4. LINGUAGEM ASSEMBLY

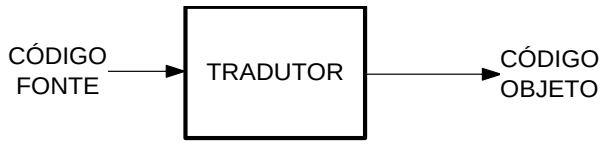
4.1. INTRODUÇÃO:

Um computador digital é uma máquina capaz de nos solucionar problemas através da execução de instruções que lhe são fornecidas. Denomina-se **programa** uma seqüência de instruções que descreve como executar uma determinada tarefa. Os **circuitos eletrônicos** de cada computador pode reconhecer e executar diretamente um **conjunto limitado de instruções simples** para quais todos os programas devem ser convertidos antes que eles possam ser executados. Estas instruções básicas raramente são mais complicadas do que:

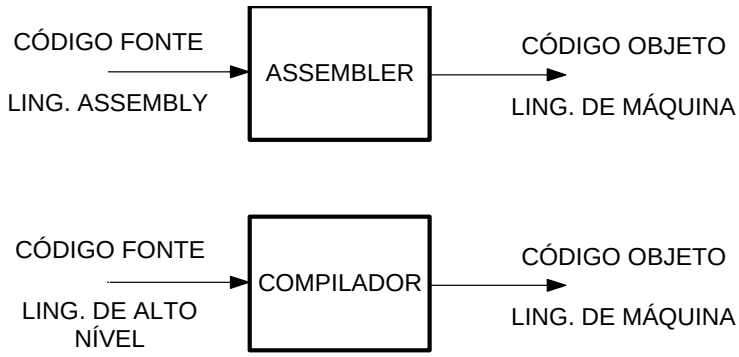
- Some dois números.
- Verifique se um número é zero.
- Mova um dado de uma parte da memória do computador para outra.

Juntas, as instruções primitivas do computador formam uma linguagem que torna possível às pessoas se comunicarem com o computador. Estas instruções são codificadas na forma de **zeros e uns** que são armazenadas em uma determinada posição na memória e constituem na única forma de programa compreendida pelo computador. Esta forma de programa é dito **código objeto** e a linguagem de zeros e uns no qual o código objeto é escrito é denominado **linguagem de máquina**.

Praticamente falando, escrever um programa em termos de zeros e uns é uma tarefa tediosa, repetitiva e sujeita a erros. Portanto, ao invés de tentar escrever o programa na linguagem de máquina, é mais sensato escrever o programa em uma linguagem mais familiar para nós seres humanos, e então usar um computador para traduzi-las em um código objeto. Um programa escrito nesta linguagem mais familiar é chamado código fonte, programa fonte ou simplesmente fonte. O programa que traduz o código fonte em código objeto é chamado tradutor, conforme ilustrado na figura abaixo:



Existem **duas formas distintas de linguagens** nas quais podemos escrever um código fonte. Estas são chamadas: **linguagens assembly** e **linguagens de alto nível** e serão descritas a seguir. Os tradutores correspondentes são chamados "assemblers" e compiladores, como mostrado na figura a seguir:

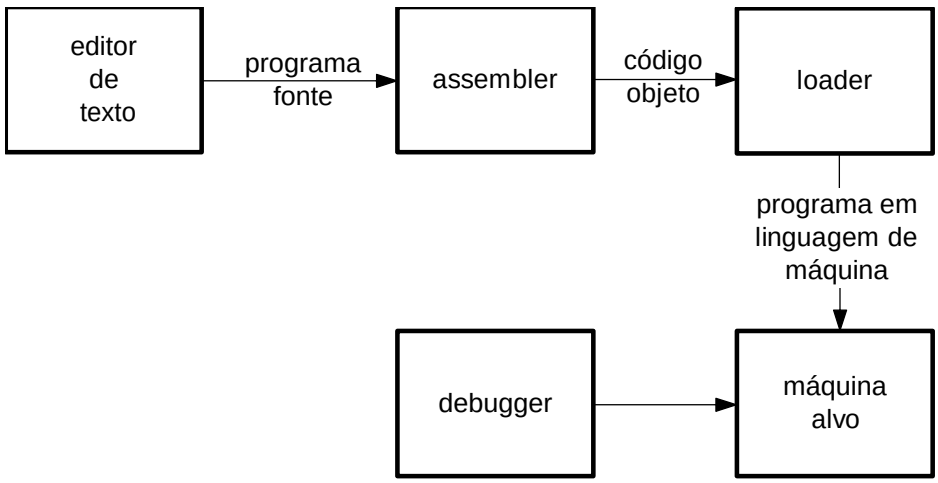


O processo de tradução poderia envolver a execução de alguma atividade final de limpeza antes da saída se tornar verdadeiramente um código de máquina. Estas atividades de limpeza são partes de processo de tradução mas, infelizmente, tem sido dado nomes distintos como relocação e linkagem. Em nosso curso, referências ao processo de tradução irão incluir toda a atividade de limpeza também.
Um programa escrito em linguagem assembly é uma **representação simbólica do programa em linguagem de máquina**. A relação entre as instruções no programa em linguagem assembly e o código objeto resultante é bastante óbvia. Uma linguagem de alto nível, por outro lado, é um dialeto não ambíguo e formalizado de alguma linguagem natural (tipicamente inglês). A relação entre as instruções na linguagem de alto nível e o código objeto resultante frequentemente não é muito evidente. A linguagem assembly dá ao programador o controle completo sobre o código objeto resultante e portanto permite a geração de um código objeto eficiente (desde que o programador seja muito eficiente). Uma linguagem de alto nível livra o programador de ter que pensar sobre o código objeto resultante e permite que o mesmo se concentre no problema que está tentando resolver. O programador está a mercê do compilador no que diz respeito à geração de um código objeto eficiente. Um compilador muito bom pode algumas vezes gerar um código objeto mais eficiente que o gerado através de um programa assembly.

4.2. LINGUAGEM ASSEMBLY

LINGUAGEM DE MÁQUINA: conjunto de instruções codificadas na forma de zeros e uns;
LINGUAGEM ASSEMBLY: é uma ferramenta de software, uma linguagem simbólica que pode ser diretamente traduzida em linguagem de máquina através de um programa tradutor chamado ASSEMBLER.

Processo de desenvolvimento de um programa usando a linguagem assembly:



O Loader, o Debugger e o programa em linguagem de máquina devem rodar na máquina ALVO; o editor de texto e o assembler podem rodar na máquina alvo ou uma outra máquina diferente. Um assembler que roda em uma máquina e produz códigos objetos para outra é chamado de CROSS-ASSEMBLER.

4.3. CARACTERÍSTICAS DA LINGUAGEM ASSEMBLY

Embora todo fabricante de microprocessadores defina uma linguagem assembly padrão para uma nova máquina quando esta sendo projetada, outros usuários podem definir linguagens assembly diferentes para a mesma máquina. Enquanto o efeito de cada instrução em linguagem de máquina é fixado no hardware, o desenvolvedor que define uma linguagem assembly é livre para especificar:

- um mnemônico para cada instrução em linguagem de máquina;
- um formato padrão para as linhas de um programa em linguagem assembly;
- formatos para especificar diferentes modos de endereçamento e outras variações de instrução;
- mecanismos para associar nomes simbólicos com endereços e outros valores numéricos;

- mecanismos para definir dados que serão carregados na memória junto com instruções quando o programa é carregado;
- diretivas que especificam como o programa deve ser traduzido (“assemblado”);

Normalmente, todas as linguagens assembly para um dado microprocessador coincidem sobre as instruções e designadores de modos de endereçamento, mas podem diferir sobre detalhes como formato de linhas, tamanho máximo dos símbolos, formato de constantes, e diretivas do assembler. Embora conceitualmente existam dúzias de diferentes linguagens assembly para um mesmo microprocessador, na prática existem somente umas poucas que se destacam: o padrão do fabricante e talvez uma ou duas versões diferentes de fornecedores de software.

4.4. FORMATOS DA LINGUAGEM ASSEMBLY

As linhas de código fonte na linguagem assembly seguem o seguinte formato:

<SÍMBOLO> <INSTRUÇÃO/DIRETIVA> <OPERANDOS> <;COMENTÁRIOS>

SIMBOLO: Um símbolo é simplesmente um identificador, uma seqüência de letras e dígitos iniciando com uma letra. O tamanho máximo do símbolo varia com os diferentes assemblers. A cada símbolo num programa assembly está associado um valor no instante em que é definido (números, string de caracteres ou posições de memória); o assembler mantêm-se informado sobre os símbolos e seus valores por uma tabela de símbolos interna.
EX.:

```
inicio:   mov AX,[dado]
```

o símbolo *inicio* possui um valor igual ao do endereço de memória da instrução mov. Em geral, o valor de um símbolo é o endereço de memória no qual a instrução ou dado correspondente está armazenado. Cada símbolo pode ser definido somente 1 vez, mas pode ser referenciado tantas vezes quanto for necessário.

INSTRUÇÃO: Este campo pode conter um mnemônico de uma instrução, que são nomes mais fáceis de serem guardados por nós humanos, associados às instruções de máquina. Ex.: MOV, ADD, MUL e JMP são todos mnemônicos de instruções correspondentes diretamente a instruções de movimentação de dados, adição, multiplicação e salto do 8086.

O assembler traduz cada mnemônico de instrução diretamente ao código de máquina correspondente. Se é inserido um mnemônico de instrução em um programa assembly, o resultado é uma instrução de máquina correspondente no código objeto gerado.

DIRETIVAS: ao contrário dos mnemônicos das instruções, as diretivas não geram nenhum código executável, ao invés, controlam vários aspectos de operação do assembler. Ex.:

- segmentos usados
- forma dos arquivos de listagem

OPERANDOS: Mnemônicos das instruções e diretivas informam ao assembler o que fazer. Operandos, por outro lado, informam ao assembler que registros, parâmetros, posição de memória e assim por diante associar com cada instrução ou diretiva. Uma instrução MOV não significa nada sozinha. Operandos são necessários para informar ao assembler onde buscar o valor e para onde deseja movimenta-lo. Zero, um ou mais operandos são necessários para as instruções e, virtualmente, qualquer número de operandos que estejam em uma única linha pode ser aceito por várias diretivas.

OPERANDOS REGISTROS: são os operandos mais comumente usados. Registros podem servir como fonte ou como destino de dados e podem ainda conter um endereço de salto sob certas circunstâncias:
Ex.:

```
mov   di,ax
push  di
xchg  ah,dl
ror   dx,al
in    al,dx
```

OPERANDOS CONSTANTES: registros são ótimos para armazenar valores intermediários, mas freqüentemente necessitamos de um valor constante como operando.

Ex.: sub si,4
A linguagem assembly permite a representação de operandos constantes de várias formas diferentes:
BINÁRIO: seqüência de 0´s e 1´s seguida pela letra B
DECIMAL: seqüência de 0..9 seguida pela letra D (opcional)
HEXADECIMAL: seqüência de 0..F seguida pela letra H (iniciando por um número)
CARACTER: qualquer caracter entre aspas
EXPRESSÕES: expressões constantes podem ser usadas quando valores constantes são aceitos (parêntesis aninhados, lógica aritmética,...)
SIMBOLOS: símbolos podem servir como operandos para muitas instruções. Usando o operador apropriado, símbolos podem servir para gera valores constantes.
Ex.: MEMWORD: DW 1

MOV AL, SIZE MEMWORD

COMENTÁRIOS: comentários não fazem nada, no sentido que não afetam o código gerado pelo assembler, mas isto não que dizer que não sejam importantes, principalmente em se tratando de programa em assembly.

