Linguagem de Máquina para 8085

Conteúdo

Introdução 2
2. Sintaxe 4
2.1 Diretivas 4
2.2. Instruções de Montagem 6
2.3. Instruções de Máquina 6
3. Sintaxe das Definições 7
4. Sintaxe dos Dados 10
5. Sintaxe das Instruções 14
5.1. Label 15
5.2. Código de Operação 17
5.3. Operandos 18
5.4. Comentários 20
6. Sintaxe do Operando 21
7. Especificação Formal da Sintaxe do “Montador” 25
7.1. Notação empregada 26
7.2. Notação BNF do “Montador” 27
8. Mensagens de Erro produzidas pelo “Montador” 30
8.1. Introdução 30
8.2. Mensagens de Erro 30
8.2.1. Erros em Diretivas 30
8.2.2. Erros em Definições 33
8.2.3. Erros em Declarações de Dados 35
8.2.4. Erros em Instruções 39
8.2.5. Erros nas Expressões 43
8.2.6. Erros em Labels 46
8.3. Mensagens de Advertência 48

Introdução

Se examinarmos o conteúdo da memória de um computador, um programa aparece como uma série de dígitos hexadecimais indistinguíveis uns dos outros. O processador ou CPU interpreta estes dígitos como códigos de instruções, endereços ou dados.

Seria possível escrever um programa neste formato, porém resultaria um processo lento e demorado. Por exemplo, o programa mostrado a seguir armazena dados na memória como é mostrado:

repete:	**MOV A,M**	**0100**	**7Eh**
	CPI 0h	**0101**	**FEh 00h**
	JNZ fim	**0104**	**C2h 0Dh 01h**
	MVI M, FFh	**0107**	**36h FFh**
	INX H	**0109**	**23h**
	JMP repete	**010A**	**C3h 00h 01h**
fim:	**HLT**	**010D**	**76h**

- O byte 7E é interpretado pelo processador como o código da instrução MOV de transferência de memória (indicada por HL) para o registrador A.
- Os bytes FE 00 correspondem ao código da instrução de comparação CPI, do acumulador com o dado imediato 00h.
- Os bytes C2 0D 01 indicam uma instrução de salto condicional para o endereço 010Dh.
- Os bytes 36 FF realizam a transferência de FF para a memória.
- O byte 23 indica que o par de registradores HL deve ser incrementado como se fosse um registrador de 16 bits.
- Os bytes C3 00 01 indicam um salto incondicional para o endereço 1000h.
- O byte 76 é a instrução HALT ou parada do processador.

O texto da esquerda é um programa escrito em “linguagem assembly”, enquanto que o da direita é o código retirado direto da memória. As diferenças entre ambos são obvias quanto à legibilidade.

O programa anterior realiza uma tarefa muito simples. Utilizando o par de registradores HL como endereço de memória carrega o seu conteúdo no acumulador. Se o conteúdo for diferente de 00 termina, caso contrário, muda o conteúdo por FFh e continua no endereço seguinte. Inclusive em um programa tão simples vemos o trabalho de trabalhar diretamente sobre a memória.

Sem dúvida, existe um problema adicional. Suponhamos que desejamos introduzir uma instrução mais ao programa proposto. Por exemplo, suponhamos que se quer especificar um endereço inicial de memória em HL. Isto se pode fazer com a seqüência de dígitos hexadecimais: 21 LL HH, onde LL representa a parte baixa doa endereço de memória e HH a alta.

	LXI H,A000h	0100	**21h 00h A0h**
repete:	MOV A,M	0103	7Eh
	CPI 0h	0104	FEh 00h
	JNZ fim	0107	**C2h 10h 01h**
	MVI M, FFh	010A	36h FFh
	INX H	010C	23h
	JMP repete	010D	**C3h 03h 01h**
fim:	HLT	0110	76h

Em negrito foram marcadas as mudanças sobre o programa original. Como vemos, na memória a introdução de uma nova instrução da lugar a efeitos colaterais que obrigam a modificar outras do programa. O risco de cometer uma falha é maior.

A ilegibilidade do programa agrava o risco de erro se for tentado adicionar mais instruções e efetuar mais mudanças.

Para evitar esta forma complexa e tediosa de trabalhar, sobretudo em programas de certa complexidade, o primeiro passo está em utilizar uma linguagem de montagem,

que proporciona uma notação das instruções completamente legível e evita ao programador ter que referir-se a endereços específicos de memória.

A linguagem montadora é, em síntese, uma seqüência de instruções que são convertidas em um código hexadecimal executável pela máquina através de um programa chamado “Montador”.

O “Montador” converte o programa fonte, escrito na “linguagem assembly”, em seu equivalente em hexadecimal, denominado programa objeto. O programa objeto é muito similar a representação na memória que terá o programa.

2. Sintaxe

Em um programa montador serão distinguidos os seguintes tipos de elementos: Diretivas, Instruções de montagem e Instruções da máquina.

2.1 Diretivas

As Diretivas oferecem informação ao montador sobre o tipo de elementos que este vai encontrar a seguir e o endereço de memória onde deve colocar os dados. São caracterizadas por irem precedidas por um ponto.

Figura 5.1. Estrutura de uma Diretiva.

Dispomos de três diretivas distintas. Estas diretivas nos permitem fazer definições (*define*), introduzir dados (*data*) e introduzir instruções (*org*). Cada diretiva declara, portanto, um bloco dentro do programa.

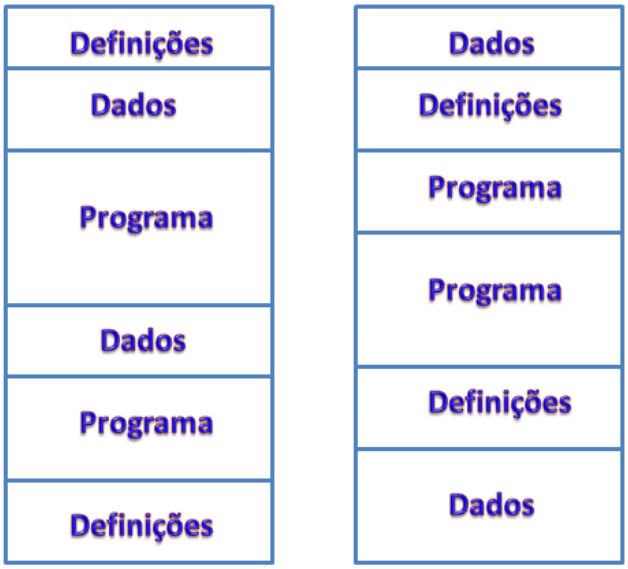


Figura 5.2. Organização Aleatória dos Blocos.

A disposição de blocos dentro do programa montador não está sujeita a nenhum tipo de restrição inicial. Além disso, é possível declarar vários blocos de um mesmo tipo.

Igualmente, nenhum bloco é imprescindível. Podem-se construir programas sem declarações, dados e, inclusive, instruções. Ainda que esta ultima pareça pouco razoável nos ser útil se quisermos unicamente introduzir dados na memória para usá-los com outro programa. Por ultimo poderemos construir um programa vazio, que será montado como tal. Geralmente, a forma mais usual de um programa será a que segue:

Figura 5.3. Organização Usual dos Blocos.

A distribuição usual por blocos é obrigatória?

Não, pode-se organizar as diretivas a seu gosto. A distribuição usual é a recomendada. A distribuição representada na figura anterior é obtida empregando as diretivas nesta ordem: DEFINE, DATA e por último ORG.

2.2. Instruções de Montagem

É um tipo de especial de instruções que unicamente são levadas em conta no processo de montagem do programa, porém que realmente não tem execução dentro da maquina uma vez terminado aquele processo.

As instruções de montagem são empregadas unicamente no processo de montagem. Desta forma, não aparecem de forma explicita dentro do código objeto, sem dúvida aparecem de forma implícita.

