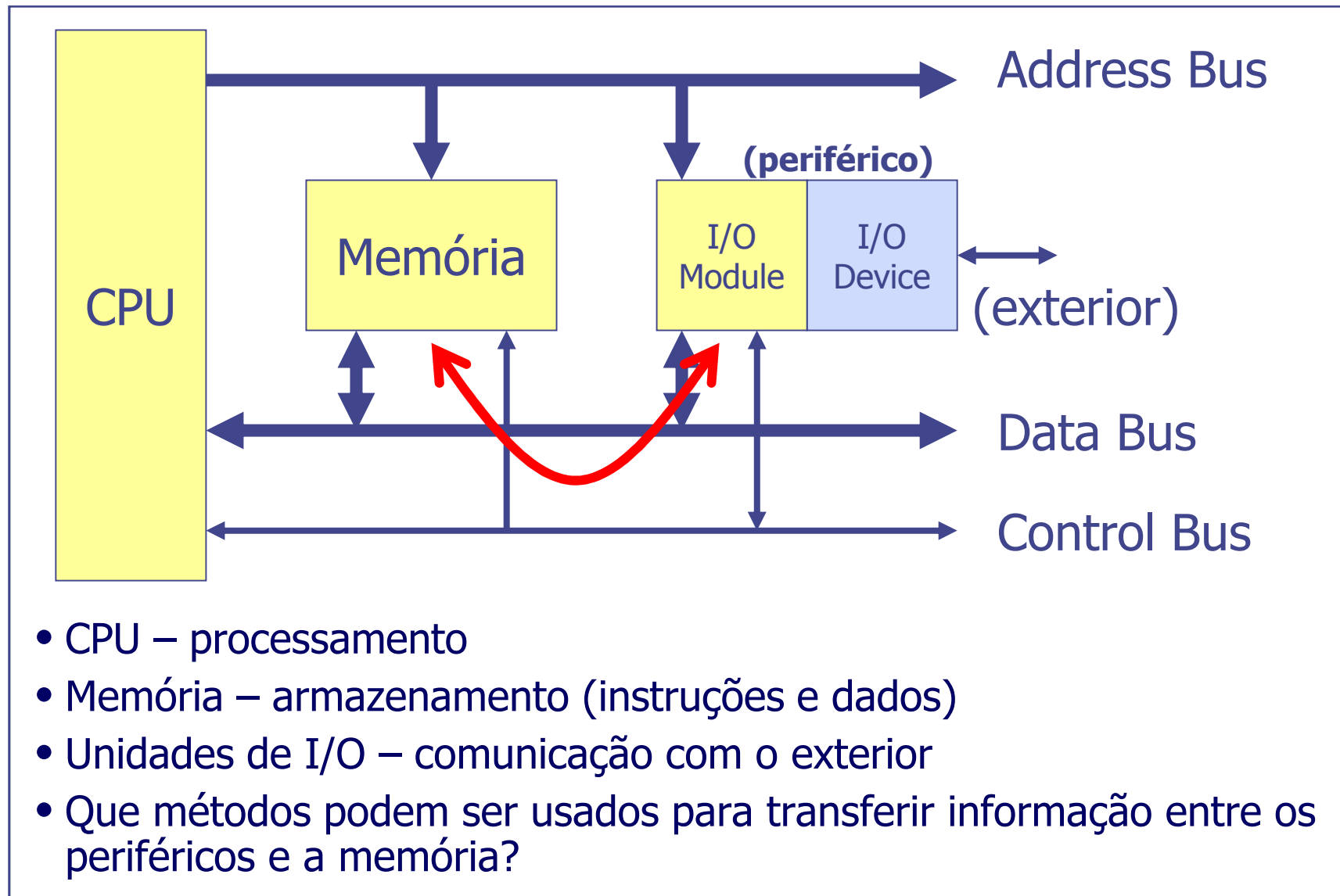


Aulas 6, 7 e 8

- Técnicas de transferência de informação entre os periféricos e a memória
 - E/S programada (*programmed I/O*)
 - E/S por interrupção (*interrupt driven I/O*)
 - E/S por acesso direto à memória (DMA)
- Interrupções:
 - As interrupções no ciclo de instrução do CPU
 - Processamento de interrupções
 - Organizações alternativas do sistema de interrupções
- Interrupções no PIC32

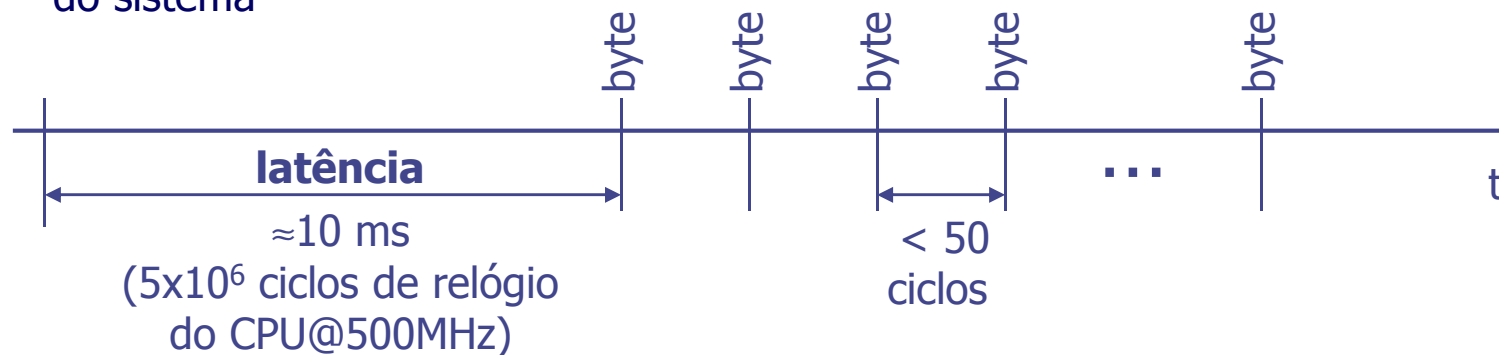
José Luís Azevedo, Arnaldo Oliveira, Tomás Oliveira e Silva, Nuno Lau

Transferência de informação entre memória e I/O



Transferência de informação entre memória e I/O

- Exemplo: transferência de informação de uma unidade de disco para a memória
 - efetuar o pedido** à unidade de disco (por exemplo sector do disco e quantidade de informação pretendida)
 - esperar** que a unidade de **disco tenha a informação disponível** na sua memória interna (informação fornecida por 1 bit de um registo de status)
 - transferir a informação da memória** da unidade de disco para a memória do sistema



- Latência**: tempo que decorre desde o pedido de informação até à disponibilização do 1º byte
- Taxa de transferência de pico (*burst*)**: nº máximo de bytes transferidos por segundo, após latência
- Taxa de transferência média**: nº total de bytes transferidos / tempo total (incluindo latência)

Técnicas de transferência de informação

1. O CPU inicia e controla a transferência de informação:

- E/S programada (***programmed I/O***)
 - O CPU toma a iniciativa – inicia e controla a transferência de informação (**POLLING**)
- E/S por interrupção (***interrupt driven I/O***)
 - O periférico sinaliza o CPU de que está pronto para trocar informação (leitura ou escrita). O CPU controla a transferência

2. O CPU não toma parte na transferência de informação:

- E/S por acesso directo à memória (**DMA**)
 - A informação é trocada diretamente entre a memória e o periférico. O CPU não toma parte no processo
 - O CPU apenas configura inicialmente o DMA e é sinalizado no final que a transferência terminou

E/S Programada

- **Exemplo:** Leitura de N caracteres de um teclado (pseudo-código)

polling {

```
nChar = 0
do {
  do {
    Read "Status register" of keyboard I/O Module
  } while ( key not pressed )
  Read character From I/O Module ("data register")
  Write character Into Memory
  nChar = nChar + 1
} while ( nChar < N )
```

- O programa bloqueia no ciclo de verificação de status (*polling*) e só avança quando for premida uma tecla
- Durante esse tempo o CPU não executa qualquer outra ação

E/S Programada (exemplo para o PIC32)

- O programa assume que no porto RB0 entra um sinal retangular e que no porto RB1 está ligado um LED. O LED comuta de estado sempre que é detetada uma transição de 0 para 1 no porto RB0.

```
# config PIC32 ports
li    $t0,SFR_BASE_HI # $t0=0xBF88
lw    $t1,TRISB($t0)  #
ori    $t1,$t1,1      # RB0 is input
andi   $t1,$t1,0xFFFD # RB1 is output
sw     $t1,TRISB($t0)  # PortB configured

polling { while0: lw    $t1,PORTB($t0) #
           andi   $t2,$t1,1          #
           beq    $t2,$0,while0      # while(RB0=0);
           lw     $t3,LATB($t0)      #
           xori   $t3,$t3,2          #
           sw     $t1,PORTB($t0)     # LATB1=!LATB1;
           while1: lw    $t1,PORTB($t0) #
                  andi   $t1,$t1,1    #
                  bne    $t1,$0,while1 # while(RB0=1);
                  j      while0       #
```

E/S programada

- O CPU tem que esperar que o periférico esteja disponível para a troca de informação. Essa espera é efetuada num ciclo de verificação da informação de status do periférico, designado por **POLLING**
- Uma parte substancial do tempo de processamento do CPU pode ser desperdiçado no ciclo de *polling*
- É uma técnica apropriada quando a velocidade do dispositivo periférico não diminui drasticamente a capacidade de processamento do CPU
- O **overhead** deste método de transferência (i.e., o número de ciclos de relógio gastos pelo CPU em tarefas que não estão diretamente relacionadas com a transferência de informação – pode ser dada em %) depende do número de vezes que o ciclo de *polling* for feito
- Uma solução para eliminar o tempo perdido no ciclo de *polling* consiste na utilização da técnica de **E/S por interrupção**

