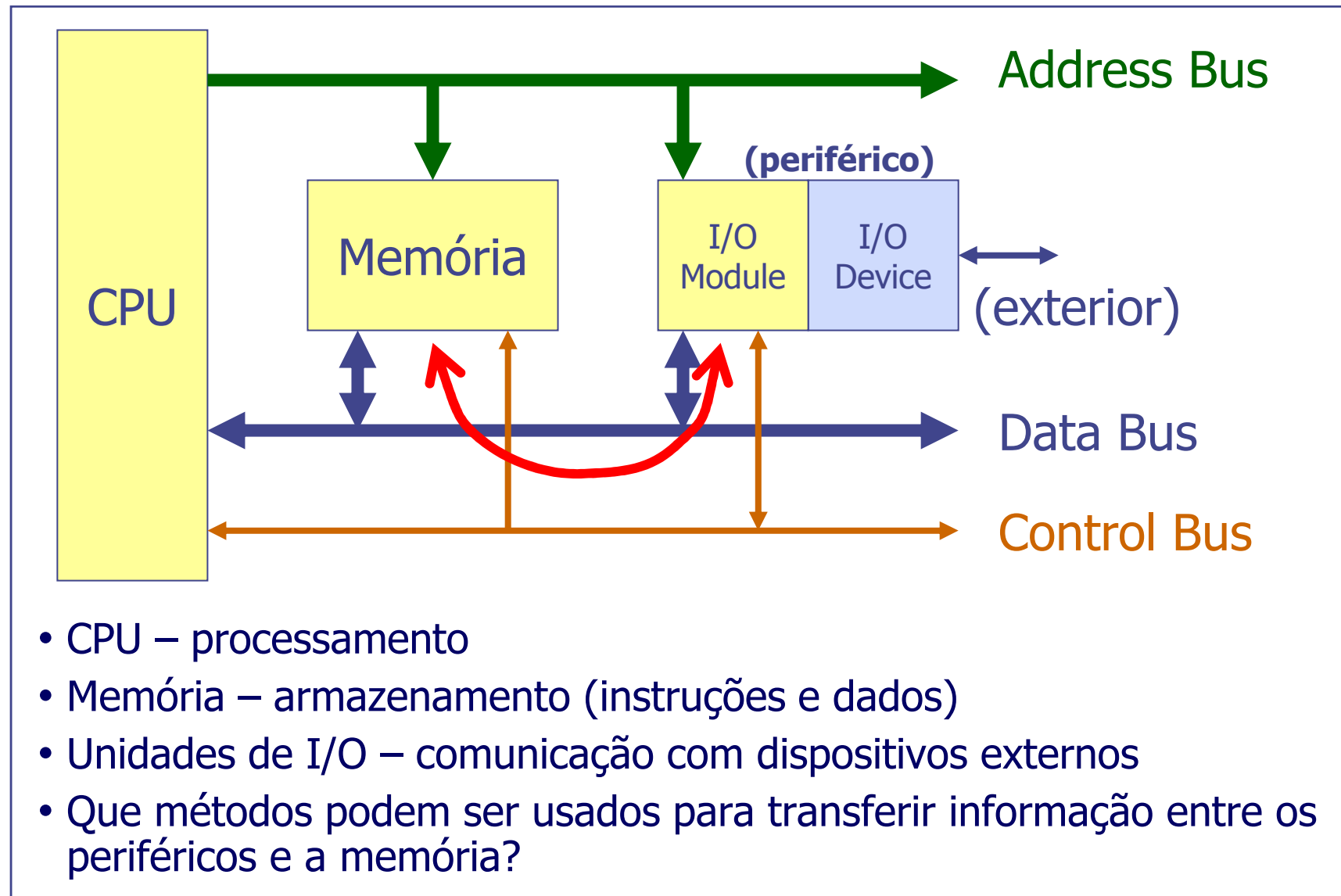


Aulas 5, 6 e 7

- 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

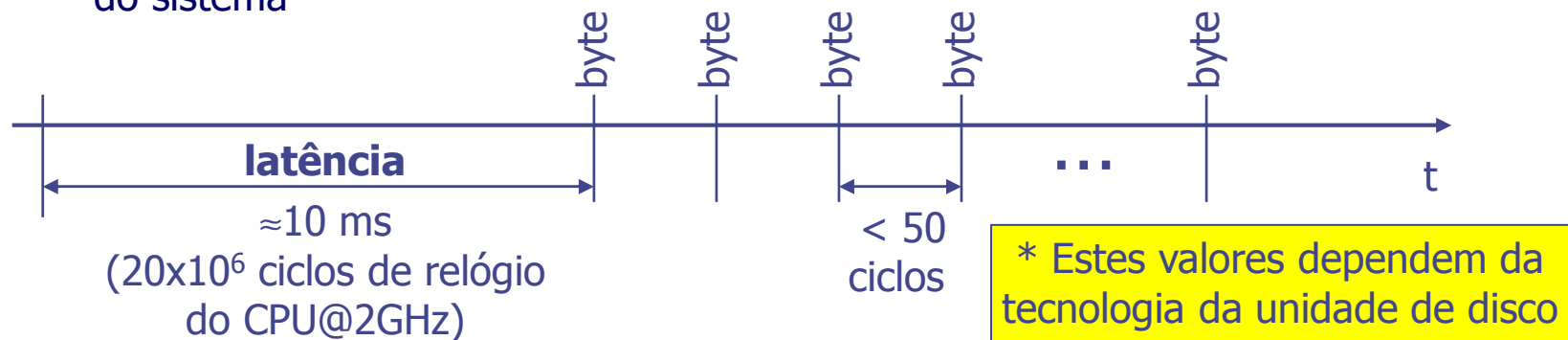
José Luís Azevedo, Bernardo Cunha, Tomás Oliveira e Silva

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 de informação
- Taxa de transferência de pico (*burst*)**: nº máximo de bytes transferidos por segundo, após decorrido o período de 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 – aguarda se necessário, 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 inicia e controla a transferência

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

- E/S por acesso direto à memória (**DMA**)
 - Um dispositivo externo ao CPU (DMA) assegura a transferência de informação diretamente entre a memória e o periférico; o CPU não toma parte no processo de transferência
 - O CPU apenas configura inicialmente o periférico e o DMA; no final o DMA sinaliza o CPU 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)

