CAPíTULO V

ASSEMBLER E SIMULADOR

5.1. CONCEITOS DO AVMAC51 E DO AVLINK

O AVMAC 8051 é um assembler (realocável), com recursos de macros para a família do MCS-51. O AVMAC51 recebe como entrada um arquivo com instruções em assemby e procede da seguinte forma:

- Envia o arquivo a um pré-processador para validar as macros e as diretivas para assemblagem condicional.
- O pré-processador produz um segundo arquivo que é enviado ao assembler propriamente dito.
- O assembler produz um arquivo objeto e um arquivo com a listagem do código-fonte.
- O objeto produzido pelo assembler é enviado (pelo usuário) ao linker, o qual produz um arquivo absoluto em formato .HEX.

Figura 5.1. Fluxograma para utilização do assembler e linker.

Para ativar o assembler usa-se o comando que está ilustrado na figura 5.2.

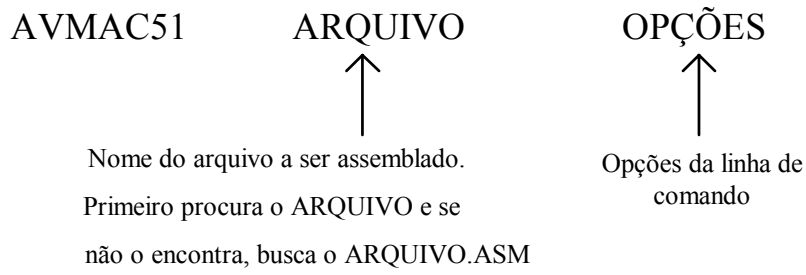


Figura 5.2. Linha de comando para a ativação do assembler.

O linker é usado da forma ilustrada na figura 5.3.

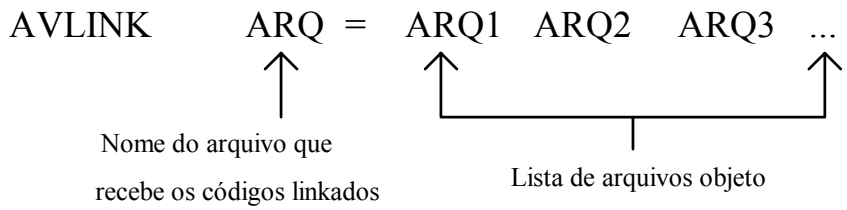


Figura 5.3. Linha de comando para a ativação do linker.

Por default, os arquivos absolutos serão feitos em formato INTEL HEX.

Um programa em linguagem assembly consiste de uma seqüência de "sentenças", cada uma ocupando uma linha do arquivo. Cada sentença pode ter até 4 campos. Os campos são separados por espaços e tabulações.

Label Operação Operando Comentário

Existem sentenças de 2 tipos:

- Instruções → do conjunto de instruções do microcontrolador
- Pseudo-instruções → que orientam o assembler.

LABEL: é um identificador que opcionalmente pode ser seguido por : (dois pontos). Ao usá-lo, é definido como um símbolo de usuário e atribui-se o endereço do primeiro byte da instrução. Os labels devem ser colocados na primeira coluna do texto.

OPERAÇÃO: contém uma instrução do microcontrolador (em linguagem assembly) ou uma pseudo-instrução (diretiva para o assembler).

OPERANDO: Pode haver nenhum, um, dois ou mais operandos; no caso de 2 ou mais operandos estes são separados por vírgulas.

COMENTÁRIO: Qualquer seqüência de caracteres precedida por ; (ponto e vírgula).

IMPORTANTE:

Labels → devem começar na primeira coluna.
Códigos → não devem começar na primeira coluna.
Comentários → pode começar na primeira coluna.

Os **símbolos** (labels) podem ter quantos caracteres se deseje mas somente os primeiros 32 são significativos. Podem ser: A → Z, 0 → 9, $.

Os **números** podem estar em outras bases diferentes da decimal, isto é, podem estar nas seguintes bases: 2, 8 ou 16; neste caso devem estar precedidos ou seguidos por:

BASE	PRECEDIDO	SEGUIDO
2	%	B
8	@	O ou Q
16	$	H

Exemplo: 254 → 0FEH ou $FE
127 → 1111111B ou %1111111

Para definir as **constantes de caracteres** usa-se ' ou "; estas são calculadas como inteiros de 16 bits. Em caso de uma letra, o byte de ordem alta é zero.

O símbolo $ é o contador de endereços do assembler (assembly-time counter). O valor de $ é o endereço do primeiro byte da instrução atual.

Nas **expressões aritméticas** os símbolos, números, caracteres e $ são referenciados como valores inteiros de 16 bits. Deve-se ter cuidado com os valores negativos, que são expressados em complemento a 2. Por exemplo, -1 é igual a FFFFH, então:

-1 + 5 = 4

-1 LT 0 => Falso, pois -1(FFFFH) não é menor que zero.

(Comparações não usam sinais).

OPERADOR	USO	RESULTADO
+	x+y	x+y
+	+x	0+x
-	x-y	x-y
-	-x	0-x
*	x*y	x*y (sem sinal)
/	x/y	x÷y (sem sinal)
MOD	x MOD y	resto de x÷y (sem sinal)
SHL	x SHL y	x deslocado à esquerda y vezes
SHR	x SHR y	x deslocado à direita y vezes
HIGH	HIGH x	MSB de x
LOW	LOW x	LSB de x
.	data.bit	endereço do bit do byte data
NOT	NOT x	negação de x (complemento 1)
AND	x AND y	AND lógico bit a bit (bitwise)
OR	x OR y	OR lógico bit a bit (bitwise)
XOR	x XOR y	XOR lógico bit a bit (bitwise)
LT	x LT y	1 se x<y, senão 0
LE	x LE y	1 se x≤y, senão 0
EQ	x EQ y	1 se x=y, senão 0
NE	x NE y	1 se x≠y, senão 0
GE	x GE y	1 se x≥y, senão 0
GT	x GT y	1 se x>y, senão 0

Figura 5.1. Lista com os operadores aritméticos.

5.2. PSEUDO-INSTRUÇÕES DO ASSEMBLER

Sempre que o assembler é chamado, este primeiro envia o arquivo fonte para o pré-processador. Pode-se evitar este passo com o seguinte comando (se não houver instruções para o pré-processador):

```
AVMAC51    ARQ.ASM    NOMACEVAL
```

Normalmente definir TRUE = 0FFFFH e FALSE = 0 evita problemas com relações.

END → deve estar presente em todo arquivo, se faltar, será gerado um.

EQU → atribui valores numéricos aos símbolos (os valores atribuídos são permanentes).

TEQ → atribui valores numéricos aos símbolos (os valores atribuídos são temporários).

OBSERVAÇÃO: é opcional a definição do tipo para os símbolos, os quais são:

CODE → memória de programa,

DATA → memória de dados interna,

XDATA → memória de dados externa,

BIT → endereços do espaço de bits,

NUMBER → qualquer coisa, é válido em qualquer lugar.

```
Exemplos:  LB      EQU     5, CODE → LB é um endereço de programa
           FLAG    EQU     P1.3,BIT → FLAG é um endereço de bit
```

DS → (Define Space) reserva espaço na RAM. Dentro do segmento de bits, DS reserva espaço em bits. Ex: DS 8 → reserva 8 bytes da RAM no segmento onde estiver, geralmente CODE.

DB → (Define Byte) reserva e inicializa espaço

Exemplos:

```
TXT     DB      "Isto é uma string", 0DH,0AH,0
TXT     DB      "Isto é uma string\r\n\0"
L1      DB      8           ;reserva 1 byte e o inicializa com 8
L2      DB      1, 2, 3     ;reserva 3 bytes e os inicializa com estes valores
```

SÍMBOLO	SIGNIFICADO
\n	nova linha
\r	retorno de carro
\t	tabulação
\0	NULL ou 0
\x	precede um hexadecimal de 2 dígitos (\x0A=line feed)

DW → (Define Word) é como DB, mas trabalha com words (16 bits). Ex: L3 DW 123H

PROC ,ENDPROC → labels globais, permitem programas modulares.

Símbolo local → começa com "L?XXXX" e só tem sentido entre PROC.....ENPROC, quando se encontra ENPROC, todos os símbolos locais são destruídos.

Exemplo:

```
BLOCO1  PROC
        MOV     A,#7
L?TOP:  CPL     P1.7        ;L?TOP é uma variável local
        DJNZ    A,L?TOP
        ENDPROC
```

INCLUDE → igual à linguagem C,

"ARQ" " " → busca arquivo no diretório corrente

<ARQ> < > → busca arquivo no diretório indicado pela variável MS/DOS "INCLUDE"

SEG → existem 4 segmentos já definidos **CODE**, **DATA**, **XDATA e BIT**. Se não foi indicado nada, usa-se o segmento CODE. A pseudo-instrução SEG permite trocar de segmentos

Exemplo:

```
        MOV     B,#20       ;Depois da assemblagem e linkagem
        SEG     DATA        ;não haverá nada entre as instruções
MARCA   DW      45H         ;MOV B,#20 e INC A. O segmento
        SEG     CODE        ;DATA será colocado em
        INC     A           ;outra parte
```

DEFSEG → permite a definição de outros segmentos, tais como: RAMDATA, ROMDATA, PILHA, etc. Na definição de um segmento pode-se informar:

ABSOLUT → não passa pela linkagem, deve ser dado o endereço inicial.

RELOCABLE → o linker define seu endereço.

ALIGNMENT → indica o tamanho do bloco para alinhamento; se for pedido um alinhamento de 100H, o linker tentará colocar o segmento em um endereço múltiplo de 100H.

BLOCK → indica ao linker que o segmento deve residir em um determinada área. Se BLOCK=PAGE então o segmento deve estar dentro de uma página de 2 KB.

OVERLAID → indica ao linker que o mesmo segmento em módulos diferentes deve estar no mesmo endereço.