E/S por interrupção

- Na técnica de E/S por interrupção quando o periférico está pronto para disponibilizar/receber informação sinaliza o CPU
- Uma interrupção, depois de reconhecida, faz com que o CPU abandone temporariamente a execução do programa em curso para executar a rotina que dá seguimento à interrupção gerada
 - A rotina associada à interrupção designa-se por **rotina de serviço à interrupção** ou ***interrupt handler***
- A transferência é também efetuada pelo CPU mas o tempo de espera é eliminado, uma vez que a interrupção ocorre quando o periférico está pronto para a troca de informação
- Esta técnica mascara o problema da longa latência descrito no exemplo de leitura de informação de uma unidade de disco (slide 3)

E/S por interrupção

- **Exemplo:** leitura de informação de um periférico
- Enviar pedido de informação ao periférico
- (CPU continua com outras tarefas...)
- Quando o periférico tiver informação disponível interrompe o CPU
- CPU atende a interrupção:
 - Suspende a execução do programa corrente
 - Salta para uma rotina de atendimento à interrupção (*interrupt handler*) que transfere a informação
 - Retoma a execução do programa suspenso

E/S por interrupção (exemplo de leitura)

```
// Configure I/O device and interrupt system
(...)
bytesReceived = 0
While(1) {
    (...) // Do other tasks/process
    (...) // ← data
}
```

void interrupt isr(void)

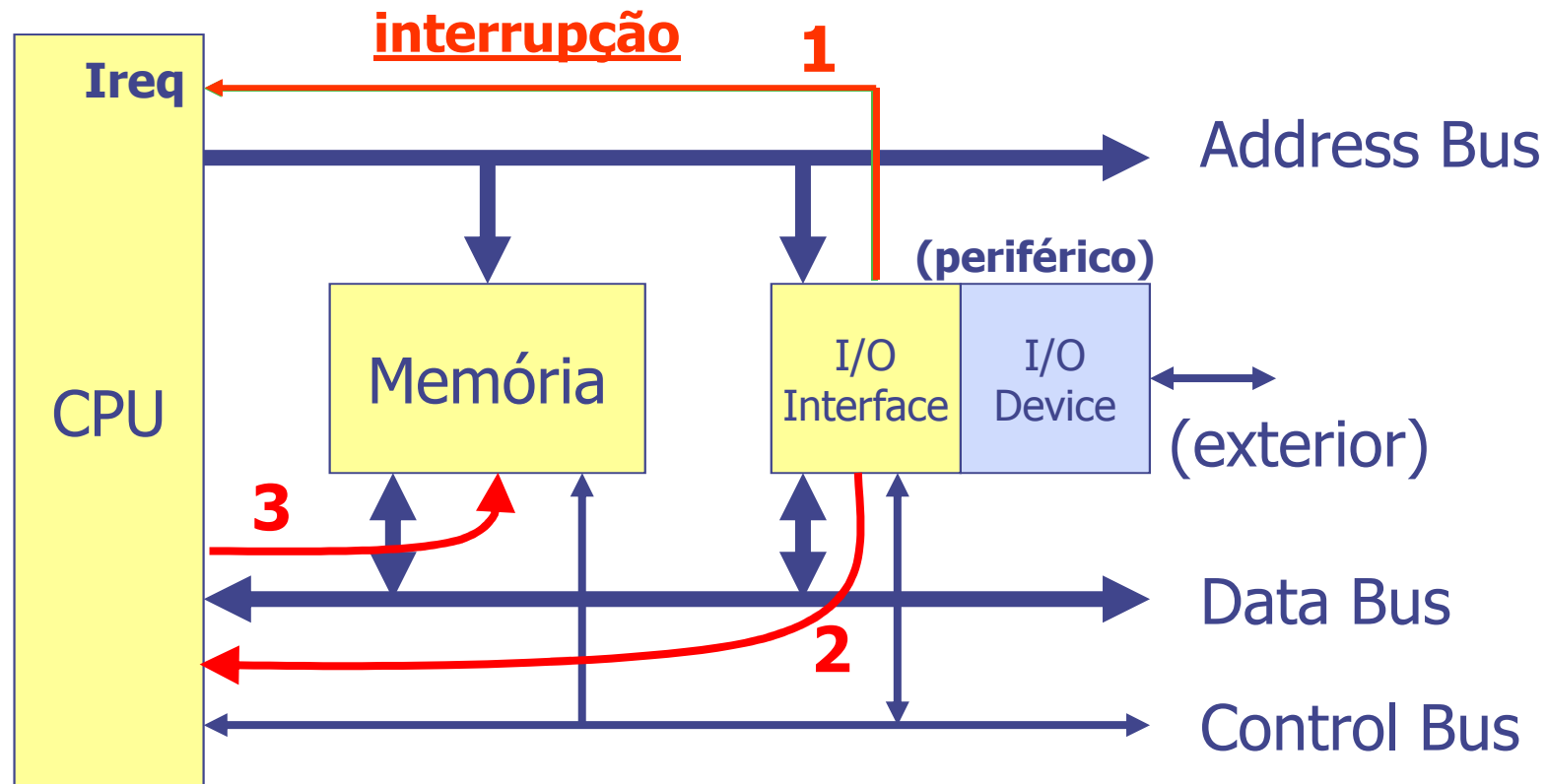
{

Read byte from I/O Module
Write byte into Memory
bytesReceived++

}

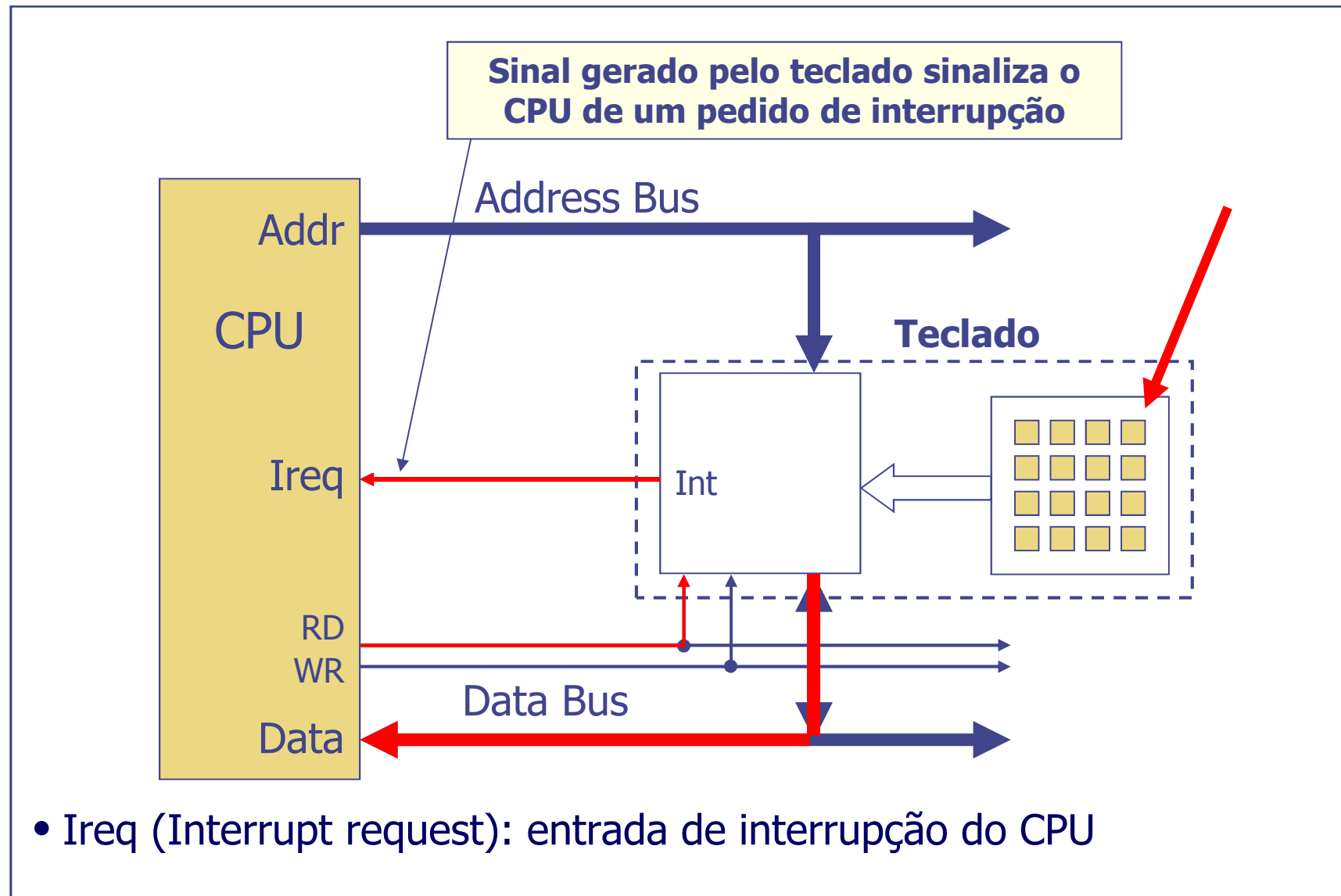
- **Não existe qualquer ciclo de espera.** O periférico gera pedido de interrupção quando está pronto a transferir a informação
- O programa em execução **pode ser interrompido a qualquer momento**
- A Rotina de Serviço à Interrupção (RSI) tem que **salvaguardar o contexto** do programa (registos internos, ...) antes de executar qualquer ação. O contexto salvaguardado tem que ser repostado antes de se terminar a RSI
- A palavra-chave "**interrupt**" distingue uma função do tipo RSI de uma função normal

E/S por interrupção



- O periférico envia pedido de interrupção ao CPU quando tiver informação disponível

E/S por interrupção (exemplo)



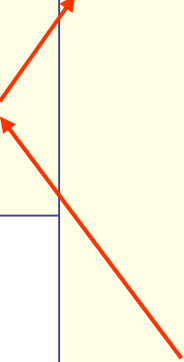
E/S por interrupção

- **Exemplo:** versão *interrupt-driven* do exemplo de comutação do estado de um LED a cada transição de 0 para 1 de um sinal de entrada

```
main:  # config PIC32 ports and interrupt system
      (...)
while: (...)  # CPU executa outras tarefas
      instr.1
      instr.2
      (...)
      instr.n
      j while

      isr: # save program context
            (...)
            lui    $t0,SFR_BASE_HI
            lw     $t1,LATB($t0)  #
            xori   $t1,$t1,2      #
            sw     $t1,LATB($t0)  # RB1=!RB1;
            # restore program context
            (...)
            eret  # exception return
```