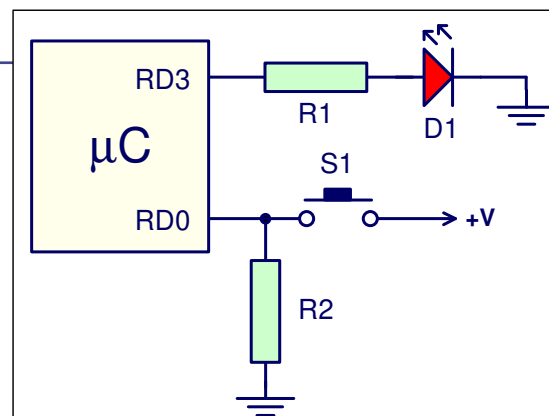
- Exemplo: comutar o estado do LED (ligado ao porto RD3) sempre que é detetada uma transição de 0 para 1 no porto RD0 (assumindo um sinal isento de *bouncing*).

```
# config PIC32 ports
lui    $t0, SFR_BASE_HI #
lw     $t1, TRISD($t0)  #
ori    $t1, $t1, 0x0001 # RD0=1
andi   $t1, $t1, 0xFFF7 # RD3=0
sw     $t1, TRISD($t0)  #

polling {
wh0: lw  $t1, PORTD($t0) # while(1) {
    andi $t2, $t1, 0x0001 #
    beq  $t2, $0, wh0     # while(RD0==0);

    lw   $t3, LATD($t0)  #
    xori $t3, $t3, 0x0008 #
    sw   $t3, LATD($t0)  # LATD3=!LATD3;

    wh1: lw  $t1, PORTD($t0) #
        andi $t1, $t1, 0x0001 #
        bne  $t1, $0, wh1     # while(RD0==1);
        j    wh0              # }
}
```



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 básica, cuja utilização pode ser justificada 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 expressa em %) depende do número de vezes que o ciclo de *polling* for executado
- 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

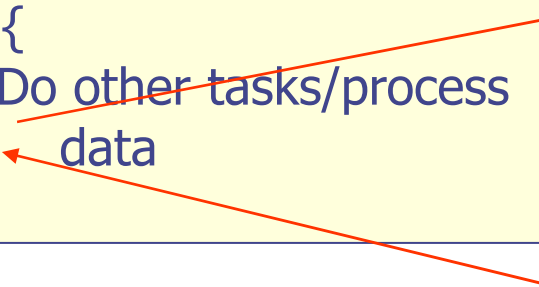
Exemplo: leitura de dados de um periférico

- CPU envia pedido de informação ao periférico (escrita num registo de controlo do periférico)
- CPU continua a execução do programa (com outras tarefas)
- Quando tiver informação disponível, o periférico gera um pedido de interrupção ao CPU
- CPU atende a interrupção:
 - Suspende a execução do programa corrente
 - Salta para a **rotina de atendimento à interrupção** (*interrupt handler*) que transfere a informação
 - Retoma a execução do programa suspenso

E/S por interrupção (exemplo de leitura)

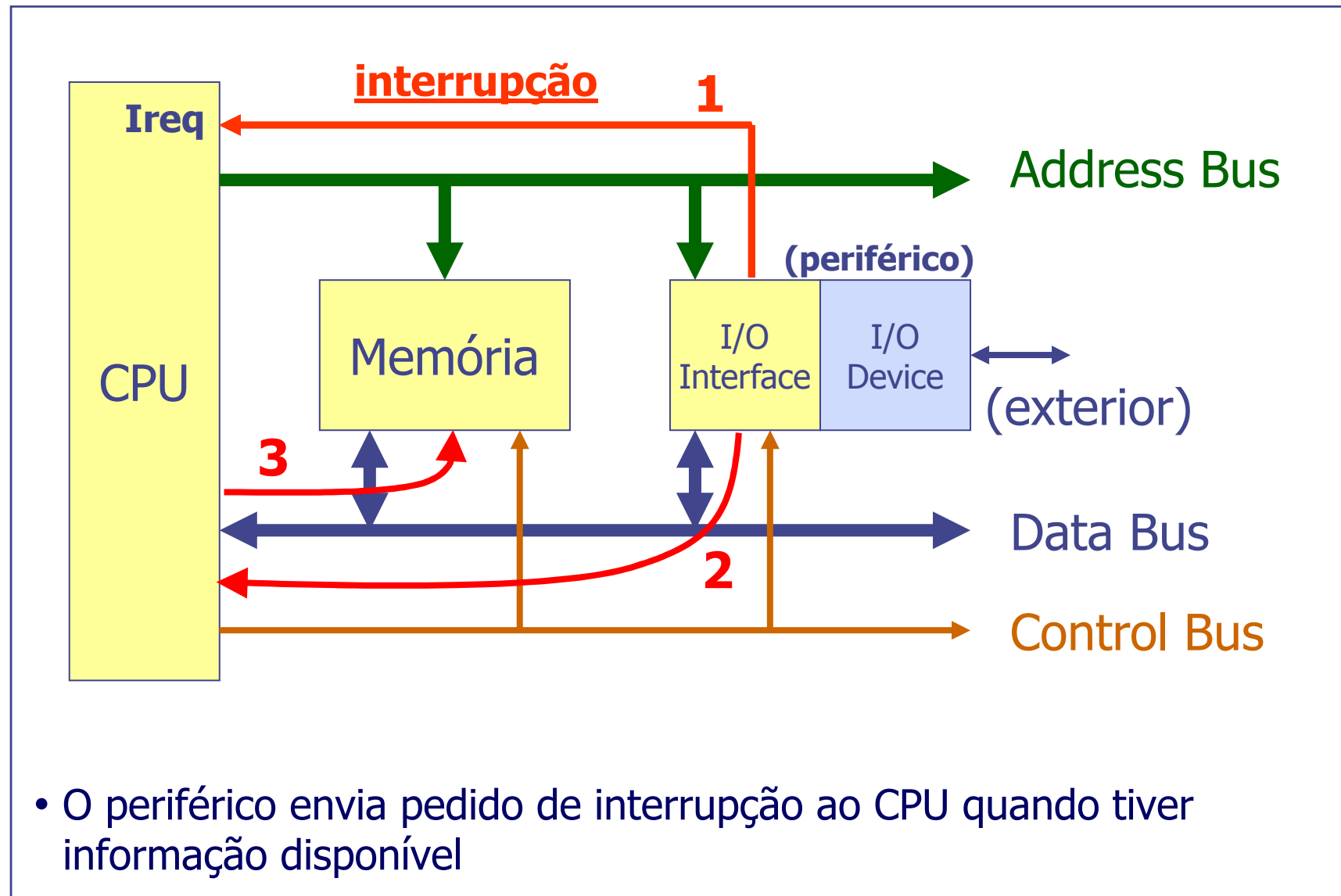
```
// Configure I/O device and interrupt system
(...)
bytesReceived = 0
While(1) {
    (...) // Do other tasks/process
    (...) // data
}
```

```
void interrupt isr(void)
{
    Read byte from I/O Module
    Write byte into Memory
    bytesReceived++
}
```

