1. Classificação *INPE-COM.1/TDL*
C.D.U.: 621.3.049.77

2. Período

4. Distribuição

interna [ ]

externa [X]

3. Palavras Chaves (selecionadas pelo autor)
MICROPROGRAMAÇÃO
MICROCÓDIGO
LINGUAGEM DE MICROPROGRAMAÇÃO
TRADUTOR

5. Relatório nº
INPE-2031-TDL/049

6. Data
Abril, 1981

7. Revisado por
Eduardo W. Bergamini

8. Título e Sub-Título

LMP, UMA LINGUAGEM DE MICROPROGRAMAÇÃO

9. Autorizado por
Nelson de Jesus Parada
Diretor

10. Setor *DEE/DEL* Código

11. Nº de cópias *11*

12. Autoria *Wilson Yamaguti*

14. Nº de páginas *105*

15. Preço

13. Assinatura Responsável

16. Sumário/Notas

Define-se uma linguagem de microprogramação denominada LMP, projetada para oferecer recursos de desenvolvimento de sistemas microprogramados. Sua meta principal é a eficiência dos microcódigos gerados em microinstruções do tipo horizontal. Pretende-se ainda, através desta linguagem, obter uma forma padronizada de documentação de microprogramas. Apresentam-se, também, algumas considerações quanto ao desenvolvimento de um tradutor para essa linguagem.

17. Observações *Tese de Mestrado em Eletrônica e Telecomunicações - Opção Sistemas Digitais e Analógicos, aprovada em 02 de dezembro de 1980.*

Aprovada pela Banca Examinadora
em cumprimento dos requisitos exigidos
para a obtenção do Título de Mestre em
Eletrônica e Telecomunicações

Dr.Flávio Roberto Dias Velasco — Presidente

Dr.Eduardo Whitaker Bergamini — Orientador

Dr.Michael Anthony Stanton — Membro da Banca -convidado-

Eng.Arry Carlos Buss Filho,MSc. — Membro da Banca

Eng.Juan Carlos Pinto de Garrido,MSc. — Membro da Banca

Wilson Yamaguti — candidato

São José dos Campos, 02 de dezembro de 1980

ÍNDICE

Pág.

ABSTRACT .... *v*

LISTA DE FIGURAS .... *vi*

LISTA DE TABELAS .... *viii*

CAPÍTULO I - INTRODUÇÃO 1

CAPÍTULO II - ARQUITETURAS MICROPROGRAMADAS E LINGUAGENS DE MICROPROGRAMAÇÃO 7

2.1 - Unidade de Controle Microprogramada 7

2.2 - Projeto e Implementação da Microinstrução 11

2.3 - Linguagens de Microprogramação 15

CAPÍTULO III - CARACTERÍSTICAS GERAIS DA LINGUAGEM LMP 23

3.1 - Estrutura da linguagem LMP 24

3.2 - Constantes e Expressões 26

3.3. - Declarações 27

3.4 - Comandos e Microcomandos 28

3.5 - Opções de Pós-processamento 29

3.6 - Opções de Controle 29

CAPÍTULO IV - DEFINIÇÃO DA LINGUAGEM LMP 31

4.1 - Componentes da linguagem 31

4.1.1 - Símbolos básicos e palavras reservadas 32

4.1.2 - Constantes 32

4.1.3 - Comentários 35

4.1.4 - Identificadores 37

4.1.5 - Microprograma 38

4.1.6 - Invocação de definição 40

4.1.7 - Opções de controle 40

4.2 - Declarações 43

4.2.1 - Declaração de memória de controle 44

4.2.2 - Declaração de campo 45

4.2.3 - Declaração de formato 49

Pág.

4.2.4 - Declaração de definição 51
4.2.5 - Declaração de microinstrução 53
4.2.6 - Declaração de rótulo 57
4.2.7 - Declaração de sub-rotina 58
4.3 - Expressões 59
4.4 - Comandos e Microcomandos 61
4.4.1 - Microcomando de atribuição 62
4.4.2 - Microcomando de concatenação 64
4.4.3 - Microcomando de configuração de campo anterior 67
4.5 - Opções de pós-processamento 67
4.5.1 - Mapeamento de PROMs 68
4.5.2 - Inversão de Bits 70
4.5.3 - Estados "don't care" 71
4.5.4 - Listagem de PROMs 72
4.5.5 - Saída em fita magnética 73

CAPÍTULO V - IMPLEMENTAÇÃO DO TRADUTOR LMP 75

5.1 - O analisador léxico ("SCANNER") 77
5.2 - O analisador sintático 79
5.3 - O analisador semântico e o gerador de código 80
5.4 - Rotinas de pós-processamento 80
5.5 - Organização da tabela de símbolos 80
5.6 - Erros 83

CAPÍTULO VI - CONCLUSÕES FINAIS 85

AGRADECIMENTOS 87

REFERÊNCIAS BIBLIOGRÁFICAS 89

APÊNDICE A - EXEMPLO DE APLICAÇÃO

ABSTRACT

A microprogramming language denominated LMP is defined in this work. It is a microprogramming language tailored for use in development of microprogrammed systems. The prime design goal of this language is to provide the ability to produce efficient microcode in the case of horizontal microprogramming. Another goal is to provide a standard form of microprogram documentation. Some considerations to implement a translator for this language are presented.

LISTA DE FIGURAS

	Pág.
I.1 - Diagrama funcional de um computador digital	1
I.2 - O sistema LMP/EMMAC para projetos de "firmware" auxiliados por computadores	5
I.3 - O processo LMP/EMMAC de geração, depuração e gravação de PROMs	6
II.1 - Controle microprogramado apresentado por Wilkes em 1951..	8
II.2 - Estrutura de uma unidade de controle microprogramada	9
II.3 - Tipos de estruturas de memória de controle	12
II.4 - Execução de uma microinstrução	14
II.5 - Relacionamento entre as linguagens de microprogramação e programação	16
II.6 - Exemplo de linguagem de microprogramação "assembly", utilizada no computador Digital Scientific META 4..........	17
II.7 - (a) Representação simbólica de uma microinstrução do computador IBM/360 modelo 30, utilizando uma linguagem do tipo fluxograma	18
(b) Exemplo de linguagem tipo fluxograma (IBM/360 modelo 30)	19
II.8 - Microprograma na linguagem ANIMIL	21
II.9 - Microprograma em PUMPKIN	22
III.1 - Estrutura de microprograma em LMP	25
IV. 1 - Exemplo de utilização de comentários em LMP	36
IV.2 - Exemplo de uma microinstrução e seus campos	46
IV.3 - Exemplo de declaração de campo correspondente à microinstrução dada na Figura IV.2	46
IV.4 - Formatos de microinstrução do HP 21MX	47
IV.5 - Declaração de campos do minicomputador HP 21MX, de microprogramação diagonal	48
IV.6 - Exemplo de declaração de formato	50
IV.7 - Organização das PROMs em uma memória de controle	70
V.1 - Estrutura básica do tradutor LMP	76
V.2 - O Analisador Léxico LMP	78
V.3 - Tabela de símbolos LMP	82
V.4 - Estrutura dos nós da Tabela de Símbolos LMP	84

Pág.

A.1 - Diagrama de blocos simplificado da unidade de controle para o exemplo de aplicação A.2

A.2 - Formatação da palavra de controle A.3

A.3 - Fluxograma do exemplo A.4

LISTA DE TABELAS

Pág.

IV.1 - Exemplo de utilização de constantes do tipo configuração de bits 35

IV.2 - Opções de listagem 42

CAPÍTULO I

INTRODUÇÃO

Com o advento de memórias semicondutoras cada vez mais rápidas, uma técnica alternativa de projeto de unidade de controle em um sistema de processamento de dados ganhou impulso considerável. Esta técnica, introduzida por M.V. Wilkes da Universidade de Cambridge em 1951 e denominada microprogramação, tem sido empregada com intensidade crescente, como uma técnica opcional de controle de circuitos lógicos.

O relacionamento típico da unidade de controle com as demais seções de um computador digital pode ser observado pelo esquema da Figura I.1 (Agrawala e Rauscher, 1976).

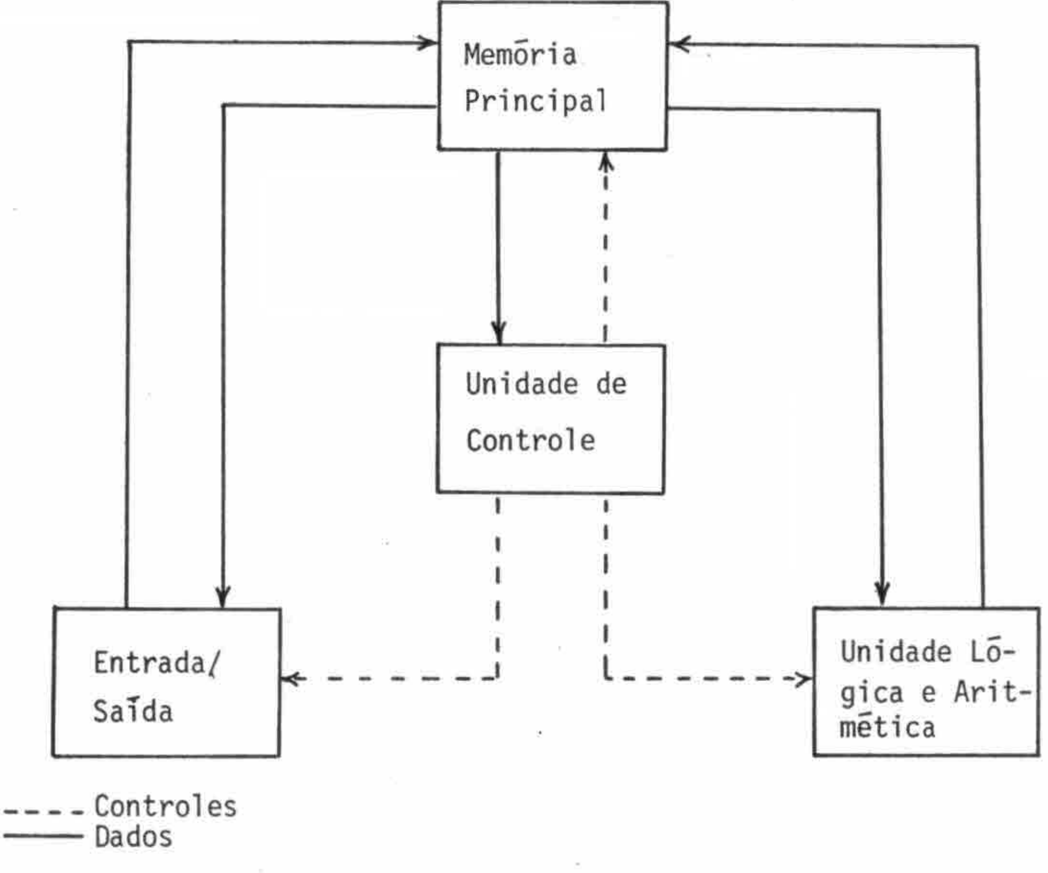


Fig. I.1 - Diagrama funcional de um computador digital.

O EMMAC é um projeto do Programa de Sistemas Digitais e Analógicos do Instituto de Pesquisas Espaciais (INPE), no qual,também, desenvolvem-se esforços no sentido de operacionalizar a linguagem LMP, de modo que ela venha a constituir um sistema completo para geração e depuração de microprogramas, como recurso de projeto auxiliado por computadores em seu laboratório.

A estrutura de geração e depuração de microprogramas, em vias de operacionalização no INPE, sistema LMP/EMMAC, é constituída pelo tradutor LMP e pelo EMMAC, esquematizados na Figura I.2.

O tradutor cruzado reside no sistema Burroughs B-6800,gerando, como saída de conexão, fitas magnéticas CCT. O EMMAC está acoplado ao minicomputador HP 21MX-E, através do qual os comandos são digitados via terminal de vídeo. Através de um destes comandos, a fita magnética com os microcódigos são armazenados em arquivos, em disco,para posterior carga nas memórias de emulação.

O sistema para o qual os microprogramas estão sendo depurados é conectado ao EMMAC, através de um conjunto de cabos. A depuração é feita através do Monitor EMMAC de maneira interativa com o microprogramador.

A Figura I.3 ilustra o processo de geração, depuração e gravação de PROM, pretendido no sistema LMP/EMMAC, enfatizando as correções sintáticas do microprograma em LMP, a depuração no EMMAC e a interação LMP-EMMAC.

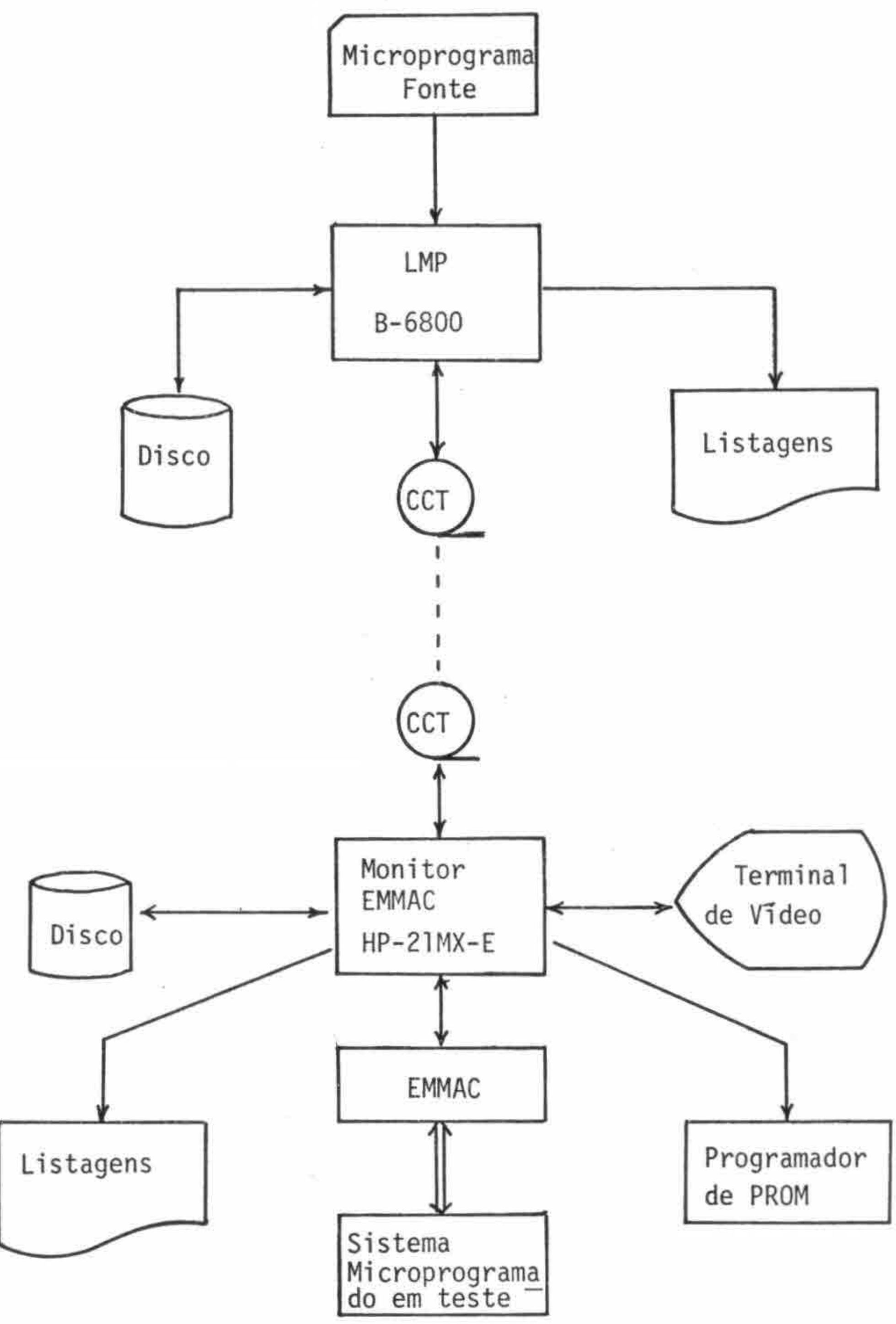


Fig. I.2 - O Sistema LMP/EMMAC para projetos de "firmware" auxiliados por computadores.

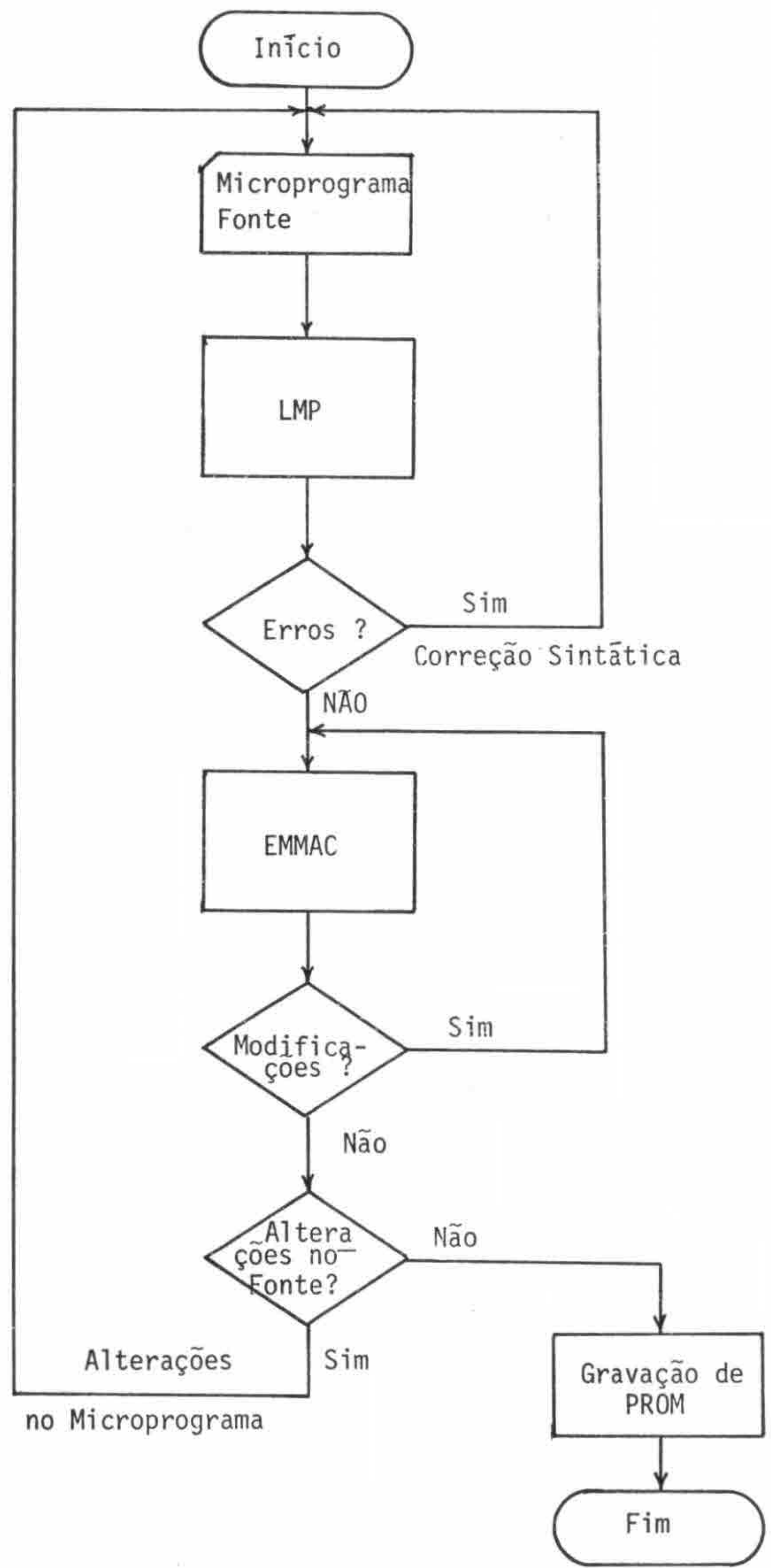


Fig. I.3 - O processo LMP/EMMAC de geração, depuração e gravação de PROMs.

CAPÍTULO II

ARQUITETURAS MICROPROGRAMADAS E LINGUAGENS DE MICROPROGRAMAÇÃO

Neste capítulo são feitas algumas considerações sobre arquiteturas microprogramadas, no sentido de caracterizar a unidade de controle microprogramada, bem como as abordagens tomadas no projeto e a implementação de uma microinstrução.

Apresentam-se, também, considerações sobre algumas linguagens de microprogramação existentes e tentativas de classificação das mesmas.

2.1 - UNIDADE DE CONTROLE MICROPROGRAMADA

A idéia original de M.V. Wilkes colocada em "The best way to design an automatic calculating machine" (Husson, 1970), apresentada numa conferência na Universidade de Manchester em 1951, sofreu algumas modificações. A maioria destas modificações foram apresentadas pelo próprio grupo em que trabalhava Wilkes.

A Figura II.1 esquematiza o controle microprogramado proposto por Wilkes, em 1951.