Transição de 0 para 1
em RB0 inicia uma
interrupção



- Não existe qualquer ciclo de espera
- O programa em execução é interrompido quando é detetada uma transição 0 para 1 no porto RB0 (CPU salta para o *Interrupt Handler*)
- Quando acaba a execução do *Interrupt Handler*, (rotina "isr") o CPU retoma a execução do programa interrompido

Exceções e interrupções

- Exceções e interrupções são eventos que, não sendo *branches* ou *jumps*, alteram o fluxo normal de execução do programa. Existem duas fontes distintas de eventos deste tipo:
 - Eventos com origem no CPU, inesperados e decorrentes da execução das próprias instruções – **exceções**
 - Por exemplo, o *overflow* aritmético ou o *fetch* de uma instrução com um OpCode desconhecido para a unidade de controlo
 - Eventos com origem externa ao CPU que surgem assincronamente com o funcionamento deste – **interrupções**. Exemplo: quando é premida uma tecla do teclado do exemplo anterior
- **Exceções**: a instrução que gera a exceção não termina
- **Interrupções**: a unidade de controlo apenas verifica se há algum pedido de interrupção pendente antes de iniciar o *fetch* de uma nova instrução
- Processamento de interrupções e exceções é semelhante

Exceções e interrupções

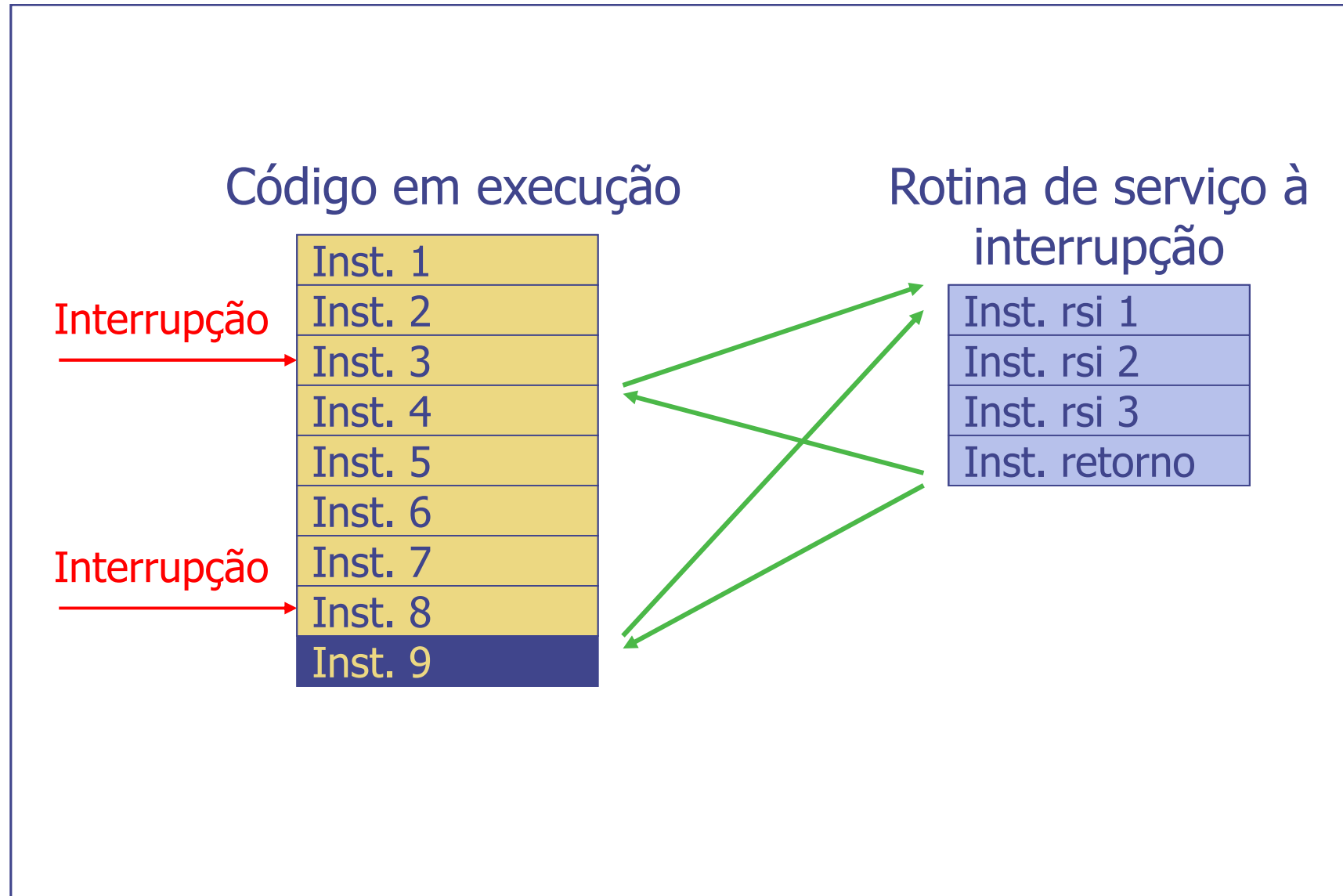
- Exemplos de dispositivos que podem gerar interrupções: teclado, rato, timers, dispositivos de comunicação, ...
- Exemplos de exceções:
 - Divisão por zero
 - *Overflow* numa operação aritmética
 - Tentativa de execução de uma instrução cujo OpCode é desconhecido
 - Acesso a um endereço de memória não alinhado (caso do MIPS)
- No MIPS a instrução "syscall" (usada nos *system calls*) usa o mesmo mecanismo das exceções no que respeita a:
 - Salvaguarda do endereço da instrução "syscall" (o retorno é feito para a instrução seguinte)
 - Salvaguarda do contexto do CPU
 - Salto para o *exception handler* e retorno ao programa que executou o "syscall"

Atendimento de interrupções e exceções

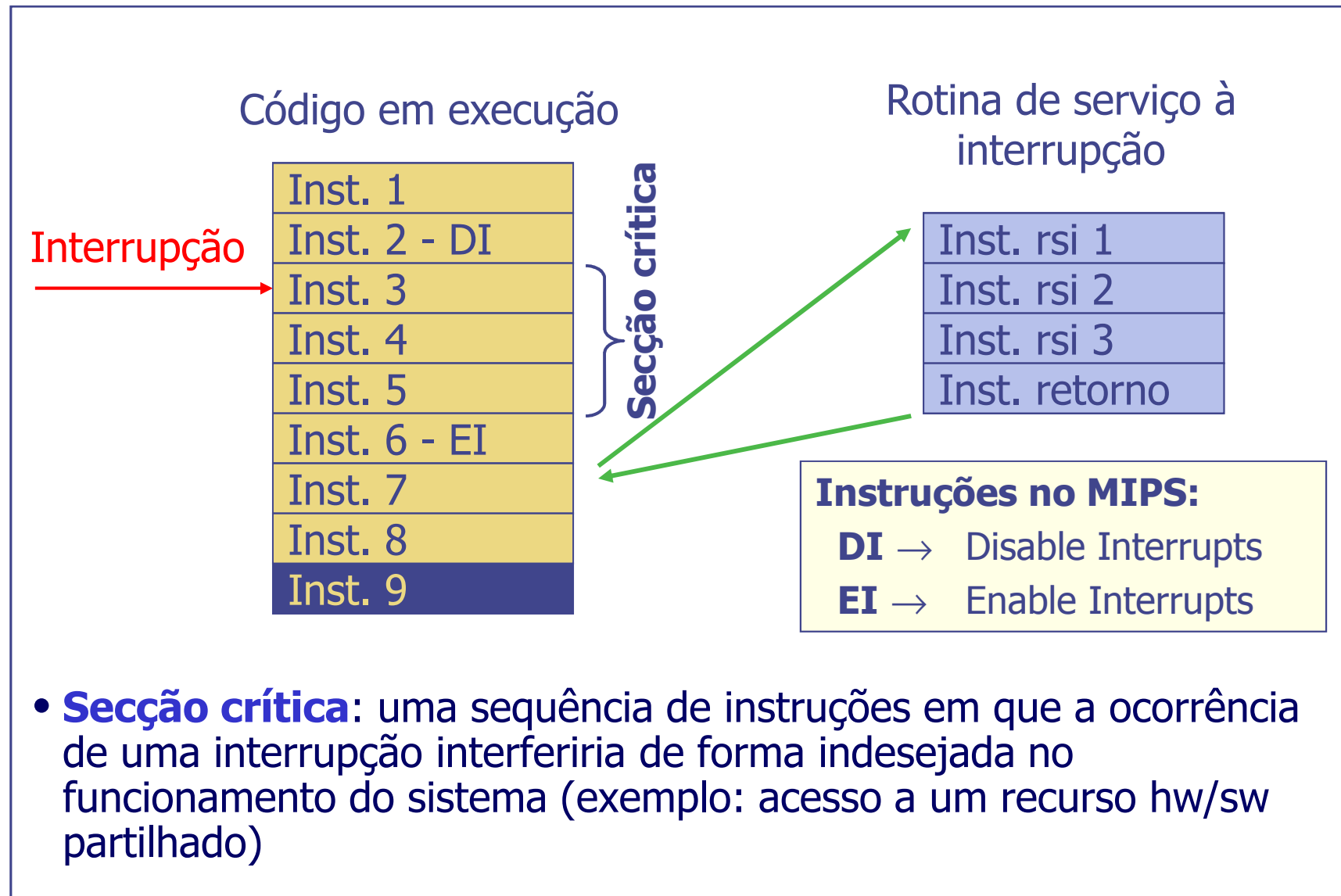
- **Exceções:** a instrução que gerou a exceção não termina, sendo a execução passada de imediato para a rotina de tratamento da exceção
- **Interrupções:** a passagem da execução para a rotina de tratamento da interrupção só acontece quando for concluída a instrução que está a ser executada no momento em que a interrupção surge
- As interrupções no ciclo de instrução do CPU:

```
while( 1 )
{
    if ( interrupt request line is active )
    {
        Process interrupt request (... , jump to Interrupt Service Routine)
    }
    Fetch instruction and increment PC
    Decode instruction
    Execute instruction and store result
}
```