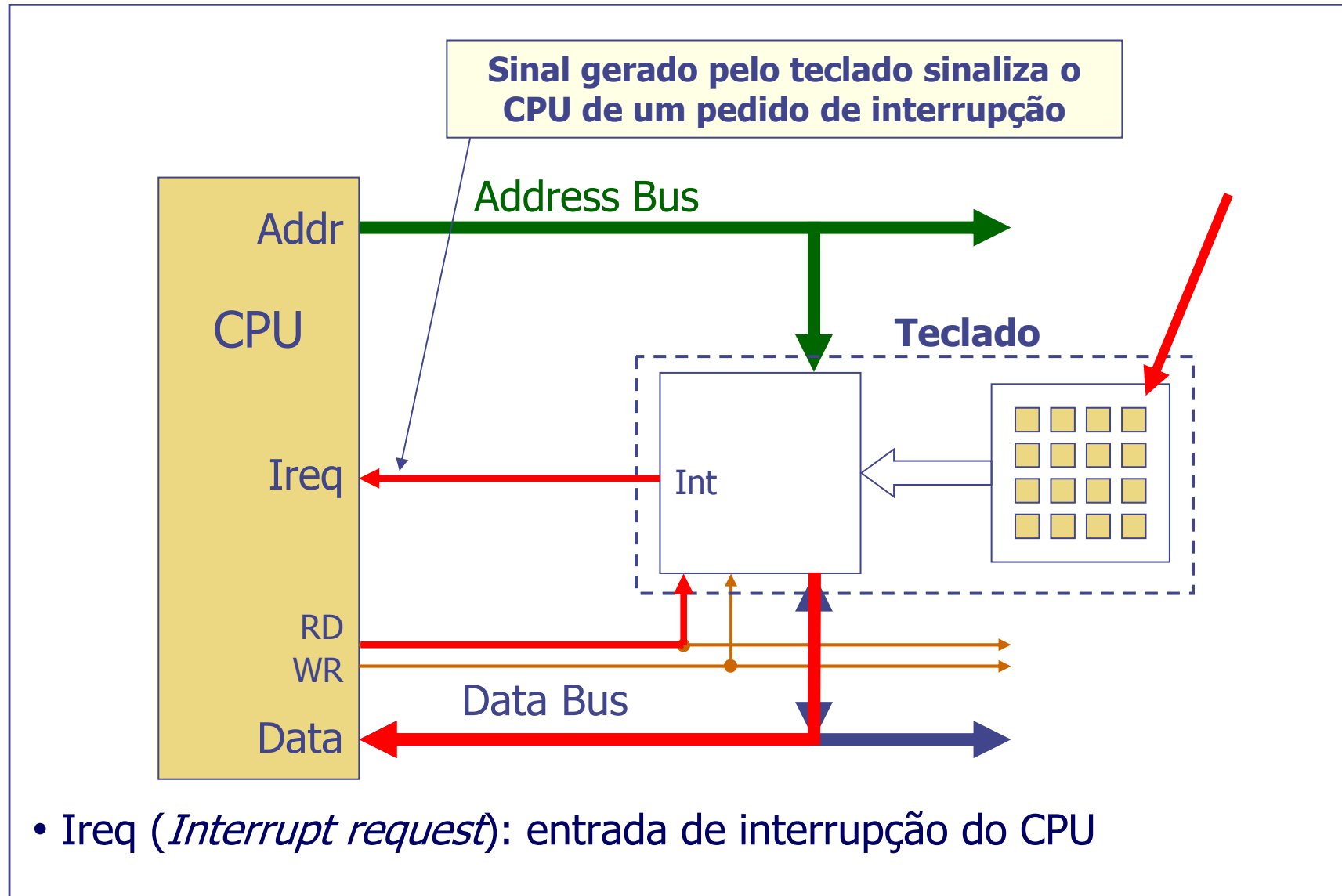


- **Não existe qualquer ciclo de espera.** O periférico gera o pedido de interrupção quando está pronto a transferir a informação
- O programa em execução **pode ser interrompido a qualquer momento**
- A Rotina de Serviço à Interrupção (RSI) tem que **salvaguardar o contexto** do programa (registos internos, ...) 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



E/S por interrupção (exemplo)



E/S por interrupção

- **Exemplo:** versão *interrupt-driven* do exemplo de comutação do estado de um LED (RD3) 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
```

Transição de 0 para 1
em RD0 inicia uma
interrupção