4.5. OPERAÇÕES EM TEMPO DE ASSEMBLY, TEMPO DE CARGA E TEMPO DE EXECUÇÃO:

TEMPO DE ASSEMBLY: são operações executadas pelo programa assembler, somente uma vez, quando o programa em linguagem assembly é traduzido. Expressões nos operandos são sempre avaliadas em tempo de assembly. Quando se vê os operadores + ou -, lembre-se que as adições e subrações são executadas pelo assembler, e não quando o programa é executado.
TEMPO DE CARGA: são executadas pelo programa LOADER, somente uma vez, quando o programa objeto é carregado na memória da máquina alvo. Todas instruções e dados constantes são depositados na memória em tempo de carga.
TEMPO DE EXECUÇÃO: são operações executadas pelo programa em linguagem de máquina na máquina alvo, cada vez que as instruções correspondentes são executadas. Por exemplo, a instrução ADD efetua uma adição em tempo de execução.

5. LINGUAGEM ASSEMBLY DO 8086/88

5.1. SINTAXE DO ASSEMBLY DO 8086/88

A linguagem não é sensível à letra maiúscula ou minúscula. Para facilitar a compreensão do texto do programa, sugere-se:

- Uso de letra maiúscula para o código;
- Uso de letra minúscula para comentários;

5.2. DECLARAÇÕES (STATEMENTS):

Como visto no capítulo anterior (item 4.4), uma linha de código em assembly pode conter uma instrução ou uma diretiva:

- Instruções, que serão convertidas em códigos de máquina;
- Diretivas, que instruem o montador assembler a realizar alguma tarefa específica:
  - Alocar espaço de memória para variáveis;
  - Criar uma subrotina (procedimento)

Formato de uma declaração (linha de programa):

[símbolo] [instrução/diretiva] [operando(s)] [;comentário]

Exemplo:

```
INICIO: MOV    CX,5h          ;inicializar o contador
```

A separação entre os campos deve ser do tipo **<espaço>** ou **<tab>**.

- O campo símbolo:

  Pode ser um rótulo de uma instrução, ou um nome de subrotina, um nome de variável, contendo de 1 a 31 caracteres, iniciando por uma letra e contendo somente letras, números e os caracteres ? @ $ _.
  (obs.: o montador assembler traduz os símbolos por endereços de memória).

  Exemplos:

Nomes válidos	Nomes inválidos:
LOOP1:	DOIS BITS
T_TEST	2abc
@caracter	A42.25
SOMA_TOTAL4	#33
$100	

- Campo de código de operação:

  Contém o código de operação simbólico (mnemônico)
  No caso de diretivas, contém o código de pseudo-instrução
  Exemplos:

instruções	diretivas
MOV	.MODEL
ADD	.STACK
INC	nome PROC
JMP	

- Campo de operandos:

  Instruções podem conter 0, 1 ou 2 operandos no 8086/88.
  Exemplos:

```
NOP          ; sem operandos: instrui para fazer nada
INC    AX    ; um operando: soma 1 ao conteúdo do registrador AX
```

```
ADD     AX,2    ; dois operandos: soma 2 ao conteúdo do registrador AX
```

No caso de **instruções** de dois operandos:

- O primeiro operando, operando destino: é um registrador ou posição de memória onde o resultado é armazenado; o conteúdo inicial é modificado;
- O segundo operando, operando fonte: não modificado pela instrução;
- Os operandos são separados por vírgula.

No caso de **diretivas**, o campo de operandos contém mais informações acerca da diretiva.

– Campo de comentário:
- Um ponto-e-vírgula (;) marca o início deste campo;
- O montador assembler ignora o resto da linha após este marcador;
- Comentários são opcionais.

Uma boa prática acerca da programação é comentar tudo e incluir a informação acerca da idéia por trás da codificação (o algorítmo).

Exemplos:

```
MOV    CX,0          ; movimenta 0 para CX (comentário desnecessário)
MOV    CX,0          ; CX conta o nº de caracteres, inicialmente vale 0
;
;
; inicialização de registradores (linha inteira de comentário)
```

5.3. FORMATO DE DADOS, VARIÁVEIS E CONSTANTES

Números: Os valores numéricos em assembly podem ser definidos no programa fonte em binário (base 2), em decimal (base 10), ou hexadecimal (base 16). A base de um valor numérico é indicada através de uma letra codificada ao final do número:

B – binário
D – decimal (opcional)
H – hexadecimal (o número deve começar por um dígito)

Caracteres ou cadeia de caracteres: podem ser definidos entre apóstrofos(‘). Por exemplo:

```
MOV    AL,’x’    ; carrega no registro AL o código ASCII do carácter ‘x’
```

O montador assembler se encarrega de substituir o caracter por um byte que corresponde ao seu código ASCII.

Variáveis: é um nome simbólico que vai estar associado à posições na memória que serão manipuladas pelo programa. A definição de “variáveis” em assembly é feita através de três diretivas básicas: DB, DW e DD. O uso de uma destas diretivas informa ao montador assembler o total de bytes em memória que vai ser alocado para uma dada variável. Na declaração de uma variável, pode-se especificar também o valor inicial que esta vai assumir quando o programa iniciar sua execução.
Em função da diretiva usada na declaração da variável, esta vai assumir um tipo e também receber um endereço de memória que será atribuído pelo montador assembler. A seguir serão mostrados alguns exemplos de declaração de variáveis (dados na memória):

Variáveis do tipo byte:

Forma geral: nome DB valor_inicial

Exemplos:

```
Um_byte     DB    00      ; byte com valor inicial zero
Caracter    DB    ‘a’     ; byte com valor inicial igual ao cód. Ascii de ‘a’
Outro_byte  DB    ?       ; usa-se a ‘?’ quando não se deseja inicializar a variável
Erro        DB    0150h   ; inválida: valor inicial não cabe num byte
```

Variáveis do tipo word:

Forma geral: nome DW valor_inicial

Exemplos:

```
Uma_Word    DW    0h      ; Word com valor inicial zero
Valido      DW    0150h   ; correto: o valor inicial pode ser armazenado em uma Word
W1          DW    ?       ; não inicializada
```

Na mesma linha de declaração de uma variável, pode-se reservar várias unidades do mesmo tipo (byte ou word), bastando para isto colocar mais valores iniciais separados por vírgula. O montador vai alocar o espaço necessário na memória em bytes adjacentes. Exemplos:

```
Vogais      DB      'a','e','i','o','u'
Dez_bytes           DB      0,1,2,3,4,5,6,7,8,9
Três_words          DW      12h,?,145
```

(Obs.: nestes casos, o nome da variável corresponde ao endereço de memória do primeiro elemento alocado)

Seqüência de caracteres podem ser reservadas usando a técnica acima ou declarando a seqüência inicial desejada entre apóstrofos. Exemplo:

```
Vogais  DB      'aeiou'  ; mesmo resultado da declaração no exemplo acima
```

Declarando grandes áreas (vetores) na memória:

A linguagem assembly ainda permite a declaração de seqüência de dados em memória através do uso da duplicação de padrões. A forma geral da diretiva usando este recurso é:

Nome Dn contador_repetição DUP (valores_iniciais)

Onde:

Dn: pode ser uma das diretivas vistas anteriormente: DB ou DW;

Contador_repetição: é um número que vai indicar quantas vezes o padrão definido dentro dos parêntesis será repetido;

Valores_iniciais: corresponde a um padrão contendo valores iniciais. Quando for mais que um, este deve ser separado por vírgula;

Exemplos;

```
Buf1    DB      1000    DUP     (0)     ; declara 1000 bytes inicializados com zeros
Buf2    DW      1000    DUP     (0)     ; declara 1000 words inicializadas com zeros
Buf3    DB      100     DUP     (1,2,3) ; declara 300 bytes inicializados com a seq. 1, 2 e 3
```

A declaração de caracteres e números podem ser combinadas em uma mesma declaração. Exemplo:

```
Mensagem    DB      'Alô mundo!', 0Ah, 0Dh, '$'
```

Algumas dicas para a declaração de variáveis são:

- Adote nomes para dados únicos e descritivos;
- Use somente DB para definir seqüência de caracteres;
- Atenção para não confundir valores escritos em hexadecimal com decimal
- Usar áreas declaradas como DB para processar dados em registradores de 8 bits e DW para registradores de 16 bits.

Constantes: nome simbólico para um dado valor constante, que é muito utilizado no programa. Para atribuir um nome a uma constante, utiliza-se a pseudo-instrução EQU (equates -> igual a) e a sintaxe é:

Nome EQU valor_da_consntante

Exemplos:

```
;declaração
LF      EQU   0Ah     ; carácter Line Feed (salto de linha)
CR      EQU   0Dh     ; carácter Carriage Return (retorno cursor para esquerda)
Linha1  EQU   'digite seu nome completo'

; uso
Mensagem      DB      linha1,LF,CR
```

É importante ressaltar que uma linha contendo declaração de constantes não vai gerar nenhum código. Trata-se somente de mais uma facilidade de programação oferecida pela linguagem assembly.

6. CONJUNTO DE INSTRUÇÕES

As instruções do 8088 podem variar em tamanho, de 1 a 6 bytes. Pode-se organizar o conjunto de instruções do 8088 em classes de instruções distintas, particularmente pelo tipo de operação associado. Assim, as seguintes classes podem ser consideradas: instruções de transferência de dados, instruções aritméticas, instruções lógicas, instruções de desvio de fluxo, instruções de entrada/saída, instruções de controle e instruções de rotação e deslocamento.

6.1. INSTRUÇÕES DE TRANSFERÊNCIA DE DADOS

As instruções de transferência de dados permitem a troca de dados entre registros internos do 8088 ou entre registro e memória. A transferência de dados entre registros pode ser feita de maneira direta, através de uma instrução MOV.

A tabela a seguir resume o conjunto de instruções de transferência de dados:

Instrução	Descrição
MOV	transfere do operando fonte para o operando destino
MOVS	move byte ou palavra numa área de memória
STOS	move byte ou palavra do acumulador para memória
LODS	move byte ou palavra da memória para acumulador
XCHG	permuta conteúdo de dois operandos
LAHF	carrega AH com bits do registro de flags
SAHF	armazena AH no registro de flags
XLAT	traduz conteúdo de AL consultando tabela de vetores
LEA	carrega offset de um operando no registro especificado
LDS	transfere quatro bytes para par de registro de 16 bits
LES	idem a LDS mas considerando o segmento extra
PUSH	salva palavra de 16 bits na pilha
PUSHF	salva flags na pilha
POP	recupera palavra de 16 bits na pilha
POPF	recupera flags na pilha

6.2. INSTRUÇÕES ARITMÉTICAS

As instruções aritméticas consistem basicamente das 4 operações (adição, subtração, multiplicação e divisão) e das instruções associadas aos ajustes decimais, assim como aquelas de comparação. As operações podem ser realizadas tanto com operandos de 8 ou de 16 bits, sendo que instruções de multiplicação e divisão podem operar sobre 16 ou 32 bits.