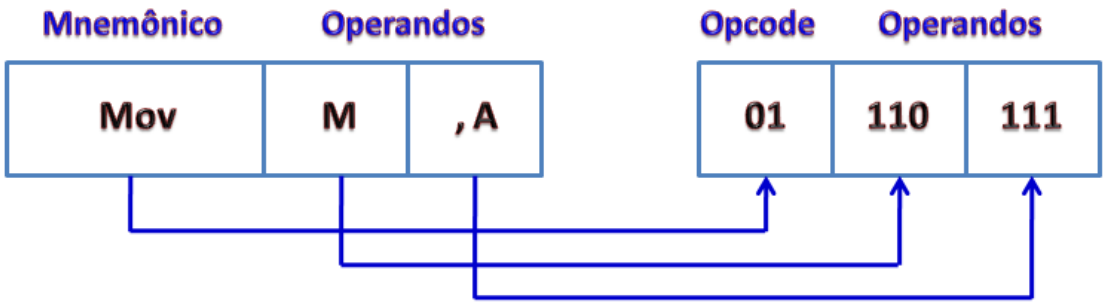
As instruções de montagem diferem conforme o bloco, ou diretiva previamente declarada, na qual estejamos. Por isso, veremos cada uma delas dentro de seu âmbito correspondente. Por hora somente diremos que existem duas classes, uma para fazer definições e outra para declarar dados.

2.3. Instruções de Máquina

Diferentemente das anteriores as instruções de maquina ou simplesmente instruções, “rodam” ou são executadas na máquina. Cada instrução tem seu código de operação correspondente na máquina.

“As instruções são introduzidas após a diretiva ***“.org”*****,** o que é o mesmo procedimento de quando se declara um bloco de programa. Mais adiante veremos com mais detalhes a sintaxe dentro deste bloco, aqui somente faremos uma pequena introdução.

Basicamente, o montador realiza uma tradução entre um mnemônico com alguns operadores para um número. Cada mnemônico representa uma operação e são denominados assim porque permitem nos recordarmos facilmente do que realiza a operação correspondente:



Instrução: Mover o Acumulador para a Memória

Neste caso o montador converte de forma automática a expressão: **“mov M, A”** em **78h.**

3. Sintaxe das Definições

O objetivo de uma definição é simplesmente colocar um nome em um número. Durante a “montagem”, cada vez que nos encontremos com um operando que contém o nome este será substituído de forma automática pelo valor correspondente com o qual foi definido.

Mais adiante, se for necessário alterar este número por alguma razão, unicamente se terá que modificar a declaração, o que nos evita a tarefa de buscar todas as aparições

do número no texto, comprovar se se trata realmente do número que queremos alterar e modificá-lo.

“As definições somente podem ser realizadas se previamente tiver sido declarada a diretiva de definições “.define”. Caso contrário será produzido um erro, já que o montador não entenderá a seqüência de léxicos.

La estrutura de um bloco de definições é a seguinte:

. define

[definição]

[definição]

[definição]

....

Existe alguma restrição sobre a tabulação e uso de espaços na diretiva e no restante das definições?

Não, as tabulações mostradas anteriormente são somente indicativas. Não tem porque realizá-las de forma estrita. A palavra ‘define’ pode estar junto ao ponto sem ser separada por espaços e as definições no início da linha.

Cada definição tem a seguinte estrutura sintática:

Nome	Valor	[Comentário]

Figura 5.4. Estrutura de uma Definição.

Onde "*nome*" declara o nome, enquanto que em "*valor*" é determinado o valor correspondente. *Nome* e *Valor* devem estar separados pelo menos por um espaço. Por exemplo:

. define

maximo FFFFh

minimo 0

limite 0CA1h

real 166

Os identificadores maximo, minimo, limite e real são nomes, enquanto que FFFFh, 0, 0CA1h e 166 são valores relacionados respectivamente aos nomes.

Os identificadores de nome não podem ser números, ou seja, devem começar ao menos por uma letra. Como máximo, devem ter um tamanho não superior a 255 caracteres. Igualmente, um nome já usado em uma definição não pode voltar a ser empregado novamente em nenhuma outra definição. Neste caso o montador mostra uma mensagem de advertência e assinala ao nome o valor correspondente da primeira definição.

Ainda que é um caso pouco provável, posto que se seguirmos a organização usual por blocos não é possível, um nome de definição não poder ser um nome já usado em um label declarado anteriormente. Na seção "***Sintaxe das Instruções***" quando são explicados os labels se faz referencia a esta limitação. Também se pode encontrar um exemplo deste erro no capitulo "**Mensagens produzidos pelo Montador**".

Cada definição é uma instrução de montagem. Como vemos não há uma instrução de maquina análoga a ela, de forma que não foca expressa de forma explicita. Sem dúvida, a instrução é tomada pelo montador que substitui cada nome pelo valor correspondente.

4. Sintaxe dos Dados

Dentro de um programa se estabelece una clara diferença entre dados e instruções. Os dados se constituem basicamente nos operandos das instruções.

A distribuição na memória de dados e instruções pode estar separada e claramente diferenciada. Isto se deve a que tanto dados como instruções são tratados da mesma forma pela maquina segundo a situação na qual esteja. Depende de como são tratados pela maquina.

Os dados podem ser de três tipos. Foram distinguidos três tipos distintos de dados simples com os que podem-se trabalhar no montador. A diferença fundamental entre estes tipos é o tamanho, como mostra a Tabela 5.1.

Tabela 5.1. Tamanho dos dados.

Tipo	Tamanho	Exemplos
Byte	8 bits	1, 10, 255, 127, -127, ...
Word	16 bits	FE00h, 65535, FFFFh, ...
String	8 bits x nº caracteres	“Este é um exemplo”, “Saída do programa”, ...

Geralmente a maioria dos dados serão do tipo *Byte*, posto que o 8085 somente opera com dados de 8 bits. Os dados de tipo *Word* são empregados com maior probabilidade para definir endereços de memória já que o 8085 é capaz de endereçar palavras de até 16 bits. As cadeias de caracteres somente proporcionam uma maior legibilidade ao programa montador posto que podem serem definidas alternativamente mediante uma seqüência de *Bytes*:

“Este é um exemplo”

45h 73h 74h 6Fh 20h 65h 73h 20h 75h 6Eh 20h 65h 6Ah 65h 6Dh 70h 6Ch 6Fh

Na primeira linha mostra-se a cadeia original, enquanto que na segunda a cadeia de bytes correspondente. Como vemos é muito mais simples de construir e modificar a notação para cadeias de caracteres.

A estrutura de um bloco de dados é a seguinte:

. data <endereço origem>

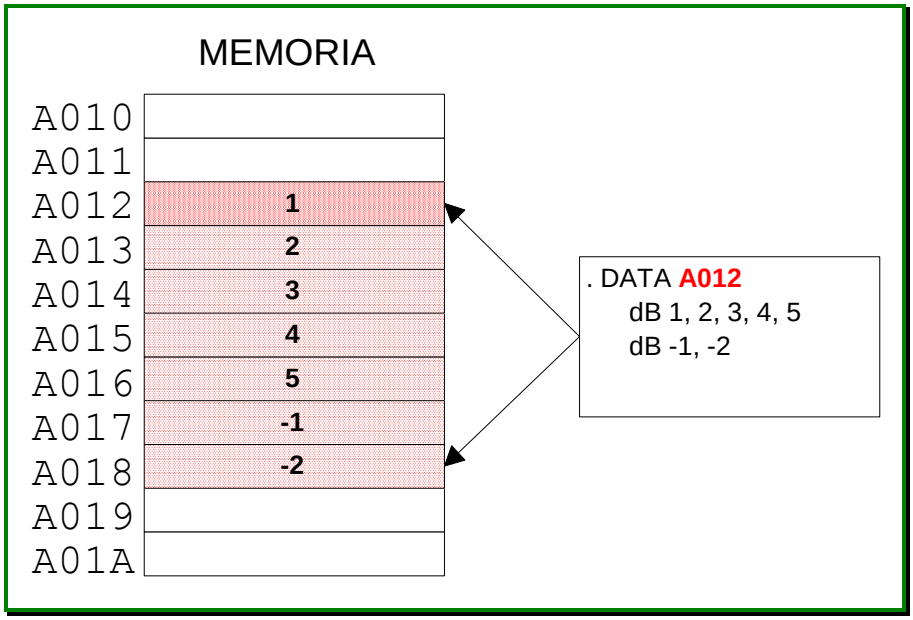
[declaração dados]

[declaração dados]

[declaração dados]

....

No endereço origem é indicado o endereço de memória no qual se começara a introduzir os dados.



Cada declaração de dados tem a seguinte sintaxe:

Figura 5.5. Estrutura de uma Declaração.

Onde:

- O *prefixo* indica o tipo de dado que vamos a introduzir. Os possíveis prefixos coincidem com os tipos descritos anteriormente:
    - *dB*: declaração de Bytes ou números de 8 bits.
    - *dW*: declaração de Words ou números de 16 bits.
    - *dS*: declaração de cadeias de caracteres.

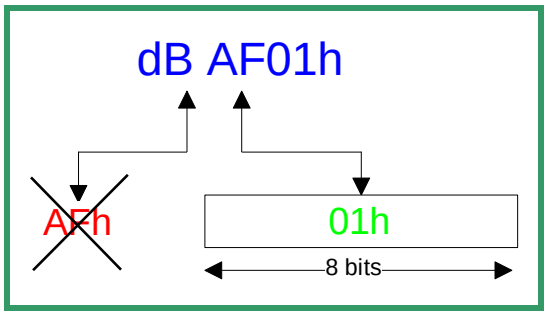
- Os prefixos não são sensíveis a "Caps Lock". Por exemplo, 'db', 'dB', 'DB' e 'Db' são reconhecidos igualmente por um prefixo de declaração de Bytes.

- Os *dados* são simplesmente uma seqüência de números ou expressões lógicas e aritméticas separadas por vírgulas ou una cadeia de caracteres.

Quando se realiza uma declaração de um dado (quer seja explicitamente ou como resultado de uma expressão aritmética e/ou lógica) como um byte, este é armazenado no próximo byte útil da memória.

Quando a declaração é de uma "Word" **os 8 bits menos significativos da expressão são armazenados no próximo byte útil da memória, enquanto que os oito bits mais significativos são armazenados no seguinte**. Como se vê, o endereço está colocado na memória em ordem inversa. Normalmente, os endereços são encontrados na memória desta forma. Esta operação se utiliza basicamente para criar uma tabela de endereços constantes.

Se a representação do dado excede o número de bits com que tenha sido declarado então se informa ao programador mediante uma advertência do "montador". **Unicamente ficam armazenados os bits menos significativos.**



Na continuação é mostrado um exemplo de declaração de dados:

```
.DATA 100h
  dB    1, 2 , 3, 4, 5, 6, 7, 8, 9, 0      ;valores
  dB    32, 25, FFh, 32, -5, -120
  dW          0, FFFFh, 01A0h      ;minhas posições de memória
  dS          "Pulse uma tecla"    ;mensagem para continuar
  dS          "Final do programa"  ;mensagem de saída
```

Além disso, é possível empregar "labels" durante uma definição, isto nos permite poder referenciar mais adiante no programa um determinado dado. ***O uso de "labels" durante uma definição fornecem bases do que, nas linguagens de mais alto nível, serão as variáveis.***

Finalmente a estrutura de uma declaração de dados é:

Figura 5.6. Estrutura Entendida de uma Declaração.

Na próxima seção veremos com mais detalhes os "labels" e como são empregados nas instruções. Desta forma veremos a utilidade real da definição de "labels" durante uma declaração de dados.

5. Sintaxe das Instruções

A estrutura de um bloco de instruções é a seguinte:

. org <endereço origem>

[instrução]

[instrução]

[instrução]

....

As instruções da linguagem de montagem vêm dadas por uma série de regras que formam a sintaxe desta. Dentro de cada instrução existem quatro partes ou campos separados:

Figura 5.7. Estrutura de uma Instrução.

Os campos Label e Comentário são prescindíveis e não tem porque estarem necessariamente em cada instrução do montador. O Código de Operação, também conhecido por *OpCode*, é completamente necessário se quisermos definir uma instrução. Os operadores dependerão do Código de Operação correspondente já que há instruções que não requerem nenhum operador enquanto que outras necessitam de vários deles.

Estes campos são separados por espaços em branco, não existe nenhuma restrição sobre o número destes que deve haver entre cada dois campos, sempre que ao menos exista um.

5.1. Label

É um campo de utilização opcional que, quando está presente, pode ter um comprimento de 1 a 255 caracteres. O primeiro caractere do label deve ser uma letra do alfabeto. Se não fosse assim, como veremos mais adiante poderia ser confundido com um endereço real de memória. Estes caracteres devem ir seguidos de dois pontos (:). Os códigos de instrução são especialmente definidos pelo montador e não podem ser utilizados como labels. Nosso montador é capaz de diferenciar entre eles, pelo que esta restrição não está presente, sem dúvida se recomenda não mesclar labels de salto com códigos de instrução.

O objetivo de um label é referenciar um endereço de memória. Isto é, quando um label está situado na declaração de um dado ou diante de uma instrução de programa esta fazendo referencia ao endereço de memória na qual se encontra o referido dado ou instrução.

Graças aos labels o programador não necessita calcular manualmente o endereço de um determinado dado que necessita nem o de uma instrução para a qual se deve saltar. Desta maneira se pode simplificar enormemente a tarefa do programador com respeito aos saltos e a carga ou armazenamento de dados.

Se levarmos em conta o significado de um label é muito fácil reconhecer as limitações que o uso destes leva. Por um lado, um label define somente um endereço. Não pode ser repetido. Portanto, isto não é possível:

```
Salto:  mov a, b
                ...
Salto:  call sub
                ...
                jmp Salto
```

É obvio que o montador não pode determinar qual endereço é o que deve ir a instrução JMP.

Não obstante, a situação contraria é possível (ainda que pouco útil), isto é, determinar o mesmo endereço a dois labels. Aa seguinte seqüência de instruções é válida:

```
Salto:

Salto2: mov a, b
        ....
        jmp Salto
```

Agora podemos justificar porque um label deve ao menos começar por uma letra. Se não for assim, o label poderia ser una seqüência completa de dígitos. Se isto ocorrer o montador será incapaz de distinguir um label de um número igual. Esta ambigüidade daria lugar a falhas.

Tabela 5.2. Exemplo de labels.

Endereço	Instrução
0100	Mov a, b
0101	0100: Mov c, d
0102	Jmp 0100

Como podemos ver existe uma clara ambigüidade sobre o que realmente se quer codificar. Por um lado se poderia interpretar como que o salto do endereço 0102 seria para onde indica o label 0100 (endereço 0101). Porém por outro lado também seria possível dizer que realmente se quer saltar para o endereço 0100.

Tampouco é possível que um label tenha o mesmo nome que uma definição previa. Como no caso anterior o montador não é capaz de distinguir a que elemento, se label ou definição, está fazendo referencia o nome encontrado.

Por ultimo, e semelhante aos nomes de definição, os labels são sensíveis a maiúsculas e minúsculas.

Resumindo:

- Os labels são sensíveis a maiúsculas e minúsculas.
- Devem iniciar ao menos por uma letra.
- Devem ser menores de 255 caracteres.
- Não podem coincidir com definições anteriores.
- Não devem coincidir com labels anteriores.

5.2. Código de Operação

É o campo mais importante da instrução já que nele se define a operação a ser realizada pela máquina (soma, subtração, salto, etc.). Cada uma das instruções tem um código determinado, que é privativo da mesma, e que deve aparecer no campo do código. Por exemplo, as letras "JMP" definem exclusivamente a operação de "salto incondicional" e nenhuma outra instrução. Não existem duas instruções com o mesmo código nem dois códigos para uma mesma instrução.

Depois das letras do campo de código, deve haver no mínimo um espaço em branco:

Jmp Salto	**Salto incondicional a Salto (label)**
Adi 7Bh	**Somar 7Bh com acumulador**
Mov a, b	**Transferir registrador B ao acumulador**

A cada seqüência de letras que definem exclusivamente uma operação se denomina mnemônico. Os mnemônicos diferentemente dos labels e dos identificadores de definições não são sensíveis a letras maiúsculas nem minúsculas. Isto é “Jmp”, “JMP”, “jmp” ou “JmP” se referem ao mesmo mnemônico.

5.3. Operandos

A informação contida neste campo é usada conjuntamente com o campo do código afim de definir com precisão a operação a ser executada pela instrução. Segundo o conteúdo do campo de código, o campo do operando pode não existir, ou consistir em uma cifra ou palavra, ou mesmo em dois, separados ambas por uma vírgula.

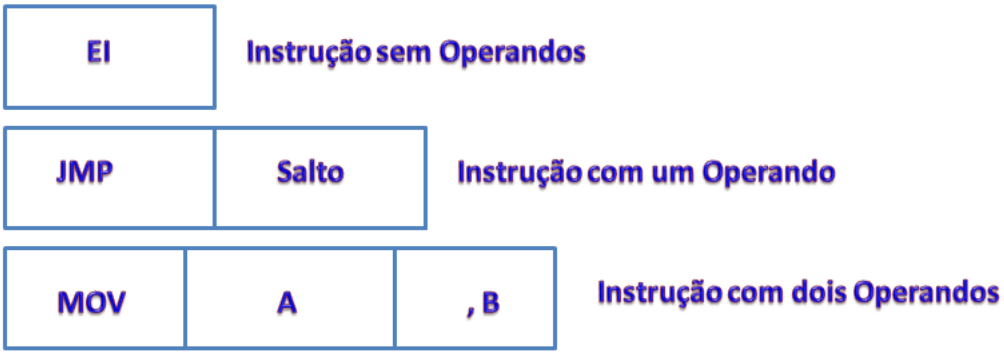


Figura 5.8. Tipos de Instruções.

Há quatro tipos de informação válidos como campo de operandos:

- **Registrador**. Um registrador (o código indicativo de uma referencia a memória) definido como fonte ou destino de dados em uma operação. Um número pode ser usado para especificar o registrador ou referencia a memória mencionados, podendo-se obter o dito número a partir de uma expressão, porém o número finalmente avaliado deve estar entre 0 e 7. A correspondência é mostrada na tabela 5.3.

Tabela 5.3. Números de registradores.

Valor	Registrador
0	B
1	C
2	D
3	E

Valor	Registrador
4	H
5	L
6	Referencia a memória
7	Acumulador (A)

Por exemplo, a instrução MVI permite a carga imediata do segundo operando no registrador indicado pelo primeiro operado. Isto é, usos típicos desta instrução seriam:

MVI A, 1 **MVI H, dir_alta** **MVI L, dir_baixa**	**Transferir 1 ao acumulador** **Transferir dir_alta (definição) ao registrador H** **Transferir dir_baixa (definição) ao registrador L**

Cada um dos registradores podem ser substituídos de diversas formas:

MVI 7, 1 **MVI 8/2, dir_alta** **MVI regL, dir_baixa**	**7 é o acumulador** **8/2 é 4 que corresponde a H** **RegL é uma definição com valor 5 (L)**

- **Par de registradores**. Um par de registradores utilizado como fonte ou destino de uma operação com dados. Os pares de registradores são especificados como se vê na tabela 5.4.

Tabela 5.4. Pares de registradores.

Especificação	Par de Registradores
B	**Registradores B e C**
D	**Registradores D e E**
H	**Registradores H e L**
PSW	**Um byte que indica o estado dos bits de condição e o registrador A.**
SP	**O registrador de 16 bits do ponteiro da stack.**

Por exemplo:

INX B	**Incrementar BC (como um registrador de 16 bits)**
PUSH PSW	**Guardar na pilha os bits de estado e o registrador A**
LXI D,0h	**Carregar HL com 0h**

- **Dato imediato**. Um dado expresso de forma imediata na instrução. O dado pode ser resultado de uma expressão. Por exemplo:

ADI 5+5	**Somar 10 ao conteúdo do acumulador**
ORI 0Fh	**OR do acumulador com 0Fh**
LXI D,10h+(MOV A,B)	**Carregar HL com 10h+78h**

- **Endereço de 16 bits.** Um endereço de memória de 16 bits ou o label de uma instrução na memória. Por exemplo:

JMP 34F0h	**Saltar para o endereço 34F0**
LXI H,A000h	**Carregar HL com A000h**

5.4. Comentários

A única regra que rege a utilização dos comentários é que estes devem ser precedidos de um ponto e vírgula (;).

AQUI: MVI C, DADH ;isto é um comentário.

O campo do comentário pode aparecer somente em uma linha, apesar de que não exista instrução na mesma:

```
; comentário sem instrução
```

6. Sintaxe do Operando

Existem nove formas de especificar o operando. Estas nove formas são detalhadas na continuação:

1. **Dado Hexadecimal**. Todo número hexadecimal é expresso na base 16 cuja representação emprega os dígitos de 0 ao 9 e as 6 primeiras letras do alfabeto (A, B, C, D, E e F). São os mais comuns quando se trabalha em "linguagem assembly". Os dados hexadecimais são seguidos de uma letra H. Por exemplo:

```
A2h (162), 10AFh (4271), FFFFh (65535), 11Ah (282)
```

2. **Dado Decimal**. Expressos na base 10 são os números que todos conhecemos com dígitos entre 0 e 9. Podem ir seguidos da letra D, ou sozinhos. Por exemplo:

```
0, 1, 255d, 35, 1050d
```

3. **Dado Octal**. Expressos em base 8, utilizam somente os dígitos do 0 ao 7. Devem ir seguidos da letra O ou Q para que sejam reconhecidos como tais. Por exemplo:

```
777q (511), 123o (83), 242q (162), 22o (18)
```

4. **Dato Binário**. Números binários são expressos na base 2 (0,1). Vão seguidos da letra B. São empregados geralmente para definir mascaras de bits. Por exemplo:

0101011101B (349), 11110000B (240)

5. **O conteúdo atual do contador de programa**. Este é definido com o caractere $ e equivale ao endereço da instrução na execução.

6. **Uma constante ASCII**. Um ou mais caracteres ASCII encerrados entre aspas simples.

'a' (65), 'b' (66), '*'(42),'?' (63)

7. **Uma instrução encerrada entre parêntesis.** Uma instrução entre parêntesis pode ser empregada como operando. O valor concreto do operando será o código hexadecimal com o qual instrução é codificada.

(MOV A,B) (65), (INX H) (35), (PUSH PSW) (245)

8. **Identificadores de Labels**. Já que lhes é assinalado um valor numérico pelo próprio montador, como vimos no caso dos registradores representados por números. Ou mesmo labels definidos pelo programador que aparecem no campo do label de outra instrução ou declaração de dado.

9. **Expressões Lógicas e Aritméticas**. Todos os operadores descritos anteriormente são expressões. As expressões lógicas e aritméticas são expressões unidas mediante os operadores: + (soma), - (subtração ou mudança de sinal), * (multiplicação), / (divisão), MOD (módulo), os operadores lógicos NOT, AND, OR, XOR, SHR (rotação para a direita), SHL (rotação para la esquerda), e parêntesis a direita e esquerda.

Todos os operadores tratam seus argumentos como quantidades de 16 bits e geram como resultados quantidades de 16 bits. Cada operador realiza a seguinte operação:

- O operador **+** gera a soma aritmética de seus operandos.

A01Bh + 00FFh = A11Ah

- O operador **-** gera a subtração aritmética de seus operandos, quando se usa como subtração (operador binário) ou como aritmética negativa quando se utiliza como mudança de sinal (operador unário).

A01Bh - 00FFh = 9F1Ch

- O operador ***** indica o produto aritmético dos dois operandos.

A01Bh * 00FFh = 9F7AE5h

- O operador **/** calcula o quociente inteiro entre os dois operandos, o resto da divisão é descartado.

A01Bh / 00FFh = A0h

- O operador **MOD** calcula o resto da divisão entre os dois operandos descartando o quociente.

A01Bh MOD 00FFh = BBh

- O operador **NOT** realiza o complemento de cada bit do operando.

NOT 0110110b = 1001001b

- O operador **AND** (ou **&**) realiza a operação lógica AND bit a bit entre os operandos.

011101b **AND** **111000b** **=** **011000b**

- O operador **OR** (ou **|**) realiza a operação lógica OR bit a bit entre os operandos.

011101b **OR** **111000b** **=** **111101b**

- O operador **XOR** (ou **^**) realiza a operação lógica O-EXCLUSIVO bit a bit entre os operandos.

011101b **XOR** **111000b** **=** **100101b**

- Os operadores **SHR** e **SHL** realizam um deslocamento do primeiro operando a direita e esquerda respectivamente o número de posições definidas pelo segundo operando, introduzindo zeros nas novas posições.

011101b **SHR** **3** **=** **000011b**

011101b **SHL** **3** **=** **101000b**

O programador deve assegurar de que o resultado gerado por uma destas operações cumpre os requisitos necessários. Se não for assim os bits mais significativos não poderão ser armazenados e se perderiam. Por exemplo:

Figura 5.4. Estrutura de uma Definição.

MVI A, NOT 0	**NOT 0 é FFFFh, MVI espera um dado imediato de 8 bits.**
MVI A, -1	**-1 é FFFFh em complemento de dois.**
MVI A, -1&(FFh)	**Forma correta de especificação.**

Múltiplos operadores podem estar presentes em uma expressão. Esta claro que a ordem em que estes são aplicados vai dar lugar a diferentes resultados. Por esta razão as expressões produzidas pelos operadores devem ser avaliadas na ordem de prioridade mostrada na tabela 5.5.

Tabela 5.5. Operadores de uma expressão.

Prioridade	
1	**Expressões entre parêntesis**
2	**Multiplicação (*), Divisão (/), MOD, SHL, SHR**
3	**Suma (+), Subtração (-)**
4	**NOT**
5	**AND**
6	**OR, XOR**

No caso das expressões entre parêntesis, o conteúdo entre os mesmos deve ser avaliado primeiro.

7. Especificação Formal da Sintaxe do "Montador"

A estrutura da "linguagem de montagem" é dada por uma série de regras que formam a sintaxe deste. A linguagem de Montagem é o primeiro escalão dentro dos níveis de abstração da maquina, ou seja, está muito próxima da própria maquina, pelo que sua complexidade como linguagem é significativamente menor que qualquer outra linguagem de mais alto nível.

A sintaxe de uma linguagem de programação se especifica mediante a gramática independente do contexto que o gera. Uma gramática independente do contexto é uma quádrupla **(N, T, P, S)** onde:

a) **N** é o conjunto de símbolos não terminais da linguagem,
b) **T** é o conjunto de símbolos terminais da linguagem,
c) **P** é o conjunto de regras de produção, e
d) **S** é o axioma ou símbolo inicial da gramática,

e além disso assegura que todas as regras de produção adotam a forma:

$$A \to \alpha \text{, onde } A \in N, \alpha \in (N \cup T)^*$$

Em outras palavras, é uma gramática de tipo 2.

7.1. Notação empregada

Para especificar formalmente a sintaxe das linguagens de programação se pode usar BNF. BNF é uma metalinguagem que se usa para especificar a sintaxe das linguagens de programação. Isto se consegue especificando em BNF a gramática que gera a linguagem correspondente.

BNF, Backus Naur Form, estabelecido por John Backus e Peter Naur (Junho 1.959, 2 de janeiro 1.960). A equivalência entre as gramáticas independentes do contexto e BNF, no sentido de ter a mesma potência expressiva, foi demonstrada por S. Ginsburg, e H.G. Pice em 1.962

Em BNF a gramática de uma linguagem de programação é expressa por meio de suas regras de produção, pelo que a sintaxe com a qual são escritas estas regras deverá por claramente de manifesto quais são:

- O símbolo inicial,
- Os símbolos não terminais e
- Os símbolos terminais da gramática da linguagem

Para ele o símbolo inicial da gramática deverá aparecer na parte esquerda da pereira regra de produção e o resto dos símbolos não terminais deverão aparecer na parte esquerda de pelo menos uma regra de produção. Diz-se que essa regra define o símbolo não terminal.

A partir do conceito de gramática independente do contexto, é obvio que nenhum símbolo terminal poderá aparecer na parte esquerda de nenhuma regra de produção.

A ordem natural que se pode seguir ao escrever as regras de produção é dada uma regra, escrever na continuação dela as regras correspondentes aos símbolos não terminais que apareçam em sua parte direita. Se algum ou alguns, destes símbolos são comuns a varias regras a regra que define este símbolo será colocada ao final de todas elas. Habitualmente as regras de produção em cuja parte direita somente aparece símbolos terminais figuram ao final da especificação.

Para especificar as regras de produção, a notação BNF é composta dos seguintes metasímbolos:

- **< >** que se usam como delimitadores dos símbolos não terminais da gramática ao escrever as regras de produção.
- **::=** que se utiliza para separar as partes esquerda e direita das regras de produção, e se lê "*se define como*".
- **|** que se usa como separador das diversas alternativas que podem aparecer na parte direita de uma regra de produção e se lê "*o*".
- **[ ]** para representar que o encerrado entre os colchetes é opcional.
- **{ }** para representar que o encerrado entre os colchetes se repete 0 ou várias vezes.

7.2. Notação BNF do "Montador"

<Programa> ::= *{<Bloco>}*

<Bloco> ::= *<Bloco definição>* | *<Bloco declaração>* | *<bloco Programa>*

<Bloco definição> ::= *<diretiva define> {<definições>}*

<Bloco declaração> ::= *<diretiva dados> {<declarações>}*

<Bloco Programa> ::= *<diretiva programa> {<instruções>}*

<diretiva define> ::= *<ponto>* **DEFINE** *[***;** *<comentários>]*

<diretiva dados> ::= *<ponto>* **DATA** *<expressão> [***;** *<comentários>]*

<diretiva programa> ::= *<ponto>* **ORG** *<expressão> [***;** *<comentários>]*

<definição> ::= *<identificador> <expressão> [***;** *<comentários>]*

<declaração> ::= *[<identificador> :] <prefixo> <expressão>* **,** *{<expressão>} [***;** *<comentários>]*

<instruções> ::= *<código operação> [<expressão reg>] [* **,** *<expressão reg>] [***;** *<comentários>]*

<expressão reg> ::= *<expressão>* | *<registrador>*

<expressão> ::= *<termo> { <operador fraco> <termo> }*

<termo> ::= *<elemento> { <operador forte> <elemento> }*

<elemento> ::= *< identificador>* | *<constante>* | **(** *<expressão>* **)**

<identificador> ::= *<letra>* | *<identificador> <letra>* | *<identificador> <digito>*

<constante> ::= *<constante hex> | <constante decimal> | <constante octal> | <constante bin> | <constante caractere>*

<constante hex> ::= *<digito hex> {<digito hex>}* **H**

<constante dec> ::= *<digito> {<digito>} [***D***]*

<constante octal> ::= *<digito oct> {<digito oct>}* **O** *| <digito oct> {<digito oct>}* **Q**

<constante bin> ::= *<digito bin> {<digito bin>}* **B**

<constante caractere> ::= ***‘*** *<letra> | <digito dec>* ***’***

<código de operação> ::= **ANA** | **AND** | **ACI** | **ADI** | .... | **XCHG** | **XTHL**

<prefixo> ::= **dB** | **dW** | **dS**

<registrador> ::= **A** | **B** | **C** | **D** | **E** | **F** | **H** | **S** | **PSW**

<operador fraco> ::= **+** | **-** | **AND** | **NOT** | **OR**

<operador forte> ::= ***** | **/**

<digito> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

<digito hex> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F**

<digito oct> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**

<digito bin> ::= **0** | **1**

<letra> ::= **a** | **b** | **c** | ... | **z** | **A** | **B** | **C** | ... | **Z**

<ponto> ::= **.**

8. Mensagens de Erro produzidas pelo “Montador”

8.1. Introdução

As mensagens de erro são geradas quando existe algum problema no código fonte a ser montado. Estes erros provocariam, caso o montador seguisse convertendo o arquivo, que o programa resultante ficasse totalmente incorreto, então no caso de erros o processo de montagem é terminado a cada vez que é produzido um erro.

Também é possível que o montador gere mensagens de advertência. Estas mensagens de advertência indicam incoerências ou ambigüidades no código fonte. Estas ambigüidades não geram um programa totalmente incorreto e assim o processo de compilação continua. Ainda que funcione este conservará certos erros que devem ser eliminados. Mais adiante veremos estes casos.

As mensagens de erro são mostradas na cor vermelha, enquanto que as mensagens de advertência são mostradas em amarelo.

A seguir descreveremos de forma detalhada as mensagens de erro e de advertência que são informadas pelo montador. Além disso, exemplos nos quais estes erros são produzidos.

8.2. Mensagens de Erro

8.2.1. Erros em Diretivas

As Diretivas oferecem informação ao montador sobre o tipo de elementos que se vai encontrar na continuação e o endereço de memória onde deve dispô-los (se corresponderem). São caracterizadas por irem precedidas por um ponto.

Erro:	Cada bloco deve começar por uma diretiva. Linha *<Nº linha>*

Cada bloco que for definido deve começar pela diretiva correspondente. Se não for assim se lança este erro. Usualmente este erro somente aparece quando iniciamos a introdução de instruções no programa ou declarações de dados a partir do zero, sem definir nenhuma diretiva previa.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplos de erro:

```
; .org 100H
      mvi b, 5
salto : mov a,b
      add b
            jmp salto
```

```
;.data 10H

dB   0, 127, FFh
dS   “Erro na diretiva”
```

Erro:	Identificador precedendo diretiva não valida. Linha *<Nº linha>*

Antes do ponto (.) que marca o começo de uma diretiva não pode haver nada salvo espaços em branco ou tabulações. No caso de que isto não se cumpra será gerada a mensagem de erro anterior.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo do erro:

```
Diretiva .org 100H
        mvi b, 5
salto : mov a,b
        add b
              jmp salto
```

	Erro:	Diretiva não reconhecida. Linha *<Nº linha>*

Somente existem as diretivas mencionadas anteriormente, isto é, DEFINE para as definições, DATA para as declarações de dados e ORG para o começo do programa. O montador não é sensível a "Caps Lock" destas letras, sempre e quando corresponde a alguma diretiva valida.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de erro:

```
.minha_diretiva 100H
        mvi b, 5
salto : mov a,b
        add b
              jmp salto
```

	Erro:	Endereço incorreto na diretiva. Linha *<Nº linha>*

As diretivas DATA e ORG necessitam um endereço de memória inicial a partir do qual se deve colocar os dados e instruções respectivamente. Tanto se não for introduzido este endereço assim como se for inválido será gerado este erro.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplos de erro:

.org 1A mvi b, 5 salto : mov a,b add b jmp salto	.data dB 0, 127, FFh dS “Não há endereço origem”

No primeiro caso a expressão é errônea, já que para definir um valor hexadecimal é necessário introduzir o sufixo ‘h’. O número não é reconhecido corretamente dando lugar ao erro. Este caso se pode estender a qualquer outro no qual a expressão que decide o endereço de memória for incorreta.

No segundo caso não foi colocado nenhum endereço. Isto da também dá lugar a um erro de montagem que gera a mensagem atual.

8.2.2. Erros em Definições

O objetivo de uma definição é simplesmente por um nome em um número. Durante a montagem, cada vez que nos encontremos com um operando que contém o nome este será substituído de forma automática pelo valor correspondente com o que foi definido.

Erro:	Valor '*<Valor>*' associado a definição '*<Definição>*' incorreto. Linha *<Nº linha>*

O valor que foi associado a um identificador de uma definição não é correto. Em uma definição somente se pode assinalar um valor (ou qualquer expressão que se resolva em um valor) a um identificador. Esta mensagem de erro indica que o valor que se foi introduzido não é valido.

- *Valor*. Seqüência de símbolos que não podem ser reconhecidas como um valor.
- *Definição*. Nome do identificador ao que se tenta determinar um valor incorreto ou não reconhecido.

- *Nº da linha*. Linha na qual se encontra o erro.

Exemplos de erro:	

.define
  pvp 180A
  iva 16
.org 100H
 	
...
	.define
	  sem_valor
	  varios 16 17
	.org 100H
	
	...

No primeiro exemplo o valor associado a “pvp” é incorreto. É um valor em hexadecimal ao que lhe falta o sufixo ‘h’. Isto é extensível a toda expressão que contenha erros.

No segundo exemplo são mostrados dois erros. No primeiro não foi especificado um valor ao que foi identificado “sem_valor”, o segundo tenta determinar vários valores de cada vez na mesma linha.

	Erro:	Colisão com a declaração previa do label: '*<Nome>*'. Linha *<Nº linha>*

Pretende-se empregar um nome identificador em uma definição que já tenha sido previamente empregada para um label.

- *Nome*. Identificador da declaração que já tenha sido empregado anteriormente.
- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo do erro:

```
.org 100H

salto: mvi a,1
       add a
       jmp salto


.define

       salto 1000h
```

	Erro:	O identificador '*<Nome>*' não é valido para uma definição. Linha *<Nº linha>*

O nome que se quer assinalar para a definição não é válido. Isto se deve a que contém caracteres inválidos ou que começa por um número.

Recordemos que o primeiro caractere de um nome de definição deve ser una letra do alfabeto. Se não for assim, como já vimos antes poderia ser confundido com um número empregado no programa.

- *Nome*. Nome da definição que não é válido.
- *Nº de linha*. Linha na qual se encontra o erro.

Exemplos de erro:	
.define	.define
1salto 1000h	12345 1000h

8.2.3. Erros em Declarações de Dados

Dentro de um programa é estabelecida uma clara diferença entre dados e instruções. Os dados constituem basicamente os operandos das instruções.

	Erro:	Declaração não valida. Expressão incorreta: '*<Valor>*'. Linha *<Nº linha>*

Existe um erro nos valores introduzidos em uma declaração de dados. Quer seja de 8 (bytes) ou 16 (words) bits, se o valor correspondente não for calculado, devido a algum erro de digitação, será gerado este erro.

- *Valor*. Seqüência de símbolos que não é possível reconhecer como um valor.
- *Nº de linha*. Linha na qual se encontra o erro.

Exemplos de erro:	
.define	.define
1salto 1000h	12345 1000h

	Erro:	Declaração incorreta. Não foram abertas “aspas”. Linha *<Nº linha>*

Na declaração de uma cadeia de caracteres não foram abertas aspas antes de introduzir a seqüência de letras correspondente. As aspas são necessárias para delimitar a cadeia que se quer armazenar na memória. Por esta razão é necessário que as cadeias de caracteres sejam expressas delimitadas por aspas.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de erro:

```
.data 0H
  dS Não foram abertas aspas”

.org 100H

salto: mvi a,1
       add a
       jmp salto
```

Erro:	Declaração incorreta. Não foram fechadas as aspas. Linha *<Nº linha>*

Como antes, porém o erro contrário. Após abrir aspas é necessário indicar o final da seqüência de caracteres, isto se realiza mediante o fechamento das aspas. Ainda que pareça uma formalidade, o fechamento das aspas pode evitar erros do programa que se devem a simples erros de digitação.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de erro:

```
.data 0H
  dS "Não foram fechadas as aspass    ; Isto também ficaria

.org 100H

salto: mvi a,1
       add a
       jmp salto
```

Como vemos, o problema de introduzir cadeias de caracteres é que se estas não forem delimitadas corretamente podem introduzir caracteres espúrios.

Erro:	Declaração incorreta. Somente se pode introduzir uma cadeia. Linha *<Nº linha>*

O prefixo de declaração de cadeias (dS) somente permite introduzir uma cadeia de caracteres, a diferença que dB e dW não é possível introduzir seqüências consecutivas na mesma linha separadas por vírgulas.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de erro:

```
.data 0H
  dS “Cadeia valida”,”Cadeia não válida”

.org 100H

salto: mvi a,1
       add a
       jmp salto
```

	Erro:	Prefixo '*<Prefixo>*' desconhecido. Linha *<Nº linha>*

O montador somente reconhece três prefixos distintos. Estes são, o prefixo de inteiros de 8 bits ou Bytes (*dB*), o prefixo de palavras de 16 bits ou Words (*dW*) e o prefixo de cadeias de caracteres ou Strings (*dS*).

O montador não é sensível a “Caps Lock” dos prefixos, porém qualquer outro prefixo é rechaçado gerando esta mensagem de erro.

- *Prefixo.* Seqüência de caracteres que o montador não reconhece como prefixo.
- *Nº de linha*. Linha na qual se encontra o erro.

Exemplos de erro:

```
.data 0H
  dB FFh
  dW FFFFh
  dS “Cadeia valida”
  dI 0a123          ;Prefixo inválido
  iva 16            ;Prefixo inválido
  mvi a,1           ;Prefixo inválido

.org 100H
```

```
salto: mvi a,1
       add a
       jmp salto
```

8.2.4. Erros em Instruções

As instruções da linguagem de montagem são dadas por uma série de regras que formam a sintaxe deste.

	Erro:	Mnemônico '<*Mnemônico*>' desconhecido. Linha <*Nº linha*>

Os mnemônicos representam o código de cada instrução. Cada uma das instruções tem um código determinado, que é privativo da mesma, e que deve aparecer no campo do código. O código montador é claramente definido por este.

- *Mnemônico.* O mnemônico não é reconhecido como una instrução própria do processador 8085.
- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de erro:

```
.data 0H
  dB FFh
  dW FFFFh
  dS “Cadeia valida”

.org 100H

salto: mvi a,1
       paddsb a,b
     add a
       jmp salto
```

	Erro:	Sintaxe incorreta na instrução. Primeiro operando inválido. Linha *<Nº linha>*

O primeiro operando da instrução não é um operando válido. Isto pode ocorrer devido a diferentes causas.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplos de erro:	
.org 100H *salto: mvi a, 5* *add a* *jmp salto+(3*)*	*.org 100H* *salto: mvi a,1* *push a* *jmp salto*

No primeiro exemplo é produzido este erro devido a que a expressão introduzida é errônea.

Toda vez que é gerado este erro também é mostrado o erro da expressão correspondente.

No segundo caso se tenta empregar um registrador não valido para a instrução (push a), por esta razão também é mostrado um erro.

	Erro:	Sintaxe incorreta na instrução. Segundo operando não é valido. Linha *<Nº linha>*

O segundo operando da instrução não é um operando válido. Isto pode ocorrer devido as diferentes causas explicadas anteriormente.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplos de erro:	
.org 100H salto: mvi a, 5+3*(5+) add a jmp salto	.org 100H salto: mov a, 1 add a jmp salto

No primeiro exemplo é produzido este erro devido a que a expressão introduzida é errônea. Quando é gerado este erro também é mostrada a expressão correspondente.

No segundo caso se tenta empregar a função *mov* com uma carga imediata, a qual não é possível (para isso existe *mvi*). Aqui também é gerada uma mensagem de erro.

	Erro:	Sintaxe incorreta na instrução. Falta operando. Linha *<Nº linha>*

A informação contida no campo operando é usada conjuntamente com o campo do código (mnemônico) afim de definir com precisão a operação a ser executada pela instrução.

Segundo o conteúdo do campo do código, o campo do operando pode não existir, ou consistir em uma cifra ou palavra, ou até mesmo os dois, separados ambos por uma vírgula.

Si a instrução necessitar mais operandos dos que os especificados será produzido este erro.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de erro:

```
.org 100H

salto: mov a
       add a
            jmp
```

MOV é uma instrução que necessita necessariamente dois operandos. Se for somente introduzido um será gerado um erro.

Da mesma forma *JMP* necessita um operando.

	Erro:	Sintaxe incorreta na instrução. Sobra operando. Linha *<Nº linha>*

Situação contraria a anterior na qual não falta um operando, e sim o contrario, há operandos demais.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de erro:

```
.org 100H

salto: mov a,b,c
       add a
            jmp salto, salto2
```

Qualquer instrução no 8085 não requer mais de dois operadores. MOV tem três, o que gera um erro.

Da mesma forma *JMP* somente necessita um operando.

	Erro:	Sintaxe incorreta na instrução. MOV não permite dois endereços de memória como operandos. Linha *<Nº linha>*

A instrução de transferência permite três tipos de transferências: transferência entre registradores (endereçamento de registrador), transferência desde a memória (endereçamento indireto de registrador) e transferência para a memória (endereçamento indireto de registrador). Em nenhum caso permite a transferência por sua vez desde e para a memória, isto é, especificar ambos operandos como endereços de memória.

- *Nº de linha*. Linha na qual se encontra o error.

Exemplo de erro:

```
.org 100H

salto: mov M,M
      add a
          jmp salto
```

8.2.5. Erros nas Expressões

Uma expressão é uma seqüência de números e operadores que podem ser resolvidas. No montador, diferente de outras linguagens de mais alto nível, as expressões deverão ser resolvidas no tempo da compilação e não da execução, já que o montador somente traduz instruções, não compila. Por esta razão, as expressões no montador não contêm variáveis nem registradores.

	Erro:	Expressão incorreta. Falta operando na expressão entre parêntesis. Linha

		<Nº linha>

Nas expressões é possível utilizar parêntesis. Se realizando uma operação nos esquecemos de colocar um operando e fechamos o parêntesis será gerado este erro.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplos de erro:
(5+)
16*75+(-)
(‘A’-15*)

	Erro:	Expressão incorreta. Operando trás operador não encontrado. Linha *<Nº linha>*

O operador não encontra o operando sobre o que se aplica.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplos de erro:
5+
16*75+
‘A’-15*

	Erro:	Expressão incorreta. Operador não reconhecido. Linha *<Nº linha>*

Se for empregado um operador diferente dos disponíveis.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplos de erro:

Sqrt(5)
Solve(x+5=0)
5 Modulo 2

Erro:	Expressão incorreta. Falta operador em expressão entre parêntesis. Linha *<Nº linha>*

Ao definir uma operação de maior prioridade mediante os parêntesis não foi especificado o operador correspondente para a referida operação.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplos de erro:
(5 16)
(16*5 75)
((‘A’ 15) (5+8))

Erro:	Expressão incorreta. Não foram fechados todos os parêntesis. Linha *<Nº linha>*

Foram abertos mais parêntesis dos que foram fechados.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplos de erro:
(16*75
(‘A’-15)*(5+5
(NOT((5+4)*6)

	Erro:	Expressão incorreta. Divisão por zero. Linha *<Nº linha>*

Na expressão se tenta realizar uma divisão por zero.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplos de erro:
7/0
(55*13)/NOT(FFh)
15/((16*32+2)-(64*8)-2)

8.2.6. Erros em Labels

O objetivo de um label é referenciar um endereço de memória. Isto é, quando uma label é situado na declaração de um dado o diante de uma instrução de programa que esta fazendo referencia ao endereço de memória na qual se encontra o dado ou instrução.

	Erro:	Não foi dado nome ao label. Linha *<Nº linha>*

Este erro aparece quando foram colocados dois pontos (:), porém não foi dado nome ao suposto label.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de erro:

```
.org 100H

  mvi b, 5
: mov a,b
  add b
  jmp salto
```

Como vemos, ainda que exista o símbolo indicativo do label (:) este realmente não existe.

	Erro:	Identificador de label inválido. Linha *<Nº linha>*

Quando o nome de um label é antecedido por caracteres diferentes de espaço ou tabulação é gerado este erro. O nome empregado para uma label não pode conter espaços em branco.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de erro:

```
.org 100H

                mvi b, 5
Label de salto: mov a,b
          add b
                jmp 123
```

	Erro:	O identificador '*<Nome>*' não é valido para o label. Linha *<Nº linha>*

O nome que se quer assinalar ao label não é valido. Isto se deve a que, ou ele contem caracteres inválidos, ou que inicia por um número. Recordemos que o primeiro caractere do label deve ser uma letra do alfabeto. Se não for assim, como já vimos antes poderia ser confundido com um endereço real de memória.

- *Nome*. Identificador assinalado ao label que não é válido.
- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de erro:

```
.org 100H

          mvi b, 5
123 :   mov a,b
      add b
          jmp 123
```

	Erro:	Colisão com a definição previa de: *<Nome>*. Linha *<Nº linha>*

Se tenta estabelecer uma label com um nome de identificação que já tenha sido previamente empregado em uma definição.

- *Nome*. Identificador do label que já tenha sido empregado anteriormente.
- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de erro:

```
.define
  salto 100h
.data (25)

.org 100H

          mvi b, 5
salto : mov a,b
      add b
          jmp salto
```

Como vemos, existe uma ambigüidade sobre de onde se deve saltar. O identificador *salto* pode referir-se ou ao label ou definição.

8.3. Mensagens de Advertência

	Advertência:	Declaração previa do label: '*<Nome>*'. Linha *<Nº linha>*

O nome com o que se quer definir o label foi previamente empregado para definir outro label, pelo que não se deveria ser reutilizado. O montador tomará a ultima declaração do label como a válida.

- *Nome*. Identificador do label que já foi empregado anteriormente.
- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de advertência:

```
.org 100H

salto : mvi b, 5
salto : mov a,b
      add b
            jmp salto
```

Como vemos, existe uma ambigüidade sobre de onde se deve saltar. Por "default" o montador optará pela última vez em que foi declarado o label de salto.

	Advertência:	Declaração previa da definição: '*<Nome>*'. Linha *<Nº linha>*

O nome que se quer aplicar na definição foi previamente empregado em outra definição, pelo que não deveria ser reutilizado. O montador assumirá a ultima definição como a válida.

- *Nome*. Identificador da definição que já tenha sido empregado anteriormente.
- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de advertência:

```
.define
  maximo 100
  minimo 0
  media 50
  maximo 120

.org 100H

salto : mvi b, 5
        add b
              jmp salto
```

O nome *Maximo é* utilizado duas vezes para realizar uma definição. Desta forma o valor de *Maximo* pode ser 100 ou 120. Por "default" será 120.

	Advertência:	A diretiva define não necessita endereço origem. Linha *<Nº linha>*

Somente as diretivas .DATA e .ORG requerem um endereço de origem a partir do qual possam situar dados ou instruções respectivamente. As definições somente são empregadas em tempo de compilação, realizando-se as oportunas substituições. Se for especificado um endereço, este é reconhecido, porém não será empregado em nenhum momento.

- *Nº de linha*. Linha na qual será encontrado o erro.

Exemplo de advertência:

```
.define A0h
  maximo 100
  minimo 0
  media 50

.org 100H

salto : mvi b, 5
        add b
              jmp salto
```

O nome *Maximo é* utilizado duas vezes para realizar uma definição. Desta forma o valor de *Maximo* pode ser 100 ou 120. Por default será 120.

	Advertência:	O endereço '*<endereço>*' não pode ser armazenado em 16 bits. Linha *<Nº linha>*

O processador 8085 somente pode trabalhar com endereços de 16 bits. Qualquer outro endereço maior de 65535 (ou FFFFh) não é endereçável.

Se um diretiva .DATA ou .ORG recebem um endereço origem maior que FFFFh será gerada esta advertência. Somente os 16 bits menos significativos do endereço será armazenado.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de erro:

```
.org 1FFFFH

            mvi b, 5
salto : mov a,b
      add b
            jmp salto
```

	Advertência:	O valor '*<Valor>*' não pode ser armazenado em um byte. Linha *<Nº linha>*

Ao realizar uma declaração de dados em forma de bytes estes devem de ser representáveis em 8 bits. Se não ocorrer isto será lançada esta advertência.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de erro:

.data 0H
  dB 256, 100h, 100000000b
 	
.org 100H
 	
  mvi b, 5
salto : mov a,b
  add b
  jmp salto

	Advertência:	O valor '*<Valor>*' não pode ser armazenado em uma word. Linha *<Nº linha>*

Como no caso anterior, com as palavras de 16 bits. Se o valor declarado não puder ser representado em 16 bits será lançada esta advertência.

- *Nº de linha*. Linha na qual será encontrado o erro.

Exemplo de erro:

.data 0H
  dW 65536, 10000h, 10000000000000000b
 	
.org 100H
 	
  mvi b, 5
salto : mov a,b
  add b
  jmp salto

	Advertências:	O primeiro operando (*<Op>*) não pode ser representado em 8

		bits. Linha *<Nº linha>*
		O segundo operando (*<Op>*) não pode ser representado em 8 bits. Linha *<Nº linha>*

Certas instruções permitem especificar um operando imediato de 8 bits. Estas advertências aparecem quando o operando imediato não pode ser representado em 8 bits.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de erro:

```
.org 100H

        mvi b, 100h
salto : mov a,b
    add b
        jmp salto
```

	Advertência:	O primeiro operando (*<Op>*) não pode ser representado em 16 bits. Linha *<Nº linha>*

Certas instruções permitem especificar um operando imediato de 16 bits. Esta advertência aparece quando o operando imediato não pode ser representado em 16 bits.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de erro:

```
.org 100H

        mvi b, FFh
salto : mov a,b
    add b
        jmp 10000h
```

	Advertência:	O segundo operando (*<Op>*) não pode representar os 16 bits. Linha *<Nº linha>*

Certas instruções permitem especificar um operando imediato de 16 bits. Esta advertência aparece quando o operando imediato não pode ser representado em 16 bits.

- *Nº de linha*. Linha na qual se encontra o erro.

Exemplo de erro:

```
.org 100H

        mvi b, FFh
salto : mov a,b
    add b
        LXI H, 10000h
```