Processamento de interrupções

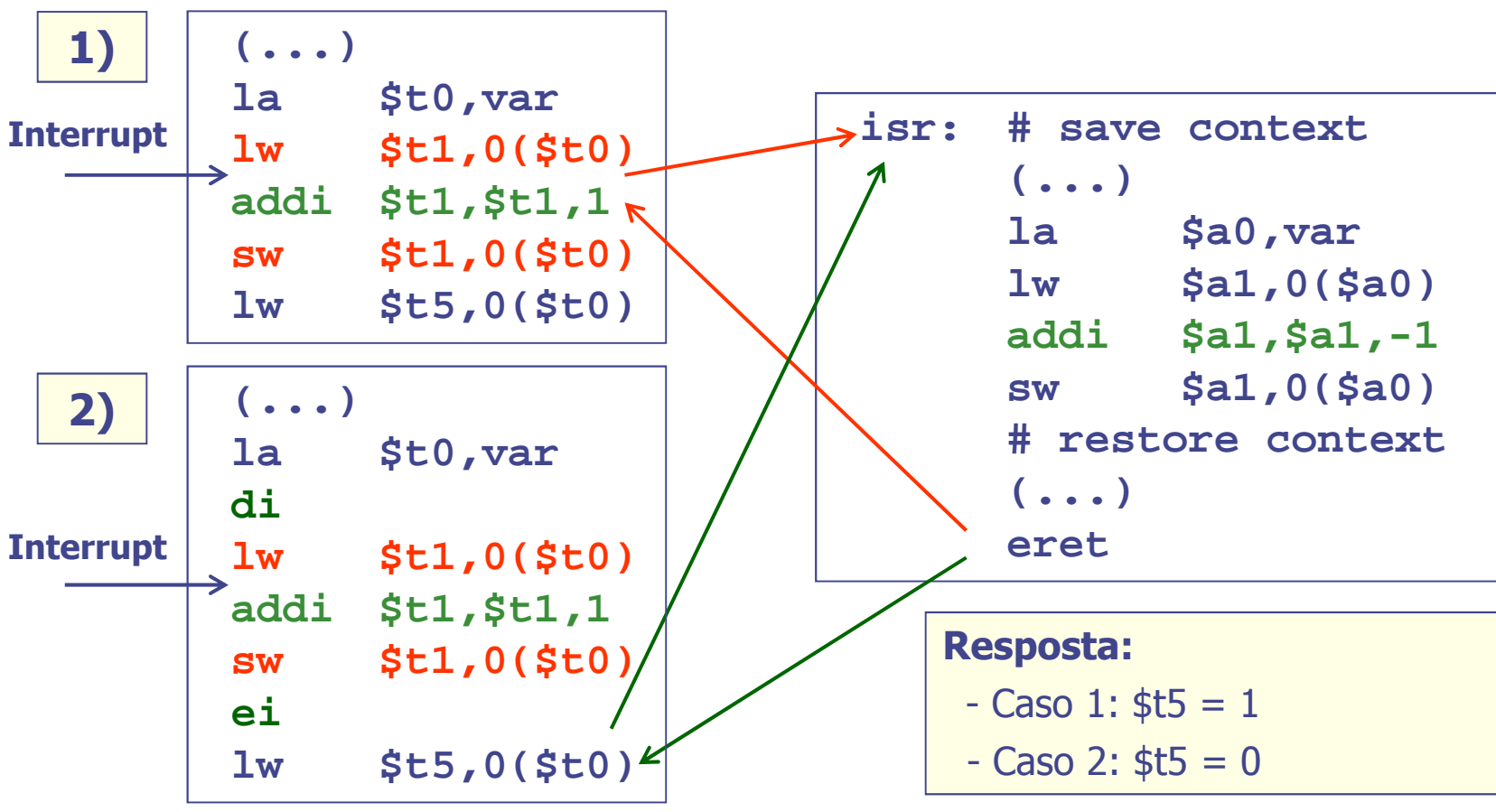


Ativação/desativação global das interrupções



Processamento de interrupções – secção crítica (exemplo)

- A variável "var" pode ser lida/atualizada na RSI e no programa principal. Se "var" tem o valor 0 antes da ocorrência da interrupção, qual o valor lido para o registo **\$t5**, no caso 1 e no caso 2?



Processamento de interrupções pelo CPU

- Em termos gerais, o processamento de uma interrupção é efetuado, pelo CPU, nos seguintes passos:
 1. Identificação da fonte de interrupção (nos casos em que tal é efetuado por hardware) e obtenção do endereço da RSI
 2. Salvaguarda do contexto atual do CPU (valor corrente do PC e de *flags* de estado associadas ao sistema)
 3. Desativação das interrupções
 4. Carregamento no PC do endereço da RSI ($PC \leftarrow \text{Endereço da RSI}$, i.e., salto para a 1ª instrução da RSI)
 5. Execução da RSI até encontrar a instrução de retorno
 6. Execução da instrução de retorno da RSI (e.g. *eret*, no MIPS)
 - Reposição do contexto salvaguardado (PC e flags) e reativação das interrupções
 - Regresso ao programa interrompido, executando a instrução que teria sido executada se não tivesse acontecido a interrupção

Processamento de interrupções pela RSI

- Ações gerais que devem ser implementadas na Rotina de Serviço à Interrupção (software):
 1. Salvaguarda do contexto do programa que foi interrompido:
registos internos do CPU → memória (*stack*) ("**PROLOGUE**")
 2. .. ações associadas ao processamento da interrupção..
 3. Reposição do contexto do programa interrompido:
registos internos do CPU ← memória (*stack*) ("**EPILOG**")
 4. Terminação da RSI com a instrução de retorno (do tipo "Return From Interrupt / exception" – eret no caso do MIPS)
- **Latência da interrupção**: define-se como o tempo que decorre desde a ocorrência do evento que desencadeia a interrupção até à execução da primeira instrução da Rotina de Serviço à Interrupção

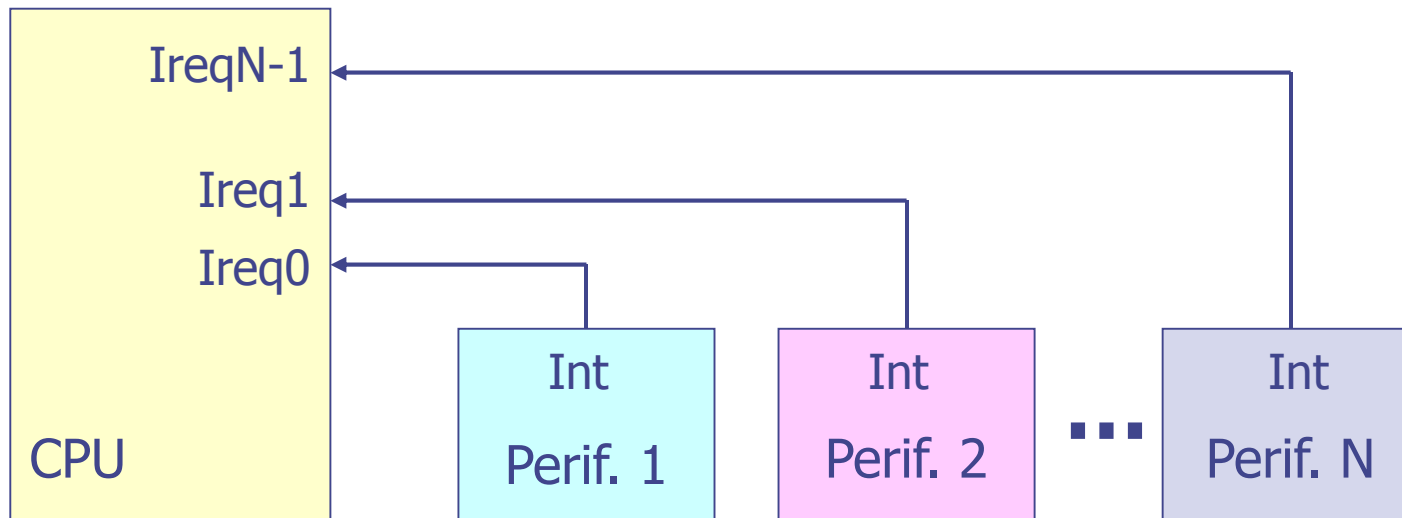
Overhead do método de transferência por interrupção

- O ***overhead*** global do método de transferência por interrupção é causado pela mudança de contexto:
 - A rotina de serviço à interrupção tem que, à entrada, **salvaguardar o contexto do programa interrompido**
 - Antes de abandonar a RSI tem que **repor o contexto salvaguardado**
 - A título de exemplo, estas 2 operações requerem, no MIPS, cerca de 80 instruções
- Em sistemas computacionais mais evoluídos, outro aspeto negativo da mudança de contexto é a que resulta da, muito provável, mudança da informação nas memórias cache (a ver mais tarde)
 - A RSI poderá utilizar zonas de memória diferentes das do programa interrompido, o que obriga à atualização das memórias *cache* com o consequente impacto no número de ciclos de relógio gastos
 - Por outro lado, o regresso ao programa interrompido tem uma consequência semelhante, obrigando à atualização das memórias cache, desta vez com as zonas de memória que o programa estava a utilizar antes de ocorrer a interrupção

