+DI]	7
Based index (displacement)	MOV CX,[BP+DI+TABELA]	11
segment override prefix	MOV AL,[ES:SI]	ea+2

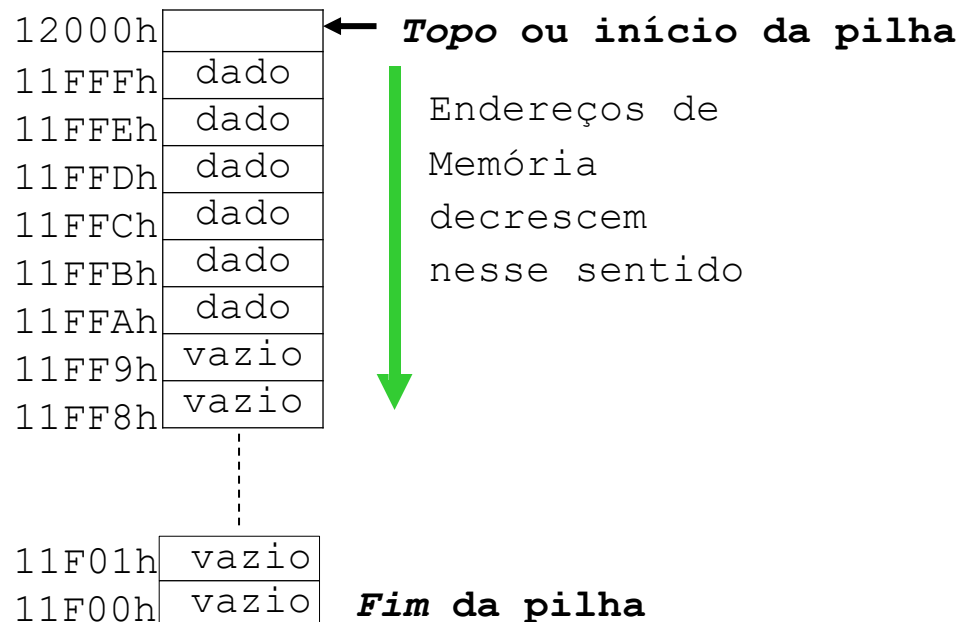
- Substituir o registro de segmento padrão adiciona 2 ciclos de relógio (ea+2).
- As instruções que envolvem puramente operações entre registradores ou dados imediatos para registradores não precisam de ciclos de relógio extras para calcular o endereço efetivo.

# A Pilha (Stack)

- A **pilha (stack)** é um bloco de memória reservado para o armazenamento temporário de dados e registros. O acesso é LAST-IN, FIRST-OUT (LIFO)
- O último local de memória utilizado na pilha é dado por (o endereço efetivo calculado pelo registro SP) e (o registro SS):

**Exemplo: Mapa de memória de uma pilha de 256 bytes:** Suponha uma pilha, de 256 bytes de tamanho, cujo topo encontra-se no endereço especificado pelo par SS:SP, como se segue:

Para SS=1000h e SP=2000h  
Assim, seu EA (*Effective Address*) é 12000h



# A Pilha (Stack)

- Os dados podem ser armazenados na pilha usando a instrução PUSH - que decrementa automaticamente SP de 2.
- A instrução POP remove os dados da pilha (e incrementa SP de 2).
- A pilha pode ter até 64K-bytes de comprimento.
- Todas as operações de pilha usando push/pop envolvem operandos do tipo word, ou seja:

push	AL	; errado
pop	AL	; errado

push	AX	; correto
pop	AX	; correto
- Entretanto, é possível criar uma “variável local” de 1 byte na pilha.

# Instruções PUSH e POP (stack)

## Exemplos:

**PUSH AX** ; armazena o AX na pilha

**POP AX** ; remove uma palavra da pilha e a carrega no AX

**PUSHF** ; armazena o registro da FLAGS na pilha.

**POPF** ; remove uma palavra da pilha e a carrega na FLAGS

**PUSH** pode ser usado com qualquer registro para salvar uma palavra (o conteúdo do registro) na pilha. Para armazenar uma word na pilha utiliza-se a ordem usual, ou seja *little-endian*.

**PUSH** também pode ser usado com dados imediatos, ou dados em memória.

**POP** é o inverso da instrução **PUSH**; ela remove uma palavra do topo da pilha. Qualquer local de memória ou registrador (ambos de 16 bits) podem ser usados como destino de uma instrução **POP**

**PUSHF** e **POPF** salva e carrega o registrador **FLAGS** de/para a pilha, respectivamente.

# Instrução XCHG

- **XCHG** troca o conteúdo de dois registros ou de um registro e memória.
- Tanto a troca de bytes como a troca de palavras são possíveis.
- Exemplos:

**XCHG AX,BX**; troca do conteúdo de AX e BX

**XCHG CL,BL**; troca do conteúdo de CL e BL

**XCHG DX,[PESO]**; troca de conteúdo de DX e memória DS:PESO

- As trocas de memória para memória usando XCHG **NÃO** são permitidas.

# Instrução XLATB

- Muitas aplicações precisam converter rapidamente um código em outro, mapeando um valor de byte para outro (por exemplo, mapeando códigos binários de teclas de um teclado para código ASCII).
- **XLATB** pode realizar uma tradução de bytes usando uma tabela de visualização contendo até 256 elementos.
- **XLATB** assume que a tabela de 256 bytes começa no endereço dado por **DS:BX** (ou seja, endereço efetivo formado pelos registros **DS** e **BX**). **AL** é usado como um índice para apontar o elemento requerido na tabela antes da execução do **XLATB**. O resultado da instrução **XLATB** é retornado no mesmo registrador **AL**.

Assuma, antes de executar **XLATB**, que **DS=1000h**,  
**BX=2000h** e **AL=80h**

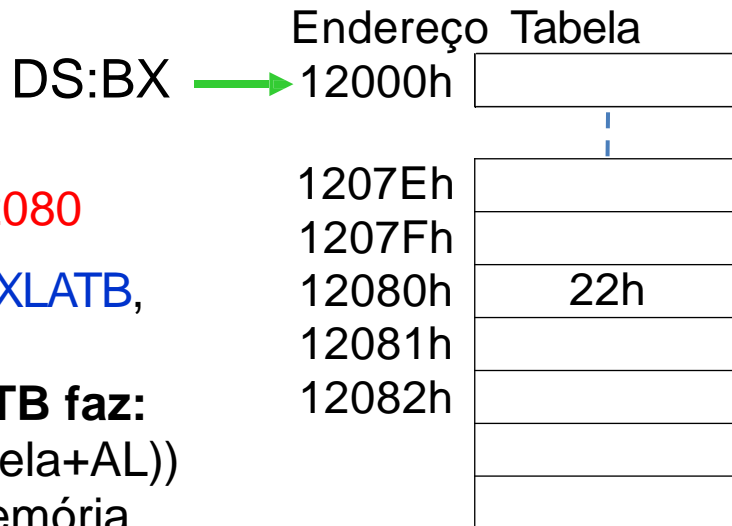
Endereço Físico= $DS:(BX+AL)=12080h = 0x12080$

```
MOV BX, Tabela
MOV AL, 80h
XLATB
```



**Depois de executar XLATB,**  
**AL=22h**

**Ou seja, XLATB faz:**  
 $AL \leftarrow \text{mem}(\text{DS}:(\text{Tabela} + \text{AL}))$   
Para mem = memória



# Instruções LEA & LDS

## LEA - Carrega endereço efetivo

- LEA carrega o offset de um endereço de memória em um registro de 16 bits. O endereço de offset pode ser especificado por qualquer um dos modos de endereçamento.

➤ **Exemplos** (com BP=1000h):

LEA AX,[BP+40h]	; [SS:1000h+40h] = [SS:1040h]; carrega 1040h em AX
LEA BX, TABELA	; carrega o offset de TABELA (no segmento de dados) para BX
LEA CX, [ES:MENSAGEM]	; carrega o offset de MENSAGEM (em segmento extra) para CX

## LDS - Carrega dados e DS

- LDS lê duas palavras de locais de memória consecutivos e as carrega no registro especificado e nos registros do segmento DS.

➤ **Exemplos** (DS=1000h inicialmente)

LDS BX,[2222h]	; cópias do conteúdo 12222h para BL, 12223h para BH, e 12224h e 12225h
	; para registro DS

LDS pode ser útil para inicializar os registros SI e DS antes de uma operação de string.

Por exemplo,

LDS SI, [offset]

- A fonte para LDS pode ser deslocamento, índice ou registro de ponteiro (exceto SP).



# LES – Carrega dado e ES

LES lê duas palavras da memória e é muito semelhante a LDS, exceto que a segunda palavra é armazenada em ES ao invés de DS.

LES é útil para inicializar DI e ES em operações de string.

**Exemplo** (com DS=1000h):

```
LES DI, [2222h]    ; carrega DI com conteúdo armazenado em 12222h e  
                   ; 12223h e carrega ES com o conteúdo armazenado  
                   ; em 12224h e 12225h
```

# Instruções LAHF e SAHF



## LAHF e SAHF

**LAHF** carrega AH com o byte de baixa ordem do registro da FLAGS;

**SAHF** Armazenar AH no byte de baixa ordem do registro da FLAG;

- Instruções muito raramente utilizadas - originalmente presentes para permitir a tradução de 8085 programas para 8086.

# Entrada e saída de dados: IN, OUT

## Instruções IN e OUT

### Exemplos:

IN AX, C8h ; lê a porta de endereço C8h (8 bits) e carrega seu conteúdo em AX;  
IN AL, DX ; lê a porta com endereço em DX e carrega seu conteúdo em AL;  
OUT 80h , AX ; coloca o conteúdo de AX na porta de endereço 80H;  
OUT DX, AX ; coloca o conteúdo de AX na porta cujo endereço está em DX;

- IN lê dados de E/S (8 bits ou 16 bits) e os carrega no acumulador (AL ou AX)
- O endereço de uma porta de 8-bits pode aparecer direto na instrução (ex.: C8h ou 80H) ou pode estar especificado em DX;
- Endereços maiores que 8-bits, necessariamente, precisam estar em DX;
- OUT carrega dados do acumulador (AL ou AX) para para uma porta (8 bits ou 16 bits)
- Somente o acumulador, AL ou AX, são permitidos nas instruções de IN e OUT.

# II. Instruções Aritméticas (20)

- O 8088 tem 20 instruções para realizar adição inteira, subtração, multiplicação, divisão e conversões de BCD

Addition	—	ADD d,s	; soma byte ou word: $d \leftarrow d+s$
	—	ADC d,s	; soma byte or word com carry (vai-1): $d \leftarrow d+s+1$
	—	INC d	; incrementa byte ou word somando 1: $d \leftarrow d+1$
	—	DAA	; <i>decimal adjust after addition</i>
	—	AAA	; <i>ASCII adjust for addition</i>
Subtraction	—	SUB d,s	; subtrai byte ou word: $d \leftarrow d-s$
	—	SBB d,s	; subtrai byte ou word com empréstimo: $d \leftarrow d-s-1$
	—	DEC d	; decrementa byte or word: $d \leftarrow d-1$
	—	DAS	; <i>decimal adjust after subtraction</i>
	—	AAS	; <i>ASCII adjust for subtraction</i>

# Instruções Aritméticas

Multiplicação

- MUL s ; multiplica byte or word, sem sinal
- IMUL s ; multiplicação inteira com sinal, signed
- AAM ; *ASCII adjust for multiply*

Divisão

- DIV s ; divide byte ou word sem sinal
- IDIV s ; divide byte ou word com sinal
- AAD ; *ASCII adjust for divide*

Outras

- NEG d ; Complemento de 2
- CBW ; converte byte para word e estende o sinal
- CWD ; converte word para double word e estende o sinal
- CMP d, s ; compara byte or word
- NOT d ; Complemento de 1

# Adição

- A adição binária de dois bytes ou duas palavras é feita utilizando:

ADD d,s

- ADD adiciona bytes ou palavras em d e s e armazena o resultado em d.
- Os operandos d e s podem usar os mesmos modos de endereçamento que em MOV.
- A adição de palavra dupla é obtida usando o bit de transporte no registro FLAGS.
- A instrução

ADC d,s

inclui automaticamente o FLAG de transporte.

# Adição

**Exemplo:** adição de duas palavras *double word* armazenadas em [x] e [y]

```
MOV AX, word [x]
```

```
MOV DX, word [x+2]
```

```
ADD AX, word [y]
```

```
ADC DX, word [y+2]
```

- A adição de números no formato BCD pode ser feita usando ADD ou ADC seguido da instrução DAA para converter o número no registro AL para uma representação BCD.
- A adição de números em sua forma ASCII é conseguida usando AAA

# ASCII adjust for Addition (AAA)

ASCII codes for the numbers 0 to 9 are 30H to 39H respectively.

The ascii adjust instructions convert the sum stored in **AL** to two-byte unpack BCD number which are placed in **AX**.

When **30H** is added to each byte, the result is the ASCII codes of the digits representing the decimal for the original number in AL.

**Example:** Register AL contains 31H (the ASCII code for 1), BL contains 39H (the ASCII code for 9).

ADD AL, BL ; produces the result 6AH which is kept in AL.

AAA ; converts 6AH in AL to 0100H in AX

Addition of 30H to each byte of AX can then produce the result 3130H (the ASCII code for 10 which is the result of 1+9)

		ASCII code	decimal
	AL	31H	1
	BL	+ 39H	+ 9
ADD AL, BL	AL	6AH	10

Depois de AAA,                      0100H

AX =

add 30H to each byte      →      3130H



# DAA Decimal adjust after addition

- Decimal adjust after addition: A instrução DAA é usada após o emprego das instruções ADD ou ADC para ajustar a resposta da soma à representação do Código BCD.
- DAA só opera com o registrador AL, ou seja, 8 bits por vez.

Exemplo de uso:

```
MOV BX,3216H
```

```
MOV DX,4029H
```

```
MOV AL,BL
```

```
ADD AL,DL
```

```
DAA
```

```
MOV CL,AL
```

```
MOV AL,BH
```

```
ADC AL,DH
```

```
DAA
```

```
MOV CH,AL ; Ao final CX=7245BCD = 3216BCD + 4029BCD
```

# Subtração

- A subtração de dois bytes ou duas palavras é realizada utilizando:

`SUB d, s` ; faz  $d \leftarrow (d-s)$

- O número no operando `s` é subtraído do operando `d` e o resultado é armazenado em `d`. `s` e `d` são encontrados usando os mesmos modos de endereçamento que `MOV`.
- Se  $s > d$ , o resultado é negativo (em forma de complemento de 2).
- A subtração de dois números *double word* é realizada usando `SUB` para subtrair as palavras de ordem baixa, e `SBB` para subtrair as palavras de ordem alta. `SBB` leva em conta o *carry* a `s` antes de realizar a subtração  $d-s$ . `SBB` atualizará o *carry* após a execução.
- A subtração de números *unpacked* BCD é obtida usando `SUB` (ou `SBB`) seguido pela instrução `DAS` (ajuste decimal após a subtração) que converte o número armazenado no acumulador para BCD.
- `AAS` é usado para corrigir a subtração (`SUB` ou `SBB`) de 2 operandos no formato *unpacked* BCD.

# Multiplicação

- **MUL s**; multiplica dois bytes ou duas *words* sem sinal. Um dos multiplicadores é o acumulador (**AX** ou **AL**) e o outro é especificado por um registro ou pelo conteúdo de um local de memória (sem endereçamento imediato).
- O resultado da multiplicação de dois bytes é armazenado no **AX**.
- Quando duas *words* são multiplicadas, o resultado tem um comprimento de 32 bits. A *word* de alta ordem é armazenada em **DX**.
- **IMUL s**; multiplica dois números com sinal (s e **AX** ou **AL**).
- **AAM** (ajuste p/ ASCII após a multiplicação) converte o conteúdo de **AL** em dois dígitos separados - um está em **AL** e o outro está em **AH**. A adição de 30H converterá então **AL** e **AH** para o código ASCII do resultado original. **AAM** dará uma resposta correta somente se cada um dos dois multiplicandos forem < 10 (ou seja, é necessário primeiro subtrair 30H dos códigos ASCII antes de multiplicar).

# Divisão

- Uma *word* **sem sinal** em AX pode ser dividida usando

**DIV s**

na qual o divisor *s* é um número de tamanho de byte em um registrador ou em memória.

- A divisão de uma *word* (previamente em **AX**) produz um quociente de 8 bits que é armazenado em **AL**, e um resto que é colocado em **AH**.
- Uma *double word* sem sinal pode ser dividida por uma *word* usando **DIV s**
- A *double word* está em **AX** e **DX** (palavra mais significativa em **DX**) e *s* é o conteúdo de um registrador de 16 bits ou o conteúdo de um local de memória. O quociente de 16 bits é colocado em **AX** e o resto de 16 bits é colocado em **DX**
- A divisão **sem sinal** de uma palavra por uma palavra é feita preenchendo **DX** com zero.
- O **IDIV** opera da mesma forma que o **DIV** para operandos sem sinal.
- O **AAD** (ajuste ASCII para divisão) é usado para converter dois números codificados em **AL** e **AH** para binário. O **AAD** é usado antes de uma divisão.
- Como regra, divide-se: 16 bits/8 bits ou 32bits/16bits

# Instruções de extensão de sinal

**CBW** (*convert byte to word*) estende o sinal, em C2, de AL para AX. A extensão do sinal envolve ou colocar zeros no byte mais significativo, AH, se o byte original for positivo, ou preencher o byte mais significativo com 1's se o byte original for negativo.