Instrução	Descrição
ADD	adição:destino= destino + fonte
ADC	idem a ADD, considerando o flag de carry
SUB	subtração: destino= fonte - destino
SBB	idem a SUB, considerando o flag de carry
INC	incrementa operando de 1: operando = operando + 1
DEC	decrementa operando de 1: operando = operando - 1
MUL	multiplica dois operandos (em 8 ou 16 bits)
IMUL	multiplica dois operandos (leva em conta o sinal)
DIV	divide acumulador por operando
IDIV	idem a DIV, mas leva em conta sinal dos operandos

DAA	ajuste decimal para adição
DAS	ajuste decimal para subtração
AAA	ajuste ASCII para adição
AAM	ajuste ASCII para multiplicação
AAD	ajuste ASCII para divisão
CBW	converte byte (8bits) em word (16 bits)
CWD	converte palavra (16 bits) em dupla palavra (32 bits)
CMP	compara dois operandos (subtração)
CMPS	compara dois bytes/words (CMPSB/CMPSW)
SCAS	compara elemento indicado por DI com acumulador
AAS	ajuste ASCII para subtração

6.3.

6.4. INSTRUÇÕES LÓGICAS

As instruções lógicas compreendem as operações lógicas efetuadas bit a bit

Instrução	Descrição
AND	AND lógico entre dois operandos
OR	OR lógico entre dois operandos
XOR	OR exclusivo entre dois operandos
TEST	comparação lógica entre dois operandos
NOT	complementação de um operando

6.5. INSTRUÇÕES DE DESVIO DE FLUXO

Nesta classe, estão organizadas as instruções que permitem efetuar desvios no sequenciamento de um programa.

Instrução	Descrição
JMP	desvio condicional no segmento de código
JCOND	desvio condicional no segmento de código
CALL	chamada de rotina
RET	retorno de rotino
INT	ativação de interrupção por software
INTO	interrupção tipo 4, se houve overflow (OF = 1)
IRET	retorno de rotina de interrupção
LOOP	repetição enquanto CX <> 0
LOOPE	repete enquanto CX <> 0 e flag zero = 1 (decrementa CX)
LOOPNE	repete enquanto CX <> 0 e flag zero = 0 (decrementa CX)

6.6. INSTRUÇÕES DE ENTRADA/SAÍDA

Correspondem às instruções de entrada e saída de dados via dispositivos de E/S.

Instrução	Descrição
IN	entrada de dados por uma porta (8 ou 16 bits)
OUT	saída de dados por uma porta (8 ou 16 bits)

6.7. INSTRUÇÕES DE CONTROLE

São aquelas instruções responsáveis pelo controle de funcionamento do microprocessador. Pode-se incluir nesta classe as instruções de habilitação de desabilitação de interrupções, a instrução de bloqueio do barramento, as instruções de controle de direção (flag de direção), etc...

Instrução	Descrição
NOP	sem operação
HLT	suspensão do processador
WAIT	sincronização do processador
LOCK	bloqueio do barramento de dados
ESC	envio de dados e ordens para coprocessadores
CLC	reseta o flag de carry
CMC	complementa o flag de carry
CLD	reseta o flag de direção
CLI	reseta o flag de interrupção (desabilita interrupções)
STC	seta o flag de carry
STD	seta o flag de direção
STI	seta o flag de interrupção (habilita interrupções)

6.8. INSTRUÇÕES DE ROTAÇÃO E DESLOCAMENTO

Permitem realizar operações de deslocamento e rotação à direita e à esquerda sobre os operandos.

Instrução	Descrição
RCL	rotação esquerda de 1 ou mais (CL) bits - através carry
RCR	rotação à direita de 1 ou mais (CL) bits - através do carry
ROL	rotação à esquerda de 1 ou mais (CL) bits
ROR	rotação à direita de 1 ou mais (CL) bits
SAL	deslocamento à esquerda de 1 ou mais (CL) bits
SAR	deslocamento à direita de 1 ou mais (CL) bits
SHR	deslocamento lógico à direita de 1 ou mais (CL) bits

7. ALGUMAS INSTRUÇÕES BÁSICAS

7.1. DESCRIÇÃO DOS FORMATOS DAS INSTRUÇÕES

Todas as instruções estudadas neste texto seguem o formato padrão de apresentação mostrado na figura 7.1.

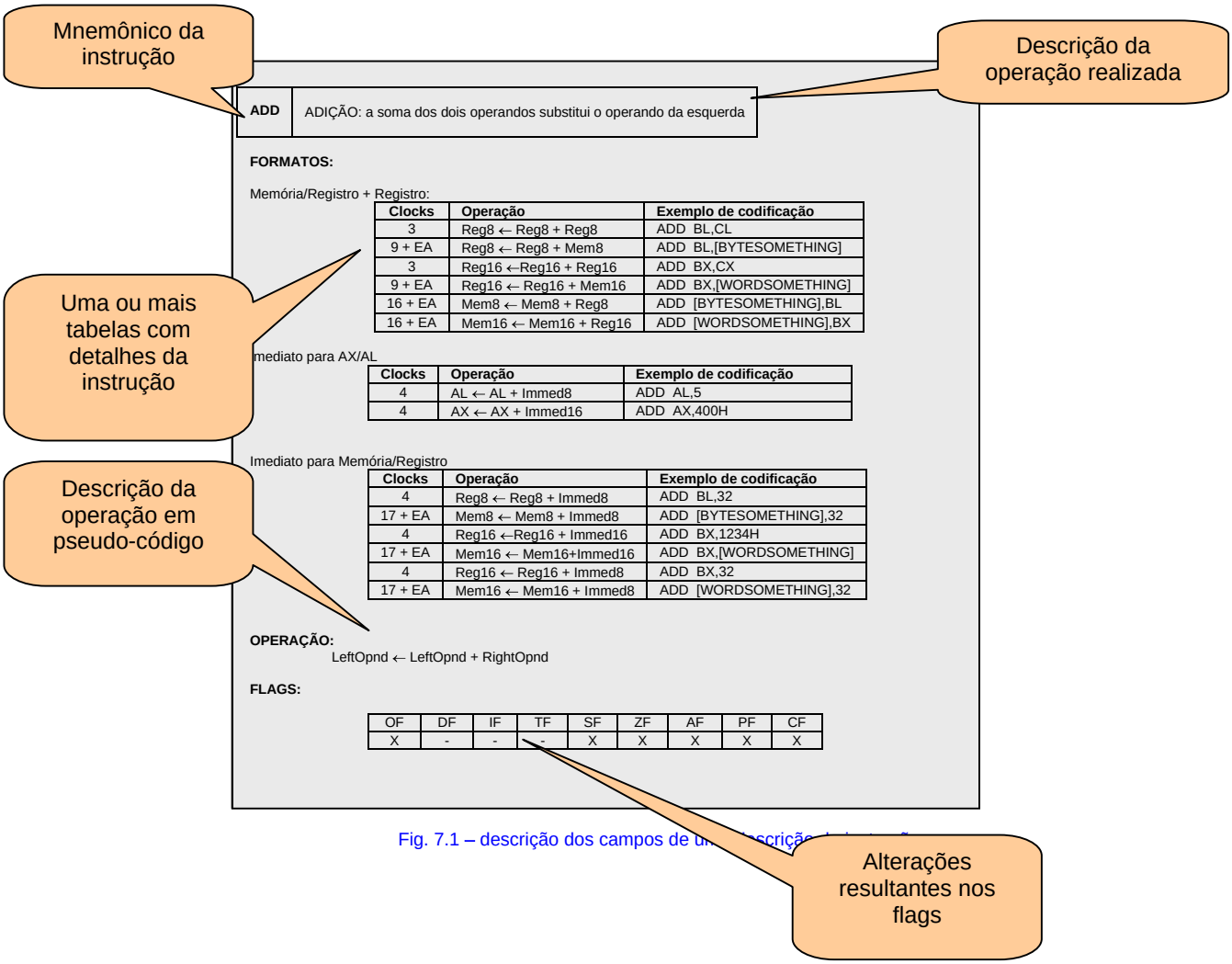


Fig. 7.1 – descrição dos campos de u

7.2. DETALHES DE ALGUMAS TABELAS

Na tabela que indica as alterações realizadas sobre os FLAGS, a linha superior representa os FLAGS individuais, e a linha inferior mostra o efeito da instrução sobre cada flag. As letras, números e símbolos usados na tabela são definidos como a mostrada a seguir:

FLAG	DEFINIÇÃO
OF	Overflow
DF	Direction (usado em operações com strings)
IF	Interrupt Enable (1=habilitado)
TF	Single Step Trap Flag (gera a interrupção 1 a cada instrução executada)
SF	Sign
ZF	Zero
AF	Auxiliary Carry (usado principalmente em operações BCD)
PF	Parity
CF	Carry

CÓDIGO EFEITO	EFEITO
X	Modificado pela instrução, resultado depende dos operandos
-	Não modificado
U	Indefinido após a operação
1	Setado para 1 pela instrução
0	Setado para 0 pela instrução

O tempo de duração de cada instrução é dado pelo número de ciclos de clock indicados nas tabelas de detalhes das instruções. Estes valores variam para uma mesma instrução em função do tipo de endereçamento utilizado, sendo nestes casos representados pela sigla EA (Effective Address), cujo valor é dado pela tabela abaixo.

Componentes do EA	Clocks
Somente deslocamento	6
Baseado/Indexado (BX,BP,SI,DI)	5
Deslocamento + Baseado/Indexado (BX,BP,SI,DI)	9
Base+ BP+DI,BX+SI, Index BP+SI,BX+SI	7 8
Deslocamento BP+DI+DISP + BX+SI+DISP Base + BP+SI+DISP Index BX+DI+DISP	11 12

7.3. TABELA DE SÍMBOLOS UTILIZADOS:

Símbolo	**Significado**
AX	Acumulador (16 bits)
AH	Acumulador (byte mais significativo)
AL	Acumulador (byte menos significativo)
BX	Registro BX (16 bits), o qual pode ser dividido e endereçado como 2 registros de 8 bits
BH	Byte mais significativo do registro BX
BL	Byte menos significativo do registro BX
CX	Registro CX (16 bits), o qual pode ser dividido e endereçado como 2 registros de 8 bits
CH	Byte mais significativo do registro CX
CL	Byte menos significativo do registro CX
DX	Registro DX (16 bits), o qual pode ser dividido e endereçado como 2 registros de 8 bits
DH	Byte mais significativo do registro DX
DL	Byte menos significativo do registro DX
SP	Stack pointer (16 bits)
IP	Instruction pointer (16 bits)
Flags	Registro de 16 bits, no qual residem nove flags
DI	Destination Index register (16 bits)
SI	Stack index register (16 bits)
CS	Code segment register (16 bits)
DS	Data segment register (16 bits)
ES	Extra segment register (16 bits)
SS	Stack segment register (16 bits)
REG8	Nome ou codificação de um registro de 8 bits da CPU
REG16	Nome ou codificação de um registro de 16 bits da CPU
LSRC, RSRC	Refere-se ao operando de uma instrução, geralmente o operando da esquerda é chamado de operando de destino e o operando a direita é chamado operando fonte
reg	Um campo que especifica um REG8 ou REG16 na descrição de uma instrução
EA	Endereço efetivo (16 bits)
r/m	Bits 2, 1 e 0 do byte MODRM usado em acessos a operandos na memória. Este campo de 3 bits definem EA, em conjunção com mode e campo w
mode	Bits 7 e 6 do byte MODRM. Este campo de 2 bits define o modo de endereçamento
w	Um campo de 1 bit na instrução, identificando instrução byte (w=0) ou instrução word (w=1)
d	Um 1 bit na instrução, “d” identifica direção, ou seja, se um registro especificado é fonte ou destino
(...)	Parêntesis significa o conteúdo de um registro ou posição de memória
(BX)	Representa o conteúdo do registro BX, o que pode significar o endereço onde um operando de 8 bits está localizado. Para ser usado em uma instrução em assembly, BX deve ser usado somente entre colchetes
((BX))	Significa um operando de 8 bits, o conteúdo da posição de memória apontada pelo conteúdo do registro BX. Esta notação é somente descritiva e para uso na descrição das instruções. Ela não pode aparecer em linhas de programa fonte
(BX)+1,(BX)	Significa um endereço (de um operando de 16 bits) cujo byte menos significativo reside na posição de memória apontada pelo conteúdo do registro BX e o byte mais significativo reside no próximo endereço sequencial, (BX) + 1
((BX)+1,(BX))	Significa um operando de 16 bits que reside no endereço indicado
addr	Endereço (16 bits) de um byte na memória
addr-low	Byte menos significativo de um endereço
addr-high	Byte mais significativo de um endereço
addr + 1:addr	Endereço de dois bytes consecutivos na memória, iniciando em addr
data	Operando imediato (8 bits se w=0 e 16 bits se w=1)
data-low	Byte menos significativo de um dado word
data-high	Byte mais significativo de um dado word
disp	Deslocamento
Disp-low	Byte menos significativo de um deslocamento de 16 bits
Disp-high	Byte mais significativo de um deslocamento de 16 bits
<-	Atribuição
+	Adição

