

UNIVERSIDADE DO MINHO

Escola de Engenharia
Departamento de Informática



Versão 1.2a

Manual da Linguagem

José Carlos Rufino Amaro
Jorge Alexandre Santos

Orientação:
Engº Fernando Mário Martins

Braga
1994

1 INTRODUÇÃO	2
2 MÁQUINA VIRTUAL DE STACK.....	3
2.1 Arquitectura	3
2.2 Princípio de Funcionamento.....	6
3 LINGUAGEM MSP	7
3.1 Gramática do MSP	7
3.2 Números negativos	9
3.3 Sintaxe e Semântica do MSP	11
3.3.1 Zona de Dados - Declaração de variáveis.....	11
3.3.2 Zona de Código - Conjunto das Instruções.....	12
3.4 Erros e Avisos de montagem	18
3.4.1 Erros.....	18
Erro Interno.....	18
Erros da Zona de Dados.....	18
Erros comuns à Zona de Dados e de Código.....	21
Erros da Zona de Código	21
Erros na declaração de <i>labels</i>	22
Erro em instruções sem argumentos.....	22
Erro específico da instrução PUSH.....	22
Erros específicos da instrução PSHA.....	23
Erros específicos de instruções de salto (JMP, JMPF, CALL)	24
Erro Externo.....	24
3.4.2 Avisos.....	25
Avisos da Zona de Dados.....	25
Avisos da Zona de Código	25
ANEXO A - TABELA DE CONVERSÃO DE E PARA C2.....	27

1 Introdução

O objectivo deste manual é descrever a linguagem *assembly* de programação MSP usada pelo WinMSP 1.2¹.

MSP são as iniciais de *Mais Simples Possível*, e com efeito, trata-se de uma linguagem bastante simples, com um reduzido número de instruções, de sintaxe e semântica muito acessíveis.

A linguagem MSP destina-se a programar uma Máquina de *Stack* Virtual. Como veremos, é recorrendo à sua *Stack* que dita máquina efectua a maior parte das suas operações. A designação de Virtual advém do facto de que tal máquina não corresponde a nenhuma máquina com existência real, sendo da responsabilidade de um programa fornecer a ilusão de que ela existe.

O WinMSP, correndo em ambiente Windows 3.1, tira partido da sua *interface* amigável e intuitiva, tornando a programação em MSP um processo mais fácil e integrado, quando comparado com versões mais antigas para DOS.

Para além da *interface* gráfica, o WinMSP traz consigo outras novidades, das quais se destacam:

- a possibilidade de usar números negativos;
- a protecção mais robusta dos acessos à Memória de Dados e à *Stack*;
- AND, OR e NOT designam agora operações lógicas sobre a totalidade de um *byte*, ao passo que antigamente eram operações *bitwise* (*bit-a-bit*); as operações *bitwise* são agora ANDB, ORB e NOTB.

A forma como o resto do Manual se encontra organizado, reflecte, aproximadamente, a seguinte estratégia para programar em *assembly* :

1. compreender a arquitectura e o funcionamento básico da máquina com que se vai trabalhar;
2. estudar o seu conjunto de instruções;
3. definir esquemas para traduzir em *assembly* as construções de uma linguagem algorítmica;
4. escrever o algoritmo do problema a resolver, numa linguagem algorítmica;
5. aplicar os esquemas de tradução definidos em (3°);
6. rever o código resultante da junção dos blocos sucessivamente obtidos por aplicação de (5°), estudando possíveis optimizações.

¹O ambiente WinMSP 1.2 encontra-se devidamente documentado no respectivo Manual do Utilizador.

2 Máquina Virtual de Stack

2.1 Arquitectura

Numa Máquina de *Stack*, podemos identificar os seguintes blocos fundamentais:

- **Memória**, para armazenar instruções e valores, dividida em 3 blocos independentes e de funcionalidade distinta:
 - Memória de Programa (ou abreviadamente MProg);
 - Memória de Dados (ou MDados);
 - Memória de Trabalho (*Stack* ou Pilha).
- **Descodificador**, para interpretar e mandar executar cada instrução;
- **Unidade Lógica e Aritmética**, onde se efectuam as operações lógicas e aritméticas básicas; os operandos são aí carregados automaticamente cada vez que se ordena a realização de uma dessas operações, e é lá que se obtém o resultado, o qual é de imediato, e também automaticamente, movido para o seu destino;
- **Input e Output**, duas posições de memória especiais, com endereços fixos e bem conhecidos (input e output) e que têm a característica especial de estarem directamente ligadas ao exterior, permitindo a entrada de valores do exterior para a máquina (i.e., para input) e a saída de valores da máquina (i.e., de output) para o exterior; concretamente, essas células de memória especiais estão ligadas, respectivamente, ao teclado e ao monitor;
- **Bus Interno**, um "corredor" interligando as diversas unidades, permitindo a circulação e comunicação de instruções e valores;
- **Program Counter (PC) (ou Instruction Pointer, IP) e Stack Pointer (SP)**, dois registos fundamentais (são aliás os únicos registos da Máquina de *Stack*!), contendo respectivamente o endereço de MProg correspondente à próxima instrução a ser executada e o endereço correspondente à primeira posição livre na *Stack* (ou seja., uma posição à frente do actual valor do topo da *Stack*);

A Figura 1 apresenta a arquitectura da Máquina de *Stack* em termos dos blocos funcionais descritos.

(Esquema do Eng. Mário Martins)

Cada célula de Memória (MProg, MDados ou *Stack*) tem a capacidade de um *byte* (sequência de 8 *bits*) e está associada a um endereço que a identifica univocamente.

Um endereço é um número inteiro começando em 0 (zero) e seguindo sequencialmente até à última célula existente. O número total de células é sempre finito e bem definido, sendo no caso presente de 32000. Portanto, a gama de endereços possíveis que permitem aceder a uma célula de Memória pode ser representada pelo intervalo [0, 31999] (equivalentemente, podemos dizer que o espaço de endereçamento na Máquina de *Stack* Virtual é limitado ao intervalo [0, 31999]).

O *byte* armazenado em cada célula de Memória pode representar uma instrução da máquina, um argumento de instrução, um valor inteiro, um carácter ou um componente de um endereço.

A interpretação do significado de cada *byte* é da responsabilidade da máquina e baseia-se no tipo de Memória à qual se está a aceder num determinado instante:

- a **Memória de Programa (MProg)** armazena instruções que vão determinar a acção da máquina, i.e., vão impor aquilo que a máquina deve executar. Um programa é a sequência de instruções presentemente armazenadas na MProg. Uma célula da MProg poderá conter o código de uma instrução ou um argumento de uma instrução. A maioria das instruções da linguagem MSP não contém argumentos. Os argumentos, a existir, localizar-se-ão imediatamente a seguir ao código da instrução.
- a **Memória de Dados (MDados)** armazena valores. Esses valores correspondem aos dados do problema ou aos resultados calculados pelo programa. Cada valor ocupa um *byte*, podendo representar grandezas numéricas no intervalo [-128,255], ou caracteres (o seu código inteiro definido de acordo com a Tabela ASCII), ou ainda endereços de posições da MDados, sendo que neste último caso se estendem por dois *bytes*. Sem entrar ainda em detalhes, diremos que a "possibilidade" de usar inteiros de 8 *bits* (que permitem representar $2^8 = 256$ valores diferentes) para cobrir o intervalo [-128,255] com $128 + 256 = 384$ valores, reside na interpretação contextual do conteúdo de cada *byte*: se em dado momento da execução do programa for necessário interpretar um *byte* como componente de um endereço, o seu valor é interpretado no intervalo [0,255], ao passo que se for oportuno interpretá-lo como valor de 8 *bits* com sinal, será convertido ao intervalo [-128,127].
- a ***Stack*** é uma memória de trabalho semelhante no conteúdo à MDados. Contudo, é acedida e usada segundo uma filosofia muito própria: os valores que se vão introduzindo na *Stack* ficam empilhados uns por cima dos outros, de tal forma que apenas se pode aceder (ler) ao valor que está no cimo da "pilha" (i.e., da *Stack*), e que corresponde ao último valor introduzido (escrito). Pelo facto de o último valor guardado ter de ser o primeiro a sair (caso se queira aceder aos outros), esta política de acesso designa-se correntemente de LIFO - *Last In First Out*. Consequentemente, apenas é necessário conhecer um único endereço: o da última posição, ou topo da *Stack*. Considera-se que o topo é o endereço da *Stack* relativo ao último valor armazenado. Assim, o próximo valor a guardar na

Stack será armazenado no endereço a seguir ao topo, e o primeiro valor a sair será sempre o do topo. A Máquina de *Stack* foi desenhada de tal forma que todas as operações esperam ter os seus operandos no topo da *Stack*, retiram-nos de lá e, em substituição, colocam no topo o resultado obtido. É pois evidente que o topo da *Stack* é dinâmico por natureza, crescendo e/ou decrescendo em conformidade com os acessos à *Stack* impostos pela execução do programa. Na Máquina de *Stack*, o topo "cresce do fim para o princípio", ou seja, do último endereço (31999) em direcção ao primeiro (0).

2.2 Princípio de Funcionamento

Para iniciar a execução de um programa é necessário que o PC contenha o endereço de MProg onde está a primeira instrução. A partir daí, e uma vez dada a "ordem" à Máquina de *Stack* para começar, o seu funcionamento é muito simples e sistemático, baseando-se num ciclo correntemente designado por *fetch-decode-execute*, e cuja essência de seguida se apresenta.

A instrução "apontada" pelo PC é carregada para o Descodificador, o qual interpreta o seu significado e emite para as outras unidades as ordens necessárias para que a acção (correspondente ao significado da instrução) seja realizada, sendo de imediato e automaticamente incrementado o PC, ou seja, avançado de modo a que fique a apontar para a próxima instrução a executar.

Terminada a realização dessa instrução, o ciclo repete-se, i.e., nova instrução é trazida para o Descodificador e o PC é devidamente incrementado, continuando desta forma a execução, até que seja decodificada uma instrução cujo significado seja, justamente, parar a execução.

As ordens enviadas pelo Descodificador às outras unidades, correspondem a um conjunto muito limitado e elementar de acções, mas que, na realidade, se combinadas de forma adequada, são suficientes para resolver qualquer problema com algoritmo.

Entre outras, essas acções podem ser:

- transferência de um dado entre o topo da *Stack* e uma das células especiais input ou output;
- realização de uma operação aritmética (adição, subtracção, multiplicação ou divisão) ou lógica (conjunção, disjunção ou negação);
- alteração da sequência normal de execução das instruções armazenadas em MProg, forçando o PC a assumir um valor diferente daquele que automaticamente toma após executar uma instrução (recordar o processo normal de execução descrito anteriormente).

O algoritmo da Figura 2, descreve, ainda que superficialmente, o funcionamento acabado de expor.

```

    ExecutarProg = ObterProxInstrução;
    enquanto ( nao Fim ) fazer
        InterpretarInstrução;
        ObterProxInstrução;
    fim enquanto

    InterpretarInstrução = se (CódigoInst = C1) então Acção1;
                        se (CódigoInst = C2) então Acção2;
                        ...
                        se (CódigoInst = Cn) então Acçãon;

    Fim = (CódigoInst = Cx);

```

Figura 2 - Algoritmo básico da Máquina Virtual de *Stack*

Nota: C1, C2, Cn e Cx denotam alguns dos códigos das instruções concretas da máquina (a apresentar no capítulo seguinte). Cx é um código especial cuja ocorrência resulta na paragem da execução do programa.

3 Linguagem MSP

3.1 Gramática do MSP

A linguagem MSP com base na qual se vão escrever os programas *assembly* para a Máquina de *Stack* Virtual, é definida pela gramática da Figura 3, com recurso à notação BNF (*Backus-Naur-Form*).

Antes porém, convém recordar alguma da simbologia desta notação:

- 'xyz': xyz é um valor bem definido na linguagem;
- <xyz>: sequência de elementos iguais a ou do tipo de xyz;
- [xyz]: xyz ocorre no máximo uma vez;
- (xyz)*: xyz ocorre zero ou mais vezes;
- (xyz)+: xyz ocorre uma ou mais vezes;

Ainda em relação à Figura 3, '\t', '\r', '\n' e '\0' correspondem a representações dos caracteres *tab*, *form-feed*, *new-line* e *null*, respectivamente.


```

<prog_ass>      ::= (<neutro>)* <declarações> (<neutro>)* <instruções>
                  (<neutro>)* <fim_de_texto>

<neutro>        ::= <comentário> | <vazio>

<comentário>    ::= (<espaço>)* ';' (<caracter>)* fim_de_linha

<espaço>        ::= ' ' | '\t' | '\r'

<caracter>      ::= qualquer_caracter_da_tabela_ASCII_menos_fim_de_linha

<fim_de_linha>  ::= '\n'

<vazio>         ::= (<espaço>)* fim_de_linha

<declarações>   ::= 'MEMORIA DE DADOS' (<neutro>)+ (<declaração>)*

<declaração>    ::= (<espaço>)* <id_var> (<espaço>)+ <endereço>
                  (<espaço>)+ ('TAM' | 'TAMA' | 'TAMAN' | 'TAMANH' |
                  'TAMANHO' ) (<espaço>)+ <tamanho>
                  [ (<espaço>)+ ('VAL' | 'VALO' | 'VALOR' | 'VALORE'
                  | 'VALORES') (<espaço>)+ ( (<valor> |
                  <valor_c/_sinal>) (<espaço>)*)+ ] (<neutro>)+

<id_var>        ::= (<alfanumerico>)+

<alfanumerico>  ::= <digito> | <letra>

<digito>        ::= '0' | '1' | ... | '9'

<letra>         ::= 'a' | ... | 'z' | 'A' | ... | 'Z' | '_'

<endereço>      ::= <int_16>

<int_16>        ::= numero_inteiro_de_16_bits_no_intervalo_[0,31999]

<tamanho>       ::= <int_16>

<valor>         ::= <int_8>

<int_8>         ::= numero_inteiro_de_8_bits_no_intervalo_[0,255]

<valor_c/_sinal> ::= <sint_8>

<sint_8>        ::= numero_inteiro_de_8_bits_no_intervalo_[-128,127]

<instruções>    ::= 'CODIGO' (<neutro>)+ ( [ (<espaço>)* <id_label>
                  ':' ] (<neutro>)* <instrução> ) *

<id_label>      ::= (<alfanumerico>)+

<instrução>     ::= (<espaço>)* <mnemonica> [ (<espaço>)+ <argumento>
                  ] (<neutro>)+

<mnemonica>     ::= 'PUSH' | 'PSHA' | ... | 'NOOP'

<argumento>     ::= <valor> | <valor_c/_sinal> | <id_var> | <id_label>
                  | <endereço> | <endr_relativo>

<endr_relativo> ::= <sint_16>

<sint_16>       ::= numero_inteiro_de_16_bits_no_intervalo_[-31999,31999]

<fim_de_texto>  ::= '\0'

```

Figura 3 - Gramática BNF do MSP.

Relativamente à gramática proposta para a linguagem *assembly* MSP, convém chamar a atenção para alguns pormenores:

- em cada linha da Zona de Dados (limitada por MEMORIA DE DADOS e por CODIGO) e da Zona de Código (limitada por CODIGO e estendendo-se até ao fim do texto do programa) não é permitida mais que uma declaração de variável e de *label*, respectivamente; também não é permitida mais de uma instrução em cada linha da Zona de Código;
- são permitidas linhas vazias ou de comentário em qualquer parte da Zona de Dados e de Código;
- identificadores de variáveis ou *labels* devem conter pelo menos uma <letra> não podendo ser sequências puras de dígitos, *underscores* (_), ou iguais à palavra reservada CODIGO;
- variáveis e *labels* podem ter identificadores iguais; além disso, tais identificadores, bem como palavras reservadas da linguagem (MEMORIA DE DADOS, TAMANHO, VALORES, CODIGO e mnemónicas) não são *case-sensitive*, ou seja, permitem qualquer combinação de maiúsculas e minúsculas para designá-los (por exemplo, "variavel1" é igual a "VARIaVEL1", "PSHA" é igual a "PSha", etc);
- é obrigatória a presença dos delimitadores das Zonas de Dados e de Código, MEMORIA DE DADOS e CODIGO, respectivamente, ainda que essas Zonas estejam vazias (o que resultaria num programa sem variáveis e/ou sem código);
- um comentário começa por um ';' e permite a ocorrência de quaisquer caracteres, à excepção de fim_de_linha, que termina o comentário;
- as palavras reservadas TAMANHO e VALORES podem ser escritas abreviadamente, até um valor "mínimo" de TAM e VAL, respectivamente;
- em números positivos, o sinal '+' é opcional; nos números negativos, o sinal '-' é obrigatório;

3.2 Números negativos

É sabido que com 8 *bits* apenas é possível representar $2^8 = 256$ valores diferentes.

Se não reservarmos nenhum *bit* para o sinal e assumirmos por defeito que todos os valores são positivos, então teremos a possibilidade de representar qualquer número na gama [0, 255].

Se reservarmos um *bit* para o sinal, então sobram 7 *bits* e estaremos condicionados a representar valores na gama $[-2^7, 2^7-1]$, ou seja [-128,127].

Existem várias formas de implementar o intervalo $[-128,127]$ com 8 *bits*, entre as quais o complemento para 2 (abreviadamente C2). Em C2 reserva-se o *bit* 7 (o mais significativo) para codificar o sinal e os restantes 7 *bits* (do 0 ao 6) são usados para representar o módulo do valor.

Não cabe aqui discutir o mecanismo de conversão de números de e para o formato C2. O que é aqui relevante é que é esse o tipo de representação que o WinMSP usa para codificar números negativos de 8 e 16 *bits*.

A Tabela 1 mostra como um dado número binário de 8 *bits* pode ser interpretado de duas formas diferentes, consoante se assuma que a sequência de 8 *bits* é uma representação sem sinal ou com sinal (em C2):

Binário de 8 <i>bits</i>	Inteiro em $[0,255]$	Inteiro em $[-128,127]$ (em C2)
1 1111111	255	-1
1 1111110	254	-2
...
1 0000001	129	-127
1 0000000	128	-128
0 1111111	127	127
0 1111110	126	126
...
0 0000001	1	1
0 0000000	0	0

Tabela 1 - Inteiros de 8 *bits* sem e com sinal.

Portanto, nada nos diz à partida, se o conteúdo de um *byte* é um número com ou sem sinal. "Um *byte* será aquilo que nós quisermos"! São as circunstâncias que ditam a forma como é interpretado o conteúdo de um *byte*, i.e., a sua interpretação é contextual e de certa forma independente da maneira como ele foi inicializado.

Um pequeno exemplo ajudará a clarificar a questão: se na Zona de Dados inicializarmos uma variável com o valor -1 e em dada altura da execução do programa usarmos o conteúdo dessa variável como parte de um endereço, então o valor binário 11111111 correspondente a -1, é na realidade interpretado como 255 (ver equivalência na Tabela 1); analogamente, se a tivéssemos inicializado com o valor 255 e mais tarde a usássemos como argumento de uma operação aritmética, o seu conteúdo seria assumido como sendo -1!

É pois evidente que a interpretação contextual está relacionada com o tipo de argumentos e respectiva gama de valores das diversas instruções do MSP. Será pois na discussão da semântica do Conjunto de Instruções que se encerrará a discussão sobre os números negativos.

No Anexo A é fornecida uma versão completa da Tabela 1.

3.3 Sintaxe e Semântica do MSP

3.3.1 Zona de Dados - Declaração de variáveis

Como é sabido, a Zona de Dados é iniciada pelas palavras reservadas MEMORIA DE DADOS e prolonga-se até ao início da Zona de Código, iniciada pela palavra reservada CODIGO.

É na Zona de Dados que se declaram e eventualmente se inicializam as diversas variáveis necessárias ao programa.

Genericamente, e recordando a sintaxe estabelecida pela Gramática do MSP, uma declaração é algo da forma:

```
<id_var> <endereço> TAM <tamanho> VAL <valor1> ... <valorN>
```

Por exemplo,

```
x 100 TAM 20 VAL 4
```

é a declaração da variável "x", no endereço 100, com tamanho 20 e com o 1º *byte* inicializado a 4, sendo os *bytes* 101 a 119 inicializados a zero, por defeito.

A semântica das declarações compreende, assim, alguns aspectos importantes:

- cada identificador de variável é único, não podendo haver mais que uma variável com determinado identificador;
- não é obrigatório declarar as variáveis contiguamente, a partir do endereço 0 (zero), podendo-se deixar "buracos" na Memória de Dados;
- o espaço reservado para uma variável não pode colidir com o reservado para outra, i.e., os intervalos [*<endereço>*, *<endereço>* + *<tamanho>* - 1] hão-de ser mutuamente exclusivos para todas as variáveis declaradas;
- a inicialização do espaço alocado a uma variável é facultativa, sendo garantida, por defeito, a inicialização desse espaço ao valor 0 (zero).
- a inicialização das variáveis pode ser feita com qualquer valor inteiro no intervalo [-128,255], com sinal opcional para os positivos; como já foi referido, esses valores estarão sujeitos a interpretação contextual durante a execução do programa;
- é perfeitamente lícito não declarar nenhuma variável na Zona de Dados; qualquer acesso que se venha a fazer à Memória de Dados há-de porém levar em conta que ela se encontra totalmente inicializada a zero (como veremos na discussão do Conjunto de Instruções, é possível ler de e escrever na Memória de Dados sem quaisquer variáveis declaradas...).

3.3.2 Zona de Código - Conjunto das Instruções

A Zona de Código é inicializada com a palavra reservada CODIGO e prolonga-se até ao final do texto do programa (o fim corresponde à ocorrência do carácter *null*, de código ASCII 0 (zero), representado na gramática por *fim_de_texto*).

Para cada instrução, é apresentada a mnemónica, o seu código em decimal, os seus argumentos (caso existam) e gama de valores (representados usando a notação BNF da gramática do MSP), e o seu significado (descrição informal, seguida de uma formal) em termos das operações sobre a *Stack* e a Memória de Dados.

Admite-se que todas as instruções sem argumentos incrementam o PC de uma unidade, imediatamente após executarem as acções respectivas, omitindo-se tal operação na sua descrição (casos em que as instruções têm argumentos serão analisados em particular).

Recorde-se ainda que a *Stack* cresce do fim do seu espaço de endereçamento (endereço 31999) para o princípio (endereço 0) e que portanto o SP diminui sempre que carregamos algo na *Stack* e aumenta sempre que extraímos algo do topo da *Stack*. O SP aponta sempre para a 1ª posição livre, imediatamente a seguir ao topo (última posição ocupada) da *Stack*. Consequentemente, qualquer leitura da *Stack* incide sobre o seu topo actual e qualquer escrita na *Stack* gera um novo topo com o valor introduzido, imediatamente a seguir ao antigo topo.

O Conjunto das Instruções, pode-se dividir em 3 grupos, cada um implementando determinada funcionalidade: manuseamento de valores e endereços, operações aritméticas e lógicas ou controlo da sequência de execução do programa.

→ Manuseamento de valores e endereços:

Mnemónica	Código	Argumento	Significado
PUSH	1	<valor> ou <valor_c/_sinal>	escreve um <int_8> ou <sint_8> na <i>Stack</i> : $Stack[SP] = \text{<sint_8>}$ ou $Stack[SP] = \text{<sint_8>}$; $SP = SP - 1$; $PC = PC + 2$;
PSHA	2	<endereço>ou <idvar>	coloca um <int_16> na <i>Stack</i> ; se o argumento não for um <endereço>, converte-se <idvar> no seu <endereço> equivalente; o <endereço> (fornecido ou resolvido) é separado em dois <int_8> (Lsb e Msb, tal que o valor de <endereço> é um <int_16> igual a $Msb * 256 + Lsb$) que são escritos na <i>Stack</i> (primeiro o Lsb, depois o Msb): $Stack[SP] = Lsb$; $SP = SP - 1$; $Stack[SP] = Msb$; $SP = SP - 1$; $PC = PC + 3$;

Mnemónica	Código	Significado
LOAD	3	<p>constrói um Endereço da Memória de Dados com dois <int_8> que lê da <i>Stack</i> e escreve na <i>Stack</i> o conteúdo da Memória de Dados nesse Endereço:</p> <p>$SP = SP + 1$; $Msb = Stack[SP]$; $SP = SP + 1$; $Lsb = Stack[SP]$; $Endereço = Msb * 256 + Lsb$; $Stack[SP] = MemóriaDados[Endereço]$; $SP = SP - 1$;</p>
LDA	4	<p>constrói um Endereço da Memória de Dados com dois <int_8> que lê da <i>Stack</i>; os conteúdos desse Endereço e do próximo (ambos relativos à Memória de Dados) tornam-se as componentes equivalentes a Lsb e Msb, respectivamente, de um outro endereço, que são então escritas na <i>Stack</i> (primeiro o Lsb, depois o Msb):</p> <p>$SP = SP + 1$; $Msb = Stack[SP]$; $SP = SP + 1$; $Lsb = Stack[SP]$; $Endereço = Msb * 256 + Lsb$; $Lsb = MemóriaDados[Endereço]$; $Stack[SP] = Lsb$; $SP = SP - 1$; $Msb = MemóriaDados[Endereço+1]$; $Stack[SP] = Msb$; $SP = SP - 1$;</p>
STORE	5	<p>lê um Valor do tipo <int_8> da <i>Stack</i> e constrói um Endereço da Memória de Dados com mais dois <int_8> que também lê da <i>Stack</i>; o primeiro <int_8> lido é escrito na Memória de Dados, no Endereço construído:</p> <p>$SP = SP + 1$; $Valor = Stack[SP]$; $SP = SP + 1$; $Msb = Stack[SP]$; $SP = SP + 1$; $Lsb = Stack[SP]$; $Endereço = Msb * 256 + Lsb$; $MemóriaDados[Endereço] = Valor$;</p>
STRA	6	<p>lê dois <int_8> (possivelmente componentes Lsb e Msb de um endereço da Memória de Dados) da <i>Stack</i>; de seguida lê mais dois <int_8> e com eles constrói um Endereço da Memória de Dados; os primeiros dois <int_8> lidos são então escritos na Memória de Dados, nas posições dadas por Endereço e Endereço +1:</p> <p>$SP = SP + 1$; $Msb2 = Stack[SP]$; $SP = SP + 1$; $Lsb2 = Stack[SP]$; $SP = SP + 1$; $Msb = Stack[SP]$; $SP = SP + 1$; $Lsb = Stack[SP]$; $Endereço = Msb * 256 + Lsb$; $MemóriaDados[Endereço] = Lsb2$; $MemóriaDados[Endereço+1] = Msb2$;</p>
IN	7	<p>escreve na <i>Stack</i> o conteúdo da posição especial <u>input</u>, resultante da leitura de um <valor_c/_sinal> através do teclado:</p> <p>$Stack[SP] = \underline{input}$; $SP = SP - 1$;</p>
OUT	8	<p>escreve no ecran (ou melhor, na posição especial <u>output</u>) o <valor_c/_sinal> resultante da leitura do topo da <i>Stack</i>:</p> <p>$SP = SP + 1$; $\underline{output} = Stack[SP]$;</p>

Mnemónica	Código	Significado
INC	28	escreve na <i>Stack</i> o conteúdo (código ASCII, do tipo <valor>) da posição especial <u>input</u> , resultante da leitura de um <caracter> através do teclado: $Stack[SP] = \text{input}; SP = SP - 1;$
OUTC	29	escreve no écran (ou melhor, na posição especial <u>output</u>) o <caracter> cujo código ASCII resulta da leitura de um <valor> do topo da <i>Stack</i> : $SP = SP + 1; \text{output} = Stack[SP];$

Observação: as instruções PUSH, LOAD e STORE operam sobre valores de um *byte* enquanto que PSHA, LDA e STRA operam sobre endereços de 2 *bytes*.

→ **Operações aritméticas e lógicas:**

• **Operações aritméticas:**

Mnemónica	Código	Significado
ADD	9	lê dois <int_8> da <i>Stack</i> e escreve a sua soma na <i>Stack</i> : $SP = SP + 1; \text{Operando2} = Stack[SP];$ $SP = SP + 1; \text{Operando1} = Stack[SP];$ $Stack[SP] = \text{Operando1} + \text{Operando2}; SP = SP - 1;$
SUB	10	lê dois <int_8> da <i>Stack</i> e escreve a sua diferença na <i>Stack</i> : $SP = SP + 1; \text{Operando2} = Stack[SP];$ $SP = SP + 1; \text{Operando1} = Stack[SP];$ $Stack[SP] = \text{Operando1} - \text{Operando2}; SP = SP - 1;$
MUL	11	lê dois <int_8> da <i>Stack</i> e escreve o seu produto na <i>Stack</i> : $SP = SP + 1; \text{Operando2} = Stack[SP];$ $SP = SP + 1; \text{Operando1} = Stack[SP];$ $Stack[SP] = \text{Operando1} * \text{Operando2}; SP = SP - 1;$
DIV	12	lê dois <int_8> da <i>Stack</i> e escreve o quociente resultante da sua divisão inteira, na <i>Stack</i> : $SP = SP + 1; \text{Operando2} = Stack[SP];$ $SP = SP + 1; \text{Operando1} = Stack[SP];$ $Stack[SP] = \text{Operando1} / \text{Operando2}; SP = SP - 1;$
ADDA	13	lê um Offset do tipo <int_8> da <i>Stack</i> e constrói um Endereço com dois <int_8> que também lê da <i>Stack</i> ; o Offset é somado ao Endereço, e o Endereço resultante é decomposto e escrito na <i>Stack</i> : $SP = SP + 1; \text{Offset} = Stack[SP];$ $SP = SP + 1; \text{Msb} = Stack[SP];$ $SP = SP + 1; \text{Lsb} = Stack[SP];$ $\text{Endereço} = \text{Msb} * 256 + \text{Lsb} + \text{Offset};$ $\text{Msb} = \text{quociente}(\text{Endereço} / 256);$ $\text{Lsb} = \text{resto}(\text{Endereço} / 256);$ $Stack[SP] = \text{Lsb}; SP = SP - 1;$ $Stack[SP] = \text{Msb}; SP = SP - 1;$

• **Operações lógicas:**

Mnemónica	Código	Significado
AND	14	lê dois <int_8> da <i>Stack</i> e escreve a sua <u>conjunção lógica</u> na <i>Stack</i> : $SP = SP + 1$; $Operando2 = Stack[SP]$; $SP = SP + 1$; $Operando1 = Stack[SP]$; $Stack[SP] = Operando1 \wedge Operando2$; $SP = SP - 1$;
OR	15	lê dois <int_8> da <i>Stack</i> e escreve a sua <u>disjunção lógica (inclusiva)</u> na <i>Stack</i> : $SP = SP + 1$; $Operando2 = Stack[SP]$; $SP = SP + 1$; $Operando1 = Stack[SP]$; $Stack[SP] = Operando1 \vee Operando2$; $SP = SP - 1$;
NOT	16	lê um <int_8> da <i>Stack</i> e escreve a sua <u>negação lógica</u> na <i>Stack</i> : $SP = SP + 1$; $Operando = Stack[SP]$; $Stack[SP] = \text{não}(Operando)$; $SP = SP - 1$;
EQ	17	lê dois <int_8> da <i>Stack</i> , verifica se são iguais e escreve o resultado dessa comparação na <i>Stack</i> : $SP = SP + 1$; $Operando2 = Stack[SP]$; $SP = SP + 1$; $Operando1 = Stack[SP]$; $Stack[SP] = (Operando1 == Operando2)$; $SP = SP - 1$;
NE	18	lê dois <int_8> da <i>Stack</i> , verifica se são diferentes e escreve o resultado dessa comparação na <i>Stack</i> : $SP = SP + 1$; $Operando2 = Stack[SP]$; $SP = SP + 1$; $Operando1 = Stack[SP]$; $Stack[SP] = (Operando1 \neq Operando2)$; $SP = SP - 1$;
LT	19	lê dois <int_8> da <i>Stack</i> , verifica se um é menor que o outro e escreve o resultado dessa comparação na <i>Stack</i> : $SP = SP + 1$; $Operando2 = Stack[SP]$; $SP = SP + 1$; $Operando1 = Stack[SP]$; $Stack[SP] = (Operando1 < Operando2)$; $SP = SP - 1$;
LE	20	lê dois <int_8> da <i>Stack</i> , verifica se um é menor ou igual que o outro e escreve o resultado dessa comparação na <i>Stack</i> : $SP = SP + 1$; $Operando2 = Stack[SP]$; $SP = SP + 1$; $Operando1 = Stack[SP]$; $Stack[SP] = (Operando1 \leq Operando2)$; $SP = SP - 1$;
GT	21	lê dois <int_8> da <i>Stack</i> , verifica se um é maior que o outro e escreve o resultado dessa comparação na <i>Stack</i> : $SP = SP + 1$; $Operando2 = Stack[SP]$; $SP = SP + 1$; $Operando1 = Stack[SP]$; $Stack[SP] = (Operando1 > Operando2)$; $SP = SP - 1$;
GE	22	lê dois <int_8> da <i>Stack</i> , verifica se um é maior ou igual que o outro e escreve o resultado dessa comparação na <i>Stack</i> : $SP = SP + 1$; $Operando2 = Stack[SP]$; $SP = SP + 1$; $Operando1 = Stack[SP]$; $Stack[SP] = (Operando1 \geq Operando2)$; $SP = SP - 1$;

• **Operações bitwise:**

Mnemónica	Código	Significado
ANDB	30	lê dois <int_8> da <i>Stack</i> e escreve a sua <u>conjunção bit-a-bit</u> na <i>Stack</i> : $SP = SP + 1$; Operando2 = <i>Stack</i> [SP]; $SP = SP + 1$; Operando1 = <i>Stack</i> [SP]; $Stack[SP] = Operando1 \& Operando2$; $SP = SP - 1$;
ORB	31	lê dois <int_8> da <i>Stack</i> e escreve a sua <u>disjunção bit-a-bit</u> na <i>Stack</i> : $SP = SP + 1$; Operando2 = <i>Stack</i> [SP]; $SP = SP + 1$; Operando1 = <i>Stack</i> [SP]; $Stack[SP] = Operando1 Operando2$; $SP = SP - 1$;
NOTB	32	lê um <int_8> da <i>Stack</i> e escreve a sua <u>negação bit-a-bit</u> (i.e, o complementar ou complemento para 1, C1) na <i>Stack</i> : $SP = SP + 1$; Operando = <i>Stack</i> [SP]; $Stack[SP] = C1(Operando)$; $SP = SP - 1$;

Observações:

- à excepção do ADDA, cujo resultado é um <endereço>, o resultado das restantes operações aritméticas é um <sint_8> e o 2º operando é retirado da *Stack* antes do 1º operando;
- nas operações lógicas o resultado é 0 (FALSO) ou 1 (VERDADEIRO) e à excepção do operador unário NOT, também o 2º operando é retirado da *Stack* antes do 1º operando; as operações bitwise (ou bit-a-bit) são um caso particular das lógicas, em que a operação lógica se efectua entre os *bits* correspondentes de cada operando (*bit* 0 com *bit* 0, ..., *bit* 7 com *bit* 7) sendo também respeitada a ordem referida de extracção dos operandos da *Stack* (à excepção do operador unário NOTB)

→ **Instruções de controlo:**

Mnemónica	Código	Argumento	Significado
JMP	23	<id_label> ou <endereço> ou <endr_relativo>	salto <u>incondicional</u> para um ponto do programa definido por um <int_16> calculado com base no argumento; esse <int_16> é um Endereço da Memória de Programa que pode estar marcado por uma <id_label>, pode ser já um <endereço> absoluto ou resultar da aplicação ao PC actual de um Offset dado por <endr_relativo> (ver Observação sobre este cálculo); a execução continua no Endereço atribuído a PC, calculado à custa do argumento: $PC = \text{Endereço}$;

Mnemónica	Código	Argumento	Significado
JMPF	18	<id_label> ou <endereço> ou <endr_relativo>	salto <u>condicional</u> para um ponto do programa definido por um <int_16> calculado com base no argumento; a determinação do Endereço a atribuir a PC é idêntica ao caso do JMP, mas a execução do salto está condicionada a um teste ao valor lógico do <int_8> do topo da <i>Stack</i> : SP = SP + 1; Valor = <i>Stack</i> [SP]; <u>Se</u> (Valor==FALSO) <u>Então</u> PC = Endereço; <u>Senão</u> PC = PC + 3;
CALL	19	<id_label> ou <endereço> ou <endr_relativo>	chamada <u>incondicional</u> de uma sub-rotina iniciada no Endereço definido por um <int_16> calculado com base no argumento; a determinação do Endereço a atribuir a PC é idêntica ao caso do JMP; porém, e depois de executar a sub-rotina (ver RET), a execução há-de prosseguir na instrução que sucede a CALL para o que é preciso salvar na <i>Stack</i> o valor apropriado do PC relativo a essa instrução (ProximoPC): ProximoPC = PC + 3; Msb = quociente(ProximoPC / 256) ; Lsb = resto(ProximoPC / 256) ; <i>Stack</i> [SP] = Lsb; SP = SP - 1 ; <i>Stack</i> [SP] = Msb; SP = SP - 1 ; PC = Endereço;
Mnemónica	Código	Significado	
RET	20	efectua o retorno de uma sub-rotina, desviando a execução do programa para um ponto salvaguardado na <i>Stack</i> , para o que lê dois <int_8> da <i>Stack</i> e constrói com eles um Endereço que é atribuído a PC: SP = SP + 1; Msb = <i>Stack</i> [SP]; SP = SP + 1; Lsb = <i>Stack</i> [SP]; Endereço = Msb * 256 + Lsb; PC = Endereço;	
HALT	21	aborta a execução do programa (independentemente de haver mais instruções para executar);	
NOOP	0	não faz nada (serve apenas para gastar tempo);	

Observações:

- um endereço absoluto é um endereço relativo ao início (endereço 0) da Memória de Programa;
- quando é dado um endereço relativo <endr_relativo> como argumento às instruções JMP, JMPF e CALL, o endereço absoluto Endereço obtém-se tomando o endereço do início da próxima instrução (PC+3) e somando-lhe o

Offset representado por <endr_relativo>, ou seja, Endereço = PC + 3 + <endr_relativo>

- o HALT deve ser usado para terminar os programas; caso contrário, e uma vez que a Memória de Programa é inicializada a zero, por defeito, e sendo zero o código do NOOP, a execução desta instrução prossegue até à última (31999) célula da Memória de Programa; outras utilizações do HALT são por exemplo reforçar o isolamento do código das sub-rotinas do resto do programa.

3.4 Erros e Avisos de assemblagem

Neste capítulo são apresentadas e discutidas as diversas situações que originam erros ou avisos durante o processo de assemblagem de um programa MSP, sobre o ambiente WinMSP 1.2.

Um aviso é um alerta para uma situação que apesar de não constituir obstáculo à geração de código executável, pode ser um sintoma de algo que não está semânticamente correcto. Certos erros, aliás, só serão detectáveis durante a execução².

Apresentam-se de seguida os erros e avisos gerados pelo WinMSP 1.2 durante a assemblagem, e em grupos que reflectem algo em comum. Para cada erro e aviso, fornece-se um código, uma descrição sumária, uma descrição desenvolvida e, quando se ache conveniente, um exemplo de uma situação que provoca o erro ou aviso em questão.

3.4.1 Erros

Erro Interno

Código: 0

Sumário: Memória de Dados e/ou Memória de Programa não disponíveis.

Descrição: Houve um erro interno que impediu a alocação da Memória de Dados e/ou de Programa. Devem-se fechar uma ou mais aplicações Windows a fim de obter memória livre suficiente para correr o WinMSP sem problemas.

Erros da Zona de Dados

Código: 1

Sumário: MEMORIA DE DADOS esperada.

Descrição: O delimitador reservado "MEMORIA DE DADOS" não foi reconhecido. Deve ocupar uma só linha e é obrigatório mesmo que não se declarem variáveis.

Código: 2

Sumário: Capacidade da Memória de Dados esgotada.

²Para uma descrição dos erros em *run-time* ver Manual do Utilizador.

Descrição: As declarações de variáveis anteriores esgotaram a Memória de Dados, não sendo possível fazer mais declarações. Esta situação é muito rara e só ocorrerá, em geral, quando se declaram *arrays* muito grandes.

Exemplo:

```
MEMORIA DE DADOS
var1 0 TAM 20000 ; OK
var2 19999 TAM 20000 ; esgota as 32000 posições disponíveis !
```

Código: 3

Sumário: Endereço de declaração inválido.

Descrição: Endereço inicial atribuído a uma variável, na sua declaração, inválido. Motivos possíveis são: endereço superior a 31999 ou colisão com uma zona previamente alocada a outra variável. Recorde-se que as declarações não são necessariamente contíguas, mas são obrigatoriamente exclusivas. As colisões também podem ser originadas pelo tamanho declarado, o que gera o erro ...

Exemplo:

```
MEMORIA DE DADOS
var1 100 TAM 20 ; OK (intervalo [100,119] reservado)
var2 110 TAM 1 ; colisão ! ( 110 pertence a [100,119], já
; reservado)
```

Código: 4

Sumário: Redefinição de identificador de variável.

Descrição: Declaração de uma variável, na Zona de Dados, cujo identificador foi já usado para outra variável.

Exemplo:

```
MEMORIA DE DADOS
x 0 TAM 1 ; OK
x 200 TAM 3 ; redeclaração de "x" !
```

Código: 5

Sumário: Endereço de variável não reconhecido.

Descrição: A seguir á declaração de uma variável é obrigatório especificar o seu endereço de declaração (i.e., o seu endereço inicial), que deve ser um número inteiro positivo de 16 *bits*, na gama [0,31999] (espaço de endereçamento da Memória de Dados).

Exemplo:

```
MEMORIA DE DADOS
y TAM 1 ; falta o endereço de declaração !
z 123abc--?? TAM 3 ; "123abc--??" não é um inteiro de 16 bits!
```

Código: 6

Sumário: TAMANHO esperado.

Descrição: A palavra reservada "TAMANHO" (ou uma das suas abreviaturas possíveis, até "TAM") não foi reconhecida. Esta palavra reservada sucede ao endereço e antecede a especificação da dimensão da variável declarada. Este separador é obrigatório.

Exemplo:

```
MEMORIA DE DADOS
a 0 100 ; separador TAM esperado entre 0 e 100 !
b 200 T 12 ; "T" não é separador da família do TAMANHO !
```

Código: 7

Sumário: Valor do TAMANHO esperado.

Descrição: A seguir à palavra reservada TAMANHO é obrigatório especificar a dimensão da variável, que deve ser um número inteiro de 16 *bits*, na gama [1,32000].

Exemplo:

```

MEMORIA DE DADOS
x 0 TAM 10 ; OK
y 20 TAM ; Oops! Esqueceu-se o valor do tamanho !

```

Código: 8

Sumário: Valor do TAMANHO inválido.

Descrição: A dimensão que se pretende atribuir à variável ou é 0 (zero), ou é superior à dimensão da Memória de Dados (32000), ou colide (total ou parcialmente) com uma zona previamente alocada a outra variável ou então é tal que o extremo direito da zona alocada à variável seria superior a 31999.

Exemplo:

```

MEMORIA DE DADOS
x 0 TAM 0 ; variável de dimensão 0 (zero) !!
y 10 TAM 40000 ; dimensão superior a 32000 !!
z 100 TAM 10 ; OK
a 90 TAM 50 ; colisão com zona reservada a "z" !!
b 31000 TAM 1000 ; há um byte "fora" da Memória de Dados !

```

Código: 9

Sumário: Valor(es) de inicialização em [-128,255] esperado(s).

Descrição: Declarando a palavra reservada VALOR (ou opcionalmente uma abreviatura até VAL), torna-se obrigatório especificar pelo menos um valor de inicialização no intervalo [-128,255], com sinal opcional para os positivos. Os restantes valores (caso o TAMANHO seja superior a um) podem ser omitidos, sendo por defeito inicializados a zero.

Exemplo:

```

MEMORIA DE DADOS
x 0 TAM 1 VAL ; Oops! Esqueceu-se o valor de inicialização !
y 10 TAM 2 ; OK. Inicialização total a zero.
z 20 TAM 3 VAL 100 ; OK. Restantes valores inicializados a zero.
a 40 TAM 1 VAL 1|b%3 ; Oops! Lixo em vez de um inteiro em
                        ; [-128,255] !

```

Código: 10

Sumário: Valores de inicialização excedem o espaço reservado.

Descrição: A lista de valores fornecida para inicializar o espaço de memória alocado a uma variável contém mais valores do que os *bytes* reservados para essa variável.

Exemplo:

```

MEMORIA DE DADOS
x 0 TAM 2 VALORES 1 2 3 ; Oops! Valor 3 em excesso !

```

Código: 11

Sumário: Valor inteiro de 8 *bits* fora do intervalo [-128,255].

Descrição: Uma constante de inicialização não pode exceder 255 nem ser inferior a -128. Valores superiores a 127 são interpretados contextualmente.

Exemplo:

MEMORIA DE DADOS

z 0 TAM 1 VAL 256 ; Valor 256 superior a 255 !

a 1 TAM 1 VAL -129 ; Valor -129 inferior a -128 !

Erros comuns à Zona de Dados e de Código

Código: 12

Sumário: Palavra inesperada ou desconhecida.

Descrição: Foi reconhecido um conjunto de caracteres sem significado para a linguagem MSP. Têm significado: identificadores alfanuméricos para variáveis e *labels* se não forem palavras reservadas; números inteiros (8/16 *bits*) com ou sem sinal (+ é opcional nos positivos). Os comentários devem ser precedidos de ';'. Este erro também ocorre se a seguir à palavra reservada CODIGO não se mudar de linha e vier algo não comentado.

Exemplo:

MEMORIA DE DADOS d7)8/&f8 ; Oops! d7)8/&f8 é lixo...

x 0 TAM 1 d7)8/&f8 ; Lixo outra vez

y 10 TAM 2 VAL 1 88d7)8/&f8 ; Lixo de novo

CODIGO 88d7)8/&f8 ; e outra vez lixo

88d7)8/&f8 ; uma linha de lixo na Zona de Código...

Código: 13

Sumário: Identificador de variável inválido ou CODIGO esperado.

Descrição: Não foi reconhecido um identificador de variável, alfanumérico e válido, ou não se conseguiu reconhecer o delimitador CODIGO que inicia a Zona de Código.

Exemplo:

MEMORIA DE DADOS

/gghg% ; uma linha de lixo na Zona de Dados

PUSH 10 ; Oops! PUSH é palavra reservada...

Erros da Zona de Código

Código: 14

Sumário: Fim de linha não encontrado (palavra inesperada) ou instrução esperada.

Descrição: Na Zona de Código, as linhas só podem ser: vazias, apenas de comentário ou com uma instrução, opcionalmente antecedida de uma *label* e/ou seguida de um comentário, não podendo coexistir duas instruções ou *labels* na mesma linha, mas sendo permitidos comentários aninhados.

Exemplo:

CODIGO

PUSH 10 PSHA x ; Duas instruções numa linha de texto !!

RET (7lijklsfd ; Lixo ...

Label1: Label2 ; Oops! Duas *labels* ...

Código: 15

Sumário: Capacidade da Memória de Programa esgotada.

Descrição: As instruções anteriores esgotaram a Memória de Programa, não sendo possível aceitar mais Código. Este erro ocorre quando a Memória de Programa livre não permite a introdução da instrução actual na sua totalidade.

Exemplo: Esta situação é muito rara em termos práticos e por isso não se fornece exemplo.

Erros na declaração de *labels***Código: 16**

Sumário: Redefinição de identificador de *label*.

Descrição: Declaração de uma *label* cujo identificador foi previamente usado para outra *label*.

Exemplo:

CODIGO

Ciclo: PSHA i

...

Ciclo: RET ; Oops! Identificador de *label* Ciclo já usado ...

Código: 17

Sumário: Faltam dois pontos (:) a uma *label* potencial.

Descrição: A sintaxe da linguagem obriga a que na declaração de uma *label*, esta seja seguida do carácter ":".

Exemplo:

CODIGO

Subrotina PSHA x ; Oops! Esqueceu-se o carácter ':' !

Fim : RET ; OK! ':' não precisa estar "colado" à *label*

Erro em instruções sem argumentos**Código: 18**

Sumário: Argumento não esperado.

Descrição: A instrução reconhecida não necessita mais argumentos que os que já foram reconhecidos. Este erro desaparece se o argumento supérfluo for suprimido ou comentado.

Exemplo:

CODIGO

ADD 1 ; ADD não tem argumentos !

PSHA x z ; PSHA só tem um argumento !

Erro específico da instrução PUSH**Código: 19**

Sumário: Argumento inteiro em [-128,255] esperado..

Descrição: A instrução PUSH coloca na *Stack* um argumento mandatório de 8 *bits*. Qualquer outro argumento é inválido. A ausência de argumento também provoca este erro.

Exemplo:

```
CODIGO
PUSH x ; x não é um inteiro de 8 bits !
PUSH ; Oops! Esqueceu-se o argumento !
```

Erros específicos da instrução PSHA**Código: 20**

Sumário: Endereço inválido (inteiro de 16 *bits* em [0..31999] esperado).

Descrição: O argumento do PSHA foi reconhecido como inteiro de 16 *bits*, mas o seu valor cai fora do espaço de endereçamento da Memória de Dados ([0..31999]).

Exemplo:

```
CODIGO
PSHA 32000 ; 32000 não pertence a [0..31999] !
```

Código: 21

Sumário: Variável desconhecida.

Descrição: Tentou-se usar como argumento do PSHA uma variável sem ter sido previamente declarada, na Zona de Dados.

Exemplo:

```
MEMORIA DE DADOS
x 0 TAM 1
CODIGO
PSHA z ; "z" não foi declarada na Zona de Dados !
```

Código: 22

Sumário: Argumento inválido (*label* em vez de variável).

Descrição: Tentou-se usar como argumento da instrução PSHA um identificador de *label*, quando provavelmente se pretendia usar um identificador de variável. Como identificadores de *labels* e variáveis podem coincidir, este erro só acontece se não há nenhuma variável declarada com um nome igual à *label*.

Exemplo:

```
MEMORIA DE DADOS
x 0 TAM 1
CODIGO
y: PSHA y ; "y" não é variável (apesar de ser label) !!asd
x: PSHA x ; OK: "x" é simultaneamente identificador de label
      ; e variável.
```

Código: 23

Sumário: Argumento inválido (não referencía nenhuma variável).

Descrição: Não se conseguiu reconhecer um argumento válido para PSHA. São argumentos válidos: um endereço de 16 *bits* (de variável ou não), ou o próprio nome de uma variável em vez do seu endereço.

Exemplo:

```
MEMORIA DE DADOS
x 0 TAM 1
CODIGO
PSHA 1234 ; OK: um endereço de 16 bits válido.
PSHA x ; OK: um identificador de variável declarada.
PSHA 0/887&5 ; Oops! Lixo...
```


Erros específicos de instruções de salto (JMP, JMPF, CALL)

Código: 24

Sumário: *Label* não definida.

Descrição: Uma *label* usada numa instrução de salto nunca foi definida, ou seja, associada a uma instrução na Zona de Código.

Exemplo:

```
CODIGO ; início da Zona de Código
CALL fim ; a label "fim" nunca chega a ser declarada !
HALT ; fim do programa
```

Código: 25

Sumário: Tentativa de salto para fora da Memória de Programa.

Descrição: O endereço absoluto ou relativo usados como argumento de uma instrução de salto, estão a provocar um salto para fora do espaço de endereçamento da Memória de Programa ([0,31999]).

Exemplo:

```
CÓDIGO
JMP -4 ; Oops! Salto para o endereço absoluto -1 !
JMP 32000 ; Oops! Salto para endereço absoluto superior a 31999!
JMP +31992 ; idem
```

Código: 26

Sumário: Argumento inválido (não faz referência à Memória de Programa).

Descrição: Não se conseguiu reconhecer um argumento válido para uma instrução de salto. São argumentos válidos: o endereço associado a uma *label* ou um endereço absoluto de 16 *bits* na gama [0..31999] ou um deslocamento relativo (positivo ou negativo) que também se traduza num endereço absoluto de 16 *bits* na gama [0..31999].

Exemplo:

```
CODIGO
JMP k(76% ; Oops! Lixo...
```

Erro Externo

Código: 27

Sumário: Programa fonte vazio.

Descrição: Não se detectou nenhum texto para assembler. Verificar conteúdo do ficheiro fonte.

3.4.2 Avisos

Avisos da Zona de Dados

Código: 0

Sumário: Memória de Dados "vazia".

Descrição: Nenhuma variável foi declarada na Zona de Dados e portanto a Memória de Dados está completamente livre e inicializada a zero.

Exemplo:

```
MEMORIA DE DADOS
CODIGO
...
```

Código: 1

Sumário: Valores não inicializados.

Descrição: Uma variável declarada na Zona de Dados não foi inicializada ou foi-o parcialmente. As células não inicializadas assumem o valor 0 (zero) por defeito.

Exemplo:

```
MEMORIA DE DADOS
x 0 TAM 1 ; nenhuma inicialização ...
y 1 TAM 3 VAL 1 ; inicialização parcial (2ª e 3ª células
                  ; a zero).
```

Código: 2

Sumário: Variável declarada, mas nunca usada.

Descrição: Uma variável declarada na Zona de Dados nunca chega a ser referenciada na Zona de Código.

Exemplo:

```
MEMORIA DE DADOS ; início da Zona de Dados
a 0 TAM 1 ; a variável "a" nunca chega a ser usada
CODIGO ; fim da Zona de Dados e início da Zona de Código
HALT ; fim do programa
```

Avisos da Zona de Código

Código: 3

Sumário: Memória de Programa "vazia".

Descrição: Como nenhuma instrução foi escrita na Zona de Código do programa, adverte-se que qualquer tentativa para o correr vai gerar erro de execução! À semelhança da Memória de Programa "vazia", as células não inicializadas assumem o valor 0 (zero) por defeito.

Exemplo:

```
MEMORIA DE DADOS
CODIGO ; Oops! Esqueceram-se variáveis e instruções !
```

Código: 4

Sumário: *Label* declarada, mas nunca usada.

Descrição: Uma *label* declarada na Zona de Código nunca é usada nessa zona.

Exemplo:

```
CODIGO
inicio: IN ; a label "inicio" jamais é referenciada no
          ; programa
...
OUT
HALT
```

Código: 5

Sumário: *Label* e variável com o mesmo nome.

Descrição: O identificador de uma *label* declarada na Zona de Código, coincide com o identificador de uma variável da Zona de Dados.

Exemplo:

```
MEMORIA DE DADOS
soma 0 TAM 1
CODIGO
soma: HALT ; "soma" identifica uma label e uma variável
          ; simultâneamente ...
```

Código: 6

Sumário: Endereço que não referencía nenhuma variável.

Descrição: Está-se a usar como argumento da instrução PSHA, um endereço de 16 *bits* válido (ou seja, na gama [0,31999]) mas que não faz referência a nenhuma célula atribuída a uma variável declarada na Zona de Dados.

Exemplo:

```
MEMORIA DE DADOS
x 0 TAM 100
CODIGO
PSHA 0 ; OK: 0 é endereço de declaração da variável "x"
PSHA 50 ; OK: 50 é endereço interior à zona reservada para "x"
PSHA 200 ; Oops! Não há nenhuma variável que possua esta célula!
```

Anexo A - Tabela de conversão de e para C2

Binário	[0,255]	[-128,127](C2)
0 0000000	0	0
0 0000001	1	1
0 0000010	2	2
0 0000011	3	3
0 0000100	4	4
0 0000101	5	5
0 0000110	6	6
0 0000111	7	7
0 0001000	8	8
0 0001001	9	9
0 0001010	10	10
0 0001011	11	11
0 0001100	12	12
0 0001101	13	13
0 0001110	14	14
0 0001111	15	15
0 0010000	16	16
0 0010001	17	17
0 0010010	18	18
0 0010011	19	19
0 0010100	20	20
0 0010101	21	21
0 0010110	22	22
0 0010111	23	23
0 0011000	24	24
0 0011001	25	25
0 0011010	26	26
0 0011011	27	27
0 0011100	28	28
0 0011101	29	29
0 0011110	30	30
0 0011111	31	31
0 0100000	32	32
0 0100001	33	33
0 0100010	34	34
0 0100011	35	35
0 0100100	36	36
0 0100101	37	37
0 0100110	38	38
0 0100111	39	39
0 0101000	40	40
0 0101001	41	41
0 0101010	42	42
0 0101011	43	43
0 0101100	44	44
0 0101101	45	45
0 0101110	46	46
0 0101111	47	47
0 0110000	48	48
0 0110001	49	49
0 0110010	50	50
0 0110011	51	51
0 0110100	52	52
0 0110101	53	53
0 0110110	54	54
0 0110111	55	55
0 0111000	56	56
0 0111001	57	57
0 0111010	58	58
0 0111011	59	59
0 0111100	60	60
0 0111101	61	61
0 0111110	62	62
0 0111111	63	63

Binário	[0,255]	[-128,127](C2)
0 1000000	64	64
0 1000001	65	65
0 1000010	66	66
0 1000011	67	67
0 1000100	68	68
0 1000101	69	69
0 1000110	70	70
0 1000111	71	71
0 1001000	72	72
0 1001001	73	73
0 1001010	74	74
0 1001011	75	75
0 1001100	76	76
0 1001101	77	77
0 1001110	78	78
0 1001111	79	79
0 1010000	80	80
0 1010001	81	81
0 1010010	82	82
0 1010011	83	83
0 1010100	84	84
0 1010101	85	85
0 1010110	86	86
0 1010111	87	87
0 1011000	88	88
0 1011001	89	89
0 1011010	90	90
0 1011011	91	91
0 1011100	92	92
0 1011101	93	93
0 1011110	94	94
0 1011111	95	95
0 1100000	96	96
0 1100001	97	97
0 1100010	98	98
0 1100011	99	99
0 1100100	100	100
0 1100101	101	101
0 1100110	102	102
0 1100111	103	103
0 1101000	104	104
0 1101001	105	105
0 1101010	106	106
0 1101011	107	107
0 1101100	108	108
0 1101101	109	109
0 1101110	110	110
0 1101111	111	111
0 1110000	112	112
0 1110001	113	113
0 1110010	114	114
0 1110011	115	115
0 1110100	116	116
0 1110101	117	117
0 1110110	118	118
0 1110111	119	119
0 1111000	120	120
0 1111001	121	121
0 1111010	122	122
0 1111011	123	123
0 1111100	124	124
0 1111101	125	125
0 1111110	126	126
0 1111111	127	127

Binário	[0,255]	[-128,127](C2)
1 0000000	128	-128
1 0000001	129	-127
1 0000010	130	-126
1 0000011	131	-125
1 0000100	132	-124
1 0000101	133	-123
1 0000110	134	-122
1 0000111	135	-121
1 0001000	136	-120
1 0001001	137	-119
1 0001010	138	-118
1 0001011	139	-117
1 0001100	140	-116
1 0001101	141	-115
1 0001110	142	-114
1 0001111	143	-113
1 0010000	144	-112
1 0010001	145	-111
1 0010010	146	-110
1 0010011	147	-109
1 0010100	148	-108
1 0010101	149	-107
1 0010110	150	-106
1 0010111	151	-105
1 0011000	152	-104
1 0011001	153	-103
1 0011010	154	-102
1 0011011	155	-101
1 0011100	156	-100
1 0011101	157	-99
1 0011110	158	-98
1 0011111	159	-97
1 0100000	160	-96
1 0100001	161	-95
1 0100010	162	-94
1 0100011	163	-93
1 0100100	164	-92
1 0100101	165	-91
1 0100110	166	-90
1 0100111	167	-89
1 0101000	168	-88
1 0101001	169	-87
1 0101010	170	-86
1 0101011	171	-85
1 0101100	172	-84
1 0101101	173	-83
1 0101110	174	-82
1 0101111	175	-81
1 0110000	176	-80
1 0110001	177	-79
1 0110010	178	-78
1 0110011	179	-77
1 0110100	180	-76
1 0110101	181	-75
1 0110110	182	-74
1 0110111	183	-73
1 0111000	184	-72
1 0111001	185	-71
1 0111010	186	-70
1 0111011	187	-69
1 0111100	188	-68
1 0111101	189	-67
1 0111110	190	-66
1 0111111	191	-65

Binário	[0,255]	[-128,127](C2)
1 1000000	192	-64
1 1000001	193	-63
1 1000010	194	-62
1 1000011	195	-61
1 1000100	196	-60
1 1000101	197	-59
1 1000110	198	-58
1 1000111	199	-57
1 1001000	200	-56
1 1001001	201	-55
1 1001010	202	-54
1 1001011	203	-53
1 1001100	204	-52
1 1001101	205	-51
1 1001110	206	-50
1 1001111	207	-49
1 1010000	208	-48
1 1010001	209	-47
1 1010010	210	-46
1 1010011	211	-45
1 1010100	212	-44
1 1010101	213	-43
1 1010110	214	-42
1 1010111	215	-41
1 1011000	216	-40
1 1011001	217	-39
1 1011010	218	-38
1 1011011	219	-37
1 1011100	220	-36
1 1011101	221	-35
1 1011110	222	-34
1 1011111	223	-33
1 1100000	224	-32
1 1100001	225	-31
1 1100010	226	-30
1 1100011	227	-29
1 1100100	228	-28
1 1100101	229	-27
1 1100110	230	-26
1 1100111	231	-25
1 1101000	232	-24
1 1101001	233	-23
1 1101010	234	-22
1 1101011	235	-21
1 1101100	236	-20
1 1101101	237	-19
1 1101110	238	-18
1 1101111	239	-17
1 1110000	240	-16
1 1110001	241	-15
1 1110010	242	-14
1 1110011	243	-13
1 1110100	244	-12
1 1110101	245	-11
1 1110110	246	-10
1 1110111	247	-9
1 1111000	248	-8
1 1111001	249	-7
1 1111010	250	-6
1 1111011	251	-5
1 1111100	252	-4
1 1111101	253	-3
1 1111110	254	-2
1 1111111	255	-1