**CWD** (*convert word to double-word*) estende o sinal de **AX**, agora sobre o registrador DX, conforme o procedimento usado em CBW

## Exemplo:

MOV AX, 0 ; AH = 0, AL = 0

MOV AL, -5 ; AX = 00FBh (=251)

CBW ; AX = FFFBh (= -5)

Para o uso de IDIV (Observe a representação em C2)

16 BITS ÷ 8 BITS

MOV AL, -128 ; 0x80

MOV BL, 2

CBW; (AX=0xFF80)

IDIV BL

; Quociente AL = -64 (0xC0) ;

; Resto AH = 0

32 BITS ÷ 16 BITS

MOV AX, -32768 ; 0x8000 (1000000000000000b)

MOV BX, 2

CWD; (DX=0xFFFF, pois o 16º BIT de AX é = 1)

IDIV BX

; Quociente AX = -16384 (0xC000)

; Resto DX = 0

# Outras Operações Aritméticas

## NEG d

Faz o complemento de 2 de d.

## CMP d,s

(*Compare*) realiza a subtração d-s, sem destruir o conteúdo de d. Como deseja-se apenas comparar 2 números, os respectivos flags da ULA são ativados conforme o resultado (se for =0, >0, <0, paridade par, paridade ímpar, se houve *carry*, se houve *carry* auxiliar, etc..)

### Exemplo:

```
MOV AL, 5      ; AL = 05h  
NEG AL         ; AL = FBh (-5)  
NEG AL         ; AL = 05h (5)
```

### Exemplo:

```
MOV AL, 5  
MOV BL, 5  
CMP AL, BL ; AL = 5, ZF = 1 (AL = BL pois  
ZeroFlag ZF=1 indica que AL-BL=0 !)
```

# III. Instruções Lógicas e de Manipulação de bits (12)

Os bits em um *byte* ou *word* podem ser manipulados diretamente usando o seguinte conjunto de 12 instruções:

Logical	AND d, s	; E Lógico resultado colocado em d
	NOT d	; Complementa cada bit de d
	OR d, s	; OU Lógico, resultado armazenado em d
	XOR d, s	; XOR, resultado colocado em d
	TEST d, s	; E para atualizar as <i>flags</i> P,Z, S. Essa operação ; não destrói o valor de d
Shift	SAL/SHL d, c	; bits de deslocamento à esquerda c vezes. Insere zero
	SAR d, c	; desloca para a direita c vezes. Insere bit de sinal
	SHR d, c	; desloca para a direita c vezes. Insere zeros

# Instruções Lógicas e de Manipulação de bits (Cont.)

Rotate	ROL d, c	gira os bits para a esquerda c vezes
	ROR d, c	gira os bits para a direita c vezes
	RCL d, c	gira os bits para a esquerda, usando o <i>carry</i> , c vezes
	RCR d, c	gira os bits para a direita, usando o <i>carry</i> , c vezes

- Os operandos de origem nas instruções lógicas podem ser dados imediatos, outro registro ou o conteúdo de um local de memória. Tanto *bytes* como *words* são permitidos.
- Para o 8088/8086, a contagem c nas instruções de rotação/deslocamento são ou =1 ou um número armazenado no registro CL.
- O operando de destino, d, pode ser um registro ou um local de memória.

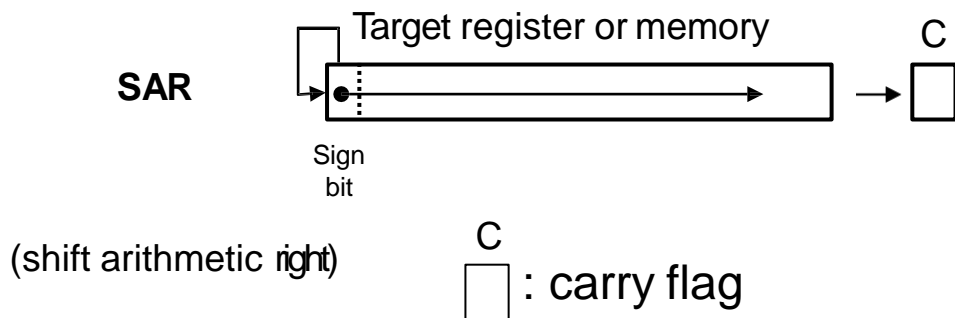
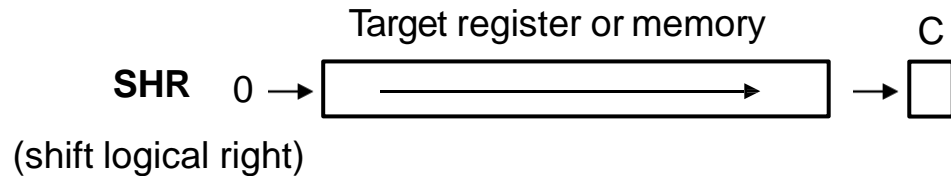
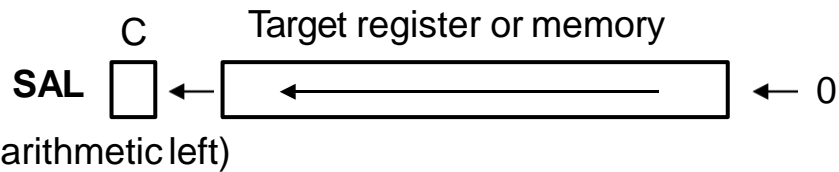
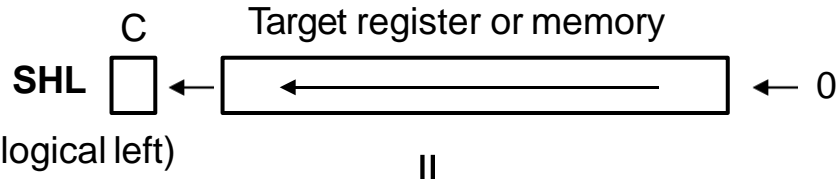


# Shift & Rotate

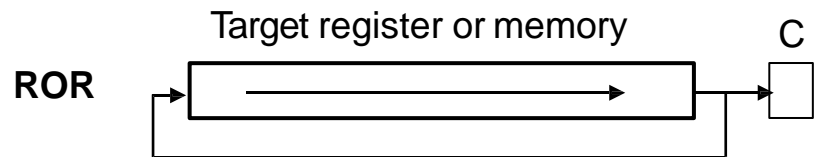
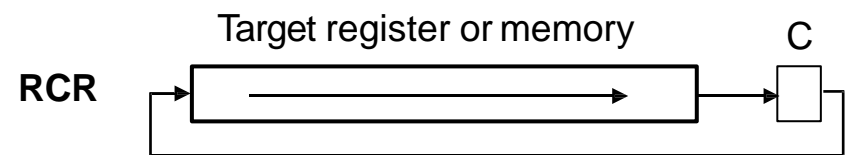
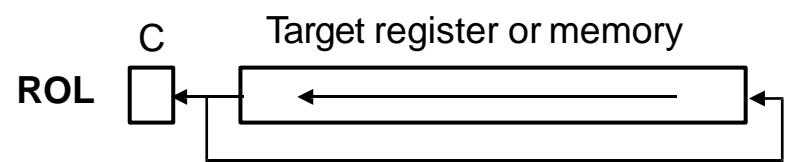
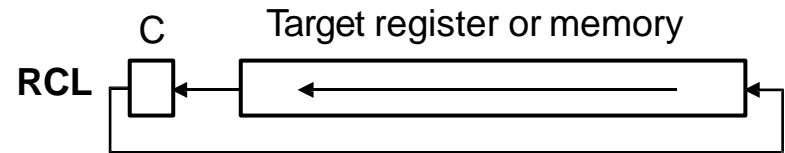
- Um deslocamento à esquerda de 1 equivale a multiplicar por 2 (com bit entrante o MSB = 0). Usando **SAL**, o bit mais significativo é deslocado para o *carry* e o bit menos significativo é preenchido com um 0.
- Um deslocamento à direita por uma posição é equivalente a dividir por dois, quando o bit de sinal é mantido (usando **SAR**).
- **ROL** e **ROR** giram os bits para a esquerda e para a direita, respectivamente. O bit de *carry* não participa da rotação mas recebe o bit MSB (**ROL**) ou o LSB (**ROR**).
- **RCL** e **RCR** giram para a esquerda e para a direita respectivamente. O bit de *carry* participa da rotação.

# Shift & Rotate (cont.)


## Shift



## Rotate



# Criando *packed* BCD a partir de caracteres ASCII



```
START: MOV BL, "5"      ; Load first ASCII digit into BL      (35H)
        MOV AL, "9"      ; Load second ASCII digit into AL    (39H)
        AND BL, 0FH       ; Mask upper 4 bits of first digit   (05H)
        AND AL, 0FH       ; Mask upper 4 bits of second digit  (09H)
        MOV CL, 04H       ; Load CL for 4 rotates required
        ROL BL, CL        ; Rotate BL 4 bit positions
        OR AL, BL         ; Combine nibbles, result in AL      (59H)
```

# IV. Instrução de Strings (6)

Uma *string* é uma série de bytes ou uma série de palavras em locais de memória sequencial. Consiste frequentemente em códigos de caracteres ASCII.

- |          |  |
|----------|--|
| REP      | - Prefixo de instrução; repete uma instrução até que CX=0                  |
| MOVS d,s | - (MOVSB, MOVSW) transferência entre posições de memória                   |
| COMPS    | - (COMPSB and COMPSW) compara 2 strings                                    |
| SCAS     | - (SCASB and SCASW) varre uma string, compara com um valor em AL ou em AX. |
| LODS s   | - (LODSB and LODSW) carrega de memória para AX ou AL                       |
| STOS d   | - (STOSB, STOSW) armazena em memória o conteúdo de Ax ou AL                |

# IV. Instrução de Strings

- As operações *String* assumem os ponteiros da *string* (ou seja, os registros que mantêm o endereço de onde a *string* é armazenada na memória):
- **DS:SI** - endereço da *string* de caracteres de origem
- **ES:DI** - endereço da *string* de destino
- **LES** e **LDS** são úteis para inicializar estes registros
- **CX** armazena a contagem para o número de vezes que a instrução de *string* deve ser repetida de modo a processar completamente a *string*.
- Os registros **SI** e índice **DI** são incrementados ou decrementados automaticamente, dependendo do valor da **flag de direção (DF)** no registro **FLAGS**. O procedimento é repetido **CX** vezes quando o prefixo **REP** é anexado à instrução *string*
- Para fazer **DF = '1'** e **DF='0'**, usa-se instruções **STD** (*set DF*) e **CLD** (*clear DF*), respectivamente.
- **DF='1'** implica em decrementar os registradores **SI** e **DI** automaticamente
- **DF='0'** implica em incrementar os registradores **SI** e **DI** automaticamente

# Instrução de Strings (cont.)

- **MOVS** transfere *byte* ou *word* de memória para memória (a única instrução capaz de transferir dados da memória para a memória)
- **MOVS** copia um dado de uma *string* de uma posição de memória (**end\_mem1**) para outra (**end\_mem2**). Os registros DS:SI e ES:DI devem ser inicializados com **end\_mem1** e **end\_mem2**, respectivamente.
- Na verdade, usa-se **MOVSB** ou **MOVSW** para mover *byte* ou *word*, respectivamente.
- Usar **MOVS** causa um erro pois não se especifica se vai mover *byte* (**MOVSB**) ou *word* (**MOVSW**).

# Prefixos Rep, REPE, REPZ, REPNE e REPNZ

- **REP** repete uma instrução de *string*, decrementando automaticamente a contagem em CX após cada repetição até que **CX** = 0. **Ex.:** (com CX=5):  
**REP MOVSB** ; repete MOVSB 5 vezes, ou seja, transfere 5 bytes.
- **REPE** (*repeat while equal*) e **REPZ** (*repeat while zero*) funcionam da mesma forma, mas são utilizados para as instruções **SCAS** e **CMPS**. **REPE** e **REPZ** têm uma condição extra (além de **CX** ≠ 0) de repetir somente quando o *flag* zero é ativado (ZF=1); ou seja, a comparação entre *strings* é interrompida assim que as duas *strings* não são iguais.
- **REPNE** e **REPNZ** também repetem enquanto **CX** ≠ 0. A condição extra é que a *flag* zero não seja ativada (ZF=0) para que a repetição continue.

- Assim, existem 3 maneiras de repetir uma instrução de *string*:

**REP** - repete se **CX** ≠ 0

**REPE, REPZ** - repete se **CX** ≠ 0 e ZF = 1

**REPNE, REPNZ** - repete se **CX** ≠ 0 e ZF = 0

# Instruções de string LODS & STOS

- **LODS** (uso: **LODSB** ou **LODSW**) carrega o acumulador com um *byte* ou *word* do local de memória **DS:SI** e incrementa automaticamente o registro **SI** se **DF=0** no registro **FLAGS** (decrementa automaticamente **SI** se **DF=1** no registro **FLAGS**).
- **LODSB** carrega **AL** com o conteúdo de memória **DS:SI** e faz **SI=SI+/-1**;
- **LODSW** carrega **AX** com o conteúdo de memória **DS:SI** e faz **SI=SI+/-2**
  
- **STOS** (uso: **STOSB** ou **STOSW**) armazena o conteúdo do acumulador, *byte* ou *word*, no do local de memória apontado por **ES:DI** e incrementa automaticamente o registro **DI** se **DF=0** no registro **FLAGS** (decrementa automaticamente **DI** se **DF=1** no registro **FLAGS**).
- **STOSB** armazena **AL** na posição de memória **ES:DI** e faz **DI=DI+/-1**;
- **STOSW** armazena **AL** na posição de memória **ES:DI** e faz **DI=SI+/-2**.



# Instrução de Strings: CMPS & SCAS

**CMPS** (compare string) compara os *bytes* ou *words* (dependendo se é usado **CMPSB** ou **CMPSW**) armazenados em DS:SI e ES:DI. **CMPSB/CMPSW** é frequentemente usado com o prefixo **REPZ**, com **CX** definindo o número de repetições, a fim de comparar duas *strings*.

**SCAS** (scan string) compara cada elemento de uma *string* com o valor armazenado em AX (**SCASW**) ou AL (**SCASB**). O elemento da *string* é apontado pelo par ES:DI. Se **SCASB/SCASW** tem prefixo **REPZ** a cadeia de caracteres é procurada pelo byte ou palavra no acumulador. Se SCAS é prefixado por REPZ, a busca é repetida até que o elemento *string* seja diferente do valor no acumulador.

**Exemplo:** (with ES:DI set to stringA and CX set to string length)

REPZ SCAS stringA	; scan stringA until item in accumulator is found
REPZ SCAS stringA	; scan until not equal to the content of accumulator

# Instruções de Controle de Fluxo de Programa

- O ponteiro de instruções (IP) tem o endereço da próxima instrução armazenada na memória. Normalmente o programa executa as instruções na mesma sequência em que elas são armazenadas na memória
- A execução sequencial de instruções pode ser deliberadamente interrompida pela alteração do conteúdo do ponteiro de instruções para executar ramificações condicionais, loops e chamada de funções.

Saltos incondicionais	JMP d	; Salto incondicional
	CALL d	; chamada de funções
	RET	; retorno de funções
Iterações	LOOP d	; loops para d até CX=0
	LOOPE/LOOPZ d	; loop do tipo <i>while equal</i>
	LOOPNE/LOOPNZ d	; loops do tipo <i>while not zero</i>
Interrupções	INT type	; interrupção de software
	IRET	; retorno da interrupção
	INTO	; interrupção por situação de ; <i>overflow</i>

# Instruções de Controle de Fluxo de Programa

**Salto**  
condicionais.  
Verifica-se os  
bits de Flag:

Sinal (S),  
Zero (Z),  
Carry (C),  
Paridade (P),  
Overflow (O)

JE/JZ d	; jump zero; if Z=1
JNE/JNZ d	; jump not zero; if Z=0
JL/JNGE d	; jump less; if (S xor O)=1
JNL/JGE d	; jump not less; if (S xor O)=0
JLE/JNG d	; less or equal; if ((S xor O) or Z)=1
JNLE/JG d	; jump greater; if ((S xor O) or Z)=0
JB/JC/JNAE d	; jump below; if C=1
JNB/JNC/JAE d	; jump not carry; if C=0
JBE/JNA d	; jump below or equal; if (C or Z)=1
JNBE/JA d	; jump above; if (C or Z)=0
JP/JPE d	; parity equal; if P=1
JNP/JPO d	; jump not parity; if P=0
JO d	; jump overflow; if O=1
JNO d	; jump no overflow; if O=0
JS d	; jump sign; if S=1
JNS d	; jump not sign; if S=0
JCXZ d	; jump if CX=0

# Salto Incondicional: JUMP

- A instrução de *jump* (salto) **JMP** carrega o ponteiro de instrução (IP) com o novo endereço de a próxima instrução, deixando todos os outros registros inalterados.
- O destino do salto pode ser **intra-segmento** e especificado por:
  - um operando, com sinal, de 8 bits que é adicionado ao IP. Um salto curto direto é produzido, por exemplo: `JMP SHORT VALOR`. Observe que o operando de 8 bits é geralmente um rótulo, e a diretiva `SHORT` diz ao *assembler* para codificar a instrução como um salto curto (ou seja, o rótulo está entre -128 a +127 bytes).
  - Um operando, com sinal, de 16 bits por exemplo. `JMP WORD PTR VALOR`. Aqui, o montador substitui o conteúdo do ponteiro de instruções pelo endereço *offset* de onde a etiqueta `VALOR` está localizada. A diretiva `WORD PTR` indica que o rótulo é um *word*.
  - O endereçamento indireto que fornece um novo endereço efetivo para IP. Exemplo: `JMP WORD PTR [BX]`; salta para o endereço dado pelo conteúdo da localização da memória apontada por `BX`.