```
isr: # save program context
    (...) # prólogo
    lui  $t0, SFR_BASE_HI
    lw   $t1, LATD($t0) #
    xori $t1, $t1, 0x0008 #
    sw   $t1, LATD($t0) # RD3=!RD3;
    # restore program context
    (...) # epílogo
    eret # exception return
```

- Não existe qualquer ciclo de espera
- O programa em execução é interrompido quando é detetada uma transição 0 para 1 na entrada de interrupção do CPU
- 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

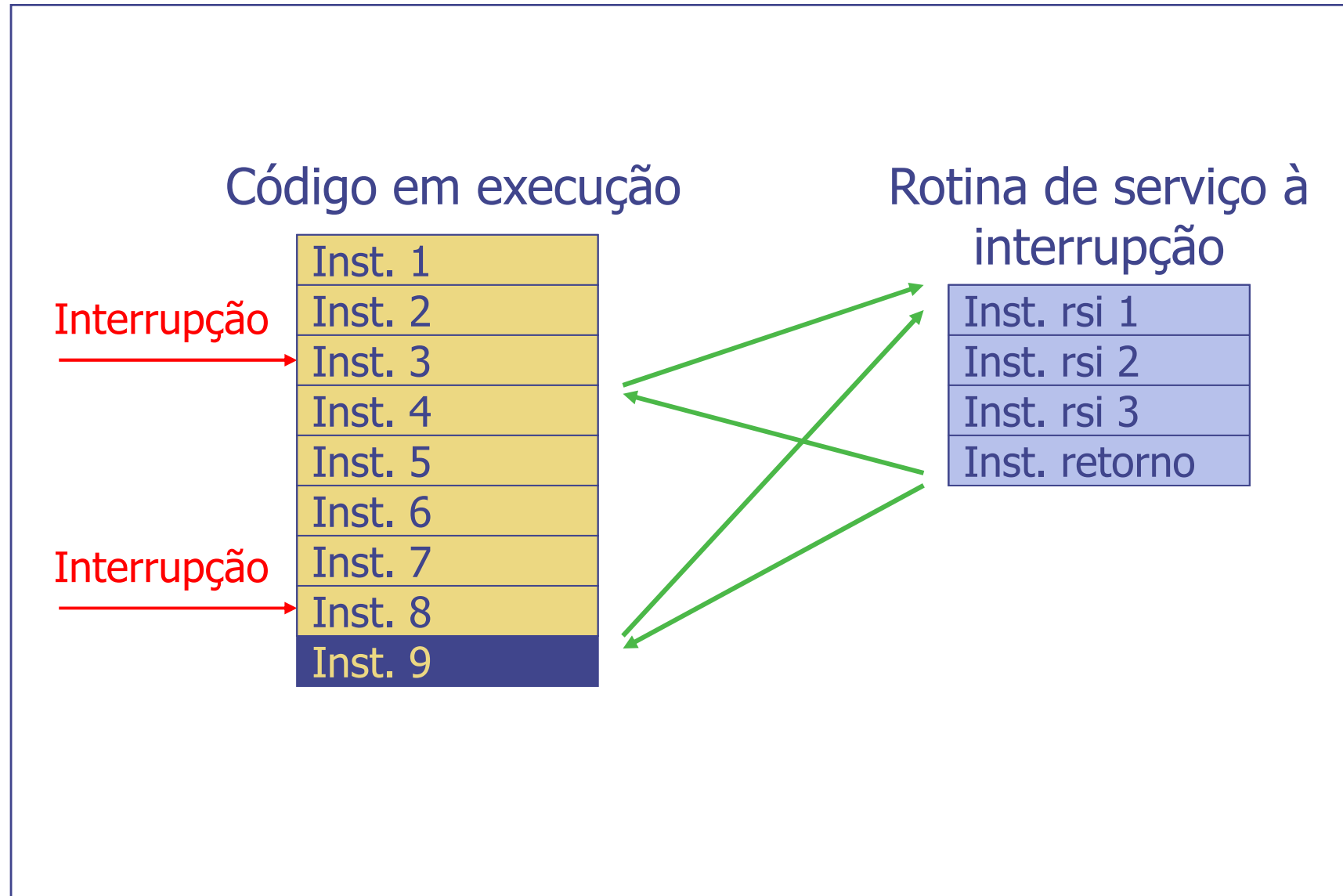
- Exemplos de dispositivos que podem gerar interrupções: teclado, rato, timers, dispositivos de comunicação, dispositivos de armazenamento, ...
- Exemplos de exceções:
 - Divisão por zero
 - *Overflow* numa operação aritmética
 - Tentativa de execução de uma instrução cujo OpCode é desconhecido
 - Acesso 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 execução do pedido
 - Reposição do contexto do CPU
 - Retorno ao programa que executou o "syscall"

Atendimento de interrupções e exceções

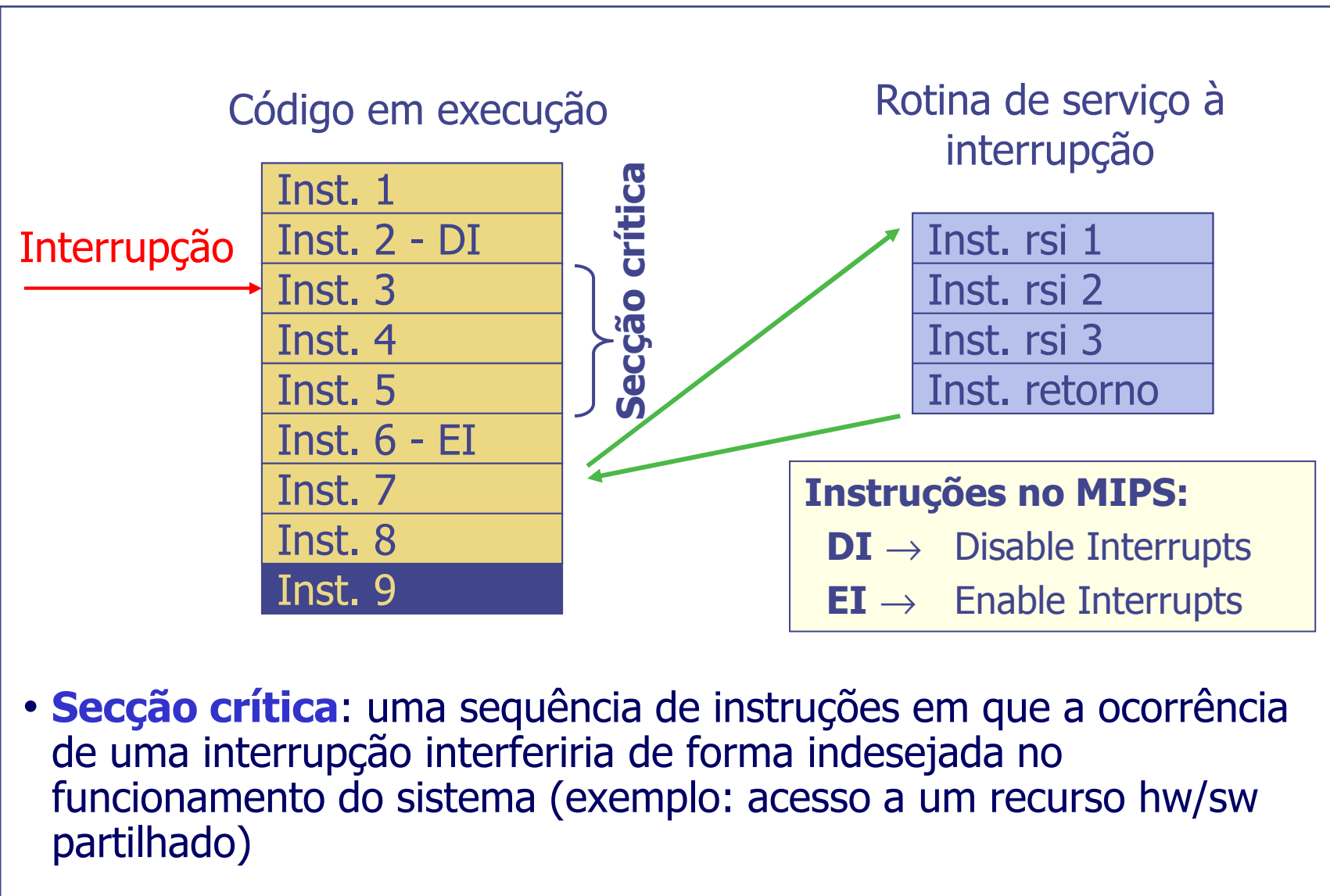
- **Exceções:** a instrução que gerou a exceção não termina, sendo a execução passada de imediato para a rotina de tratamento da exceção
- **Interrupções:** a passagem da execução para a rotina de tratamento da interrupção só acontece quando for concluída a instrução que está a ser executada no momento em que a interrupção surge
- As interrupções no ciclo de execução de instruções 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 and read operands
    Execute operation and store result
}
```


