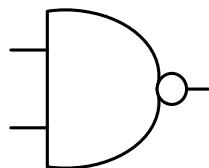


# Aula 0

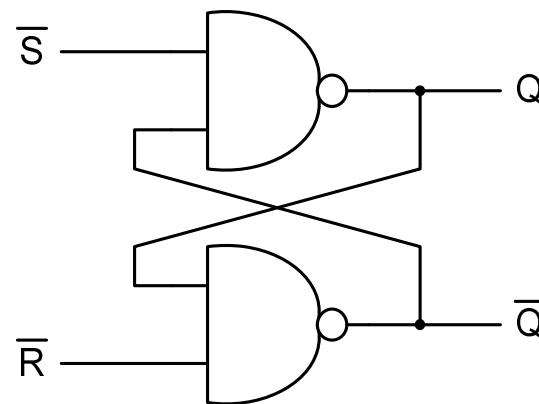
No princípio, era o verbo...



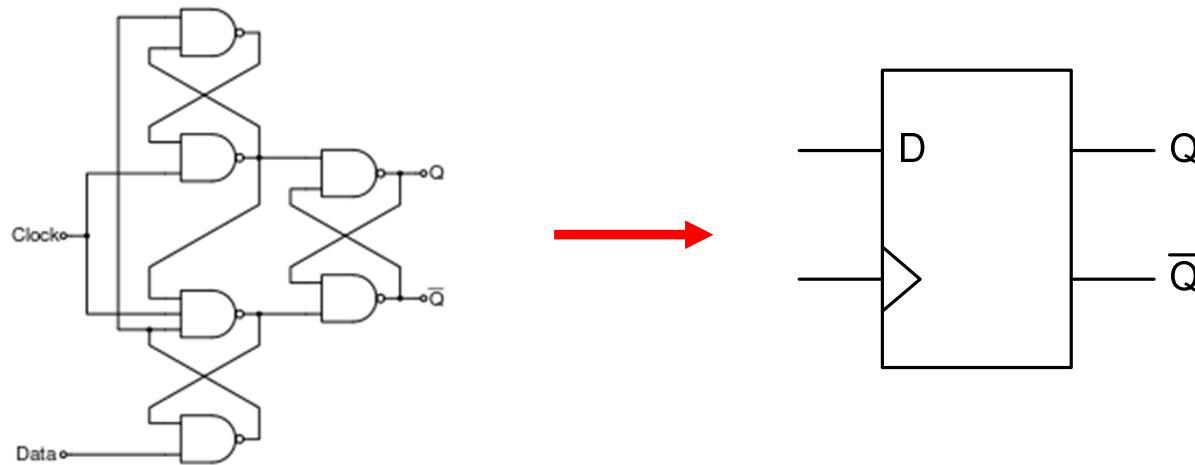
José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

# Era uma vez...

- E do verbo se fez...

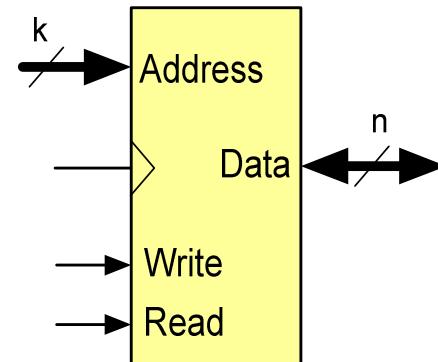
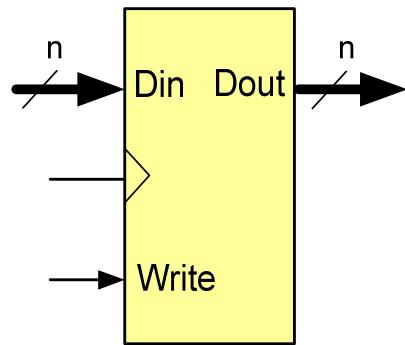


- E do Latch SR surgiu...

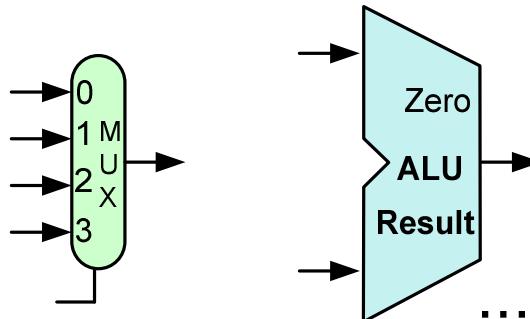


# Era uma vez...

- E do FF tipo D construíram-se:

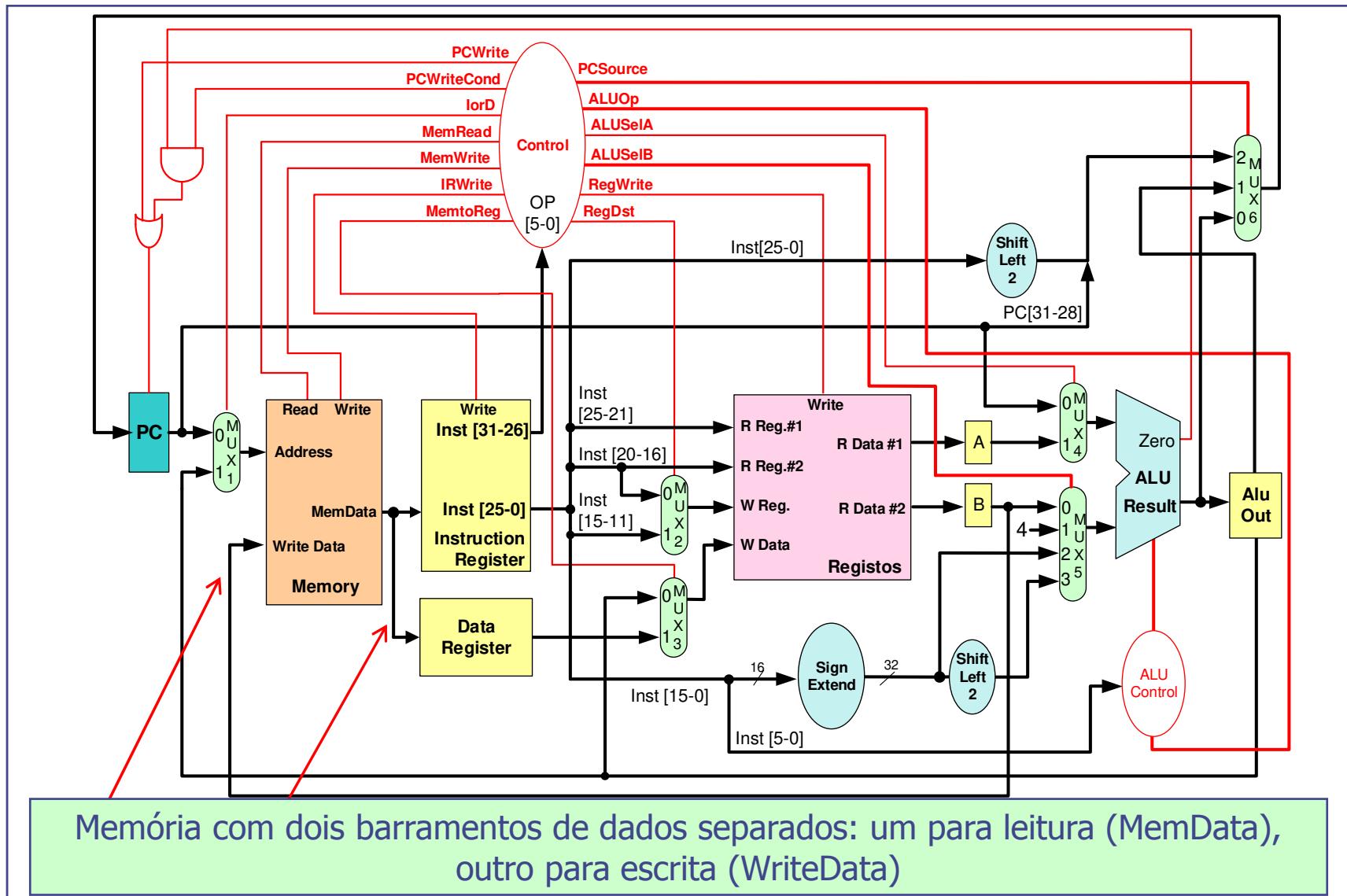


- E do verbo também nasceram:

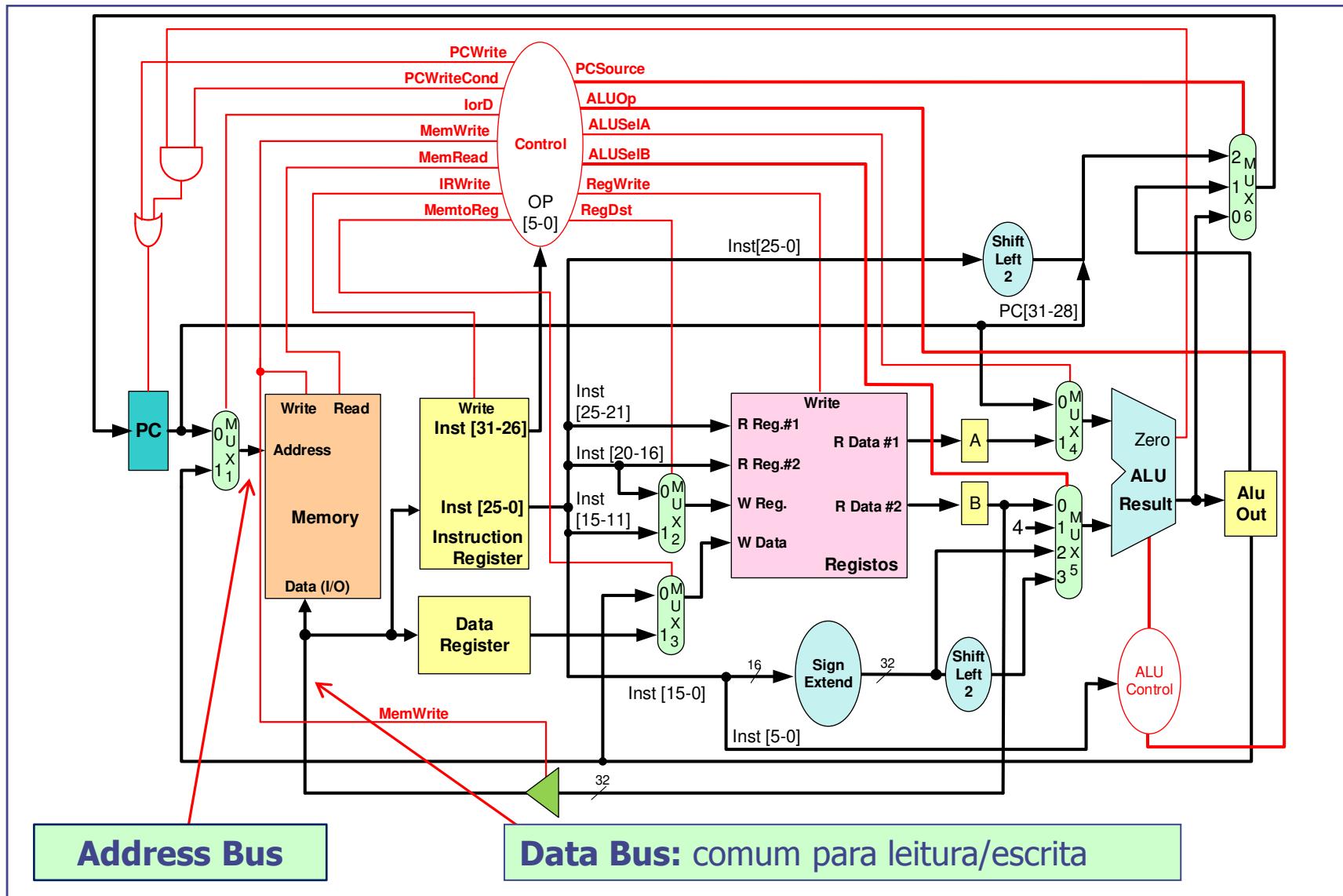


- E de tudo isto resultou...

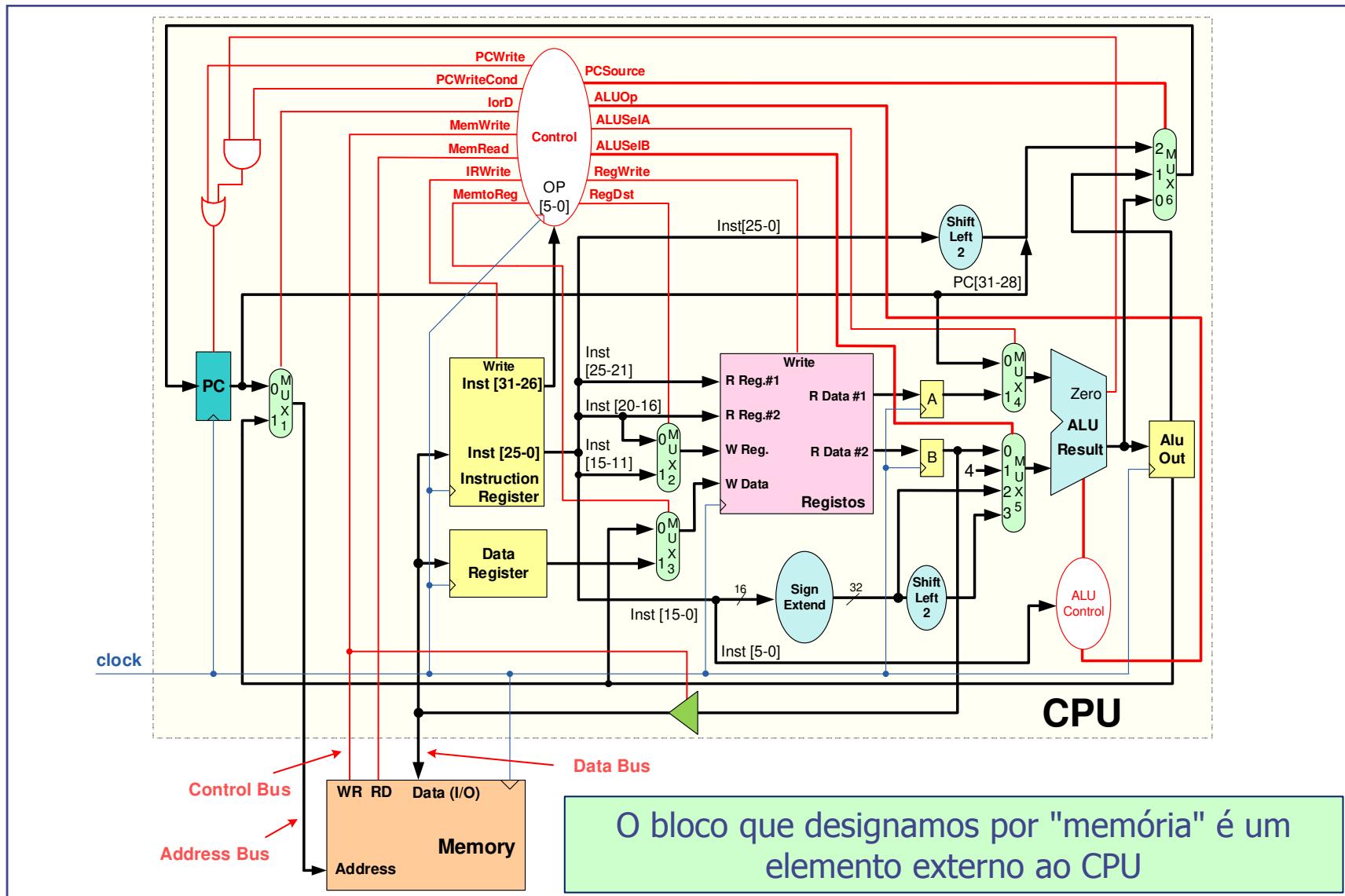
# Versão *multi-cycle* simplificada de uma arquitetura MIPS



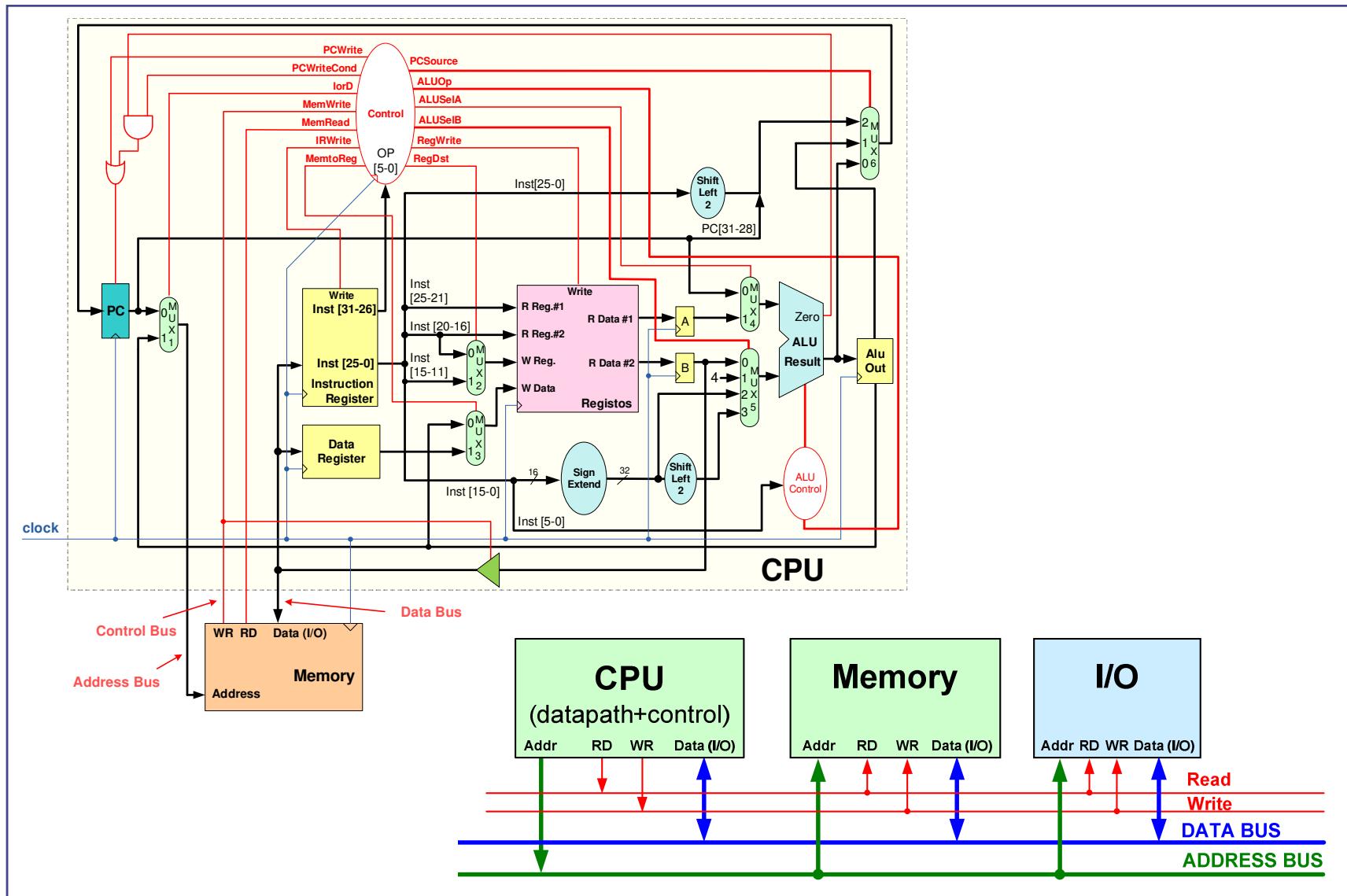
# Versão *multi-cycle* simplificada de uma arquitetura MIPS



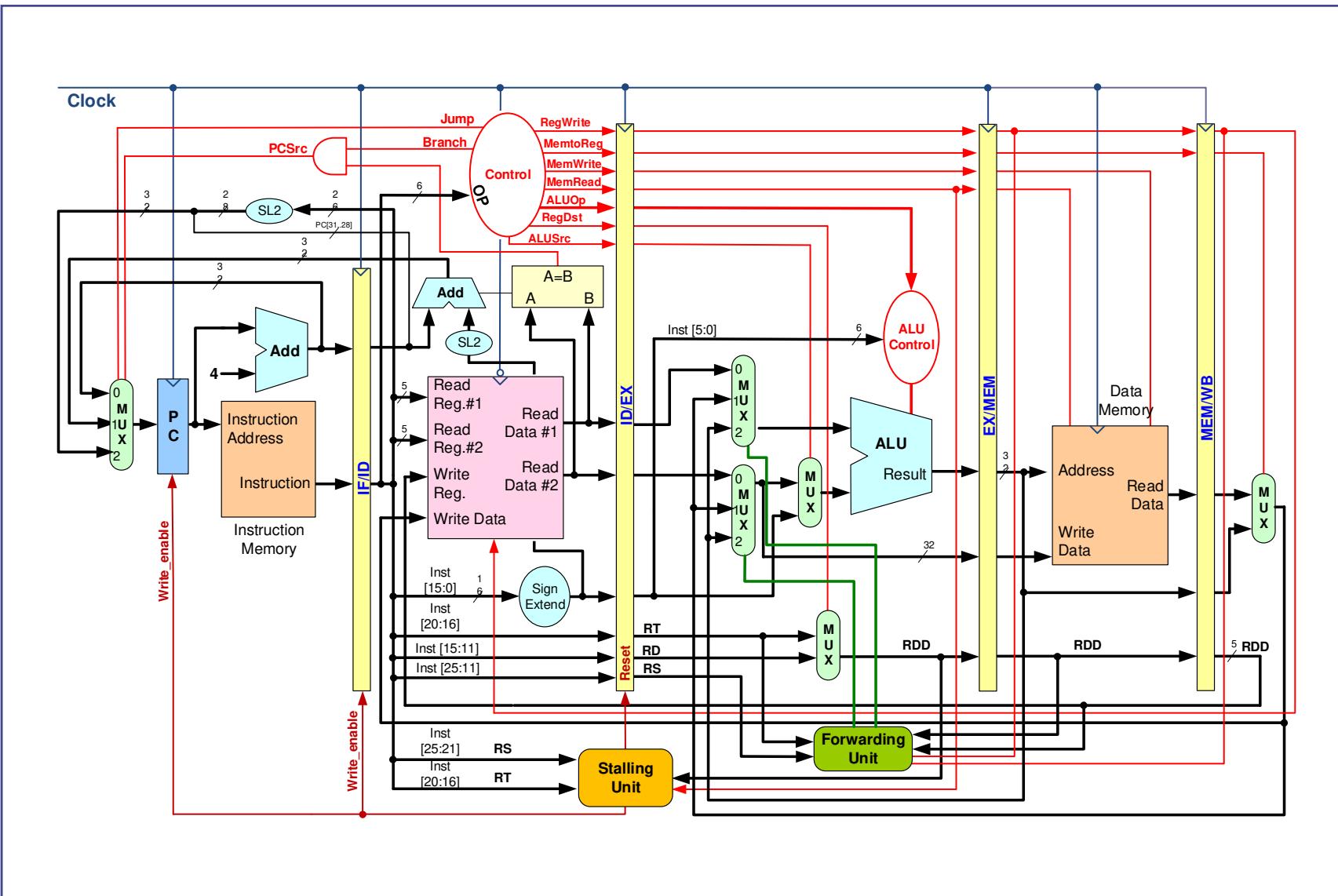
# Versão *multi-cycle* simplificada de uma arquitetura MIPS



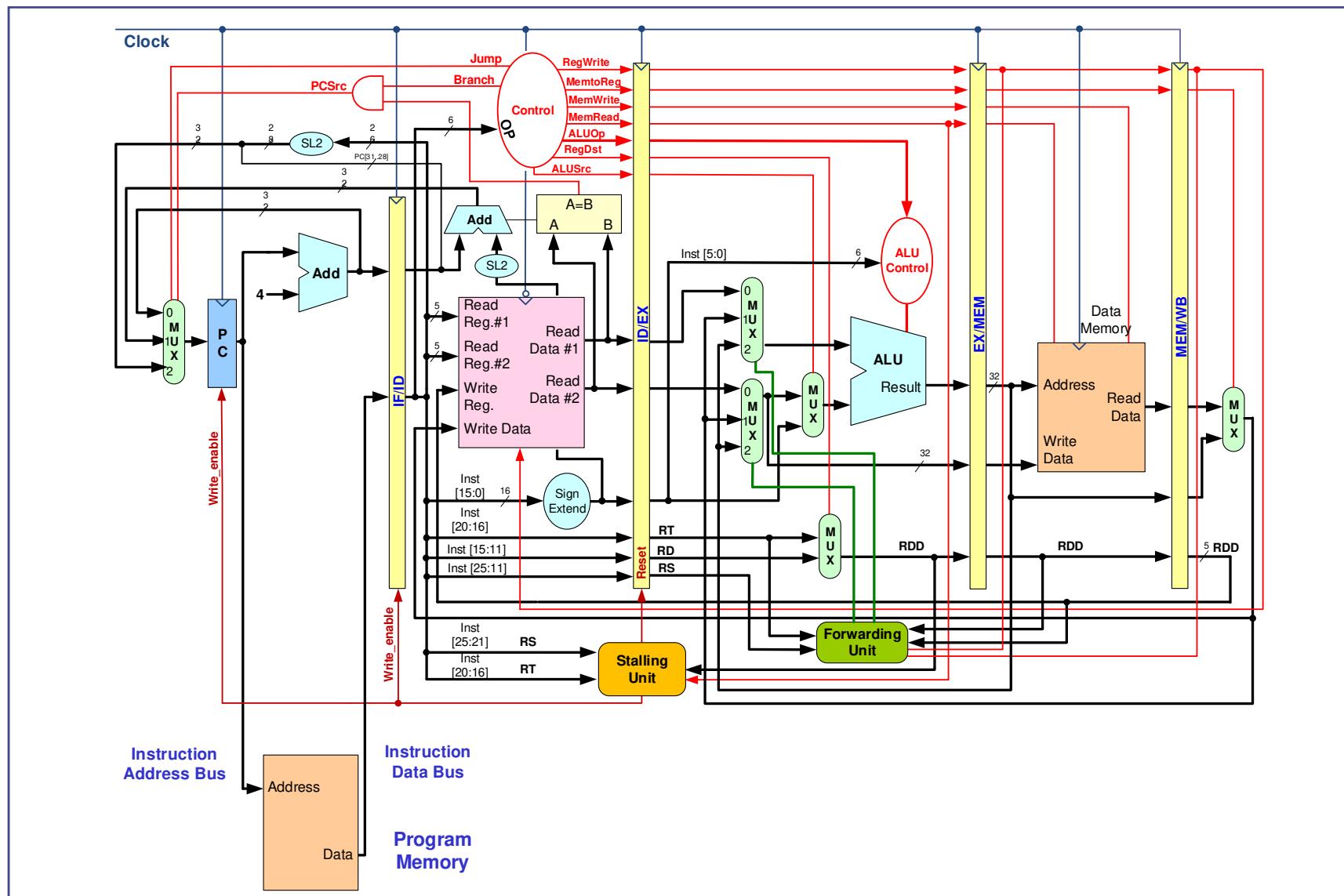
# Arquitetura de von-Neumann



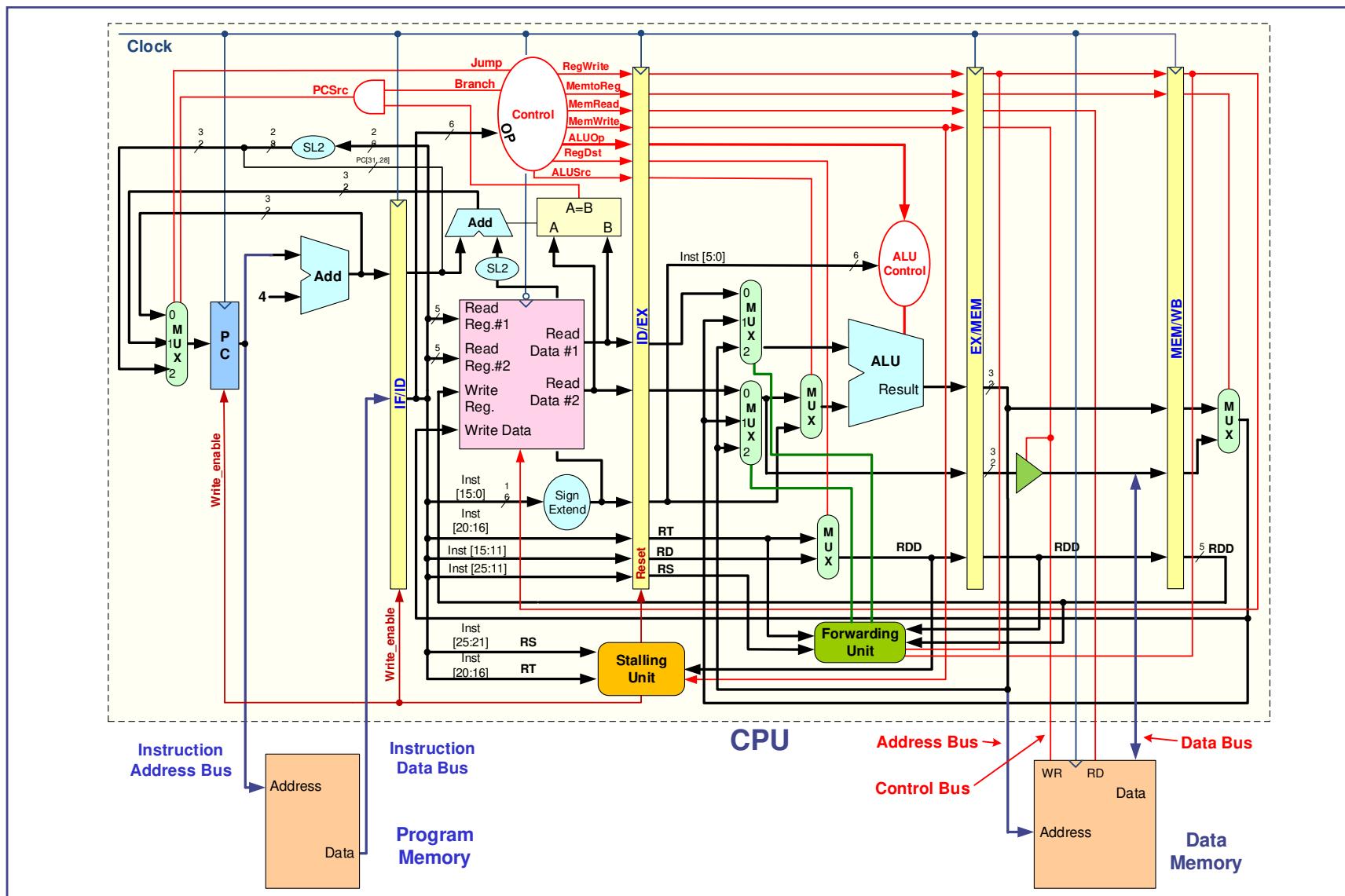
# Versão simplificada de uma arquitetura MIPS *pipelined*



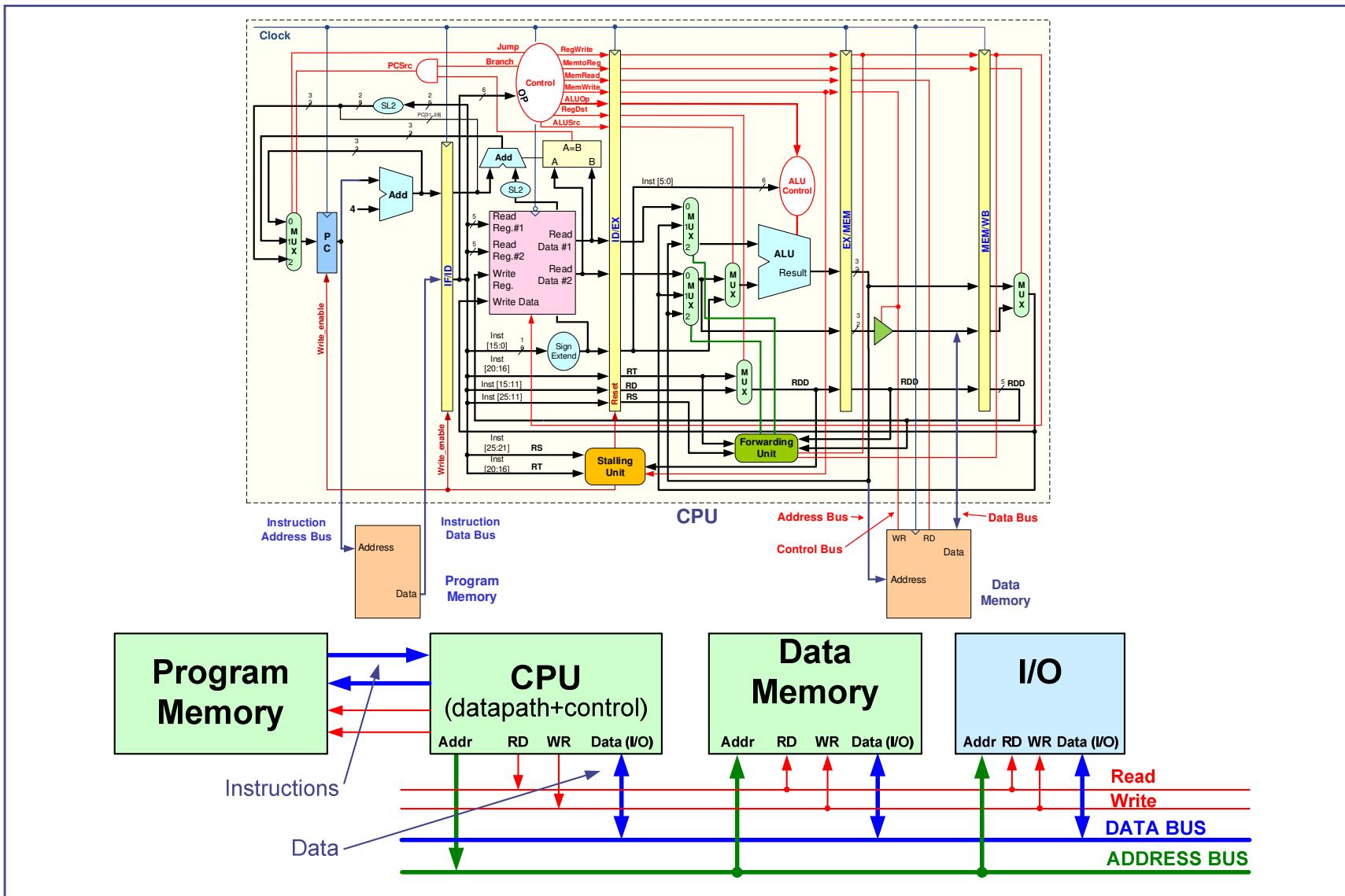
# Versão simplificada de uma arquitetura MIPS *pipelined*



# Versão simplificada de uma arquitetura MIPS *pipelined*



# Arquitetura de Harvard



# Aula 1

- Microprocessadores *versus* microcontroladores
- Sistemas embebidos
- Desenvolvimento de aplicações para microcontroladores
- O Microcontrolador PIC32 da Microchip
- Ferramentas de desenvolvimento para a placa DETPIC32

José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

# Microcontroladores *versus* Microprocessadores

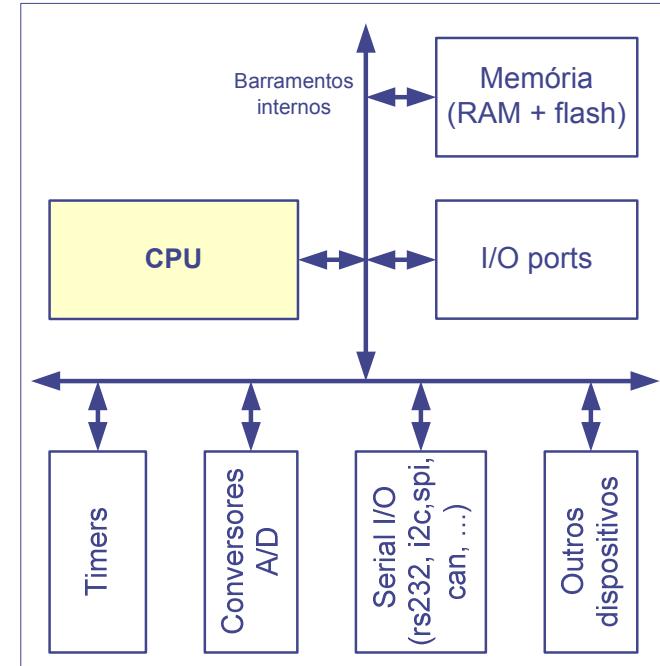
- Microprocessador:
  - Circuito integrado com um (ou mais) CPU
  - Não tem memória interna (além do banco de registos)
  - Os barramentos estão disponíveis no exterior
  - Para obter um sistema completo é necessário acrescentar RAM, ROM e periféricos
  - Pode operar a frequências elevadas (> 3GHz)
  - Sistemas computacionais de uso geral
- Microcontrolador:
  - Circuito integrado inclui CPU, RAM, ROM, periféricos
  - Frequência de funcionamento normalmente baixa
  - Baixo consumo de energia
  - Disponibiliza uma grande variedade de periféricos e interfaces com o exterior
  - Rapidez de resposta a eventos externos (Sistemas de Tempo Real)
  - Utilizado em tarefas específicas (por exemplo controlo da velocidade de um motor)

# Sistema embedido

- Sistema computacional especializado
  - realiza uma tarefa específica ou o controlo de um determinado dispositivo
- Tem requisitos próprios e executa apenas tarefas pré-definidas
- Recursos disponíveis, em geral, mais limitados que num sistema computacional de uso geral (e.g. menos memória, ausência de dispositivos de interação com o utilizador)
- Tem, em regra, um custo inferior a um sistema computacional de uso geral
- Pode ser implementado com base num microcontrolador
- Pode fazer parte de um sistema computacional mais complexo
- Exemplos de aplicação:
  - eletrónica de consumo, automóveis, telecomunicações, domótica, robótica, iot, ...

# Microcontrolador – principais características

- Dispositivo programável que integra, num único circuito integrado, 3 componentes fundamentais:
  - Uma Unidade de Processamento
  - Memória (volátil e não volátil)
  - Portos de I/O (E/S)
- Inclui outros dispositivos de suporte (periféricos), tais como:
  - Timers
  - Conversor A/D
  - Serial I/O (rs232, i2c, spi, can, ...)
  - ...
- Barramentos (dados, endereços e controlo) interligam todos estes dispositivos (não estão, geralmente, acessíveis externamente)
- Externamente há, em geral, pinos que podem ser configurados programaticamente para diferentes funções (versatilidade)



# Processo de desenvolvimento de aplicações para µC

- Computador *host* (e.g. PC) / Computador *target* (µC)
  - Estas plataformas são, geralmente, distintas (CPU, sistema operativo, dispositivos de interface com o utilizador, ...)
- **Edição do programa** numa linguagem de alto nível (por ex. C), ou, em casos pontuais, em *assembly* do microcontrolador
- **Geração do código** usando um *cross-compiler* / *cross-assembler*
  - Um *cross-compiler* (compilador-cruzado) é um compilador que corre na plataforma A (o *host*, e.g. o PC) e que gera código executável para a plataforma B (o *target*, e.g. o µC)
  - A utilização de *cross-compilers* / *cross-assemblers* é a regra no desenvolvimento de aplicações para microcontroladores uma vez que, geralmente, estes não disponibilizam os recursos necessários e as interfaces adequadas
- **Transferência para a memória do microcontrolador** (geralmente memória não volátil) do código produzido pelo *cross-compiler* / *cross-assembler*
- **Teste e depuração** (*debug*) do programa

# Transferência de programas para o microcontrolador

- Para a transferência de um programa executável para a memória do microcontrolador pode ser utilizado um dos seguintes métodos:
  - **Programa-monitor** (software)
  - **Bootloader** (software)
  - **In-Circuit Debugger** (hardware)
- Programas-monitor e *bootloaders*:
  - Executam no arranque do sistema
  - A comunicação com o *host* é efetuada por RS232 / USB
- *In-Circuit Debugger*
  - Dispositivo externo proprietário, i.e., específico para um dado fabricante
  - Pode usar uma interface de comunicação standard (JTAG) ou uma interface proprietária

# Transferência de programas para o microcontrolador

- **Programa-monitor:** é um programa que reside, de forma permanente, na memória não volátil do microcontrolador:
  - disponibiliza funções de transferência e execução de programas
  - implementa outras funções úteis no *debug* de novos programas, como por exemplo, visualização do conteúdo de registos internos do microprocessador, visualização do conteúdo da memória, execução passo a passo, etc.
- **Bootloader:** é um programa que reside, de forma permanente, na memória do microcontrolador e que disponibiliza apenas funções básicas de transferência e execução de um programa.

# Transferência de programas para o microcontrolador

- **In-Circuit Debugger (ICD)**: um dispositivo de hardware controlado por software no *host* que permite a transferência e execução controlada de um programa num microcontrolador



- O ICD é, normalmente, necessário para a transferência inicial de um programa-monitor ou de um *bootloader*.

# Tecnologias de memória não volátil

- **ROM** – programada durante o processo de fabrico
- **PROM** – *Programmable Read Only Memory*: programável uma única vez
- **EPROM** – *Erasable PROM*: escrita em segundos, apagamento em minutos (ambas efetuadas em dispositivos especiais)
- **EEPROM** – *Electrically Erasable PROM*
  - O apagamento e a escrita podem ser efetuados no próprio circuito em que a memória está integrada
  - O apagamento é feito byte a byte
  - Escrita muito mais lenta que leitura
- **Flash EEPROM** (tecnologia semelhante à EEPROM)
  - A escrita pressupõe a inicialização (*reset*) prévia das zonas de memória a escrever
  - O *reset* é feito por blocos (por exemplo, blocos de 4 kB) o que torna esta tecnologia mais rápida que a EEPROM
  - O *reset* e a escrita podem ser efetuados no próprio circuito em que a memória está integrada
  - Escrita muito mais lenta que a leitura

# Microcontrolador PIC32 da Microchip

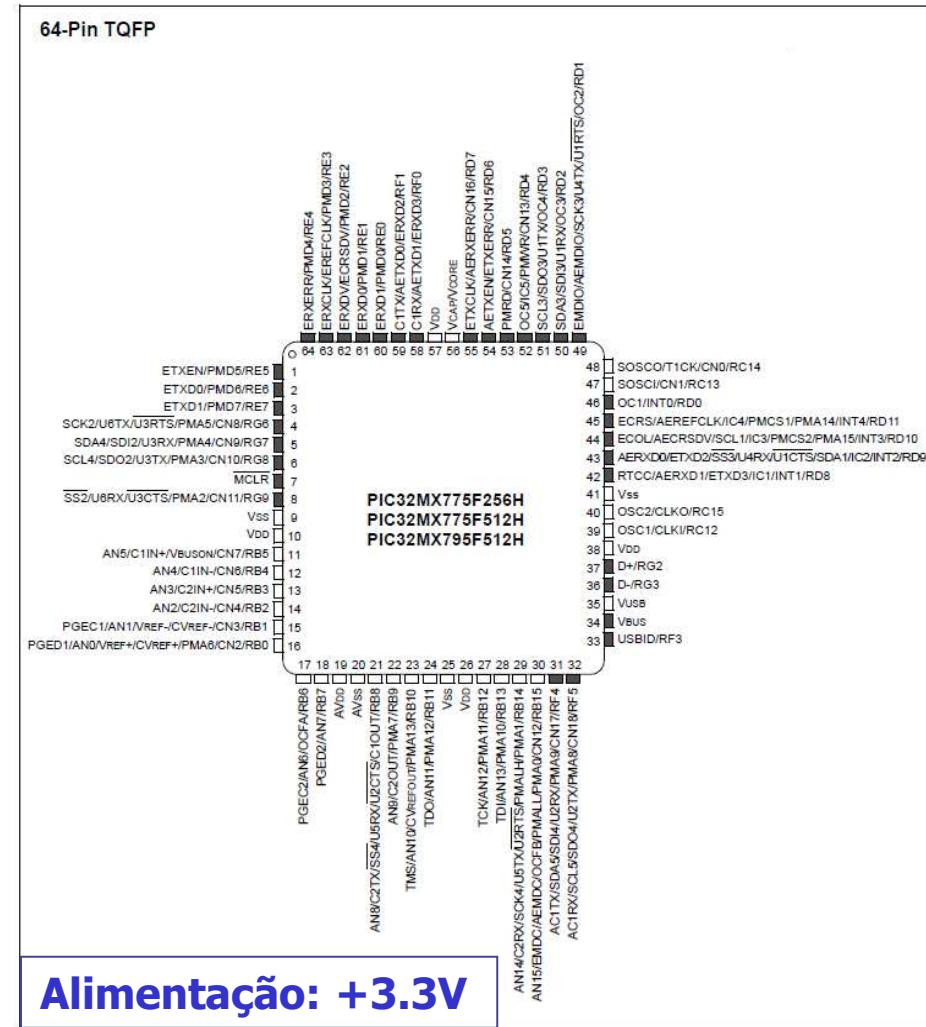
- Microcontrolador **PIC32MX795F512H**:
  - CPU MIPS
  - Conjunto alargado de periféricos
  - Memória flash: 512 kB (+12 kB Boot flash)
  - Memória RAM: 128 kB
  - Versão de 64 pinos (também disponível em 100 e 121 pinos)
- **CPU**:
  - MIPS32 M4K (core 32-bits com 5 estágios de *pipeline*)
    - Com coprocessador 0 (exceções e interrupções, gestão de memória)
    - **Não** dispõe de *Floating Point Unit* (coprocessador 1)
  - 32 registos de 32 bits (\$0 a \$31)
  - Espaço de endereçamento de 32 bits
  - Organização de memória: *byte-addressable*
  - Max. frequência de relógio: 80 MHz
- Documentação completa em (link válido em 23/01/2024):



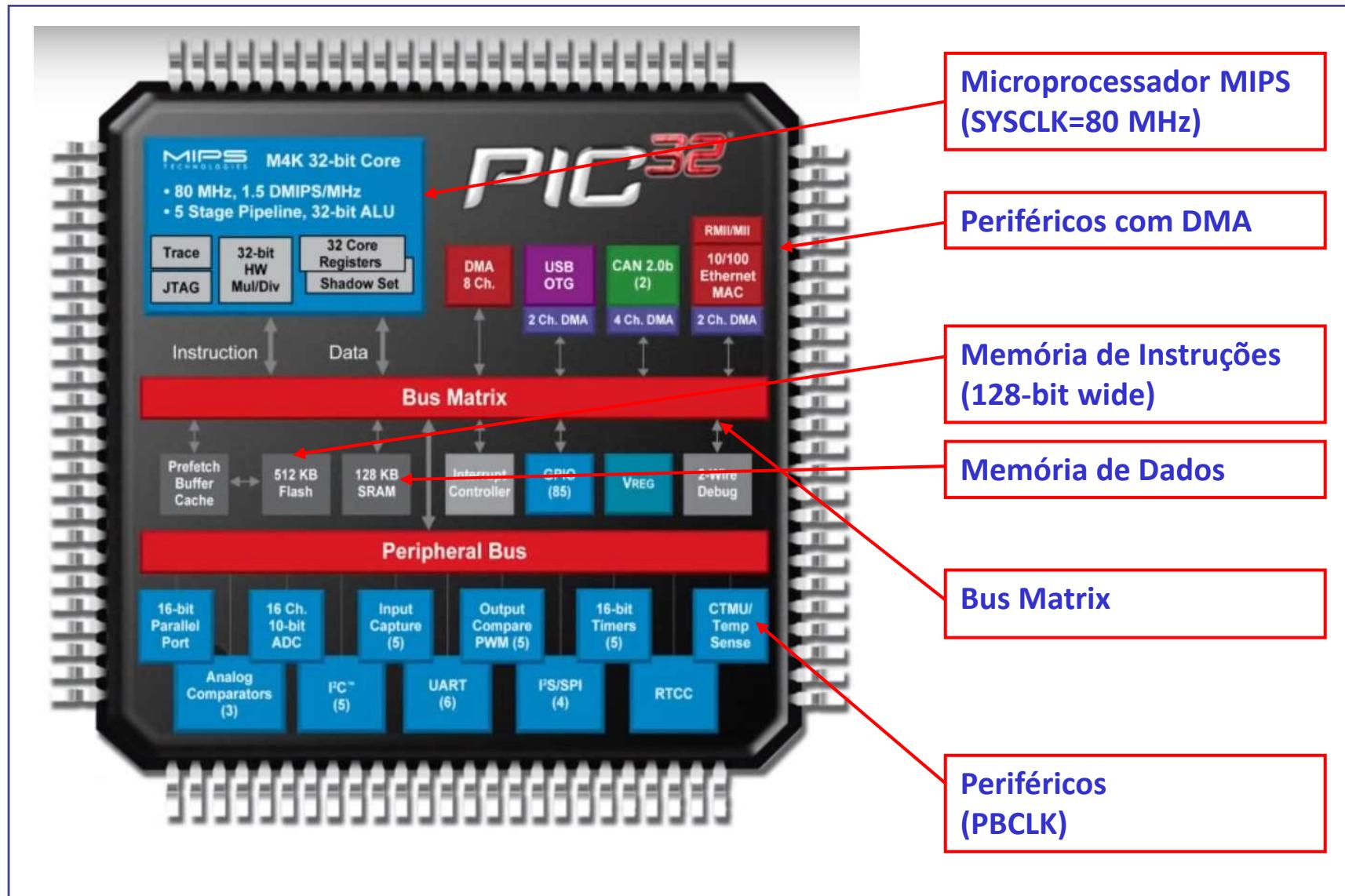
<https://www.microchip.com/en-us/product/PIC32MX795F512H>

# Microcontrolador PIC32MX795F512H

- Elevado nível de multiplexagem nos pinos do circuito integrado (cada pino pode ter, na versão de 64 pinos, até 9 funções distintas)
- Função desempenhada por cada pino depende de programação



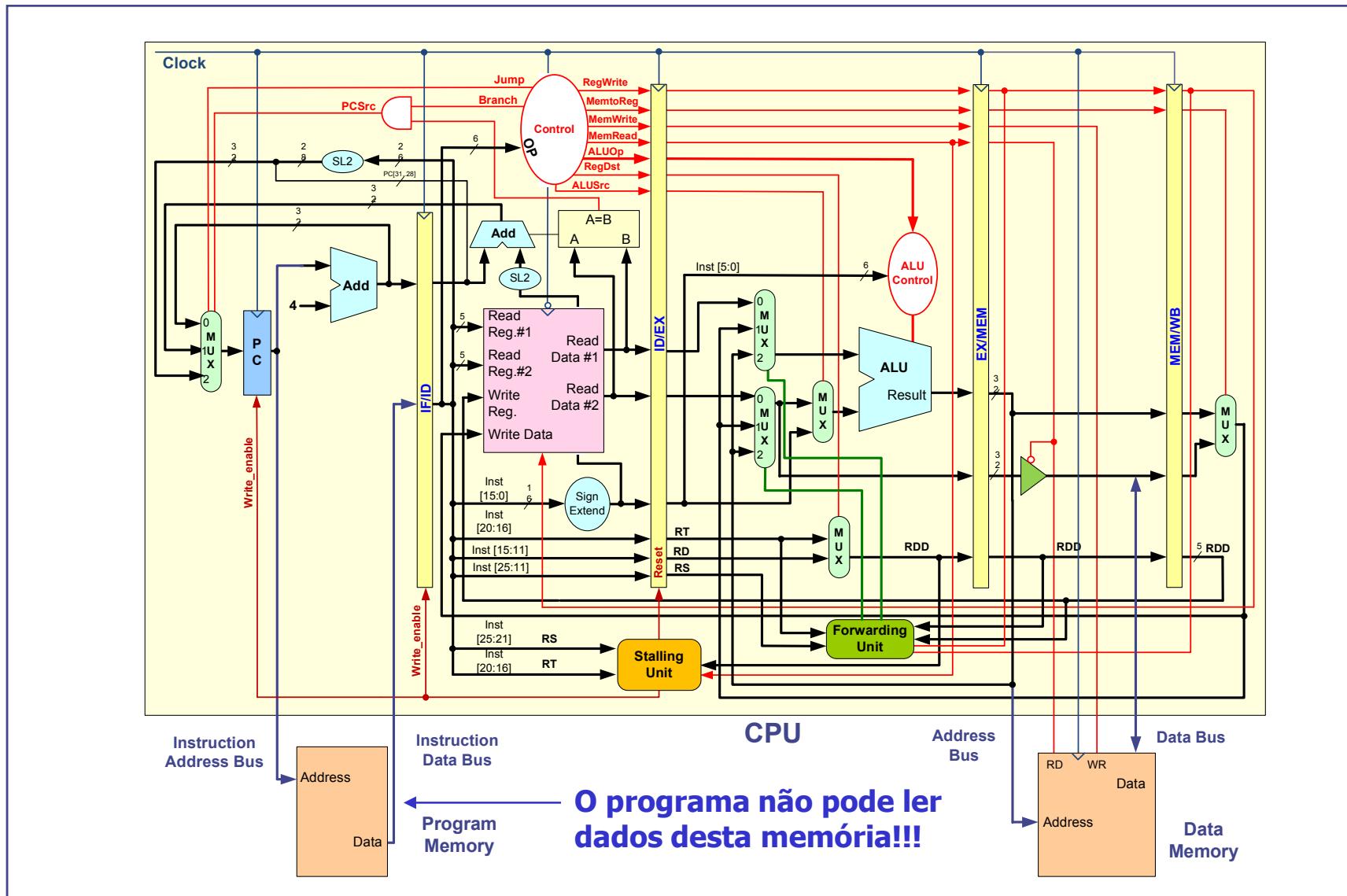
# Microcontrolador PIC32MX795F512H



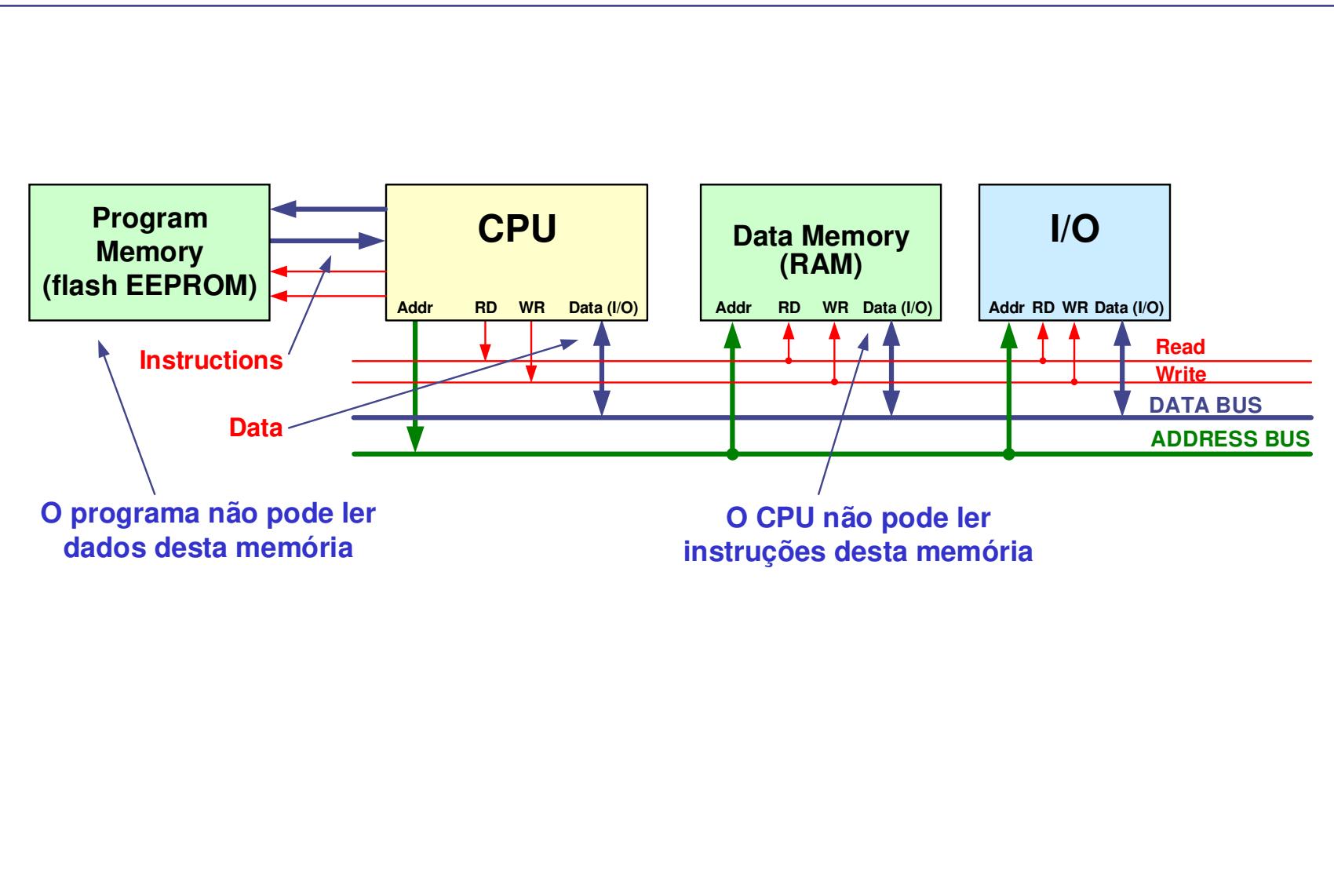
# Microcontrolador PIC32MX795F512H

- O MIPS do PIC32 é baseado numa **arquitetura de Harvard** (memória de instruções e dados separadas). Esta opção evita o *hazard* estrutural na implementação *pipelined* que aconteceria com uma única memória.
- Numa arquitetura de Harvard:
  - Existem dois espaços de endereçamento independentes: um para o programa e outro para dados.
  - Apenas o bloco encarregue da leitura das instruções da memória (*instruction fetch*) tem acesso à memória de programa.
  - O programa não pode ler dados da memória de instruções.
  - O CPU não pode ler instruções da memória de dados
  - É difícil o tratamento das constantes (por exemplo *strings*) uma vez que estas não podem ser armazenadas juntamente com o programa na memória de instruções (tipicamente uma memória não volátil)

# Versão simplificada de uma arquitetura MIPS *pipelined*

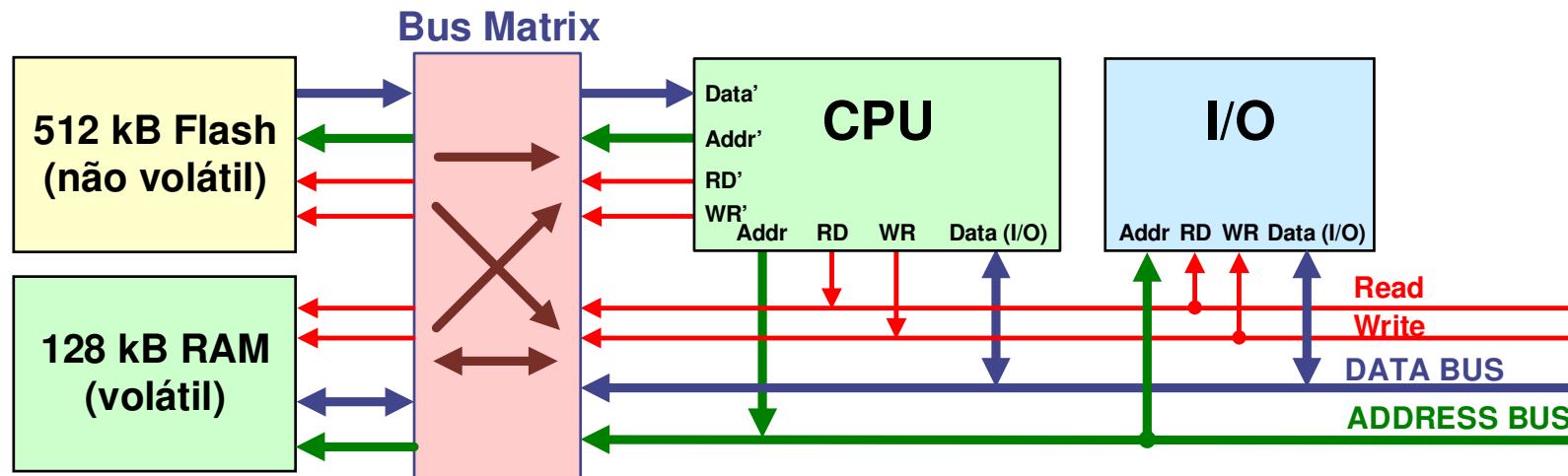


# Arquitetura de Harvard convencional



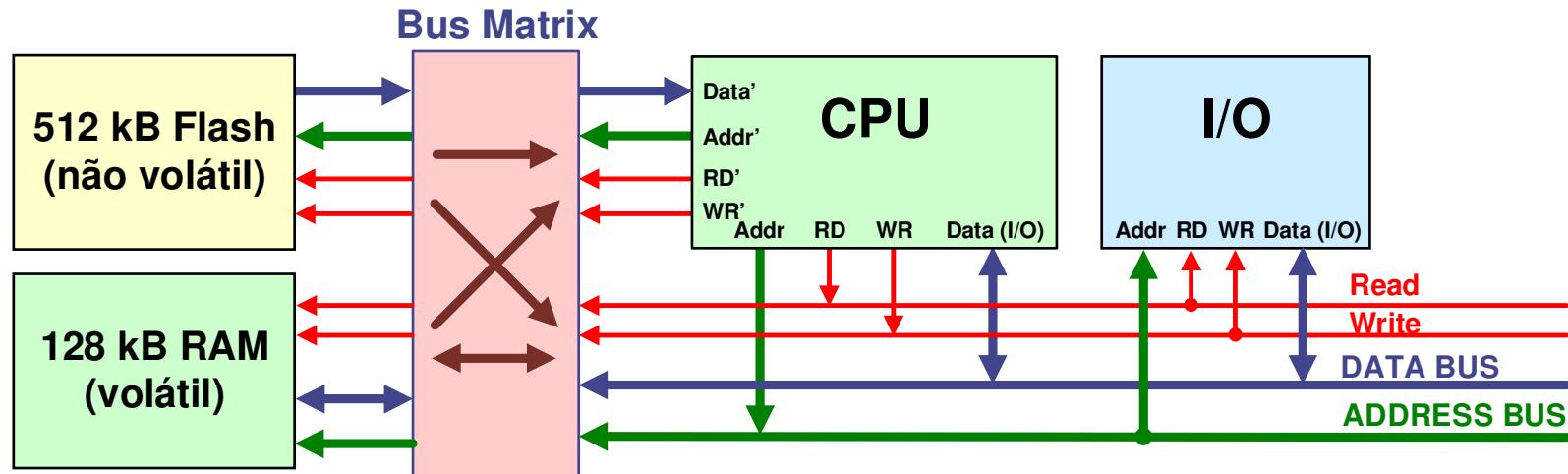
# Microcontrolador PIC32MX795F512H

- Solução implementada no PIC32 que resolve o problema dos dados constantes da arquitetura de Harvard: **Bus Matrix**



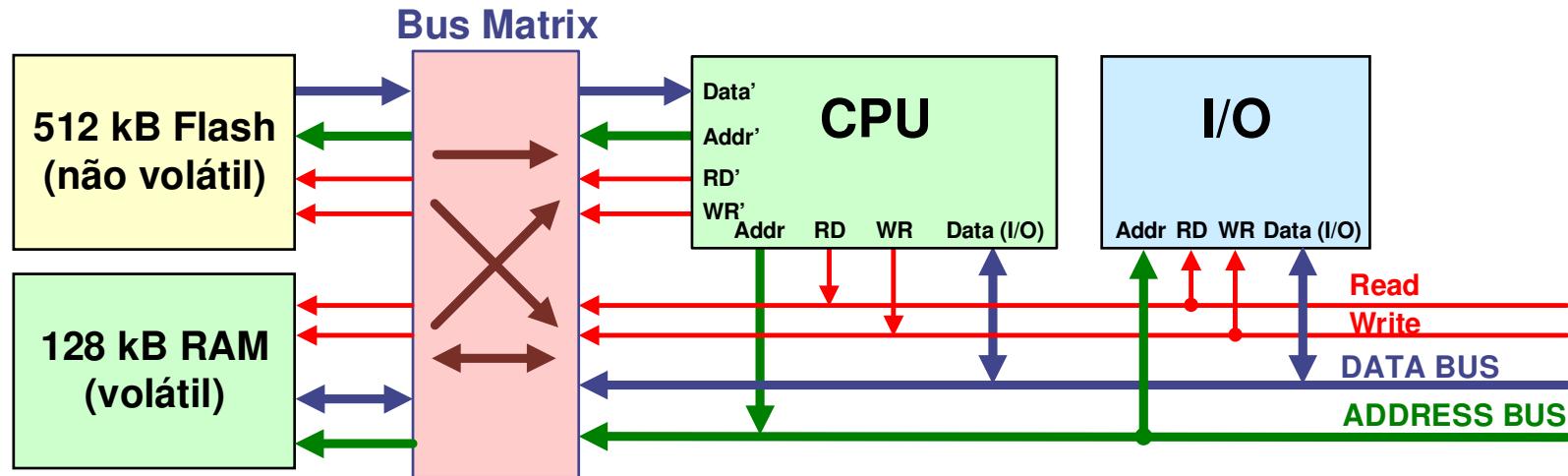
- O *Bus Matrix* é um comutador (*switch*) de alta velocidade que funciona à mesma frequência do CPU (SYSCLK)
- Estabelece ligações ponto a ponto entre os módulos do microcontrolador, em particular, entre o CPU e a memória RAM ou entre o CPU e a memória *Flash*
- O *peripheral bus* também liga ao *Bus Matrix* e pode ser configurado para trabalhar a uma frequência igual ou inferior à do CPU (PBCLK)

# Microcontrolador PIC32MX795F512H



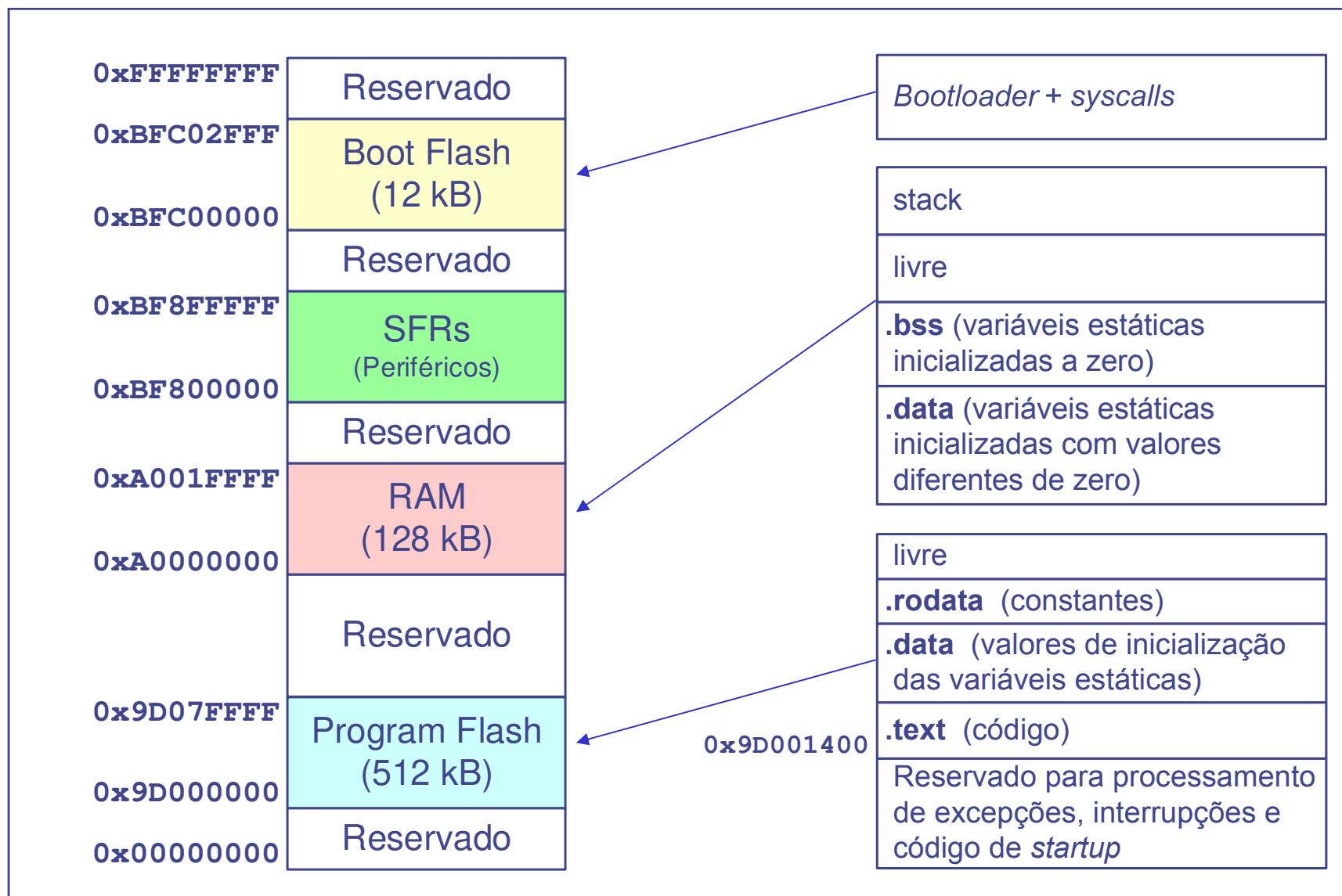
- Com o *Bus Matrix*, o espaço de endereçamento aparece, na visão do programador, como um espaço linear unificado (instruções e dados residem no mesmo espaço de endereçamento, cada um deles ocupando uma gama de endereços única)
- O CPU pode então executar programas que residem quer na *Flash* quer na RAM
- O programa gerado pelo *host* (instruções + dados constantes) pode ser armazenado na totalidade na memória *Flash* – programa pode aceder a qualquer momento à *Flash* para ler dados (por exemplo *strings*)

# Microcontrolador PIC32MX795F512H



- Para o programador, o PIC32 comporta-se como uma arquitetura de *von Neumann*: um único espaço de endereçamento onde residem dados e instruções
- Para que o programa tenha acesso a qualquer zona da memória Flash (para leitura) ou da memória RAM (para leitura ou escrita), basta definir adequadamente o respetivo endereço

# Mapa de memória do PIC32 (versão DETPIC32)



# Ferramentas de desenvolvimento DETPIC32

- Edição
  - GVim, gedit, geany...
- *Cross-compiler / cross-assembler*
  - **gcc** com *back-end* para MIPS (gcc-pic32)
  - Gera, entre outros, ficheiros ".hex" e ".map"
- Ferramentas desenvolvidas especificamente para DETPIC32
  - **bootloader** – programa previamente gravado na *boot flash* do PIC32; lê informação do porto de comunicação e escreve na memória *Flash*
  - **ldpic32** – programa para transferir ficheiro ".hex" para PIC32 (atua em conjunto com o *bootloader*)
  - **pterm** – programa terminal para comunicação com a placa DETPIC32, permitindo a interação com o utilizador
  - **hex2asm** – faz o *disassembly* do ficheiro ".hex" (utiliza o ficheiro ".map", para evidenciar secções / símbolos relevantes)

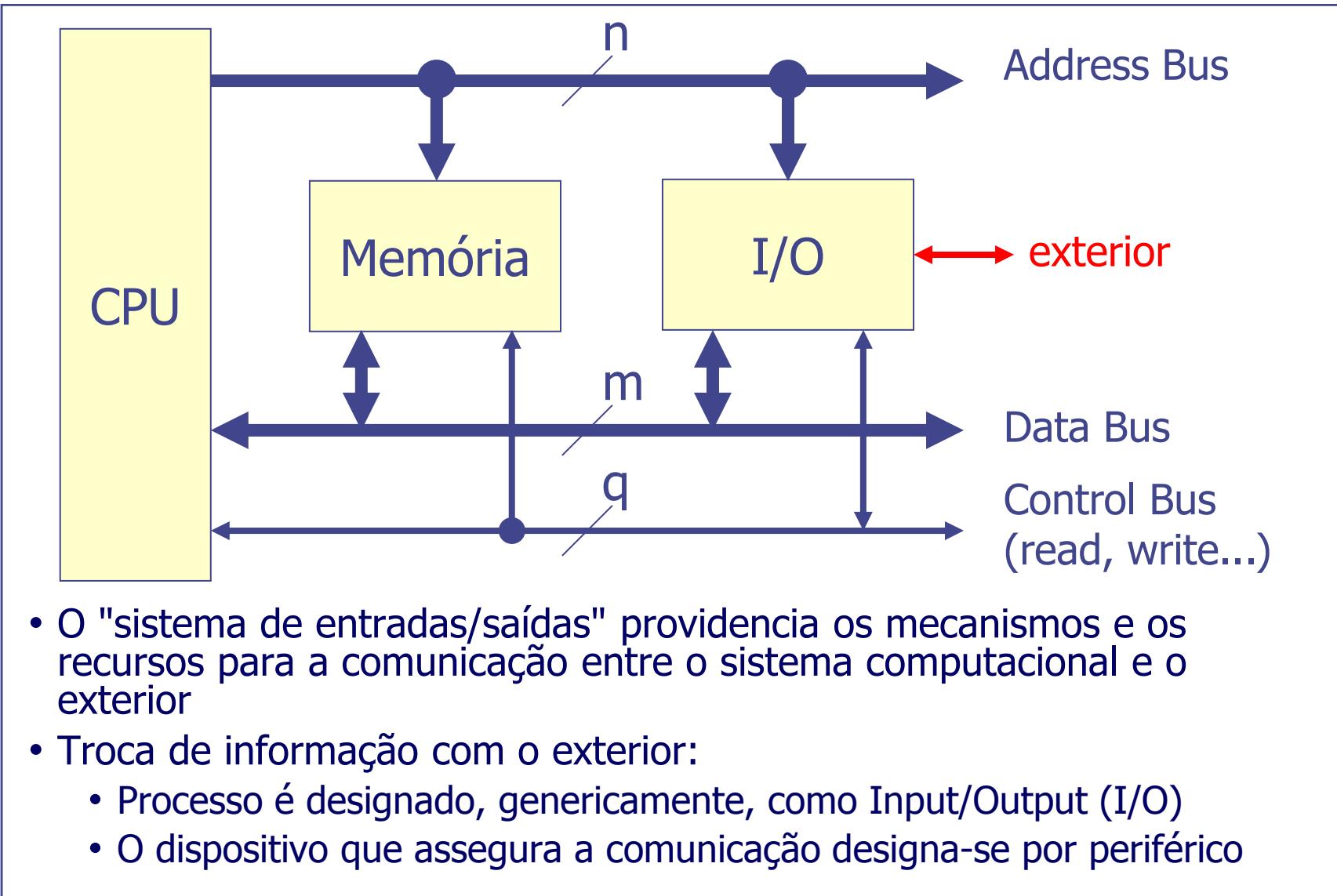


## Aulas 2 e 3

- Noção de periférico; estrutura básica de um módulo de I/O; modelo de programação
- Endereçamento das unidades de I/O
- Descodificação de endereços e geração de sinais de seleção de memória e unidades de I/O
- Mapeamento no espaço de endereçamento de memória
- Exemplo de um gerador de sinais de seleção programável

José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

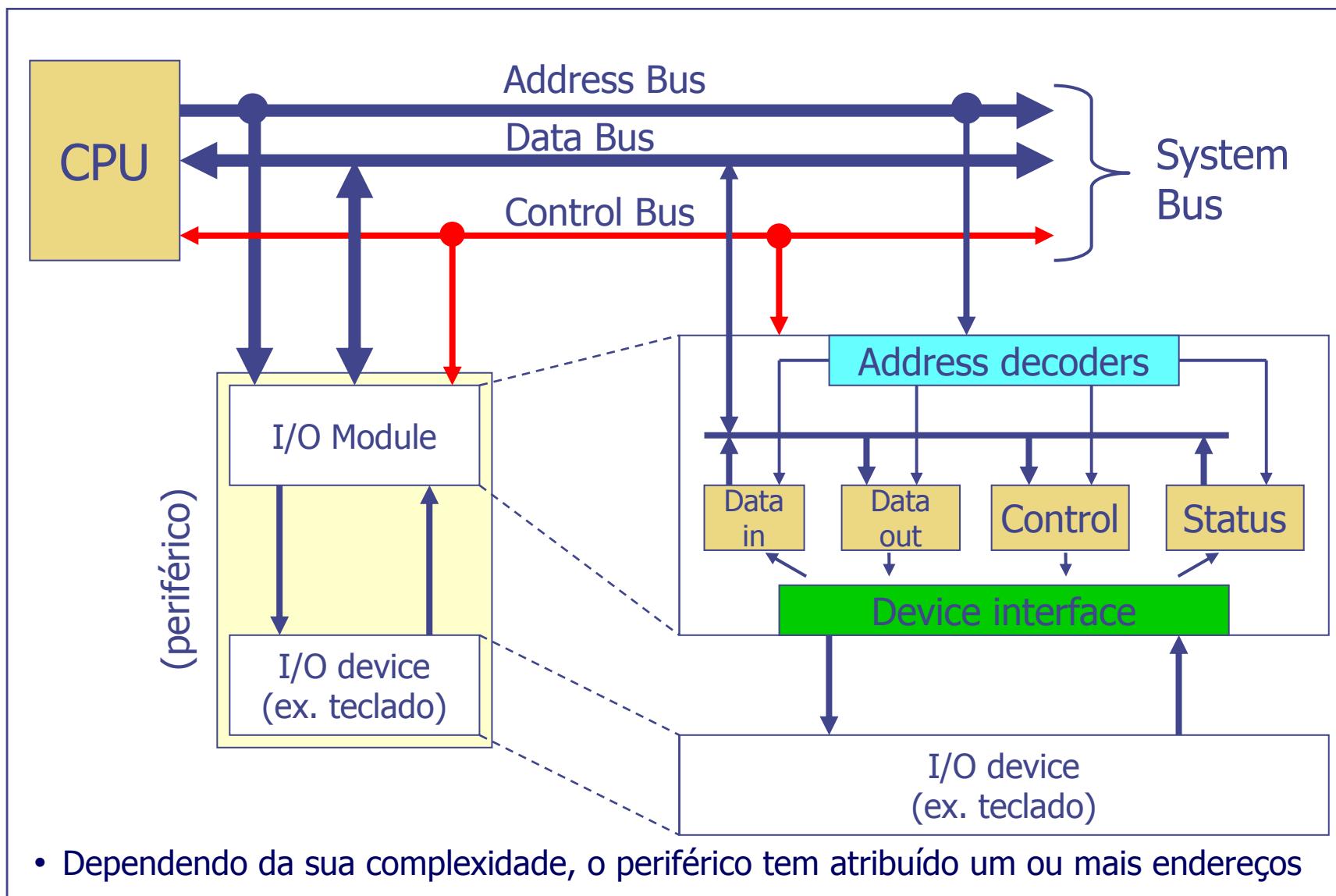
# Introdução



# Introdução

- Dispositivos periféricos:
  - grande variedade (por exemplo: teclado, rato, unidade de disco ...)
  - com métodos de operação diversos
  - assíncronos relativamente ao CPU
  - geram diferentes quantidades de informação com diferentes formatos a diferentes velocidades (de alguns bits/s a dezenas de Megabyte/s)
  - mais lentos que o CPU e a memória
- É necessária uma interface que providencie a adaptação entre as características intrínsecas do dispositivo periférico e as do CPU / memória
  - Módulo de I/O

# Módulo de I/O



# Módulo de I/O

- O módulo de I/O pode assim ser visto como um módulo de compatibilização entre as características e modo de funcionamento do sistema computacional e o dispositivo físico propriamente dito
- Ao nível do hardware:
  - Adequa as características do dispositivo físico de I/O às características do sistema digital ao qual tem que se ligar. O periférico liga-se ao sistema através dos barramentos, do mesmo modo que todos os outros dispositivos (ex. memória)
- Na interação com o dispositivo físico:
  - Lida com as particularidades do dispositivo, nomeadamente, formatação de dados, deteção e gestão de situações de erro, ...
- Ao nível do software:
  - Adequa o dispositivo físico à forma de organização do sistema computacional, disponibilizando e recebendo informação através de registos; esta solução esconde do programador a complexidade e os detalhes de implementação do dispositivo periférico

# Módulo de I/O

- O módulo de I/O permite ao processador ver um modelo simplificado do periférico, escondendo os detalhes de funcionamento interno
- Com a adoção do módulo de I/O, o dispositivo periférico, independentemente da sua natureza e função, passa a ser encarado pelo processador como uma coleção de registos de dados, de controlo e de *status*
- A comunicação entre o processador e o periférico é assegurada por operações de escrita e de leitura, em tudo semelhantes a um acesso a uma posição de memória
  - Ao contrário do que acontece na memória, o valor associado a estes endereços pode mudar sem intervenção do CPU
- O conjunto de registos e a descrição de cada um deles são específicos para cada periférico e constituem o que se designa por **modelo de programação do periférico**

# Módulo de I/O – modelo de programação

- **Data Register(s) (*Read/Write*)**
  - Registo(s) onde o processador coloca a informação a ser enviada para o periférico (*write*) e de onde lê informação proveniente do periférico (*read*)
- **Status Register(s) (*Read only*)**
  - Registo(s) que engloba(m) um conjunto de bits que dão informação sobre o estado do periférico (ex. operação terminada, informação disponível, situação de erro, ...)
- **Control Register(s) (*Write only* ou *Read/Write*)**
  - Registo(s) onde o CPU escreve informação sobre o modo de operação do periférico (comandos)
  - É comum um só registo incluir as funções de controlo e de *status*. Nesse caso, um conjunto de bits desse registo está associado a funções de controlo (*read/write* ou *write only bits*) e outro conjunto a funções de status (*read only bits*)

# Comunicação entre o CPU e outros dispositivos

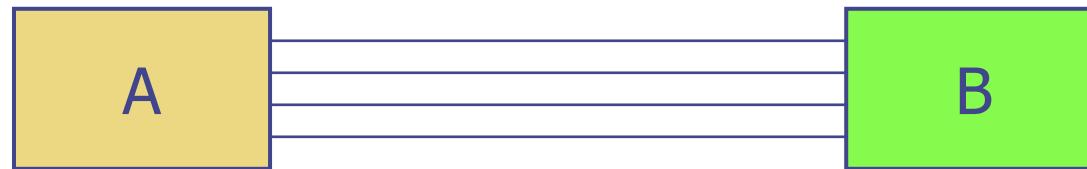
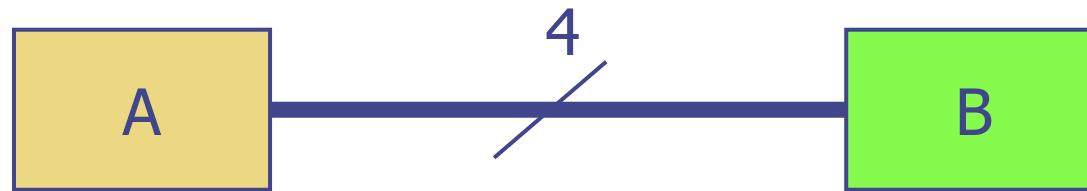
- A iniciativa da comunicação é sempre do CPU, no contexto da execução das instruções
- A comunicação entre o CPU e um dispositivo genérico envolve a existência de um **protocolo** que ambas as partes conhecem e respeitam
- Apenas duas operações podem ser efetuadas no sistema:
  - **Write** (fluxo de informação: CPU → dispositivo externo)
  - **Read** (fluxo de informação: CPU ← dispositivo externo)
- Uma operação de acesso do CPU a um dispositivo externo envolve:
  - Usar o barramento de endereços para especificar o **endereço do dispositivo a aceder**
  - Usar o barramento de controlo para sinalizar qual a operação a realizar (*read* ou *write*)
  - O barramento de dados assegura a transferência de dados, no sentido adequado, entre as duas entidades envolvidas na comunicação

# Seleção do dispositivo externo

- **Operação de escrita** (CPU → dispositivo externo)
  - apenas 1 dispositivo deve receber a informação colocada pelo CPU no barramento de dados
- **Operação de leitura** (CPU ← dispositivo externo)
  - apenas 1 dispositivo pode estar ativo no barramento de dados
  - os dispositivos, quando inativos, têm que estar eletricamente desligados do barramento de dados
  - é obrigatório utilizar portas **Tri-State** na ligação do dispositivo ao barramento de dados
- Num sistema computacional há múltiplos circuitos ligados ao barramento de dados
  - unidades de I/O
  - circuitos de memória
- Há, pois, necessidade de, a partir do endereço gerado pelo CPU, **selecionar apenas um** dos vários dispositivos existentes no sistema

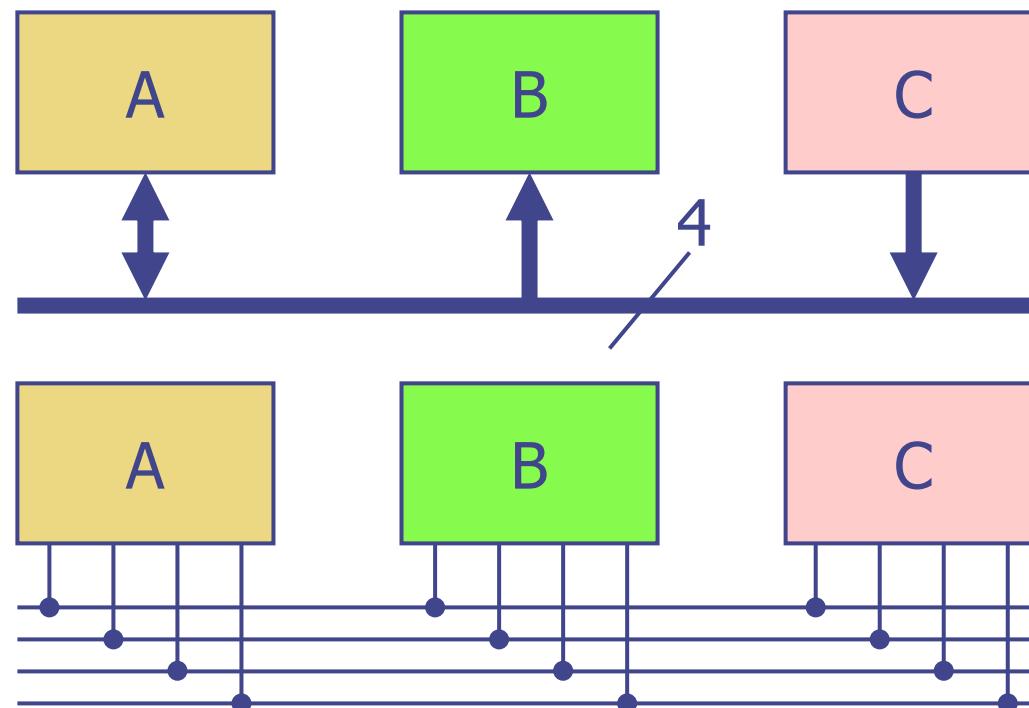
# Barramento simples (revisão)

- Barramento (bus) - conjunto de ligações (fios) agrupadas, geralmente, segundo uma dada função; cada ligação transporta informação relativa a 1 bit.
- Exemplo – barramento de 4 bits que liga os dispositivos A e B



# Barramento partilhado (revisão)

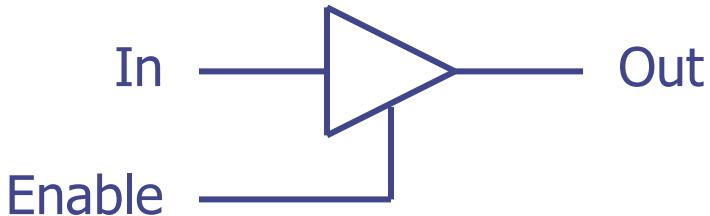
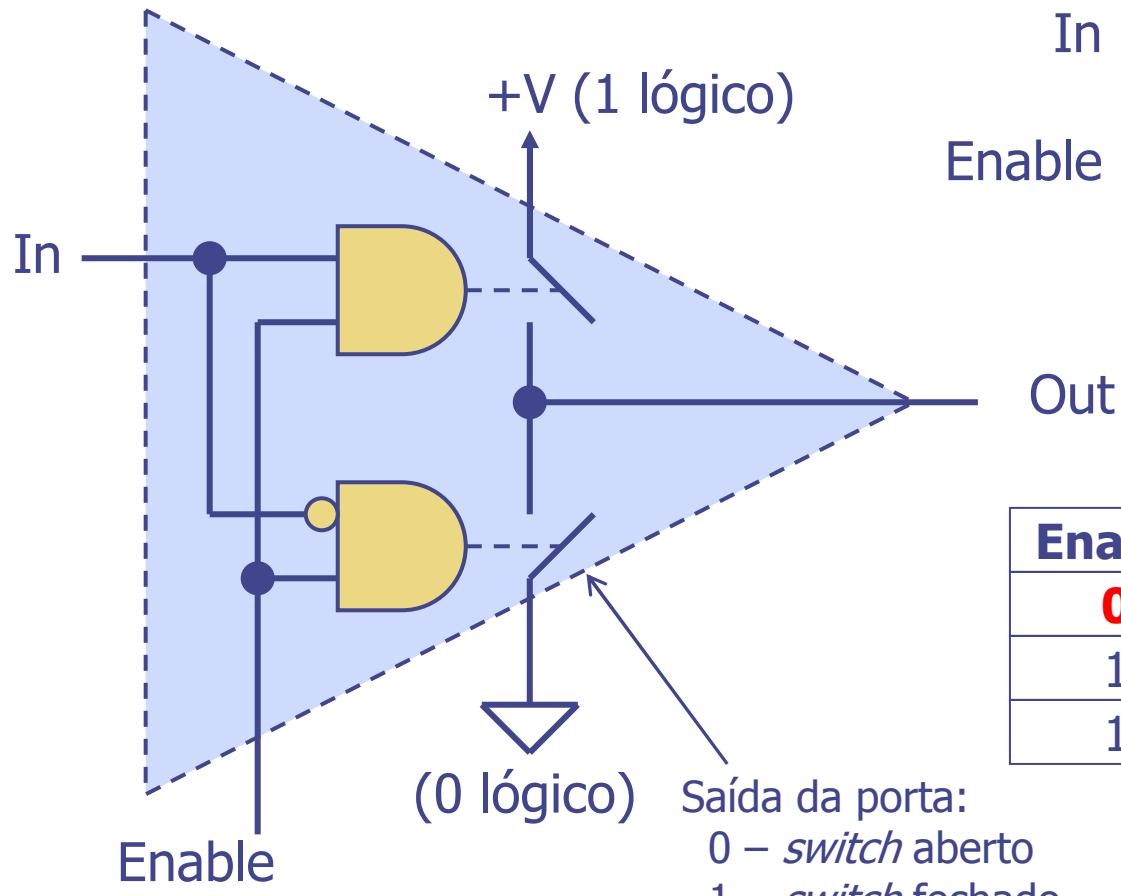
- Barramento partilhado (*shared bus*) - barramento que interliga vários dispositivos
- Exemplo: barramento de interligação entre os dispositivos A, B e C



- A comunicação pode realizar-se de A para B, de C para B ou de C para A

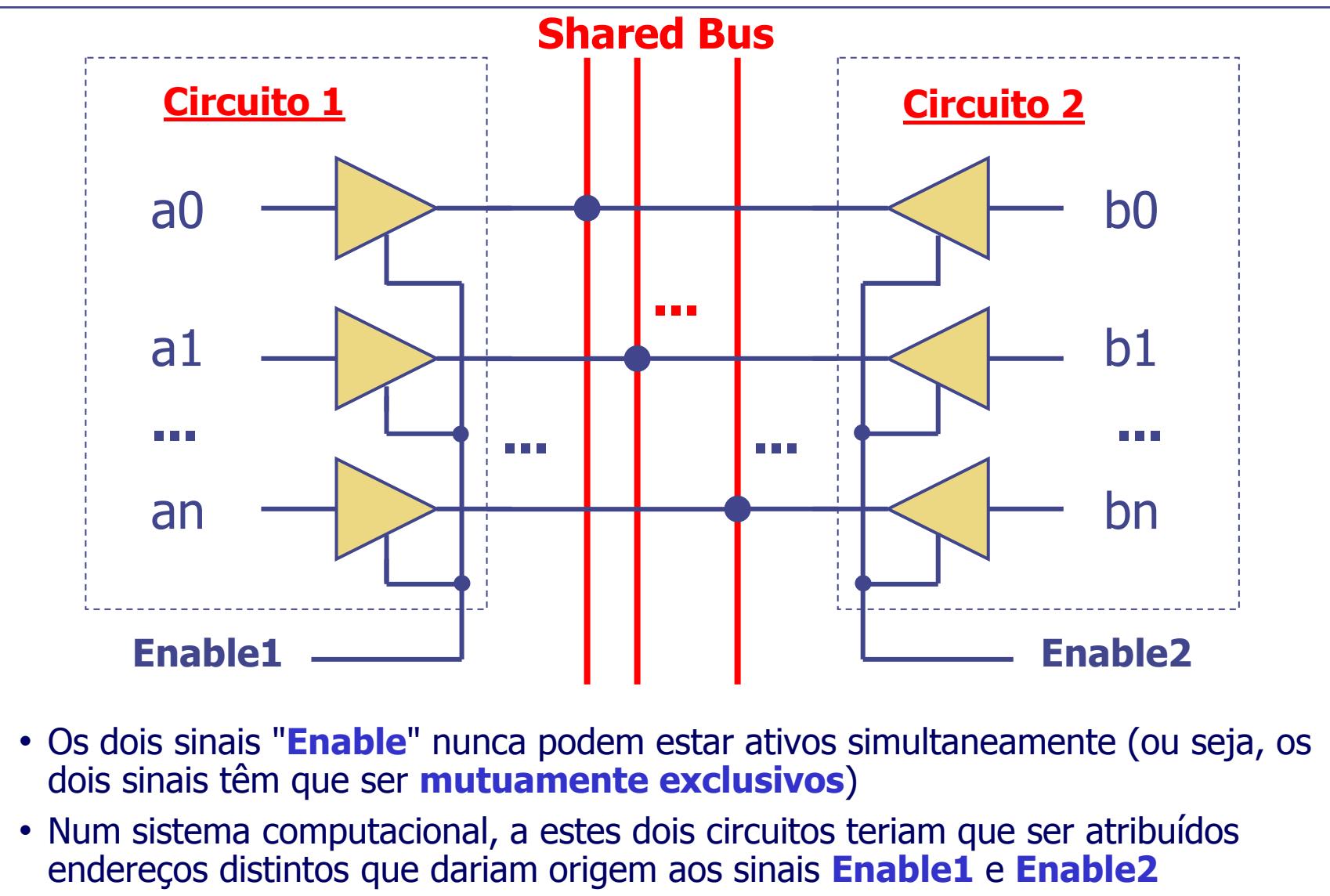
# Porta *Tri-State*

- Modelo de funcionamento de uma porta *Tri-State*

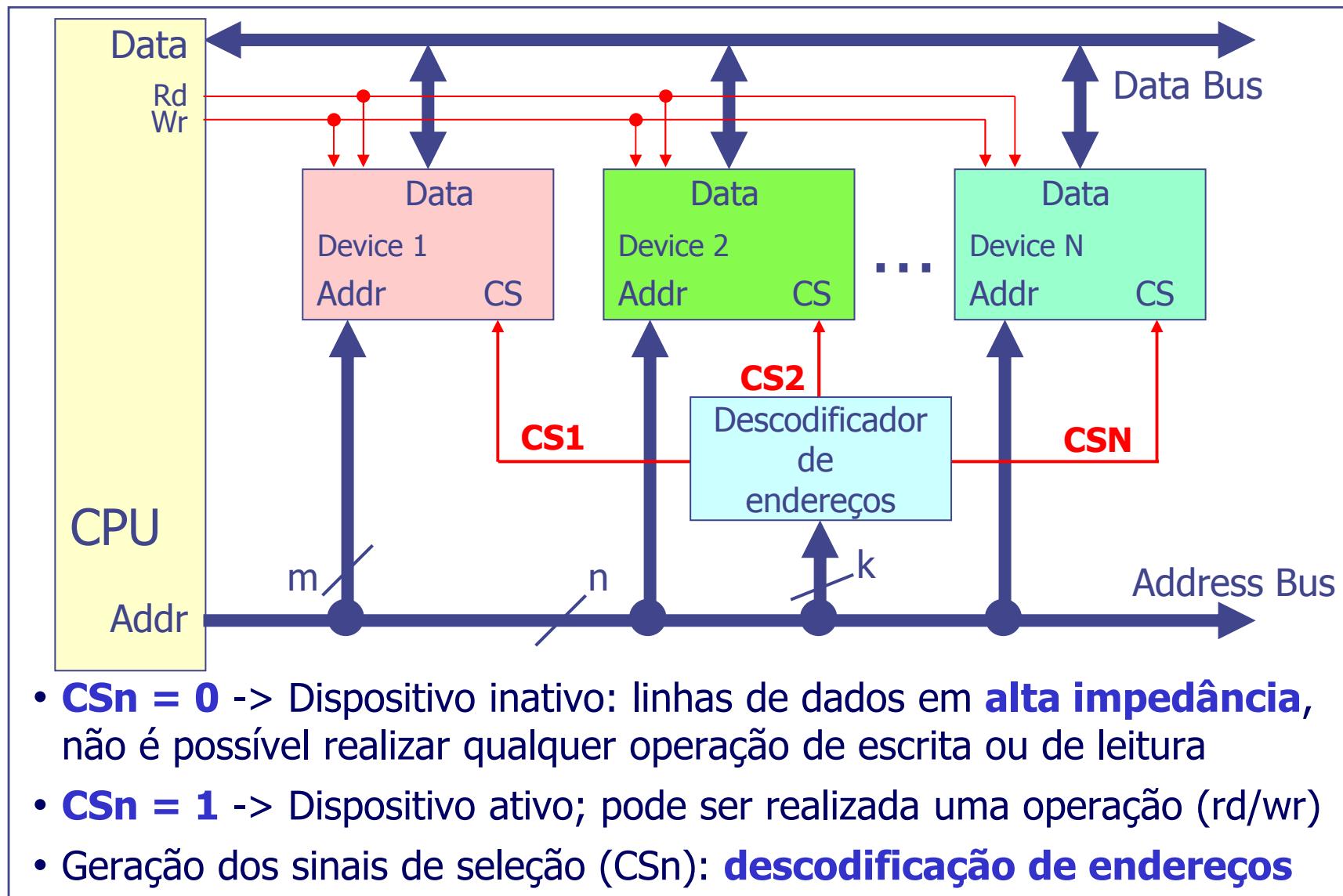


Enable	In	Out
0	X	Hi-Z
1	0	0
1	1	1

# Ligação a um barramento partilhado



# Seleção do dispositivo externo



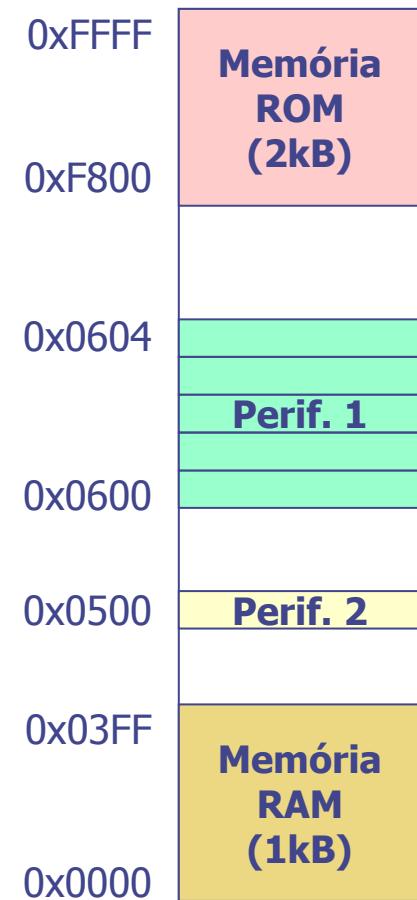
# Seleção do dispositivo externo

- Para ser possível o acesso a todos os recursos disponíveis, o dispositivo externo pode necessitar de apenas um endereço ou de uma gama contígua de endereços
- Exemplos (supondo uma organização de memória do tipo *byte-addressable*) :
  - Para aceder a todas as posições de uma memória de 1kB são necessários 1024 endereços consecutivos (10 bits do barramento de endereços)
  - Para ser possível o acesso aos 5 registos (de 1 byte cada) de um periférico são necessários 5 endereços consecutivos (3 bits do barramento de endereços)
  - Para aceder a um porto de saída de 1 byte (por exemplo implementado como um registo de 8 bits) será apenas necessário 1 endereço
- A implementação do descodificador de endereços é feita a partir de um mapa de endereços que mapeia no espaço de endereçamento do processador a gama de endereços necessária para cada dispositivo do sistema

# Mapeamento no espaço de endereçamento

- Exemplo de mapeamento de dispositivos, considerando um espaço de endereçamento de 16 bits ( $2^{16} = 64k$ ,  $A_{15}-A_0$ ), e uma organização do tipo *byte-addressable*:
  - memória RAM de  $1k \times 8$  (1 kB), memória ROM de  $2k \times 8$  (2 kB), periférico1 com 5 registos de 1 byte, periférico2 com 1 registo de 1 byte
  - Possível mapa de endereços:

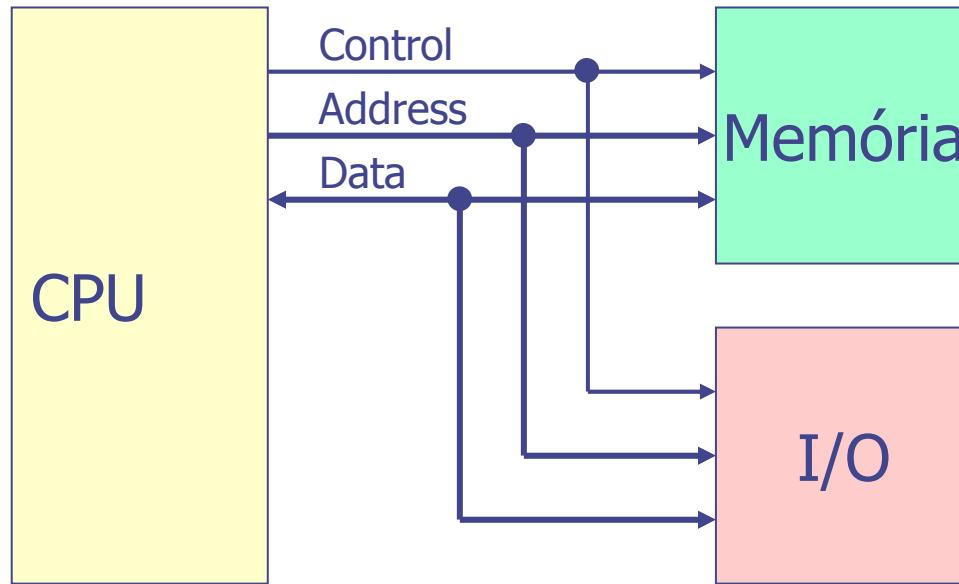
Dispositivo	Dimensão (bytes)	Endereço Inicial	Endereço Final	Nº bits do <i>address bus</i>
RAM, $1k \times 8$	1024	0x0000	0x03FF	10
ROM, $2k \times 8$	2048	0xF800	0xFFFF	11
Periférico 1	5	0x0600	0x0604	3
Periférico 2	1	0x0500	0x0500	0



Se os registos dos periféricos fossem de 32 bits, quais seriam as gamas de endereços necessárias ?