# Salto Incondicional: JUMP

O destino do JMP pode ser inter-segmentos. Neste caso, tanto o IP quanto o CS devem ser alterados. Os novos valores para IP e CS podem ser especificados indiretamente (por exemplo, JMP DWORD PTR [SI]) ou diretamente (por exemplo, JMP DWORD PTR CASA. IP é substituído pela primeira *word* apontada pelo operando e CS é substituído pela próxima *word* (não iremos praticá-lo).

---

## Resumo do Jump : 5 tipos

- |                           |  |
|---------------------------|--|
| (1). Direct short jump :  | $IP := IP + \text{disp8}$                        |
| (2). Direct near jump :   | $IP := IP + \text{disp16}$                       |
| (3). Direct far jump :    | $IP := \text{disp16}, CS := \text{base16}$       |
| (4). Indirect near jump : | $IP = [\text{reg}], \text{ or } IP = \text{reg}$ |
| (5). Indirect far jump :  | $IP = [\text{reg}], CS := [\text{reg}+2]$        |

# Exemplo de Salto (condicional e incondicional)



```
BACK:  IN AL, 80h
        CMP AL, 255
        JE SAIR
        JMP BACK
SAIR:   HLT
```

```
BACK:  IN AL, 80H
        CMP AL, 255
        JE BACK
        JMP FIM
AQUI:  CALL CALCULA
        |
        |
        |
FIM:    HLT
```

# Salto Condicional

- Somente salto curto (ou seja, o destino do salto é especificado por um offset de 8 bits, com sinal, que é adicionado ao IP) é possível para as instruções de salto condicional do 8088/8086. Os nomes de registro não podem ser usados como operando de uma instrução de salto condicional.
- Para executar um salto condicional fora da faixa de 8 bits, é necessário combinar um salto condicional curto com um salto incondicional.
- O conjunto de instruções de salto condicional abrange todas as combinações possíveis de zero, *carry*, *overflow*, sinal e *flags* de paridade no registro FLAGS.
- Além dos testes para as várias combinações de C, O, S, Z, e P
- Há uma instrução para testar se o registro CX é zero.

# 8086 Conditional Jump Instructions

MNEMONIC	CONDITION TESTED	"JUMP IF ..."
JA/JNBE	(CF or ZF)=0	above/not below nor equal
JAE/JNB	CF=0	above or equal/not below
JB/JNAE	CF=1	below/not above nor equal
JBE/JNA	(CF or ZF)=1	below or equal/not above
JC	CF=1	carry
JE/JZ	ZF=1	equal/zero
JG/JNLE	((SF xor OF) or ZF)=0	greater/not less nor equal
JGE/JNL	(SF xor OF)=0	greater or equal/not less
JL/JNGE	(SF xor OF)=1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF)=1	less or equal/not greater
JNC	CF=0	not carry
JNE/JNZ	ZF=0	not equal/not zero
JNO	OF=0	not overflow
JNP/JPO	PF=0	not parity/parity odd
JNS	SF=0	not sign
JO	OF=1	overflow
JP/JPE	PF=1	parity/parity equal
JS	SF=1	sign
JCXZ	CX=0	Jump if CX=0
JECXZ	ECX=0	Jump if ECX=0 (386

Nota:

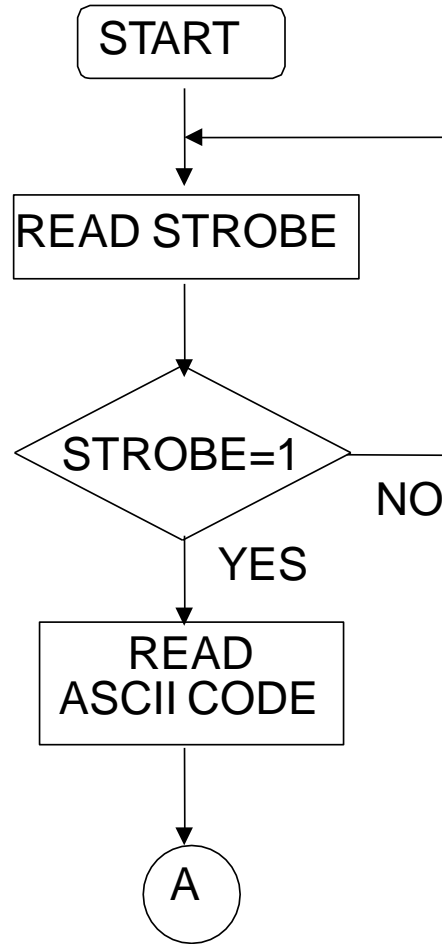
“*above*” e “*below*” referem-se à relação de dois

valores sem sinal;

“*greater*” e “*less*” referem-se à relação de dois valores com sinal.



# Ex.: Lendo código ASCII quando um sinal de strobe for ativado



FLOWCHART

## PSEUDOCODE

REPEAT

    READ KEYPRESSED STROBE

UNTIL STROBE = 1

READ ASCII CODE FOR KEY PRESSED

# Leitura do código ASCII quando um sinal de *strobe* está presente

; FUNÇÃO	: O programa lê um código em ASCII logo depois que ;
; EXECUTADA	um sinal de <i>strobe</i> é enviado por um teclado
; REGISTRADORES	: CS, DX, AL
; USADOS	
; PORTAS DE 8 bits	: <u>FFFAH</u> – Porta com bit de <i>Strobe</i> no LSB
	: <u>FFF8H</u> – Porta que recebe o dado em ASCII
	;

Código de  
máquina

Código em Assembly

0000 BAFFFA  
0003 EC  
0004 24 01  
0006 74 FB  
0008 BA FFF8  
000B EC

LOOK\_AGAIN:

MOV DX, FFFAH	; Point DX at strobe port
IN AL, DX	; Read keyboard strobe
AND AL, 01	; Mask extra bits and set flags
JZ LOOK_AGAIN	; If strobe is low then keep looking
MOV DX, 0FFF8H	; else point DX at data port
IN AL, DX	; Read in ASCII code

# Loops

O 8088/8086 tem 3 instruções que são projetadas para executar uma seção de código para um número fixo de vezes:

LOOP d;

LOOPE/LOOPZ d;

LOOPNE/LOOPNZ d;

$$\text{LOOP} \equiv \begin{cases} \text{DEC CX} \\ \text{JNZ d} \end{cases}$$

- O operando usado em loop é um offset de 8 bits com sinal, normalmente especificado por um rótulo em assembly.
- LOOP decrementa CX e verifica se CX = 0. Se não for, o LOOP salta (curto) para o destino especificado.
- LOOPE é semelhante ao LOOP, exceto que um teste adicional é realizado. LOOPE decrementa CX e verifica se CX=0 e se a *flag* Z=1. LOOPE só saltará se CX ≠ 0 e se Z=1.
- LOOPNZ também é semelhante. LOOPNZ decrementa CX e testa para ver se CX=0 e se a flag Z=0. LOOPNZ só saltará se se CX ≠ 0 e se Z=1.
- Os loops podem ser aninhados (loops dentro dos loops) e o STACK (PILHA) é frequentemente usado para armazenar e recuperar o registro do contador dos loops externos.

# Procedures e Programação Modular

- *Procedures* são sequências agrupadas de instruções que normalmente desempenham uma função específica (por exemplo, encontrar a média de vários números, esperar por uma determinada duração, ...) e que podem ser usadas em diferentes pontos do programa principal.
- **Vantagens** do uso de *procedures*:
  - permite que o programa seja dividido em módulos mais simples
  - economiza espaço de memória e esforço na escrita do programa
  - programas mais legíveis, permitindo a manutenção de futuros programas e uma depuração mais fácil
- **Desvantagens** do uso de *procedures*:
  - maior sobrecarga do programa e, portanto, programas mais lentos

# Procedures e Programação Modular

*Procedures* são executados utilizando a instrução **CALL**

**CALL** tem um operando que normalmente é o rótulo do endereço inicial do *procedure*, através de registro, podem ser usados modos de endereçamento direto e indireto. **CALL** pode iniciar *procedure* que estejam próximos (no mesmo segmento) ou distantes (em um segmento diferente), e as diretivas assembler **WORD PTR** ou **DWORD PTR** são usadas com **CALL** para especificar se deve substituir somente **IP** ou ambos **IP** e **CS**.

**CALL** difere fundamentalmente da instrução **JMP**, uma vez que **CALL** armazena automaticamente alguns conteúdos de registro para o **STACK** antes de entrar no procedimento.

# ***Procedures: CALL NEAR e CALL FAR***

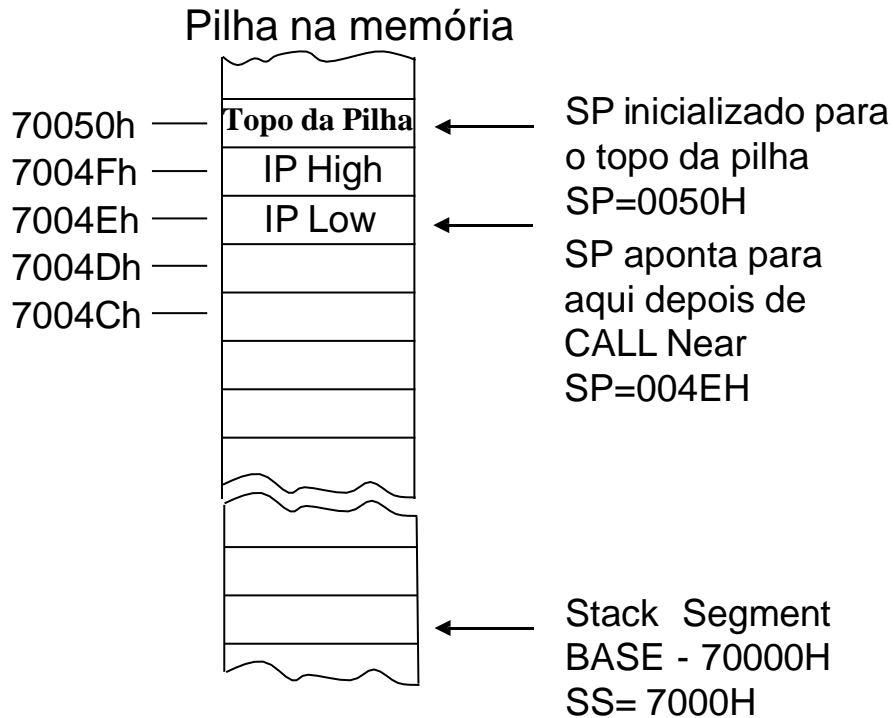
- Uma chamada de um procedure pode ser do tipo NEAR ou do tipo FAR
- Para **CALL NEAR** (ou seja, o procedimento está no mesmo segmento), **CALL** salva o conteúdo do IP na pilha antes de entrar no procedimento e **RET** restaura o IP para retornar ao programa principal.
- Para **CALL FAR** (ou seja, um procedimento em um segmento diferente) envolve salvar tanto o IP quanto os registros CS na pilha antes que os novos valores de IP e CS sejam carregados com o endereço de início do procedimento.
- Abaixo, DELAYS pode estar no mesmo segmento ou em outro segmento.

DELAYS:

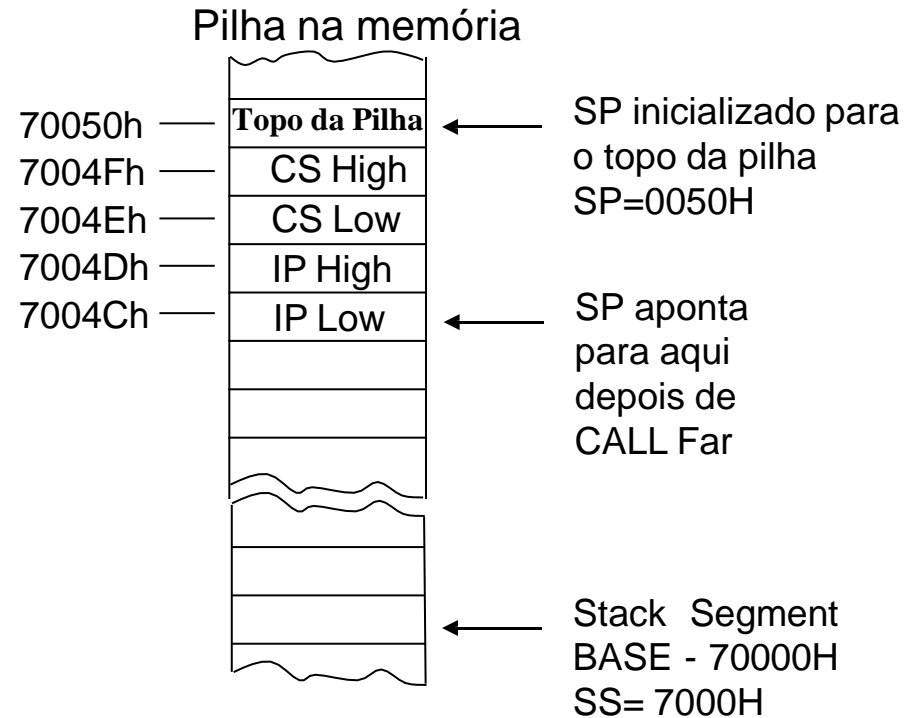
MOV CX,1000H

FRED: NOP ; No **OP**eration  
LOOP FRED  
RET

# Uso da Pilha (Stack)



CALL near



CALL far

# Exemplo: Uso da Pilha (Stack)

Endereço de  
memória –  
par cs:ip

CS : IP

075F:1000H

075F:1002H

075F:1006H

075F:1009H

Trecho de um  
Programa Principal  
que faz um “call”

mov ax, bx

add [1400H], ax

call EXEMPLO

cmp ax, 0

...

Endereço de  
memória de  
EXEMPLO

par cs:ip

CS : IP

075F:2001H

075F:2002H

075F:2003H

075F:0004H

075F:0005H

Procedure:  
EXEMPLO

EXEMPLO:

PUSHF

PUSH AX

PUSH BX

PUSH CX

.....

POP CX

POP BX

POP AX

POPF

RET

Suponha que EXEMPLO esteja localizado  
para aonde aponta o par CS:IP = 075FH:2001H



# CALL, PUSH, POP e RET (1)

Ex: Rotina  
EXEMPLO

Execução de EXEMPLO (call tipo near) e seus efeitos na pilha

EXEMPLO:

PUSHF

PUSH AX

PUSH BX

PUSH CX

.

.

POP CX

POP BX

POP AX

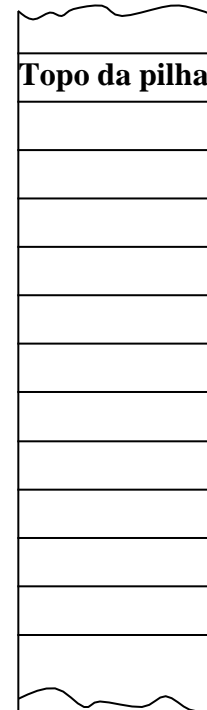
POPF

RET

Antes de CALL

SP Stack in memory

0050h →



Empilhando:  
Ação das  
instruções  
CALL e push

SP

Desempilhando:  
Ação das  
instruções pop e  
RET

# CALL, PUSH, POP e RET (1)

Ex: Rotina  
EXEMPLO

Execução de EXEMPLO (call tipo near) e seus efeitos na pilha

EXEMPLO:

PUSHF

PUSH AX

PUSH BX

PUSH CX

.

.

POP CX

POP BX

POP AX

POPF

RET

Antes de CALL

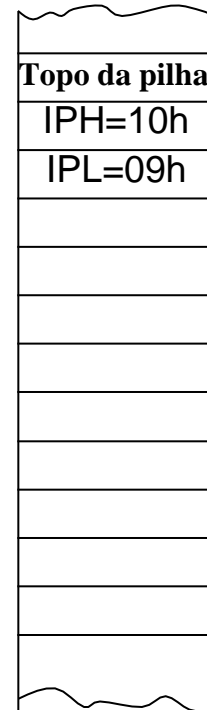
Depois de CALL

Empilhando:  
Ação das  
instruções  
CALL e push

SP Stack in memory

0050h →

004Eh →



SP

Desempilhando:  
Ação das  
instruções pop e  
RET

# CALL, PUSH, POP e RET (1)

## Ex: Rotina EXEMPLO

## Execução de EXEMPLO (call tipo near) e seus efeitos na pilha

EXEMPLO:

PUSHF

PUSH AX

PUSH BX

PUSH CX

.