Processamento de interrupções

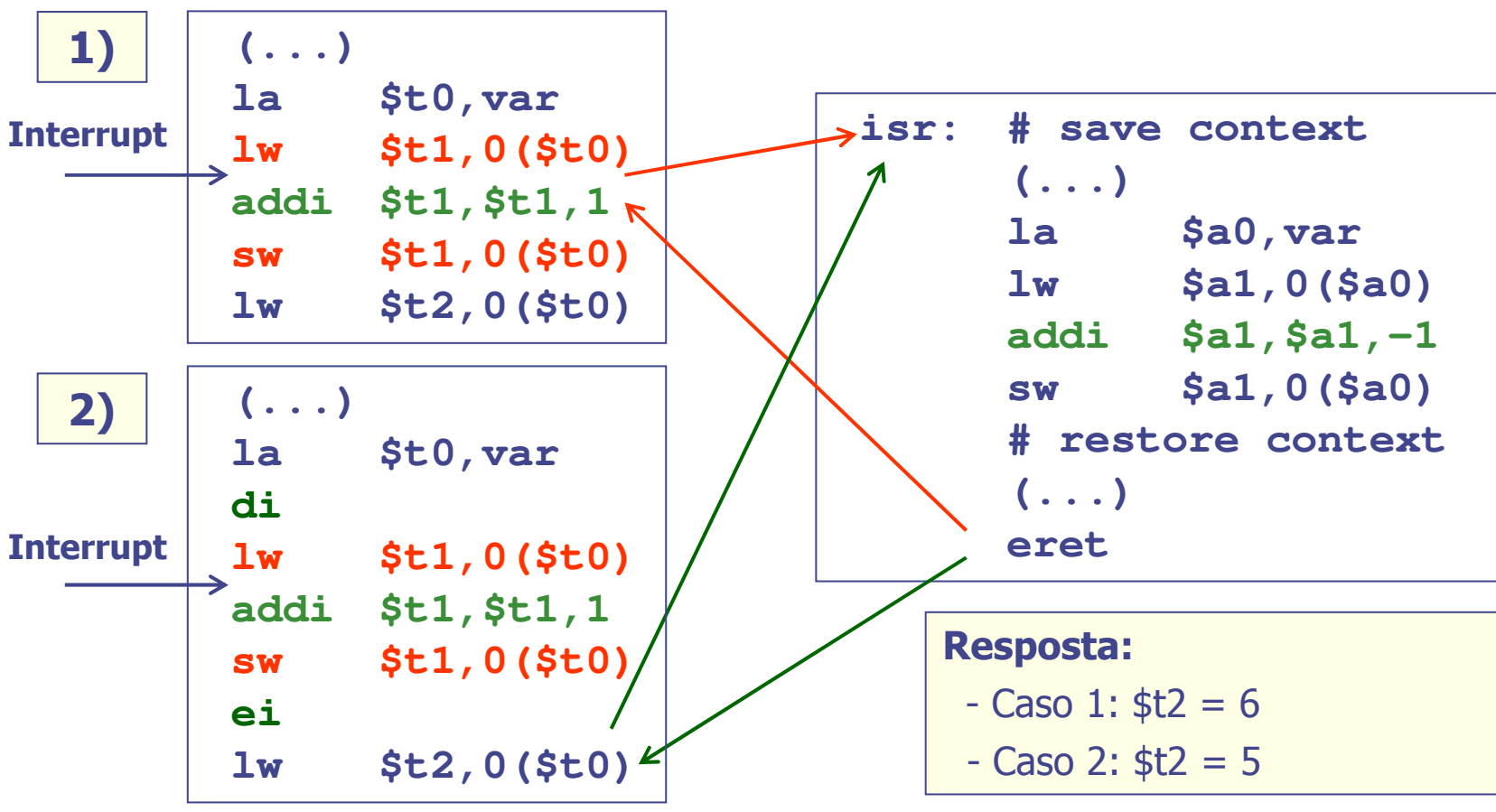


Ativação/desativação global das interrupções



Processamento de interrupções – secção crítica (exemplo)

- A variável "**var**" pode ser lida e alterada na RSI e no programa principal. Se "**var**" tem o valor 5 antes da ocorrência da interrupção, qual o valor lido para o registo **\$t2**, 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, com a execução da 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*) ("**PRÓLOGO**")
 2. .. ações associadas ao processamento da interrupção..
 3. Reposição do contexto do programa interrompido:
registos internos do CPU ← memória (*stack*) ("**EPILOGO**")
 4. Conclusão 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 (pontos 1 a 4 do slide anterior mais eventual conclusão de secção crítica do código)

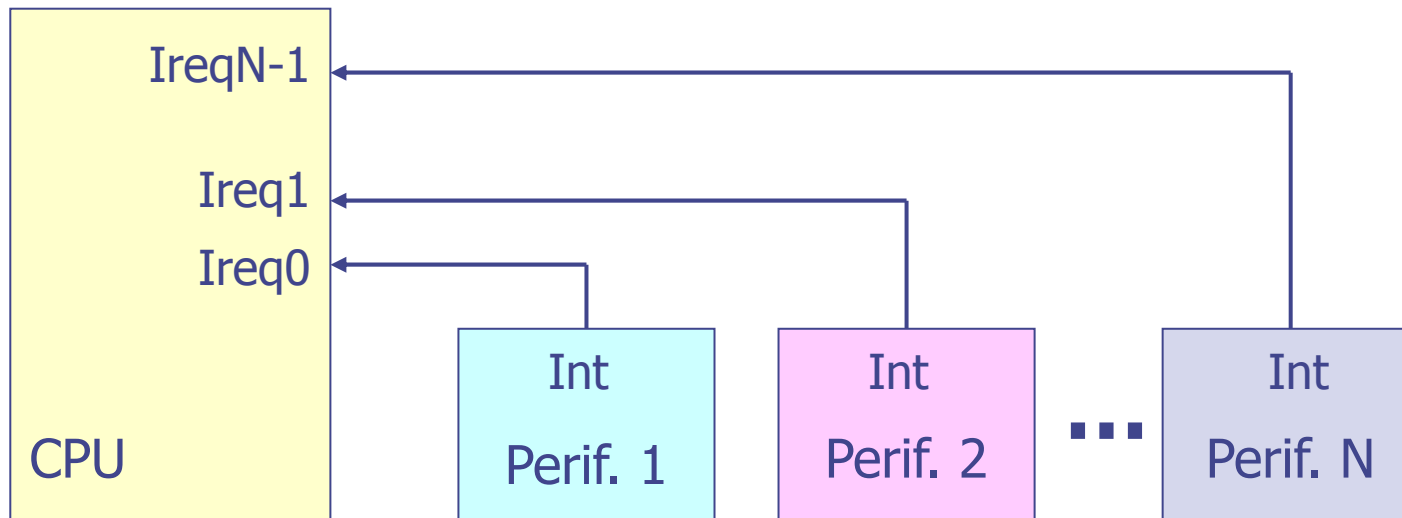
Overhead do método de transferência por interrupção

- O **overhead** global do método de transferência por interrupção é, no essencial, 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 do PIC32, cerca de 50 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