Organização do sistema de interrupções

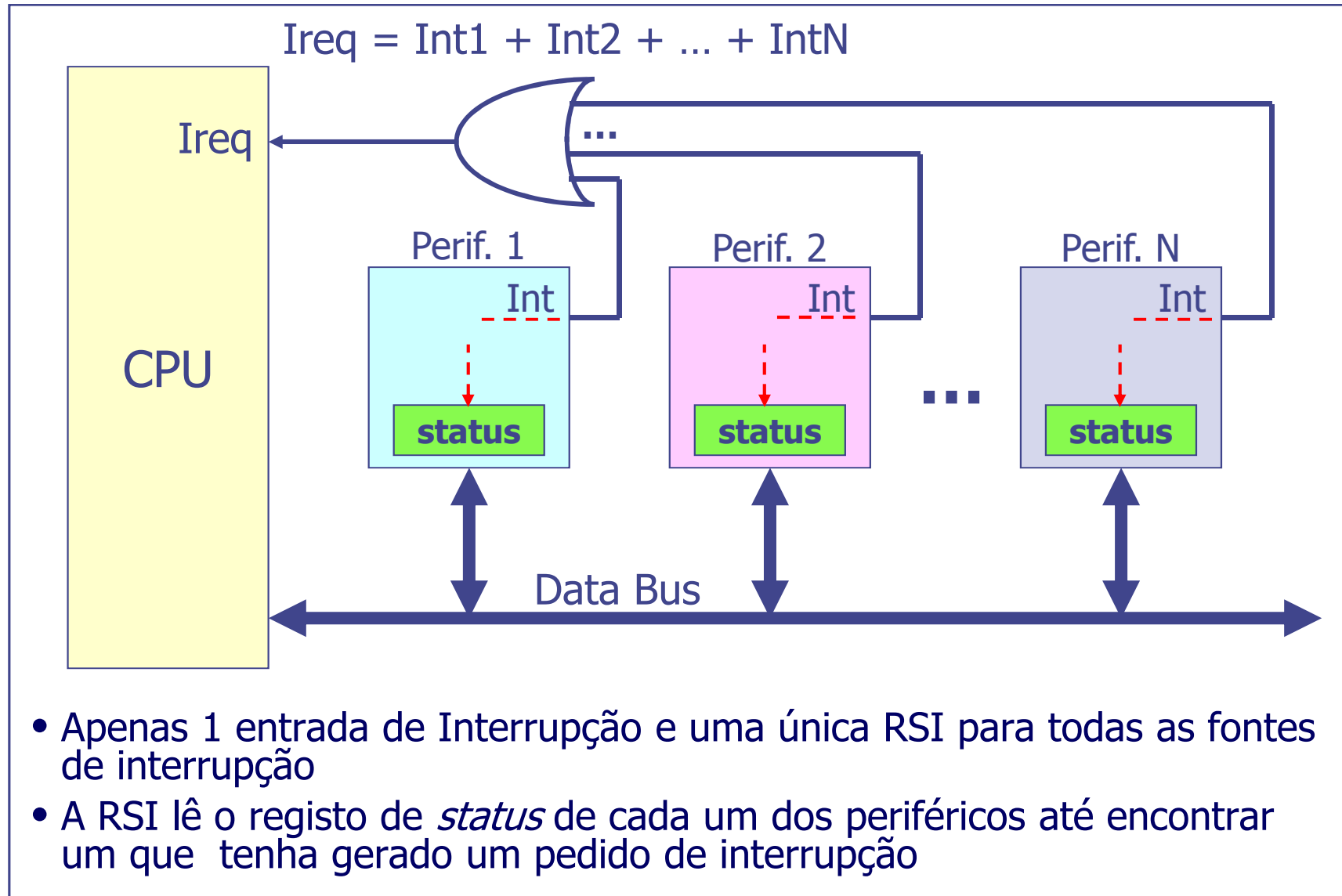
- Num sistema real vários periféricos podem ter a capacidade de gerar interrupções
- Como organizar o sistema de interrupções para permitir a ligação de vários periféricos?
 - **Múltiplas linhas de interrupção**
 - **Identificação da fonte de interrupção por software**
 - **Interrupções vetorizadas** (identificação da fonte de interrupção por hardware)
- Como gerir pedidos simultâneos de interrupção (qual a ordem do atendimento)?
- Como atribuir diferentes níveis de prioridade a diferentes fontes de interrupção?

Múltiplas linhas de interrupção



- Identificação automática da fonte de interrupção
- Uma RSI para cada fonte de interrupção
- Número máximo de dispositivos que podem gerar interrupção é igual ao número de linhas de interrupção do CPU
- Cada linha tem atribuída uma prioridade fixa (pode ser usado um *priority encoder*)
- No caso de haver 2 ou mais linhas de interrupção ativas simultaneamente, o CPU atende em 1º lugar a de mais alta prioridade

Identificação da fonte de interrupção por *software*



Identificação da fonte de interrupção por software

- Exemplo de organização da Rotina de Serviço à Interrupção

```
void interrupt general_isr(void)
{
    Read status, peripheral 1
    If( interrupt_bit = ON) {
        peripheral_isr_1()
    }

    Read status, peripheral 2
    If( interrupt_bit = ON) {
        peripheral_isr_2()
    }

    (...)

    Read status, peripheral n
    If( interrupt_bit = ON) {
        peripheral_isr_n()
    }
}
```

Funções específicas para tratamento dos pedidos de interrupção de cada fonte

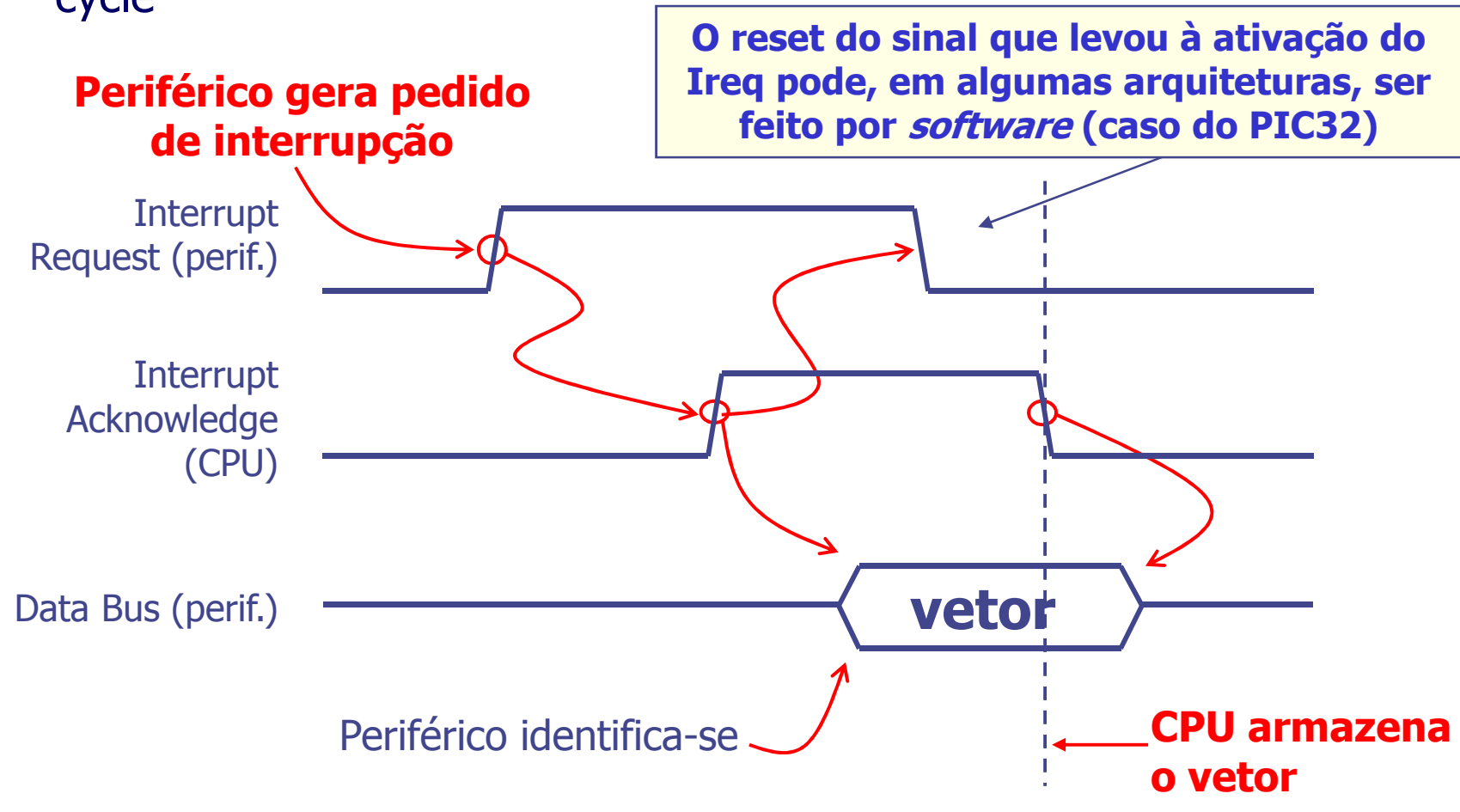
No caso de pedidos de interrupção simultâneos, a ordem pela qual os periféricos são "questionados" determina a prioridade no atendimento

Interrupções vetorizadas

- CPU tem apenas 1 entrada de interrupção
- A identificação da fonte é feita por hardware
- Cada periférico possui um identificador único, designado por **vetor**
- Durante o processo de atendimento, na fase de identificação da fonte, o periférico gerador da interrupção identifica-se através do seu vetor
- Este vetor vai ser usado depois como índice de uma tabela que especifica como chegar à RSI
- **Uma RSI para cada vetor de interrupção**

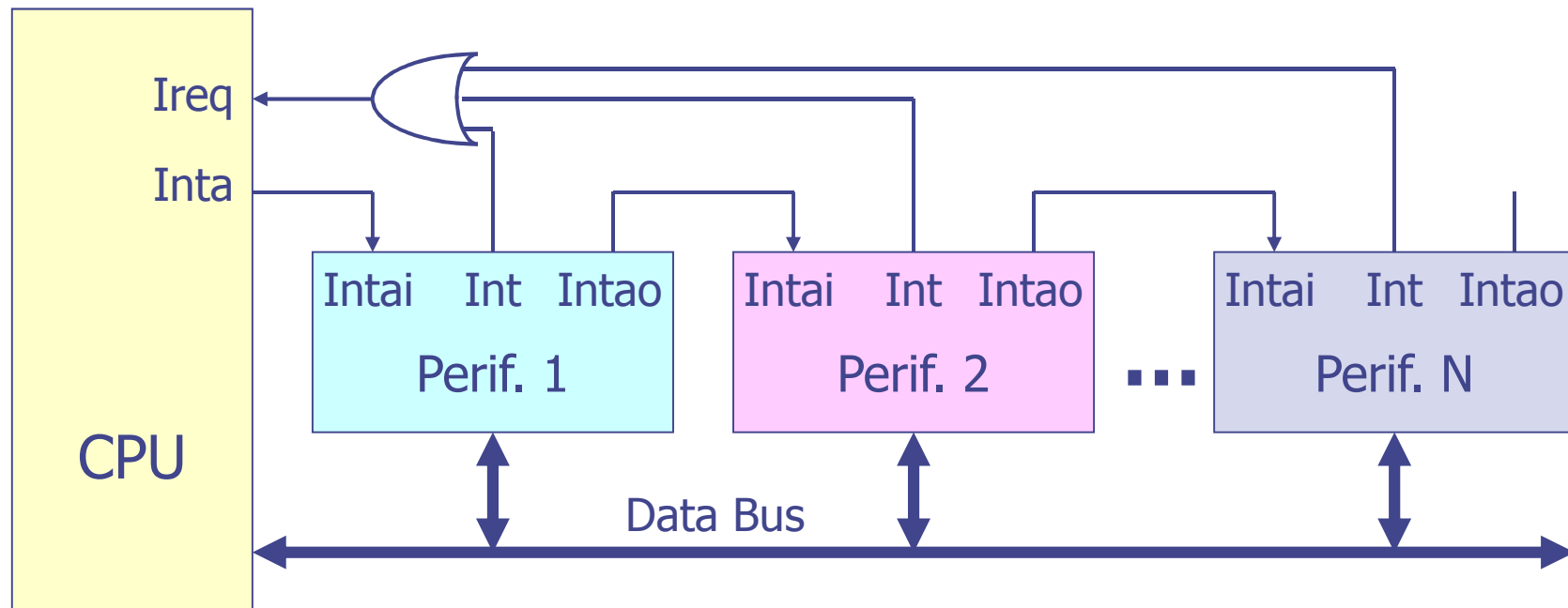
Interrupções vetorizadas

- A identificação da fonte de interrupção é feita por hardware num processo genericamente designado por "Interrupt acknowledge cycle"



Interrupções vetorizadas – *daisy chain*

- Periféricos podem estar organizados numa estrutura *daisy chain*



$$Ireq = Int1 + Int2 + \dots + IntN$$

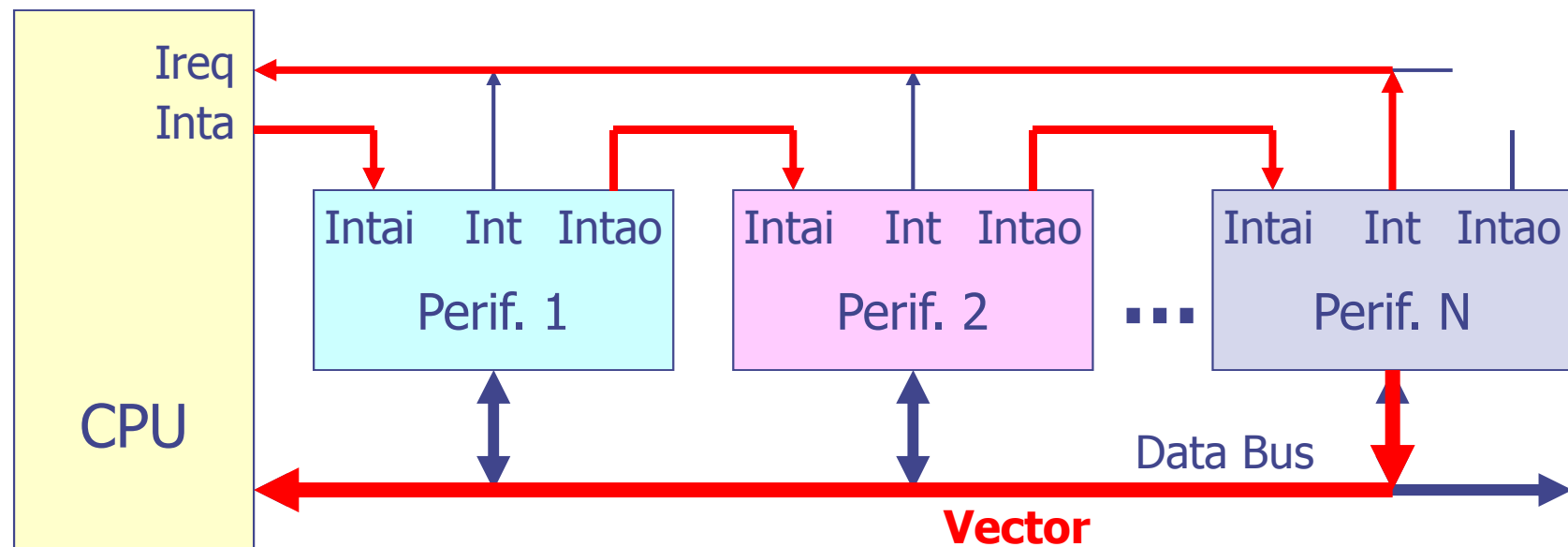
Intai/o - Interrupt Acknowledge in/out

Interrupções vetorizadas – *daisy chain*

- Genericamente, o procedimento de identificação da fonte de interrupção num esquema de interrupções vetorizadas em que os periféricos estão organizados numa cadeia *daisy chain* é o seguinte:
 1. Quando o CPU deteta o pedido de interrupção ("Ireq") e está em condições de o atender ativa o sinal "Interrupt Acknowledge" ("Inta")
 2. O sinal "Inta" percorre a cadeia até ao periférico que gerou a interrupção
 3. O periférico que gerou a interrupção coloca o seu identificador (vetor) no barramento de dados
 4. O CPU lê o vetor e, a partir dele, obtém o endereço da RSI a executar (por exemplo a partir de uma tabela previamente preenchida); de seguida salta para a RSI

Interrupções vetorizadas – *daisy chain*

- Exemplo de identificação da fonte de interrupção

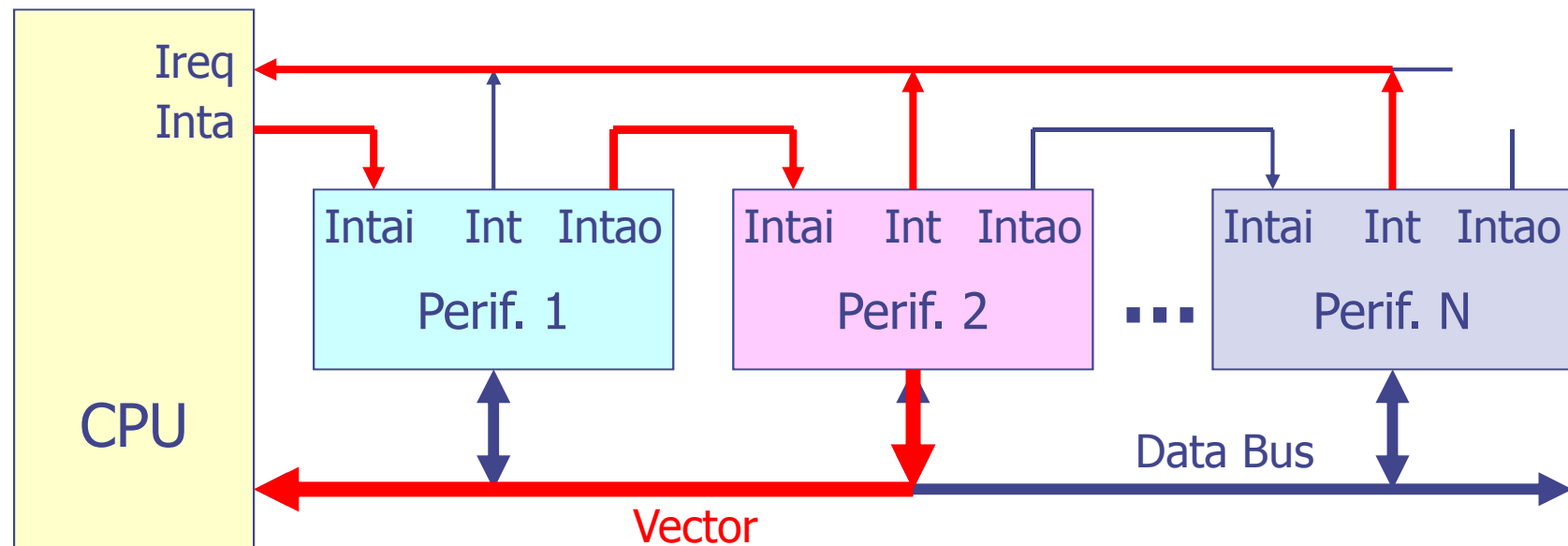


$$Ireq = Int1 + Int2 + \dots + IntN$$

Intai/Intao - Interrupt Acknowledge in/out

Interrupções vetorizadas – *daisy chain*

- Exemplo de identificação da fonte de interrupção, no caso em que dois periféricos têm a linha de interrupção ativa

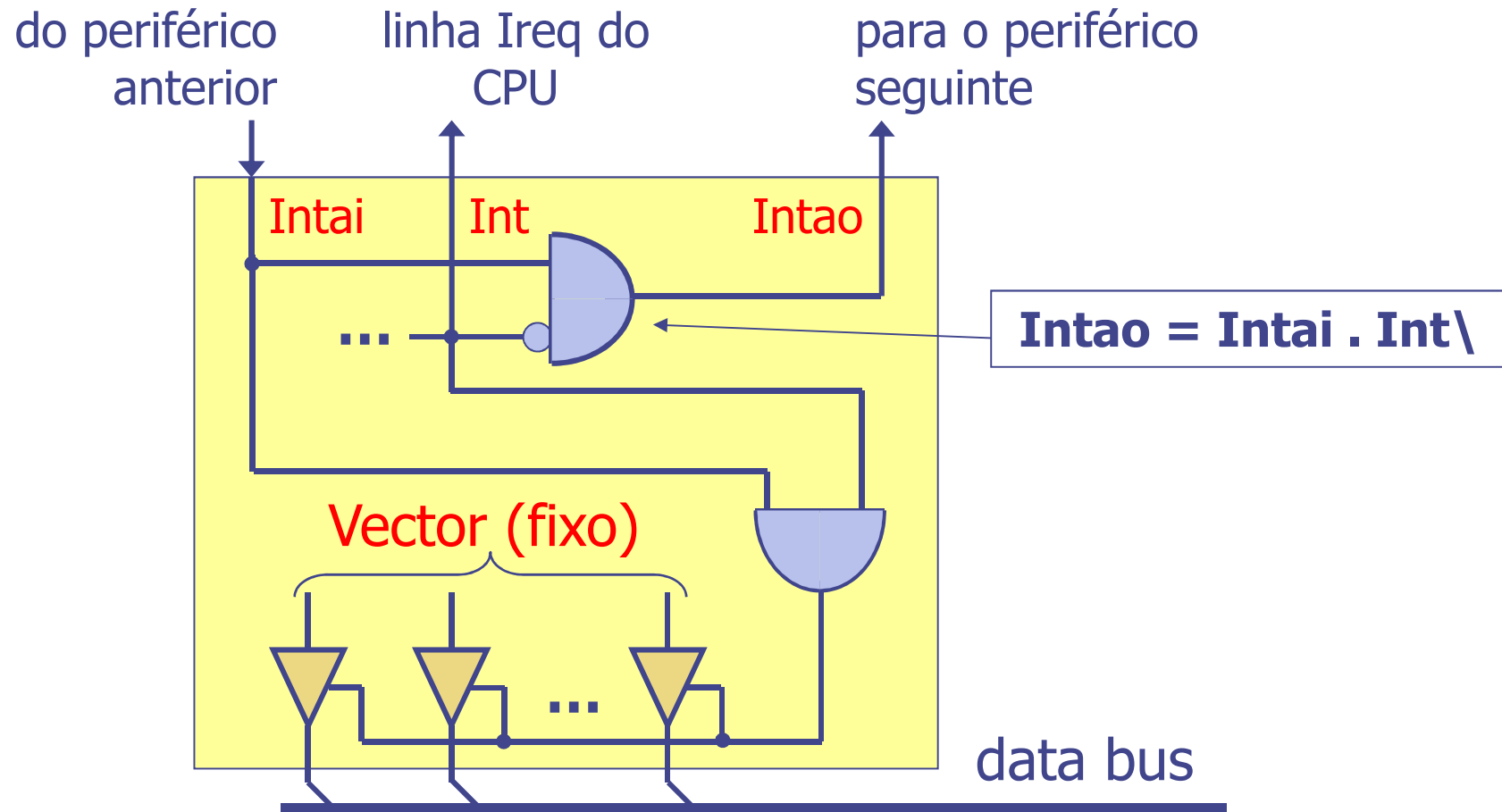


$$Ireq = Int1 + Int2 + \dots + IntN$$

- A ordem de colocação dos periféricos na cadeia, relativamente ao CPU, determina a sua prioridade

Interrupções vetorizadas – *daisy chain*

- Estrutura típica do periférico (arbitragem e identificação)

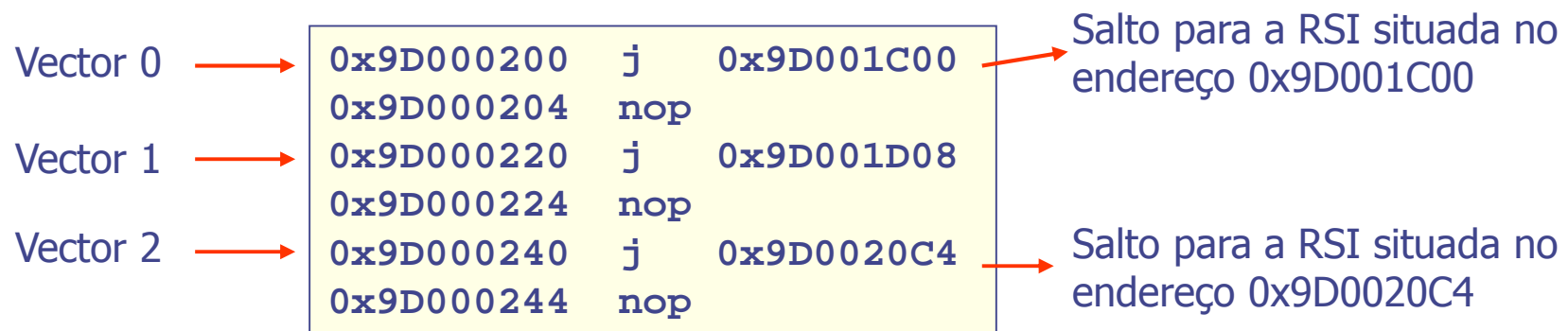


Interrupções vetorizadas – tabela de vetores

- Há duas formas de organizar a tabela de interrupções que associa um dado vetor a uma RSI
 1. A tabela é inicializada com os endereços de todas as RSI
 2. A tabela é inicializada com instruções de salto para as RSI
- No primeiro caso, na fase inicial do processamento da interrupção o CPU acede à tabela, usando como índice o vetor
- O valor lido da tabela é carregado no *Program Counter*
- Este modelo é usado, por exemplo, na arquitetura Intel x86

Interrupções vetorizadas – tabela de vetores

- No segundo caso, são colocadas na tabela de interrupções instruções de salto para as RSI (em vez dos seus endereços)
- No processamento da interrupção o CPU usa o vetor como *offset* para saltar (jump) para a posição da tabela onde está, em geral, uma instrução de salto incondicional para a RSI a executar.
- Este modelo é o usado na arquitetura MIPS.
- O exemplo seguinte ilustra esta forma de organização, para 3 vetores:



Interrupções no PIC32

- O PIC32 pode ser configurado em um de dois modos:
 - **Single-vector mode** – um único vetor (0) para todas as fontes de interrupção, ou seja, identificação da fonte por software
 - **Multi-vector mode** – Interrupções vetorizadas (vetores definidos pelo fabricante para todas as fontes – ver PIC32MX7XX Family Data Sheet – Interrupt Controller)
- **Na placa DETPIC32 o sistema de interrupções está configurado para "multi-vector mode"**
- O sistema de interrupções do PIC32 é baseado num módulo de gestão exterior ao CPU (controlador de interrupções)
- Até 96 fontes de interrupção (75 no PIC32MX7xx) das quais 5 fontes externas com configuração de transição ativa (*rising* ou *falling edge*)
- Até 64 vetores (51 no PIC32MX7xx)
- O controlador de interrupções permite, entre outras coisas, a configuração das prioridades de cada fonte
- Funciona como um **priority encoder** enviando para o CPU o pedido pendente de maior prioridade (identificado com vetor e prioridade)

Interrupções no PIC32

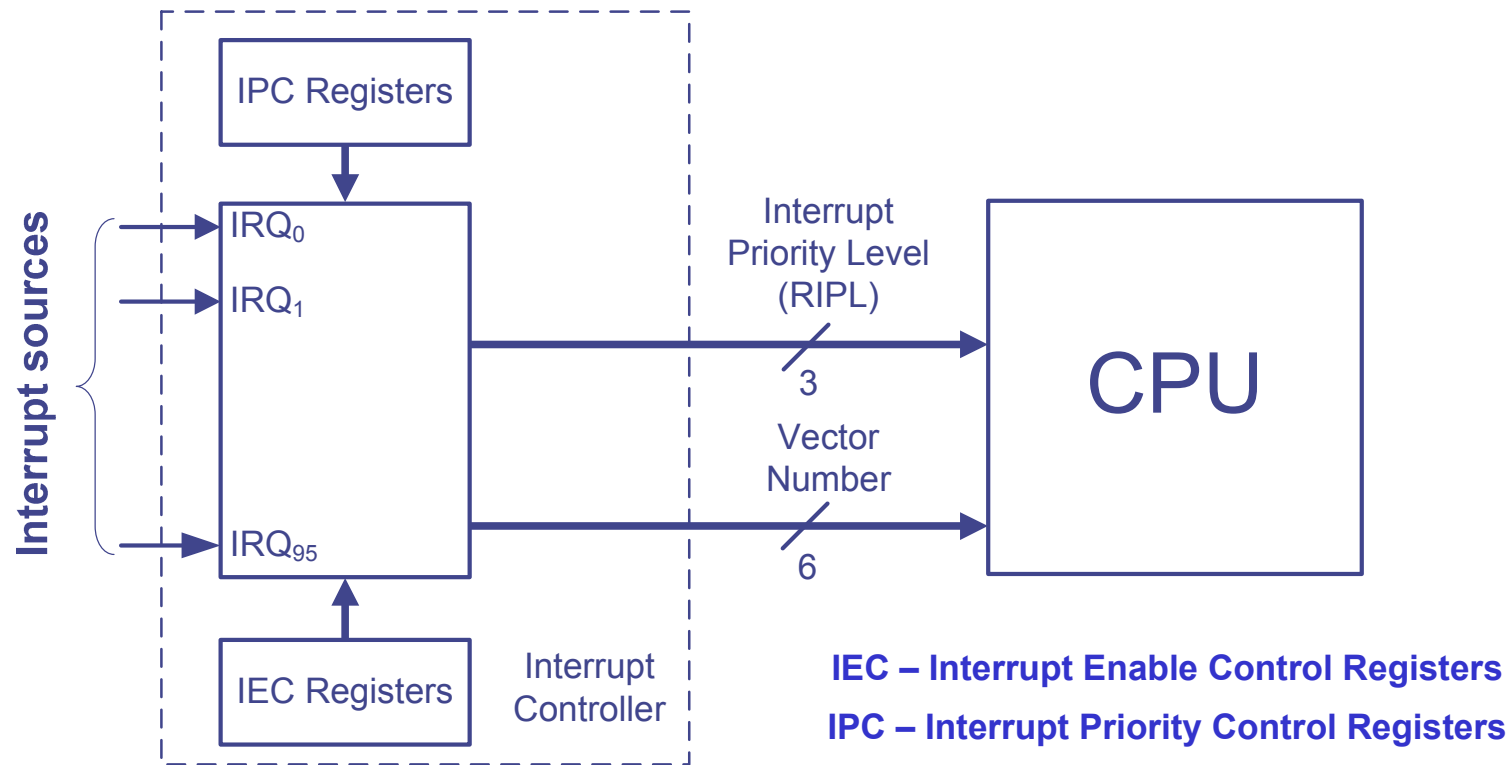
- **IEC0, IEC1, IEC2** – Interrupt Enable **Control Registers**
 - Registos através dos quais se pode habilitar / desativar (enable / disable) qualquer fonte de interrupção. Cada módulo do PIC32 que pode gerar interrupções usa 1 ou mais bits destes registos
- **IPC0, IPC1, ..., IPC12** – Interrupt Priority **Control Registers**
 - Registos através dos quais se pode configurar, com 3 bits, a prioridade de cada uma das fontes de interrupção (0 a 7)
- **IFS0, IFS1, IFS2** – Interrupt Flag **Control / Status Registers**
 - Flags de sinalização da ocorrência de interrupções, de todas as fontes possíveis. Cada módulo do PIC32 que pode gerar interrupções usa 1 ou mais bits destes registos
- **INTCON** – Interrupt **Control Register**
 - Configura a polaridade da transição das fontes de interrupção externa (rising edge / falling edge)

Interrupções no PIC32

- Cada fonte de interrupção tem associado um conjunto de bits de configuração e de status
- **Interrupt Enable Bit** – bit definido em um dos registos **IECx** (Interrupt Enable Control Registers), através do qual se pode fazer o *enable* ou o *disable* de uma dada fonte de interrupção. O nome do bit é, normalmente, formado pela sigla identificativa da fonte, terminada com o sufixo **–IE** (e.g. **T1IE**, *Timer1 Interrupt Enable*)
- **Interrupt Flag** – bit definido em um dos registos **IFSx** (Interrupt Flag Status Registers) e designado com o sufixo **–IF** (e.g. **T1IF**, *Timer1 Interrupt Flag*). Este bit é ativado automaticamente quando ocorre uma interrupção. A desativação é da responsabilidade do programador
- **Priority Level** – 3 bits definidos em um dos registos **IPCx** (Interrupt Priority Control Registers), designado com o sufixo **–IP** (e.g. **T1IP**, *Timer1 Interrupt Priority*)
 - 7 níveis de prioridade (1 a 7); a prioridade 0 significa fonte *disabled*

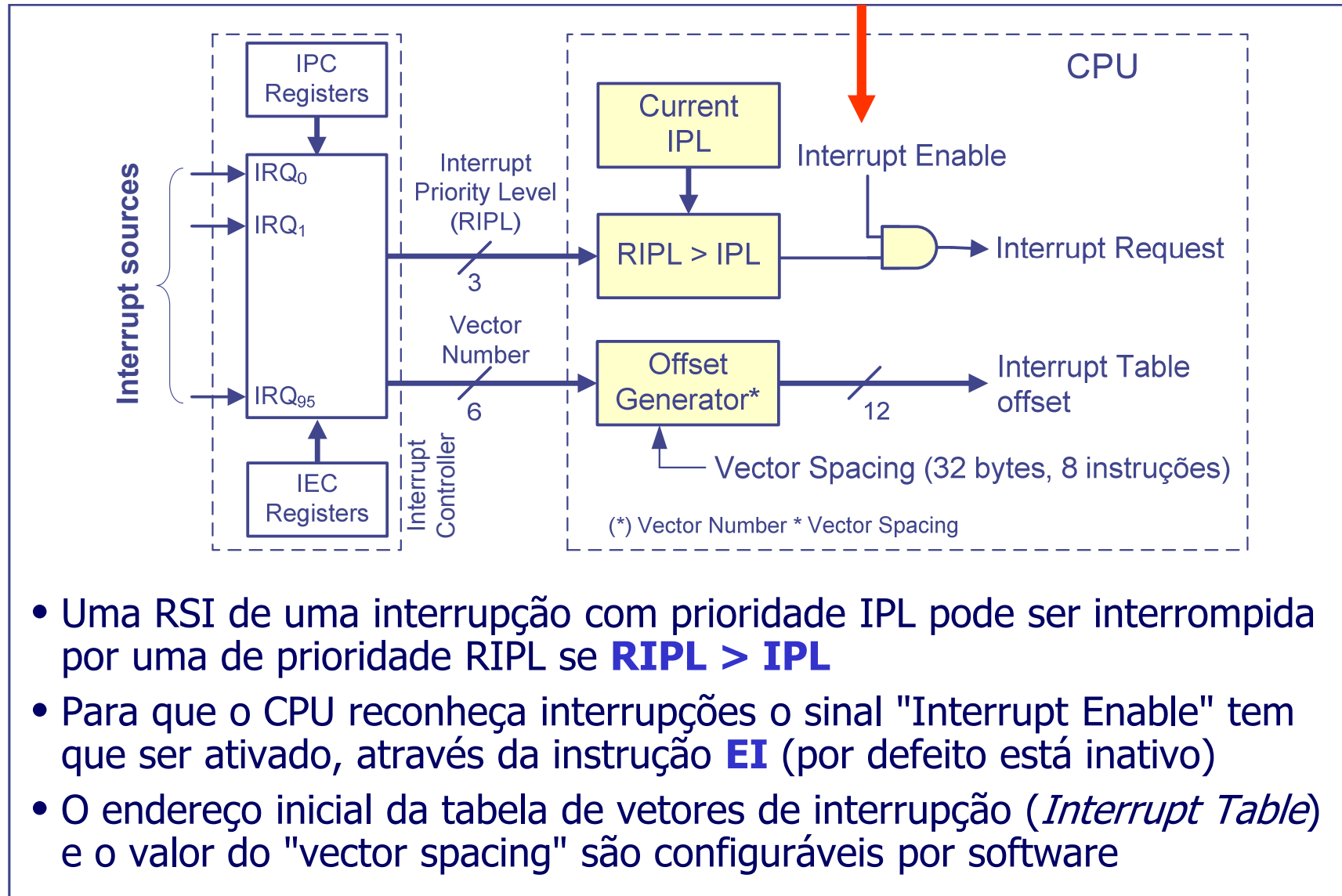
Interrupções no PIC32

- Fontes de interrupção:
 - Internas: até 91, de periféricos internos; externas: 5



- O pedido pendente com maior prioridade é encaminhado para o CPU (identificado pelo vetor e pela prioridade – RIPL)

Interrupções no PIC32



Exemplo de uma tabela de vetores no PIC32

#vector_7 (INT1, External Interrupt 1)

0x9D0002E0 0x0B40074D j 0x9D001D34

0x9D0002E4 0x00000000 nop

#vector_8 (T2 - Timer2)

0x9D000300 0x0B4006C3 j 0x9D001B0C

0x9D000304 0x00000000 nop

#vector_19 (INT4 - External Interrupt 4)

0x9D000460 0x0B40077A j 0x9D001DE8

0x9D000464 0x00000000 nop

- Na placa DETPIC32 o endereço inicial da tabela de vetores (a que corresponde o vetor 0) é **0x9D00200**.

Rotina de Serviço à Interrupção no PIC32

TABLE 7-1: INTERRUPT IRQ, VECTOR AND BIT LOCATION

Interrupt Source ⁽¹⁾	IRQ Number	Vector Number	Interrupt Bit Location			
			Flag	Enable	Priority	Sub-Priority
Highest Natural Order Priority						
CT – Core Timer Interrupt	0	0	IFS0<0>	IEC0<0>	IPC0<4:2>	IPC0<1:0>
CS0 – Core Software Interrupt 0	1	1	IFS0<1>	IEC0<1>	IPC0<12:10>	IPC0<9:8>
CS1 – Core Software Interrupt 1	2	2	IFS0<2>	IEC0<2>	IPC0<20:18>	IPC0<17:16>
INT0 – External Interrupt 0	3	3	IFS0<3>	IEC0<3>	IPC0<28:26>	IPC0<25:24>
T1 – Timer1	4	4	IFS0<4>	IEC0<4>	IPC1<4:2>	IPC1<1:0>

Interrupt Function
Vector Number
Function Name

```
void _int_( 4 ) isr_t1(void)
{
    ...
    IFS0bits.T1IF = 0; // Reset T1 Interrupt Flag
}
```

Exemplo de programação com interrupções no PIC32

```
void main(void)
{
    configIO();           // Config IO and Interrupt
                          //   Controller
    EnableInterrupts();   // Enable Interrupt System
    while(1)
    {
        ...
    }
}

// IO Configuration function
void configIO(void)
{
    ...
    IFS0bits.T1IF = 0;    // Reset Timer 1 interrupt flag
    IPC2bits.T1IP = 2;    // Set priority level to 2
    IEC0bits.T1IE = 1;    // Enable Timer 1 interrupts
    ...
}
```

Exemplo de programação com interrupções no PIC32

```
// Interrupt Service routine - Timer2
void __int__( 8 ) isr_t2(void)
{
    ...
    IFS0bits.T2IF = 0;    // Reset Timer 2 Interrupt Flag
}

// Interrupt Service routine - External Interrupt 1
void __int__( 7 ) isr_ext_int1(void)
{
    ...
    IFS0bits.INT1IF = 0; // Reset External Interrupt 1 Flag
}

// Interrupt Service routine - External Interrupt 4
void __int__( 19 ) isr_ext_int4(void)
{
    ...
    IFS0bits.INT4IF = 0; // Reset External Interrupt 4 Flag
}
```