-	Subtração
*	Multiplicação
/	Divisão
%	Módulo
&	AND
|	OR
||	EXCLUSIVE OR

7.4. MOV	MOVIMENTA DADOS: o operando da direita (fonte) é copiado para o operando da esquerda (destino). O operando da direita não é modificado. Nenhum flag é afetado. Não permite operações de movimentação memória-memória

FORMATOS:

Memória/Registro para ou de Registro:

Clocks	Operação	Exemplo de codificação
9 + EA	Mem8 <- Reg8	MOV [BYTESOMETHING],CL
2	Reg8 <- Reg8	MOV BL,AL
9 + EA	Mem16 <- Reg16	MOV [WORDSOMETHING],CX
8 + EA	Reg8 <-Mem8	MOV CL,[BYTESOMETHING]
8 + EA	Reg16 <- Mem16	MOV CX,[WORDSOMETHING]

Endereçamento direto a memória para ou de AX/AL:

Clocks	Operação	Exemplo de codificação
10	AL <- Mem8	MOV AL,[BYTESOMETHING]
10	AX <- Mem16	MOV AX,[WORDSOMETHING]
10	Mem8 <- AL	MOV [BYTESOMETHING],AL
10	Mem16 <-AX	MOV [WORDSOMETHING],AX

Imediato para registro:

Clocks	Operação	Exemplo de codificação
4	Reg8 <- Immed8	MOV CL,5
4	Reg16 <- Immed16	MOV SI,400H

Imediato para memória:

Clocks	Operação	Exemplo de codificação
10 + EA	Mem8 <- Immed8	MOV [BYTESOMETHING],15
10 + EA	Mem16 <- Immed16	MOV [WORDSOMETHING],1234H

Registro para ou de Registro de Segmento:

Clocks	Operação	Exemplo de codificação
2	Reg16 <- Sreg*	MOV AX,DS
10 + EA	Sreg* <-Reg16	MOV DS,AX

* CS não permitido

OPERAÇÃO:

LeftOpnd <- RightOpnd

FLAGS:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

7.5. ADD	ADIÇÃO: a soma dos dois operandos substitui o operando da esquerda

FORMATOS:

Memória/Registro + Registro:

Clocks	Operação	Exemplo de codificação
3	Reg8 <- Reg8 + Reg8	ADD BL,CL
9 + EA	Reg8 <- Reg8 + Mem8	ADD BL,[BYTESOMETHING]
3	Reg16 <-Reg16 + Reg16	ADD BX,CX
9 + EA	Reg16 <- Reg16 + Mem16	ADD BX,[WORDSOMETHING]
16 + EA	Mem8 <- Mem8 + Reg8	ADD [BYTESOMETHING],BL
16 + EA	Mem16 <- Mem16 + Reg16	ADD [WORDSOMETHING],BX

Imediato para AX/AL

Clocks	Operação	Exemplo de codificação
4	AL <- AL + Immed8	ADD AL,5
4	AX <- AX + Immed16	ADD AX,400H

Imediato para Memória/Registro

Clocks	Operação	Exemplo de codificação
4	Reg8 <- Reg8 + Immed8	ADD BL,32
17 + EA	Mem8 <- Mem8 + Immed8	ADD [BYTESOMETHING],32
4	Reg16 <-Reg16 + Immed16	ADD BX,1234H
17 + EA	Mem16 <- Mem16+Immed16	ADD BX,[WORDSOMETHING]
4	Reg16 <- Reg16 + Immed8	ADD BX,32
17 + EA	Mem16 <- Mem16 + Immed8	ADD [WORDSOMETHING],32

OPERAÇÃO:

LeftOpnd ← LeftOpnd + RightOpnd

FLAGS:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

7.6. SUB	SUBTRAÇÃO: o resultado da subtração do operando da direita do operando da esquerda substitui o operando da esquerda

FORMATOS:

Memória/Registro + Registro:

Clocks	Operação	Exemplo de codificação
3	Reg8 <- Reg8 - Reg8	SUB BL,CL
9 + EA	Reg8 <- Reg8 - Mem8	SUB BL,[BYTESOMETHING]
3	Reg16 <-Reg16 - Reg16	SUB BX,CX
9 + EA	Reg16 <- Reg16 - Mem16	SUB BX,[WORDSOMETHING]
16 + EA	Mem8 <- Mem8 - Reg8	SUB [BYTESOMETHING],BL
16 + EA	Mem16 <- Mem16 - Reg16	SUB [WORDSOMETHING],BX

Imediato para AX/AL

Clocks	Operação	Exemplo de codificação
4	AL <- AL - Immed8	SUB AL,5
4	AX <- AX - Immed16	SUB AX,400H

Imediato para Memória/Registro

Clocks	Operação	Exemplo de codificação
4	Reg8 <- Reg8 - Immed8	SUB BL,32
17 + EA	Mem8 <- Mem8 - Immed8	SUB [BYTESOMETHING],32
4	Reg16 <-Reg16 - Immed16	SUB BX,1234H
17 + EA	Mem16 <- Mem16-Immed16	SUB BX,[WORDSOMETHING]
4	Reg16 <- Reg16 - Immed8	SUB BX,32
17 + EA	Mem16 <- Mem16 - Immed8	SUB [WORDSOMETHING],32

OPERAÇÃO:

LeftOpnd <- LeftOpnd - RightOpnd

FLAGS:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

7.7. INC	INCREMENTO DE 1: o operando é incrementado de 1

FORMATOS:

Registro de 16 bits:

Clocks	Operação	Exemplo de codificação
2	Reg16 <- Reg16 + 1	INC CX

Memória/ Registro de 8 bits

Clocks	Operação	Exemplo de codificação
3	Reg8 <- Reg8 + 1	INC BL
15 + EA	Mem8 <- Mem8 + 1	INC [BYTESOMETHING]
15 + EA|	Mem16 <- Mem16 + 1	INC [WORDSOMETHING]

OPERAÇÃO:

Operand ← Operand + 1

FLAGS:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	-

7.8. DEC	DECREMENTO DE 1: o operando é decrementado de 1

FORMATOS:

Registro de 16 bits:

Clocks	Operação	Exemplo de codificação
2	Reg16 <- Reg16 - 1	DEC CX

Memória/ Registro de 8 bits

Clocks	Operação	Exemplo de codificação
3	Reg8 <- Reg8 - 1	DEC BL
15 + EA	Mem8 <- Mem8 - 1	DEC [BYTESOMETHING]
15 + EA|	Mem16 <- Mem16 + 1	DEC [WORDSOMETHING]

OPERAÇÃO:

Operand <- Operand - 1

FLAGS:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	-

8. ESTRUTURA DE UM PROGRAMA EM LINGUAGEM ASSEMBLY

A estrutura geral de uma programa fonte em assembly está mostrada no quadro abaixo. Cada item destacado será explicado a seguir:

```
.MODEL SMALL   ; define o modelo de memória

.STACK 100h    ; define o tamanho da área de pilha em bytes

.DATA          ;  indica o início da declaração de dados

.CODE    ;  indica o início da declaração de código (instruções)

inicio:  mov  ax,@data
         mov  ds,ax




         mov  ah,4ch    ; função DOS para terminar programa
         int  21h       ; encerra um programa

END inicio
```

8.1. MODELO DE MEMÓRIA

É uma diretiva que define o tamanho que os segmentos de código e dados devem ter. A tabela abaixo resume as opções existentes:

Modelo	Descrição
SMALL	Código em um segmento; Dados em um segmento;
MEDIUM	Código em mais de um segmento; Dados em um segmento;
COMPACT	Código em um segmento; Dados em mais de um segmento;
LARGE	Código em mais de um segmento; Dados em mais de um segmento; Nenhum array maior que 64Kbytes;
HUGE	Código em mais de um segmento; Dados em mais de um segmento; Arrays maiores que 64Kbytes;

Para os programas que estaremos implementando o modelo a ser adotado será o SMALL. A diretiva de definição de segmentos deve ser declarada antes da definição de qualquer segmento.

Além de definir o tamanho dos segmentos, a diretiva .MODEL define o tipo default de codificação a ser adotada pelo montador assembler para as instruções de chamada/retorno de subrotinas (ver item correspondente a estas instruções mais adiante no texto para maiores esclarecimentos).

8.2. SEGMENTO DE PILHA

Esta diretiva vai reservar uma área na memória correspondente à área de pilha. O operando da diretiva é o tamanho em bytes que está sendo especificado para a área de pilha que vai ser usada pelo programa. Vale ressaltar que, diferentemente de quando se programa em linguagem de alto nível, o programador assembly deve estimar o tamanho da área de pilha. O valor usado como exemplo, 100h (=256) é uma quantidade razaoável e que vai atender a todos os programas exemplos que estaremos implementando.

8.3. SEGMENTO DE DADOS

Esta diretiva vai definir o segmento de dados, ou seja, neste bloco que vai estar as declarações das variáveis que estarão sendo manipuladas pelo programa. Pode-se também fazer a atribuição de símbolos para constantes usando a diretiva EQU vista no item 5.3.

8.4. SEGMENTO DE CÓDIGO

Esta diretiva define o inicio do segmento de código, que é o local onde serão codificadas as instruções. Dentro do segmento de código, as instruções podem ser organizadas em subrotinas, conforme iremos estudar no item 11.

8.5. CONCLUSÃO

No ambiente que estaremos desenvolvendo nossos programas (DOS), o sistema operacional já se encarrega de inicializar os segmentos de código (CS) e o segmento de pilha (SS) com base nas informações que estão contidas no programa executável gerado. Os registros de segmentos de dados (DS) e extra (ES), este quando utilizado, devem ser inicializados pelo programa. Recomenda-se que esta inicialização seja a primeira operação realizada pelo programa e portanto, todos os programas que estaremos implementando começam com as instruções:

```
MOV   AX,@data
MOV   DS, AX
```

A palavra reservada @data é substituida pelo sistema operacional no momento da carga do programa e corresponde ao valor do endereço de memória (segmento) onde foi alocada a área de dados que vai ser usada pelo programa. Percebam que esta informação é passada pelo sistema operacional e que o mesmo já poderia inicializar este registro, assim como faz com o CS e SS, porém este não o faz.

O término de um programa fonte é indicado pela diretiva **END**. Qualquer linha abaixo da linha contendo esta diretiva é ignorada pelo montador assembler.