Neste esquema têm-se duas matrizes denominadas C e S, que constituem a memória de controle. A informação de cada matriz é determinada pela ativação de uma das linhas endereçadas pela árvore de decodificação. O conteúdo da matriz C atua diretamente nos recursos funcionais do processador, enquanto que, o conteúdo da matriz S fornece a informação da próxima linha a ser ativada. Todo este processo é sequenciado por pulsos de relógio.

Há ainda, a possibilidade de efetuar um desvio condicional, com escolha da linha a ser ativada, mediante a ocorrência ou não de certa condição armazenada no "flip-flop" FF.

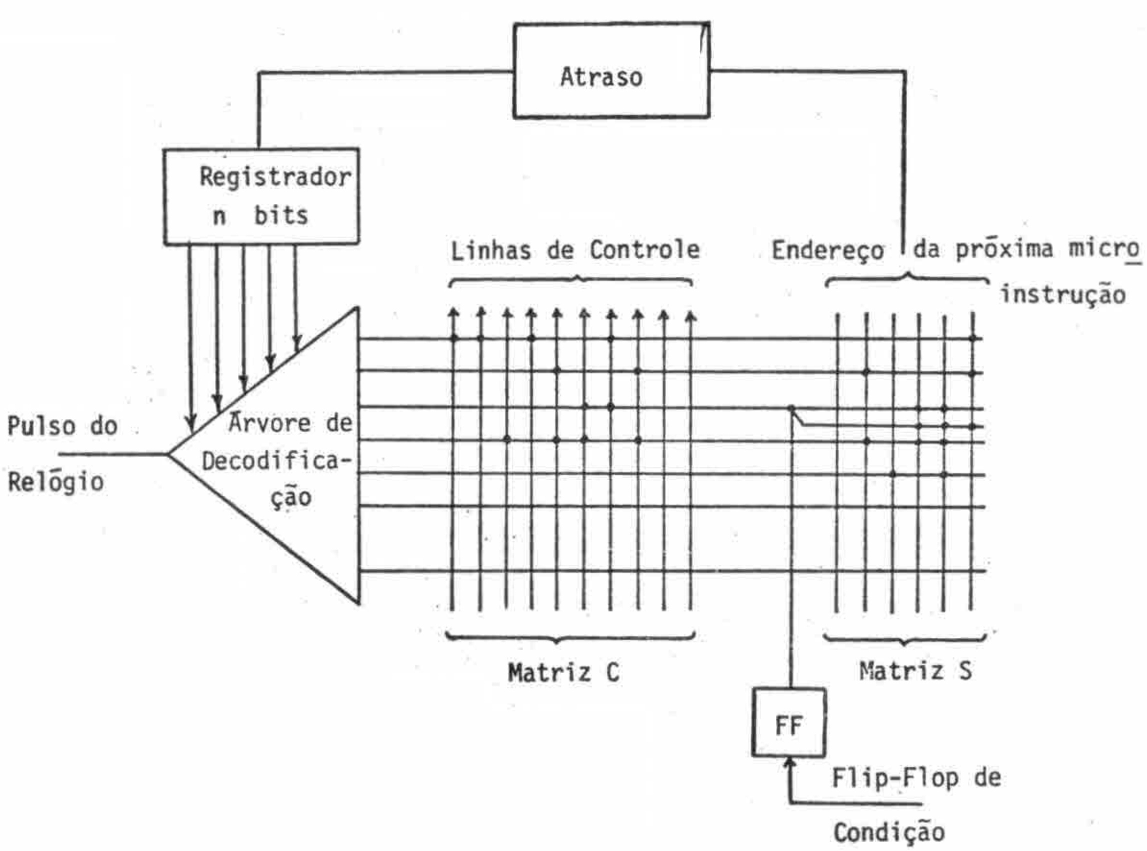


Fig. II.1 - Controle microprogramado apresentado por Wilkes em 1951.

FONTE: Husson (1970) p.24.

Para o esquema da Figura II.1 pode-se caracterizar:

a) o Sequenciador, como sendo aquele que gera o endereço da próxima microinstrução (matriz S).

b) a Memória de Controle, onde são armazenadas as microinstruções (matriz C).

c) a Microinstrução representada pelas linhas de controle.

A implementação da unidade de controle que utiliza a microprogramação tornou-se muito atraente após o surgimento de memórias monolíticas bipolares de alta velocidade, com custos decrescentes. A disponibilidade de outros circuitos integrados do tipo LSI, que implementam funções cada vez mais complexas, reduzindo sensivelmente o número de circuitos necessários à unidade de controle e a outras partes de um computador.

A Figura II.2 apresenta a estrutura típica, atual, de uma unidade de controle microprogramada.

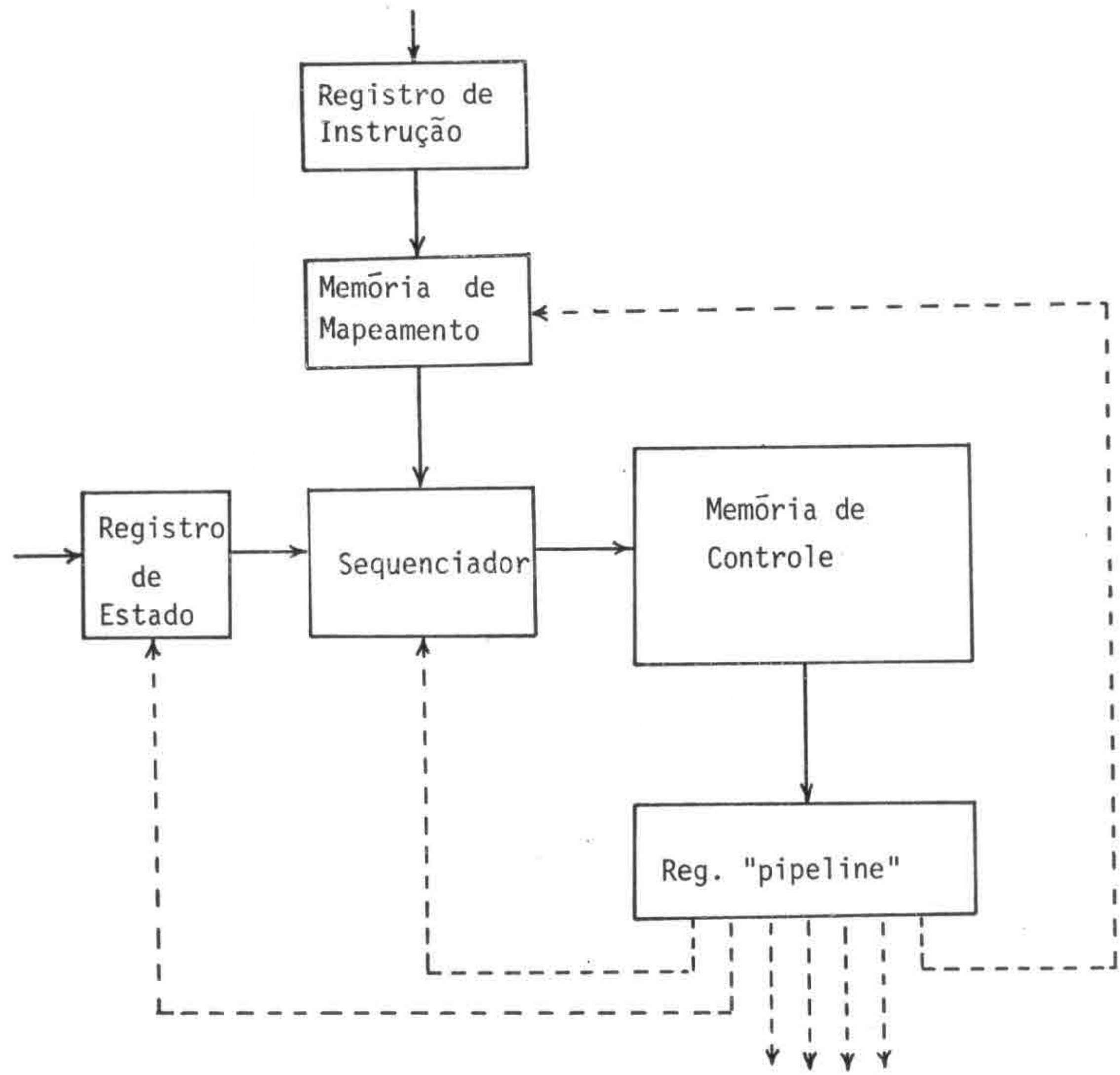


Fig. II.2 - Estrutura de uma unidade de controle microprogramada.

A instrução de máquina de um dado computador é transferido da memória principal para o registro de instrução, no ciclo denominado busca de instrução ("fetch"). A partir do conteúdo deste registro, calcula-se o endereço da memória de controle, onde está armazenado o conjunto de microinstruções que executam a instrução de máquina, através do controle direto dos recursos funcionais envolvidos.

Após a execução desta instrução, é executada novamente a micro-rotina de busca de instruções, repetindo-se o processo.

A complexidade do sequenciador varia de um simples registro contador àquele realizado com sofisticados circuitos LSI, que atuam no fluxo de controle das microinstruções.

Outro fato a ressaltar é a maneira pela qual pode ser estruturada a memória de controle. Segundo Agrawala e Rauscher (1976), sete estruturas básicas podem ser consideradas:

a) uma microinstrução por palavra;

b) duas microinstruções por palavra;

c) estrutura em blocos, isto é, a memória é dividida em páginas;

d) estrutura partilhada em duas áreas de armazenamento com palavras de tamanhos diferentes. A área de palavra de tamanho menor mapeia o endereço da outra memória, de modo que esta atue sobre os recursos funcionais do processador, o qual requer, em princípio, menor número de bits de controle no conjunto;

e) estrutura hierarquizada em dois níveis, onde o nível inferior interpreta os bits do nível superior, de maneira semelhante, como uma memória de controle interpreta as instruções de máquina;

f) estrutura "cache", onde os microcódigos são armazenados numa memória mais lenta, por exemplo a memória principal, e, quando forem solicitados, são transferidos para a memória de controle para execução, Desta forma, a memória de controle tem o seu tamanho reduzido;

g) microinstrução implementada na própria memória principal, da qual é efetuada diretamente a busca e a execução de microinstrução.

Na Figura II.3 estão esquematizadas as várias estruturas da memória de controle.

Nos computadores comerciais, o tamanho da palavra de controle varia tipicamente de 16 a 100 bits, e o tamanho da memória entre 256 a 4096 posições.

2.2 - PROJETO E IMPLEMENTAÇÃO DA MICROINSTRUÇÃO

No projeto da microinstrução, deve-se considerar não só os recursos do processador, a serem controlados, mas também, a distribuição desses controles em campos e formatos que simplifiquem a decodificação e a execução das mesmas. Inúmeras são as maneiras de realizar a estruturação das microinstruções. A classificação de cada tipo de estrutura está relacionada com o número de microoperações que a compõem.

A microprogramação é dita vertical quando cada microinstrução executa apenas uma microoperação, enquanto que na denominada de horizontal, a microinstrução contém diversas microoperações que são executadas simultaneamente, o que acarreta um tamanho maior de palavra e aumento da velocidade da máquina. Há, ainda, a microprogramação diagonal que apresenta características intermediárias. A divisão entre a microprogramação vertical e a horizontal se torna difícil com circuitos cada vez mais complexos, onde uma dada microoperação realiza um conjunto de tarefas simultâneas, correspondentes a várias microoperações em circuitos anteriores.

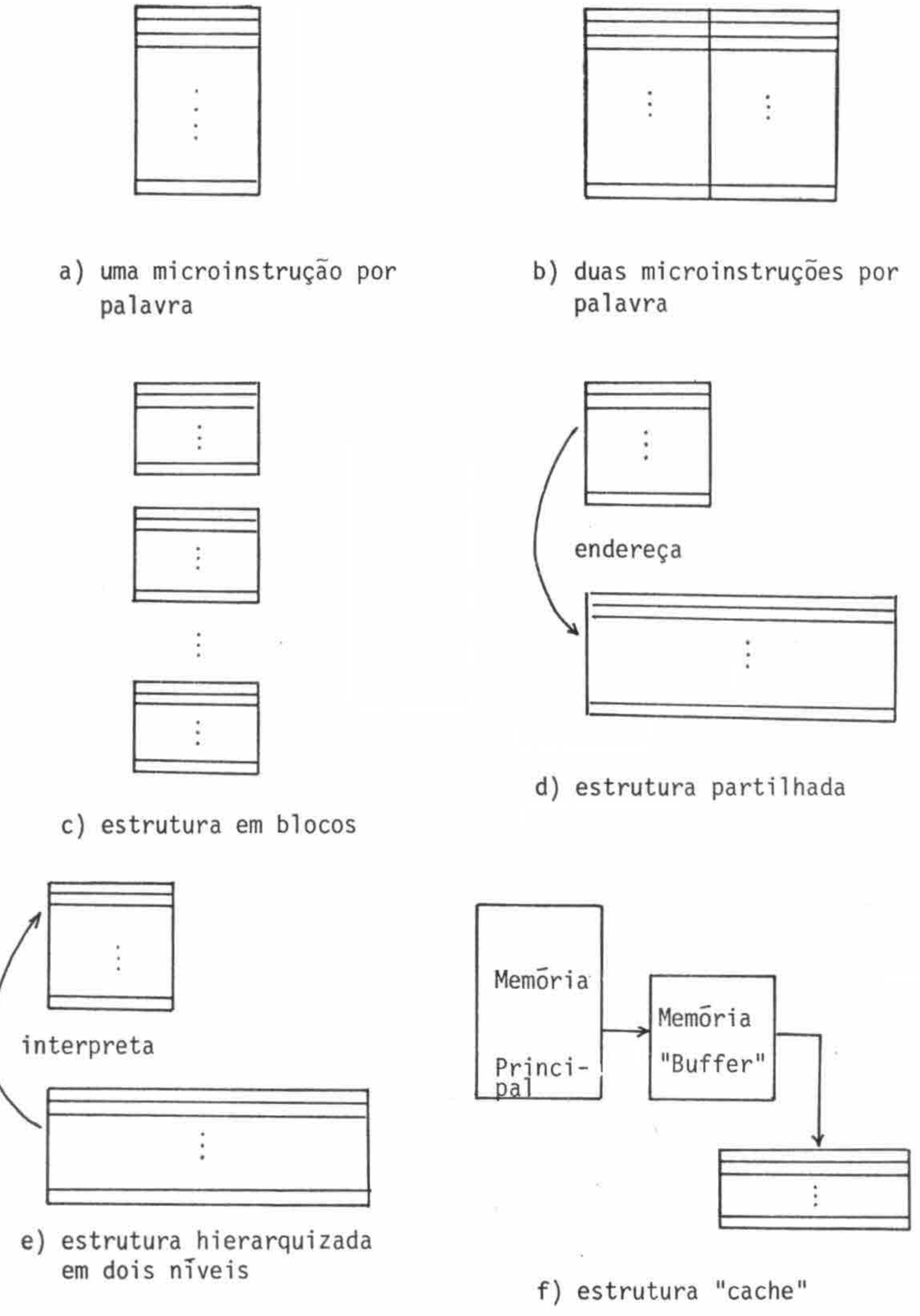
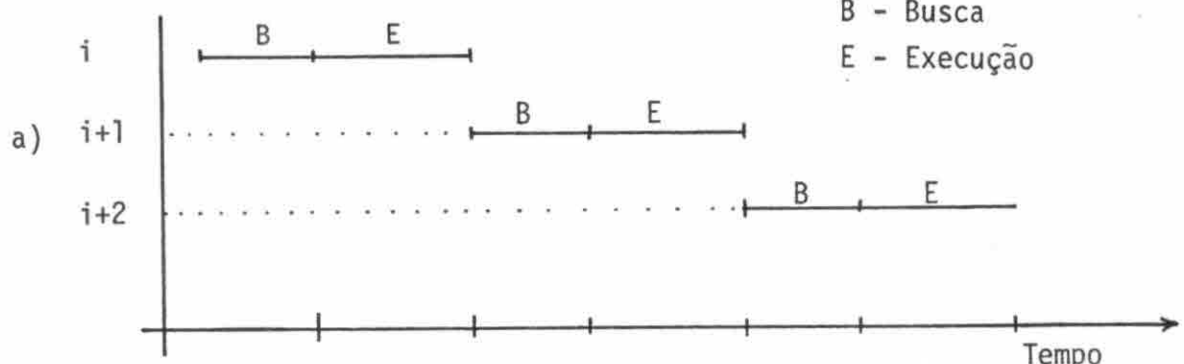


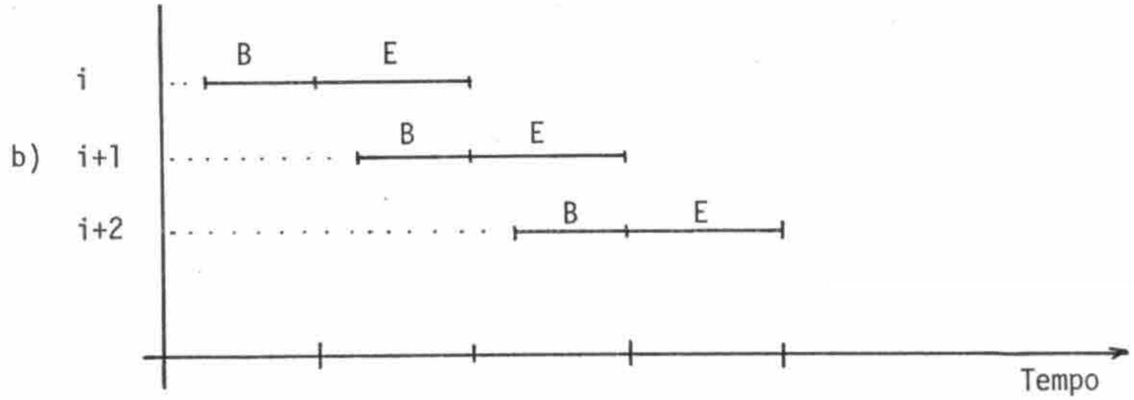
Fig. II.3 - Tipos de estruturas de memória de controle.

Outra característica de uma microinstrução diz respeito ao seu grau de codificação. Num projeto mais simples, cada bit da microinstrução representa ou não a ativação de um microoperação. Denomina-se este tipo de microinstrução de não-codificada, relembrando a proposta original de Wilkes. No entanto, certas microoperações podem ser combinadas em campos, de forma que se tornem codificadas em um pequeno grupo de bits, de modo que, para cada combinação de código,seja executada uma única microoperação, a cada pulso de relógio. É comum agrupar campos codificados com a finalidade de obter uma palavra de controle de tamanho menor.

Com o intuito de aumentar a velocidade do sistema, usualmente é utilizado um registro "pipeline", que permite a busca de uma microinstrução ("fetch"), enquanto se processa a execução da microinstrução anterior.

A Figura II.4 apresenta os fluxos de microinstruções no tempo, com ou sem o registro de "pipeline". A Figura II.4 (a)apresenta a implementação série, onde a saída da memória de controle atua no sistema, sem o uso desse registro de "pipeline". A Figura II.4 (b) e (c), utiliza-se este registro, possibilitando realizar a busca da próxima microinstrução e a execução daquela contida no registro de "pipeline". Nos casos de desvios condicionais, onde a execução da próxima microinstrução depende de condições resultantes da execução da microinstrução corrente, a busca poderá ser retardada ou não. Há casos de sistemas em que o próximo endereço é estabelecido estatisticamente; caso haja acerto, a busca e a execução são realizadas em paralelo como no caso (b); caso ocorra a condição menos provável, a busca é realizada com o endereço correto, retardando, assim, a execução da microinstrução, como mostra a Figura II.4 (c).





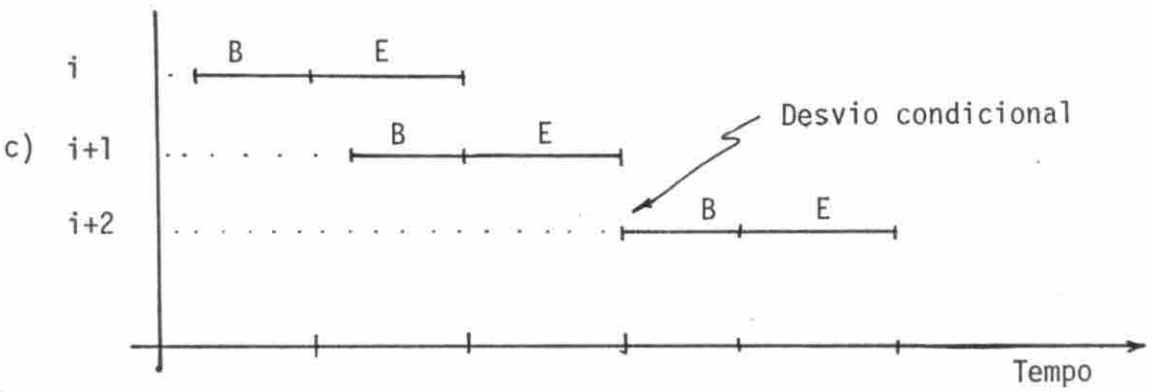


Fig. II.4 - Execução de uma microinstrução.

- (a) implementação série (b) implementação paralela ("pipelined") e (c) implementação paralela com ocorrência de desvio condicional.

Existem, naturalmente, outras considerações a serem feitas, como por exemplo, a sincronização e as fases do relógio. Porém, não se pretende esgotar o assunto de microprogramação, mas, sim, fornecer subsídios para uma linguagem que gere o conteúdo da memória de controle, aliviando o microprogramador de trabalho enfadonho. Maiores detalhes poderão ser encontrados em Agrawala e Rauscher (1976), em Husson (1970) e em Alexandridis (1978).

2.3 - LINGUAGENS DE MICROPROGRAMAÇÃO

As linguagens de microprogramação visam gerar microcódigos a partir de uma linguagem fonte, na tentativa de eliminar erros de codificação, bem como, fornecer uma forma de documentação do microprograma para futuras verificações, melhorias ou modificações.

O desenvolvimento de linguagens de microprogramação recebeu forte influência das linguagens tradicionais de programação, principalmente no caso de microprogramação vertical, onde as microinstruções se assemelham aos códigos de máquina.

Desta forma, de acordo com Agrawala e Rauscher (1976), as linguagens de microprogramação podem ser classificadas de maneira análoga à existente em linguagens de programação, como ilustrada na Figura II.5.

É interessante observar que as linguagens de microprogramação oferecem recursos variados que dificultam a sua classificação. A definição entre os limites de uma dada classe para outra é vaga, e a quantidade dessas classes e suas atribuições variam, como se verificou nas publicações disponíveis.

As linguagens de microcódigo correspondem ao nível mais baixo das linguagens de programação, que são representadas pela sequência de bits agrupadas em microinstruções, análogas aos códigos de máquina (linguagem de máquina).

	Alto nível(independente de máquina)			Baixo nível (dependente de máquina)			
Linguagens de programação	Orientado para problema (ECAP,COGO)	Orientado para procedimento (PL/I, FORTRAN, COBOL)		Macro-Assembly		Assembly	Máquina
Linguagens de microprogramação	Orientado para problema	Orientado para procedimento		Transferência de Registros com MACRO	Transferência de Registros	Assembly e Fluxograma	Microcódigo
		Independente de máquina	Dependente de máquina				
	Alto nível			Baixo nível			

Fig. II.5 - Relacionamento entre as linguagens de microprogramação e programação.

As linguagens de microprogramação do tipo "assembly", também conhecidas como "meta-assembly", possibilitam a utilização de mnemônicos ou formas simbólicas semelhantes às linguagens de programação do tipo "assembly". Estas linguagens são utilizadas pela maioria dos computadores microprogramados, como por exemplo no Digital Scientific META 4 (Figura II.6)

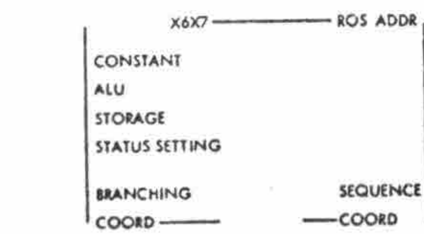
```
* META 4 MICROPROGRAM TO PERFORM A BLOCK MOVE INSTRUCTION
*  MOVE N WORDS OF MEMORY FROM ONE LOCATION TO ANOTHER.
*   OVERLAPPING FIELDS CAUSE UNDEFINED RESULT.
*   REGISTER USAGE:
*        SP REGISTER - ADDRESS OF SOURCE FIELD, UPDATED AS WE GO ALONG.
*        DP REGISTER - ADDRESS OF DESTINATION FIELD, ALSO UPDATED.
*        L REGISTER - LENGTH OF SOURCE & DESTINATION FIELDS, = 0 ON RETURN
*
MOVE BRZ  L         RET    W      IF L IS ZERO, JUST RETURN.
LOOP MOVE SP MA            MR     READ A WORD FROM SOURCE FIELD,
     ADDI SP SP     1             AND INCREMENT SP
     MOVE DP MA                   SELECT CORRESPONDING WORD IN DESTINATION
     ADDI DP DP     1              FIELD, AND INCREMENT DP.
     MOVE MD MD            PZ,MW  WRITE SOURCE WORD INTO DESTINATION
     SUBI L  L      1             DECREMENT LENGTH,
     BNZ  L         LOOP   W       AND DO ANOTHER IF NOT EXHAUSTED.
RET  RETURN
```

Fig. II.6 - Exemplo de linguagem de microprogramação "assembly", utilizada no Computador Digital Scientific META 4.

FONTE: Agrawala e Rauscher (1976), p.90.

Nas linguagens do tipo fluxograma ("flowchart"), utilizam-se símbolos e disposições gráficas semelhantes aos empregados em um fluxograma. Elas são particularmente empregadas em vários modelos IBM/360 que utilizam microprogramação horizontal. Cada microinstrução é representada por uma figura retangular, na qual são definidos os conteúdos dos vários campos, como mostra a Figura II.7 (a) e (b).

As linguagens de transferência de registros (RTL) utilizam formato semelhante ao de um comando de atribuição, de modo a indicar a transferência de conteúdo de um registro para outro. Estas admitem, ainda, operações algébricas simples e uma correspondência unívoca entre os comandos da linguagem e as microinstruções geradas. Essas linguagens são interessantes nos casos de microprogramação horizontal, onde o número elevado de campos da microinstrução dificulta o emprego de linguagem do tipo "assembly" ou mesmo, as do tipo fluxograma.



Format of Symbolic Representation

01 — 115D
1101
R + KH → DC
WRITE
HZ → S4, LZ → S5
G4, G5
C4
C4
CD

Fig. II.7 (a) - Representação simbólica de uma microinstrução do computador IBM/360,modelo 30, utilizando uma linguagem do tipo fluxograma.

FONTE: Agrawala e Rauscher (1976), p.92.

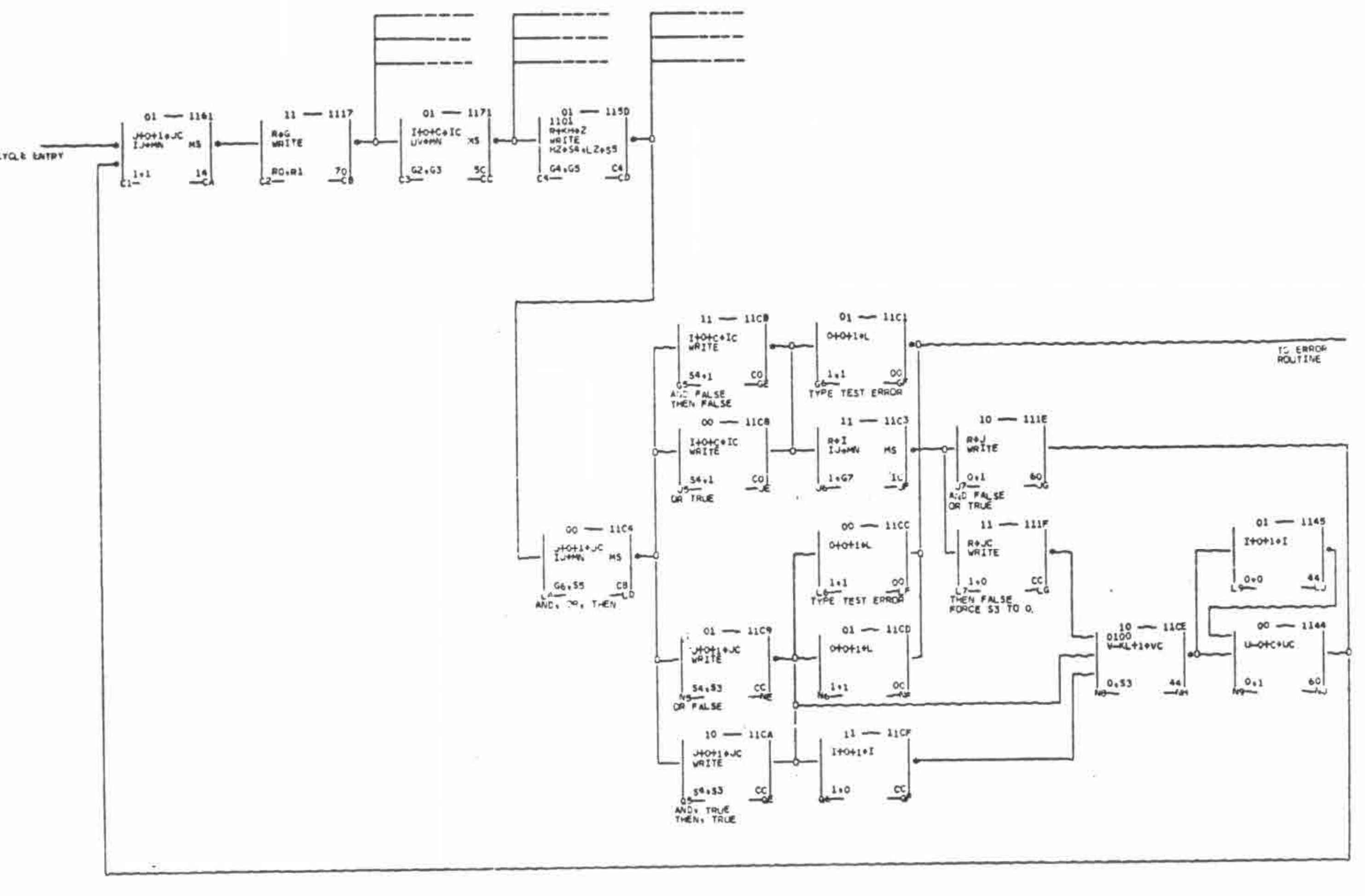


Fig. II.7 (b) - Exemplo de linguagem tipo fluxograma (IBM/360 modelo 30).

FONTE: Agrawala e Rauscher (1976), p.93.

No caso de adicionar recursos de macro parametrizados,há um aumento substancial na capacidade dessas linguagens. A Figura II.8 apresenta a linguagem ANIMIL, como exemplo de linguagem do tipo RTL.

As linguagens orientadas para procedimentos dependentes de máquina são aquelas que oferecem recursos do tipo "IF ... THEN ... ELSE ... ", "WHILE ... DO ...", "BEGIN ... END", expressões aritméticas, lógicas, variáveis, etc. Porém, características funcionais do processador refletem-se na linguagem. Nessas linguagens, a correspondência de um para um, entre os comandos e as microinstruções, não se mantém. Normalmente, o compilador gera, a partir de subcomandos fornecidos sequencialmente, a configuração de campos simultâneos em uma ou mais microinstruções. A eficiência de geração de códigos fica comprometida. A Figura II.9 apresenta a linguagem PUMPKIN, como exemplo de linguagem orientada para procedimento dependente de máquina.

As linguagens orientadas para procedimento independentes de máquina são aquelas semelhantes às linguagens de programação Algol, Fortran, Cobol, PL/I e outras. Nessas linguagens, as capacidades e características do processador podem ser descritas através das declarações, o que permitiria a utilização de microprogramas de um dado computador em outro. A geração de códigos, relativamente eficientes, também é uma das dificuldades de implementação dos compiladores dessas linguagens.

Em termos práticos, concluiu-se que quanto maior o número de campos simultâneos em uma microinstrução (tipo horizontal),maior é a dificuldade de geração de códigos relativamente eficientes, a partir de linguagens de alto nível, sendo interessante, para estes casos, o emprego de linguagem a nível de transferência de registros, como adotada na linguagem LMP proposta.

À medida que a microinstrução se torna vertical, o emprego de linguagens de alto nível sofre menos restrições e a implementação se torna mais simples, evitando-se, assim, algoritmos complexos de otimização de códigos.

```
ADDRESS          MICROINSTRUCTION
-------          ----------------
0 "SPAU MICROPROGRAM EXAMPLE - ADD COMPLEX NUMBERS
0      ASSUME BSM 0 CONTAINS N PAIRS OF COMPLEX NUMBERS,
0       EACH COMPLEX NUMBER IS STORED IN ONE 32 BIT BSM WORD.
0       SUM SUCCESSIVE PAIRS OF NUMBERS AND STORE THEM IN BSM 1.
0       ASSUME SUMS DO NOT OVERFLOW."
0
0
0 "INITIALIZE"
0
0      A5: BARA-W(1); $  "BUFFER ADDRESS REGISTER A POINTS TO INPUT
1                          DATA, WHOSE ADDRESS IS STORED IN W(1)"
1      A6: BARB-W(2); $ "BUFFER ADDRESS REGISTER B POINTS TO OUTPUT
2                          DATA, WHOSE ADDRESS IS STORED IN W(2)"
2      A5: INCA-W(3);  A6: INCB-W(3); $
3          "SET ADDRESS INCREMENT REGISTERS TO 1, WHICH IS IN W(3)"
3      CTRI - W(4); $ "SET CTRI TO THE NUMBER OF PAIRS
4                        OF COMPLEX NUMBERS TO BE ADDED,
4                        WHICH IS STORED IN W(4)"
4
4
4 "FILL UP PIPE".
4
4      X(1)-BUFA;  A5: BARA-BARA+INCA;  $
5          "READ THE FIRST COMPLEX NUMBER INTO X(1)
5           INCREMENT THE POINTER TO THE INPUT DATA"
5
5      Y(1)-BUFA;  A5: BARA-BARA+INCA;  DECI;  $
6          "READ THE SECOND COMPLEX NUMBER INTO Y(1)
6           INCREMENT THE POINTER TO THE INPUT DATA
6           DECREMENT COUNTER AS HAVE JUST READ ONE PAIR OF NUMBERS"
6
6
6 "LOOP TO SUM NUMBERS AND PRINT THEM OUT"
6
6 LOOP# A1: R1-X1(1)+Y1(1);  A2: R2-X2(1)+Y2(1);
6       X(1)-BUFA;  A5: BARA-BARA+INCA;  $
7           "ADD REAL AND IMAGINARY PARTS OF THE TWO COMPLEX NUMBERS
7            READ THE FIRST NUMBER OF PAIR INTO X(1)
7            INCREMENT POINTER TO INPUT DATA"
7
7     BUFB-R1R2;  A6: BARB-BARB+INCB;  Y(1)-BUFA;
7            A5: BARA-BARA+INCA;
7     DECI; IF NOT CTRI THEN GO TO LOOP; $
8           "WRITE COMPLEX SUM INTO BUFFER 1
8            INCREMENT POINTER TO OUTPUT DATA
8            READ SECOND NUMBER OF PAIR INTO Y(1)
8            INCREMENT POINTER TO INPUT DATA
8            DECREMENT COUNTER I AND IF ALL DATA HAS NOT BEEN READ
8                THEN CONTINUE LOOPING"
8
8
8  "FLUSH PIPE"
8
8     A1: R1-X1(1)+Y1(1); A2: R2-X2(1)+Y2(1); $
9           "ADD REAL AND IMAGINARY PARTS OF THE TWO COMPLEX NUMBERS"
9
9
9      BUFB-R1R2; $$  "WRITE THE LAST SUM INTO BUFFER 1"
```

Fig. II.8 - Microprograma na linguagem ANIMIL.

FONTE: Agrawala e Rauscher (1976), p.94-95.

```
PROC PARSERX(RX);
        DCL 1 RX DWORD, "32 BIT RX INSTRUCTION"
             2 OPCODE BIT(8), "OPERATION CODE"
             2 REG1 BIT (4), "SOURCE/TARGET REGISTER"
             2 STORAGE_ADDRESS BIT (20),
                  3 INDEX BIT(4), "INDEX REG"
                  3 BASE BIT (4), "BASE REG"
                  3 DISPLACEMENT BIT(12);

        DCL PARSED_OPCODE WORD IN LSB(8); "LOCAL STORE B, LOC 8"
        DCL REGNO WORD IN LSB (9);
        DCL EFFEC_ADDRH WORD IN LSB(10);
        DCL EFFEC_ADDRL WORD IN LSB(11);

        "SIMULATED GENERAL PURPOSE REGISTERS"
        DCL 1 GPR (0:15) DWORD IN BSM1(100), "BUFFER STORE"
             2 HIGH WORD,
             2 LOW WORD;

    "THIS ROUTINE IS PASSED A /360 RX TYPE INSTRUCTION AS A
PARAMETER, STORES THE OPCODE AND REGISTER  OPERAND IN LOCAL STORE, AND
CALCULATES THE EFFECTIVE ADDRESS OF THE STORAGE OPERAND (BASE REGISTER
+ INDEX REGISTER + 12 BIT DISPLACEMENT). IT THEN CALLS EXECRX TO
EXECUTE THE INSTRUCTION."

        PARSED_OPCODE <- OPCODE;  "SET OPCODE IN LSB"
        REGNO <- REG1; "SET REG # IN LSB"
        EFFEC_ADDRH <- 0; "ZERO HIGH 16 BITS OF EA"

"CARRYOUT RETURNS THE VALUE OF THE ADDER CARRY WHEN EVALUATING THE
EXPRESSION USED AS AN ARGUMENT.  AS A SIDE EFFECT, THE 16 BIT RESULT
MAY BE ASSIGNED WITHIN THE FUNCTION."

        EFFEC_ADDRH <- CARRYOUT(EFFEC_ADDRL <-
             DISPLACEMENT+LOW(BASE)) + HIGH(BASE);
        IF INDEX ¬=0 THEN
             IF CARRYOUT(EFFEC_ADDRL <- EFFEC_ADDRL+LOW(INDEX))
                  THEN EFFEC_ADDRH <- EFFEC_ADDRH+1;

        "ZERO HIGH BYTE OF SUM (24 BIT ADDRESSING)"
        EFFEC_ADDRH <- (EFFEC_ADDRH+HIGH(INDEX)) & X'00FF';

        CALL EXECRX; "EXECUTE THE RX INSTRUCTION"

END PARSERX;                                "C. BERGMAN, AUG 73"
```

Fig. II.9 - Microprograma em PUMPKIN.

FONTE: Agrawala e Rauscher (1976), p.97.

CAPÍTULO III

CARACTERÍSTICAS GERAIS DA LINGUAGEM LMP

Apresentam-se neste capítulo certas características da linguagem que às vezes não transparecem na descrição mais formal, como a realizada no Capítulo IV. Algumas considerações são efetuadas quanto ao seu projeto.

A linguagem LMP tem como objetivo básico a introdução de facilidades para a geração e manuseio de microcódigos de sistemas microprogramados, como meio de projeto auxiliado por computador("Computer Aided Design-CAD").Em função de interesses de projeto no INPE, procurou-se desenvolver recursos de geração de sistemas com microprogramação horizontal, como por exemplo, uma unidade aritmética de ponto flutuante ou uma unidade de processamento de imagens e sinais, onde o fator mais importante é o tempo.

Além da geração de microcódigos, deseja-se obter, com esta linguagem, uma forma de padronização dos microprogramas para facilitar a documentação dos mesmos, e, principalmente, para uso na manutenção desses sistemas.

Para alcançar a meta de boa documentação, é interessante que a linguagem seja de alto nível; porém, a necessidade de geração de microcódigos, de forma eficiente, restringe o seu emprego, principalmente no caso de microinstrução do tipo horizontal.

A linguagem LMP resultou de um compromisso com as necessidades citadas; situa-se,conforme a classificação apresentada no Capítulo II, entre as linguagens consideradas de baixo nível. Mais precisamente, é do tipo RTL ("Register Transfer Language"), acrescida de recursos de macro.

Procurou-se ainda, de certa forma, obter uma melhor estruturação da linguagem através da utilização de blocos, de forma semelhante àqueles empregados em linguagens do tipo Algol. Assim sendo, um bloco delimitado por "BEGIN" e "END" poderia representar uma região do microprograma, capaz de gerar microcódigos que desempenham alguma função ou tarefa lógica. A idéia se estende também para o caso de declaração de sub-rotina, constituída por um conjunto de microinstruções; neste conjunto, a última microinstrução deve ser implementada a nível de microoperação, com o retorno ao endereço seguinte à microinstrução que realizou a chamada a esta sub-rotina.

Uma vez que o sistema para o qual se geram os microcódigos pode variar, a linguagem tem caráter descritivo. Esta descrição de circuito ("hardware") compreende a declaração de memória de controle e a estruturação da palavra de controle. Admitiu-se, para tanto, que uma microinstrução ocupa uma palavra de controle (veja estruturas de memória de controle apresentadas no Capítulo II).

3.1 - ESTRUTURA DA LINGUAGEM LMP

A estrutura adotada para o microprograma fonte é apresentada na Figura III.1, onde foi adotada a seguinte convenção:

DECLMEM - Declaração de Memória de Controle
DECL - Declaração
CMD - Comando
MCMD - Microcomando
OPPC - Opção de Pós-processamento

Na estrutura apresentada na Figura III.1, é interessante ressaltar que num dado bloco, as declarações devem anteceder aos comandos. No bloco principal deve ser fornecido, como primeira declaração,a de memória de controle.

```
BEGIN
    %
    % MICROPROGRAMA FONTE
    % ******************
    %
    % DECLARAÇÃO DA MEMÓRIA DE CONTROLE
    %
    DECLMEM;
    %
    % DEMAIS DECLARAÇÕES
    %
    DECL; DECL;
    :
    DECL;
    %
    % COMANDOS  E MICROCOMANDOS
    %
    CMD; CMD; ... CMD;
    MCMD, MCMD, MCMD, ... MCMD;
    BEGIN
        DECL; DECL; ...
        CMD; ... CMD
    END
    CMD; ... CMD;
    %
    % OPÇÕES DE PÓS-PROCESSAMENTO
    %
    OPPC; OPPC;
    :
    OPPC
END
```

Fig. III.1 - Estrutura de microprograma em LMP.

O paralelismo das microoperações que controlam recursos simultâneos do processador é representado pela associação do conceito de microcomando com o de microoperação, e a de comando com a microinstrução, exceto no caso de bloco.

A linguagem pode ser encarada, também, como sendo basicamente um instrumento que serve para transferir configurações binárias a registros de tamanhos variáveis (campos). Cada uma dessas transferências deve corresponder a uma microoperação executável pelos recursos funcionais do processador, que deve ser microprogramado. Um dado conjunto de transferências (cada uma delas é um microcomando) pode ser agrupado e delimitado por ";", gerando uma microinstrução composta de microoperações executáveis simultaneamente.

Os recursos de macro inseridos na linguagem LMP podem ser do tipo:

a) inserção de texto parametrizada ou não a nível de microprograma fonte, dado pela declaração de definição e sua invocação;

b) inserção de microinstrução a nível de microcódigo, dada pela declaração de microinstrução e sua invocação através do microcomando de concatenação.

3.2 - CONSTANTES E EXPRESSÕES

As constantes podem representar tanto uma configuração binária como um número decimal equivalente. Elas são utilizadas em vários pontos do microprograma fonte. Estas constantes podem ser expressas em várias bases:

- binária;
- octal;
- decimal;
- hexadecimal.

A diferença básica entre a configuração binária e o número é que a primeira representa uma configuração de bits com tamanho definido, enquanto a segunda é encarada como um valor numérico que tem uma representação equivalente em bits, mas cujo tamanho não é definido de forma explícita (Seção 4.1.2 - Constantes).

As expressões foram inseridas na linguagem de forma a oferecer facilidades quanto ao manuseio de endereços de microprograma via rótulos, identificadores de sub-rotinas, identificadores de campo, ou mesmo, de valores numéricos.

O cálculo de expressão é efetuado durante a compilação, utilizando o próprio ALGOL.

3.3 - DECLARAÇÕES

O tradutor LMP, como um recurso de CAD ("Computer Aided Design") para microprogramação, é uma ferramenta de uso geral, que visa a geração de microcódigos de diversos sistemas microprogramados. Essa diversidade de aplicações faz com que seja necessária a descrição de certas características do circuito a ser microprogramado.

Assim sendo, as declarações se destinam à descrição dessas características, bem como a das entidades que descrevem as facilidades de programação inseridas na linguagem.

A descrição da memória de controle é feita pela declaração de memória de controle, onde implicitamente se define o tamanho da palavra de controle. No caso, uma palavra de controle corresponde a uma microinstrução.

A nomenclatura dos campos de uma microinstrução é feita através da declaração de campo. E a declaração de microinstrução permite a associação de um identificador a uma dada microinstrução.

As demais declarações da linguagem LMP procuram oferecer as facilidades encontradas usualmente nas linguagens de programação. Essas facilidades se referem às declarações de rótulos, de sub-rotinas e de definições.

No caso da declaração de sub-rotina, é necessária a geração, no final do corpo da mesma, de uma microoperação que, quando executada, realize o retorno ao endereço seguinte à invocação dessa mesma sub-rotina.

3.4 - COMANDOS E MICROCOMANDOS

Os comandos são aqueles que na realidade geram os micro códigos, fazendo uso das entidades declaradas. A cada comando corresponde a geração de uma palavra de controle.

O termo microcomando foi introduzido com a finalidade de associar, a nível de linguagem, uma ou mais microoperações. Desta forma, um dado comando é constituído por um conjunto de microcomandos, de maneira análoga a uma microinstrução que é constituída pelo conjunto de microoperações simultâneas. Cada microoperação atua sobre um recurso específico do sistema a ser microprogramado.

Na realidade, cada microcomando provê recursos de atribuição de valores a campos da palavra de controle.

3.5 - OPÇÕES DE PÓS-PROCESSAMENTO

As opções de pós-processamento provêem facilidades quanto ao manuseio e à documentação dos microcódigos gerados.

Esta fase atua sobre a memória de controle gerada numa forma semifinal (dita forma intermediária) pela fase anterior. Essas duas fases são descritas no Capítulo V, onde são feitas as considerações de implementação de um tradutor LMP.

Os resultados da fase de opções de pós-processamento são constituídos dos microcódigos na forma final, colocadas em diversos meios de armazenamento, principalmente em fitas magnéticas para conexão ao EMMAC (Emulador de Memórias de Microcontrole Auxiliadas por Computador).

3.6 - OPÇÕES DE CONTROLE

As opções de controle permitem o gerenciamento da memória de controle quanto à alocação dos microcódigos gerados. Essas opções permitem, ainda, o controle de listagem do microprograma fonte.

A execução dessas opções foram atribuídas ao Analisador Léxico, que executa a chamada de rotinas de tratamento de cada uma das opções. A partir daí, pode-se dizer que há possibilidade dessas aparecerem em qualquer ponto do microprograma fonte (Seção 4.1.7 - Opções de Controle).

CAPÍTULO IV

DEFINIÇÃO DA LINGUAGEM LMP

A linguagem LMP é definida em termos de sua sintaxe, através da notação BNF ("Backus Naur Form"), que possibilita a definição de uma linguagem de maneira formal e clara.

As produções da linguagem LMP foram estruturadas na forma mais explícita possível, acrescida de comentários, a fim de facilitar o entendimento.

4.1 - COMPONENTES DA LINGUAGEM

Sintaxe:

<componentes da linguagem> :: = <símbolos básicos>|
<palavras reservadas>|
<constantes>|
<comentários>|
<identificador>|
<microprograma>|
<invocação de definição>|
<opções de controle>

Cada uma dessas entidades é apresentada com maiores detalhes nas próximas seções.

4.1.1 - SÍMBOLOS BÁSICOS E PALAVRAS RESERVADAS

Sintaxe:

<símbolos básicos> :: = <letra>|<dígito>|<ponto>|
<caracteres especiais>

<letra> :: = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|X|Y|W|Z

<dígito> :: = 0|1|2|3|4|5|6|7|8|9

<ponto> :: = .

<caracteres especiais> :: = ; | <espaço> | , | + | - | / | * |
: | = | $ | " | % | (|) | & | ← |
' | [|] | # | @

<espaço> :: = <espaço simples>|<espaço><espaço simples>

<espaço simples> :: = {uma posição em branco}

<palavras reservadas> :: = BEGIN | END | MEMORY | LABEL | FIELD |
FORMAT | DEFINE | MICRO | PROCEDURE |
COMMENT | $RESERVE | $ORIGIN | $SETLIST |
$RESETLIST | SOURCE | INTER | BIN | OCT |
DEC | HEX | TABSIMB | MAP | INVERT | ALL |
DONTCARE | EQL | FOR | FMAG

Para o tradutor, a linguagem é encarada como uma sequência de símbolos básicos e de palavras reservadas.

4.1.2 - CONSTANTES

Sintaxe:

<constantes> :: = <configuração de bits>|
<número>

```
<configuração de bits> :: = <configuração binária>|
                             <configuração octal>|
                             <configuração decimal>|
                             <configuração hexadecimal>

<configuração binária> :: = "<cordão binária>"B
<configuração octal> :: = " <cordão octal>"Q
<configuração decimal> :: = "<cordão decimal>"D
<configuração hexadecimal> :: = "<cordão hexadecimal>"H
<número>  :: = <número binário>|
               <número octal>|
               <número decimal>|
               <número hexadecimal>

<número binário> :: = <cordão binário>B
<número octal> :: = <cordão octal>Q
<número decimal> :: = <cordão decimal>|
                      <cordão decimal>D

<número hexadecimal> :: = <dígito><cordão hexadecimal>H|
                          0<letra hexadecimal>< cordão   hexadeci-
                           mal>H

<cordão binário> :: = <dígito binário>|
                      <cordão binário><dígito binário>

<cordão octal> :: = <dígito octal>|
                    <cordão octal><dígito octal>

<cordão decimal> :: = <dígito>|
                      <cordão decimal><dígito>

<cordão hexadecimal> :: = <caractere hexadecimal>|
                          <cordão hexadecimal><caractere  hexadeci-
                          mal>

<dígito binário> :: = 0|1

<dígito octal> :: = 0|1|2|3|4|5|6|7
```

<letra hexadecimal> :: = A|B|C|D|E|F

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

As constantes podem ser expressas nas bases: binária, caracterizada por B; octal, caracterizada por Q; decimal, caracterizada por D; e hexadecimal, caracterizada por H. Estas várias bases foram inseridas na linguagem LMP, com o intuito de oferecer flexibilidade e clareza na documentação do microprograma.

As constantes do tipo configuração de bits são empregadas na declaração de microinstrução (Seção 4.2.5) e representam implicitamente um conjunto de bits, cujo tamanho é função da base utilizada.

A Tabela IV.1 apresenta alguns exemplos de constantes do tipo configuração de bits e sua configuração binária equivalente.

Vale ressaltar que, no caso de configuração decimal, é utilizada a representação BCD ("Binary Coded Decimal").

As contantes do tipo número são utilizadas com a finalidade de fornecer o valor correspondente à base associada.

Exemplos de números válidos:

123, 01101111B (=111_{10}), 377Q (=255_{10}),

0A5H (=165_{10}), 23H (=35_{10}), 1023D (=1023_{10})

Exemplos de números inválidos:

A5H, 130112B, 139Q, 01A5

TABELA IV.1

EXEMPLO DE UTILIZAÇÃO DE CONSTANTES DO TIPO CONFIGURAÇÃO DE BITS

CONTANTES	BASE	CONFIGURAÇÃO BINÁRIA EQUIVALENTE	TAMANHO EM BITS
"01101111"B	BINÁRIA	01101111	8
"01"B	BINÁRIA	01	2
"147"Q	OCTAL	001100111	9
"120"D	DECIMAL	000100100000	12
"C3"H	HEXADECIMAL	11000011	8
"02"H	HEXADECIMAL	00000010	8

O tamanho máximo de uma constante é função do seu tipo e da capacidade do tradutor implementado. No caso da implementação efetuada, a constante poderá ser representada por até 48 bits, que correspondem a uma palavra do B-6800.

4.1.3 - COMENTÁRIOS

Sintaxe:

<comentários> :: = <comentários tipo COMMENT>|
<comentário tipo "%">

<comentário tipo COMMENT> :: = COMMENT {qualquer sequência de caracteres sem ";"}

<comentário tipo "%"> :: = %{qualquer sequência de caracteres até o final do registro lógico do microprograma fonte}

Os comentários assumem importância fundamental quanto à documentação do microprograma. A utilização deste recurso é recomendado com muita ênfase, no sentido de facilitar a manutenção dos microprogramas e melhorar a inteligibilidade dos mesmos. Os comentários podem ser colocados em qualquer posição entre as várias entidades do microprograma, sejam nas declarações, sejam nos comandos, ou mesmo dentro de um comando, no caso do comentário do tipo "%".

A Figura IV.1 apresenta um exemplo de utilização de comentários em LMP.

```
BEGIN

    COMMENT: - MICROPROGRAMA DE TESTE 01 - 06.11.80
               UNIDADE DE CONTROLE DO MODEM 4800 BPS
               PROJETO MODEM - GSD/DEE - INPE;

    %
    % DECLARAÇÃO DA MEMÓRIA DE CONTROLE
    %
    MEMORY WC [0:511, 0:71];
    %
    % ROTINAS DE CONTROLE
    %
    PROCEDURE ALU;
    BEGIN
    COMMENT: - ROTINA DE CONTROLE DA UNIDADE LÓGICA E ARITMÉTICA;
    .
    :
    END;
    .
    :
    %
    % DEFINIÇÃO DE CAMPOS DA PALAVRA DE CONTROLE
    %
```

Fig. IV.1 - Exemplo de utilização de comentários em LMP.

(continua)

```
FIELD OPCODE = WC [ 0:5 ], % CÕDIGO DE OPERAÇÃO
      A = WC [ 6:9 ], % ENDEREÇO A
      B = WC [ 10:13 ], % ENDEREÇO B
      FLAG = WC [ 14:15 ], % FLAG DE CONDIÇÃO
      ADR = WC [ 16:24 ] ; % ENDEREÇO DE MEMÕRIA
.
:
FORMAT TIPO.01 (OPCODE, A,ALU,ADR), % FORMATO TIPO.01
       TIPO.02 (OPCODE, B,FLAG,IR); % FORMATO TIPO.02
.
:
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ***** FASE DE PÕS-PROCESSAMENTO
%
INVERT ALL;
DONTCARE EQL 1 FOR ALL;
LIST ALL;
END % FIM DO MICROPROGRAMA EXEMPLO
```

Fig. IV.1 - Conclusão

4.1.4 - IDENTIFICADORES

Sintaxe:

<identificador> :: = <letra>|
<identificador><letra>|
<identificador><dígito>|
<identificador><ponto>

Os identificadores não têm significados intrínseco como as palavras reservadas, que são utilizadas como identificação das várias entidades declaradas na linguagem; uma vez declaradas,suas validades se estendem ao bloco corrente e aos mais internos. Num mesmo bloco, não é permitido utilizar um identificador já declarado para identificar uma outra entidade.

Exemplos de identificadores válidos:

CONTROLE, MUX1, MUX2, L1, L2, MUX.01,
MUX.02, ALU...., SOMA

Exemplos de identificadores inválidos:

125, $MUX, %CONTR, 3MUX

4.1.5 - MICROPROGRAMA

Sintaxe:

```
<microprograma> :: = BEGIN
                     <declaração de memória de controle>;
                     <corpo do microprograma>;
                     <lista de opções de pós-processamento>
                      END

<corpo do microprograma> :: = <lista de comandos>|
                              <lista de declarações>;<lista de co
                               mandos>

<lista de declarações> :: = <declaração>
                            <lista de declarações>;<declaração>
```

<lista de comando> :: = <comando>|
<lista de comandos>;<comando>

<lista de opções de pós-processamento> :: = <opção de pós-processamento>|
<lista de opções de pós-processamento>

Com respeito à sintaxe do microprograma vale ressaltar:

a) Todo microprograma é delimitado pelas palavras reservadas: BEGIN e END.

b) Em todo microprograma deve-se inicialmente declarar a memória de controle, onde deverá residir o microprograma objeto ou os microcódigos gerados a partir do microprograma fonte. Com esta declaração ficam determinadas as dimensões da memória de controle e o tamanho da microinstrução ou palavra de controle.

c) Todo identificador referenciado num comando deverá ter sido declarado anteriormente.

d) Um microprograma é basicamente constituído por três etapas distintas. A primeira é dada pela lista de declarações que descrevem as características ligadas ao circuito em desenvolvimento e algumas facilidades de programação. A segunda etapa compreende a a sequência de comandos que geram, em última análise, os microcódigos numa forma intermediária. E na etapa final, são fornecidos as opções de pós-processamento, que permitem gerar os microcódigos na forma final.

Os elementos de uma etapa do microprograma não podem ser misturados com os de outra; caso isto aconteça, o tradutor LMP deverá fornecer mensagem de erro correspondente.

4.1.6 - INVOCAÇÃO DE DEFINIÇÃO

A invocação de definição faz uso de entidades ainda não definidas, mas que serão apresentadas nas próximas seções. Assim, a invocação de definição será apresentada na Seção 4.2.4, quando definida a declaração de definição.

4.1.7 - OPÇÕES DE CONTROLE

Sintaxe:

```
<opções de controle> :: = $RESERVE   <expressão>|
                          $ORIGIN    <expressão>|
                          $SETLIST   <opção de listagem>|
                          $RESETLIST <opção de listagem>

<opção de listagem> :: =  SOURCE|
                          INTER |
                          TABSIMB|
                         <formato de saída>FIELD

<formato de saída> ::  =  BIN | OCT | DEC | HEX
```

As opções de controle permitem o gerenciamento da memória de controle quanto à alocação dos microcódigos gerados. Essas opções possibilitam, ainda, o controle de listagem do microprograma fonte e dos códigos gerados na forma intermediária (isto é, apresenta o estado "don't care" como sendo o caractere "X"), além de tabelas auxiliares com vista à documentação e facilidades de depuração.

O controle de alocação dos microcódigos gerados é feita pelas opções $RESERVE e $ORIGIN.

A opção $RESERVE incrementa o contador de alocação de microprograma (MILC), pelo valor fornecido na expressão; isto é, reserva tantas posições de memória quanto forem estabelecidas pela expressão dada.

Como exemplo de utilização, tem-se:

$RESERVE 5

$RESEVE 5*2+4/2 equivale a $RESERVE 12

$RESERVE 20Q

A opção $ORIGIN carrega o contador de alocação de microprograma com o valor fornecido na expressão. Desta forma, a opção $ORIGIN se assemelha à instrução ORG em uma linguagem "assembly".

O caractere "$" é utilizado para representar o contador de alocação de microprograma, e o mesmo pode ser empregado em uma expressão.

Exemplificando, pode-se ter:

$ORIGIN 0

$ORIGIN $+5

$ORIGIN 256

$ORIGIN 1000B

As opções $SETLIST e $RESETLIST atuam na habilitação ou não de listagem de: microprograma fonte, no caso da opção de listagem SOURCE; microcódigos com estados "don't care", no caso de INTER;tabela de símbolos, no caso de TABSIMB; e campos no formato especificado, no caso de FIELD.

A Tabela IV.2 apresenta, de forma condensada, essas opções de listagem e a ação tomada por omissão ("default") das mesmas.

Cada uma dessas opções deverá ser especificada a partir da primeira posição do registro lógico. No caso de entrada por cartões, a opção deverá ser perfurada a partir da coluna 1, contendo somente uma opção de controle.

TABELA IV.2

OPÇÕES DE LISTAGEM

OPÇÃO	OMISSÃO ("DEFAULT")	AÇÃO TOMADA
SOURCE	ATIVADO	Lista o microprograma fonte
INTER	DESATIVADO	Lista a forma intermediária em binário e com estados "don't care"
TABSIMB	DESATIVADO	Lista a tabela de símbolos e as tabelas auxiliares
FIELD	DESATIVADO	Lista a forma intermediária de maneira análoga a INTER, porém agrupados em campos, em um dos seguintes formatos de saída: - BINÁRIO (BIN) - OCTAL (OCT) - DECIMAL (DEC) - HEXADECIMAL (HEX)

Exemplo de utilização das opções $SETLIST e $RESETLIST:

```
BEGIN
...
$RESETLIST SOURCE      % A listagem do microprograma é desativada
                       % até que seja novamente ativada pela opção
                       % $SETLIST SOURCE
```

```
...
$SETLIST TABSIMB
$SETLIST INTER
END
```

4.2 - DECLARAÇÕES

Sintaxe:

<declaração> : : = <declaração de campo>|
<declaração de formato>|
<declaração de definição>|
<declaração de microinstrução>|
<declaração de rótulo>|
<declaração de sub-rotina>

As declarações podem ser classificadas em dois tipos:

a) aquelas que descrevem as características do processador para o qual se está gerando os microcódigos; isto é, as declarações dos campos da microinstrução e a declaração dos formatos destas microinstruções;

b) aquelas que auxiliam na microprogramação, através de recursos da linguagem. São as declarações de definição, de microinstrução, de rótulo e de sub-rotina. Nas declarações de definição e de microinstrução, estão contidas as potencialidades da linguagem na geração de microcódigos, bem como, na documentação e na facilidade de leitura do microprograma.

É através das declarações que as entidades do programa são definidas e a validade desta identificação se estende ao bloco corrente e àqueles mais internos.

4.2.1 - DECLARAÇÃO DE MEMÓRIA DE CONTROLE

Sintaxe:

<declaração de memória de controle> :: = MEMORY <identificador de memória><especificação de memória>

<identificador de memória> :: = <identificador>

<especificação de memória> :: = [<tamanho da memória>,<comprimento da palavra>]

<tamanho da memória> :: = <par delimitador>

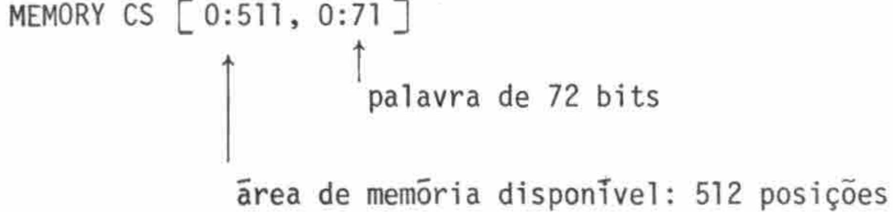
<comprimento da palavra> :: = <par delimitador>

<par delimitador> :: = <número>:<número>

Através desta declaração, associa-se um identificador à memória de controle, bem como especificam-se a área disponível e a nomenclatura dos bits da palavra de controle.

É conveniente notar que esta declaração é necessária em qualquer microprograma e deve ser a primeira declaração a constar no bloco principal.

Exemplos:



MEMORY MEMCONT [40H : 1000H, 0:31]

microinstrução com 32 bits, onde o bit 0 é o mais significativo e o bit 31 o menos significativo

área disponível a partir de 40H a 1000H

4.2.2 - DECLARAÇÃO DE CAMPO

Sintaxe:

<declaração de campo> :: = FIELD <lista de especificação de campo>

<lista de especificação de campo> :: = <especificação de campo>| <lista de especificação de campo>,<especificação de campo>

<especificação de campo> :: = <identificador de campo> = <identificador de memória de controle>[<par delimitador de campo>] | <identificador de campo> = <identificador de memória de controle>[<par delimitador de campo>](<constante>)

<identificador de campo> :: = <identificador>

<par delimitador de campo> :: = <número>:<número>

Através da declaração de campo, associa-se a cada campo da microinstrução um identificador. Basicamente, a linguagem LMP deverá fornecer meios para o preenchimento destes campos, com a configuração adequada de bits.

Como exemplo de uma declaração de campo, pode-se dar uma palavra de controle, como na Figura IV.2, pertencente a uma memória de controle identificada por CS. A declaração de campo correspondente é apresentada na Figura IV.3.

0 ... 11	12 ... 15	16 ... 20	21 ... 23	24 ... 31
ADR	MUX	ALU	TEST	VALOR

Fig. IV.2 - Exemplo de uma microinstrução e seus campos.

```
FIELD   ADR = CS [0:11]              % CAMPO DE ENDEREÇO
        MUX = CS [12:15] (0111B),    % CONTROLE DO MULTIPLEX
                                     % VALOR INICIAL = 0111B
        ALU = CS [16:20] (0),        % CONTROLE DA ALU
                                     % VALOR INICIAL IGUAL A ZERO,
                                     % ISTO É, O MESMO QUE 00000B
        TEST = CS [21:23],           % TESTE DE CONDIÇÃO
        VALOR = CS [24:31];          % CONSTANTE
```

Fig. IV.3 - Exemplo de declaração de campo correspondente à microinstrução dada na Figura IV.2.

É permitido ainda atribuir um valor inicial a um dado campo, através de uma constante entre parênteses, situada logo após a declaração correspondente.

Os campos que não tiverem sido associados com quaisquer valores, durante o corpo do microprograma, serão considerados como contendo estados "don't care". Exceção é feita no caso do campo possuir valor inicial associado que, por omissão ("DEFAULT") numa determinada microinstrução, assume o valor declarado inicialmente.

É importante notar a ordem com que se especifica o par delimitador de campo. Os valores mais significativos de cada campo são aqueles que apresentam os valores mais baixos, como convencionado na declaração de memória de controle para a descrição da palavra de controle.

A codificação de uma microinstrução, pode estar associado mais do que um formato, sendo permitido declarar vários identificadores diferentes entre si, associados a um mesmo conjunto de bits da palavra de controle. Na Figura IV.4 são apresentados os formatos de microinstrução do minicomputador HP 21MX, e na Figura IV.5, as declarações de campos correspondentes.

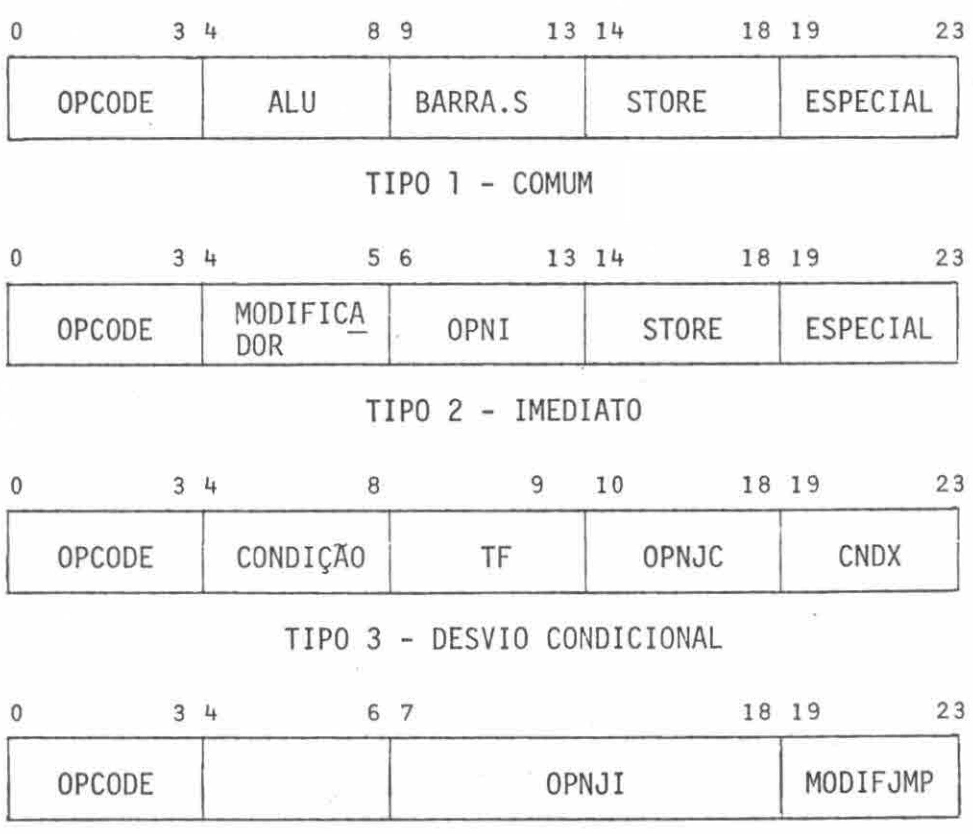


Fig. IV.4 - Formatos de microinstrução do HP21MX, baseados em Hewlett-Packard, 1977.

```
FIELD
        %
        % TIPO 1
        %
        OPCODE      = CS [0:3] ,        % OPERAÇÃO
        ALU         = CS [4:8] ,        % CONTROLE DA ALU
        BARRA.S     = CS [9:13] ,       % CONTROLE DO BARRAMENTO S
        STORE       = CS [14:18],       % ARMAZENAMENTO
        ESPECIAL    = CS [19:23],       % MICROOPERAÇÕES ESPECIAIS
        %
        % TIPO 2
        %
        OPIMD       = CS [0:3] ,        % OPERAÇÃO DO TIPO IMEDIATO
        MODIFICADOR= CS [4:5] ,         % MODIFICADOR DE FUNÇÃO
        OPNI        = CS [6:13] ,       % OPERANDO IMEDIATO
        %
        % TIPO 3
        %
        OPJMP       = CS [0:3 ],        % OPERAÇÃO DE DESVIO
        CONDIÇÃO    = CS [4:8 ],        % CONDIÇÃO DE DESVIO
        TF          = CS [9:9 ],        % VERDADE OU FALSA
        OPNJC       = CS [10:18] ,      % OPERANDO DE DESVIO CONDI-
                                        % CIONAL
        CNDX        = CS [19:23]  ,     % CÓDIGO ESPECIAL
        %
        % TIPO 4
        %
        OPJMP        = CS[0:3]  ,       % OPERAÇÃO DE DESVIO INCON-
                                        % DICIONAL
        MODIFJMP     = CS[19:23] ,      % MODIFICADOR DE DESVIO
        OPNJI        = CS[7 :18]        % OPERANDO DE DESVIO INCON-
                                        % DICIONAL
```

Fig. IV.5 - Declaração de campos do minicomputador HP 21MX, de microprogramação diagonal.

4.2.3 - DECLARAÇÃO DE FORMATO

Sintaxe:

<declaração de formato> :: = FORMAT <lista de especificações de formato>

<lista de especificações de formato> :: = <especificação de formato>|
<lista de especificações de formato>, <especificação de formato>

<especificação de formato> :: = <identificador de formato>(<lista de identificador de campo ou microinstrução>)

<identificador de formato> :: = <identificador>

<lista de identificador de campo ou microinstrução> :: = <identificador de campo ou microinstrução>|
<lista de identificador de campo ou microinstrução>,<identificador de campo ou microinstrução>

<identificador de campo ou microinstrução> :: = <identificador de campo>|
<identificador de microinstrução>

Com o intuito de diminuir a palavra de controle, a codificação, em certos campos, é realizada de modo que estes sejam compartilhados. Desta forma, é possível mais de um formato de microinstrução. A Figura IV.4 mostra os formatos de microinstrução do minicomputador HP 21MX.

Definiu-se esta declaração com a finalidade de caracterizar a coerência dos campos de um dado formato. Ela também possibilita a verificação e a detecção de possíveis erros de microprogramação. Esta verificação também pode ser realizada com identificadores de microinstrução.

Deve-se ainda observar que todos os identificadores de campo ou de microinstrução, utilizados na declaração de formato, devem ser declarados com antecipação no microprograma fonte. Caso contrário, haverá emissão de mensagem de erro pelo tradutor LMP.

A Figura IV.6 apresenta o exemplo de declaração associada aos formatos de microinstrução do HP 21MX.

```
FORMAT
TIPO.1 (OPCODE, ALU, BARRA.S, STORE, ESPECIAL), % COMUM
TIPO.2 (OPIMD, MODIFICADOR, OPNI, STORE, ESPECIAL), % IMEDIATO
TIPO.3 (OPJMP, CONDIÇÃO, TF, OPNJC, CNDX), % DESVIO CONDICIONAL
TIPO.4 (OPJMP, OPNJI, MODIFJMP) % DESVIO INCONDICIONAL
```

Fig. IV.6 - Exemplo de declaração de formato.

4.2.4 - DECLARAÇÃO DE DEFINIÇÃO

Sintaxe:

<declaração de definição> :: = DEFINE <lista de especificações de definição>

<lista de especificações de definição> :: = <especificação de definição>|
<lista de especificações de definição>,<especificação de definição>

<especificação de definição> :: = <identificador de definição>=<contexto> #

<identificador de definição> :: = <identificador>

<contexto> :: = {qualquer sequência de caracteres válidos, diferentes de # , que inserida no microprograma fonte adquire significado válido}

<invocação de definição> :: = <identificador de definição><parâmetros de texto real>

<parâmetros de texto real> :: = <vazio>|
(<lista de segmentos de texto real>)

<vazio> :: = {conjunto vazio de caracteres}

<lista de segmentos de texto real> :: = <segmento de texto real>|
<lista de segmentos de texto real>,<segmento de texto real>

<segmento de texto real> :: = {texto real sem conter parentes descasados ou vírgulas parentizadas}

A idéia básica da declaração de definição é a de possibilitar o recurso de macro a nível de microprograma fonte, através de substituição ou inserção de texto quando é feita uma invocação de definição.

Com esta declaração, o microprogramador poderá associar um identificador a uma sequência de caracteres, que pode ter sido definida no contexto. Quando este identificador for referenciado no microprograma fonte, o contexto por ele identificado passa a ser analisado, neste ponto, como constituinte real do microprograma fonte.

Existem dois tipos de declaração de definição:

a) <u>Paramétrica</u>: Ela apresenta no contexto, uma ou mais vezes, a sequência de caracteres ?nn. O índice nn é um número que representa a ordem do parâmetro formal. Na invocação de definição, o parâmetro real, alocado mais à esquerda da lista de parâmetros reais, substitui o parâmetro formal ?nn de menor valor, e assim sucessivamente.

b) <u>Simples</u>: Esta não apresenta parâmetros formais, ou melhor, caracteriza-se pela ausência da sequência de caracteres ?nn no contexto.

Exemplos de declaração de definição:

```
DEFINE  R0 = 00H # ,
        R1 = 01H # ,
        R2 = 02H # ,
        STATUS = 11B # ;

DEFINE  IFCOND = TEST: = ?01  , JMPADR: = ?02  , CTL: = 1 #  ,
        CALL = ADR: = ?01  , MUX & ADD #  ;
```

Na invocação de definição correspondente, poder-se-ia ter:

```
IFCOND (STATUS, LOOP),  % ONDE LOOP É UM IDENTIFICADOR DE RÓTULO
CALL (FETCH),           % FETCH É UM IDENTIFICADOR DE SUB-ROTINA,
                        % ANTERIORMENTE DECLARADA
```

A restrição no segmento de texto real, onde não é permitida a existência de parenteses descasados ou de vírgulas com parenteses, visa a eliminação de ambiguidades que poderiam ocorrer. O exemplo abaixo ilustra situações deste tipo:

Na invocação de TEXTO (AB, TIPO (ALU, ADR), 3)

parâmetros: AB
TIPO (ALU, ADR)
3

ou,

parâmetros: AB
TIPO (ALU
ADR)
3

4.2.5 - DECLARAÇÃO DE MICROINSTRUÇÃO

Sintaxe:

<declaração de microinstrução> :: = MICRO <lista de especificação de microinstrução>

<lista de especificação de microinstrução :: = <especificação de microinstrução>|
<lista de especificação de microinstrução>,<especificação de microinstrução>

<especificação de microinstrução> :: = <identificador de microinstrução> <corpo da microinstrução> #

<identificador de microinstrução> :: = <identificador>

<corpo da microinstrução> :: = <especificação de formato livre>

<especificação de formato livre> :: = <especificação de campo de formato livre>|
<especificação de formato livre><especificação de campo de formato livre>

<especificação de campo de formato livre> :: = <número><tipo do campo>|
<configuração de bits>

<tipo do campo> :: = X|
<configuração de bits modificada>|
<variáveis de microinstrução>

<configuração de bits modificada> :: = <configuração de bits>|
<configuração de bits><lista de modificadores>

<lista de modificadores> :: = <modificador>|
<lista de modificadores><modificador>

<modificador> :: = '|-| / |←

<variáveis de microinstrução> :: = V|
V<valor por omissão>|
V<lista de atributos>|
V<lista de atributos><valor por omissão>

<valor por omissão> :: = <constante>
<constante> <lista de modificadores>

<lista de atributos> :: = <atributo>|
<lista de atributos><atributo>

<atributo> :: = '|-| / | ←

O objetivo desta declaração é permitir ao microprogramador associar identificadores a cordões de bits para geração de microinstrução, quando de suas invocações através de microcomandos de concatenação (Seção 4.4.2 - Microcomando de concatenação).

É também um recurso de macro, porém com certas restrições, e sua aplicabilidade se dá a nível de microcódigo.

A declaração de microinstrução é constituída por uma sequência de especificações de formato livre, separadas por vírgulas. Cada uma dessas especificações definem uma região da palavra de controle (campo em formato livre) e o seu tipo; podendo ser um conjunto de bits "don't care" (tipo X), uma configuração de bits, ou denotar uma variável de microprogramação (tipo V).

Definem-se, para um dado campo, atributos e modificadores; o atributo é um modificador permanente associado ao campo, enquanto o modificador tem caráter temporário.

Os atributos e os modificadores são operados da esquerda para a direita e assumem os seguintes significados:

a) Complemento de um (')

b) Complemento de dois (-)

c) Truncamento à esquerda ou à direita (/)

d) Justificação à esquerda (←)

Os complementos de um e de dois são efetuados de maneira usual, gerando uma configuração binária do mesmo tamanho.

Por omissão desses modificadores ou atributos, um dado campo será preenchido, justificando-se à direita, e os bits mais significativos serão completados com zeros. Esse modo de preenchimento poderá ser alterado através do atributo e/ou do modificador ← .

No caso do campo a ser preenchido possuir um tamanho menor que o da configuração a ser armazenada, há necessidade de utilizar o truncamento (/); caso contrário, uma mensagem de erro será emitida. O truncamento será efetuado à esquerda, no caso de justificação à direita, e à direita, no caso de justificação à esquerda.

Exemplos:

```
MICRO  GOTO = "0001"B, 3X, 10V, 15X # ,
       TEST = 4X, 3V "000"B, 25X # ,
       CALL = "0010"B-, 3V ←"000"B, 10V, 15X #
```

Configurações correspondentes:

```
GOTO:   0001XXXVVVVVVVVVVXXXXXXXXXXXXXXX

TEST:   XXXXVVVXXXXXXXXXXXXXXXXXXXXXXXXX
            ↑
            valor por omissão = 000B

CALL:   1110VVVVVVVVVVVVVXXXXXXXXXXXXXX
        └─┬┘└┬┘└────┬────┘
          │  │      └── 2º parâmetro
          │  └──── 1º parâmetro (valor por omissão=000B)
          └──── complemento de dois de 0010B
```

4.2.6 - DECLARAÇÃO DE RÓTULO

Sintaxe:

<declaração de rótulo> : : = LABEL <lista de identificador de rótulo>

<lista de identificador de rótulo> : : = <identificador de rótulo>| <lista de identificador de rótulo>, <identificador de rótulo>

<identificador de rótulo> : : = <identificador>

Através desta declaração, especifica-se um conjunto de identificadores que serão utilizados no microprograma fonte,associados às posições de memória de controle que se deseja referenciar (veja Comando rotulado na Seção 4.4).

Esta declaração é particularmente interessante na implementação dos microcódigos de controle de fluxo (desvios incondicionais, condicionais etc.).

Exemplos:

LABEL INICIO, FIM;

LABEL L1, L2, LOOP;

LABEL ERRO.FLAG.OVR, ERRO.DE.PARIDADE, TESTE;

4.2.7 - DECLARAÇÃO DE SUB-ROTINA

Sintaxe:

<declaração de sub-rotina> :: = PROCEDURE <identificador sub-rotina>;
<corpo da sub-rotina>

<identificador de sub-rotina> :: = <identificador>

<corpo da sub-rotina> :: = <bloco>|
<comando composto>

A declaração de sub-rotina define um conjunto de comandos (microinstruções) estruturados num bloco ou num comando composto, ao qual é associado um identificador, para posterior referência no microprograma fonte.

Este conjunto de comandos deverá gerar um conjunto de microinstruções, que deverão ser alocadas numa região da memória de controle, cujo início é associado ao identificador da sub-rotina. A microoperação que executa o retorno da sub-rotina deverá estar contida na última microinstrução do conjunto de microcódigos associados a essa sub-rotina.

Exemplo:

```
PROCEDURE FETCH;
   BEGIN
   ATRIB, OPN1: = R1;
   ...
   RETORNO, CR2: = 15  % RETORNO É IMPLEMENTADO NESTA MICROINSTRU
   END                 % ÇÃO
```

4.3 - EXPRESSÕES

Sintaxe:

```
<expressão> :: = <termo>|
                 <expressão> + <termo>|
                 <expressão> - <termo>
<termo> :: = <fator>|
             <termo>*<fator>|
             <termo>/<fator>
<fator> :: = (<expressão>)|
             <número>|
             $|
             <identificador de sub-rotina>|
             <identificador de rótulo>|
             <identificador de campo>
```

As expressões permitem algumas facilidades quanto às operações aritméticas e quanto às referências aos identificadores de sub-rotina, de rótulo, de campo, ou mesmo, do contador de alocação de microinstrução.

Os operadores aritméticos +, -, * e / têm significado convencional de soma, subtração, multiplicação e divisão,respectivamente. No caso do operador de divisão, a operação é realizada como sendo entre variáveis do tipo INTEGER da linguagem ALGOL.

Quanto à referência ao identificador de sub-rotina ou ao identificador de rótulo, o valor associado se refere, respectivamente, ao endereço onde a sub-rotina foi alocada, ou ao endereço da memória de controle que representa o rótulo.

No caso do identificador de campo, certo cuidado é necessário. O valor associado corresponde ao último valor atribuído a este campo, podendo até mesmo corresponder ao valor por omissão ("default"), quando da sua declaração. Porém, se o campo estiver no estado "don't care", a expressão será inválida.

Quanto ao contador de alocação de microinstrução, o seu valor é representado pelo símbolo $. O endereço de microprograma corresponde ao endereço onde será armazenada, pelo respectivo comando, a microinstrução.

Exemplo de expressões válidas:

```
10 + 5 * (3 + 2)
LOOP + 3            %+% ONDE LOOP FOI DECLARADO COMO SENDO UM ROTULO
FETCH               %+% ONDE FETCH FOI DECLARADO COMO SENDO UMA SUB-
                      % ROTINA, E O VALOR ASSOCIADO CORRESPONDE AO
                      % ENDEREÇO INICIAL, ONDE FOI ALOCADO A SUB-RO
                      % TINA FETCH
$ - 2               %+% A EXPRESSÃO EQUIVALE AO VALOR DO CONTADOR DE
                      % ALOCAÇÃO DE MICROINSTRUÇÃO MENOS DOIS,
ALU                 %+% ONDE ALU FOI DECLARADO COMO IDENTIFICADOR DE
                      % CAMPO E POSSUI UM VALOR ASSOCIADO
```

Exemplo de expressões inválidas:

```
-10+5*(3+2)    % UMA EXPRESSÃO NÃO PODE SER INICIADA COM O OPERADOR -
               %
-(5*2+100B)    % IDEM AO ANTERIOR
               %
INCR + 30      % ONDE INCR FOI DECLARADO COMO SENDO UMA MICROINS
               % TRUÇÃO, E, PORTANTO, A SUA REFERÊNCIA NA EXPRES
               % SÃO É INVÁLIDA
               %
ALU            % ONDE ALU FOI DECLARADO COMO IDENTIFICADOR DE CAM
               % PO E O MESMO CONTÉM ESTADOS "DON'T CARE"
```

4.4 - COMANDOS E MICROCOMANDOS

Sintaxe:

```
<comando> :: = <comando simples>|
               <comando rotulado>

<comando rotulado> :: = <identificador de rótulo>:<comando sim
                         ples>

<comando simples> :: = <bloco>|
                       <comando composto>|
                       <lista de microcomandos>

<bloco> :: = BEGIN <lista de declarações>;<lista de comandos>
             END

<comando composto> :: = BEGIN <lista de comandos>END

<lista de comandos> :: = <comando>|
                         <lista de comandos>;<comando>

<lista de microcomandos> :: = <microcomando>|
                              <lista de microcomandos>, <microco
                              mando>
```

<microcomando> : : = <atribuição> |
<concatenação> |
<configuração de campo anterior>

Os comandos são aqueles que na realidade geram a memória de controle. A cada comando correspondente a geração de uma microinstrução, que é composta de microoperações simultâneas. Os microcomandos correspondem a estas microoperações a nível de linguagem. No caso de um microcomando de concatenação, poderá haver geração simultânea de uma ou mais microoperações.

A idéia de associar blocos ou comandos compostos a conjuntos de microinstruções, que executem funções ou tarefas definidas, é possibilitar a estruturação da documentação do microprograma. O objetivo é incrementar a sua inteligibilidade para futuras alterações e manutenção dos mesmos.

4.4.1 - MICROCOMANDO DE ATRIBUIÇÃO

Sintaxe:

<atribuição> : : = <atribuição lado esquerdo>: = <expressão> |
<atribuição lado esquerdo>: = <configuração de bits>

<atribuição lado esquerdo> : : = <identificador de campo> |
<identificador de memória de controle>
[<par delimitador>]

O microcomando de atribuição permite o preenchimento de um dado campo ou de uma região da palavra de controle, com uma configuração de bits determinada pela expressão correspondente.

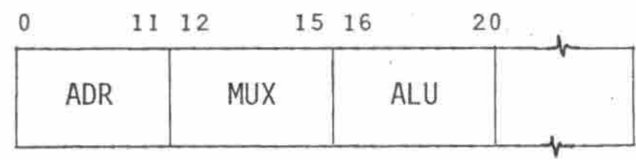
No cálculo da configuração de bits a ser armazenada no campo ou região da palavra de controle, a partir da expressão, duas situações podem ocorrer:

a) A configuração de bits calculada menor que o tamanho do campo ou da região da palavra de controle. Neste caso, a configuração de bits é armazenada, justificando-se à direita; isto é, as posições mais significativas são inseridas com bits iguais a zero.

b) A configuração de bits calculada maior que o tamanho do campo ou da região da palavra de controle. Neste caso, haverá emissão de mensagem notificando o microprogramador.

Estas situações são ilustradas a seguir.

Considerem-se a microinstrução abaixo e as declarações correspondentes:




```
FIELD  ADR = CS [0:11],    % ONDE CS É O IDENTIFICADOR DE MEMÓ
                           % RIA DE CONTROLE
       MUX = CS [12:15],
       ALU = CS [16:20]
```

1) ADR: = 2*4 + 16 é equivalente a ADR: = 24 ou ADR: = 11000B. Assim, o campo ADR conterá a seguinte configuração

000000011000B

2) MUX: = $ - 2 , onde $ equivale a 40. Assim, a configuração resultante será: 101000B,que provocará emissão de mensagem, uma vez que o campo MUX possui somente 4 bits de tamanho.

3) CS [16:20]: = 37Q, equivale a ALU: = 37Q, e a configuração armazenada contém todos os bits iguais a um: 11111B.

4.4.2 - MICROCOMANDO DE CONCATENAÇÃO

Sintaxe:

<concatenação> :: = <lista de invocação de microinstrução>

<lista de invocação de microinstrução> :: = <invocação de microinstrução>|
<lista de invocação de microinstrução> & <invocação de microinstrução>

<invocação de microinstrução> :: = <identificador de microinstrução> (<lista de parâmetros reais de microinstrução>)

<lista de parâmetros reais de microinstrução> :: = <parâmetro real>|
<lista de parâmetros reais de microinstrução>, <parâmetro real>

<parâmetro real> :: =@|
<parâmetro de microinstrução>|
<parâmetro de microinstrução><lista de modificadores>

<parâmetro de microinstrução> :: = <constante>|
<identificador de rótulo>|
<identificador de sub-rotina>

<lista de modificadores> :: = <modificador>|
<lista de modificadores><modificador>

<modificador> :: = '| - | / | ←

O microcomando de concatenação permite a montagem de uma microinstrução, através da concatenação de microinstruções já declaradas na declaração de microinstrução.

Neste processo de montagem faz-se uma verificação da existência ou não de regiões conflitantes, ou seja, a tentativa de atribuir uma certa configuração de bits a uma região solicitada. Caso uma das regiões já possua uma determinada configuração, ocorrerá o conflito e a emissão de mensagem de erro correspondente.

No caso de utilizar o parâmetro @, o valor definido para o parâmetro formal é o valor por omissão ("default"), estabelecido na declaração da microinstrução correspondente. E se não existir valor por omissão, o parâmetro @ será considerado inválido, causando emissão de erro.

O significado dos modificadores encontra-se na Seção 4.2.5 - Declaração de microinstrução.

Como ilustração de utilização de um microcomando de concatenação, considerar-se-ão as seguintes declarações:

```
LABEL LOOP;
DEFINE OVERFLOW = "111"B # ;
PROCEDURE FETCH;
   BEGIN
   ...
   END;
```

```
%
% MICROINSTRUÇÕES
% MICRO GOTO = "0001"B, 3X, 10V, 15X # ,
        TEST  4X, 3V "000"B, 25X #  ,
        CALL  "0010"B, 3X, 10V, 15X # ,
        ADD   17X, "01101"B, 10X # ,
        SUB   17X, "01110"B, 10X #  ;
  .
  .
  .
LOOP: ALU: = 7, ...;
  .
  .
  .
```

Exemplos de microcomandos de concatenação válidos:

a) GOTO (LOOP) & ADD;

A GOTO (LOOP) corresponde:

0001XXXVVVVVVVVVVXXXXXXXXXXXXXXX

E a ADD:

XXXXXXXXXXXXXXXXX01101XXXXXXXXXX

Como resultado final de GOTO (LOOP) & ADD, tem-se:

0001XXX000000100001101XXXXXXXXXX

Onde X representa o estado "don't care".

V representa o campo declarado como variável.

LOOP foi suposto associado ao endereço: 0000001000B

b) CALL (FETCH) & SUB & TEST (OVERFLOW)

Analogamente com o microcomando acima, tem-se como resultado final a seguinte configuração interna:

0010111000011000101110XXXXXXXXXX

Exemplos de microcomandos de concatenação inválidos:

c) GOTO (LOOP) & ADD & SUB

Existe conflito nas microoperações de ADD e de SUB,pois as mesmas são mutuamente exclusivas.

d) CALL (@) & TEST

O parâmetro real @ não pode ser empregado, pois o valor por omissão não foi declarado.

4.4.3 - MICROCOMANDO DE CONFIGURAÇÃO DE CAMPO ANTERIOR

Sintaxe:

<configuração de campo anterior> :: = <identificador de campo>

Este microcomando foi introduzido para facilitar a microprogramação de certos campos que são modificados com pouca frequência. Este é um fato que pode ser observado em experiências com geração de microprogramas.

O tradutor LMP armazena, para cada campo, a informação da última configuração assumida. A simples referência a um certo campo dentro de um comando, equivalerá a um microcomando de atribuição com o último valor assumido.

4.5 - OPÇÕES DE PÓS-PROCESSAMENTO

Sintaxe:

<Opções de pós-processamento> :: = <mapeamento de PROMs>|
<inversão de bits>|

<estados "don't care">|
<listagem de PROMs>|
<saída em fita magnética>

As opções de pós-processamento constituem-se na etapa final de geração dos microcódigos, já a nível de PROM, onde será implementada eletricamente a memória de controle do sistema.

Essas opções permitem ainda a documentação de cada PROM, através da listagem dos seus conteúdos, e da interconexão com o sistema HP-EMMAC, por meio de armazenamento dos microcódigos em fita magnética.

O sistema HP, constituído pelos minicomputadores HP2116B e HP21MX-E, monitora o Emulador de Memória de Microcontrole, cujo conjunto permite a depuração e a queima final das PROMs do sistema em desenvolvimento.

A forma intermediária gerada na etapa anterior à de pós-processamento tem por objetivo permitir ao microprogramador levantar algumas estatísticas quanto ao aproveitamento e/ou utilização da memória de controle, através dos estados "don't care". Esta forma intermediária já é quase a final, com exceção dos estados "don't care".

A opção de pós-processamento, estados "don't care", permite a definição dos mesmos, para a forma mais adequada ao circuito, de modo a evitar a queima desnecessária das posições deste tipo de uma dada PROM.

4.5.1 - MAPEAMENTO DE PROMs

Sintaxe:

<mapeamento de PROMs> :: = MAP <lista de PROMs>

<lista de PROMs> :: = <especificação de PROM>|
<lista de PROMs>,<especificação de PROMs>

<especificação de PROM> :: = <identificador de PROM> = <identificador de memória de controle> [<posição de endereçamento>,<posição na palavra de controle>]

<posição de endereçamento> :: = <par delimitador>

<posição na palavra de controle> :: = <par delimitador>

Esta opção permite associar os identificadores às PROMs e, ao mesmo tempo, especificá-las.

Na organização de memória de controle (Figura IV.7), as PROMs utilizadas têm tamanhos diferentes. Estas diferenças são caracterizadas na especificação de cada uma delas, no exemplo dado a seguir:

```
MAP   PROM.11 = CS [ 0:255, 0:7 ] ,
      PROM.12 = CS [ 0:255, 8:15 ],
      PROM.13 = CS [ 0:255, 16:19 ] ,
      PROM.14 = CS [ 0:255, 20:35 ] ,
      PROM.15 = CS [ 0:255, 36:39 ] ,
      PROM.16 = CS [ 0:255, 40:43 ] ,
      PROM.17 = CS [ 0:255, 44:47 ] ,
      PROM.21 = CS [ 256:767, 0:7 ] ,
      PROM.22 = CS [ 256:767, 8:15] ,
      .
      .
      .
      PROM.47 = CS [ 896:1023, 44:47 ];
```

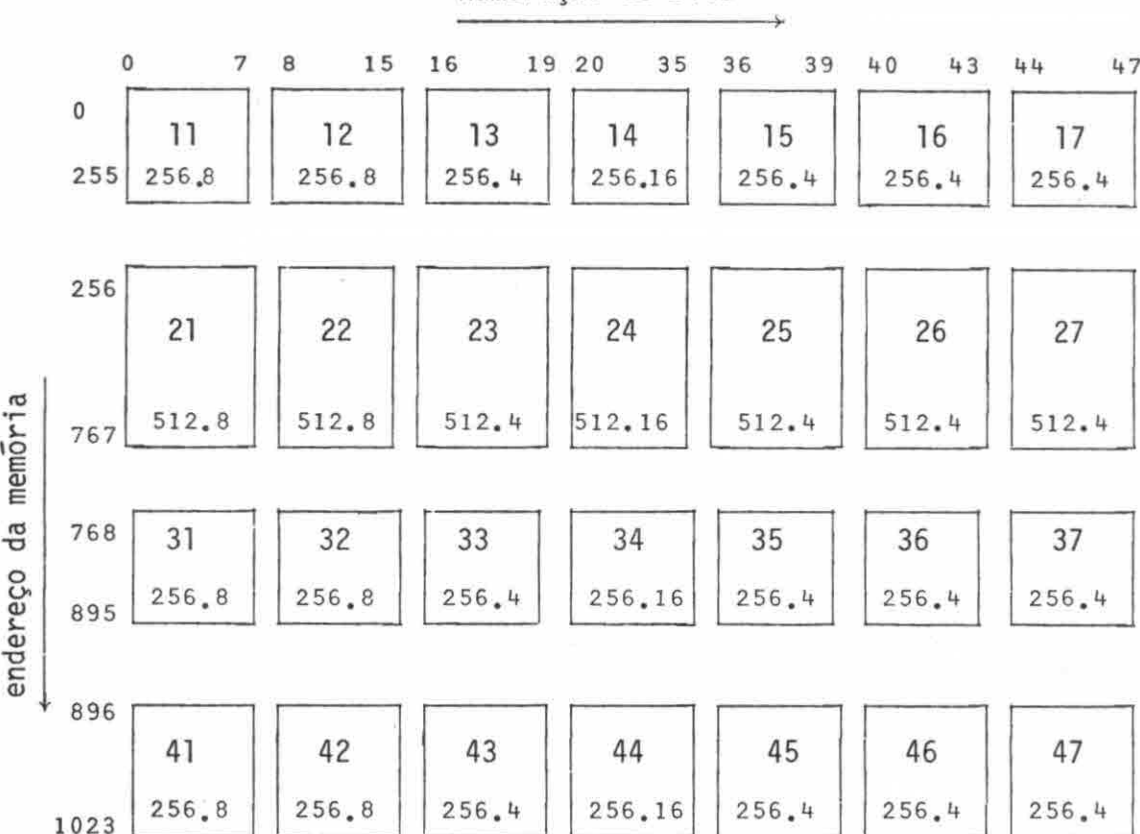


Fig. IV.7 - Organização das PROMs em uma memória de controle.

4.5.2 - INVERSÃO DE BITS

Sintaxe:

<inversão de bits> :: = INVERT <lista de especificação de bits>|
INVERT ALL

<lista de especificação de bits> :: = <especificação de bits>|
<lista de especificação de bits>,<especificação de bits>

<especificação de bits> :: = <identificador de campo>|
<identificador de memória de controle>[<posição na palavra de controle>]

<posição na palavra de controle> :: =<par delimitador>

Através da opção de pós-processamento de inversão de bits, pode-se complementar uma região da palavra de controle ou de um campo. Somente os estados "don't care" não são afetados por esta opção.

Exemplos:

a) INVERT ALL, realiza a complementação de todos os bits da memória de controle.

b) INVERT ADR, ALU, realiza a complementação dos campos ADR e ALU, independentemente da posição da palavra de controle na memória.

4.5.3 - ESTADOS "DON'T CARE"

Sintaxe:

<estados "don't care"> :: = DONTCARE EQL <dígito binário>FOR <lista de especificação de bits>|
DONTCARE EQL <dígito binário>FOR ALL

<lista de especificação de bits> :: = <especificação de bits>|
<lista de especificação de bits>,<especificação de bits>

<especificação de bits> :: = <identificador de campo>|
<identificador de memória de controle>[<posição na palavra de controle>]

<posição na palavra de controle> :: = <par delimitador>

Através desta opção de pós-processamento, os estados "don't care" são definidos para melhor adequação às memórias PROMs, utilizadas na implementação da memória de controle.

Por omissão ("default"), o tradutor LMP considerará o estado "don't care" como zero, na geração final das PROMs.

Permite-se, ainda, a utilização desta opção tantas vezes quantas se fizerem necessárias. Caso uma dada região seja definida várias vezes, a última definição feita é a que terá validade.

Exemplos de utilização:

a) DONCARE EQL 1 FOR ALL

b) DONCARE EQL 0 FOR ALU, ADR

4.5.4 - LISTAGEM DE PROMs

Sintaxe:

<listagem de PROMs> :: = LIST <formato de saída> ALL |
LIST <formato de saída sequência de PROMs>

<sequência de PROMs> :: = <identificador de PROM>|
<sequência de PROMs>,<identificador de PROM>

<formato de saída> :: = BIN | OCT | DEC | HEX

Através desta opção de pós-processamento, os conteúdos das memórias PROMs podem ser listados na impressora, para documentação ou conferição posterior à gravação das mesmas.

As PROMs devem ser identificadas anteriormente com a opção de mapeamento de PROMs.

Exemplos:

a) LIST BIN ALL , todas as PROMs serão listados na impressora

b) LIST HEX PROM.11, PROM.12, as PROMs PROM.11 e PROM.12 serão listados em hexadecimal.

4.5.5 - SAÍDA EM FITA MAGNÉTICA

Sintaxe:

<saída em fita magnética> :: = FMAG

Esta opção de pós-processamento visa a geração de uma fita magnética tipo CCT, contendo os microcódigos já gerados pelo tradutor LMP, para posterior utilização no sistema HP-EMMAC.

CAPÍTULO V

IMPLEMENTAÇÃO DO TRADUTOR LMP

Paralelamente ao presente trabalho, procurou-se implementar um tradutor para a linguagem de microprogramação LMP, que permitisse verificar a utilidade dessa linguagem como meio de geração de microcódigos.

Na implementação desse tradutor, inseriram-se e/ou implementaram-se muitas das idéias de Gries (1971) e de Aho e Ulman (1978).

A estrutura básica do tradutor LMP é apresentada na Figura V.1. O tradutor implementado no B-6800 (DPD/INPE) é essencialmente orientado pela sintaxe da linguagem. Assim, o Algoritmo de Análise Sintática é o núcleo do processo de tradução, que controla os analisadores Léxico ("Scanner") e Semântico, e o Gerador de Códigos.

O Analisador Léxico é ativado pelo Analisador Sintático, para adquirir a sequência de caracteres correspondentes ao microprograma fonte, fornecendo os símbolos da linguagem.

O Analisador Sintático efetua a análise sintática do microprograma fonte e, quando necessário, requisita ao Analisador Semântico e ao Gerador de Códigos, a função de verificar e armazenar as informações semânticas numa estrutura de tabelas, e, ao mesmo tempo, gerar uma forma intermediária. Esta forma intermediária não corresponde a nenhuma das formas usuais encontradas na literatura, e praticamente, corresponde à forma final dos microcódigos. As diferenças básicas ocorrem nos estados "don't care", que deverão ser definidos nas opções de pós-processamento.

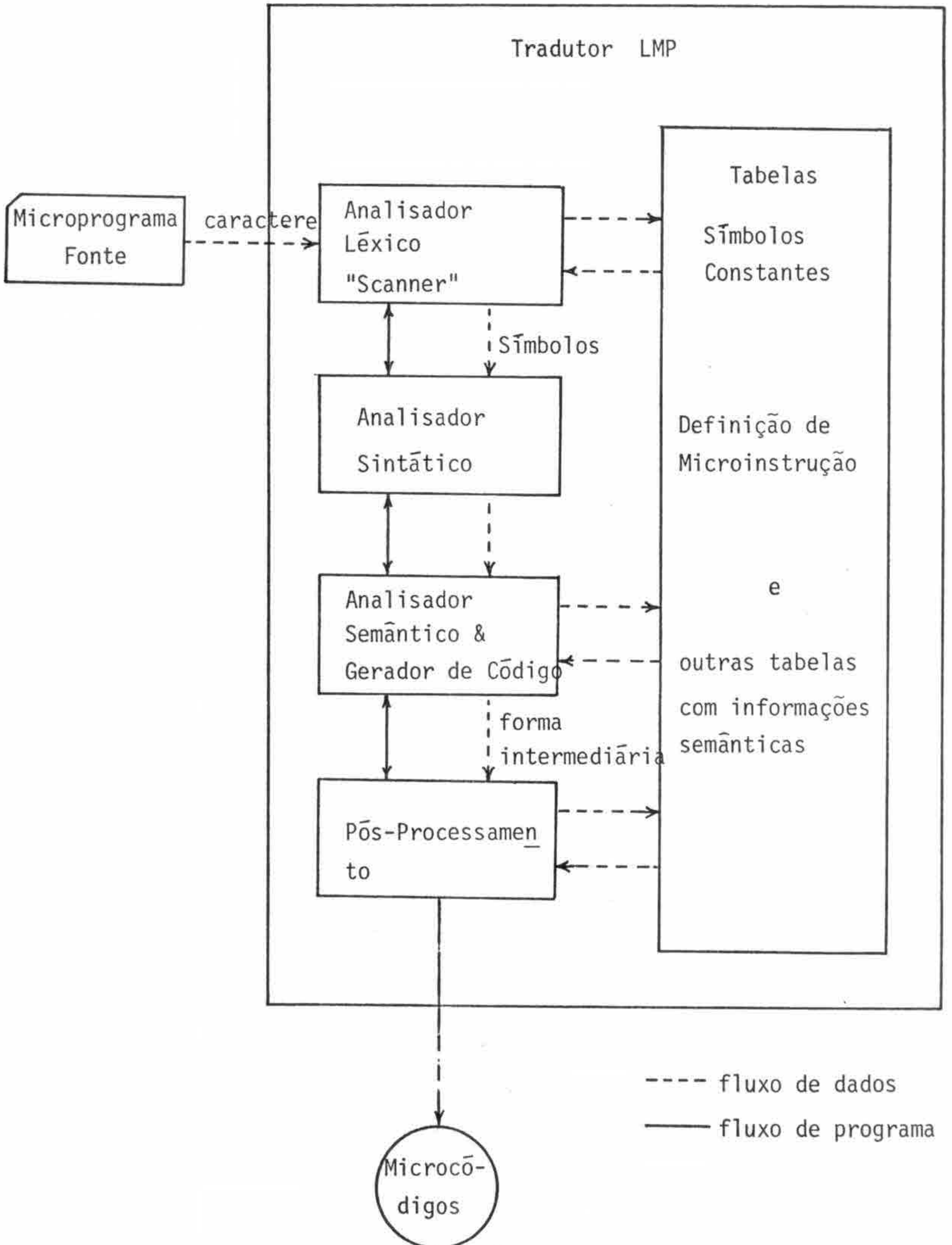


Fig. V.1 - Estrutura básica do tradutor LMP.

Essa forma intermediária poderá fornecer informações de interesse do microprogramador, quanto ao aproveitamento da memória de controle, através da medida e das listagens dos estados "don't care".

5.1 - O ANALISADOR LÉXICO ("SCANNER")

A função do Analisador Léxico é a de analisar o microprograma fonte, caractere por caractere, verificando a sua validade e identificando os elementos da linguagem, tais como: identificadores, constantes, delimitadores, palavras reservadas, etc.

Cada elemento identificado é passado ao Analisador Sintático e associado a um número inteiro que o caracteriza, de acordo com uma tabela predefinida internamente ao tradutor.

Os comentários são tratados pelo Analisador Léxico de forma autônoma em relação ao Analisador Sintático (Figura V.2).

De acordo com Gries (1971), toda definição e expansão de macros, existentes na linguagem, devem ser tratados pelo Analisador Léxico. Os recursos de macro implementados na linguagem LMP correspondem às declarações de definição e de microinstrução, que visam, respectivamente, a inserção de textos no microprograma fonte e a geração de microinstrução concatenada, através do microcomando de concatenação.

No caso da declaração de definição, o texto associado é armazenado na Tabela de Definição e, quando de sua invocação, através do identificador de definição associado, a análise sintática é feita sem se levar em conta a origem dos símbolos, seja ela proveniente de Tabelas de Definição ou de cartões do microprograma fonte.

No caso da declaração de microinstrução, além da análise léxica dos caracteres de entrada, faz-se uma verificação quanto à sua própria validade.

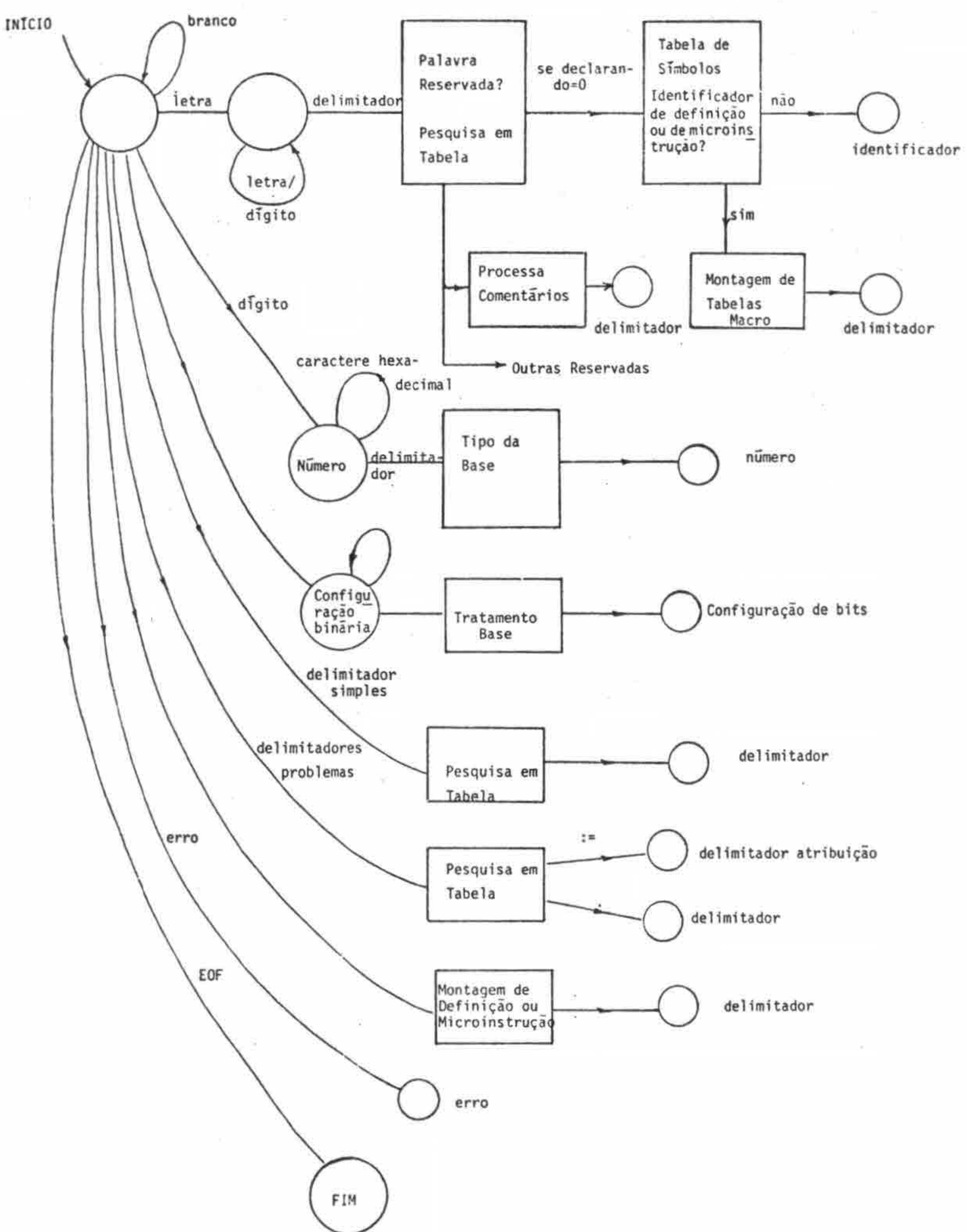


Fig. V.2 - O Analisador Léxico LMP.

Pela simples invocação do identificador de microinstrução, no microprograma, faz-se a inserção da microinstrução correspondente, alterada ou não, em função dos demais microcomandos.

O tratamento das constantes é feito também pelo Analisador Léxico, que os repassa ao Analisador Sintático com a configuração binária e seu tamanho, no caso da constante ser do tipo configuração de bits, e o valor decimal correspondente, no caso da constante ser do tipo número.

5.2 - O ANALISADOR SINTÁTICO

Na implementação do Analisador Sintático LMP, utiliza-se a técnica de "parsing" do tipo "Top-down".

Tentou-se, no início deste projeto, levantar as possíveis vantagens e/ou desvantagens da utilização de técnicas "Bottom-up" em relação ao "Top-down". Esta análise não foi tarefa trivial.

A implementação do tradutor LMP teve também como objetivo ser simples e flexível às possíveis mudanças das produções da linguagem, assim como, oferecer facilidade de depuração e manutenção.

A escolha de um tipo de "parsing" ou outro, aparentemente, não é perceptível a nível do sistema global, uma vez que as condições oferecidas pelo computador hospedeiro (B-6800) e sua linguagem de alto nível (Algol) fazem com que a implementação do Analisador Sintático "Top-down" descendente recursivo seja uma boa opção.

Certas tarefas foram criteriosamente deixadas para o Analisador Léxico, de modo a simplificar a análise sintática. No caso, as mensagens de erro sintático puderam ser bem específicas, oferecendo, assim, boa orientação na correção de microprogramas incorretos.

5.3 - O ANALISADOR SEMÂNTICO E O GERADOR DE CÓDIGO

Estes recursos são chamados pelo Analisador Sintático para verificar a correção semântica das construções da linguagem, para armazenar informações ligadas a essas construções na Tabela de Símbolos e, ao mesmo tempo, para gerar a forma intermediária (comentada anteriormente).

Nessa análise e geração de códigos, utilizam-se algumas rotinas bem determinadas para acesso às tabelas que constituem a Tabela de Símbolos; entre elas, aquela que armazena os microcódigos na forma intermediária. A estruturação básica da Tabela de Símbolos é apresentada na Seção 5.5.

5.4 - ROTINAS DE PÓS-PROCESSAMENTO

As rotinas de pós-processamento devem gerar os microcódigos, na forma final, para serem transportados a um Programador de PROM ou ao Emulador de Memória de Microcontrole Auxiliado por Computador (EMMAC), que está acoplado ao minicomputador HP 21MX-E.

As rotinas de pós-processamento atuam na forma intermediária, de acordo com as opções definidas no microprograma, estabelecendo os estados "don't care" em um dos níveis lógicos. Podem realizar,ainda, a inversão selecionada de bits e a listagem dos microcódigos para documentação.

5.5 - ORGANIZAÇÃO DA TABELA DE SÍMBOLOS

Na organização da Tabela de Símbolos, procurou-se atingir os objetivos que permitem armazenar informações estruturadas em blocos e que tentam minimizar os tempos de acesso às tabelas que a compõem.

Devido às peculiaridades da linguagem LMP,acredita-se que num microprograma usual far-se-á utilização intensa de identificadores, resultando em tabelas relativamente grandes, o que poderia tornar o acesso do tipo sequencial dispendioso em tempo.

Optou-se então pelo endereçamento do tipo HASH, que, com um dispêndio adicional relativamente pequeno de memória em relação às outras técnicas, permitiu uma acesso rápido, com poucas comparações.

A organização da Tabela de Símbolos é apresentada na Figura V.3, onde se verifica a existência de cinco tabelas lógicas que a compõe. Esta organização foi baseada nos esquemas apresentados por Aho e Ullman (1978) e por Wulf et alii (1975).

Na implementação dessas tabelas, procurou-se utilizar os recursos oferecidos pela linguagem Algol, de modo a minimizar o espaço ocupado pelas mesmas.

A tabela HASH, denominada THASH, ocúpa 33 palavras do B-6800, divididas em 101 campos úteis para endereçamento da tabela de Referência, TIDREF.

A tabela TIDREF contém uma parte ativa, onde são armazenadas todas as referências aos identificadores ativos, isto é, declarados no bloco corrente e nos blocos envolventes. Uma outra parte, denominada inativa, armazena referências aos identificadores dos blocos desativados, que serão utilizados para geração posterior de mapas de auxílio ao microprogramador.

A tabela TBLOCK armazena informações da estrutura em blocos e os ponteiros para a tabela TIDREF, que permitem localizar o início da área relativa ao bloco em questão e da tabela TSTRING, onde fica armazenada a configuração de caracteres ligados a cada identificador.

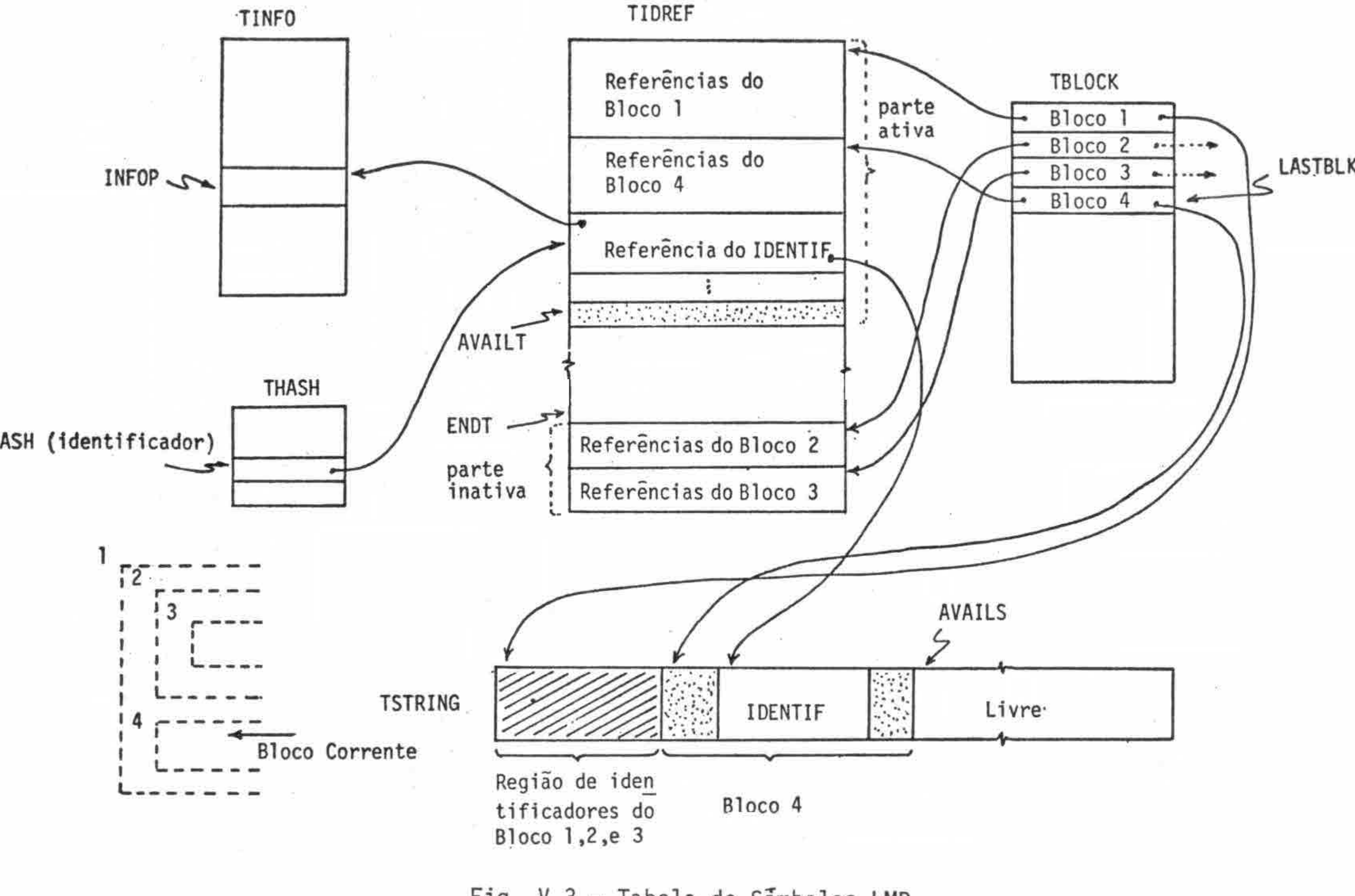


Fig. V.3 - Tabela de Símbolos LMP.

As demais informações semânticas são armazenadas na tabela TINFO.

A Figura V.4, mostra, com um pouco mais de detalhes, os nós de cada tabela, bem como o seu relacionamento. Os mnemônicos utilizados para cada campo de um nó foram empregados na implementação realizada. Não será feito um detalhamento de projeto, pois não é este o objetivo deste trabalho, porém, procurou-se evidenciar as idéias que nortearam a implementação do tradutor LMP.

5.6 - ERROS

Os erros podem ser detectados em três níveis:

a) Análise Léxica

b) Análise Sintática

c) Análise Semântica e Geração de Códigos

Em qualquer dos casos, faz-se a sinalização e a emissão da mensagem de erro correspondente.

Em caso de erro, nenhuma ação corretiva é tomada. Tenta-se, apenas, evitar a propagação do mesmo no microprograma fonte, percorrendo-se os caracteres de entrada, até que seja encontrado o final da declaração, o do comando ou o da opção de pós-processamento. Em termos de símbolos da linguagem, faz-se a análise até que seja encontrado um dos seguintes símbolos: ";", "BEGIN" ou "END".

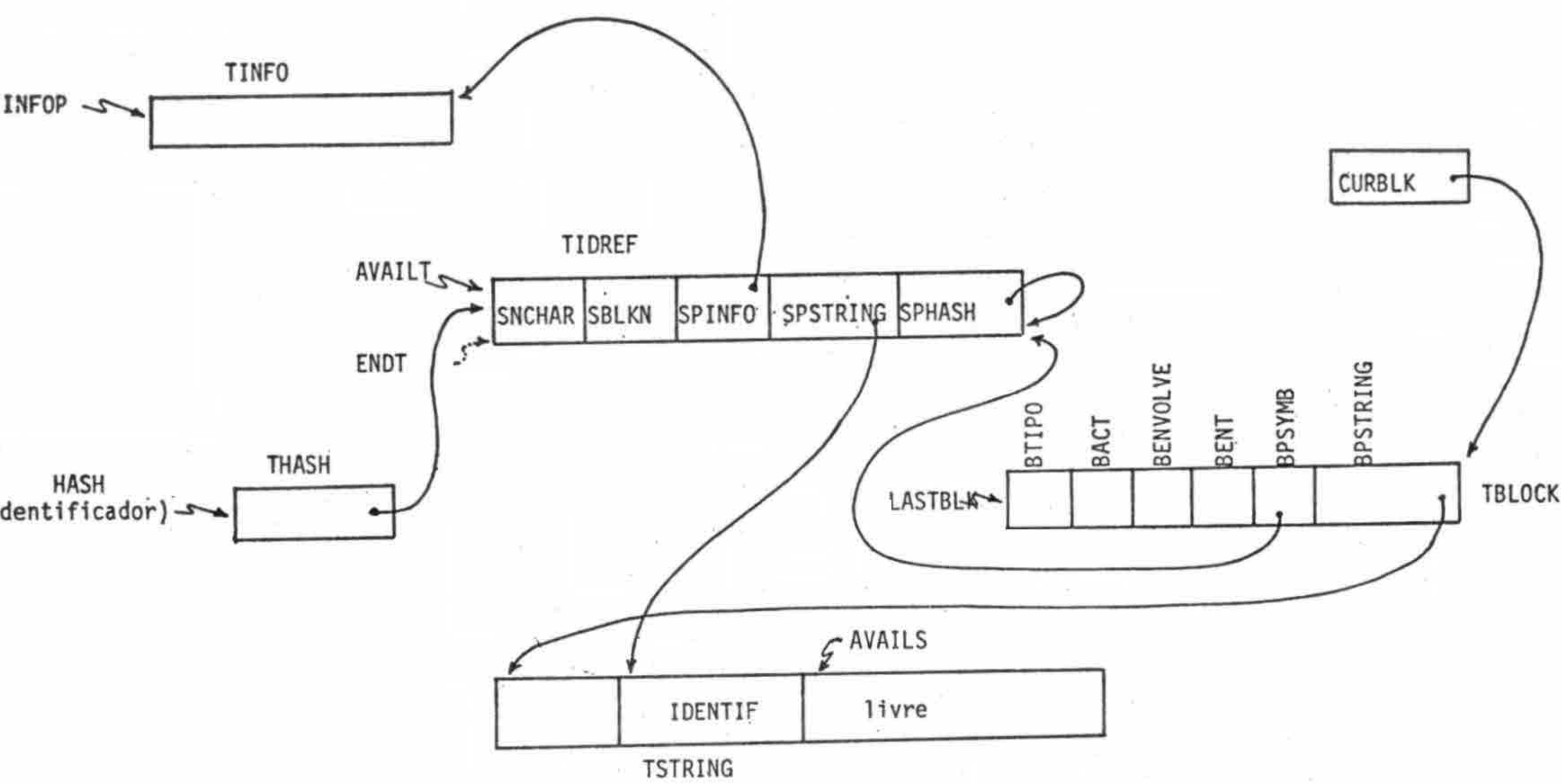


Fig. V.4 - Estrutura dos nós da Tabela de Símbolos LMP.

CAPÍTULO VI

CONCLUSÕES FINAIS

A linguagem LMP pode ser considerada como mais um recurso de CAD ("Computer Aided Design") no desenvolvimento de sistemas microprogramados. Sua aplicabilidade em termos de microprogramação é muito vasta, uma vez que a linguagem, em si, não depende especificamente do sistema para o qual são gerados os microcódigos. Usualmente, linguagens de microprogramação têm sido desenvolvidas especificamente para processadores específicos, e implementadas em baixo nível, principalmente as destinadas à microprogramação do tipo horizontal.

Acredita-se que dentro dos objetivos propostos para o Programa de Sistemas Digitais e Analógicos, a linguagem LMP deverá ser uma ferramenta bastante utilizada, principalmente por este laboratório do INPE. Atualmente, estão em desenvolvimento um MODEM de 4800BPS, um computador de 16 bits e uma unidade aritmética de ponto flutuante. Todos eles utilizam técnicas de microprogramação nos seus desenvolvimentos. Estas técnicas deverão ser empregadas também na implementação de controladores. Em especial, o desenvolvimento de computadores para supervisão de bordo de satélites deverá utilizar extensamente esta técnica. Neste caso, a linguagem LMP será uma ferramenta importante.

O principal mérito do presente trabalho se constitui na proposta desta linguagem, cujo tradutor está sendo implementado à parte.

A real avaliação do tradutor implementado só poderá ser feita após período de intensa utilização e, principalmente, quando a totalidade de seus recursos forem implantados.

Como futuras extensões deste trabalho, poder-se-ia mencionar a possibilidade de ter uma linguagem LMP, que possa contar com novas declarações e comandos de forma análoga à apresentada por DeWitt (1976).

Outra adição à linguagem LMP poderá ser fornecida pela inserção de novas opções de pós-processamento. A conexão direta do minicomputador HP 21MX-E, onde está acoplado o EMMAC, com o CPD/B-6800 do INPE, onde está implantada a linguagem CDL ("Computer Design Language"), deverá ser muito interessante. Este esquema de conexão poderá validar microprogramas de forma cruzada, bem como a lógica de projeto, sem prévia necessidade da montagem física do circuito.

O Apêndice A apresenta um exemplo de utilização da linguagem LMP.

AGRADECIMENTOS

Dentre as diversas pessoas que colaboraram direta ou indiretamente na execução deste trabalho, destacam-se as seguintes:

Dr. Eduardo Whitaker Bergamini, orientador, pelo apoio e interesse demonstrado em todas as fases do desenvolvimento do presente trabalho.

Dr. Flávio Roberto Dias Velasco, pelas sugestões na implementação do tradutor LMP.

Sr. José Benedito Soares Jr. pela colaboração na depuração do tradutor em desenvolvimento, Neusa Maria Dias Bicudo pela revisão de linguagem, e Sueli Inácia pelo trabalho de datilografia.

REFERÊNCIAS BIBLIOGRÁFICAS

AGERWALA, T. Microprogram optimization: a survey. *IEEE Transactions on Computers*, C-25(10):962-973, Oct. 1976.

AGRAWALA, A.K.; RAUSCHER, T.G. *Foundations of microprogramming: architecture, software and applications.* New York, Academic, c 1976.

ADVANCED MICRO DEVICES. (AMD). *AMDASM/80 reference manual - MDS resident microassembler.* Sunnyavale, Ca., 1977.

———. *A microprogrammed 16-bit computer.* Sunnyvale, Ca., c 1976 (seções).

———. *Microprogramming handbook and Am 2900 emulation.* 2 ed. Sunnyvale, Ca., c 1976a.

AHO, A.V.; ULLMAN, J.D. *Principles of compiler design.* Reading, Addison Wesley, 1978.

ALEXANDRIDIS, N.A. Bit-sliced microprocessor architecture. *Computer*, 11(6):56-80, Jun. 1978.

AMARAL, P.F.S. *Emulador de memórias de microcontrole auxiliado por computador.* São José dos Campos, INPE, maio, 1979. (INPE-1489-TDL/009).

BROWN, P.J. *Macroprocessors and techniques for portable software.* New York, John Wiley, c 1974.

BURROUGHS. *B6600/B7700 Algol language: reference manual.* Detroit, 1974.

DEWITT, D. Extensibility - a new approach for designing machine independent microprogramming languages. *Sigmicro Newsletter*, 7(3): 33-41, Sept. 1976. Micro 9 Proceedings.

DUBBS, E.W.; PARSONS, R.L.; PETERSON, J.E. A microprogram design system translator. In: GALEY, J.M.; KLEIR, R.L. *Microprogramming: a tutorial on the Queen Mary.* Long Beach, Ca., IEEE Computer Society, 1975. p. 145-147.

ECKHOUSE, R.H. A high level microprogramming language (MPL). In: AFIPS CONFERENCE, Atlantic City, 1971. Proceedings. v. 38, p. 169-177.

GRIES, D. *Compiler construction for digital computers.* New York, John Wiley, c 1971.

HANKINS, T.H. *Algol user's guide for the B6700 computer.* San Diego, 1974.

HEWLETT-PACKARD (HP). *21MX-Series computers - BCS and DOS microprogramming reference mannual.* Cupertino, Ca., 1977.

HUSSON, S.S. *Microprogramming principles and practices.* Englewood Cliffs, N.J., Prentice-Hall, c 1970.

KNUTH, D.E. *The art of computer programming.* 2 ed. Reading, Addison-Wesley, v. 1, 1975.

LAWS Jr., B.A. *A Microbe: a self commenting microassembler sigmicro Newsletter,* 8(3):61-65, Sept. 1977. Micro 10 Proceedings.

LLOYD, G.; VANDAM, A. Design considerations for microprogramming languages. In: AFIPS CONFERENCE, Chicago, 1974. Proceedings. v. 43, p. 537-543.

MALIK, K.; LEWIS, T. Design objectives for high level microprogramming languages. In: ANNUAL MICROPROGRAMMING WORKSHOP, 11., Pacific Grove, Ca., 1978. Proceedings. p. 154-160.

MALLET, P.W. Approaches to design of high level language for microprogramming. In: ANNUAL WORKSHOP ON MICROPROGRAMMING, 7., Palo Alto, Ca., 1974. p. 66-73.

MICK, J.R. Am 2900 bipolar microprocessor family. In: ANNUAL WORKSHOP ON MICROPROGRAMMING, 8., Chicago, 1975. Proceedings. p. 56-63.

ROHL, J.S. *An introduction to compiler writing.* New York, American Elsevier, c 1975.

TSUCHIYA, M.; GONZALEZ, M.J. Toward optimization of horizontal microprograms. *IEEE Transactions on Computers,* C-25(10):992-999, Oct. 1976.

WULF, W.; JOHNSSON, R.K.; WEINSTOCK, C.B.; HOBBS, S.O.; GESCHKE, C.M. *The design of an optimizing compiler*. New York, American Elsevier, c 1975.

APÊNDICE A

EXEMPLO DE APLICAÇÃO

O exemplo que se segue baseia-se em um outro fornecido pela ADVANCED MICRO DEVICES (1977), e ilustra o emprego da linguagem LMP, após adequada adaptação.

A Figura A.1, apresenta o diagrama de blocos simplificado da unidade de controle, para o qual serão gerados os microcódigos do exemplo. A Figura A.2 apresenta a palavra de controle e sua especificação por campos.

O algoritmo da Figura A.3 foi utilizado como o exemplo de aplicação da linguagem LMP, apresentado a seguir.

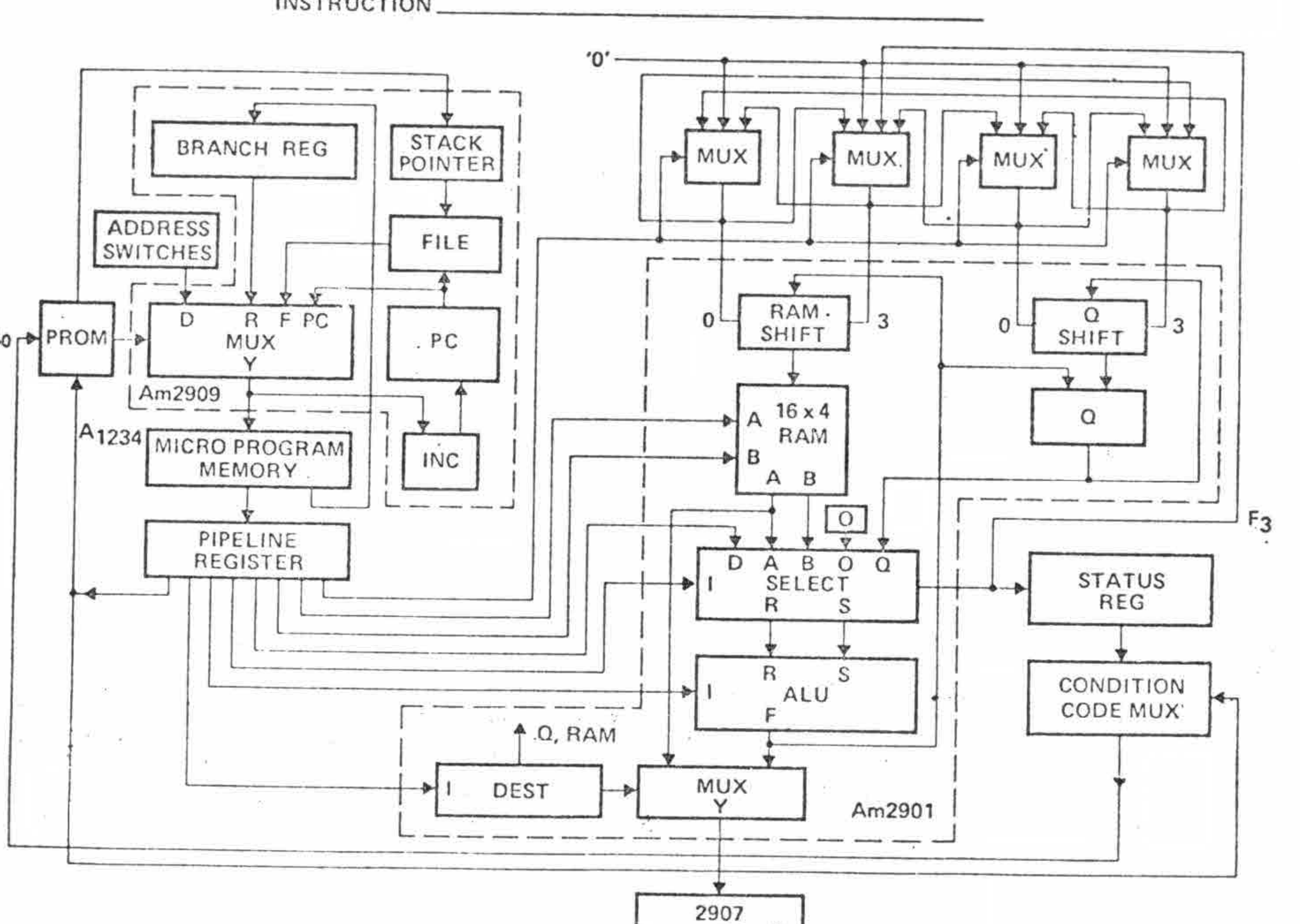


Fig. A.1 - Diagrama de blocos simplificado da unidade de controle, para o exemplo de aplicação.

FONTE: Advanced Micro Devices (1976a), p.25.

RAM & MUX SELECT	7				6				5				4				3				2				1				0			
RAM LOCATION	U9				U7				U8				U6				U5				U4				U3				U2			
LMP →	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
BIT NUMBER	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIT DEFINITION	BR_3	BR_2	BR_1	BR_0	P_3	P_2	P_1	P_0	MUX_1	I_8	I_7	I_6	MUX_0	I_2	I_1	I_0	C_n	I_5	I_4	I_3	A_3	A_2	A_1	A_0	B_3	B_2	B_1	B_0	D_3	D_2	D_1	D_0
FIELD DEFINITION	BRANCH ADDRESS				NEXT INSTRUCTION CONTROL				MUX_1	DESTINATION CONTROL			MUX_0	SOURCE SELECT			C_n	ALU			"A"				"B"				"D"			

CODE	FUNCTION
0	BRANCH REGISTER IF F ≠ 0
1	BRANCH REGISTER
2	CONTINUE
3	BRANCH MAP (D SWITCHES)
4	JUMP TO SUBROUTINE IF F ≠ 0
5	JUMP TO SUBROUTINE
6	RETURN FROM SUBROUTINE
7	FILE REFERENCE
8	END LOOP AND POP IF F = 0
9	PUSH (AND CONTINUE)
10	POP (AND CONTINUE)
11	END LOOP AND POP IF C_{n+4}
12	BRANCH REGISTER IF F = 0
13	BRANCH REGISTER IF F_3
14	BRANCH REGISTER IF OVR
15	BRANCH REGISTER IF C_{n+4}

	LOAD	Y =
0	F → Q	F
1	NOTHING	F
2	F → B	A
3	F → B	F
4	F/2 → B Q/2 → Q	F
5	F/2 → B	F
6	2F → B 2Q → Q	F
7	2F → B	F

	R	S
0	A	Q
1	A	B
2	O	Q
3	O	B
4	O	A
5	D	A
6	D	Q
7	D	O

	F
0	R + S
1	S − R
2	R − S
3	R ∨ S
4	R ∧ S
5	$\bar{R}$ ∧ S
6	R ⊻ S
7	$\overline{R \veebar S}$

		TYPE	DOWN*			
0	0	ZERO	0 → RAM_3	0 → Q_3	0 → RAM_0	0 → Q_0
0	1	ROTATE	RAM_0 → RAM_3	Q_0 → Q_3	RAM_3 → RAM_0	Q_3 → Q_0
1	0	ROTATE DOUBLE	RAM_0 → Q_3	Q_0 → RAM_3	RAM_3 → Q_0	Q_3 → RAM_0
1	1	ARITHMETIC DOUBLE	F_3 (Sign) → RAM_3	RAM_0 → Q_3	Q_3 → RAM_0	0 → Q_0

Fig. A.2 - Formatação da palavra de controle.

FONTE: Advanced Micro Devices(1976a), p.26.

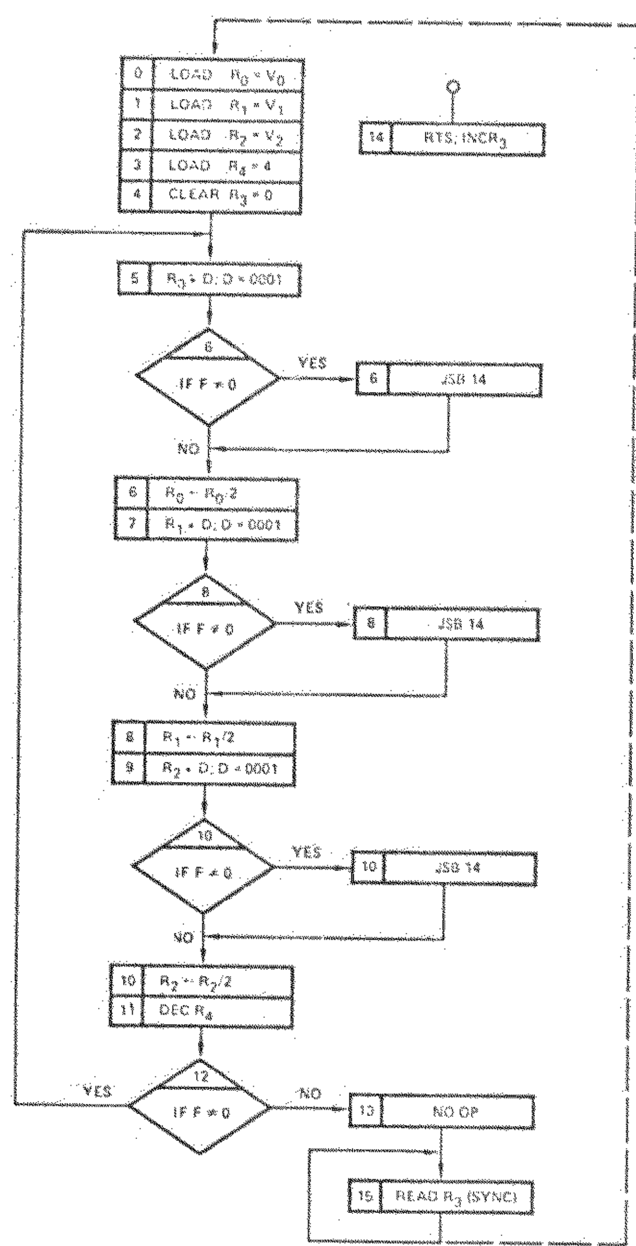


Fig. A.3 - Fluxograma do exemplo.

FONTE: Advanced Micro Devices (1976 a), p.28.

```
BEGIN
COMMENT  EXEMPLO DE APLICAÇÃO DA LINGUAGEM LMP
UTILIZANDO A ARQUITETURA DO "KIT" EDUCATIVO E DE AVALIAÇÃO
AM 2900 DA ADVANCED MICRO DEVICES;
%
% DECLARAÇÃO DE MEMÓRIA DE CONTROLE
%
MEMORY CS[0:255, 0:31];
%
% DECLARAÇÕES DOS CAMPOS DA PALAVRA DE CONTROLE
%
FIELD  BADR    = CS [0: 3] , % ENDER. DE DESVIO
       NEXT    = CS [4: 7] , % CONTROLE DO PRÓXIMO ENDER.
       MUX1    = CS [8:8 ] , % CONTR DO MUX BIT1
       DEST    = CS [9:11] , % DESTINO
       MUX0    = CS[12:12] , % CONTR DO MUX BIT0
       SOURCE  = CS[13:15] , % FONTE
       CN      = CS[16:16] , %
       ALU     = CS[17:19] , % OPER. NA ALU
       A       = CS[20:23] , % RAM REG. A
       B       = CS[24:27] , % RAM REG. B
       D       = CS[28:31] ; % LINHA D
%
% DEFINIÇÕES
%
%
DEFINE
       % *** REGISTROS ***
       R0  = "0"H #,
       R1  = "1"H #,
       R2  = "2"H #,
       R3  = "3"H #,
       R4  = "4"H #,
       R5  = "5"H #,
       R6  = "6"H #,
       R7  = "7"H #,
```

```
        RAMF    = 3 #,
        RAMQD   = 4 #,
        RAMD    = 5 #,
        RAMQU   = 6 #,
        RAMU    = 7 #,
        %
        % *** SELEÇÃO DO PRÕXIMO ENEDEREÇO ***
        %
        BRFNO          = "0"H #, % BRANCH REGISTER IF F ≠ 0
        BR             = "1"H #, % BRANCH REGISTER
        CONT           = "2"H #, % CONTINUE
        BM             = "3"H #, % BRANCH MAP
        JSRFNO         = "4"H #, % JUMP SUBROUTINE IF ≠ 0
        JSR            = "5"H #, % JUMP SUBROUTINE
        RTS            = "6"H #, % RETURN FROM SUBROUTINE
        STKREF         = "7"H #, % FILE REFERENCE
        LOOPFNO        = "8"H #, % END LOOP AND POP IF F = 0
        PUSH           = "9"H #, % PUSH AND CONINUE
        POP            = "A"H #, % POP AND CONTINUE
        LOOPCOUT       = "B"H #, % END LOOP AND POP IF CN+4
        BRFEQO         = "C"H #, % BRANCH REG IF F = 0
        BRF3           = "D"H #, % BRANCH REG IF F3
        BROVR          = "E"H #, % BRANCH REG IF OVR
        BRCOUT         = "F"H #, % BRANCH REG IF CN+4
        %
        % *** GERAIS ***
        %
        CNO  = "0"B #,
        CN1  = "1"B #,
        LOW  = "0"B #,
        HIGH = "1"B #,
        ZERO = "0"B #,
        ONE  = "1"B#;  % FIM DAS DEFINIÇÕES
%
% DECLARAÇÃO DE MICROINSTRUÇÃO
%
```

```
R8  = "8"H ≠ ,
R9  = "9"H ≠ ,
R10 = "A"H ≠ ,
R11 = "B"H ≠ ,
R12 = "C"H ≠ ,
R13 = "D"H ≠ ,
R14 = "E"H ≠ ,
R15 = "F"H ≠ ,
%
% ***OPERANDOS FONTES PARA O AM2901 ***
%
AQ = "0"Q ≠,
AB = "1"Q ≠,
ZQ = "2"Q ≠,
ZB = "3"Q ≠,
ZA = "4"Q ≠,
DA = "5"Q ≠,
DQ = "6"Q ≠,
DZ = "7"Q ≠,
%
% *** FUNÇÕES NA ALU ***
%
ADD   = 0 ≠, %  R + S
SUBR  = 1 ≠, %  S - R
SUBS  = 2 ≠, %  R - S
OR    = 3 ≠, %  R OR S
AND   = 4 ≠, %  R AND S
NOTRS = 5 ≠, %  R/ AND S
EXOR  = 6 ≠, %  R XOR S
EXNOR = 7 ≠, %  (R XOR S)/
%
%CONTROLE DE DESTINO AM2901 ***
%
QREG   = 0 ≠,
NOP    = 1 ≠,
RAMA   = 2 ≠,
```

```
       MICRO
       LOADREG = 4X, "2"H,          % CONTINUE
                 X, "3"Q,           % RAMF
                 X, "7"Q,           % DZ ( D + ZERO)
                 X, "3"Q,           % R OR S
                 4X,4V ,            % REG A SER CARREGADO
                 4V  # ,            % VALOR DE CARGA
%
       IFCALL  = 4V,                % ENDEREÇO DE DESVIO SE CONDIÇÃO
                                    % FALSA
                 4V,                % PRÕXIMO ENDEREÇO
                 X,"5"Q ,           % RAM D
                 X, "3"Q ,          % ZB
                 X, "3"Q ,          % R OR S
                 4X, 4V,            % REGISTRO
                 4X # ;
%
% DECLARAÇÕES DE RÕTULOS
%
LABEL LOOP.1, LOOP.2;
%
% DECLARAÇÃO DE SUBROTINA
%
PROCEDURE INCR3;
BEGIN
$ORIGIN 14
  NEXT:= RTS,  % RETORNO
  DEST:= RAMF , B:= R3, SOURCE:= ZB, ALU:= ADD, % R3 := R3 + 1
  CN:= CN1
END ; % PROCEDURE INCR3 SERÃ ALOCADA NO ENDEREÇO 14
%
% MICROPROGRAMA PRINCIPAL
% ======================
%
$ORIGIN 0
       LOADREG(R0,15); % V0 = 15
       LOADREG(R1, 9); % V1 = 9
```

```
        LOADREG( R2, 0);   % V2= 0
        LOADREG(R3 , 4);
        NEXT, DEST ,       % MESMA CONFIG. ANTERIOR
        SOURCE:= ZB ,      % B := F
        ALU:= AND,
        B:= R3 ;           % CLEAR R3
LOOP.1: NEXT, DEST, SOURCE:=DA ,
        ALU, A:=R0 , B:= R0 , D:= 1; % R0 AND D E D= 0001
        IFCALL(INCR3, JSRFNO, R0); % CALL SUBROTINA INCR3 SE F ≠ 0
                                   % IF ASSOCIADO À MICROINSTRUÇÃO
                                   % 6 DO DIAGRAMA DE BLOCOS
        :

LOOP.2: NEXT:= BR, BADR:= LOOP.2, SOURCE := ZB, ALU:= OR,
        B:= R3;                    % READ R3 (SYNC)
:
END
```