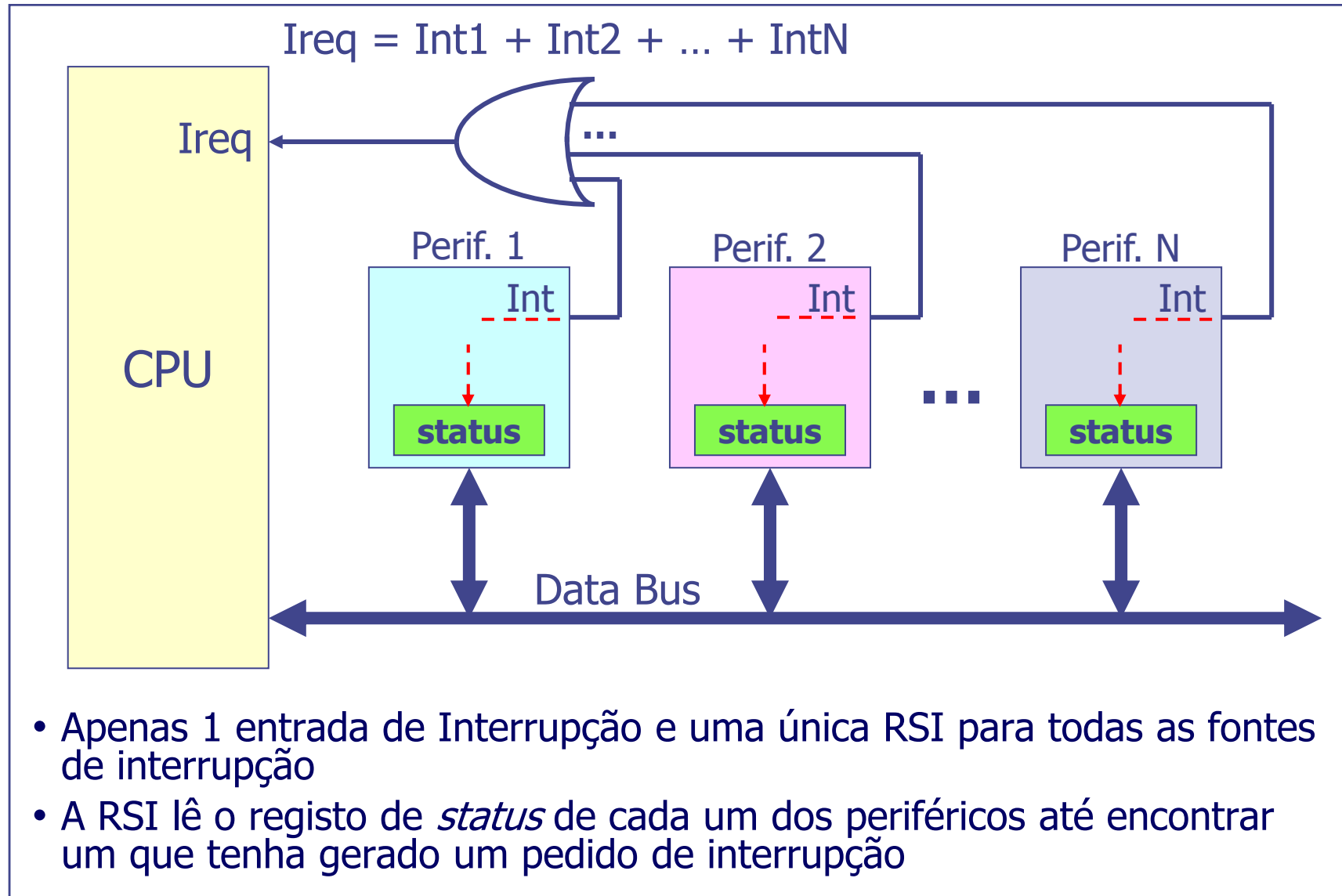
- Num sistema real é expectável que vários periféricos possam ter a capacidade de gerar interrupções
- Como organizar o sistema de interrupções para permitir a ligação de vários periféricos?
 - **Múltiplas linhas de interrupção**
 - **Identificação da fonte de interrupção por software**
 - **Interrupções 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 register of peripheral 1  
    If( interrupt_bit = ON) {  
        peripheral_isr_1()  
    }  
  
    Read status register of peripheral 2  
    If( interrupt_bit = ON) {  
        peripheral_isr_2()  
    }  
  
    (...)   
  
    Read status register of 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

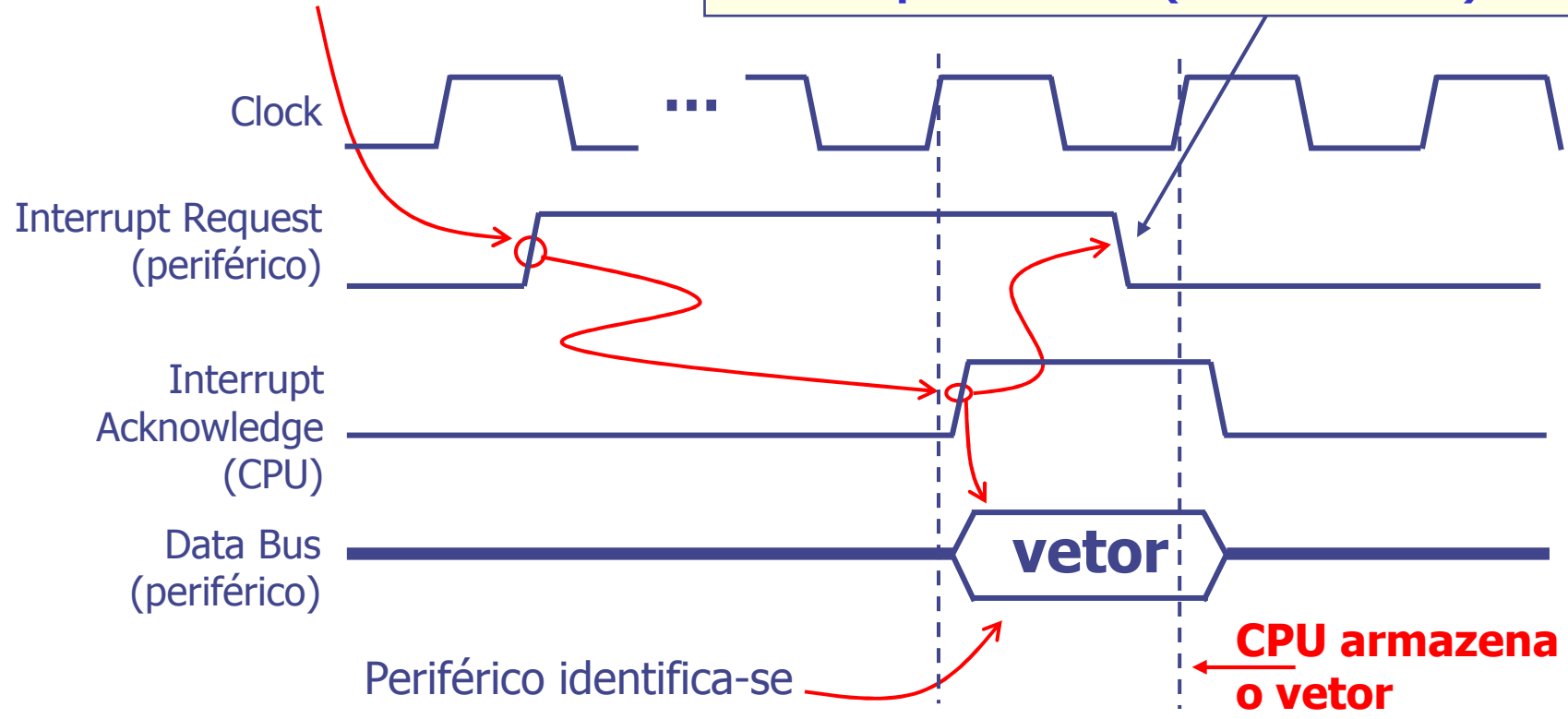
- CPU tem apenas 1 entrada de interrupção
- A identificação da fonte é feita por hardware
- Cada periférico possui um identificador único, designado por **vetor**
- Uma RSI para cada vetor de interrupção
- Durante o processo de atendimento, na fase de identificação da fonte, o periférico gerador da interrupção identifica-se através do seu vetor
- O vetor vai ser usado depois como índice de uma tabela que contém: ou o endereço de cada uma das RSI, ou instruções de salto incondicional para as RSI