Pelo fato da linguagem assembly não possuir um ponto de entrada pré-definido, o equivalente ao “main” da linguagem C, devemos definir este ponto de entrada explicitamente no código fonte. Isto é feito adicionado-se à direita da diretiva END o rótulo da primeira instrução que vai ser executada pelo programa. Consequentemente, devemos definir este rótulo associado à primeira instrução no bloco de instruções e se considerarmos que as duas primeiras instruções são as duas mostradas aciama, teremos por exemplo:

```
...
.CODE
...
    inicio:     MOV   AX, @data
                MOV   DS, AX
                ...


END inicio  ; a palavra inicio é o nome do rótulo da primeira instrução que será executada pelo programa
```

9. INSTRUÇÕES DE CONTROLE DE FLUXO

Praticamente todos os programas possuem alguma forma de laços e/ou desvios, exceção aos que são bastante simples. Um exemplo de necessidade comum aos programas é testar uma condição para determinar se o programa deve terminar ou não. Em assembly, a implentação de um laço ou desvio implica em transferir o controle da execução para outra instrução que não seja aquela que está codificada imediatamente a seguir à instrução que está sendo executada. O 8086 oferece uma série de instruções e mecanismos que permitem implementar a transferência de controle mudando o que seria a sequencia normal de execução. Neste item estaremos estudando estas instruções e as formas de codificá-las de modo a implementar as estruturas lógica de que dispomos nas linguagens de alto nível.

9.1. LAÇOS DE REPETIÇÃO

Para os casos em que se deseja repetir uma sequência de instruções um certo número de vezes, o 8086 oferece a instrução **LOOP** que é apresentada em detalhes no final deste item. A operação realizada pela instrução LOOP pode ser descrita como:

- Decrementa de um o valor de CX;
- Verifica o valor resultante em CX e se for diferente de zero, salta para a instrução indicada no operando da instrução LOOP, se for igual a zero, continua a execução na instrução seguinte;

Como pode ser observado, a instrução LOOP requer que um valor inicial seja armazenado no registro CX. O operando da instrução LOOP é um rótulo definido na primeira instrução da sequência que vai fazer parte do laço. Tomemos como exemplo a inicialização com zeros de um vetor de bytes na memória, que foi declarado como:

```
V1      DB      100 DUP (?)
```

A implementação desta inicialização usando a instrução LOOP seria:

```
        ...
        MOV     SI,0    ; SI é o registro usado como um indice para o acesso ao vetor na memória
        MOV     CX,100  ; CX recebe o número de vezes que o laço vai se repetir

volta:  MOV     [V1+SI],0       ; escreve na posição do vetor dada por V1+SI o valor 0
        INC     SI              ; incrementa o indice de acesso ao vetor
        LOOP    volta           ; decrementa um de CX e salta para a instrução rotulada por volta
                                ; se CX é <> 0
        MOV     ...             ; continua a execução a partir desta instrução quando terminar o laço
```

9.2. SALTO INCONDICIONAL

No desenvolvimento de programas em assembly é inevitável o uso de instrução que efetuam saltos, que são equivalentes a GOTOs em outros linguagens. A instrução que realiza este tipo de operação no 8086 é o **JMP**.

O operando da instrução de desvio incondicional JMP é o rótulo da instrução para a qual se deseja saltar. O exemplo abaixo mostra o uso da instrução JMP (Obs.: é um exemplo ilustrativo, observar que trata-se de um “laço eterno”, ou seja, uma vez que o programa entra no laço não sai mais. Este tipo de laço deve ser evitado)

```
                ...
                MOV     AX, 1
                MOV     BX, 1
continua:       ADD     AX, BX
                JMP     continua  ; salta incondicionalmente para  a instrução rotulada por continua
```

9.3. SALTOS CONDICIONAIS

O 8086 oferece um série de instruções que realizam ou não o salto baseado no valor atual de um ou mais **flags de status** (ver definição dos flags de status em 3.3.4). Os flags de status tem seus valores atualizados durante a execução de instruções lógicas ou aritméticas (exemplo: observar as mudanças efetuadas nos flags pelas instruções SUB/ADD/INC e DEC). O uso de uma instrução de desvio condicional é sempre precedido de uma instrução que garantidamente atualize o(s) flag(s) que está(ão) sendo testado(s).
Os mnemônicos das instruções de desvio condicional começam pela letra J e é seguido do tipo da condição que vai ser testada. O operando é o rótulo da instrução para a qual se deseja desviar. O formato geral das instruções de desvio condicional é:

JXXX rótulo_de_destino

Onde:

XXX é uma condição dependente de algum dos Flags de Status;

E a ação executada pela instrução segue o modelo:

- Se a condição XXX é verdadeira, a próxima instrução a ser executada é aquela definida pelo rotulo_de_destino;
- Se a condição XXX é falsa, a próxima instrução a ser executada é aquela que imediatamente segue a instrução de salto;

Existem no 8086 três classes de instruções de saltos condicionais, a saber:

- Saltos de flags simples: dependem do valor de algum flag específico;
- Saltos sinalizados: dependem do resultado de uma operação realizada sobre números com sinal
- Saltos não sinalizados: dependem do resultado de uma operação realizada sobre números sem sinal;

Saltos de flags simples

Para cada um dos flags de status existentes, existe no 8086 um par de instruções que realizam o salto baseado no seu estado atual, uma para a condição de estar ligado (=1) e outro para a condição de estar desligado (=0). Por exemplo para o flag de zero (ZF), temos as instruções de desvio condicional JZ e JNZ. Vamos tomar o exemplo usado no item 9.1, ou seja, a inicialização com zeros de um vetor de bytes na memória:

```
;declaração do vetor na memória
.DATA
V1      DB      100 DUP (?)

.CODE
...
MOV     SI,0     ; SI é o registro usado como um indice para o acesso ao vetor na memória
MOV     AL,100 ; AL é o contador que vai ser testado

volta:  MOV     [V1+SI],0       ; escreve na posição do vetor dada por V1+SI o valor 0
        INC     SI              ; incrementa o indice de acesso ao vetor
        DEC     AL              ; decrementa o contador AL de um. Esta instrução vai atualizar o flag
                                ; de zero ZF, que vai ser testado pela próxima instrução
        JNZ     volta           ; salta se o resultado no contador AL é <> de zero
        MOV     ...             ; continua a execução a partir desta instrução quando o contador for zero
```

Observar o código acima que conseguimos o mesmo efeito da instrução LOOP usando instruções de desvio condicional.

Saltos sinalizados e não sinalizados

Para a execução de desvios baseados nas condições do tipo maior, menor, maior-ou-igual e menor-ou-igual, é preciso levar em conta o tipo de dado que se está trabalhando. Além disto, **a operação que estabelece nos flags a condição de teste é a subtração**. A tabela abaixo resume as instruções de desvio condicionais enquadradas neste requisito:

Tipo de teste (condição para realizar o desvio)	Dados com sinal	Dados sem sinal
Igual	JE	JE
Diferente	JNE	JNE
Maior	JG	JA
Menor	JL	JB
Maior-ou-igual	JGE	JAE
Menor-ou-igual	JLE	JBE

Considere a implentação de um desvio se o valor de AL for maior que o valor de BL, considerando dados sem sinal:

```
SUB    AL,BL       ; a instrução de subtração estabelece nos flags as condições para o teste
JA     é_maior     ; efetua o salto se o valor de AL e maior que o valor em BL
                   ; executa esta instrução se a condição do salto não for satisfeita
```

Observar que no exemplo anterior, para o estabelecimento da condição de teste entre dois elementos através da operação de subtração, tivemos que alterar o valor de AL, que recebeu o resultado da subtração. Nem sempre isto é desejável. Considere a situação em que se deseja somente testar, sem alterar os valores dos operandos que estão sendo testados. No exemplo acima teriamos que ter armazenado uma cópia do conteúdo de AL antes de efetuar a subtração. Para eliminar este incoveniente, o 8086 oferece uma instrução que realiza uma subtração entre os operandos e atualiza os flags, porém sem modificar os operandos: **CMP.** Usando esta instrução no exemplo anterior, temos:

```
CMP    AL,BL       ; efetua a subtração, porém não altera os operandos. Estabelece nos
                   ; flags as condições para o teste
JA     é_maior     ; efetua o salto se o valor de AL e maior que o valor em BL
                   ; executa esta instrução se a condição do salto não for satisfeita
```

9.4. INSTRUÇÕES VISTAS NESTE CAPÍTULO

LOOP	**CONTROLE DE LAÇO**: fornece controle de interação. Para usar a instrução LOOP deve-se carregar em CX um valor ser sinal da quantidade de interações e então codificar o LOOP no final da série de instruções que farão parte das interações. Cada vez que o LOOP é executado o registro CX é decrementado e um desvio condicional para o topo do laço é executado. A condição testada para efetuar o desvio é se **CX é diferente de zero após o decremento.**

FORMATOS:

Clocks	Operação	Exemplo de codificação
18 ou 6	Dec CX; salta se CX <> 0	LOOP TARGETLABEL

OPERAÇÃO:

```
CX  <- CX – 1;
if (CX<>0) then do
   IP <- IP + displacement;
end if
```

FLAGS:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

JMP	**Salto incondicional:** existem dois tipos de salto intra-segmento: um que é relativo ao IP e é especificado por um rótulo para determinar o endereço de destino; e um no qual o endereço de destino é obtido de um registro ou variável na memória sem modifica-lo.

FORMATOS:
No interior de um segmento, relativo ao IP:

Clocks	Operação	Exemplo de codificação
15	IP <- IP + Disp16	JMP NEAR_LABEL

No interior de um segmento, indireto:

Clocks	Operação	Exemplo de codificação
11	IP <- Reg16	JMP SI
18 + EA	IP <- Mem16	JMP WORD PTR[SI]

OPERAÇÃO:

```
if IP-relative then do
        IP <- IP + Disp16;
else do;
        IP <- (EA);
end if;
```

FLAGS:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Jcond	**Salto condicional:** Saltos condicionais testam flags, os quais presumivelmente foram "*setados*" de alguma forma por uma instrução executada previamente. Devido a existência de vários significados e formas de usos ao interpretar estados particulares de flags, são oferecidos diferentes mnemônicos para cada interpretação resultando em um mesmo op-code. Isto significa que alguns op-codes são, de fato, sinônimos de outros. Como um exemplo, considere um programa que tenha comparado um caracter com outro em AL poderia saltar se os dois eram iguais (JE), enquanto em outra situação se pretende simplesmente testar um bit de AX através da instrução AND usando uma máscara e considerar se o resultado é zero ou não para efeito de desvio (neste caso se usaria JZ, um sinônimo de JE). Em todos os casos, se a condição especificada no salto condicional é verdadeira, o valor do deslocamento é adicionado ao IP. Os limites de faixas de valores de saltos condicionais são 127 bytes abaixo ou 126 bytes acima da instrução.

FORMATOS:

Clocks	Operação	Exemplo de codificação
16 ou 4	Salta se acima	JA TARGETLABEL
16 ou 4	Salta se acima ou igual	JAE TARGETLABEL
16 ou 4	Salta se abaixo	JB TARGETLABEL
16 ou 4	Salta se abaixo ou igual	JBE TARGETLABEL
16 ou 4	Salta se carry setado	JC TARGETLABEL
16 ou 4	Salta se igual	JE TARGETLABEL
16 ou 4	Salta se maior	JG TARGETLABEL
16 ou 4	Salta se maior ou igual	JGE TARGETLABEL
16 ou 4	Salta se menor	JL TARGETLABEL
16 ou 4	Salta se menor ou igual	JLE TARGETLABEL
16 ou 4	Salta se não acima	JNA TARGETLABEL
16 ou 4	Salta se nem acima e nem igual	JNAE TARGETLABEL
16 ou 4	Salta se não abaixo	JNB TARGETLABEL
16 ou 4	Salta se nem abaixo e nem igual	JNBE TARGETLABEL
16 ou 4	Salta se não carry setado	JNC TARGETLABEL
16 ou 4	Salta se não igual	JNE TARGETLABEL
16 ou 4	Salta se não maior	JNG TARGETLABEL
16 ou 4	Salta se nem maior nem igual	JNGE TARGETLABEL
16 ou 4	Salta se não menor	JNL TARGETLABEL
16 ou 4	Salta se nem maior nem igual	JNGE TARGETLABEL
16 ou 4	Salta se não menor	JNL TARGETLABEL
16 ou 4	Salta se nem menor nem igual	JNLE TARGETLABEL
16 ou 4	Salta se não overflow setado	JNO TARGETLABEL
16 ou 4	Salta se não paridade setado	JNP TARGETLABEL
16 ou 4	Salta se positivo	JNS TARGETLABEL
16 ou 4	Salta se não zero	JNZ TARGETLABEL
16 ou 4	Salta se overflow setado	JO TARGETLABEL
16 ou 4	Salta se paridade setado	JP TARGETLABEL
16 ou 4	Salta se paridade par	JPE TARGETLABEL
16 ou 4	Salta se paridade impar	JPO TARGETLABEL
16 ou 4	Salta se sinal setado	JS TARGETLABEL
16 ou 4	Salta se zero	JZ TARGETLABEL

OPERAÇÃO:

```
if condition is true then do;
    IP <- IP + Disp16;
end if;
```

FLAGS:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

CMP	COMPARA DOIS OPERANDOS: os flags são setados como decorrência da subtração do operando da direita do operando da esquerda. Nenhum dos operandos são modificados.

FORMATOS:

Memória/Registro com Registro:

Clocks	**Operação**	**Exemplo de codificação**
3	flags <- Reg8 - Reg8	CMP BL,CL
9 + EA	flags <- Reg8 - Mem8	CMP BL,[BYTESOMETHING]
3	flags <-Reg16 - Reg16	CMP BX,CX
9 + EA	flags <- Reg16 - Mem16	CMP BX,[WORDSOMETHING]
9 + EA	flags <- Mem8 - Reg8	CMP [BYTESOMETHING],BL
9 + EA	flags <- Mem16 - Reg16	CMP [WORDSOMETHING],BL

Imediato para AX/AL

Clocks	**Operação**	**Exemplo de codificação**
4	flags <- AL - Immed8	CMP AL,5
4	flags <- AX - Immed16	CMP AX,400H

Imediato para Memória/Registro

Clocks	**Operação**	**Exemplo de codificação**
4	flags <- Reg8 - Immed8	CMP BL,32
10 + EA	flags <- Mem8 - Immed8	CMP [BYTESOMETHING],32
4	flags <-Reg16 - Immed16	CMP BX,1234H
10 + EA	flags <- Mem16 – Immed16	CMP [WORDSOMETHING],1234H
4	flags <- Reg16 - Immed8	CMP BX,32
10 + EA	flags <- Mem16 - Immed8	CMP [WORDSOMETHING],32

OPERAÇÃO:

flags <- LeftOpnd - RightOpnd

FLAGS:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

10. INSTRUÇÕES LÓGICAS, DE DESLOCAMENTO E DE ROTAÇÃO

O 8086/88 oferece um sub-conjunto de instruções que permitem realizar operações a nível de bit em um byte (8 bits) ou numa word (16 bits). Nas linguagens de alto nível estas manipulações nem sempre são diretas.

10.1. INSTRUÇÕES LÓGICAS

As instruções lógicas disponíveis no 8086/88 são **AND, OR, XOR** e **NOT** e são normalmente usadas para:

- resetar (reset) ou limpar (clear) um bit: 1 -> 0
- setar (set) um bit: 0 -> 1
- examinar bits (verificar seu valor individualmente)
- realizar máscaras para manipular bits

O formato de uso das instruções lógicas é:

```
AND   destino, fonte
OR    destino, fonte
XOR   destino, fonte
NOT   destino
```

A tabela verdade dos operadores lógicos relacionados acima é:

a	b	a **AND** b	a **OR** b	a **XOR** b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

a	**NOT** a
0	1
1	0

Alguns exemplos de uso das instruções lógicas:

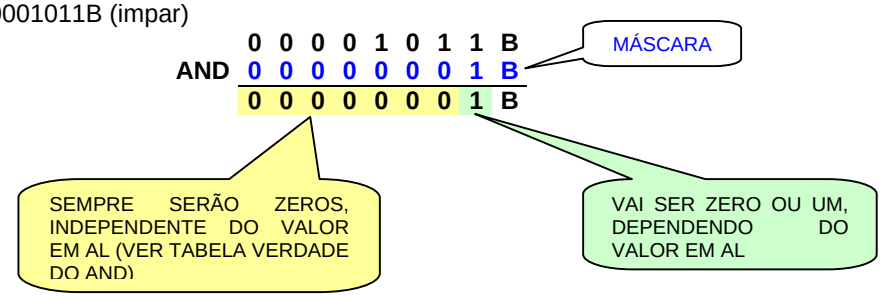
Ex10.1.: Verificação se um número é par ou impar - O valor do bit menos significativo de um número binário b_0 determina se o mesmo é par ou impar: quando for igual a zero é par e quando for igual a um é impar. Assim, aplicando a operação lógica AND sobre o dado a ser testado e uma máscara escolhida adequadamente podemos testar o estado do bit menos significativo, como mostrado na seqüência abaixo, que testa o valor armazenado no registro AL:

```
        AND   AL, 00000001B     ; o resultado desta operação será zero ou um, dependendo do valor
                                ; do b0 em AL. E por ser uma operação lógica, o flag de zero será
                                ; atualizado
        JZ    é_par             ; salta para o rótulo é_par  se o resultado do AND foi zero
                                ; se não efetuar o salto, o valor em AL era impar
        ..
        JMP   pula
é_par:  ...


pula:
```

Vamos observar o resultado da operação considerando alguns valores em AL

a) AL = 11 = 00001011B (impar)



b) AL = 22 = 00010110B (par)

```
      0 0 0 1 0 1 1 0 B
AND   0 0 0 0 0 0 0 1 B
      ---------------
      0 0 0 0 0 0 0 0 B
```

Ex10.2.: Conversão de minúscula p/ maiúscula e vice-versa – a tabela ASCII é uma representação em um byte de caracteres (ver apêndice A). Uma característica que a tabela ASCII possui com relação aos caracteres alfabéticos é que os códigos de uma certa letra maiúsculas e sua correspondente minúscula mudam somente no bit 5 (b_5). Assim, temos que para os códigos das letras maiúsculas, o bit 5 será sempre igual a zero e para as letras minúsculas será sempre igual a um, como por exemplo:

		b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
'a'	-	0	1	1	0	0	0	0	1
'A'	-	0	1	0	0	0	0	0	1

		b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
'b'	-	0	1	1	0	0	0	1	0
'B'	-	0	1	0	0	0	0	1	0

Uma possível implementação de seqüência de instruções para a conversão de **maiúscula -> minúscula** seria, considerando que a letra maiúscula está em AL:

```
OR      AL, 00100000B       ; a máscara e a instrução lógica foram escolhidas para ligar o bit 5 de
                            ; AL, sem alterar os outros bits.
```

A operação contrária (**minúscula -> maiúscula**), seria:

```
AND     AL,11011111B        ; a máscara e a instrução lógica foram escolhidas para zerar o bit 5 de
                            ; AL, sem alterar o outros bits.
```

O 8086/88 oferece ainda uma outra instrução lógica: **TEST**, que realiza a mesma operação da instrução AND, porém sem alterar o valor dos operandos. Como não altera o valor dos operandos, esta instrução é utilizada somente como a instrução que precede um desvio condicional. Por exemplo, a instrução AND AL,00000001B do exemplo 10.1, que foi usada para testar se o conteúdo de AL é par ou impar, está alterando o valor de AL. Para evitar que o valor de AL seja perdido, a instrução AND pode ser substituída por TEST, ficando então:

```
        TEST        AL, 00000001B       ; atualiza os flags, porém não altera o valor dos
                                        ; operandos
```

10.2. INSTRUÇÕES DE DESLOCAMENTO

Efetuam o deslocamento a nível de bit de um byte ou uma word. Algumas aplicações seriam:

- deslocar um bit para a esquerda é equivalente a multiplicar por 2
- deslocar um bit para a direita é equivalente a dividir por 2
- os bits deslocados para fora são perdidos

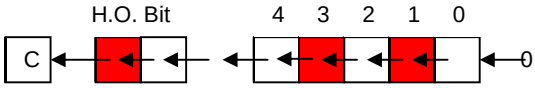
Formato das instruções:

```
        Sxx     destino, 1      ; usado para deslocar 1 bit
        Sxx     destino, CL     ; quando o deslocamento for > que 1 bit, deve ser feito
                                ; através de CL
```

Sxx corresponde a uma das instruções de deslocamento disponíveis:

SHL/SAL – delocamento para a esquerda

Os mnemônicos SHL e SAL são sinônimos. Eles representam a mesma instrução. Estas instruções movem cada bit no operando de destino a quantidade de vezes especificada no operando da direita. Bits zero preenchem as posições vazias à esquerda do operando de destino. A cada deslocamento, o bit mais significativo é copiado para o Carry Flag:



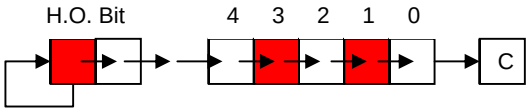
- O carry flag contém o ultimo bit deslocado da posição contendo o bit mais significativo do operando destino.
- O flag de zero vai sinalizar (=1) se o resultado no operando de destino for zero
- O flag de sinal vai conter o valor resultante no bit mais significativo do operando de destino.

Desde que um valor inteiro quando deslocado uma posição para a esquerda é equivalente a multiplicar o valor por 2, podemos realizar as seguintes operações de multiplicação

```
shl     ax, 1   ;Equivalente a AX*2
mov     cl, 2
shl     ax, cl   ;Equivalente a AX*4
mov     cl, 3
shl     ax, cl   ;Equivalente a AX*8
mov     cl, 4
shl     ax, cl   ;Equivalente a AX*16
mov     cl, 5
shl     ax, 5   ;Equivlente a AX*32
mov     cl, 6
shl     ax, 6   ;Equivalente a AX*64
```

SAR – deslocamento aritmético para a direita

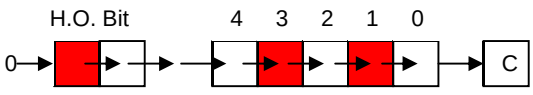
A instrução SAR desloca todos o bits do operando de destino um bit para a direita, replicando o bit mais significativo:



- O carry flag contém o ultimo bit deslocado da posição menos significativo do operando de destino.
- O flag de zero vai sinalizar (=1) se o resultado no operando de destino for zero
- O flag de sinal vai conter o valor resultante no bit mais significativo do operando de destino.