.

POP CX

POP BX

POP AX

POPF

RET

Antes de CALL

Depois de CALL

Depois de PUSHF

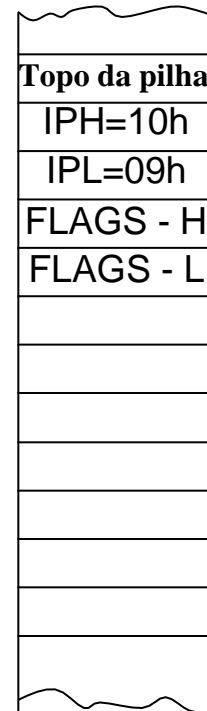
Empilhando:  
Ação das  
instruções  
CALL e push

SP Stack in memory

0050h →

004Eh →

004Ch →



SP

Desempilhando:  
Ação das  
instruções pop e  
RET

# CALL, PUSH, POP e RET (1)

## Ex: Rotina EXEMPLO

## Execução de EXEMPLO (call tipo near) e seus efeitos na pilha

EXEMPLO:

**PUSHF**

**PUSH AX**

**PUSH BX**

**PUSH CX**

.

.

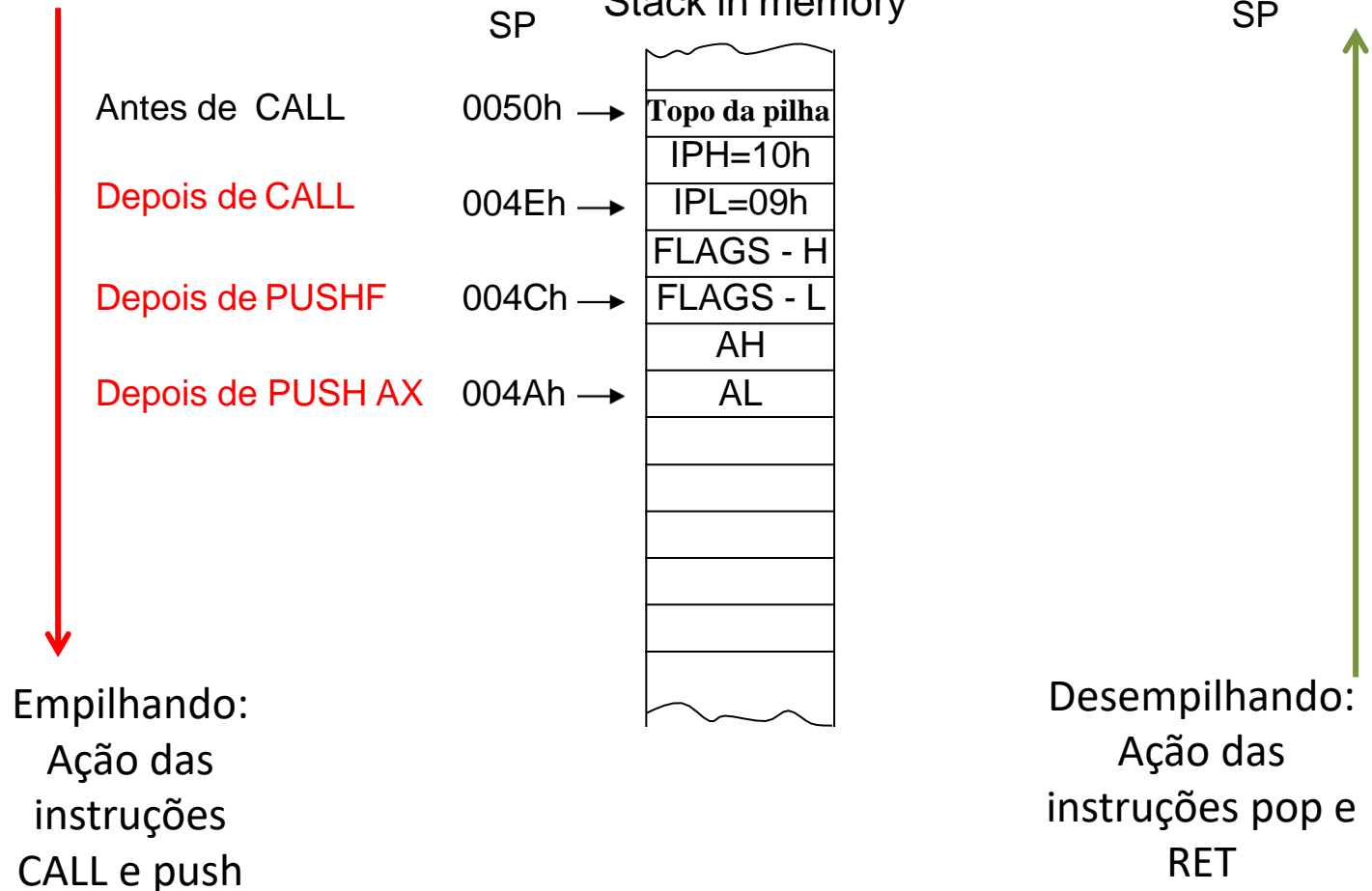
**POP CX**

**POP BX**

**POP AX**

**POPF**

**RET**



# CALL, PUSH, POP e RET (1)

## Ex: Rotina EXEMPLO

## Execução de EXEMPLO (call tipo near) e seus efeitos na pilha

EXEMPLO:

PUSHF

PUSH AX

PUSH BX

PUSH CX

.

.

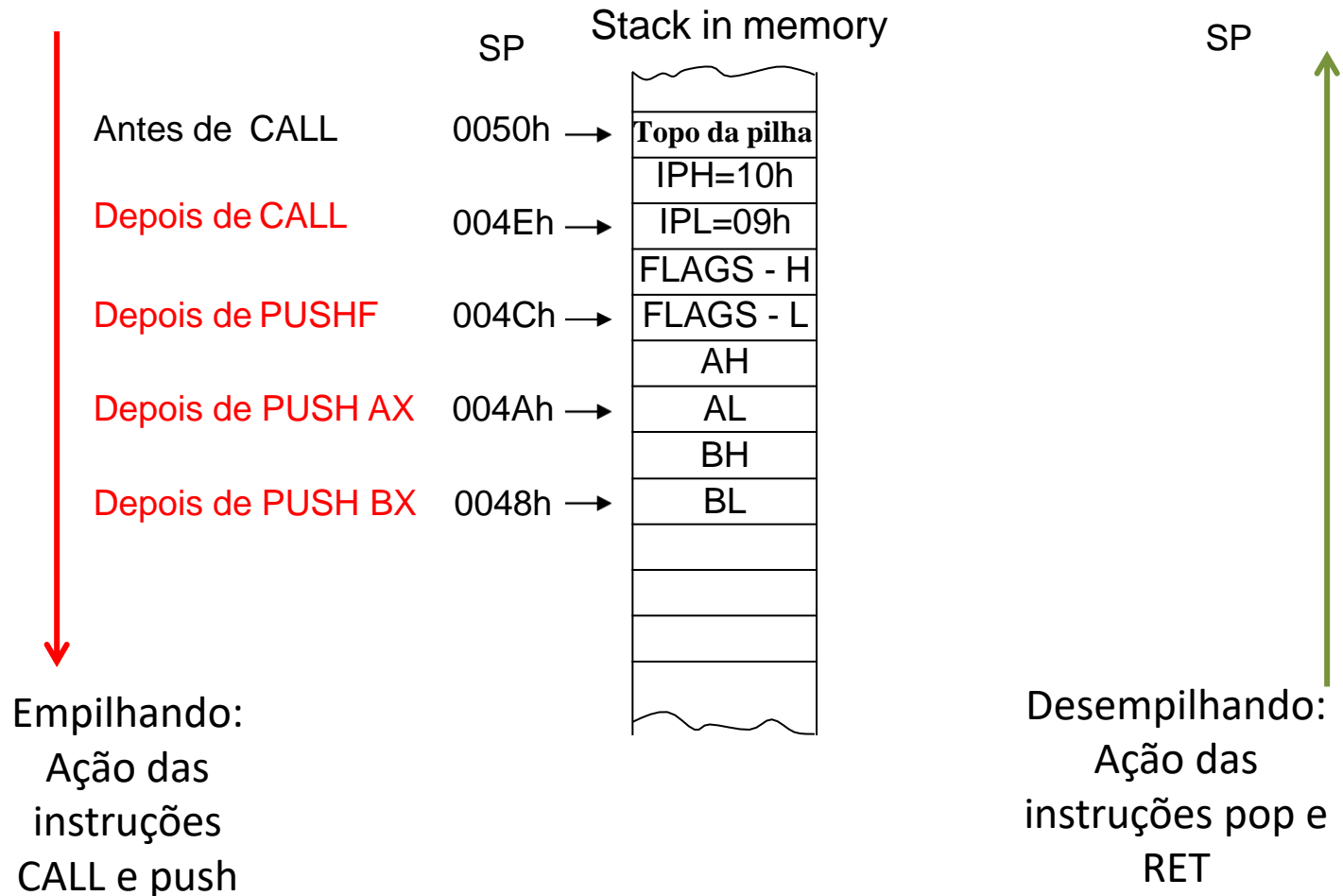
POP CX

POP BX

POP AX

POPF

RET



# CALL, PUSH, POP e RET (1)

## Ex: Rotina EXEMPLO

## Execução de EXEMPLO (call tipo near) e seus efeitos na pilha

EXEMPLO:

PUSHF

PUSH AX

PUSH BX

PUSH CX

.

.

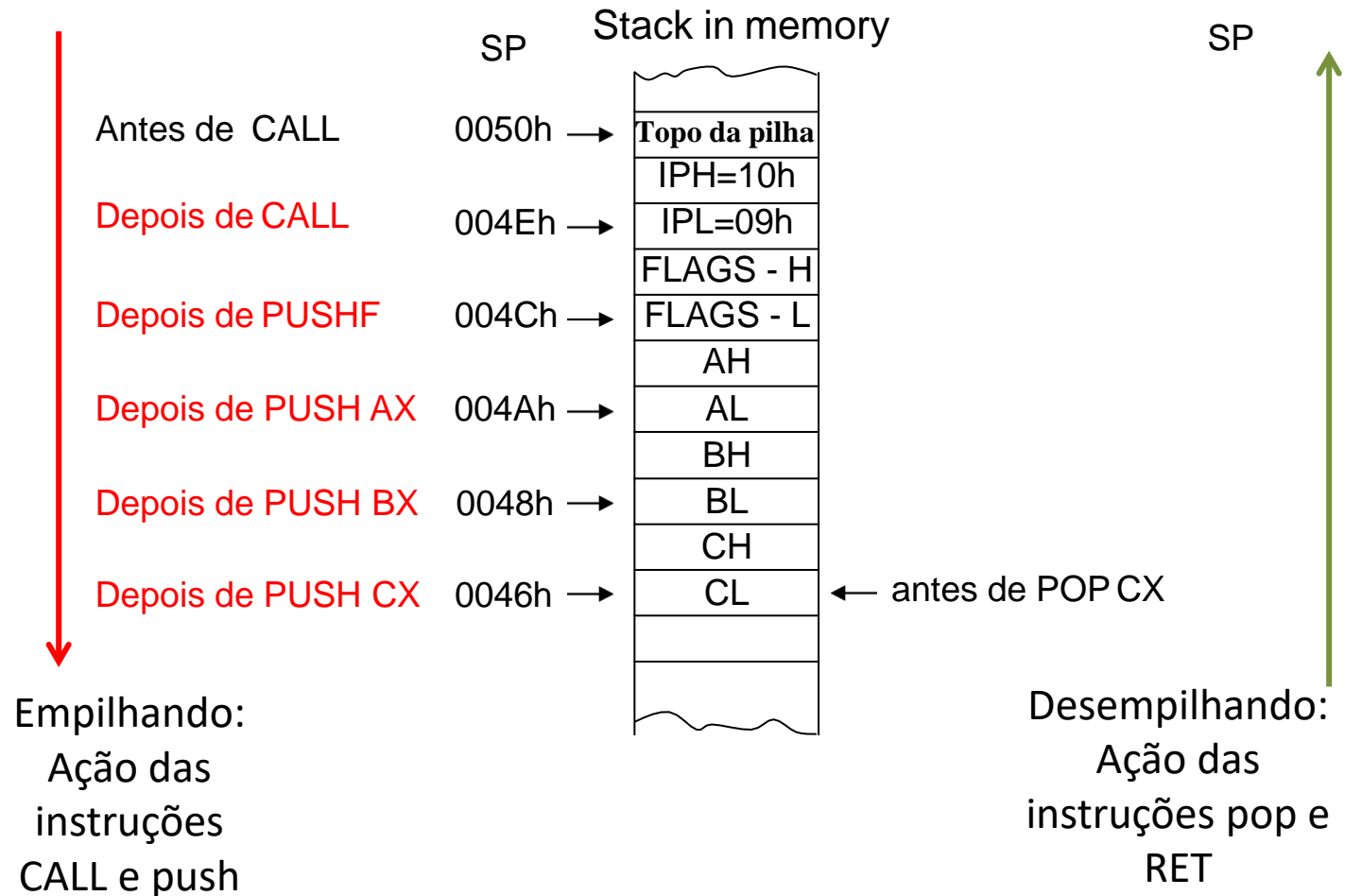
POP CX

POP BX

POP AX

POPF

RET



# CALL, PUSH, POP e RET (1)

## Ex: Rotina EXEMPLO

## Execução de EXEMPLO (call tipo near) e seus efeitos na pilha

EXEMPLO:

PUSHF

PUSH AX

PUSH BX

PUSH CX

.

.

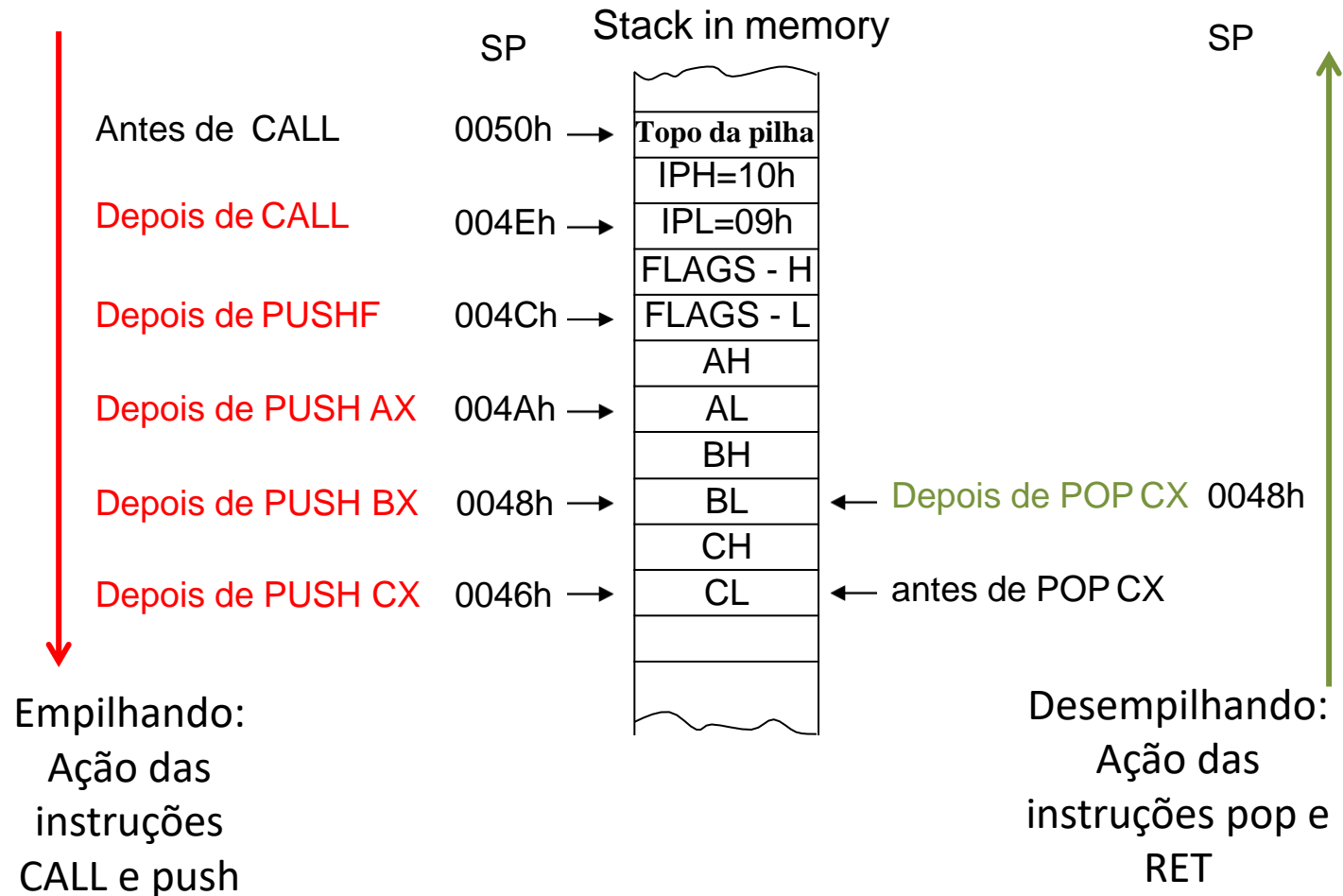
POP CX

POP BX

POP AX

POPF

RET



# CALL, PUSH, POP e RET (1)

## Ex: Rotina EXEMPLO

## Execução de EXEMPLO (call tipo near) e seus efeitos na pilha

EXEMPLO:

PUSHF

PUSH AX

PUSH BX

PUSH CX

.