Interrupções vetorizadas

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

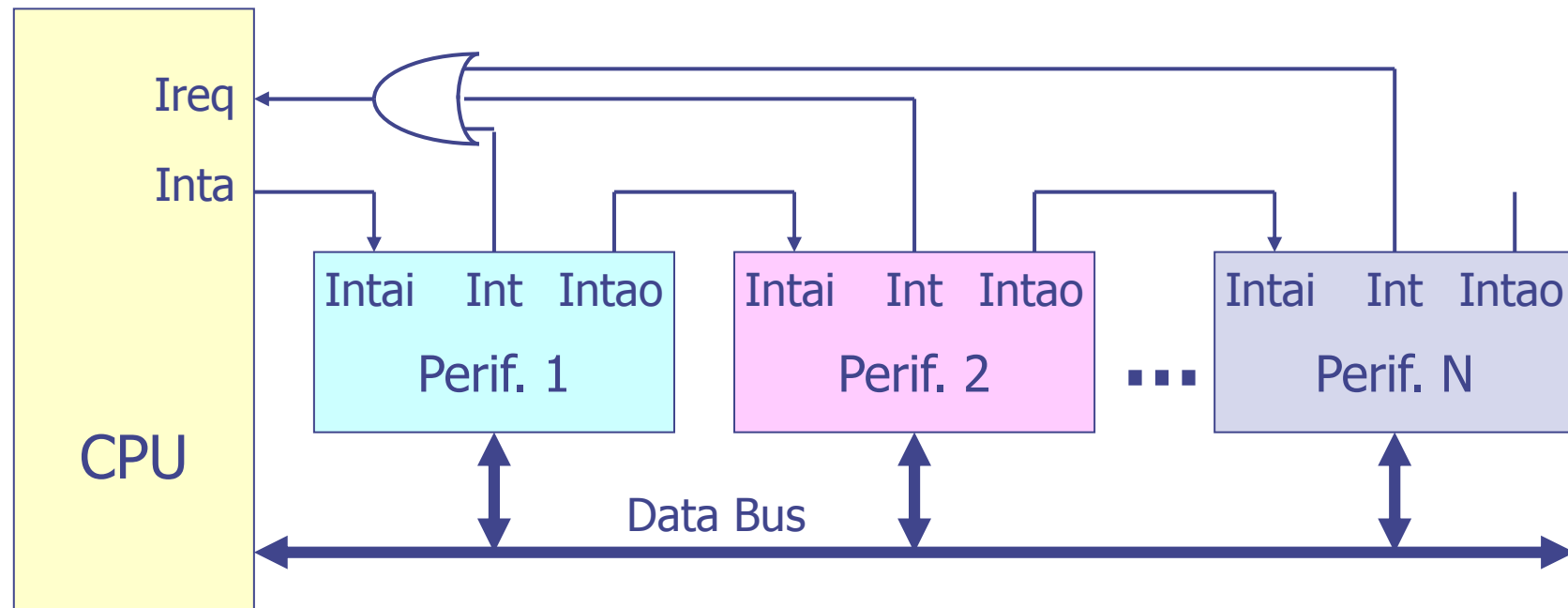
Periférico gera pedido de interrupção

O reset do sinal que levou à ativação do Ireq pode, em algumas arquiteturas, ser feito por *software* (caso do PIC32)



Interrupções vetorizadas – *daisy chain*

- Periféricos podem estar organizados numa estrutura *daisy chain*



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

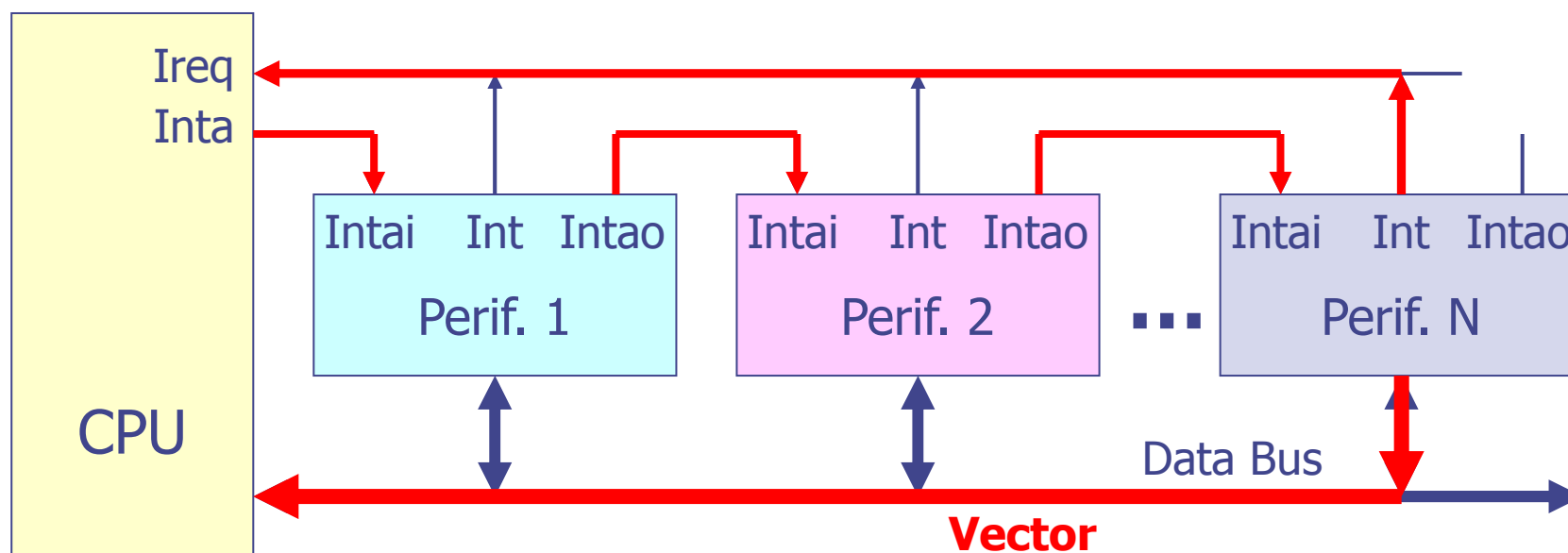
Intai/o - Interrupt Acknowledge in/out

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 e bloqueia a propagação do sinal "Interrupt Acknowledge"
 4. O CPU lê o vetor e usa-o como índice da tabela que contém os endereços das RSI ou instruções de salto para as RSI.