SHR – deslocamento lógico para a direita

A instrução SHR desloca todos os bits do operando de destino para a direita. Zeros são inseridos no bit mais significativo a cada bit deslocado

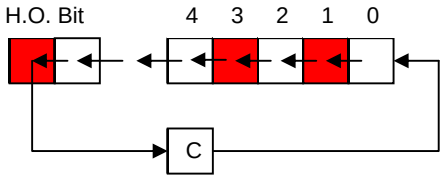


- O carry flag contém o ultimo bit deslocado da posição menos significativo do operando de destino.
- O flag de zero vai sinalizar (=1) se o resultado no operando de destino for zero
- O flag de sinal vai conter o valor resultante no bit mais significativo do operando de destino.

10.3. INSTRUÇÕES DE ROTAÇÃO

RCL – rotação para a esquerda com carry flag

A rotação para a esquerda com o carry flag, como o próprio nome diz, rotaciona bits para a esquerda no operando de destino através do carry flag e carrega seu valor atual para o bit menos significativo do operando de destino:

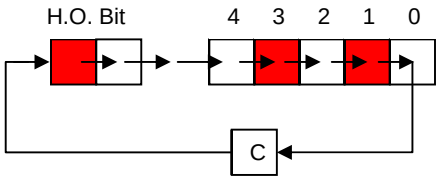


Notar que se o operando é rotacionado n+1 bits, onde n é o tamanho do operando (8 para byte e 16 para word), o resultado final no operando de destino não se altera.

- O carry flag contém o ultimo bit deslocado do bit mais significativo do operando de destino
- A instrução RCL não altera o valor dos flags de zero, sinal, paridade e carry auxiliary.

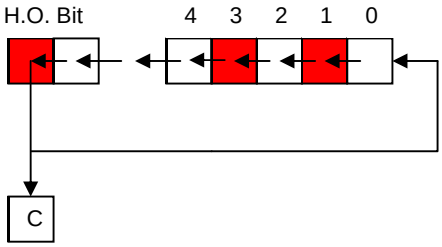
RCR – rotação para a direita com carry flag

Idem a anterior, porém efetuando a rotação para a direita



ROL – rotação para a esquerda sem carry

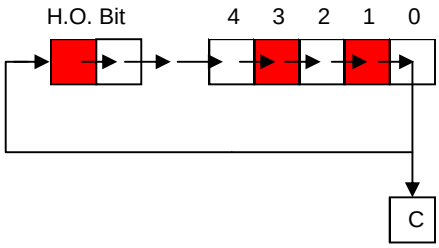
Similar a instrução RCL, ou seja, rotaciona o operando de destino para a esquerda o número de bits especificado. A diferença é que o bit mais significativo é deslocado diretamente para o bit 0. Uma cópia do bit mais significativo também é feita para o carry flag.



Os flags alterados são os mesmos para a instrução RCL.

ROR – rotação para a direita sem carry

Idem ao ROL, porém rotacionando para a direita.



11. SUBROTINAS

Antes de iniciar a apresentação das instruções e diretivas relacionadas à declaração, chamada e retorno de subrotinas vamos abordar as instruções que efetuam a manipulação de pilha. Estas instruções serão usadas para a passagem de parâmetros para as subrotinas, conforme veremos adiante.

11.1. INSTRUÇÕES DE MANIPULAÇÃO DE PILHA

O 8086/88 possui mecanismos internos (registradores/instruções) que implementam uma estrutura de dados de uma dimensão organizadas em algum trecho de memória. Esta estrutura é uma pilha (ou stack) com a seguinte política: o ultimo que entre é o primeiro que sai (LIFO = Last-In First-Out). A posição da pilha mais recentemente acrescida é o **topo da pilha**. Os registros do 8086 relacionados à pilha são:

SS -> registro de segmento cujo valor define o início do segmento de pilha (base da pilha).
SP -> seu valor corresponde ao endereço do topo da pilha (define o deslocamento do topo em relação à base).

No ambiente que estaremos trabalhando, o sistema operacional (DOS) se encarrega de reservar e inicializar os registros **SS:SP** para a área de memória que foi alocada para ser a pilha utilizada pelo programa. O tamanho desta pilha, em bytes, é indicada pela diretiva .**STACK** ***valor*** definida no programa fonte. A pilha cresce do topo para a base. As instruções disponíveis para a manipulação direta da pilha no 8086/88 são:

Instruções para colocar dados na pilha:

```
PUSH   fonte
PUSHF                ; salva o valor atual dos flags na pilha
```

Onde **fonte** é:

- um registrador de 16 bits
- uma palavra de memória ou variável de 16 bits (de tipo DW)

A execução de **PUSH** resulta nas seguintes ações:

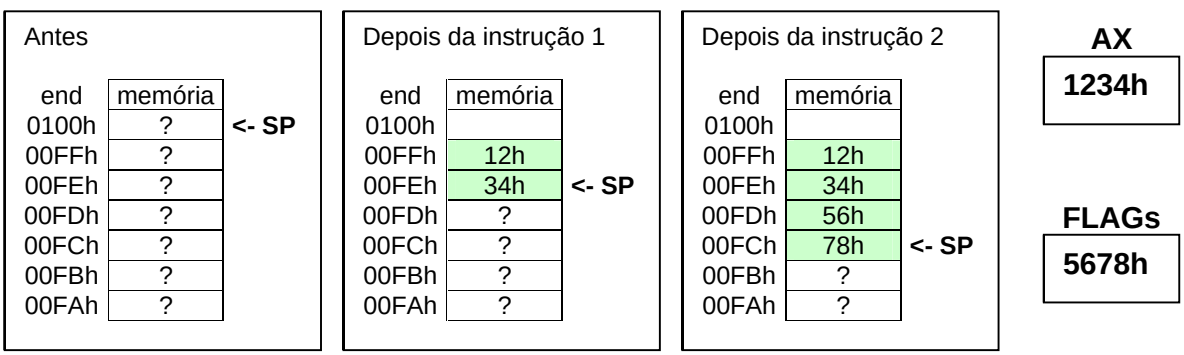
- o registrador **SP** (stack pointer) é decrementado de 2
- uma cópia do conteúdo da **fonte** é armazenada na pilha de forma que:
  - a posição **SS:SP** -> armazena o byte menos significativo da fonte
  - a posição **SS:(SP+1)** -> armazena o byte mais signitivativo
- o conteúdo da **fonte** não é alterado

A execução de PUSHF, que não possui operando, resulta:

- o registrador SP (stack pointer) é decrementado de 2
- uma cópia do conteúdo do registrador de FLAGs é armazenada no pilha

Exemplo de operação:

```
...
PUSH   AX      ; instrução 1
PUSHF          ; instrução 2
```



Instruções para retirar dados da pilha:

POP destino
POPF

Onde destino é:

- um registrador de 16 bits
- uma palavra de memória ou variável de 16 bits (de tipo DW)

A execução de POP resulta nas seguintes operações:

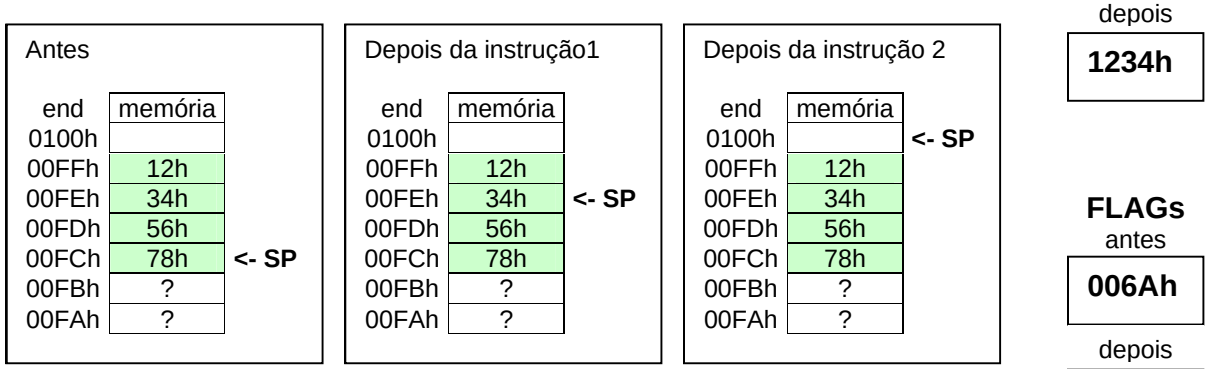
- o conteúdo das posições **SS:SP** (byte menos significativo) e **SS:(SP+1)** (byte mais significativo) é copiada para o destino
- o registrador SP (stack pointer) é decrementado de 2

A execução de POPF, que não possui operando, resulta:

- o conteúdo das posições **SS:SP** (byte menos significativo) e **SS:(SP+1)** (byte mais significativo) é copiada para o registrador de **FLAGs**
- o registrador SP (stack pointer) é decrementado de 2

Exemplo de operação:

```
...
POPF          ; instrução 1
POP   AX      ; instrução 2
```

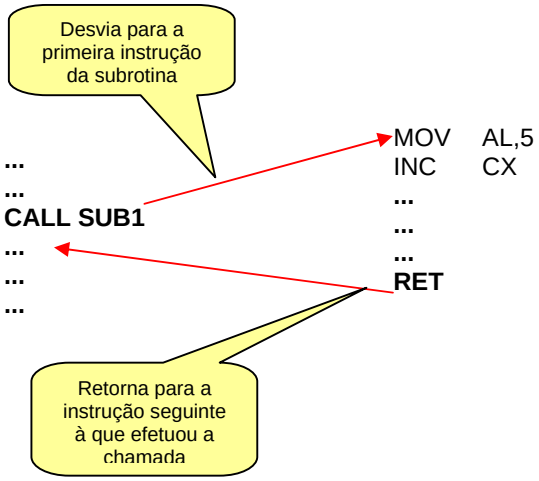


AX
antes
F0D3h
depois
1234h

FLAGs
antes
006Ah
depois
5678h

11.2. INSTRUÇÕES DE CHAMADA/RETORNO SUBROTINA

Subrotinas são blocos de instruções que são chamados para executar a partir de várias posições do programa. Cada vez que uma subrotina é chamada, as instruções que pertencem a subrotina são executadas, e então a execução retorna de volta para a instrução seguinte a que efetuou a chamada da subrotina.
As instruções do 8086/88 para chamar e retornar de uma subrotina são **CALL** e **RET**, respectivamente.



11.3.

11.4.

11.5.

11.6.