.

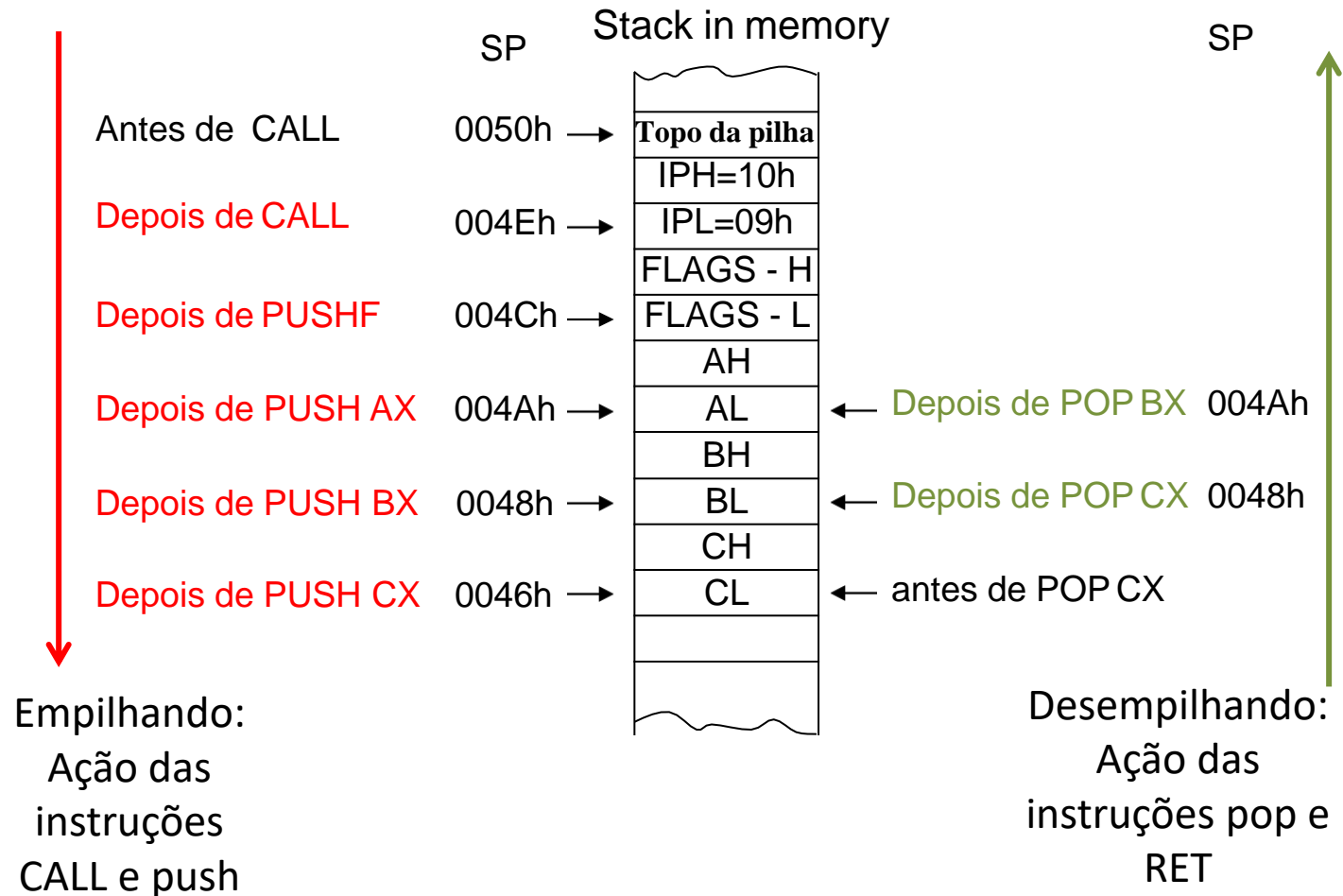
POP CX

POP BX

POP AX

POPF

RET





# CALL, PUSH, POP e RET (1)

## Ex: Rotina EXEMPLO

## Execução de EXEMPLO (call tipo near) e seus efeitos na pilha

EXEMPLO:

PUSHF

PUSH AX

PUSH BX

PUSH CX

.

.

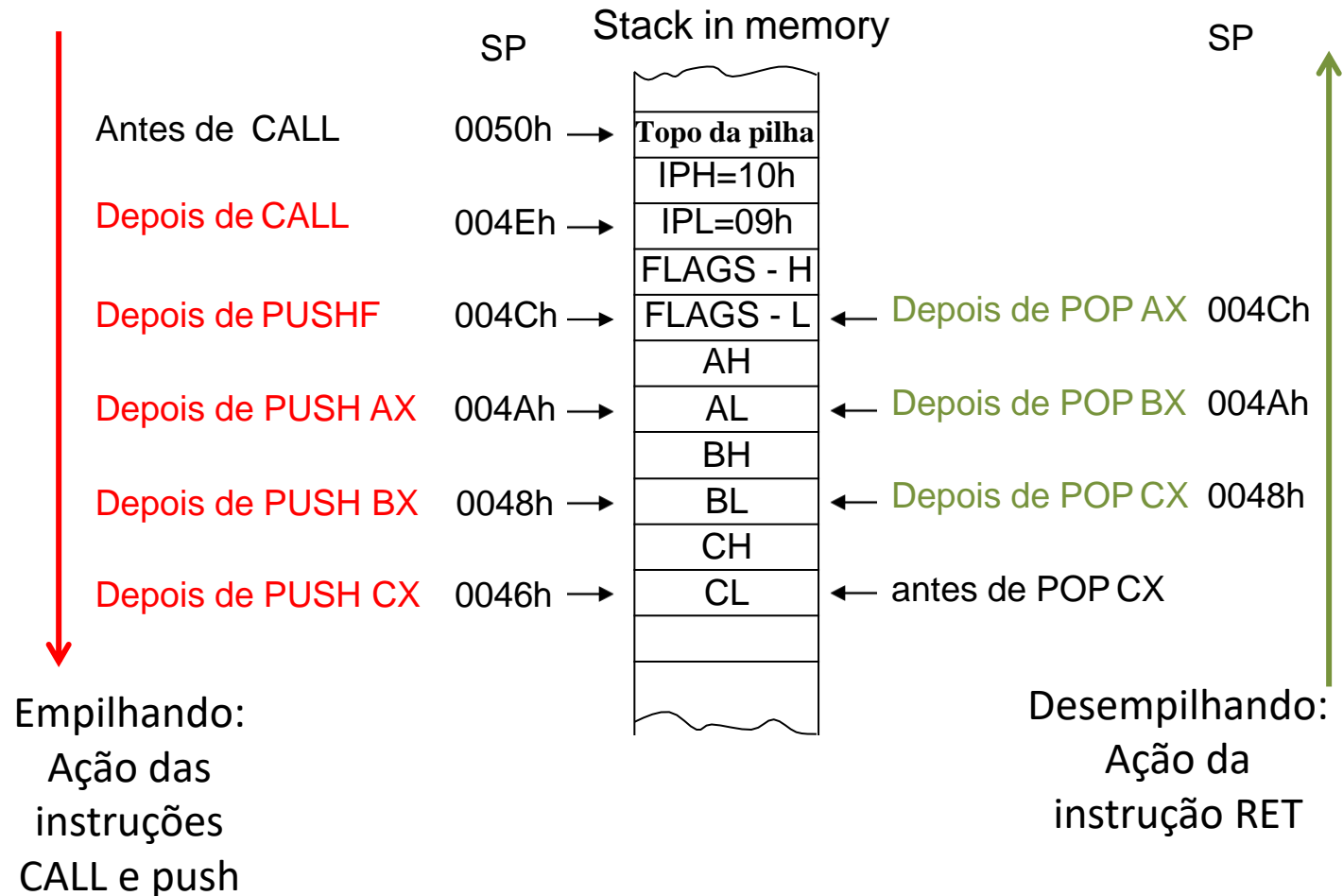
POP CX

POP BX

POP AX

POPF

RET



# CALL, PUSH, POP e RET (1)

## Ex: Rotina EXEMPLO

## Execução de EXEMPLO (call tipo near) e seus efeitos na pilha

EXEMPLO:

PUSHF

PUSH AX

PUSH BX

PUSH CX

.

.

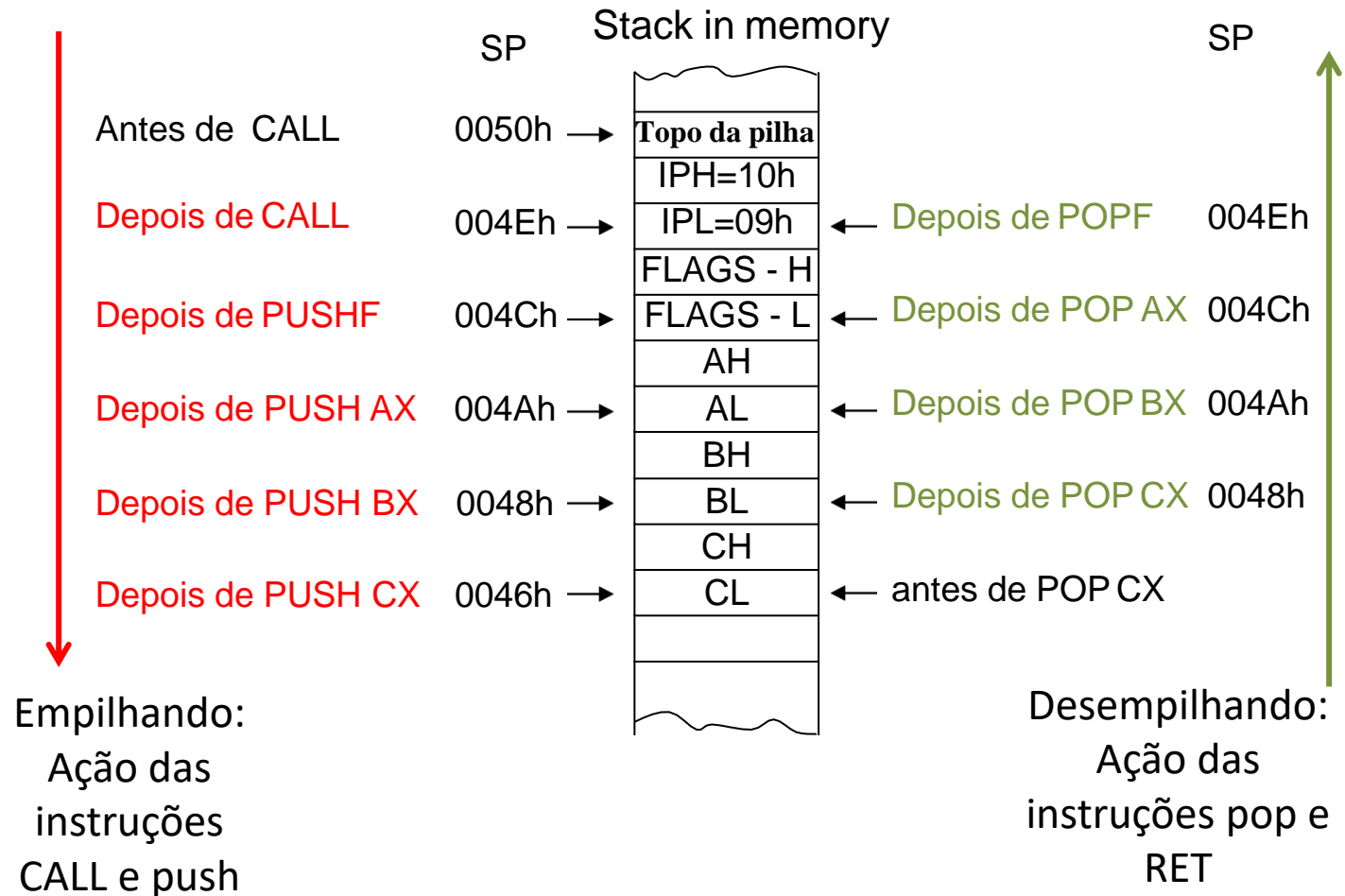
POP CX

POP BX

POP AX

POPF

RET



# CALL, PUSH, POP e RET (1)

## Ex: Rotina EXEMPLO

## Execução de EXEMPLO (call tipo near) e seus efeitos na pilha

EXEMPLO:

PUSHF

PUSH AX

PUSH BX

PUSH CX

.

.

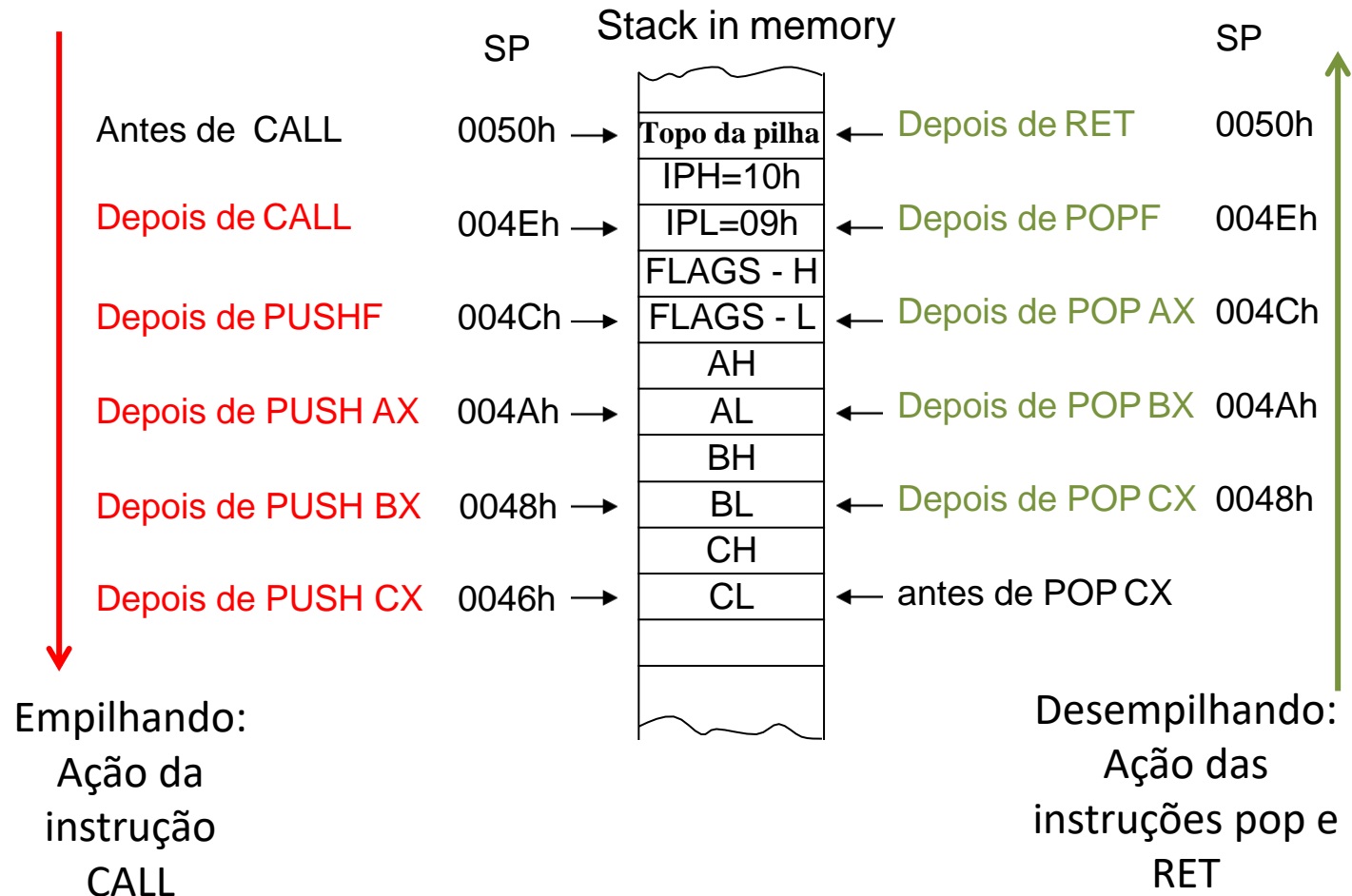
POP CX

POP BX

POP AX

POPF

RET



# Exemplo: Passagem de parâmetros pela Pilha (Stack)

Endereço de memória –  
par cs:ip

CS : IP

<u>Trecho de um Programa Principal que faz um "call"</u>	
075F:1000H	mov ax, 1000H
075F:1003H	push ax
075F:1004H	mov ax, ABCDH
075F:1007H	push ax
075F:1008H	call pixel_xy
075F:100BH	...