CLASS → define a classe do segmento; existem 4 classes: CODE, DATA, XDATA, BIT

Exemplos:

```
DEFSEG STACK, CLASS=DATA                ;stack é um segmento redirecionável
DEFSEG KUMQUATS                         ;KUMQUATS é CODE por default
DEFSEG BLOCK, START=200H                ;BLOCK é segmento CODE, iniciando em 200H.
DEFSEG TRW, ALIGN=PAGE                  ;TRW é segmento tipo CODE e deve começar
                                        ;com endereço múltiplo de 800H (dentro de páginas
                                        ; de 2k)
DEFSEG NEWIO, START = 50H, CLASS=BIT    ;NEWIO é um segmento na área de
                                        ; bits, começando em 50H.
```

OBSERVAÇÃO: DEFSEG só define o segmento; para entrar em um segmento qualquer é necessário usar a instrução SEG.

ORG → origem.

PUBLIC, EXTERN → permite compartilhar símbolos definidos em arquivos (módulos) distintos. Ex: PUBLIC LABEL e EXTERN LABEL_EXT

O PRÉ-PROCESSADOR (ADP)

O pré-processador é chamado ADP. O assembler necessita encontrá-lo pois é a primeira ação a ser realizada. Todas as pseudo-operações relativas ao pré-processador começam com o caracter %.

%IF - %ELSE - %ELSEIF - %ENDIF → sem comentários

%SWITCH - %CASE - %DEFAULT - %ENDSW → idêntico à linguagem C.

Ex:
```
%SWITCH PARAM
%CASE 1
...
%CASE 9
....
%DEFAULT
...
%ENDSW
```

%REPT - %ENDREPT → repetir uma seção do programa.

Ex:
```
%REPT 5
...
%ENDREPT
```

%FOR - %ENDFOR → sem comentários.

Ex:
```
%FOR I = 1 TO 10
...
%ENDFOR
```

%MACRO → permite definir trechos de programa e representá-los por um símbolo.

Exemplo:

Definir um macro que represente 3 rotações para a esquerda:

```
RL3     %MACRO          ;exemplo de macro sem parametros
        RL      A
        RL      A
        RL      A
        %ENDM
```

Utilização desta macro:

```
        ADD     A,#5
        RL3
```

```
        SUBB    A,B
```

Definição de uma macro para representar a soma de três variáveis:

```
SOMA    %MACRO      x,y,z     ;exemplo de macro com parametros
        MOV         A,z
        ADD         A,y
        MOV         x,A
        %ENDM
```

Utilização da macro recém definida:

```
        MOV         B,A
        SOMA        R7,R6,R5
        MOV         R3,A
```

%IFB - %IFNB → (If Blank - If Not Blank) se foi atribuído ou não valor a um parâmetro (blank significa que não foi atribuído nenhum valor); geralmente usado nas macros.

%IFEQ - %IFNEQ → compara dois parâmetros.

%GENSYM → gerar símbolos automaticamente

Exemplo: A utilização do FOR pode resultar em repetição de um label:

```
        % FOR       I       IN "P1.7","P1.6","P1.5"
        AQUI        JNB     I,AQUI
        % ENDFOR
```

Vai resultar em:

```
        AQUI    JNB     P1.7,AQUI
        AQUI    JNB     P1.6,AQUI
        AQUI    JNB     P1.5,AQUI
```

O label AQUI foi definido três vezes, o que é um erro. A solução é usar GENSYM:

```
        % FOR       I   IN  "P1.7","P.6","P1.5"
        % GENSYM    AQUI
        AQUI    JNB     I,AQUI
        % ENDFOR
```

Vai resultar em:

```
        ??0000  JNB     P1.7,??0000
        ??0001  JNB     P1.6,??0001
        ??0002  JNB     P1.5,??0002
```

Notar que se evitou a repetição do mesmo label.

OPÇÕES DO ASSEMBLER AVMAC51

As opções podem ser colocadas na linha de comando quando o assembler é invocado ou em um arquivo fonte, usando a linha chamada "linha de opção". Ela deve iniciar na primeira coluna.

```
$option      opções,....
```

OBSERVAÇÕES:

- Duas instruções genéricas JMP e CALL são fornecidas. O assembler decide qual instrução usar em função da distância, de forma que obtenha o menor tamanho. Isso é válido para referências para trás. Em referências para adiante é obrigatório o uso de LJMP, AJMP ou LCALL, ACALL.
- Todos os nomes dos SFR e dos bits já estão definidos.
- Também já estão definidos os seguintes endereços de interrupções

```
RESET  →  0      ;endereço a ser executado depois do RESET
EXTI0  →  3      ;interrupção EXTERNA 0
TIMER0 →  08H    ;interrupção do TIMER 0
EXTI1  →  13H    ;interrupção EXTERNA 1
TIMER1 →  1BH    ;interrupção do TIMER 1
SINT   →  23H    ;interrupção da PORTA SERIAL
```

- Na listagem do arquivo com extensão .PRN gerado, o campo de endereços apresentará os seguintes símbolos:

&	→	segmento definido pelo usuário	@	→	segmento PAGE0 (se houver)
‘	→	segmento CODE	*	→	label externo
“	→	segmento DATA			

5.3. O LINKER - AVLINK

O AVLINK é um sistema interativo para linkar (ligar) objetos. Este programa é invocado usando:

```
AVLINK  ARQ = ARQ  START (100H)
```

A partir do ARQ.OBJ é gerado o ARQ.HEX, onde o programa começará a partir do endereço absoluto 100H.

Outros arquivos podem ser gerados:

ARQ.SYM → arquivo de símbolos.

ARQ.MAP → indica onde cada parte do programa está localizada.

AVLINK → invoca linker em modo interativo

AVLINK @ARQ → lê arquivo e aguarda comandos

OPÇÕES (listadas com a opção –h) **:**

MAP = mapfile → redireciona o arquivo .MAP (MAP = COM). (MA=mapfile)

NoMAP → não gera .MAP. (-NM)

ORDER (seg nombre, seg nombre...) → força que os segmentos sejam carregados na seqüência especificada. (-OC seg1, seg2, ...)

HEX = arquivo imagem → redireciona a saída .HEX. (HX = filename)

NoOut → não gera .HEX. (-NO)

Out Format = format_type → formato do arquivo absoluto: (OF= mot | mot0 | tek)

HEX → Intel HEX.

TEK → Tektronix.

MOT → Motorola.

SYmbols = symfile → redireciona a saída .SYM. (SY=filename)

NoSymbols → não gera arquivo .SYM (default). (-NS)

SYmbols → gera arquivo .SYM. (-SY)

PlainDump → tabela de símbolos. (-PD)

QUiet → não gera mensagens, incluindo os warnings. (-QU)

STart (clase,endereço) → endereço de início para cada classe.

ToP (clase,endereço) → endereço limite para cada classe.

RecLen= tamanho → tamanho do record nos arquivos .HEX (default = 32). (-RL=tam)

PutSeg (Segname, endereço) → realoca o segmento especificado em Segname. (-PS(seg,addr))

DefSym (symname, endereço) → define um símbolo público. (-DS(name,val))

MaxLen (segname, tamanho) → tamanho máximo para um segmento. (-ML(seg,len))

ShowsMods → coloca no arquivo .MAP a informação de cada arquivo .OBJ. (-SM)

ShowPublics → mostra no .MAP os labels públicos de cada arquivo. (-SP)

SortbyName → ordena a lista gerada por ShowPublics. (-SN)

RumsAt(segname, addr) → ?.

HEXFORM → é um programa que ordena os arquivos .HEX de forma que os endereços mais baixos apareçam primeiro. Modelo para a utilização:

```
HEXFORM     ARQ_IN.HEX      ARQ_OUT.HEX
            (não ordenado)      (ordenado)
```

Serve também para converter entre .HEX e o formato binário com a sintaxe: hexform ARQ=ARQ

AVLIB → permite a criação de bibliotecas. Modelo para a utilização:

```
AVLIB       ART.LIB = ARQ1.OBJ, ARQ2.OBJ
```

Se um programa que usa a biblioteca ART.LIB, mas chama somente ARQ1, então ARQ2 não será linkado.

AVREF → gera a referência cruzada. Modelo para a utilização:

AVREF TESTE = TESTE.SYM, ARQ1.XRF, ARQ2.XRF, ARQ3.XRF

(Os arquivos ARQ1.XRF, ARQ2.XRF, ARQ3.XRF foram gerados por AVLINK).

(O arquivo TESTE.HEX foi produzido pelos arquivos ARQ1.OBJ, ARQ2.OBJ, ARQ3.OBJ).

5.4. O FORMATO INTEL.HEX

O formato INTEL.HEX representa informações binárias através de um texto ASCII imprimível. Ele é organizado em uma série de records. Dentro de cada record o dado binário é representado por dígitos (hexadecimal) ASCII, 2 dígitos para cada byte.

Cada record tem um endereço de carregamento a partir do qual os bytes devem ser carregados. Para cada record também há um checksum de 8 bits que é um complemento a 2 da soma dos bytes de dados, contador, tipo de record e endereço. Notar que os valores binários dos bytes, e não os códigos ASCII, são usados no cálculo do checksum.

O checksum obtido somado ao que está no record deve resultar em zero.

RECORD DE DADOS

:	**COUNT** 2 char	**END. CARGA** 4 char	**TIPO** 00	**DADOS** (2*count) char	**CHECKSUM** 2 char	**CR**	**LF**

RECORD FINAL

:	**COUNT** 00	**END. INÍCIO** 4 char	**TIPO** 01	**CHECKSUM** 2 char	**CR**	**LF**

CR - Carriage Return **LF - Line Feed**

Figura 5.4. Descrição do formato INTEL.HEX.

Exemplo: Record com os dados 1,2,3,4,5 colocados a partir do endereço 1234H e com endereço de início em 0000H.

: 05 1234 00 01 02 03 04 05 A6

: 00 0000 01 FF

(foram deixados espaços para claridade)

Checksum para os records:

05 + 12H + 34H + 00 + 01 + 02 + 03 + 04 + 05 = 5AH → 0 - 5AH = A6

00 + 00 + 00 + 01 = 01 → 0 - 01 = 0FFH

5.5. PROGRAMAS EXEMPLO

A seguir são apresentados três programas fonte (.ASM) com a listagem (.PRN) gerada pelo assembler. Por simplicidade, todos os programas iniciam no endereço 0.

PROGRAMA FONTE EXEMPLO 1:

```
;EXEMPLO1.ASM
;
;Somar dois numeros BCD de 4 digitos : R1R2R3 = R4R5 + R6R7

        SEG     CODE
INICIO  MOV     A,R7        ; R7 + R5
        ADD     A,R5
        DA      A
        MOV     R3,A        ; R3 = R7 + R5
        ;
        MOV     A,R6        ; R6 + R4 + Carry
        ADDC    A,R4
        DA      A
        MOV     R2,A        ; R2 = R6 + R4 + Carry
        CLR     A
        ADDC    A,#0        ; usar o Carry
        MOV     R1,A        ; R1 = Carry
        END
```

LISTAGEM DO ARQUIVO EXEMPLO1.PRN GERADO PELO ASSEMBLER:

```
                1   ;EXEMPLO1.ASM
                2   ;
                3   ;Somar dois numeros BCD de 4 digitos : R1R2R3 = R4R5 + R6R7
                4
                5           SEG     CODE
0000' EF        6   INICIO  MOV     A,R7        ; R7 + R5
0001' 2D        7           ADD     A,R5
0002' D4        8           DA      A
0003' FB        9           MOV     R3,A        ; R3 = R7 + R5
                10          ;
0004' EE        11          MOV     A,R6        ; R6 + R4 + Carry
0005' 3C        12          ADDC    A,R4
0006' D4        13          DA      A
0007' FA        14          MOV     R2,A        ; R2 = R6 + R4 + Carry
0008' E4        15          CLR     A
0009' 34 00     16          ADDC    A,#0        ; usar o Carry
000B' F9        17          MOV     R1,A        ; R1 = Carry
                18          END
```

PROGRAMA FONTE EXEMPLO 2:

```
; EXEMPLO2.ASM
;
; Converter para BCD um numero binario entre 0 e 999
; R7R6(BCD) ← R5R4(Bin)
; Esta rotina esta no exemplo E10 caso c

            SEG     CODE
BB39        CLR     A
            MOV     R7,A    ;ZERAR R7 E R6
            MOV     R6,A

            MOV     A,R5
            JZ      LB1     ;SE R5=0, SALTAR ESSA FASE
LB2         MOV     A,R6    ;SOMAR 256 BCD TANTAS VEZES
            ADD     A,#56H  ;QUANTO EH O VALOR DE R5
            DA      A
            MOV     R6,A
            MOV     A,R7
            ADDC    A,#2
            DA      A
            MOV     R7,A
            DJNZ    R5,LB2  ;ZERO R5

LB1         MOV     A,R5    ;CONVERTE R4 PARA BCD
            MOV     B,#100
            DIV     AB      ;SEPARA CENTENAS
            ADD     A,R7
            DA      A
            MOV     R7,A
            MOV     A,B
            MOV     B,#10
            DIV     AB      ;SEPARA DEZENAS
            SWAP    A
            ADD     A,B
            ADD     A,R6    ;SOMA COM O QUE EXISTE
            DA      A
            MOV     R6,A
            MOV     A,R7
            ADDC    A,#0
            DA      A
            MOV     R7,A
            SJMP    $
            END
```

LISTAGEM DO ARQUIVO EXEMPLO2.PRN GERADO PELO ASSEMBLER:

```
                1   ; EXEMPLO2.ASM
                2
                3   ; Converter para BCD um numero binario entre 0 e 999
                4   ; R7R6(BCD) <-- R5R4(Bin)
                5   ; Esta rotina está no exemplo E10 C
                6
                7           SEG     CODE
0000' E4        8   BB39    CLR     A
0001' FF        9           MOV     R7,A        ;ZERAR R7 E R6
0002' FE        10          MOV     R6,A
                11
0003' ED        12          MOV     A,R5
0004' 60 0C     13          JZ      LB1         ;SE R5=0, PULAR ESSA FASE
0006' EE        14  LB2     MOV     A,R6        ;SOMAR 256 BCD TANTAS VEZES
0007' 24 56     15          ADD     A,#56H      ;QUANTO EH O VALOR DE R5
0009' D4        16          DA      A
000A' FE        17          MOV     R6,A
000B' EF        18          MOV     A,R7
000C' 34 02     19          ADDC    A,#2
000E' D4        20          DA      A
000F' FF        21          MOV     R7,A
0010' DD F4     22          DJNZ    R5,LB2      ;ZERO R5
                23
0012' EC        24  LB1     MOV     A,R5        ;CONVERTE R4 PARA BCD
0013'75 F0 64   25          MOV     B,#100
0016' 84        26          DIV     AB          ;SEPARA CENTENAS
0017' 2F        27          ADD     A,R7
0018' D4        28          DA      A
0019' FF        29          MOV     R7,A
001A' E5 F0     30          MOV     A,B
001C'75 F0 0A   31          MOV     B,#10
001F' 84        32          DIV     AB          ;SEPARA DEZENAS
0020' C4        33          SWAP    A
0021' 25 F0     34          ADD     A,B
0023' 2E        35          ADD     A,R6        ;SOMA COM O QUE EXISTE
0024' D4        36          DA      A
0025' FE        37          MOV     R6,A
0026' EF        38          MOV     A,R7
0027' 34 00     39          ADDC    A,#0
0029' D4        40          DA      A
002A' FF        41          MOV     R7,A
002B' 80 FE     42          SJMP    $           ;LOOP ETERNO
                43          END
```

PROGRAMA FONTE EXEMPLO 3:

```
;EXEMPLO3.ASM

;Converter um byte nos 2 caracteres HEXA ASCII correspondentes
;               AB (HEXA ASCII) ← A
;Exemplo: se Acc=0110 1010 → Acc=36H e B=41H
;Observacao: ASCII(6)=36H e ASCII(A)=41H
;Corresponde ao exemplo E12 do capitulo 4

;Para ilustrar este programa sera criado o segmento "PROGRAMA"
;do tipo CODE, realocavel.

            DEFSEG      PROGRAMA, CLASS=CODE        ;CRIAR SEGMENTO
            SEG         PROGRAMA                    ;ENTRAR NO SEGMENTO
HEXASC      MOV         B,A         ;GUARDAR UM BYTE
            ANL         A,#0FH      ;SEPARAR LSNIBBLE
            ACALL       NIBASC      ;CONVERTER NIBBLE → ASCII
            XCH         A,B         ;RECUPERAR UM BYTE
            ANL         A,#0FOH     ;SEPARAR MSNIBBLE
            SWAP        A
            ACALL       NIBASC
            SJMP        $           ;PARAR EXECUÇÃO

;Rotina para converter uma nibble no ASCII correspondente
NIBASC      PUSH        ACC         ;GUARDAR NIBBLE
            CLR         C
            SUBB        A, #10      ;A-10 → CY=1 ==> A<10
            POP         ACC         ;     → CY=0 ==> A<=10
            JNC         NIBASC1
            ADD         A, #30H     ;30H ... 39H
            RET
NIBASC1     ADD         A, #37H     ;44H ... 46H
            RET
            END
```

LISTAGEM DO ARQUIVO EXEMPLO3.PRN GERADO PELO ASSEMBLER:

```
                1   ;EXEMPLO3.ASM
                2
                3   ;Converter um byte nos 2 caracteres HEXA ASCII correspondentes
                4   ;AB (HEXA ASCII) <-- A
                5   ;Exemplo: se Acc=0110 1010 → Acc=36H e B=41H
                6   ;Observacao: ASCII(6)=36H e ASCII(A)=41H
                7   ;Corresponde ao exemplo E12
                8
                9   ;Para ilustrar este programa sera criado o segmento "PROGRAMA",
                10  ;do tipo CODE, realocavel.
                11
                12          DEFSEG      PROGRAMA, CLASS=CODE    ;CRIAR
SEGMENTO
                13              SEG     PROGRAMA  ;ENTRAR EM SEGMENTO
0000& F5 F0     14  HEXASC      MOV     B,A       ;GUARDAR UM BYTE
0002& 54 0F     15              ANL     A,#0FH    ;SEPARAR LSNIBBLE
0004& ....X     16              ACALL   NIBASC    ;CONVERTER NIBBLE → ASCII
0006& C5 F0     17              XCH     A,B       ;RECUPERAR UM BYTE
0008& 54 F0     18              ANL     A,#0F0H   ;SEPARAR MSNIBBLE
000A& C4        19              SWAP    A
000B& ....X     20              ACALL   NIBASC
000D& 80 FE     21              SJMP    $         ;PARAR EXECUÇÃO
                22
                23  ;Rotina para converter uma nibble em ASCII correspondente
000F& C0 E0     24  NIBASC      PUSH    ACC       ;GUARDAR NIBBLE
0011& C3        25              CLR     C
0012& 94 0A     26              SUBB    A,#10     ;A-10 → CY=1 ==> A<10
0014& D0 E0     27              POP     ACC       ;      → CY=0 ==> A<=10
0016& 50 03     28              JNC     NIBASC1
0018& 24 30     29              ADD     A,#30H    ;30H ... 39H
001A& 22        30              RET
001B& 24 37     31  NIBASC1     ADD     A,#37H    ;41H ... 46H
001D& 22        32              RET
                33              END
```

5.6. CONCEITOS DO AVSIM 8051

O AVSIM51 é feito para ser usado com o AVMAC51. Este simulador oferece muitos recursos:

- Uma "CPU visual" na tela,
- Todos os registros, flags, stack, portas e memória,
- Desassembler (Inverse assembler),
- Single step,
- Undo (back trace),
- Rodar todo o programa,
- Simular I/O através de arquivos,
- Pode-se aplicar patches diretamente usando mnemônicos,
- Contador de ciclos para medir tempo de execução,
- Breakpoints.

Chama-se o programa com: AVSIM51 [keystrokes]. A versão 1.0 suporta apenas um argumento.

AVSIM51 <CPU tipo> FL <command file_name>

Para que o AVSIM51 rode eficientemente devem estar presentes três arquivos:

- AVSIM51.OVR
- AVSIM51.REG
- AVSIM51.HLP

Se não foi especificado na linha de comando, o AVSIM51, ao ser invocado, pode especificar o tipo de CPU.

Existem dois modos de operação (troca-se de modo usando **ESC**):

- Command mode → Executar comando
- Display mode → Acesso dentro da CPU

5.6.1. Modo Comando

Neste modo o cursor sempre está em uma linha de comando - próximo na parte inferior do monitor. Dá assim a possibilidade de executar todos os comandos.

O modo comando é feito por menus. Se o menu é muito extenso, usa-se a barra de espaço para mostrar o resto. Os comandos são selecionados por:

- barra highlighted
- uma letra.

Alguns comandos pedem uma linha para complementá-lo. A seqüência **CRTL + C** é usada para abortar qualquer comando ou menu e retornar ao menu principal.

5.6.2. Modo Display

Neste modo o cursor está em um dos campos da CPU que aparece na tela. Tudo o que pode ser acessado pelo cursor está em highlight.

Cada campo na janela pertence a um dos tipos:

- HEX
- ASCII
- BINARY
- CODE

5.7. COMANDOS DO AVSIM51

Antes de iniciar o estudo dos comandos, é necessário uma série de observações.

Sempre que um valor é necessário pode-se usar uma expressão aritmética. Uma expressão é composta por uma ou mais constantes conectadas por operadores.

uma expressão:	20H - (BF_BASE-1)
múltiplas expressões:	MAIN , "AB", $-1, '0a'
sintaxe de endereço:	[adr space:] expressão → D:BF_BASE

Os valores numéricos podem estar na base 2, 8, 16 ou 10.

@ → indica base 8 quando usado com os mnemônicos

@ → em expressões indica o operador indireto.

Há diversos outros operadores:

. → um ponto é um símbolo onde está o resultado da última expressão
'.+100' → expressão anterior +100.

\+ → suma.

\- → subtração ou negação.

@ → operador indireto.

( ) → até 4 níveis.

Quando um endereço é solicitado, o espaço de endereçamento pode ser explicitamente especificado como:

Code	C	Ex: C: Main
Data	D	Ex: D: BUFFER
External	X	Ex: X: 0A

5.7.1. Comandos de Ambiente:

-Carregar arquivos:
- Load Prog → atribui memória e carrega o programa
- Load Data → atribui memória e carrega os dados
- Load symbols → carrega a tabela de .símbolos .do Cross.Assembler
- Load Avocet → Load Symbols (.PRN)
- Load Avocet → Load Program (.HEX ou .MIK)

-Arquivo de comando: evita a repetição dos comandos freqüentes.
- Load → executa arquivo de comando.
- Open → abrir arquivo de comando
- Close → fechar arquivo de comando
- Restart → reiniciar arquivo de comando

-Atribuir memória: Set memory_map: → atribui espaço de dados externo.
(como ROM ou RAM e não pode ser liberada)

5.7.2. Comandos para a Execução de Programas:

5.7.2.1. Breakpoints:

Existem diversos tipos de breakpoints:

- De acordo com o acesso: read/write
  write only
- De acordo com o endereço: endereço de memória
  faixa de endereço
  registrador
- De acordo com a duração: sticky → somente é desativado pelo usuário
  dynamic → se auto destrói quando ocorre

O número de breakpoints que podem estar ativos simultaneamente depende somente da memória disponível no PC.

Dois breakpoints tipo "sticky" são instalados automaticamente:

→ todo endereço de memória não definido

→ toda faixa de memória definida como ROM tem o tipo "write only break point".

Breakpoints incondicionais: sempre interrompem a execução quando há acesso a um endereço.

- Set sticky_bkpt → só é desativado pelo usuário.
- Set dynamic_bkpt → se auto apaga quando ocorre.

Acesso: read/write ou write only

Endereço: valor ou faixa de valores

Para retardar uma interrupção até n passos a partir do breakpoint, entre com um valor decimal antes da seleção de acesso/endereço.

Breakpoint dinâmicos incondicionais podem ser ativadas usando "break point cursor keys" (F2, F3, F4).

Quando um breakpoint ocorre, a execução do programa se detém antes da instrução que ativa o breakpoint.

→ Se é dinâmico, ele é apagado e a execução do programa pode prosseguir.

→ Se é do tipo "sticky" é necessário usar "single step".

Os comandos que operam com os breakpoints são:

- View breakpoints.
- Reset breakpoints.
- Reset all.
- Reset trap list.

5.7.2.2. Condições (Value, Range, Mask, Indirect)

VALUE → deve ser fornecido um valor de 8 bits o qual será comparado com o conteúdo, endereço ou registro sujeito a breakpoint. Exemplo: interrompe quando o conteúdo de R0 seja 0F0H.

RANGE → deve-se definir um limite inferior e um superior; quando o conteúdo estiver nesta faixa (inclusive) o breakpoint é ativado.

MASK → máscara de 8 bits com 0, 1 e X (don't care). Quando os bits da máscara coincidem com o conteúdo do endereço do registro é produzido um trap. Exemplo: mask = 1XXX XXX0.

INDIRECT → agora o conteúdo do registro é interpretado como uma endereço e é usado para buscar um dado na memória, o qual é comparado com o valor. Um segundo valor de 16 bits (offset) deve ser fornecido para ser somado com o conteúdo do registro (index) e assim produzir um endereço alvo.

5.7.2.3. Pass Points:

São idênticos aos breakpoints, mas em vez de gerar um trap, eles incrementam um contador de 32 bits.

5.7.2.4. Opcode Traps:

O PC é usado para apontar um dado (um opcode neste caso), que será comparado com um dado especificado. A opção pede um mnemônico completo ou, pelo menos, o primeiro byte é usado na comparação.

Exemplos: JNZ $ interrompe em toda instrução que tenha JNZ,
MOV R0,#7 provocar um trap em todos MOV R0, #X.

5.7.2.5. Execute Command:

Permite executar um comando como se fosse um interpretador. Pode ser usado para desviar para qualquer lugar (JMP adr) ou para forçar o retorno de uma subrotina.

5.7.3. Comandos de Display:

DUMP 1 / 2 Absolute → endereço de inicio da janela que mostra a área de dados. Com o cursor dentro da janela, basta usar <SCROLL UP> ou <SCROLL DOWN>.

DUMP 1 / 2 Indirect → deve ser fornecido uma endereço ou registro e também o valor de offset. O endereço indireto aparece em highlight. Exemplo: usa-se para monitorar transferências com @R1 ou @R0.

5.7.4. Comandos de I/O:

Permite que se unam arquivos aos pinos do microcontrolador e assim se simulem as entradas e saídas. Em um simulador não existem dispositivos externos para interagir com os pinos do microcontrolador. São fornecidos dois recursos para simular atividade externa:

- I/O Interativo → o usário manualmente gera os sinais nos pinos de interesse.
- I/O por Arquivo → as entradas e saídas vão e vem dos arquivos.

Um arquivo de I/O é uma cadeia de bytes, onde cada bit está conectado a um pino externo do microcontrolador. Também deve-se indicar a velocidade (em ciclos de máquina) que será usada para varrer o arquivo. A recepção de caracteres ASCII pela porta serial é simulada ao unir os 7 bits menos significativos do arquivo de I/O ao SBUF e o bit mais significativo ao bit RI do SCON.

Podem ser gerados arquivos de entrada e arquivos de saída. O comando UNDO não somente retrocede a instrução como também retrocede os arquivos de I/O corretamente.

I/O, OPEN, (Y or N) rewind, IO rate, Mapbit:

I/O rate n → a cada n ciclos de máquina será feita uma transferência.

Mapbit → <I/O bit (0-7)> <memo bit(0-7)>, <endereço> <IN/OUT>:

Exemplo: 7, 3, P1, OUT → irá direcionar o bit P1.3 para o bit 7 do arquivo de saída.

7,RI, IN → o valor do bit RI (do SCON) virá do bit 7 do arquivo de entrada.

View I/O files → mostra todos os arquivos de I/O abertos, a designação dos bits e o valor do "rate counter".

5.7.5. Comandos de Memória:

Set Memory_map → Define memória adicional RAM ou ROM, a qual é inicializada com zeros.
View Memory_map→ Ver os espaços de endereços de cada área.
EXPRESION → Coloca o valor da expressão na posição do cursor.
Memory Clear → Preencher com zeros a área especificada.
Memory Fill → Preencher com um valor a área especificada.
Memory Move → Deslocar áreas de memória, mesmo quando há superposição.
Memory Search → Buscar determinado caracter na memória.

5.7.6. Incremental Cross-Assembler:

Patch code → Permite entrar com os mnemônicos em vez de dados hexadecimais (o endereço usado é o que está no PC). Para alterar um byte de uma instrução pode-se usar a área que está à direita do PC.
Open Output File → Grava todas as instruções que foram digitadas para o patch.

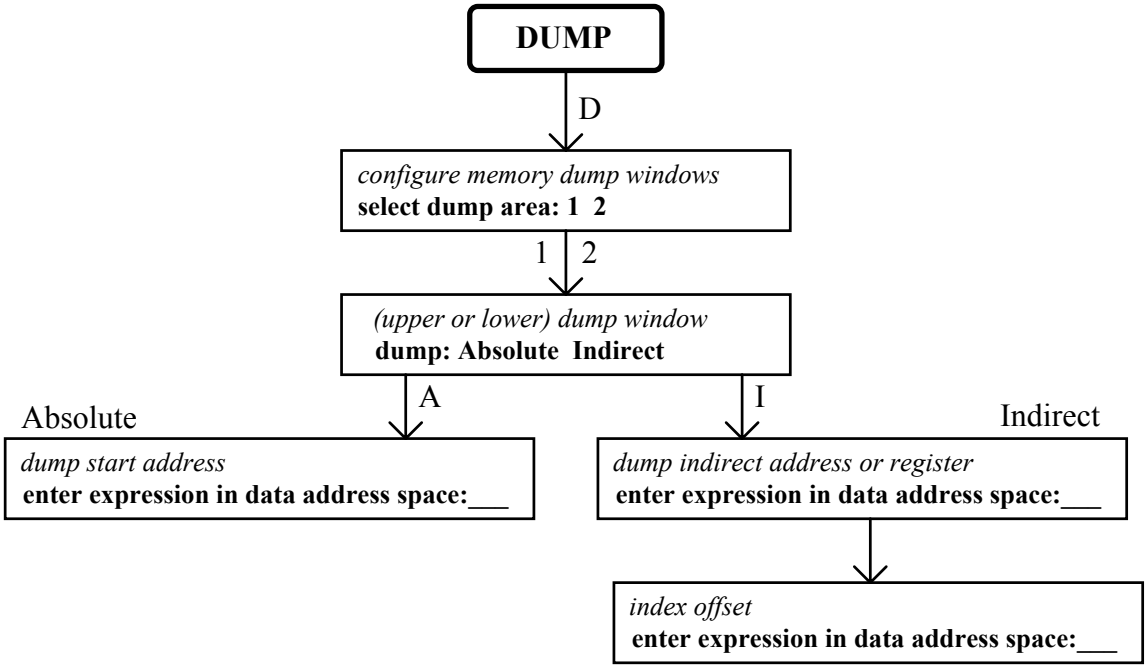
5.8. FLUXOGRAMA DE OPERAÇÃO DO AVSIM51

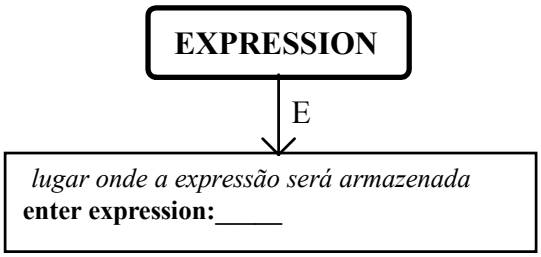
A seguir será apresentado o fluxograma de todas as opções que existem no AVSIM51.

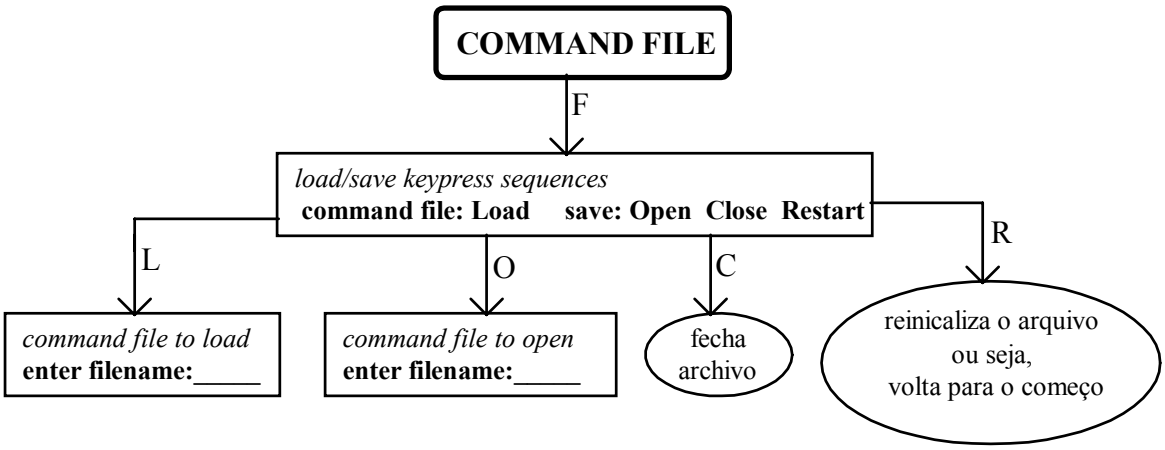
Usar-se-á a seguinte convenção:

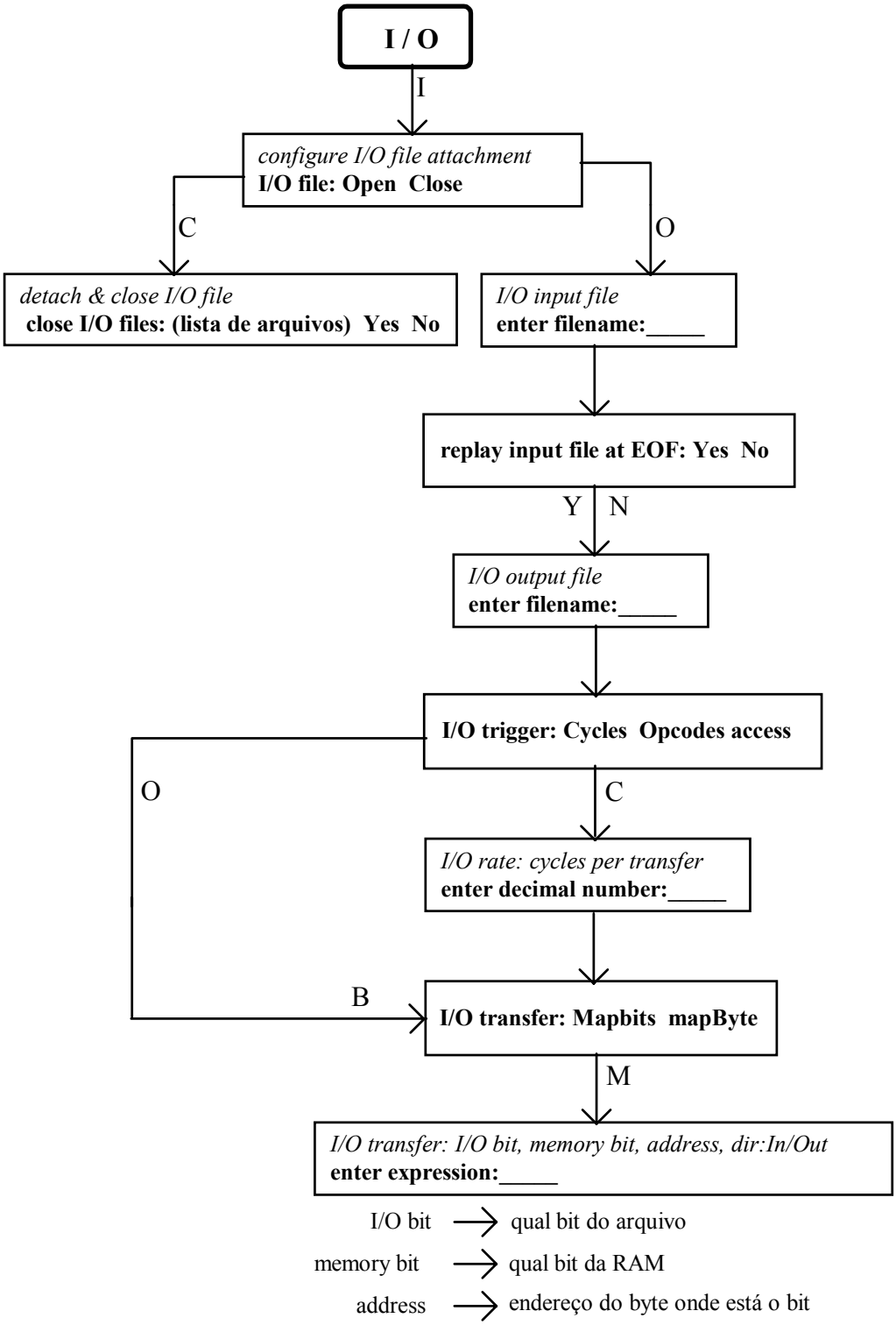
em itálicos → as informações exibidas pelo AVSIM.

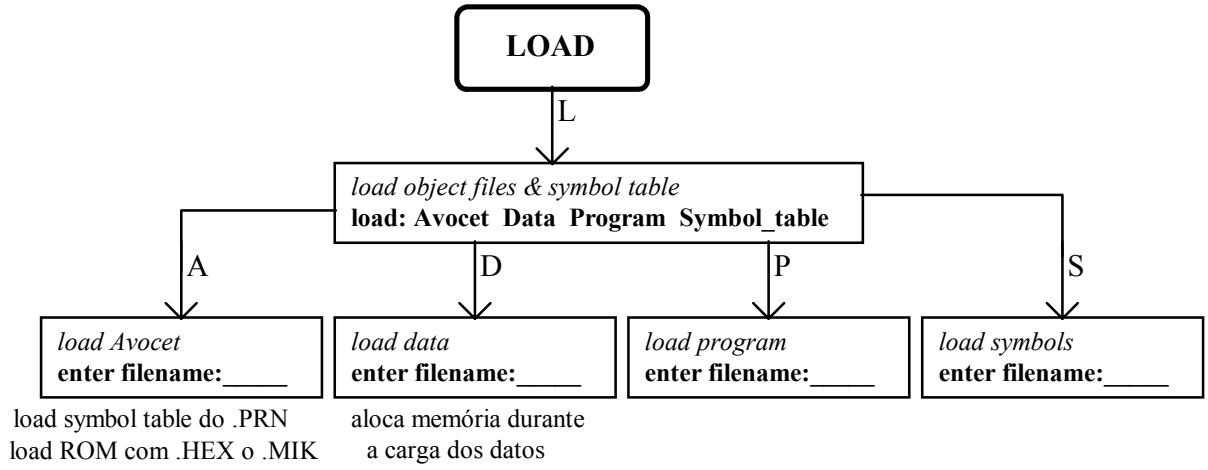
em negrito → as solicitações de entradas, seja através de letra, highlight ou expressão.

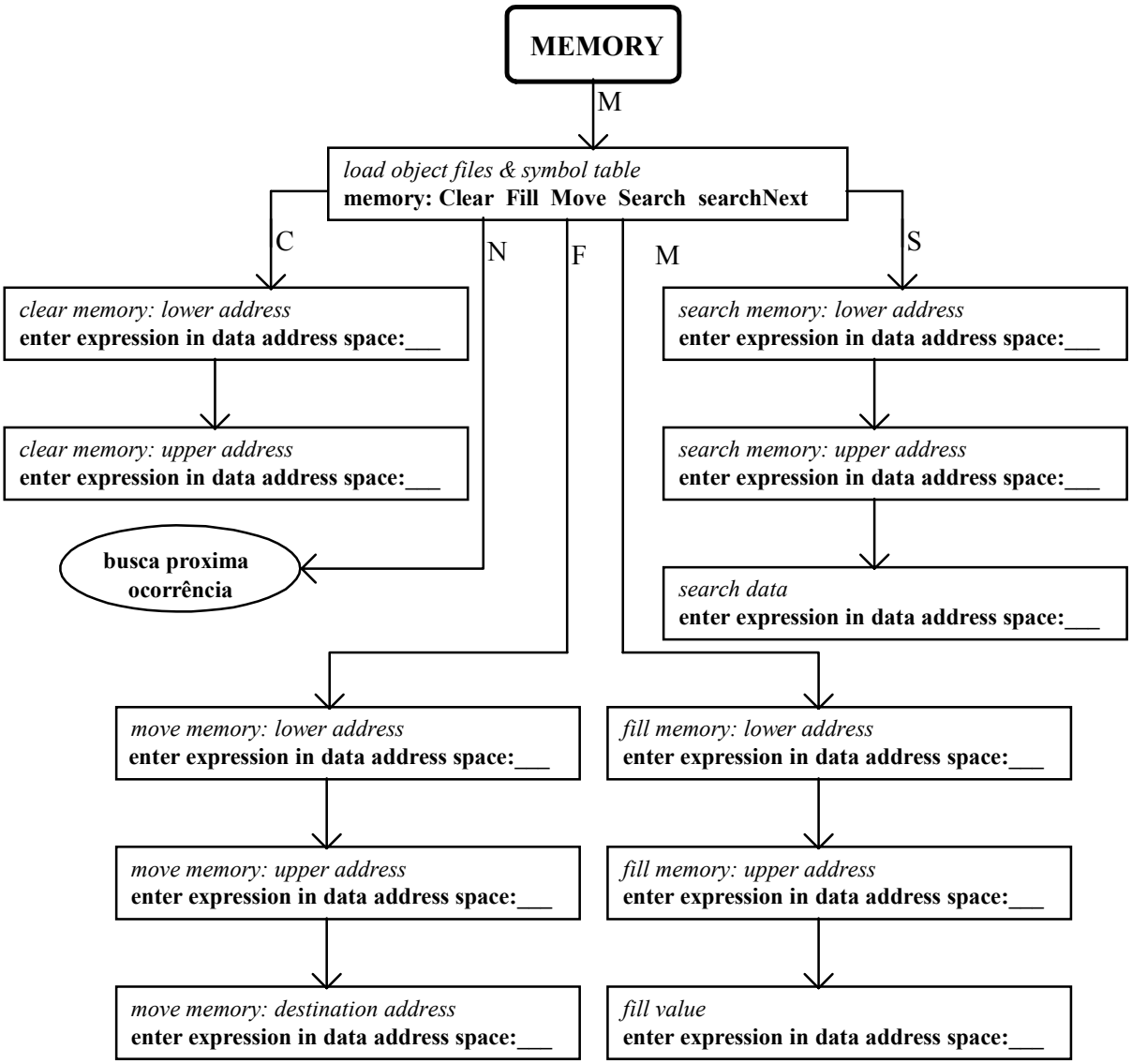

DUMP
D
configure memory dump windows
select dump area: 1 2
1
2
(upper or lower) dump window
dump: Absolute Indirect
A
I
Absolute
Indirect
dump start address
enter expression in data address space:___
dump indirect address or register
enter expression in data address space:___
index offset
enter expression in data address space:___

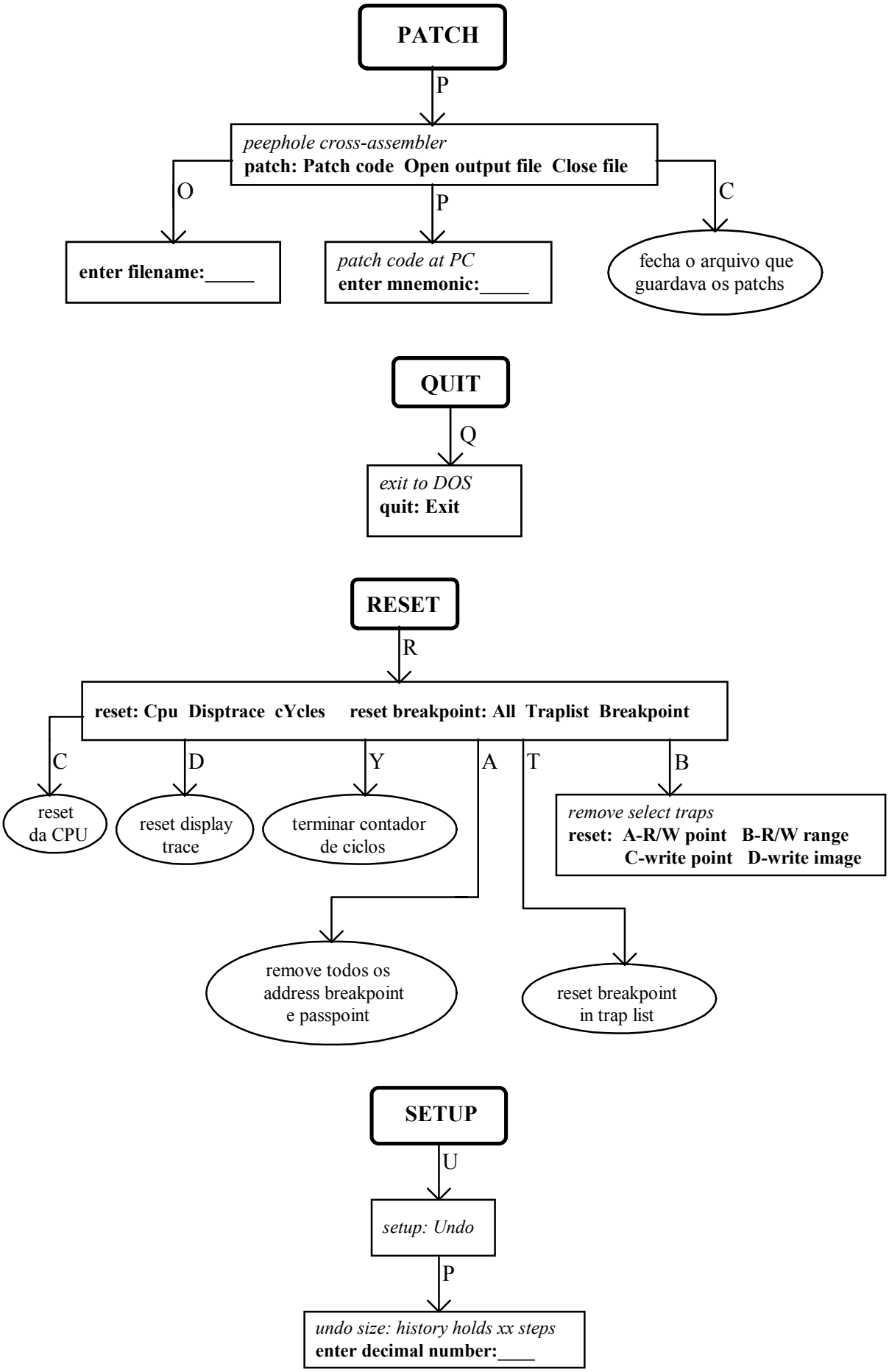

EXPRESSION
E
lugar onde a expressão será armazenada
enter expression:_____

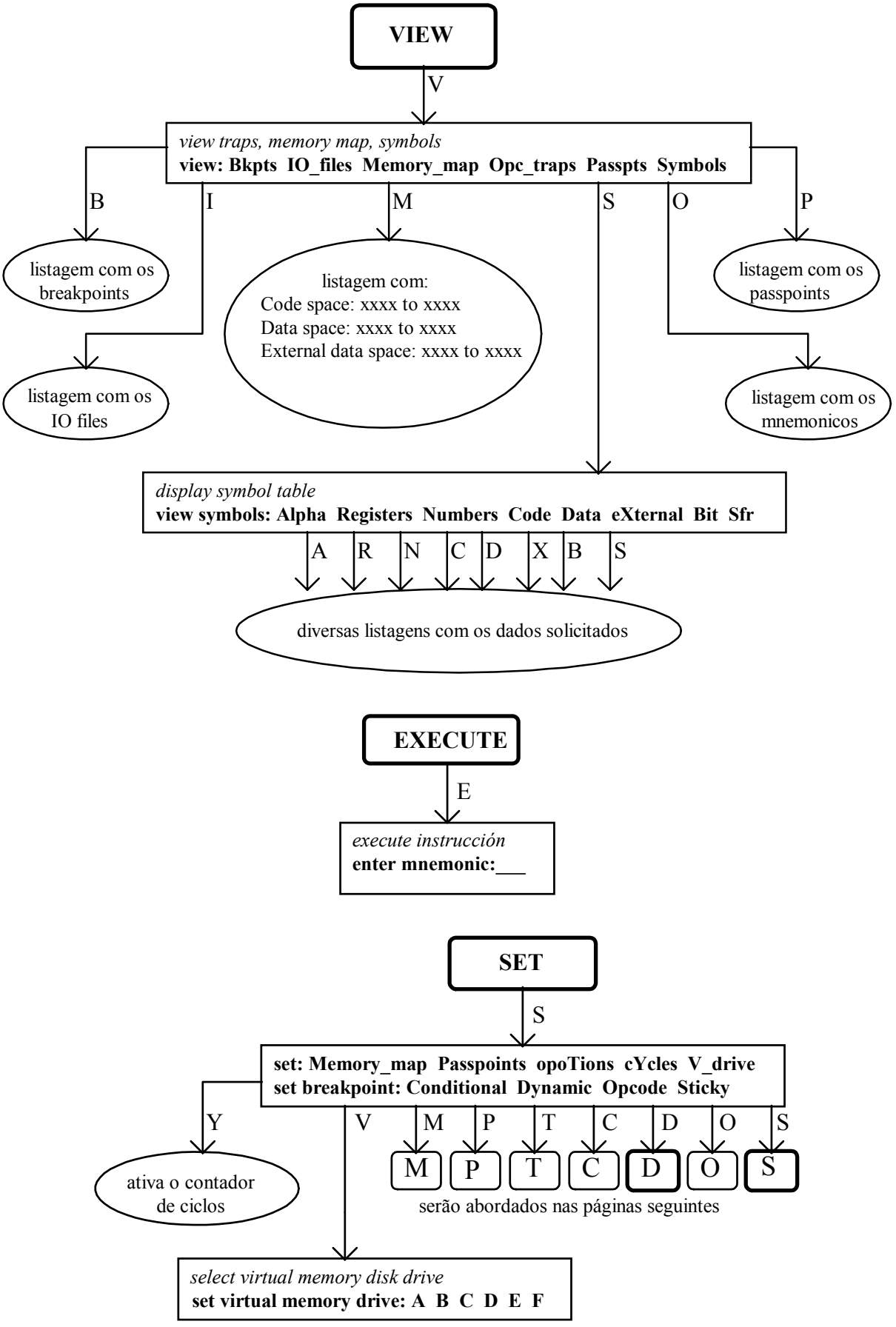

COMMAND FILE
F
load/save keypress sequences
command file: Load save: Open Close Restart
L
O
C
R
command file to load
enter filename:_____
command file to open
enter filename:_____
fecha
archivo
reinicaliza o arquivo
ou seja,
volta para o começo

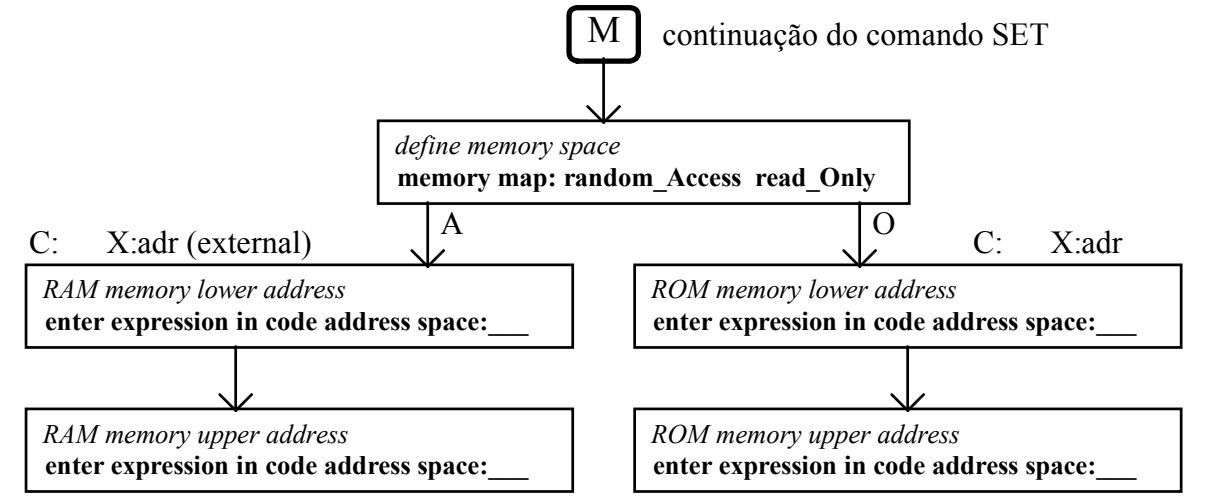

I / O
I
configure I/O file attachment
I/O file: Open Close
C
O
detach & close I/O file
close I/O files: (lista de arquivos) Yes No
I/O input file
enter filename:_____
replay input file at EOF: Yes No
Y
N
I/O output file
enter filename:_____
I/O trigger: Cycles Opcodes access
O
C
I/O rate: cycles per transfer
enter decimal number:_____
B
I/O transfer: Mapbits mapByte
M
I/O transfer: I/O bit, memory bit, address, dir:In/Out
enter expression:_____
I/O bit → qual bit do arquivo
memory bit → qual bit da RAM
address → endereço do byte onde está o bit

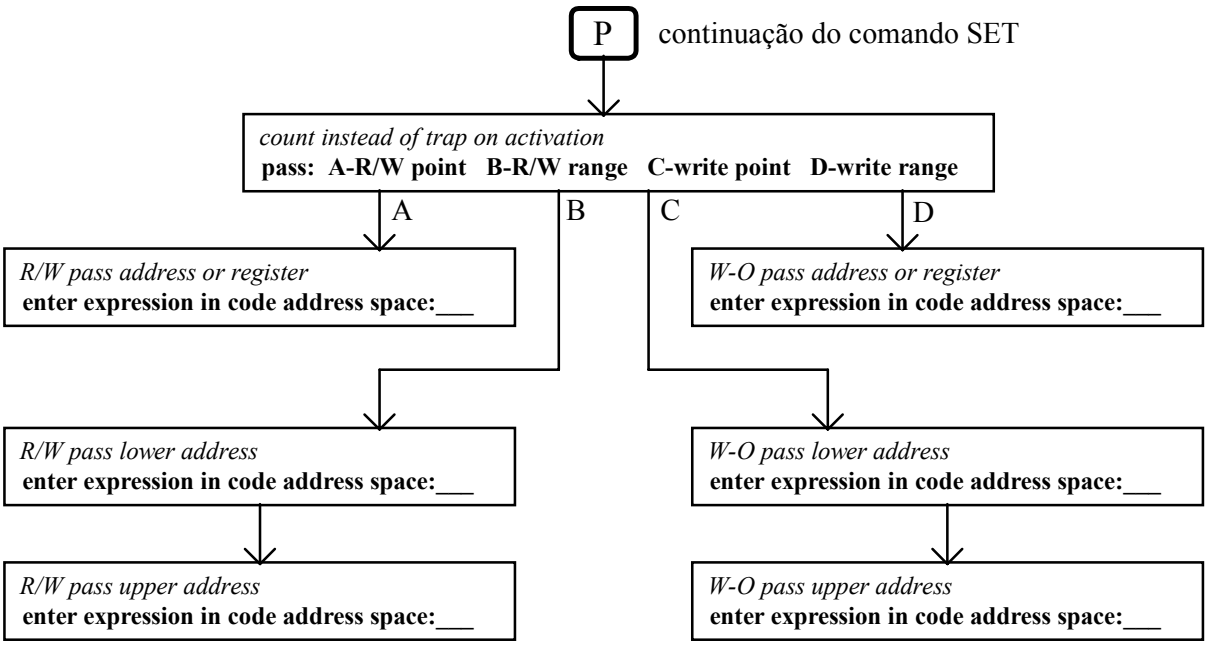

LOAD
L
load object files & symbol table
load: Avocet Data Program Symbol_table
A
D
P
S
load Avocet
enter filename:_____
load data
enter filename:_____
load program
enter filename:_____
load symbols
enter filename:_____
load symbol table do .PRN
load ROM com .HEX o .MIK
aloca memória durante
a carga dos datos

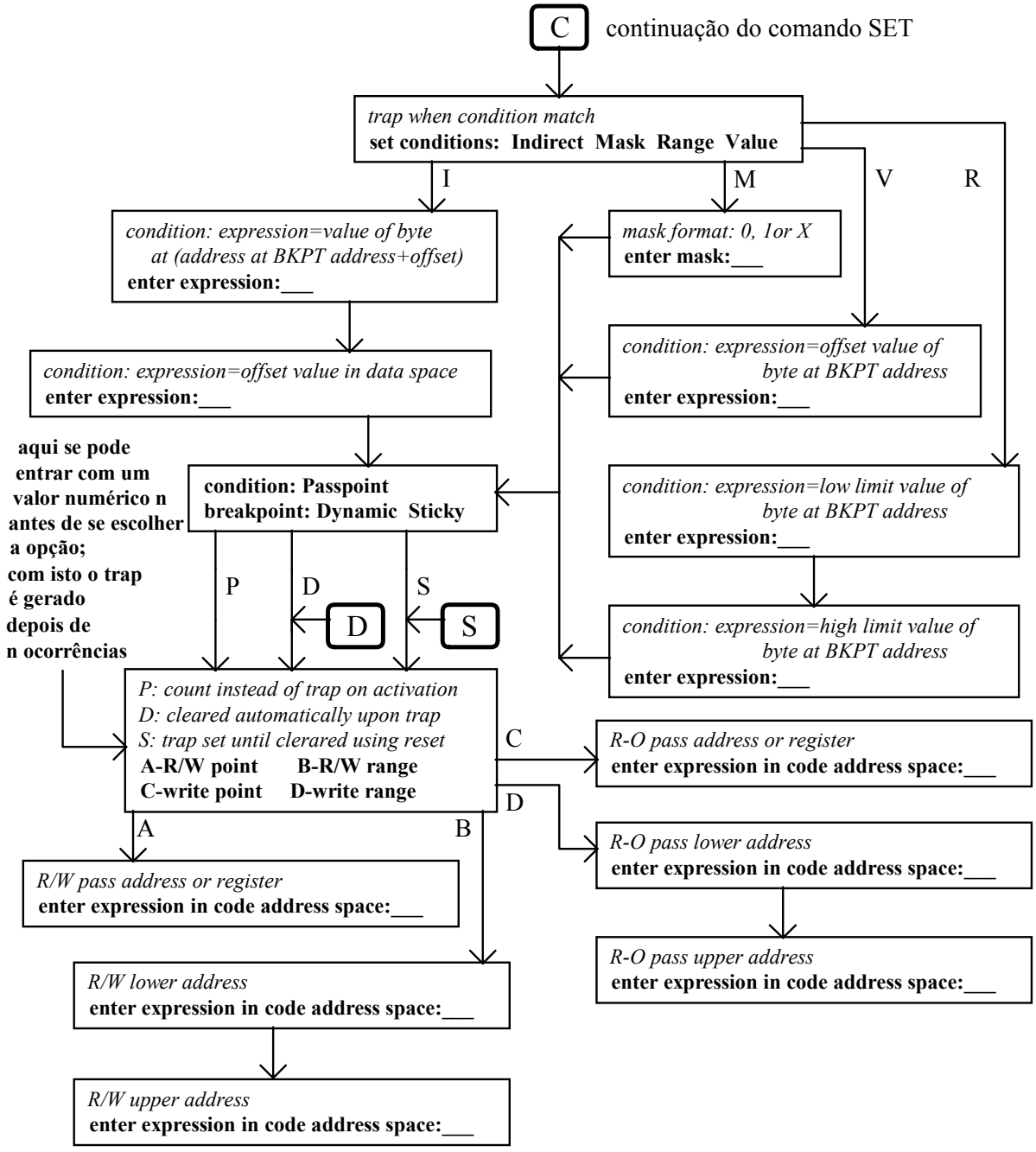

MEMORY
M
load object files & symbol table
memory: Clear Fill Move Search searchNext
C
N
F
M
S
clear memory: lower address
enter expression in data address space:___
clear memory: upper address
enter expression in data address space:___
busca proxima
ocorrência
search memory: lower address
enter expression in data address space:___
search memory: upper address
enter expression in data address space:___
search data
enter expression in data address space:___
move memory: lower address
enter expression in data address space:___
move memory: upper address
enter expression in data address space:___
move memory: destination address
enter expression in data address space:___
fill memory: lower address
enter expression in data address space:___
fill memory: upper address
enter expression in data address space:___
fill value
enter expression in data address space:___

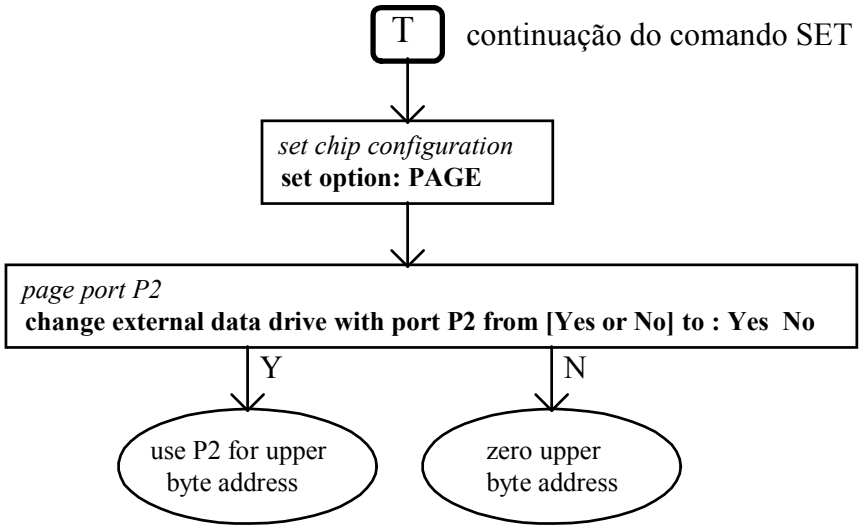

PATCH
P
peephole cross-assembler
patch: Patch code Open output file Close file
O
P
C
enter filename:_____
patch code at PC
enter mnemonic:_____
fecha o arquivo que guardava os patchs
QUIT
Q
exit to DOS
quit: Exit
RESET
R
reset: Cpu Disptrace cYcles reset breakpoint: All Traplist Breakpoint
C
D
Y
A
T
B
reset da CPU
reset display trace
terminar contador de ciclos
remove select traps
reset: A-R/W point B-R/W range
C-write point D-write image
remove todos os address breakpoint e passpoint
reset breakpoint in trap list
SETUP
U
setup: Undo
P
undo size: history holds xx steps
enter decimal number:____

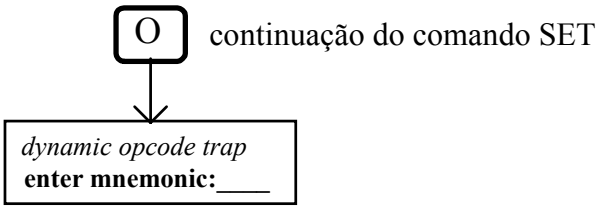

VIEW
V
view traps, memory map, symbols
view: Bkpts IO_files Memory_map Opc_traps Passpts Symbols
B
I
M
S
O
P
listagem com os breakpoints
listagem com os IO files
listagem com:
Code space: xxxx to xxxx
Data space: xxxx to xxxx
External data space: xxxx to xxxx
listagem com os passpoints
listagem com os mnemonicos
display symbol table
view symbols: Alpha Registers Numbers Code Data eXternal Bit Sfr
A R N C D X B S
diversas listagens com os dados solicitados
EXECUTE
E
execute instrucción
enter mnemonic:___
SET
S
set: Memory_map Passpoints opoTions cYcles V_drive
set breakpoint: Conditional Dynamic Opcode Sticky
Y
V
M P T C D O S
ativa o contador de ciclos
M P T C D O S
serão abordados nas páginas seguintes
select virtual memory disk drive
set virtual memory drive: A B C D E F

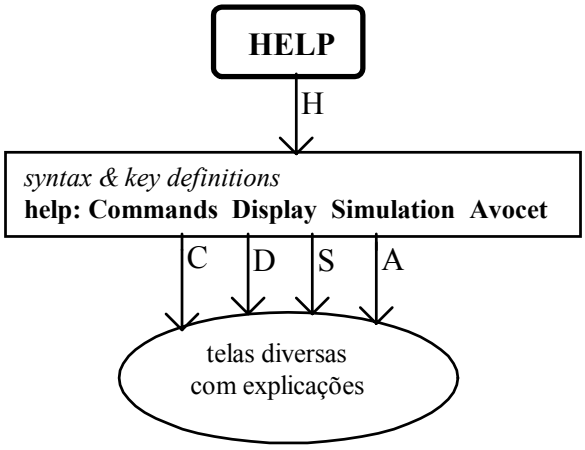

M
continuação do comando SET
define memory space
memory map: random_Access read_Only
A
O
C: X:adr (external)
C: X:adr
RAM memory lower address
enter expression in code address space:___
ROM memory lower address
enter expression in code address space:___
RAM memory upper address
enter expression in code address space:___
ROM memory upper address
enter expression in code address space:___


P
continuação do comando SET
count instead of trap on activation
pass: A-R/W point B-R/W range C-write point D-write range
A
B
C
D
R/W pass address or register
enter expression in code address space:___
W-O pass address or register
enter expression in code address space:___
R/W pass lower address
enter expression in code address space:___
W-O pass lower address
enter expression in code address space:___
R/W pass upper address
enter expression in code address space:___
W-O pass upper address
enter expression in code address space:___


C
continuação do comando SET
trap when condition match
set conditions: Indirect Mask Range Value
I
M
V
R
condition: expression=value of byte at (address at BKPT address+offset)
enter expression:___
mask format: 0, 1or X
enter mask:___
condition: expression=offset value in data space
enter expression:___
condition: expression=offset value of byte at BKPT address
enter expression:___
aqui se pode entrar com um valor numérico n antes de se escolher a opção; com isto o trap é gerado depois de n ocorrências
condition: Passpoint
breakpoint: Dynamic Sticky
condition: expression=low limit value of byte at BKPT address
enter expression:___
P
D
S
D
S
condition: expression=high limit value of byte at BKPT address
enter expression:___
P: count instead of trap on activation
D: cleared automatically upon trap
S: trap set until clerared using reset
A-R/W point B-R/W range
C-write point D-write range
C
R-O pass address or register
enter expression in code address space:___
D
A
B
R-O pass lower address
enter expression in code address space:___
R/W pass address or register
enter expression in code address space:___
R-O pass upper address
enter expression in code address space:___
R/W lower address
enter expression in code address space:___
R/W upper address
enter expression in code address space:___


T
continuação do comando SET
set chip configuration
set option: PAGE
page port P2
change external data drive with port P2 from [Yes or No] to : Yes No
Y
N
use P2 for upper byte address
zero upper byte address


O
continuação do comando SET
dynamic opcode trap
enter mnemonic:____


HELP
H
syntax & key definitions
help: Commands Display Simulation Avocet
C
D
S
A
telas diversas com explicações

5.9. TELAS DE AJUDA

5.9.1. Ajuda para os Comandos

5.9.2. Ajuda para o Display

5.9.3. Ajuda para a Simulação

RUN:

F1 ➔ Go Toggle : roda simulação

F10 ➔ Single step : roda passo a passo

F9 ➔ Undo : desfaz última instrução

BKPT:

F2 ➔ move o cursor do breakpoint para cima

F4 ➔ move o cursor do breakpoint para baixo

F3 ➔ BKP SET: seta um breakpoint dinâmico no cursor

F5 ➔ Speed: velocidade de simulação

ALT+F5 ➔ Label Toggle "Label": endereços e operandos simbolicamente
"ADDR": sem símbolos na desassemblagem

F6 ➔ Display Toggle "ON": a tela é atualizada a cada instrução
"OFF": somente janelas de trace são atualizadas até que um trap ocorra

ALT+F6 ➔ Trace Toggle "ON": a janela é atualizada mesmo quando o display está OFF
"OFF": a janela é atualizada quando o disply está ON

F8 ➔ Skip/Toggle: sai ou entra passo a passo em instruções CALL

Teclas de controle de modo:

ESC ➔ alterna modo display / comando

F7 ➔ modo do cursor: Hex, ASCII, BIN

CTRL+PgUp ➔ alterna o modo de scroll (rolamento)

Movimento do cursor:

setas do teclado ➔ movem 1 posição em cada sentido

HOME/END ➔ primeiro/último caracter da janela

PgUp/PgDn ➔ rola uma janela para cima/baixo

Movimento entre janelas:

RETURN ➔ vai para a última posição alterada

CTRL + $\rightarrow$: move para a janela à direita

CTRL + $\leftarrow$: move para a janela à esquerda

Teclas de edição de objetos:

+/- ➔ incrementa/decrementa byte/word/flag

Ins ➔ alterna byte/nibble/bit (dependendo da posição)

CTRL-END ➔ apaga da posição para o final do objeto

CTRL-HOME ➔ apaga o objeto inteiro

5.9.4. Ajuda para o AVOCET

5.9.5. Tela do Simulador