# Endereçamento das unidades de I/O

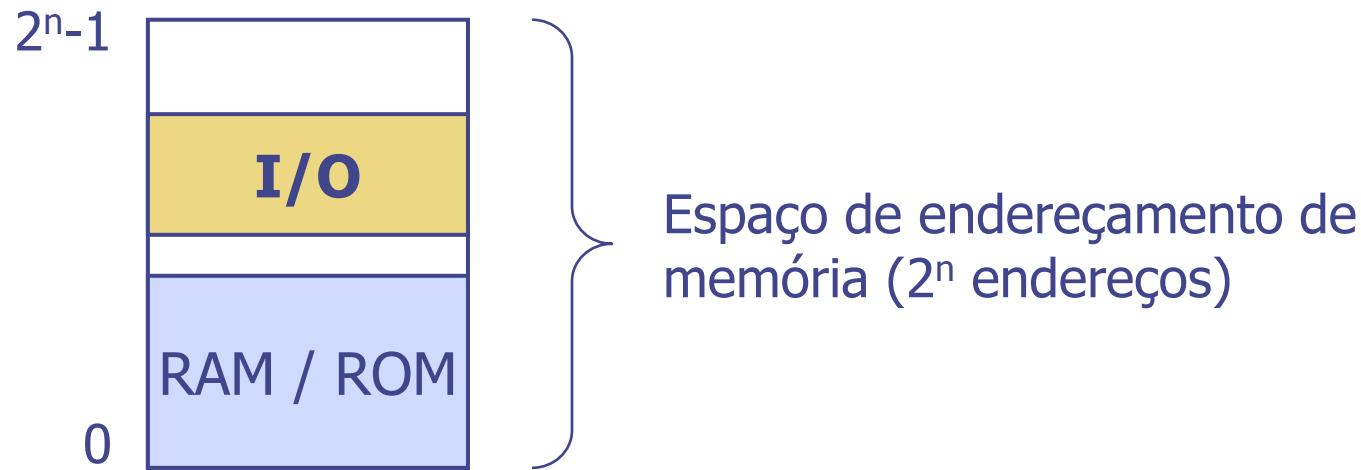
- *Memory-mapped I/O*



- Memória e unidades de I/O coabitam no mesmo espaço de endereçamento
- Uma parte do espaço de endereçamento é reservada para periféricos

# Endereçamento das unidades de I/O

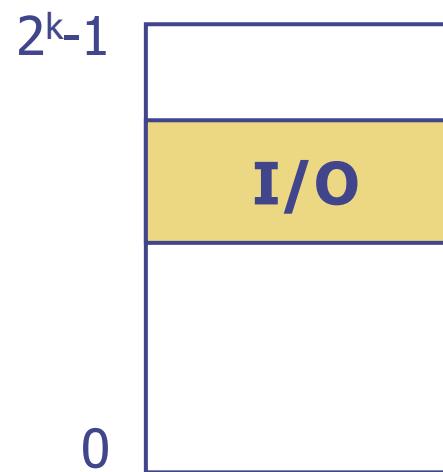
- *Memory-mapped I/O*



- As unidades de I/O são atribuídos endereços do espaço de endereçamento de memória
- O acesso às unidades de I/O é feito com as mesmas instruções com que se accede à memória (**lw** e **sw** no caso do MIPS)

# Endereçamento das unidades de I/O

- I/O Isolado

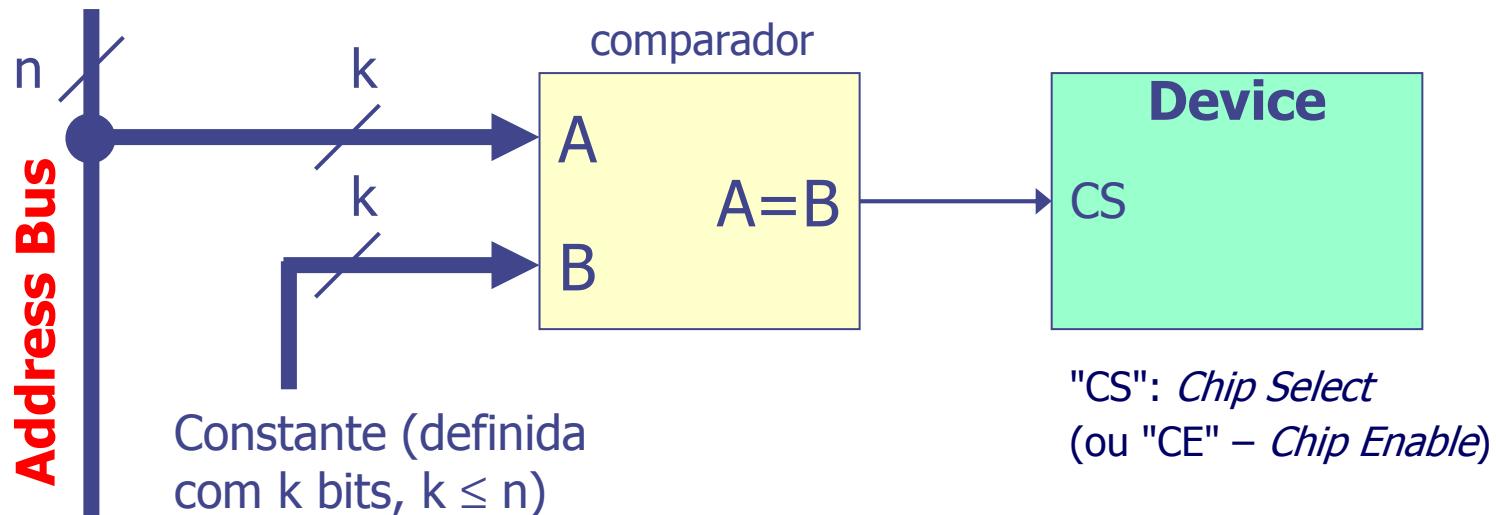


$$\mathbf{IO / M\ \backslash = 1}$$

Espaço de endereçamento de periféricos ( $2^k$  endereços)

- Memória e periféricos em espaços de endereçamento separados
- Sinal do barramento de controlo indica a qual dos espaços de endereçamento (I/O ou memória) se destina o acesso; por exemplo IO/M\:
  - $\text{IO/M\ \backslash = 1} \rightarrow$  acesso ao espaço de endereçamento de I/O
  - $\text{IO/M\ \backslash = 0} \rightarrow$  acesso ao espaço de endereçamento de memória
- O acesso às unidades de I/O é feito com instruções específicas

# Descodificação de endereços



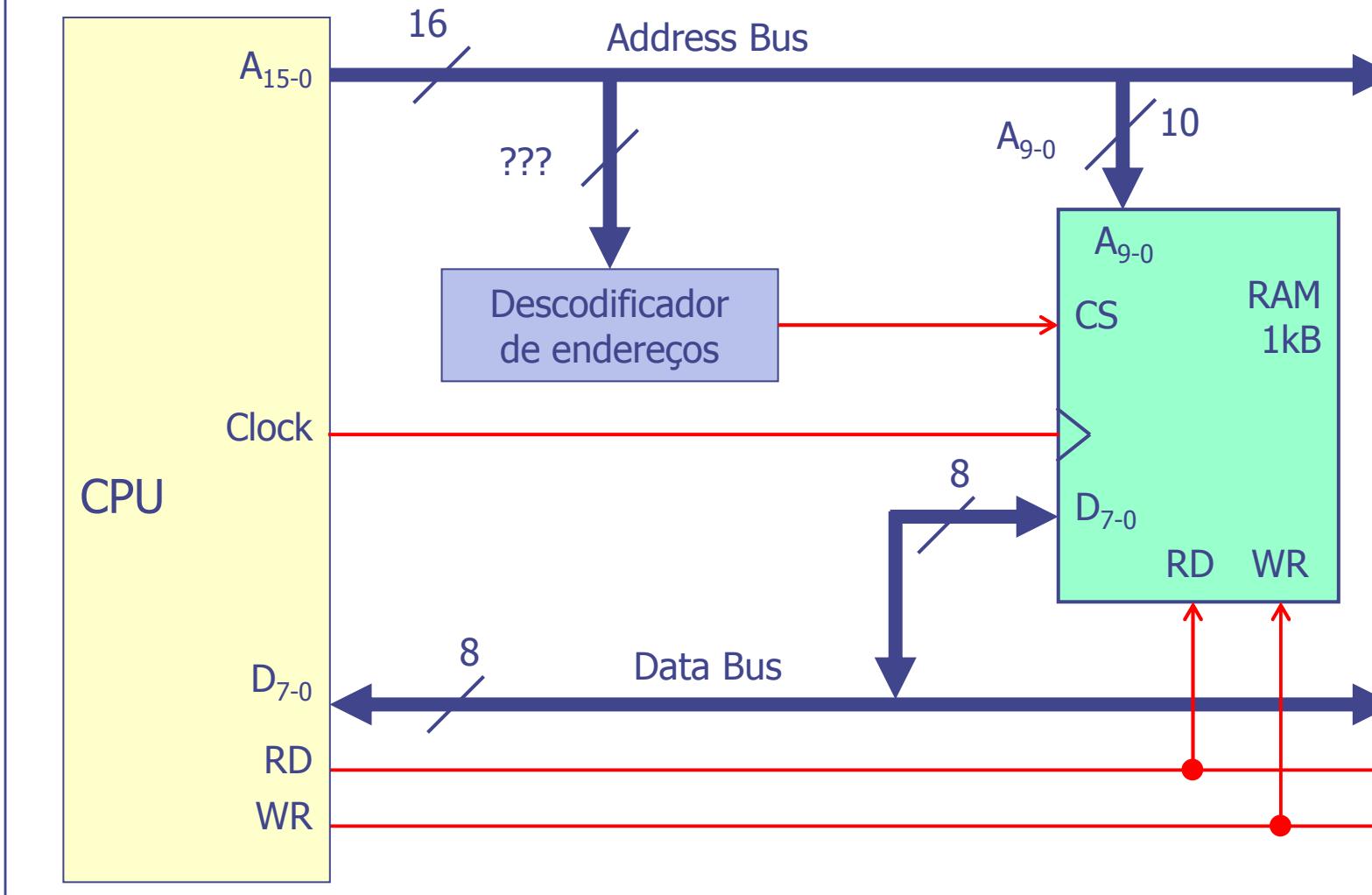
- O dispositivo é selecionado quando a combinação binária presente nos " $k$ " bits do *address bus* for igual à constante (entrada B)
- Exemplo com  $n=16$  e  $k=4$ 
  - Entrada A: 4 bits mais significativos do barramento de endereços
  - Entrada B:  $1000_2$
  - Sinal de seleção ativo na gama: [0x8000, 0xFFFF]

# Descodificação de endereços

- Supondo um CPU com um espaço de endereçamento de 16 bits, memória *byte-addressable*, e um barramento de dados de 8 bits
  - 16 bits ( $A_{15}-A_0$ )  $\rightarrow (2^{16}=64\text{ k})$
  - 8 bits ( $D_7-D_0$ )
- **Exemplo 1:** ligação de uma memória RAM de 1 kByte ao CPU
  - 1 kByte ( $1\text{k} \times 8$ ) - 10 bits de endereço ( $2^{10}=1\text{k}$ )
- **Exemplo 2:** ligação de um porto de saída de 1 byte ao CPU

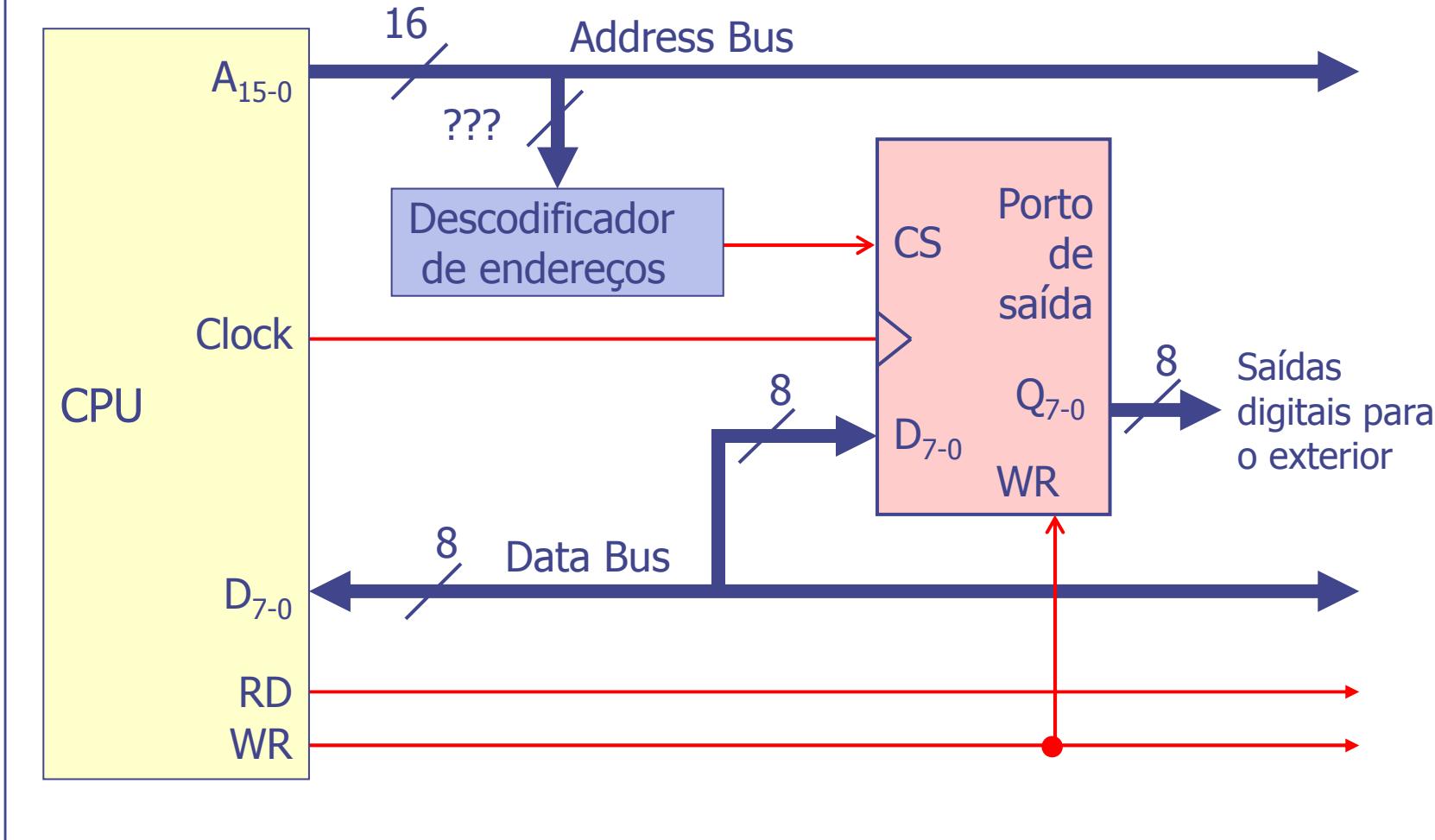
# Descodificação de endereços

- Exemplo 1: ligação de uma memória RAM de 1 kByte ao CPU



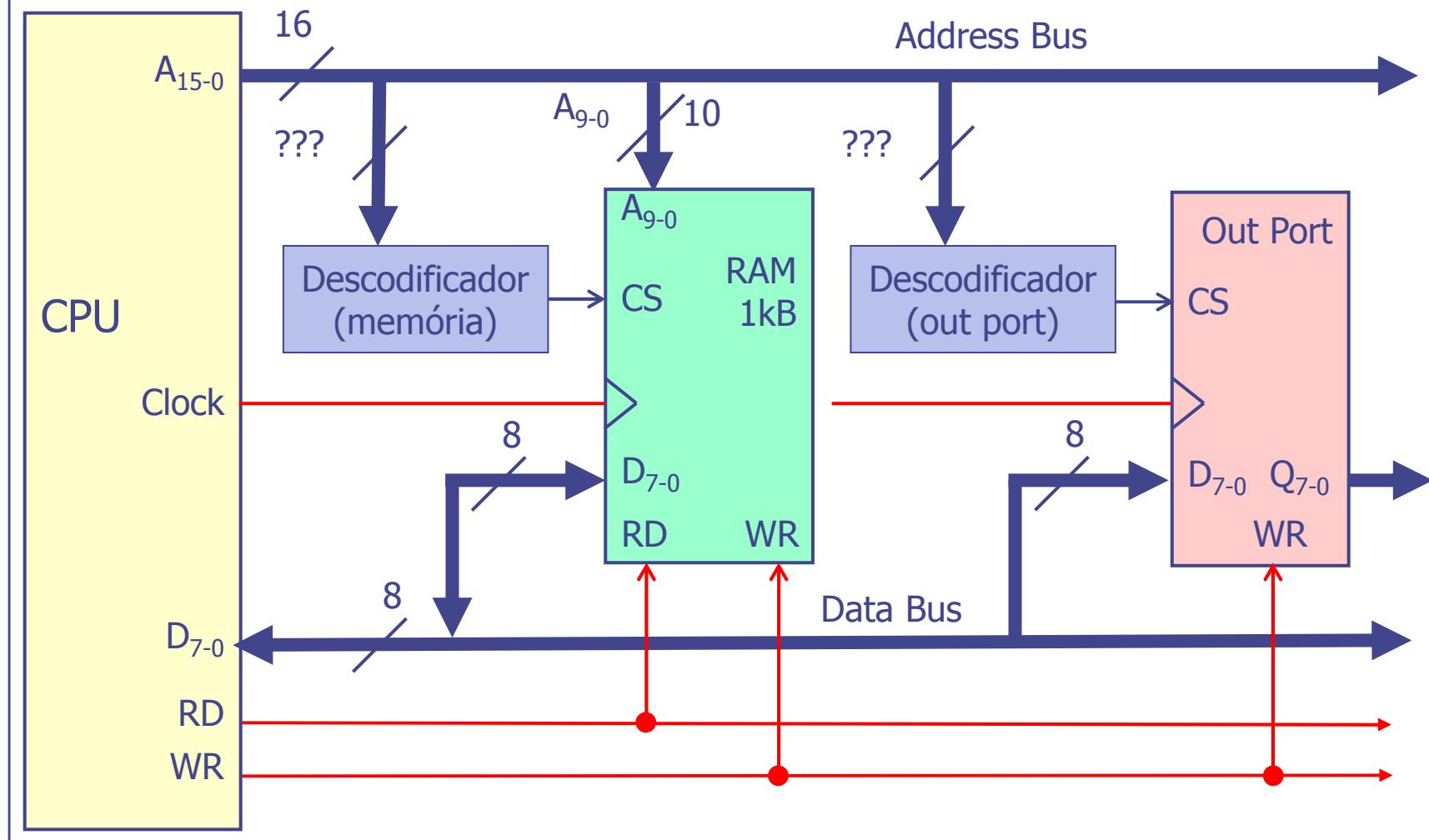
# Descodificação de endereços

- Exemplo 2: ligação de um porto de saída de 1 byte ao CPU



# Descodificação de endereços

- Ligaçāo do porto de saída e da memória



# Descodificação de endereços

- Descodificação total
  - Para uma dada posição de memória / registo de periférico existe apenas um endereço possível para acesso
  - Todos os bits relevantes são descodificados
- Descodificação parcial
  - Vários endereços possíveis para aceder à **mesma posição de memória/registo de um periférico**
  - Apenas alguns bits são descodificados
  - Conduz a circuitos de descodificação mais simples (e menores atrasos)

# Descodificação de endereços

- Mapa de endereços, num espaço de endereçamento de 16 bits (para o exemplo dos slides anteriores):

Dispositivo	Dimensão (bytes)	Endereço Inicial	Endereço Final	Nº bits do <i>address bus</i>
RAM, 1k x 8	1024	0x0000	0x03FF	10
Porto de saída	1	0x4100	0x4100	0

- Descodificador de endereços do porto de saída
  - Quais os bits a usar no descodificador de endereços?
- Descodificador de endereços da memória RAM
  - Quais os bits a usar no descodificador de endereços?

# Descodificação de endereços

Dispositivo	Dimensão (bytes)	Endereço Inicial	Endereço Final	Nº bits do <i>address bus</i>
RAM, 1k x 8	1024	0x0000	0x03FF	10
Porto de saída	1	0x4100	0x4100	0

- Porto de saída – **descodificação total:**

$0x4100 = 0100\ 0001\ 0000\ 0000$

$$An \setminus \Leftrightarrow \overline{An}$$

$CS = A15 \setminus . A14 . A13 \setminus . A12 \setminus . A11 \setminus . A10 \setminus . A9 \setminus . A8 .$   
 $A7 \setminus . A6 \setminus . A5 \setminus . A4 \setminus . A3 \setminus . A2 \setminus . A1 \setminus . A0 \setminus$

- Porto de saída – **descodificação parcial:**

- Não usar, por exemplo, os dois bits menos significativos

$CS = A15 \setminus . A14 . A13 \setminus . A12 \setminus . A11 \setminus . A10 \setminus . A9 \setminus . A8 .$   
 $A7 \setminus . A6 \setminus . A5 \setminus . A4 \setminus . A3 \setminus . A2 \setminus$

- Gama de ativação do CS: [0x4100, 0x4103]

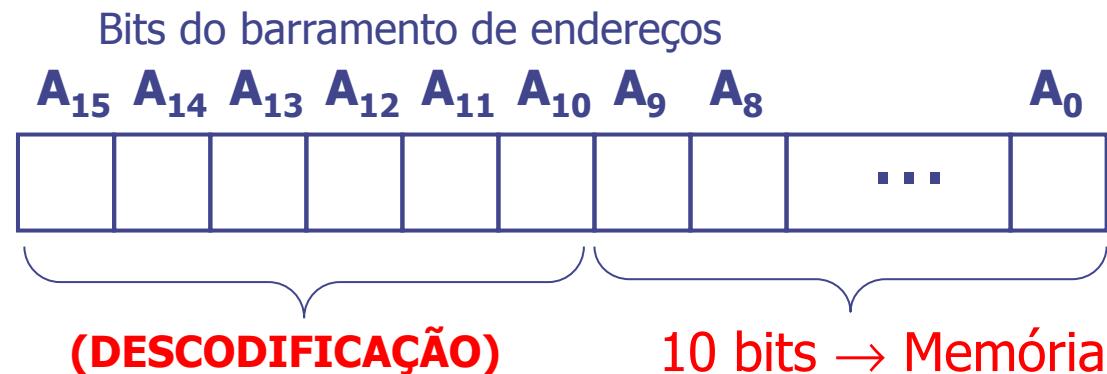
# Descodificação de endereços

Dispositivo	Dimensão (bytes)	Endereço Inicial	Endereço Final	Nº bits do <i>address bus</i>
RAM, 1k x 8	1024	0x0000	0x03FF	10
Porto de saída	1	0x4100	0x4100	0

- Memória RAM



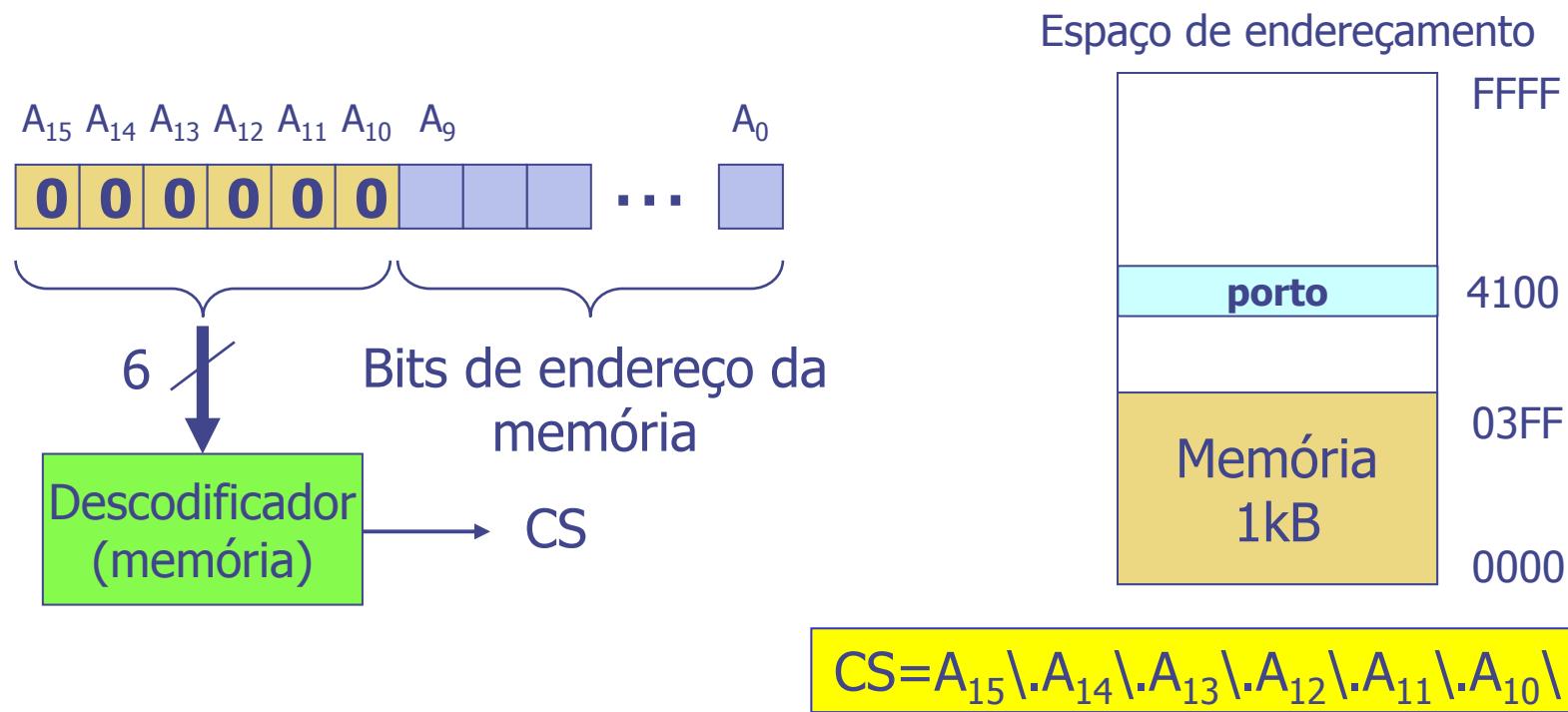
# Descodificação de endereços



- Para garantir que a memória de 1kB está mapeada a partir do endereço 0x0000 (na gama 0x0000-0x03FF), há várias soluções possíveis; vamos analisar as seguintes 3:
  - 1) **descodificação total** – usar os 6 bits A15 a A10 (e.g. **000000**)
  - 2) **descodificação parcial** – usar A15, A14, A13 e A12 e ignorar A11 e A10 (e.g. **0000xx**)
  - 3) **descodificação parcial** – usar apenas A13, A12, A11 e A10 e ignorar A15 e A14 (e.g. **xx0000**)
- Que implicações têm estas escolhas? Quais garantem zonas de endereçamento exclusivas para o porto de saída e para a memória?

# Descodificação de endereços – descodificação total

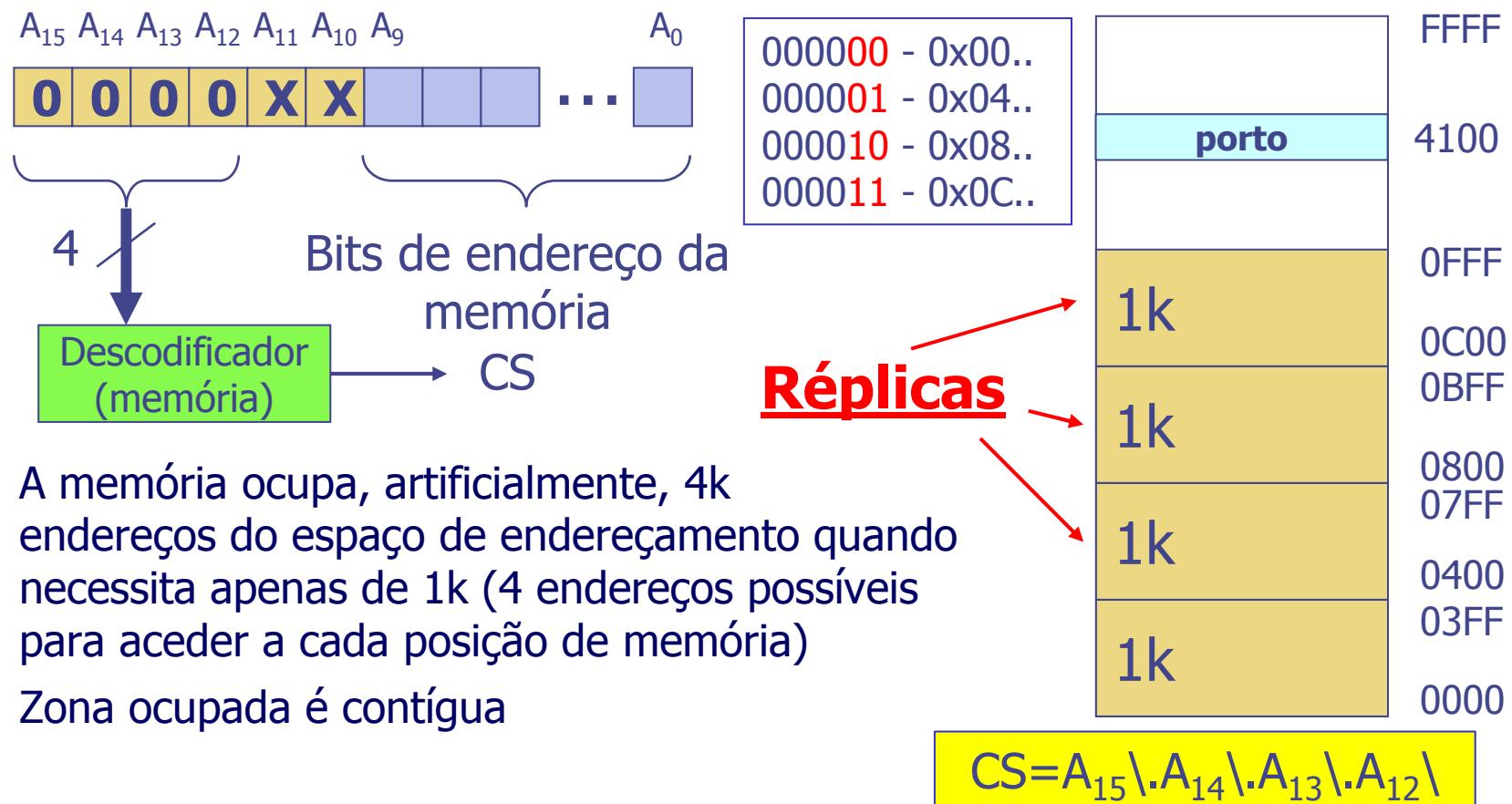
- Solução 1 – utilizar todos os bits possíveis, e.g. **000000**. Isto significa que um endereço só é válido para aceder à memória se tiver os 6 bits mais significativos a 0



- A memória ocupa 1k do espaço de endereçamento
- Apenas 1 endereço possível para aceder a cada posição de memória

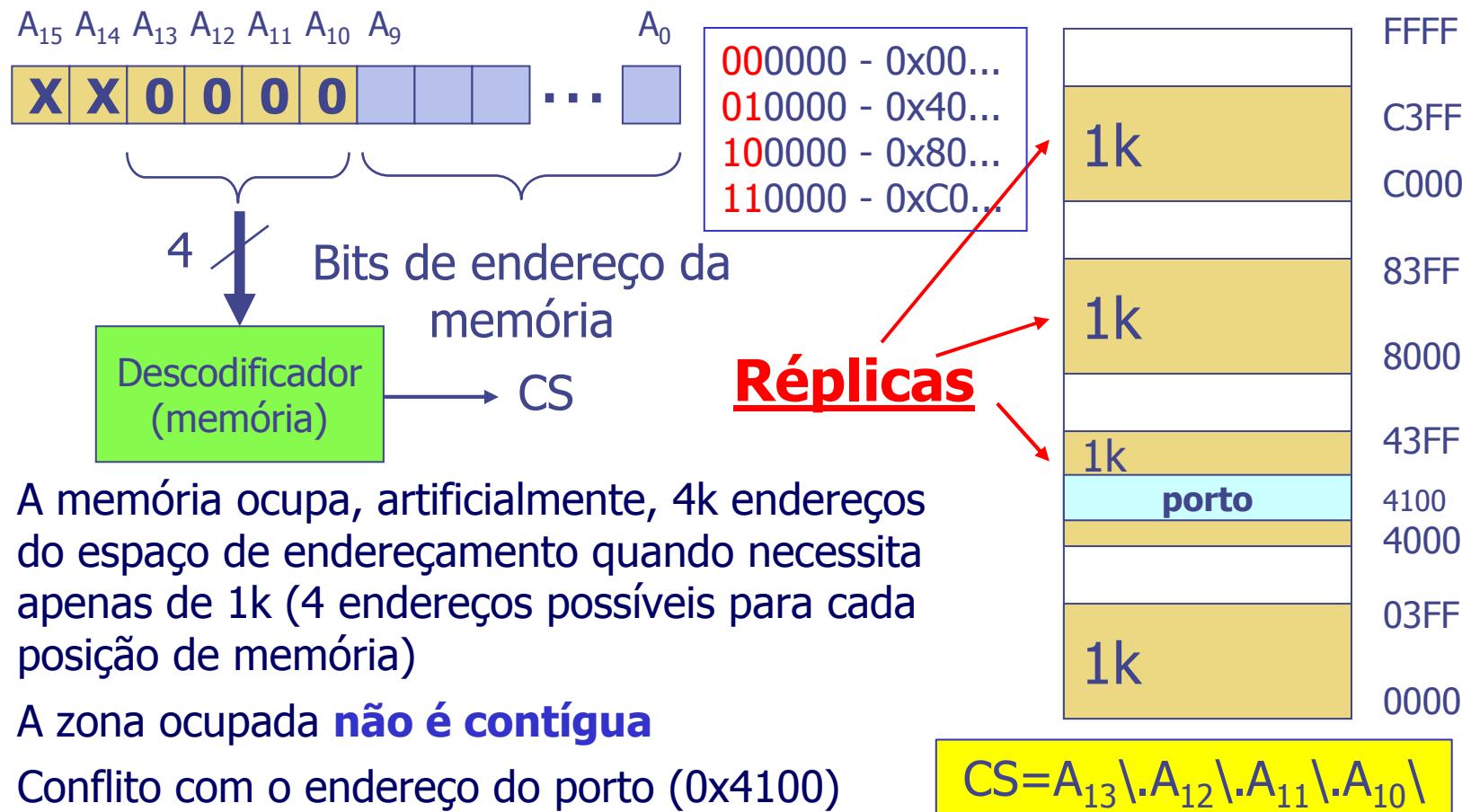
# Descodificação de endereços – descodificação parcial

- Solução 2 – usar A<sub>15</sub>, A<sub>14</sub>, A<sub>13</sub> e A<sub>12</sub> e ignorar A<sub>11</sub> e A<sub>10</sub>, e.g. **0000xx**. Isto significa que um endereço válido para aceder à memória não depende do valor dos bits A<sub>11</sub> e A<sub>10</sub>, mas tem que ter os bits A<sub>15</sub> a A<sub>12</sub> a 0.



# Descodificação de endereços – descodificação parcial

- Solução 3 – usar A<sub>13</sub>, A<sub>12</sub>, A<sub>11</sub> e A<sub>10</sub> e ignorar A<sub>15</sub> e A<sub>14</sub>, e.g. **xx0000**. Isto significa que um endereço válido para aceder à memória não depende do valor dos bits A<sub>15</sub> e A<sub>14</sub>, mas tem que ter os bits A<sub>13</sub> a A<sub>10</sub> a 0

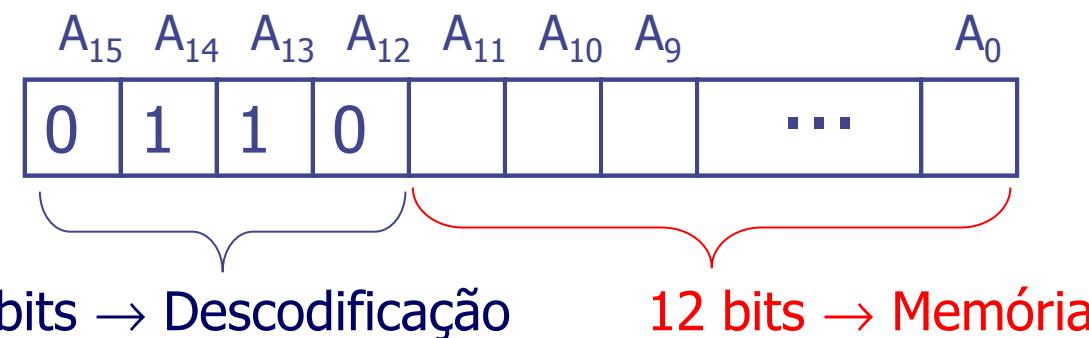


- A memória ocupa, artificialmente, 4k endereços do espaço de endereçamento quando necessita apenas de 1k (4 endereços possíveis para cada posição de memória)
- A zona ocupada **não é contígua**
- Conflito com o endereço do porto (0x4100)

# Descodificação de endereços – exercício

- Escrever a equação lógica do descodificador de endereços para uma memória de 4 kByte, mapeada num espaço de endereçamento de 16 bits, que respeite os seguintes requisitos:
  - Endereço inicial: 0x6000; descodificação total.

$$4 \text{ kByte} = 2^{12} (2^{12} - 1 = 0\text{xFFFF})$$



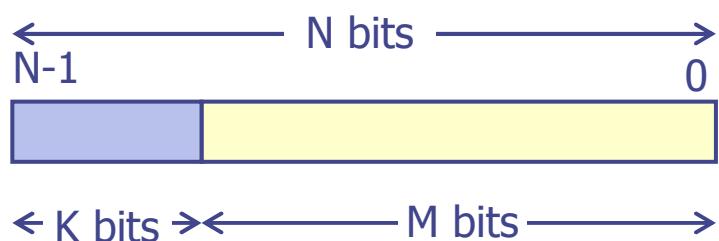
0110000000000000 (0x6000)

0110111111111111 (0x6FFF)

- Lógica positiva:  $CS = A_{15}\backslash \cdot A_{14} \cdot A_{13} \cdot A_{12}\backslash$
- Lógica negativa:  $CS\backslash = A_{15} + A_{14}\backslash + A_{13}\backslash + A_{12}$

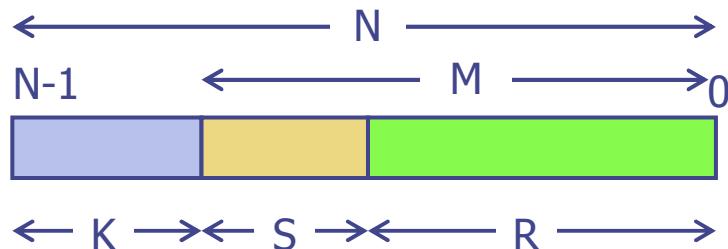
# Gerador de sinais de seleção programável

- Como se viu anteriormente, os N bits do espaço de endereçamento podem, para efeitos de descodificação de endereços e endereçamento, ser divididos em dois grupos: M bits usados para endereçamento dentro da gama descodificada e os restantes K bits usados para descodificação



- Dimensão da gama descodificada:  $2^M$
- Endereço inicial da gama descodificada é definida pela combinação binária dos K bits
- Número de gamas que podem ser descodificadas:  $2^K$

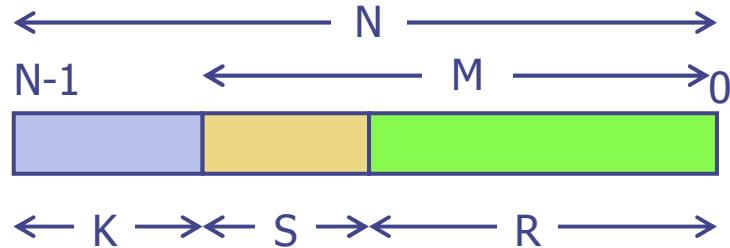
- O mesmo método pode ser aplicado para a sub-divisão dos M bits da gama descodificada: S bits usados para descodificação, R bits usados para endereçamento



- Número de sub-gamas que podem ser descodificadas:  $2^S$
- Dimensão da sub-gama descodificada:  $2^R$
- Endereço inicial da sub-gama descodificada é definido pelo conjunto dos bits K e S

# Gerador de sinais de seleção programável - exemplo

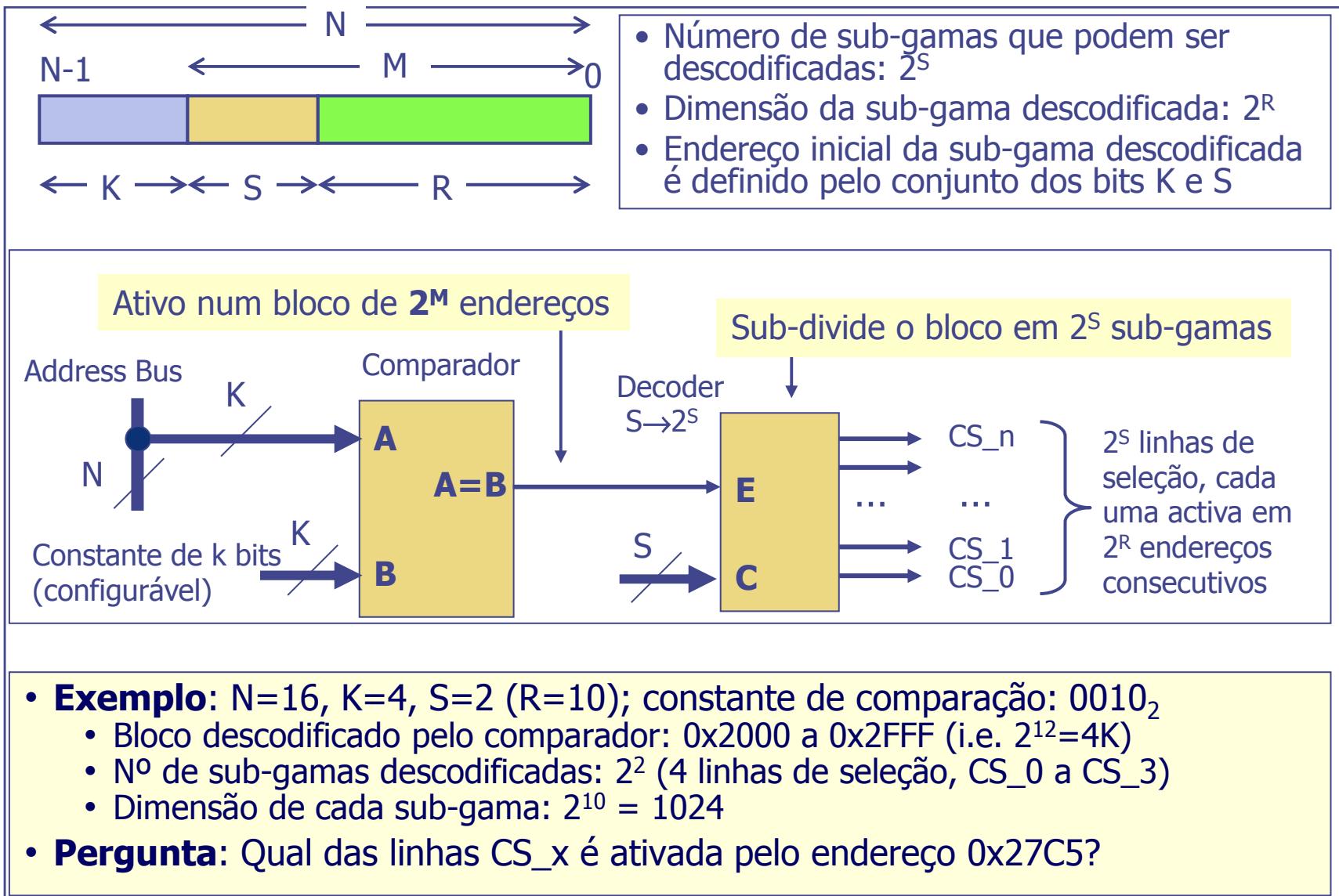
- Exemplo para um espaço de endereçamento de 8 bits (N=8)



<b>01000000</b> – <b>01000111</b>	: 0x40 – 0x47
<b>01001000</b> – <b>01001111</b>	: 0x48 – 0x4F
<b>01010000</b> – <b>01010111</b>	: 0x50 – 0x57
<b>01011000</b> – <b>01011111</b>	: 0x58 – 0x5F

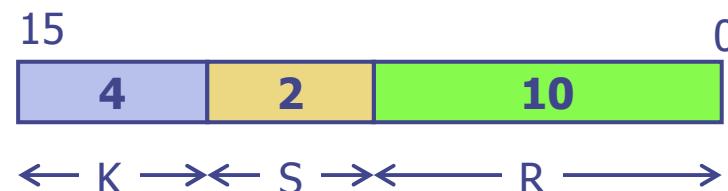
- $M=5$  ( $K=3$ ),  $S=2$  e  $R=3$ 
  - $N = 8$  – espaço de endereçamento com  $2^8 = 256$  endereços
  - $M = 5$  – gama descodificada com  $2^5 = 32$  endereços
  - $S = 2$  – número de sub-gamas  $2^2 = 4$
  - $R = 3$  – dimensão da sub-gama:  $2^3 = 8$  endereços
- A gama de  $2^5$  endereços foi sub-dividida em  $2^2$  gamas iguais, de  $2^3$  endereços cada
- O endereço inicial do bloco de  $2^2$  gamas é definido pela combinação binária usada nos  $K$  bits. Ex: **010** -> endereço inicial = **0x40**
  - gama0: 0x40 – 0x47, gama1: 0x48 – 0x4F, gama2: 0x50 – 0x57, gama3: 0x58 – 0x5F

# Gerador de sinais de seleção programável - implementação



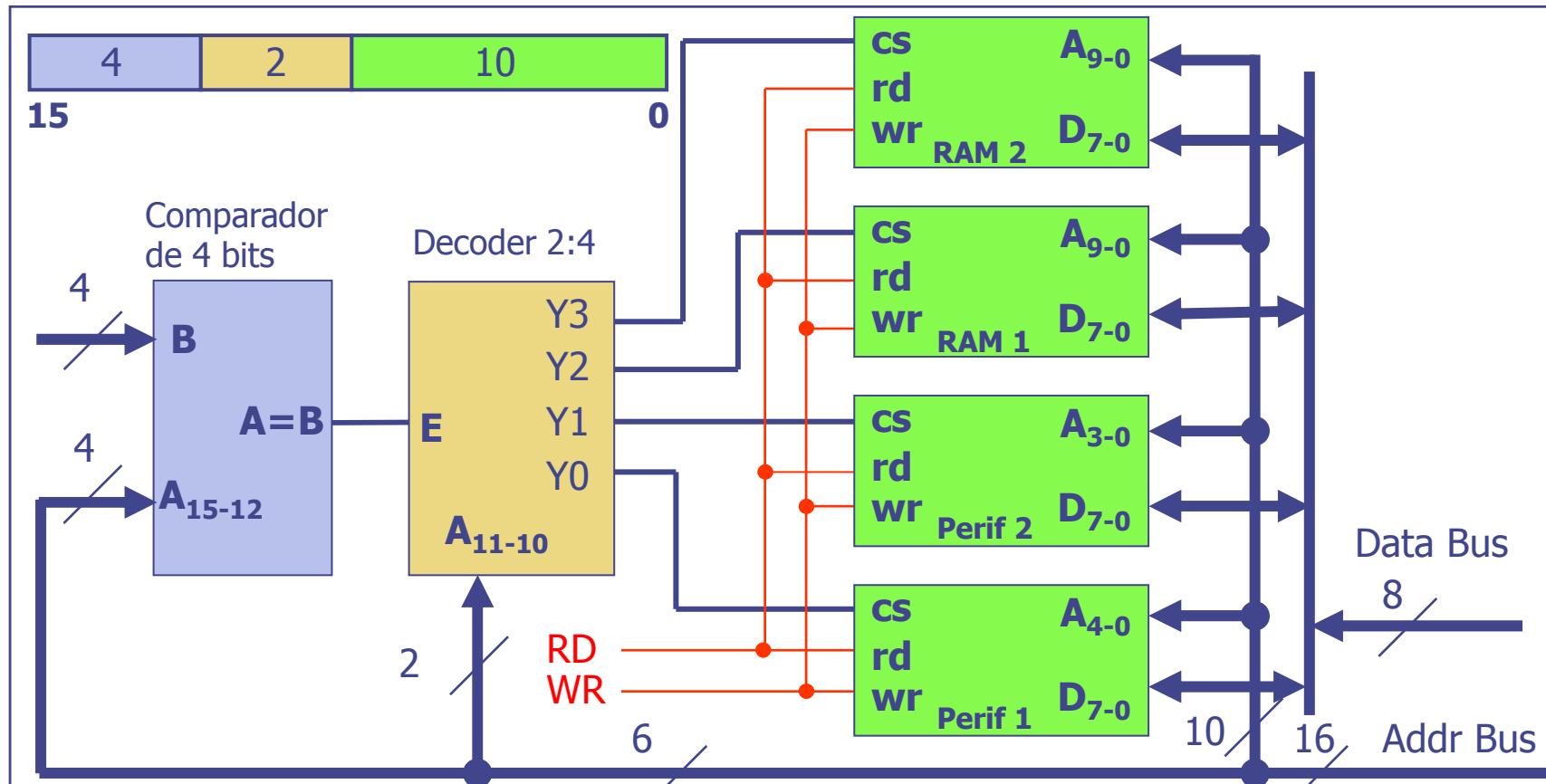
# Gerador de sinais de seleção programável – exercício

- Considerando um espaço de endereçamento de 16 bits, usar o modelo de gerador de sinais de seleção programável para implementar um descodificador de endereços para:
  - Duas memórias RAM de 1 kByte cada
  - Um periférico com 32 registos internos
  - Um periférico com 16 registros internos
- Assuma que o descodificador pode usar o espaço de endereçamento a partir do endereço 0xB000



- Solução:
  - 4 sinais de seleção ( $S=2$ ), cada um ativo em 1024 endereços consecutivos ( $R=10$ )
  - Constante de comparação usada no comparador:  $1011_2$

# Gerador de sinais de seleção programável - exercício



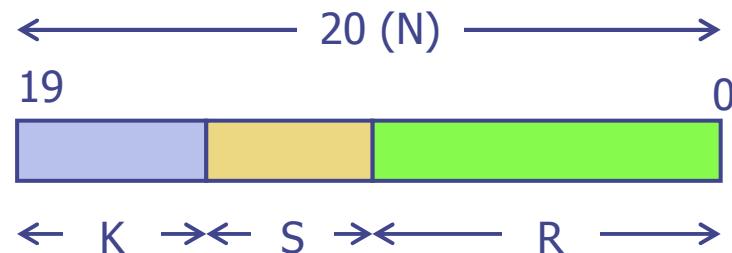
- Indique qual o dispositivo selecionado quando o endereço é 0xB935
- Construa o mapa de memória com os endereços inicial e final em que cada uma das 4 linhas de seleção está ativa
- Construa o mapa de endereços para os 4 dispositivos
- Indique todos os possíveis endereços para aceder ao primeiro registo do periférico 1

# Descodificação de endereços - exercícios

1. Para o exemplo do slide 29, suponha que no descodificador apenas se consideram os bits A15, A13 e A11, com os valores 1, 0 e 0, respectivamente.
  - a) Apresente a expressão lógica que implementa este descodificador: i) em lógica positiva e ii) em lógica negativa.
  - b) Indique os endereços inicial e final da gama-base descodificada e de todas as réplicas.
2. Suponha que, no exercício do slide 33, não se descodificaram os bits A14 e A12, resultando na expressão  $CS\backslash = A15 + A13\backslash$ 
  - a) Indique as gamas do espaço de endereçamento de 16 bits ocupadas pela memória.
  - b) Indique os endereços possíveis para aceder à 15<sup>a</sup> posição da memória.
3. Escreva as equações lógicas dos 4 descodificadores necessários para a geração dos sinais de seleção para cada um dos dispositivos do exemplo do slide 16.
4. Para o exemplo do slide 36, determine a gama de endereços em que cada uma das linhas  $CS\_x$  está ativa, com a constante de comparação  $0010_2$

## Gerador de sinais de seleção programável - exercícios

1. Pretende-se gerar os sinais de seleção para 4 memórias de 8 kByte, a mapear em gamas de endereços consecutivas, de modo a formar um conjunto de 32 kByte. O endereço inicial deve ser configurável. Para um espaço de endereçamento de 20 bits:



- a) Indique o número de bits dos campos  $K$ ,  $S$  e  $R$ , supondo descodificação total.
- b) Esboce o circuito digital que implementa este descodificador
- c) Indique os endereços inicial e final para a primeira, segunda e última gamas de endereços possíveis de serem descodificadas.
- d) Para a última gama de endereços, indique os endereços inicial e final atribuídos a cada uma das 4 memórias de 8k
- e) Suponha que o endereço 0x3AC45 é um endereço válido para aceder ao conjunto de 32k. Indique os endereços inicial e final da gama que inclui este endereço. Indique os endereços inicial e final da memória de 8K à qual está atribuído este endereço

# Gerador de sinais de seleção programável - exercícios

1. Pretende-se gerar os sinais de seleção para os seguintes 4 dispositivos: 1 porto de saída de 1 byte, 1 memória RAM de 1 kByte (*byte-addressable*), 1 memória ROM de 2 kByte (*byte-addressable*), 1 periférico com 5 registos de 1 byte cada um. O espaço de endereçamento a considerar é de 20 bits.
  - a) Desenhe o gerador de linhas de seleção para estes 4 dispositivos, baseando-se no modelo discutido nos slides anteriores e usando a mesma sub-gama para o periférico e para o porto de saída de 1 byte.
  - b) Especifique a dimensão de todos os barramentos e quais os bits que são usados.
  - c) Desenhe o mapa de memória com o endereço inicial e final do espaço efetivamente ocupado por cada um dos 4 dispositivos, considerando para o conjunto um endereço-base por si determinado.
2. O periférico com 5 registos, do exercício anterior, tem um barramento de endereços com três bits. Suponha que esses bits estão ligados aos bits A0, A1 e A2 do barramento de endereços do CPU.
  - a) Usando o descodificador desenhado no exercício anterior, indique os 16 primeiros endereços em que é possível aceder ao registo 0 (selecionado com A0, A1 e A2 a 0)
  - b) Repita o exercício anterior supondo que os 3 bits do barramento de endereços do periférico estão ligados aos bits A2, A3 e A4 do barramento de endereços.

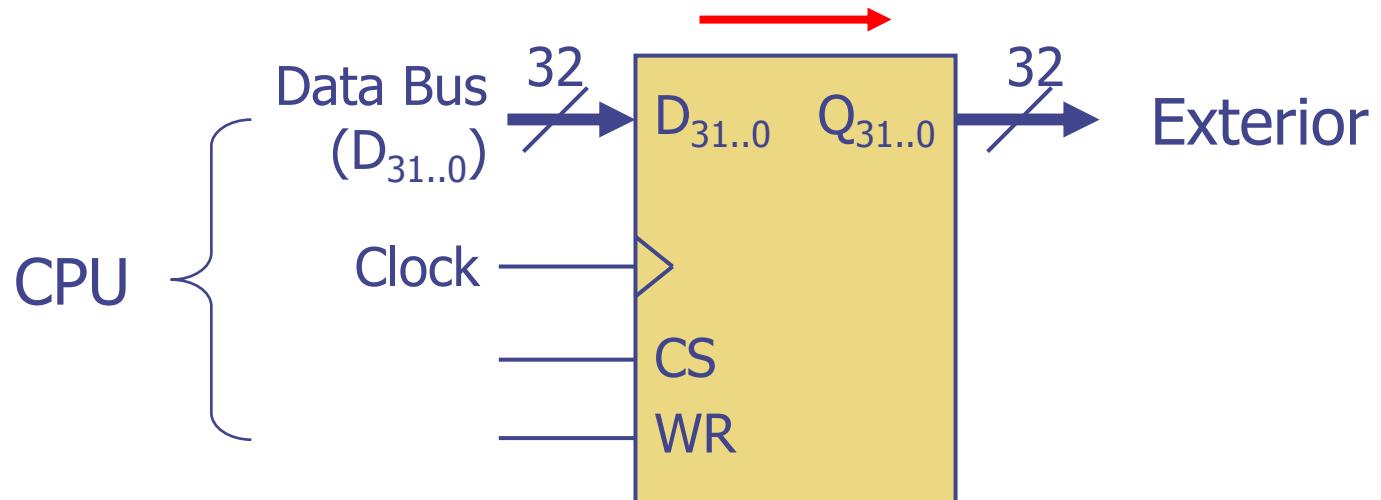
## Aula 4

- Estrutura básica de portos de I/O
- Portos de I/O no PIC32
  - Estrutura básica de um porto de I/O de 1 bit
  - Estrutura dos portos de I/O de "n" bits.
- Exemplos de programação em *assembly*

José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

# Porto de saída de 32 bits

- Porto de saída de 32 bits (constituído por um único registo de 32 bits)

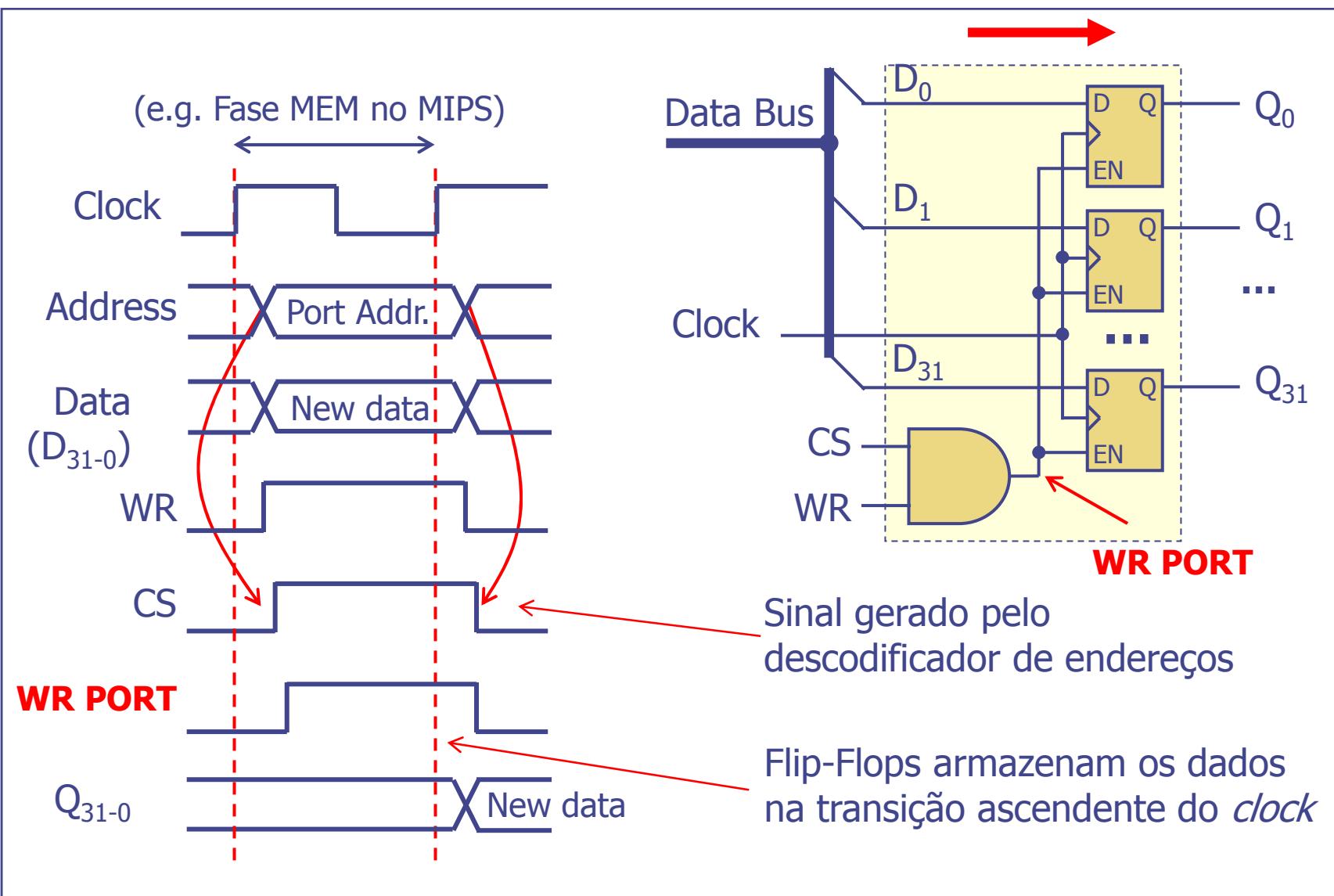


- O porto armazena informação proveniente do CPU, transferida durante uma operação de escrita na memória (estágio MEM nas instruções "**sw**", no caso do MIPS)
- A escrita no porto é feita na transição ativa do relógio se os sinais "**cs**" e "**wR**" estiverem ambos ativos
- O sinal "**cs**" é gerado pelo descodificador de endereços: fica ativo se o endereço gerado pelo CPU coincidir com o endereço atribuído ao porto

# Porto de saída de 32 bits (descrição em VHDL)

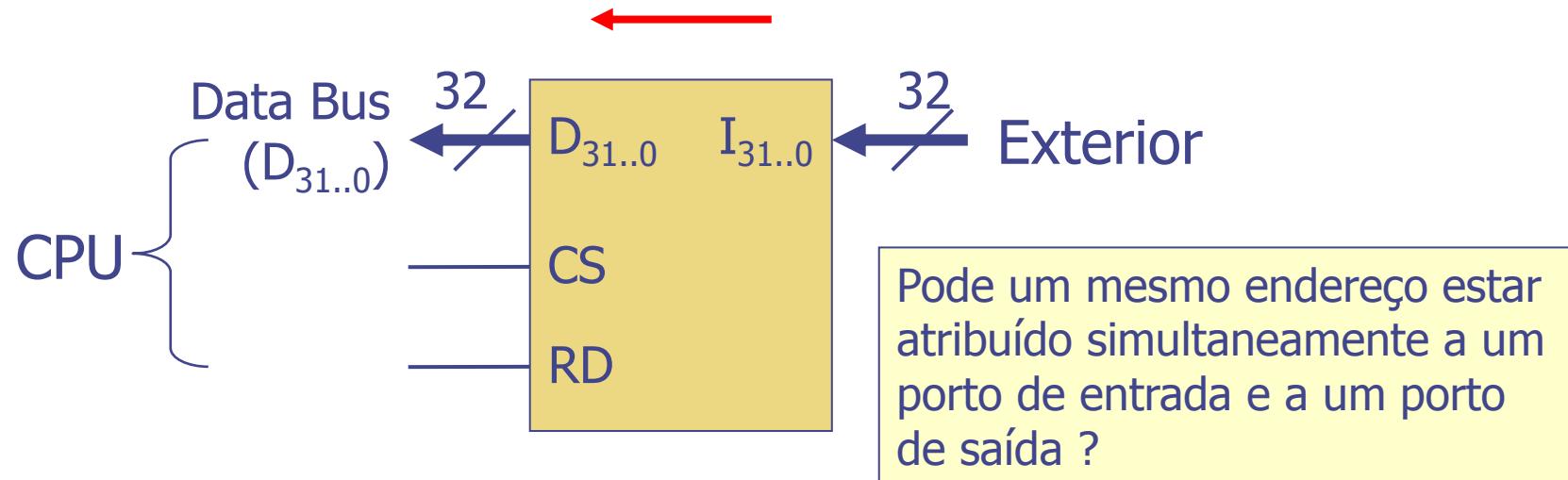
```
entity OutPort is
    port(clk, wr, cs : in std_logic;
          dataIn      : in std_logic_vector(31 downto 0);
          dataOut     : out std_logic_vector(31 downto 0));
end OutPort;
architecture behav of OutPort is
begin
    process(clk)
    begin
        if(rising_edge(clk)) then
            if(cs = '1' and wr = '1') then
                dataOut <= dataIn;
            end if;
        end if;
    end process;
end behav;
```

# Porto de saída de 32 bits



# Porto de entrada de 32 bits

- Porto de entrada de 32 bits (em geral, um porto de entrada não tem capacidade de armazenamento)

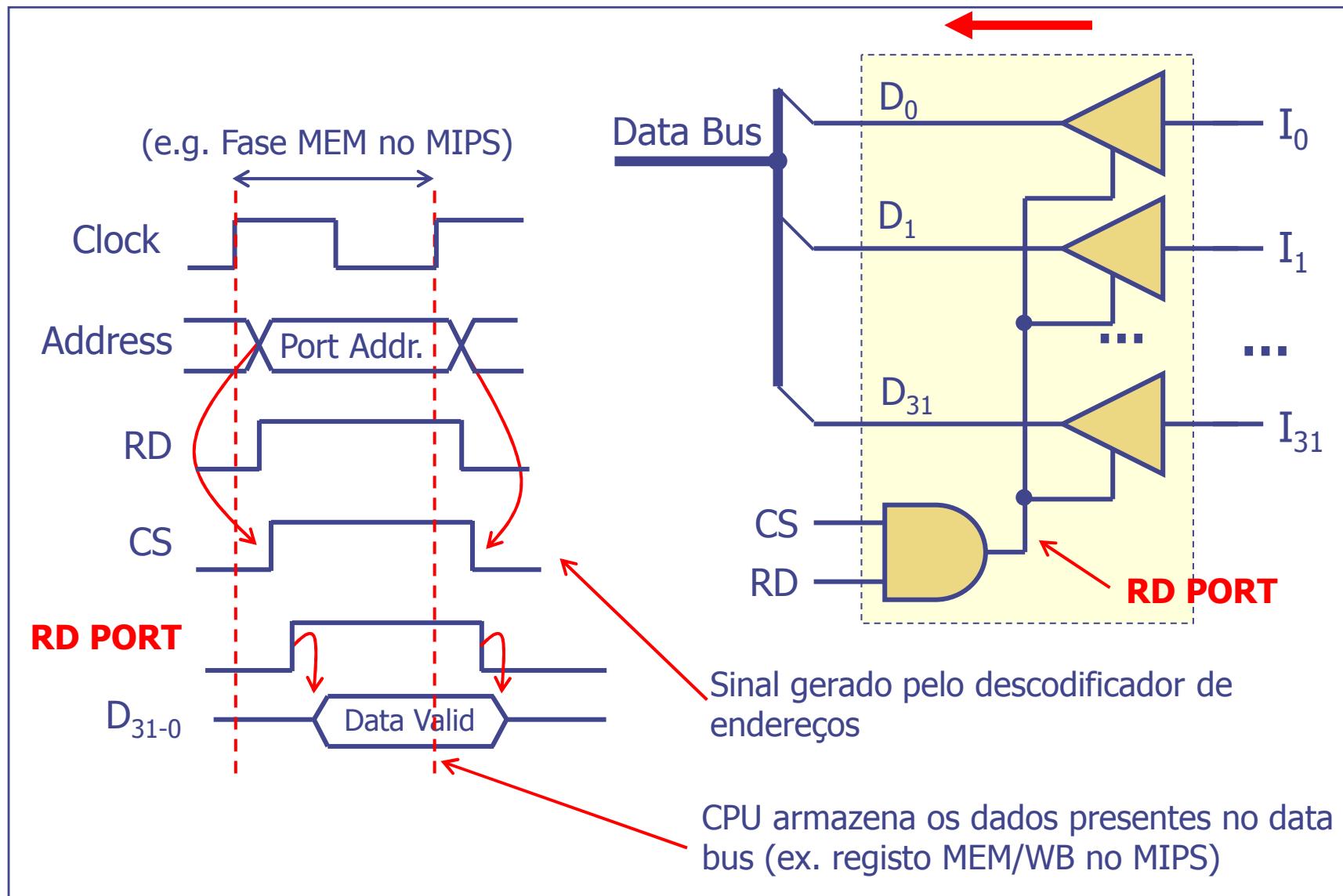


- A informação presente nas 32 linhas de entrada ( $I_{31..0}$ ) é transferida para o CPU durante uma operação de leitura (estágio MEM nas instruções "**lw**", no caso do MIPS)
- As saídas  $D_{31..0}$  têm obrigatoriamente portas *tri-state* que só são ativadas quando estão ativos, simultaneamente, os sinais "**CS**" e "**RD**"
- Ao nível do porto, a operação de leitura é assíncrona, pelo que não é necessário o sinal de relógio

# Porto de entrada (descrição em VHDL)

```
entity InPort is
    port(rd, cs  : in  std_logic;
          dataIn  : in  std_logic_vector(31 downto 0);
          dataOut : out std_logic_vector(31 downto 0));
end InPort;
architecture behav of InPort is
begin
    process(rd, cs, dataIn)
    begin
        if(cs = '1' and rd = '1') then
            dataOut <= dataIn;
        else
            dataOut <= (others => 'Z');
        end if;
    end process;
end behav;
```

# Porto de entrada de 32 bits



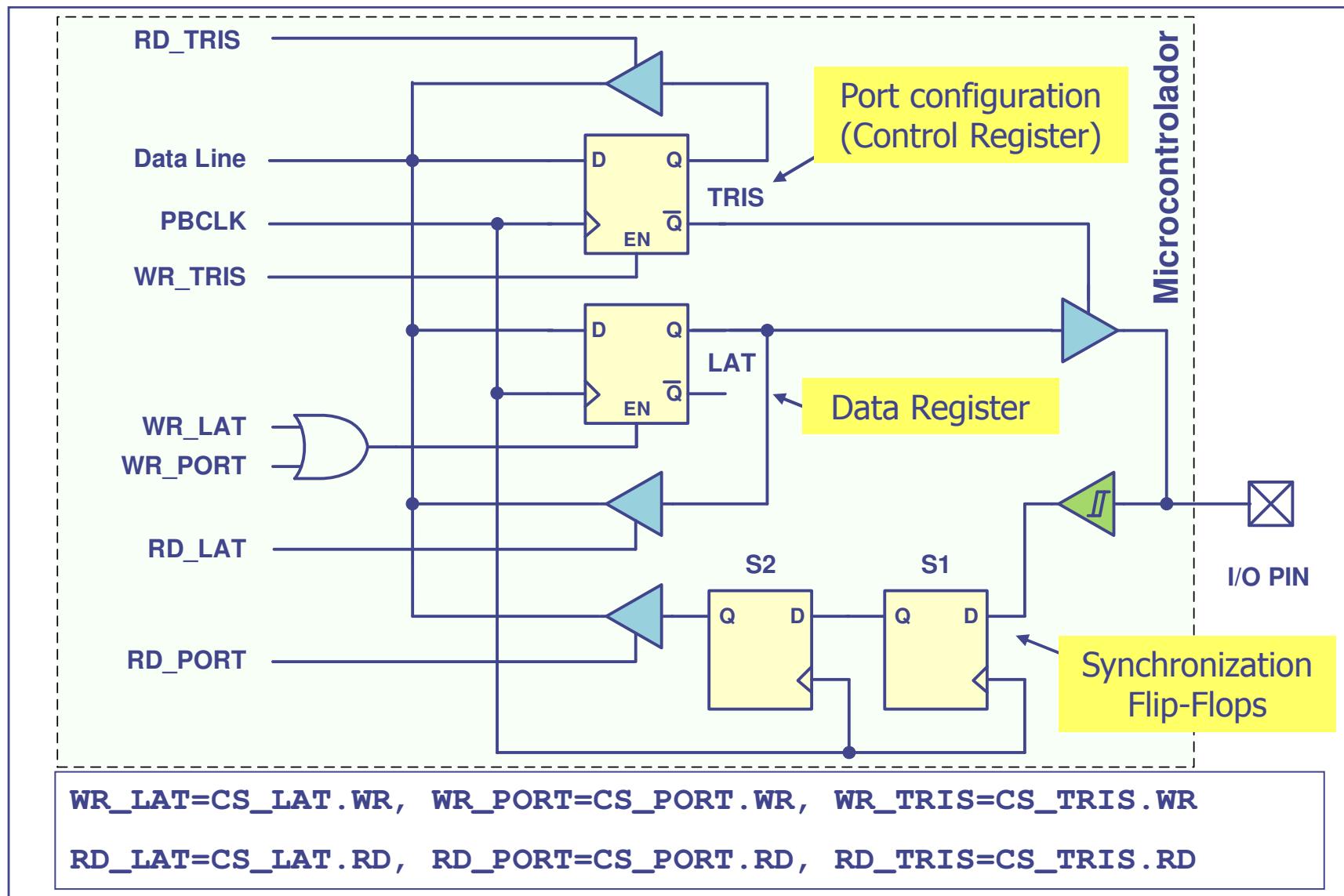
# Portos de I/O no PIC32

- O microcontrolador PIC32MX795F512H disponibiliza vários portos de I/O, com várias dimensões (16 bits, no máximo)
  - Porto B (RB): 16 bits, I/O
  - Porto C (RC): 2 bit, I/O
  - Porto D (RD): 12 bits, I/O
  - Porto E (RE): 8 bits, I/O
  - Porto F (RF): 5 bits, I/O
  - Porto G (RG): 4 de I/O + 2 I
- Cada um dos bits de cada um destes portos pode ser configurado, por programação, como entrada ou saída
  - **um porto de I/O de n bits do PIC32 é um conjunto de n portos de I/O de 1 bit**

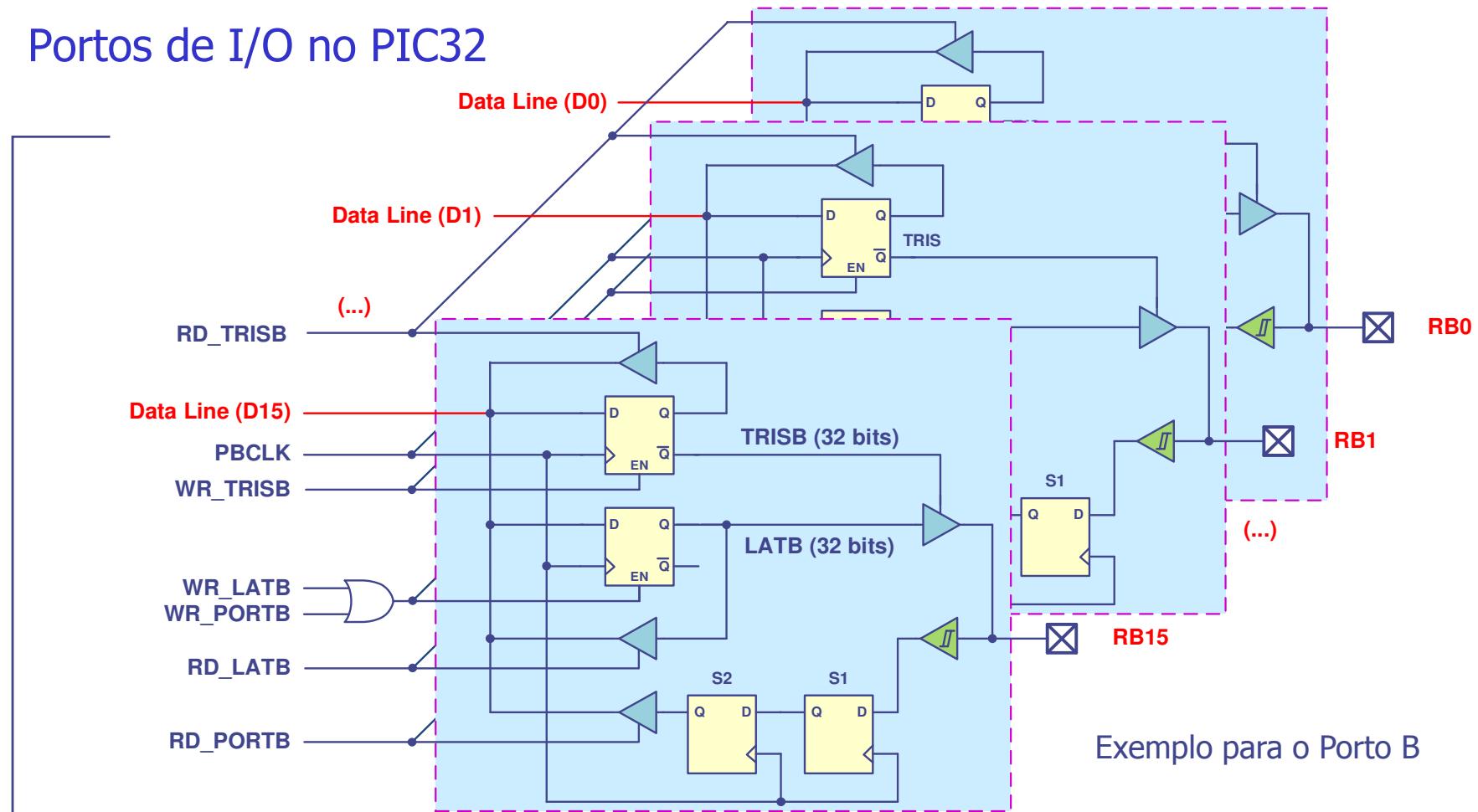
# Portos de I/O no PIC32

- Cada um dos portos (B a G) tem associado um total de 12 registos **de 32 bits**. Desses, os que vamos usar são:
  - **TRIS** – usado para configuração do porto (entrada ou saída)
  - **PORT** – usado para ler valores de um porto de entrada
  - **LAT** – usado para escrever valores num porto de saída
- A configuração de cada um dos bits de um porto, como entrada ou como saída, é feita através dos registos **TRIS** ("Tri-state" *registers*)
  - bit **n** do registo TRIS = 1: bit **n** do porto configurado como entrada
  - bit **n** do registo TRIS = 0: bit **n** do porto configurado como saída
- Exemplo para o porto E (8 bits): **TRISE** = 000...10101010<sub>2</sub>
  - portos RE0, RE2, RE4 e RE6 configurados como saída
  - portos RE1, RE3, RE5 e RE7 configurados como entrada

# Modelo simplificado de um porto de I/O de 1 bit no PIC32



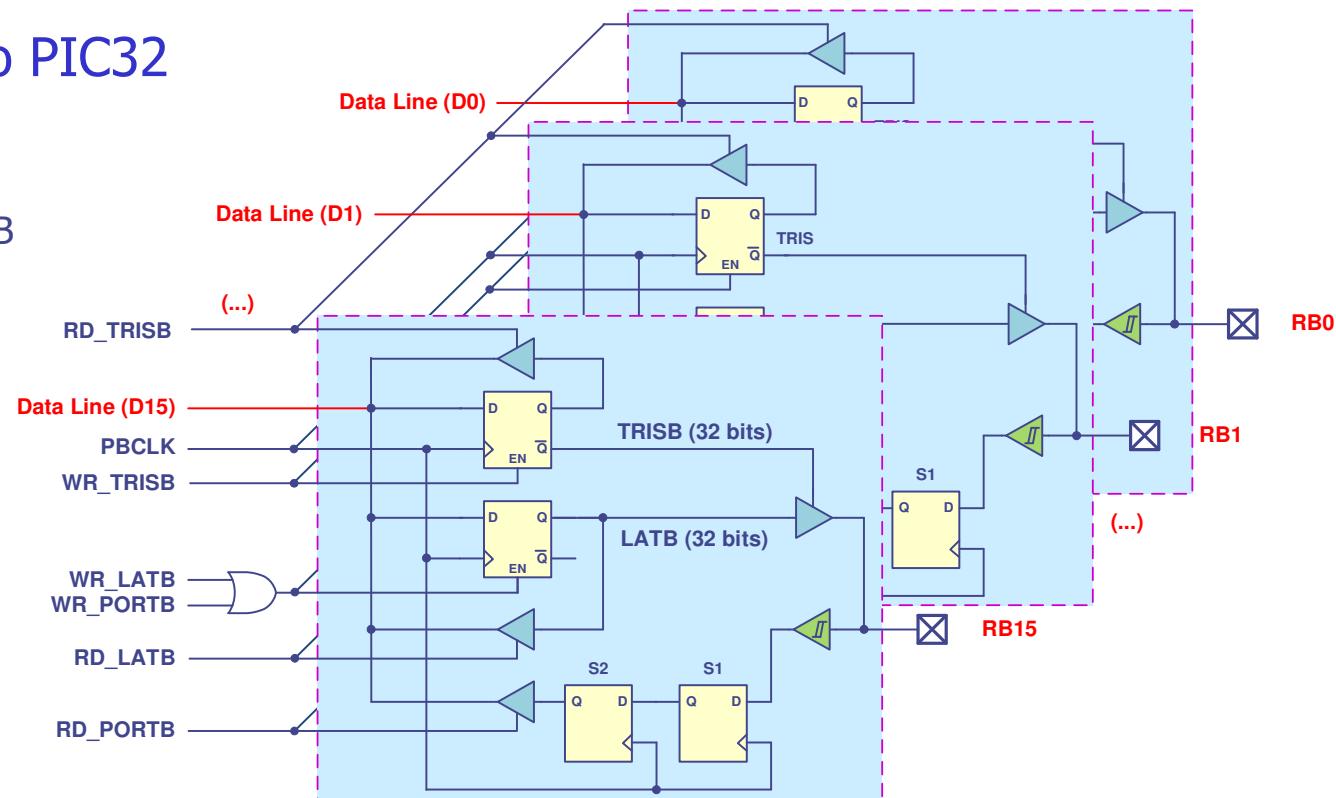
## Portos de I/O no PIC32



- Registro TRISx (TRISB, TRISC, ...) agrupa todos os flip-flop TRIS dos portos de I/O de 1 bit; registos de 32bits (16 MSbits são irrelevantes)
- Registro LATx (LATB, LATC, ...) é o registo de dados e agrupa todos os flip-flops LAT dos portos de I/O de 1 bit; registos de 32 bits (16 MSbits são irrelevantes)

# Portos de I/O no PIC32

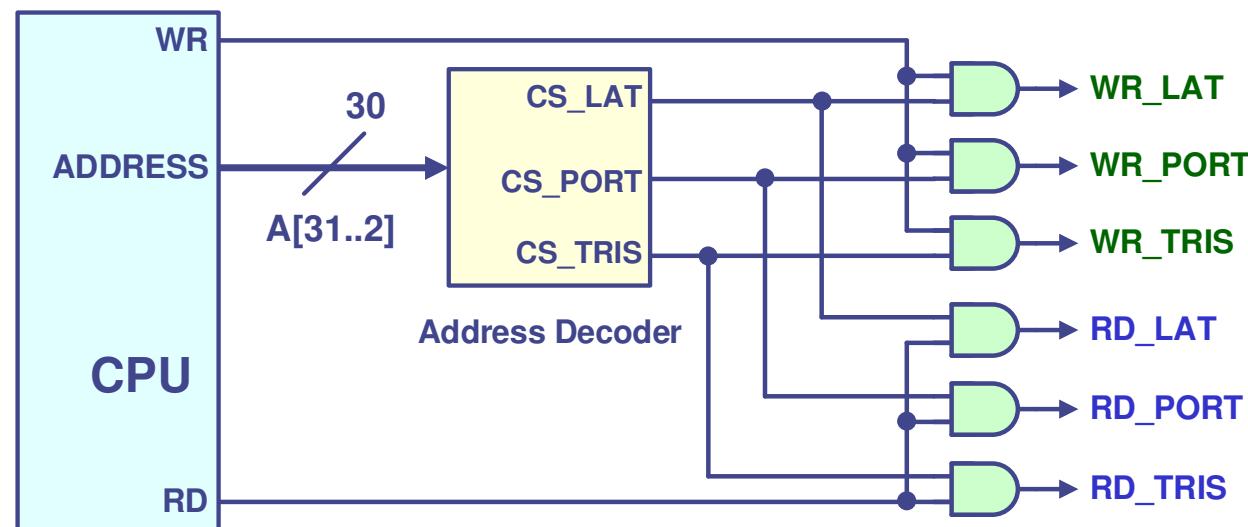
Exemplo para o Porto B



- Cada porto de entrada inclui uma porta *schmitt trigger* (comparador com histerese) que tem o objetivo de melhorar a imunidade ao ruído
- No porto de entrada, o sinal externo é sincronizado através de 2 *flip-flops* (S1 e S2). Esta configuração visa resolver os possíveis problemas causados por meta-estabilidade decorrentes do facto de o sinal externo ser assíncrono relativamente ao *clock* do CPU
  - Os dois *flip-flops*, em conjunto, impõem um atraso de, até, dois ciclos de relógio na propagação do sinal até ao barramento de dados do CPU

# Portos de I/O no PIC32

- A escrita no porto é feita no endereço referenciado pelo identificador **LAT<sub>x</sub>**, em que x é a letra que identifica o porto; a leitura do porto é feita do endereço referenciado por **PORT<sub>x</sub>**
- Os portos estão mapeados no espaço de endereçamento unificado do PIC32 (ver aula 1), em endereços definidos pelo fabricante
- Os sinais que permitem a escrita e a leitura dos 3 registos de um porto (TRIS, PORT e LAT) são obtidos por descodificação de endereços, em conjunto com os sinais RD e WR



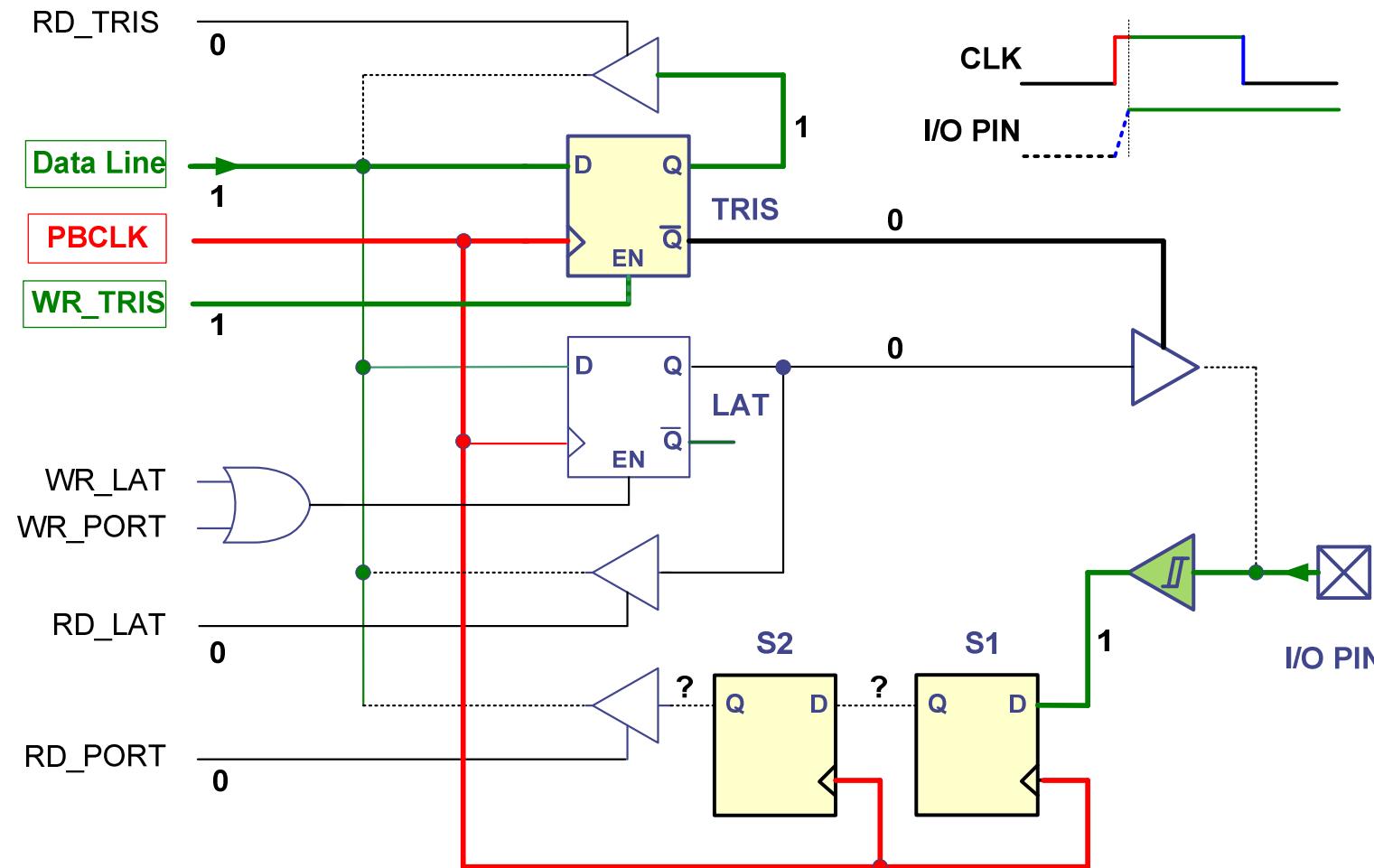
Exemplos de  
endereços:

TRISB: 0xBF886040  
PORTB: 0xBF886050  
LATB: 0xBF886060

TRISE: 0xBF886100  
PORTE: 0xBF886110  
LATE: 0xBF886120

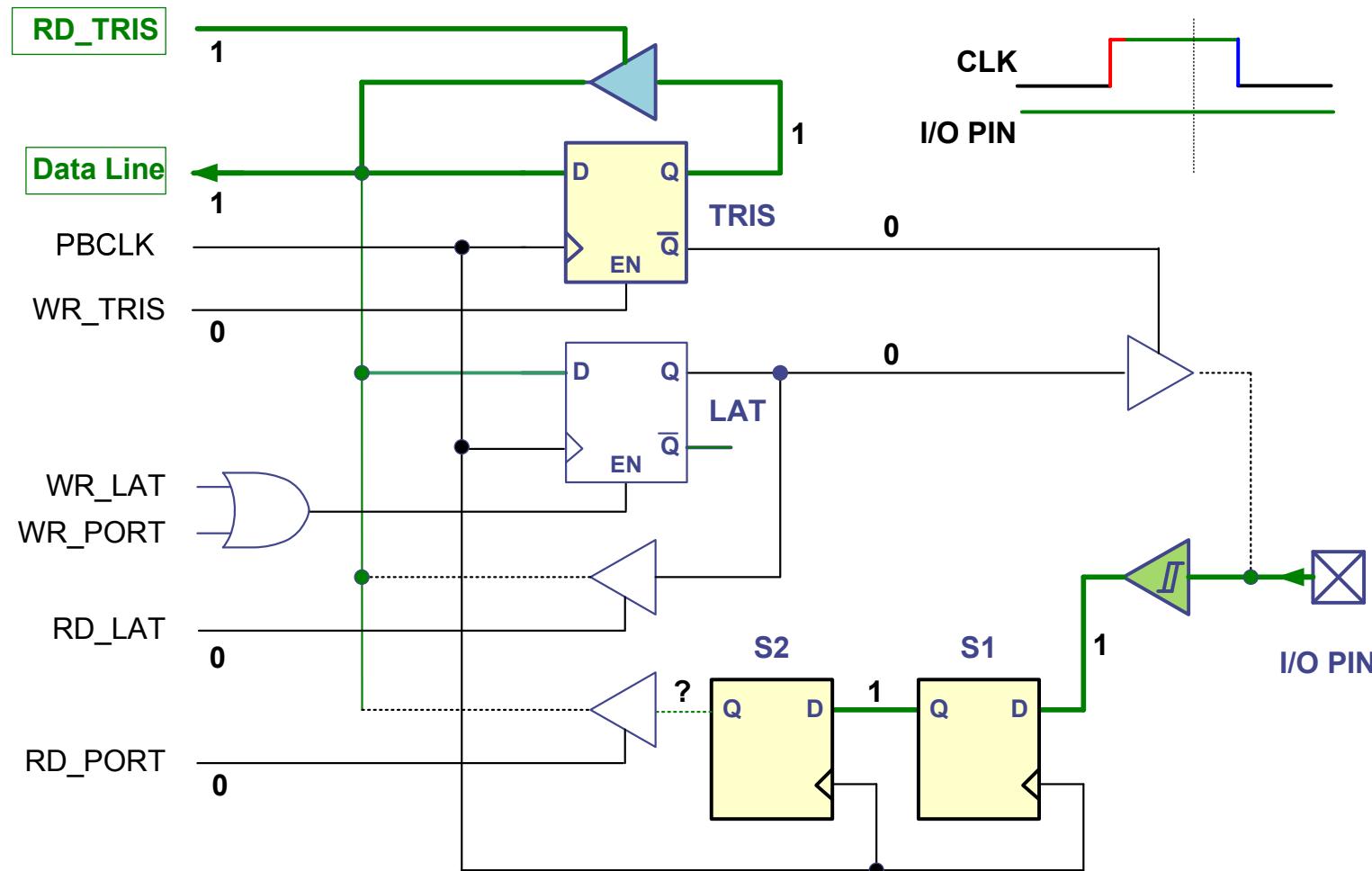
# Modelo simplificado de um porto de I/O de 1 bit no PIC32

Definir pino como porto de entrada – Escrita do valor '1' no TRIS



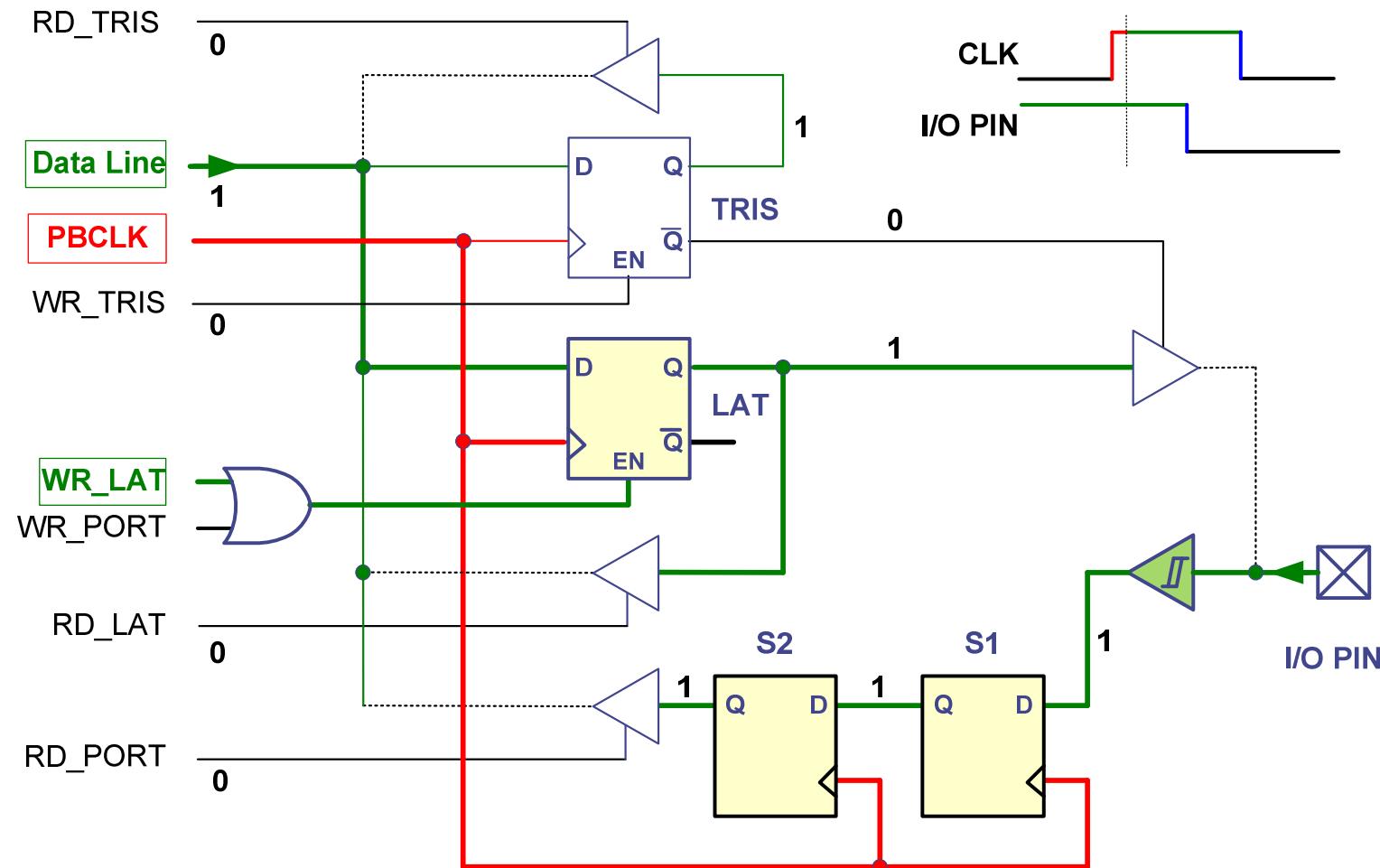
# Modelo simplificado de um porto de I/O de 1 bit no PIC32

Ler o valor do registro TRIS



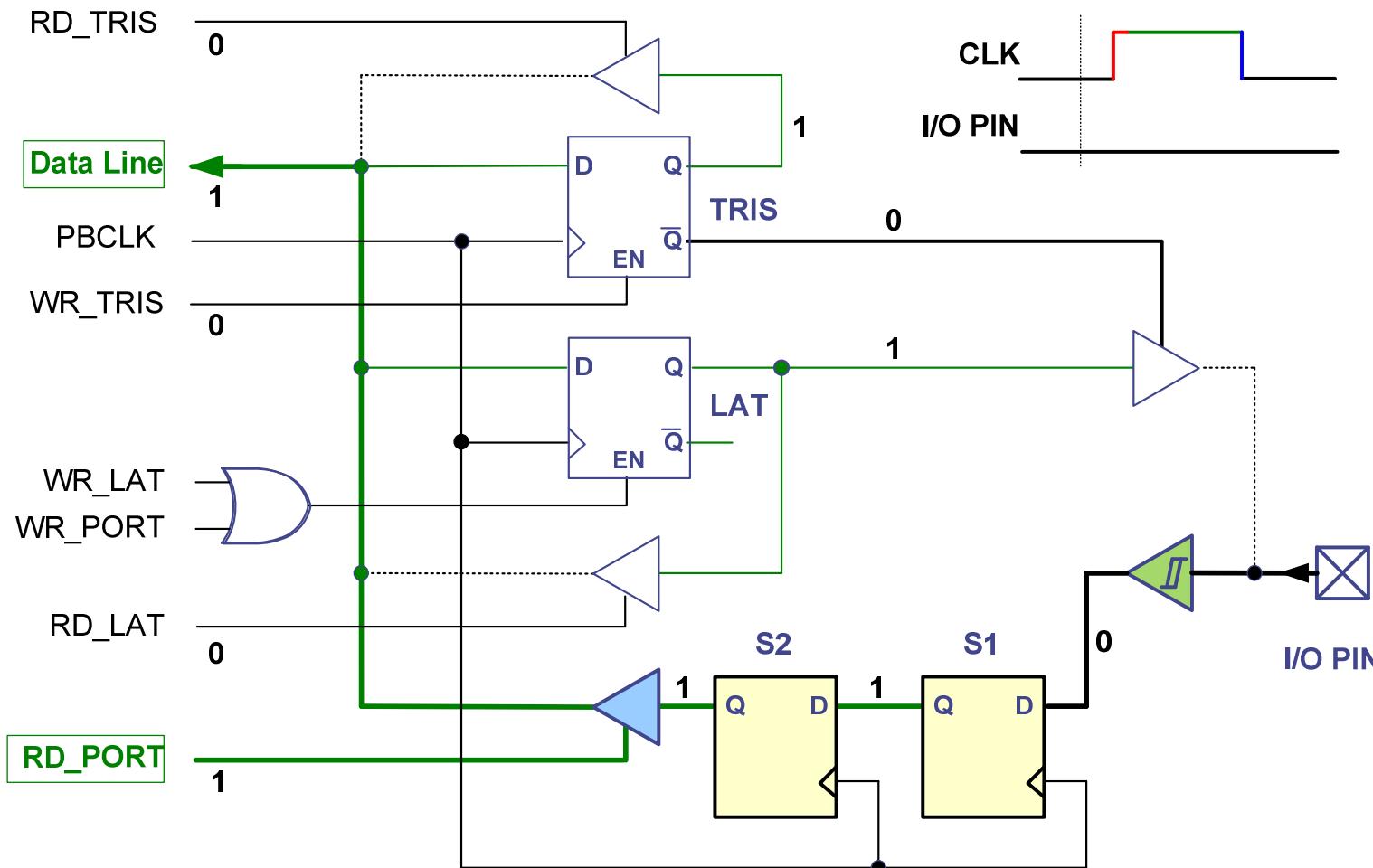
# Modelo simplificado de um porto de I/O de 1 bit no PIC32

Escrita do valor '1' no registo LAT (não influencia a saída)



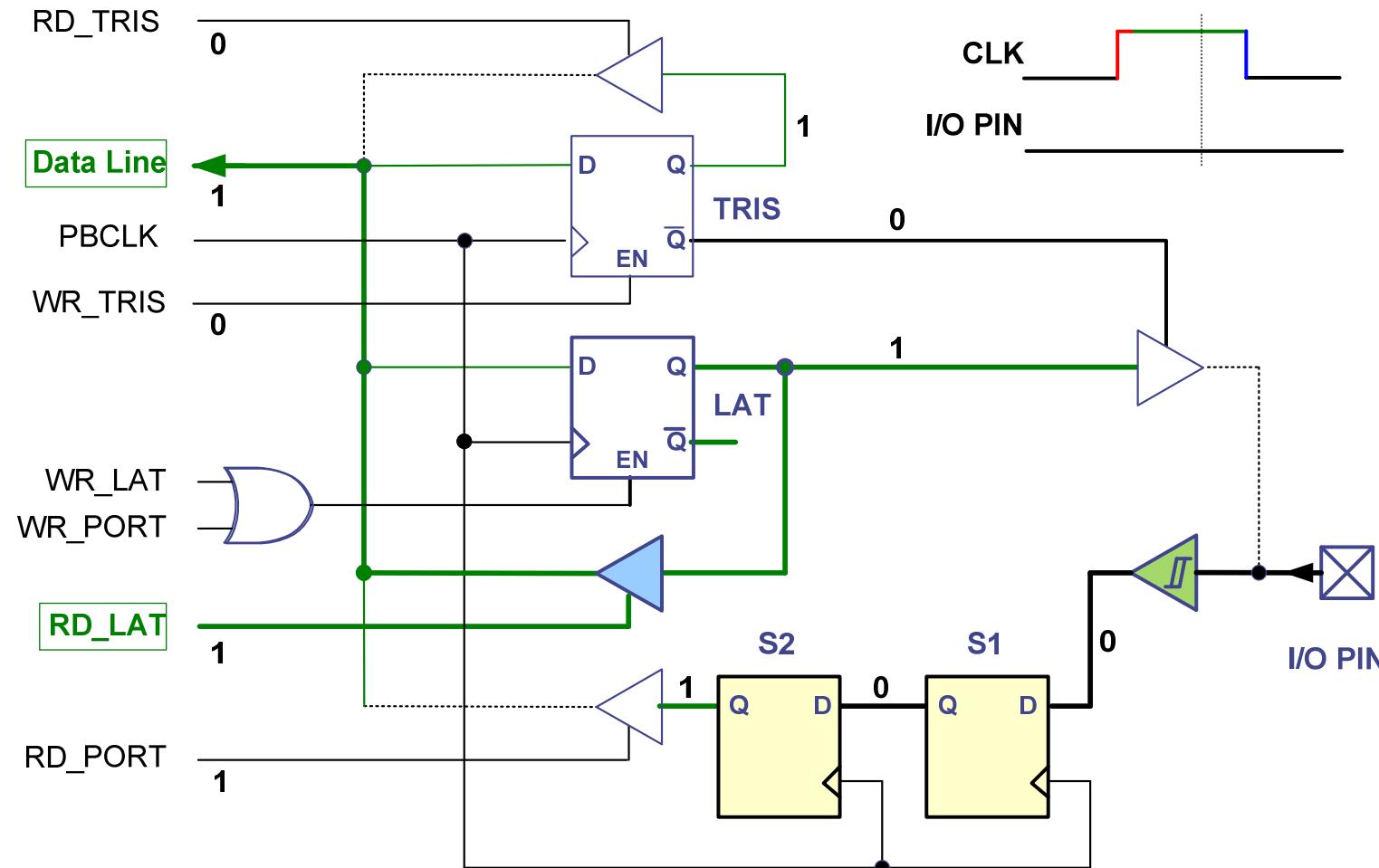
# Modelo simplificado de um porto de I/O de 1 bit no PIC32

Leitura do Porto de entrada (com 2 ciclos de relógio de atraso)



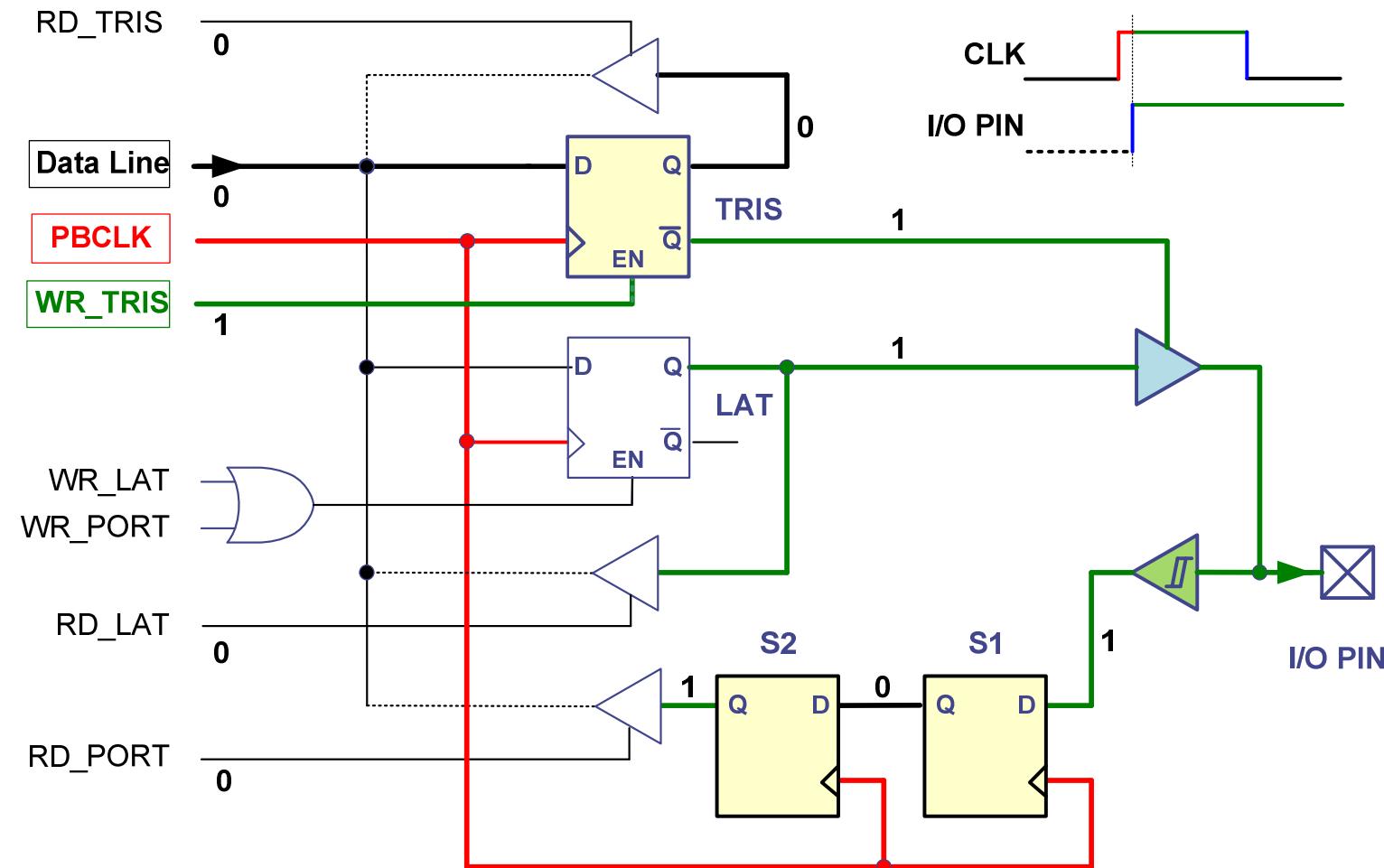
# Modelo simplificado de um porto de I/O de 1 bit no PIC32

Leitura do valor do registo LAT (pino é uma entrada)



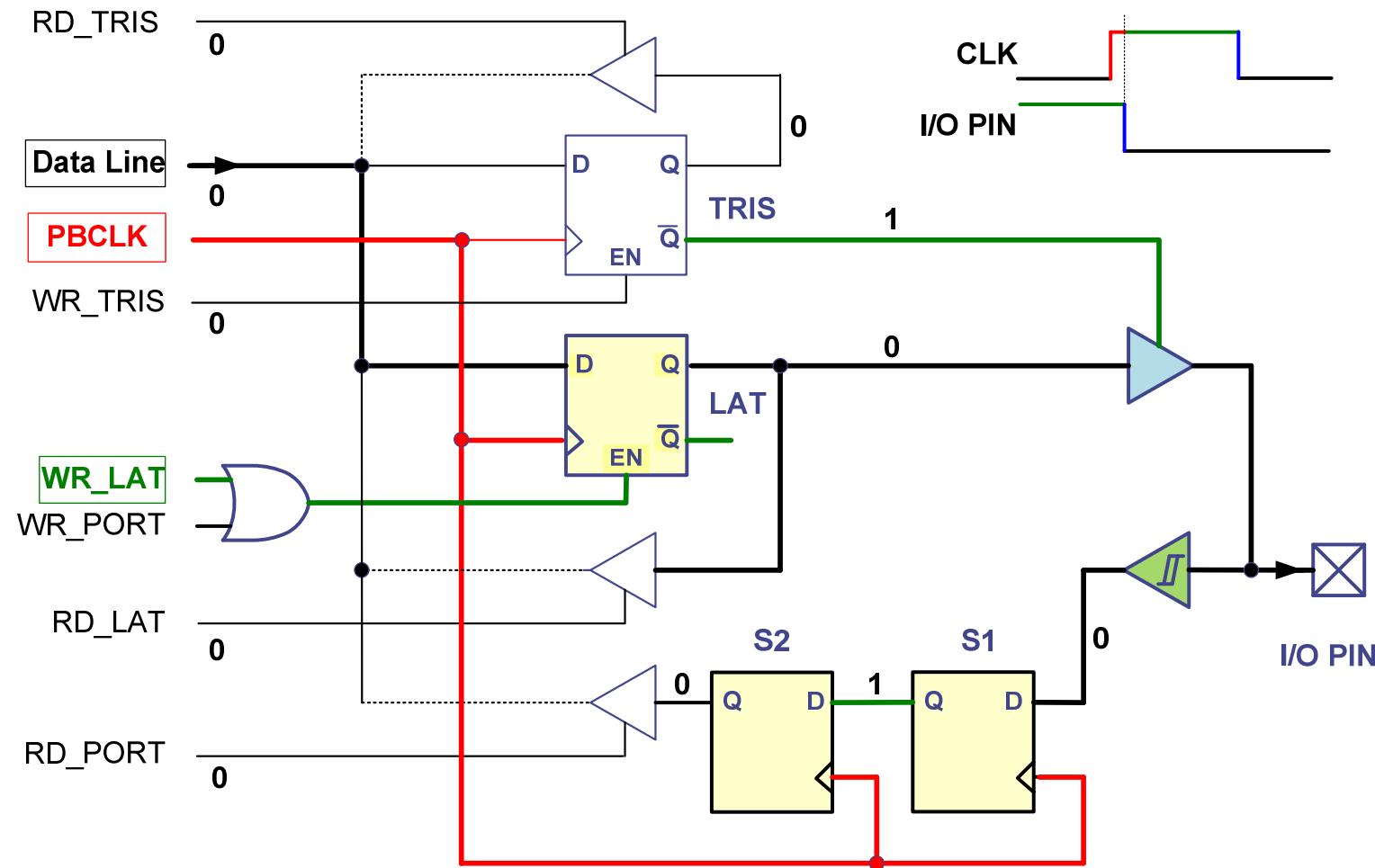
# Modelo simplificado de um porto de I/O de 1 bit no PIC32

Definir pino como porto de saída – Escrita do valor '0' no TRIS

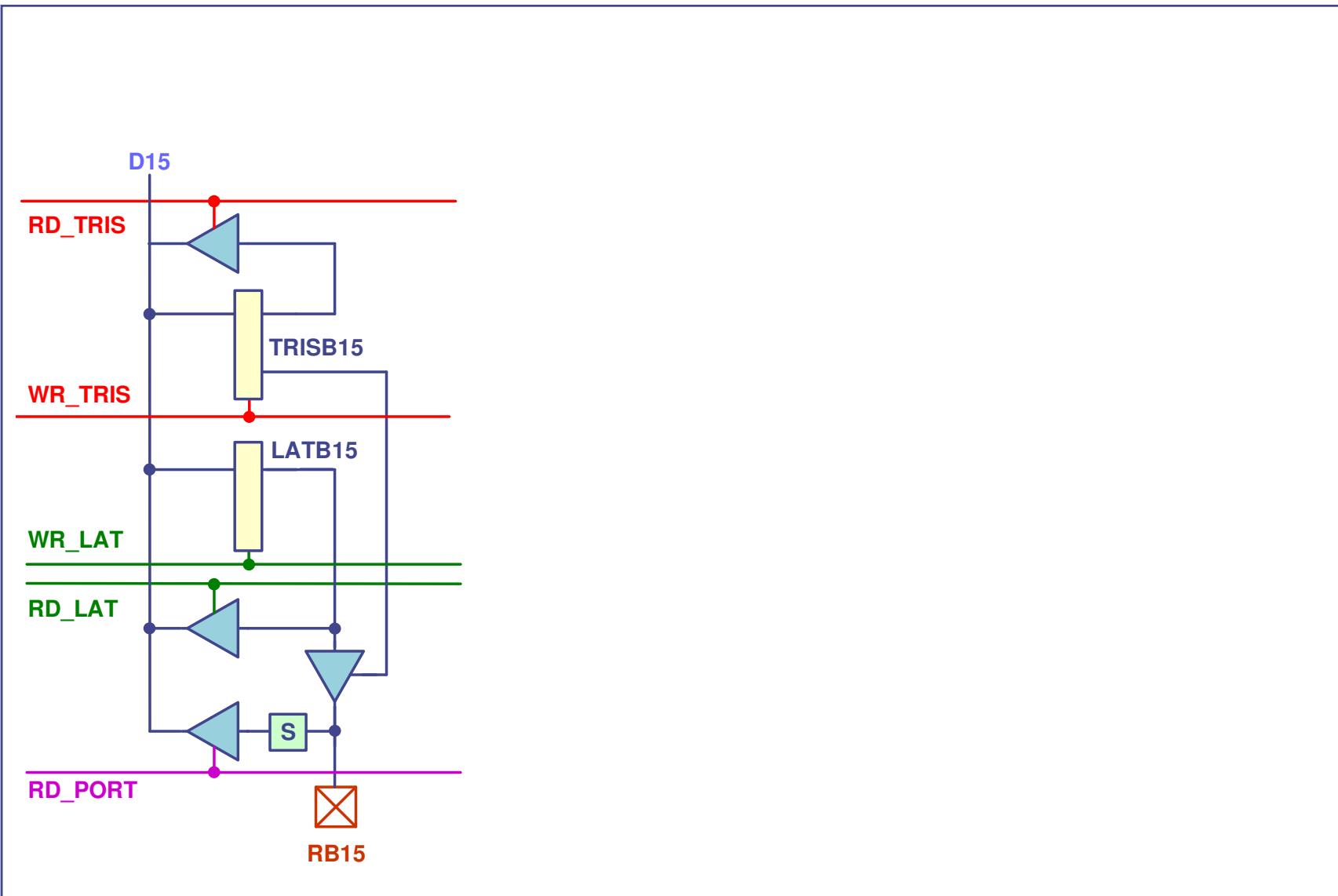


# Modelo simplificado de um porto de I/O de 1 bit no PIC32

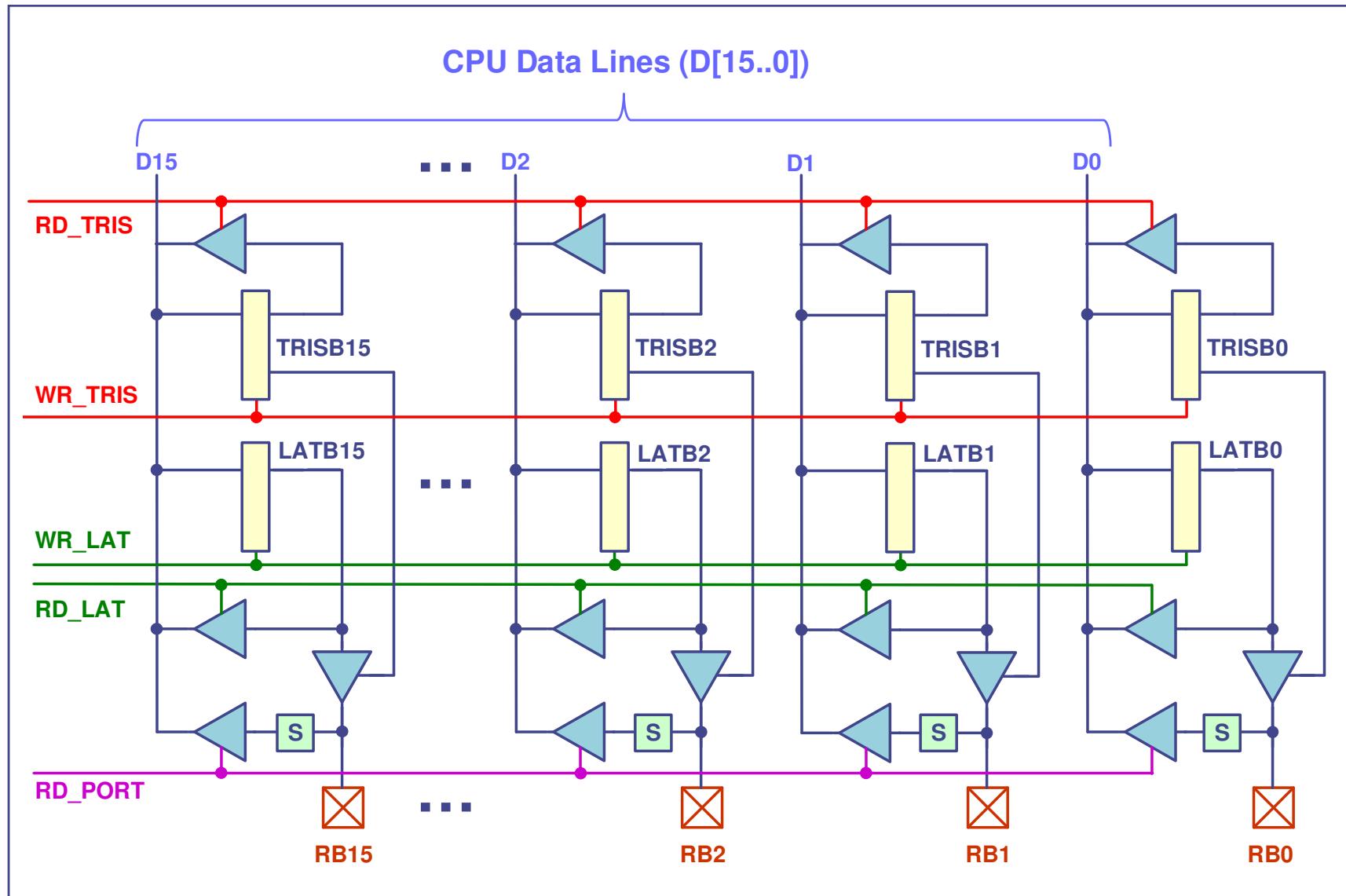
Alterar o valor do porto de saída – Escrita do valor '0' no LAT



# Modelo simplificado de um porto de I/O no PIC32



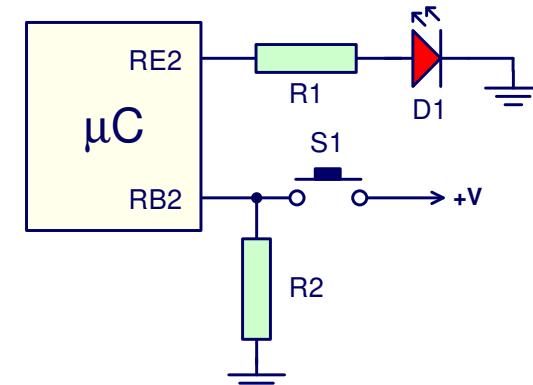
# Modelo simplificado de um porto de I/O no PIC32



# Exemplo de configuração/utilização dos portos

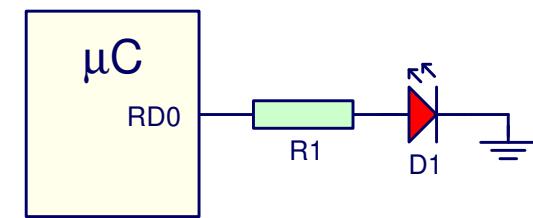
- Exemplo 1:

- Acender o LED D1 enquanto o *switch* S1 estiver premido; LED ligado ao porto RE2 e *switch* ligado ao porto RB2



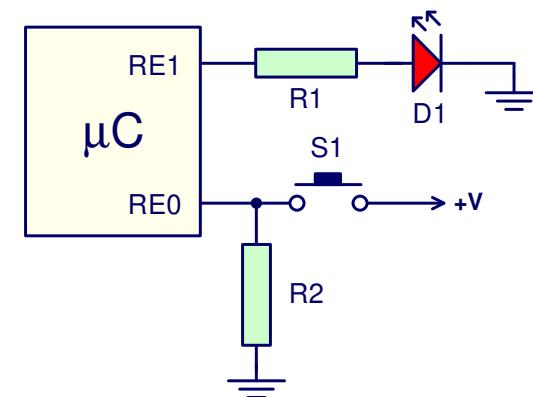
- Exemplo 2:

- Gerar no porto RD0 um sinal de 1 Hz com *duty-cycle* de 10% (i.e. RD0=1 durante 0.1s, RD0=0 durante 0.9s)



- Exemplo 3:

- Manter o LED D1 apagado enquanto o *switch* S1 estiver premido, e aceso na situação contrária; LED D1 ligado ao porto RE1 e *switch* ligado ao porto RE0



# Exemplo de configuração/utilização dos portos

- Definição dos endereços dos portos:

```
.equ ADDR_BASE_HI, 0xBF88

.equ TRISB, 0x6040      # TRISB address: 0xBF886040
.equ PORTB, 0x6050      # PORTB address: 0xBF886050
.equ LATB, 0x6060        # LATB address: 0xBF886060

.equ TRISD, 0x60C0      # TRISD address: 0xBF8860C0
.equ PORTD, 0x60D0      # PORTD address: 0xBF8860D0
.equ LATD, 0x60E0        # LATD address: 0xBF8860E0

.equ TRISE, 0x6100      # TRISE address: 0xBF886100
.equ PORTE, 0x6110      # PORTE address: 0xBF886110
.equ LATE, 0x6120        # LATE address: 0xBF886120

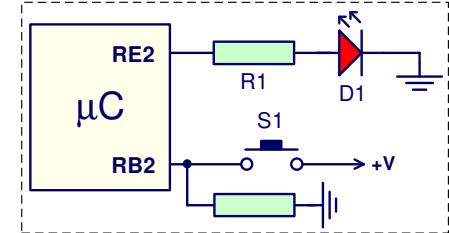
.data

.text
.globl main
```

# Exemplo de configuração/utilização dos portos

- Exemplo 1: Ler o valor do porto de entrada (RB2) e escrever esse valor no porto de saída (RE2)

```
.text
.globl main
main: lui $t0,ADDR_BASE_HI # $t0=0xBF880000
      lw  $t1,TRISB($t0)    # Address: BF880000 + 00006040
      ori $t1,$t1,0x0004    # bit2 = 1 (IN)
      sw  $t1,TRISB($t0)    # RB2 configured as IN
      lw  $t1,TRISE($t0)    # Read TRISE register
      andi $t1,$t1,0xFFFFB   # bit2 = 0 (OUT)
      sw  $t1,TRISE($t0)    # RE2 configured as OUT
loop: lw   $t1,PORTB($t0)    # Read PORTB register
      andi $t1,$t1,0x0004    # Reset all bits except bit 2
      lw   $t2,LATE($t0)     # Read LATE register
      andi $t2,$t2,0xFFFFB   # Reset bit 2
      or   $t2,$t2,$t1        # Merge data
      sw  $t2,LATE($t0)     # Write LATE register
      j    loop
```



# Exemplo de configuração/utilização dos portos

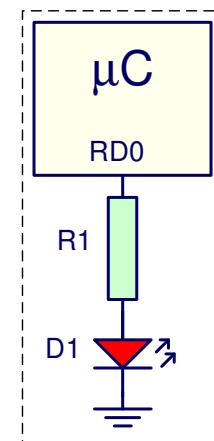
- Exemplo 2: gerar no bit 0 do porto D (RD0) um sinal de 1 Hz com *duty-cycle* de 10% (i.e. RD0=1 durante 0.1s, RD0=0 durante 0.9s)

```
.text
.globl main
main: lui $t0,ADDR_BASE_HI # 16 MSbits of port addresses
      lw   $t1,TRISD($t0)    # Read TRISD register
      andi $t1,$t1,0xFFFF    # Modify bit 0 (0 is OUT)
      sw   $t1,TRISD($t0)    # Write TRISD (port configured)

loop: lw   $t1,LATD($t0)    # Read LATD
      ori  $t1,$t1,0x0001    # Modify bit 0 (set)
      sw   $t1,LATD($t0)    # Write LATD

# wait 100 ms (e.g., using MIPS core timer)
      lw   $t1,LATD($t0)    # Read LATD
      andi $t1,$t1,0xFFFF    # Modify bit 0 (reset)
      sw   $t1,LATD($t0)    # Write LATD

# wait 900 ms (e.g., using MIPS core timer)
      j    loop
```



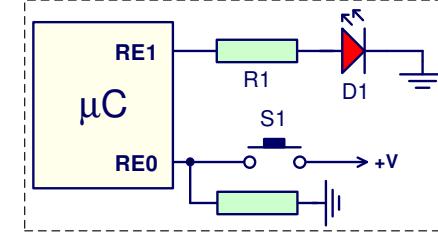
# Exemplo de configuração/utilização dos portos

- Exemplo 3: em ciclo infinito, ler o valor do porto de entrada (RE0) e escrever esse valor, negado, no porto de saída (RE1)

```

.text
.globl main
main: lui $t0,ADDR_BASE_HI # 16 MSbits of port addresses
      lw  $t1,TRISE($t0)    # Read TRISE register
      ori $t1,$t1,0x0001    # bit0 = 1 (IN)
      andi $t1,$t1,0xFFFFD # bit1 = 0 (OUT)
      sw  $t1,TRISE($t0)    # TRISE configured
loop: lw   $t1,PORTE($t0)    # Read PORTE register
      andi $t1,$t1,0x0001    # Reset all bits except bit 0
      xori $t1,$t1,0x0001    # Negate bit 0
      sll $t1,$t1,1          #
      lw   $t2,LATE($t0)     # Read LATE register
      andi $t2,$t2,0xFFFFD    # Reset bit 1
      or   $t2,$t2,$t1        # Merge data
      sw  $t2,LATE($t0)      # Write LATE register
      j   loop

```



# Programação de portos I/O - exercício

1. Pretende usar-se o porto RB do microcontrolador PIC32MX795F512H para realizar a seguinte função (em ciclo fechado):

O byte menos significativo ligado a este porto é lido com uma periodicidade de 100ms. Com um atraso de 10ms, o valor lido no byte menos significativo é colocado, em complemento para 1, no byte mais significativo desse mesmo porto. Escreva, em *assembly* do MIPS, um programa que execute esta tarefa.

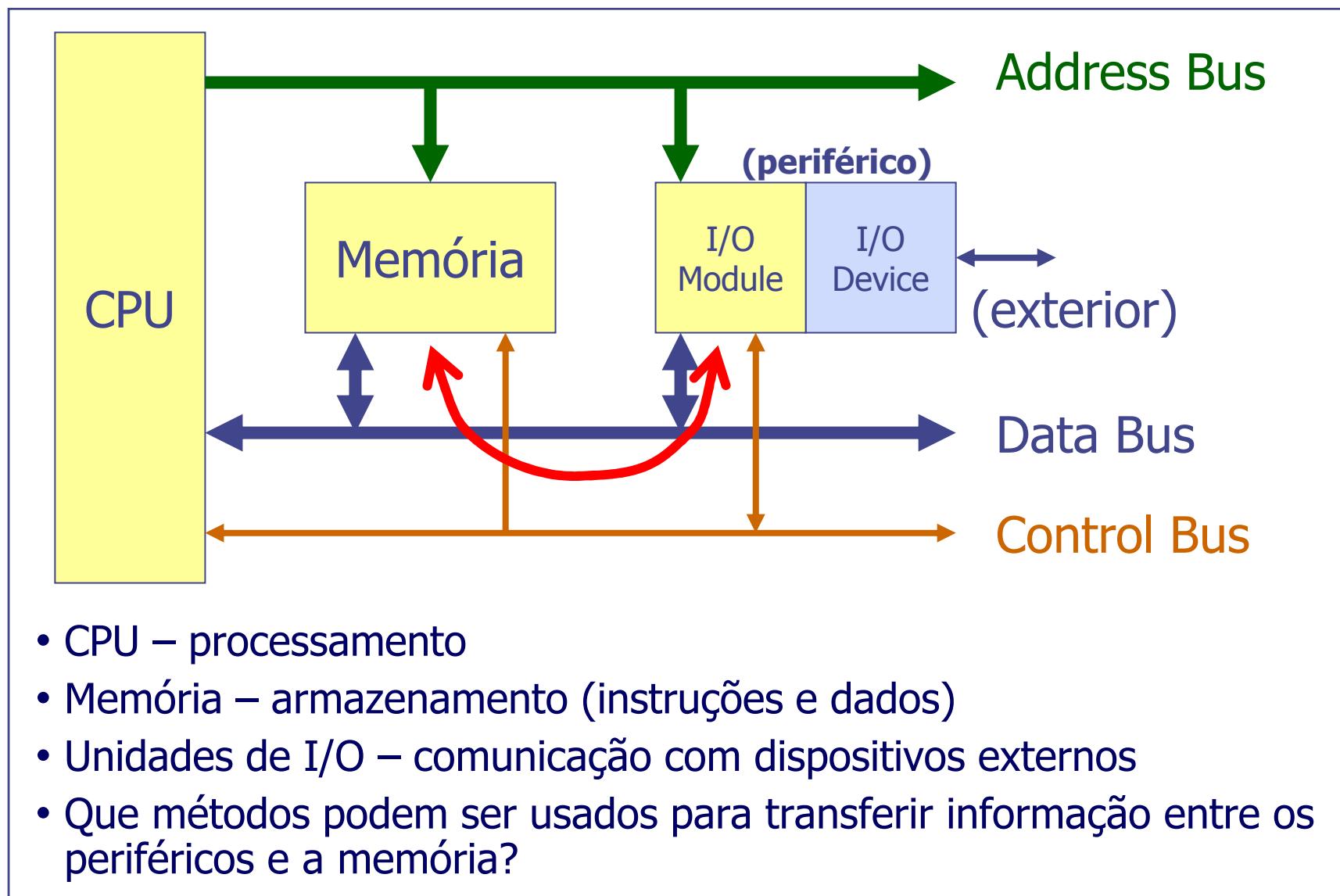
- a) configure o porto RB para executar corretamente a tarefa descrita
- b) efetue a leitura do porto indicado
- c) execute um ciclo de espera de 10ms
- d) efetue a transformação da informação lida para preparar o processo de escrita naquela porto
- e) efetue, no byte mais significativo, o valor resultante da operação anterior
- f) execute um ciclo de espera de 90ms
- g) regresse ao ponto b)

## 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

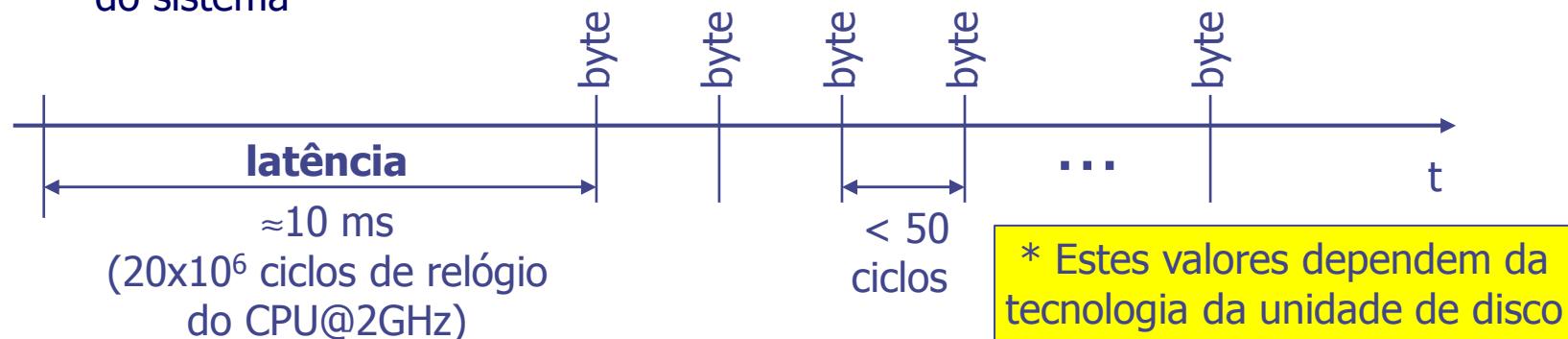
José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

# 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
  1. **efetuar o pedido** à unidade de disco (por exemplo sector do disco e quantidade de informação pretendida)
  2. **esperar** que a unidade de **disco tenha a informação disponível** na sua memória interna (informação fornecida por 1 bit de um registo de status)
  3. **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)

```
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 )
```

*polling* {

- 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

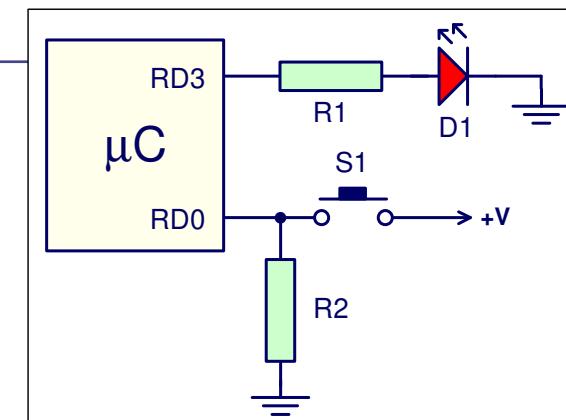
# E/S Programada (exemplo para o PIC32)

- 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,ADDR_BASE_HI    #
lw     $t1,TRISD($t0)      #
ori    $t1,$t1,0x0001 # RD0=1
andi   $t1,$t1,0xFFFF7 # 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;
}

{ polling
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

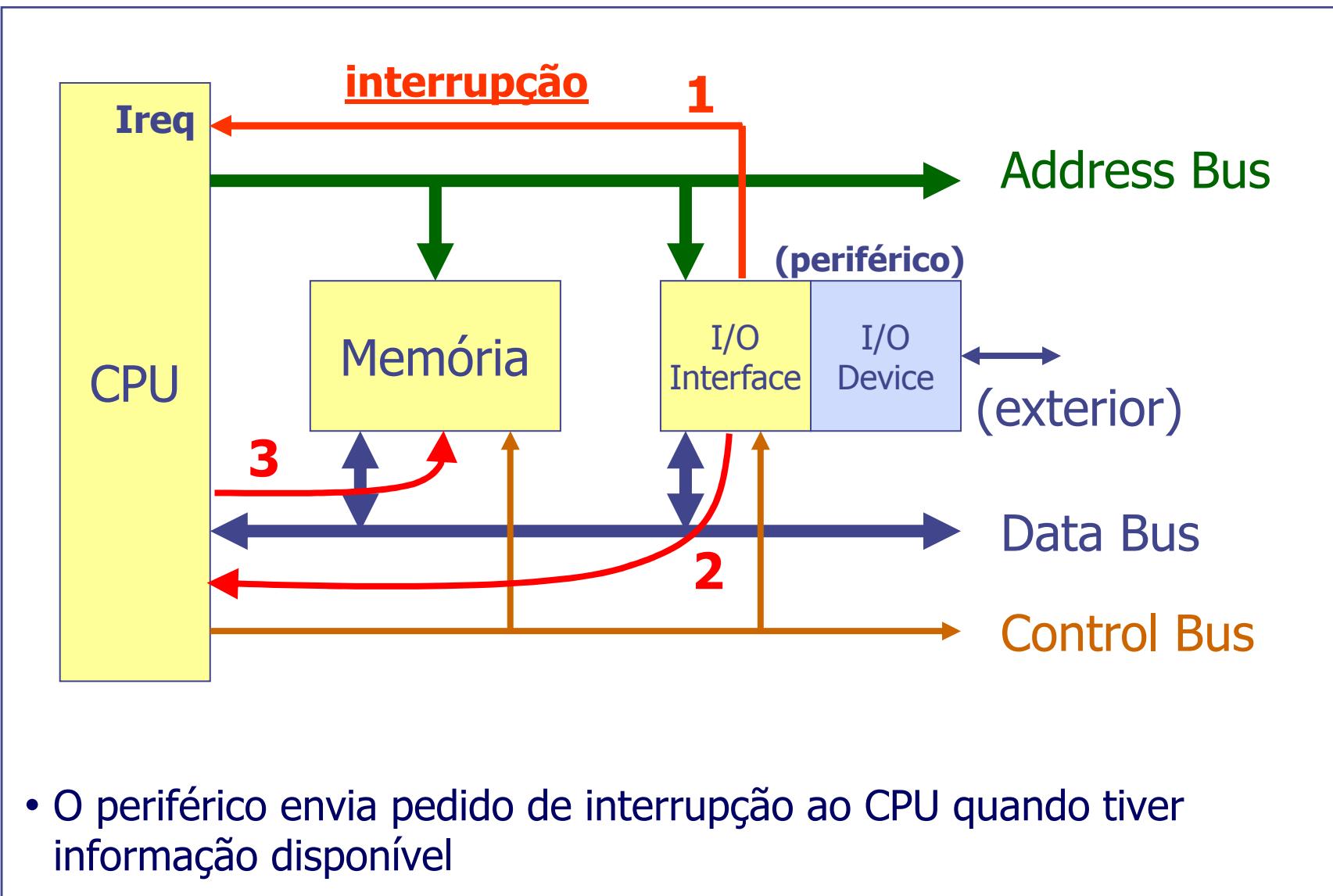
# E/S por interrupção (exemplo de leitura)

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

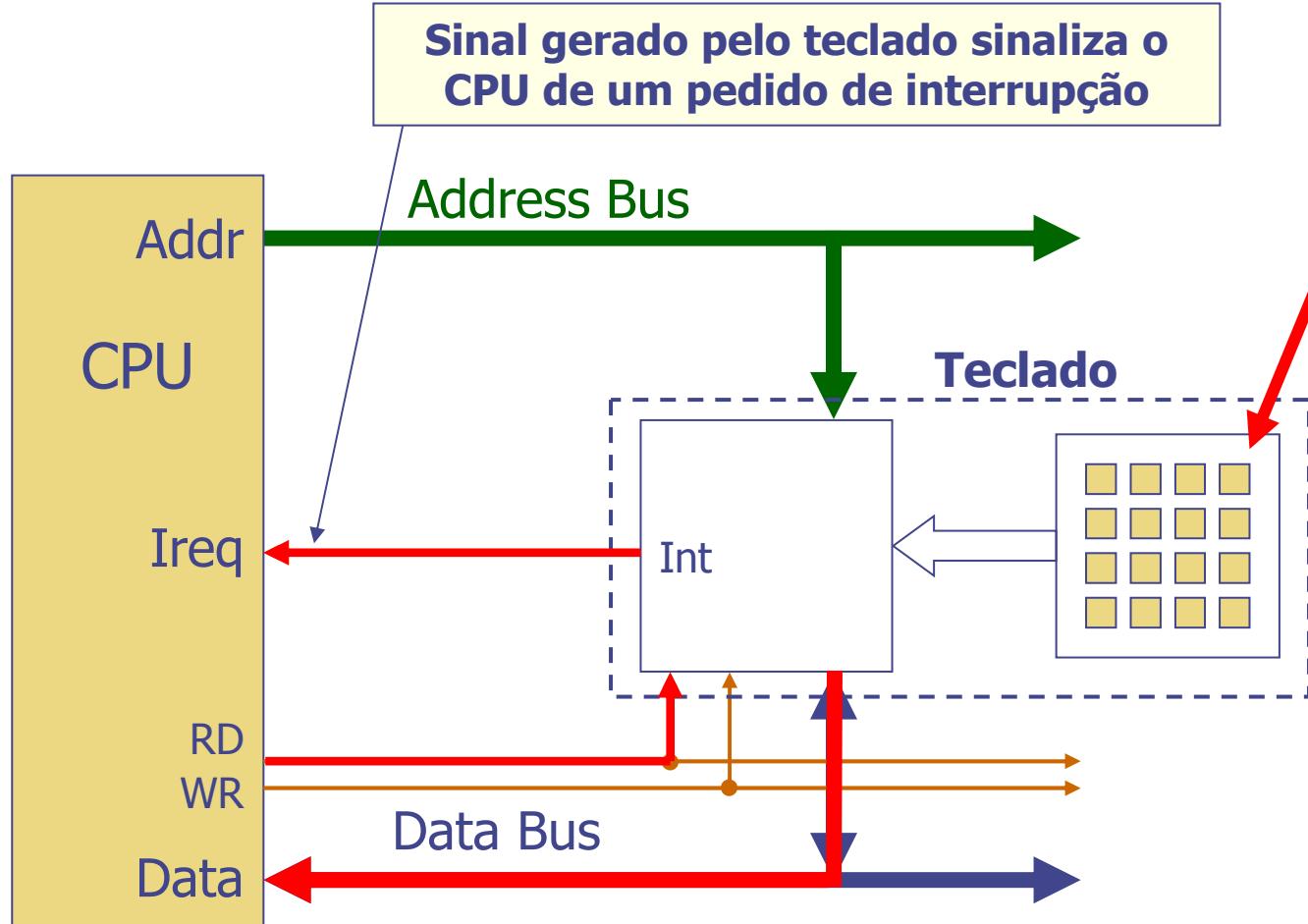
```
void interrupt isr(void)
{
    Read byte from I/O Module
    Write byte into Memory
    bytesReceived++
}
```

- **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 reposto 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)



- Ireq (*Interrupt request*): entrada de interrupção do CPU

# 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,ADDR_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

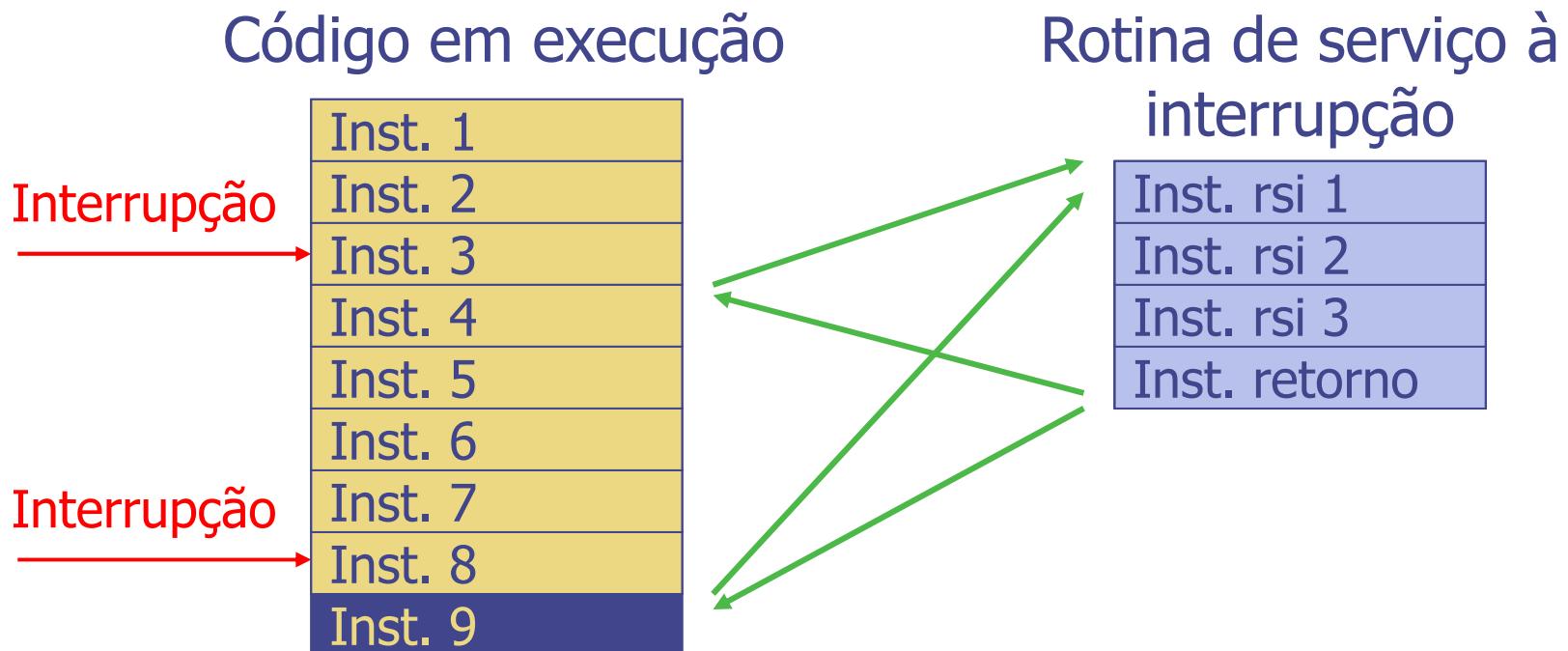
- Exemplos de dispositivos que podem gerar interrupções: teclado, rato, timers, dispositivos de comunicação, dispositivos de armazenamento, ...
- Exemplos de exceções:
  - Divisão por zero
  - *Overflow* numa operação aritmética
  - Tentativa de execução de uma instrução cujo OpCode é desconhecido
  - 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

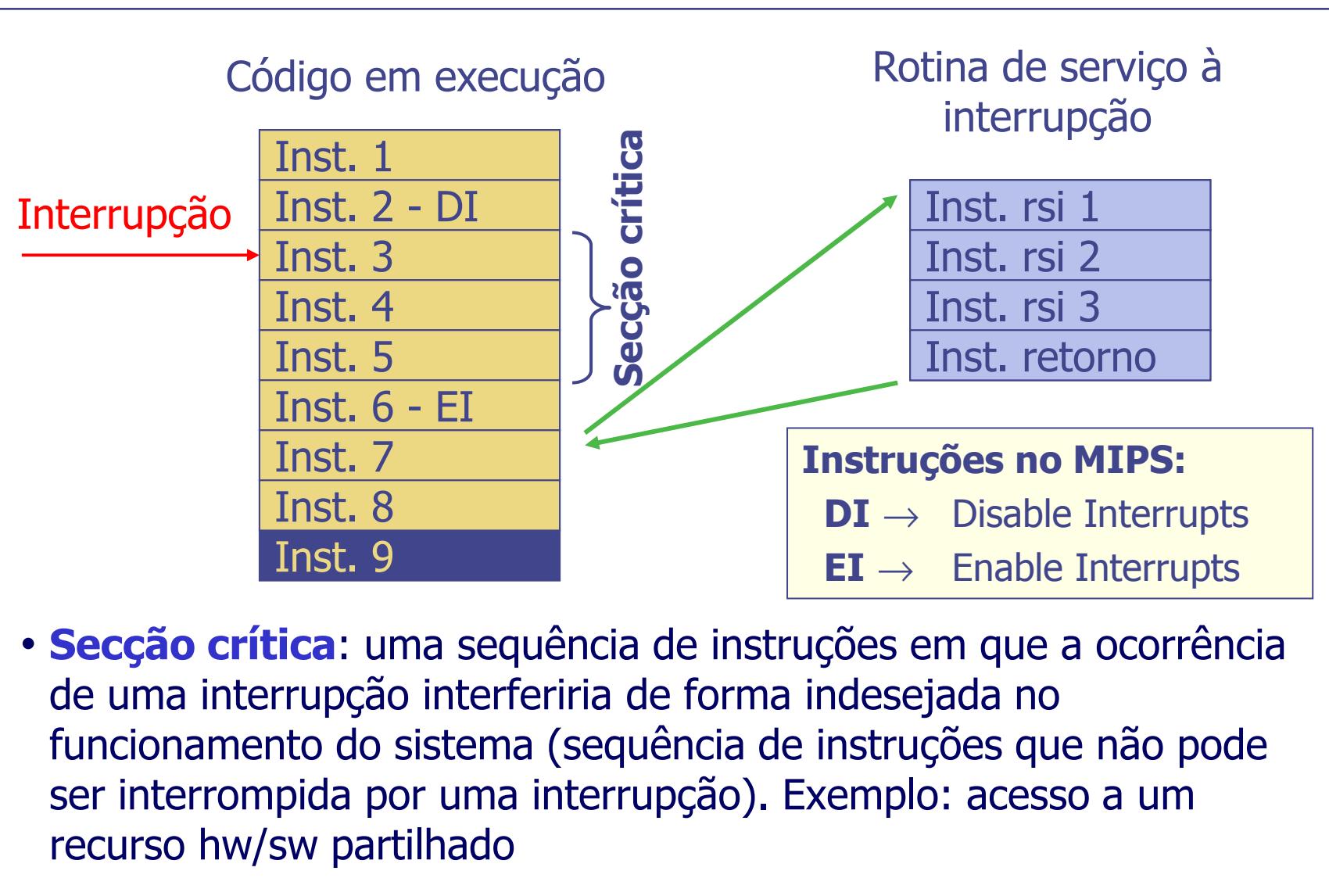
- **Exceções:** a instrução que gerou a exceção não termina, sendo a execução passada de imediato para a rotina de tratamento da exceção
- **Interrupções:** a passagem da execução para a rotina de tratamento da interrupção só acontece quando for concluída a instrução que está a ser executada no momento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

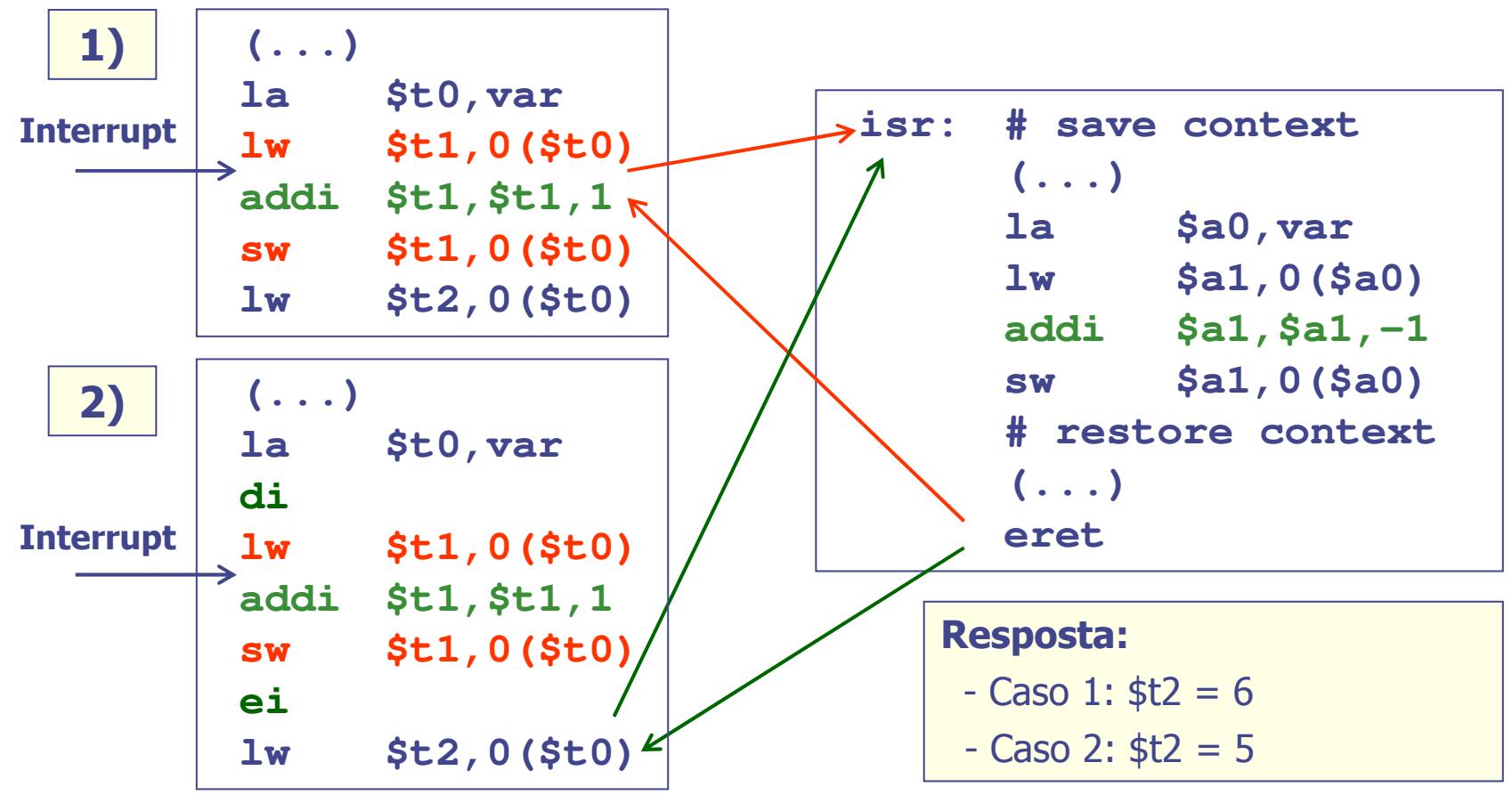


# Ativação/desativação global das interrupções



# 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 ( $PC \leftarrow$  Endereço da RSI, i.e., salto para a 1<sup>a</sup> 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)

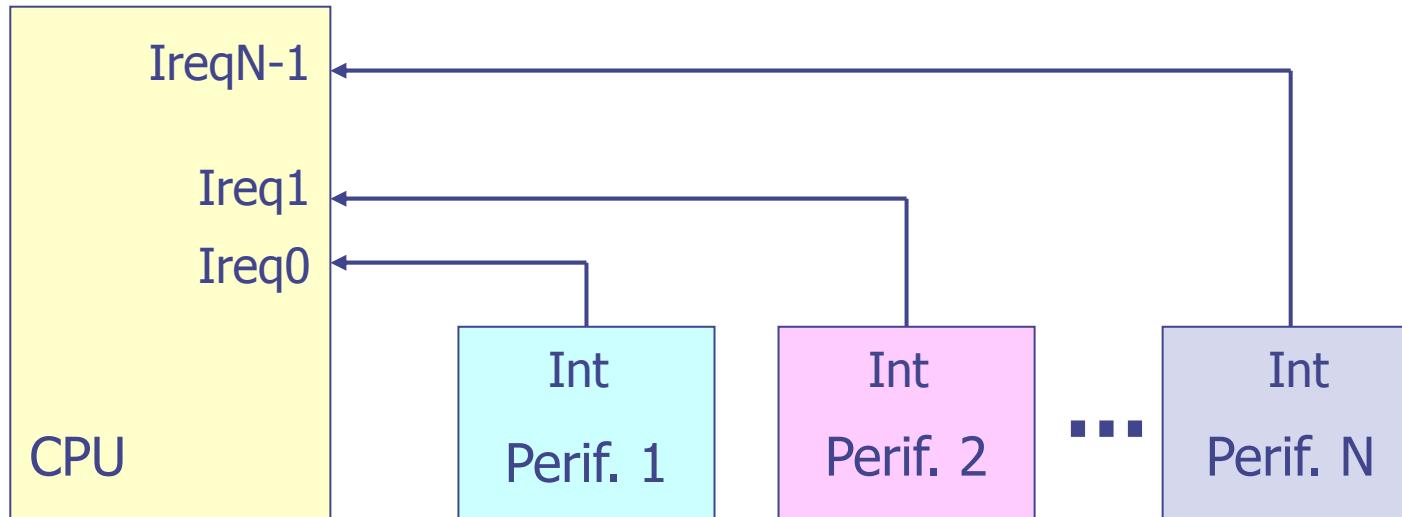
## *Overhead* do método de transferência por interrupção

- 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

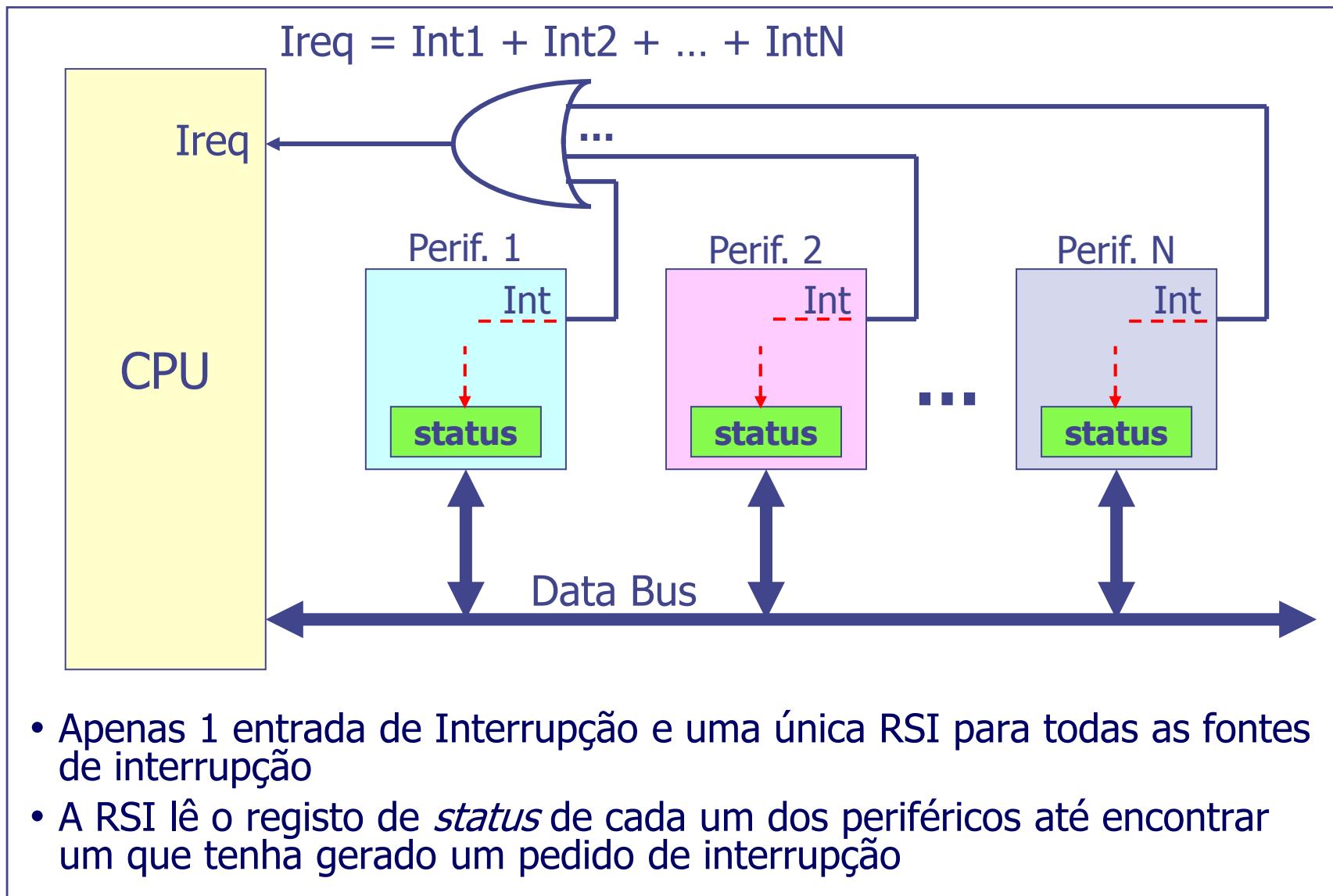
- Num sistema real é expectável que vários periféricos possam ter a capacidade de gerar interrupções
- Como organizar o sistema de interrupções para permitir a ligação de vários periféricos?
  - **Múltiplas linhas de interrupção**
  - **Identificação da fonte de interrupção por software**
  - **Interrupções 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

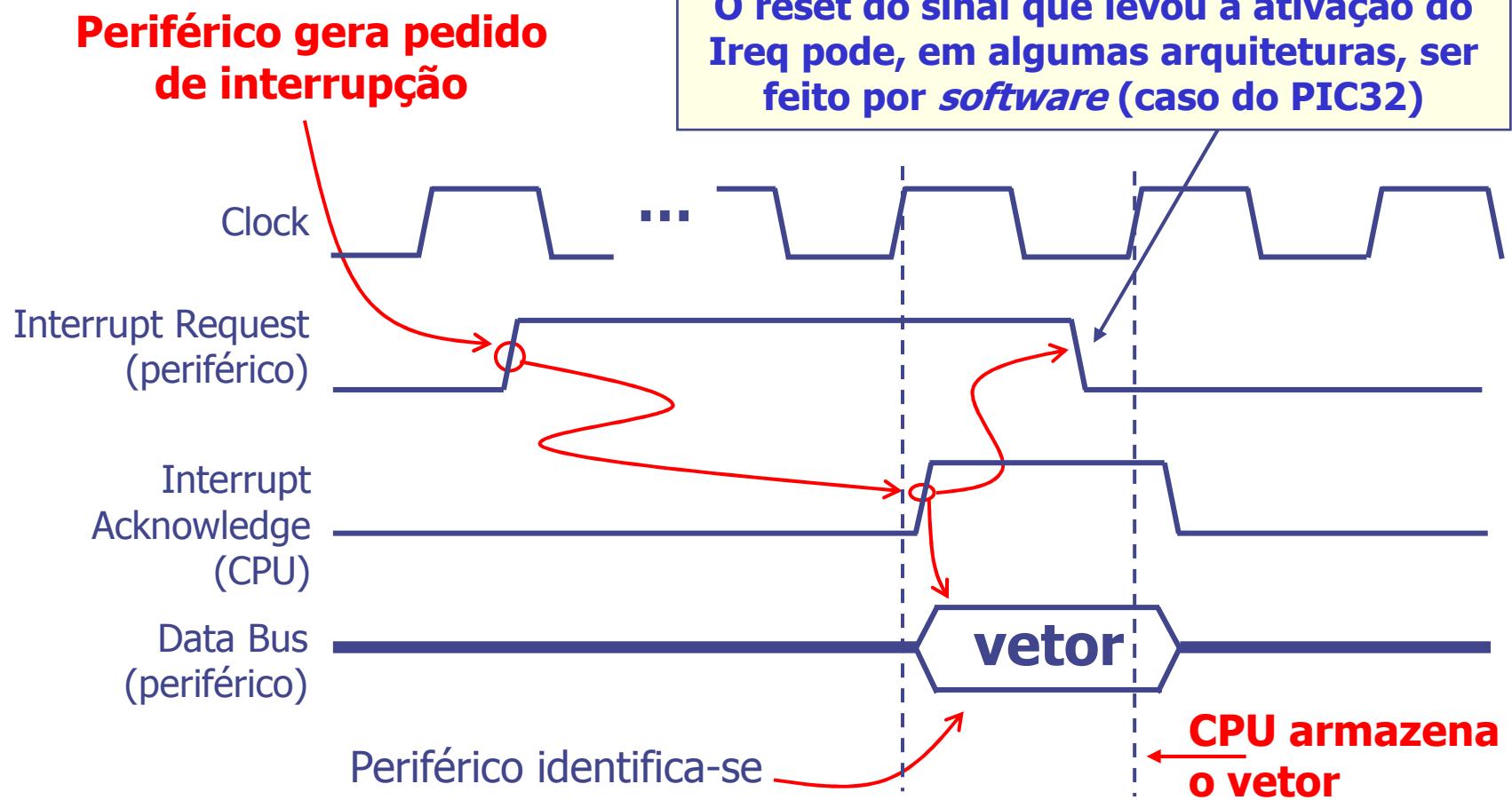
No caso de pedidos de interrupção simultâneos, a ordem pela qual os periféricos são "questionados" determina a prioridade no atendimento

# 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

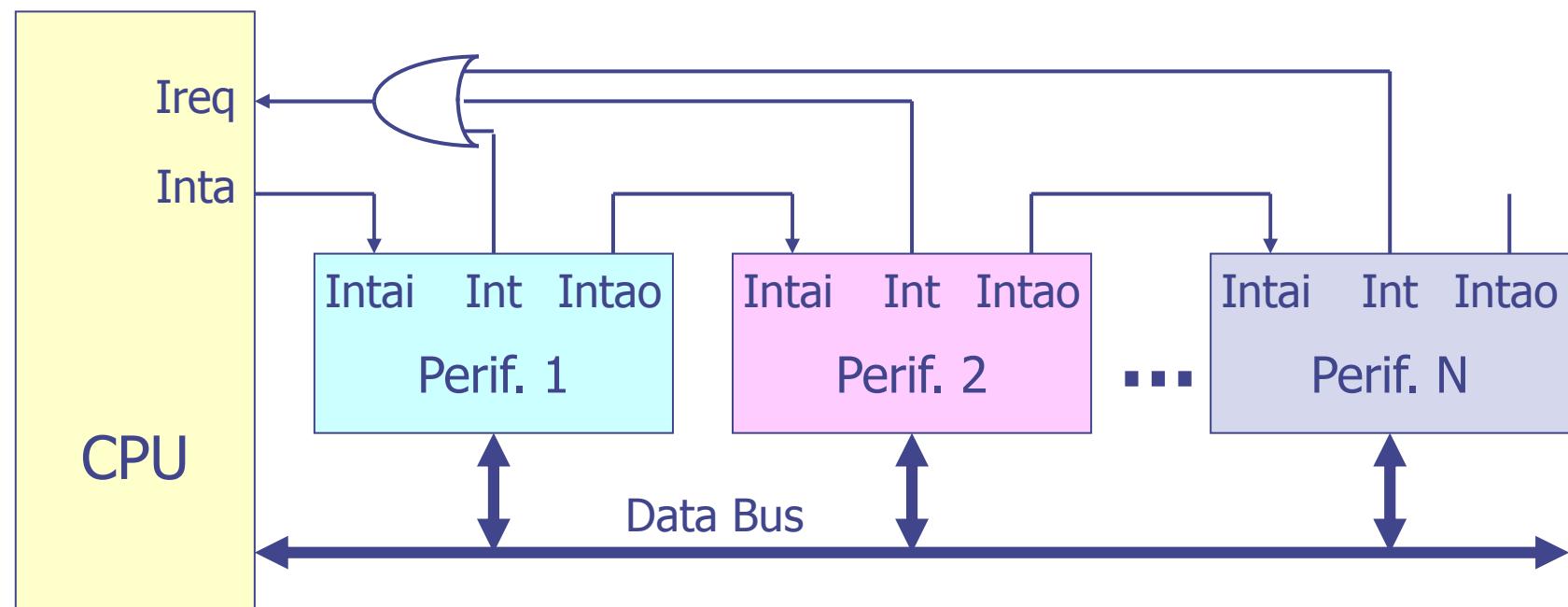
# Interrupções vetorizadas

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



# Interrupções vetorizadas – *daisy chain*

- Periféricos podem estar organizados numa estrutura *daisy chain*



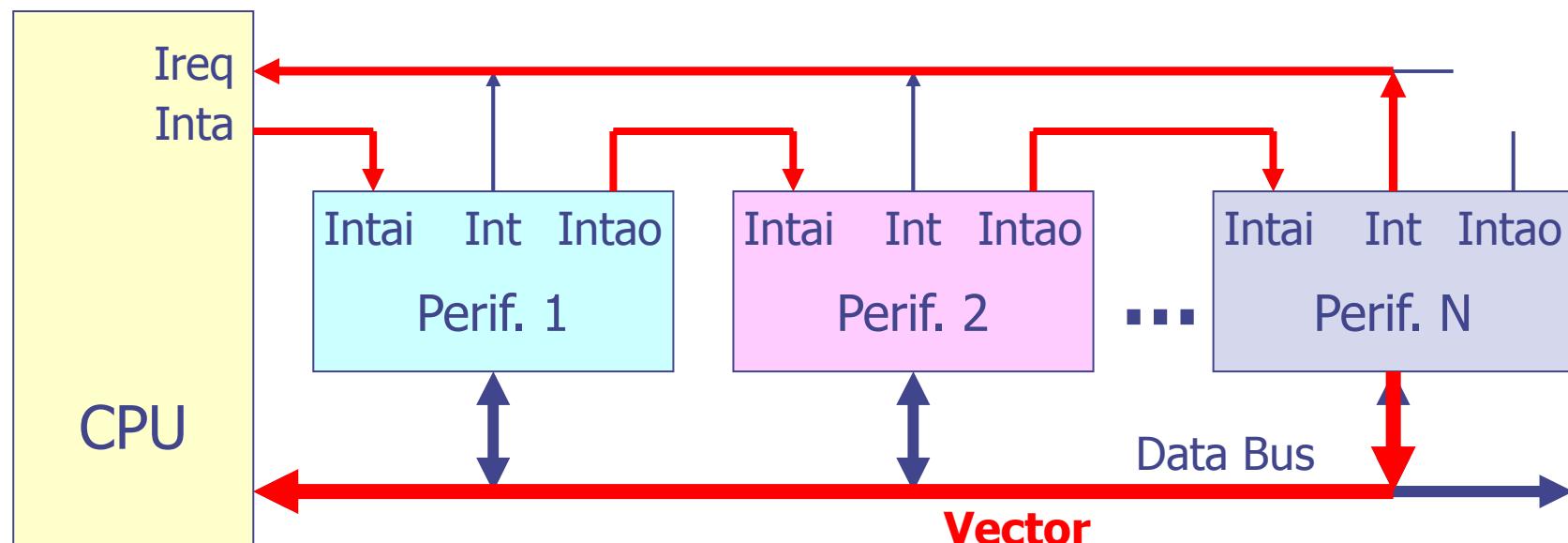
**Intai/o - Interrupt Acknowledge in/out**

# 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*

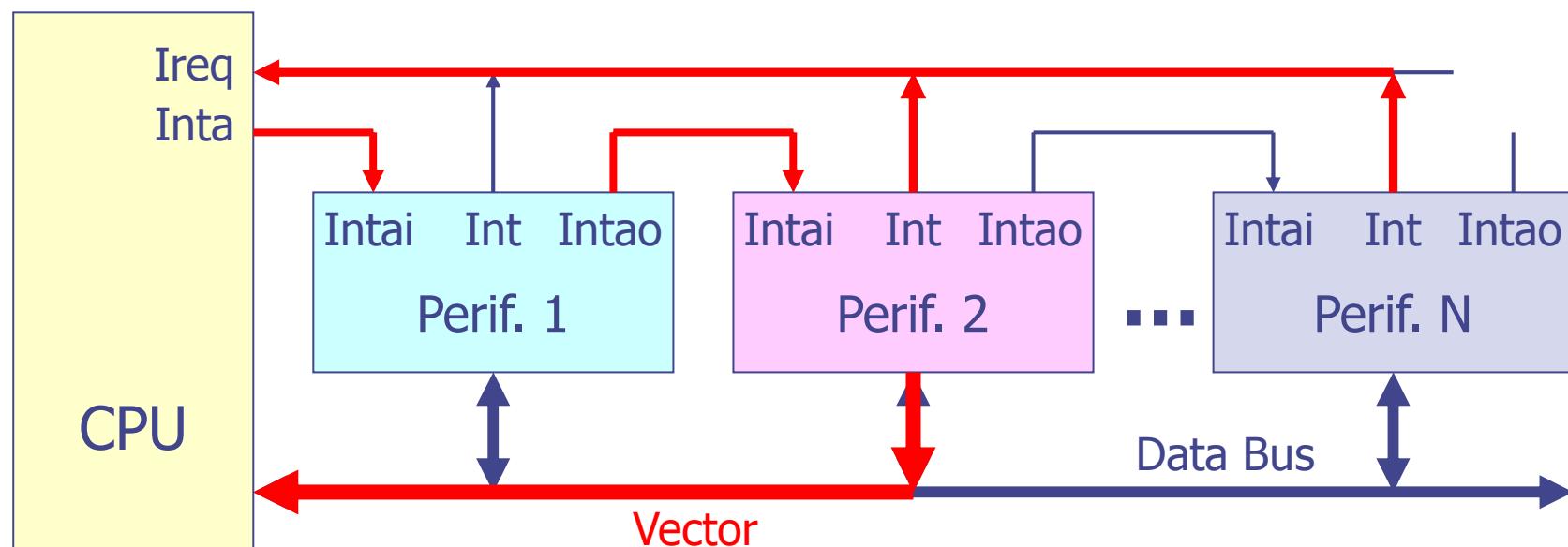
- Exemplo de identificação da fonte de interrupção



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

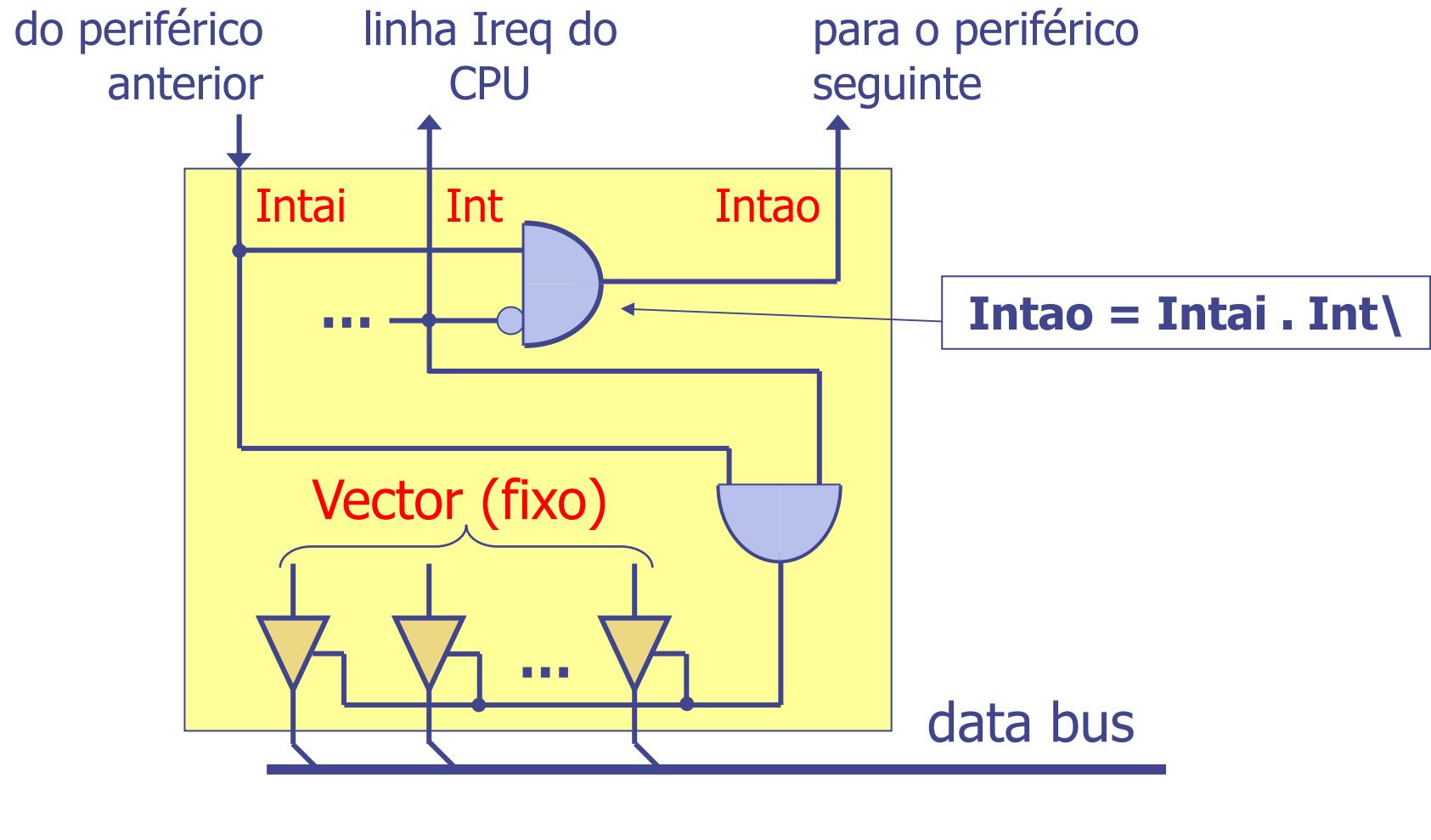


$$Ireq = Int_1 + Int_2 + \dots + Int_N$$

- 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)

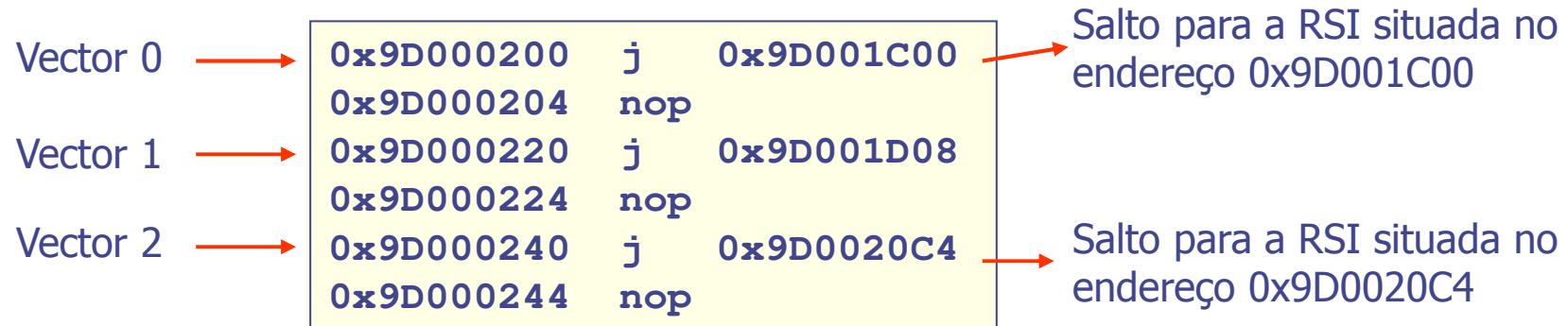


# Interrupções vetorizadas – tabela de vetores

- 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:



# Aula 8

- O sistema de interrupções do PIC32

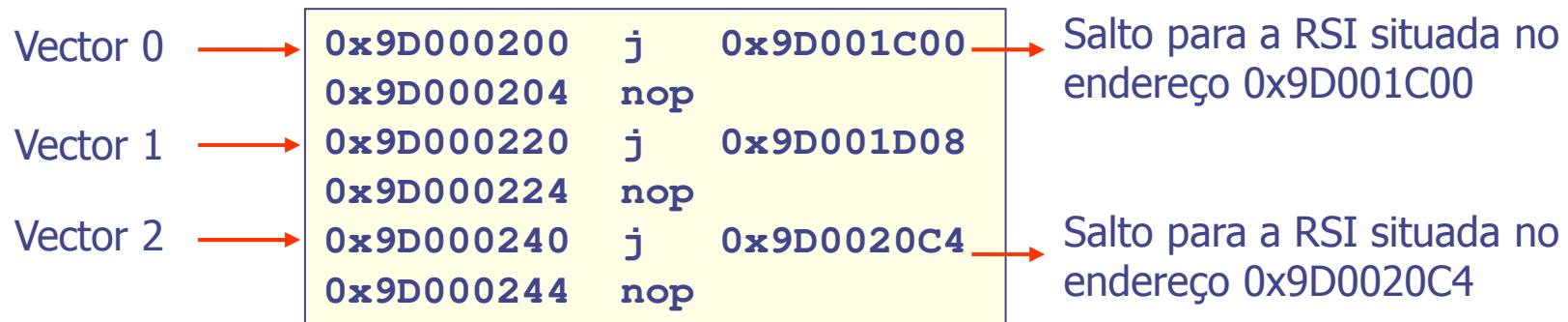
José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

# 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)

# Tabela de vetores (multi-vector mode)

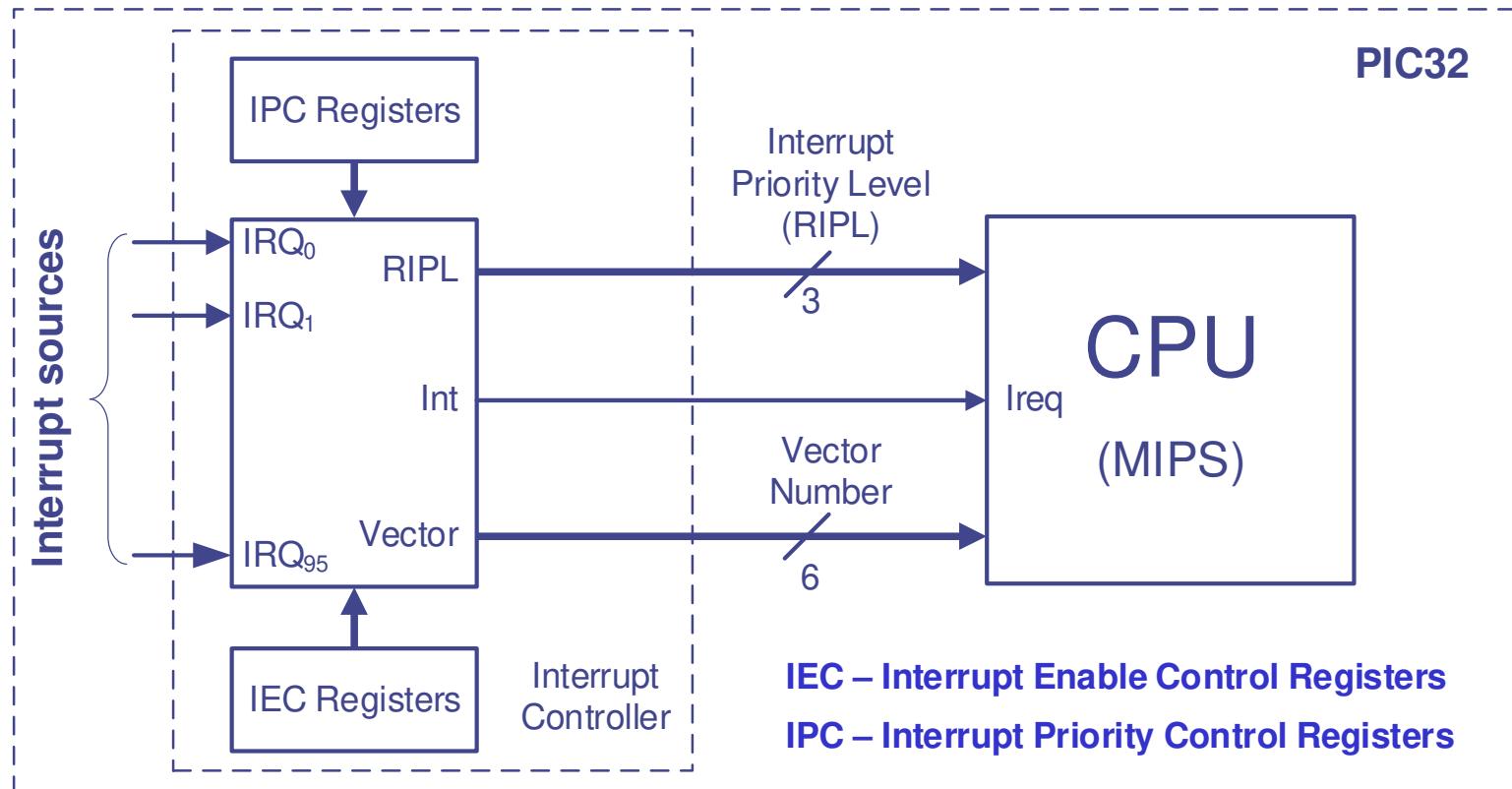
- **Tabela inicializada com instruções de salto para as RSI:** são colocadas na tabela de vetores, instruções de salto para as RSI
- No processamento da interrupção o CPU usa o vetor para calcular o endereço da tabela de vetores e faz um salto para esse endereço (ou seja, PC <= endereço calculado)
- Cada posição da tabela tem, em geral, uma instrução de salto incondicional para a RSI a executar
- O espaçamento entre elementos da tabela pode ser configurado para 32, 64, 128, 256 ou 512 bytes (valor por defeito: 32 = 0x20)



- **Exemplo:** vetor=2, base\_address=0x9D000200, espaçamento=32  
 $\text{endereço\_tabela} = \text{vetor} * 0x20 + 0x9D000200 = 0x9D000240$

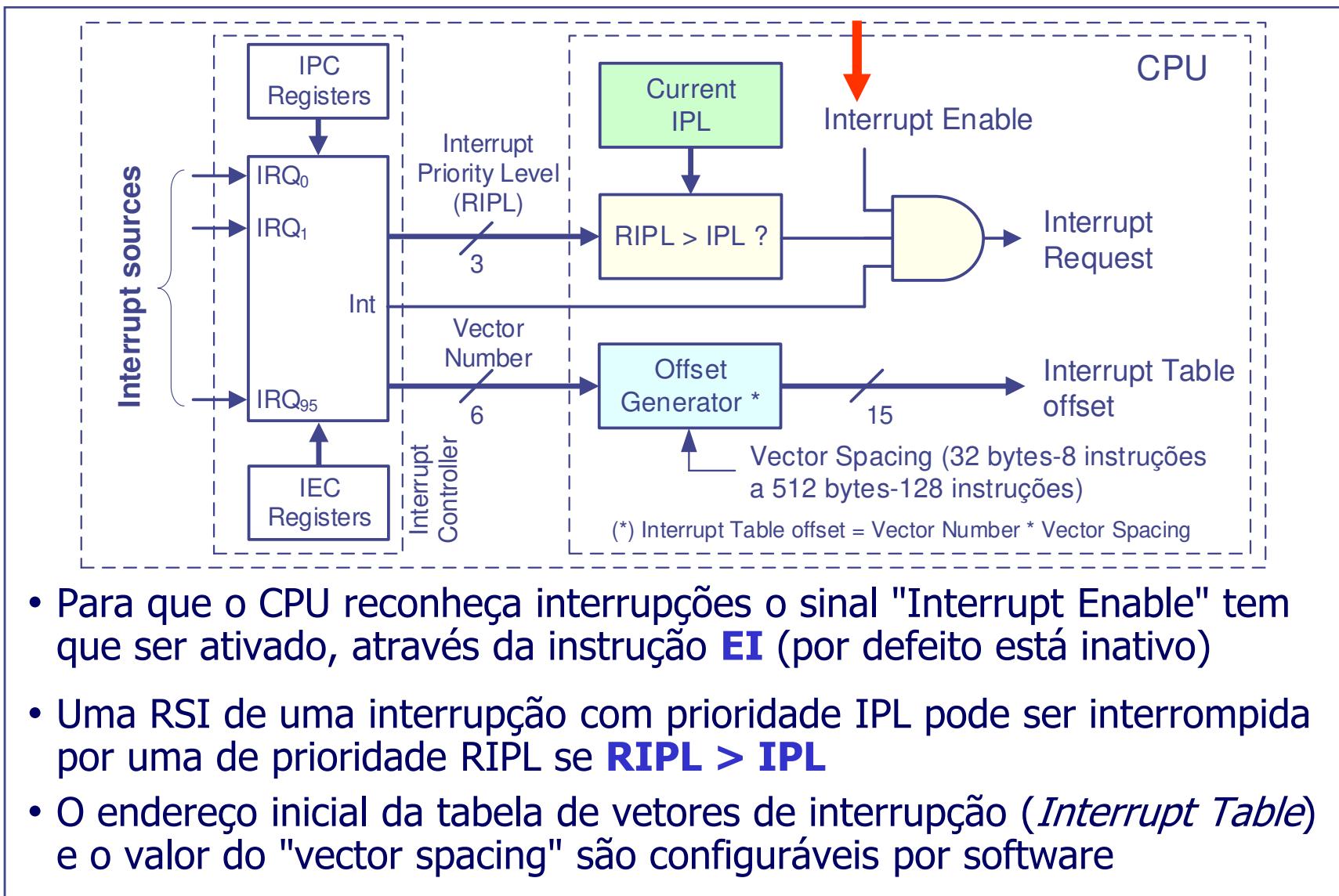
# Interrupções no PIC32

- 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



- Para que o CPU reconheça interrupções o sinal "Interrupt Enable" tem que ser ativado, através da instrução **EI** (por defeito está inativo)
- Uma RSI de uma interrupção com prioridade IPL pode ser interrompida por uma de prioridade RIPL se **RIPL > IPL**
- O endereço inicial da tabela de vetores de interrupção (*Interrupt Table*) e o valor do "vector spacing" são configuráveis por software

# Interrupções no PIC32

- **IEC0, IEC1, IEC2** – Interrupt Enable **Control Registers**
  - Registros 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**
  - Registros 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 **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 ativa 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 num dos registos **I<sub>E</sub>C<sub>x</sub>** (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*)
- **Priority Level** – conjunto de 3 bits definido num dos registos **I<sub>P</sub>C<sub>x</sub>** (Interrupt Priority Control Registers), designado através da sigla da fonte com o sufixo **IP** (e.g. **T1IP**, *Timer1 Interrupt Priority*)
  - 7 níveis de prioridade (1 a 7); a prioridade 0 significa fonte *disabled*
- **Interrupt Flag** – bit definido num dos registos **I<sub>F</sub>S<sub>x</sub>** (Interrupt Flag Status Registers) e designado com a sigla da fonte 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

# Exemplo de uma tabela de vetores no PIC32

```
#vector_7 (INT1, External Interrupt 1)  
  
0x9D0002E0      0xB40074D    j      0x9D001D34  
0x9D0002E4      0x00000000    nop
```

```
#vector_8 (T2 - Timer2)  
  
0x9D000300      0xB4006C3    j      0x9D001B0C  
0x9D000304      0x00000000    nop
```

```
#vector_19 (INT4 - External Interrupt 4)  
  
0x9D000460      0xB40077A    j      0x9D001DE8  
0x9D000464      0x00000000    nop
```

- Na placa DETPIC32 o endereço inicial da tabela de vetores (a que corresponde o vetor 0) é 0x9D000200.

# Rotina de Serviço à Interrupção no PIC32

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

Interrupt Source <sup>(1)</sup>	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>

Fonte

```

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

```

Interrupt Function

Vector Number

Function Name

TABLE 7-1: [PIC32MX7XX Family Data Sheet#page 74 - 76]

# Exemplo de programação com interrupções no PIC32

```
int main(void)
{
    configIO();          // Config IO and Interrupt
                        // Controller
    EnableInterrupts(); // Enable Interrupt System. Macro
                        // definida em detpic32.h como:
                        //     asm volatile("ei")

    while(1)
    {
        ...
    }
    return 0;
}

// IO Configuration function
void configIO(void)
{
    ...
    IFS0bits.T1IF = 0;    // Reset Timer 1 interrupt flag
    IPC1bits.T1IP = 2;    // Set priority level to 2
    IEC0bits.T1IE = 1;    // Enable Timer 1 interrupts
    ...
}
```

# Exemplo de programação com interrupções no PIC32

```
// Interrupt Service routine - Timer1
void __int_( 4 ) isr_t1(void)
{
    ...
    IFS0bits.T1IF = 0;    // Reset Timer 1 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
}
```

# Aula 9

- Transferência de informação por DMA
  - Configuração hardware
  - Passos de uma transferência
- Modos de funcionamento
  - Modo "bloco"
  - Modo "burst"
  - Modo "cycle-stealing"
- Configurações

José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

# Introdução

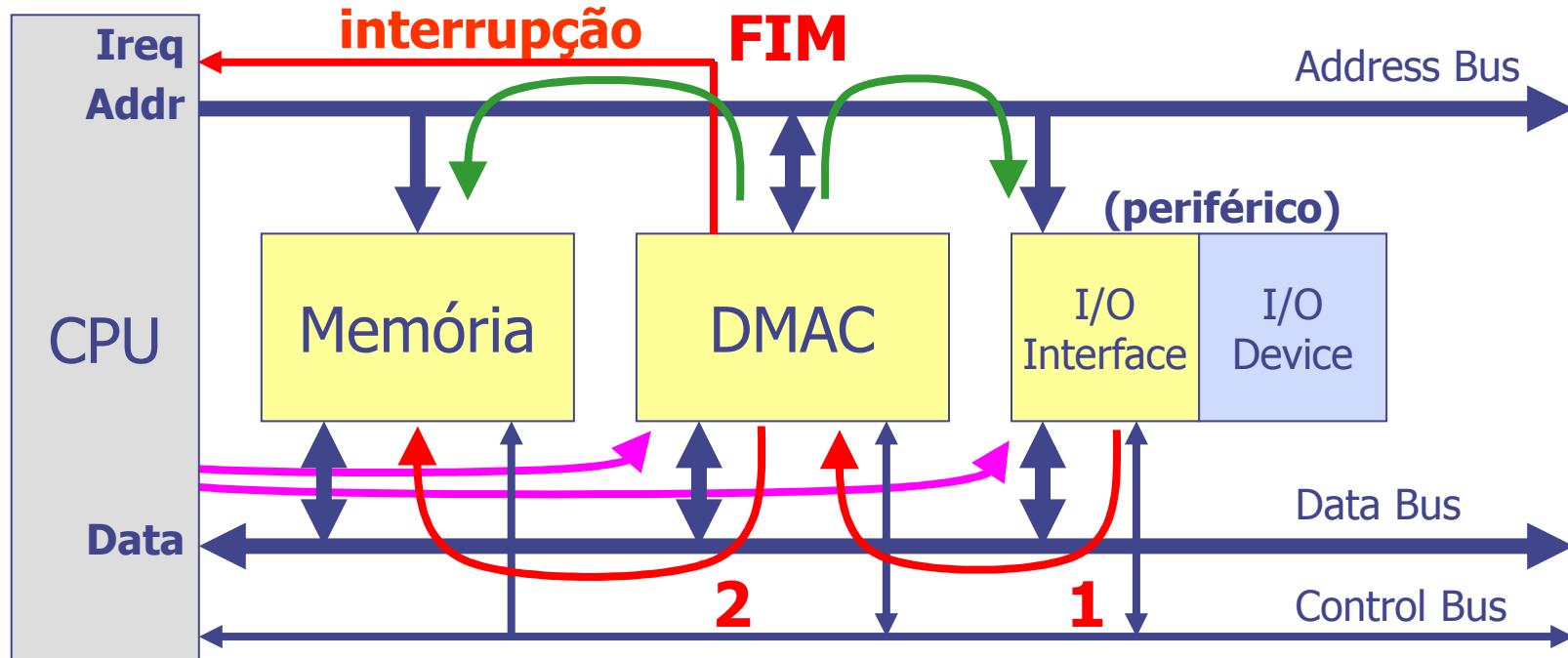
- O problema da (possível) longa latência apresentada pelos periféricos é adequadamente tratada pela técnica de E/S por interrupção
- Esta técnica não resolve, contudo, a questão da transferência a taxas elevadas, uma vez que o limite é sempre imposto pelo facto de o CPU ter que executar um programa para efetuar a transferência
- Por exemplo, transferir informação de um periférico para a memória implica, do ponto de vista temporal:
  - O tempo de latência de resposta à interrupção + tempo de execução do prólogo da RSI
  - O tempo para executar instruções de: leitura de uma *word* do módulo de E/S do periférico, de escrita dessa *word* no endereço destino da memória e de atualização dos endereços origem e destino
  - Para cada iteração, o tempo necessário para verificar se todos os dados foram já transferidos (envolve a leitura de um ou mais registo do módulo de E/S do periférico e a verificação dos bits de status necessários)

# Introdução

- Acresce ainda o facto de que, durante o período de tempo em que o CPU está a realizar esta transferência, se encontra completamente ocupado a realizar esta tarefa, diminuindo assim a sua eficiência global na realização de outras tarefas
- A solução para mitigar o problema identificado é designada por **DMA** (do inglês *Direct Memory Access*) e consiste na transferência de informação do periférico diretamente para a memória (ou o contrário), sem intervenção do processador
- **Exercício 1:** um programa para transferir dados de 32 bits de um periférico para a memória é implementado com um ciclo com 10 instruções. Admitindo que o CPU funciona a 100 MHz e que o programa em causa apresenta um CPI de 2, determine a taxa de transferência máxima, em Bytes/s, que se consegue obter (suponha um barramento de dados de 32 bits)
- **Exercício 2:** admita que, de uma forma não realista, o programa do exercício anterior era constituído por apenas 2 instruções, e o CPI era 1. Qual seria nesse caso a taxa de transferência?

# Transferência por Acesso Direto à Memória (DMA)

- Transferência de dados entre a memória e um periférico sem a intervenção do CPU. Um periférico especializado (DMAC – DMA Controller) efetua essa transferência



- Quando o controlador de DMA termina a transferência de todas as palavras, gera uma interrupção
- A transferência é **exclusivamente feita por hardware** pelo controlador de DMA, i.e., não é executado qualquer programa

# Controlador de DMA (DMAC)

- Um controlador de DMA é um periférico que, do ponto de vista do modelo de programação, é semelhante a qualquer outro periférico
- Disponibiliza um conjunto de registos internos a que o CPU acede para configurar uma transferência de dados, nomeadamente:
  - Endereço origem da informação (endereço de leitura)
  - Endereço destino da informação (endereço de escrita)
  - Quantidade de informação a transferir (nº de bytes/words)
- Pode também ter registos com informação sobre o estado de uma transferência
- Do ponto de vista da operação realizada, o controlador de DMA é um periférico especializado na transferência de dados (cópia), de forma autónoma, em hardware, após programação feita pelo CPU
- Durante a transferência, o controlador de DMA tem a capacidade de controlar os barramentos de endereços, dados e controlo, como se fosse um CPU

# Controlador de DMA (DMAC)

- Para efetuar uma transferência de um bloco de dados de **n** palavras, o controlador de **DMA** realiza, no essencial, ao nível dos barramentos, as mesmas operações que seriam realizadas por um programa executado pelo CPU:
  - **Lê** uma palavra (*byte ou word*) do dispositivo fonte (do **source address**) para um registo interno – "**fetch**"
  - **Escreve** a palavra guardada no registo interno no passo anterior, no dispositivo destino (no **destination address**) – "**deposit**"
  - Incrementa *source address* e *destination address*
  - Incrementa o **número de palavras transferidas**
  - Se não transferiu a totalidade do bloco, repete desde o início
- Para realizar estas tarefas o DMAC necessita de **controlar os barramentos** de **endereços** e de **dados** e ainda os sinais **rd** e **wr**
- Note-se que a capacidade do DMAC para gerir os barramentos do sistema entra em conflito com capacidade idêntica do CPU

# Controlador de DMA (DMAC)

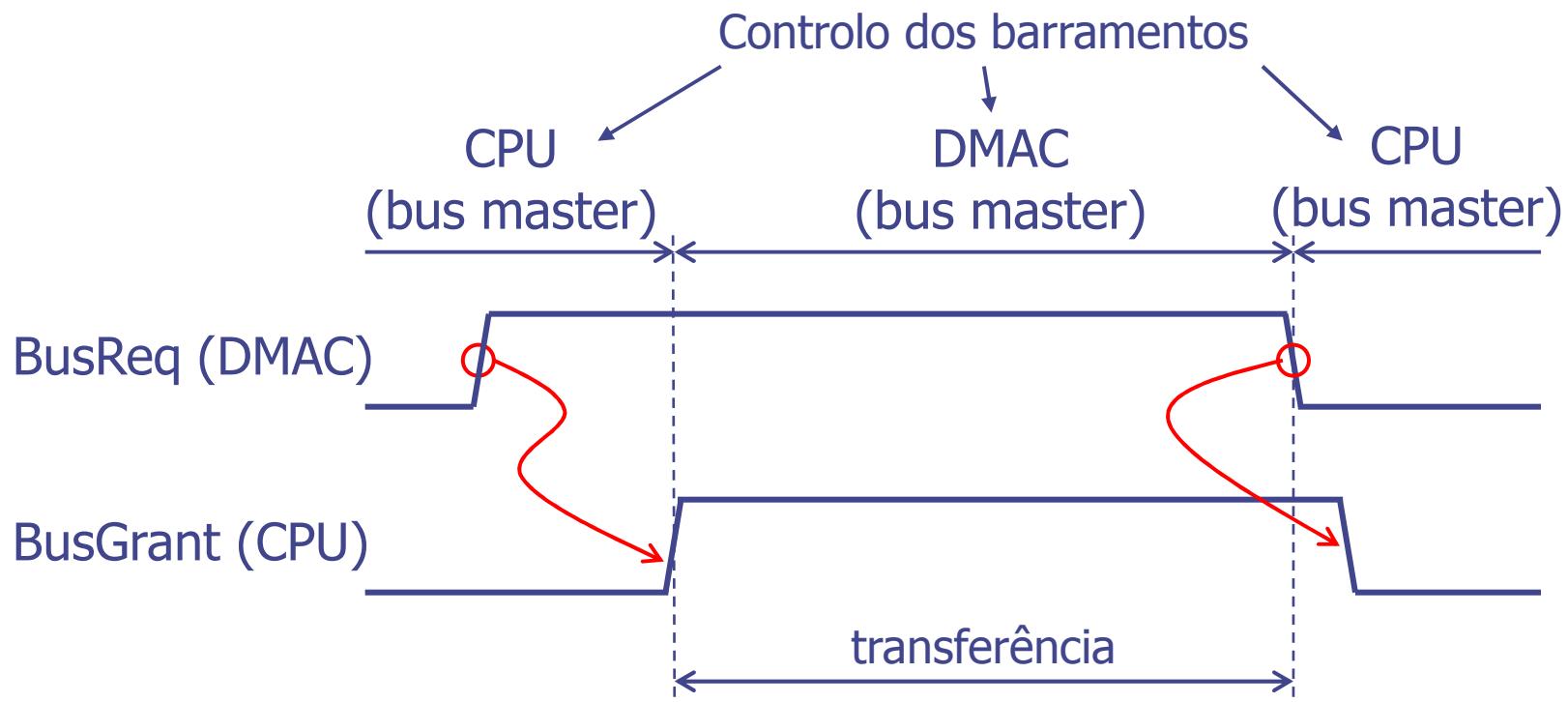
- Apenas um dos dois dispositivos, CPU e DMAC, pode estar ativo no barramento de cada vez... Por defeito é o CPU
- Isto é, só pode haver, em cada instante, um "**bus master**" (só um dispositivo que é "bus master" pode controlar os barramentos)
- Quando o DMAC necessita de aceder aos barramentos para realizar uma transferência, tem que primeiro ter autorização do CPU para ser o "bus master"
- O DMAC só inicia a transferência de informação quando tem a confirmação, por parte do CPU, de que é "bus master"
- O controlo dos barramentos por parte do DMAC é sempre temporário
- Quais são então os passos que o DMAC tem que seguir para fazer uma transferência de dados?

# Controlador de DMA – passos para uma transferência

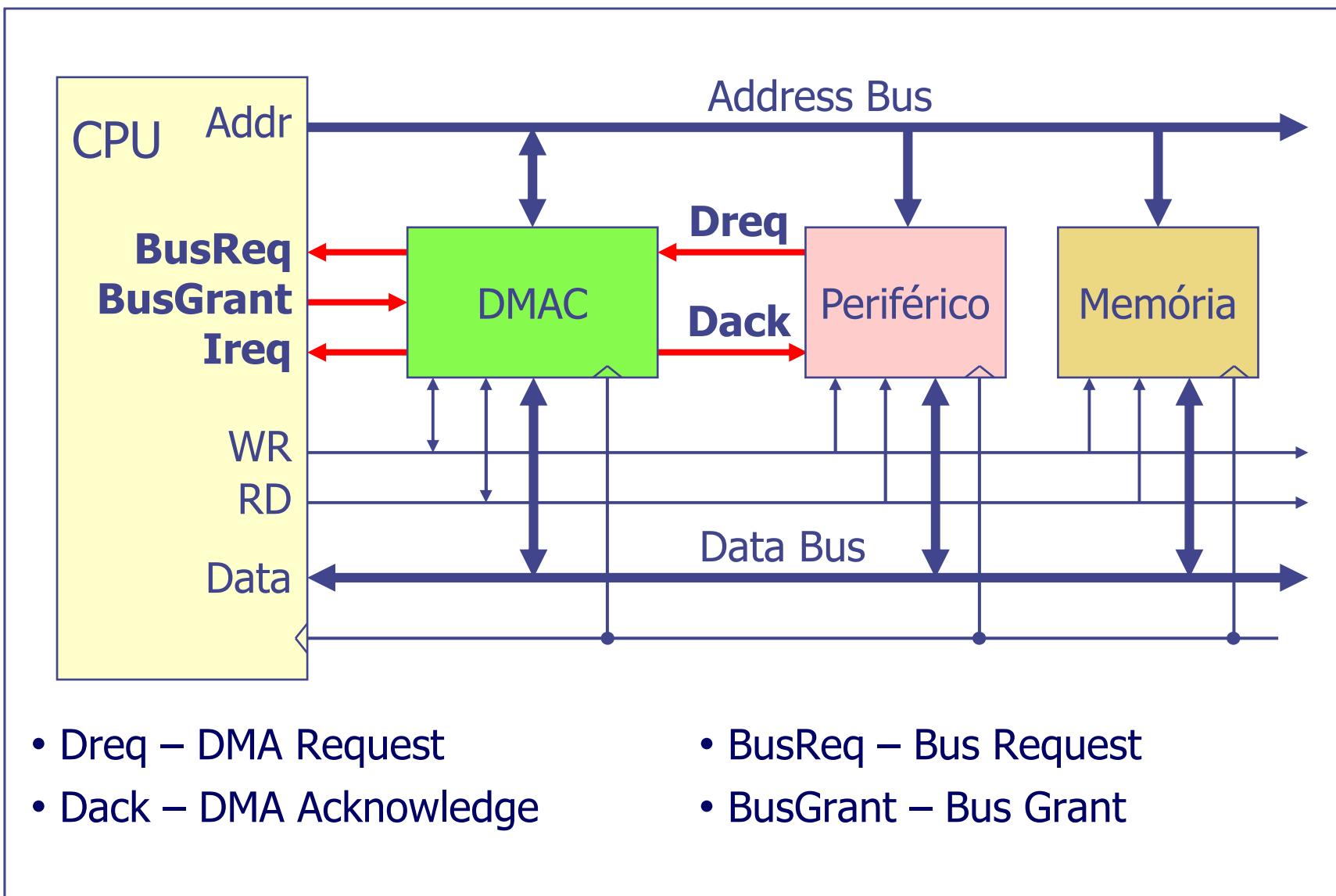
- DMAC pede ao CPU para ser *bus master*
- Espera até ter a confirmação do CPU
- Efetua a transferência:
  - Lê uma palavra (*byte* ou *word*) do dispositivo fonte (do *source address*) para um registo interno
  - Escreve a palavra guardada no registo interno, no passo anterior, no dispositivo destino (no *destination address*)
  - Incrementa *source address* e *destination address*
  - Incrementa o número de palavras transferidas
  - Se não transferiu a totalidade do bloco, repete
- Retira o pedido para ser *bus master* libertando, simultaneamente, os barramentos (ou seja, as suas ligações aos barramentos de dados, de endereços e de controlo passam a funcionar como entradas)

# Controlador de DMA

- Para se tornar um "bus master", o DMAC:
  1. ativa o sinal "**BusReq**" (*Bus Request*) e
  2. espera pela ativação do sinal "**BusGrant**" (CPU ativa *Bus Grant* quando está em condições de libertar os barramentos)



# DMA – Hardware

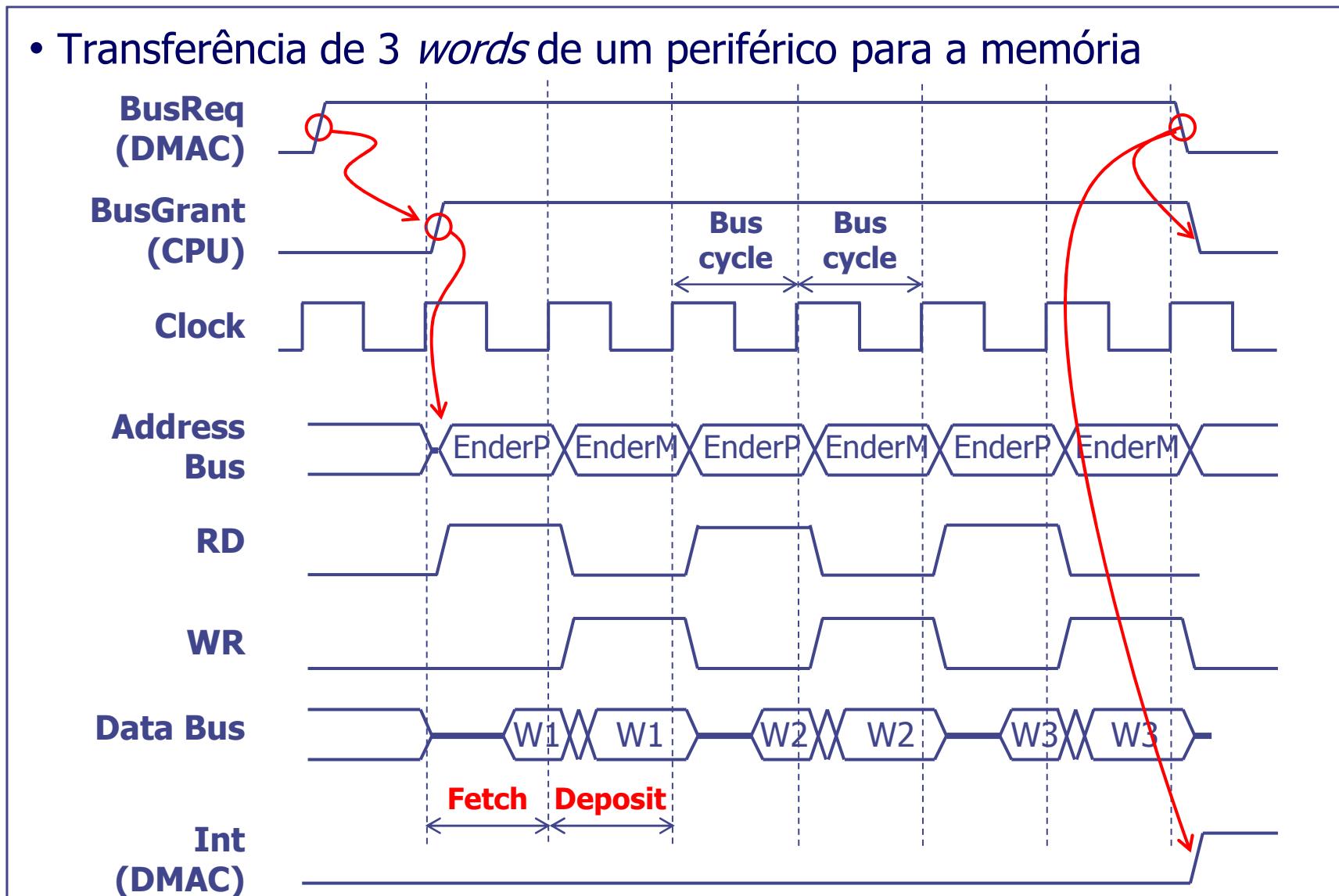


# DMA – Modos de operação

- **Bloco** – o DMAC assume o controlo dos barramentos até todos os dados terem sido transferidos
- **Burst** (rajada)
  - O DMAC transfere até atingir o número de palavras pré-programado ou até o periférico não ter mais informação pronta para ser transferida
  - Se não foi transferida a totalidade da informação:
    - O periférico pode desativar o sinal Dreq o que leva o DMAC a desativar o sinal BusReq e a libertar os barramentos
    - Logo que o periférico ative de novo o sinal Dreq o DMAC volta a ativar o sinal BusReq e, logo que seja "bus master", continua no ponto onde interrompeu
- **Cycle Stealing**
  - O DMAC assume o controlo dos barramentos durante 1 *bus cycle* e liberta-os de seguida ("rouba" 1 *bus-cycle* ao CPU) - transfere parcialmente 1 palavra (*fetch* ou *deposit*)
  - O CPU só liberta os barramentos nos ciclos em que não acede à memória (por exemplo, no estágio MEM de uma instrução aritmética na arquitetura MIPS, pipelined)
  - A transferência é mais lenta, mas o impacto do processo de DMA no desempenho do CPU é nulo - o DMAC aproveita os ciclos que não são, de qualquer modo, usados pelo CPU

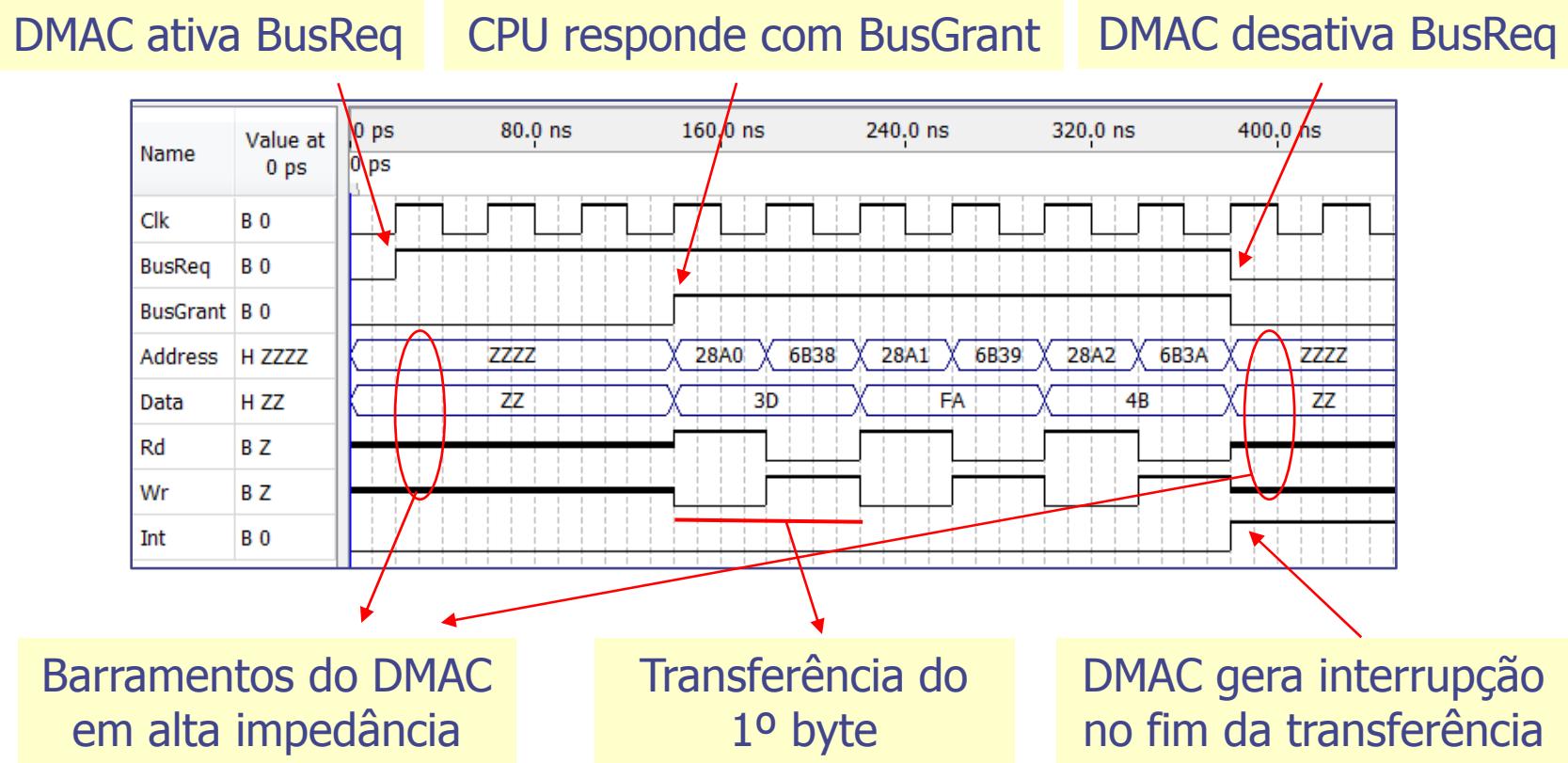
# Modo Bloco (exemplo)

- Transferência de 3 *words* de um periférico para a memória

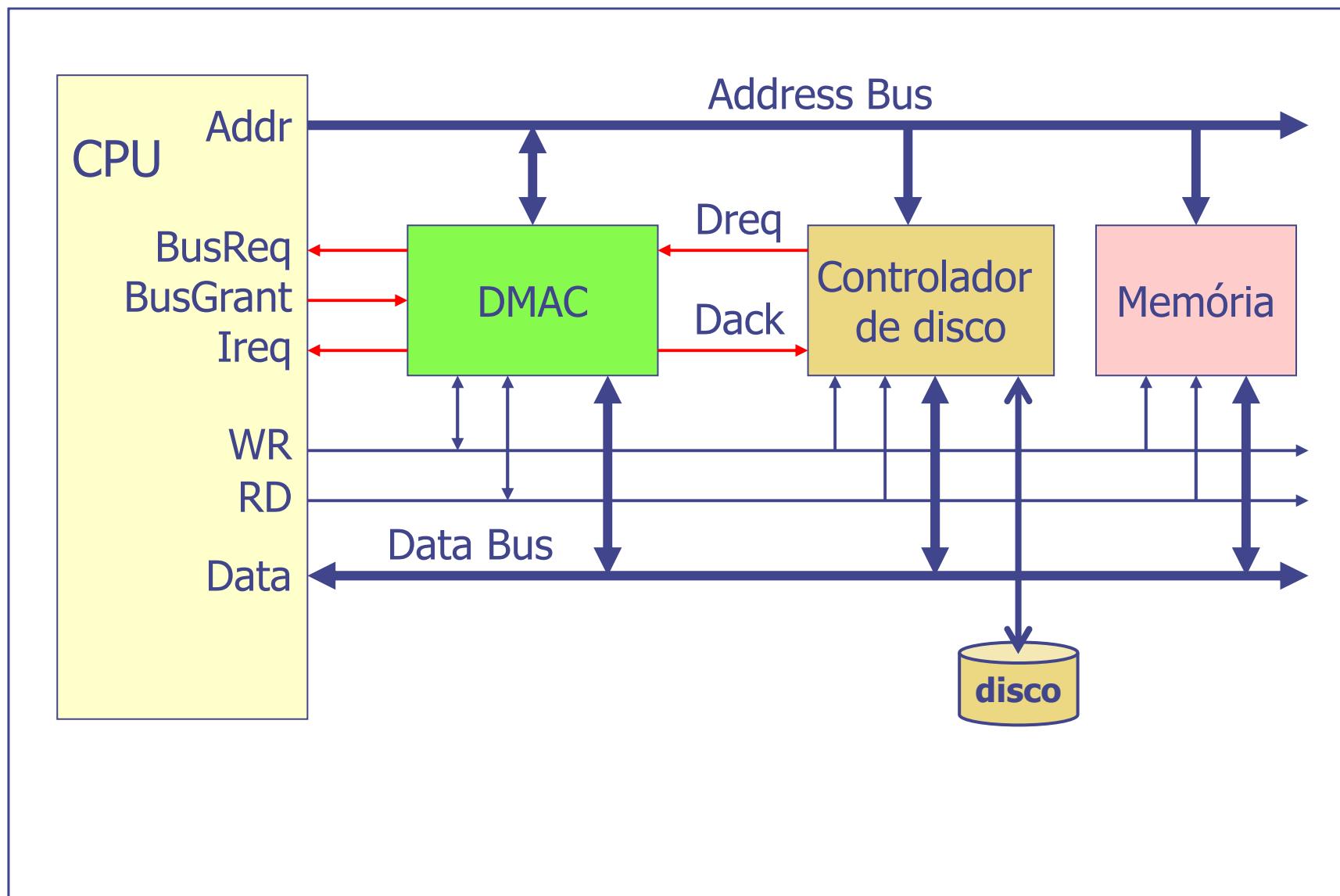


# Modo Bloco (exemplo)

- Exemplo de uma transferência de 3 bytes (memória *byte-addressable*, barramento de dados de 8 bits):
  - Endereço de início na origem: 0x28A0, [0x28A0, 0x28A2]
  - Endereço de início no destino: 0x6B38, [0x6B38, 0x6B3A]



# DMA – Hardware (exemplo)



## Modo Bloco – Exemplo de uma transferência por DMA

1. O CPU envia um comando ao controlador do disco (DiskCtrl): leitura de um dado sector, número de palavras
2. O CPU programa o DMAC: endereço inicial da zona de dados a transferir (Controlador do disco), endereço inicial da zona destino (Memória), número de palavras a transferir
3. O CPU pode continuar com outras tarefas
4. Quando o DiskCtrl tiver lido a informação pedida para a sua zona de memória interna, ativa o sinal **Dreq** do DMAC (sinalizando dessa forma o DMAC de que a informação está pronta para ser transferida)
5. O DMAC ativa o sinal **BusReq**, pedindo autorização ao CPU para ser *bus master*, e fica à espera...
6. Logo que possa, o CPU coloca os seus barramentos em alta impedância e ativa o sinal **BusGrant** (o que significa que o DMAC passou a ser o *bus master*)
7. O DMAC ativa o sinal **Dack** e o DiskCtrl, em resposta, desativa o sinal **Dreq**

## Modo Bloco – Exemplo de uma transferência por DMA

8. O DMAC efetua a transferência: i) lê do endereço-origem para um registo interno e ii) escreve do registo interno para o endereço-destino (incrementa endereços e número de palavras transferidas). Durante esta fase o CPU está impedido de aceder à memória de dados
9. Quando o DMAC termina a transferência:
  - Desativa o sinal **Dack**
  - Deixa de controlar os barramentos, isto é, os barramentos de dados, de endereços e de controlo passam a ser entradas
  - Desativa o sinal **BusReq**
  - Ativa o sinal de interrupção
10. O CPU quando deteta a desativação do BusReq desativa também o sinal BusGrant e pode novamente usar os barramentos
11. Logo que possa, o CPU atende a interrupção gerada pelo DMAC

# Modo "Cycle-Stealing"

- Numa transferência em modo "cycle-stealing", o DMAC efetua a seguinte sequência de operações:

**1. Torna-se bus master**

2. Fetch: lê 1 palavra do dispositivo fonte (do *source address*)

**3. Liberta os barramentos**

4. Espera durante um tempo fixo pré-determinado (Ex. 1T)

**5. Torna-se bus master**

6. Deposit: escreve 1 palavra no dispositivo destino (no *destination address*)

**7. Liberta os barramentos**

8. Incrementa *source address* e *destination address*

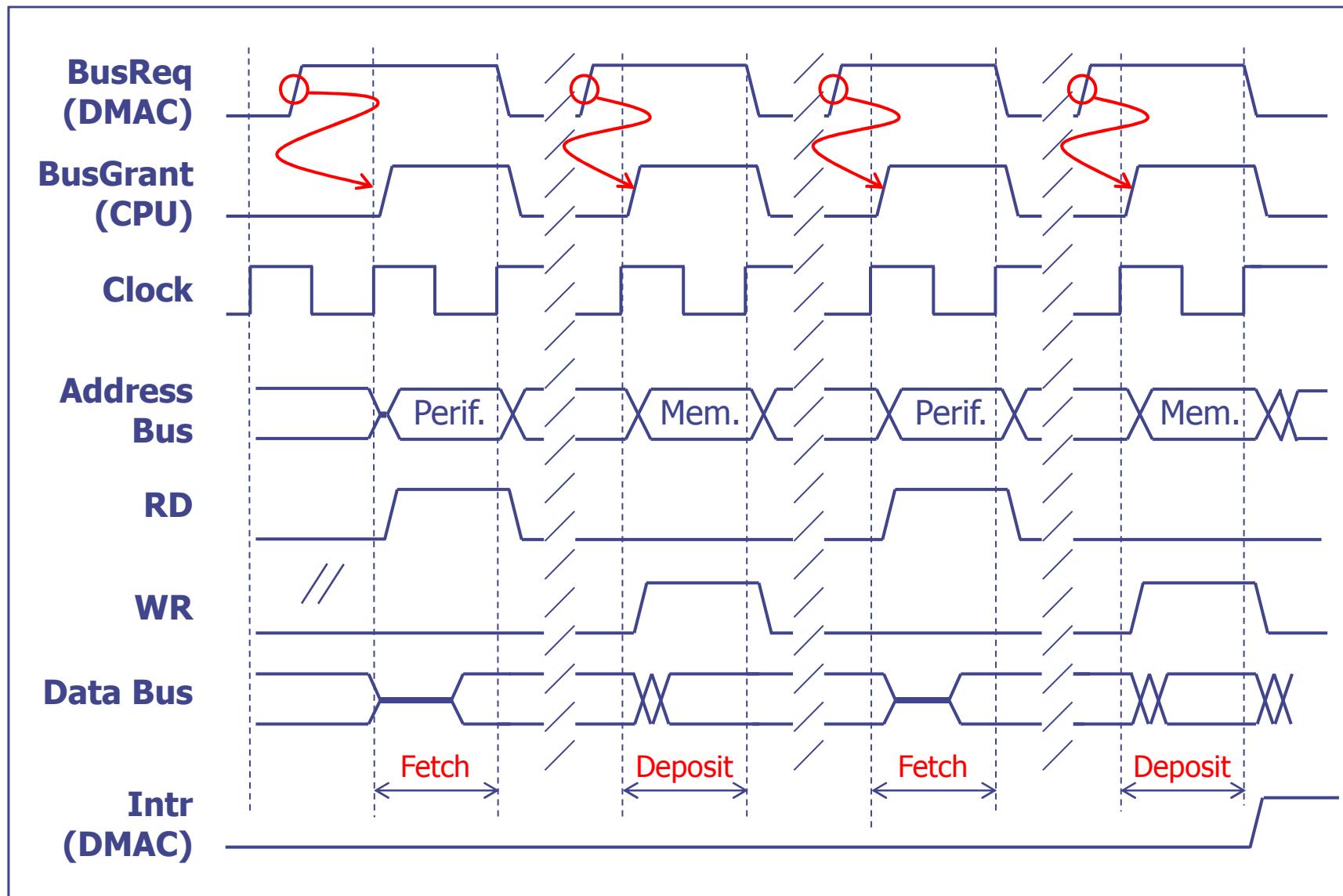
9. Incrementa o nº de bytes/words transferidos

10. Espera durante um tempo fixo pré-determinado (Ex. 1T)

11. Se não transferiu a totalidade do bloco, repete desde 1

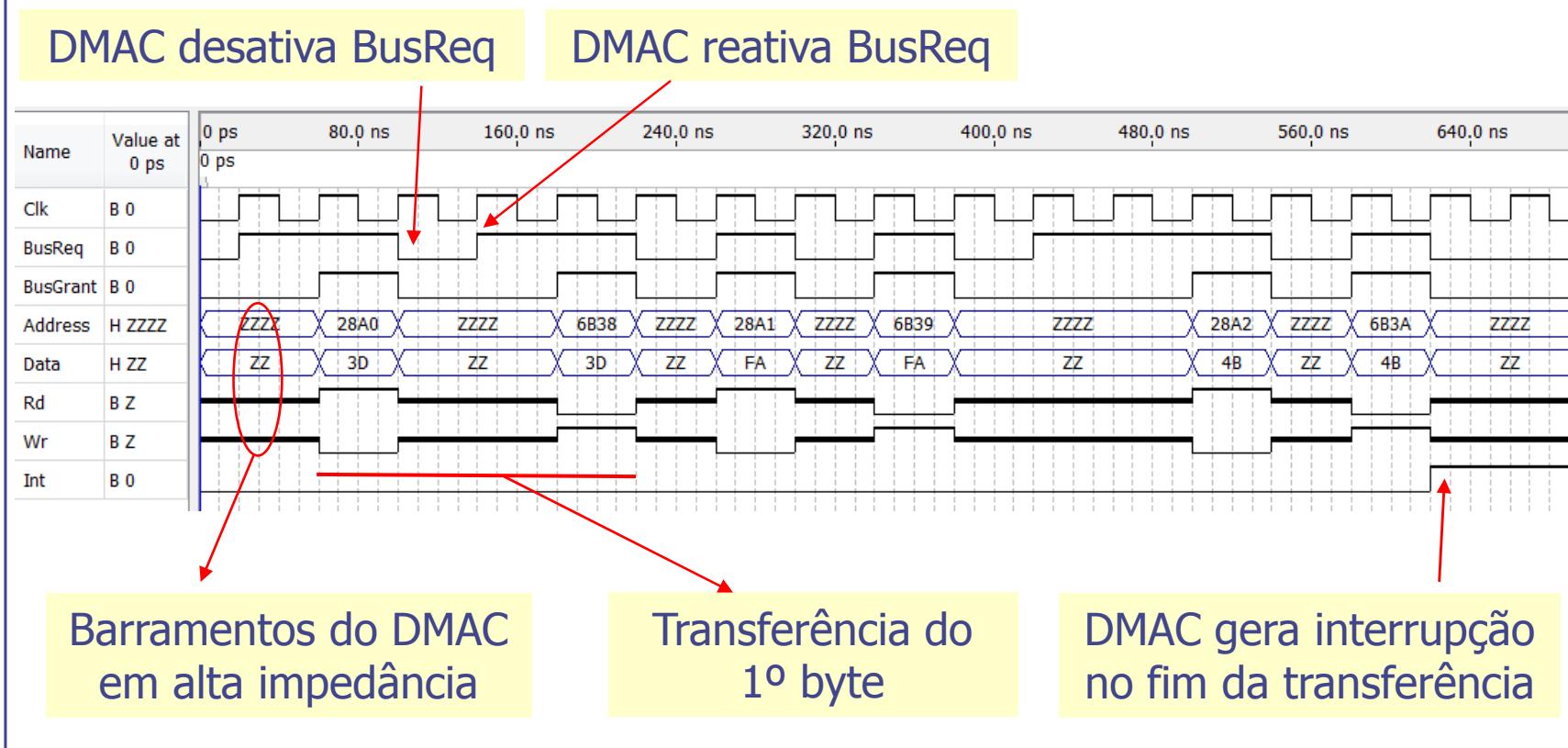
12. Após a transferência da totalidade dos dados, ativa o sinal de interrupção

# Modo "Cycle-Stealing" (exemplo)

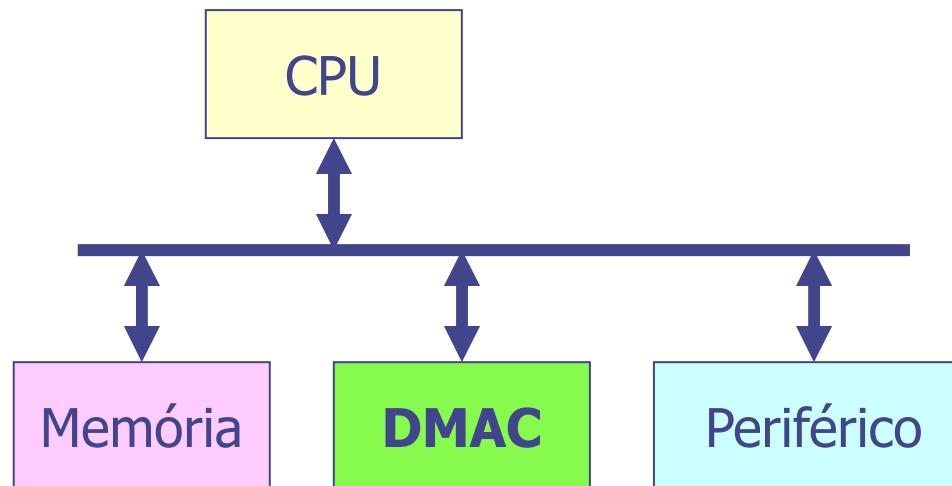


# Modo "Cycle-Stealing" (exemplo)

- Exemplo de uma transferência de 3 bytes (memória *byte-addressable*, barramento de dados de 8 bits):
  - Endereço de início na origem: 0x28A0, [0x28A0, 0x28A2]
  - Endereço de início no destino: 0x6B38, [0x6B38, 0x6B3A]

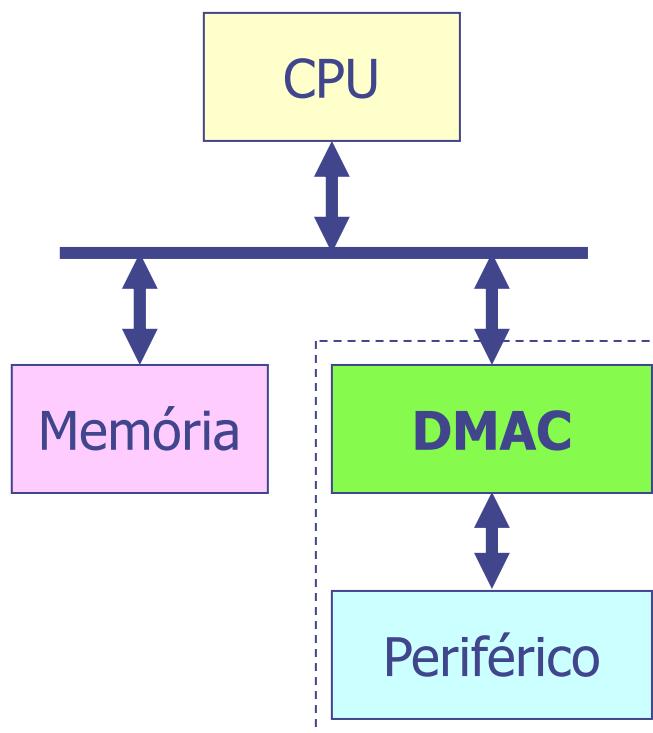


# Configurações – Canal de DMA



- O DMAC fornece um serviço de transferência genérico
- Pode ser usado para transferir informação:
  - de I/O para memória
  - de memória para I/O
  - de memória para memória
- Para a transferência de 1 palavra o barramento é usado 2 vezes (2 "bus cycles" por palavra)
- Em modo "cycle stealing", por cada palavra transferida, o CPU liberta os barramentos 2 vezes

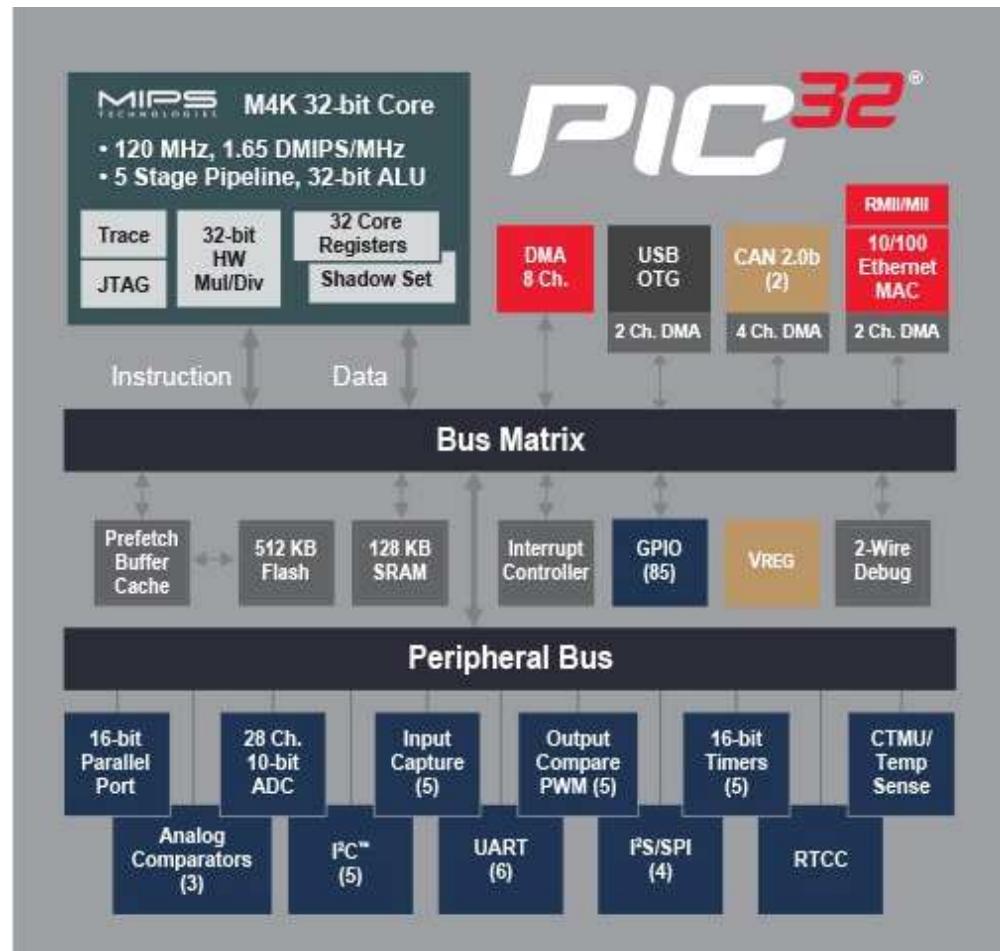
# Configurações – DMA dedicado



- Um periférico tem o seu próprio controlador de DMA (**DMAC dedicado**)
- Para a transferência completa de 1 palavra o barramento é usado apenas 1 vez (1 "bus cycle" por palavra):
  - DMAC → memória ou memória → DMAC

# Controladores de DMA no PIC32

- Controlador DMA de 8 canais (transferência memória/periférico e memória/memória)
- Controlador dedicado no módulo USB
- Controlador dedicado no módulo CAN
- Controlador dedicado no módulo Ethernet



# Exercícios

- Suponha um DMAC não dedicado de 32 bits (i.e. com barramento de dados de 32 bits), a funcionar a 100 MHz. Suponha ainda que são necessários 2 ciclos de relógio para efetuar uma operação de leitura ou escrita (i.e. 1 "bus cycle" é constituído por 2 ciclos de relógio).
- **Exercício 1** – Determine a taxa de transferência desse DMAC (expressa em Bytes/s), supondo um funcionamento em modo bloco.
- **Exercício 2** – Determine a taxa de transferência de pico desse DMAC (expressa em Bytes/s), supondo um funcionamento em modo "cycle-stealing" e um tempo mínimo entre operações elementares de 1 ciclo de relógio ("fetch", 1T mínimo, "deposit", 1T mínimo).
- **Exercício 3** – Repita os exercícios 1 e 2 supondo um DMAC dedicado com as características referidas anteriormente.

# Exercícios

- **Exercício 4** – Admita uma arquitetura em que o ciclo de barramento ( $bT$ ) é igual 2ns. Suponha que o CPU programou um controlador de DMA em modo "*cycle-stealing*" para ter um tempo de espera entre "*fetch*" e "*deposit*" igual a  $1*bT$  e um tempo de espera entre o "*deposit*" e o próximo "*fetch*" igual a  $2*bT$ . Sabendo que o barramento de dados é de 16bits, determine a taxa de transferência de pico desse controlador de DMA expresso em Bytes/s.
- **Exercício 5** – Admita agora que, para as mesmas condições indicadas no exercício anterior, o tempo médio de resposta a um pedido de *BusReq* é, para um ciclo completo de transferência, de  $2.5*bT$ . Qual seria, nesse caso, a taxa média de transferência nesse período (expressa em palavras/s).
- **Exercício 6** – Repita os exercícios 4 e 5 supondo um controlador de DMA dedicado com as características referidas anteriormente.

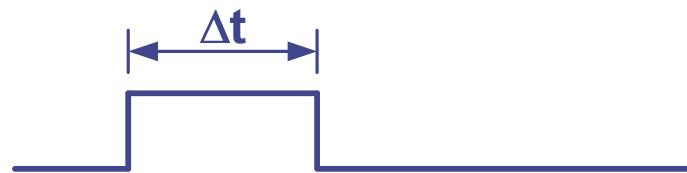
# Aula 10

- *Timers*
  - Aplicações
  - Princípio de funcionamento
  - Divisão de frequência e controlo do *duty-cycle*
- *Watchdog timer*

José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

# Introdução

- Um *timer* é um dispositivo periférico de suporte que permite, no essencial, a medição de tempo, partindo de uma referência temporal conhecida
- Alguns exemplos de aplicações típicas de *timers*:
  - geração de um evento com uma duração controlada;  
**exemplo:** geração de um impulso com uma duração definida,  $\Delta t$

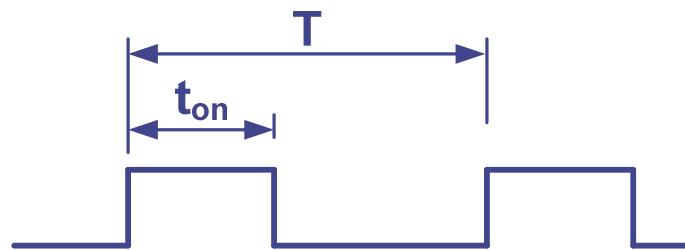


- geração de um evento periódico com período controlado;  
**exemplo:** geração de um impulso com um período definido,  $T$



# Introdução

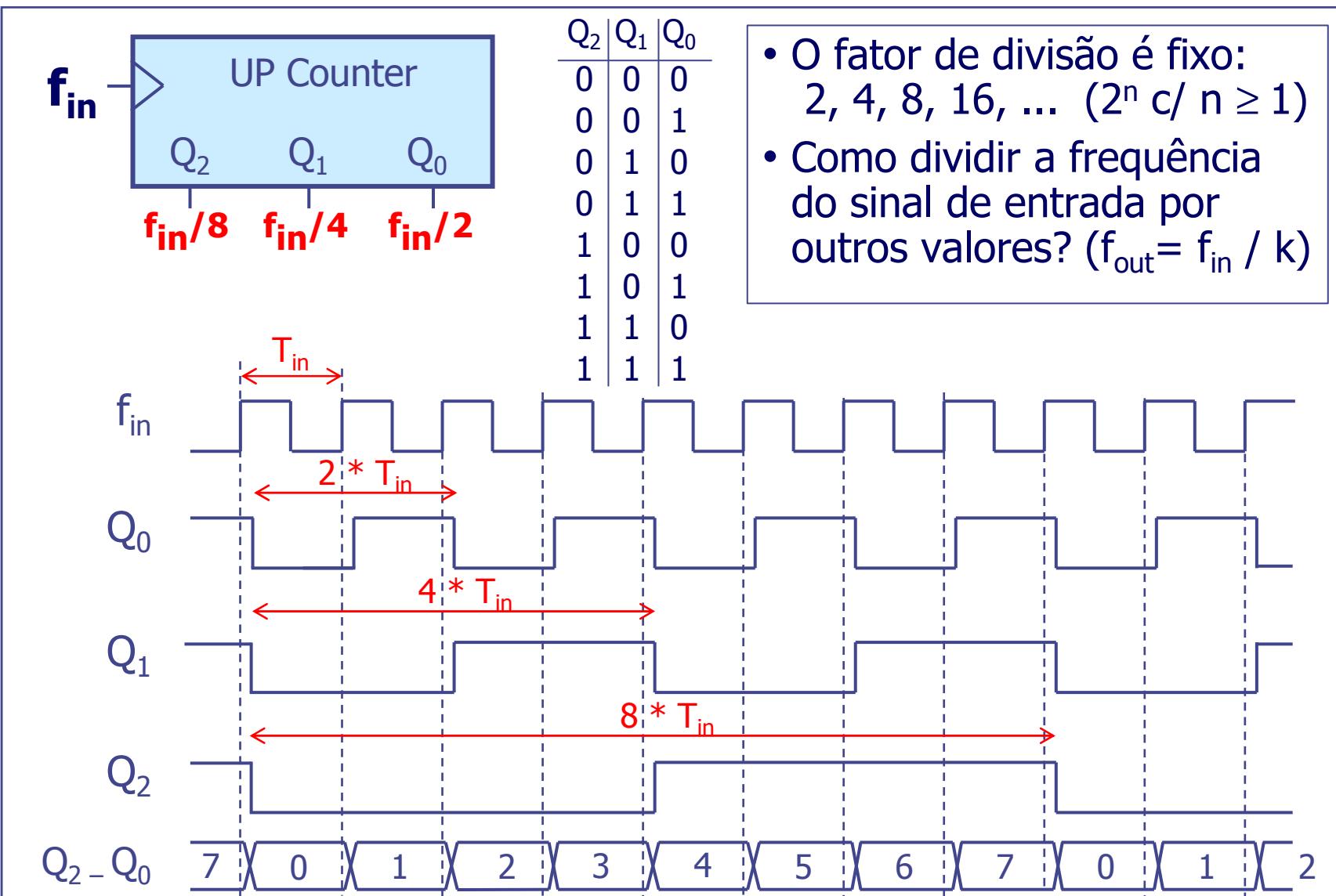
- Geração de um evento periódico com período e duração controlados. Exemplo: geração de um sinal periódico com um período de 10 ms e um "duty-cycle" de 40%:



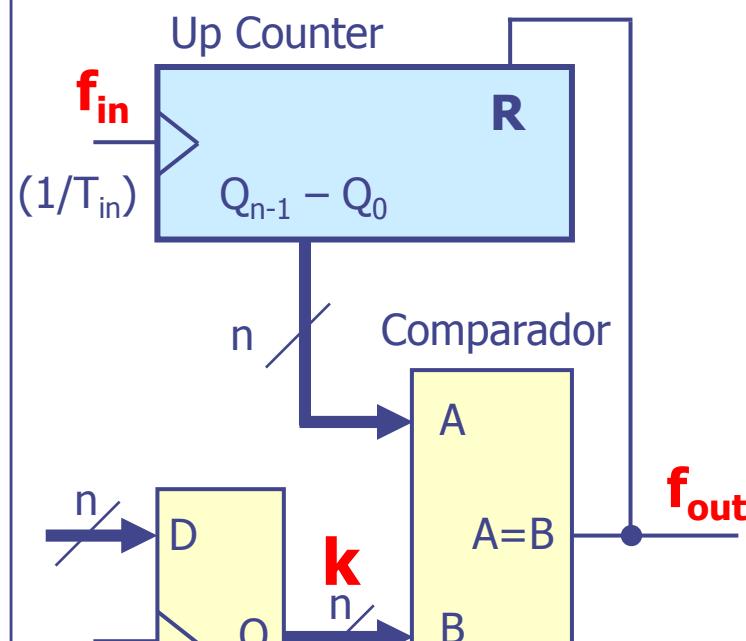
$$\text{Duty-cycle} = (t_{on} / T) * 100 \ [\%]$$

- "ton" é o tempo durante o qual o sinal está no nível lógico 1, num período
- a possibilidade de alterar o valor de "ton" sem alterar o valor de  $T$  é útil em muitas situações e designa-se por **PWM (Pulse Width Modulation)** – modulação por largura de pulso)
- O funcionamento dos *timers* baseia-se sempre na contagem de ciclos de um sinal de relógio com frequência conhecida

# Divisão de frequência



# Divisão de frequência (versão 1)

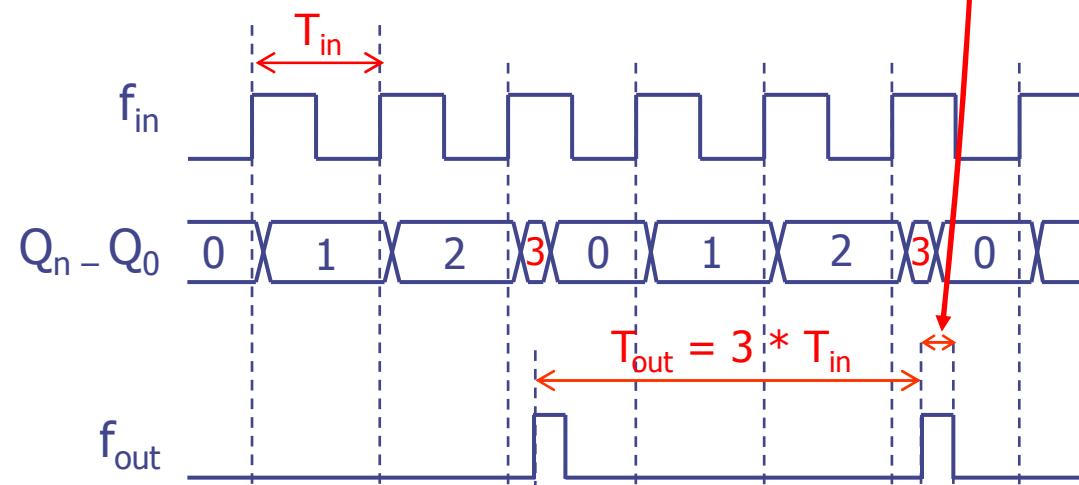


Registro de  
n bits

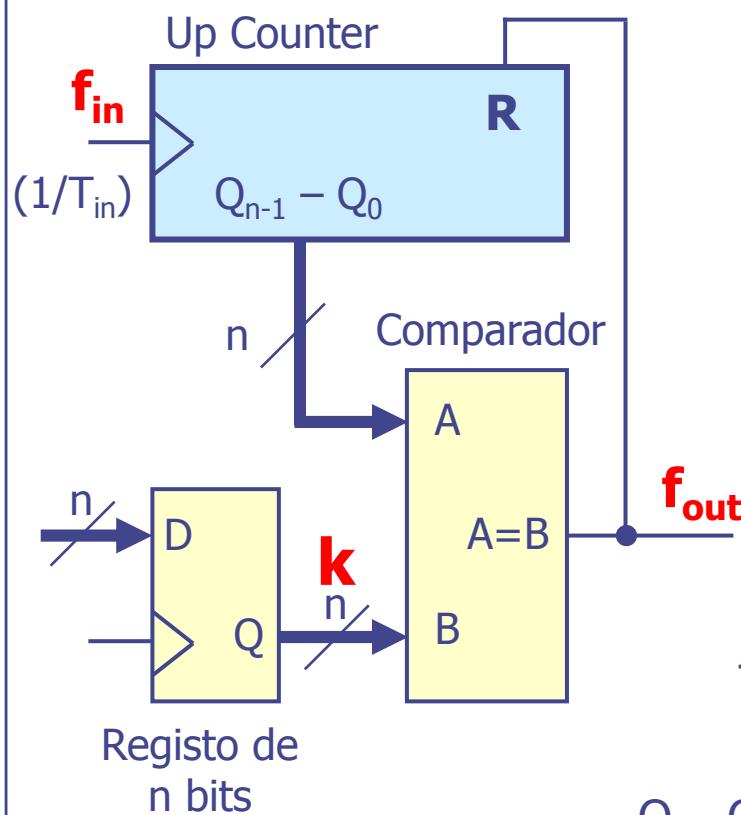
$$k = \frac{f_{in}}{f_{out}}$$

- **Reset do contador assíncrono:** o tempo a 1 do sinal **R** não é controlado (2 atrasos de propagação)
- O período do sinal de saída é:  
 $T_{out} = k * T_{in}$ , ou seja:
  - $f_{out} = f_{in} / k$

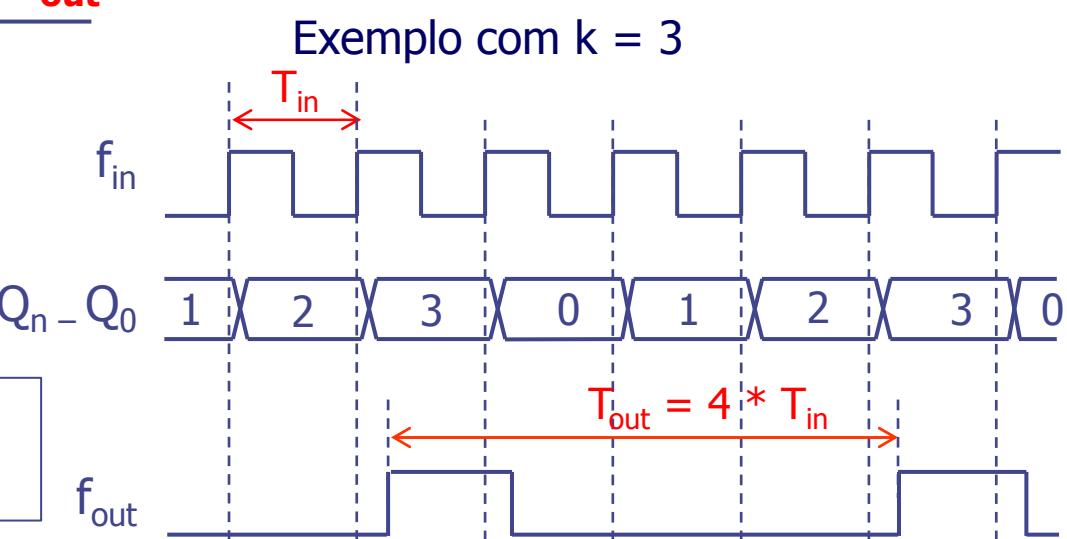
Exemplo com  $k = 3$



# Divisão de frequência (versão 2)



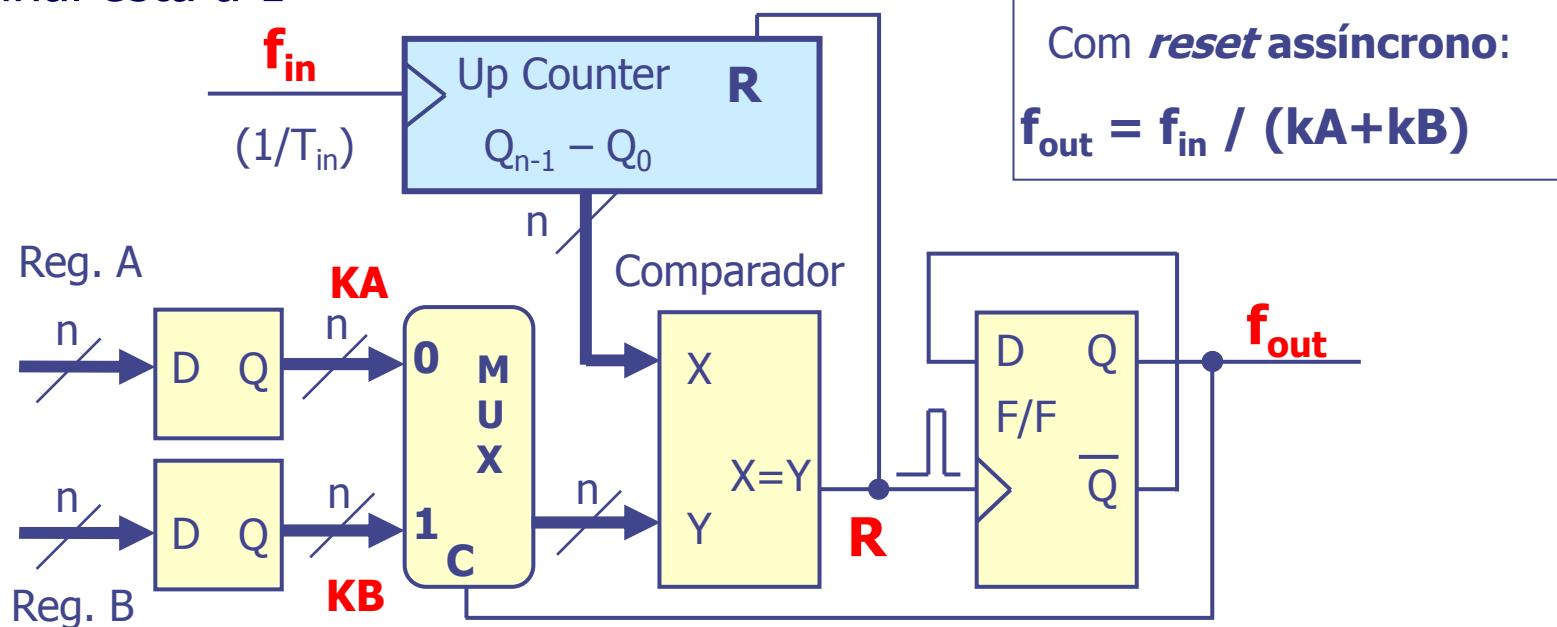
- **Reset do contador síncrono:** o tempo a 1 do sinal de saída ( $f_{out}$ ) é 1 ciclo de relógio do sinal de entrada ( $T_{in}$ )
- O período do sinal de saída é:  
 $T_{out} = (k+1) * T_{in}$ , ou seja:
  - $f_{out} = f_{in} / (k+1)$
  - $k = (f_{in} / f_{out}) - 1$



$$k = \frac{f_{in}}{f_{out}} - 1 \quad (k > 0)$$

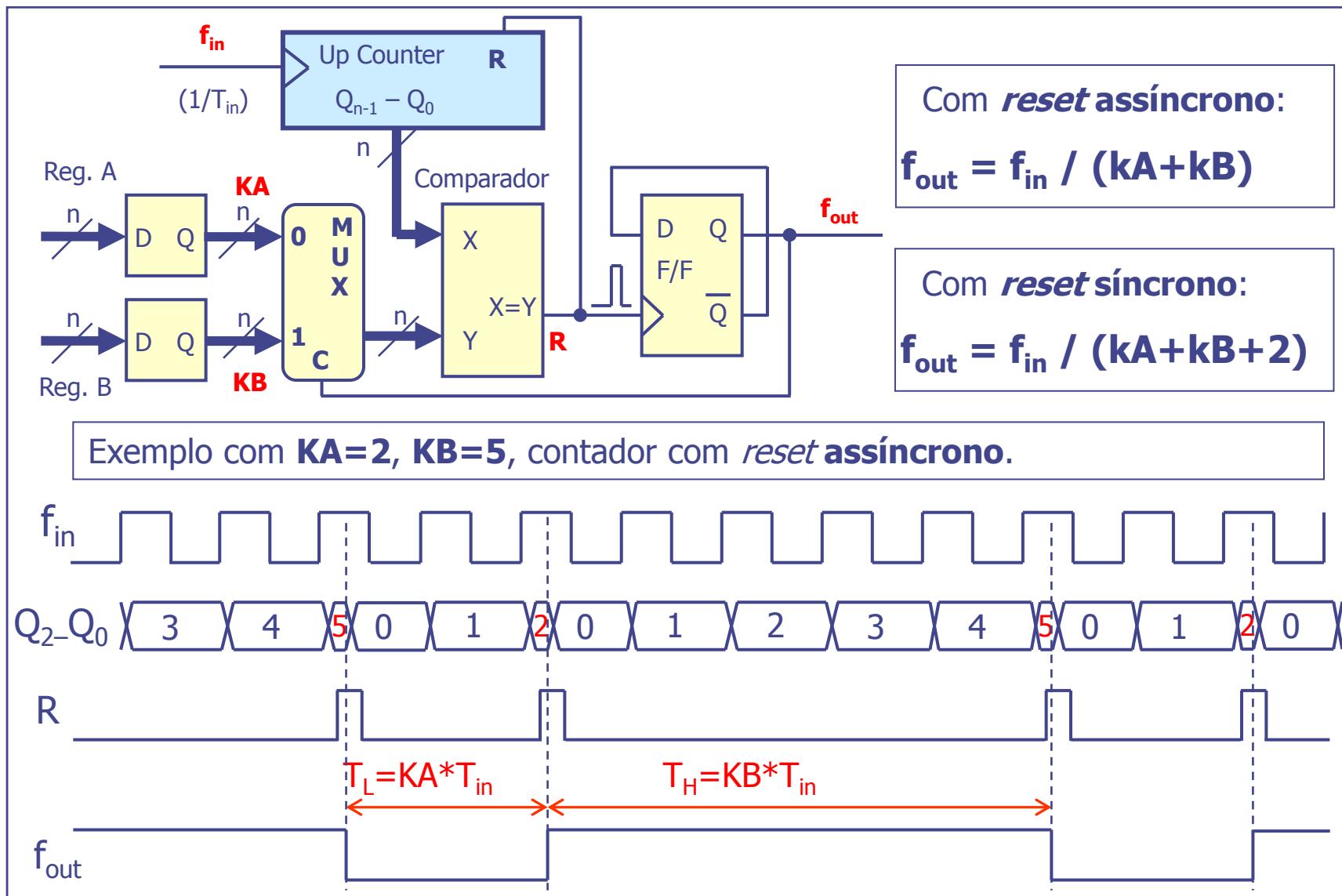
# Controlo de frequência e *duty-cycle*

- Para este tipo de aplicação, o timer tem que permitir o controlo do período do sinal de saída, bem como do tempo durante o qual esse sinal está a 1



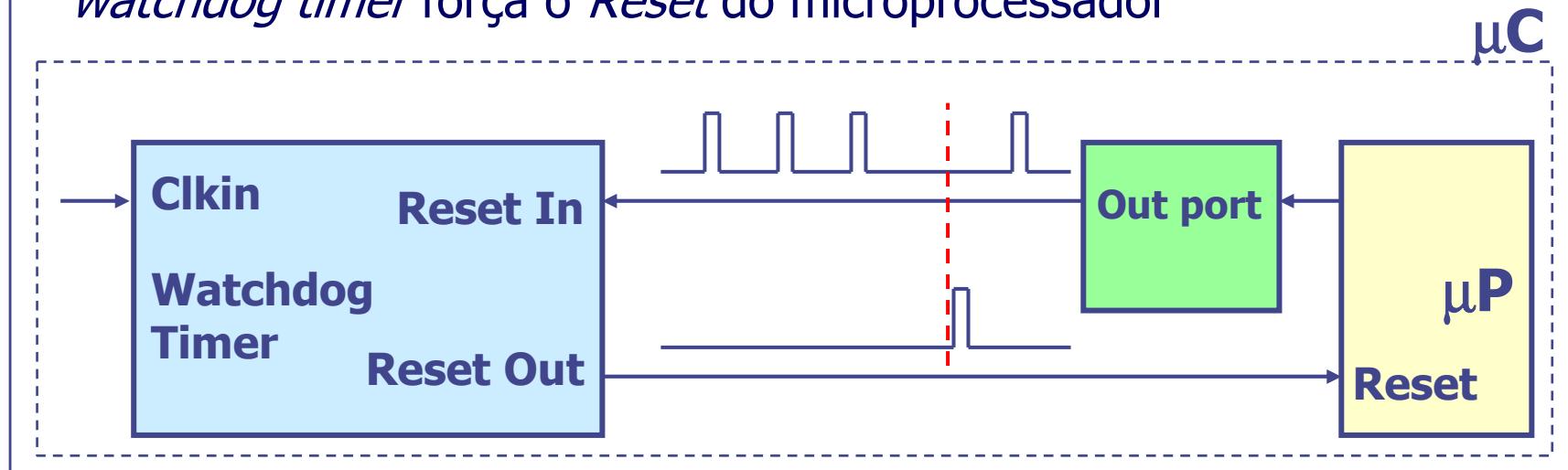
- Quando a saída  $Q$  do flip-flop está a 1, a entrada  $Y$  do comparador toma o valor de  $KB$ ; caso contrário, toma o valor de  $KA$ . Logo, o tempo durante o qual o sinal de saída está a 1 depende de  $KB$ , e durante o qual está a 0 depende de  $KA$

# Controlo de frequência e *duty-cycle*

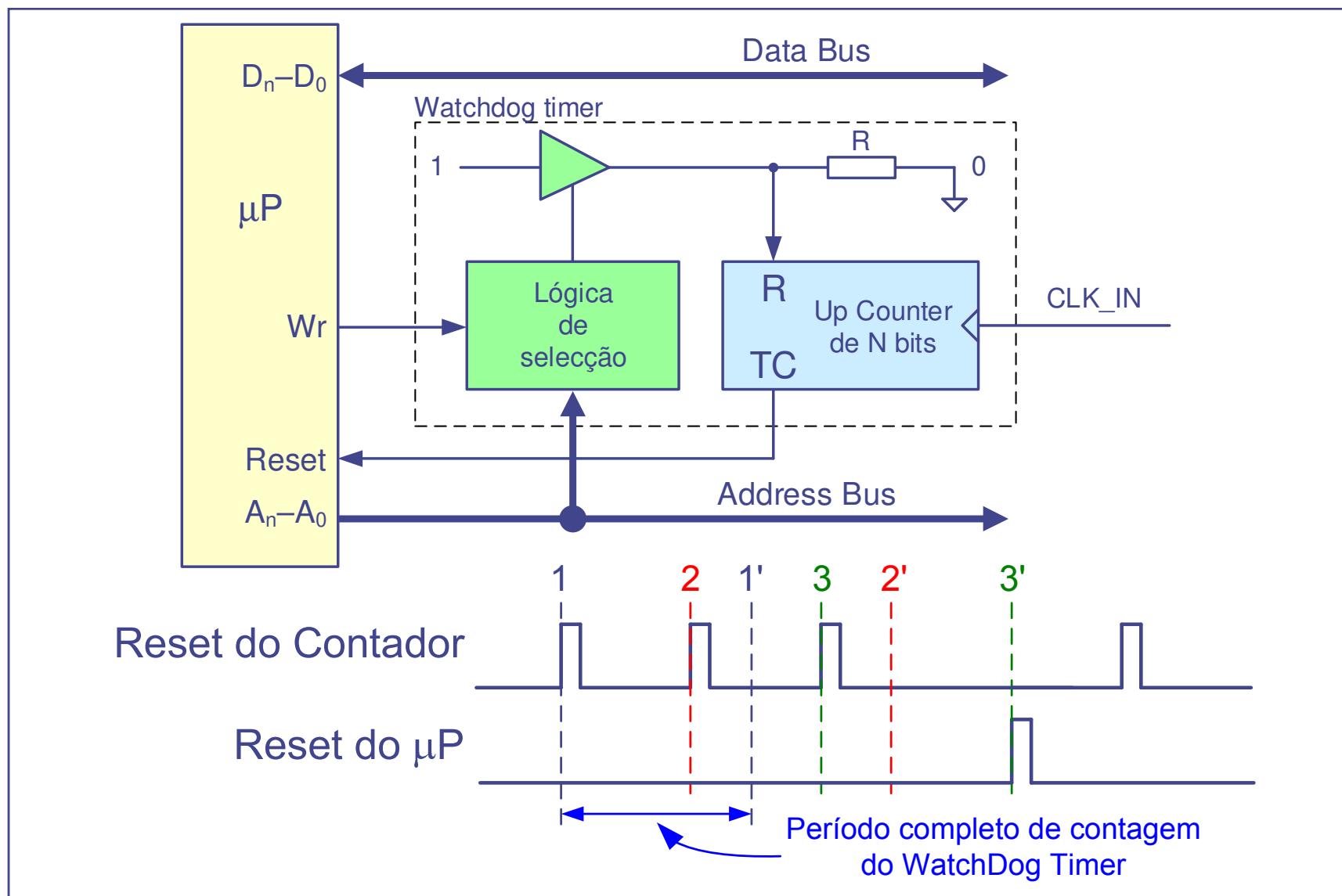


# *Watchdog Timer* (temporizador "cão de guarda")

- Sistemas baseados em microcontrolador podem assegurar funções de controlo críticas que não podem falhar
- Como garantir que um crash do microprocessador não compromete o funcionamento global do sistema?
- Um *watchdog timer* tem como função monitorizar a operação do microprocessador e, em caso de falha, forçar o seu reinício
- Situação mais comum: se o microprocessador não atuou a entrada de *Reset* do *watchdog timer* ao fim de um tempo pré-determinado o *watchdog timer* força o *Reset* do microprocessador



# Watchdog Timer – exemplo de implementação



# *Watchdog Timer* – exemplo de utilização

- A aplicação no microcontrolador executa em ciclo infinito
- O *watchdog timer* é ativado quando o programa inicia. O *reset* da sua contagem é feito regularmente no corpo do ciclo (no exemplo, `clearWatchdogTimer()`)

```
void main(void)
{
    enableWatchdogTimer();
    (...)

    while(1)
    {
        (...)

        clearWatchdogTimer();
    }
}
```

- Caso haja uma falha no microprocessador que implique a quebra de execução do ciclo, a função "clearWatchdogTimer()" deixa de ser chamada e o *watchdog timer* deixa de ser reiniciado:
  - Quando o contador do *watchdog timer* atinge o valor máximo da contagem força um *reset* ao microprocessador

# Aula 11

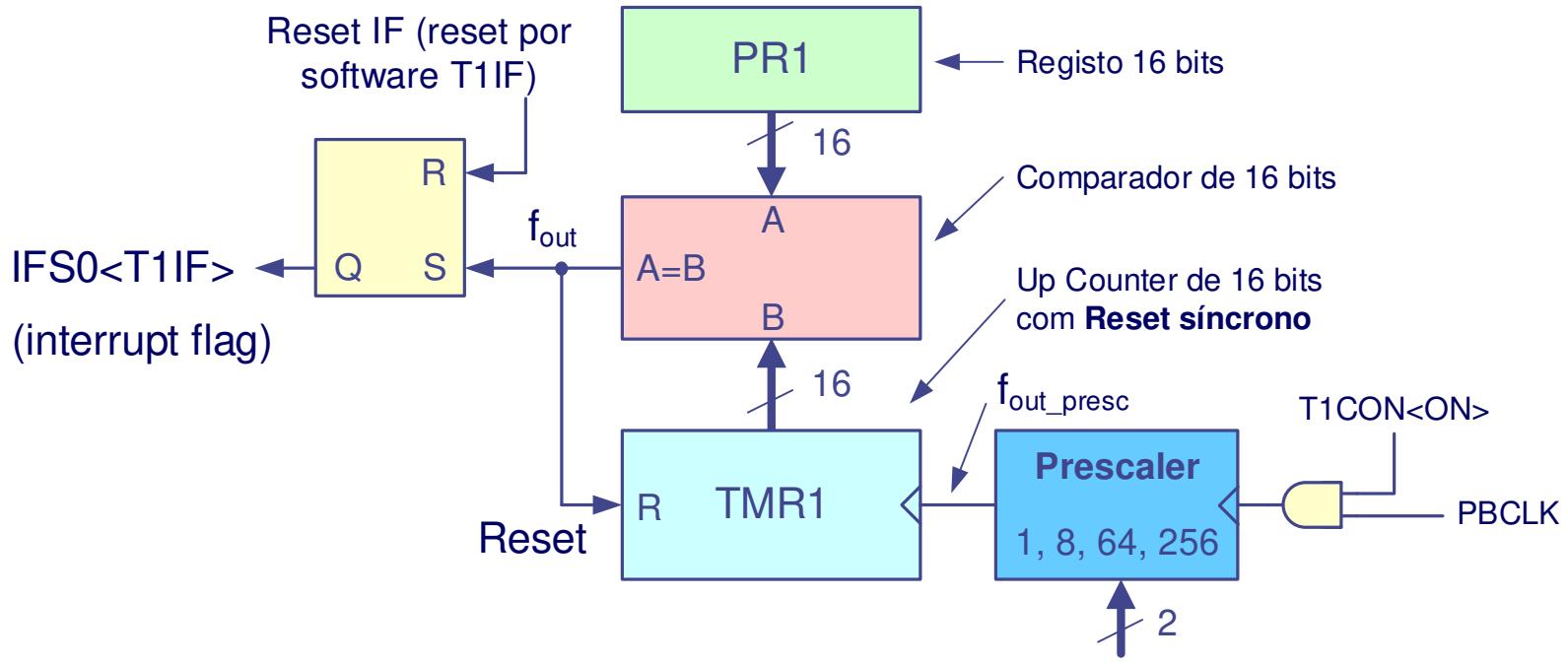
- *Timers no PIC32*
  - Estrutura e funcionamento
  - Geração de sinais PWM (*output compare module*)

José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

## *Timers* no PIC32

- A série PIC32MX7xx disponibiliza **5 *timers* de 16 bits**, designados por **T1, T2, T3, T4 e T5**
- T2, T3, T4 e T5 têm a mesma estrutura e apresentam o mesmo modelo de programação. São designados pelo fabricante como ***timers tipo B***
- T2 a T5 podem ser agrupados 2 a 2 para formar 2 *timers* de 32 bits (T2 e T3 e/ou T4 e T5)
- O T1 é designado como ***timer tipo A***; tem uma estrutura semelhante aos restantes e pequenas diferenças no modelo de programação
- A frequência-base de entrada para os *timers* é dada pelo *Peripheral Bus Clock (PBCLK)*. Na placa DETPIC32 a frequência de PBCLK é metade da frequência de CPU, i.e. **PBCLK = 20 MHz**
- Os *timers* do PIC32 não têm saída acessível no exterior. Podem ser usados para gerar interrupções (todos) ou como base de tempo para a geração de sinais com "duty-cycle" configurável (T2 e T3)

# PIC32 – *timer* tipo A (T1)



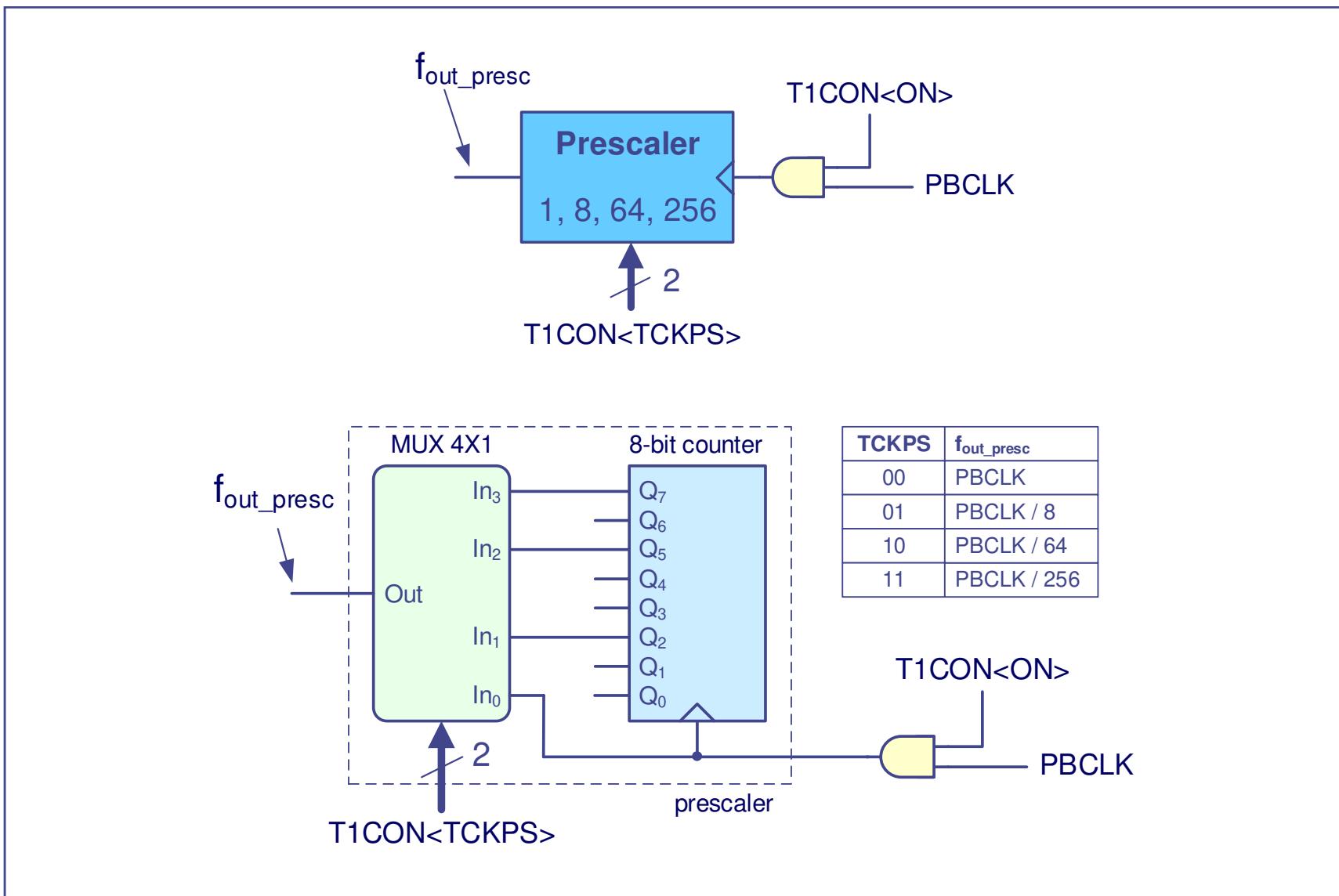
- Frequência à entrada do contador de 16 bits:

$$f_{out\_presc} = \frac{PBCLK}{K_{prescaler}}$$

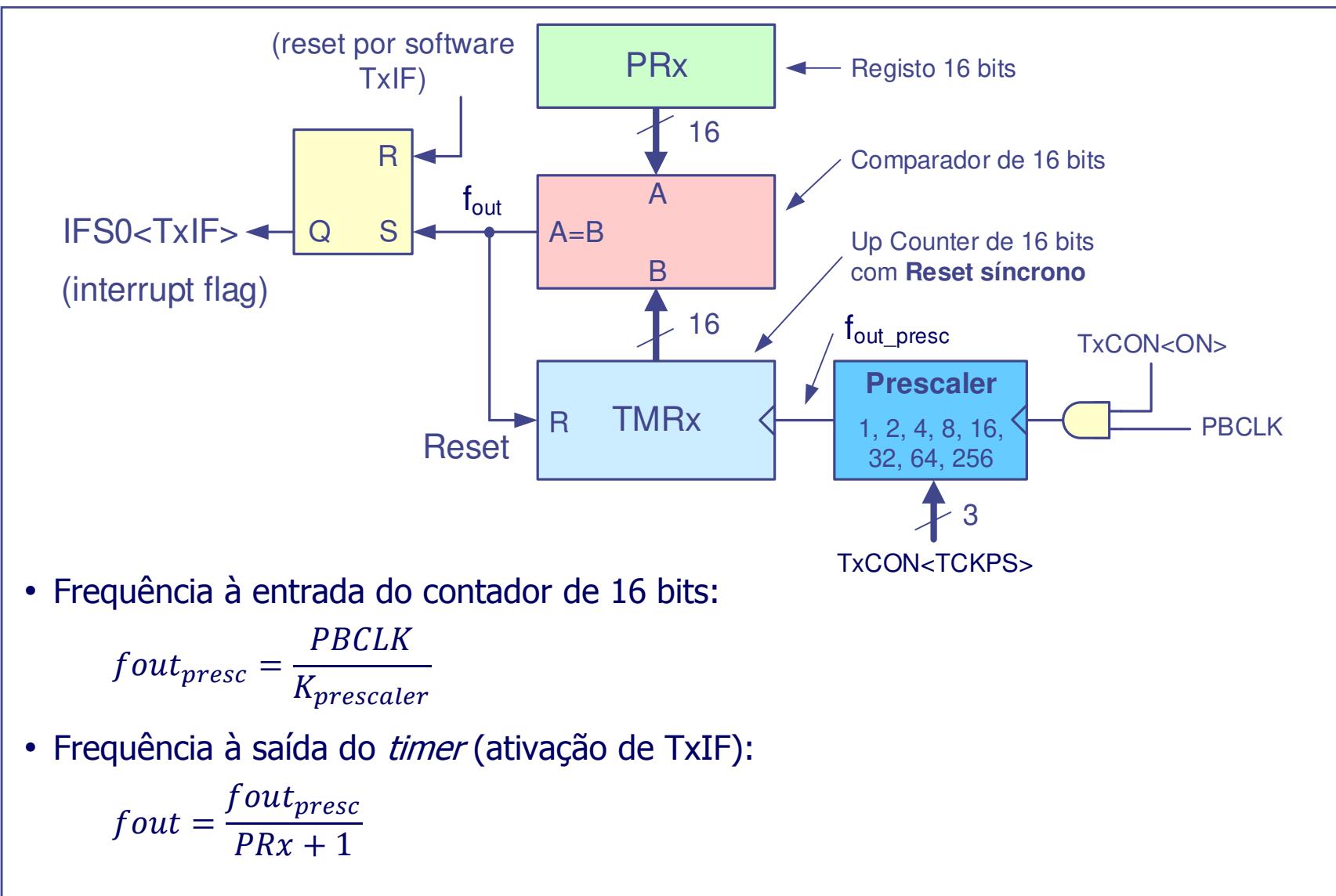
- Frequência à saída do *timer* (ativação de T1IF):

$$f_{out} = \frac{f_{out\_presc}}{PR1 + 1}$$

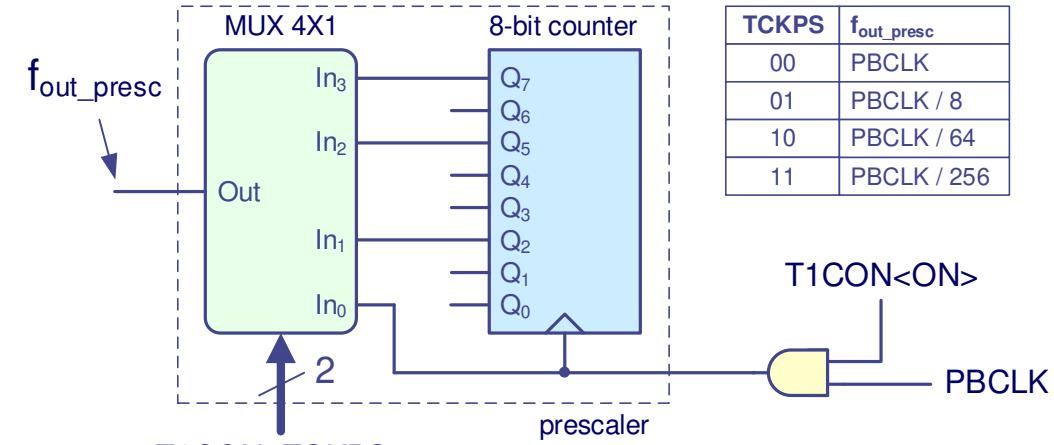
# PIC32 – timer tipo A (prescaler)



# PIC32 – *timers* tipo B (T2 a T5)

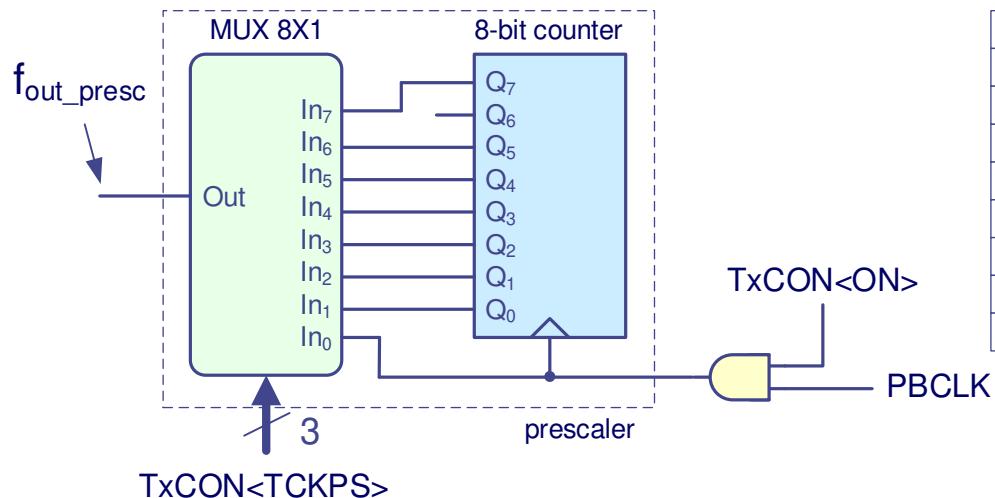


# PIC32 –prescalers

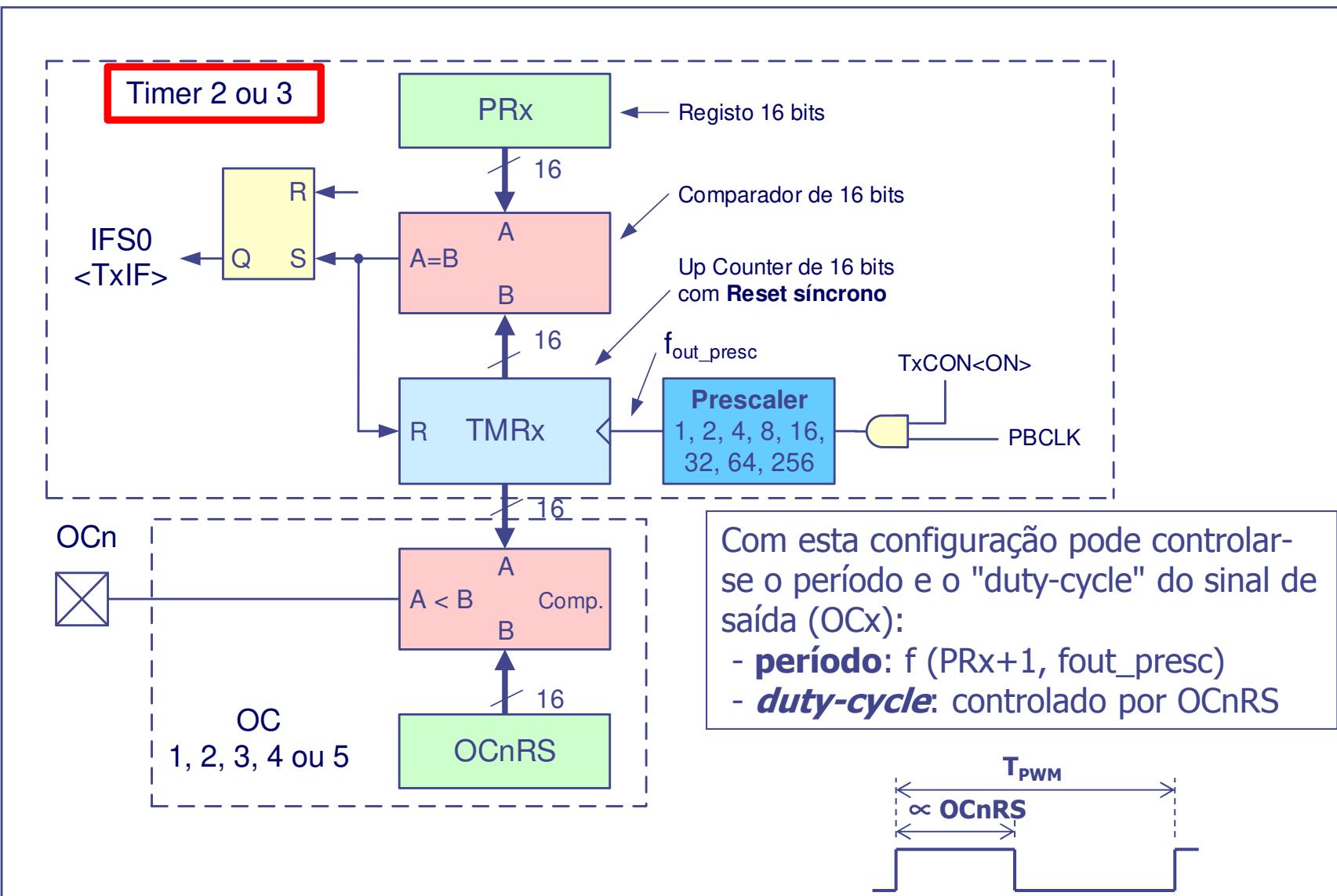


**prescaler tipo A**

**prescaler tipo B**



# PIC32 – controlo de período e "duty-cycle"

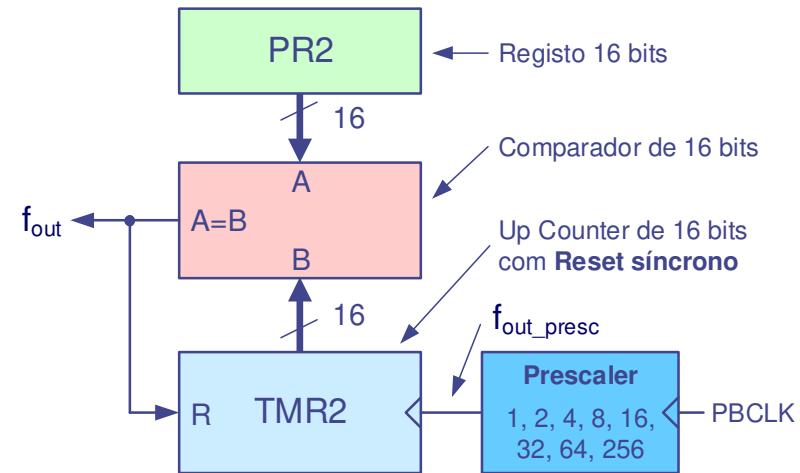


# Exercício

Calcular as constantes para gerar um sinal PWM com uma frequência de 8 Hz e um *duty-cycle* de 20%, usando T2 como referência e OC1 como saída (PBCLK = 20 MHz)

$$f_{out} = \frac{f_{out\_presc}}{PR2 + 1} = \frac{\frac{PBCLK}{K_{prescaler}}}{PR2 + 1}$$

$$K_{prescaler} = \frac{PBCLK}{(PR2 + 1) * f_{out}}$$



## 1. Cálculo da constante de divisão do *prescaler*

$$K_{prescaler} \geq \left\lceil \frac{PBCLK}{(65535 + 1) * 8} \right\rceil = 39$$

$$K_{prescaler} = 64$$

Valor máximo da constante PR2

# Exercício (continuação)

2. Cálculo da constante de divisão do *timer* (PR2), com Kprescaler=64

$$f_{out\_presc} = \frac{PBCLK}{K_{prescaler}} = \frac{20 * 10^6}{64} = 312500 \text{ Hz}$$

$$PR2 = \left( \frac{f_{out\_presc}}{fout} \right) - 1 = \frac{312500}{8} - 1 = 39062$$

3. Cálculo de OC1RS (*duty-cycle* de 20%):

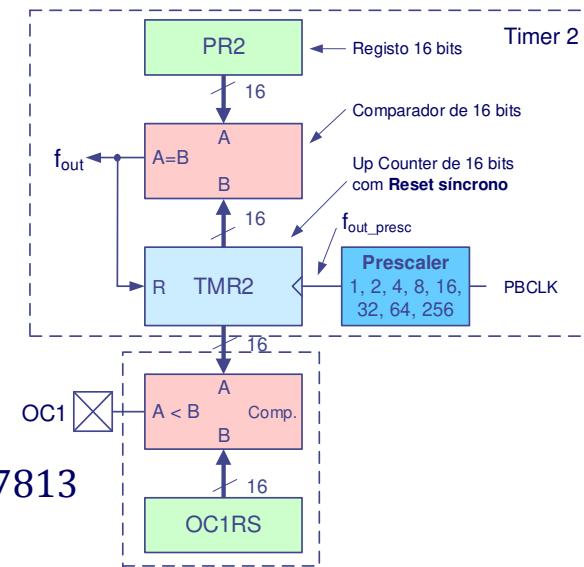
$$OC1RS = \frac{(PR2 + 1) * dutyCycle}{100} = \frac{(39062 + 1) * 20}{100} = 7813$$

3.1 Alternativa ao cálculo de OC1RS:

$$t_{ON} = 0.2 * T_{out} = \frac{0.2}{8} = 25 \text{ ms} \quad \text{tempo a 1 (t}_{ON}\text{) do sinal de saída}$$

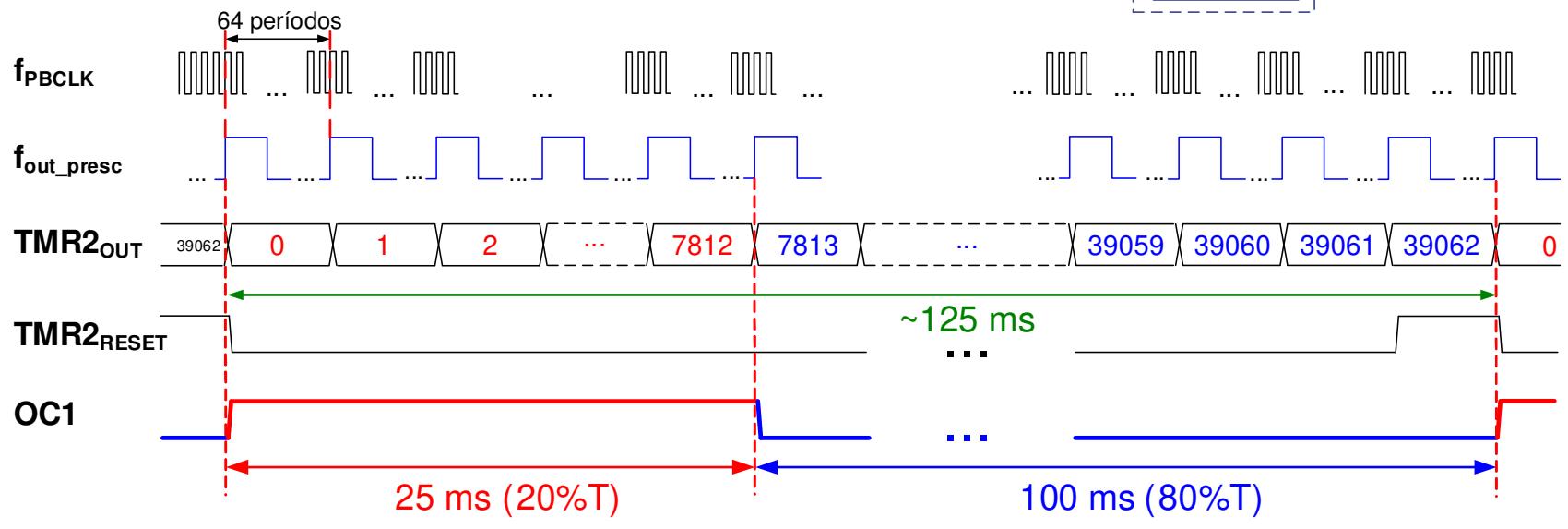
$$T_{IN} = \frac{1}{f_{out\_presc}} = \frac{1}{312500} = 3.2 \text{ } \mu\text{s} \quad \text{período do sinal à entrada do } timer$$

$$OC1RS = \frac{t_{ON}}{T_{IN}} = \frac{25 * 10^{-3}}{3.2 * 10^{-6}} = 7813$$



# Exercício (comportamento temporal)

- Settings:
  - Pre-scale factor = 64
  - PR2 value = 39062
  - OC1RS = 7813
- $f_{PBCLK} = 20 \text{ MHz}$
- $f_{out\_presc} = 312.5 \text{ KHz}$
- $TMR2(OC1)_{\text{period}} = 125 \text{ ms} (f=8\text{Hz})$
- Duty-cycle = 20%



# PIC32 – Resolução do sinal PWM

- A resolução de um sinal PWM dá uma medida do número de níveis com que se pode variar o *duty-cycle* do sinal
- Pode ser definida como:
  - **Resolução =  $\log_2 (T_{\text{PWM}} / T_{\text{IN}})$**
  - em que  $T_{\text{PWM}}$  é o período do sinal PWM gerado e  $T_{\text{IN}}$  é o período do sinal à entrada do gerador de PWM
- Para o caso do PIC32:
  - **Resolução =  $\log_2 ( T_{\text{PWM}} / (T_{\text{PBCLK}} * \text{Prescaler}) )$** , ou, mais simplesmente:
  - **Resolução =  $\log_2 ( PRx + 1 )$**
- Exercícios:
  - determine o valor das constantes PRx e OCnRS para a geração de um sinal com uma frequência de 100 Hz e 25% de *duty-cycle*, supondo PBCLK = 20 MHz, maximizando a resolução do sinal PWM; determine o valor das constantes para um *duty-cycle* de 80%
  - determine a resolução do sinal PWM que obteve; determine a resolução do sinal de PWM do exemplo do slide anterior

# Exercícios

1. Pretende-se gerar um sinal com uma frequência de 85 Hz.  
Usando o Timer T2 e supondo PBCLK = 50 MHz:
  - calcule o valor mínimo da constante de divisão a aplicar ao *prescaler* e indique qual o valor efetivo dessa constante
  - calcule o valor da constante PR2
2. Repita o exercício anterior, supondo que se está a usar o Timer T1
3. Pretende-se gerar um sinal com uma frequência de 100 Hz e 25% de "duty-cycle". Usando o módulo "output compare" OC5 e como base de tempo o Timer T3 e supondo ainda PBCLK = 40 MHz:
  - determine o valor efetivo da constante de *prescaler* que maximiza a resolução do sinal PWM
  - determine o valor das constantes PR3 e OC5RS
  - determine a resolução do sinal de PWM obtido

# Anexos

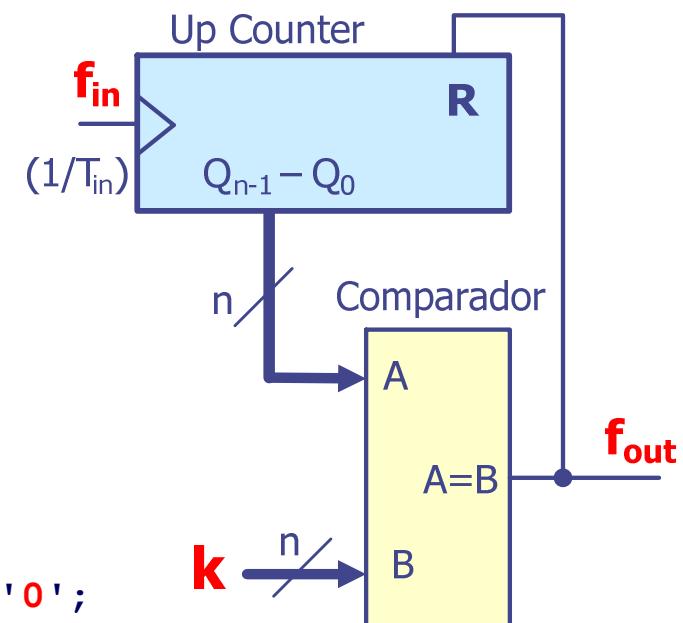
- Exemplos de modelação em VHDL de:
  - divisor de frequência genérico
  - *timer* tipo A do PIC32
  - gerador de PWM do PIC32

# Exemplo de um divisor de frequência (VHDL)

```

entity FreqDivider is
    generic(N      : positive := 16);
    port( fin     : in  std_logic;
          k       : in  std_logic_vector(N-1 downto 0);
          fout    : out std_logic);
end FreqDivider;
architecture synchronous of FreqDivider is
    signal s_counter, s_k : natural range 0 to ((2 ** N)-1) := 0;
begin
    s_k <= to_integer(unsigned(k));
    process(fin)
    begin
        if(rising_edge(fin)) then
            if(s_counter = s_k) then
                s_counter <= 0;
            else
                s_counter <= s_counter + 1;
            end if;
        end if;
    end process;
    fout <= '1' when s_counter = s_k else '0';
end synchronous;

```



# PIC32 – Modelação em VHDL do *timer* tipo A

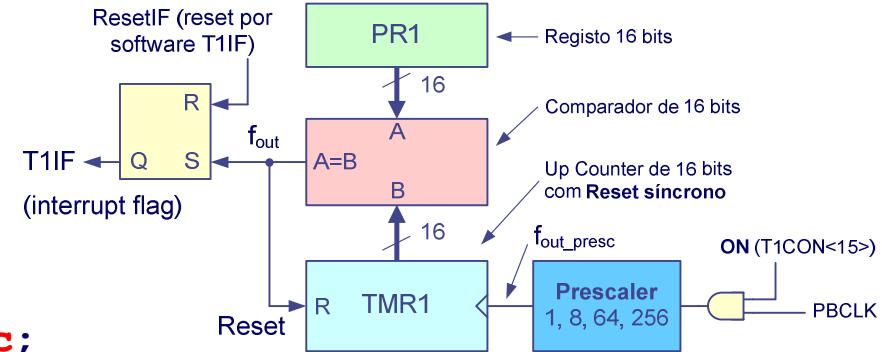
```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity TimerPIC32_A is
    port( PBCLK      : in std_logic;
          T1On       : in std_logic;
          ResetIF    : in std_logic;
          Presc     : in std_logic_vector(1 downto 0);
          PR1       : in std_logic_vector(15 downto 0);
          T1IF      : out std_logic);
end TimerPIC32_A;

architecture synchronous of TimerPIC32_A is
    signal s_counter, s_pr1 : natural range 0 to (2**16-1) := 0;
    signal s_precounter : unsigned(7 downto 0);
    signal s_foutPresc  : std_logic;
begin
    -- (continua)

```

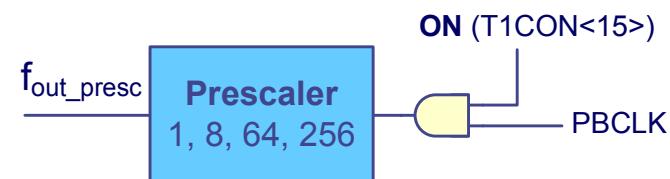


# PIC32 – Modelação em VHDL do *timer* tipo A

```
-- Prescaler (divide PBCLK frequency by 1, 8, 64 or 256)
process (PBCLK, T1On, s_precounter, s_kprescale)
begin
    if(rising_edge(PBCLK)) then
        if(T1On = '1') then
            s_precounter <= s_precounter + 1;
        end if;
    end if;
end process;

s(foutPresc) <= PBCLK when Presc = "00" else -- Div 1
                    s_precounter(2) when Presc = "01" else -- Div 8
                    s_precounter(5) when Presc = "10" else -- Div 64
                    s_precounter(7);                      -- Div 256
```

-- (continua)

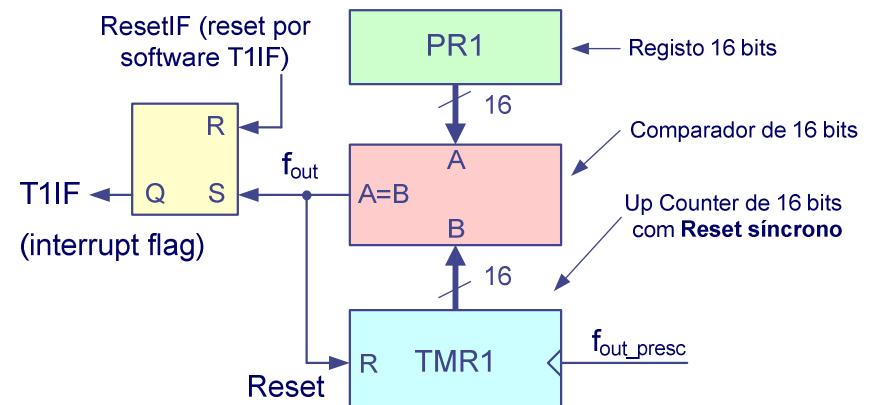


# PIC32 – Modelação em VHDL do *timer* tipo A

```

-- Timer (input clock is the signal produced by the prescaler)
s_pr1 <= to_integer(unsigned(PR1));
process(s(foutPresc, ResetIF)
begin
    if(rising_edge(s(foutPresc)) then
        if(s_counter = s_pr1) then
            T1IF <= '1'; s_counter <= 0;
        else
            s_counter <= s_counter + 1;
        end if;
    end if;
    if(ResetIF = '1') then
        T1IF <= '0';
    end if;
end process;
end synchronous;

```

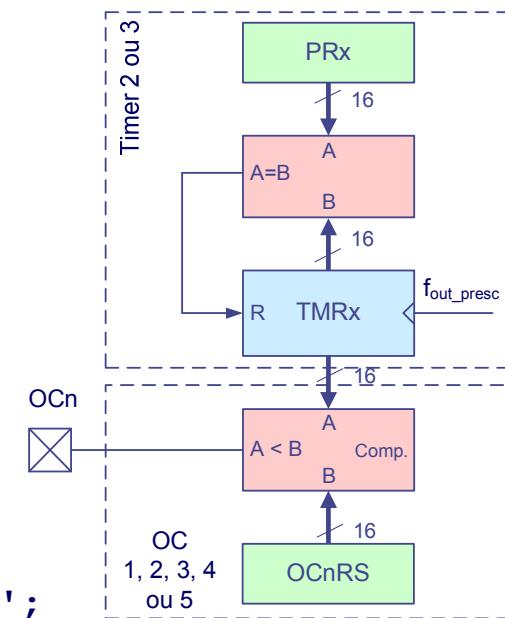


# PIC32 – Modelação em VHDL do gerador de PWM

```

entity FreqDividerDC is
    port( foutPresc : in std_logic;
          PRx      : in std_logic_vector(15 downto 0);
          OCnRS   : in std_logic_vector(15 downto 0);
          OCn     : out std_logic);
end FreqDividerDC;
architecture synchronous of FreqDividerDC is
    signal s_counter : natural range 0 to (2**16-1) := 0;
    signal s_prx, s_ocnrs : natural range 0 to (2**16-1);
begin
    s_ocnrs <= to_integer(unsigned(OCnRS));
    s_prx <= to_integer(unsigned(PRx));
    process(foutPresc)
    begin
        if(rising_edge(foutPresc)) then
            if(s_counter = s_prx) then
                s_counter <= 0;
            else
                s_counter <= s_counter + 1;
            end if;
        end if;
    end process;
    OCn <= '1' when s_counter < s_ocnrs) else '0';
end synchronous;

```



## Aulas 12 e 13

- Barramentos paralelo *vs* barramentos série
- Barramentos série
  - Princípio de funcionamento
  - Sincronização de relógio entre transmissor e receptor
  - Modos de transmissão de dados: transmissão orientada ao bit, transmissão orientada ao byte
  - Topologias de ligação
  - Elementos de uma ligação série

José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

# Introdução

- Barramentos: interligação dos blocos de um sistema de computação
  - CPU, memória, unidades de I/O
- Tipos de dispositivos ligados a um barramento:
  - **Bus Master** – Dispositivo que pode iniciar e controlar uma transferência de dados (exemplos: Processador, Módulo de I/O com DMA)
  - **Bus Slave** – Dispositivo que só responde a pedidos de transferências de dados, i.e., não tem capacidade para iniciar uma transferência (exemplos: Memória, Módulo de I/O sem DMA)
- **Barramento de um só Master**: só há um dispositivo no barramento com capacidade para iniciar e controlar transferências de informação
- **Barramento Multi-Master**: mais que um dispositivo capaz de iniciar e controlar transferências de informação (exemplos: vários CPUs, 1 ou mais controladores de DMA, um ou mais módulos de I/O com DMA)

# Introdução

- **Barramentos paralelo:** os dados são transmitidos em paralelo (através de N Linhas). Incluem:
  - **Barramento de dados:**
    - Suporta a transferência de informação entre os blocos
    - O número de linhas (largura do barramento) determina quantos bits podem ser transferidos simultaneamente; a largura do barramento é um fator determinante no desempenho do sistema
  - **Barramento de endereços:**
    - Especifica a origem/destino da informação
    - O número de linhas define a dimensão do espaço de endereçamento (determina a capacidade máxima de memória que o sistema pode ter:  $2^N$  palavras, sendo N o número de bits do barramento de endereços)
  - **Barramento de controlo:**
    - Conjunto de sinais que especificam operações, sinalizam eventos, efetuam pedidos, ...

# Introdução

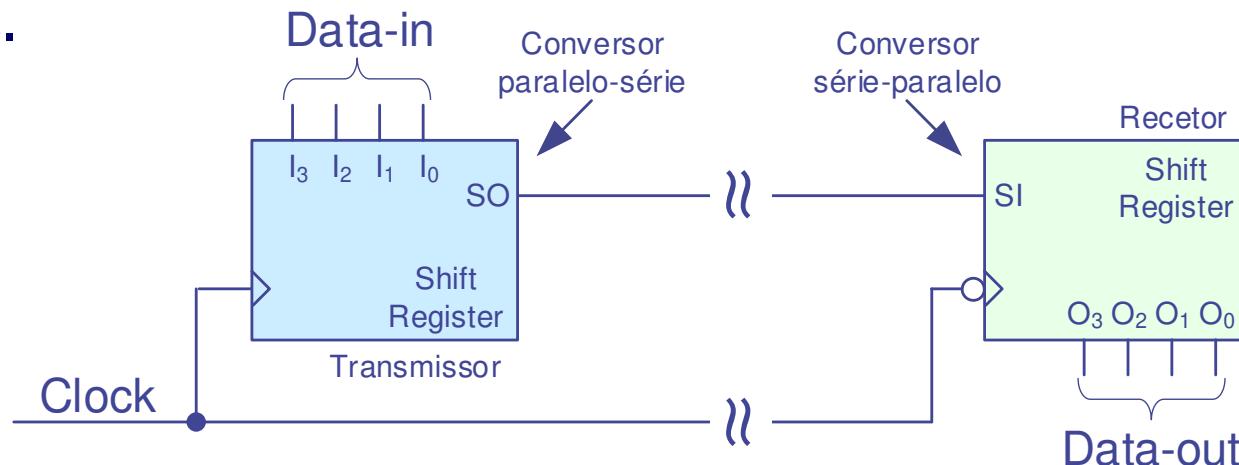
- A transmissão paralela, com relógio comum, a débitos elevados coloca problemas de vária ordem, nomeadamente:
  - Controlo do tempo de "skew" das linhas do barramento
  - Dificuldade em anular a interferência provocada por fontes de ruído externas
  - Interferência mútua, isto é, entre sinais adjacentes ("crosstalk")
  - Elevado número de fios de ligação e custo associado
  - Fichas de ligação volumosas e caras (possivelmente com contactos dourados)
- **Barramentos série:** os dados são serializados no transmissor, ou seja, transmite-se 1 bit de cada vez (tipicamente 1 bit a cada ciclo de relógio)
  - Comunicação série

# Introdução

- Vantagens dos barramentos série (ao nível físico):
  - Simplicidade de ligação de cablagem
  - Diminuição de custos de interligação
  - Possibilidade de transmissão a distâncias elevadas (em par diferencial)
  - Débito elevado
- Tipos de comunicação série
  - **Simplex**: comunicação apenas num sentido (TX -> RX); usada, por exemplo, em telemetria, para leitura remota de sensores
  - **Half-duplex**: comunicação nos dois sentidos, mas apenas um de cada vez (é usada uma só linha)
  - **Full-duplex**: Comunicação simultânea nos dois sentidos (são usadas duas linhas)

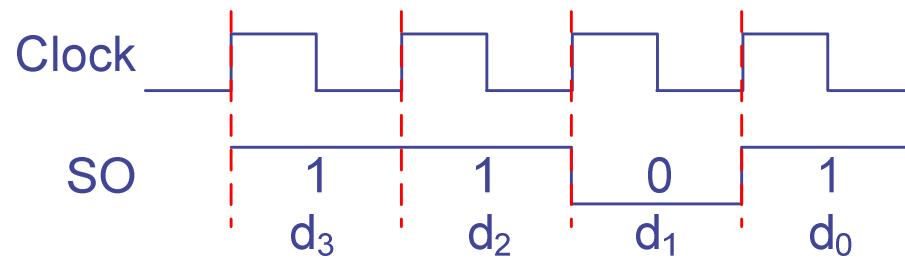
# Introdução

- Diz-se que se está na presença de um barramento ou interface série sempre que exista uma só "linha" (suporte) para transferência de dados.



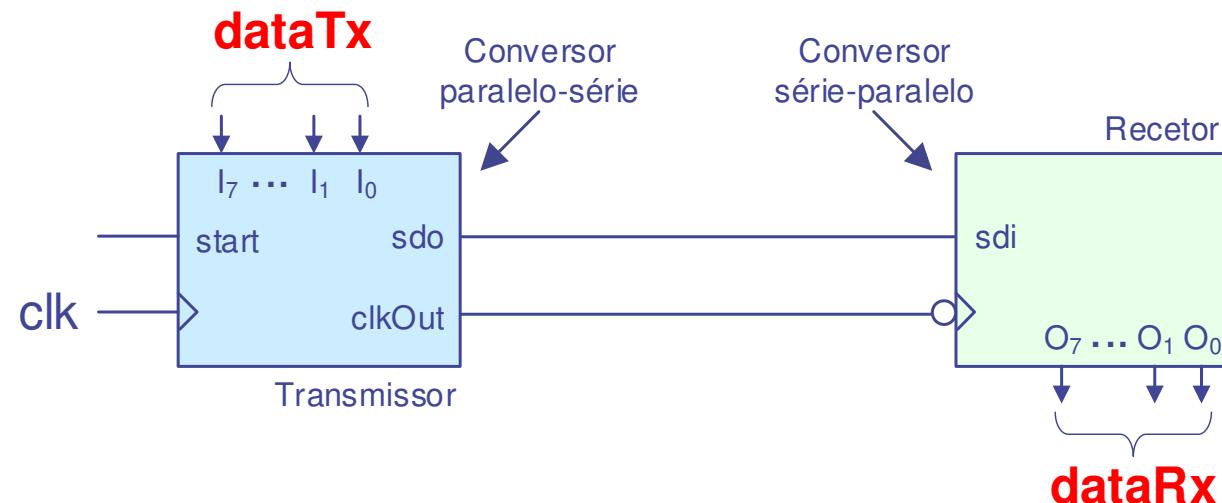
Para transmissão bidirecional podem existir 2 "linhas" separadas, uma para transmissão e outra para receção

- Exemplo  
(Data-in = 1101)

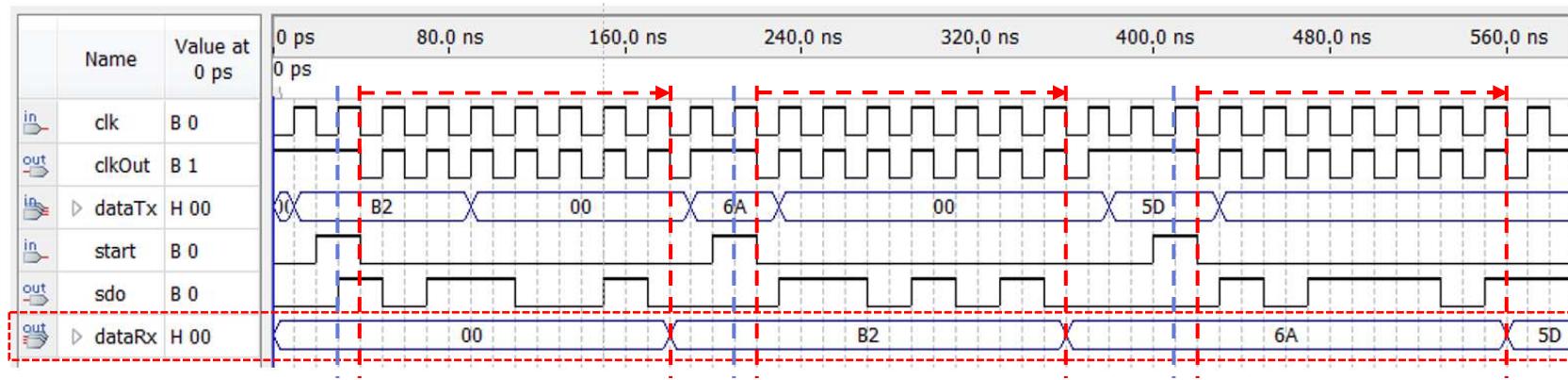


# Introdução

- Exemplo em que o transmissor gera o sinal de relógio (o sinal "start" dá início à transmissão)

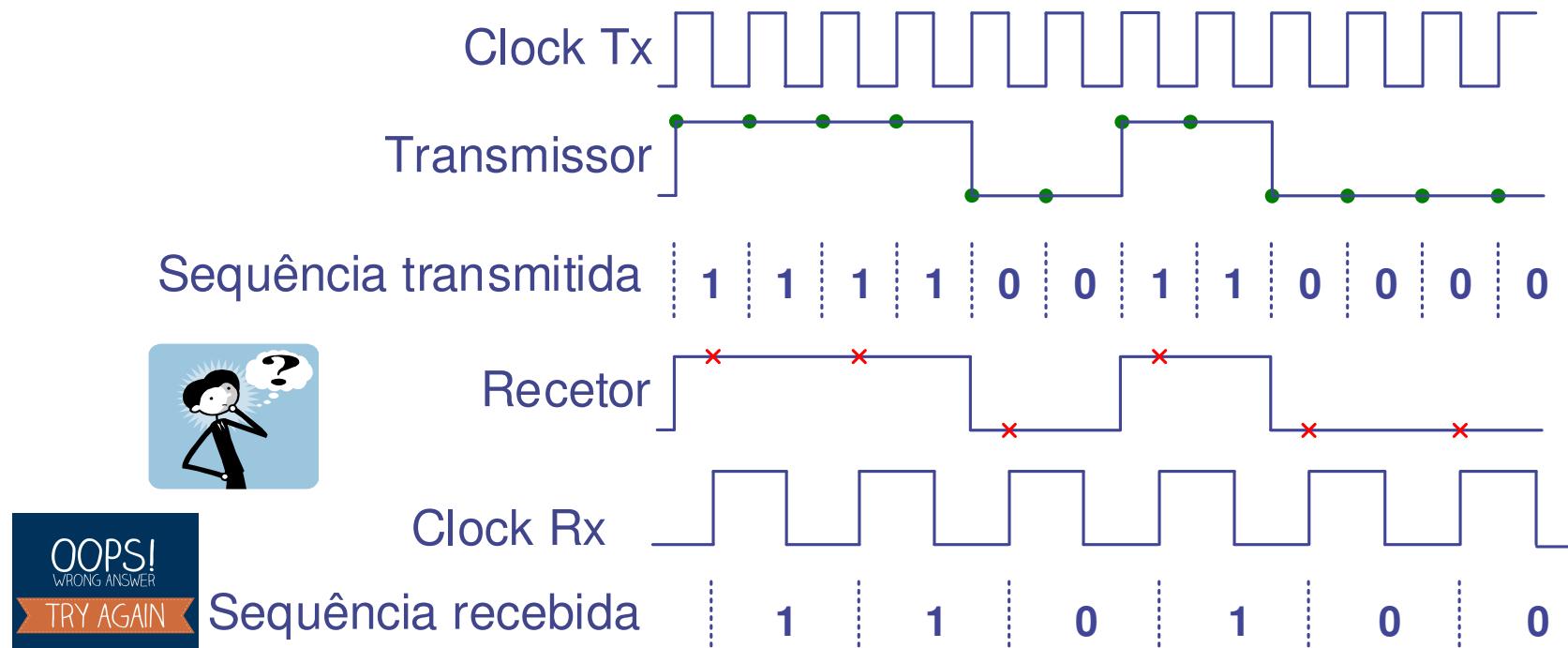


- Exemplo com transmissão da sequência: 0xB2, 0x6A, 0x5D



# Sincronização entre transmissor e recetor

- O sincronismo é obtido através da utilização do mesmo relógio no transmissor e no recetor, ou de relógios independentes que terão que estar sincronizados durante a transmissão



- Caso sejam distintos, os relógios do Tx e do Rx têm de estar sincronizados para que a amostragem do sinal seja realizada nos instantes corretos

# Sincronização entre transmissor e recetor

- **Transmissão Síncrona**

- O sinal de relógio é transmitido de forma explícita através de um sinal adicional, ou na codificação dos dados
- Os relógios do transmissor e do recetor têm de se manter sincronizados
- Quando o relógio não é explicitamente transmitido, o relógio do recetor é recuperado a partir das transições de nível lógico na linha de dados

- **Transmissão Assíncrona**

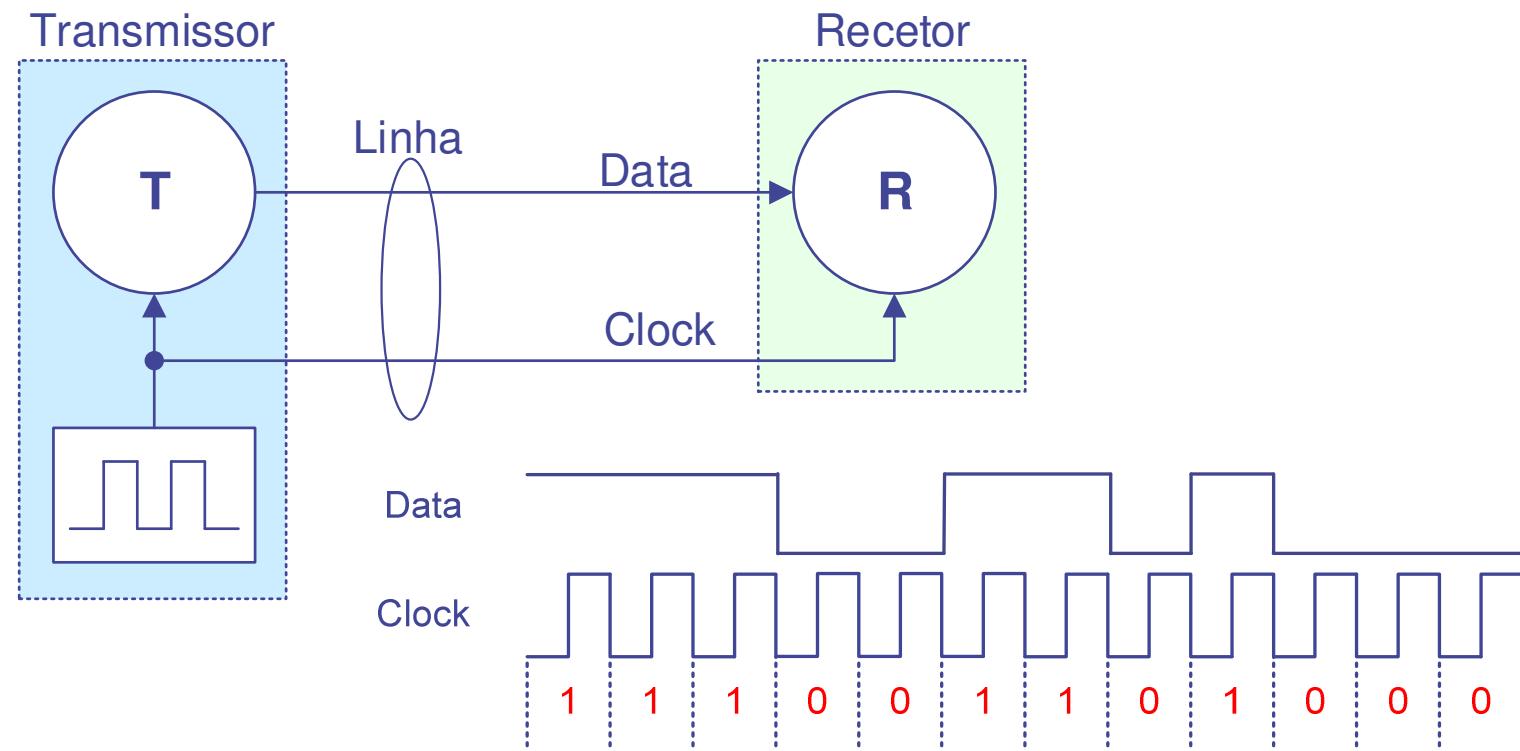
- Não é usado relógio na transmissão, nem há recuperação do relógio na receção
- É necessário acrescentar bits para sinalizar o princípio e o fim da transmissão (e.g. start bit, stop bit), que permitam ao recetor proceder à amostragem do sinal recebido, com o menor erro temporal possível

# Técnicas de sincronização do relógio

- Transmissão síncrona
  - **Relógio explícito do transmissor**
    - Exemplo: SPI
  - **Relógio explícito do receptor**
  - **Relógio explícito mutuamente-sincronizado**
    - Exemplo: I2C
  - **Relógio codificado ("self-clocking")**
    - Exemplo: USB, Ethernet
- Transmissão assíncrona
  - **Relógio implícito**
    - Exemplo: RS-232, CAN

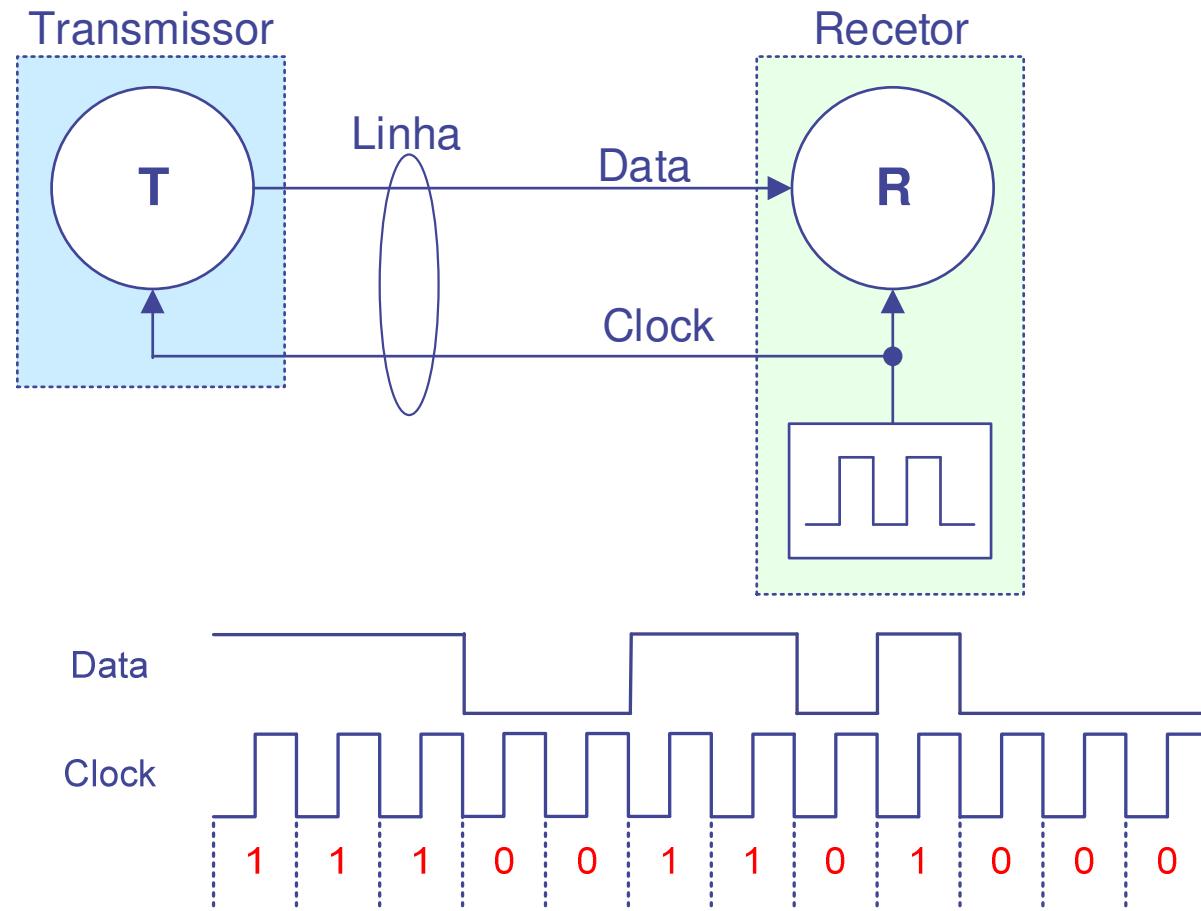
# Sincronização de relógio

- **Relógio explícito do transmissor**
- O transmissor envia os dados e informação de relógio em linhas separadas
- A linha de relógio pode ser vista como um sinal "Valid"



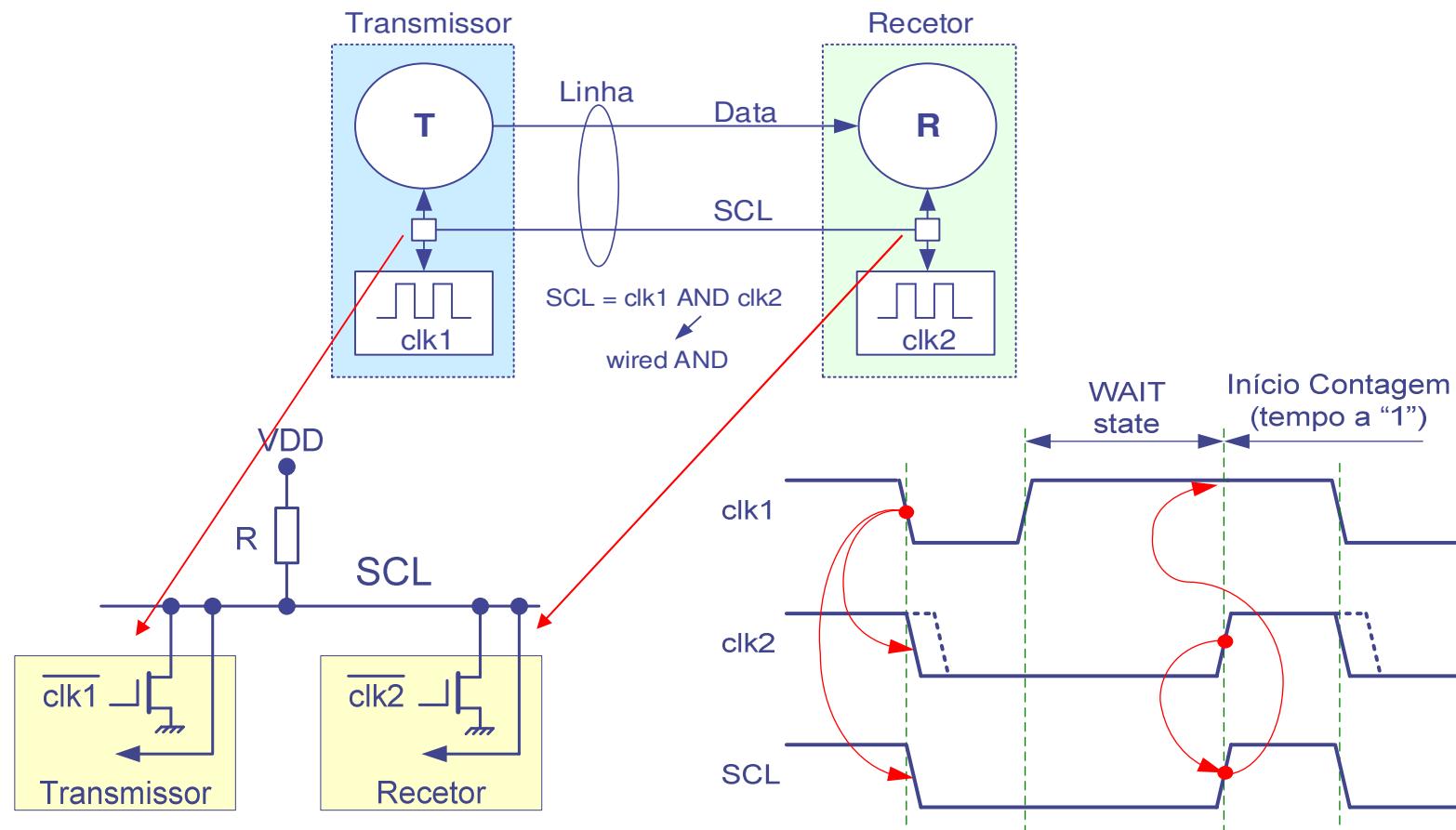
# Sincronização de relógio

- **Relógio explícito do receptor**
- Semelhante ao anterior, sendo o receptor a gerar o sinal de relógio



# Sincronização de relógio

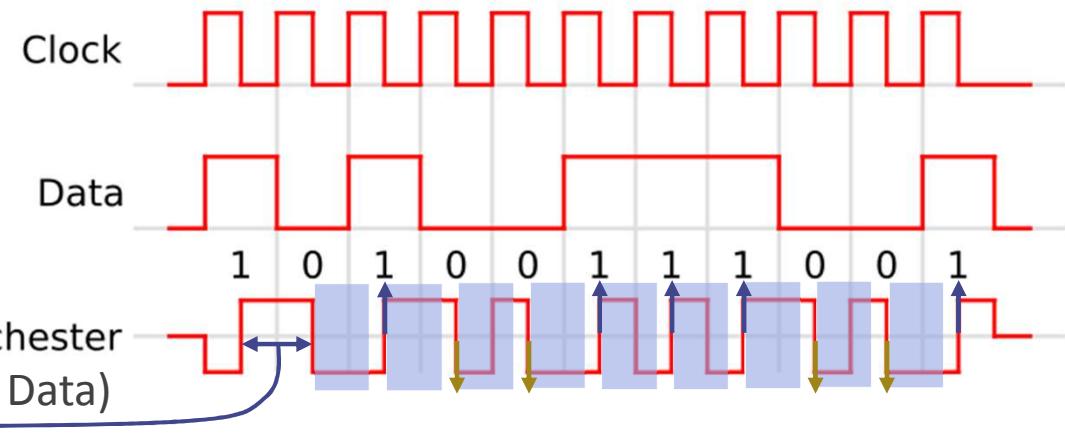
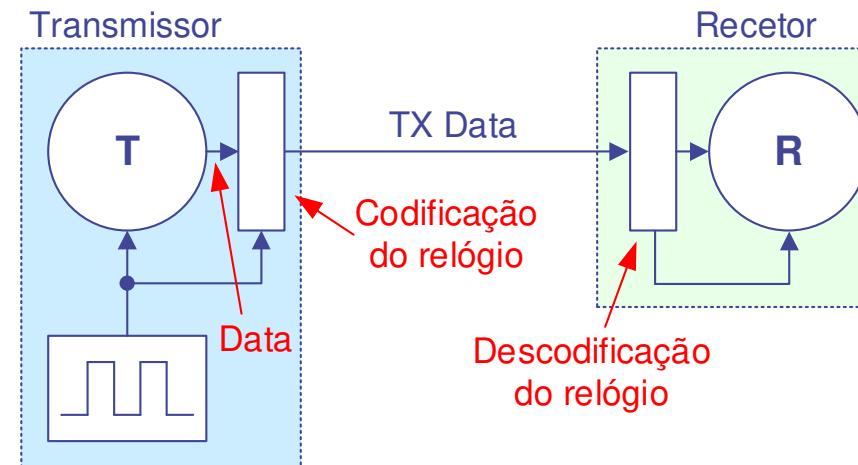
- Relógio explícito mutuamente-sincronizado ("clock stretching")
- Transmissor e Recetor sincronizam-se mutuamente



# Sincronização de relógio

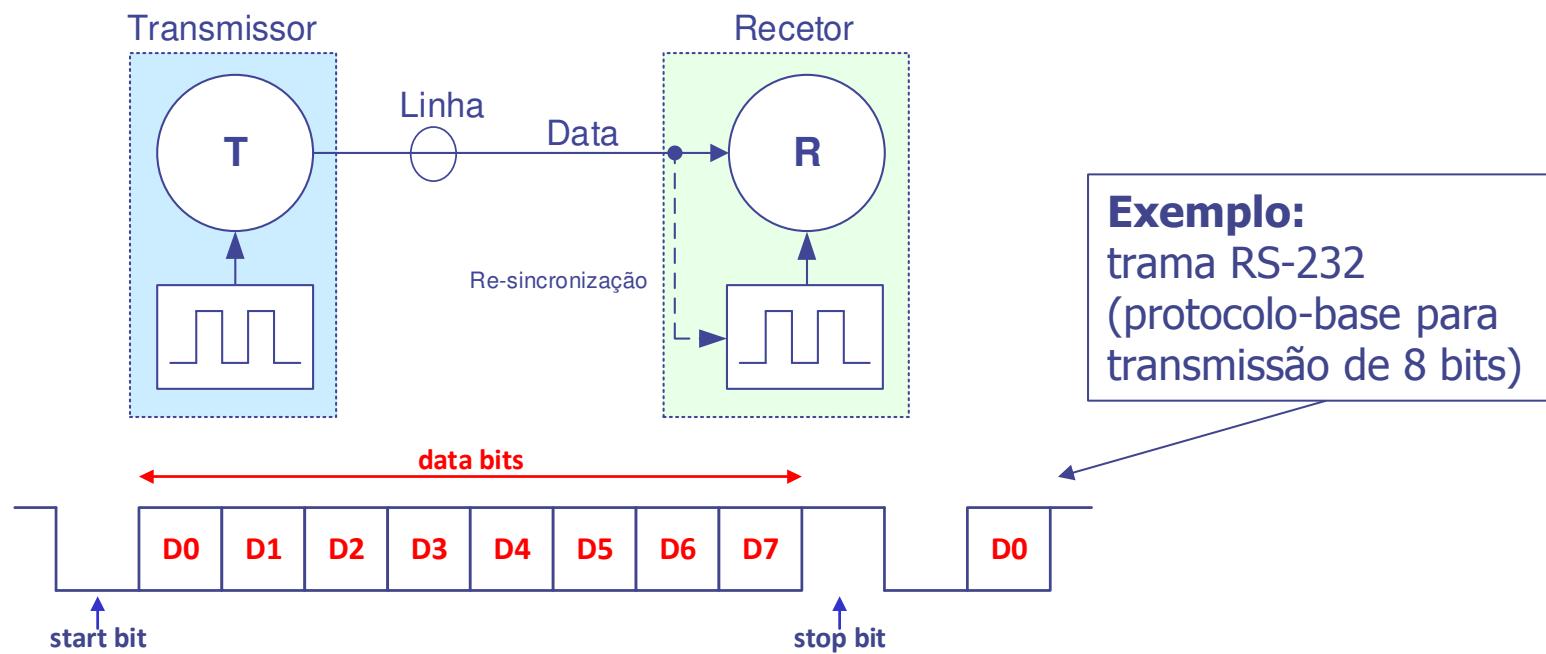
- **Relógio codificado**
- Relógio enviado, em forma codificada, conjuntamente com os dados

- Exemplo: codificação Manchester
- A informação transmitida resulta do "ou exclusivo" entre o bit a transmitir e o sinal de relógio



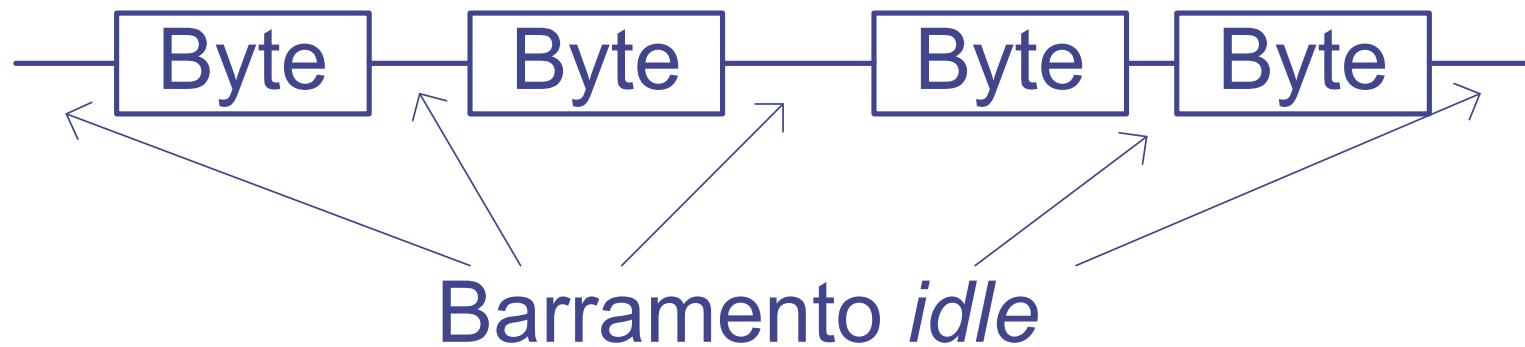
# Sincronização de relógio (transmissão assíncrona)

- **Relógio implícito**
- Os relógios são locais (i.e. não há comunicação do relógio)
- O relógio do receptor é sincronizado ocasionalmente com o do transmissor por meio da receção de símbolos específicos
- Entre instantes de sincronização o desvio dos relógios depende da estabilidade/precisão dos relógios do transmissor e do receptor



## Transmissão de dados – transmissão orientada ao Byte

- O envio de um byte é a operação atómica (indivisível) do barramento
- Cada byte é encarado como independente dos restantes
- Não há restrições temporais para a transmissão em sequência de 2 bytes



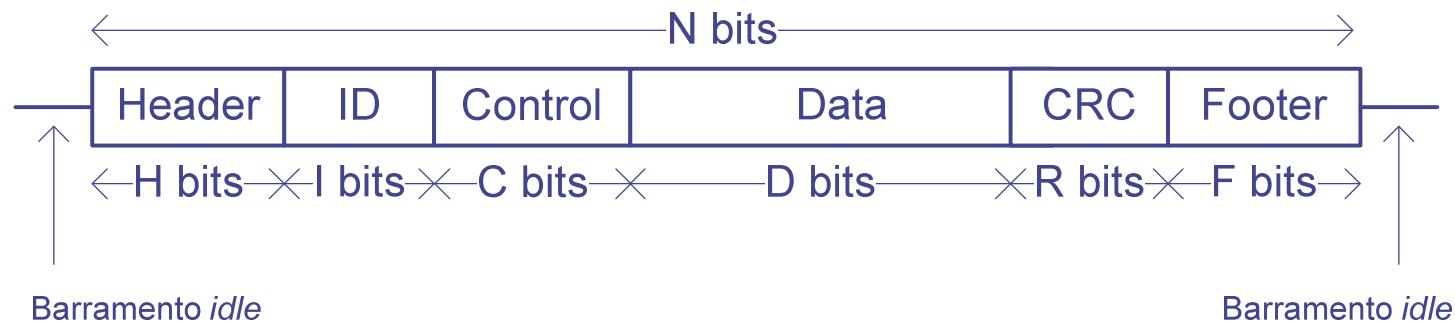
- Alguns bytes podem estar reservados para estruturar a informação
- Exemplo de transmissão orientada ao byte: RS232

# Transmissão de dados – transmissão orientada ao bit

- A informação é organizada em tramas (sequência de bits intercalada entre duas situações de meio livre)
- As tramas são constituídas por um símbolo de sincronização (delimitador, constituído por 1 ou mais bits) seguido por uma sequência de bits de comprimento arbitrário
- As tramas podem conter campos com diferentes funções:
  - Sincronização: sinalização de início e de fim da trama
  - Arbitragem de acesso ao meio (em barramentos multi-master)
  - Identificação. Diversas formas possíveis:
    - Quem produz
    - Qual o destino
    - Identificação da informação que circula na trama
    - ...
  - Quantidade de informação transmitida
  - Dados
  - Detecção de erros de transmissão

# Transmissão de dados – transmissão orientada ao bit

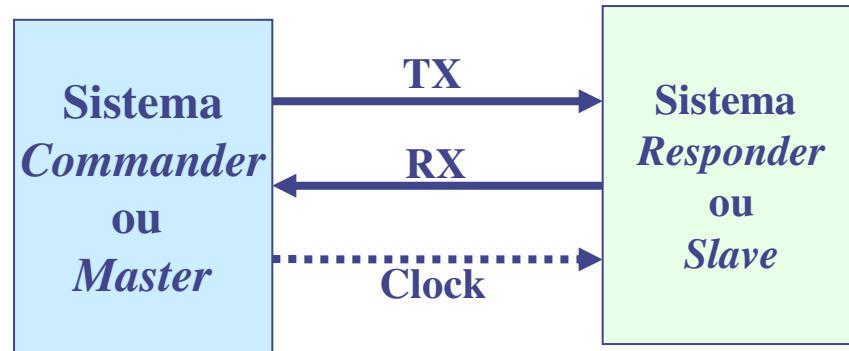
- Exemplo de estrutura de uma trama



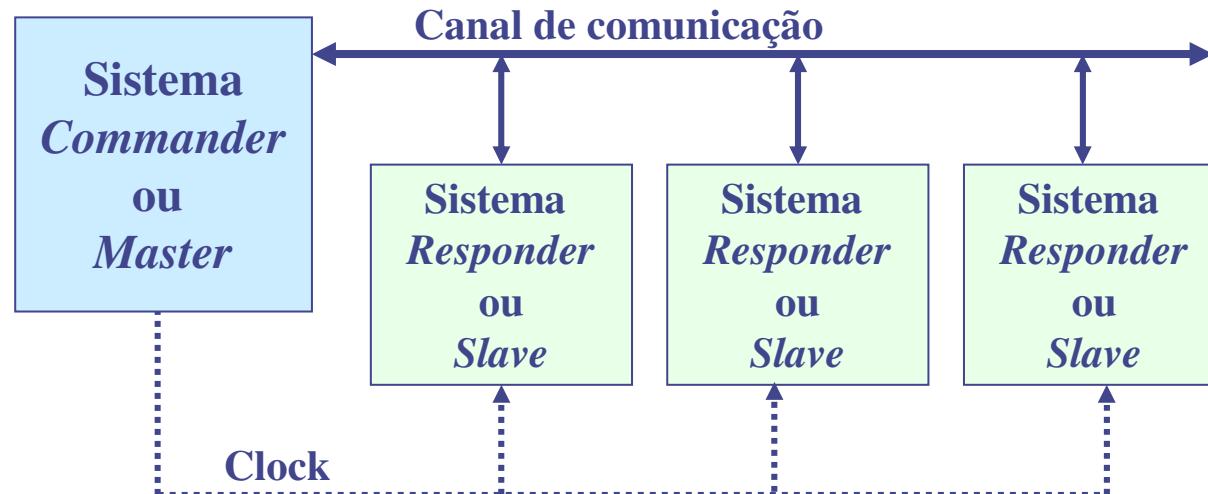
- Exemplo de transmissão orientada ao bit: barramento CAN ("Controller Area Network")
  - "Header" e "footer": delimitadores de início e fim de trama
  - Data: campo de dados
  - CRC ("cyclic redundancy check"): código usado para detetar, no receptor, erros na comunicação
    - Uma forma simples de CRC consiste em somar todos os bytes transmitidos (soma truncada com R bits) – *checksum*
    - O receptor soma todos os bytes recebidos e compara com a soma recebida

# Topologias

- Comunicação ponto a ponto ("half-duplex" ou "full-duplex"):

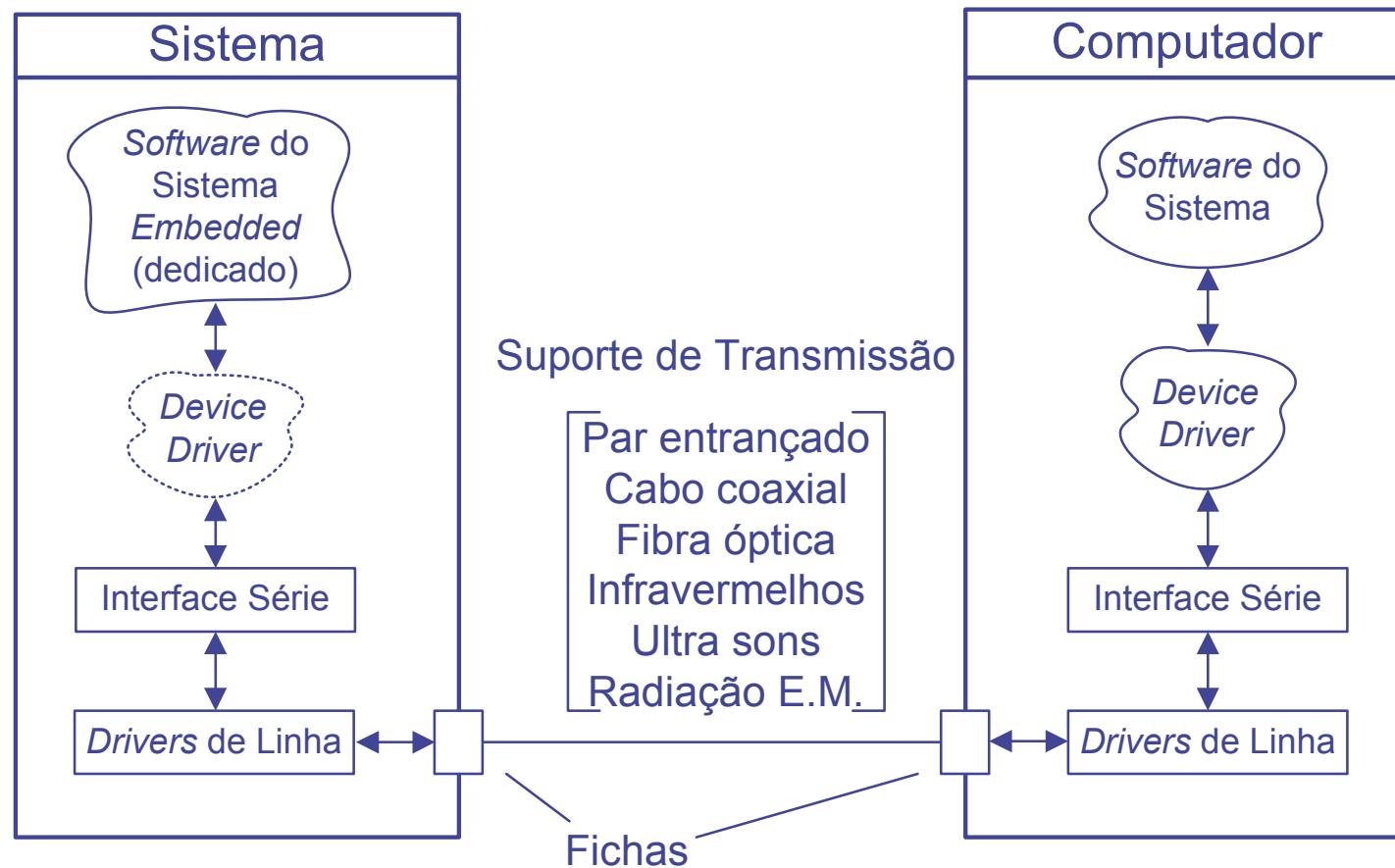


- Comunicação multiponto ("half-duplex"):



# Elementos de uma ligação série

- Exemplo de uma ligação série entre um sistema embutido ("embedded" ou dedicado) e um computador de uso geral (PC)



# Aula 14

- A interface SPI (*Serial Peripheral Interface*)
- Sinalização
- Sequência de operação
- Arquiteturas de ligação
- Tipos de transferências
- Passos de configuração de um *master SPI*

José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

# Introdução

- SPI – sigla para "Serial Peripheral Interface"
- Interface definida inicialmente pela Motorola (Microwire da National Semiconductor é um *subset* do protocolo SPI)
- O SPI é utilizado para comunicar com uma grande variedade de dispositivos:
  - Sensores de diverso tipo: temperatura, pressão, etc.
  - Cartões de memória (MMC / SD)
  - Circuitos: memórias, ADCs, DACs, Displays LCD (e.g. telemóveis), comunicação entre corpo de máquinas fotográficas e as lentes, ...
  - Comunicação entre microcontroladores
- **Ligaçāo a curtas distâncias (dezenas de cm)**

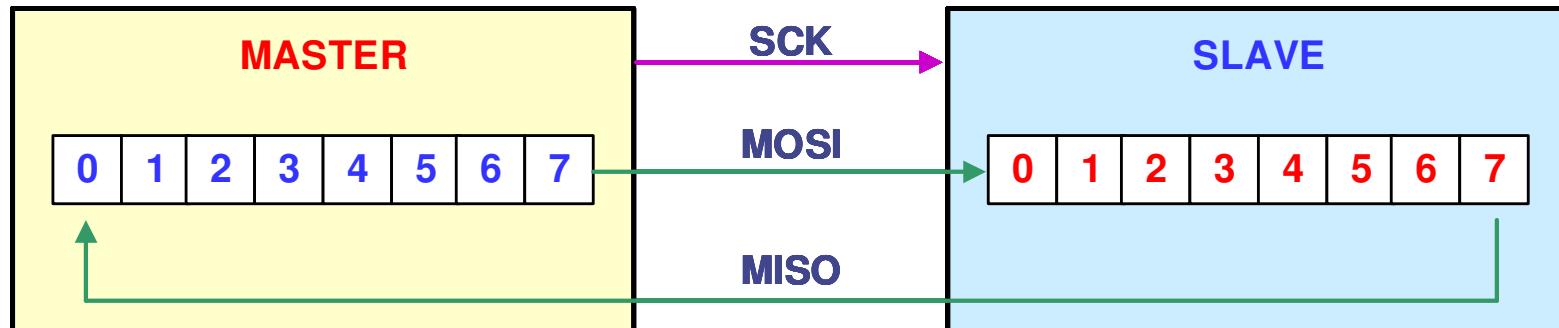
# Descrição geral

- Arquitetura "Master-Slave" com ligação ponto a ponto
- Comunicação bidirecional "full-duplex"
- Comunicação síncrona (relógio explícito do *master*)
  - Relógio é gerado pelo *master* que o disponibiliza para todos os *slaves*
  - Não é exigida precisão ao relógio - os bits vão sendo transferidos a cada transição de relógio. Isto permite utilizar um oscilador de baixo custo no *master* (não é necessário um cristal de quartzo)
- Fácil de implementar por hardware ou por software
- Não são necessários "line drivers" (ou "transceivers") - circuitos de adaptação ao meio de transmissão. Os níveis lógicos correspondem aos da diferença de potencial de alimentação dos dispositivos (e.g. 3.3V)

# Descrição geral

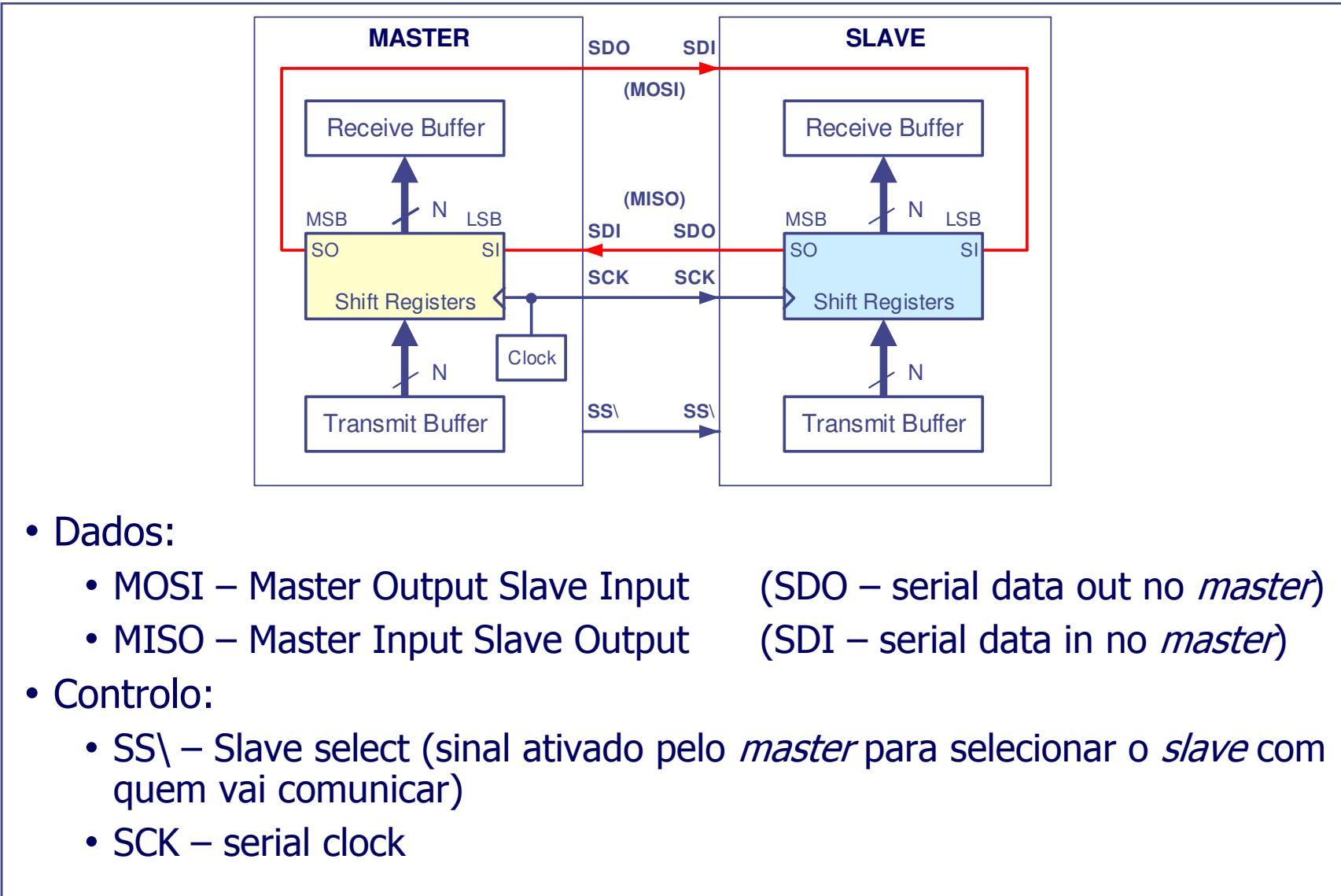
- Arquitetura "Master-Slave"
  - O sistema só pode ter um *master*
  - O *master* é o único dispositivo no sistema que pode controlar o relógio
- Um *master* pode estar ligado a vários *slaves*: para cada comunicação, apenas 1 *slave* é selecionado pelo *master* (daí ligação ponto a ponto)
- O *master* inicia e controla a transferência de dados
- Sinalização:
  - **SCK** – clock
    - Relógio gerado pelo *master* que sincroniza a transmissão/recepção de dados
  - **MOSI** – Master Output Slave Input (SDO no *master*)
    - Linha do *master* para envio de dados para o *slave*
  - **MISO** – Master Input Slave Output (SDI no *master*)
    - Linha do *slave* para enviar dados para o *master*
  - **SS** – Slave select
    - Linha do *master* que seleciona o *slave* com quem vai comunicar

# Descrição geral – esquema de princípio



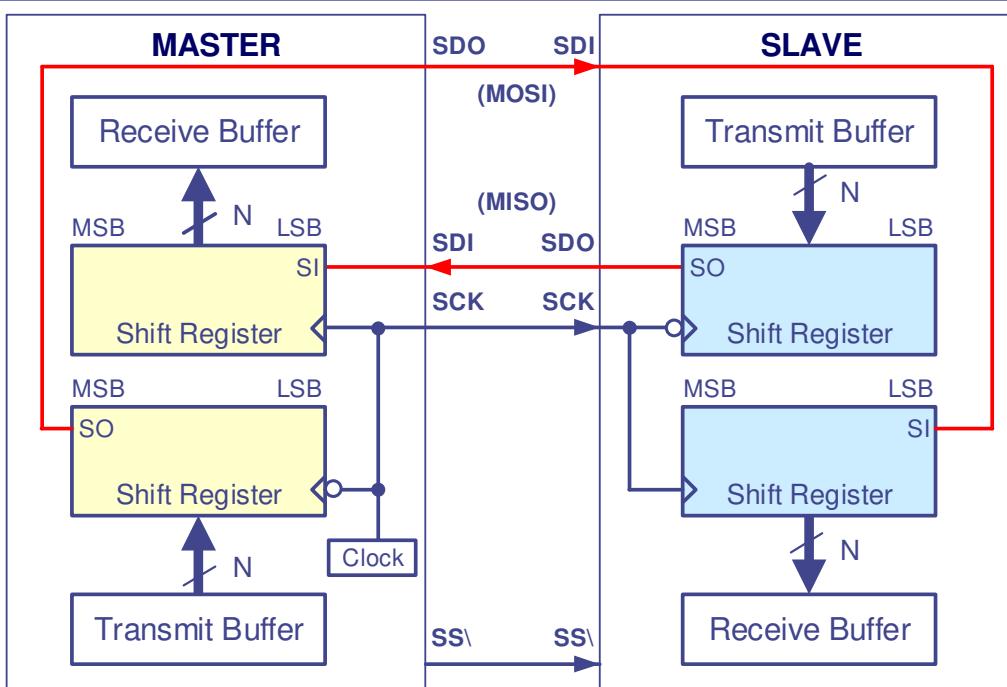
- Transmissão "full-duplex" baseada em dois *shift-registers* (um no *master* e outro no *slave*)
- Em cada ciclo de relógio:
  - O *master* coloca 1 bit na linha MOSI e o *slave* recebe-o
  - O *slave* coloca 1 bit na linha MISO e o *master* recebe-o
- Ao fim de N ciclos de relógio o *master* enviou uma palavra de N bits e recebeu do *slave* uma palavra com a mesma dimensão – "Data Exchange"
- Esta sequência é realizada mesmo quando é pretendida uma comunicação unidirecional

# Sinalização



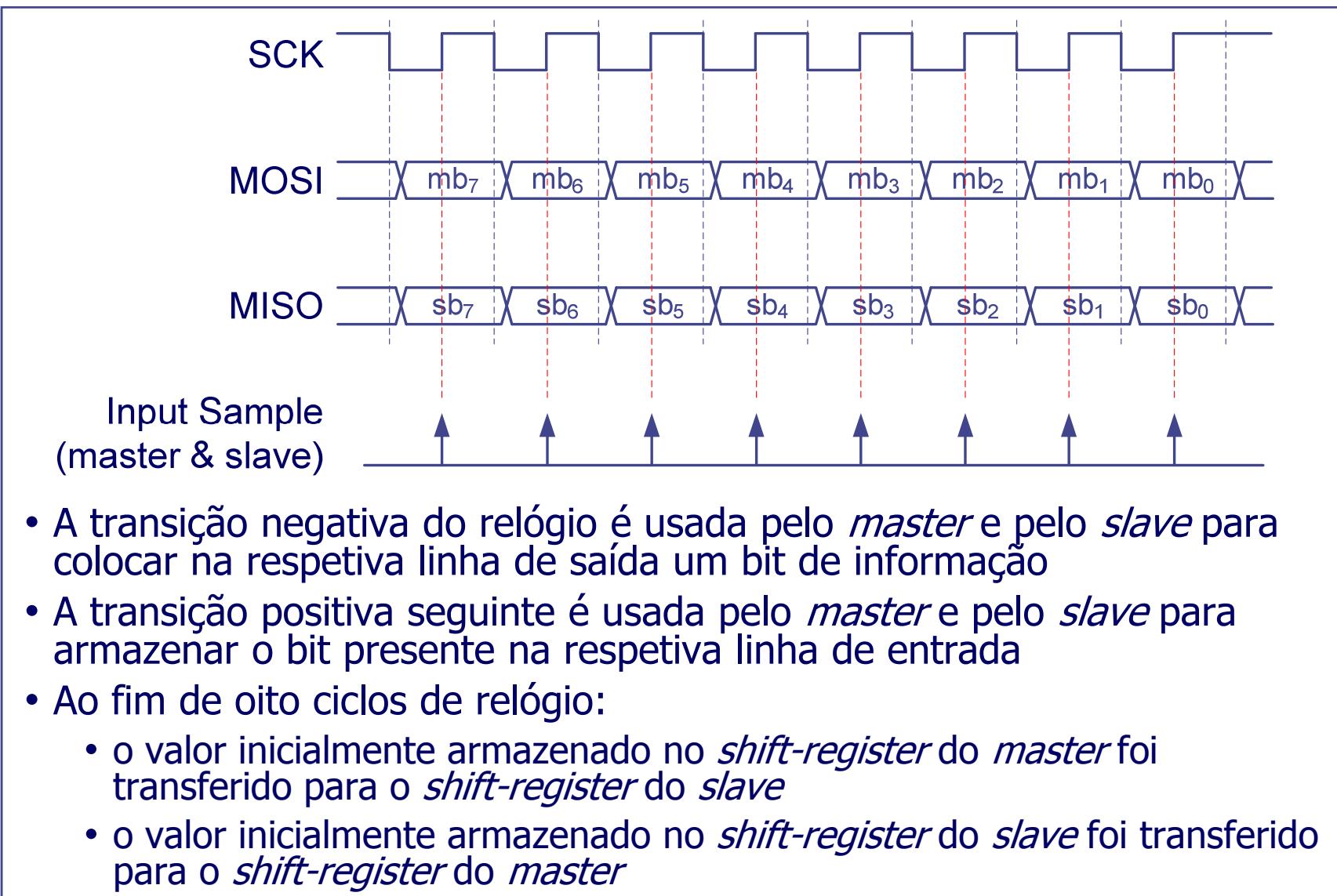
- Dados:
  - MOSI – Master Output Slave Input (SDO – serial data out no *master*)
  - MISO – Master Input Slave Output (SDI – serial data in no *master*)
- Controlo:
  - SS\ – Slave select (sinal ativado pelo *master* para selecionar o *slave* com quem vai comunicar)
  - SCK – serial clock

# Sinalização



- O sinal de relógio tem um "duty-cycle" de 50%
- No exemplo da figura:
  - *master* e *slave* usam a transição negativa do relógio para colocarem 1 bit na linha (*master* na linha MOSI, *slave* na linha MISO)
  - Na transição positiva seguinte, o *master* armazena o valor presente na linha MISO e o *slave* armazena o valor que se encontra na linha MOSI

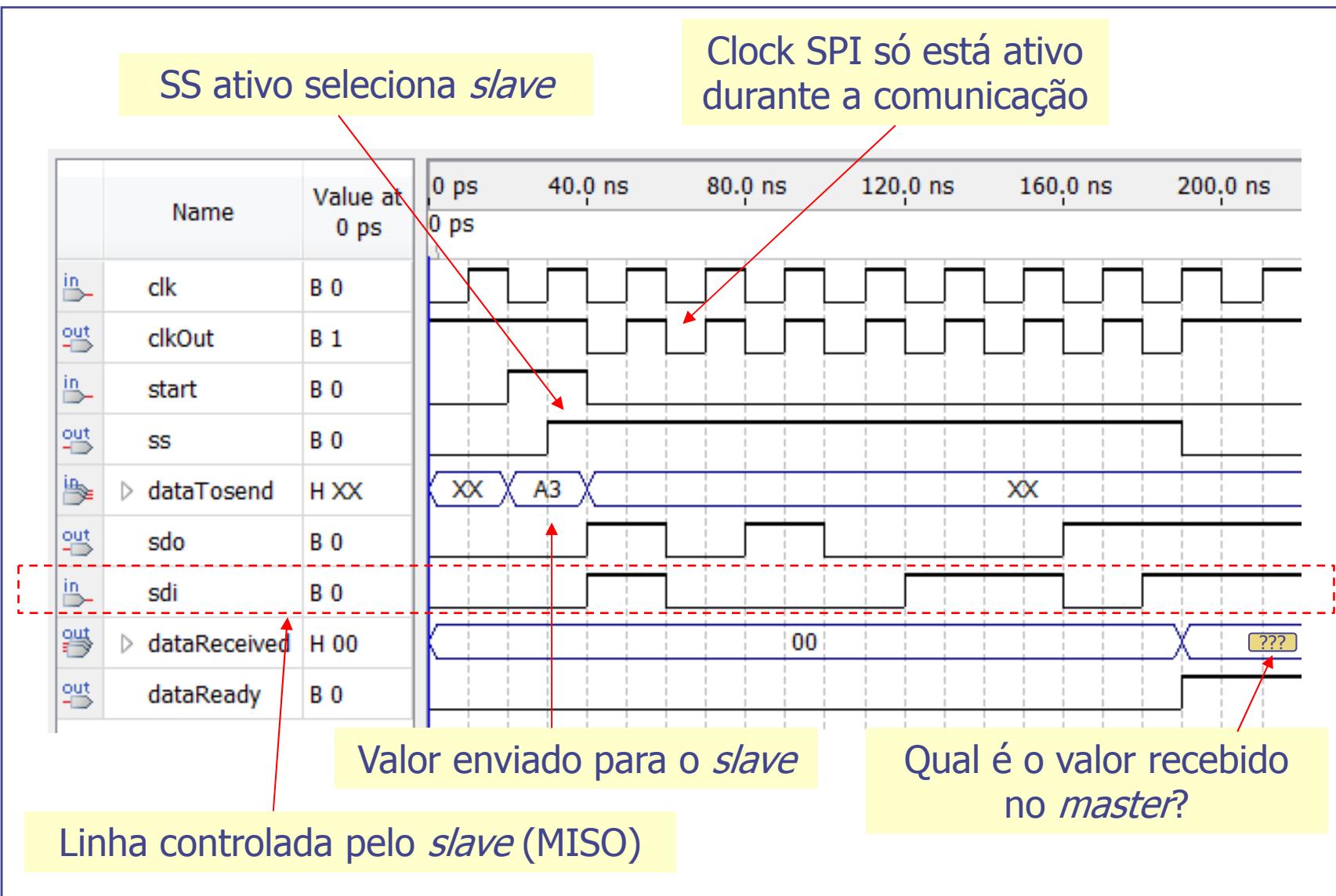
# Operação – exemplo



# Operação

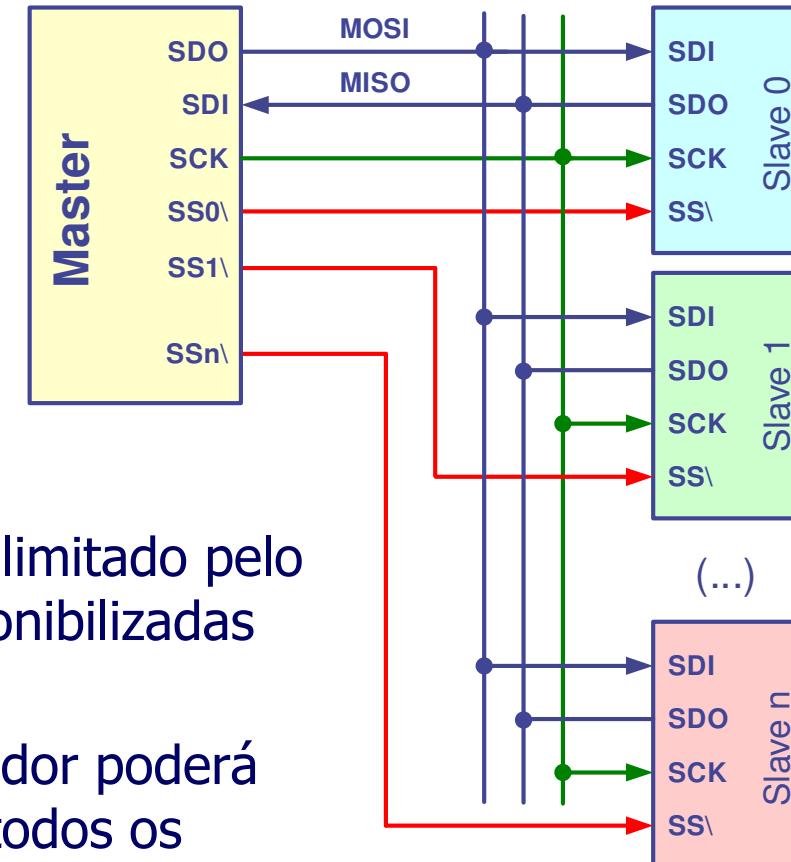
- O *master* ativa a linha SS\ do *slave* com que vai comunicar
- O *master* ativa o relógio que vai ser usado para sincronizar a troca de informação com o *slave* com quem vai comunicar
- Em cada ciclo do relógio, por exemplo na transição positiva
  - O *master* coloca na linha MOSI um bit de informação que é lido pelo *slave* na transição de relógio oposta seguinte
  - O *slave* coloca na linha MISO um bit de informação que é lido pelo *master* na transição de relógio oposta seguinte
- O *master* desativa a linha SS\ e desativa o relógio (que fica estável, por exemplo, no nível lógico 1)
  - Só há relógio durante o tempo em que se processa a transferência
- No final, o *master* e o *slave* trocaram o conteúdo dos seus *shift-registers*

# Simulação de um *master* SPI



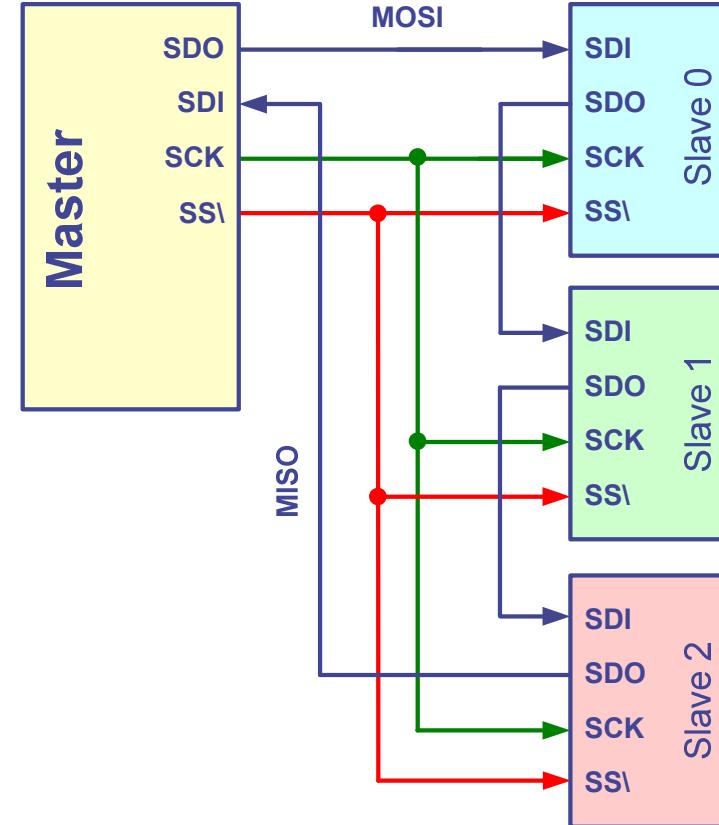
# Arquiteturas de ligação – *slaves* independentes

- Sinais de seleção ("slave select") independentes
- Em cada instante apenas um  $SSx\backslash$  está ativo, isto é, apenas 1 *slave* está selecionado
- Os sinais SDO dos *slaves* (MISO) não selecionados estão em alta impedância
- O número máximo de *slaves* está limitado pelo número de linhas de seleção disponibilizadas pelo *master*
- Alternativamente, o microcontrolador poderá gerar, através de portos digitais, todos os sinais  $SSx\backslash$  necessários para comunicar com os *slaves*, ultrapassando a limitação anterior



# Arquiteturas de ligação – Daisy Chain (cascata)

- Sinal "slave select" comum, SDO/SDI ligados em cascata
- Todos os *slaves* recebem o mesmo sinal de relógio gerado pelo *master*
- A saída de dados de cada *slave* liga à entrada de dados do seguinte
- Para que esta arquitetura funcione o *slave* tem de ser capaz de armazenar uma sequência de N bits enviados durante 1 ciclo de comando e enviar para a sua saída a mesma sequência de N bits durante o ciclo de comando seguinte
  - Enquanto o SS estiver ativo o *slave* ignora o comando recebido e envia-o para a saída DO no ciclo de comando seguinte
  - O *slave* apenas executa o comando quando o sinal SS é desativado



# Tipos de transferências

- O SPI funciona sempre em modo "data exchange", isto é, o processo de comunicação envolve sempre a troca do conteúdo dos *shift-registers* do *master* e do *slave*
- Cabe aos dispositivos envolvidos na comunicação usar ou descartar a informação recebida
- Podem considerar-se os seguintes cenários de transferência:
  - **Bidirecional:** são transferidos dados válidos em ambos os sentidos (*master* → *slave* e *slave* → *master*)
  - **Master → slave (operação de escrita):** *master* transfere dados para o *slave*, e ignora/descarta os dados recebidos
  - **Slave → master (operação de leitura):** *master* pretende ler dados do *slave*; para isso transfere para o *slave* uma palavra com informação irrelevante (por exemplo 0); o *slave* ignora/descarta os dados recebidos

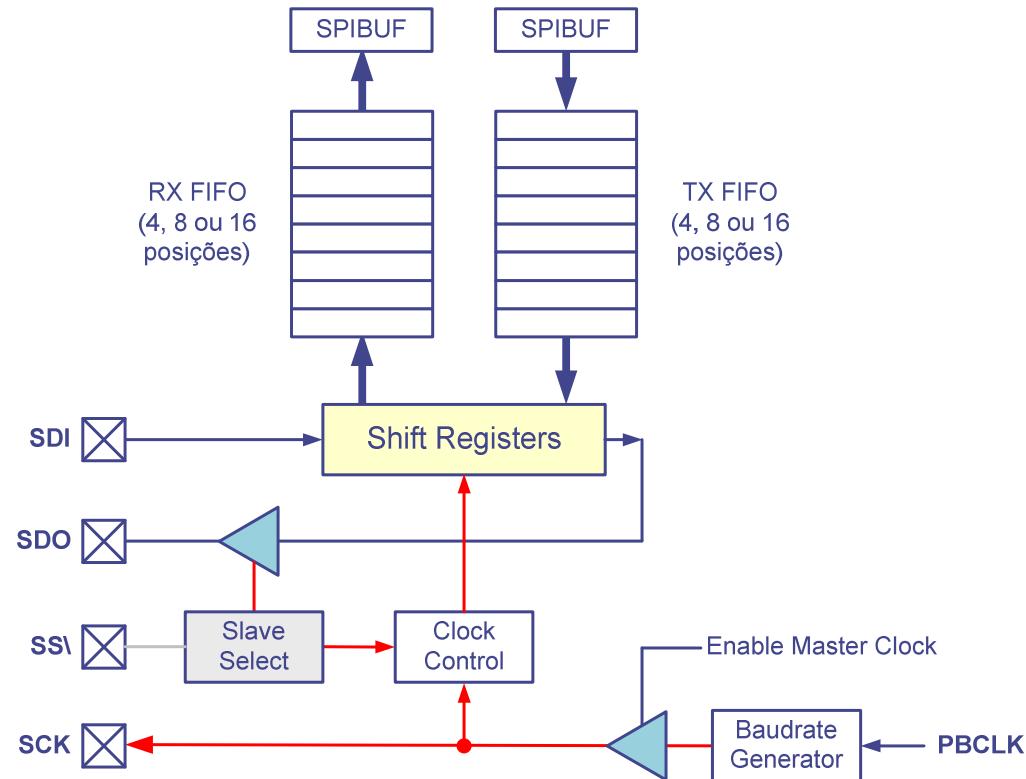
# Configuração de um *master* SPI

- Antes de iniciar a transferência há algumas configurações que são efetuadas no *master* (através do seu modelo de programação) para adequar os parâmetros que definem a comunicação às características do *slave* com que vai comunicar:
  1. Configurar a frequência de relógio
  2. Configurar o nível lógico de repouso ("idle") do sinal de relógio
  3. Especificar qual o flanco do relógio usado para a transmissão (a receção é efetuada no flanco oposto). Esta configuração é feita em função das características do *slave* com o qual o *master* vai comunicar:
    - Transmissão no flanco ascendente (consequentemente, a receção é feita no flanco descendente)
    - Transmissão no flanco descendente (consequentemente, a receção é feita no flanco ascendente)

# Interface SPI no PIC32

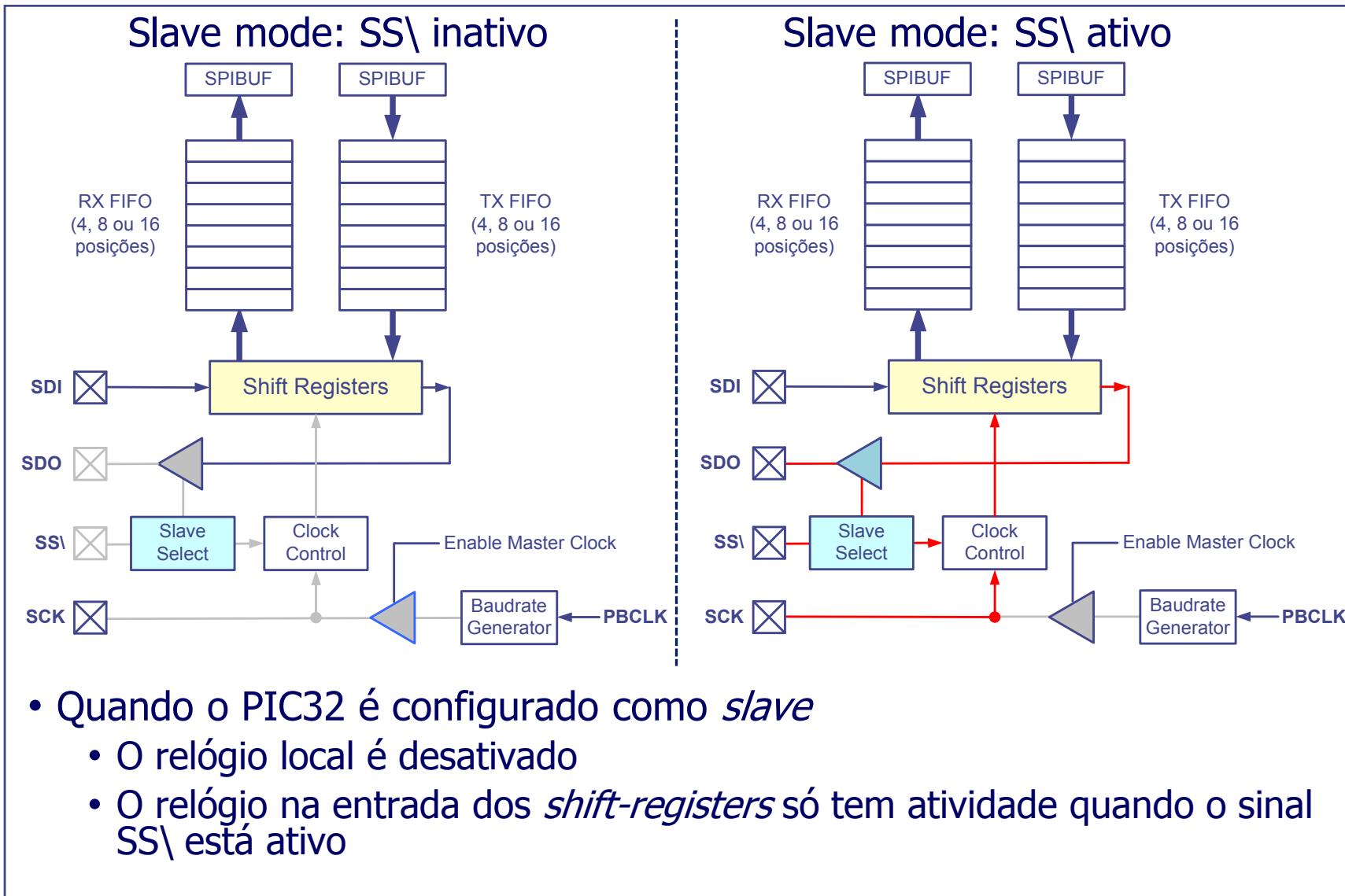
- O PIC32MX795F512H disponibiliza 3 módulos de comunicação SPI
- Cada um dos módulos pode ser configurado para funcionar como *master* ou como *slave*
- Comprimento de palavra configurável: 8, 16 ou 32 bits
- *Shift-registers* separados para receção e transmissão
- Os registos de receção e transmissão são FIFOs:
  - 16 posições se o comprimento de palavra for 8 bits
  - 8 posições se o comprimento de palavra for 16 bits
  - 4 posições se o comprimento de palavra for 32 bits
- Cada uma dos módulos pode ser configurado para gerar interrupções em função da ocupação dos FIFOs (e.g. TX FIFO tem, pelo menos, 1 posição livre; RX FIFO tem, pelo menos, 1 palavra disponível para ser lida)

# Interface SPI no PIC32



- Quando o PIC32 é configurado como *master*
  - O relógio local é ativado durante cada transmissão
  - O buffer 3state do SDO está sempre ativo
  - A entrada SS\ é ignorada

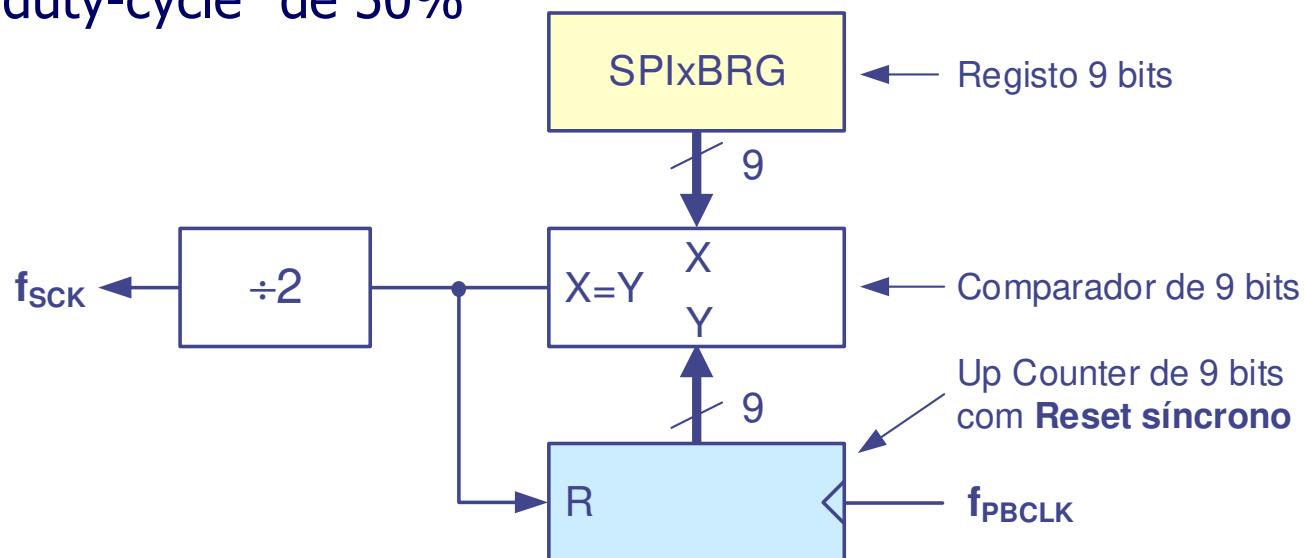
# Interface SPI no PIC32



- Quando o PIC32 é configurado como *slave*
  - O relógio local é desativado
  - O relógio na entrada dos *shift-registers* só tem atividade quando o sinal SS\ está ativo

# Interface SPI no PIC32 – gerador de relógio

- Utiliza uma arquitetura semelhante à de um timer, em que o sinal de relógio de entrada é o Peripheral Bus Clock (20 MHz na placa DETPIC32).
- Com a divisão por 2 à saída do comparador obtém-se um relógio com "duty-cycle" de 50%



- $f_{SCK} = f_{PBCLK} / (2 * (\text{SPIxBRG} + 1))$ , em que SPIxBRG representa a constante armazenada no registro com o mesmo nome

# Aula 15

- O barramento CAN (*Controller Area Network*)
- Características fundamentais
- Aplicações
- Topologia da rede e codificação
- Tipos de tramas
- Deteção de erros
- Filtros de aceitação de mensagens
- Arbitragem

José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

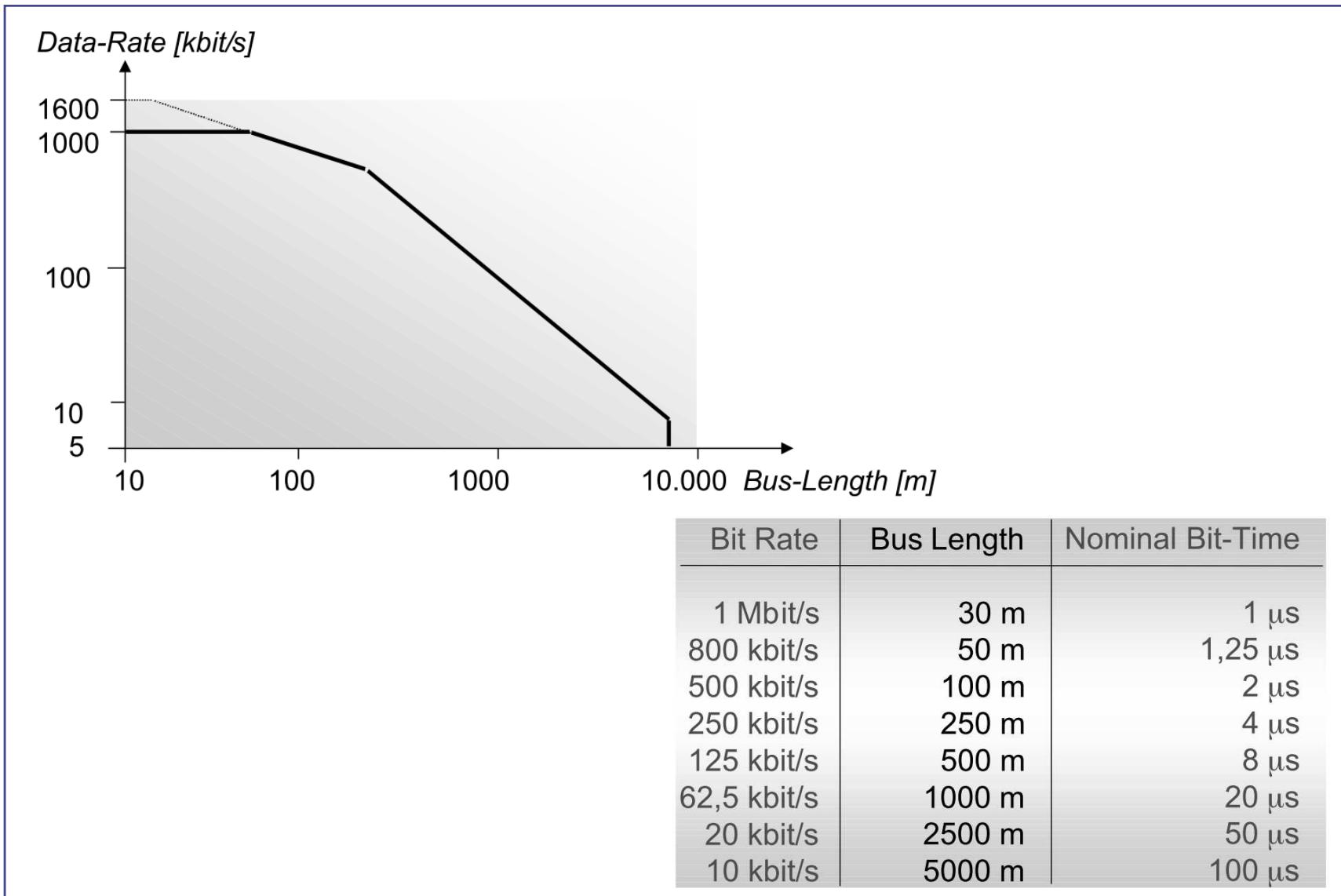
# Introdução

- Desenvolvido em 1991 (versão 2.0) pela Bosch para simplificar as cablagens nos automóveis  
(<http://esd.cs.ucr.edu/webres/can20.pdf>)
- Utiliza comunicação diferencial em par entrançado
- Taxas de transmissão até 1 Mbit/s
- Adequado a aplicações de segurança crítica; elevada robustez
  - Tolerância a interferência eletromagnética
  - Capacidade de detetar diferentes tipos de erros
  - Baixa probabilidade de não deteção de um erro de transmissão ( $4.7 \times 10^{-11}$ )
- Atualmente usado num leque muito variado de aplicações
  - Comunicação entre subsistemas de um automóvel
  - Aviónica, Aplicações industriais, Domótica, Robótica
  - Equipamentos médicos, ...

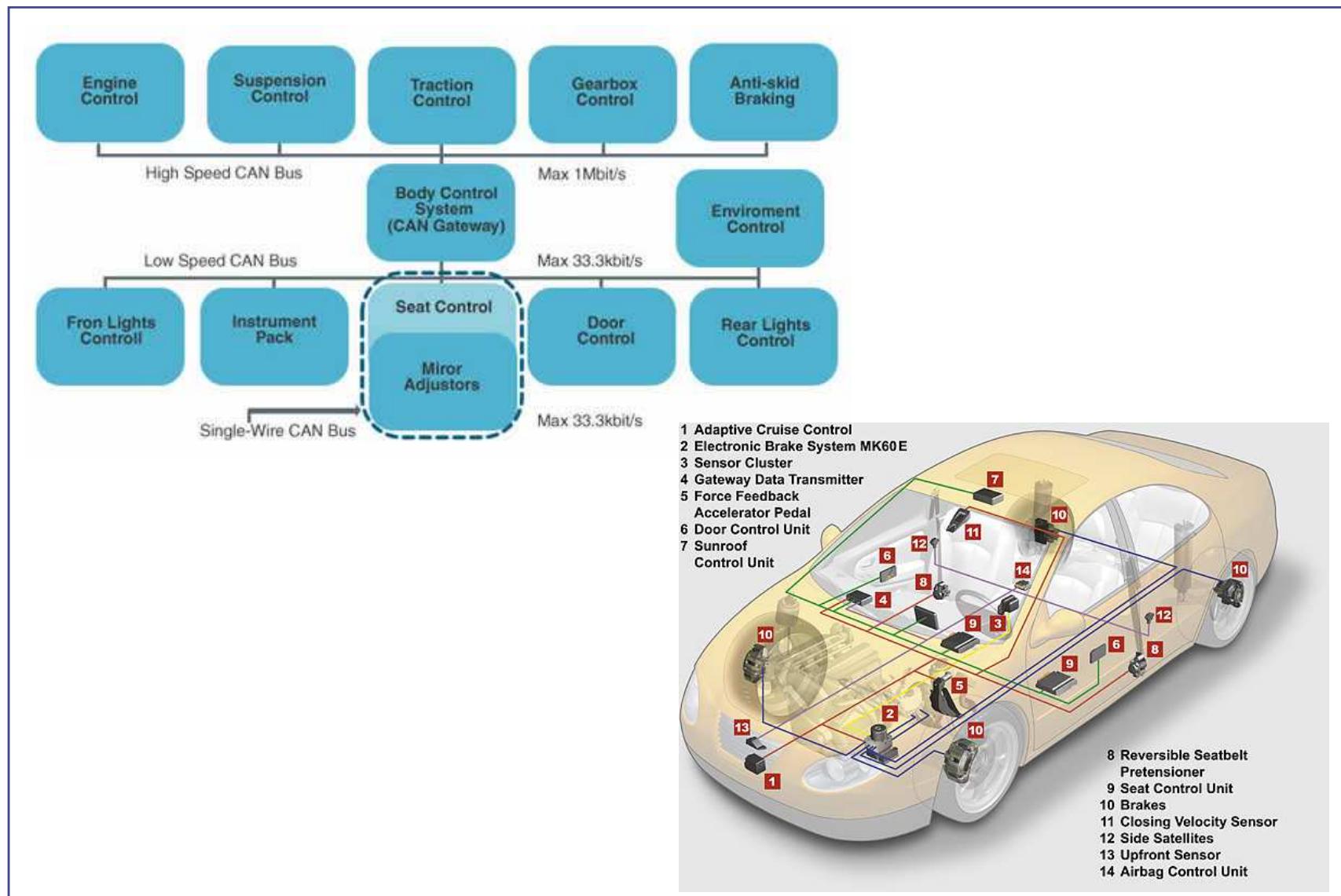
# Introdução

- Transmissão em "broadcast": a informação enviada pelo transmissor pode ser recebida por todos os nós ao mesmo tempo
- Comunicação bidirecional "half-duplex"
- A informação produzida é encapsulada em tramas
- O CAN é um barramento "multi-master": qualquer nó do barramento pode produzir informação e iniciar uma transmissão
- Uma vez que dois ou mais nós podem querer aceder simultaneamente ao barramento para transmitir, tem que haver uma forma de arbitrar o acesso ao meio
- No CAN cada mensagem tem um ID único que identifica a natureza do seu conteúdo; esse ID determina também a prioridade da mensagem e, consequentemente, a prioridade no acesso ao barramento

# Comprimento máximo do barramento

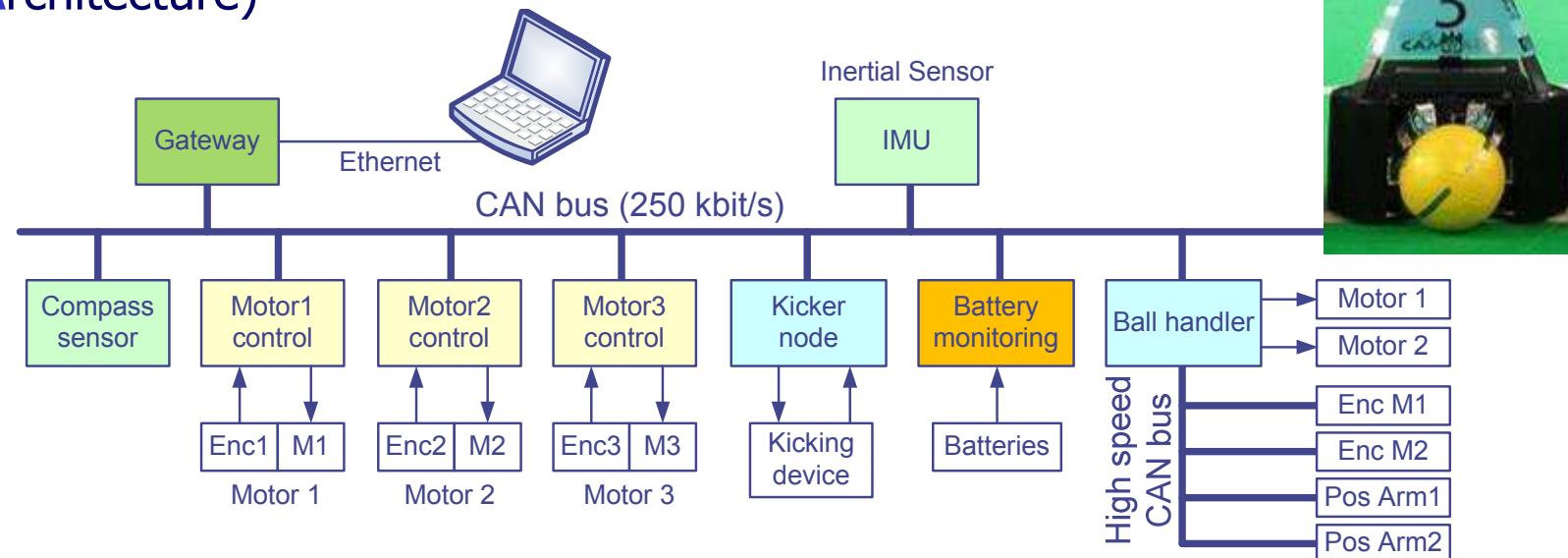


# Exemplos de aplicação



# Exemplos de aplicação

- Infraestrutura sensorial e de atuação dos robots da equipa de futebol robótico do DETI: CAMBADA (Cooperative Autonomous Mobile roBots with Advanced Distributed Architecture)

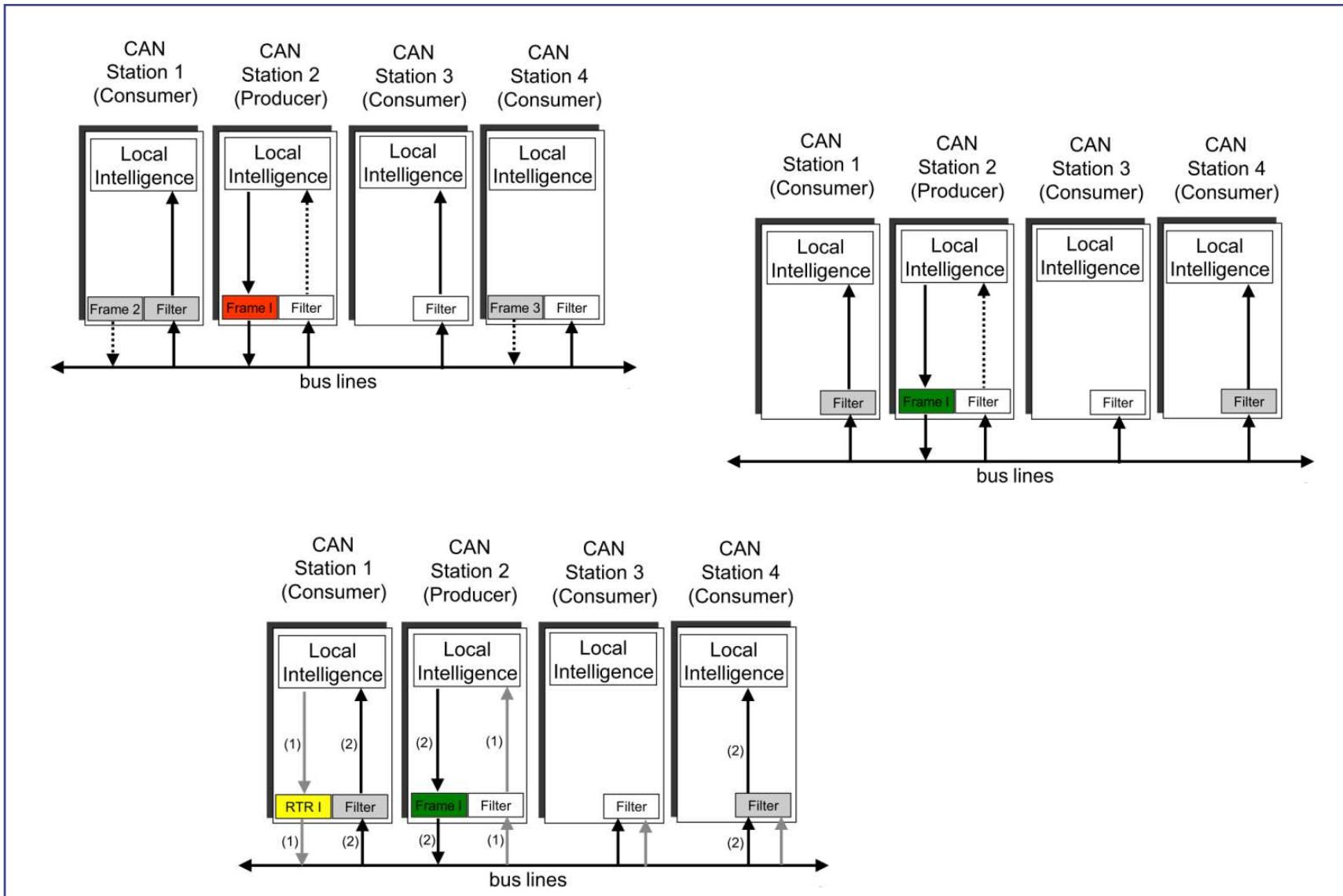


- Arquitetura distribuída em que cada nó desempenha uma tarefa ou conjunto de tarefas relacionadas
- O sistema é facilmente alterável; por exemplo, acrescentar um novo sensor não implica qualquer alteração na estrutura existente (basta ligar o novo nó ao barramento CAN)

# Características fundamentais

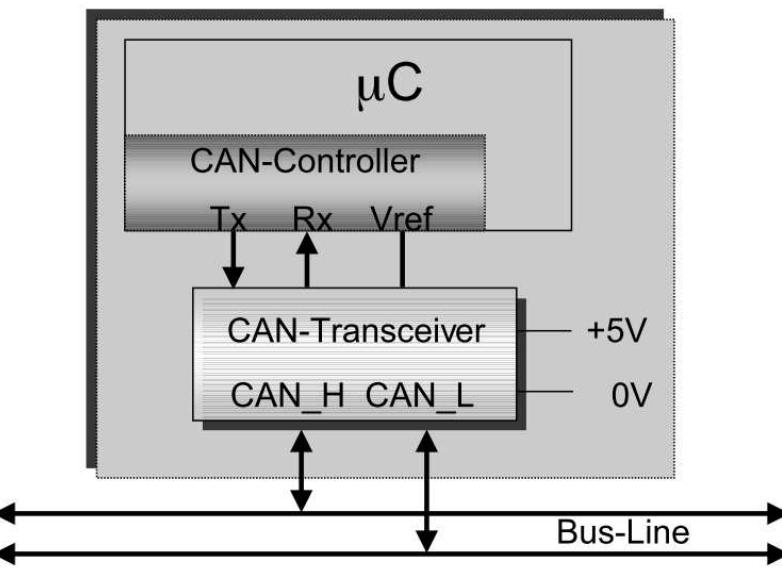
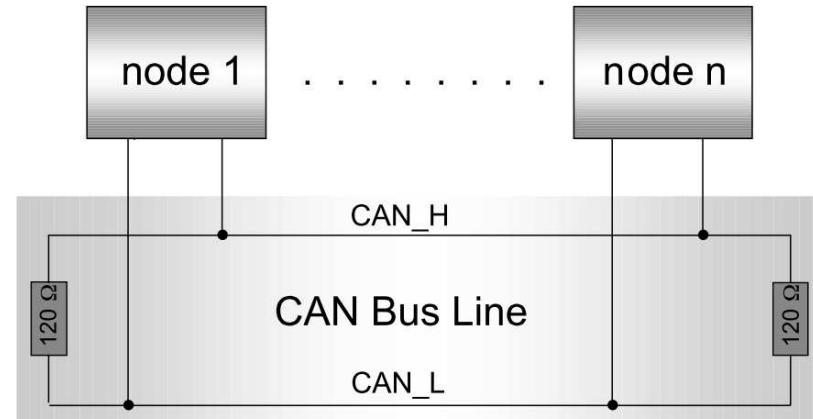
- Sincronização de relógio:
  - **Relógio implícito** (comunicação assíncrona, i.e. não há transmissão do relógio - o transmissor e o receptor têm relógios locais independentes)
- Transmissão orientada ao bit
- Barramento série "multi-master"
  - Diversos nós trocam mensagens encapsuladas em tramas
- Paradigma produtor-consumidor / Transmissão em "broadcast"
  - Identificação do conteúdo da mensagem (não existe identificação do nó de origem ou de destino)
- Capacidade de Remote Transmission Request
- Correção de erros baseada em retransmissão

# Características fundamentais

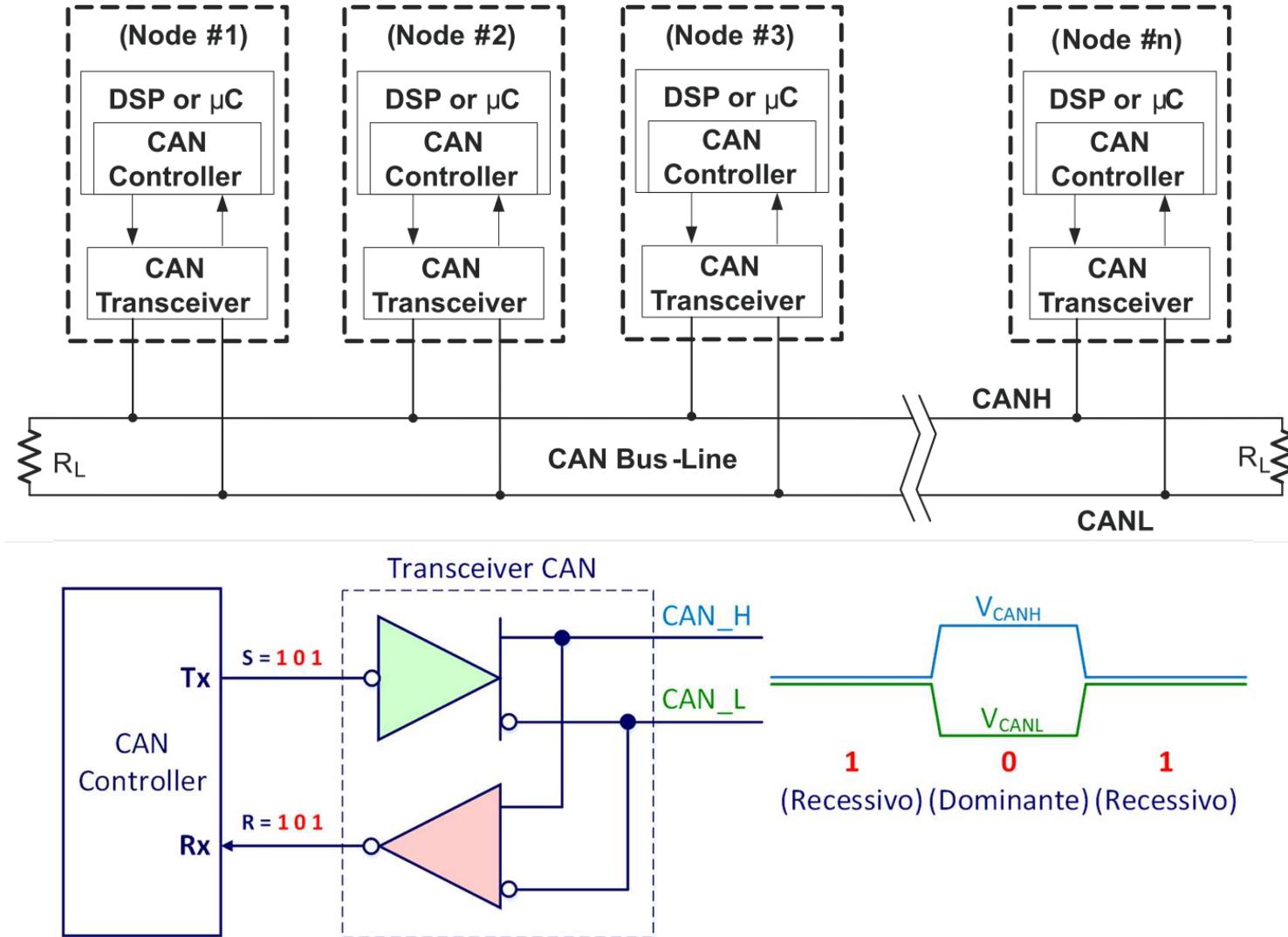


# Topologia da rede e estrutura de um nó

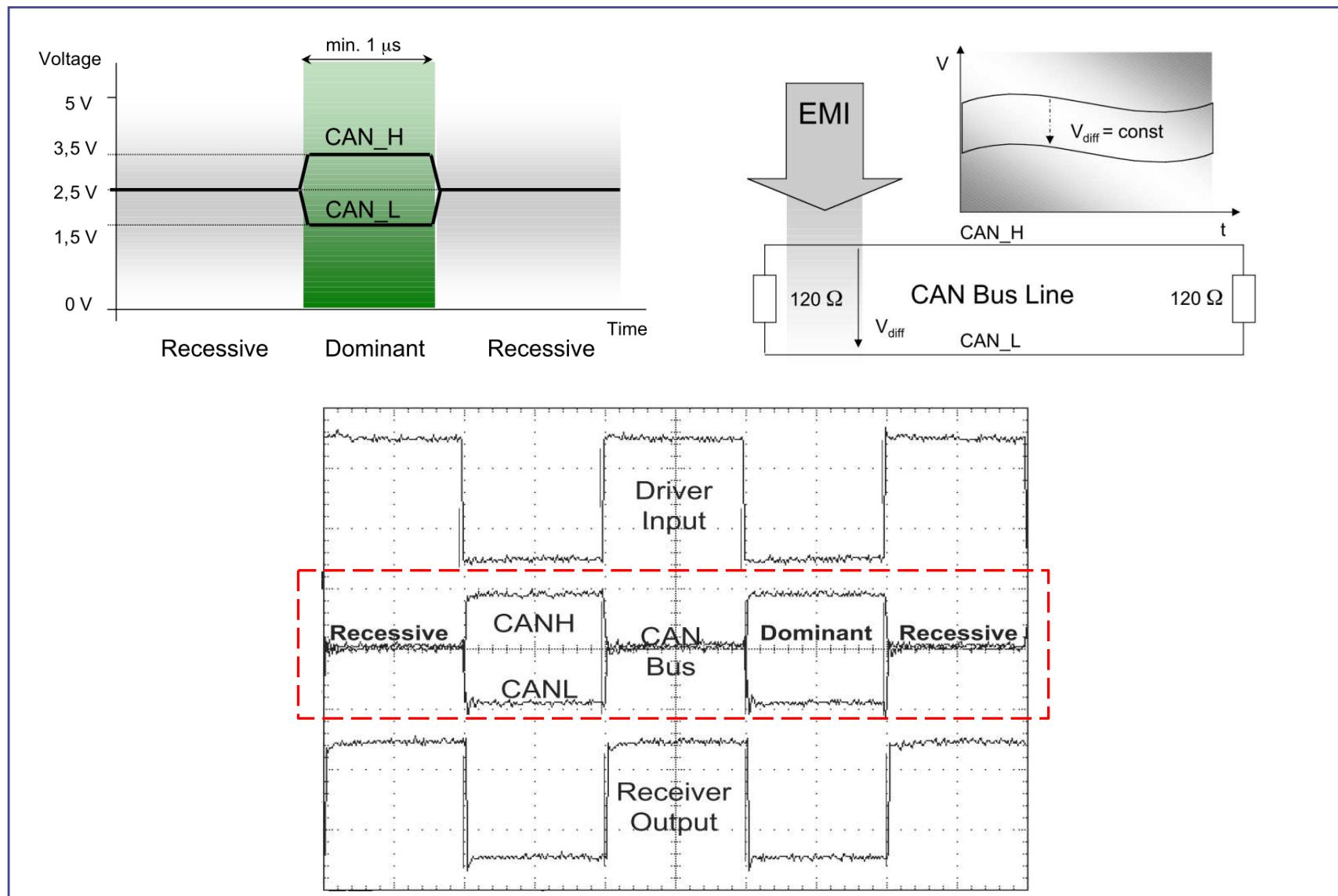
- Comunicação **diferencial, par entrancado**
- Na transmissão, o "transceiver" transforma o nível lógico presente na linha Tx em duas tensões e coloca-as nas linhas **CAN\_H** e **CAN\_L**
- Na receção, o "transceiver" discrimina o nível lógico pela **diferença de tensão entre CAN\_H e CAN\_L** e o resultado é enviado através da linha Rx para o controlador CAN



# Topologia da rede e estrutura de um nó



# Transmissão diferencial



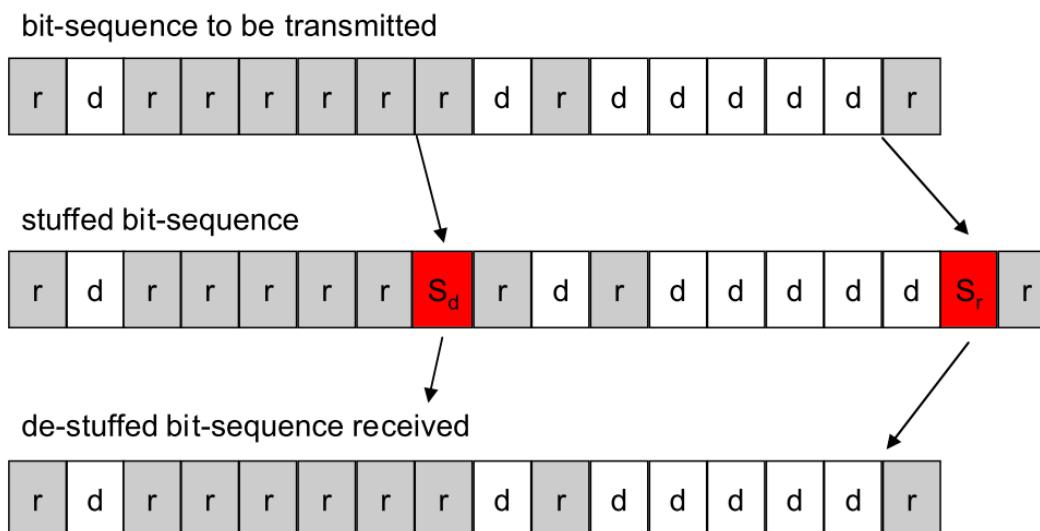
# Codificação

- Codificação Non-Return-to-Zero - bit recessivo ('1')/dominante ('0')



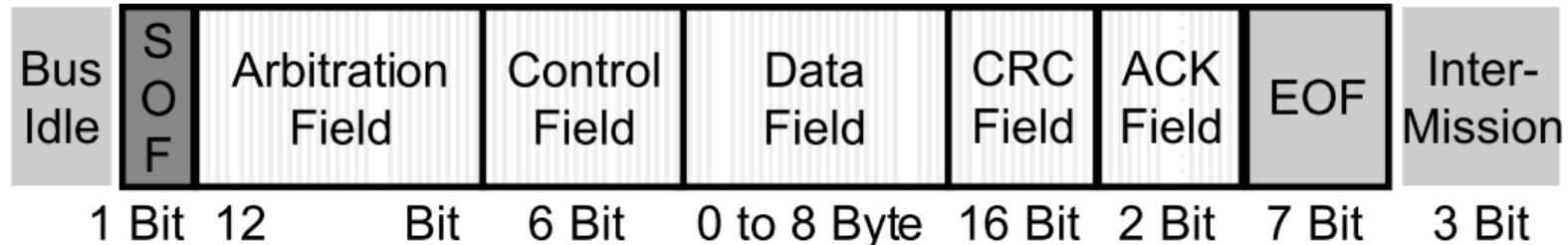
- **"Bit-stuffing"**

- Por cada 5 bits iguais é inserido 1 bit de polaridade oposta

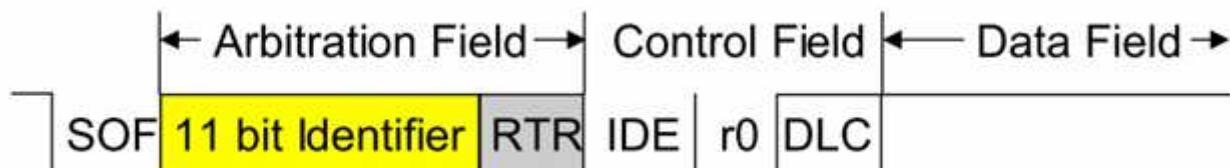


- Garante um tempo máximo entre transições da linha de dados, assegurando que há transições suficientes para manter sincronizados os instantes de amostragem de dados em cada um dos nós

# Formato da trama de dados (CAN 2.0A)

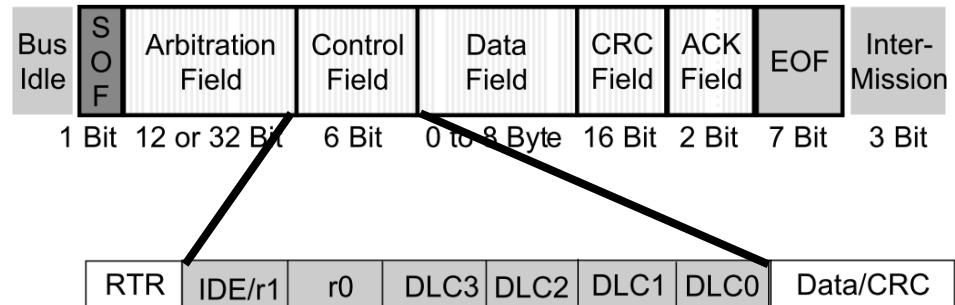


- **SOF** (Start of Frame)
  - Bit dominante ('0') indica o início da trama
  - Usado para sincronização dos instantes de amostragem dos nós receptores
- **Arbitration**
  - **Identifier** (11 bits) – identificador da mensagem que também serve para arbitragem entre diferentes *masters* que podem iniciar a transmissão das suas tramas em simultâneo (id mais baixo, maior prioridade)
  - **RTR** (1 bit) Remote Transmission Request - dominante numa trama de dados
- Standard Frame Format (CAN 2.0A):



# Formato da trama de dados (CAN 2.0A)

- **IDE** (identifier extension)
  - Bit dominante ('0') significa trama standard (CAN 2.0A, 11-bit identifier)
  - Bit recessivo ('1') significa trama CAN 2.0B (com identificador estendido de 29 bits)
- **r0** – reservado
- **DLC3 – DLC0**
  - Número de bytes de dados (0 a 8)
- **Data** (Campo de dados)
  - 0 a 8 bytes (0 a 64 bits)
  - MSBit first (/byte)



No. of Data Bytes	Data Length Code (DLC)			
	DLC3	DLC2	DLC1	DLC0
0	d	d	d	d
1	d	d	d	r
2	d	d	r	d
3	d	d	r	r
4	d	r	d	d
5	d	r	d	r
6	d	r	r	d
7	d	r	r	r
8	r	d/r	d/r	d/r



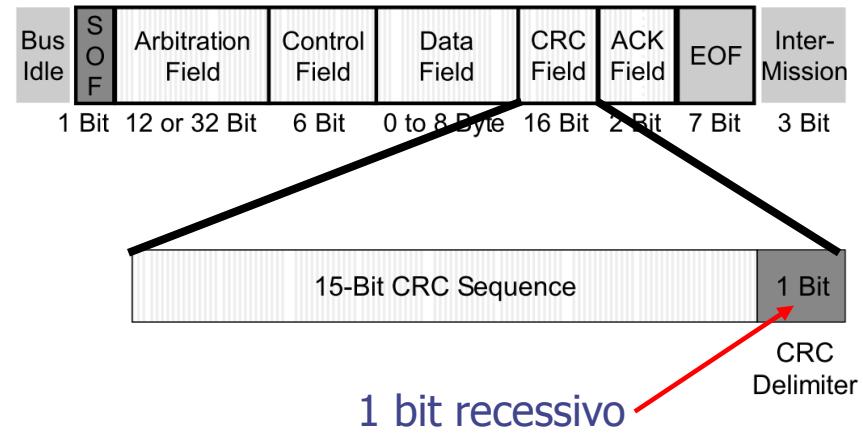
min. length of Data Field = 0 Byte



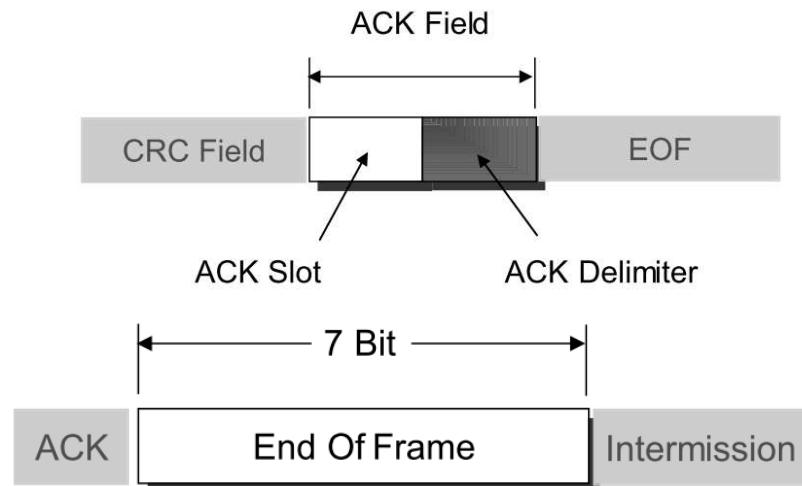
max. length of Data Field = 8 Byte

# Formato da trama de dados (CAN 2.0A)

- **CRC** (Cyclic Redundancy Check)
  - Deteção de erros
  - Produtor e consumidor calculam a sequência de CRC com base nos bits transmitidos/recebidos
  - Produtor transmite a sequência CRC calculada
  - Consumidor compara a sequência CRC calculada localmente com a recebida do produtor
- **ACK** (Acknowledge)
  - Validação da trama (ACK Slot)
  - Recessivo (produtor)
  - Dominante (1+ consumidores)
- **EOF** (End of Frame)
  - Terminação da trama (7 bits recessivos)
- **IFS** (interframe/intermission)
  - Mínimo de 3 bits recessivos



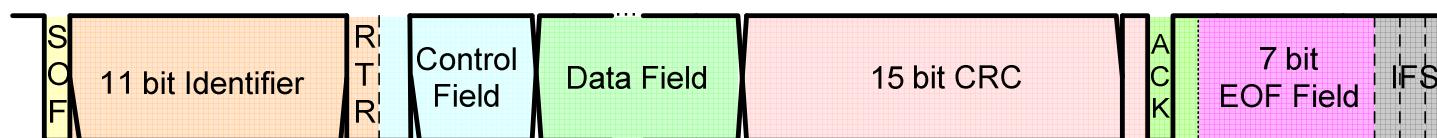
*Remark: The CRC Delimiter is a fixed formatted recessive bit.*



# Tipos de tramas

- **Data Frame**

- Usada no envio de dados de um nó produtor para o(s) consumidor(es); numa trama de dados o bit RTR está a '0' (dominante)



- **Remote Transmission Request Frame**

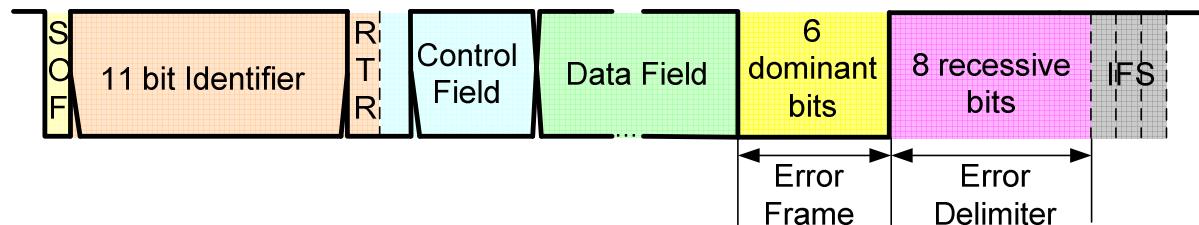
- Enviada por um nó consumidor a solicitar (ao produtor) a transmissão de uma trama de dados específica (trama tem o campo RTR a '1' – recessivo, o que a diferencia de uma trama de dados)



# Tipos de tramas

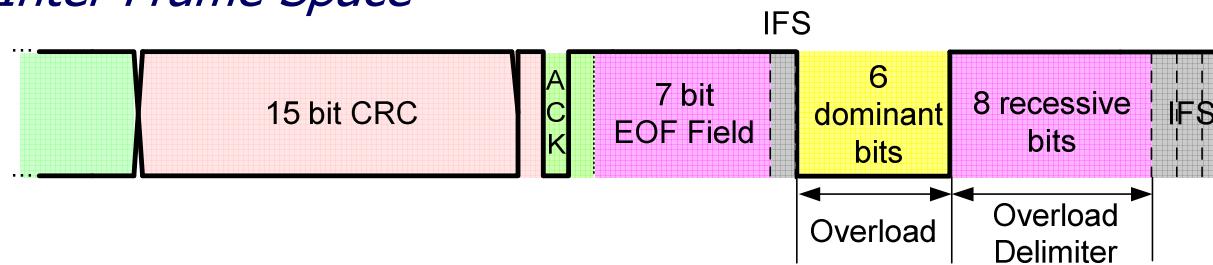
- **Error Frame**

- Usada para reportar um erro detetado (a trama de erro sobrepõe-se a qualquer comunicação invalidando uma transmissão em curso)



- **Overload Frame**

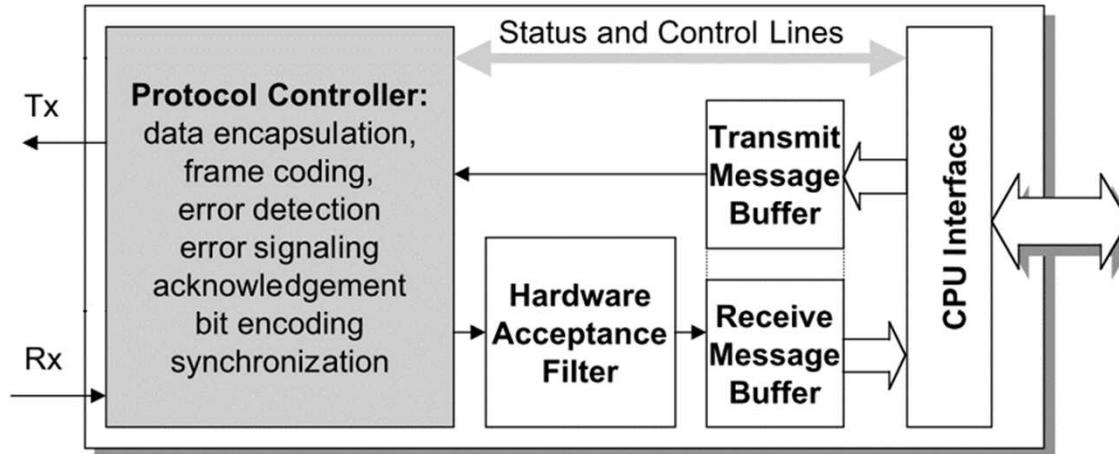
- Usada para atrasar o envio da próxima trama (enviada por um nó em situação de sobrecarga que não teve tempo para processar a última trama enviada). Deve iniciar-se durante os dois primeiros bits do *Inter Frame Space*



# Deteção de erros de comunicação

- São usados vários métodos de deteção de erros. Se a receção de uma trama falha em qualquer um deles essa trama não é aceite e é gerada uma trama de erro que força o produtor a reenviar
- **CRC Error** – o CRC calculado não coincide com o CRC recebido
- **Acknowledge Error** – o produtor não recebe um bit dominante ('0') no campo ACK, o que significa que a mensagem não foi recebida por nenhum nó da rede (todos os nós fazem o "acknowledge" da receção da trama)
- **Form Error** – esta verificação analisa campos da mensagem que devem ter sempre o valor lógico '1' (recessivo): EOF, delimitador do ACK e delimitador do CRC; se for detetado um bit dominante em qualquer destes campos é gerado um erro
- **Bit Error** – cada bit transmitido é analisado pelo produtor da mensagem: se o produtor lê um valor que é o oposto do que escreveu gera um erro (exceções: identificador, ACK)
- **Stuffing Error** – se, após 5 bits consecutivos com o mesmo nível lógico não for recebido um de polaridade oposta, é gerado um erro

# Arquitetura típica de um controlador CAN



- O controlador CAN implementa o protocolo em hardware
- O "CPU interface" assegura, tipicamente, a comunicação com o CPU de um microcontrolador (registos de controlo, estado e dados – buffers)
- O "**hardware acceptance filter**" filtra as mensagens recebidas com base no seu ID. Por programação é possível especificar quais os IDs das mensagens que serão copiadas para o "Receive Message Buffer" (i.e., que serão disponibilizadas ao microcontrolador)
- Este mecanismo de filtragem ao descartar mensagens não desejadas, reduz a carga computacional no microcontrolador

# Filtros de aceitação de mensagens e máscaras

- O CAN é um barramento de tipo "**broadcast**", ou seja, uma mensagem transmitida por um nó é recebida por todos os nós da rede
- O controlador CAN de cada nó lê todas as mensagens que circulam no barramento e coloca-as num registo temporário designado por "Message Assembly Buffer" (MAB)
- Logo que uma mensagem válida é recebida no MAB, é aplicado um **mecanismo de filtragem** que permite que apenas as mensagens de interesse para o nó sejam copiadas para o buffer de receção (as restantes são descartadas)
- A filtragem é feita por verificação dos bits do identificador da mensagem

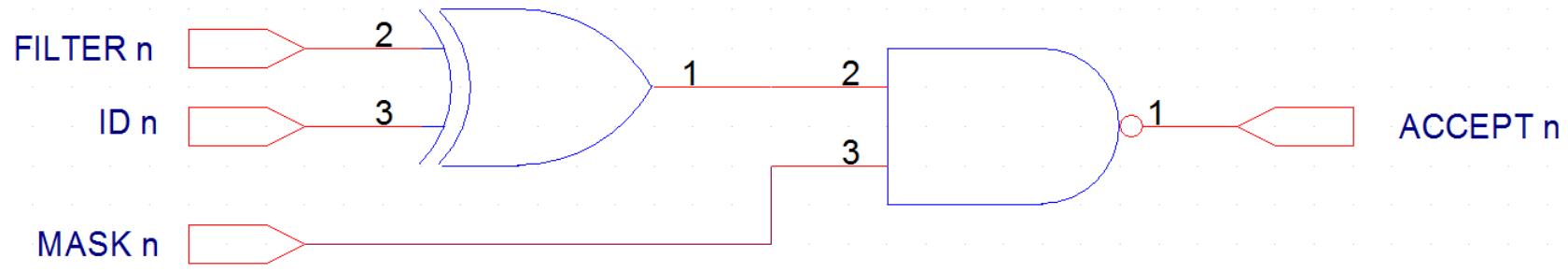
# Filtros de aceitação de mensagens e máscaras

- O mecanismo de filtragem é constituído por um conjunto de **filtros** e **máscaras**: na sua forma mais simples, a mensagem só é copiada para o buffer de receção se o identificador da mensagem igualar um dos filtros de aceitação (previamente configurados por software)
- As máscaras fornecem flexibilidade adicional ao permitir definir quais os bits do identificador que têm que ser iguais aos definidos nos filtros e quais os que são aceites incondicionalmente

Mask bit n	Filter bit n	Message Identifier bit n	Accept/Reject bit n
0	X	X	Accept
1	0	0	Accept
1	0	1	Reject
1	1	0	Reject
1	1	1	Accept

- ACCEPT = ACCEPT<sub>10</sub> . ACCEPT<sub>9</sub> . . . . ACCEPT<sub>0</sub>
- Se ACCEPT=1, a mensagem é copiada para o buffer de receção

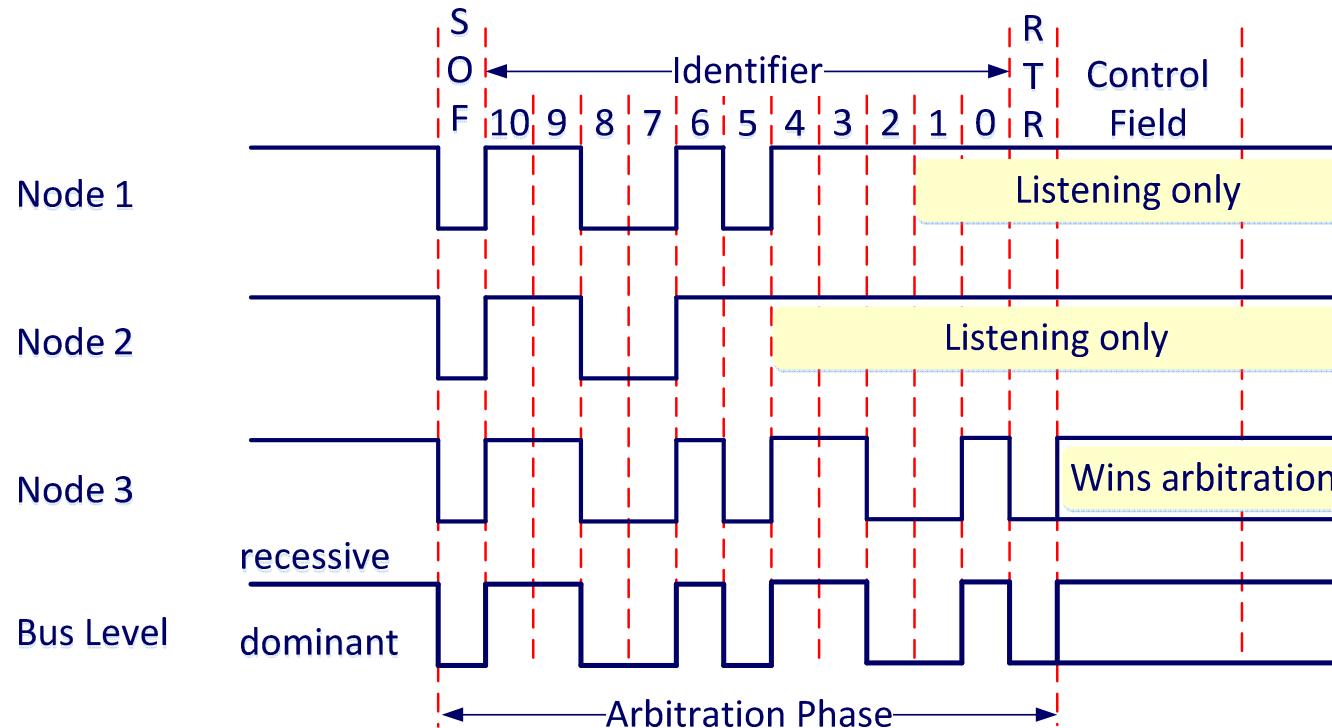
# Filtros de aceitação de mensagens e máscaras



- $\text{ACCEPT} = \text{ACCEPT}_{10} . \text{ACCEPT}_9 . \dots . \text{ACCEPT}_0$
- Se  $\text{ACCEPT}=1$ , a mensagem é copiada para o buffer de receção
- Exemplos (ID de 11 bits):
  - Máscara com o valor 0x000: todas as mensagens são aceites
  - Máscara com o valor 0x7FF, filtro com o valor 0x1F4: apenas a mensagem com o ID 0x1F4 é aceite
  - Máscara com o valor 0x7FC, filtro com o valor 0x230: são aceites as mensagens com os Ids 0x230, 0x231, 0x232 e 0x233

# Controlo de acesso ao meio – Arbitragem

- Realizada durante os campos ID e RTR das tramas (*arbitration field*)
- Baseada em bit recessivo / bit dominante



- O nó produtor da mensagem com o identificador de menor valor binário ganha o processo de arbitragem e transmite os seus dados (um identificador com todos os bits a '0' tem a mais alta prioridade)

# Aula 16

- A interface RS-232C
- Estrutura das tramas
- Codificação dos sinais
- Sincronização de relógio
- Tolerância na frequência dos relógios do emissor e do receptor

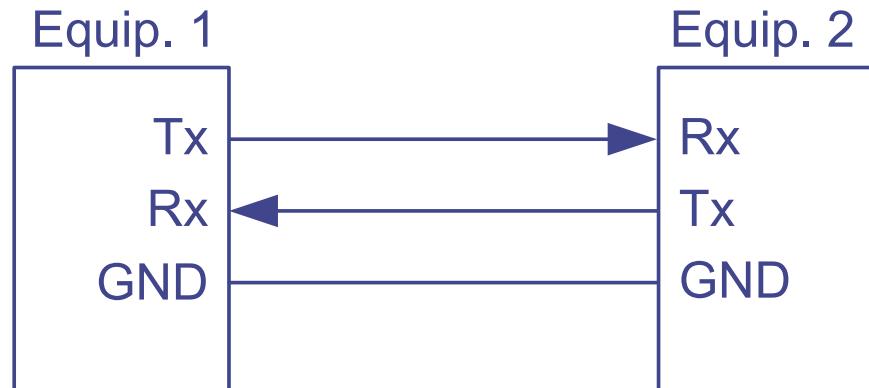
José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

# Introdução

- RS-232C – Standard (1969) para comunicação série assíncrona entre um Equipamento Terminal de Dados (DTE, e.g. computador) e um Equipamento de Comunicação de Dados (DCE, e.g. Modem)
- Permite comunicação bidirecional, *full-duplex*
- Transmissão orientada ao byte
- Conheceu uma grande utilização, que se estendeu muito para além do seu objetivo inicial (ligar DTEs a modems)
- Com o aparecimento do USB os computadores deixaram de disponibilizar comunicação RS-232C
- Por ser um modo de comunicação série muito fácil de implementar e de programar continua a ser muito usado em microcontroladores
- Apareceram no mercado conversores USB/RS-232C que permitem a ligação a PCs de equipamentos que implementam RS-232C

# Sinalização

- Na sua forma mais simples, a implementação da norma RS-232C requer apenas a utilização de 2 linhas de sinalização e uma linha de massa



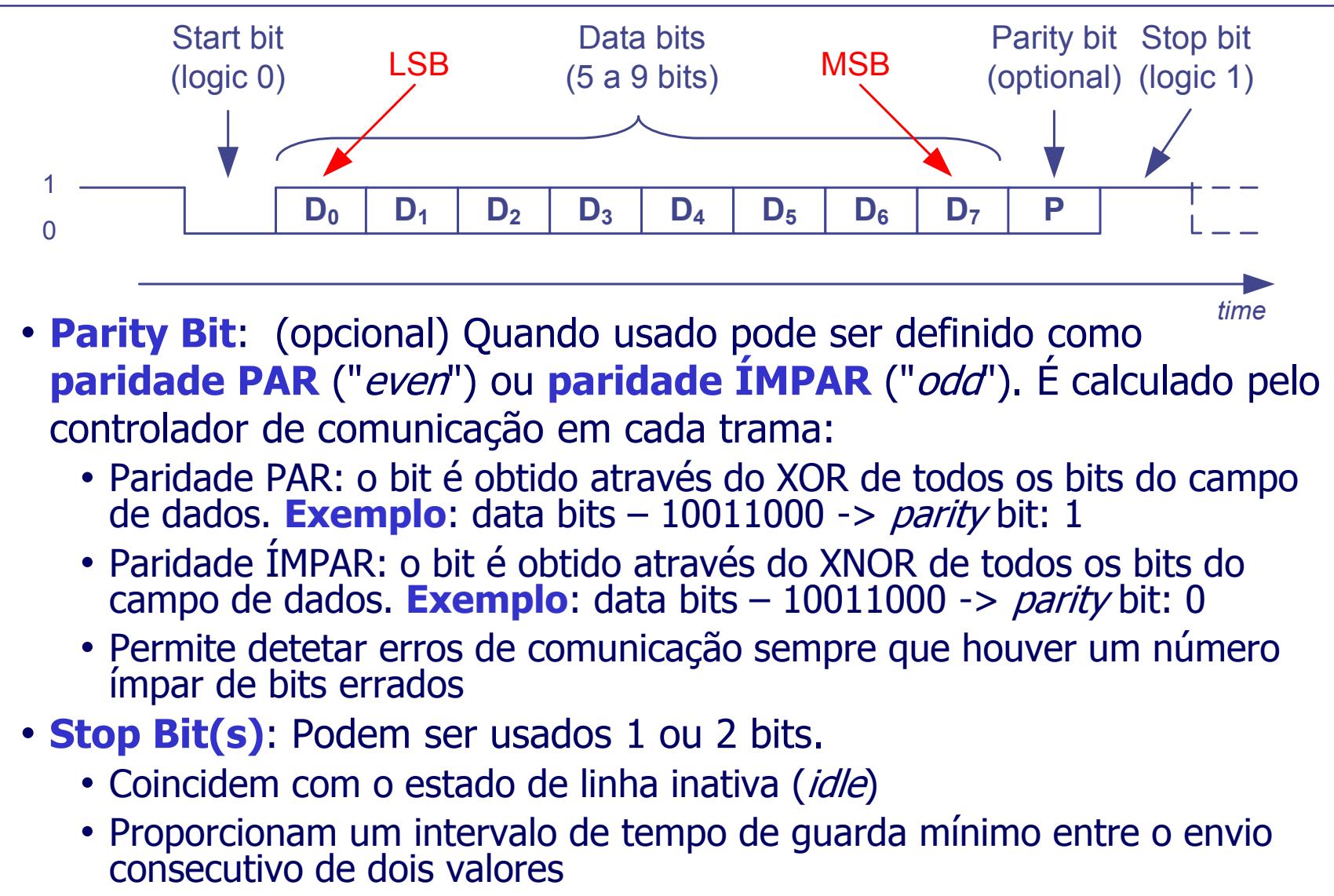
- Podem ser usadas linhas adicionais para protocolar a troca de informação entre os dois equipamentos (*handshake*)
  - RTS (*Request to send*)
  - CTS (*Clear to send*)
  - DTR (*Data terminal ready*)
  - DSR (*Data set ready*)

**A norma original definia um total de 12 sinais (sendo 9 apenas para *handshaking!*)**

# Alguns problemas da norma RS-232C

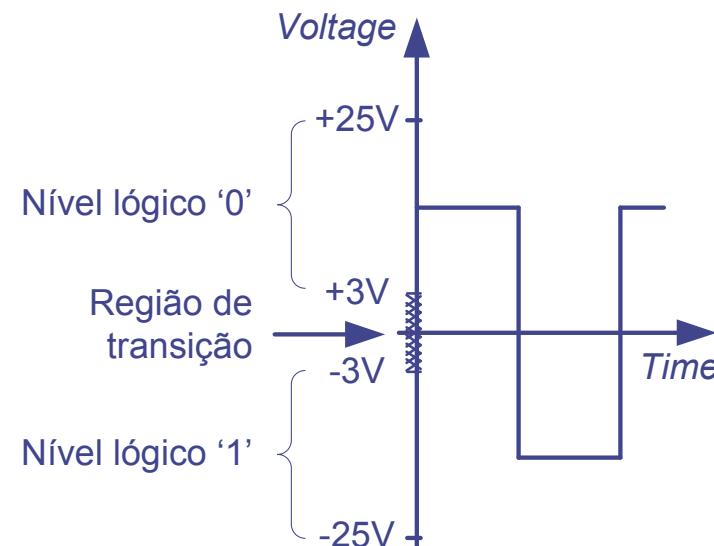
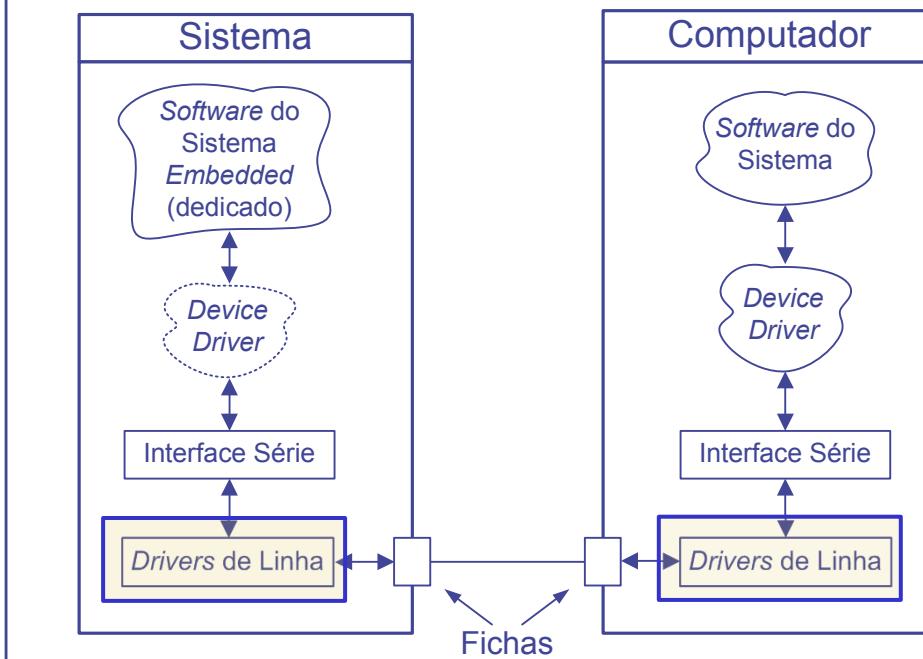
- A nível físico (na linha de comunicação) os níveis lógicos são codificados com tensões simétricas (por exemplo +10V e -10V)
- Consumo de energia elevado
- Sinalização *single-ended*
  - Sinal é diferença entre tensão num fio e *common ground* (0V)
  - Baixa imunidade ao ruído
  - Impõe limitação na velocidade / distância
- Apenas suporta ligações ponto-a-ponto (implementações multi-ponto não standard)
- A norma era suficientemente vaga para permitir implementações proprietárias que, na prática, dificultavam ou mesmo impossibilitavam a interligação entre equipamentos de fabricantes diferentes

# Estrutura de uma trama RS-232C



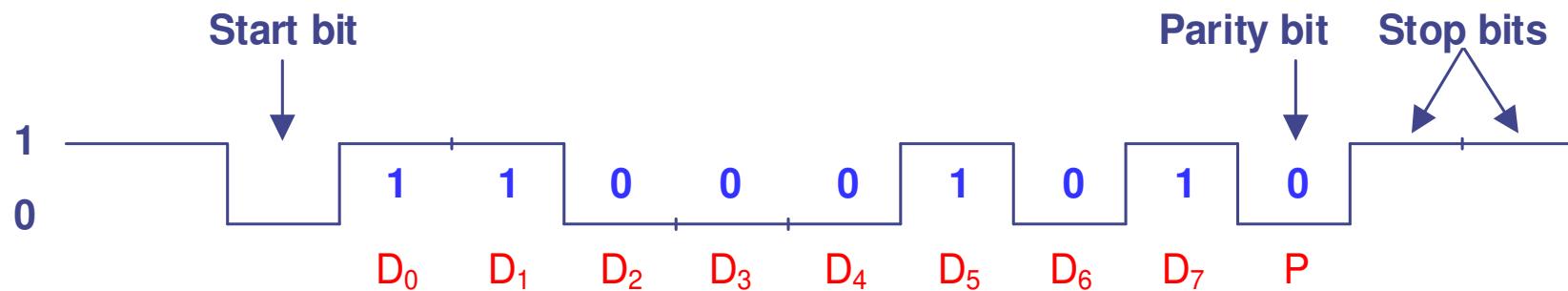
# Camada física – codificação dos níveis lógicos

- Numa ligação física RS-232C os bits da trama são codificados em NRZ-L (*Non Return to Zero - Level*)
  - Nível lógico 1: codificado com uma tensão negativa (na gama -3V a -25V)
  - Nível lógico 0: codificado com uma tensão positiva (na gama +3V a +25V)
- A codificação e descodificação da trama com estes níveis de tensão é assegurada por circuitos eletrónicos designados por **drivers de linha**

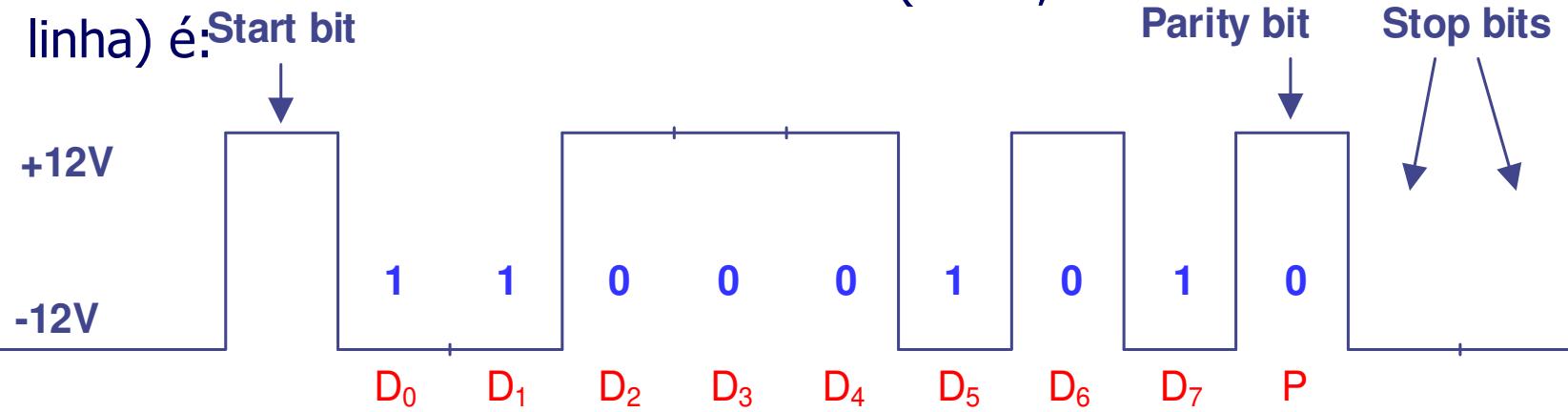


## Exemplo: 8 bits de dados, 2 stop bits, paridade par

- A trama gerada pelo controlador de comunicação série RS-232C (e.g. UART-Universal asynchronous receiver/transmitter) para transmitir o valor 0xA3 é:



- A trama codificada em níveis RS-232C (isto é, à saída do driver de linha) é:

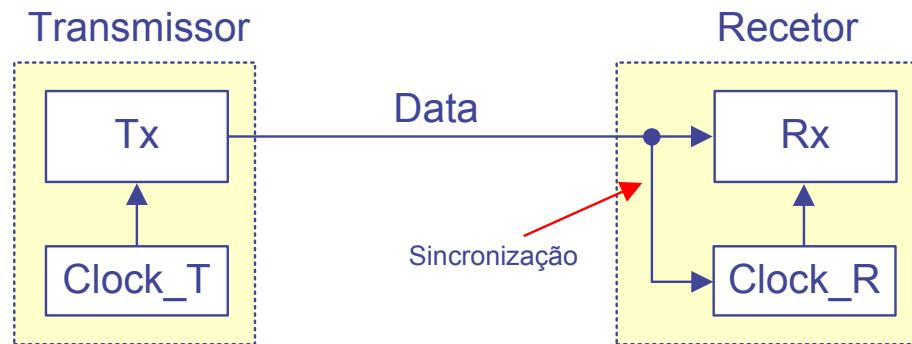


# Baudrate (taxa de transmissão)

- O **baudrate** é, genericamente, o número de símbolos transmitidos por segundo. A cada símbolo pode corresponder um ou mais bits de dados
- A taxa de transmissão de dados bruta (*gross bit rate*) corresponde ao número de bits transmitidos por segundo (bps) (o *baudrate* não deve ser confundido com *gross bit rate*)
- No caso do RS-232C a cada símbolo está associado um único bit, logo o *baudrate* e o *gross bit rate* coincidem
- Exemplos comuns de *baudrates* em RS-232C [bps]: 600, 1200, 4800, 9600, 19200, 38400, 57600, 115200, 230400
- No exemplo anterior o número total de bits a serem transmitidos é 12
  - 1 start bit, 8 bits de dados, 1 bit de paridade, 2 stop bits
  - considerando um *baudrate* de 57600 bps a transmissão completa de uma trama demora  $\sim 208 \mu\text{s}$  ( $12 / 57600$ )
  - o bit rate líquido é  $(8 * 57600) / 12$ , i.e., 38400 bps
  - o byte rate é:  $38400 / 8 = 4800 \text{ bytes/s}$

# Recepção de dados

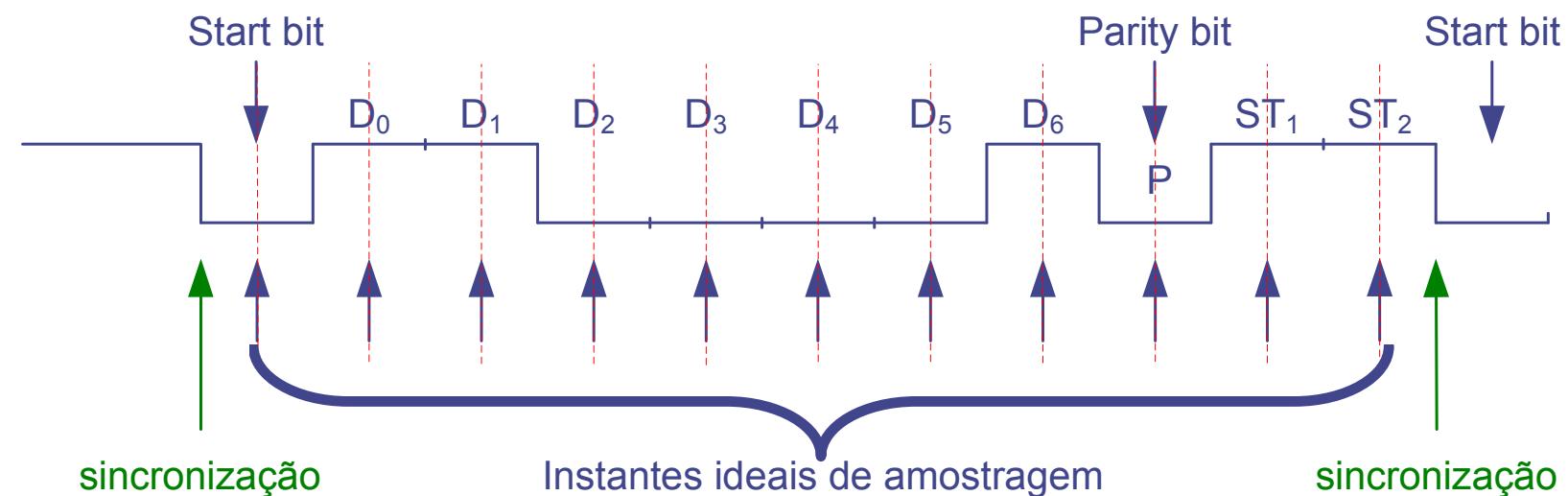
- Sincronização de relógio: **relógio implícito**
  - Comunicação assíncrona (i.e. não há transmissão do relógio)
  - O transmissor e o receptor têm relógios locais (independentes)



- A sincronização do processo de receção de dados é assegurada **no início da receção de cada nova trama** (sinalizado pelo **start bit**: transição de "1" para "0" na linha após um período de inatividade, por exemplo depois da receção completa de uma trama)
- Este método deve ser robusto, dentro de certos limites, a diferenças de frequência entre os relógios do transmissor e do receptor
  - Imprecisão na geração do relógio
  - Constante de divisão dos timers (que geram o relógio) não inteiras

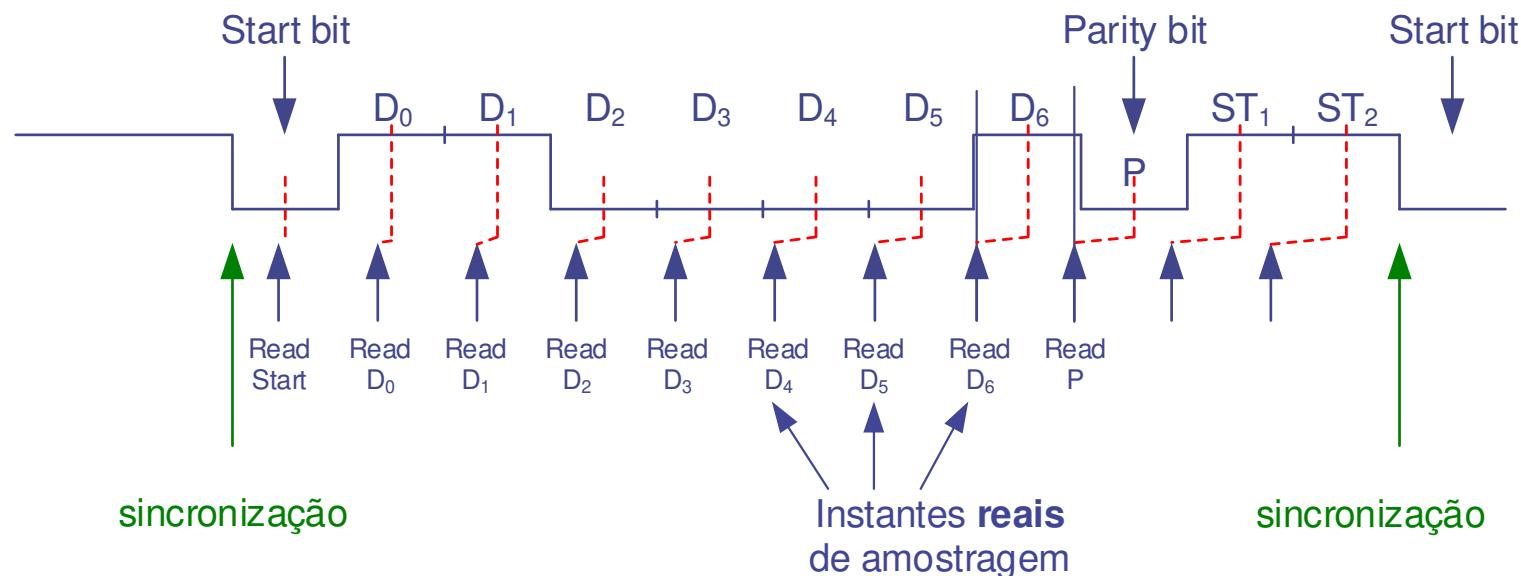
# Recepção de dados

- Para que a comunicação se processe corretamente, o transmissor e o receptor têm que estar **configurados com os mesmos parâmetros**:
  - Estrutura da trama: **nº de bits de dados, tipo de paridade, número de stop bits**
  - **Baudrate** (relógios com a mesma frequência)
- O receptor deve sincronizar-se pelo flanco negativo (transição do nível lógico "1" para o nível lógico "0") da linha (Start bit) e, idealmente, **fazer as leituras a meio do intervalo reservado a cada bit**
- Exemplo da receção do valor 0x43: estrutura da trama 7, 0, 2 (7 data bits, odd parity, 2 stop bits)



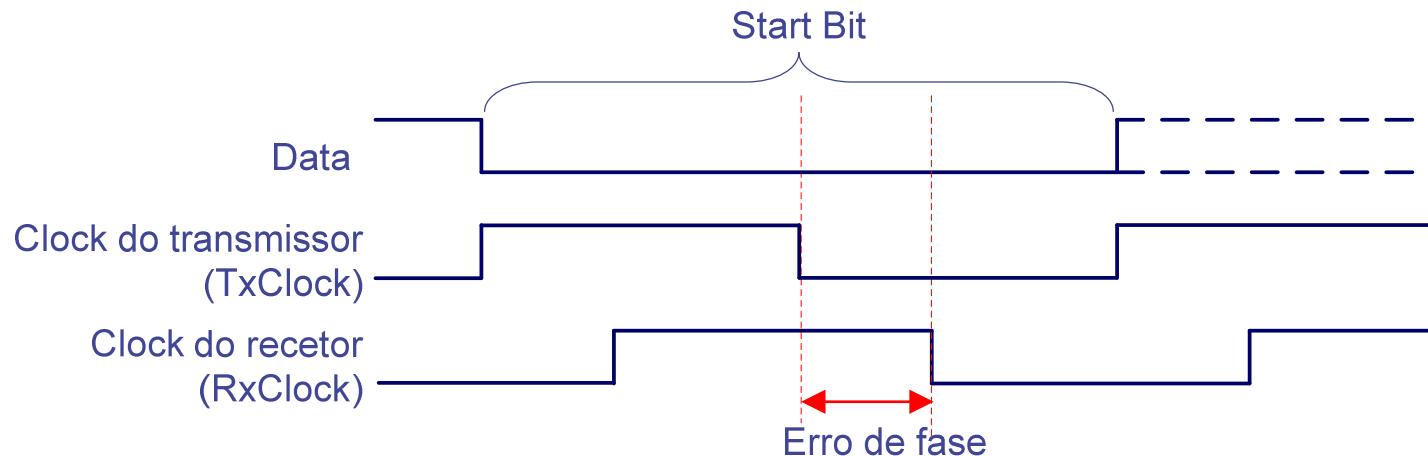
# Sincronização

- Entre instantes de sincronização o desvio dos relógios depende da estabilidade/precisão dos relógios do transmissor e do receptor
- Exemplo em que a receção não é corretamente efetuada devido a um desvio da frequência dos relógios do transmissor e do receptor



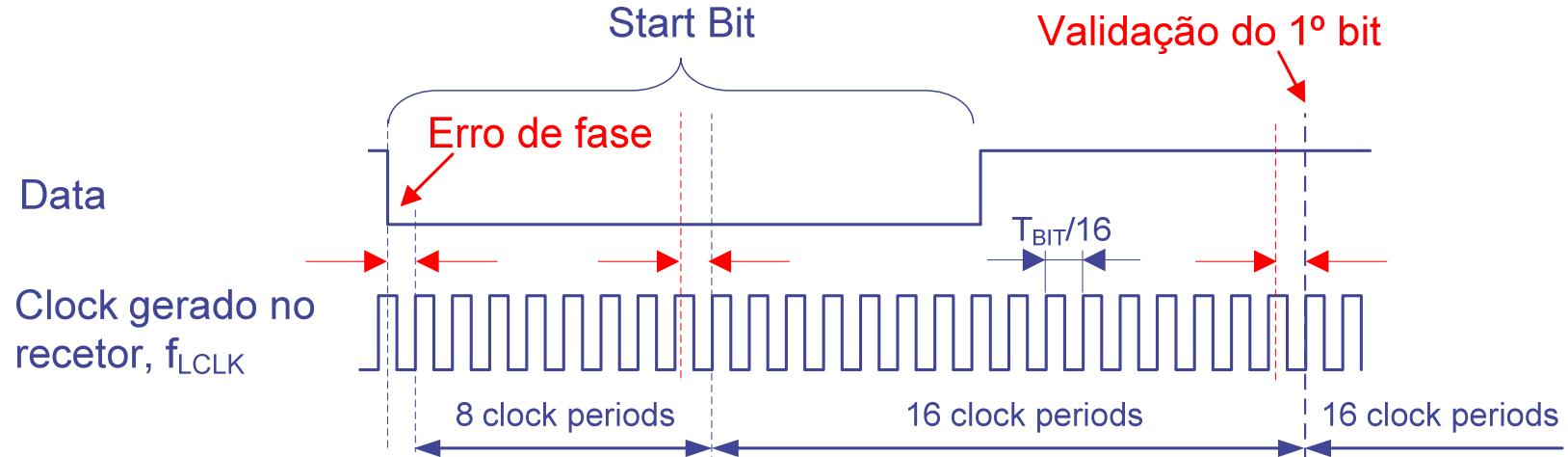
- Neste exemplo o receptor detetaria dois erros:
  - **Erro de paridade** (se o bit  $D_6$  for detetado como 1): o bit de paridade devia ser 0 e é lido como 1
  - **Erro de framing**: é detetado o nível lógico 0 no instante em que era esperado um stop bit (nível lógico 1)

# Sincronização



- Mesmo que a frequência dos relógios do transmissor e do recetor seja a mesma, subsiste o erro de fase que pode impedir a correta validação da informação (idealmente a meio do "tempo de bit")
- Sincronizar a fase do relógio do recetor com a do transmissor é tecnicamente complicado
- Em vez disso, é mais simples gerar no recetor um relógio com uma frequência N vezes superior ao relógio do transmissor e sincronizar a receção a partir desse relógio (designado a seguir por  $f_{LCLK}$ )

# Sincronização



- Por exemplo, se  $N = 16$ , o erro de fase máximo desse relógio, relativamente ao aparecimento do start, é  $T/16$ , em que  $T$  é o período do relógio do transmissor (ou seja, o "tempo de bit",  $T_{BIT}$ )
- O erro de fase mantém-se até ao fim da receção da trama corrente, mas os instantes de validação estão bem definidos:
  - "Start bit", validado ao fim de 8 ciclos de relógio
  - Restantes bits validados a cada 16 ciclos de relógio

# Sincronização

- O relógio local (**LCLK**) deverá então ter, idealmente, uma frequência igual a:

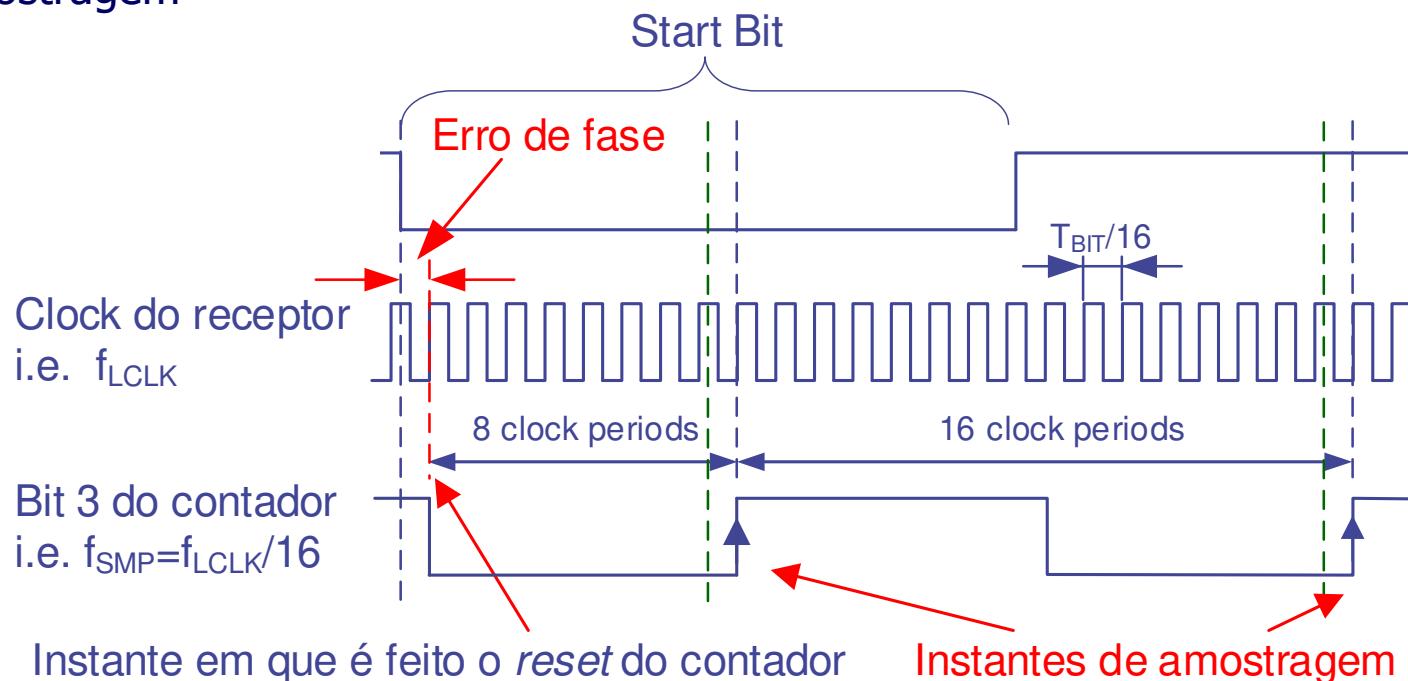
$$f_{LCLK} = N * f_{TCLK}$$

em que  $f_{TCLK} = 1/T_{BIT}$  é a frequência do relógio de transmissão

- N é normalmente designado por **fator de sobreamostragem**
- Valores típicos de N: 4, 16, 64
- Esse relógio não é sincronizado com o sinal da linha, logo impõe um erro de fase (que é sempre inferior a um período,  $\Delta_1 < T_{LCLK}$  )
- Utilizando um relógio com N=16 o erro de fase máximo é  **$T_{BIT}/16$** . Para N=64, o erro de fase máximo será  **$T_{BIT}/64$**
- A utilização de fatores de sobreamostragem elevados nem sempre é possível (frequência da fonte de relógio disponível, constante de divisão do timer)

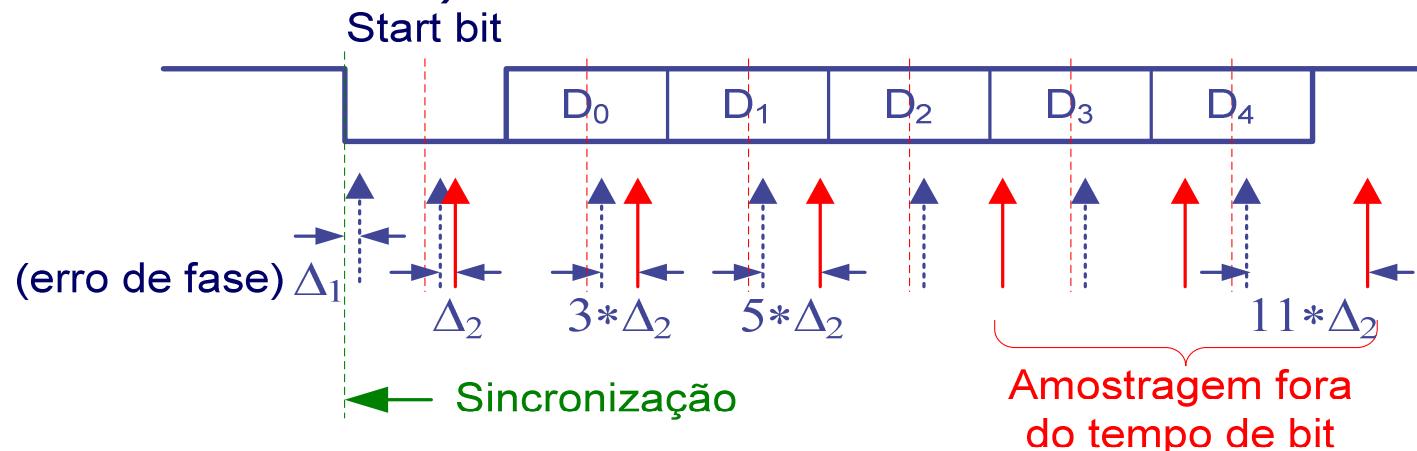
# Sincronização – exemplo de implementação

- Considerando um fator de sobreamostragem ( $N$ ) de 16, o tempo de bit equivale a 16 períodos do relógio local ( $16T_{LCLK}$ ). O erro de fase máximo é  $T_{BIT}/16$
- $f_{SMP}$  é a frequência usada pelo receptor para a amostragem de cada bit da trama
- Usando um contador de 4 bits (com o relógio  $f_{LCLK}$ ) como divisor de frequência por 16 (i.e.  $f_O = f_{SMP} = f_{LCLK}/16$ ) a sincronização é trivial: basta fazer o reset (síncrono) desse contador quando é detetado o start bit
- As transições ascendentes do bit 3 do contador (MSBit) definem os instantes de amostragem



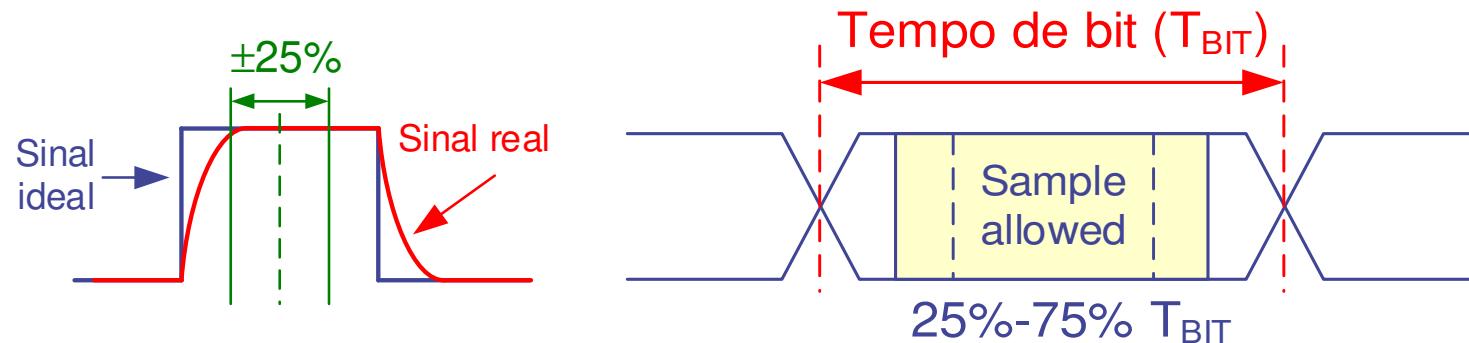
# Sincronização – erro do instante de amostragem

- Os erros nos instantes de amostragem podem ter duas causas distintas:
  - **Erro de fase ( $\Delta_1$ )**: erro cometido ao determinar o instante inicial de sincronização
  - **Erro provocado por desvio de frequência ( $\Delta_2$ )**: a frequência dos relógios do transmissor e do receptor não são exatamente iguais (e.g. tolerância dos cristais de quartzo dos osciladores, constantes de divisão dos timers). Este erro é cumulativo e proporcional ao comprimento da trama (no caso da figura abaixo, no bit  $D_4$  o erro é  $11 * \Delta_2$ )
- O efeito cumulativo destas fontes de erro pode originar erros na receção (como vimos no slide 11)



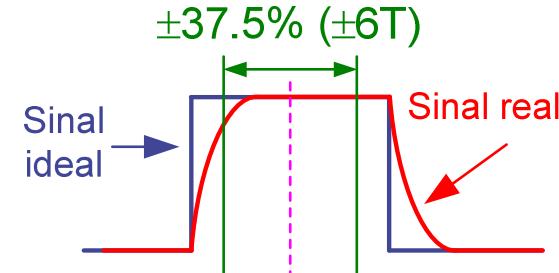
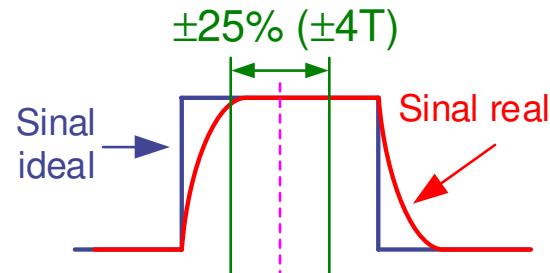
# Máximo desvio de frequência entre emissor e receptor

- É comum considerar-se como zona segura de amostragem do bit:
  - **Pior caso** (cabos longos com efeito capacitivo pronunciado, velocidades de transmissão elevadas, ...):  $\pm 25\%$  do tempo de bit, em torno do instante ideal de amostragem, i.e., a meio
  - **Caso ideal** (cabos curtos e de acordo com as especificações, velocidades moderadas, ...):  $\pm 37,5\%$  do tempo de bit, em torno do instante ideal de amostragem



# Máximo desvio de frequência entre emissor e receptor

- Considerando um fator de sobreamostragem ( $N$ ) de 16, o tempo de bit equivale a 16 períodos do relógio local ( $T_{BIT} = 16T_{LCLK}$ )
- Assim, o desvio máximo admitido no instante de amostragem de um bit é de  $\pm 4T_{LCLK}$  (pior caso,  $\pm 0.25 \times 16$  ciclos) a  $\pm 6T_{LCLK}$  (caso ideal,  $\pm 0.375 \times 16$  ciclos)

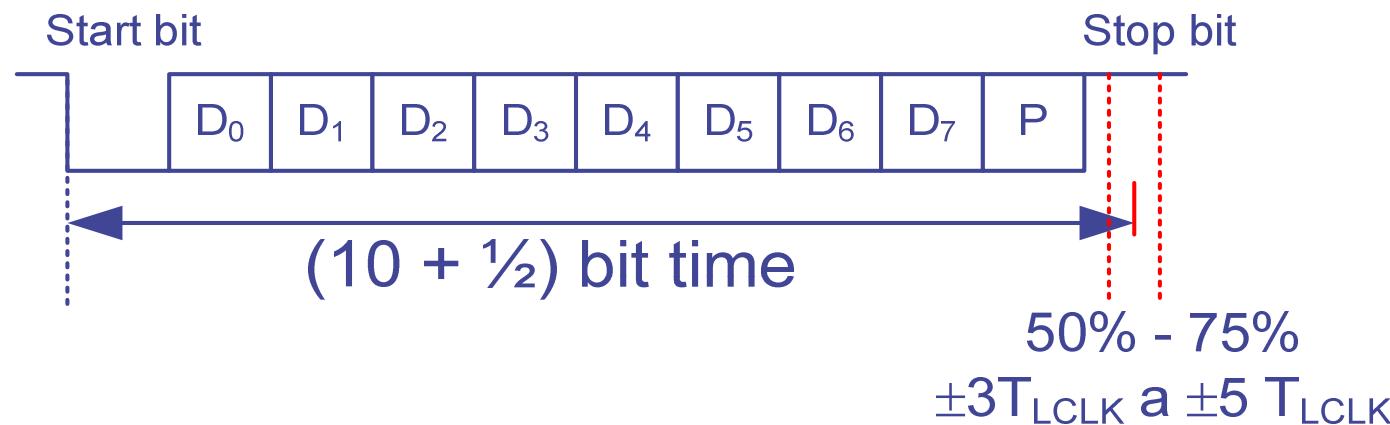


- Como há um erro intrínseco máximo de  $1T_{LCLK}$  devido ao erro de fase, então o desvio máximo aceitável, resultante da diferença de frequência dos relógios, é de  $\pm 3T_{LCLK}$  a  $\pm 5T_{LCLK}$  (em cada instante de amostragem)

**Esta correção do erro de fase para a obtenção do desvio máximo aceitável é conservativa. Porquê?**

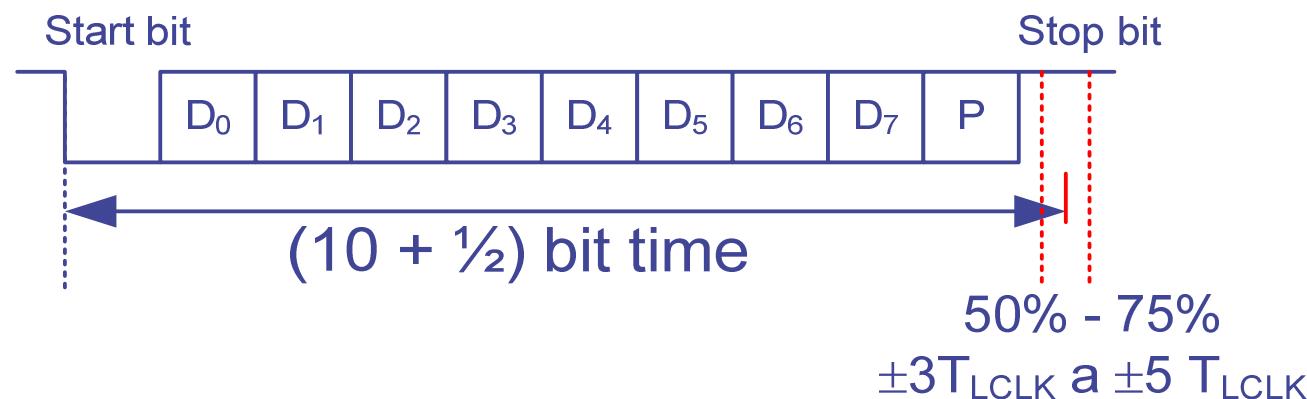
# Máximo desvio de frequência entre emissor e recetor

- Ao contrário do erro de fase, o erro provocado por desvio de frequência é cumulativo e diretamente proporcional ao comprimento da trama
- Por outro lado é necessário garantir que o último bit da trama (independentemente do comprimento da trama) é sempre amostrado dentro da zona segura ( $\pm 3T_{LCLK}$  a  $\pm 5T_{LCLK}$ , relativamente ao instante ideal de amostragem)



## Máximo desvio de frequência entre emissor e recetor

- Consideremos então o caso em que pretendemos amostrar uma das tramas mais longas (start bit, 8 data bits, bit de paridade e 1 stop bit).

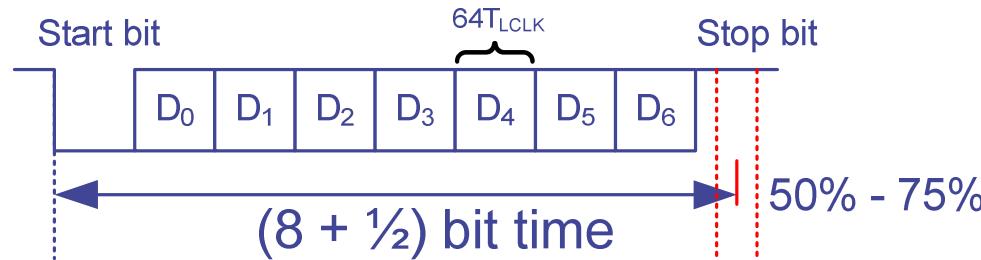


- Continuando a admitir que o relógio local tem um fator de sobreamostragem de N=16, para amostrar esta trama serão então necessários  $10.5 \times 16 = 168$  períodos do relógio local
- Assim, a máxima discrepância que poderá ser tolerada entre os relógios do transmissor e do recetor é  $\Delta T = \pm 3/168 \approx \pm 1.8\%$  (pior caso) a  $\Delta T = \pm 5/168 \approx \pm 3.0\%$  (caso "ideal")

# Máximo desvio de frequência entre emissor e recetor

- Como vimos, a máxima discrepância que poderá ser tolerada entre os relógios do transmissor e do recetor é  $\Delta T = \pm 3/168 \approx \pm 1.8\%$  (pior caso) a  $\Delta T = \pm 5/168 \approx \pm 3.0\%$  (caso "ideal") para as condições indicadas.
- **Exemplo:** vamos assumir que a taxa de transmissão é de 115200 bps, 8 data bits, parity bit, 1 stop bit e N=16. Para os dois casos-limite, para que a comunicação se processe sem erros, qual deve ser a gama de frequência do relógio do recetor?
- $T_{LCLK} = \frac{1}{(16*115200)} \gg f_{LCLK} = 1.843.200 \text{ Hz (valor ideal)}$
- $T_{LCLK} = \frac{(1 \pm 0.018)}{(16*115200)} \gg f_{LCLK} \in [1.810.609, 1.876.986] \text{ Hz (pior caso)}$
- $T_{LCLK} = \frac{(1 \pm 0.03)}{(16*115200)} \gg f_{LCLK} \in [1.798.515, 1.900.206] \text{ Hz (melhor caso)}$
- **Exercício:** Admita que a configuração da comunicação é 38400 bps, 7 bits sem paridade, 1 stop bit e N = 64. Calcule o valor de frequência ideal no recetor e os intervalos admissíveis dessa frequência para os casos limite (tente resolver sozinho. A solução está na versão em PDF)

# Exercício: resolução



- Intervalo de validação (melhor caso):  $\pm \left( \left( \frac{75\%}{2} * 64 \right) - 1 \right) = \pm 23T_{LCLK}$
- Intervalo de validação (pior caso):  $\pm \left( \left( \frac{50\%}{2} * 64 \right) - 1 \right) = \pm 15T_{LCLK}$
- Nº de períodos de relógio para amostrar a trama  $= 8.5 * 64 = 544$
- Máxima discrepância temporal (melhor caso):  $\frac{\pm 23}{544} \approx \pm 4.3\%$
- Máxima discrepância temporal (pior caso):  $\frac{\pm 15}{544} \approx \pm 2.76\%$
- $T_{LCLK} = \frac{1}{(64*38400)} \gg f_{LCLK} = 2.457.600 \text{ Hz (valor ideal)}$
- $T_{LCLK} = \frac{(1\pm 0.0276)}{(64*38400)} \gg f_{LCLK} \in [2.391.592, 2.527.355] \text{ Hz (pior caso)}$
- $T_{LCLK} = \frac{(1\pm 0.043)}{(64*38400)} \gg f_{LCLK} \in [2.356.280, 2.568.025] \text{ Hz (melhor caso)}$

# Aula 17

- *Device drivers*
- Princípios gerais
- Caso de estudo: *device driver* para uma UART
- Princípio de operação e estruturas de dados
- Funções de interface com a aplicação
- Funções de serviço de interrupções e interface com o hardware

José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

# Introdução

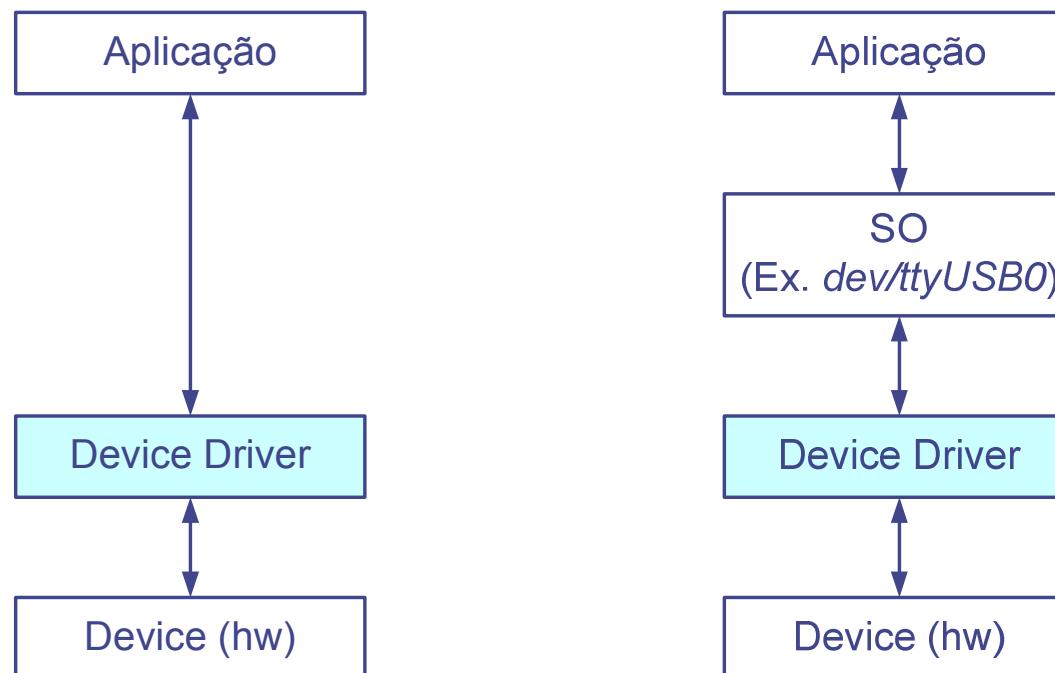
- O **número** de **periféricos** existentes é muito **vasto**:
  - Teclado, rato, placas (gráfica, rede, som, etc.), disco duro, *pen drive*, scanner, câmara de vídeo, etc.
- Estes periféricos apresentam características distintas:
  - **Operações suportadas**: leitura, escrita, leitura e escrita
  - **Modo de acesso**: carater, bloco, etc.
  - **Representação da informação**: ASCII, UNICODE, *Little/Big Endian*, etc.
  - **Largura de banda**: alguns bytes/s a MB/s
  - **Recursos utilizados**: portos (I/O, *memory mapped*), interrupções, DMA, etc.
  - **Implementação**: diferentes dispositivos de uma dada classe podem ser baseados em implementações distintas (e.g. diferentes fabricantes, diferentes modelos) com reflexos profundos na sua operação interna

# Introdução

- As aplicações/Sistemas Operativos (SO) **não podem conhecer todos os tipos de dispositivos** passados, atuais e futuros com um nível de detalhe suficiente para realizar o seu controlo a baixo nível!
- **Solução:** Criar uma camada de abstração que permita o acesso ao dispositivo de forma independente da sua implementação
- ***Device driver***
  - Um programa que permite a outro programa (aplicação, SO) interagir com um dado dispositivo de hardware
  - Implementa a camada de abstração e lida com as particularidades do dispositivo controlado
  - Como o *Device Driver* tem de lidar com os aspetos específicos da implementação física, o seu fornecimento é assegurado pelo fabricante
- **Aspetos-chave:**
  - Abstração, uniformização de acesso, independência entre aplicações/SO e o hardware

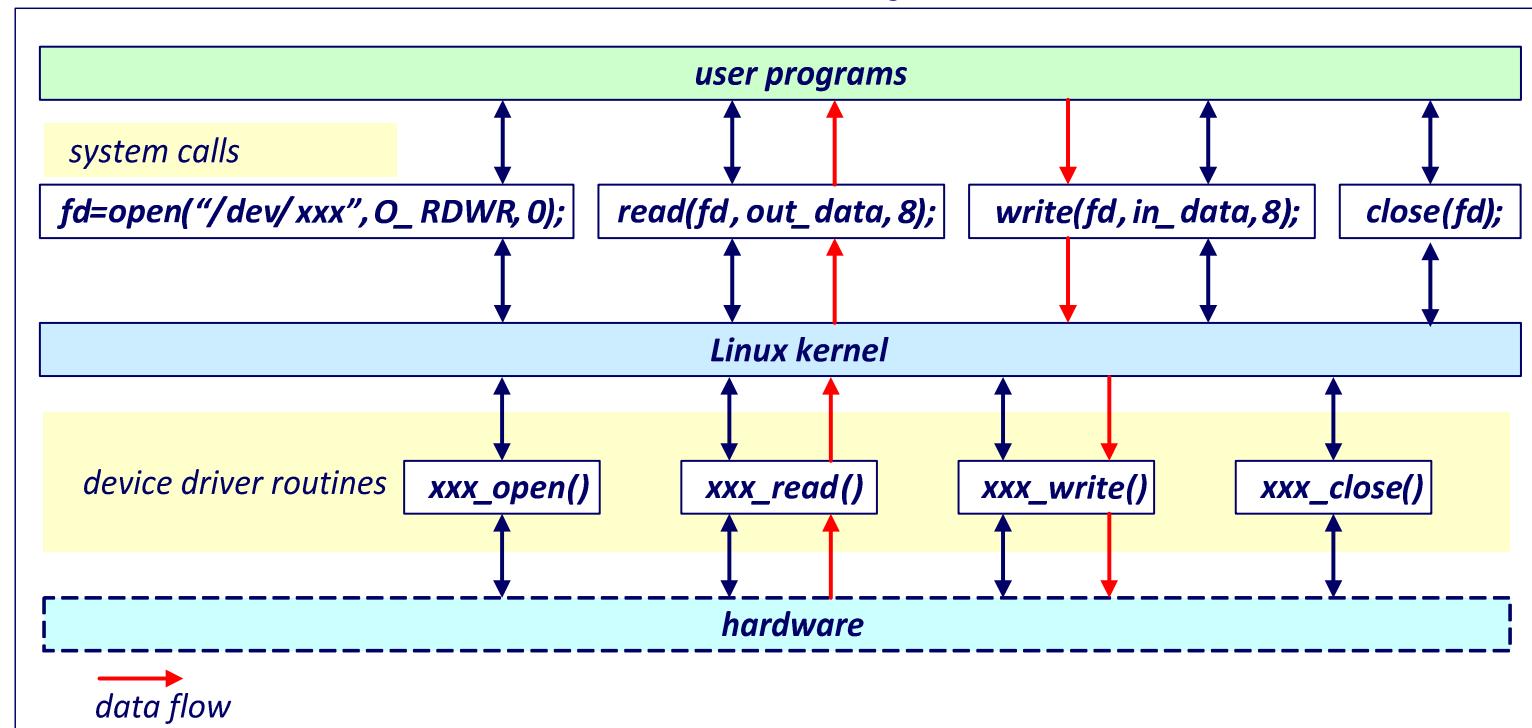
# Princípios gerais

- O acesso, por parte das aplicações, a um *device driver* é diferente num sistema embutido e num sistema computacional de uso geral (com um Sistema Operativo típico, e.g. Linux, Windows, Mac OS):
  - Aplicações em sistemas embutidos acedem, tipicamente, de forma direta aos *device drivers*
  - Aplicações que correm sobre SO acedem a funções do SO (*system calls*); o kernel do SO, por sua vez, acede aos *device drivers*



# Princípios gerais

- O Sistema Operativo especifica classes de dispositivos e, para cada classe, uma interface que estabelece como é realizado o acesso a esses dispositivos
  - A função do *device driver* é traduzir as chamadas realizadas pela aplicação/SO em ações específicas do dispositivo
  - Exemplos de classes de dispositivos: interface com o utilizador, armazenamento de massa, comunicação, ...



# Exemplo de um *device driver*: comunicação série

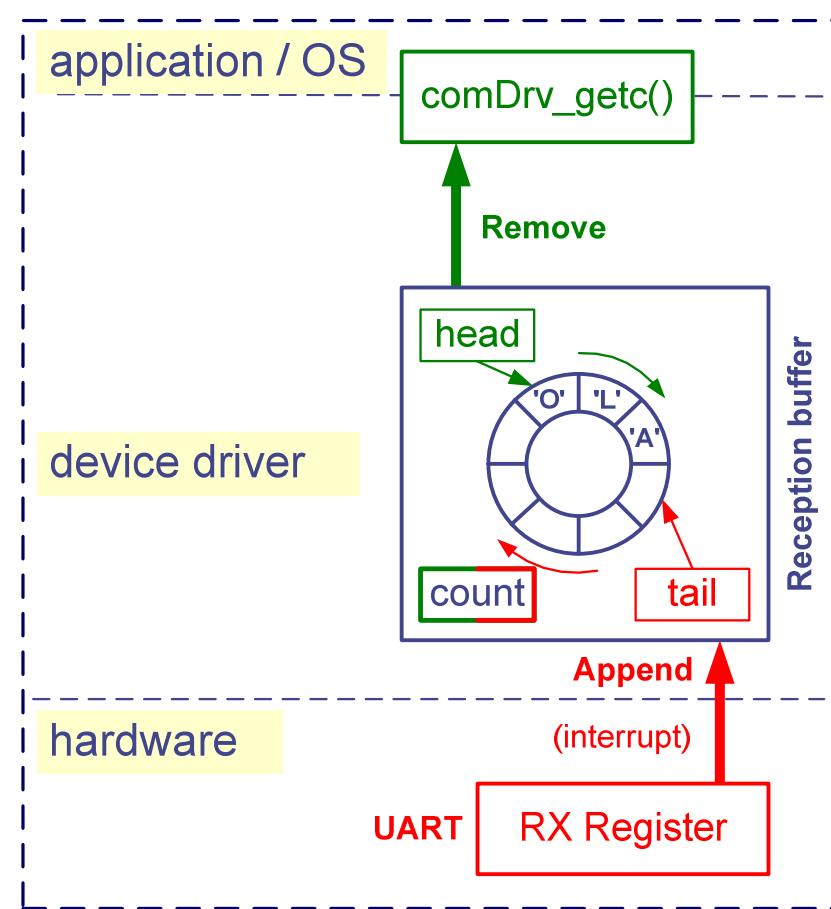
- Admitindo que é fornecida uma biblioteca que apresenta a seguinte interface:
  - `void comDrv_init(int baudrate, char dataBits,  
                      char parity, char stopBits);`
  - `char comDrv_getc(void); // read a character`
  - `void comDrv_putc(char ch); // write a character`
  - `void comDrv_close(void);`
- **Do ponto de vista da aplicação:**
  - Do ponto de vista funcional é relevante qual o modelo/fabricante do dispositivo de comunicação série?
  - Se o dispositivo de comunicação for substituído por outro com arquitetura interna distinta, sendo fornecida uma biblioteca com interface compatível, é necessário alterar a aplicação?

# Caso de estudo

- Aspetos-chave da implementação de um *device driver* para uma UART RS232 (*Universal Asynchronous Receiver Transmitter*) para executar num sistema com microcontrolador (i.e., sem sistema operativo)
- Princípio de operação
  - Desacoplamento da transferência de dados entre a UART e a aplicação, realizada por meio de FIFOs (um FIFO de transmissão e um de receção). Do ponto de vista da aplicação:
    - A **transmissão** consiste em copiar os dados a enviar para o FIFO de transmissão do *device driver*
    - A **recepção** consiste em ler os dados recebidos que residem no FIFO de receção do *device driver*
  - A transferência de dados entre os FIFOs e a UART é realizada por interrupção, i.e., sem intervenção explícita da aplicação
  - Um FIFO pode ser implementado através de um *buffer* circular

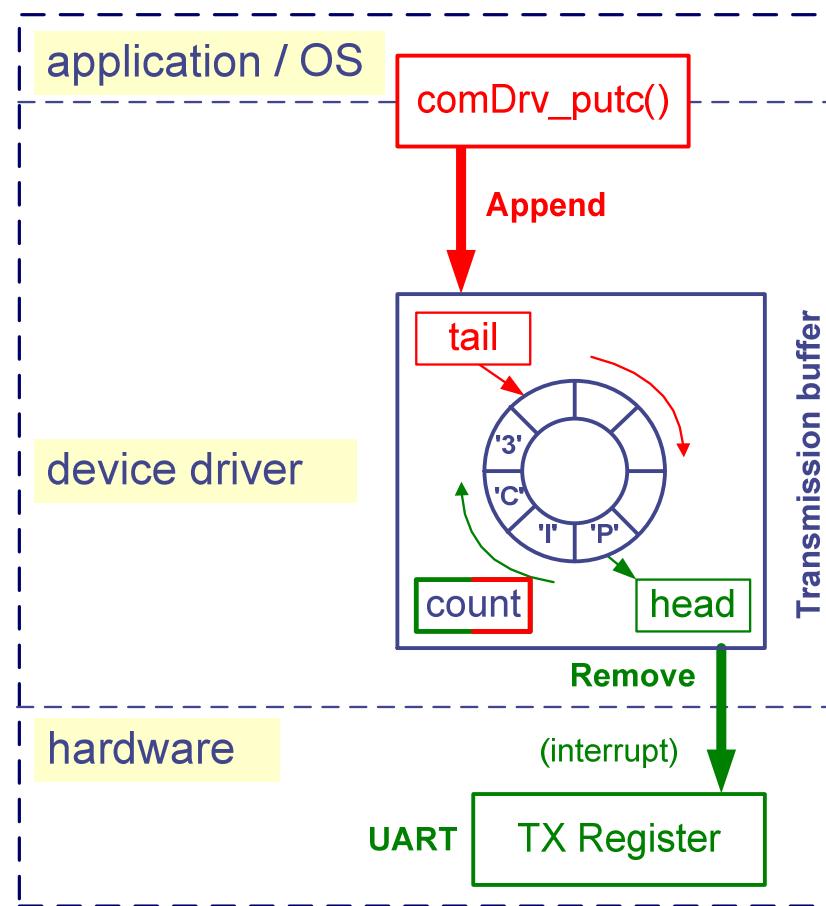
# Princípio de operação – receção

- "**tail**" – posição do buffer circular onde a **rotina de serviço à interrupção escreve** o próximo caracter lido da UART
- "**head**" – posição do buffer circular de onde a função **comDrv\_getc()** lê o próximo caracter
- "**count**" – número de caracteres residentes no buffer circular (ainda não lidos pela aplicação)
- **O acesso à variável "count" tem que ser feito numa secção crítica do código. Porquê?**



# Princípio de operação – transmissão

- "**tail**" – posição do buffer circular onde a função **comDrv\_putc()** escreve o próximo caracter
- "**head**" – posição do buffer circular de onde a **rotina de serviço à interrupção lê** o próximo caracter a enviar para a UART
- "**count**" – número de caracteres residentes no buffer circular (ainda não enviados para a UART)
- **O acesso à variável "count" tem que ser feito numa secção crítica do código. Porquê?**



# Implementação – FIFO

- FIFO - Buffer circular implementado através de um *array* linear:

```
#define BUF_SIZE 32

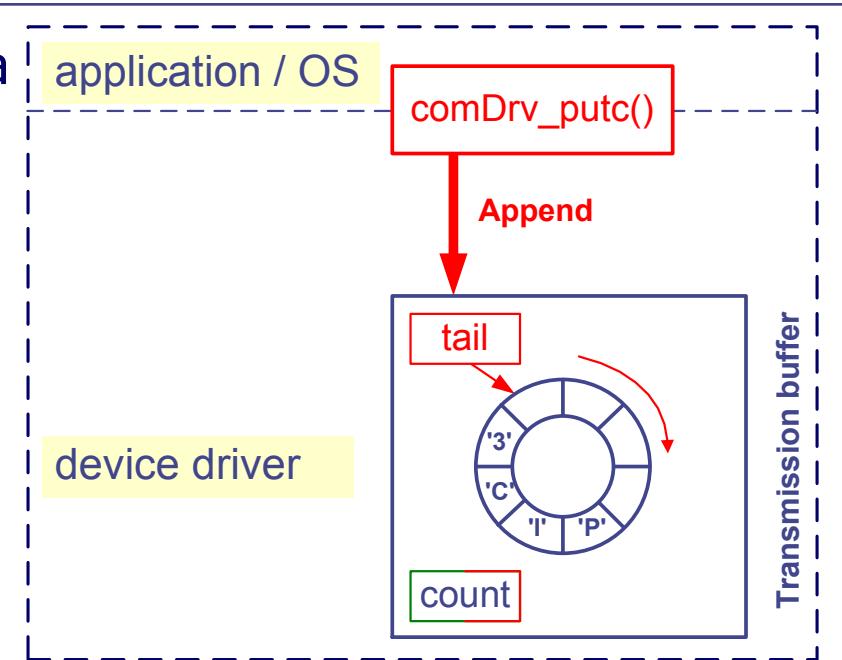
typedef struct
{
    unsigned char data[BUF_SIZE];
    unsigned int head;
    unsigned int tail;
    unsigned int count;
} circularBuffer;

circularBuffer txb; // Transmission buffer
circularBuffer rxb; // Reception buffer
```

- A constante "BUF\_SIZE" deve ser definida em função das necessidades previsíveis de pico de tráfego.
- Se "BUF\_SIZE" for uma potência de 2 simplifica a atualização dos índices do buffer circular (podem ser encarados como contadores módulo  $2^N$  e podem ser geridos com uma simples máscara)

# Implementação – Função de transmissão

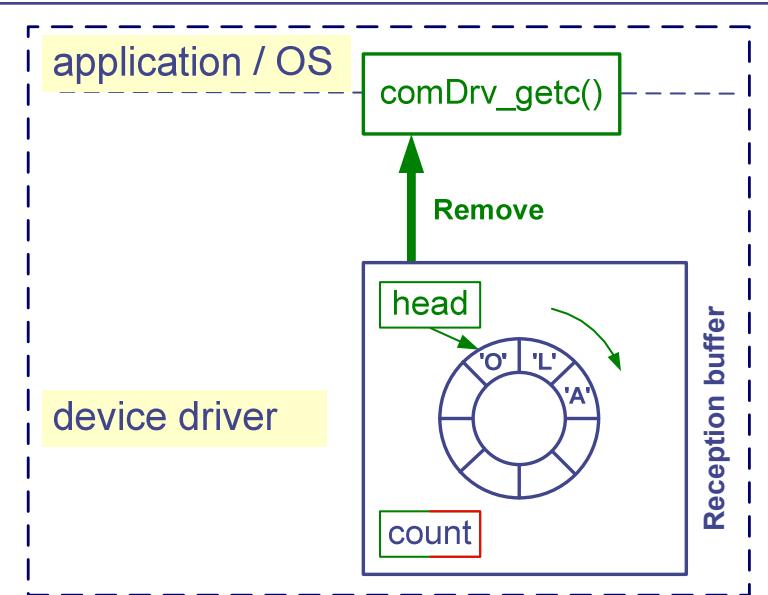
- A função de transmissão, evocada pela aplicação, copia o caracter para o **buffer de transmissão** (posição "tail"), incrementa o índice "tail" e o contador



```
void comDrv_putc(char ch)
{
    Wait while buffer is full (txb.count==BUF_SIZE)
    Copy "ch" to the buffer ("tail" position)
    Increment "tail" index (mod BUF_SIZE)
    Increment "count" variable
}
```

# Implementação – Função de receção

- A função de receção, evocada pela aplicação, verifica se há caracteres no **buffer de receção** para serem lidos e, caso haja, retorna o carácter presente na posição "**head**", incrementa o índice "**head**" e decrementa o contador

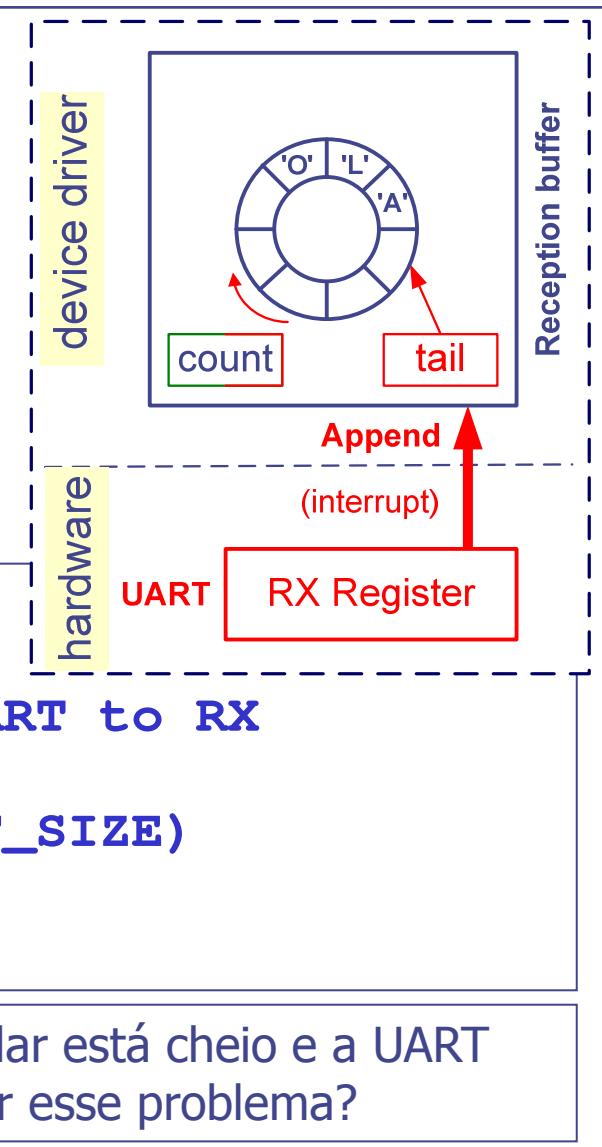


```
int comDrv_getc(char *pchar)
{
    If "count" variable is 0 then return false
    Copy character at position "head" to *pchar
    Increment "head" index (mod BUF_SIZE)
    Decrement "count" variable
    return true;
}
```

# Implementação – RSI de receção

- A rotina de serviço à interrupção da receção é executada sempre que a UART recebe um novo caractere
- O caractere recebido pela UART deve então ser copiado para o **buffer de receção**, na posição "**tail**"; a variável "**count**" deve ser incrementada e o índice "**tail**" deve ser igualmente incrementado

```
void interrupt isr_rx(void)
{
    Copy received character from UART to RX
        buffer ("tail" position)
    Increment "tail" index (mod BUF_SIZE)
    Increment "count" variable
}
```

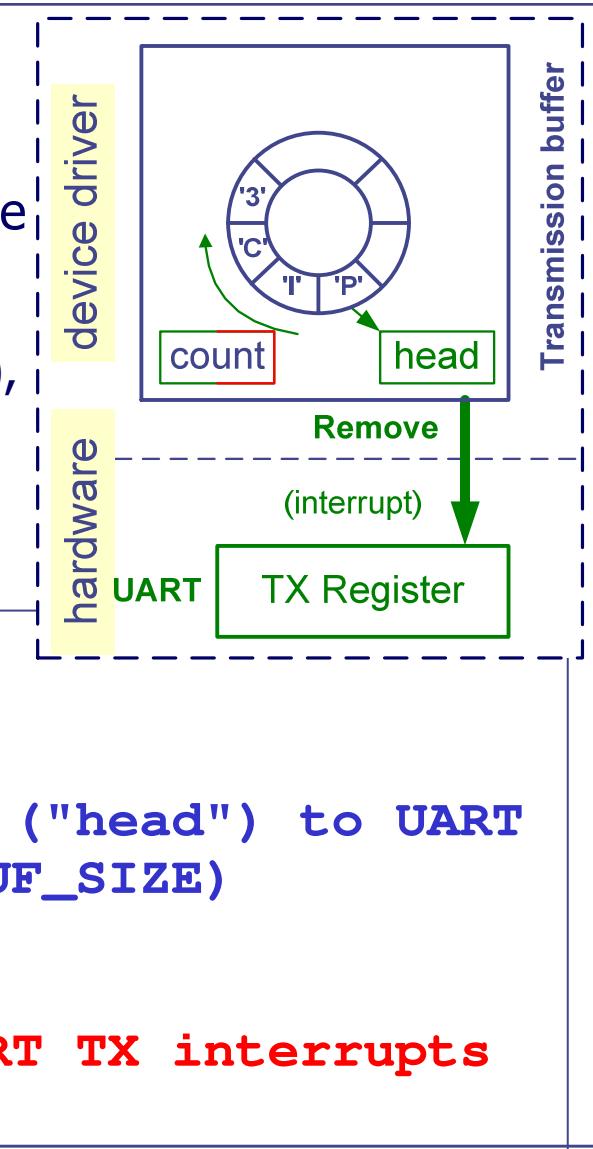


O que acontece no caso em que o *buffer* circular está cheio e a UART recebe um novo caractere? Como resolver esse problema?

# Implementação – RSI de transmissão

- A UART gera, normalmente, uma interrupção de transmissão quando tiver disponibilidade para transmitir um novo caractere
- As tarefas a implementar na respetiva rotina de serviço à interrupção são:
  - Se o número de caracteres no **buffer de transmissão** for maior que 0 ("count" > 0), copiar o conteúdo do buffer na posição "head" para a UART
  - Decrementar a variável "count" e incrementar o índice "head"

```
void interrupt isr_tx(void)
{
    If "count" > 0 then {
        Copy character from TX buffer ("head") to UART
        Increment "head" index (mod BUF_SIZE)
        Decrement "count" variable
    }
    If "count" == 0 then disable UART TX interrupts
}
```



# Atualização do TX "count" - Secção crítica

```
void comDrv_putc(char ch)
{
    Wait while buffer is full (count==BUF_SIZE)
    Copy "ch" to the transmission buffer ("tail")
    Increment "tail" index (mod BUF_SIZE)
    Disable UART TX interrupts
    Increment "count" variable
    Enable UART TX interrupts
}

void interrupt isr_tx(void)
{
    if "count" > 0 then
    {
        Copy character from TX buffer ("head") to UART
        Increment "head" index (mod BUF_SIZE)
        Decrement "count" variable
    }
    if "count" == 0 then disable UART TX interrupts
}
```

**Secção crítica  
("count" é um  
recurso partilhado)**

**Se a UART estiver disponível para  
transmitir, desencadeia a imediata  
geração da interrupção de transmissão**

# Atualização do RX "count" - Secção crítica

```
int comDrv_getc(char *pchar)
{
    If "count" variable is 0 then return false
    Copy character at position "head" to *pchar
    Increment "head" index (mod BUF_SIZE)
    Disable UART RX interrupts
    Decrement "count" variable
    Enable UART RX interrupts
    return true;
}
```

Secção crítica  
("count" é um  
recurso partilhado)

```
void interrupt isr_rx(void)
{
    Copy received character from UART to RX buffer
        ("tail" position)
    Increment "tail" index (mod BUF_SIZE)
    Increment "count" variable
}
```

# Aulas 18 e 19

- Tecnologias de memória
- Organização genérica de um circuito de memória a partir de uma célula básica
- Memória SRAM (*Static Random Access Memory*) :
  - organização de células básicas num array
  - ciclos de acesso para leitura e escrita: diagramas temporais
  - construção de módulos de memória SRAM
- Memória DRAM (*Dynamic Random Access Memory*) :
  - célula básica; organização interna
  - ciclos de acesso para leitura e escrita: diagramas temporais
  - refrescamento: modo "RAS only"
  - construção de módulos de memória DRAM

José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

# Introdução – conceitos básicos

- RAM – Random Access Memory
  - Designação para memória volátil que pode ser lida e escrita
  - Acesso "random"
- ROM – Read Only Memory
  - Memória não volátil que apenas pode ser lida
  - Acesso "random"

(Acesso "random" - tempo de acesso é o mesmo para qualquer posição de memória)

# Introdução – conceitos básicos

- Tecnologias:
  - Semicondutor
  - Magnética
  - Ótica
  - Magneto-ótica
- Memória volátil:
  - Informação armazenada perde-se quando o circuito é desligado da alimentação: RAM (SRAM e DRAM)
- Memória não volátil:
  - A informação armazenada mantém-se até ser deliberadamente alterada: EEPROM, Flash EEPROM, tecnologias magnéticas

# Memória não volátil – evolução histórica

- **ROM** – programada durante o processo de fabrico (1965)
- **PROM** – Programmable Read Only Memory: programável uma única vez (1970)
- **EPROM** – Erasable PROM: escrita em segundos, apagamento em minutos (ambas efectuadas em dispositivos especiais) (1971)
- **EEPROM** – Electrically Erasable PROM (1976)
  - O apagamento e a escrita podem ser efetuados no próprio circuito em que a memória está integrada
  - O apagamento é feito byte a byte
  - Escrita muito mais lenta que leitura
- **Flash EEPROM** (tecnologia semelhante à EEPROM) (1985)
  - A escrita pressupõe o prévio apagamento das zonas de memória a escrever
  - O apagamento é feito por blocos (por exemplo, blocos de 4 kB) o que torna esta tecnologia mais rápida que a EEPROM
  - O apagamento e a escrita podem ser efetuados no próprio circuito em que a memória está integrada
  - Escrita muito mais lenta que leitura

# Tecnologias de memória

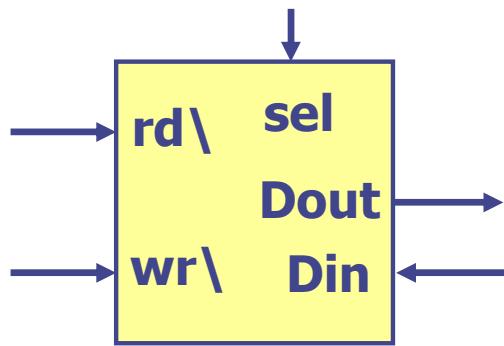
Tecnologia	Tempo Acesso	\$ / GB
SRAM	0,5 – 2,5 ns	\$500 - \$1000
DRAM	35 - 70 ns	\$10 - \$20
Flash	5 – 50 us	\$0,75 - \$1
Magnetic Disk	5 - 20 ms	\$0,005 - \$0,1

(Dados de 2012)

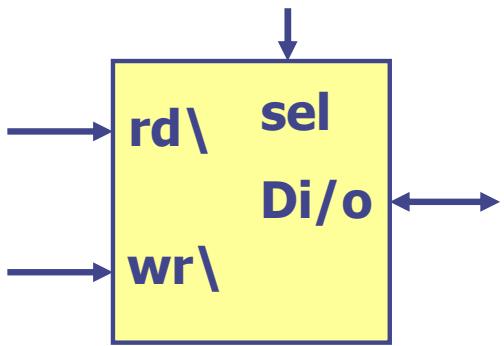
- SRAM - Static Random Access Memory
- DRAM - Dynamic Random Access Memory
- Dadas estas diferenças de custo e de tempo de acesso, é vantajoso construir o sistema de memória como uma hierarquia onde se utilizem todas estas tecnologias

# Organização básica de memória

- Uma memória pode ser encarada como uma coleção de N registos de dimensão K ( $N \times K$ )
- Cada registo é formado por K células, cada uma delas capaz de armazenar 1 bit
- Uma célula de memória (de 1 bit) pode ser representada por:



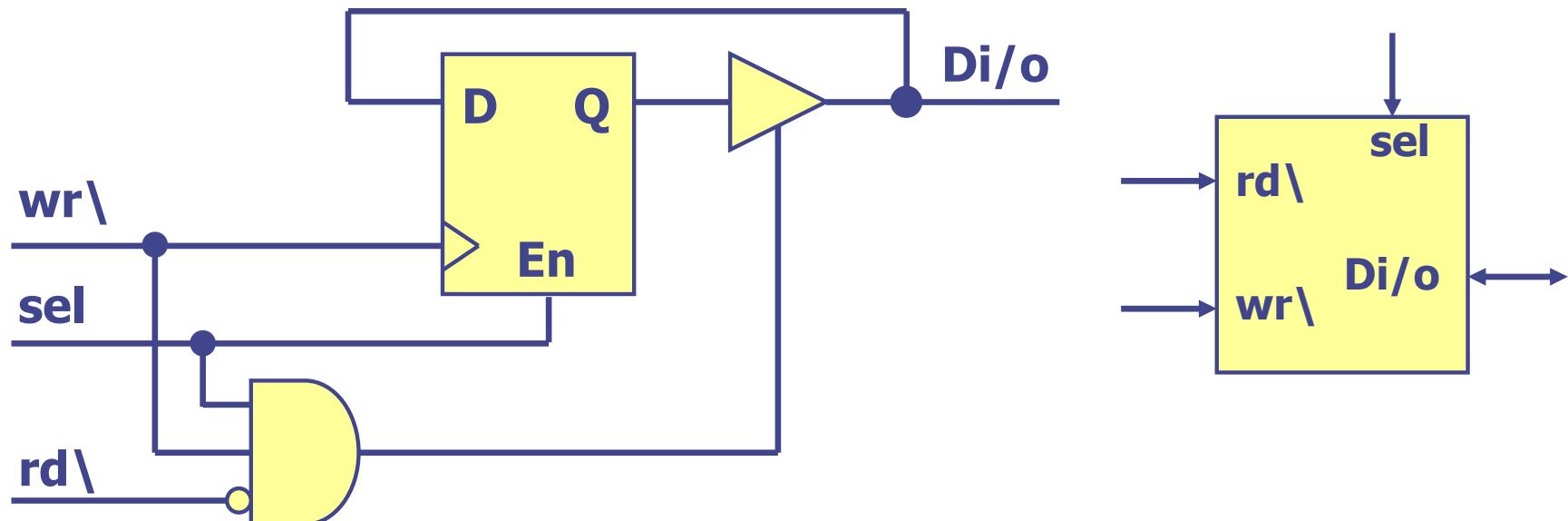
Din	– Data In (1 bit)
Dout	– Data Out (1 bit)
sel	– Select
rd\'	– Read\
rw\'	– Write\



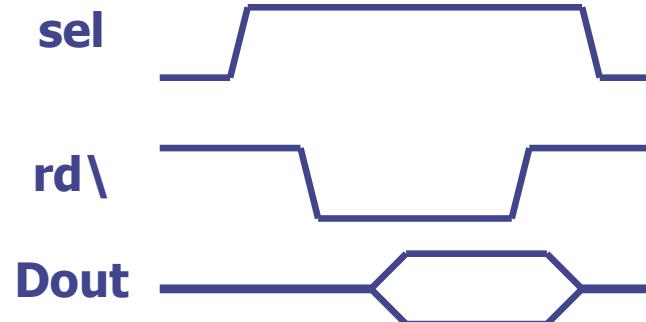
Di/o	– Data In/Out (1 bit)
sel	– Select
rd\'	– Read\
rw\'	– Write\

# Organização básica de memória

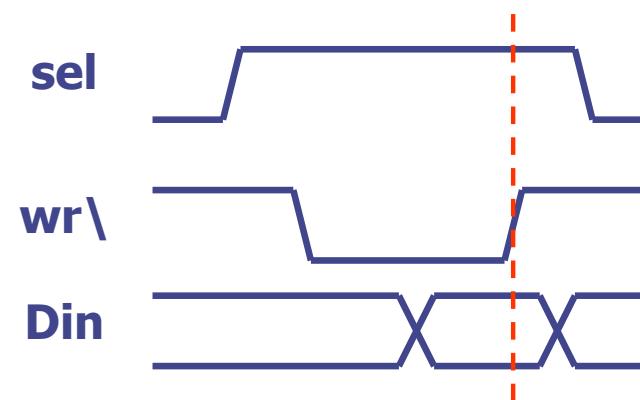
- Uma possível implementação de uma célula de memória é:



Operação de leitura



Operação de escrita

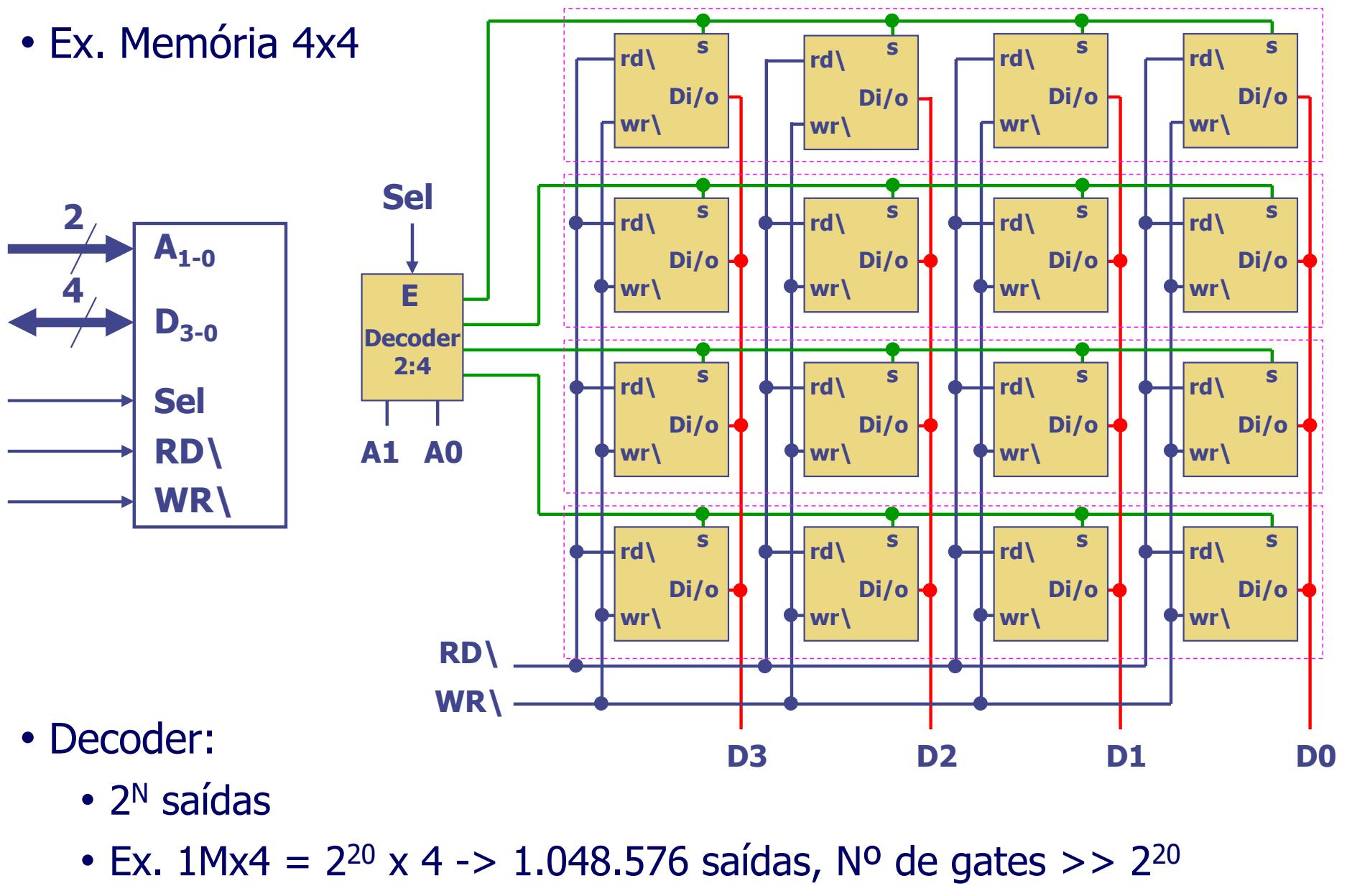


# Agrupamento de células de memória

- Através do agrupamento de células-base pode formar-se uma memória de maior dimensão
- O que é necessário especificar:
  - O **número total de Words (N)** que a memória pode armazenar
  - O **Word size (K = 1, 4, 8, 16, 32, ...)**  
(Número total de bits = word size \* nº words)
- Exemplos:
- 1k x 8
  - 8 bits / word
  - $1k = 2^{10} \rightarrow 10$  linhas de endereço  $\rightarrow 1.024$  endereços
- 1M x 4
  - 4 bits / word
  - $1M = 2^{20} \rightarrow 20$  linhas de endereço  $\rightarrow 1.048.576$  endereços

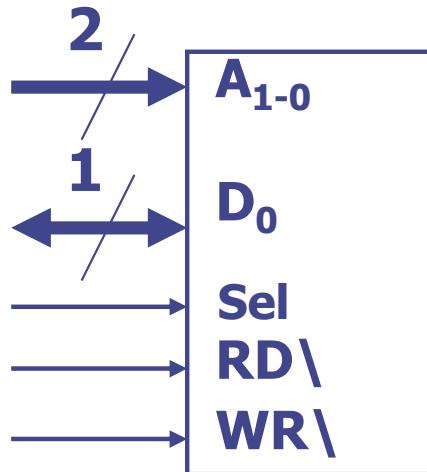
# Organização 2D

- Ex. Memória 4x4

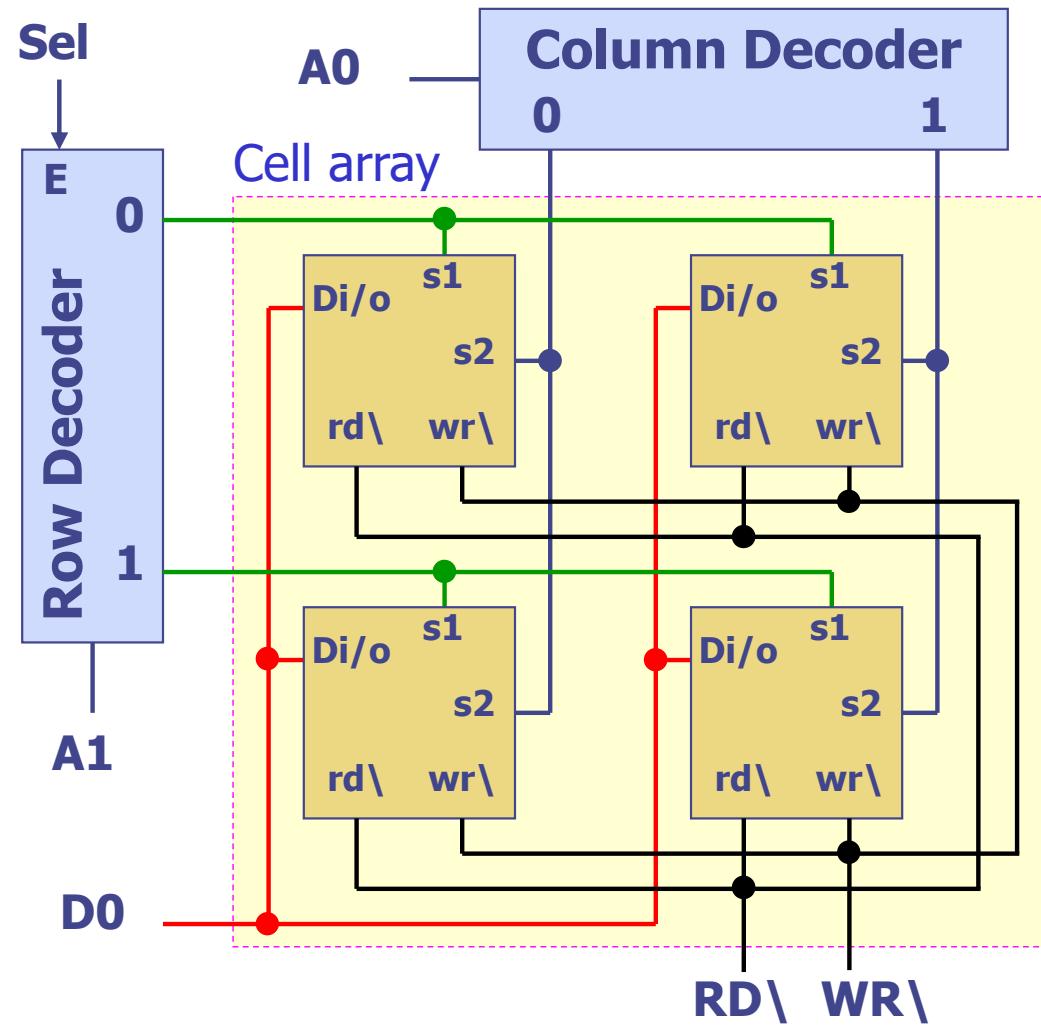


# Organização em matriz (conceito)

- Ex. Memória 4x1



(Célula seleccionada: S<sub>1</sub>.S<sub>2</sub>=1)



Q1. E se a memória fosse de 16x1? e se fosse 8x1? e 1Mx1?

Q2. E se a memória fosse de 4x2? E se fosse 4x4?

# Memória do tipo RAM (volátil)

- **SRAM – Static RAM**

- Vantagens:

- Rápida
    - Informação permanece até que a alimentação seja cortada

- Inconvenientes:

- Implementações típicas: 6 transistores / célula
    - Baixa densidade, elevada dissipação de potência
    - Custo/bit elevado

- **DRAM – Dynamic RAM**

- Vantagens:

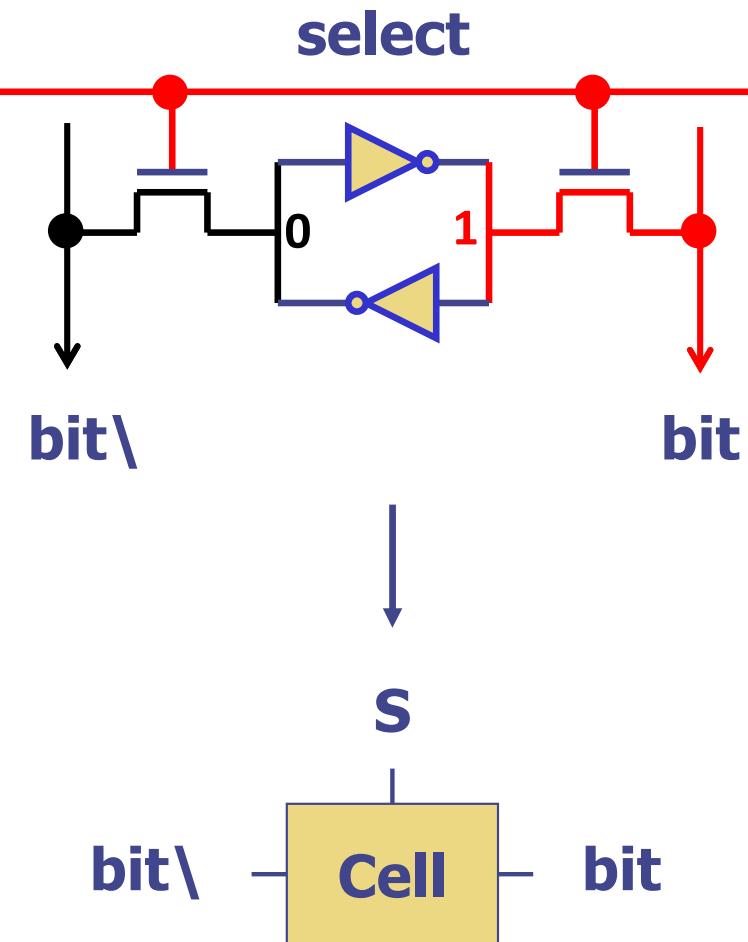
- Implementações típicas: (1 transistor + 1 condensador) / célula
    - Alta densidade, baixa dissipação de potência
    - Custo/bit baixo

- Inconvenientes:

- Informação permanece apenas durante alguns mili-segundos (necessita de *refresh* regular – daí a designação "dynamic")
    - Mais lenta (pelo menos 1 ordem de grandeza) que a SRAM

# RAM estática (SRAM)

- 6 transístores / célula



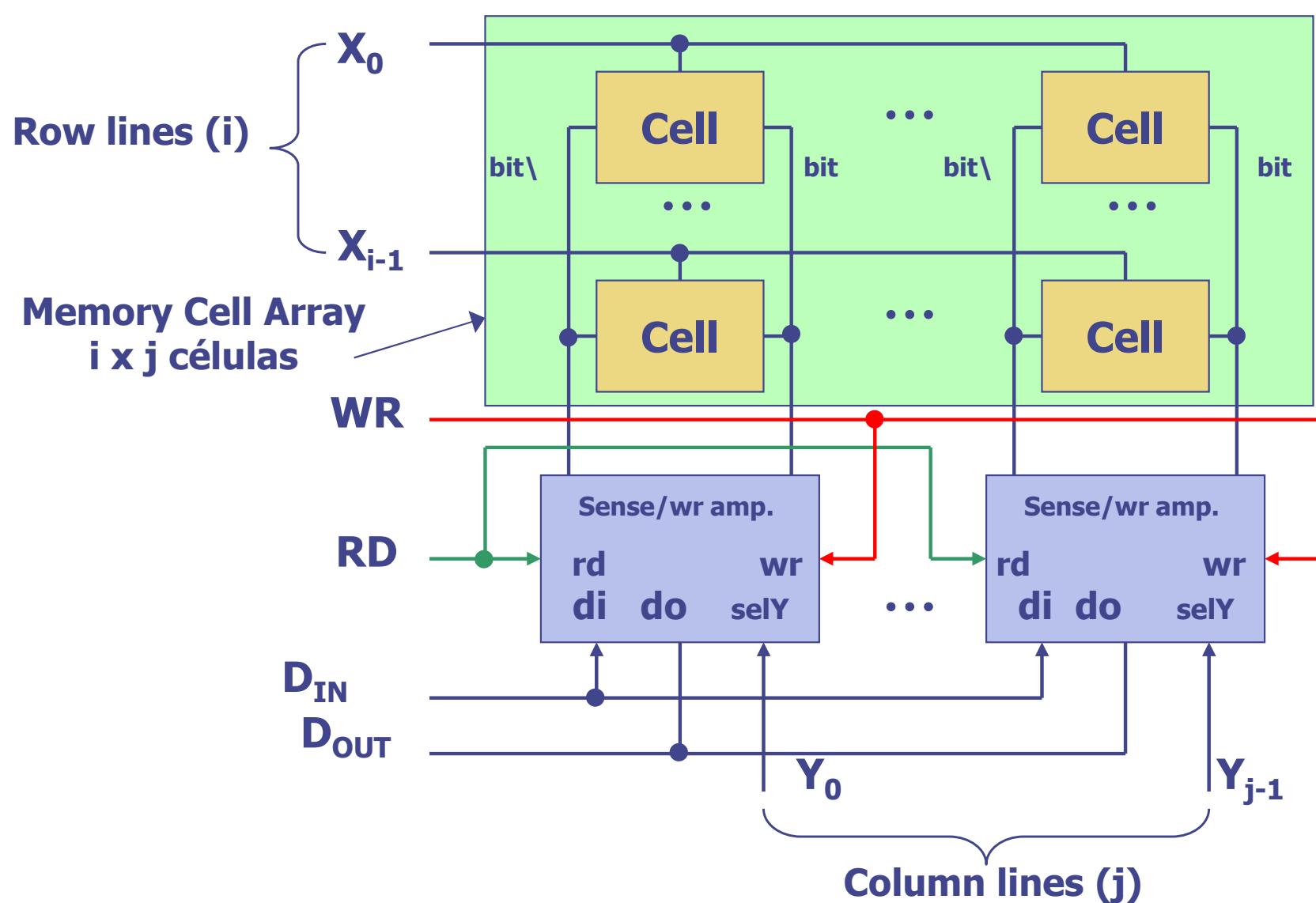
- **Write**

- Colocar a informação em "bit" (e "bit\"). Exemplo: para a escrita do valor lógico "1" – "bit"=1, "bit\ "=0
- Ativar a linha "select"

- **Read**

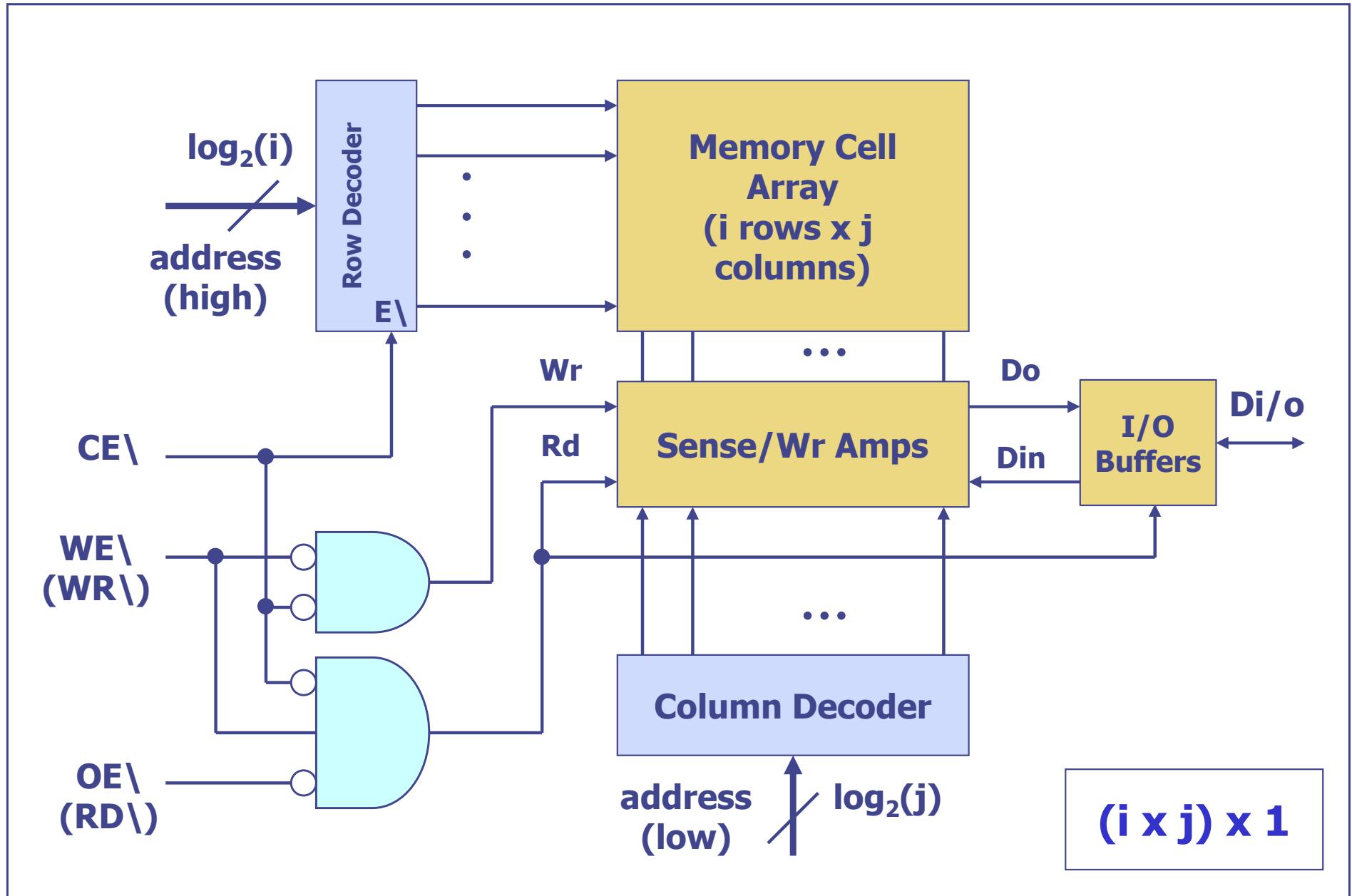
- Ativar a linha "select"
- O valor lógico armazenado na célula é detetado pela diferença de tensão entre as linhas "bit" e "bit\"

# SRAM - Organização interna

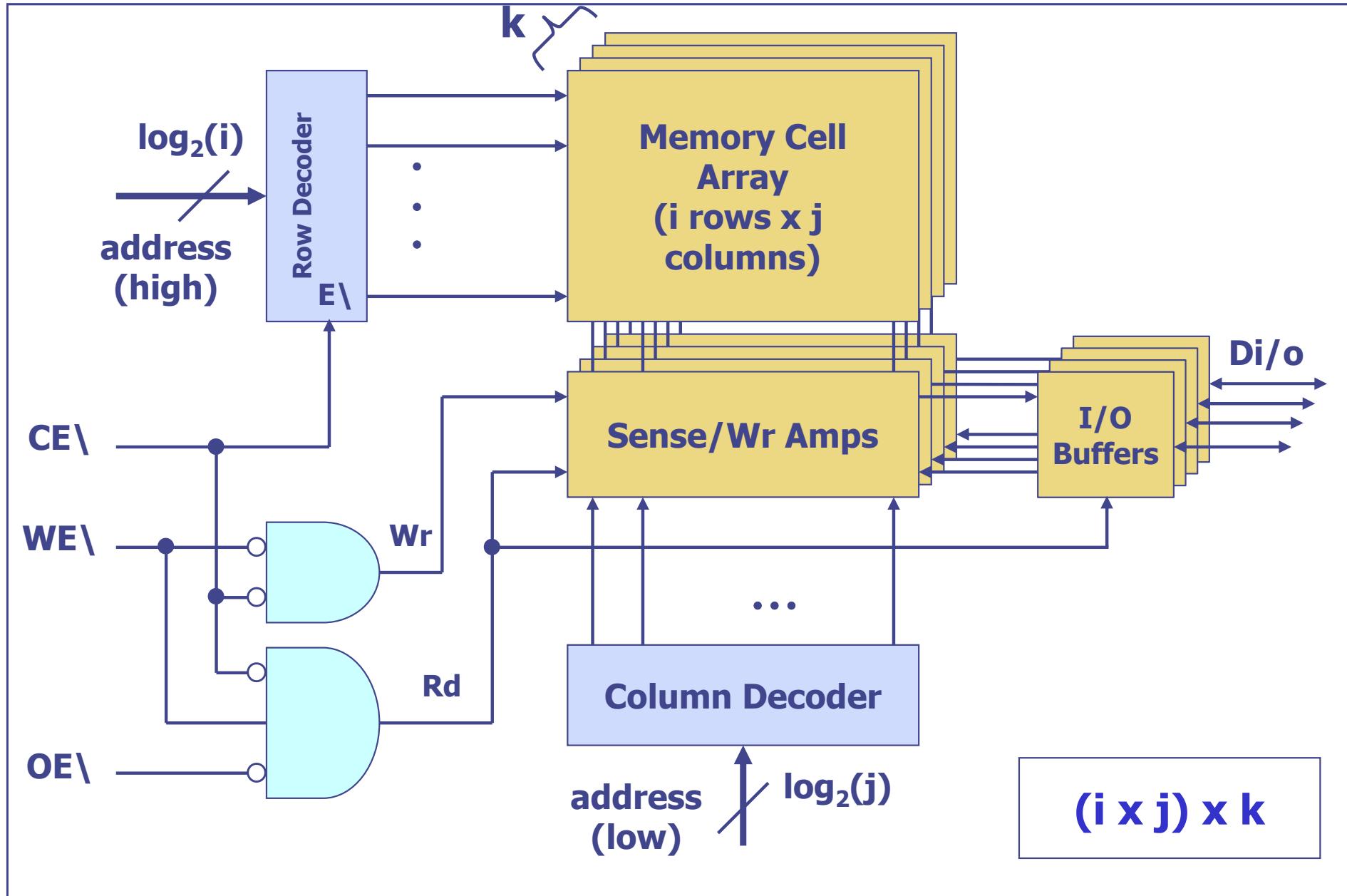


- Se  $selY=0$ , as linhas "bit" e "bit\" ficam em alta impedância

# SRAM - Organização interna

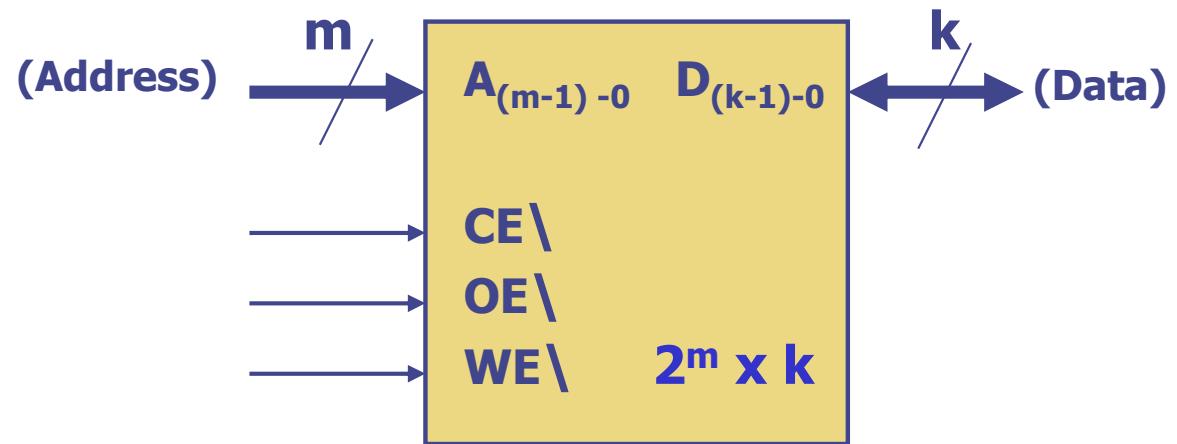


# SRAM - Organização interna



# SRAM - Bloco funcional

- Diagrama lógico (interface assíncrona)

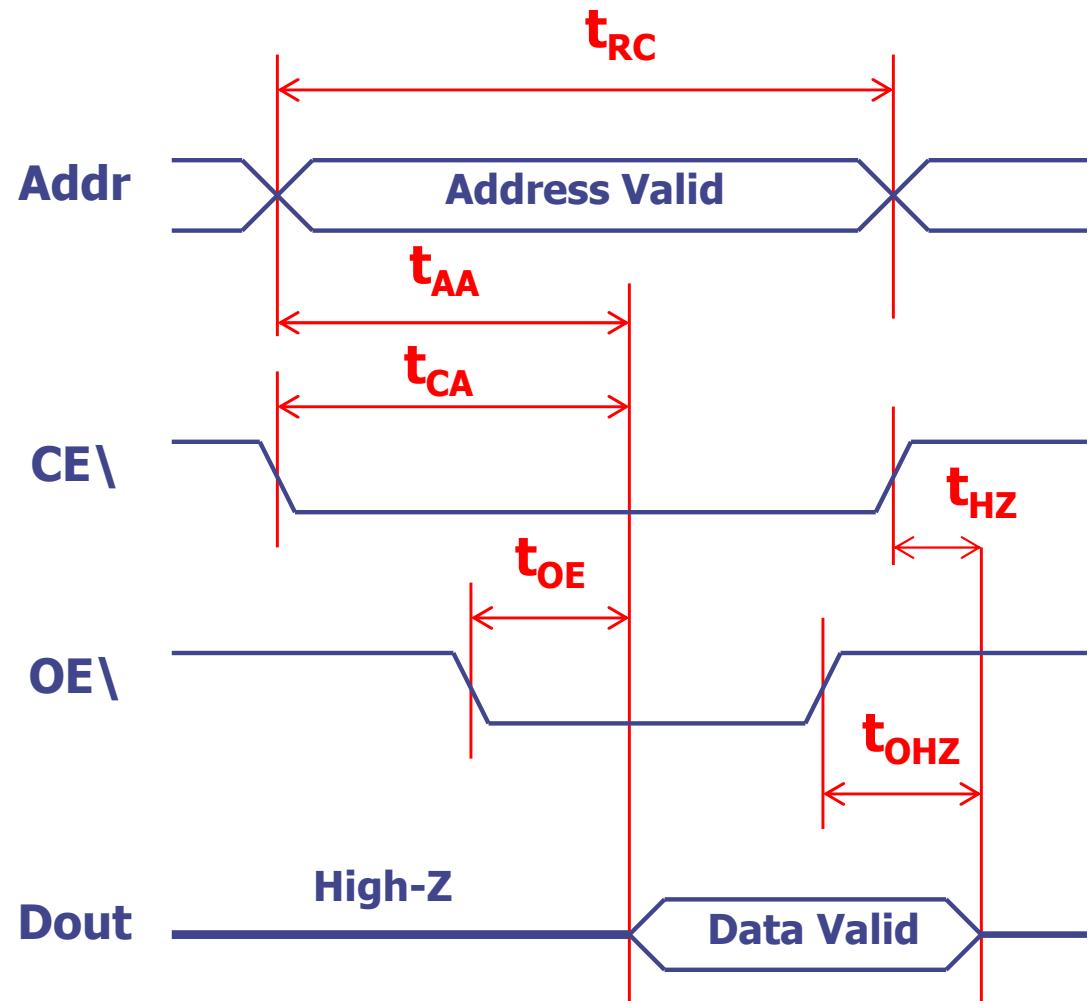


- Tabela de verdade

CE\	OE\	WE\	Operação
1	X	X	High-Z
0	1	1	High-Z
0	X	0	Escrita
0	0	1	Leitura

# SRAM – Ciclo de Leitura

- Diagrama temporal típico de um ciclo de leitura de uma memória SRAM (interface assíncrona)



$t_{RC}$	- Read Cycle Time
$t_{AA}$	- Address Access Time
$t_{CA}$	- CE\ Access Time
$t_{OE}$	- Output Enable to Output Valid
$t_{HZ}$	- CE\ to Output in High-Z
$t_{OHZ}$	- OE\ to Output in High-Z

# SRAM – Ciclo de leitura

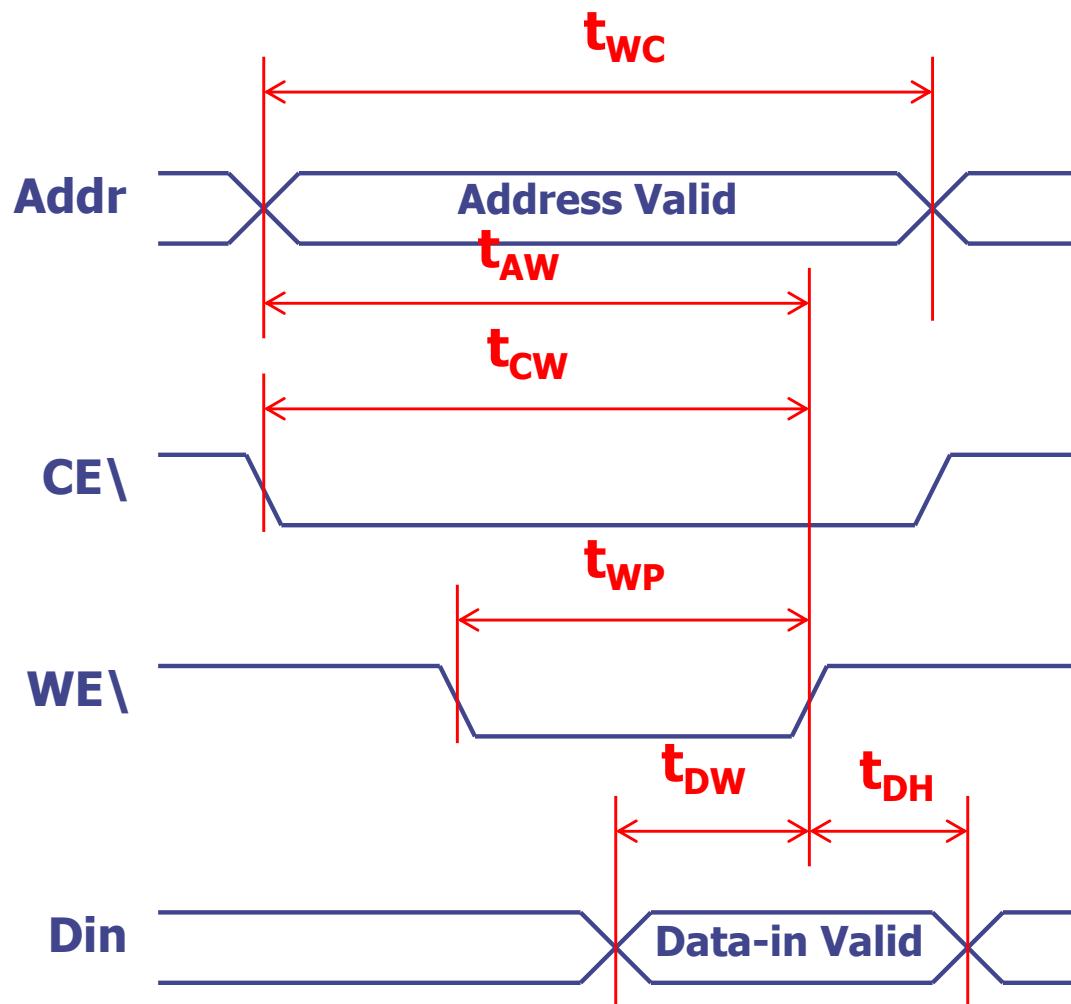
- Valores indicativos (em ns) dos parâmetros associados a um ciclo de leitura de uma memória SRAM:

Parameter	Symbol	Min.	Max.
Read Cycle Time	$t_{RC}$	1.5	
Address Access Time	$t_{AA}$		1.5
CE\ Access Time	$t_{CA}$		1.5
Output Enable to Output Valid	$t_{OE}$		0.7
CE\ to Output in High-Z	$t_{HZ}$		0.6
OE\ to Output in High-Z	$t_{OHZ}$		0.6

- **Cycle Time**: tempo de acesso mais qualquer tempo adicional necessário antes que um segundo acesso possa ter início
- **Access Time**: tempo necessário para os dados ficarem disponíveis no barramento de saída da memória
- **Taxa de transferência**: taxa a que os dados podem ser transferidos de/para uma memória ( $1 / \text{cycle\_time}$ )

# SRAM – Ciclo de Escrita

- Diagrama temporal típico de um ciclo de escrita de uma memória SRAM



$t_{WC}$  - Write Cycle Time

$t_{AW}$  - Address Valid to End of Write

$t_{CW}$  - CE\ to End of Write

$t_{WP}$  - Write Pulse Width

$t_{DW}$  - Data Valid to End of Write  
(data setup time)

$t_{DH}$  - Data Hold Time

# SRAM – Ciclo de Escrita

- Valores indicativos (em ns) dos parâmetros associados a um ciclo de escrita de uma memória SRAM:

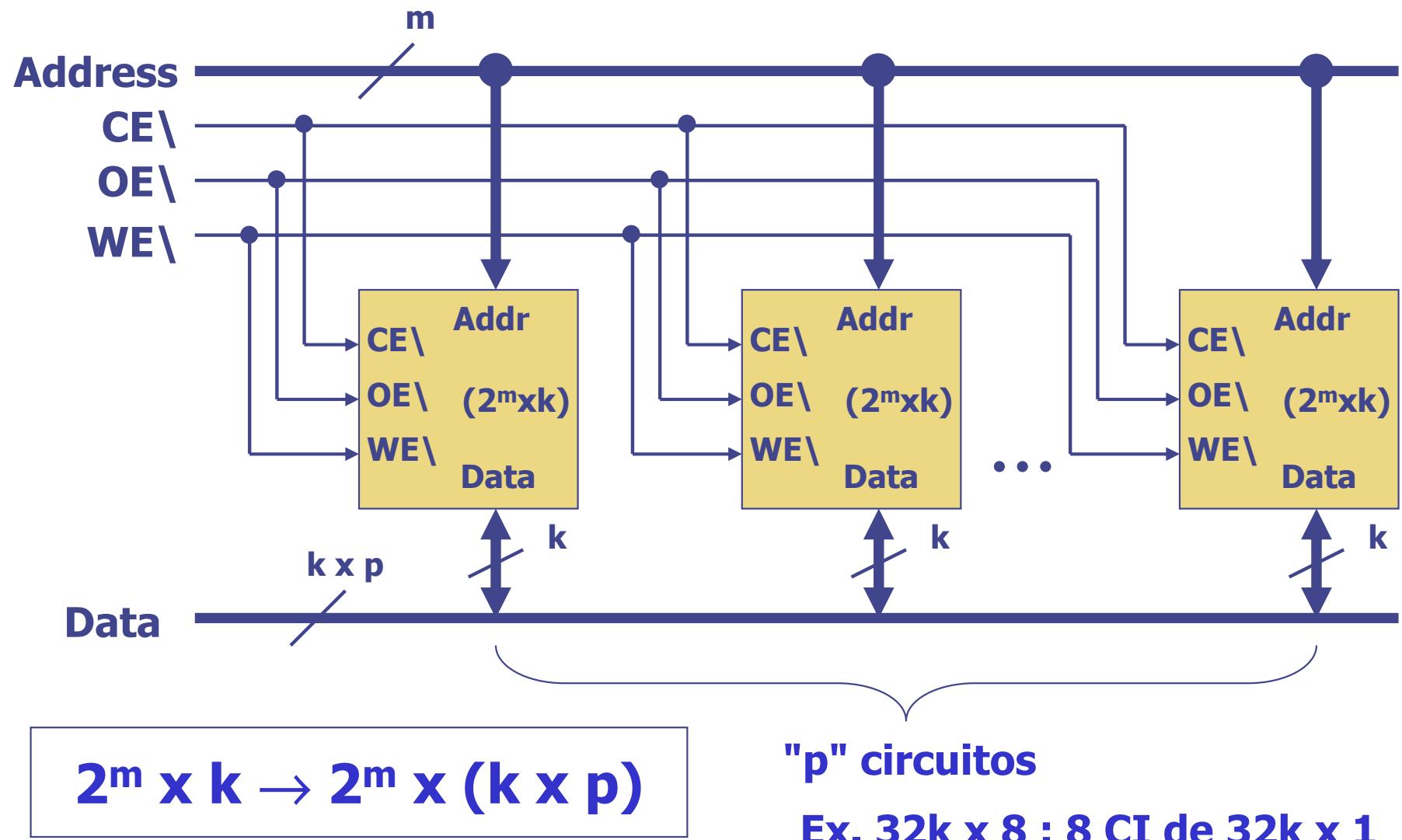
Parameter	Symbol	Min.	Max.
Write Cycle Time	$t_{WC}$	1.5	
Address Valid to End of Write	$t_{AW}$	1.0	
CE\ to End of Write	$t_{CW}$	1.0	
Write Pulse Width	$t_{WP}$	1.0	
Data Valid to End of Write	$t_{DW}$	0.7	
Data Hold Time	$t_{DH}$	0	

# Aumento da capacidade de armazenamento

- É frequente ter-se necessidade de memórias com uma capacidade de armazenamento superior à capacidade individual dos circuitos disponíveis comercialmente
- Nessa situação recorre-se à construção de módulos de memória que resultam do agrupamento de circuitos de acordo com o aumento pretendido
- Assim, a construção de um módulo de memória pode envolver as duas fases seguintes, ou apenas uma delas, em função dos circuitos disponíveis e dos requisitos finais de armazenamento:
  - **Aumento da dimensão da palavra.** Exemplo: a partir de C.I.s de 32K x 1, construir uma memória de 32K x 8
  - **Aumento do número total de posições de memória.** Exemplo: a partir de C.I.s de 32K x 8, construir uma memória de 128K x 8

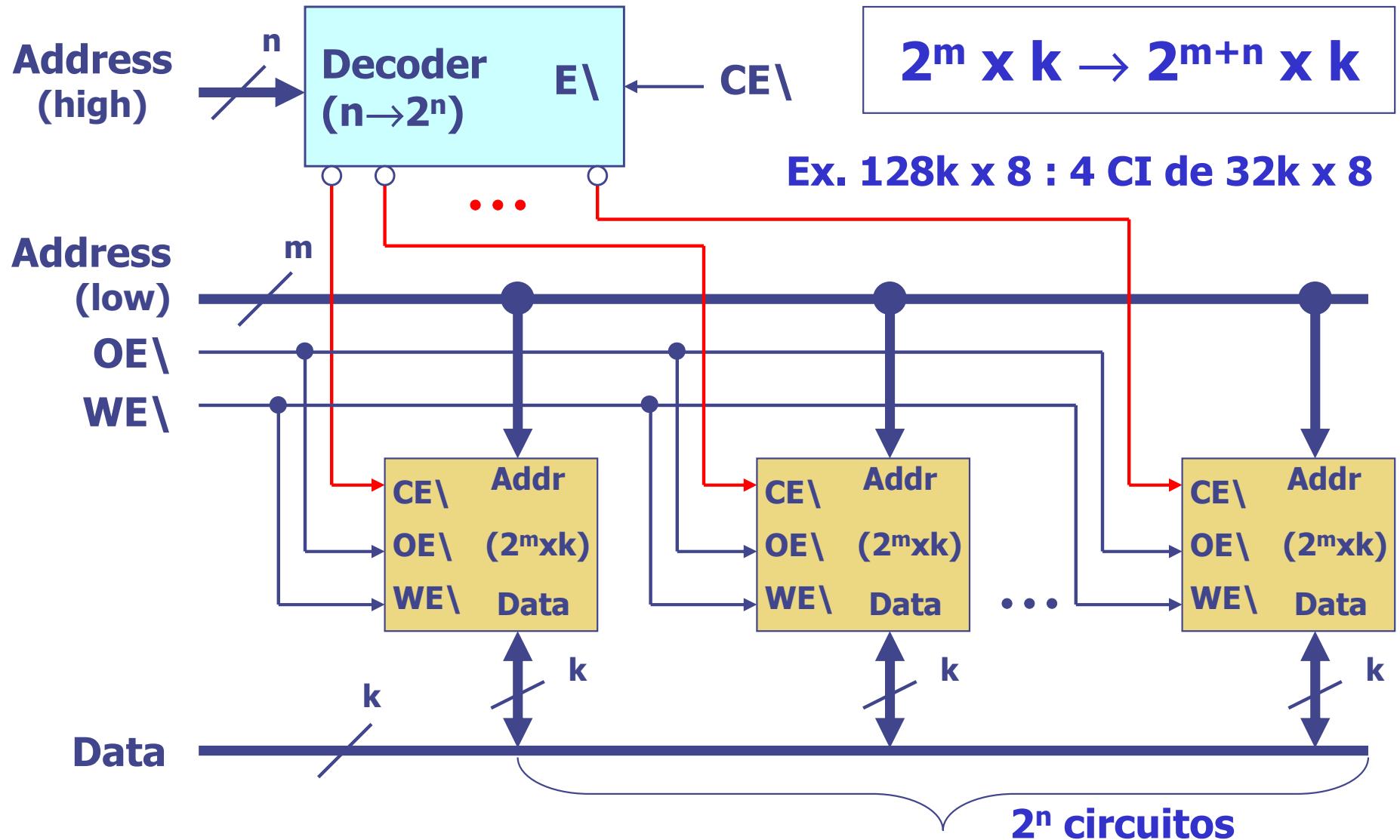
# Módulo de memória SRAM

- Aumento da dimensão da palavra



# Módulo de memória SRAM

- Aumento do número total de posições de memória



# Memória do tipo RAM (volátil)

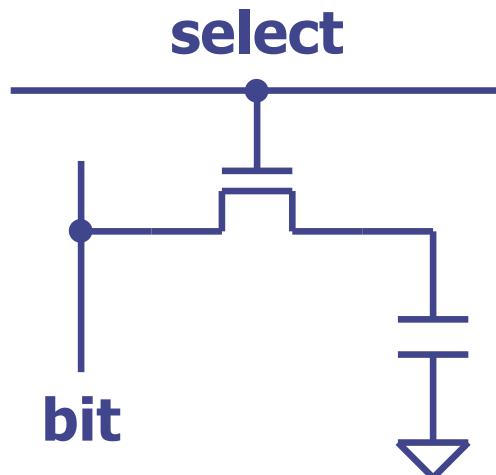
- **SRAM – Static RAM**

- Vantagens:
  - Rápida
  - Informação permanece até que a alimentação seja cortada
- Inconvenientes:
  - Implementações típicas: 6 transistores / célula
  - Baixa densidade, elevada dissipação de potência
  - Custo/bit elevado

- **DRAM – Dynamic RAM**

- Vantagens:
  - Implementações típicas: (1 transistor + 1 condensador) / célula
  - Alta densidade, baixa dissipação de potência
  - Custo/bit baixo
- Inconvenientes:
  - Informação permanece apenas durante alguns mili-segundos (necessita de *refresh* regular – daí a designação "dynamic")
  - Mais lenta (pelo menos 1 ordem de grandeza) que a SRAM

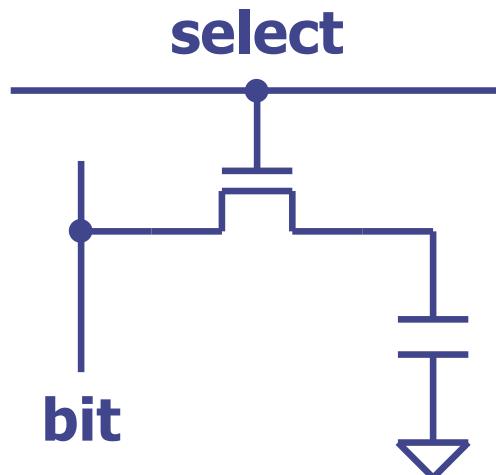
# RAM Dinâmica (DRAM)



- Condensador com uma capacidade muito pequena (dezenas de fF ( $1 \text{ fF} = 10^{-15} \text{ F}$ ))

- Na ausência de leitura, o condensador descarrega "lentamente"
- Informação permanece na célula apenas durante alguns milisegundos
- É obrigatório fazer o refreshamento ("refresh") periódico da carga do condensador
- A operação de leitura é destrutiva (descarrega o condensador)
- Após uma operação de leitura é necessário repor a carga no condensador

# RAM Dinâmica (DRAM)



- **Write**

- Colocar dado na linha "bit"
- Ativar a linha "select"

- **Read**

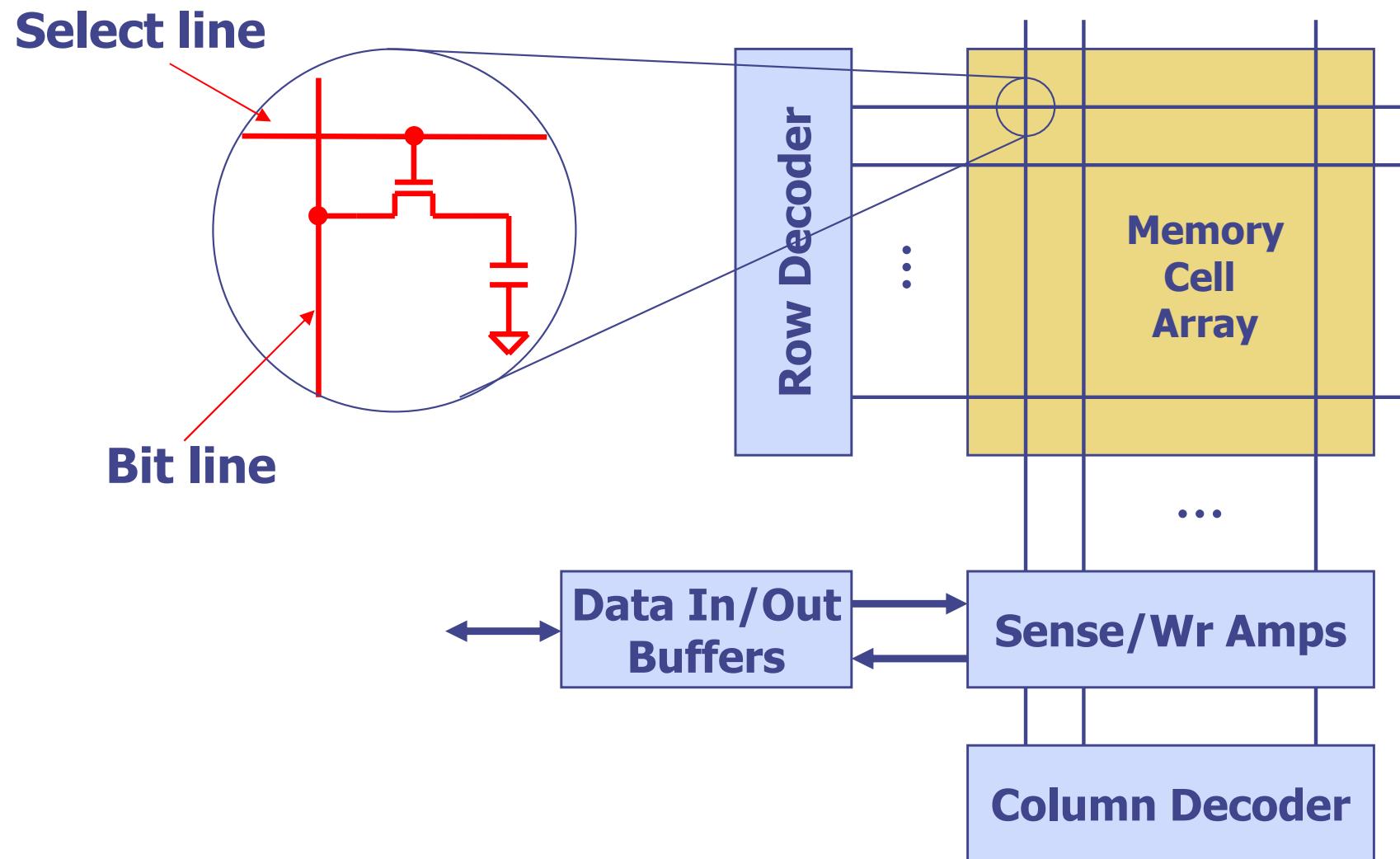
- Pre-carregar a linha "bit" a  $VDD/2$
- Ativar a linha "select"
- Valor lógico detetado pela diferença de tensão na linha bit, relativamente a  $VDD/2$
- Restauro do valor da tensão no condensador (write)

- **Refresh da célula**

- Operação interna idêntica a uma operação de "Read"

# RAM Dinâmica (DRAM)

- Organização em matriz

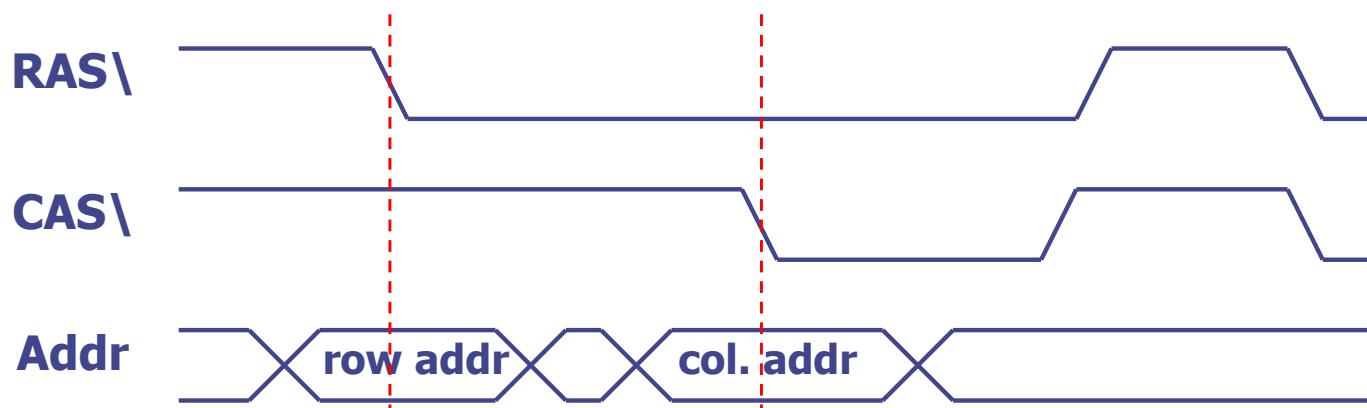


# RAM Dinâmica (DRAM)

- O endereço de acesso à memória é dividido em 2 partes:

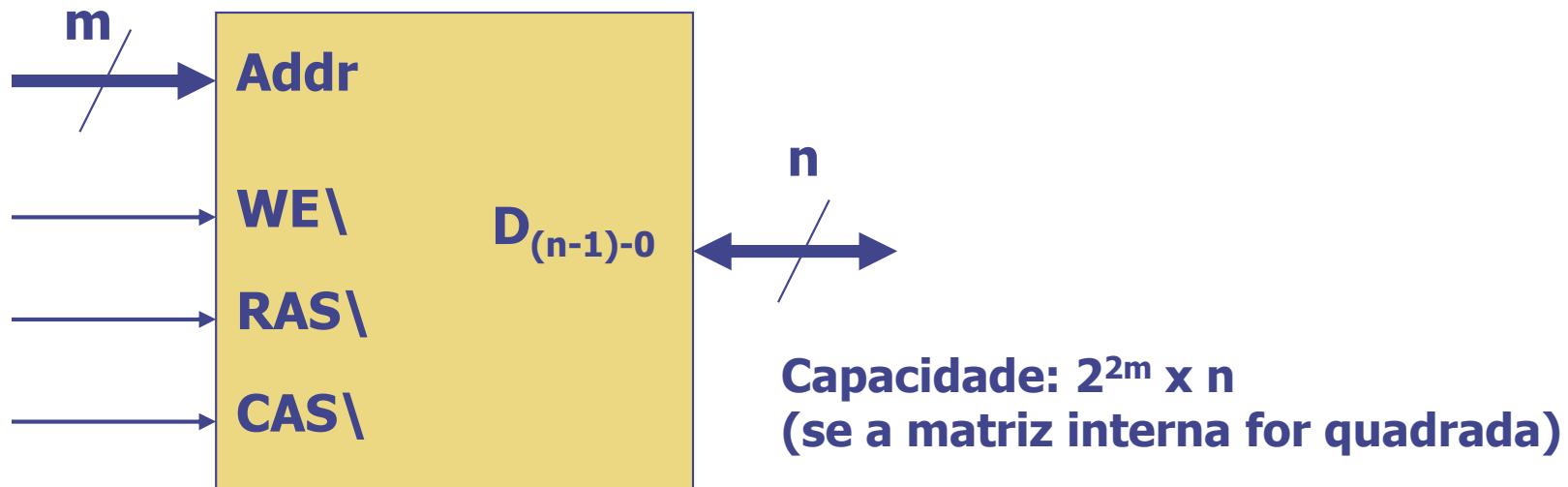
**Address:**      **Row Address**      **Column Address**

- O barramento de endereços é multiplexado: primeiro é enviado o **endereços de linha** e depois é enviado o **endereço de coluna**
- A multiplexagem no tempo é feita com 2 strobes independentes
  - **RAS** – Row Address Strobe
  - **CAS** – Column Address Strobe



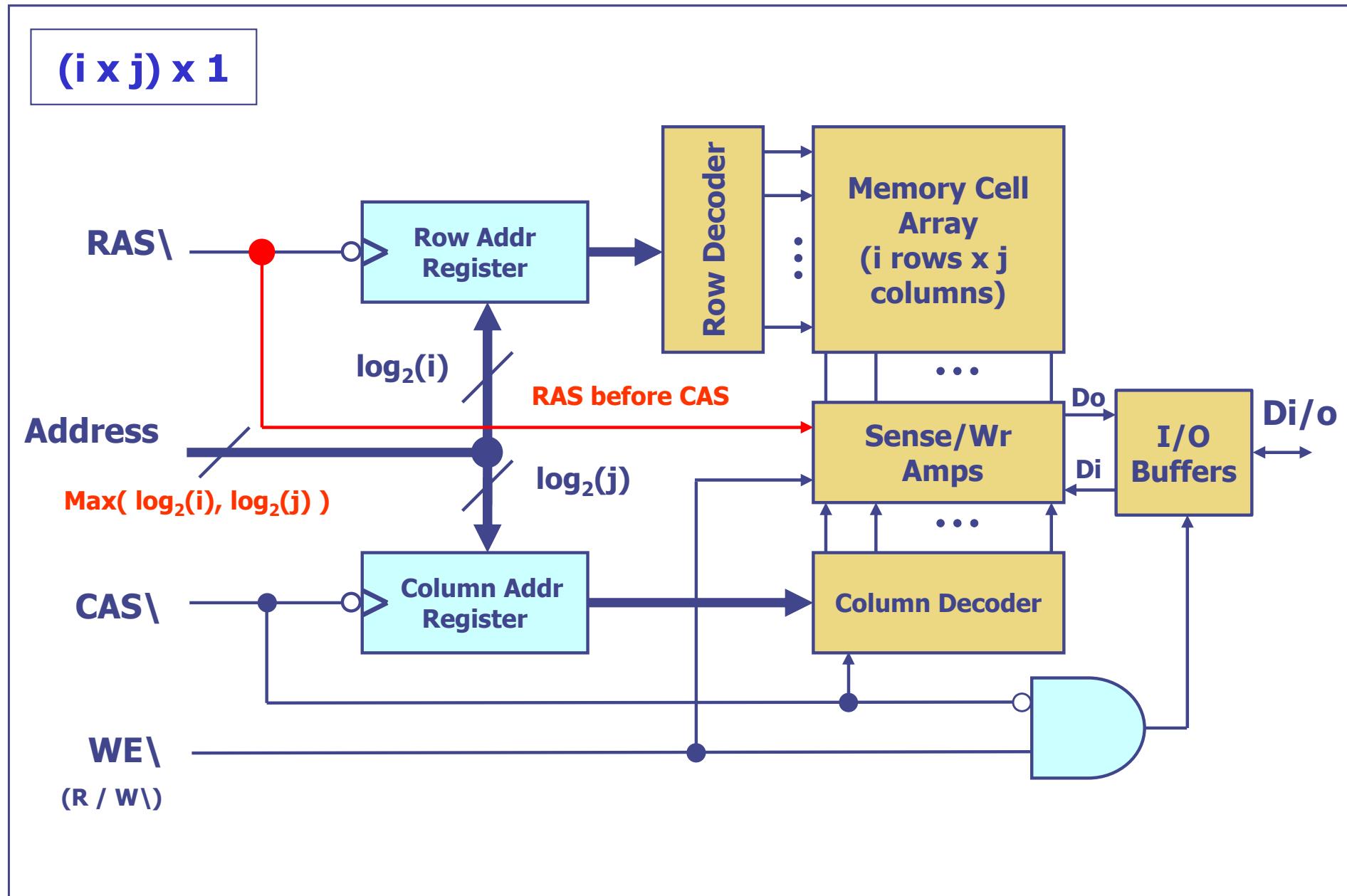
- As transições do RAS e do CAS são usadas para armazenar internamente os endereços de linha e de coluna, respectivamente
- Linha CAS funciona também como "chip-enable"

# DRAM - Diagrama lógico



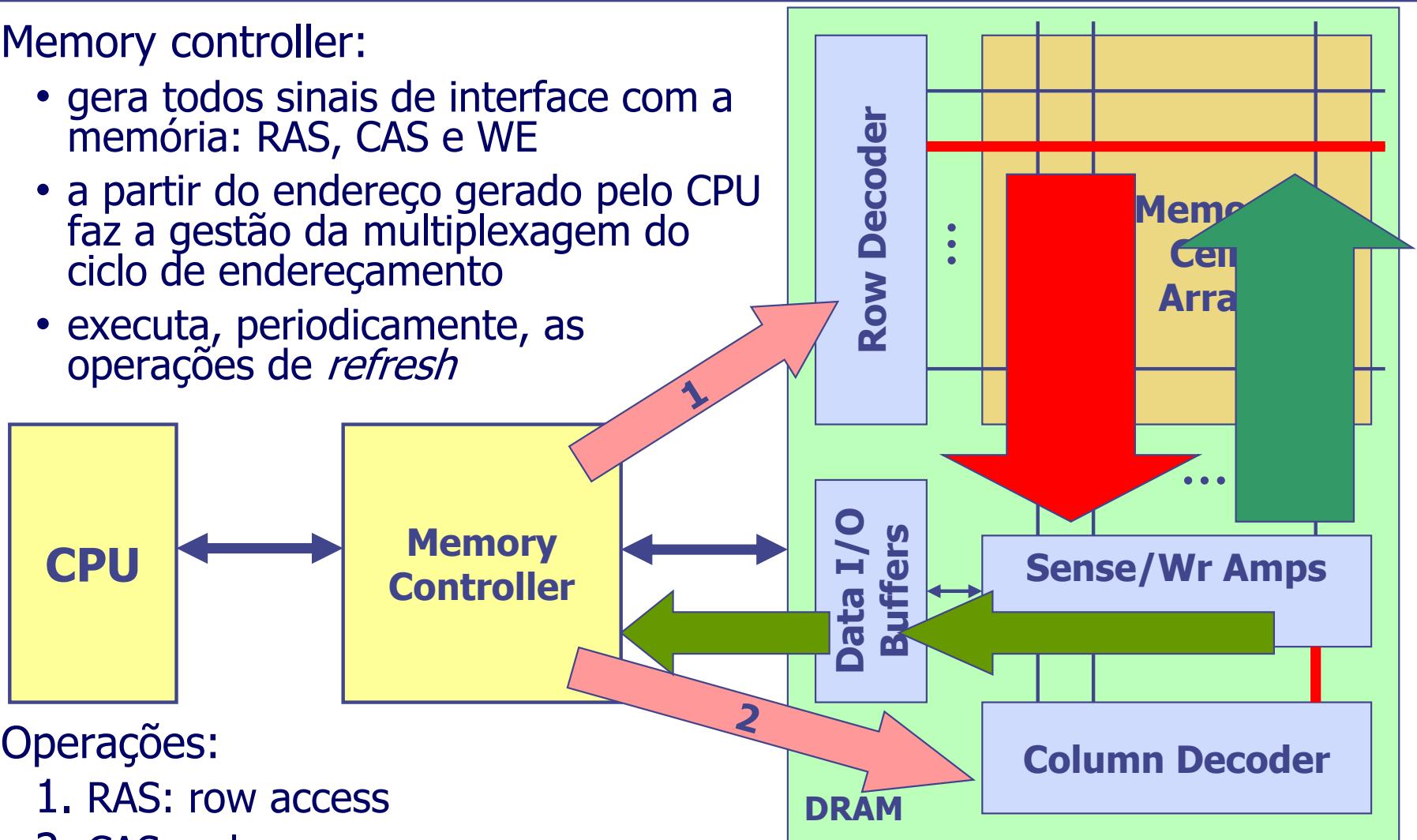
- WE\= 0 → escrita; WE\=1 → leitura ( $\equiv$  R/W\)
- RAS\: valida endereço da linha na transição descendente
- CAS\: valida endereço da coluna na transição descendente

# DRAM – Diagrama de blocos conceptual



# DRAM – Leitura

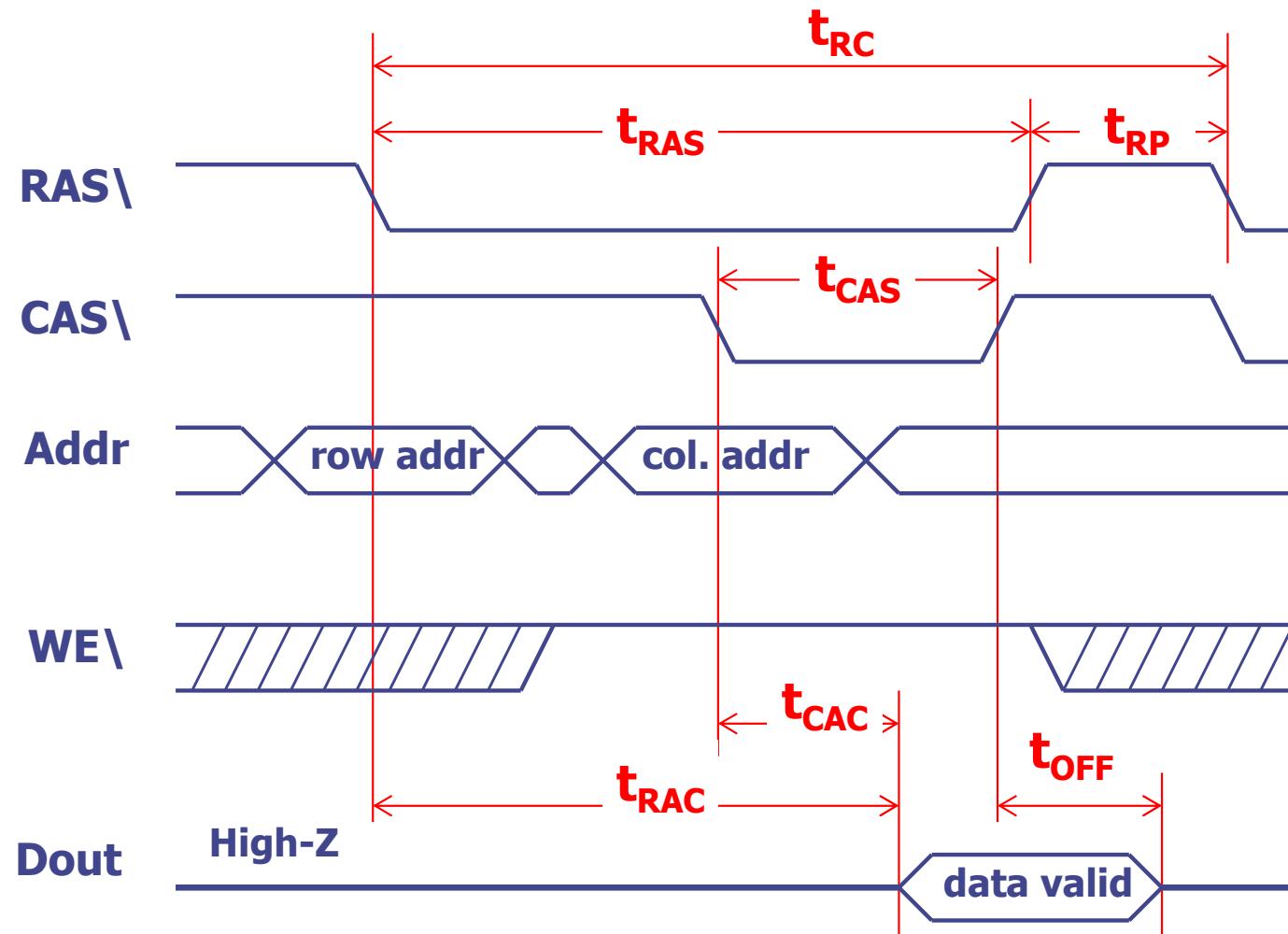
- Memory controller:
  - gera todos sinais de interface com a memória: RAS, CAS e WE
  - a partir do endereço gerado pelo CPU faz a gestão da multiplexagem do ciclo de endereçamento
  - executa, periodicamente, as operações de *refresh*



- Operações:
  1. RAS: row access
  2. CAS: column access
- Buffer de linha (*row buffer*) armazena temporariamente todos os bits de uma linha de células da matriz

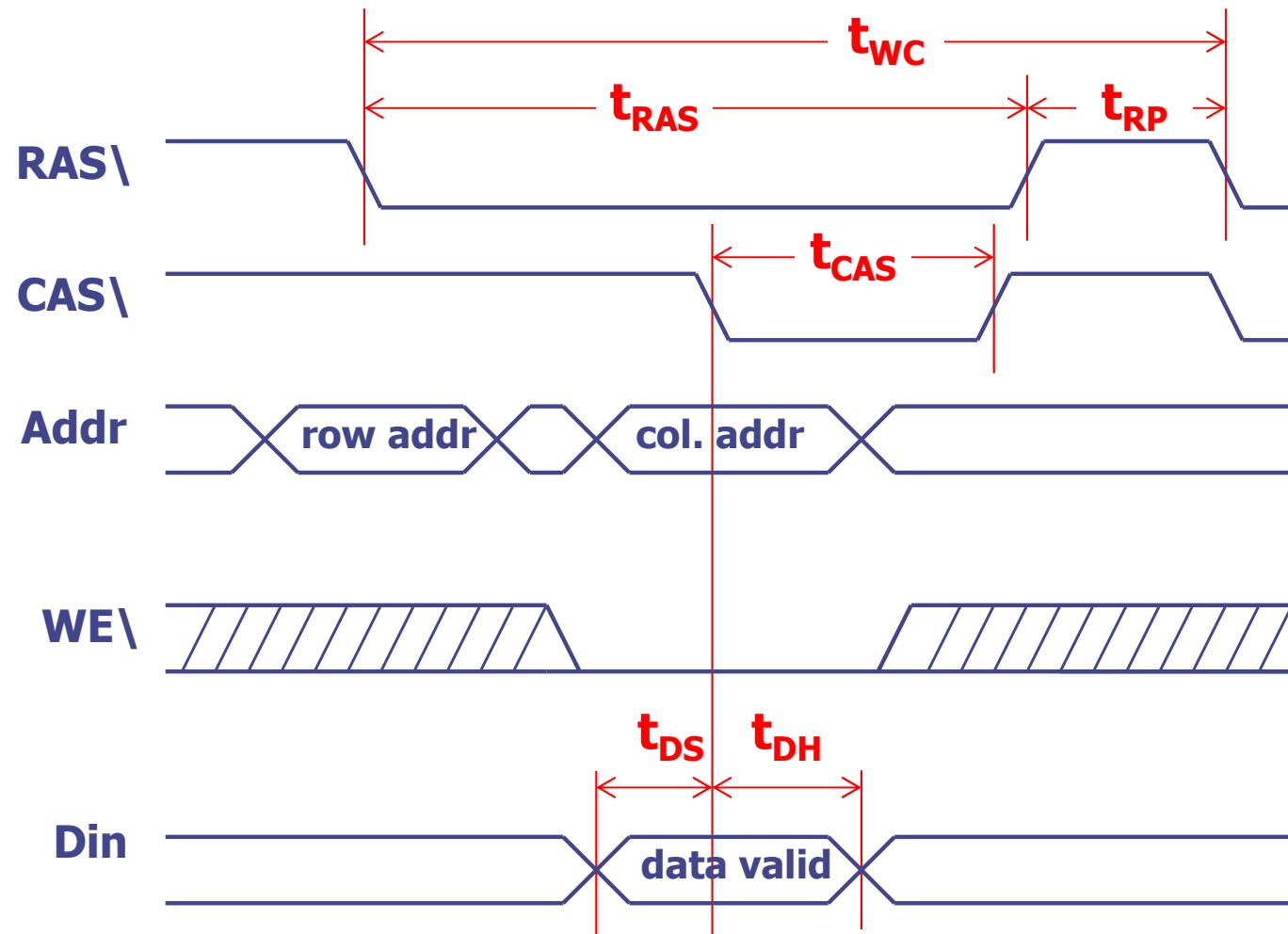
# DRAM – Ciclo de Leitura

- Diagrama temporal típico de um ciclo de leitura de uma memória DRAM



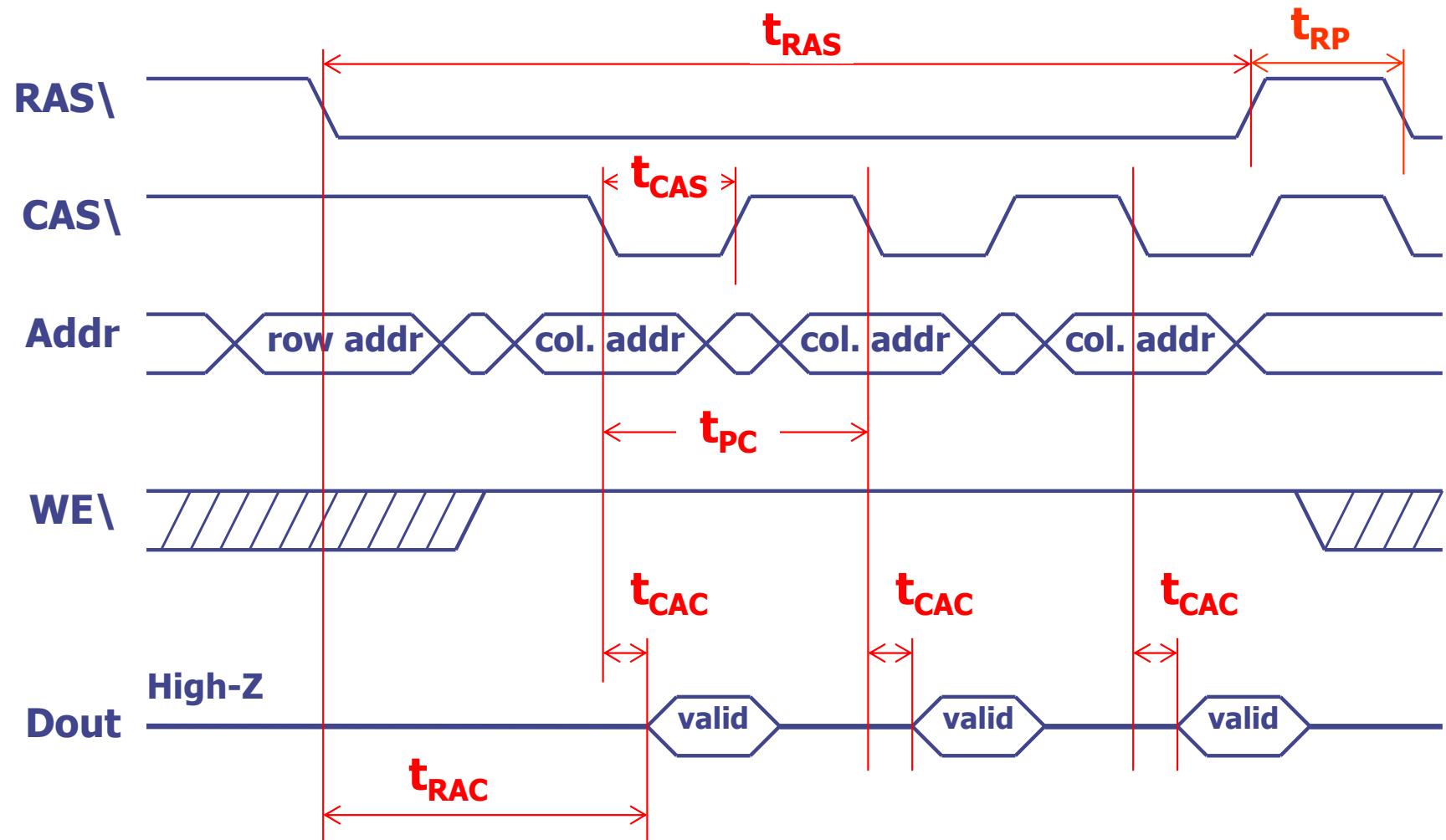
# DRAM – Ciclo de Escrita

- Diagrama temporal típico de um ciclo de escrita (*early write*) de uma memória DRAM



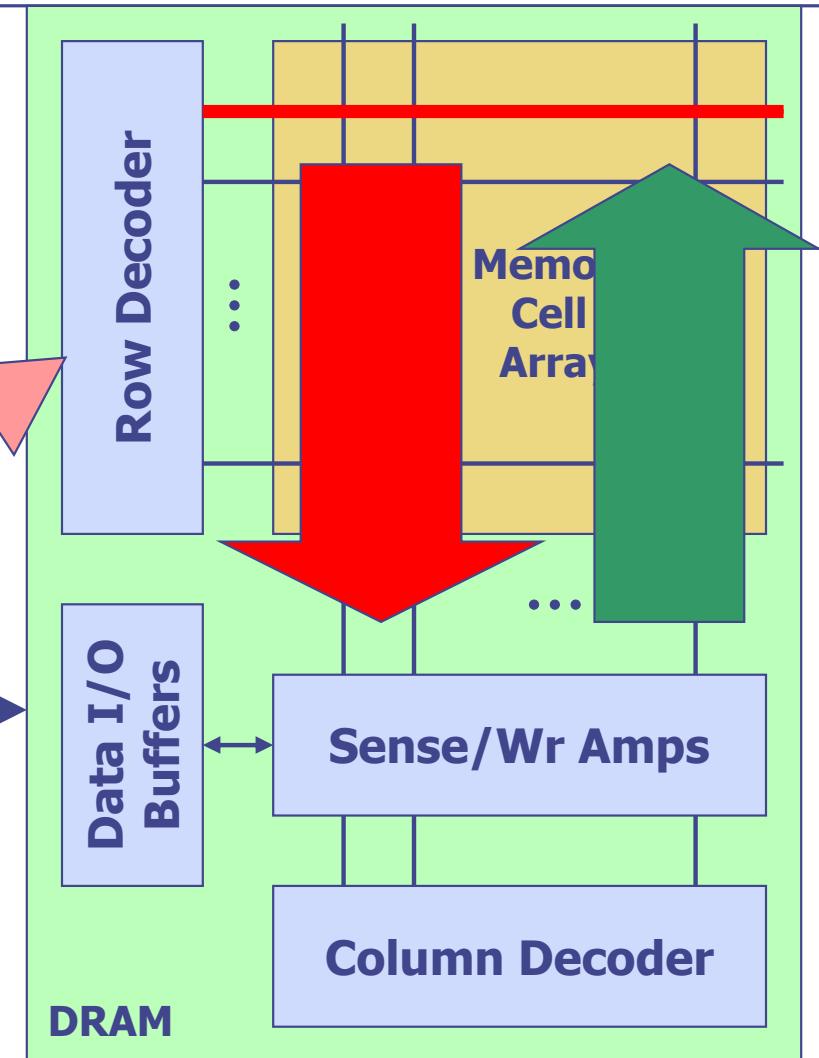
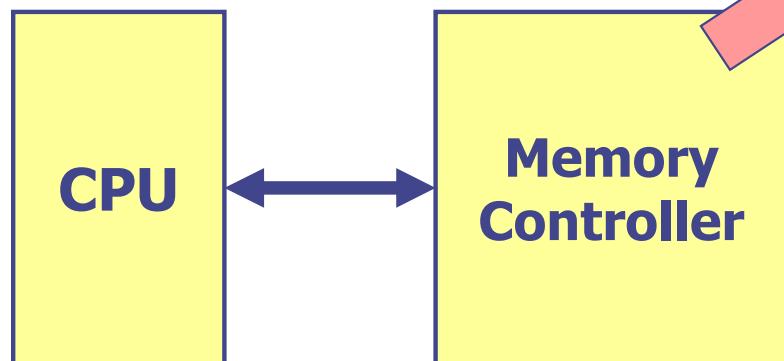
# DRAM – Ciclo de Leitura em *page mode*

- Diagrama temporal típico de um ciclo de leitura de uma memória DRAM, em modo paginado (*page mode*)



# DRAM – Refrescamento

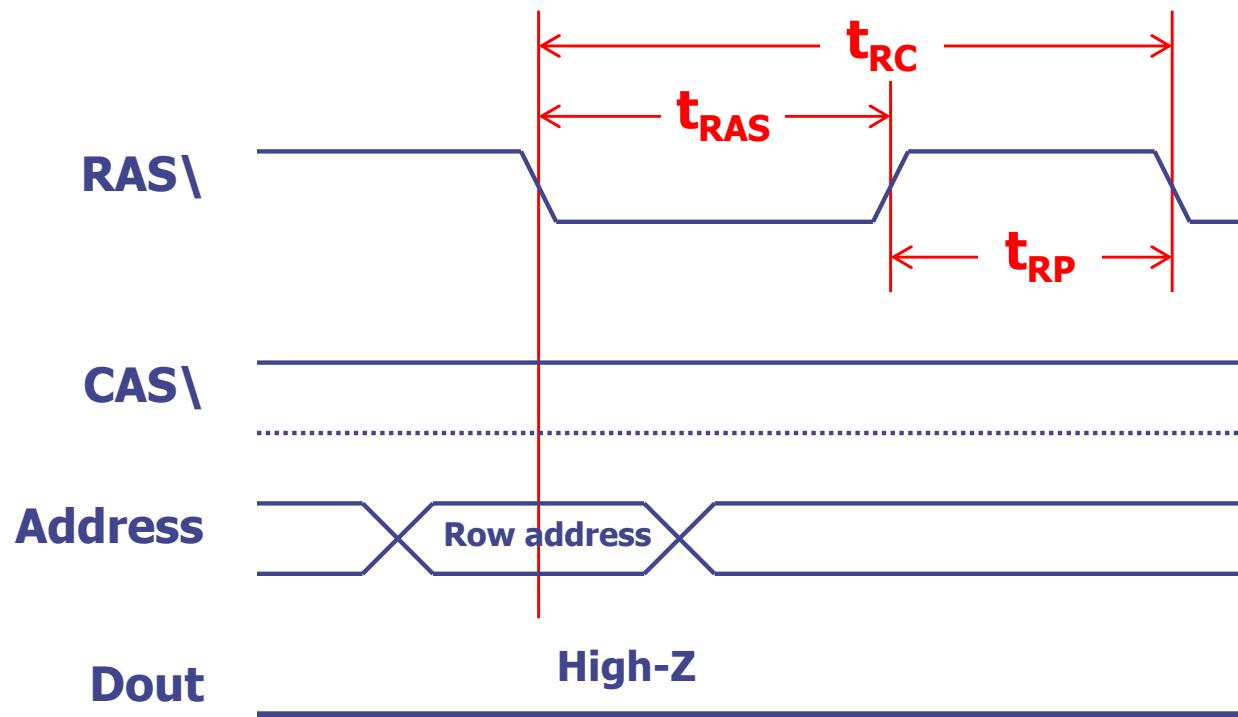
Operação interna de leitura (não há troca de informação com o exterior)



O refrescamento é efetuado, de forma sequencial, para cada uma das linhas da matriz

# DRAM Refresh – RAS Only

- O *refresh* é feito simultaneamente em **todas as células da mesma linha da matriz** (especificada no address bus, no momento da ativação do sinal RAS\)
- O sinal CAS\ mantém-se inativo durante o processo



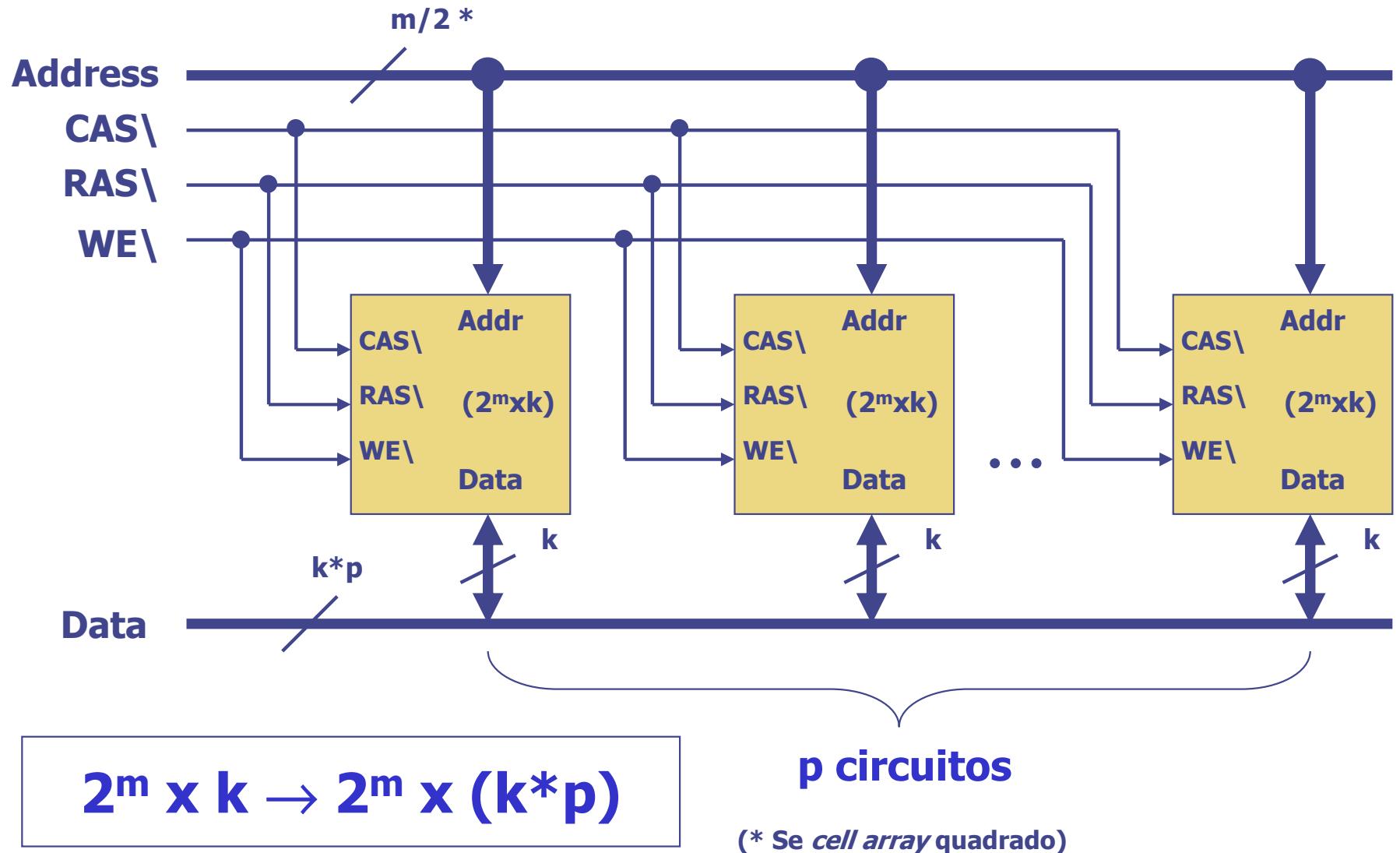
# DRAM - Parâmetros principais

- Valores indicativos (em ns) dos tempos indicados nos diagramas temporais de leitura e escrita de uma memória DRAM com um tempo de acesso de 55 ns:

Parameter	Symbol	Min.	Max.
<b>Read or Write Cycle Time</b>	$t_{RC}$	<b>100</b>	
<b>RAS\ precharge time</b>	$t_{RP}$	<b>45</b>	
<b>Page mode cycle time</b>	$t_{PC}$	<b>35</b>	
<b>RAS\ pulse width</b>	$t_{RAS}$	<b>55</b>	<b>10000</b>
<b>CAS\ pulse width</b>	$t_{CAS}$	<b>28</b>	<b>10000</b>
<b>Data-in setup time</b>	$t_{DS}$	<b>5</b>	
<b>Data-in hold time</b>	$t_{DH}$	<b>14</b>	
<b>Output buffer turn-off delay</b>	$t_{OFF}$		<b>15</b>
<b>Access time from RAS\</b>	$t_{RAC}$		<b>55</b>
<b>Access time from CAS\</b>	$t_{CAC}$		<b>28</b>

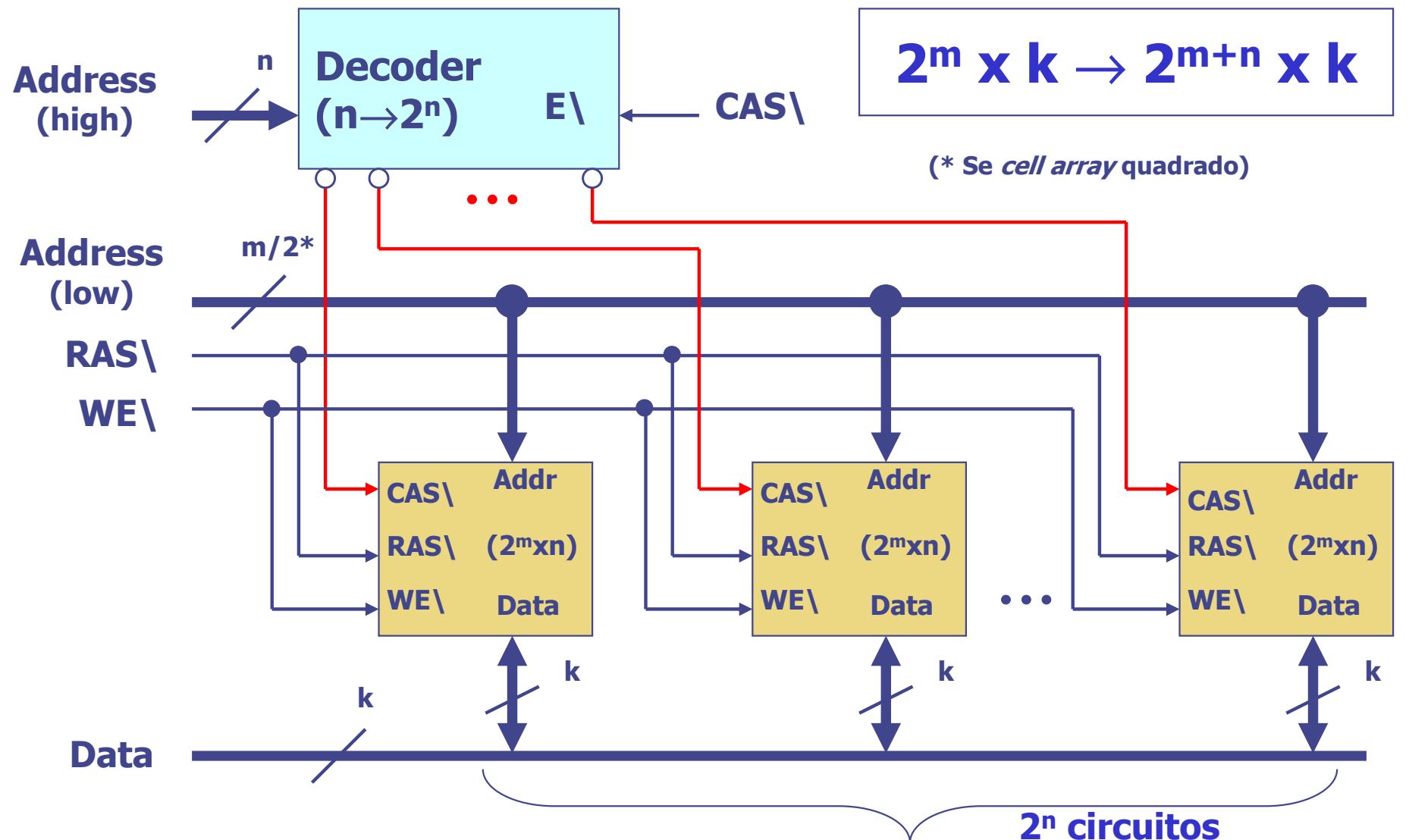
# Módulo de memória DRAM

- Aumento da dimensão da palavra



# Módulo de memória DRAM

- Aumento do número total de posições de memória



# Melhorias de desempenho da DRAM

- **Fast Page Mode**
  - Adiciona sinais de temporização que permitem acessos repetidos ao buffer de linha (sem outro tempo de acesso à linha)
- **Synchronous DRAM (SDRAM)**
  - Adiciona um sinal de relógio à interface DRAM, para facilitar a sincronização de transferências múltiplas
  - Múltiplos bancos, cada um com o seu buffer de linha
- **Double Data Rate (DDR SDRAM)**
  - Transferência de dados tanto no flanco ascendente como no flanco descendente do sinal de relógio (duplica a taxa de transferência de pico)
  - Versão atual: DDR5 (2021->). Exemplo: DDR5-6400, 6400 Milhões de transferências por segundo, relógio de 3.2 GHz
- Estas técnicas melhoram a largura de banda, mas não a latência

Name		Release year	Chip			Bus			Voltage (V)
Gen	Standard		Clock rate (MHz)	Cycle time (ns)	Pre-fetch	Clock rate (MHz)	Transfer rate (MT/s)	Bandwidth (MB/s)	
DDR	DDR-200	1998	100	10	2n	100	200	1600	2.5
	DDR-266		133	7.5		133	266	2133 $\frac{1}{3}$	
	DDR-400		200	5		200	400	3200	
DDR2	DDR2-400	2003	100	10	4n	200	400	3200	1.8
	DDR2-533		133 $\frac{1}{3}$	7.5		266 $\frac{2}{3}$	533 $\frac{1}{3}$	4266 $\frac{2}{3}$	
	DDR2-800		200	5		400	800	6400	
	DDR2-1066		266 $\frac{2}{3}$	3.75		533 $\frac{1}{3}$	1066 $\frac{2}{3}$	8533 $\frac{1}{3}$	
DDR3	DDR3-800	2007	100	10	8n	400	800	6400	1.5/1.35
	DDR3-1600		200	5		800	1600	12800	
	DDR3-1866		233 $\frac{1}{3}$	4.29		933 $\frac{1}{3}$	1866 $\frac{2}{3}$	14933 $\frac{1}{3}$	
DDR4	DDR4-1600	2014	200	5	8n	800	1600	12800	1.2/1.05
	DDR4-2400		300	3 $\frac{1}{3}$		1200	2400	19200	
	DDR4-2666		333 $\frac{1}{3}$	3		1333 $\frac{1}{3}$	2666 $\frac{2}{3}$	21333 $\frac{1}{3}$	
	DDR4-3200		400	2.5		1600	3200	25600	
DDR5	DDR5-3200	2020	200	5	16n	1600	3200	25600	1.1
	DDR5-4000		250	4		2000	4000	32000	
	DDR5-5600		350	2.86		2800	5600	44800	
	DDR5-6400		400	2.5		3200	6400	51200	
	DDR5-7200		450	2.22		3600	7200	57600	

## Aulas 20 e 21

- Organização da memória de um sistema computacional
- Hierarquia do sistema de memória
- Localidade temporal e espacial
- A memória cache
  - Princípio de funcionamento
  - Cache com mapeamento associativo
  - Cache com mapeamento direto
  - Cache com mapeamento parcialmente associativo

José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

# Introdução

- Pretensão do utilizador:
  - Uma memória rápida e com grande capacidade de armazenamento
  - Que custe o preço de uma memória lenta... ☺
- Solução perfeita para este dilema não existe

Tecnologia	Tempo Acesso	\$ / GB
SRAM	0,5 – 2,5 ns	\$500 - \$1000
DRAM	35 - 70 ns	\$10 - \$20

(Dados de 2012)

- A organização da memória de um sistema computacional resulta de um compromisso entre:
  - Velocidade, Capacidade, Custo, Consumo energético
  - Menor tempo de acesso: maior custo por bit
  - Maior capacidade: maior tempo de acesso

# Introdução

- Memória DRAM (Dynamic RAM)

Ano	Capacidade (max. por chip)	Access Time	\$ / Mb
1980	64 kbit	250 ns	\$1500
1983	256 kbit	185 ns	\$500
1985	1 Mbit	135 ns	\$200
1989	4 Mbit	110 ns	\$50
1992	16 Mbit	90 ns	\$15
1996	64 Mbit	60 ns	\$10
1998	128 Mbit	60 ns	\$4
2000	256 Mbit	55 ns	\$1
2004	512 Mbit	50 ns	\$0.25
2007	1024 Mbit	45 ns	\$0.05
2010	2 Gbit	40 ns	\$0.03
2012	4 Gbit	35 ns	\$0.001

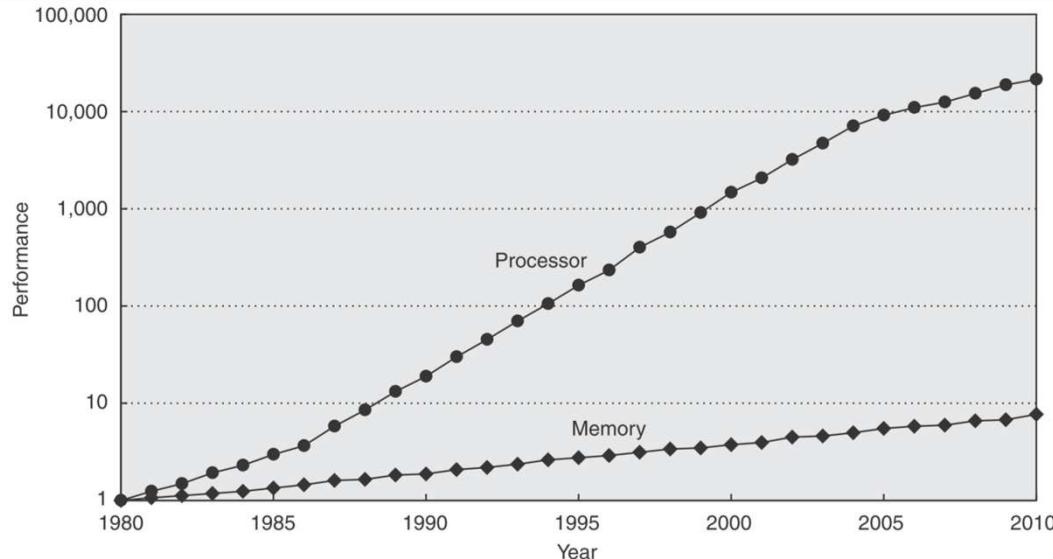
# Introdução

- O processador deve ser alimentado de instruções e dados a uma taxa que não comprometa o desempenho do sistema

- A diferença entre o desempenho do processador e da memória (DRAM) tem vindo a aumentar

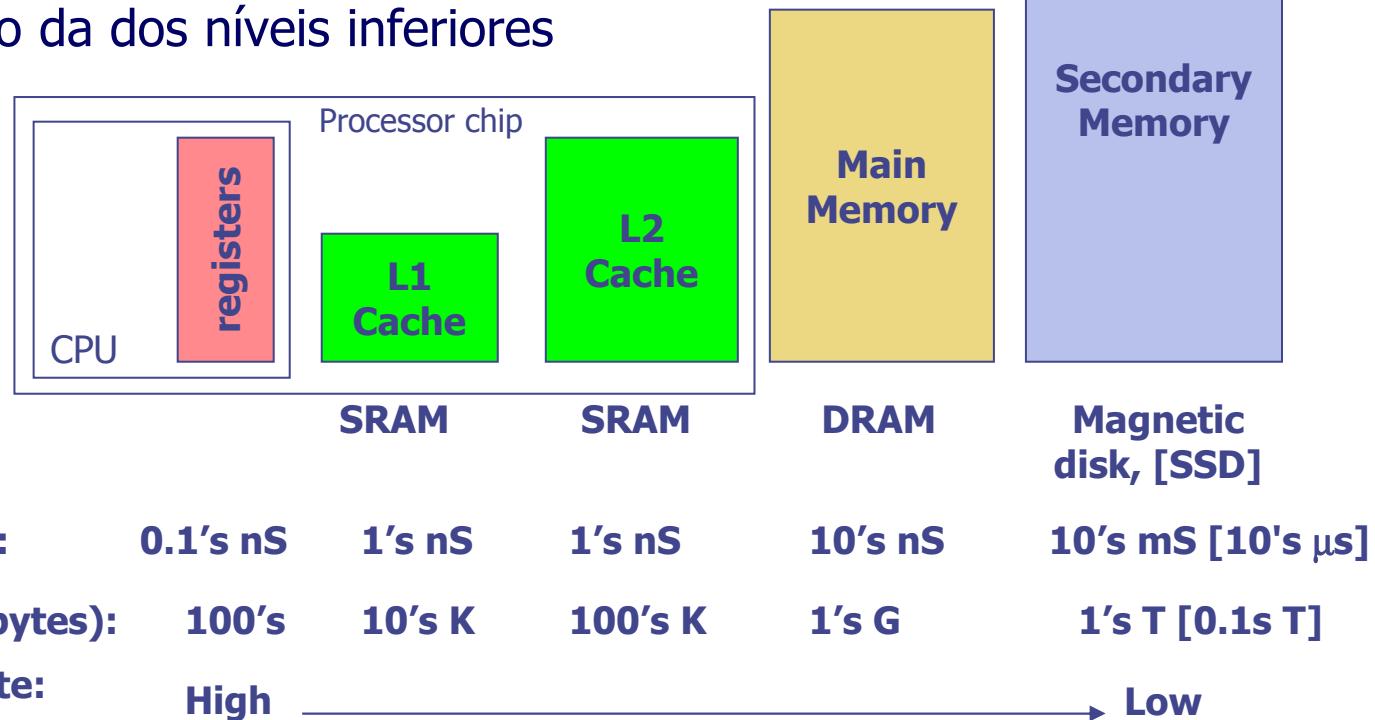
- Solução:

- Guardar a informação mais vezes utilizada pelo CPU numa memória rápida (*static RAM*) de pequena capacidade
- Aceder raramente à memória principal (mais lenta) para obter a informação em falta (apenas quando necessário)
- Transferir blocos de informação da memória principal para a memória rápida
- Conceito: **cache**



# Hierarquia de memória

- Memória organizada em níveis
- A informação nos níveis superiores é um subconjunto da dos níveis inferiores



- A informação circula apenas entre níveis adjacentes da hierarquia
- **Bloco** – quantidade de informação que circula entre níveis adjacentes (n bytes)

# Hierarquia de memória

- Solução 1 – expor a hierarquia
  - Alternativas de armazenamento: registos internos do CPU, memória rápida, memória principal, disco
  - Cabe ao programador utilizar racionalmente estas alternativas de armazenamento
  - Exemplo de processador que usa esta técnica: Cell microprocessor (Cell Broadband Engine Architecture; usado por ex. na PlayStation 3 e algumas televisões)
- Solução 2 – esconder a hierarquia
  - Modelo de programação:
    - Tipo de memória único
    - Espaço de endereçamento único
  - A máquina gere automaticamente o acesso ao sistema de memória
  - Solução usada na maioria dos processadores contemporâneos

# Hierarquia de memória

- A hierarquia de memória combina uma memória adaptada à velocidade do processador (de pequena dimensão) com uma (ou mais) menos rápida, mas de maior dimensão
- A memória rápida armazena um sub-conjunto da informação residente na memória principal.
- Uma vez que a memória com que o processador interage diretamente é de pequena dimensão, a eficiência da hierarquia resulta do facto de se copiar informação para a memória rápida poucas vezes e de se aceder a essa informação muitas vezes (antes de surgir a necessidade de a substituir)
- Para se tirar partido deste esquema, a probabilidade de a informação (que o processador necessita) estar nos níveis mais elevados da hierarquia tem que ser elevada
- O que torna essa probabilidade elevada ?

# Localidade

- **Princípio da localidade:** os programas não acedem à memória (dados e instruções) de forma aleatória mas usam tipicamente endereços que se situam na vizinhança uns dos outros
- Ou seja, num dado intervalo de tempo um programa acede a uma zona reduzida do espaço de endereçamento
- O princípio da localidade manifesta-se de duas formas:
  - **Localidade no espaço (spatial locality):** Se existe um acesso a um endereço de memória então é provável que os endereços contíguos sejam também acedidos
  - **Localidade no tempo (temporal locality):** Se existe um acesso a um endereço de memória então é provável que esse mesmo endereço seja acedido novamente no futuro próximo

# Localidade

- **Localidade espacial:**

"a informação que o processador necessita de seguida, tem uma elevada probabilidade de estar próxima da que consome agora"

- Exemplos: instruções de um programa; processamento de um *array*
- Quanto maior for o bloco (zona contígua de memória) de informação existente na memória rápida, melhor

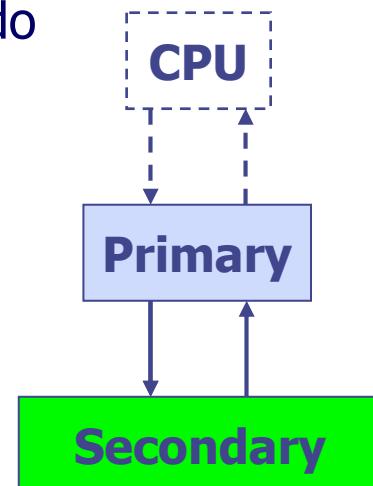
- **Localidade temporal:**

"a informação que o processador consome agora tem uma elevada probabilidade de ser novamente necessária num curto espaço de tempo"

- Exemplos: variável de controlo de um ciclo, instruções de um ciclo
- Quantos mais blocos de informação estiverem na memória rápida, melhor

# Hierarquia de memória com 2 níveis

- Nível primário (superior) rápido e de pequena dimensão
- Nível secundário (inferior) mais lento mas de maior dimensão
  - O nível primário contém os blocos de memória mais recentemente utilizados pelo CPU
  - Os pedidos de informação são sempre dirigidos ao nível primário, sendo o nível secundário envolvido apenas quando a informação pretendida não está nesse nível
    - Se os dados pretendidos se encontram num bloco do nível primário então existe um "hit"
    - Caso contrário ocorre um "miss"
  - Na ocorrência de um "miss" acede-se ao nível secundário e transfere-se o bloco que contém a informação pretendida



# Hierarquia de memória com 2 níveis

- A taxa de sucesso (**hit ratio**) é dada por:

$$hit_{ratio} = \frac{nr_{hits}}{nr_{accesses}}$$

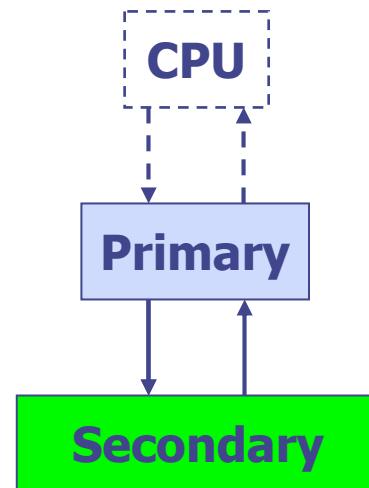
- A taxa de insucesso (**miss ratio**) é dada por:

$$miss_{ratio} = 1 - hit_{ratio}$$

- O tempo de acesso no caso de um "hit" designa-se **hit time**
- O tempo de substituir um bloco do nível superior e enviar os dados para o processador é designado por **penalty time (miss penalty)**
- De um modo simplificado, o "penalty time" é dado por:

$$penalty_{time} = hit_{time} + t_{mem}$$

em que " $t_{mem}$ " é o tempo de acesso ao nível secundário



# Hierarquia de memória com 2 níveis

- O tempo médio de acesso à informação é então:

$$T_{access} = hit_{ratio} * hit_{time} + (1 - hit_{ratio}) * penalty_{time}$$

$$T_{access} = hit_{ratio} * hit_{time} + (1 - hit_{ratio}) * (hit_{time} + t_{mem})$$

- **Exercício:** Assumindo um *hit\_ratio* de 95%, um tempo de acesso ao nível superior de 5ns e um tempo de acesso ao nível inferior de 50ns, calcular o tempo médio de acesso à memória
  - $T_a = 0,95 * 5 + 0,05 * (50 + 5) = 7,5\text{ns}$   
Ou seja, aproximadamente 6,7 vezes mais rápido que o acesso direto ao nível secundário
- Quando o acesso a uma palavra no nível superior falha, acede-se ao nível seguinte não apenas a essa palavra, mas também às que estão em endereços adjacentes (vizinhas) - **bloco**

# Memória cache

- **cache**: nível de memória que se encontra entre o CPU e a memória principal (primeiras utilizações no início da década de 60)

- Exemplo - **cache read**:

Cache recebe endereço (MemAddr) do CPU

O bloco que contém MemAddr está na cache?

**SIM (hit)**:

Lê a cache e envia para o CPU o conteúdo de MemAddr

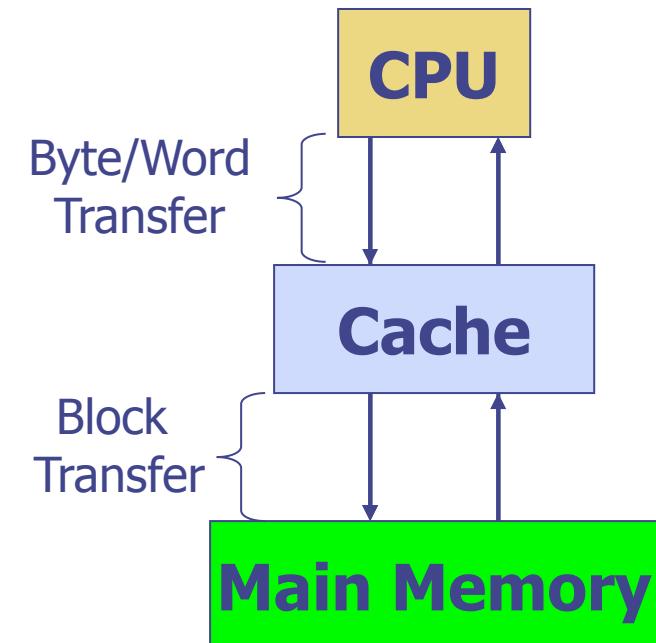
**NÃO (miss)**:

Encontra espaço na cache para um novo bloco

Acede à memória principal e lê o bloco que contém o endereço MemAddr

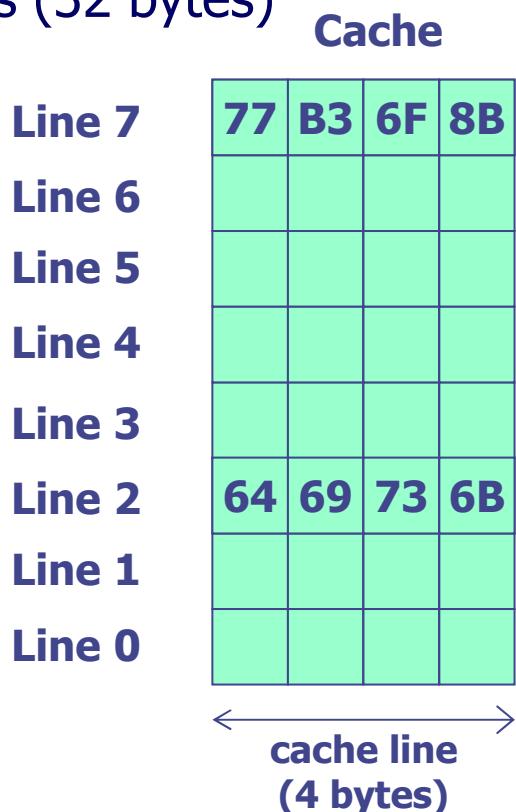
Conteúdo de Memaddr é enviado para o CPU

- É comum os computadores recentes incluírem 2 ou mais níveis de cache; a cache é normalmente integrada no mesmo circuito integrado do processador



# Memória cache

- Exemplo: memória de 64K (16 bits de endereço), cache de 8 linhas de 4 bytes (32 bytes)



- Quais os endereços da memória principal representados nesta cache?

Memory Address (16 bits)

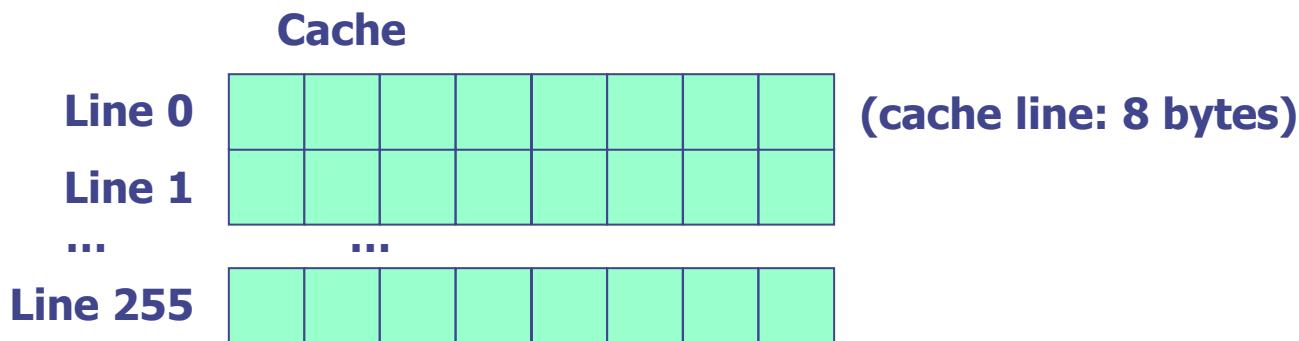
8B	FFFF	Block 3FFF
6F	FFFE	
B3	FFFD	
77	FFFC	
...		
6B	000B	Block 2
73	000A	
69	0009	
64	0008	
20	0007	Block 1
74	0006	
72	0005	
65	0004	
73	0003	Block 0
6E	0002	
49	0001	
0A	0000	

# Memória cache

- As operações da cache são transparentes para o processador
- O processador gera um endereço e pede que seja feita uma operação de leitura ou escrita e esse pedido é satisfeito pelo sistema de memória:
  - é desconhecido para o processador se é a cache ou a memória principal a satisfazer o pedido
- Na organização da cache e na implementação da respetiva unidade de controlo é necessário tomar em consideração um conjunto de aspetos, nomeadamente:
  - Como saber se um determinado endereço está na cache?
  - Se está na cache, onde é que está?
  - Quando ocorre um miss, onde colocar um novo bloco na cache?
  - Quando ocorre um miss, qual o bloco a retirar da cache?
  - Como tratar o problema das operações de escrita de modo a manter a coerência da informação nos vários níveis?
- Implementação tem de ser eficiente! Realizável em hardware

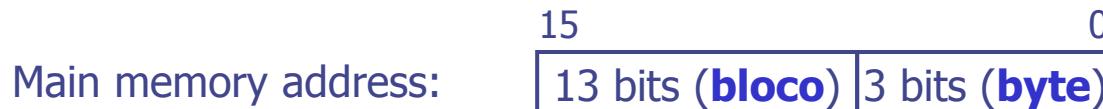
# Organização dos sistemas de cache

- Há basicamente 3 formas de organizar os sistemas de cache:
  - Cache **totalmente associativa** ("fully associative")
  - Cache **com mapeamento direto** ("direct mapped")
  - Cache **parcialmente associativa** ("set associative")
- A metodologia de organização de cada uma delas é apresentada a seguir, partindo do seguinte conjunto de especificações-base:
  - Espaço de endereçamento de 16 bits (memória principal organizada em bytes, com 64 kBytes)
  - Memória cache de 2 kBytes, em que cada linha tem 8 bytes ( $2^3$ ) (ou seja, cada bloco tem uma dimensão de 8 bytes)



# Organização dos sistemas de cache

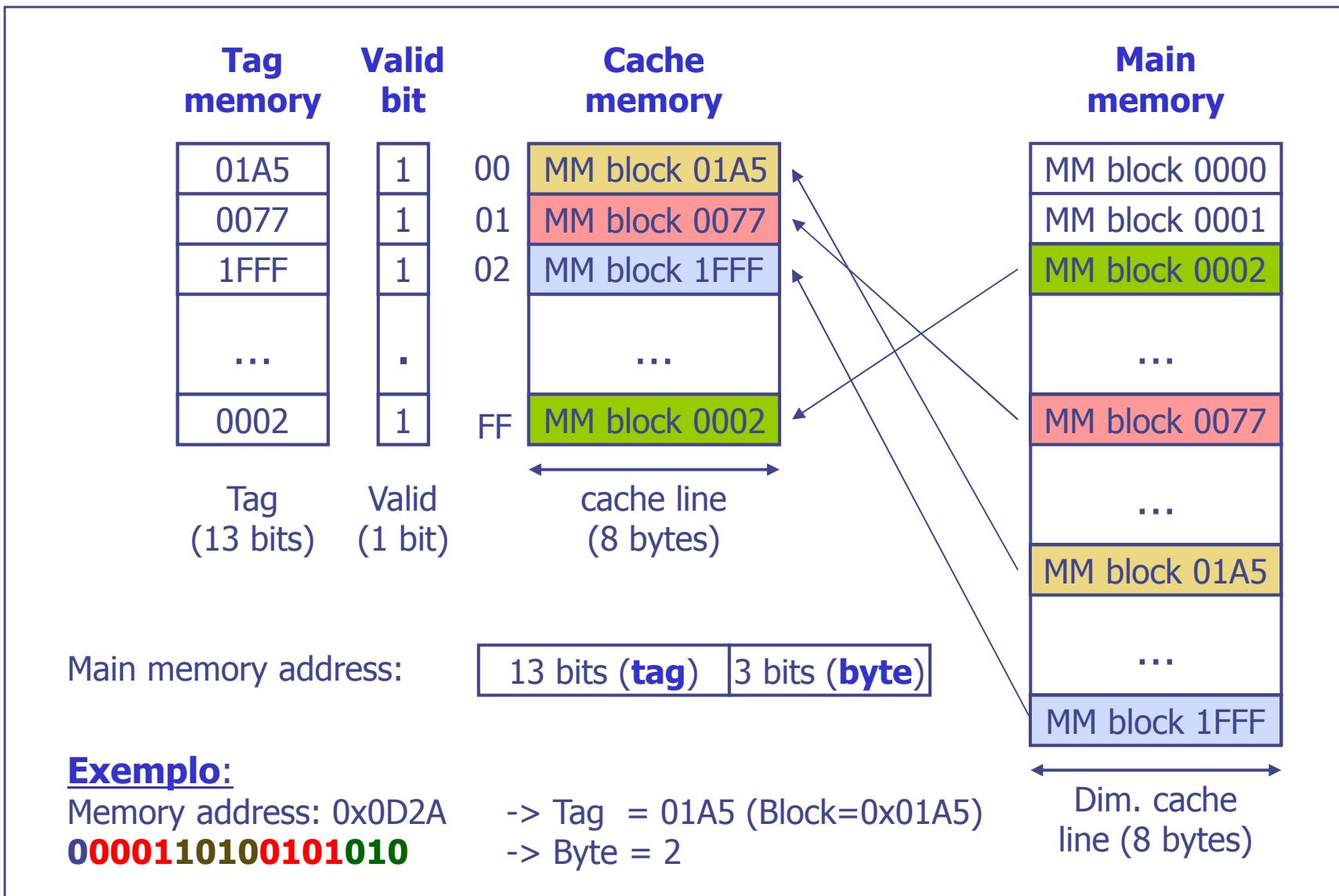
- Especificação-base para os exemplos de organização que se seguem:
  - Espaço de endereçamento de 16 bits (0x0000 a 0xFFFF)
  - Memória cache de 2 kBytes, em que cada linha tem 8 bytes ( $2^3$ ), significa que a cache tem 256 linhas ( $2^{11} / 2^3 = 256$ )
- Com estes valores, a memória principal pode ser vista como sendo constituída por um conjunto de  $2^{13}$  blocos contíguos, de 8 bytes cada ( $2^{16}/2^3 = 2^{13}$ ): 8k blocos (numerados de 0000 a 0x1FFF)
- Assim, no endereço de 16 bits, os
  - 3 bits menos significativos identificam o **byte dentro do bloco**
  - 13 bits mais significativos identificam o **bloco**



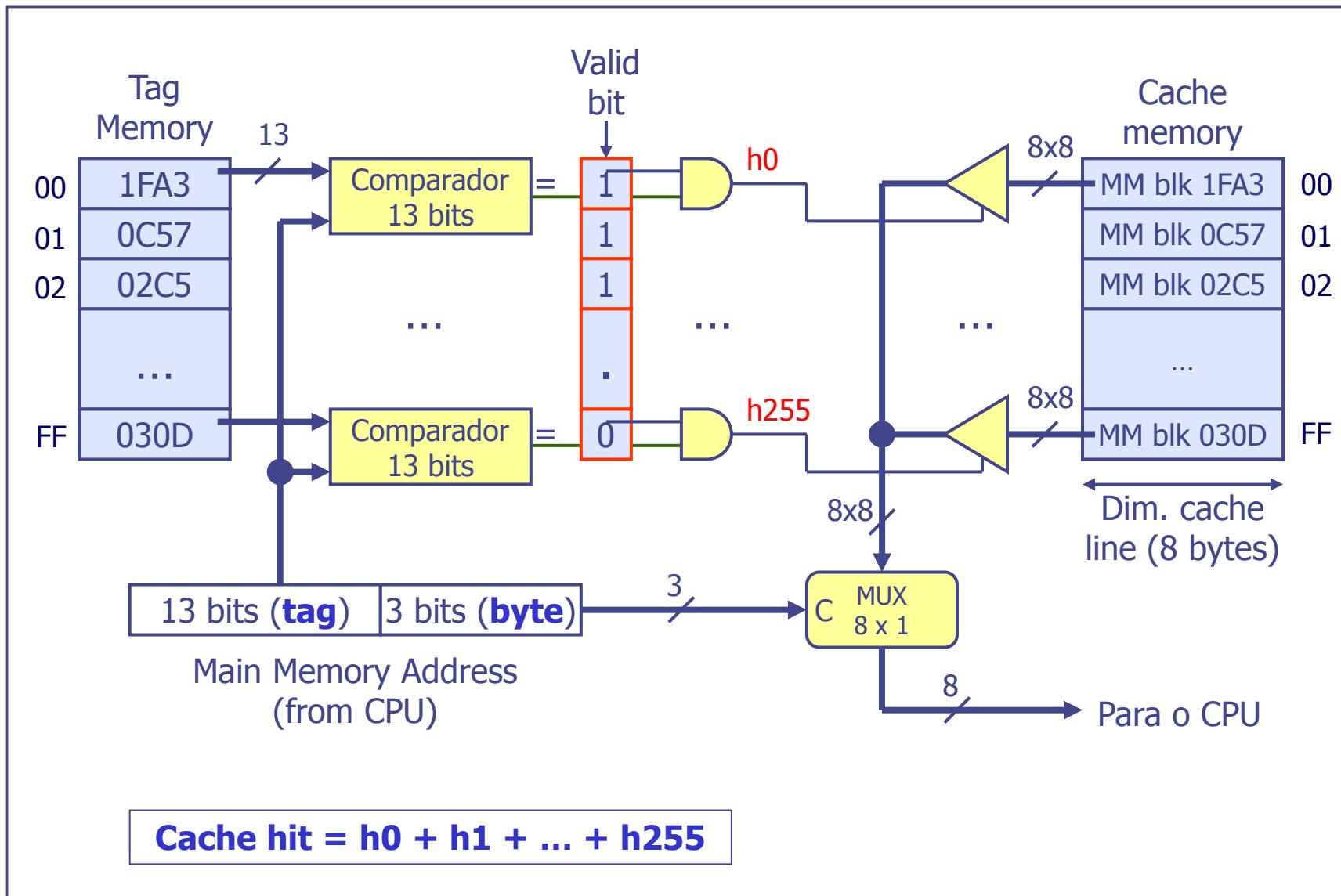
- Exemplo. Address = 0x001A : **00000000000011010**

Bloco 3, de 0x0018 a 0x001F, byte 2 dentro do bloco

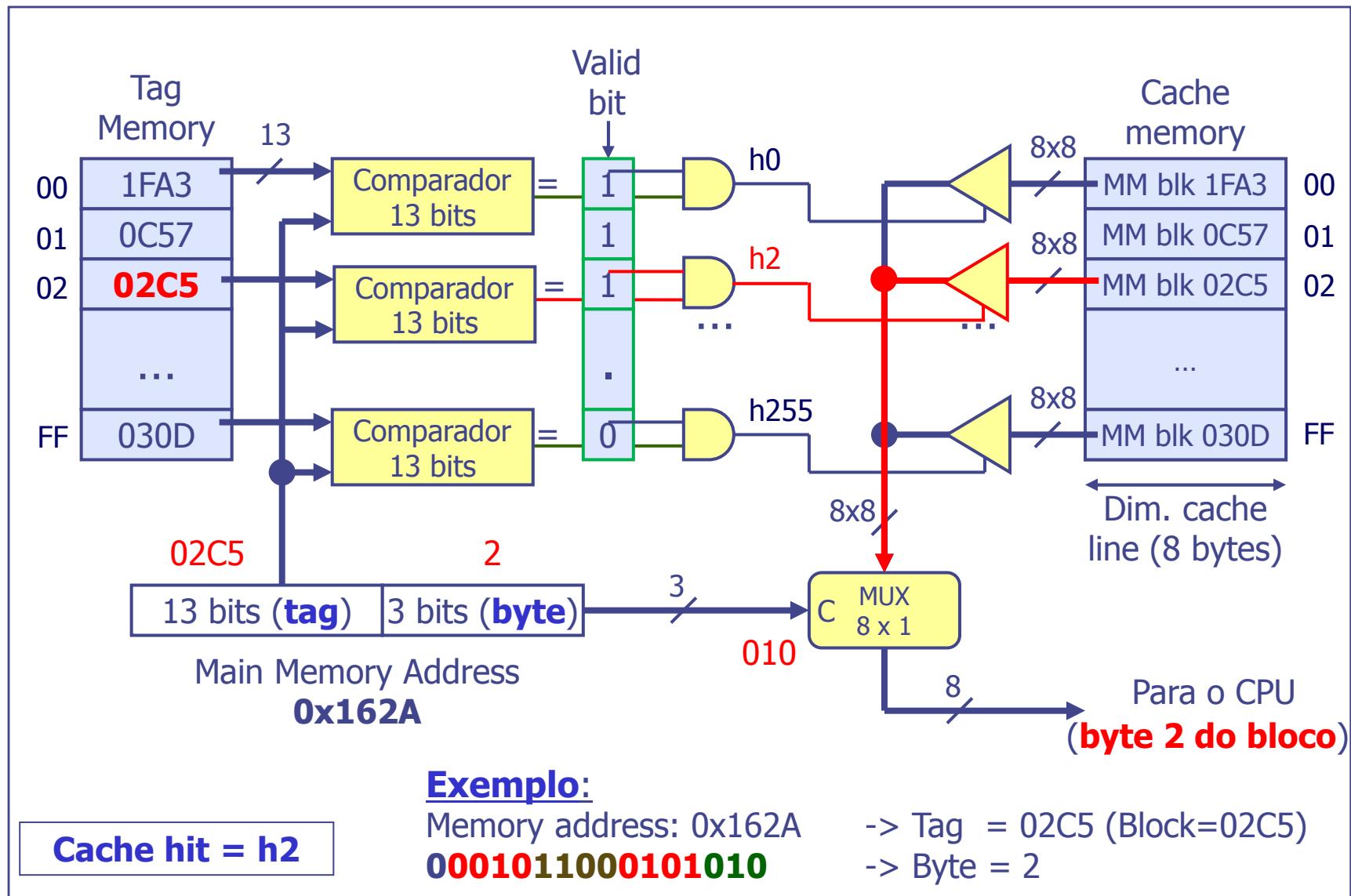
# Cache com mapeamento associativo



# Cache com mapeamento associativo



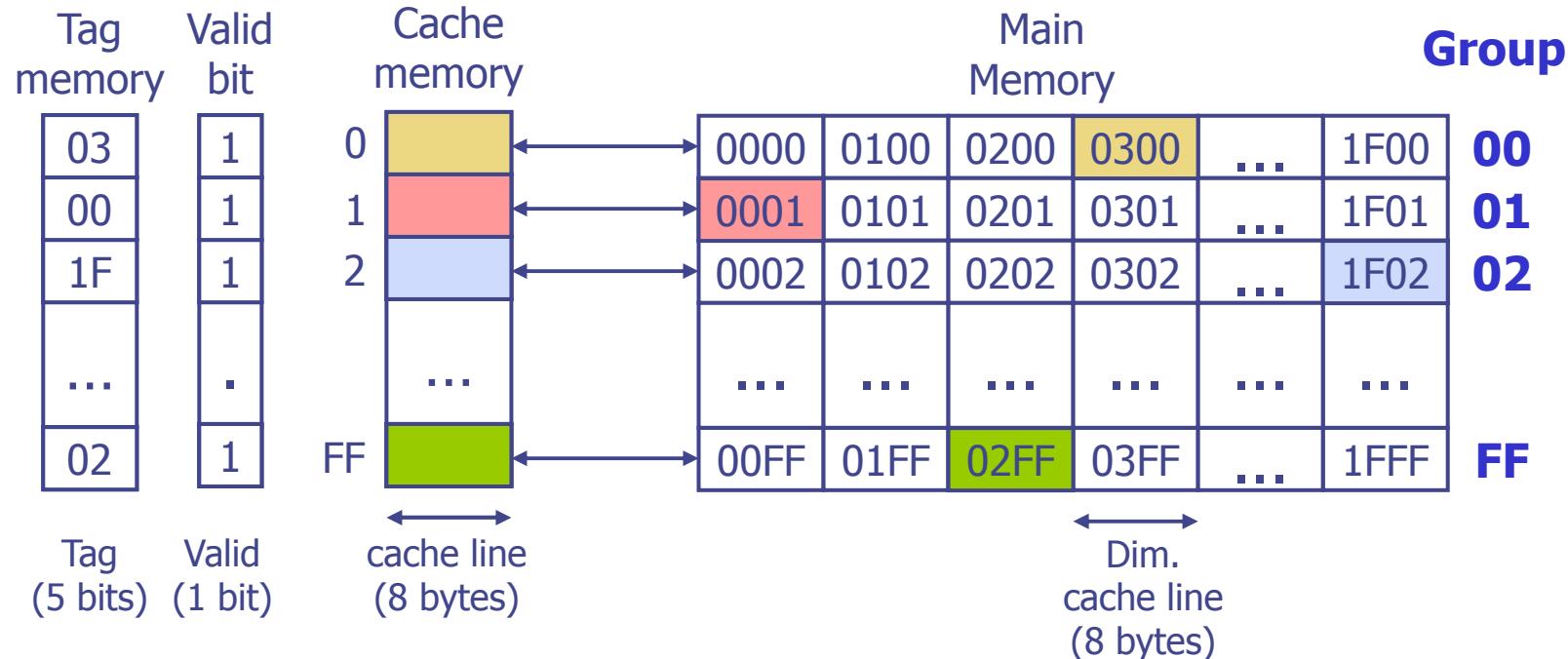
# Cache com mapeamento associativo (exemplo)



# Cache com mapeamento associativo

- Vantagens:
  - Qualquer bloco da memória principal pode ser colocado em qualquer posição da cache
- Inconvenientes:
  - A "tag" tem que ter todos os bits do número do bloco, o que numa implementação realista origina comparadores com uma dimensão elevada
  - Todas as entradas da memória "tag" têm de ser analisadas, de forma a verificar se um endereço se encontra na cache
  - Muitos comparadores, custo elevado

# Cache com mapeamento direto



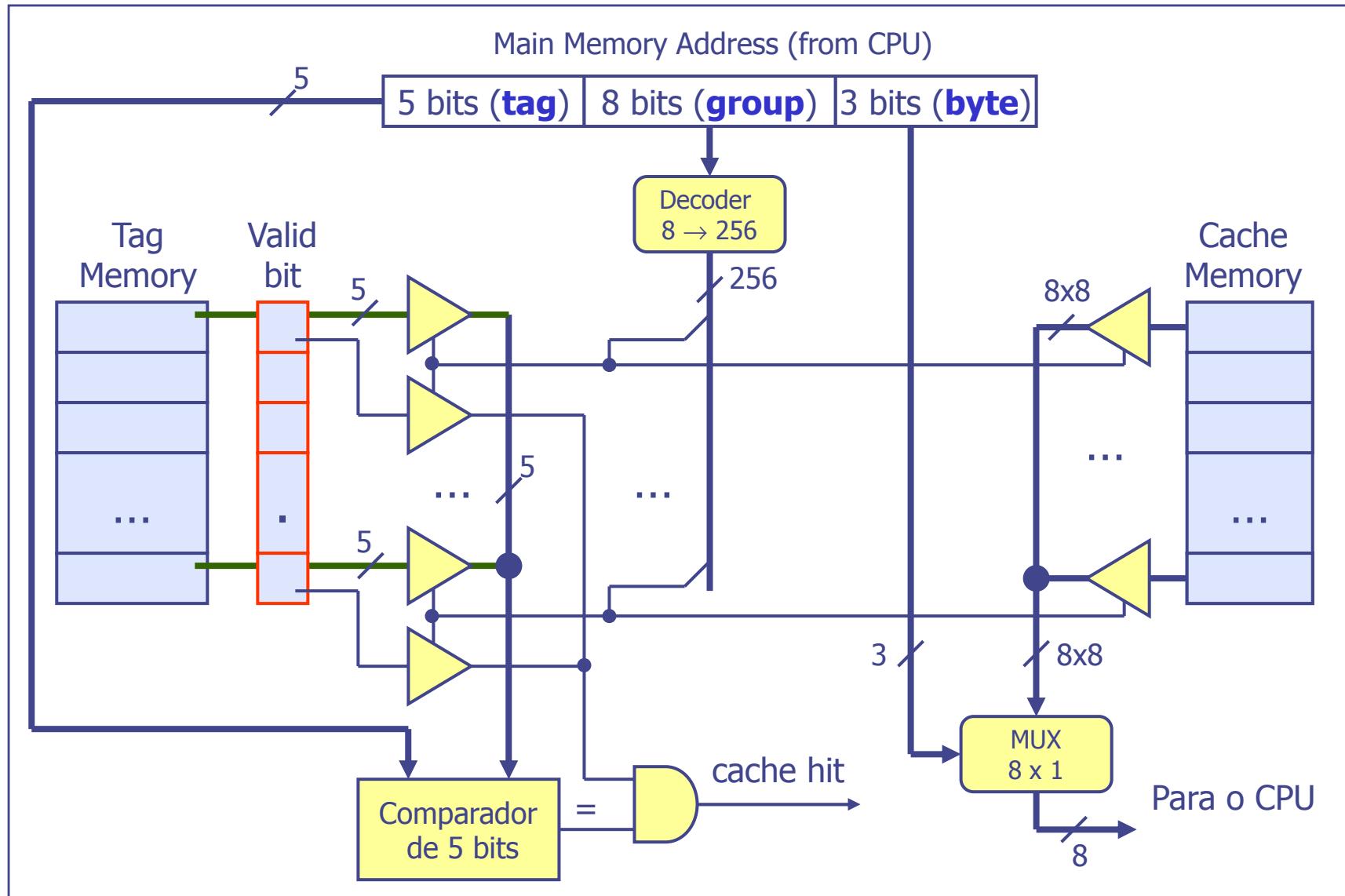
**Main memory address:** 5 bits (**tag**) | 8 bits (**group**) | 3 bits (**byte**)

## Exemplo:

Memory address = 0xF813  
**1111100000010011**

-> Tag = 0x1F (Block = 0x1F02)  
-> Group = 0x02  
-> Byte = 3

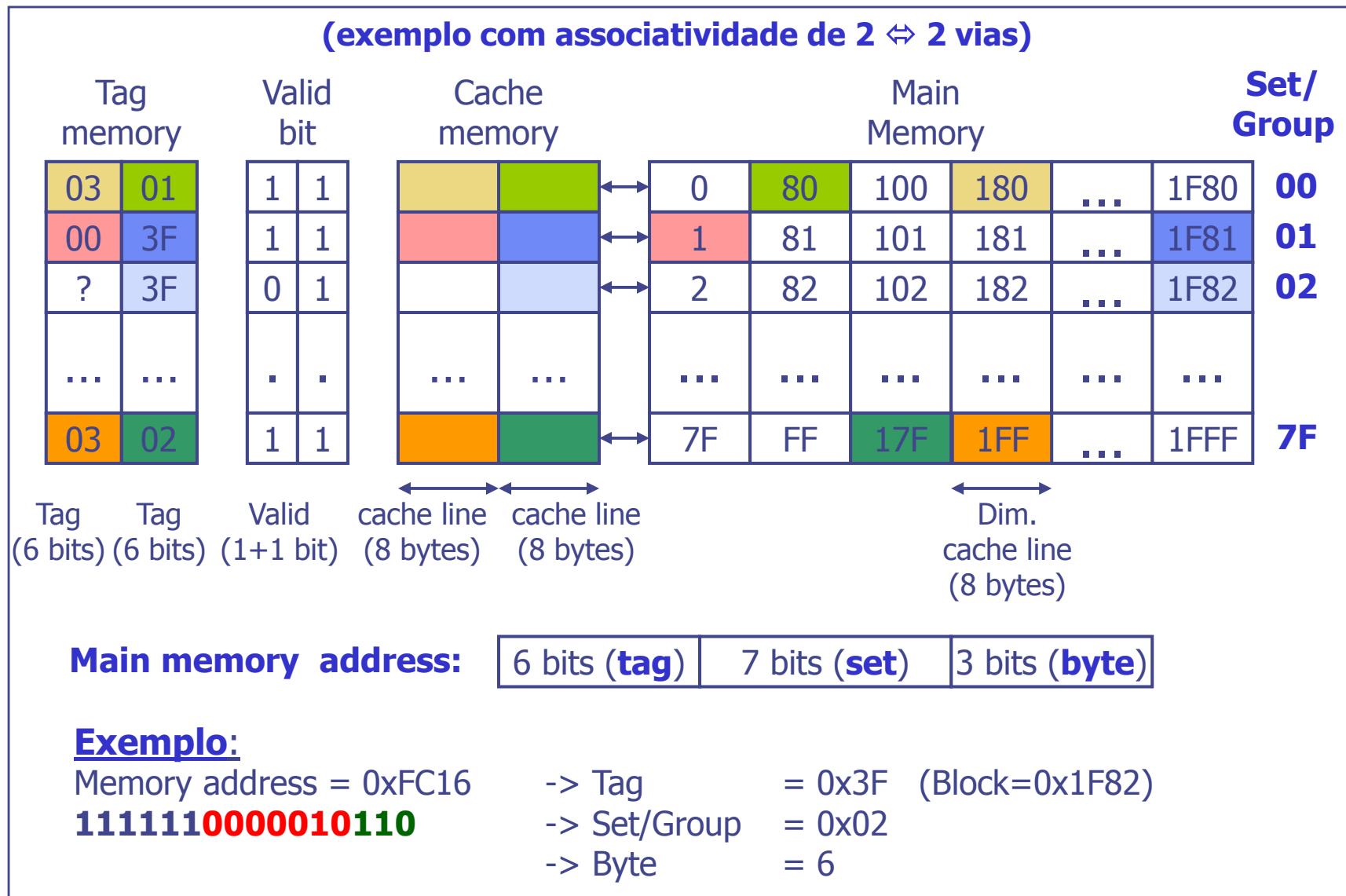
# Cache com mapeamento direto



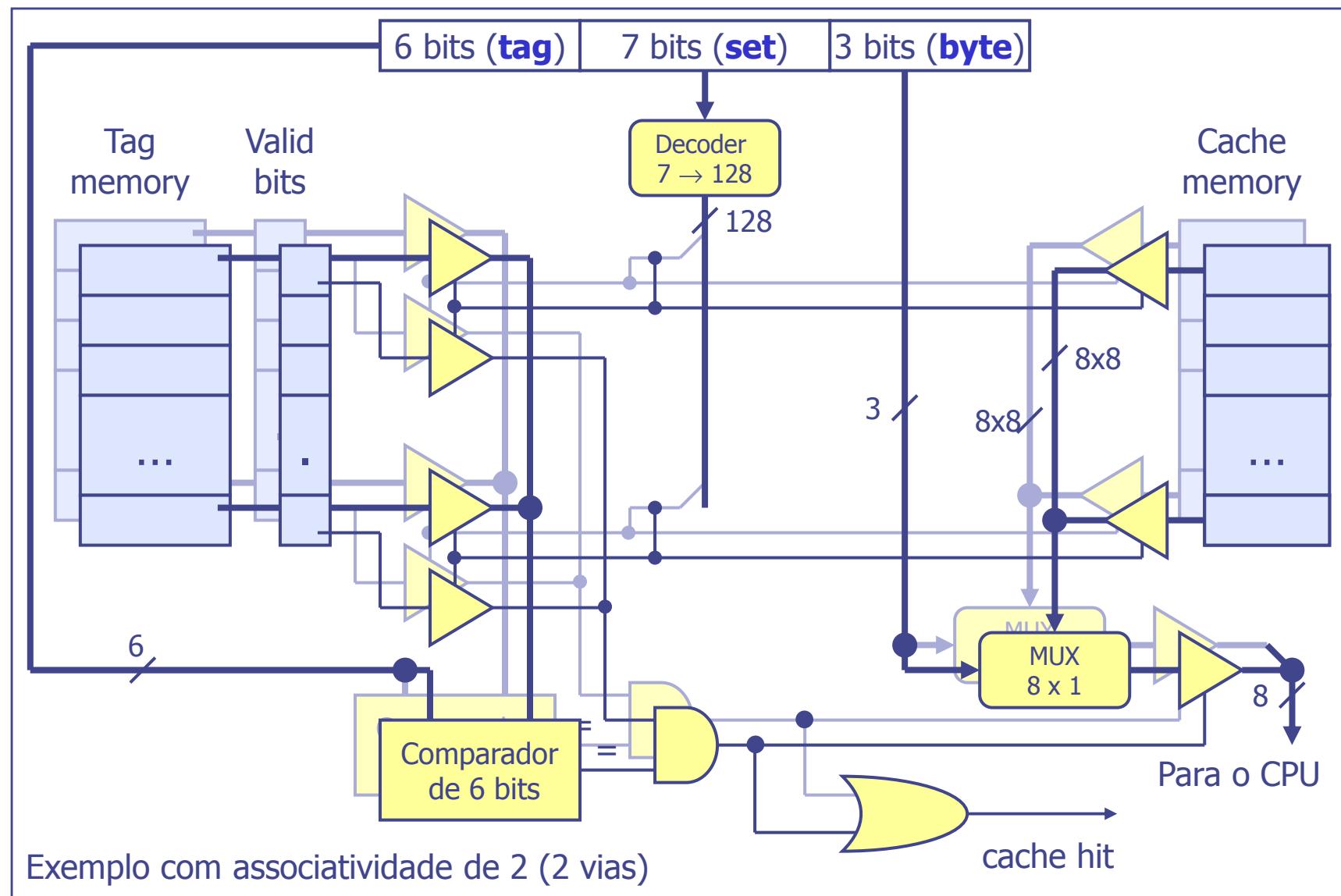
# Cache com mapeamento direto

- A cada bloco da memória principal é associada uma linha da cache; o mesmo bloco é sempre colocado na mesma linha
- Maior vantagem:
  - Simplicidade de implementação
- Maior desvantagem:
  - Vários blocos têm associada a mesma linha
  - Num dado instante apenas um bloco de um dado grupo pode residir na cache, o que tem como consequência que alguns blocos podem ser substituídos e recarregados várias vezes, mesmo existindo espaço na cache para guardar todos os blocos em uso
- Uma melhoria óbvia deste tipo de cache seria permitir o armazenamento simultâneo de mais que um bloco do mesmo grupo
- É esta melhoria que é explorada na cache **parcialmente associativa ("set associative")**

# Cache com mapeamento parcialmente associativo



# Cache com mapeamento parcialmente associativo

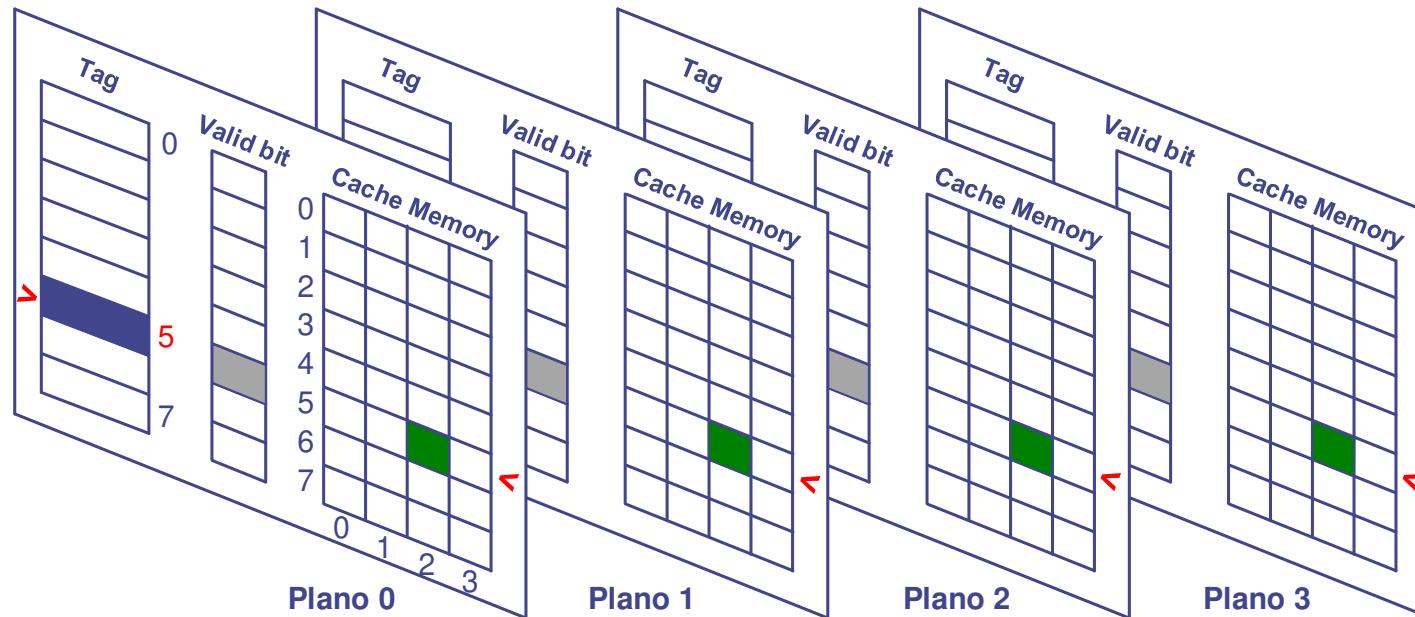


# Cache parcialmente associativa

- A cache parcialmente associativa é semelhante à cache com mapeamento direto, mas a primeira permite que mais que um bloco de um mesmo grupo possa ser carregado na cache
- Uma cache com associatividade de 2, ou seja, com 2 vias, permite que 2 blocos do mesmo grupo possam estar simultaneamente na cache
- A divisão do endereço de memória é igual ao do mapeamento direto (o campo "group" é normalmente designado por "set" - conjunto), mas agora há "n" possíveis lugares onde um dado bloco de um mesmo grupo pode residir; "n" é o número de vias da cache e nesse caso diz-se que a cache tem associatividade de "n"
- As "n" vias onde o bloco pode residir têm que ser procuradas simultaneamente; o hit da cache resulta do OR dos hits de cada uma das vias
- (Block-set associative cache / n-way-set associative cache)

# Cache com mapeamento parcialmente associativo

- Esquematização de uma cache com **associatividade de 4**, de **128 bytes**, com **blocos de 4 bytes** ( $128 / 4 = 32$  bytes por via,  $32 / 4 = 8$  linhas)
- Endereço gerado pelo CPU (16 bits) 0x6A96: **0110101010010110**



- A comparação dos 11 bits mais significativos (A15-A5) do endereço com as "tags" é feita simultaneamente nas 4 vias
- A posição da memória "tag" onde é feita a comparação é determinada pelos bits A4-A2 do endereço (posição 5 no exemplo)
- A via onde ocorre o hit (se ocorrer) fornece o byte armazenado na cache na posição determinada pelos bits A1-A0 do endereço (posição 2 no exemplo)

# Políticas de substituição de blocos

- As políticas de substituição de blocos na cache, na ocorrência de um miss, são várias. As mais utilizadas são as seguintes:
- **FIFO** – First in first out: é substituído o bloco que foi carregado há mais tempo
- **LRU** – Least Recently Used: é substituído o bloco da cache que está há mais tempo sem ser referenciado
- **LFU** – Least Frequently Used: substituído o bloco menos acedido
- **Random**: substituição aleatória (testes indicam que não é muito pior do que LRU)

# Políticas de escrita

- **Write-through**

- Todas as escritas são realizadas simultaneamente na cache e na memória principal
- Se endereço ausente na cache, atualiza apenas a memória principal (**write-no-allocate**)
- A memória principal está sempre consistente

- **Write-back**

- Valor escrito apenas na cache; novo valor é escrito na memória quando o bloco da cache é substituído
- "**Dirty bit**" (este bit é ativado quando houver uma escrita em qualquer endereço do bloco presente na linha da cache)
- Se endereço ausente na cache, carrega o bloco para a cache e atualiza-o (**write-allocate**)
- Mais complexo do que "write-through"

# Exemplo – Intel Core i7-6500U

- Número de cores: 2
  - Core 1: L1 data (32 KB), L1 instr (32 KB), L2 data+instr (256 KB)
  - Core 2: L1 data (32 KB), L1 instr (32 KB), L2 data+instr (256 KB)
  - L3: (4 MB) partilhada pelos 2 cores

Cache	Size	Associativity	Line Size
L1 Data	2 x 32 KB	8-way set associative	64 bytes
L1 Instructions	2 x 32 KB	8-way set associative	64 bytes
L2	2 x 256 KB	4-way set associative	64 bytes
L3	4 MB (shared cache)	16-way set associative	64 bytes

- Qual a dimensão do bloco das caches L1, L2 e L3?
- Qual o número de linhas das caches L1, L2 e L3?

# Exercícios

1. Qual a posição dos endereços de bloco 1 e 29 numa cache de mapeamento direto com 8 linhas?
2. Considere um sistema computacional com um espaço de endereçamento de 32 bits e uma cache com 256 blocos de 8 bytes cada um.
  - Qual o tamanho em bits dos campos tag, group e byte supondo que a cache é: 1) associativa; 2) mapeamento direto; 3) com associatividade de 2.
3. Para a implementação de uma cache com 64KB de dados e blocos de 4 bytes num sistema computacional com um espaço de endereçamento de 32 bits, quantos bits de armazenamento são necessários supondo:
  - uma cache associativa;
  - uma cache com mapeamento direto;
  - uma cache com associatividade de 2.

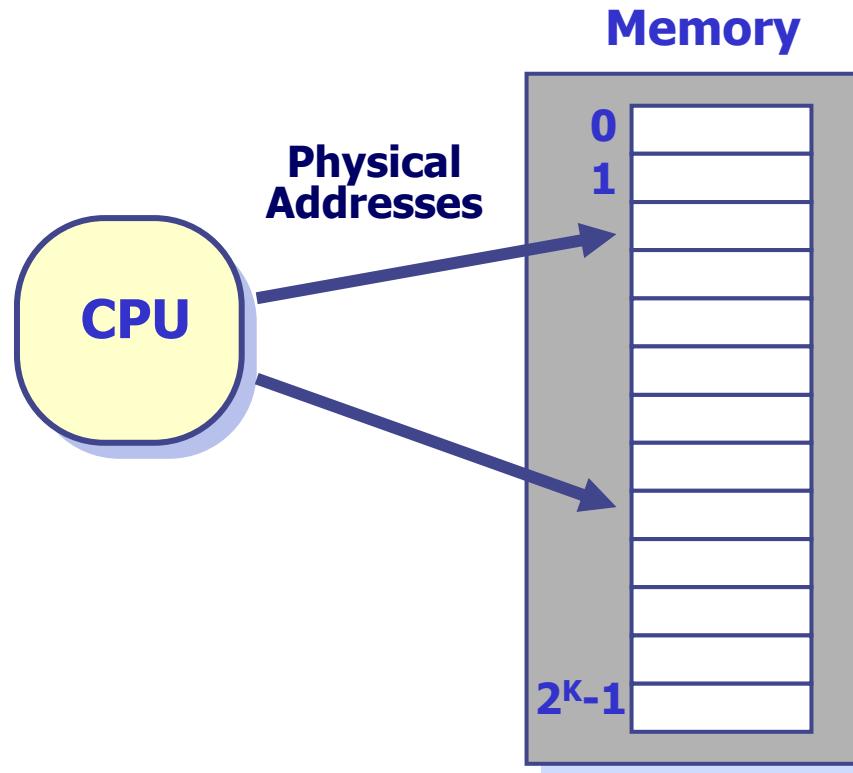
# Aulas 22, 23 e 24

- Memória Virtual
  - Motivações para a sua utilização
  - Endereço virtual, endereço físico
  - *Page Table*. Tradução de endereços. *Page Fault*
  - "Translation-lookaside buffer" (TLB)
  - Integração da memória virtual com TLB e *cache*

José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

# Sistema apenas com memória física

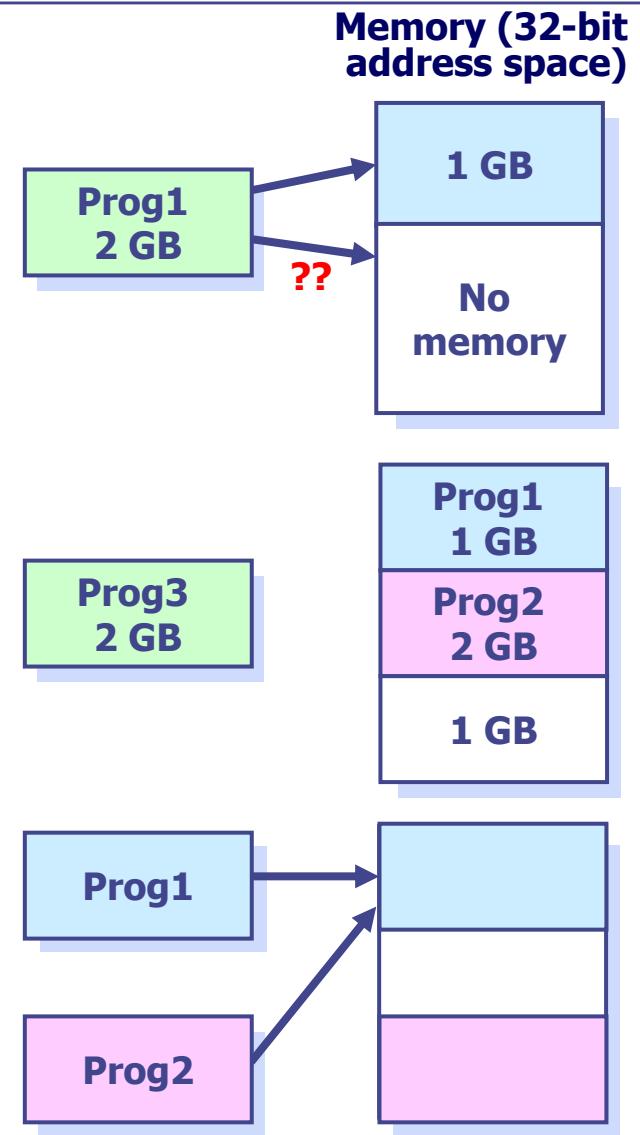
- Num sistema apenas com memória física



- Os endereços gerados pelo CPU apontam diretamente para as posições de memória a que este pretende aceder

# Sistema apenas com memória física – problemas

- Memória insuficiente
  - O que acontece se o programa Prog1 tentar aceder a mais do que 1 GB de memória?
- Gestão da memória disponível
  - Supondo que o sistema tem 4 GB de memória, os programas Prog1 e Prog2 usam 3 GB. Quando Prog1 terminar ficam 2GB disponíveis, mas Prog3 não pode executar porque não existe um bloco de 2 GB
  - Fragmentação da memória
- Segurança
  - Um programa pode escrever fora da zona de memória que lhe está atribuída



# Sistema apenas com memória física – problemas

- Num sistema multi-processo, a memória disponível tem de ser partilhada pelos processos em execução
  - Cada processo tem as suas próprias necessidades de memória
- A memória pode facilmente ser corrompida se um processo escrever (de forma involuntária ou intencional) na zona de memória atribuída a outro processo
  - O processo que viu a sua zona de memória alterada falhará por razões que nada têm a ver com a sua lógica de funcionamento
- Como gerir adequadamente a memória disponível entre os vários processos?
- O que fazer se a memória necessária para executar todos os processos for superior à memória física disponível?
- Como garantir que um processo não escreve na zona de memória atribuída a outro processo?

# Memória virtual

- É uma abstração que permite uma utilização eficiente do sistema de memória em sistemas multi-processo (mantendo o modelo de memória hierárquica)
- Esta técnica é usada em sistemas de computação geral baseados em microprocessador, com sistema operativo (não usado em sistemas simples baseados em microcontrolador)
- O conceito de "memória virtual" não é novo: descrito pela primeira vez por Kilburn et al. em 1962

# Motivações para a utilização de Memória virtual

- Eficiência na utilização da memória
  - Memória deve ser partilhada pelos vários processos em execução
  - Em memória apenas deve residir a informação necessária
- Segurança
  - Devem existir mecanismos de segurança que impeçam que um processo altere as zonas de memória dos outros processos
- Transparência
  - Um processo deve ter acesso à memória de que necessita (eventualmente mais do que a memória física DRAM)
  - Um processo deve correr como se toda a memória lhe pertencesse
- Partilha de memória
  - Vários processos devem poder aceder à mesma zona de memória (de forma controlada)

# Memória virtual

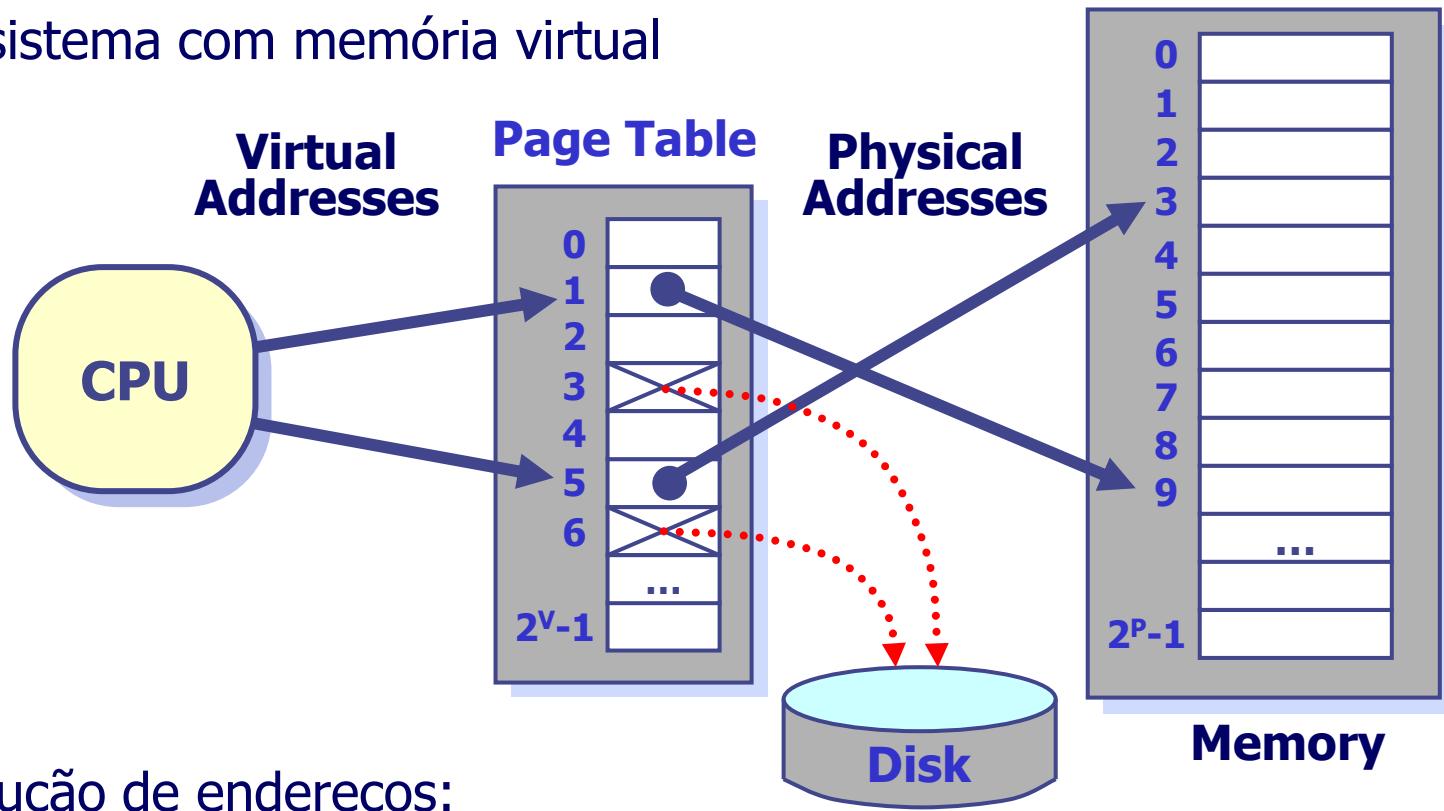
- Utilização eficiente da memória física:
  - Vários programas podem estar em execução "simultânea" no computador
  - A memória pode estar fragmentada
  - A quantidade de memória total necessária para executar todos os programas pode ser superior à memória física disponível
  - No entanto, apenas uma fração dessa quantidade de memória está activamente a ser usada num determinado instante de tempo
  - Assim, na memória principal apenas é necessário ter as "zonas ativas" de todos os programas (instruções e dados) a correr no computador (as restantes podem estar em disco)
  - O sistema operativo pode reservar mais memória para um processo ou libertar memória que já não esteja a ser usada, de acordo com as necessidades

# Memória virtual

- Segurança:
  - Múltiplos processos partilham a mesma memória física
  - É necessário usar mecanismos de proteção que assegurem que um dado processo apenas acede (leitura e/ou escrita) às zonas de memória que lhe foram atribuídas
  - Endereços usados pelos processos não são endereços da memória física
- Transparência
  - Cada processo tem o seu próprio espaço de endereçamento que pode usar na totalidade, de forma exclusiva
  - Cada processo executa como se fosse o único a ocupar a memória
- Partilha de memória
  - Com a implementação de mecanismos de proteção torna-se possível a partilha de zonas de memória entre processos

# Princípio de funcionamento

- Um sistema com memória virtual



- Tradução de endereços:
  - Os endereços virtuais gerados pelo CPU são convertidos para endereços físicos através de uma tabela, designada por "Page Table"
  - A tradução de endereços tem que ser temporalmente eficiente

# Nomenclatura-base

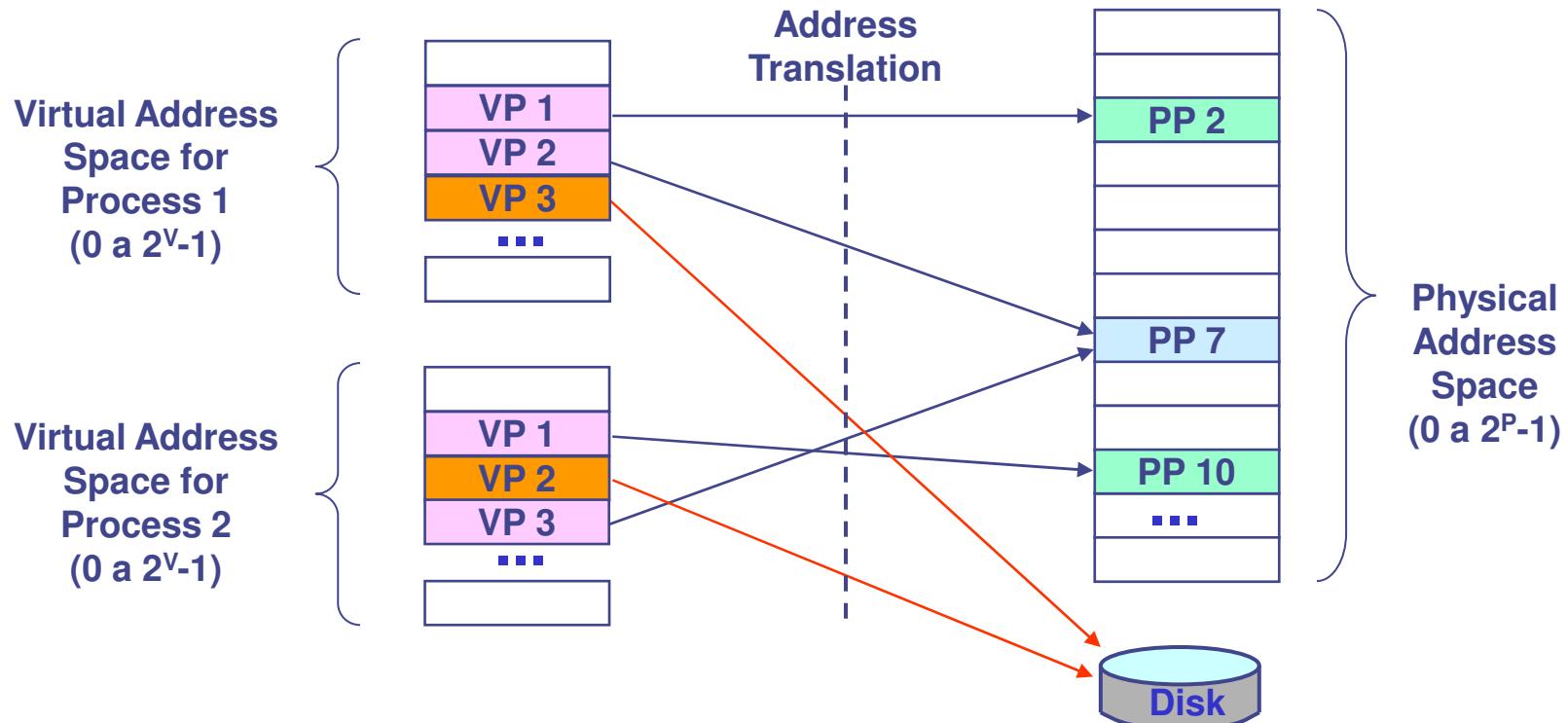
- **Endereço virtual** (*virtual address*)
  - Um endereço gerado pelo CPU
  - O **espaço de endereçamento virtual** é a coleção de todos os endereços virtuais (0 a  $2^V-1$ , num espaço de endereçamento de V bits)
- **Endereço físico** (*physical address*)
  - Um endereço da memória principal (e.g. DRAM) ou do disco
  - O **espaço de endereçamento físico** é a coleção de todos os endereços físicos (0 a  $2^P-1$ , num espaço de endereçamento de P bits)
- **Tradução de endereços** (*address translation*)
  - O processo pelo qual um endereço virtual é traduzido num endereço físico
  - Um endereço virtual é traduzido num endereço físico quando o CPU acede à memória (para ler uma instrução ou para aceder a uma palavra de dados)

# Princípio de funcionamento

- Os espaços de endereçamento virtual e físico são divididos em blocos; os blocos têm a mesma dimensão nos dois espaços de endereçamento
  - Na terminologia de memória virtual estes blocos designam-se por **"páginas"**
- Cada processo tem o seu próprio **espaço de endereçamento virtual**
  - O espaço de endereçamento virtual de todos os processos pode seguir o mesmo modelo: ter início no mesmo endereço, ter a primeira instrução no mesmo endereço, ter o *data segment* na mesma zona da memória, etc.
  - A atribuição de **páginas físicas a páginas virtuais** é feita pelo sistema operativo
  - Quando o processo é carregado em memória, o sistema operativo faz a atribuição dos endereços físicos a endereços virtuais (**relocation** – construção da *Page Table*)
  - A relocação permite que um programa possa ser carregado em qualquer página de memória

# Princípio de funcionamento

- Espaço de endereçamento virtual de V bits
- Espaço de endereçamento físico de P bits

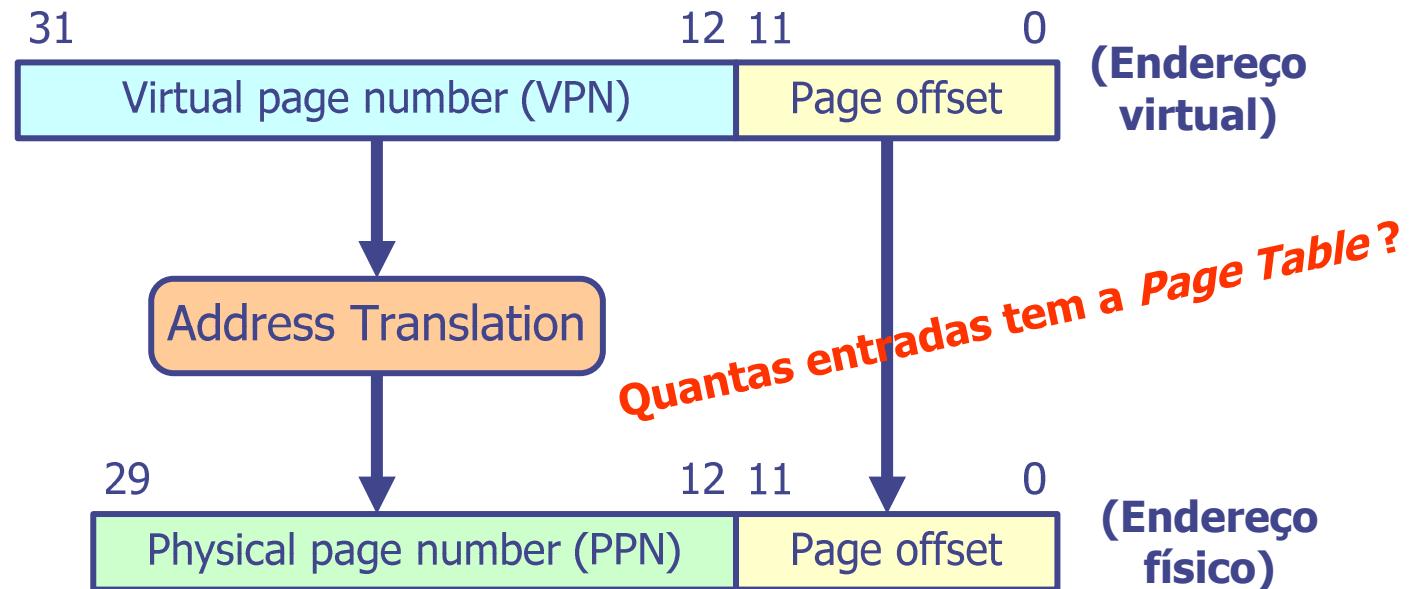


VP ≡ Virtual page  
PP ≡ Physical page

Neste exemplo a página física 7 é partilhada pelos dois processos com endereços virtuais diferentes (exemplo de um zona de memória partilhada: leitura ou leitura/escrita)

# Princípio de funcionamento

- Mapeamento entre um endereço virtual (V bits) e um endereço físico (P bits) - exemplo c/ **V = 32 bits** e **P = 30 bits** e páginas de 4 kBytes

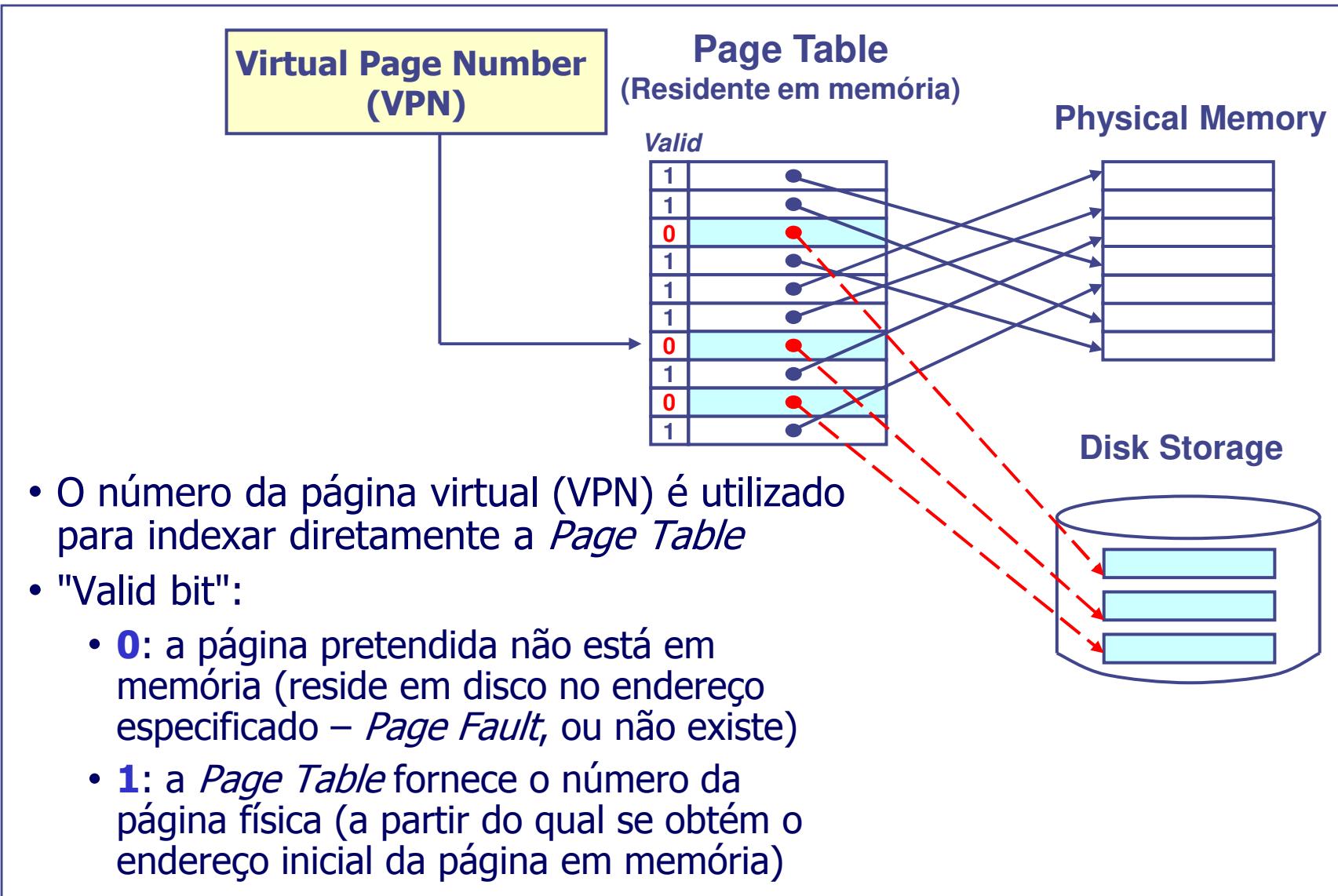


- Espaço de endereçamento virtual:  $2^{32} = 4 \text{ GB}$
- Espaço de endereçamento físico:  $2^{30} = 1 \text{ GB}$
- Dimensão da página:  $2^{12} = 4 \text{ kB}$
- Número de páginas da memória virtual:  $2^{20} = 1 \text{ M}$
- Número de páginas da memória física:  $2^{18} = 256\text{k}$

# Tradução de endereços

- A *Page Table* contém um número de entradas igual ao número máximo de páginas virtuais (para o exemplo dado anteriormente, a tabela teria  $2^{20}$  entradas). Por esta razão não é necessária uma "tag"
- O número da página virtual (VPN) é utilizado para indexar diretamente a *Page Table*
- A tradução de endereços tem que ser rápida, uma vez que ocorre em cada acesso do CPU à memória. Realizada por hardware!

# Tradução de endereços

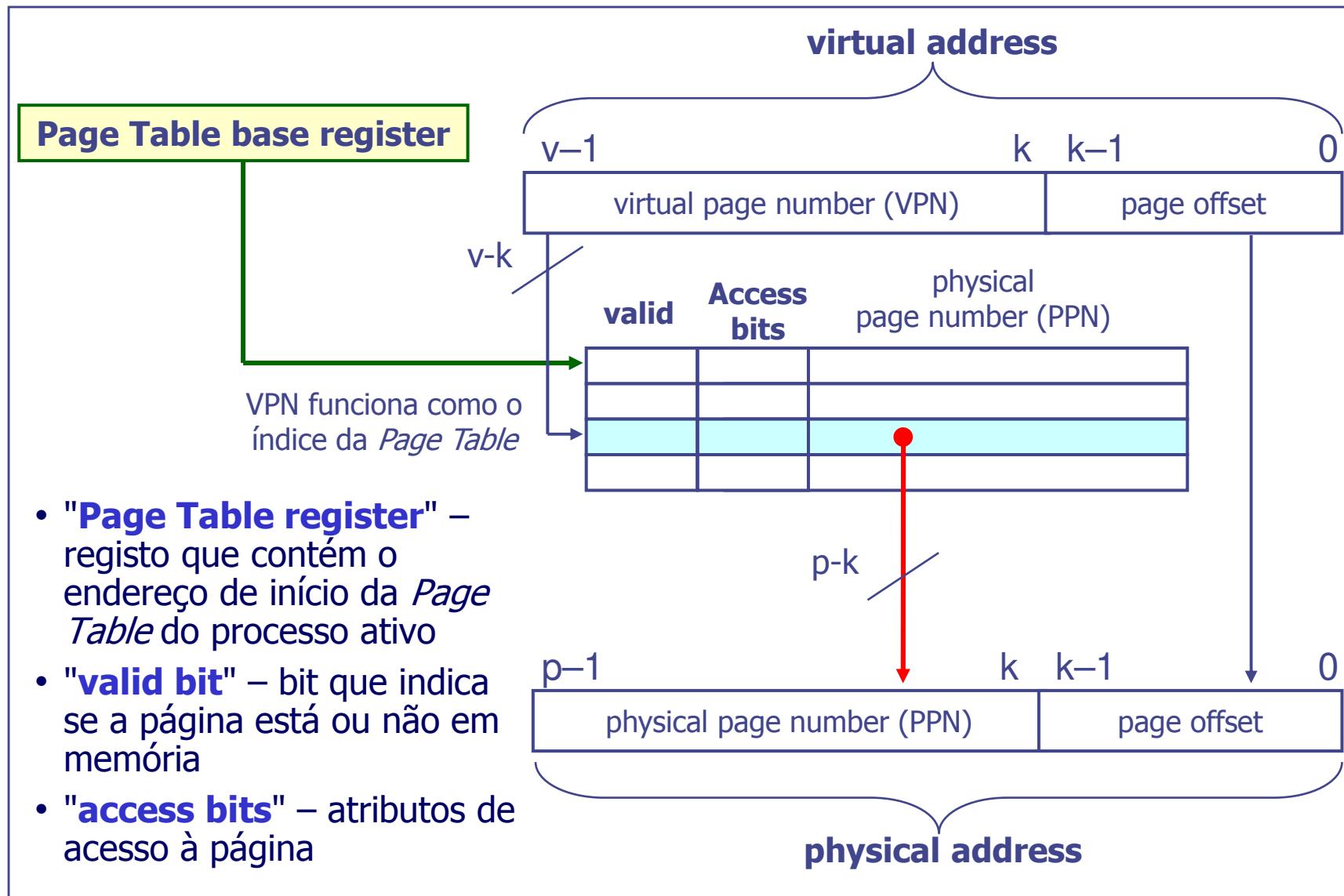


- O número da página virtual (VPN) é utilizado para indexar diretamente a *Page Table*
- "Valid bit":
  - **0**: a página pretendida não está em memória (reside em disco no endereço especificado – *Page Fault*, ou não existe)
  - **1**: a *Page Table* fornece o número da página física (a partir do qual se obtém o endereço inicial da página em memória)

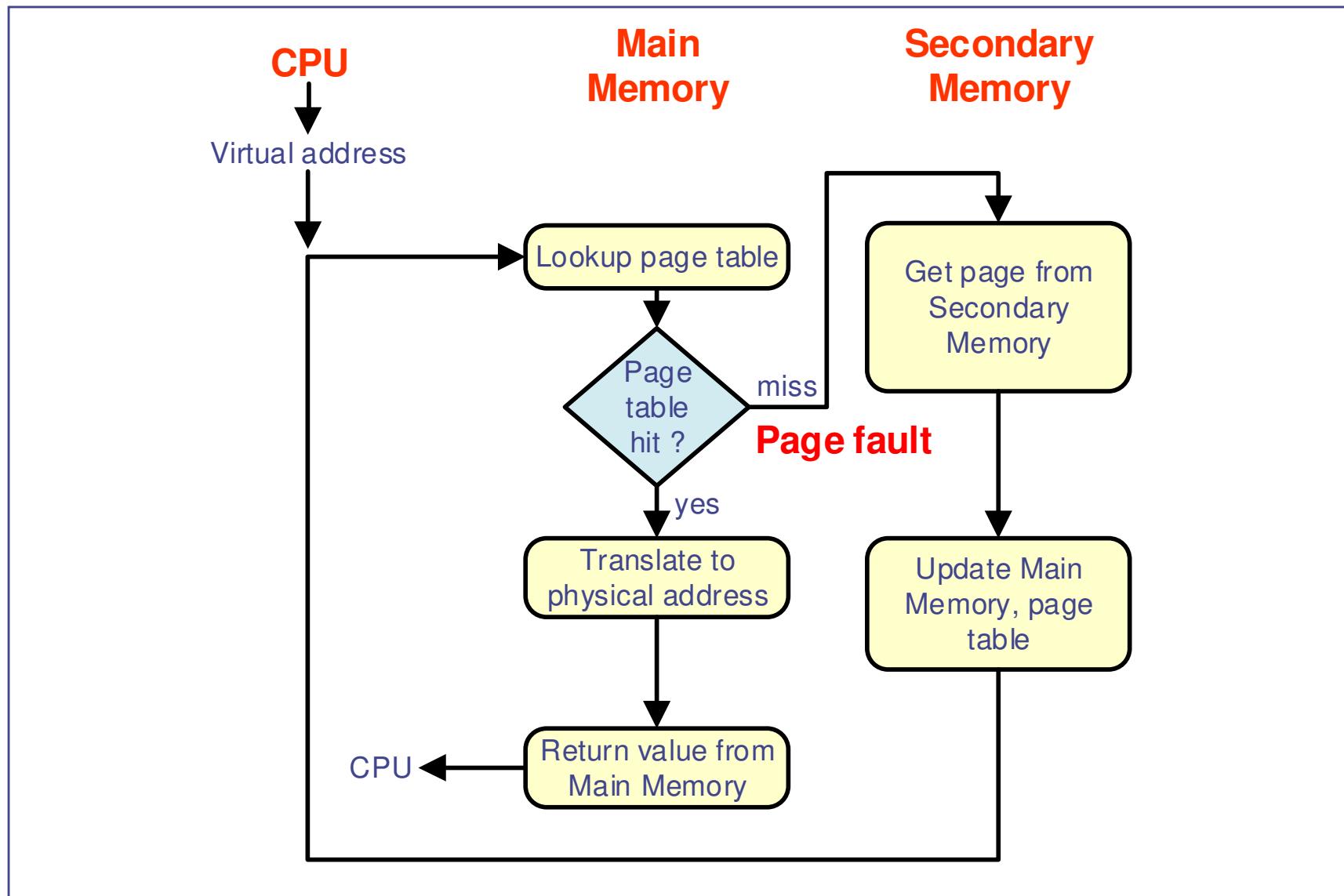
# Operação

- Tradução
  - Uma *Page Table* por processo
  - O Virtual Page Number (VPN), obtido do endereço gerado pelo processador, forma o índice de acesso à *Page Table*
- Obtenção do endereço físico da página
  - A entrada da tabela correspondente ao VPN contém informação sobre a página
  - Se "valid bit = 1" a página reside na memória
    - O endereço de acesso é obtido da entrada da tabela correspondente ao VPN concatenado com o "page offset"
  - Se "valid bit = 0" a página reside no disco ou não existe
    - *Page Fault*
    - Copiar para a memória a página pretendida (pode envolver a substituição de uma página residente em memória)
    - "valid bit" passa a "1"
- Proteção
  - Os atributos de acesso à página (Read, Read/Write, Execute) são verificados em cada tradução (para cada VPN)

# Tradução de endereços

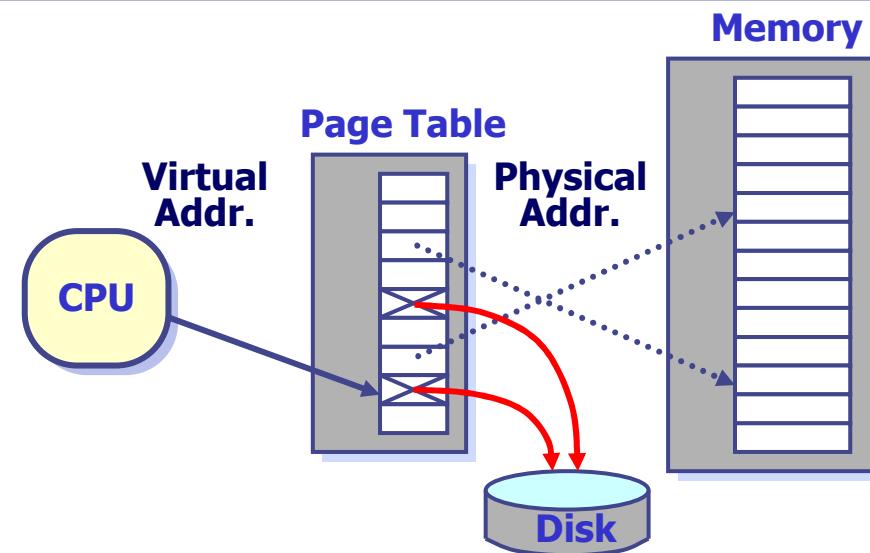


# "Page Fault"

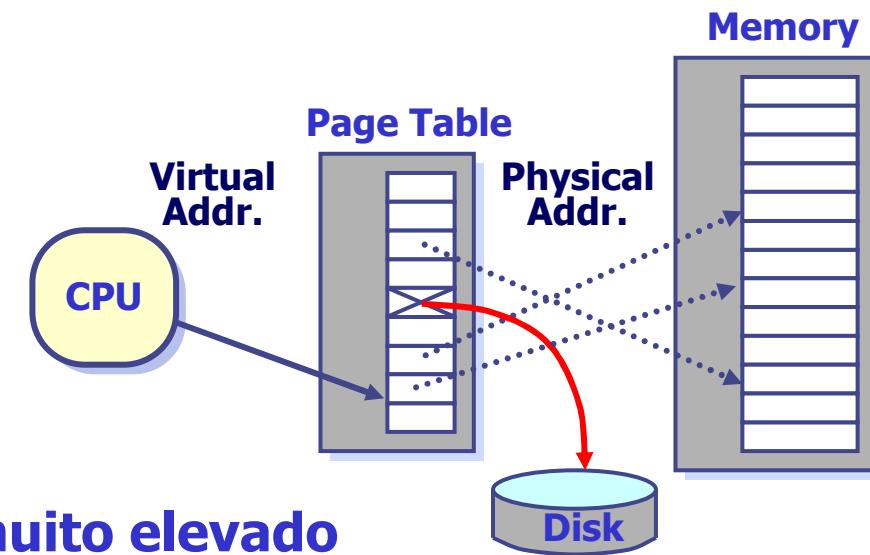


# "Page Fault"

- Ocorrência de um *Page Fault*



- Após um *Page Fault*



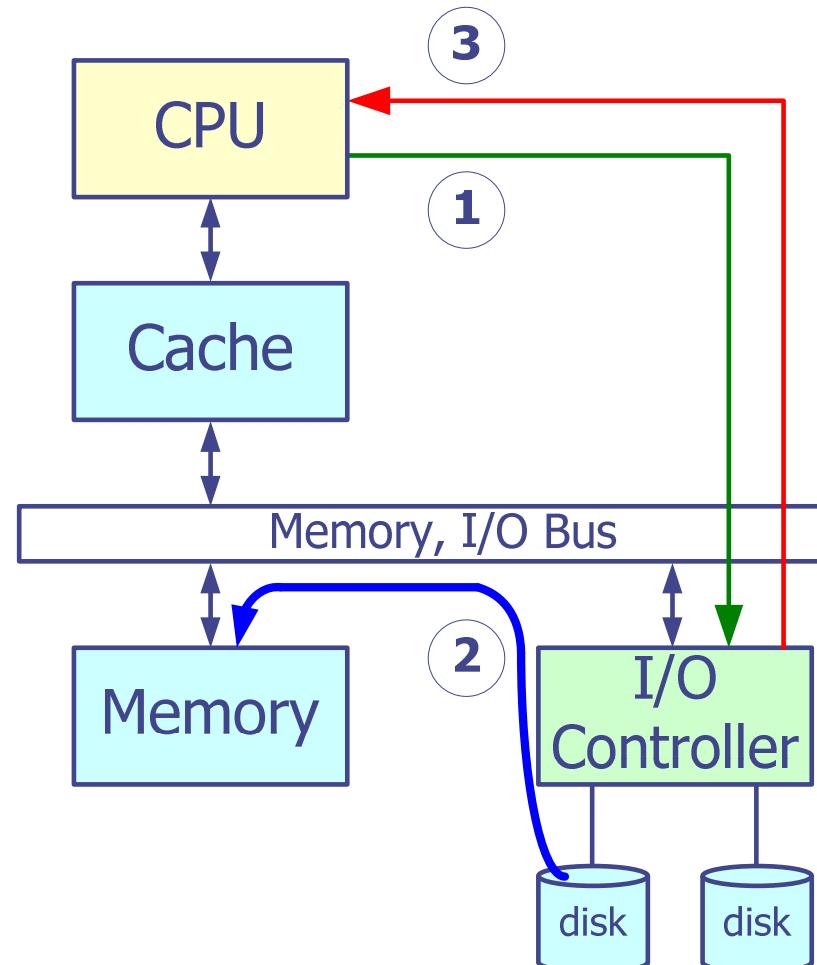
**custo de um *Page Fault* é muito elevado**

# "Page Fault"

- Se a página que o CPU pretende aceder não está em memória ("Valid bit" da *Page Table* com o valor 0), é gerado um *Page Fault*
- Um *Page Fault* tem um custo elevado em termos temporais
- Os *Page Faults* são tratados por software:
  - O gestor de memória (Memory Management Unit - MMU) gera uma exceção que transfere o controlo para o Sistema Operativo
  - O Sistema Operativo decide onde colocar, na memória, a página em falta e desencadeia a respetiva transferência a partir do disco. Se não houver espaço na memória o SO tem que começar por substituir uma página existente e isso pode envolver a cópia dessa página para o disco
  - O processamento completo de um *Page Fault* pode demorar dezenas de milhões de ciclos de relógio
  - O Sistema Operativo suspende o processo corrente e lança outro
  - O processo suspenso é retomado quando o *Page Fault* for resolvido, numa decisão também da responsabilidade do Sistema Operativo

# Parte do processamento de um Page Fault

1. SO configura o "I/O controller" para transferir, para memória, a página em falta (bloco):
  - Dimensão do bloco
  - Endereço de início no disco
  - Endereço destino na memória (escolhido pelo SO)
2. A transferência processa-se por DMA
3. O "I/O controller" sinaliza o fim da transferência:
  - Gera uma interrupção
  - O SO retoma a execução do processo suspenso



# Política de substituição de páginas na memória

- Quando ocorre um *Page Fault* e a memória física está toda ocupada, é necessário decidir qual a página que tem que sair
- A política normalmente usada é:
  - LRU (Least Recently Used - é substituída a página que está há mais tempo sem ser referenciada)
  - NRU (Not Recently Used) - é uma versão simplificada do LRU
- Implementação do NRU
  - Cada VPN de uma *Page Table* contém um "reference bit"
  - Periodicamente os "reference bits" são colocados a zero pelo Sistema Operativo
  - Quando uma página é acedida (*touched*) o "reference bit" é colocado a 1
  - As páginas candidatas a serem substituídas são as que têm o "reference bit" a 0 (é substituída aleatoriamente uma dessas páginas)

# Política de escrita

- **Write-back** - a utilização de um esquema do tipo "write-through" seria impraticável uma vez que a escrita de uma página no disco demora um tempo que penalizaria fortemente o desempenho do sistema
  - Escrita em disco página a página
  - Página só é escrita em disco quando necessita de ser retirada da memória física
- **Dirty bit**
  - Se "dirty bit" = 0, a página não necessita de ser escrita em disco aquando da sua substituição (*overwrite*)
  - Se "dirty bit" = 1, a página que vai ser substituída foi alterada. Antes de ser substituída, essa página é copiada para o disco; após essa operação, a nova página é então copiada do disco para a zona da memória que ficou livre
- **Write-allocate**
  - Escrita numa página que não reside em memória física: carrega essa página para a memória e escreve

# Implementação de mecanismos de proteção

- Um processo não pode interferir de maneira nenhuma com o funcionamento de outro (seja de forma involuntária ou intencional). Isso é garantido porque:
  - Os espaços de endereçamento de cada processo são independentes
  - Cada processo tem a sua própria *Page Table*, gerida exclusivamente pelo Sistema Operativo
- Dentro do mesmo processo:
  - Diferentes zonas do espaço de endereçamento podem ter diferentes permissões de acesso. Por exemplo, uma zona de instruções não tem permissão de escrita, uma zona de dados não tem permissão de execução, etc.
  - Os atributos de acesso a cada página (Read, Read/Write, Execute) são verificados em cada tradução de endereços (para cada VPN)

# Implementação de mecanismos de proteção

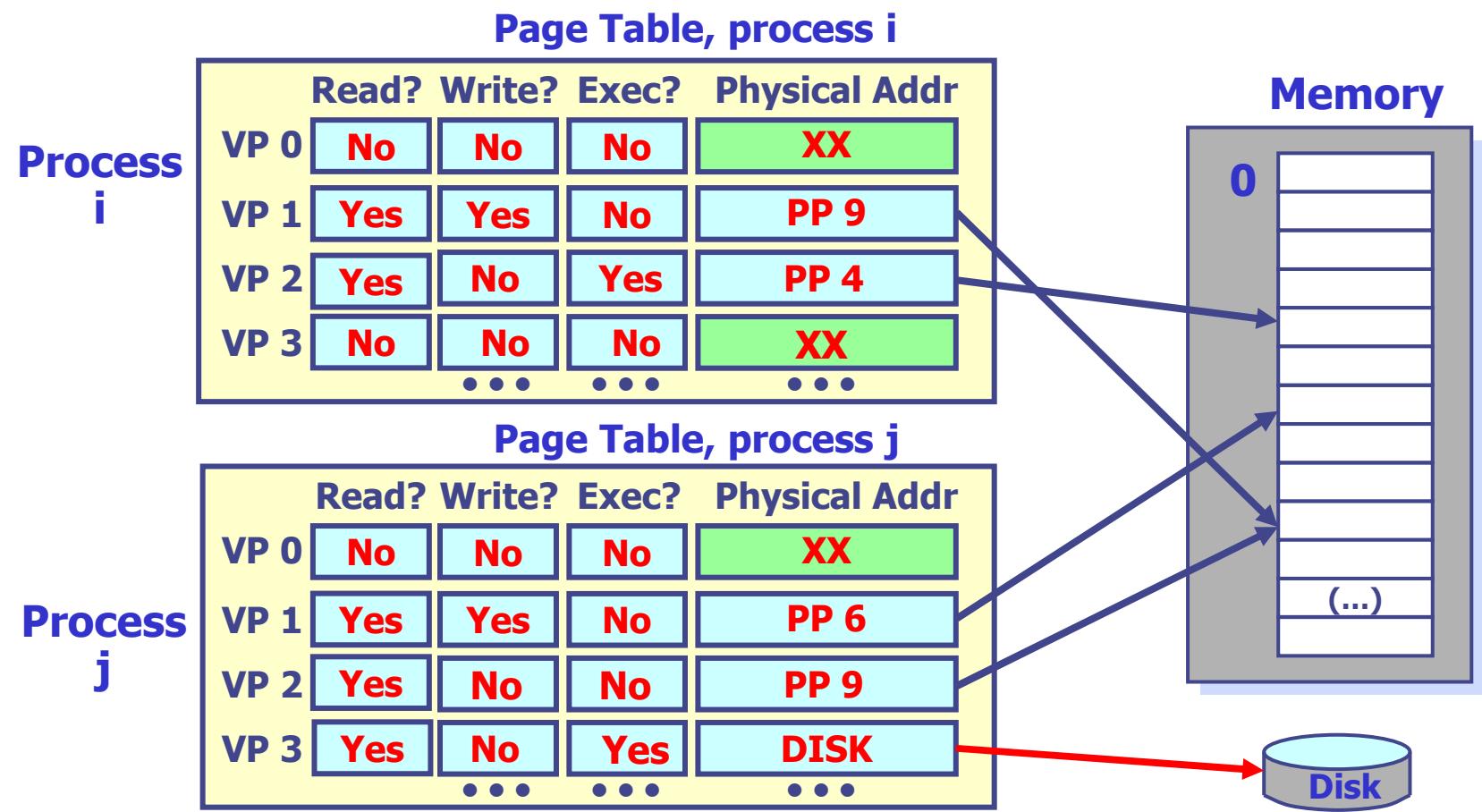
- Cada entrada da *Page Table* contém informação relativa a atributos de acesso do processo respetivo:
  - Read
  - Read / Write
  - Execute

**Page Table**

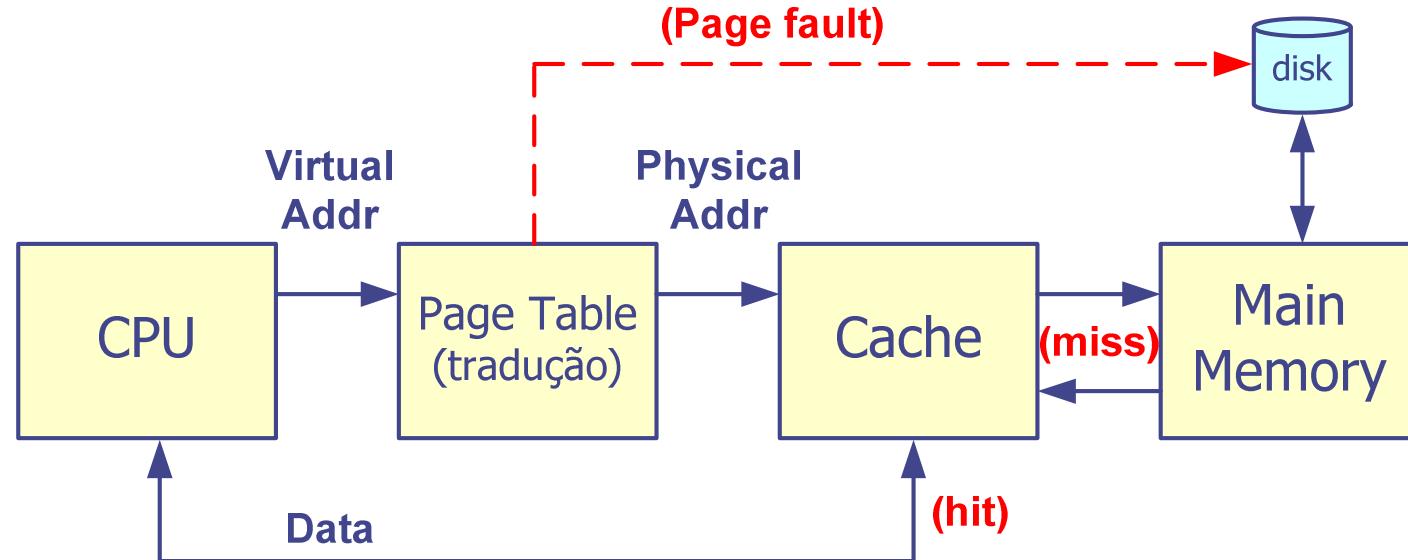
Process	Read?	Write?	Exec?	Physical Addr
VP 0	No	No	No	XX
VP 1	Yes	No	No	PP 9
VP 2	Yes	Yes	No	PP 4
VP 3	Yes	No	Yes	PP 7
	• • •	• • •	• • •	• • •

# Implementação de mecanismos de proteção

- O hardware gera uma exceção se ocorrer uma tentativa de violação dos atributos de acesso (e.g. no SO Windows: "General protection fault", no SO Linux: "Segmentation fault")



# Memória virtual + cache



- A *Page Table* reside na memória principal. Isso significa que cada acesso à memória virtual implica dois acessos à memória física:
  - 1 acesso para indexar a *Page Table* e obter o endereço físico
  - 1 acesso para ler/escrever os dados
- Será esta uma solução viável? Como resolver este problema?

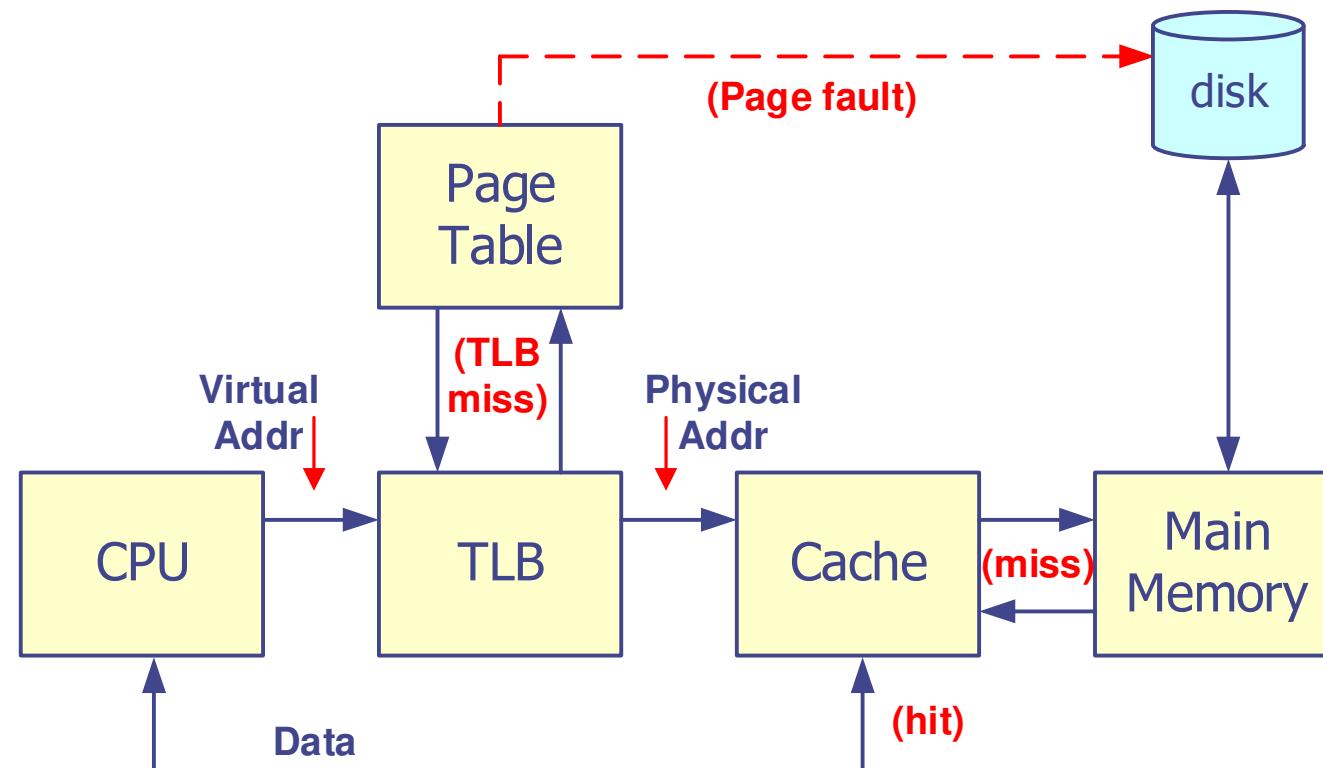
# Translation-lookaside buffer (TLB)

- Os acessos à memória virtual contêm localidade espacial e temporal
- Uma forma de resolver o problema é ter uma parte da *Page Table* numa memória rápida, semelhante a uma cache, onde se encontram as entradas da tabela mais recentemente utilizadas
- Esta memória é normalmente designada por "translation-lookaside buffer" - TLB
- A TLB é tipicamente implementada como uma memória associativa com procura paralela
- Páginas residentes em disco não são referenciadas na TLB

# Translation-lookaside buffer (TLB)

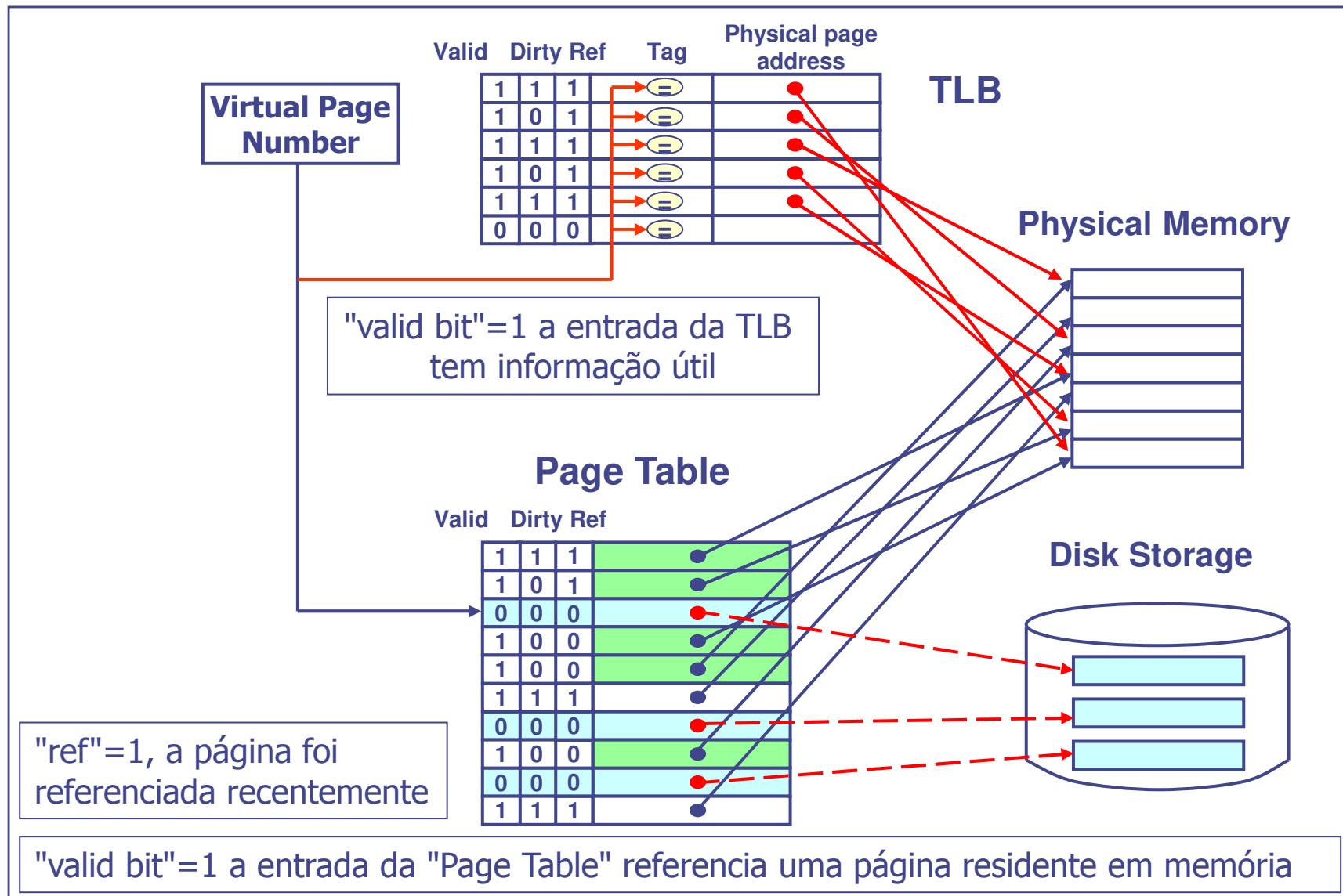
- Em cada acesso, no processo de tradução, verifica-se se o número da página virtual (VPN) está na TLB
- Se VPN está na TLB: "Hit" – o endereço da página física é obtido de imediato da TLB
- Se VPN não está na TLB: "Miss" – a *Page Table* é verificada e pode originar, ou não, um *Page Fault*.
  - Se o acesso à *Page Table* não originar um *Page Fault*, o endereço físico e os atributos da página são copiados da *Page Table* para a TLB (pode envolver a substituição de uma entrada na TLB)
  - Se o acesso à *Page Table* originar um *Page Fault*, é necessário fazer todo o processamento do *Page Fault* que culmina com a atualização da *Page Table*. A seguir o endereço físico e os atributos da página são copiados da *Page Table* para a TLB (pode envolver a substituição de uma entrada na TLB)

# TLB + Memória virtual + cache

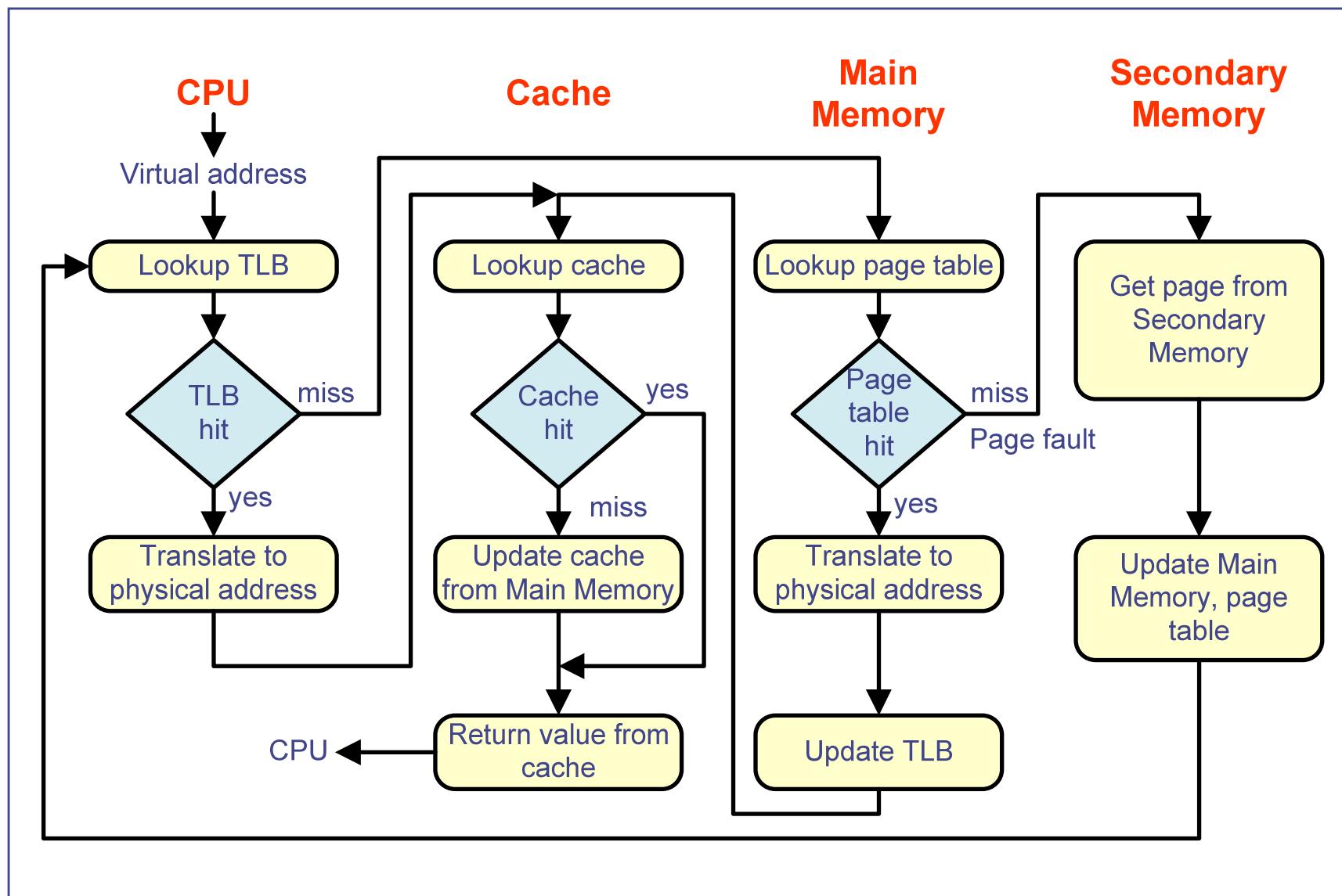


- Se a página virtual está na TLB (hit) a tradução é muito rápida
- Se a página virtual não está na TLB (miss) é necessário consultar a *Page Table*, que reside na memória principal, e atualizar a TLB
- No acesso à *Page Table* pode ocorrer um *Page Fault*

# Translation-lookaside buffer (TLB)



# Operação da hierarquia de memória

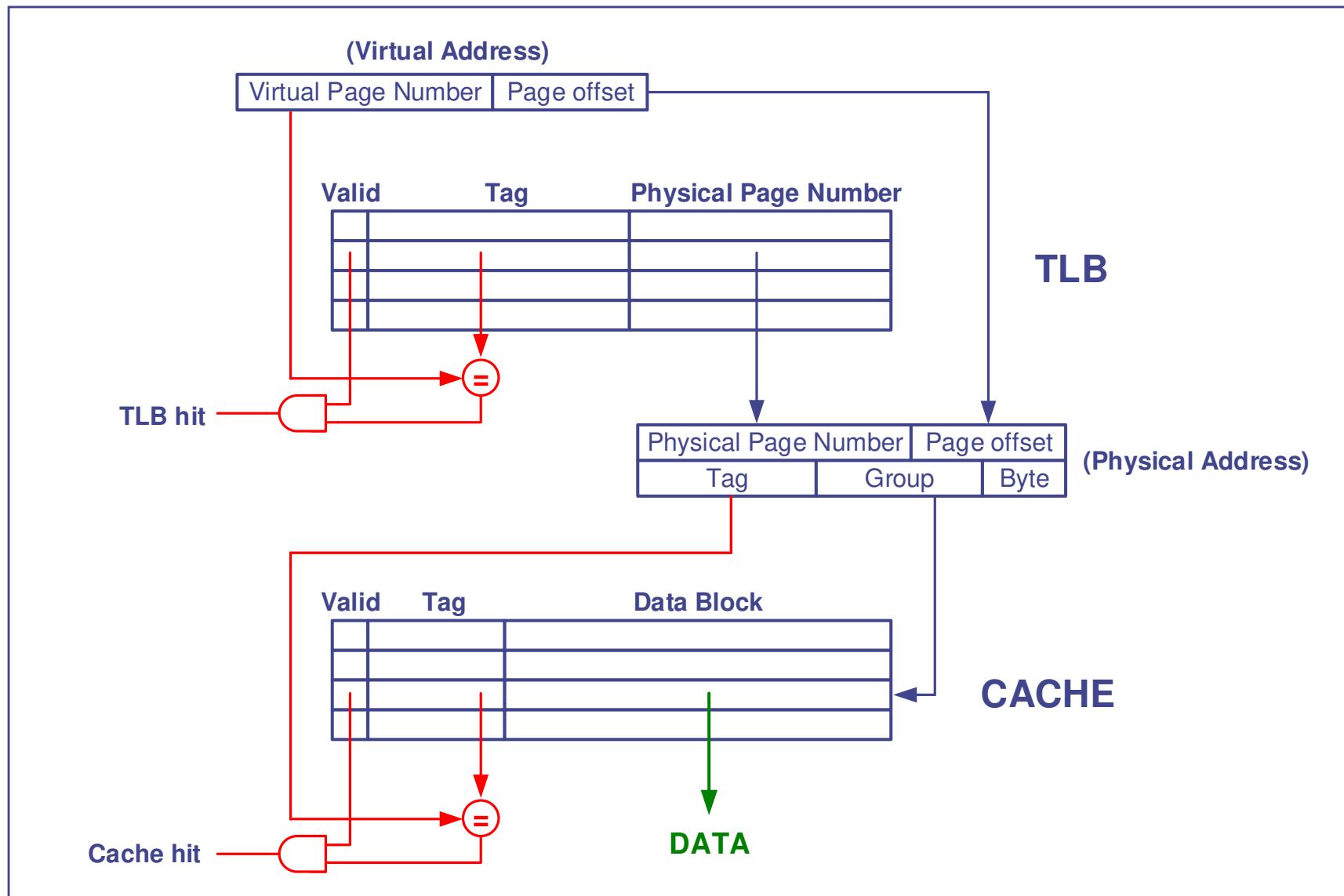


# Tradução de endereços com a TLB

- Combinações possíveis dos eventos iniciados por
  - TLB (miss / hit)
  - Page Table (miss / hit, miss ≡ Page Fault)
  - Cache (miss / hit)

TLB	Page Table	Cache	Possível?
miss	miss	miss	Sim! Os dados não estão em memória
miss	miss	hit	Não! A página não está em memória, hit na cache impossível
miss	hit	miss	Sim! A página está em memória, dados não disponíveis na cache
miss	hit	hit	Sim! A página está em memória, dados na cache
hit	miss	miss	Não! A página não está referenciada na "Page Table"
hit	miss	hit	Não! A página não está referenciada na "Page Table"
hit	hit	miss	Sim! Página referenciada na TLB, dados não disponíveis na cache
hit	hit	hit	Sim! Página referenciada na TLB, dados disponíveis na cache

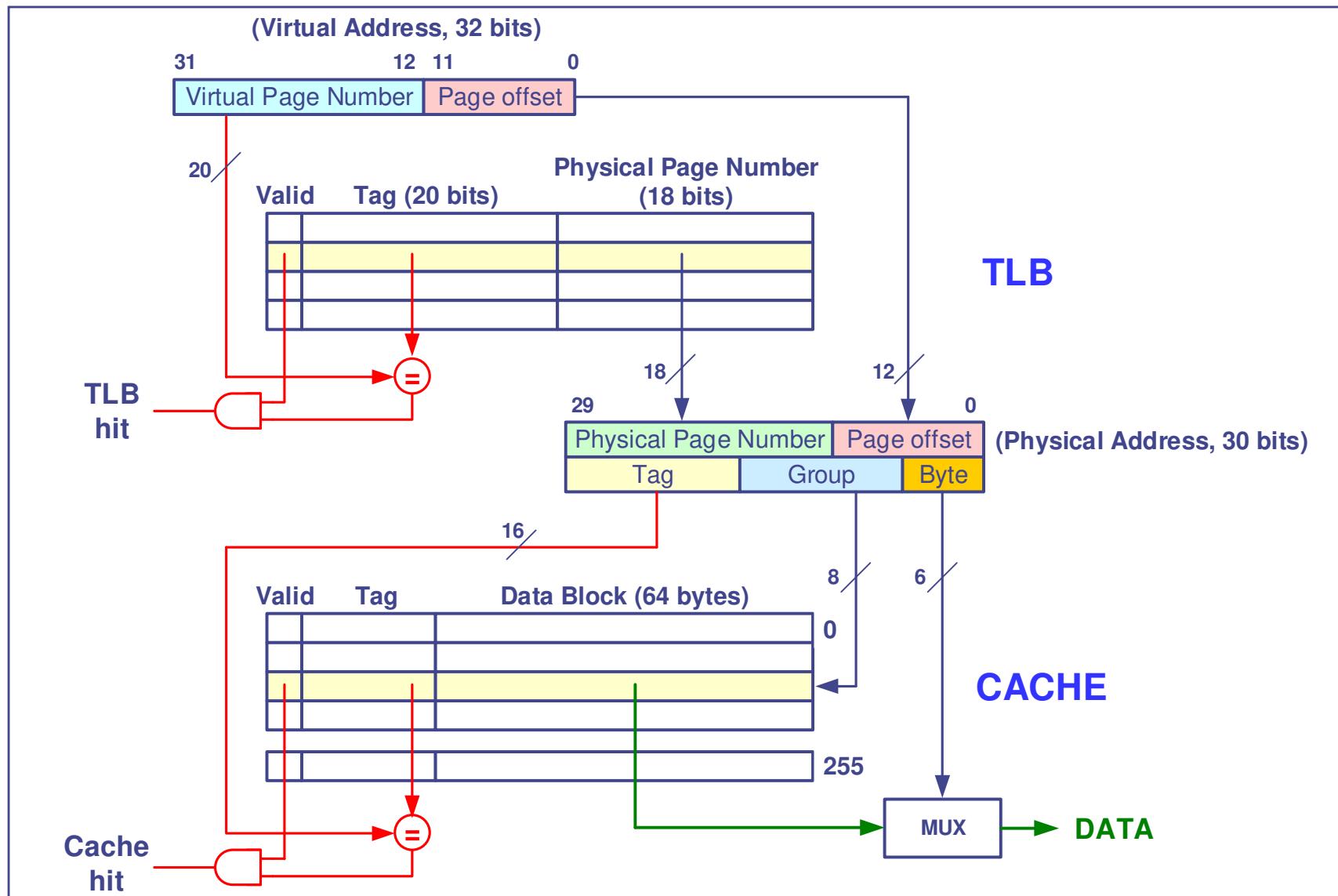
# Tradução de endereços com a TLB



# Tradução de endereços com a TLB - Exemplo

- Memória virtual:
  - Espaço de endereçamento virtual: 32 bits
  - Espaço de endereçamento físico: 30 bits
  - Dimensão da página: 4 KB (12 bits)
    - VPN – 20 bits
    - PPN – 18 bits
- Cache de 16 kBytes
  - Mapeamento direto
  - Nº de linhas: 256 (8 bits)
  - Dimensão do bloco 64 bytes (6 bits)

# Tradução de endereços com a TLB - Exemplo



## Em resumo...

- A memória virtual introduz uma indireção entre o endereço virtual gerado pelo CPU e o endereço físico da memória
- Isto permite
  - Mapear memória em disco (memória "ilimitada")
  - Gerir de forma eficiente a memória disponível e evitar a fragmentação
  - Impedir que um processo aceda à zona de memória de outro processo ou que escreva em zonas de memória atribuídas ao processo, mas onde não tem permissão de escrita (segurança)
- Cada acesso à memória envolve a tradução do endereço virtual para um endereço físico
  - A memória é organizada em páginas
  - Cada processo tem uma *Page Table*, localizada em memória, com o mapeamento entre páginas virtuais e páginas físicas e com os respetivos atributos de acesso
  - A tradução de endereços é acelerada através da TLB que contém um subconjunto das entradas da *Page Table*

# Exercício

- Complete a seguinte tabela, preenchendo as quadrículas em falta e substituindo o ? pelo valor adequado. Utilize as seguintes unidades: K =  $2^{10}$  (Kilo), M =  $2^{20}$  (Mega), G =  $2^{30}$  (Giga), T =  $2^{40}$  (Tera), P =  $2^{50}$  (Peta) ou E =  $2^{60}$  (Exa).

Virtual address size (n)	# Virtual addresses (N)	Maior endereço virtual (hexadecimal)
8	$2^8 = 256$	0xFF
	$2^? = 64\text{ K}$	
		0xFFFFFFFF
	$2^? = 256\text{ T}$	
64		

# Exercício

- Determine o número de entradas da *Page Table* (PTE) para as seguintes combinações de número de bits do espaço de endereçamento virtual ( $n$ ) e dimensão da página ( $P$ ):

<b>n</b>	<b>P</b>	<b># PTEs</b>
16	4 KB	
16	8 KB	
32	4 KB	
32	8 KB	
48	4 KB	

# Exercício

- Suponha um espaço de endereçamento virtual de 32 bits e um espaço de endereçamento físico de 24 bits:
  - determine o número de bits dos campos: VPN (virtual page number), VPO (virtual page offset), PPN (physical page number), PPO (physical page offset) para as dimensões de página P:

P	VPN	VPO	PPN	PPO	# virtual pages	# physical pages
1 KB						
2 KB						
4 KB						
8 KB						

- para cada dimensão de página, determine o número de páginas virtuais e físicas

# Exercício

- Considere um sistema de memória virtual com um espaço de endereçamento virtual de 26 bits, páginas de 512 bytes, em que cada entrada da "Page Table" está alinhada em endereços múltiplos de 2, tem 16 bits de dimensão, e está organizada do seguinte modo:

**Valid** bit, **Dirty** bit, **Read** flag, **Write** flag, **PPN**

- em quantas páginas está organizado o espaço de endereçamento virtual? Quantas entradas tem a "Page Table"?
- qual a dimensão do espaço de endereçamento físico?
- em quantas páginas está organizado o espaço de endereçamento físico?
- Suponha que o "Page Table register" do processo em execução tem o valor 0x1A0 e que no endereço 0x252 (e 0x253) está armazenado o valor 0xA26C
  - quais os atributos da página física referenciada por essa entrada da tabela? Onde reside a página física?
  - qual é o VPN representado nessa entrada da "Page Table"?
  - qual o endereço inicial e final da página física?
  - qual a gama de endereços virtuais que indexa esta entrada da "Page Table"?