Endereço de memória de  
pixel\_xy  
par cs:ip

CS : IP

<u>Procedure: pixel_xy</u>	
pixel_xy:	
075F:2001H	PUSH BP
075F:2002H	MOV BP,SP
075F:2004H	MOV AX, 0
075F:0007H	.
....	.
075F:0055H	POP BP
075F:0056H	RET 4

Suponha que pixel\_xy esteja localizado para  
aonde aponta o par CS:IP = 075FH:2001H

# CALL, PUSH, POP e RET (2)

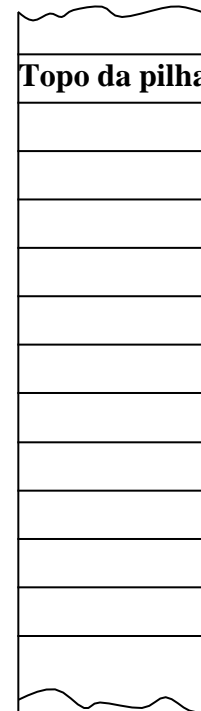
## Execução de pixel\_xy (call tipo near) e seus efeitos na pilha

```
....  
mov    ax, 1000H  
push   ax  
mov    ax, ABCDH  
push   ax  
call   pixel_xy  
...
```

```
pixel_xy:  
PUSH BP  
MOV BP,SP  
  
....  
POP BP  
RET 4
```

Antes de CALL

SP      Stack in memory  
0050h →



Empilhando:  
Ação das  
instruções  
CALL e push

Desempilhando:  
Ação das  
instruções pop e  
RET

# CALL, PUSH, POP e RET (2)

## Execução de EXEMPLO (call tipo near) e seus efeitos na pilha

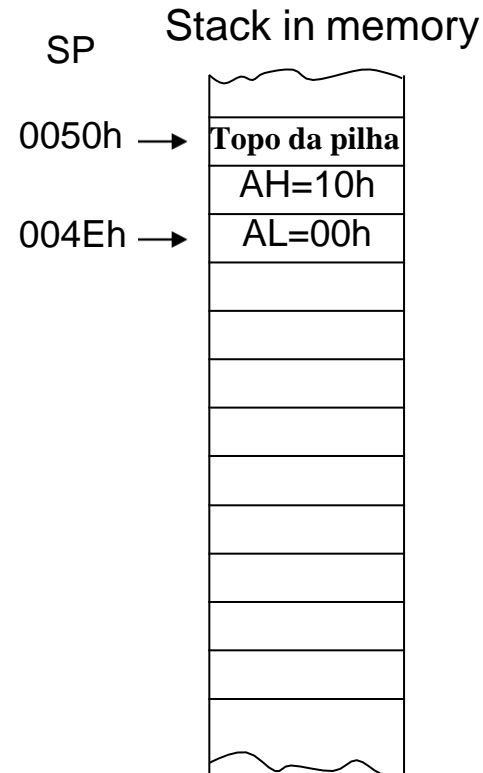
```
....  
mov    ax, 1000H  
push  ax  
mov    ax, ABCDH  
push   ax  
call   pixel_xy  
....
```

```
pixel_xy:  
PUSH BP  
MOV  BP,SP  
  
....  
POP  BP  
RET  4
```

Antes de CALL

Depois de **push ax**

Empilhando:  
Ação das  
instruções  
CALL e push



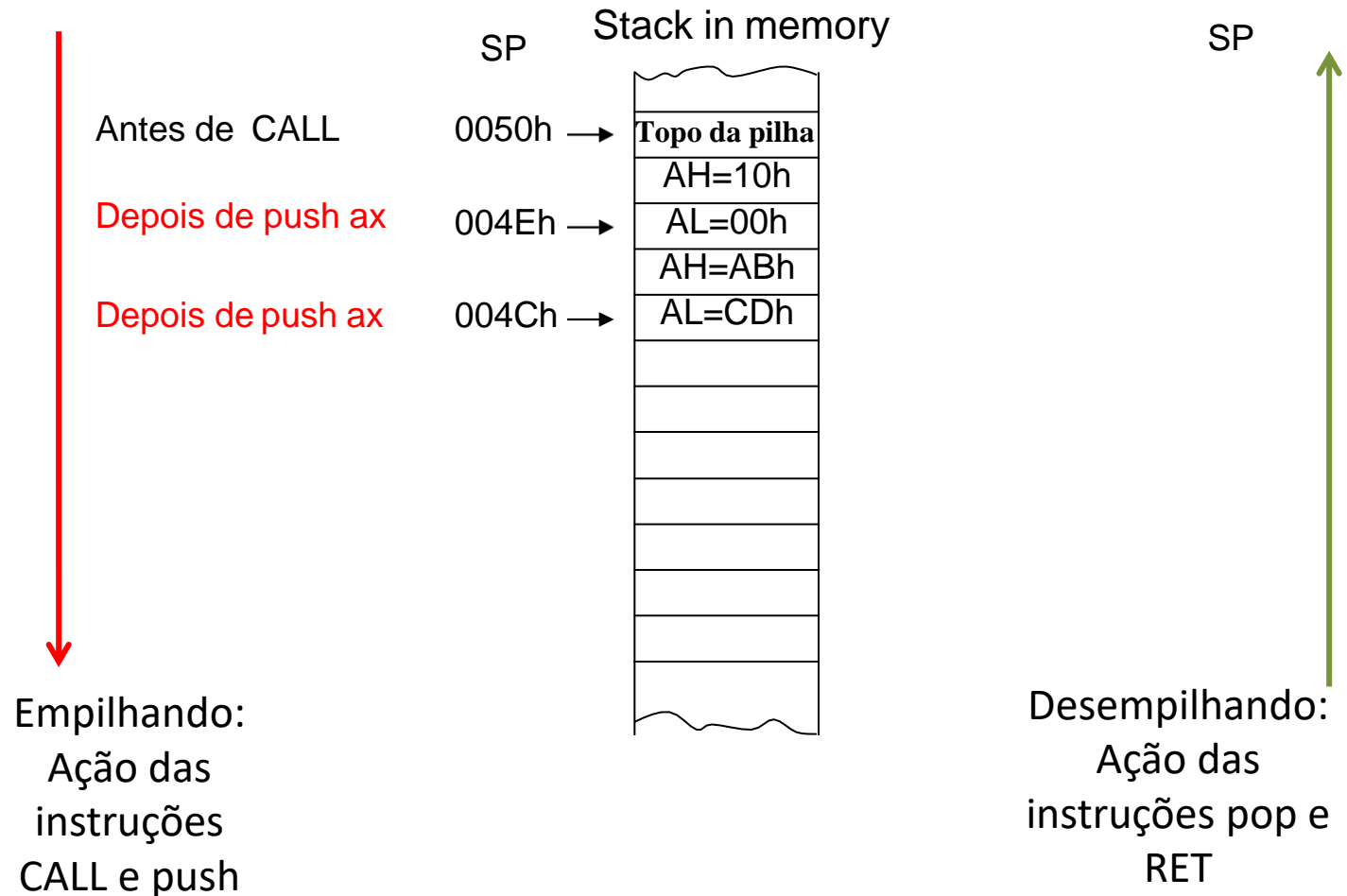
Desempilhando:  
Ação das  
instruções pop e  
RET

# CALL, PUSH, POP e RET (2)

## Execução de pixel\_xy (call tipo near) e seus efeitos na pilha

```
....  
mov    ax, 1000H  
push   ax  
mov    ax, ABCDH  
push   ax  
call   pixel_xy  
...
```

```
pixel_xy:  
PUSH BP  
MOV BP,SP  
  
....  
POP BP  
RET 4
```

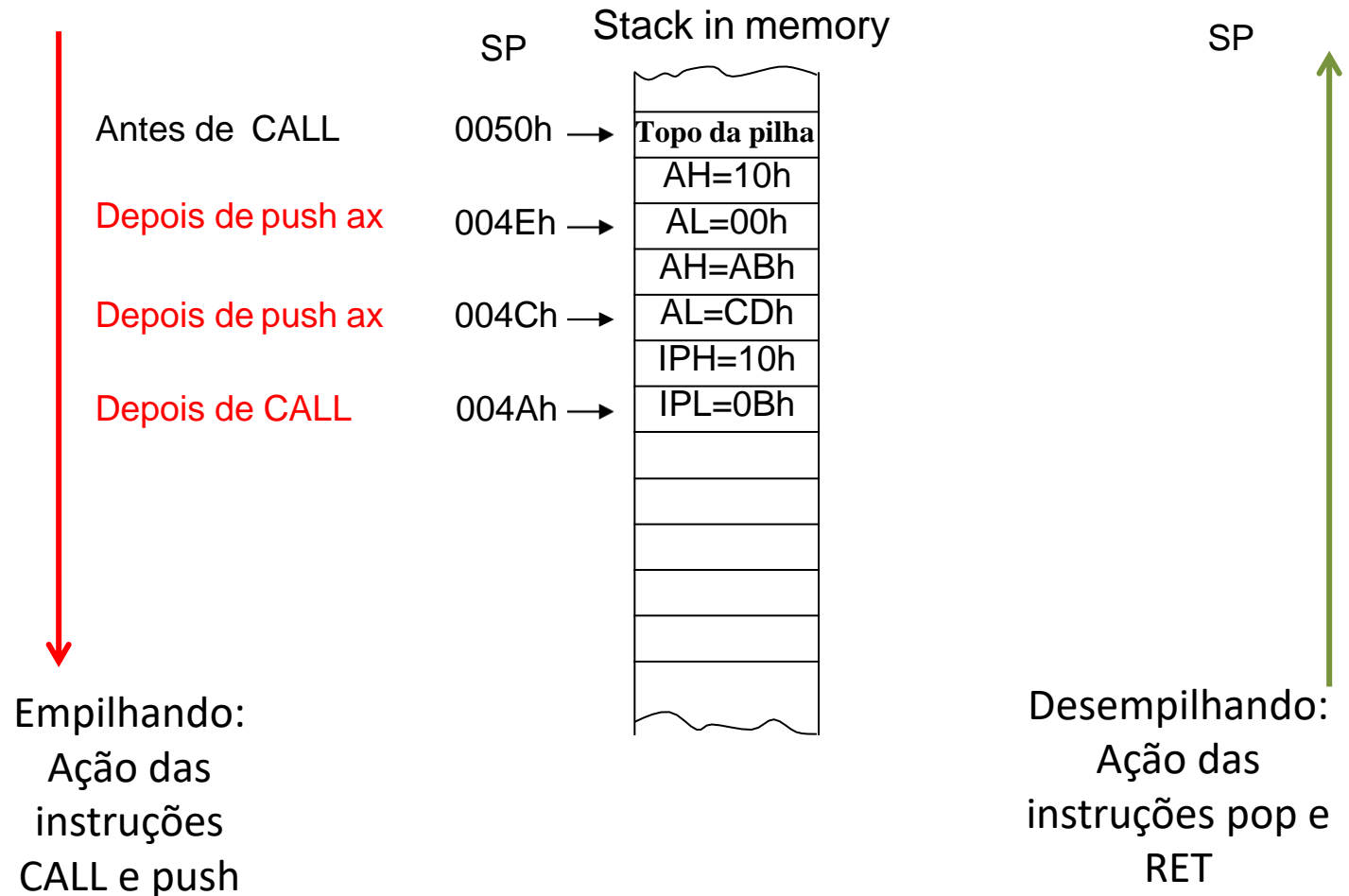


# CALL, PUSH, POP e RET (2)

## Execução de pixel\_xy (call tipo near) e seus efeitos na pilha

```
....  
mov    ax, 1000H  
push   ax  
mov    ax, ABCDH  
push   ax  
call   pixel_xy  
...
```

```
pixel_xy:  
PUSH BP  
MOV BP,SP  
  
....  
POP BP  
RET 4
```



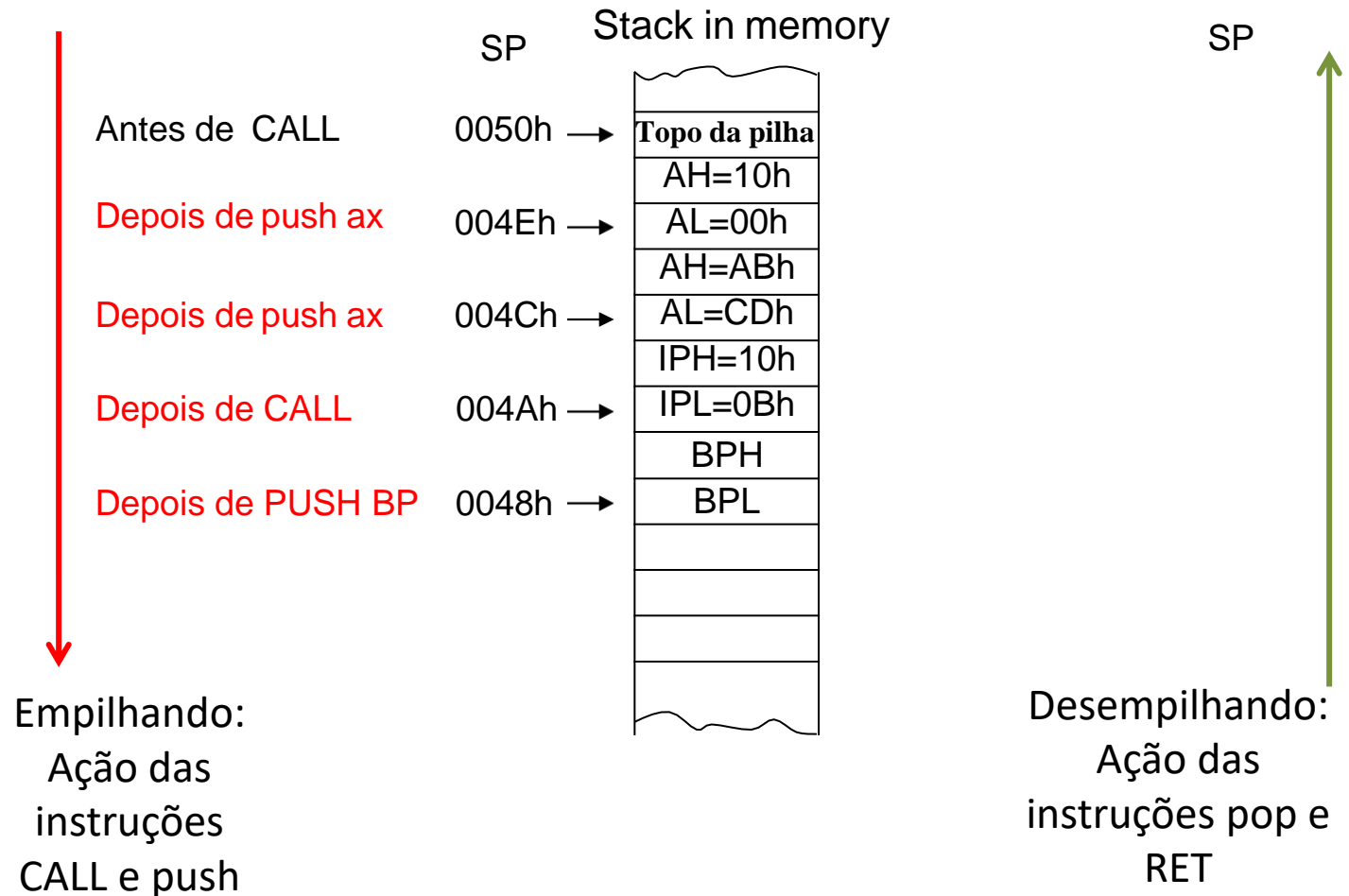


# CALL, PUSH, POP e RET (2)

## Execução de pixel\_xy (call tipo near) e seus efeitos na pilha

```
....  
mov    ax, 1000H  
push   ax  
mov    ax, ABCDH  
push   ax  
call   pixel_xy  
...
```

```
pixel_xy:  
PUSH BP  
MOV    BP,SP  
  
....  
POP    BP  
RET    4
```