Interrupções vetorizadas – *daisy chain*

- Exemplo de identificação da fonte de interrupção

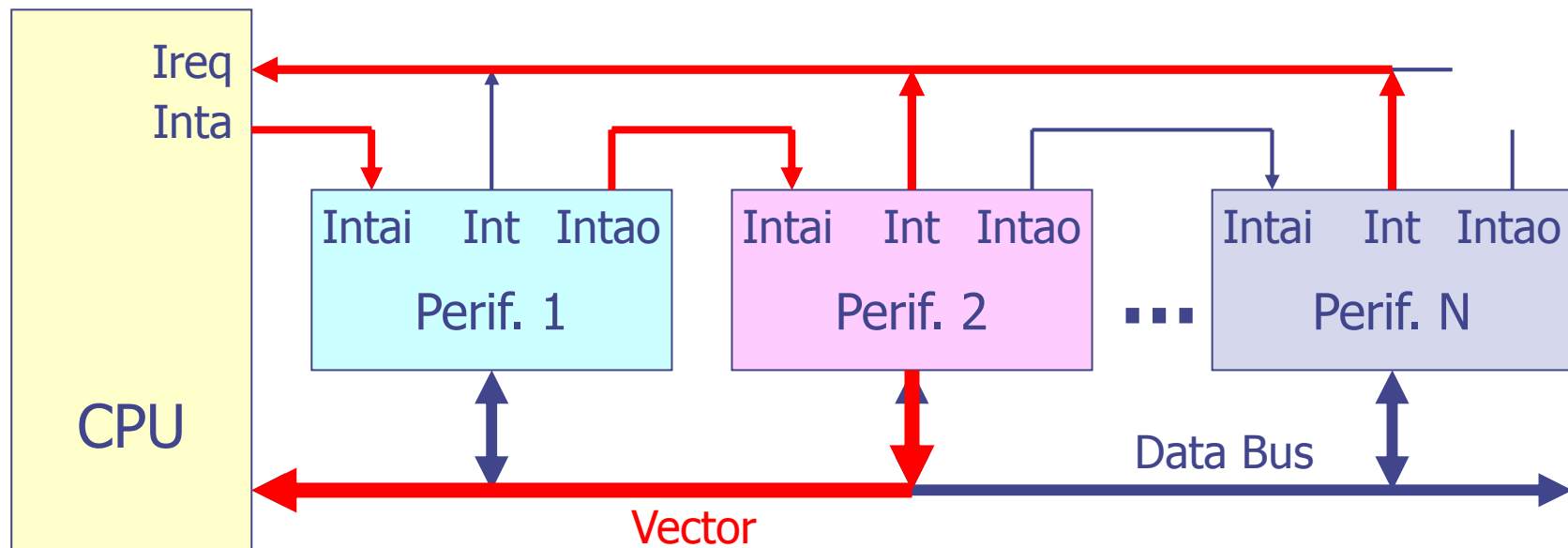


$$Ireq = Int1 + Int2 + \dots + IntN$$

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

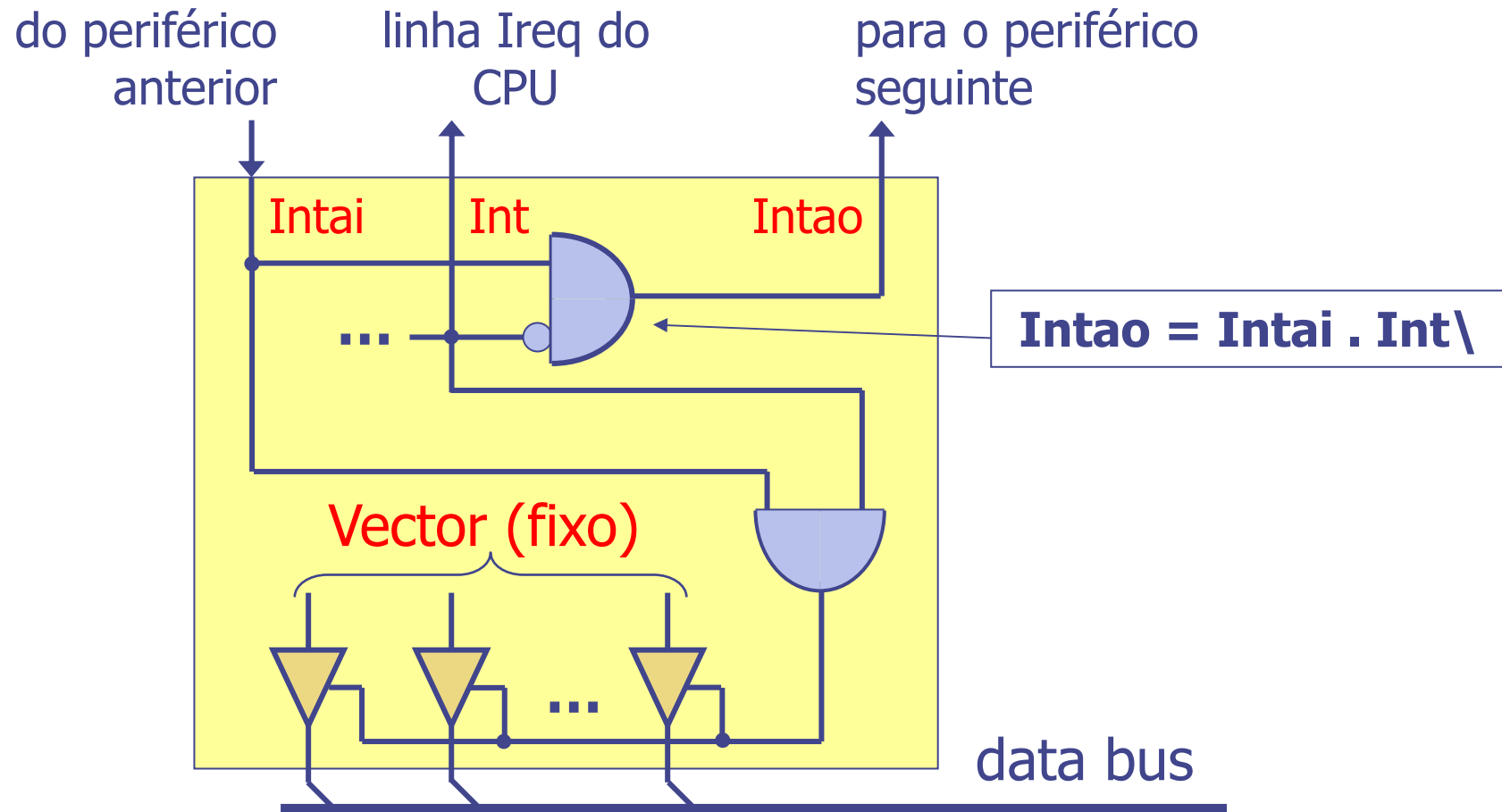


$$Ireq = Int1 + Int2 + \dots + IntN$$

- 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
- **Tabela inicializada com os endereços de todas as RSI:**
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

- **Tabela inicializada com instruções de salto para as RSI:** 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:

