19. INTERRUPÇÕES

O controlo simultâneo de vários acontecimentos externos por parte do CPU, recorrendo exclusivamente ao teste continuado de entradas, torna o desenho das aplicações complexo, principalmente se tivermos que contabilizar tempos e realizar vários outros processamentos em simultâneo com a observação dos vários eventos externos. Para contornar esta dificuldade, os CPUs em geral, põem disponível uma entrada IRQ (*Interrupt Request*), através da qual se pode accionar um mecanismo denominado por interrupção. Este mecanismo consiste no seguinte: o CPU, sempre que termina a execução de uma instrução, testa o pino de entrada IRQ e caso este se encontre activo, gera uma chamada para uma rotina ISR (*Interrupt Service Routine*) que executará a acção pretendida pelo dispositivo que promoveu o pedido de interrupção, retornando em seguida ao programa interrompido.

Este mecanismo tem várias implicações na arquitectura do processador uma vez que são necessários assegurar os seguintes itens:

- Dar à aplicação o controlo de inibição e permissão para o CPU aceitar o pedido de interrupção;
- Quando ocorrer a interrupção suspender a execução em curso;
- Preservar o valor do PC corrente para assegurar o retorno ao programa interrompido, bem como do estado de execução do CPU, ou seja, preservar os valores dos vários registos do CPU;
- Inibir o CPU de aceitar nova interrupção.
- Vectorizar o CPU para a ISR;

Cada um dos itens tem diversas formas de resolução dependente da arquitectura CISC ou RISC e do fabricante do CPU.

19.1 Permissão e Inibição

Quanto à inibição e permissão de interrupção, ou são disponibilizadas instruções específicas, ou se utiliza um bit específico de um registo do programador. Como é fácil entender o CPU ao atender uma interrupção tem que ficar automaticamente inibido de interrupções para poder prosseguir executando instruções da ISR sem ser interrompido descontroladamente. Este facto levanta um problema no retorno ao programa interrompido, uma vez que é necessário repor na íntegra o estado de execução do CPU, inclusive o estado de aceitação das interrupções, dai que o retorno e a permissão de interrupção tenham que ser feita de forma indivisível.

19.2 Preservar estado de execução

Quanto ao preservar do valor corrente do PC, é por norma utilizado o mecanismo disponível no CPU para dar suporte à implementação de rotinas (STACK ou registo LINK). Quanto à preservação do valor dos restantes registos do CPU ou fica ao cuidado da ISR utilizando uma qualquer estrutura de dados, ou existe uma duplicação de vários registos do CPU e que são comutados no momento da interrupção.

19.3 Vectorização

Depois de ser salvo o contexto do programa interrompido procede-se à vectorização para a ISR e que consiste em colocar como conteúdo do registo PC o endereço da ISR associada à interrupção.

O estabelecimento do endereço da ISR pode ser feito de várias formas, como por exemplo:

- Valor constante estabelecido pelo fabricante do CPU;
- Leitura a partir do bus de uma instrução (Intel 8085);
- Leitura a partir do bus de um índice de uma tabela onde são estabelecidos os endereços das respectivas ISRs (Pentium).

19.4 Retorno de interrupção

No retorno ao programa interrompido a arquitectura tem que providenciar mecanismos que possibilitem simultaneamente o retorno e a reposição do estado de aceitação de interrupções. E novamente aqui existem várias soluções: ou uma instrução de permissão de interrupções com efeito retardado para possibilitar a execução da instrução de retorno com o CPU ainda inibido, ou uma instrução específica que execute de forma integrada a reposição do estado do processador e o retorno ao programa interrompido. Também se pressupõe que o processamento realizado pela ISR levou a que o periférico que gerou a interrupção retire o pedido.

19.5 Pedido de interrupção

O CPU pode disponibilizar uma ou mais entradas para pedido de interrupção (*Interrupt Request*). Estas entradas podem ter várias naturezas de excitação, nomeadamente: *level-sensitive*, *edge-trigger-sensitive*, *edge-level-sensitive*. Esta variedade de sensibilidades está directamente associada ao tipo de dispositivos que requerem interrupção. As entradas de interrupção com natureza *level-sensitive* só pode ser utilizada por periférico que tenham a capacidade de retirar o pedido de interrupção quando forem acedidos pela ISR, caso contrário, deverão utilizar entradas de interrupção com sensibilidade *edge-trigger-sensitive* ou *edge-level-sensitive*, evitando desta forma que o pedido ainda permaneça no momento de retornar da ISR.

19.6 Interrupções no PDS16

O PDS16 põe disponível uma entrada de IRQ denominada nEXINT, activada com valor lógico zero e com natureza de excitação *level-sensitive*, ou seja, requer interrupção enquanto a entrada permanecer activa. Como já foi anteriormente referido, as entradas de interrupção com este tipo de sensibilidade requerem que o periférico não mantenha o pedido no momento de retornar da ISR (ver exemplo 1). O PDS16 amostra a entrada nEXINT a meio de T4 como mostra a Figura 19-1, e caso esteja activa, transita de T4 para estado de interrupção TINT, onde desenvolve as várias acções que lhe estão associadas e que adiante descreveremos. A passagem do CPU pelo estado de interrupção é assinalada nos bits de estado S1, S0, possibilitando que o dispositivo que requereu interrupção seja informado antecipadamente do facto da aceitação da interrupção por parte do CPU e assim dispor de mais tempo antes de ser acedido. Esta informação, como veremos mais adiante, é utilizada pelos dispositivos geradores e gestores de interrupções.

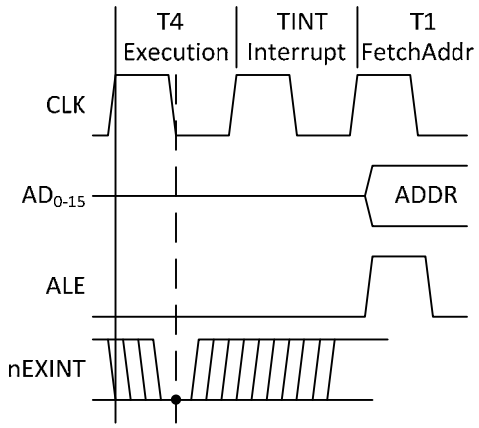

Figura 19-1 Diagrama temporal do estado de interrupção

A introdução do mecanismo de interrupções no PDS16 implica alterações à estrutura inicial para poder responder aos vários itens anteriormente descritos. Estas alterações terão uma complexidade acrescida, pelo facto de se pretender que a estrutura responda ao esquema de prioridades *fully nested*. Um esquema de prioridades *fully nested* permite que uma ISR seja interrompida por uma interrupção mais prioritária. Assim sendo, será criado um segundo banco de registos, com a estrutura mostrada na Figura 19-2, e serão adicionados dois bits ao registo PSW, um para inibição e permissão de interrupções denominado IE (*Interrupt Enable*) e outro, denominado BS (*Bank Select*), que estabelece qual dos bancos de registos se encontra seleccionado (aplicação ou interrupção). Para responder a esta especificação o *register file,* passa a ter a estrutura apresentada na Figura 19-3.

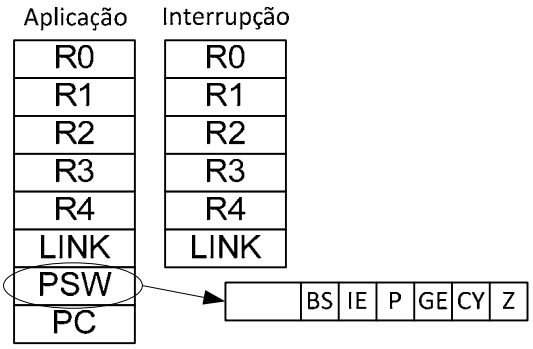

Figura 19-2

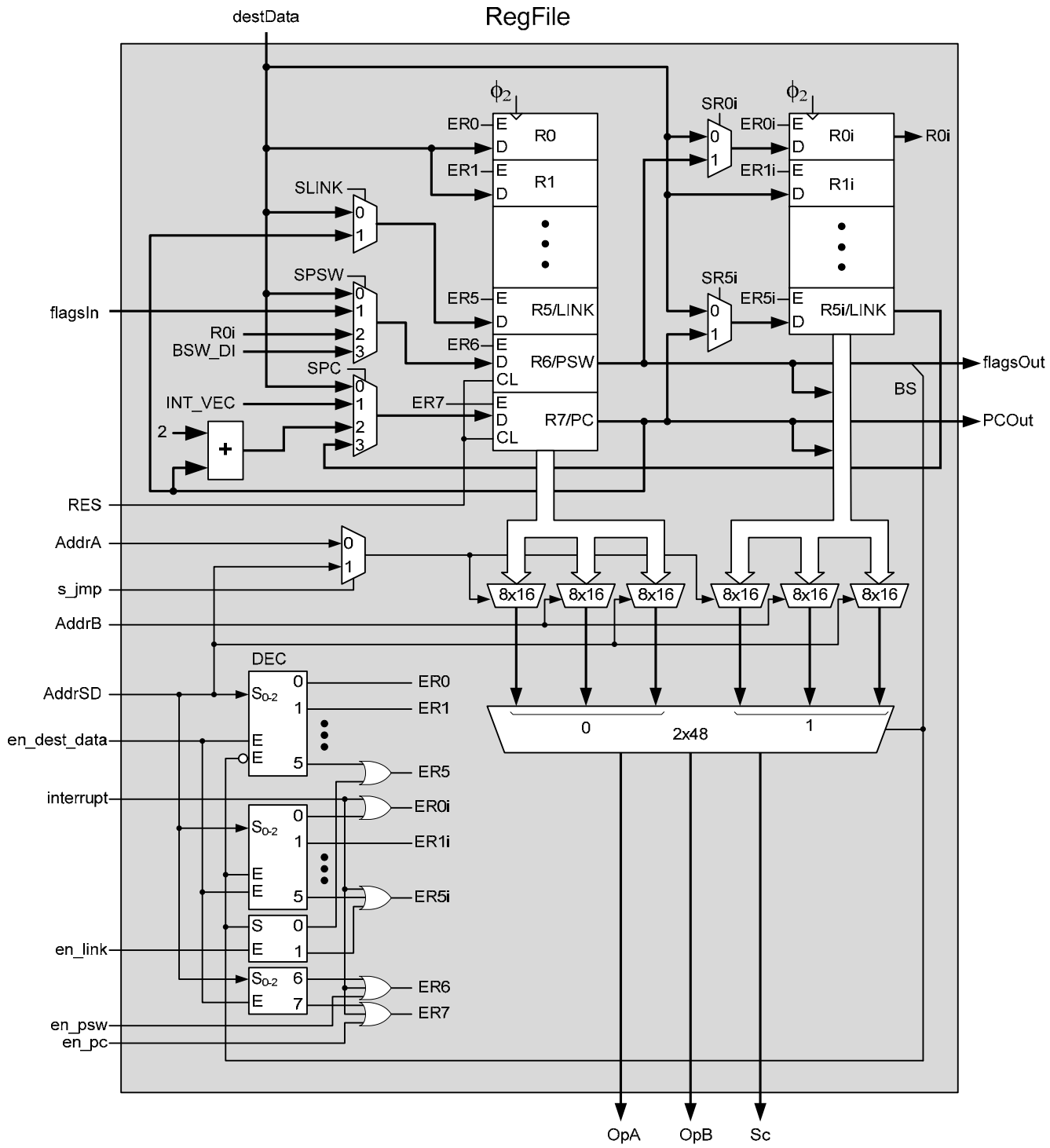
RegFile
destData
ER0
ER1
ER5
ER6
ER7
R0
R1
R5/LINK
R6/PSW
R7/PC
SR0i
SR5i
ER0i
ER1i
ER5i
R0i
R1i
R5i/LINK
SLINK
SPSW
SPC
flagsIn
R0i
BSW_DI
INT_VEC
2
RES
AddrA
s_jmp
AddrB
AddrSD
en_dest_data
interrupt
en_link
en_psw
en_pc
DEC
8x16
2x48
BS
flagsOut
PCOut
OpA
OpB
Sc

Figura 19-3

19.6.1 Processamento da interrupção no PDS16

Aquando de uma interrupção o CPU realiza as seguintes acções em simultâneo:

- Copia o valor do registo PSW para o registo R0 do banco de interrupção;
- Coloca a 1 o bit BS do registo PSW tornando desta forma activo o banco de interrupção;
- Coloca a zero o *bit* IE do registo PSW inibindo assim as interrupções;
- Copia o valor do registo PC para o registo LINK do banco de interrupção;
- Coloca o valor 2 no registo PC, vectorizando desta forma o processamento para o ponto de entrada da interrupção.

É de lembrar, que os vários registos são *edge-trigger* sincronizados por ϕ_2, e que por essa razão assiste-se como que a um deslocamento de informação entre os vários registos envolvidos.
Para garantir indivisibilidade entre as várias acções que são necessárias realizar para o retorno da interrupção foi criada a instrução IRET, que copia em simultâneo o valor do registo LINK do banco de interrupção para o registo PC, e copia o valor do registo R0 do banco de interrupção para o registo PSW. Se admitirmos que o registo R0 do banco de interrupção mantém o valor de PSW no momento da interrupção, então é reposto o estado do programa interrompido e a permissão às interrupções e é também restabelecido como banco seleccionado o banco de aplicação. Após o retorno da interrupção se a entrada EXINT estiver activa executa ainda uma instrução do programa interrompido.

19.7 Gestão de interrupções

É natural nos sistemas baseados em microprocessadores existirem várias fontes de interrupção. Como o CPU põe disponível uma única entrada para pedido de interrupção, o sistema deverá dispor de um dispositivo periférico que faça a gestão dos vários pedidos de interrupção contemplando: as diferentes sensibilidades, a simultaneidade dos pedidos, a prioridade entre cada um dos pedidos e gerar interrupção ao CPU. Este periférico, normalmente denominado por PIC (*Peripheral Interrupt Controller*) põe disponíveis várias entradas de interrupção e permite através de programação:

- Definir a sensibilidade da excitação de cada uma das entradas de interrupção (*Level*, *Edge trigger* ou *Edge Level*);
- Inibir e desinibir individualmente cada entrada de interrupção;
- Definir um qualquer esquema de prioridades (*Linear, Round Robin, Rotation, Fully Nested,* etc.) que assegure que, de entre os vários pedidos de interrupção presentes, é indicado ao CPU o índice do mais prioritário segundo o esquema de prioridades estabelecido.

19.8 PIC_V1

A título de exemplo, pensemos num PIC que denominaremos por PIC_V1 com a seguinte especificação:

- Suporta quatro entradas de interrupção;
- Permite definir a sensibilidade de excitação de cada entrada (*Level sensitive*, *Edge trigger*);
- Permite mascarar individualmente cada uma das entradas de interrupção;
- Implementa o esquema de prioridades linear.

19.8.1 Tipo de excitação

As entradas de interrupção IRQ0-3 podem ser individualmente programadas para gerarem pedidos de interrupção quando o valor lógico presente na entrada transita de 0 para 1 (*edge-triggered-sensitive*), ou enquanto se mantêm a 1 (*level-sensitive*). A definição do tipo de excitação, faz-se colocando a zero, ou a um, os bits do registo CRT_S (*Interrupt Sensitive*).

19.8.2 Inibição de interrupção

Cada uma das fontes de interrupção pode individualmente ser desinibida ou inibida colocando a um, ou a zero, o respectivo bit no registo **CTR_M** (*Interrupt Mask*).

19.8.3 Prioridade das interrupções

O esquema de prioridades implementado pelo PIC_V1 é um esquema de prioridades linear, ou seja, o **IRQ** mais prioritário é o **IRQ3** e o menos prioritário é o **IRQ0**. Caso a sensibilidade à interrupção seja *edge-triggered,* a leitura do vector por parte do CPU, implica colocar a zero o pedido de interrupção.

19.8.4 Interrupção espúria

Dado que as entradas de interrupção podem ser programadas com sensibilidade *level-sensitive*, pode acontecer a geração de interrupções espúrias por parte do PIC_V1. Este tipo de interrupções acontece, quando a entrada que pediu interrupção e que por acaso é única, é desactivada no intervalo de tempo que medeia entre o momento em que o CPU atende a interrupção e o momento em que lê o vector de interrupção do PIC. Tal situação leva o PIC a gerar o vector de interrupção zero. Para ultrapassar este problema o PIC_V1 constrói o vector de interrupção com três bits sendo o bit de maior peso controlado pelo sinal **nINTP** e os dois de menor peso pelo *priority encoder*. Desta forma uma interrupção espúria terá como vector o valor 4.

Na Figura 19-4 é apresentado o diagrama de blocos do PIC_V1.

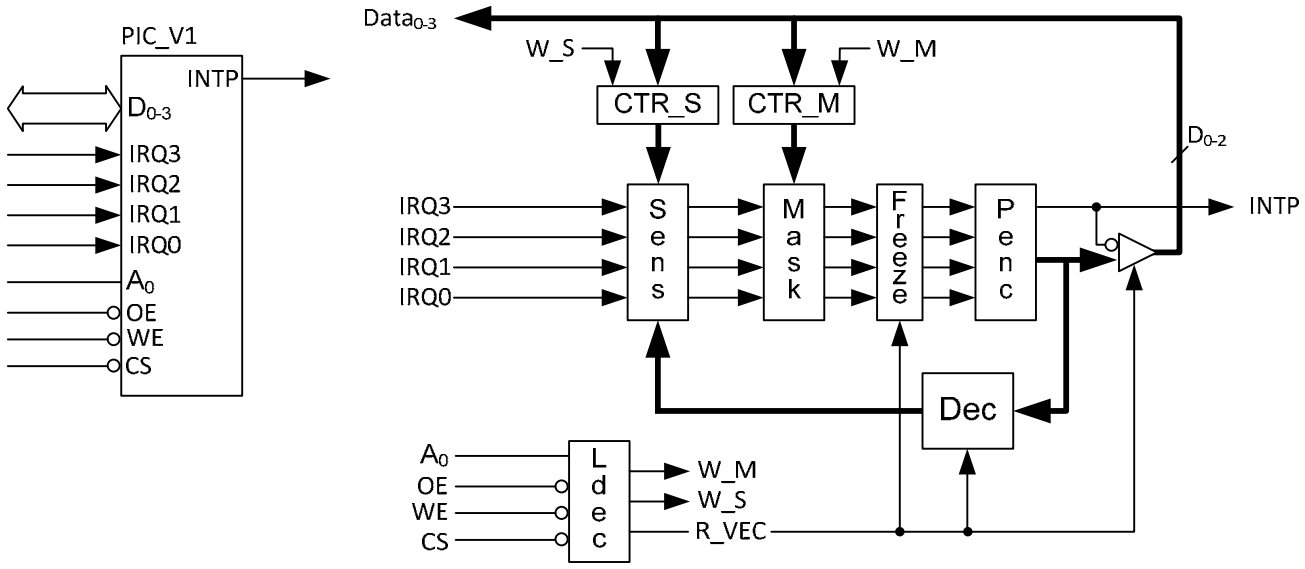

Figura 19-4

Na Figura 19-5 é apresentada uma possível implementação do PIC_V1.

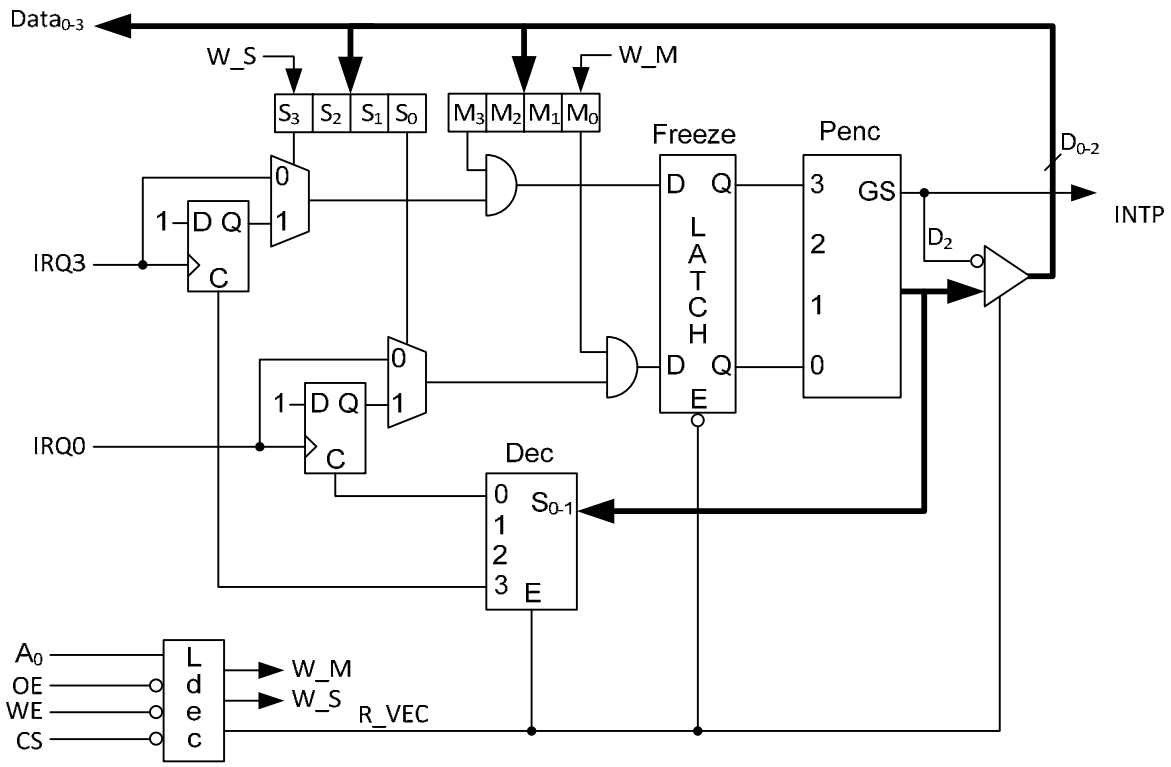

Figura 19-5

Nota: A existência do registo *Freeze* (congelar) tem como objectivo garantir a estabilidade do bus de dados durante a leitura do vector da interrupção por parte do CPU. A não existência deste registo levaria a que a chegada de uma interrupção de nível superior ao que estava a ser indicado no momento de conclusão da leitura do vector, poderia produzir um vector errado, isto é, que não correspondesse nem à nova interrupção nem à interrupção preterida (ex: interrupção preterida ser a 1, a nova interrupção ser a 2, pode levar o *priority encoder* a produzir momentaneamente o valor 0 ou 3).

Uma outra característica importante que a estrutura apresenta é a de permitir a discussão das prioridades até ao momento da leitura do vector, ou seja, permite que desde o momento em que se gerou o pedido de interrupção até que o CPU leia o vector, sejam aceites interrupções mais prioritárias que aquela que originou o pedido.

Como se pode observar na Figura 19-5, a leitura do vector de interrupção, coloca a zero o *flip-flop* que lhe está associado.

ISR (*Interrupt Service Routine*)

A rotina que executa as acções associadas a uma interrupção é frequentemente denominada de ISR (*Interrupt Service Routine*). Na entrada de interrupção (endereço 2) dado que é um espaço de acesso directo, não deverá ser ocupado pelo código da ISR. Neste endereço deverá existir um redireccionamento

(**jmp**) para a ISR ou para uma rotina de *switch* caso o elemento que requereu a interrupção seja um PIC. Vejamos alguns exemplos de exploração da interrupção.

19.9 Exercícios

Exemplo 1:

Activar uma saída A, se após a activação de um botão B, for gerado pelo botão B um determinado número de impulsos num intervalo de 10 segundos. Como mostra a Figura 19-6, a solução apresentada utiliza o botão B ligada à entrada de interrupção através de um *flip-flop* tipo D *edge-trigger*. Esta solução tem como objectivo garantir que a ISR só evocada uma vês por cada activação de B. O tempo de 10 segundos é contabilizado utilizando uma rotina de temporização por software. A solução mostra ainda a utilização de uma estrutura stack para salvar o PSW e o endereço de retorno, no sentido de permitir o aninhamento de funções, no caso deste exemplo a função `clearIntRequest()`. Admita que a frequência de *clock* do CPU é de 1KHz / T=1ms

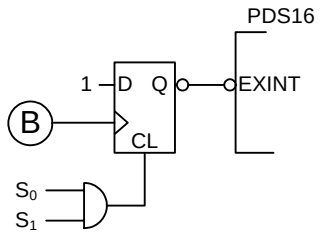

Figura 19-6

```
      .equ  IO_PORT_ADDR,0xff00
      .equ  IE_MASK,10000b
      .equ  ID_MASK,00000b
      .equ  CONST_1_SEG,125
      .equ  VAL_CONT_IMPULSOS,6

      .section start
      .org  0
      jmp   main
      jmp   isrEntry

      .data
sp:   .space      2

cont_impulsos:
      .space      1

      .text
; void clearIntRequest()
clearIntRequest:
      ldih  r0,#high(IO_PORT_ADDR)
      ldi   r1,#1             ;clear flip-flop
      stb   r1,[r0,#1]
      ldi   r1,#0
      stb   r1,[r0,#1]
      ret

isrEntry:
      ld    r1,sp
;push LINK
      st    r5,[r1,#0]
;push PSW
      st    r0,[r1,#1]
;update SP
      addf  r1,r1,#4
```

```
      st    r1,sp
      jmpl  clearIntRequest         ; clear interrupt requeste
      ldb   r0,cont_impulsos        ; incrementa contador de impulsos
      inc   r0
      stb   r0,cont_impulsos
isrReturn:
      ld    r1,sp
      subf  r1,r1,#4
      ld    r0,[r1,#1]  ; pop PSW
      ld    r5,[r1,#0]  ; pop LINK
      st    r1,sp
      iret

; void delay(int segundos)
delay:
      ldi   r1,#CONST_1_SEG
delay1:
      dec   r1                ;4ms
      jnz   delay1            ;4ms
      dec   r0
      jnz   delay
      ret

main:
;iniciar stack pointer de interrupção
      ldi   r0,#low(stack)
      ldih  r1,#high(stack)
      orl   r0,r0,r1
      st    r0,sp
main_1:
      ldi   r0,#0
      stb   r0,cont_impulsos
      jmpl  clearIntRequest
;desinibir interrupções
      ldi   PSW,#IE_MASK
main_2:
      ld    r0,cont_impulsos  ;esperar pela activaçao de B
      anl   r0,r0,r0
      jz    main_2            ;if (cont_impulsos==0)
      ldi   r0,#10            ;delay 10 segundos
      jmpl  delay
;inibir interrupções
      ldi   PSW,#ID_MASK
      ldb   r1,cont_impulsos
      ldi   r0,#VAL_CONT_IMPULSOS
      subr  r0,r0,r1
      jnz   main_1            ;if (cont_impulsos != VAL_CONT_IMPULSOS)
      jmpl  actuar_A          ;rotina que realizaria a activação da saída A
      jmp   main_1

stack:
```

Exemplo 2:

Pretende-se contar o número vezes que os sinais X e Y produzem a sequência mostrada na Figura 19-7 b), ou seja, uma transição descendente em X quando Y está a um, seguida de um impulso negativo em Y enquanto X ainda permanece em zero. Para detecção da sequência vamos utilizar a entrada de interrupção num esquema de amostragem dos sinais X e Y. Para tal, a entrada de interrupção é activada por uma onda quadrada com a frequência de 10Hz que se considera superior ao ritmo de variação dos sinais X e Y. Os sinais X e Y são observados através dos bits de menor peso do porto de entrada do SDPD16 Figura 19-7 c). Para termos interrupções a cada transição ascendente do sinal de 10Hz, é necessário conferir natureza edge trigger à entrada de interrupção como mostra a Figura 19-7 a). Para detectar a sequência dos sinais X e Y tal como mostra a figura b), a ISR implementa uma máquina de estados com o diagrama de transição de estados mostrado na Figura 19-7 d).

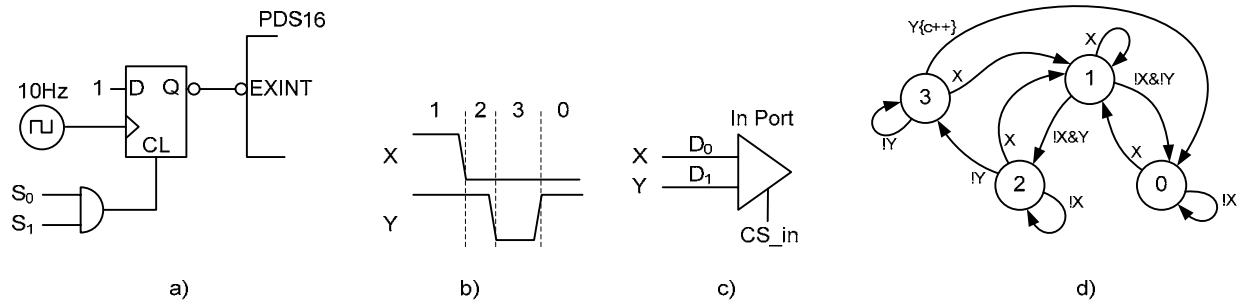

Figura 19-7

```
        .section start
        .org      0
        jmp       main
        ld        r7,[r7,#0]
        .word     intSwitchEntry      ; Admitindo que o corpo do main é muito extenso
                                      ; e que o offset do jump não é suficiente para
                                      ; atingir a referência intSwitchEntry

        .equ      IO_PORT_ADDR,0xff00
        .equ      IE_MASK,10000b

        .data
switch_tab:
        .word     isr_stat0,isr_stat1,isr_stat2,isr_stat3
isr_stat:
        .space    1
psw:    .space    2

        .text
main:
        ldi       r0,#0
        st        r0,c
        st        r0,isr_stat
;desinibir interrupções
        ldi       PSW,#IE_MASK
        ...
```

```
corpo do main
       ...

intSwitchEntry:
       st       r0,psw
       ldih     r0,#IO_PORT_ADDR
       ldi      r1,#1              ;clear flip-flop
       stb      r1,[r0,#1]
       ldi      r1,#0
       stb      r1,[r0,#1]         ;clear flip-flop
       ldb      r0,[r0,#1]         ;r0 = in_port (parametro de entrada no estado)
       ldi      r1,#low(switch_tab);switch case
       ldb      r2,isr_stat
       ld       r7,[r1,r2]         ;switch case
c:     .space   2             ;contador de sequências
isr_stat0:                    ;Estado 0
       shr      r0,r0,#1,0
       jnc      isrReturn     ; if (!X)
go_stat_1:                    ; if (X)
       ldi      r1,#1
       st       r1,isr_stat
       jmp      isrReturn

isr_stat1:                    ;Estado 1
       shr      r0,r0,#1,0
       jc       isrReturn     ; if (X)
       shr      r0,r0,#1,0
       jnc      go_stat_0     ; if (!X & !Y)
       ldi      r1,#2         ; if (!X & Y)
       st       r1,isr_stat
       jmp      isrReturn
go_stat_0:
       ldi      r1,#0
       st       r1,isr_stat
       jmp      isrReturn

isr_stat2:                    ;Estado 2
       shr      r0,r0,#1,0
       jc       go_stat_1     ; if X
       shr      r0,r0,#1,0
       jc       isrReturn     ; if (!X & Y)
       ldi      r1,#3         ; if (!X & !Y)
       st       r1,isr_stat
       jmp      isrReturn

isr_stat3:                    ;Estado 3
       shr      r0,r0,#1,0
       jc       go_stat_1     ; if (X)
       shr      r0,r0,#1,0
       jnc      isrReturn     ; if (!Y)
       ld       r0,c          ; if (Y) c++
       inc      r0
       st       r0,c
       jmp      go_stat_0

isrReturn:
```

```
        ld        r0,psw
        iret
```

Exemplo 3:

Entrada de interrupção utilizando o gestor de interrupções PIC_V1. A ISR corre sem permissão de interrupções;

```
       .section start
       .org 0
       jmp   main
       jmp   int_entry

       .text
int_entry:
       ld    r1,sp
;push LINK
       st    LINK,[r1,#0]
;push PSW image
       st    r0,[r1,#1]
       addf  r1,r1,#4
;update SP
       st    r1,sp
;read interrupt index
       ld    r0,pic_base_addr
       ldb   r0,[r0,#0]
       ldi   r1,#7
       anl   r0,r0,r1
       add   r0,r0,r0    ;dobra por que são words
;r0= device_handler_addr
       ldi   r1,#low(isr_tab_addr)
       ldih  r2,#high(isr_tab_addr)
       orl   r1,r1,r2
;call device_handler
       ld    r0,[r0,r1]
       jmpl  r0,#0
;dev_handler ret intry point
       ld    r1,sp
       subf  r1,r1,#4
       ld    r0,[r1,#1]        ; pop PSW
       ld    LINK,[r1,#0]      ; pop LINK
       st    r1,sp
       iret
isr_tab_addr:
       .word isr0,isr1,isr2,isr3
       .word spurious_int
main:
;r0=PIC_BASE_ADDR
       ldi   r0,#low(PIC_BASE_ADDR)
       ldih  r1,#high(PIC_BASE_ADDR)
       orl   r0,r0,r1
       st    r0,pic_base_addr
;Programar PIC
       ldi   r1,#INT_SENS_DEF
```

```
      stb   r1,[r0,#CTR_S]
      ldi   r1,#INT_MASK
      stb   r1,[r0,#CTR_M]
;init stack pointer interrupt
      ldi   r0,#low(STACK_BASE_ADDR)
      ldih  r1,#high(STACK_BASE_ADDR)
      orl   r0,r0,r1
      st    r0,sp
;desinibir interrupções
      ld    PSW,#IE_MASK
;inicio da aplicação
```

Exemplo 4:

Entrada de interrupção utilizando o PIC mas implementando o critério de prioridades *Fully Nested*. A ISR corre com permissão de interrupções;

```
main() {
  initIntEntrySP();
  initDevHandlerDataStruct();
  int_mask=START_MASK
  initPic(int_mask,INT_SENS);
  initAplDataStract();
  enableInt();
  aplication();
}

void intEntry() {
  push(LINK);
  push(PSW);
  push(int_mask);
  index=readPIC();
  int_mask=tab_mask[index];
  setPicIntMask(int_mask);

  switchDevHand(index);

  int_mask=pop();
  setPicIntMask(int_mask);
  R0=pop();
  LINK=pop();
  reti();
}

devHandler_n() {
  loadSPdevHand_n();
  setAplRegBank();
  pushAll();
  enableInt();
  body();
  desableInt();
  popAll();
  setIntRegBank();
  return;
}
```

Exemplo 5:

Pretende-se construir baseado num microprocessador PDS16 um sistema envolvendo vários dispositivos físicos. Entre os vários dispositivos, existe um teclado de 16 teclas organizado em matriz espacial 4x4 e um mostrador constituído por dois *displays* de 7 segmentos + ponto decimal.
Pretende-se que a recolha de informação do teclado e a apresentação dos caracteres no mostrador se processe de forma paralela e independente da aplicação.
O processo de recolha de informação do teclado deverá ser feito através de um porto de saída de 4 bits e um porto de entrada também de 4 bits e obedecendo às seguintes especificações:

- O reconhecimento e identificação das teclas é realizado por software;
- A detecção é *edge-triggered*, e a disciplina *two-key-lockout;*
- Deve constituir um processo autónomo, ou seja, não deve requerer processamento por parte da aplicação;
- A informação recolhida do teclado é posta disponível à aplicação através de uma função `char getKey()`, que quando evocada devolve o valor numérico da tecla que tenha sido premida.
- Os valores das teclas que forem sendo premidas, caso a aplicação não as consuma de imediato, vão sendo armazenadas até ao limite de 8.

O mostrador é constituído por dois *displays* de 8 LEDs ligados em ânodo comum como é mostrado na Figura 19-8.

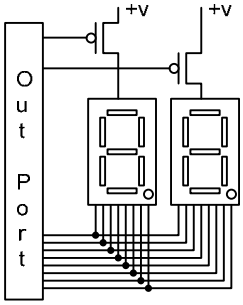

Figura 19-8

No sentido de minimizar o número de linhas de saída, é utilizado um esquema de multiplexagem das linhas que controlam os segmentos. Os oito segmentos de cada um dos *displays* estão ligados a um bus de 8 bits onde é colocada a informação a ser afixada. Existem duas linhas que controlam através de transístores o ânodo dos LEDs dos *displays* e que permitem activar o *display* ao qual a informação presente no bus se destina.
A afixação de informação no mostrador processa-se da seguinte forma: alternadamente (e a um ritmo suficientemente elevado para que não ocorra cintilação) vai-se colocando no bus informação para cada um dos *displays* e sincronizadamente activa-se o respectivo ânodo. Este refrescamento associado à persistência da visão humana resulta que os dois *displays* parecem encontrar-se activos em simultâneo.
O mostrador deverá apresentar a seguinte especificação:

- Refrescamento é realizado por software;
- Deve constituir um processo autónomo, ou seja, não deve requerer processamento por parte da aplicação;

Põe disponível uma função, `void DispWrite(char graf, char data, char dot)` que quando evocada desloca para a esquerda a informação presente no mostrador e afixa no dígito mais à direita do mostrador o carácter **`data`**.
O parâmetro **`graf`**, informa se **`data`** é um numérico hexadecimal ou um gráfico, correspondendo no caso de ser gráfico, o segmento **a** do display, ao bit D0 de **`data`** e o segmento **g** ao bit D6. No caso de ser numérico, é afixado um dígito hexadecimal.
No caso de ser numérico o valor de **`data`** é truncado de forma a ficar compreendido entre 0 e 15. O parâmetro **`dot`** informa se o dígito a afixar tem ou não ponto decimal.

Solução:
Porque o processo de reconhecimento e identificação de uma tecla premida, é autónomo assim como o refrescamento dos dígitos do mostrador, e ambos realizados por software, sem recorrer a periféricos específicos, a solução passa obrigatoriamente pelo uso do mecanismo de interrupção.
No caso do teclado poderemos adoptar uma de duas soluções: adicionar hardware externo para produzir interrupção aquando da actuação de uma tecla como mostra a Figura 19-9, ou análise da matriz, por interrupção temporizada.

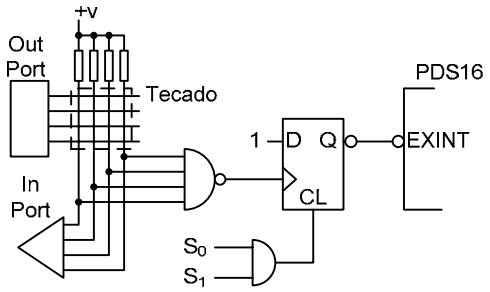

Figura 19-9

Vamos utilizar a segunda solução pois não necessita de hardware específico e permite realizar paralelamente outras tarefas, tais como por exemplo, o refrescamento do mostrador. Quanto às tarefas inseridas nesta interrupção é necessário acautelar o tempo gasto por estas, pois podem por em perigo o atendimento de outras interrupções por parte do CPU, uma vez que este se encontra com as interrupções inibidas durante o processamento da interrupção.
Para a implementação do processo de recolha de informação do teclado e refrescamento do mostrador, vamos utilizar um temporizador, que gerará interrupções com um ritmo superior ao da actuação de um teclado por um humano e garanta um refrescamento sem cintilação do mostrador, devendo ser no entanto o mais baixo possível para não diminuir a eficiência da aplicação.
Tendo em consideração que o tempo de reacção de um humano para actuação de uma tecla é da ordem das décimas de segundo e o ritmo de refrescamento de um dígito é no mínimo de 100Hz, optaremos pelo tempo de refrescamento que é o mais exigente. Uma vez que o mostrador é constituído por dois dígitos temos um tempo mínimo de refrescamento do mostrador de 5ms.

Processamento do teclado

Todo o processo de recolha de informação do teclado é realizado pela rotina de serviço da interrupção.

A ISR() implementa um autómato de três estados com o seguinte comportamento:

- 1º estado, aguarda que seja actuada uma tecla;
- 2º estado, descodifica e armazena o índice da tecla premida;
- 3º estado, aguarda que todas as teclas sejam desactivadas.

A detecção de que se existe alguma tecla premida, faz-se aplicando o valor lógico 0 a todas as linhas, e de seguida lê-se o porto de entrada testando se alguma coluna toma o valor lógico 0.

A identificação de qual a tecla premida, faz-se aplicando de forma sequencial o valor lógico 0 a cada uma das linhas da matriz, até que surja o valor lógico 0 numa das colunas. Quando tal acontece está determinada o índice da linha onde existe ligação entre linha e coluna. Em seguida, com a amostra das colunas que foi recolhida no passo anterior, verifica-se qual o índice da coluna que se encontra a zero, ficando assim encontrado o índice da coluna. Multiplicando o índice da linha pelo número de colunas da matriz e somando o índice da coluna fica determinado o índice na matriz onde se encontra ligado uma linha com uma coluna.

A solução que a seguir se descreve contempla a filtragem da intermitência de contacto produzida no fecho e abertura das teclas, fenómeno denominado por *bounce* (ver capitulo 6.8).

Esta intermitência, em teclados de baixa qualidade, pode ter várias transições com uma duração de cerca de 5ms, que levaria à detecção de actuações múltiplas de uma mesma tecla.

Para obviar este problema, no estado do autómato onde é detectado o fecho ou abertura de um contacto, não se realiza mais processamento associado à transição, apenas se promove a mudança de estado do autómato. Como o tempo entre estados é de 5ms, leva a que o tempo de transição entre estados absorva o tempo de *bounce*. Isto porque, supostamente o tempo de *bounce* é inferior a 5ms.

```
        .equ    DISP_SEG_PORT, 0xff00
        .equ    DISP_ANODO_PORT, 0xff01
        .equ    KEY_PORT, 0xff02

        .section startup
        .org    0
        jmp     main
        ld      r7,[r7,#0]
        .word   ISRTimer

lk_isr_1:
        .space  2
lk_isr_2:
        .space  2
r0_isr:
        .space  2
```

```
;************
;  ring_buf
;************
        .equ    MAX_CHAR,8
ring_buf:
buf:    .space  MAX_CHAR
c:      .space  1
g_ix:   .space  1
p_ix:   .space  1

        .equ    CONT,(c-ring_buf)
        .equ    GET_IX,(g_ix-ring_buf)
        .equ    PUT_IX,(p_ix-ring_buf)



key_stat:
        .space  1
disp_stat:
        .space  1
disp0:
        .space  1
disp1:
        .space  1


;************************
;      void main()
;************************
main:   jmpl    initKey
        jmpl    initDisp
        ldi     r6,#10000b      ;permitir interrupções
main_1: jmpl    getKey          ;le tacla do buffer
        add     r6,r0,#1        ;testa valor devolvido sem afectar o valor
        jz      main_1
        mov     r1,r0
        ldi     r0,#CAR         ;indica que é caracter sem ponto decimal
        ldi     r2,#NO_DOT
        jmpl    dispWrite
        jmp     main_1

        .equ    CAR,0
        .equ    NO_DOT,0

;************************
; software do keyboard
;************************
initKey:
        ldi     r0,#low(ring_buf)
        ldih    r0,#high(ring_buf)
        ldi     r1,#0
        stb     r1,[r0,#CONT]
        stb     r1,[r0,#GET_IX]
        stb     r1,[r0,#PUT_IX]
        stb     r1,key_stat
        ldi     r0,#low(KEY_PORT)
        ldih    r0,#high(KEY_PORT)
        ldi     r1,#0
        stb     r1,[r0,#0]      ; coloca todas as linhas a zero
        ret

;char getKey();
getKey: ldi     r0,#low(ring_buf)
        ldih    r0,#high(ring_buf)
        ldb     r1,[r0,#CONT]
        orl     r1,r1,r1
        jz      getKeyEnd       ; contador a zero no key
        dec     r1
        stb     r1,[r0,#CONT]   ; cont--
        ldb     r1,[r0,#GET_IX]
```

```
        ldb     r2,[r0,r1]      ; r2=ring_buf[get_ix]
        inc     r1              ; get_ix++
        ldi     r3,#MAX_CHAR-1
        anl     r1,r1,r3
        stb     r1,[r0,#GET_IX]         ; get_ix=(get_ix++) % 8
        mov     r0,r2
        ret
getKeyEnd:
        ldi     r0,#0
        dec     r0              ; return -1
        ret

; int writeBuf(char car)
writeBuf:
        ldi     r1,#low(ring_buf)
        ldih    r1,#high(ring_buf)
        ldb     r2,[r1,#CONT]
        sub     r6,r2,#MAX_CHAR
        jz      writeBufEnd     ; buffer full descarta caracter
        inc     r2
        stb     r2,[r1,#CONT]  ; cont++
        ldb     r2,[r1,#PUT_IX]         ; r2=put_ix
        stb     r0,[r1,r2]      ; ring_buf[put_ix]=r0
        inc     r2              ; put_ix++
        ldi     r3,#MAX_CHAR-1
        anl     r2,r2,r3
        stb     r2,[r1,#GET_IX]         ; get_ix=(get_ix++) % 8
        ldi     r0,#0           ; return 0
        ret
writeBufEnd:
        ldi     r0,#0
        dec     r0              ; return -1
        ret

;*******************************
        .equ    LINHAS,4
scanKey:
        st      r5,lk_isr_2
        ldi     r0,#low(KEY_PORT)
        ldih    r0,#high(KEY_PORT)
        ldb     r1,key_stat
        sub     r6,r1,#2        ;testa key_stat sem alterar o valor
        jz      stat_2
        sub     r6,r1,#1
        jz      stat_1
stat_0: ldb     r1,[r0,#0]
        ldi     r2,#0x0f
        anl     r1,r1,r2
        sub     r1,r1,r2
        jz      stat_0_end
        ldi     r0,#1
        stb     r0,key_stat
stat_0_end:
        ret
stat_1: ldi     r2,#LINHAS
        ldi     r4,#0xF7        ; para linha 3 igual a zero
        ldi     r1,#2
        stb     r1,key_stat     ; transita obrigatoriamente para o estado 2
stat_1_1:
        stb     r4,[r0,#0]      ; coloca a zero a linha r2-1
        ldb     r1,[r0,#0]      ; le as colunas
        ldi     r3,#0x0f
        anl     r1,r1,r3        ;
        subr    r1,r1,r3
        jnz     stat_1_find_colun
        shr     r4,r4,#1,1
        dec     r2
        jz      stat_1_fail
        jmp     stat_1_1
stat_1_find_colun:
```

```
        ldi     r3,#0
        stb     r3,[r0,#0]      ; coloca todos os bits do porto a zero para o estado 2
        dec     r2              ; ajustar o número de linhas para indice de linha
stat_1_2:
        shr     r1,r1,#1,0
        jnc     stat_1_colun
        inc     r3
        jmp     stat_1_2
stat_1_colun:
        shl     r2,r2,#2,0      ; linha*(numero de colunas)
        add     r0,r2,r3        ; key=linha*(numero de colunas) + coluna
        jmpl    writeBuf
        ld      r5,lk_isr_2
        ret
stat_1_fail:
        ldi     r1,#0
        stb     r1,[r0,#0]      ; coloca todos os bits do porto a zero para o estado 2
        ld      r5,lk_isr_2
        ret

stat_2: ldb     r1,[r0,#0]      ; espera que levantem a tecla premida
        ldi     r2,#0x0f
        anl     r1,r1,r2
        sub     r1,r1,r2
        jnz     stat_2_end
        ldi     r1,#0
        stb     r1,[r0,#0]      ; coloca todas as linhas a zero
        stb     r1,key_stat
stat_2_end:
        ld      r5,lk_isr_2
        ret

;***********************
; software do display
;***********************
initDisp:
        ldi     r0,#low(DISP_ANODO_PORT)
        ldih    r0,#high(DISP_ANODO_PORT)
        ldi     r1,#0
        stb     r1,disp_stat
        stb     r1,disp0
        stb     r1,disp1
        ldi     r1,#3
        stb     r1,[r0,#0]
        ret

; void dispWrite(char graf, char data, char dot)
dispWrite:
        anl     r0,r0,r0
        jz      disp_car
disp_1: ldb     r0,disp0
        stb     r0,disp1
        not     r1,r1
        stb     r1,disp0
        ret
disp_car:
        ldi     r0,#low(tab_7seg)
        ldih    r0,#high(tab_7seg)
        ldi     r3,#0x0f
        anl     r1,r1,r3
        ldb     r1,[r0,r1]
        orl     r2,r2,r2
        jz      disp_1
        ldi     r2,#0x80
        orl     r1,r1,r2
        jmp     disp_1
tab_7seg:
        .byte   0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x67,0x77,0x7c,0x39,0x5e,0x79,0x71
```

```
; void scanDisp()
scanDisp:
        ldi     r0,#low(DISP_SEG_PORT)
        ldih    r0,#high(DISP_SEG_PORT)
        ldi     r1,#low(DISP_ANODO_PORT)
        ldih    r1,#high(DISP_ANODO_PORT)
        ldb     r2,disp_stat
        anl     r2,r2,r2
        notf    r2,r2
        stb     r2,disp_stat
        jz      scanDispMust
        ldb     r2,disp0
        stb     r2,[r0,#0]
        ldi     r2,#ANODO_LEAST
        stb     r2,[r1,#0]
        ret
scanDispMust:
        ldb     r2,disp1
        stb     r2,[r0,#0]
        ldi     r2,#ANODO_MUST
        stb     r2,[r1,#0]
        ret

        .equ    ANODO_LEAST,10b
        .equ    ANODO_MUST,01b

;***********************************
;
; software da ISR associada ao timer
;***********************************
;
ISRTimer:
        st      r5,lk_isr_1
        st      r0,r0_isr
        jmpl    scanKey
        jmpl    scanDisp
        ld      r5,lk_isr_1
        ld      r0,r0_isr
        iret

        .end
```