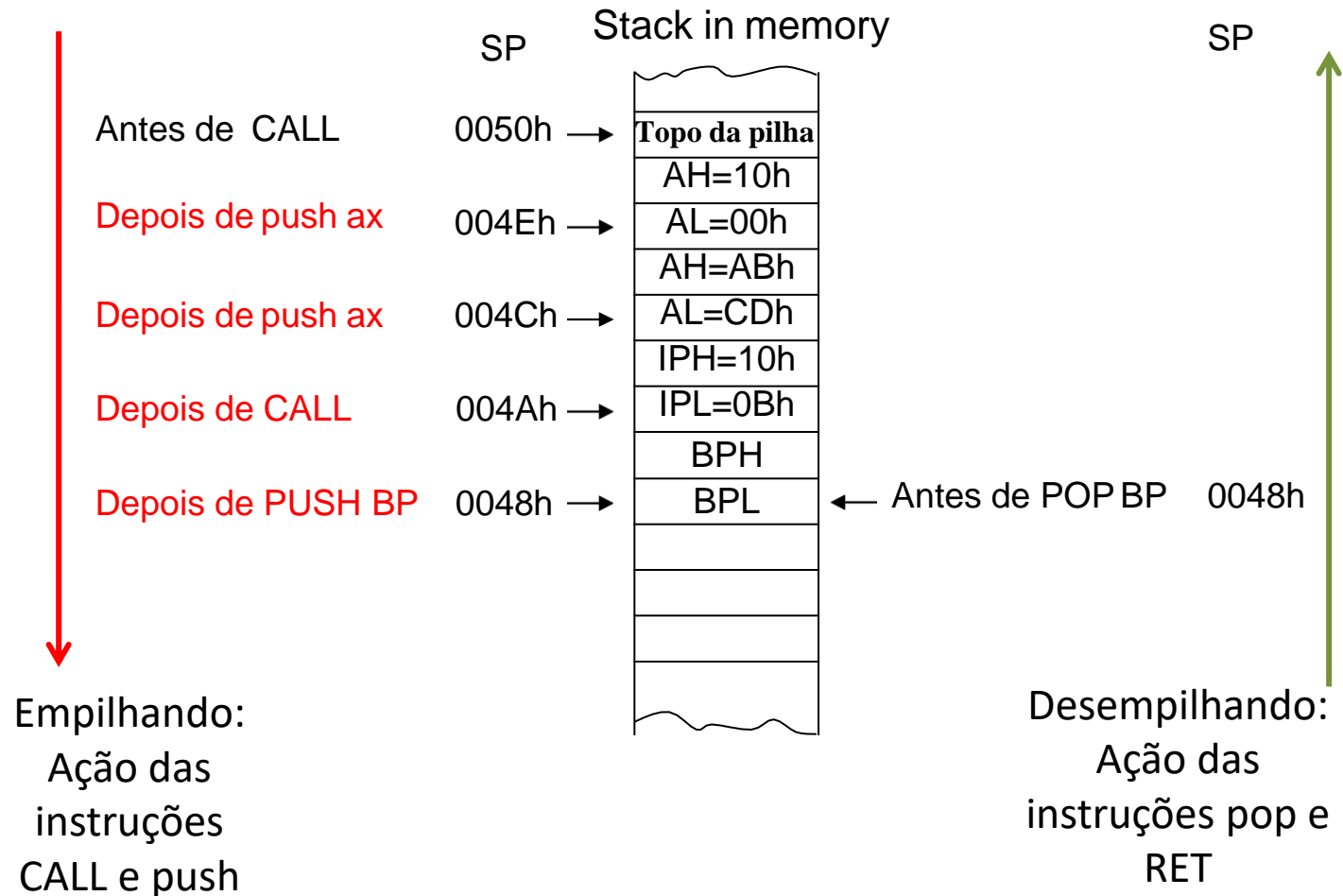


# CALL, PUSH, POP e RET (2)

## Execução de pixel\_xy (call tipo near) e seus efeitos na pilha

```
....  
mov ax, 1000H  
push ax  
mov ax, ABCDH  
push ax  
call pixel_xy  
....
```

```
pixel_xy:  
PUSH BP  
MOV BP,SP  
  
....  
POP BP  
RET 4
```

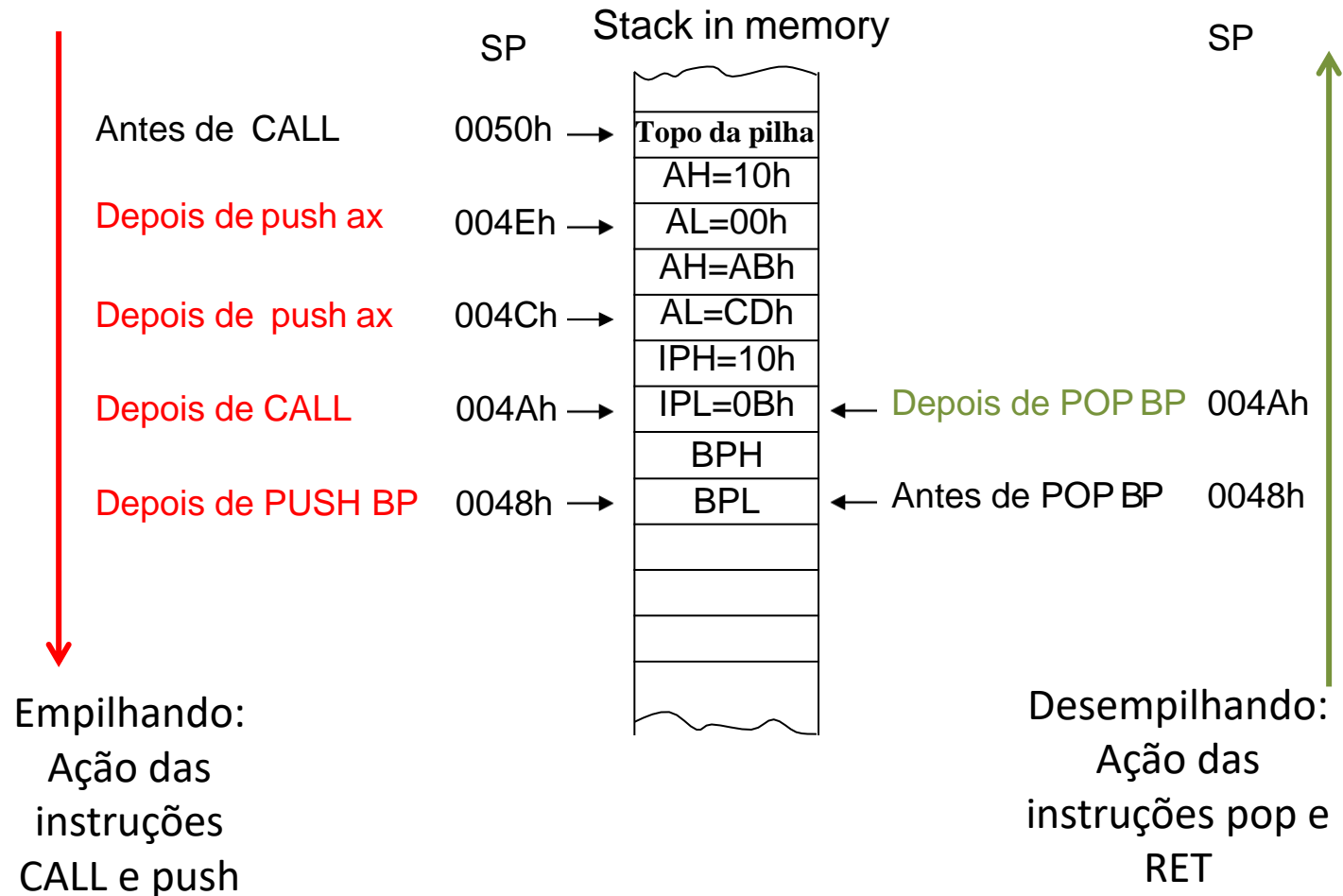


# CALL, PUSH, POP e RET (2)

## Execução de pixel\_xy (call tipo near) e seus efeitos na pilha

```
....  
mov    ax, 1000H  
push   ax  
mov    ax, ABCDH  
push   ax  
call   pixel_xy  
....
```

```
pixel_xy:  
PUSH BP  
MOV BP,SP  
  
....  
POP BP  
RET 4
```

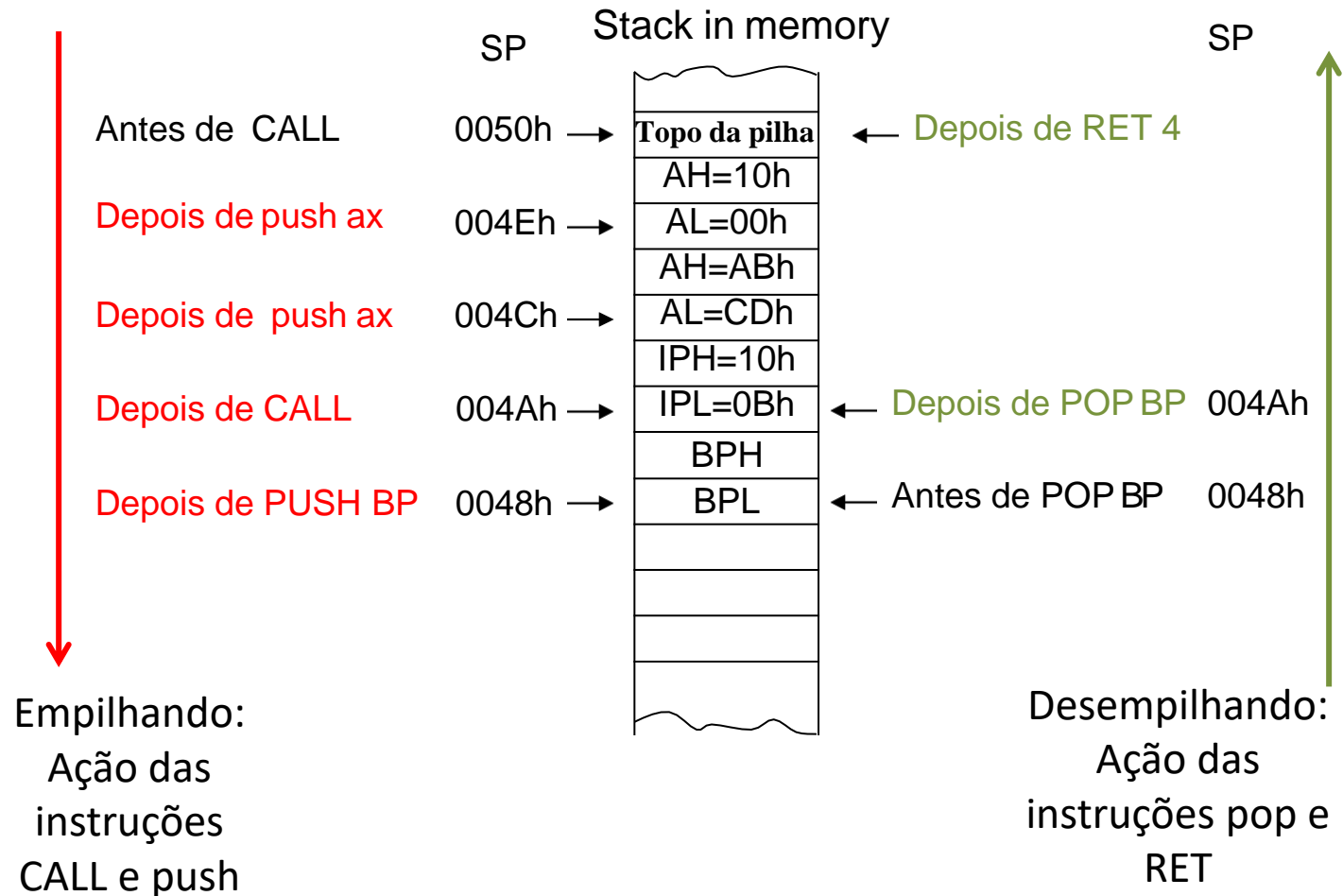


# CALL, PUSH, POP e RET (2)

## Execução de pixel\_xy (call tipo near) e seus efeitos na pilha

```
....  
mov ax, 1000H  
push ax  
mov ax, ABCDH  
push ax  
call pixel_xy  
....
```

```
pixel_xy:  
PUSH BP  
MOV BP,SP  
  
....  
POP BP  
RET 4
```



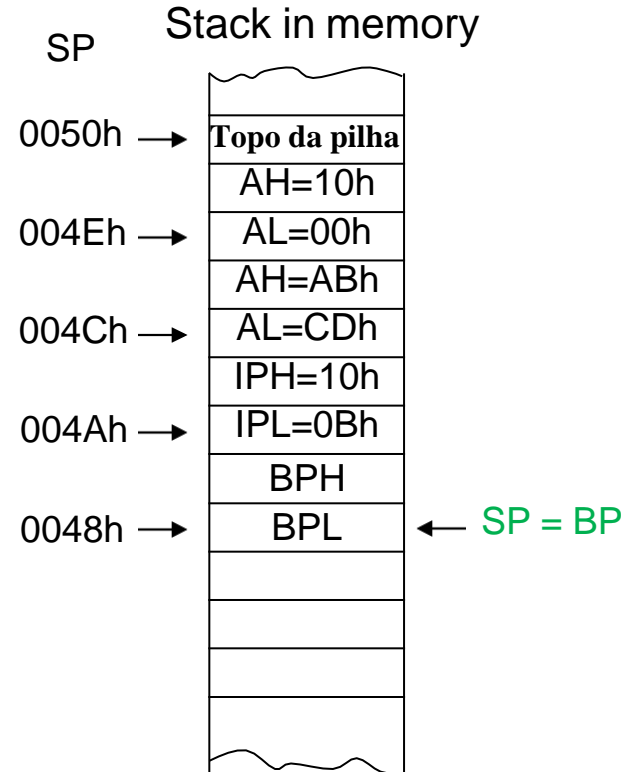
# Acessando parâmetros passados pela Pilha

## Execução de pixel\_xy (call tipo near) e seus efeitos na pilha

```
....  
mov    ax, 1000H  
push   ax  
mov    ax, ABCDH  
push   ax  
call   pixel_xy  
....
```

Antes de CALL

```
pixel_xy:  
PUSH BP  
MOV BP, SP  
MOV AX, WORD[BP+6]  
SUB  AX, WORD[BP+4]  
....  
POP BP  
RET 4
```



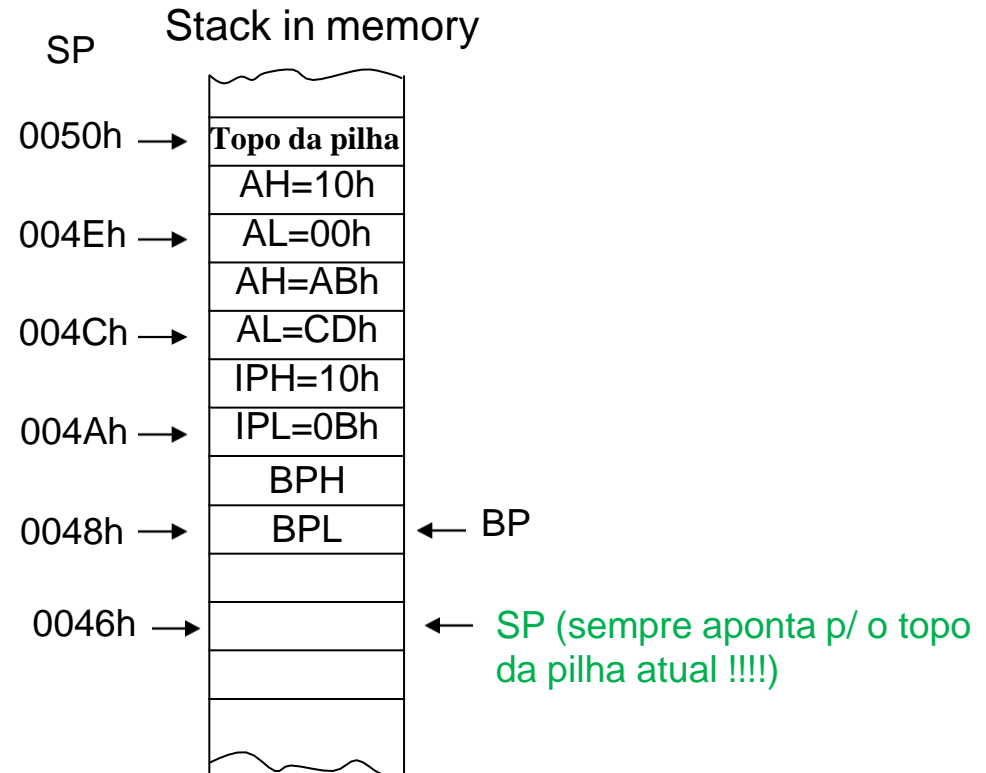
# Criando Variável Local na Pilha

```
....  
mov     ax, 1000H  
push    ax  
mov     ax, ABCDH  
push    ax  
call    pixel_xy  
....
```

## Execução de pixel\_xy (call tipo near) e seus efeitos na pilha

Antes de CALL

```
pixel_xy:  
    PUSH    BP  
    MOV     BP, SP  
    MOV     AX, WORD[BP+6]  
    SUB     AX, WORD[BP+4]  
    SUB     SP, 2  
    MOV     WORD[BP-2], 0xCADA  
    ....  
SAINDO DE PIXEL_XY:  
    ADD     SP, 2  
    POP     BP  
    RET     4
```



# Criando Variável Local na Pilha

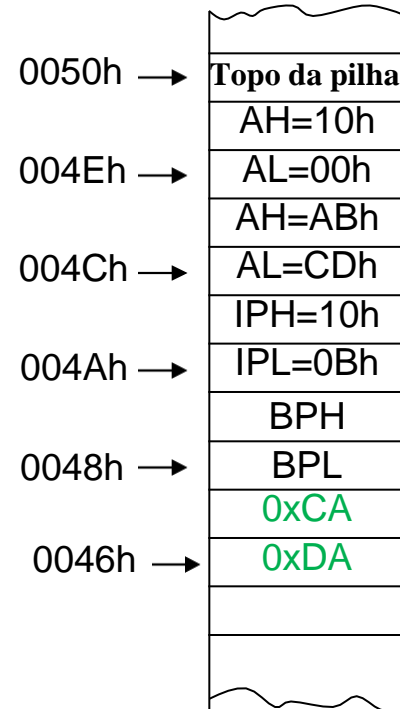
```
....  
mov     ax, 1000H  
push    ax  
mov     ax, ABCDH  
push    ax  
call    pixel_xy  
....
```

## Execução de pixel\_xy (call tipo near) e seus efeitos na pilha

Antes de CALL

```
pixel_xy:  
    PUSH    BP  
    MOV     BP, SP  
    MOV     AX, WORD[BP+6]  
    SUB     AX, WORD[BP+4]  
    SUB     SP, 2  
    MOV     WORD[BP-2], 0xCADA  
    ....  
;SAINDO DE PIXEL_XY:  
    ADD     SP, 2  
    POP     BP  
    RET     4
```

SP      Stack in memory



← BP

← BP-2 = SP  
(SP sempre aponta p/ o topo da pilha atual !!!!)

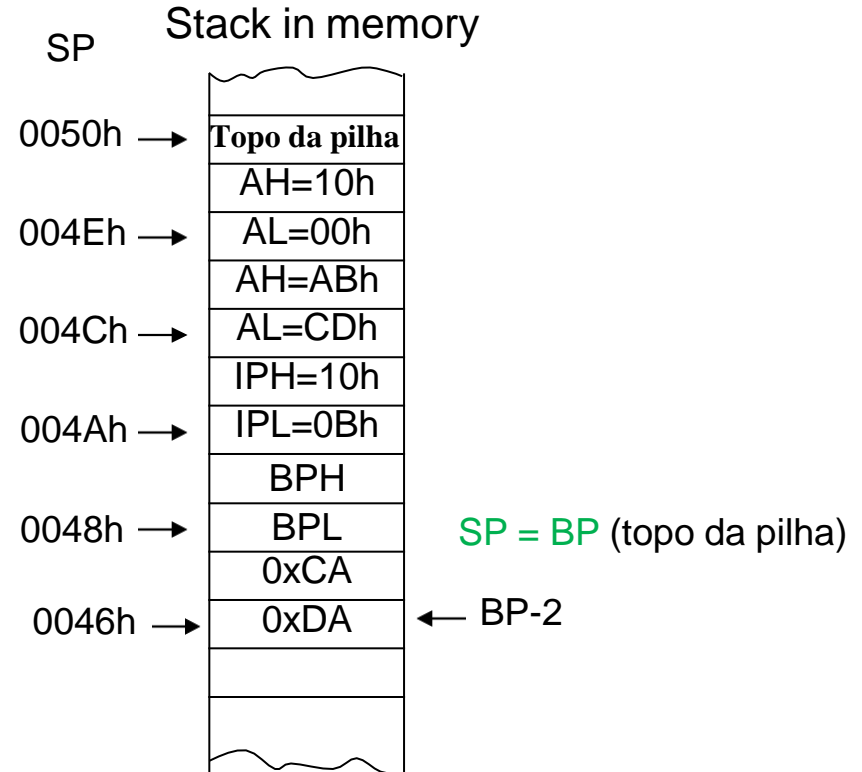
# Criando Variável Local na Pilha

```
....  
mov     ax, 1000H  
push    ax  
mov     ax, ABCDH  
push    ax  
call    pixel_xy  
....
```

## Execução de pixel\_xy (call tipo near) e seus efeitos na pilha

Antes de CALL

```
pixel_xy:  
    PUSH    BP  
    MOV     BP, SP  
    MOV     AX, WORD[BP+6]  
    SUB     AX, WORD[BP+4]  
    SUB     SP, 2  
    MOV     WORD[BP-2], 0xCADA  
    ....  
  
SAINDO:  
    ADD     SP, 2  
    POP     BP  
    RET     4
```





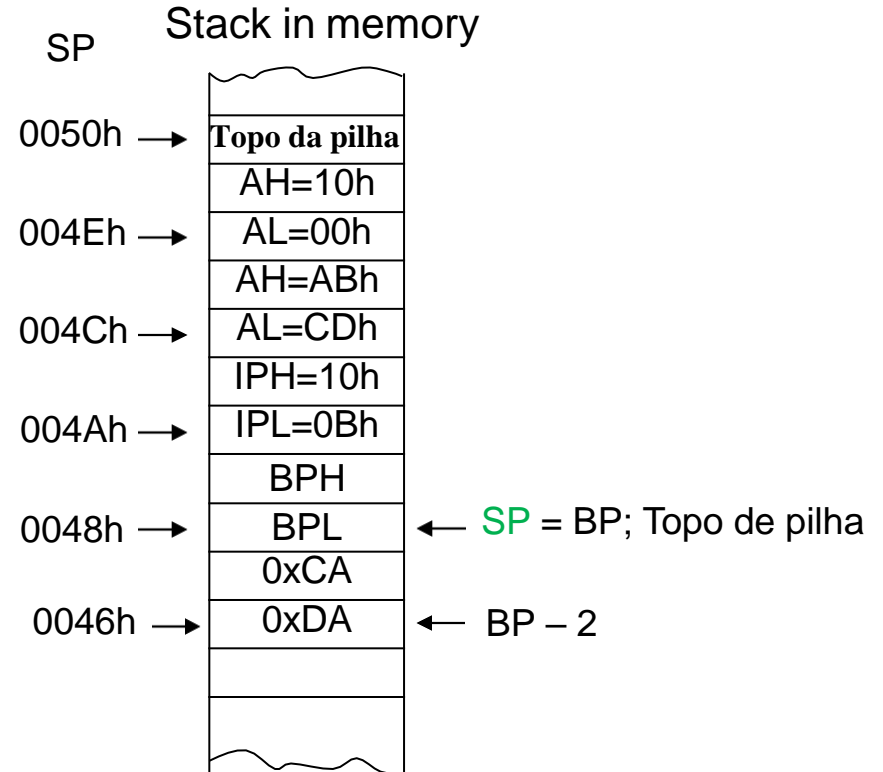
# Criando Variável Local na Pilha

```
....  
mov     ax, 1000H  
push    ax  
mov     ax, ABCDH  
push    ax  
call    pixel_xy  
....
```

## Execução de pixel\_xy (call tipo near) e seus efeitos na pilha

Antes de CALL

```
pixel_xy:  
    PUSH    BP  
    MOV     BP,SP  
    MOV     AX, WORD[BP+6]  
    SUB     AX, WORD[BP+4]  
    SUB     SP, 2  
    MOV     WORD[BP-2], 0xCADA  
    ....  
  
SAINDO:  
    ADD     SP, 2  
    POP     BP  
    RET     4
```



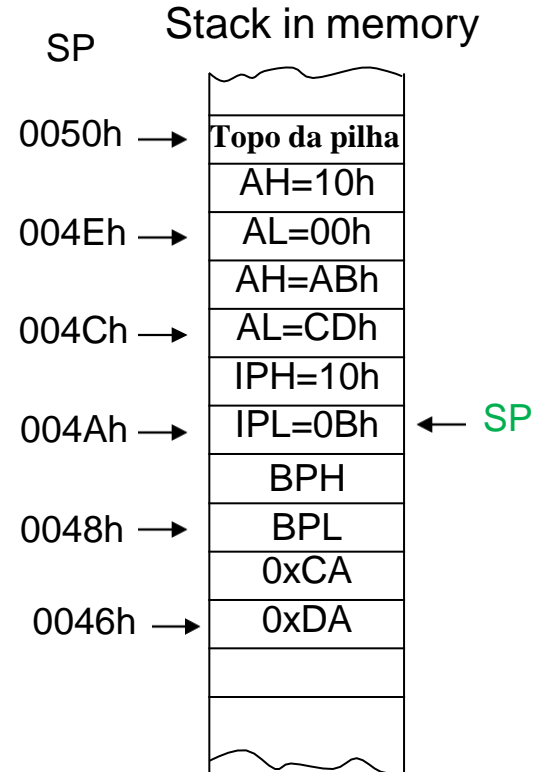
# Criando Variável Local na Pilha

```
....  
mov     ax, 1000H  
push    ax  
mov     ax, ABCDH  
push    ax  
call    pixel_xy  
....
```

## Execução de pixel\_xy (call tipo near) e seus efeitos na pilha

Antes de CALL

```
pixel_xy:  
    PUSH    BP  
    MOV     BP, SP  
    MOV     AX, WORD[BP+6]  
    SUB     AX, WORD[BP+4]  
    SUB     SP, 2  
    MOV     WORD[BP-2], 0xCADA  
    ....  
  
SAINDO:  
    ADD     SP, 2  
    POP     BP  
    RET     4
```



A partir de agora BP não será mais representado pois seu valor foi restaurado com a execução de **POP BP**

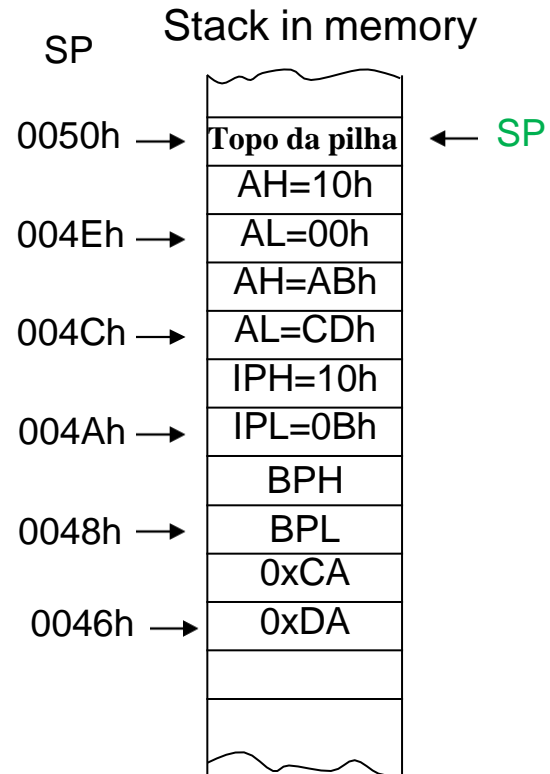
# Criando Variável Local na Pilha

```
....  
mov     ax, 1000H  
push    ax  
mov     ax, ABCDH  
push    ax  
call    pixel_xy  
....
```

## Execução de pixel\_xy (call tipo near) e seus efeitos na pilha

Antes de CALL

```
pixel_xy:  
    PUSH    BP  
    MOV     BP, SP  
    MOV     AX, WORD[BP+6]  
    SUB     AX, WORD[BP+4]  
    SUB     SP, 2  
    MOV     WORD[BP-2], 0xCADA  
    ....  
  
SAINDO:  
    ADD     SP, 2  
    POP     BP  
    RET     4
```



# Instruções de Interrupção

- O programa pode gerar uma interrupção de software usando as instruções INT ou INTO
- No tocante ao uso da PILHA, *Interrupts (INT)* se assemelham a *Procedures* exceto que uma INT empilha automaticamente mais registradores na *PILHA*. Durante o atendimento de uma INT o  $\mu P$  faz:
  - (1) Empilha (PUSH) o registrador de FLAGS para o STACK
  - (2) Desabilita as *flags* T (Trap flag) e IF (*interrupt flag*)
  - (3) Empilha (PUSH) CS para o STACK
  - (4) Carrega o novo valor de CS a partir da tabela de interrupção
  - (5) Empilha (PUSH) IP para o STACK
  - (6) Carrega o novo valor para IP a partir da tabela de interrupção
  - (7) Executar instrução a partir do novo valor de CS:IP

# Interrupt Vector Table

- O operando de 8 bits depois da instrução INT especifica o tipo de interrupção.
- O endereço dos procedimentos de tratamento de interrupção é armazenado em uma tabela de vetores de interrupção no início da memória, e pode ser derivado multiplicando o vetor de interrupção por 4.
- As funções das interrupções de 0 a 31 são reservadas pela Intel para funções especiais para o 8086-Pentium. As funções de interrupções 32 em diante podem ser definidas pelo usuário.
- Algumas das funções de interrupção especificadas pela Intel diferem das funções de interrupção reais que são implementadas no PC IBM. As funções de interrupção especificadas pela Intel são listadas a seguir:

# Tabela de Interrupções

Vector Number	Address	Microprocessor	Function
0	0H-3H	8086-80486/Pentium	Divide error
1	4H-7H	8086-80486/Pentium	Single step
2	8H-BH	8086-80486/Pentium	NMI (hardware interrupt)
3	CH-FH	8086-80486/Pentium	Breakpoint
4	10H-13H	8086-80486/Pentium	Interrupt on overflow
5	14H-17H	80286-80486/Pentium	BOUND interrupt
6	18H-1BH	80286-80486/Pentium	Invalid opcode
7	1CH-1FH	80286-80486/Pentium	Coprocessor emulation interrupt
8	20H-23H	80386-80486/Pentium	Double fault
9	24H-27H	80386	Coprocessor segment overrun
10	28H-2BH	80386-80486/Pentium	Invalid task state segment
11	2CH-2FH	80386-80486/Pentium	Segment not present
12	30H-33H	80386-80486/Pentium	Stack fault
13	34H-37H	80386-80486/Pentium	General protection fault
14	38H-3BH	80386-80486/Pentium	Page fault
15	3CH-3FH		Reserved*
16	40H-43H	80286-80486/Pentium	Floating-point error
17	44H-47H	80486SX	Alignment check interrupt
18	48H-4BH	Pentium	Machine check exception
19-31	4CH-7FH	8086-80486/Pentium	Reserved*
32-255	80H-3FFH	8086-80486/Pentium	User interrupts

# Tabela de Interrupção

Memory Address	Table Entry	Vector Definition	
3FE	CS 255	Vector 255 <sub>10</sub>	User Available
3FC	IP 255		
≈	≈		
82	CS 32	Vector 32 <sub>10</sub>	
80	IP 32		
7E	CS 31	Vector 31 <sub>10</sub>	
7C	IP 31		
≈	≈		
16	CS 5	Vector 5	Reserved
14	IP 5		
12	CS 4	Vector 4 – Overflow	
10	IP 4		
0E	CS 3	Vector 3 - Breakpoint	
0C	IP 3		
0A	CS 2	Vector 2 - NMI	
08	IP 2		
06	CS 1	Vector 1 – Single Step	
04	IP 1		
02	CS Value – Vector 0 (CS 0)	Vector 0 – Divide Error	
00	IP Value – Vector 0 (IP 0)		
	← 2 Bytes →		

# Atribuições de Interrupção no PC IBM e compatíveis

- ⌘ To return to DOS, assembler programs on IBM PCs must end with INT 21. Register AH should contain 4CH to specify the DOS terminate functions.

Number	Function
0	Divide error
1	Single step (debug)
2	Nonmaskable interrupt pin
3	Breakpoint
4	Arithmetic overflow
5	Print screen key and BOUND instruction
6	Illegal instruction error
7	Coprocessor not present interrupt
8	Clock tick (hardware) (Approximately 18.2 Hz)
9	Keyboard (hardware)
A	Hardware interrupt 2 (system bus) (cascade in AT)
B - F	Hardware interrupt 3-7 (system bus)
10	Video BIOS
11	Equipment environment
12	Conventional memory size
13	Direct disk service
14	Serial COM port service
15	Miscellaneous service
16	Keyboard service
17	Parallel port LPT service
18	ROM BASIC
19	Reboot
1A	Clock service
1B	Control-bread handler
1C	User timer service
1D	Pointer for video parameter table
1E	Pointer for disk drive parameter table
1F	Pointer for graphics character pattern table
20	Terminate program
21	DOS service
22	Program termination handler
23	Control C handler
24	critical error handler
25	Read disk
26	Write disk
27	Terminate and stay resident
28	DOS idle
2F	Multiplex handler
70-770	Hardware interrupts 8-15 (AT style computer)



# Instruções de Interrupção

Mnemonic	Meaning	Format	Operation	Flags Affected
CLI	Clear interrupt flag	CLI	$0 \rightarrow (IF)$	IF
STI	Set interrupt flag	STI	$1 \rightarrow (IF)$	IF
INT $n$	Type $n$ software interrupt	INT $n$	$(Flags) \rightarrow ((SP) - 2)$ $0 \rightarrow TF, IF$ $(CS) \rightarrow ((SP) - 4)$ $(2 + 4 \cdot n) \rightarrow (CS)$ $(IP) \rightarrow ((SP) - 6)$ $(4 \cdot n) \rightarrow (IP)$	TF, IF
IRET	Interrupt return	IRET	$((SP)) \rightarrow (IP)$ $((SP) + 2) \rightarrow (CS)$ $((SP) + 4) \rightarrow (Flags)$ $(SP) + 6 \rightarrow (SP)$	All
INTO	Interrupt on overflow	INTO	INT 4 steps	TF, IF
HLT	Halt	HLT	Wait for an external Interrupt or reset to occur	None
WAIT	Wait	WAIT	Wait for TEST input to go active	None

SP: stack pointer

# **Estado interno do 8088/8086 após a sua inicialização/reset**

CPU COMPONENT	CONTENT
Flags	Clear
Instruction Pointer	0000h
CS Register	FFFFh
DS Register	0000h
SS Register	0000h
ES Register	0000h
Queue	Empty