Para que a subrotina saiba retornar para a instrução seguinte à instrução CALL que efetuou a sua chamada, o endereço da instrução para a qual ela deve retornar deve ser preservado. O local onde este endereço é armazenado é a pilha e por ser um endereço composto de dois elementos (segmento e offset), existe dois tipos de instrução de chamada de subrotina: uma que guarda somente o offset do endereço de retorno e uma outra que guardo o segmento e o offset do endereço de retorno. A seguir, será descrito o funcionamento das instruções **CALL/RET** que armazenam apenas o offset:

Instrução de chamada de subrotina intra-segmento:

CALL nome

A execução da instrução **CALL** vai resultar em:

- O conteúdo do **IP**, registro que contém o **offset do endereço** da próxima instrução a ser executada (instrução após a instrução **CALL**) é armazenado na pilha
- **IP** recebe o **offset do endereço** da primeira instrução da subrotina chamada

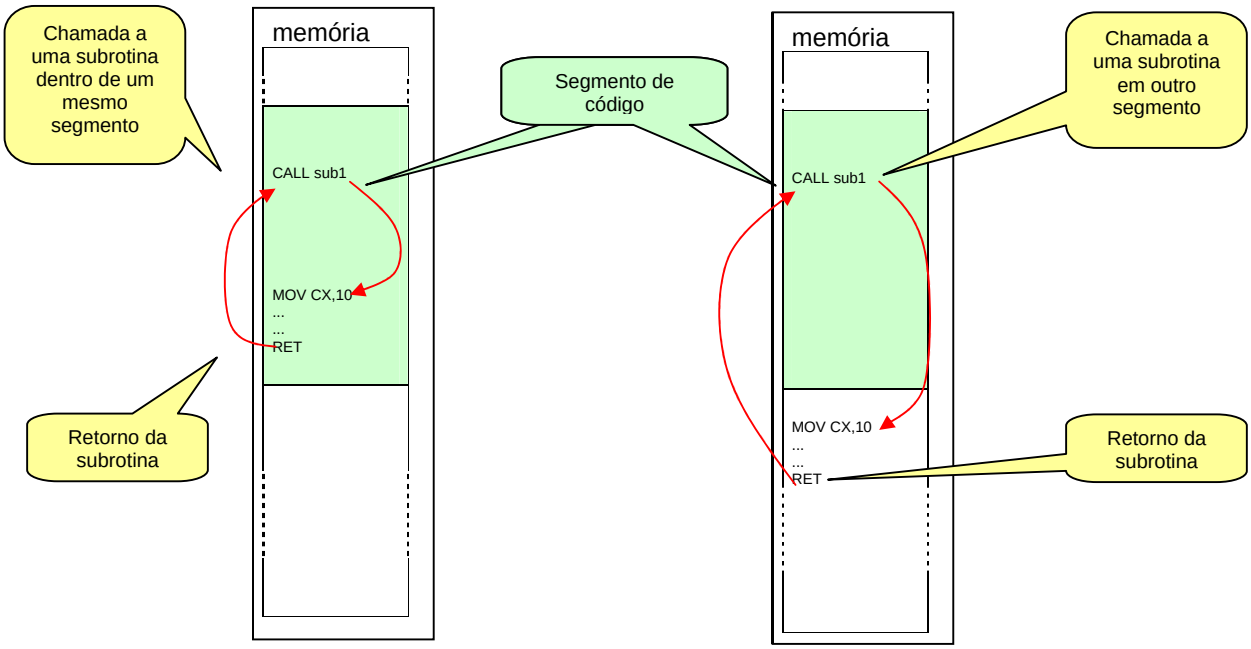
Instrução de retorno:

RET

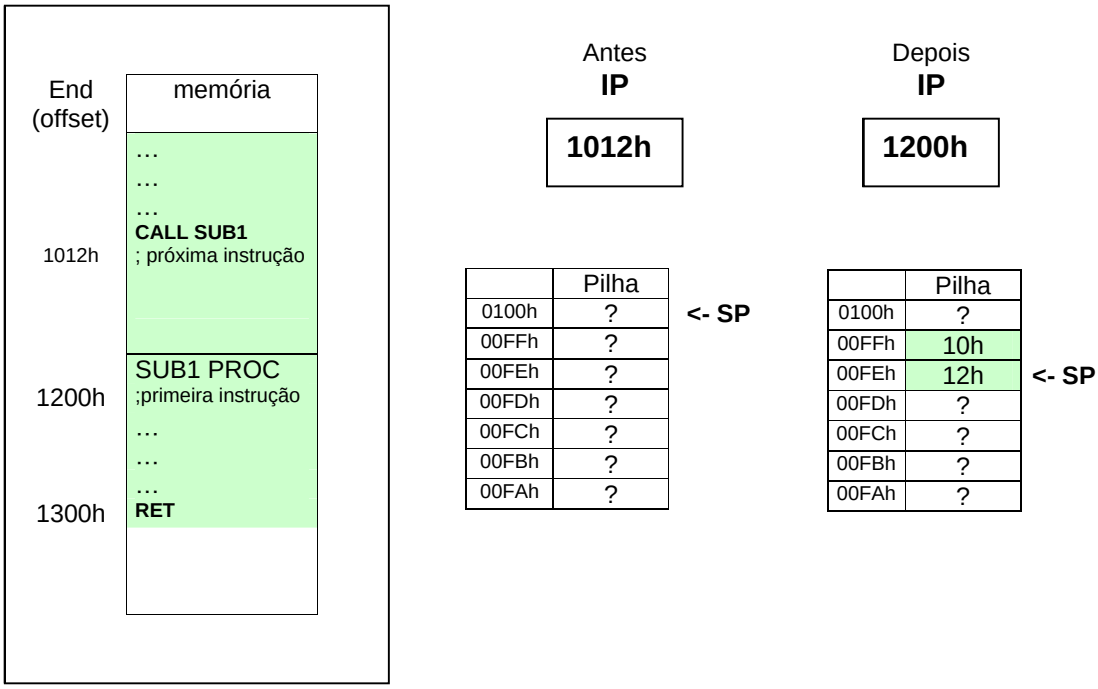
A execução da instrução **RET** vai resultar em:

- Retira do topo da pilha o offset do endereço de retorno e armazena este valor no registro IP

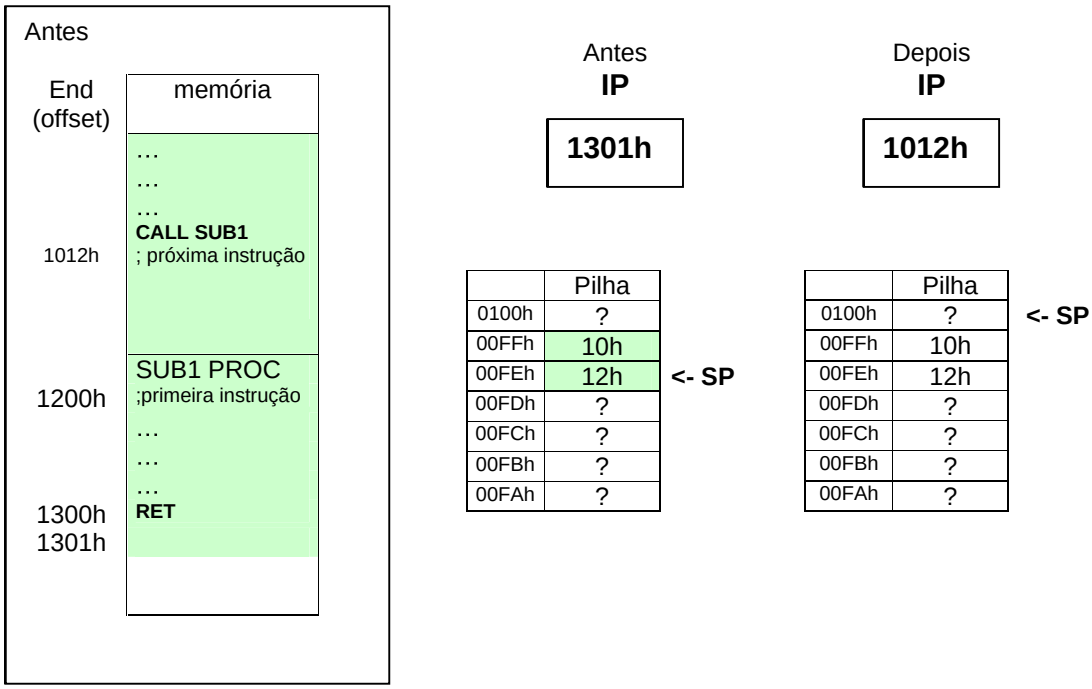
O pares de instruções CALL/RET que armazenam somente o offset do endereço retorno permite a chamada de subrotinas cujo código estejam armazenadas no mesmo segmento. Caso contrário é preciso usar as instruções CALL/RET que armazenam segmento e offset do endereço de retorno.



Mecanismo de chamada (intra-segmento):



Mecanismo de retorno (intra-segmento):



Diretiva de declaração de subrotina

Como foi visto acima, uma instrução de chamada de subrotina (**CALL**) pode ser codificada de duas formas e o montador assembler deve ser informado qual o código de máquina que vai ser gerado. Uma diretiva **PROC** é usada para definir um nome para a subrotina assim como o seu tipo, que pode ser NEAR ou FAR. A sintaxe para o uso da diretiva PROC é:

Nome_sub **PROC** *<tipo>*

; corpo da subrotina

Nome_sub **ENDP**

Onde:
Nome_sub é o nome atribuído à subrotina, que vai ser usado no operando da instrução CALL quando se desejar chamar a subrotina.
<*tipo*> informa se o código de máquina gerado pelo montador assembler vai ser FAR (inter-segmento, armazena o segmento e o offset do endereço de retorno) ou se vai ser NEAR (intra-segmento, armazena somente o offset do endereço de retorno). Este parâmetro é opcional e quando não é definido, o montador assembler vai codificar as instruções CALL/RET em função da opção na diretiva **.MODEL**.

Diferentemente das linguagens de alto nível, a definição de subrotina em assembly não define uma estrutura de blocos nem escopo para símbolos declarados do interior da diretiva **PROC/ENDP**. Assim, apesar de não recomendável, podemos ter a situação abaixo:

```
P1      PROC  NEAR
        MOV   AX, 15
lab:    ADD   DX, AX
P2      PROC  NEAR          ; inicia a declaração de outra subrotina, sem ter terminado
                            ; primeira
        MOV   AX, 0
        CMP   AX, [cont]
        JE    salto
        DEC   [cont]
salto:  MOV   AX, 0
        RET                 ; termina P2 e P1 aqui
P2      ENDP
        CMP   DX, 10        ; esta instrução nunca será executada
        JE    lab
        RET
P1      ENDP
```

11.7. PASSAGEM DE PARÂMETROS

Quando a quantidade de parâmetros for pequena, estes podem ser passados via registrador, o que simplifica bastante a implementação. Por exemplo, uma subrotina para zerar um vetor de bytes qualquer na memória pode ser definida como recebendo os seguintes parâmetros: BX -> endereço do primeiro byte do vetor na memória e CX o tamanho do vetor em bytes.

Uma outra forma de passagem de parâmetros seria através da pilha. Para demonstrar a utilização da passagem de parâmetros pela pilha, vamos considerar um programa bastante simples em Pascal e o seu equivalente em linguagem assembly

```
PROGRAM TESTE;

VAR
        Var1    : integer;
        Var2    : integer;
        Total : integer;



PROCEDURE SOMA(V1, V2: integer);
BEGIN
        Total := V1 + V2;
END;



BEGIN
        Var1 := 3;
        Var2 := 2;
        Soma(Var1, Var2);
END.
```

```
.MODEL small
.STACK 100h
.DATA
        Var1    DW    ?       ; declaração de
        Var2    DW    ?       ; variáveis
        Total   DW    ?       ; equivalentes


.CODE

Soma    PROC  NEAR
        PUSH  BP              ; BP é o registro usado para
        MOV   BP, SP          ; acessar memória
        MOV   AX, [BP+6]      ; busca o primeiro parâmetro na pilha
        MOV   AX, [BP+4]      ; busca o segundo parâmetro na pilha
        MOV   [Total], AX
        RET   4               ; termina a subrotina, retirando
Soma    ENDP                  ; da pilha os parâmetros

Inicio:
        MOV   AX,@data
        MOV   DS, AX
        MOV   [Var1], 3
        MOV   [Var2], 5
        PUSH  [Var1]          ; carrega na pilha o 1° parâmetro
        PUSH  [Var2]          ; carrega na pilha o 2° parâmetro
        CALL  Soma            ; chama a subrotina
        MOV   AH, 4Ch         ; termina o programa
        INT   21h
END Inicio
```

No momento em que os parâmetros são buscados: MOV AX, [BP + 6] e MOV AX, [BP + 4] a pilha possui o seguinte estado: