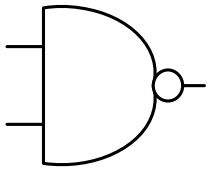
Aula 1

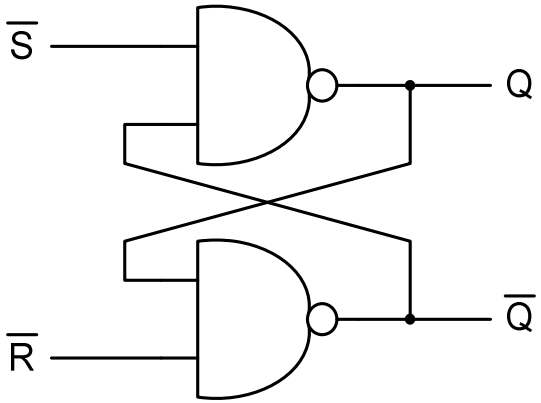
No princípio, era o verbo...



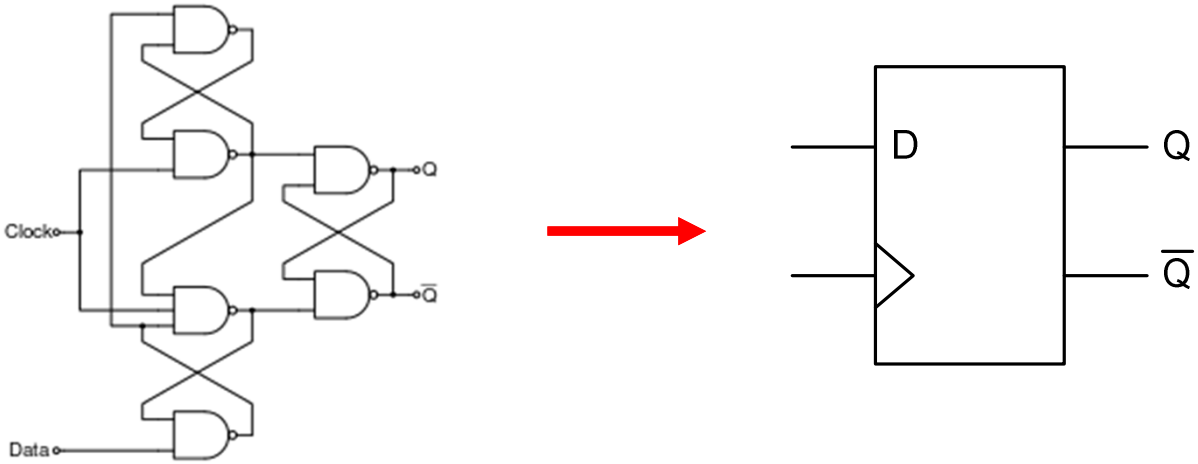
José Luís Azevedo, Arnaldo Oliveira, Tomás Silva, Bernardo Cunha

Era uma vez...

- E do verbo se fez...

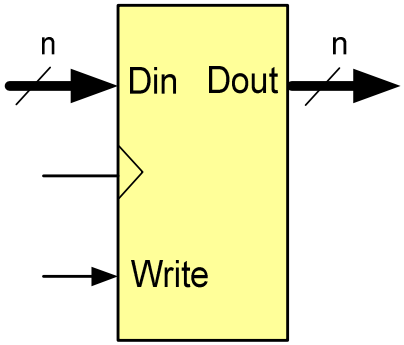


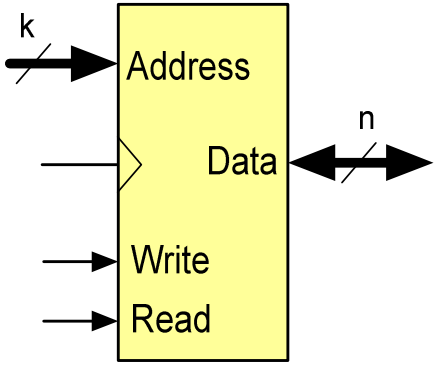
- E do Latch SR surgiu...



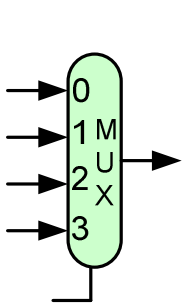
Era uma vez...

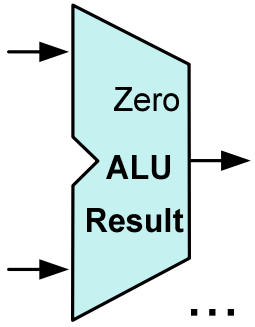
- E do FF tipo D construíram-se:





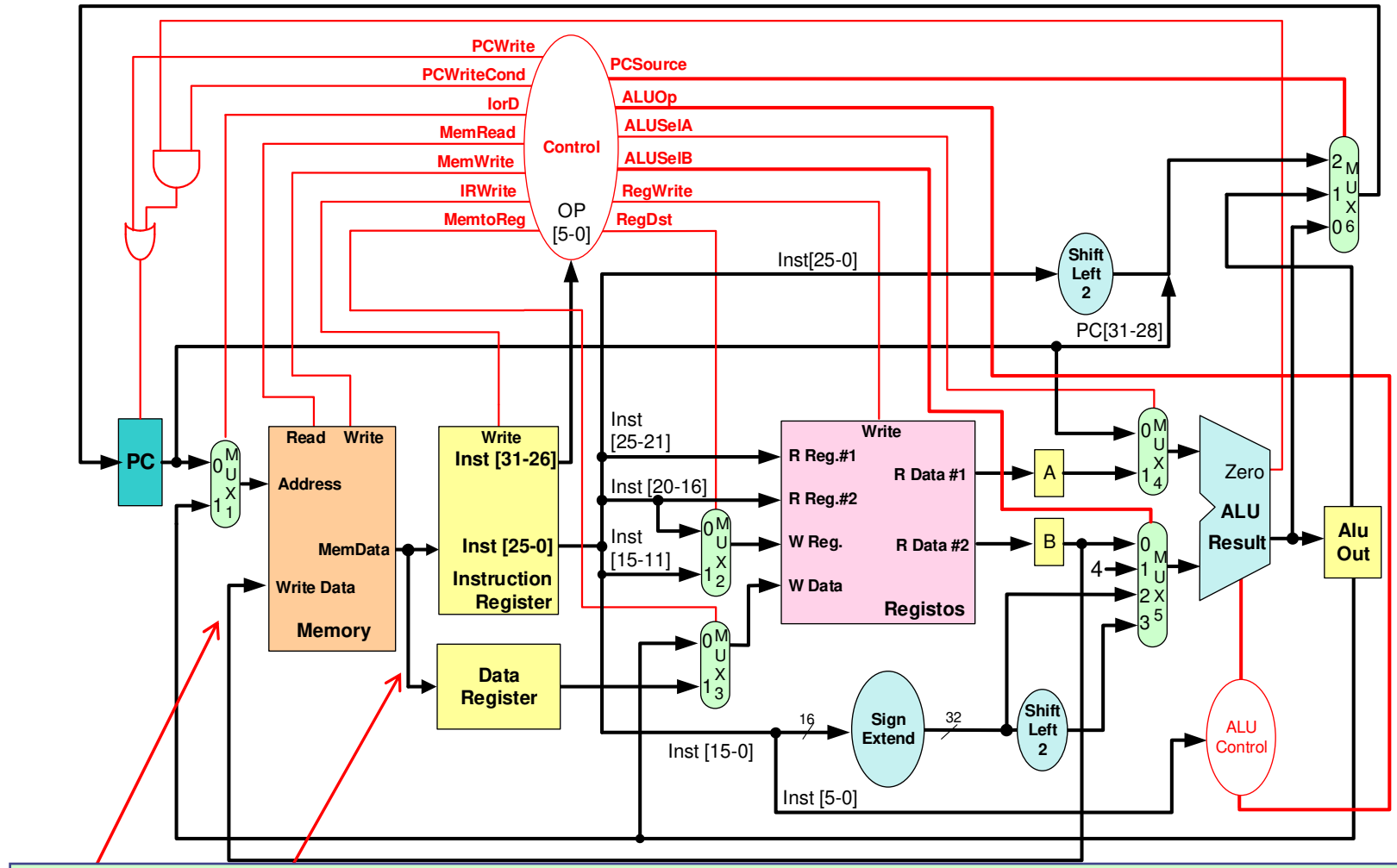
- E do verbo também nasceram:





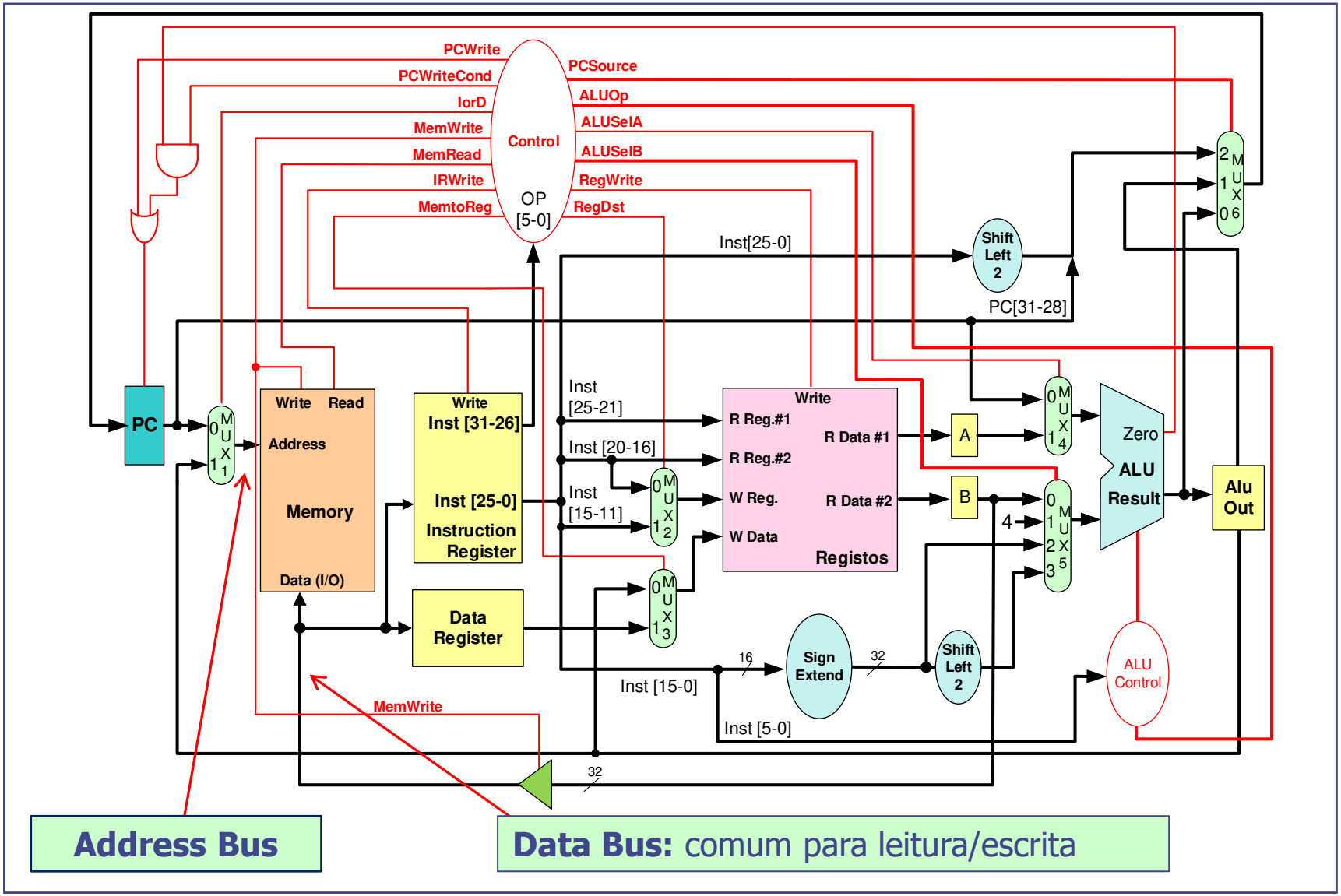
- E de tudo isto resultou…

Versão *multi-cycle* simplificada de uma arquitetura MIPS

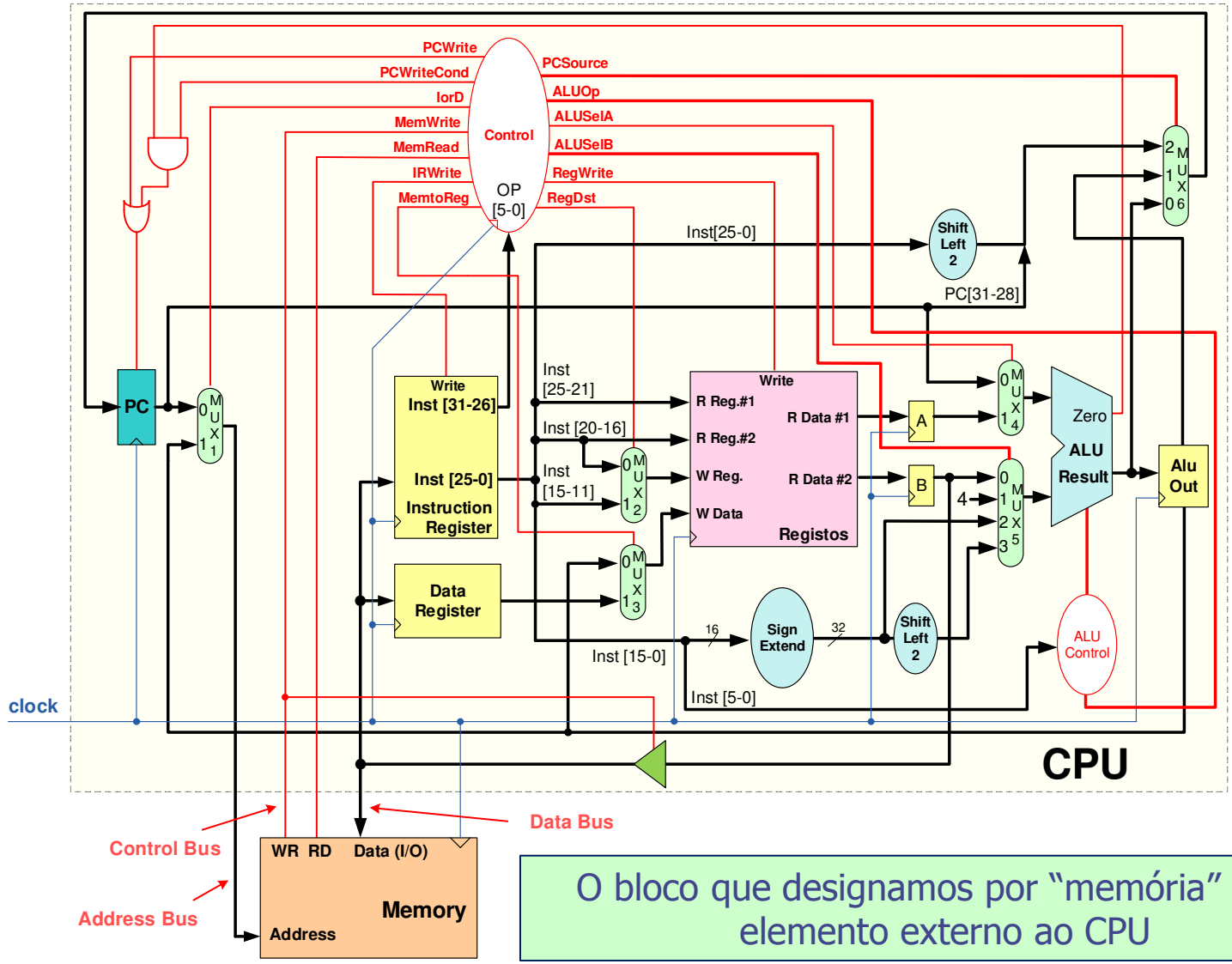


Memória com dois barramentos de dados separados: um para leitura (MemData), outro para escrita (WriteData)

Versão *multi-cycle* simplificada de uma arquitetura MIPS

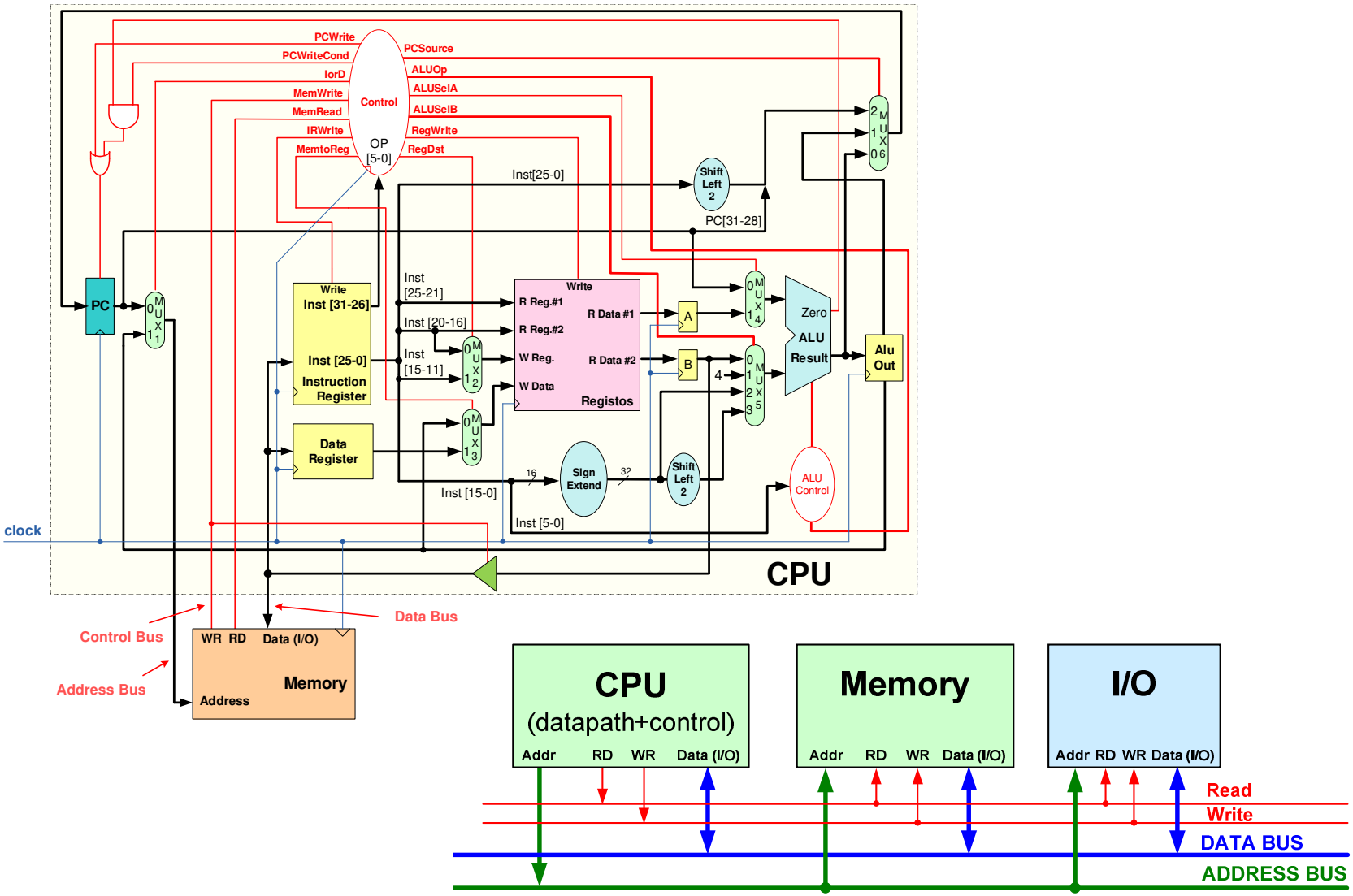


Versão *multi-cycle* simplificada de uma arquitetura MIPS

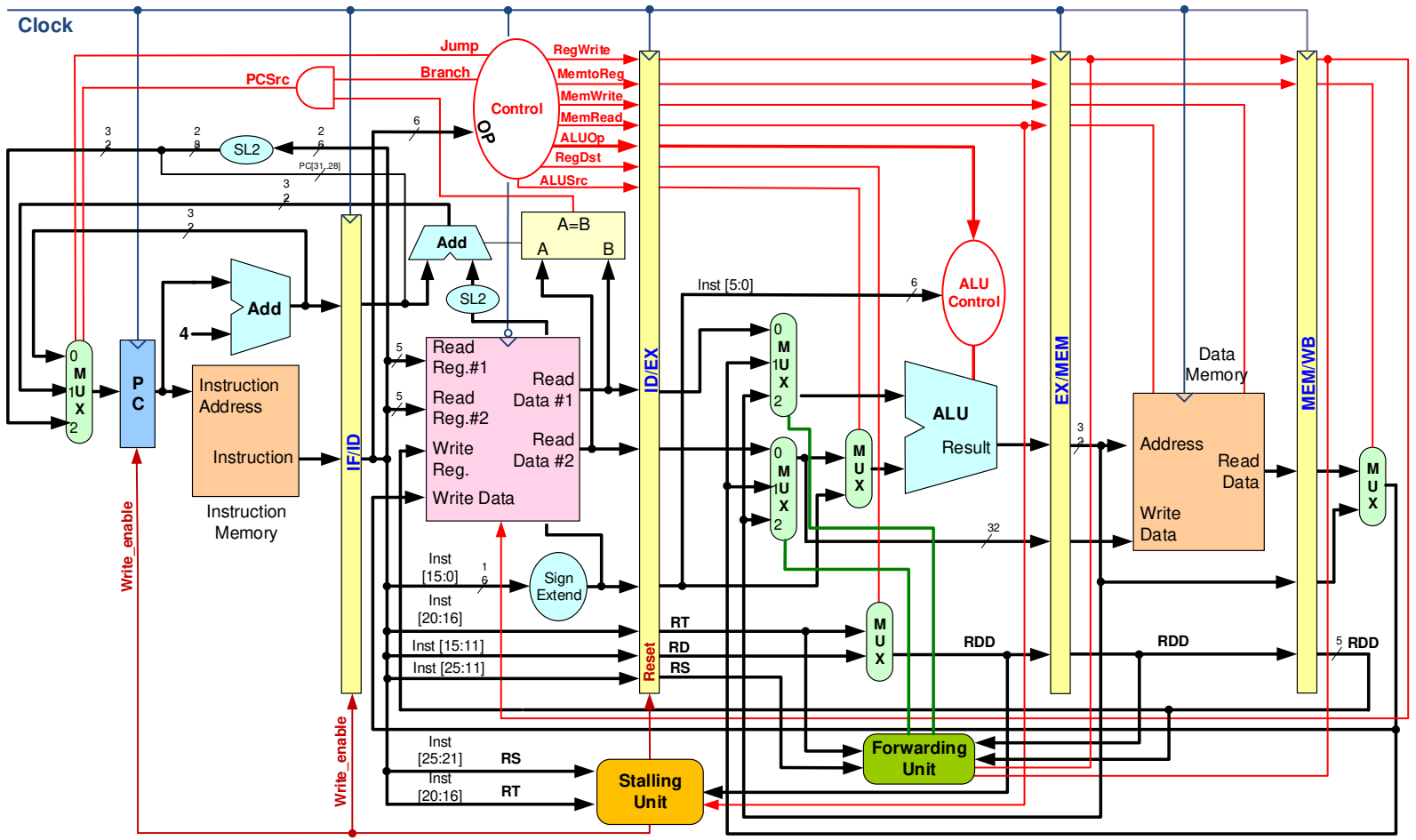


O bloco que designamos por "memória" é um elemento externo ao CPU

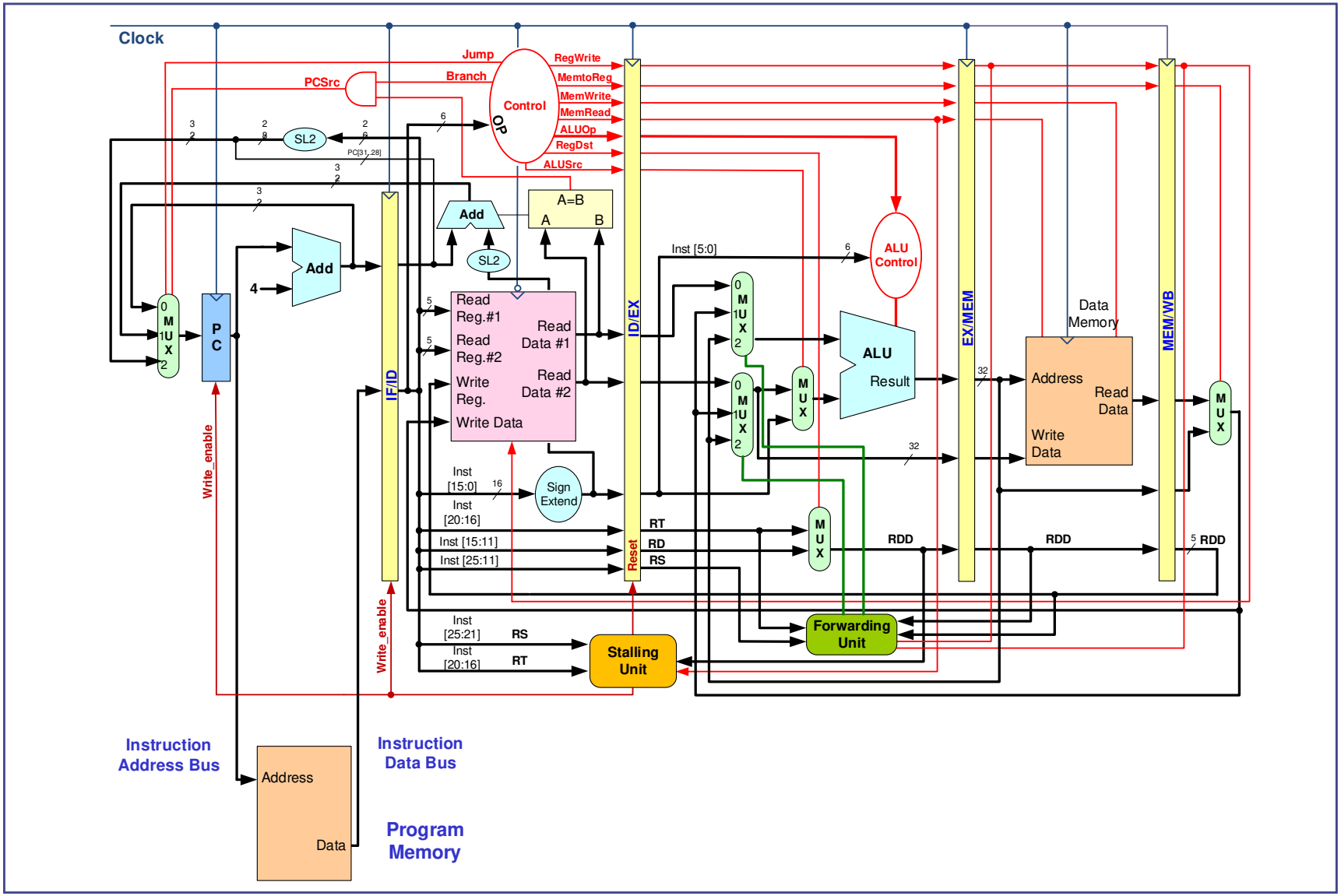
Arquitetura de *von-Neumann*



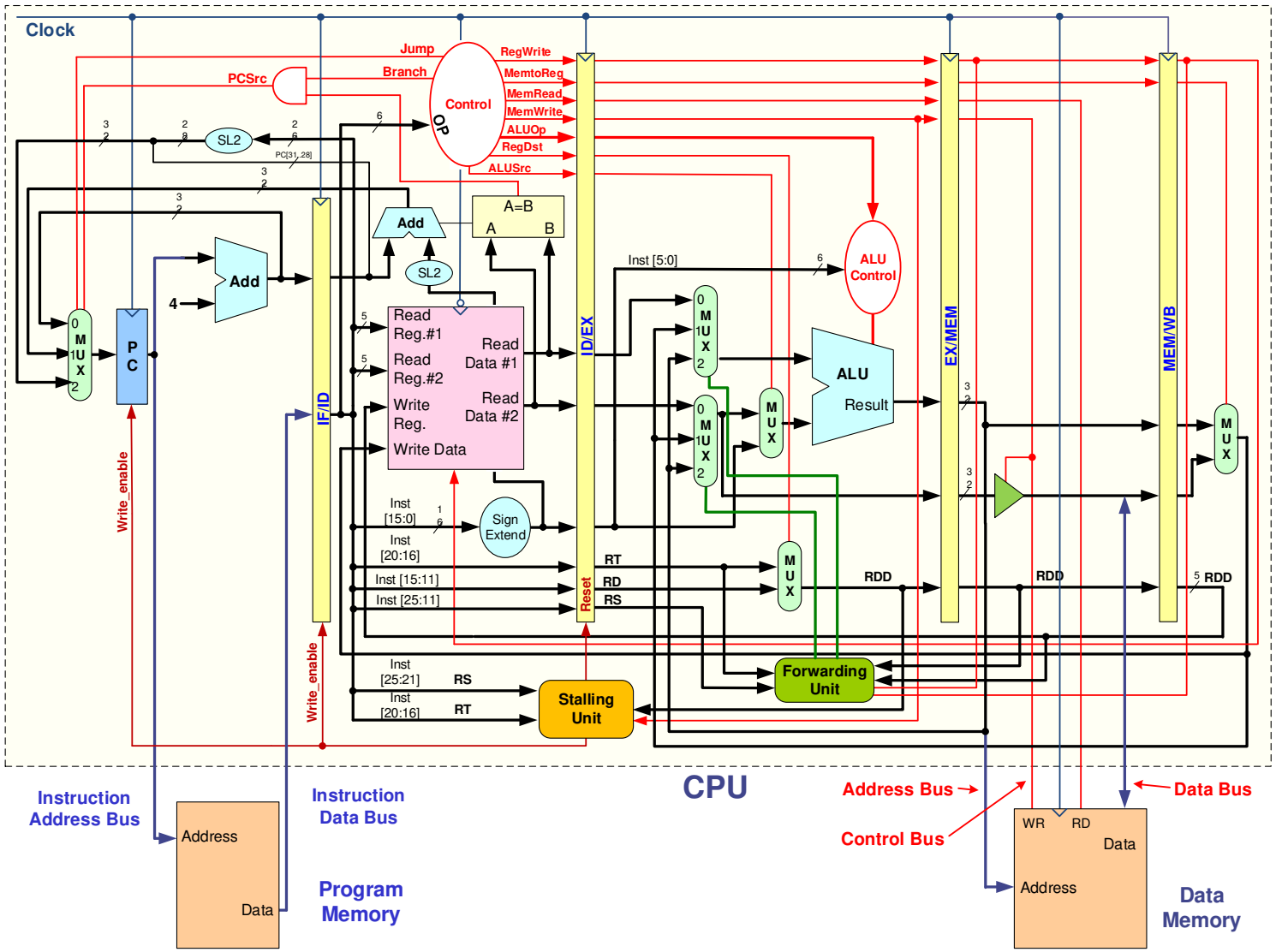
Versão simplificada de uma arquitetura MIPS *pipelined*



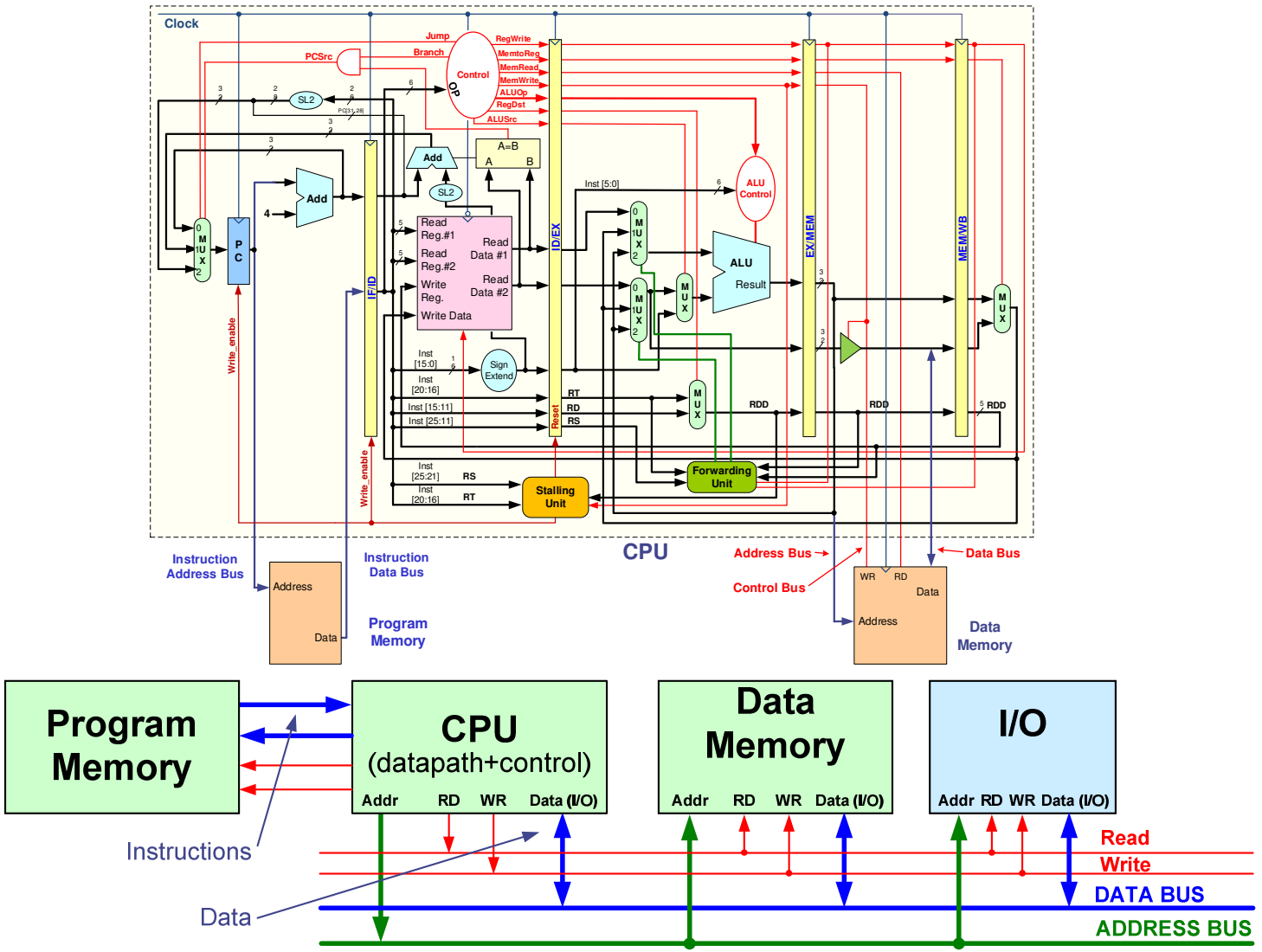
Versão simplificada de uma arquitetura MIPS *pipelined*



Versão simplificada de uma arquitetura MIPS *pipelined*



Arquitetura de Harvard



Aula 2

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

José Luís Azevedo, Arnaldo Oliveira, Tomás Silva, Bernardo Cunha

Microcontroladores *versus* Microprocessadores

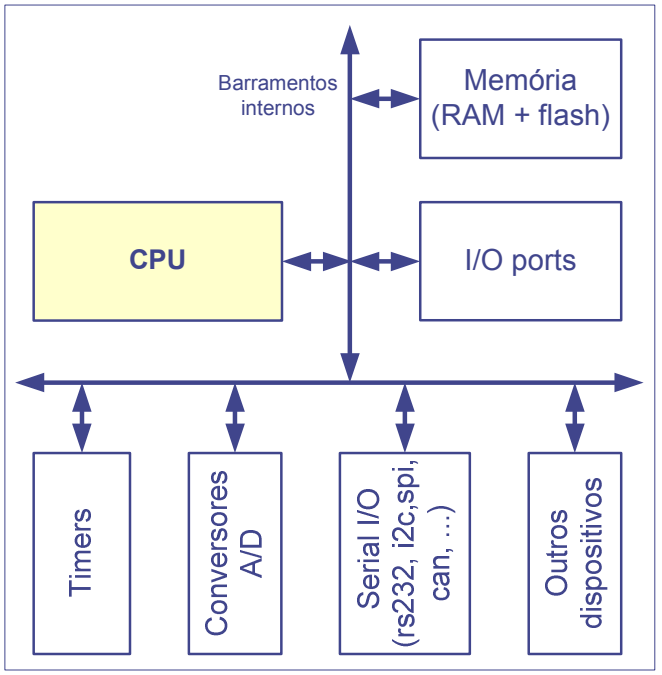
- Microprocessador:
  - Circuito integrado com um (ou mais) CPU
  - Não tem memória interna (além do banco de registos)
  - Os barramentos estão disponíveis no exterior
  - Para obter um sistema completo é necessário acrescentar RAM, ROM e periféricos
  - Pode operar a frequências elevadas (> 3GHz)
  - Sistemas computacionais de uso geral
- Microcontrolador:
  - Circuito integrado inclui CPU, RAM, ROM, periféricos
  - Frequência de funcionamento normalmente baixa (< 200 MHz)
  - 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 embebido

- 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 caraterísticas

- 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

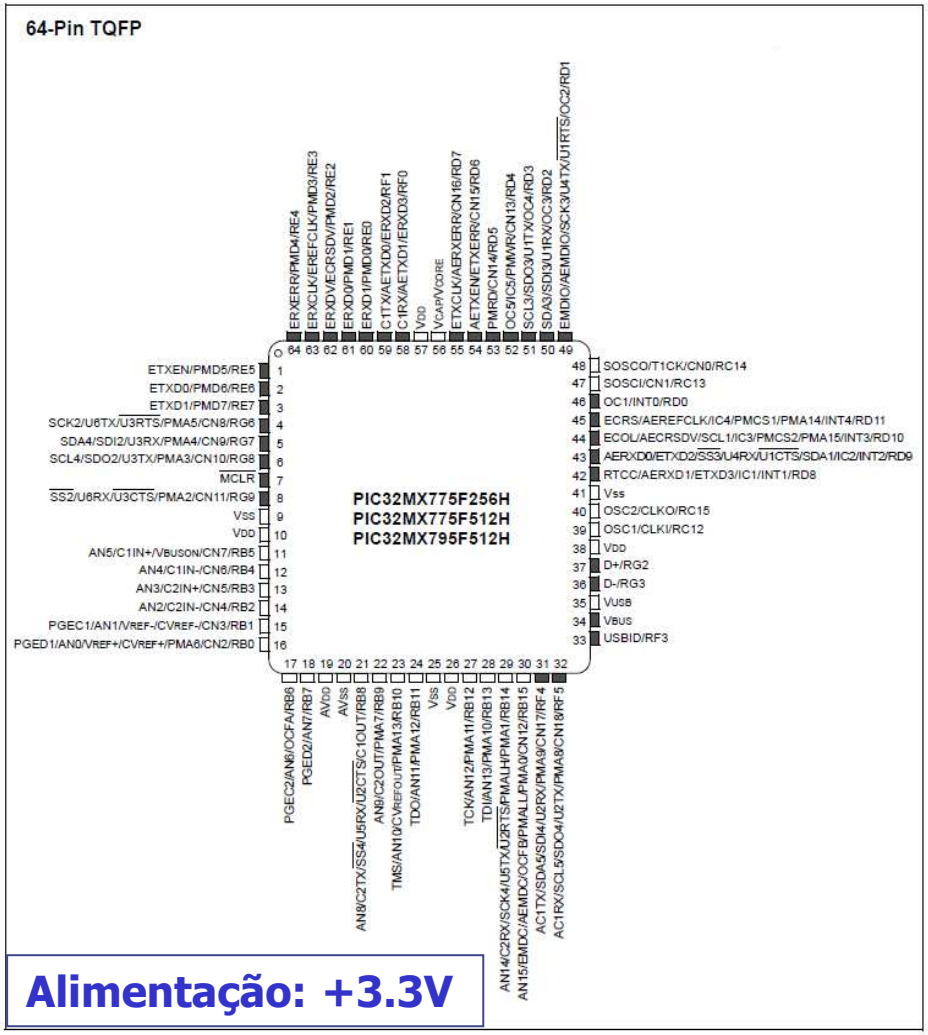
- Microcontrolador **PIC32MX795F512H**:
  - CPU MIPS
  - Conjunto alargado de periféricos
  - Memória flash: 512 kB (+12 kB Boot flash)
  - Memória RAM: 128 kB
  - Versão de 64 pinos (também disponível em 100 e 121 pinos)

- **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 detalhada em (link válido em 03/03/2021):

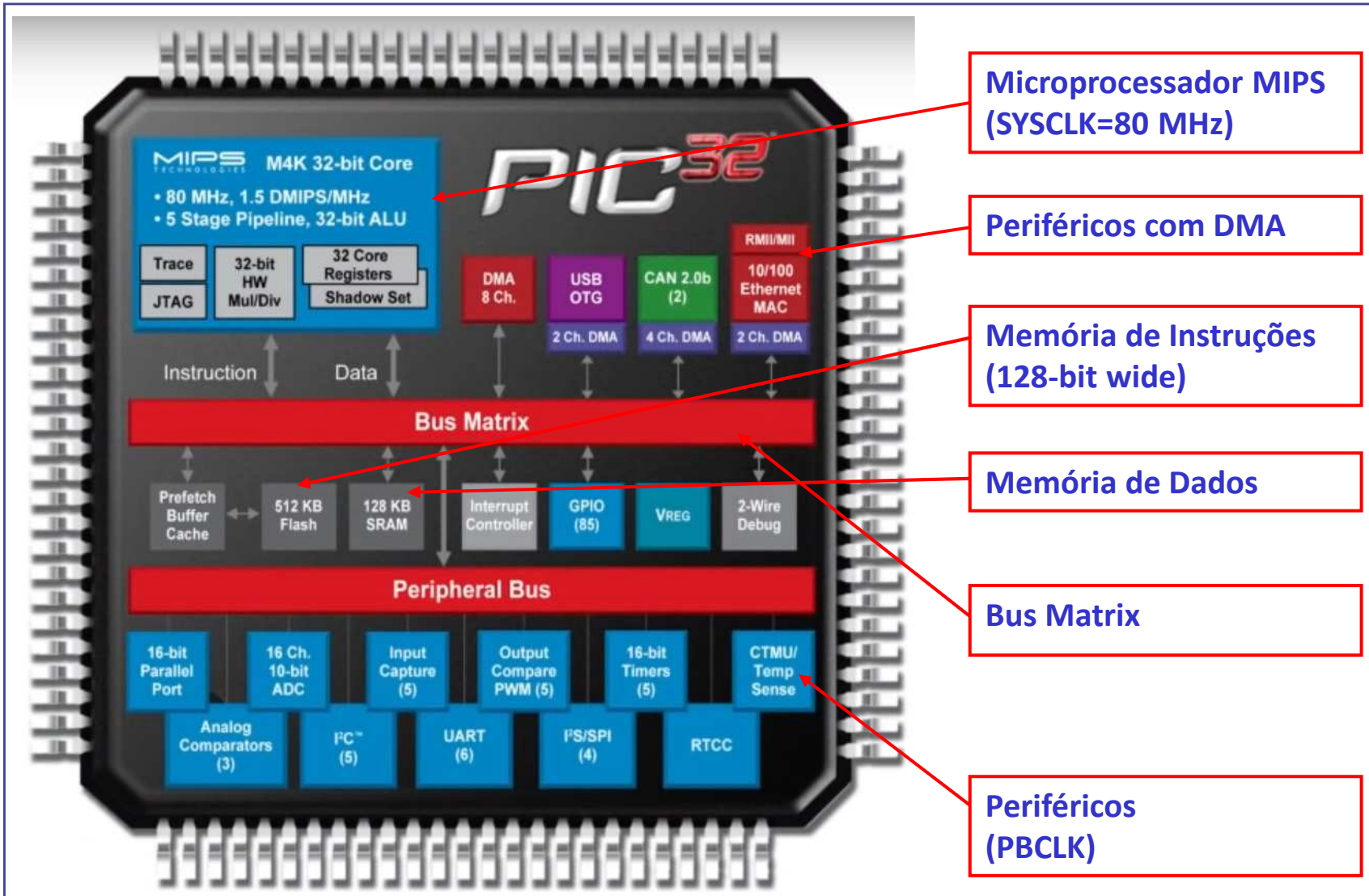
  http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en545655#1

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



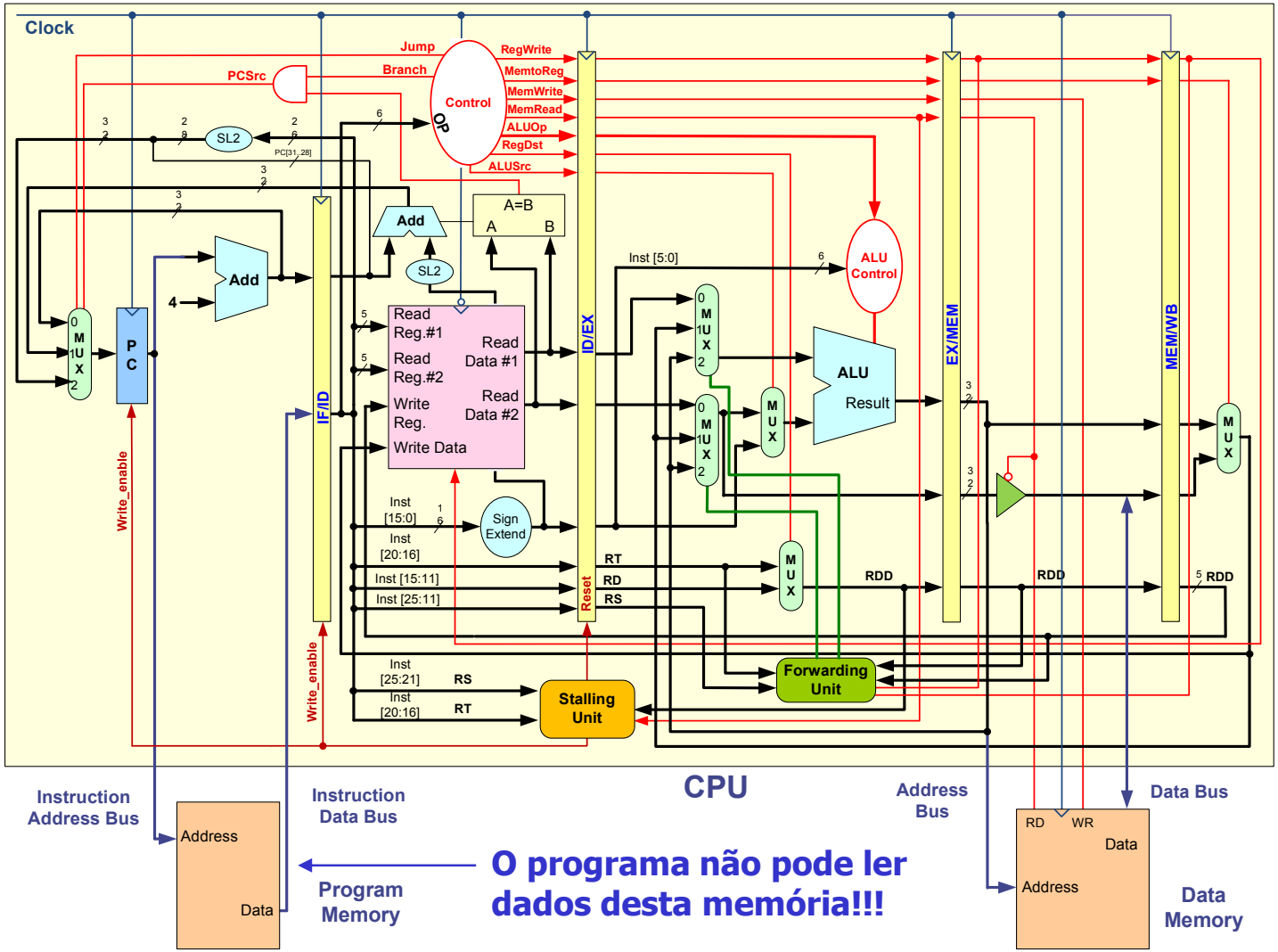
Microcontrolador PIC32MX795F512H



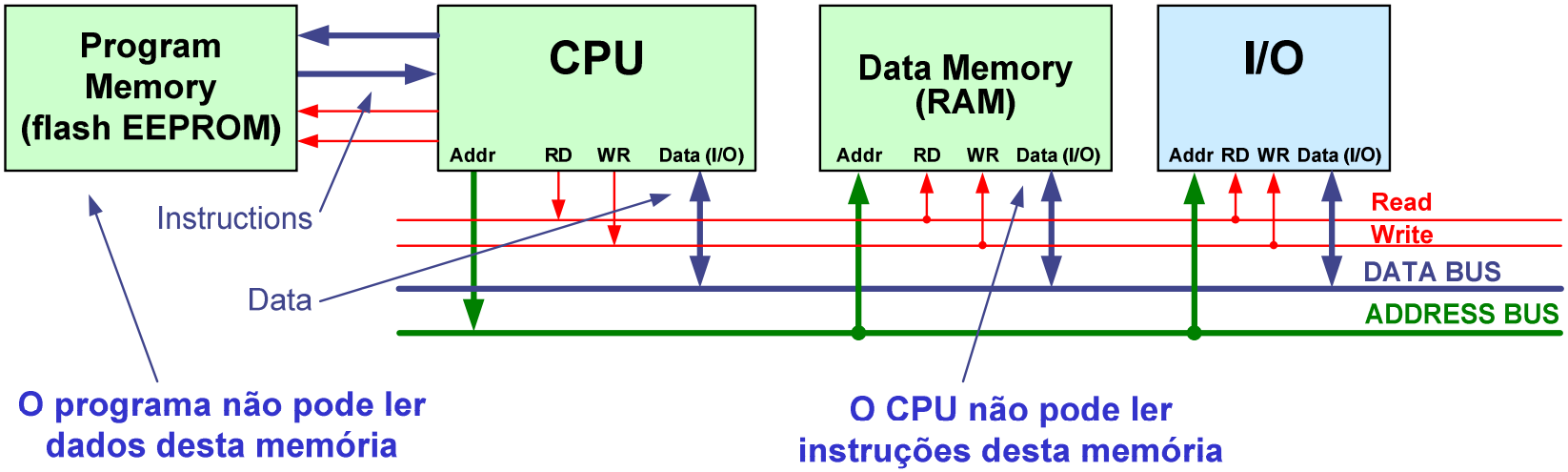
Microcontrolador PIC32MX795F512H

- 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*

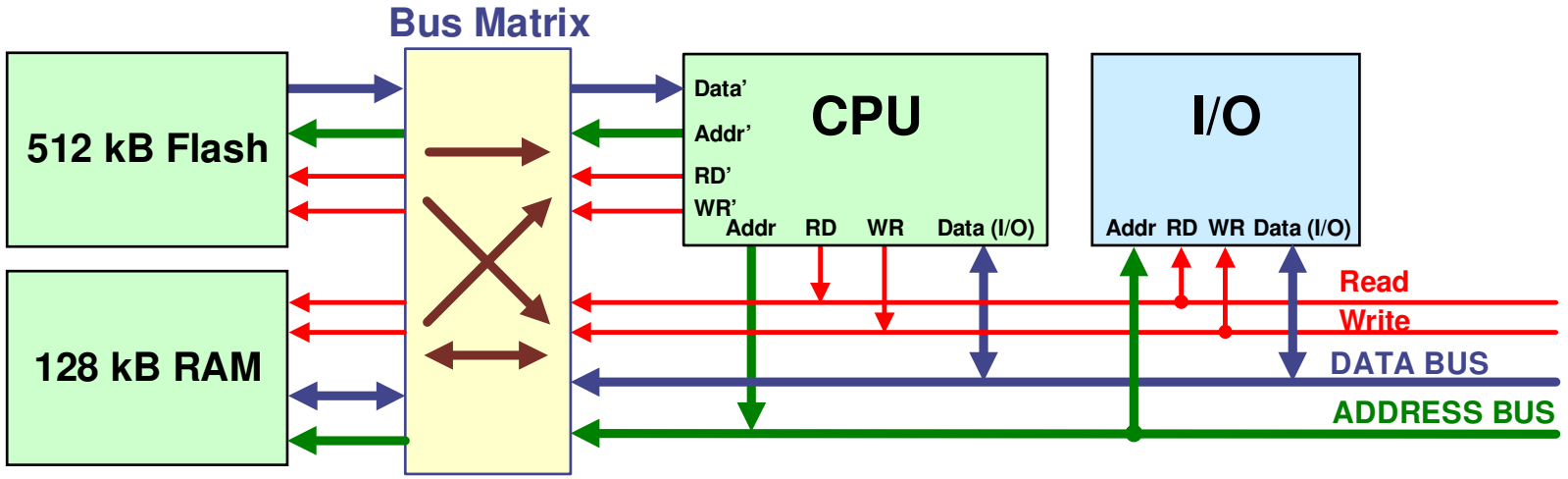


Arquitetura de Harvard convencional



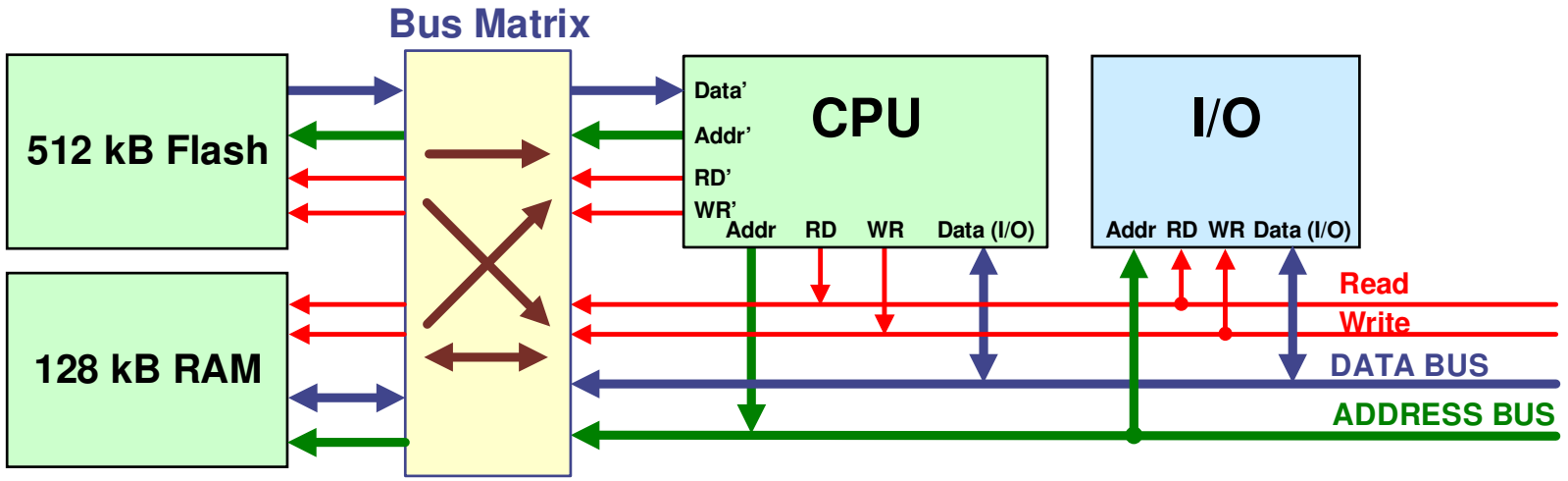
Microcontrolador PIC32MX795F512H

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



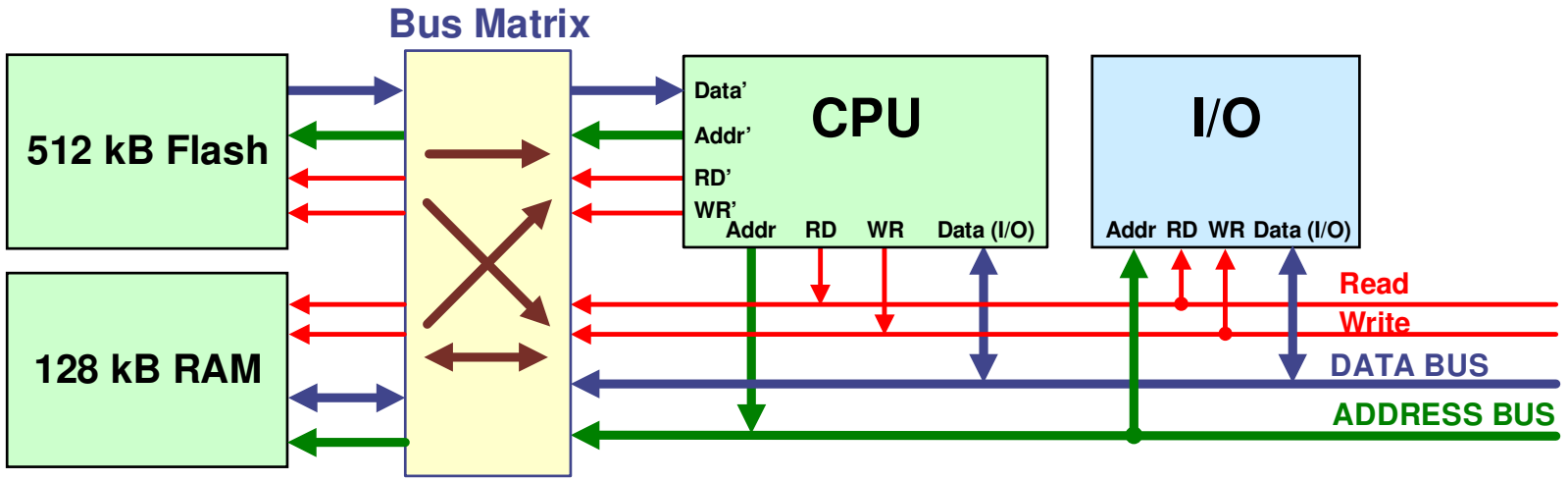
- O *Bus Matrix* é um comutador (*switch*) de alta velocidade que funciona à mesma frequência do CPU (SYSCLK)
- Estabelece ligações ponto a ponto entre os módulos do microcontrolador, em particular, entre o CPU e a memória RAM ou entre o CPU e a memória *Flash*
- O *peripheral bus* também liga ao *Bus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

Microcontrolador PIC32MX795F512H



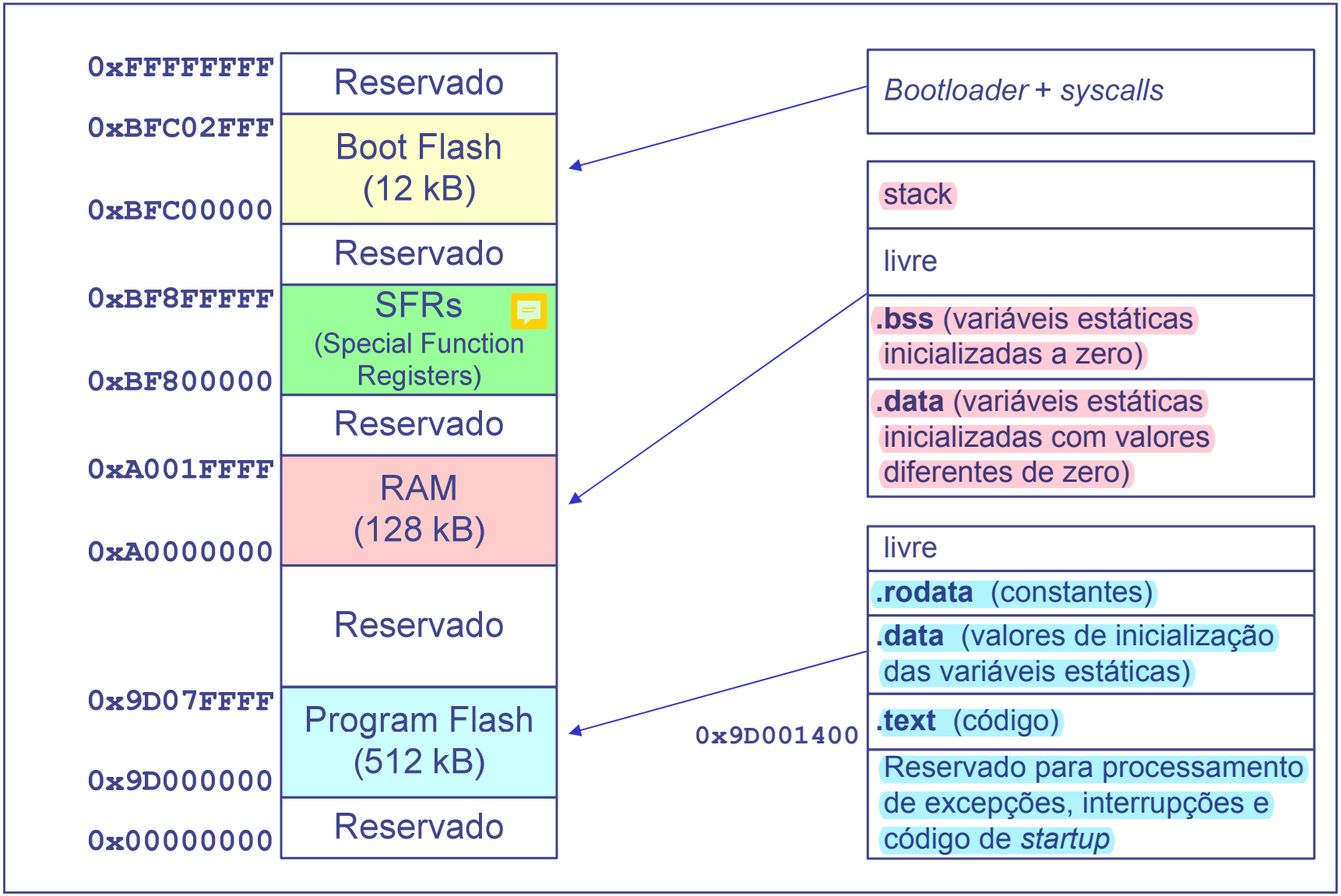
- Com o *Bus Matrix*, o espaço de endereçamento aparece, na visão do programador, como um espaço linear unificado (instruções e dados residem no mesmo espaço de endereçamento, cada um deles ocupando uma gama de endereços única)
- O CPU pode então executar programas que residem quer na *Flash* quer na 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

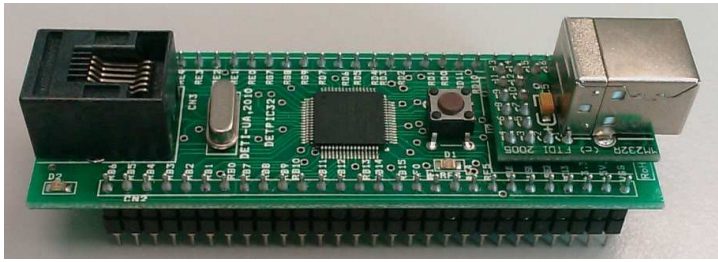


- Para o programador, o PIC32 comporta-se como uma arquitetura de *von Neumann*: um único espaço de endereçamento onde residem dados e instruções
- Para que o programa tenha acesso a qualquer zona da memória Flash (para leitura) ou da memória RAM (para leitura ou escrita), basta definir adequadamente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

Mapa de memória do PIC32 (versão DETPIC32)



Ferramentas de desenvolvimento DETPIC32



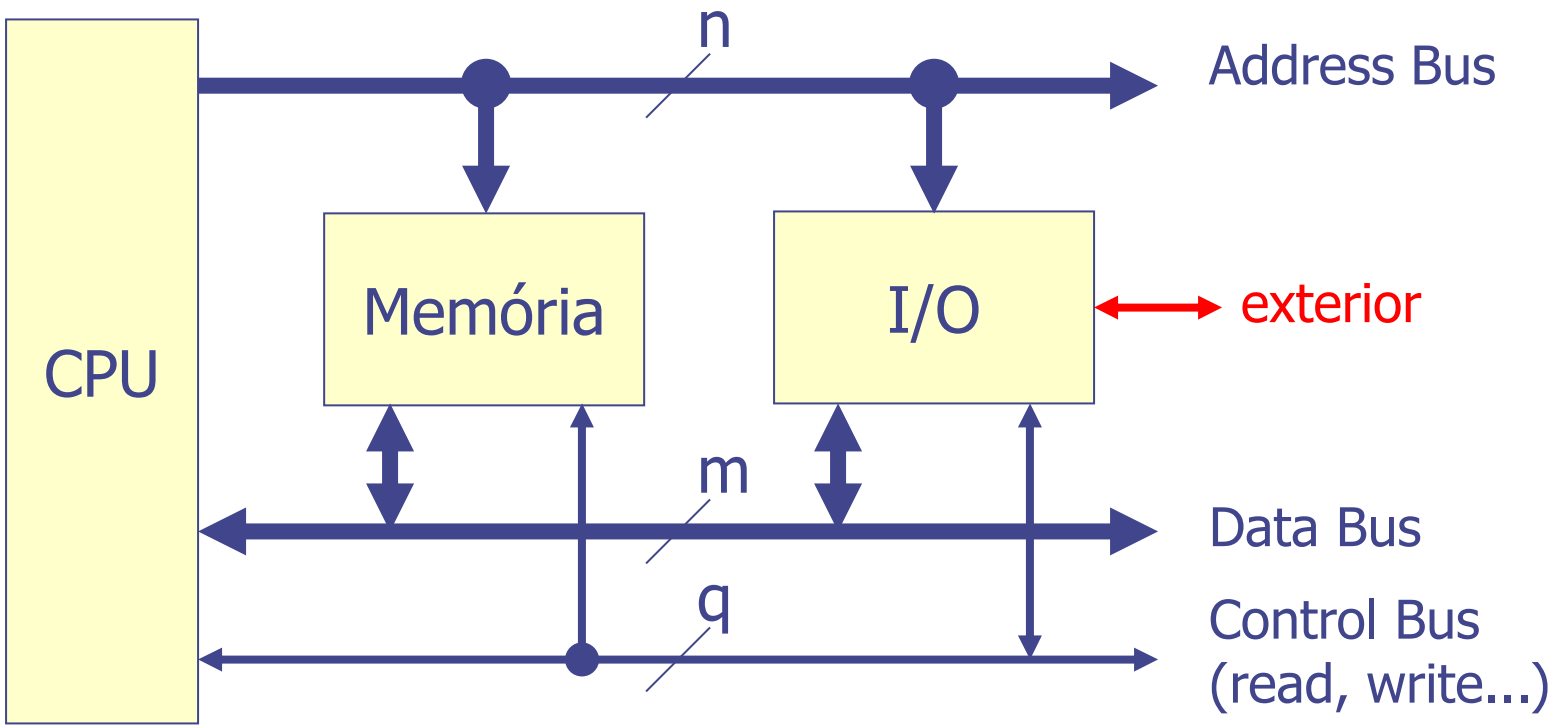
- Edição
  - GVim, gedit, geany…
- *Cross-compiler* | *cross-assembler*
  - **gcc** com *back-end* para MIPS (gcc-pic32)
  - Gera, entre outros, ficheiros ".hex" e ".map"
- Ferramentas desenvolvidas especificamente para DETPIC32
  - ***bootloader*** – programa previamente gravado na *boot flash* do PIC32; 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 *disassemble* do ficheiro ".hex" (utiliza o ficheiro ".map", para evidenciar secções / símbolos relevantes)

Aulas 3, 4 e 5

- 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
- Estrutura básica de um porto de I/O de 1 bit no microcontrolador PIC32. Estrutura dos portos de I/O de "n" bits.

José Luís Azevedo, Arnaldo Oliveira, Tomás Silva, Bernardo Cunha

Introdução



- O "sistema de entradas/saídas" providencia os mecanismos e os recursos para a comunicação entre o sistema computacional e o exterior
- Troca de informação com o exterior:
  - Processo é designado, genericamente, como Input/Output (I/O)
  - O dispositivo que assegura a comunicação designa-se por periférico

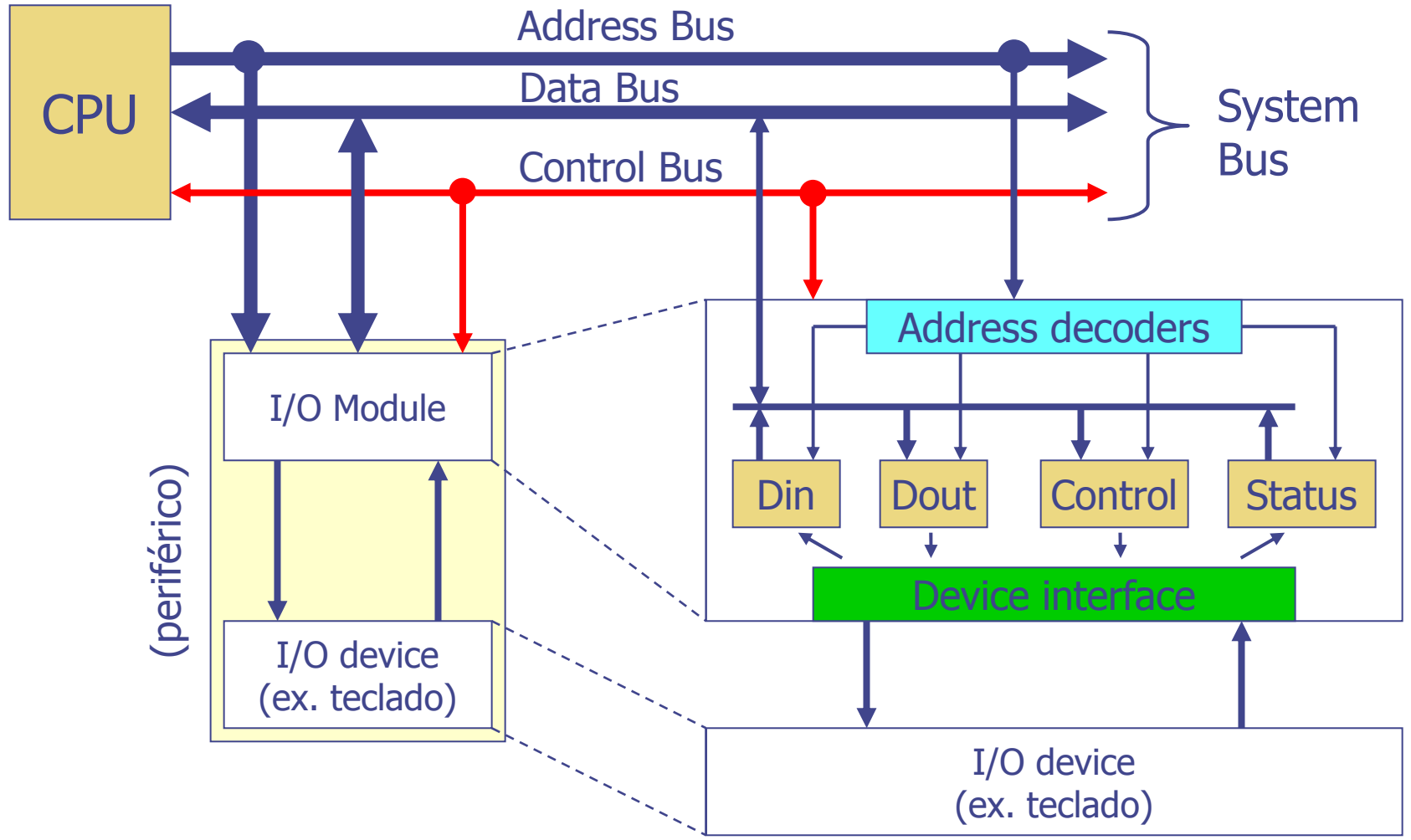
Introdução

- 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



- Dependendo da sua complexidade, o periférico tem atribuído um ou mais endereços

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)
- 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

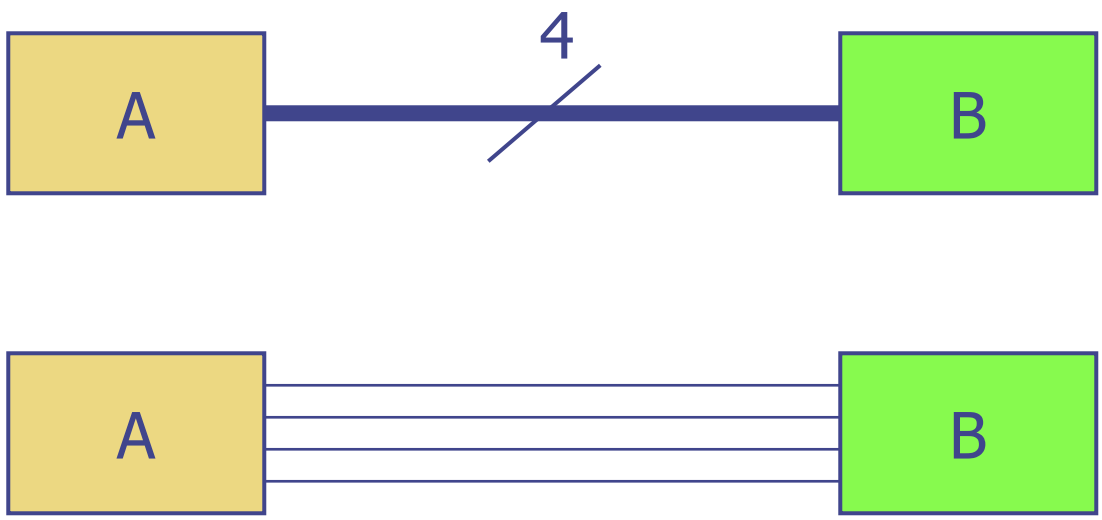
- A iniciativa da comunicação é sempre do CPU, no contexto da execução das instruções
- A comunicação entre o CPU e um dispositivo genérico envolve a existência de um **protocolo** que ambas as partes conhecem e respeitam
- Apenas duas operações podem ser efetuadas no sistema:
  - ***Write*** (fluxo de informação: CPU $\rightarrow$ dispositivo externo)
  - ***Read*** (fluxo de informação: CPU $\leftarrow$ 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

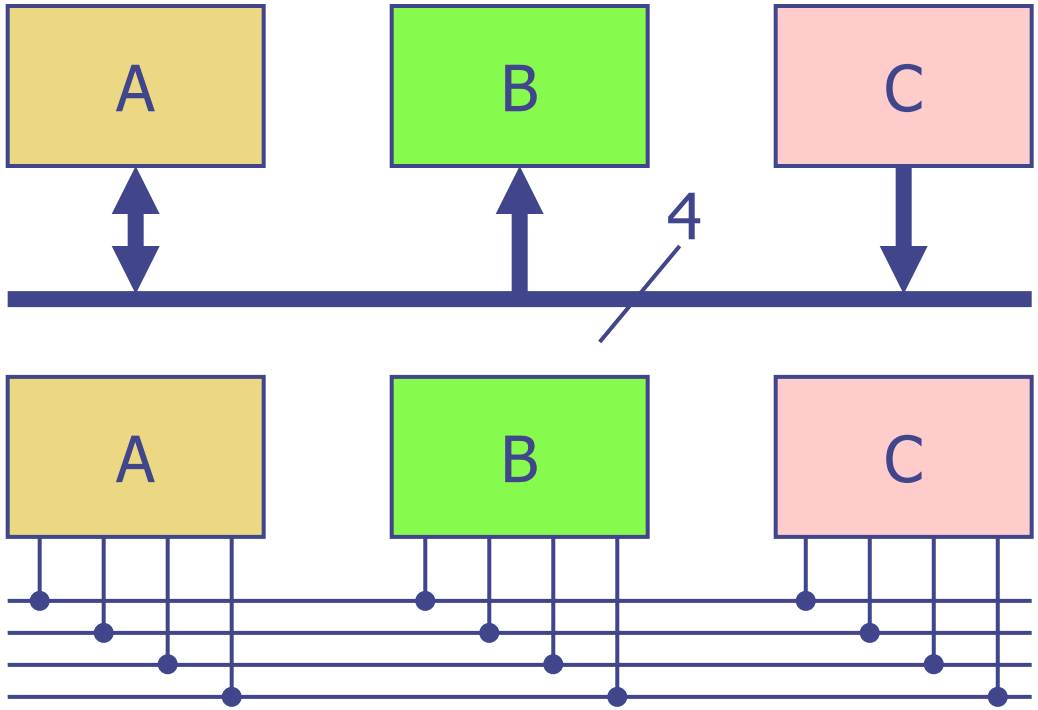
Barramento simples (revisão)

- 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)

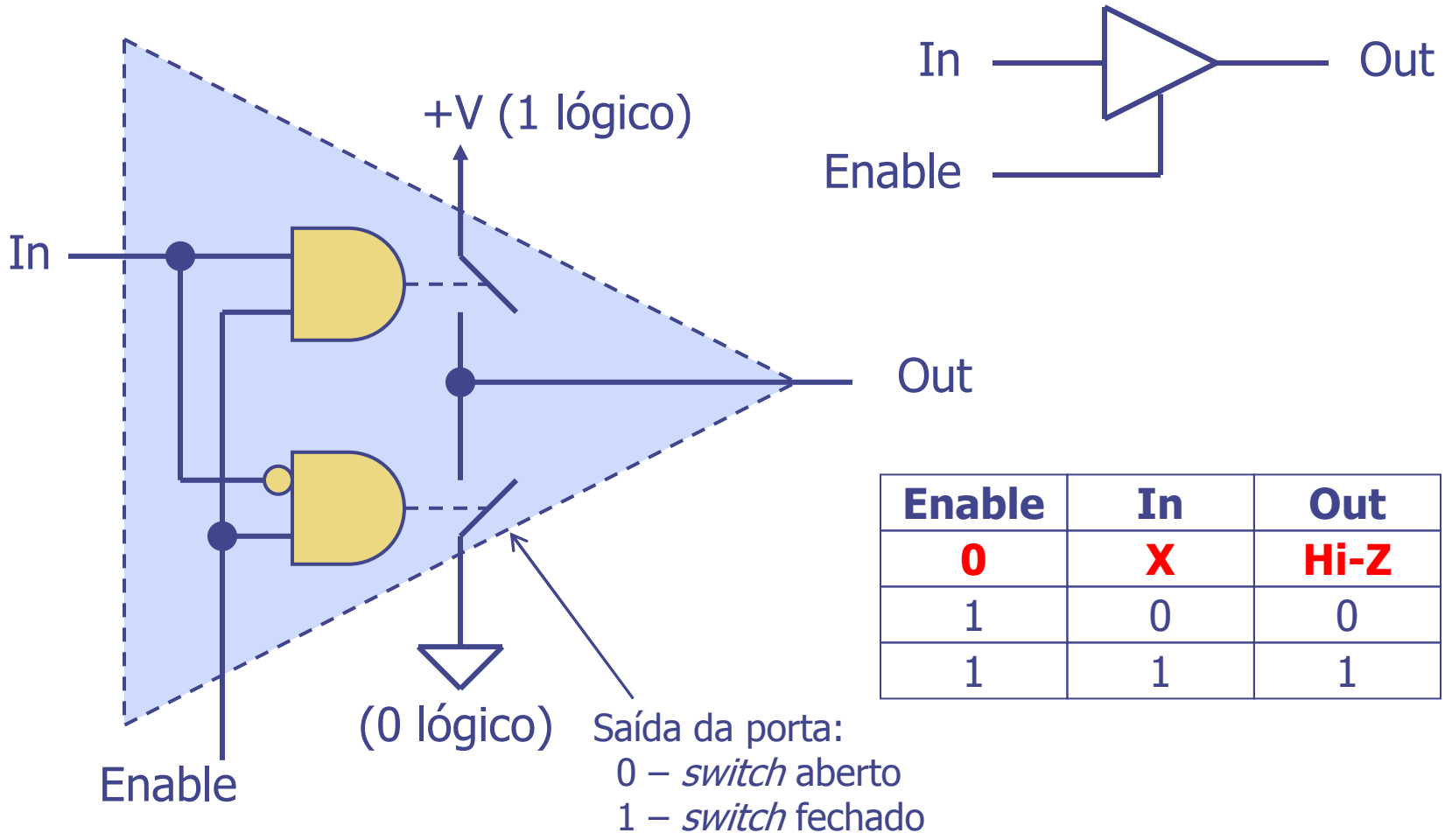
- Barramento partilhado (*shared bus*) - barramento que interliga vários dispositivos
- Exemplo: barramento de interligação entre os dispositivos A, B e C



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

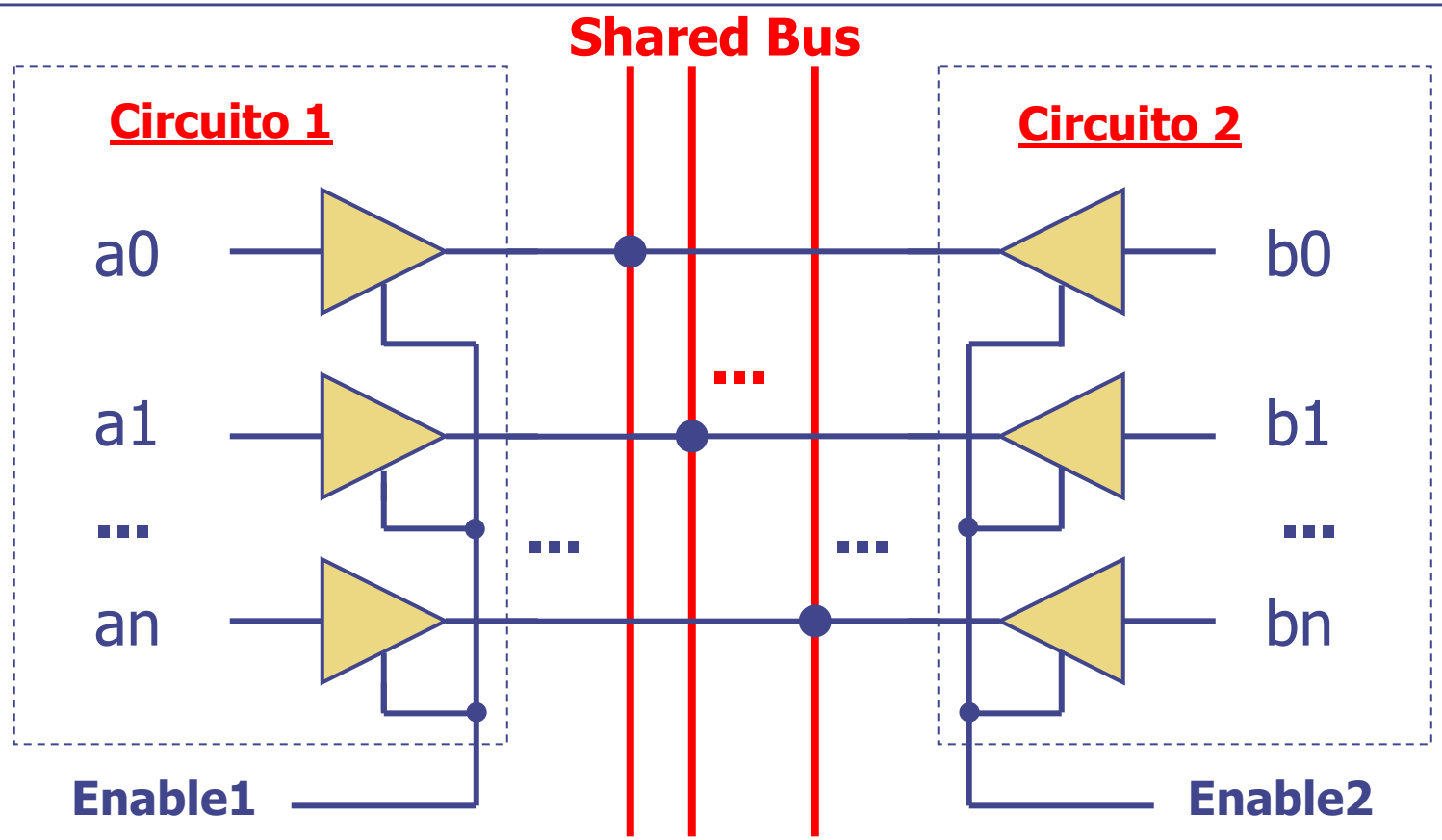
Porta *Tri-State*

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



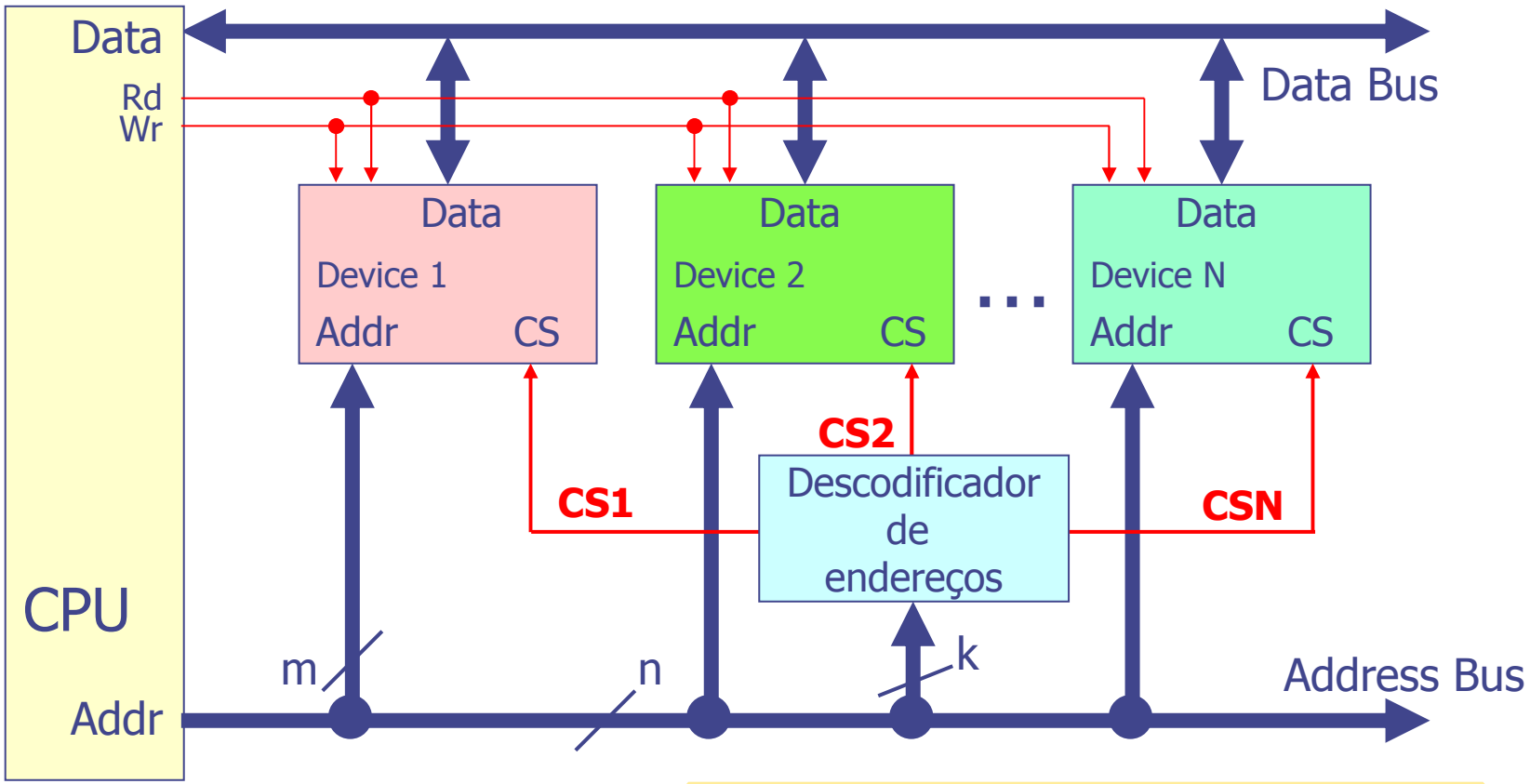
Enable	In	Out
0	**X**	**Hi-Z**
1	0	0
1	1	1

Ligação a um barramento partilhado



- Os dois sinais "**Enable**" nunca podem estar ativos simultaneamente (ou seja, os dois sinais têm que ser **mutuamente exclusivos**)
- Num sistema computacional, a estes dois circuitos teriam que ser atribuídos endereços distintos que dariam origem aos sinais **Enable1** e **Enable2**

Seleção do dispositivo externo



- **CSx = 0** -> Dispositivo inativo: linhas de dados em **alta impedância**, não é possível realizar qualquer operação de escrita ou de leitura
- **CSx = 1** -> Dispositivo ativo; pode ser realizada uma operação (rd/wr)
- Geração dos sinais de seleção (CSx): **descodificação de endereços**

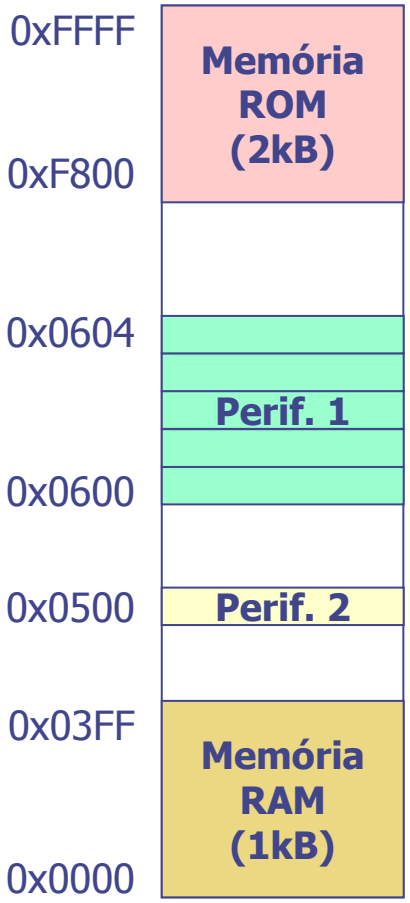
Seleção do dispositivo externo

- 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

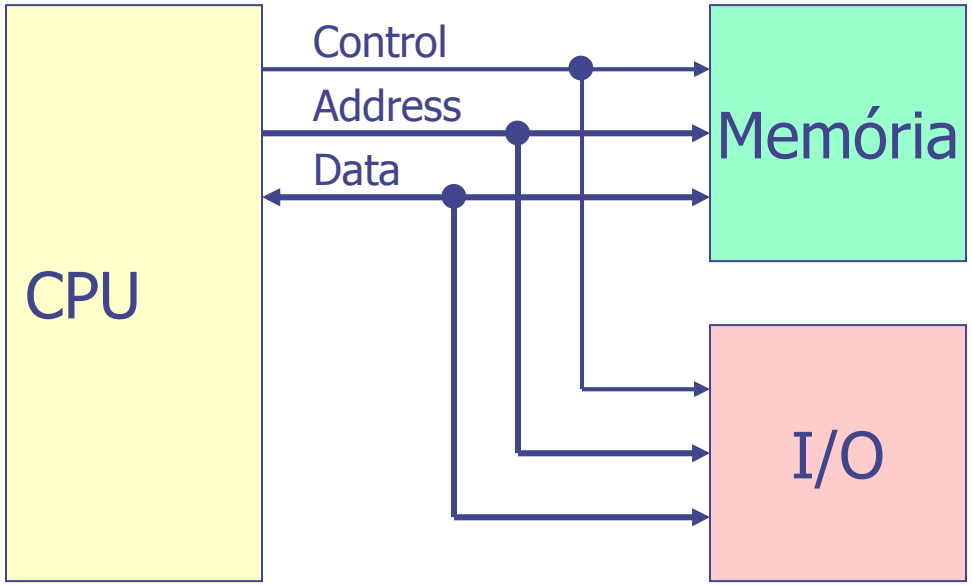
- Exemplo de mapeamento de dispositivos, considerando um espaço de endereçamento de 16 bits (2^{16} = 64k, A_{15}-A_0), e uma organização do tipo *byte-addressable*:
  - memória RAM de 1k x 8 (1 kB), memória ROM de 2k x 8 (2 kB), periférico com 5 registos, periférico com 1 registo
- Possível mapa de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

Dispositivo	Dimensão (bytes)	Endereço Inicial	Endereço Final	Nº bits do *address bus*
RAM, 1k x 8	1024	0x0000	0x03FF	10
ROM, 2k x 8	2048	0xF800	0xFFFF	11
Periférico 1	5	0x0600	0x0604	3
Periférico 2	1	0x0500	0x0500	0



Endereçamento das unidades de I/O

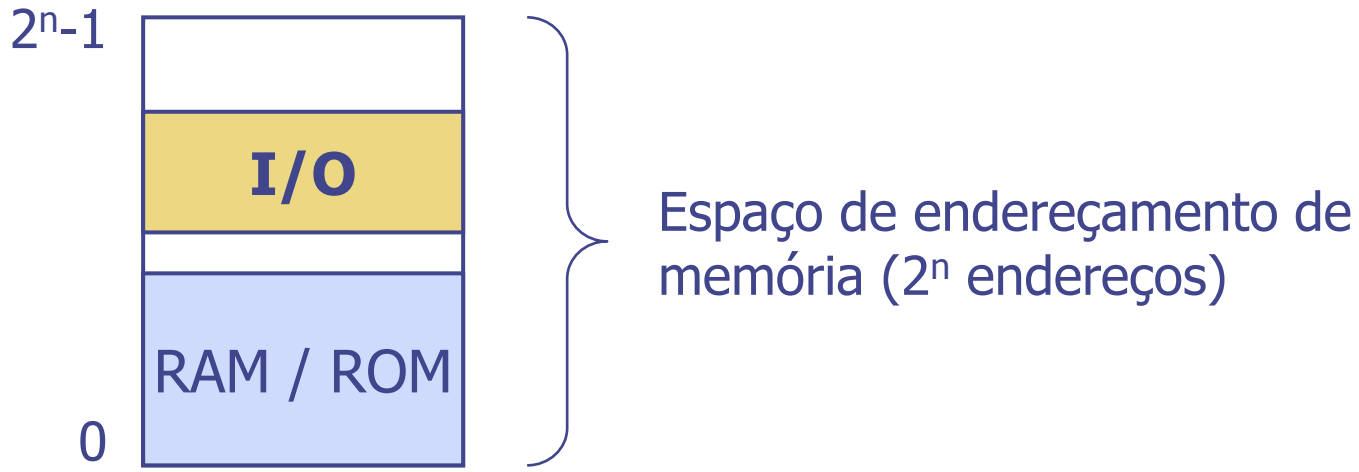
- *Memory-mapped* I/O



- 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

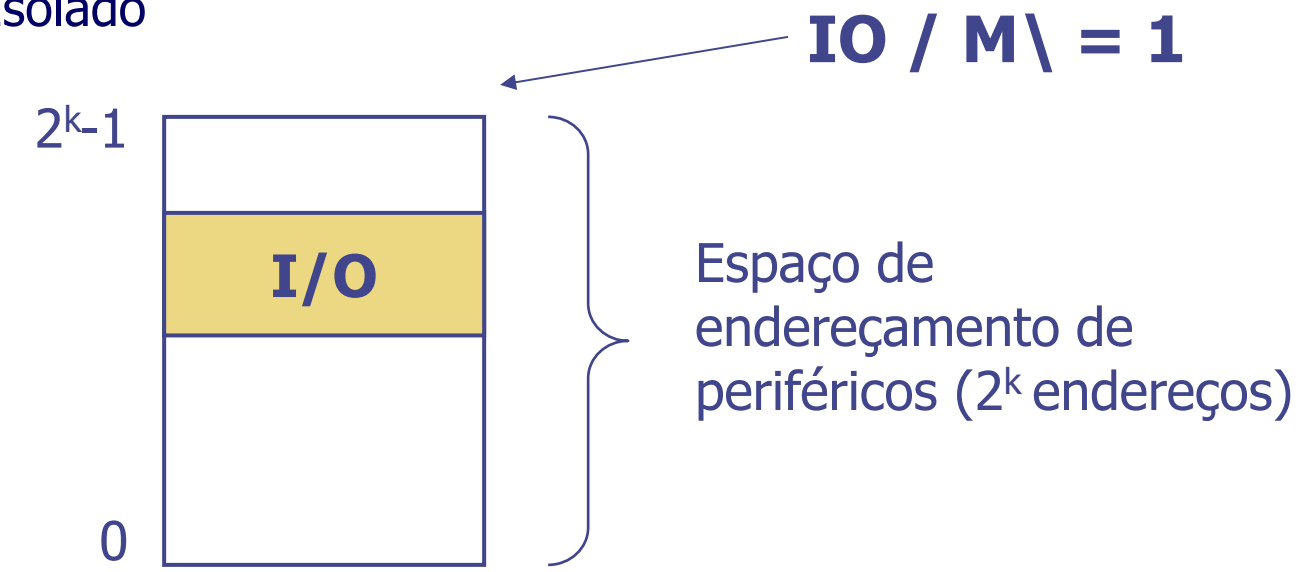
- *Memory-mapped* I/O



- Às 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 acede à memória (**`lw`** e **`sw`** no caso do MIPS)

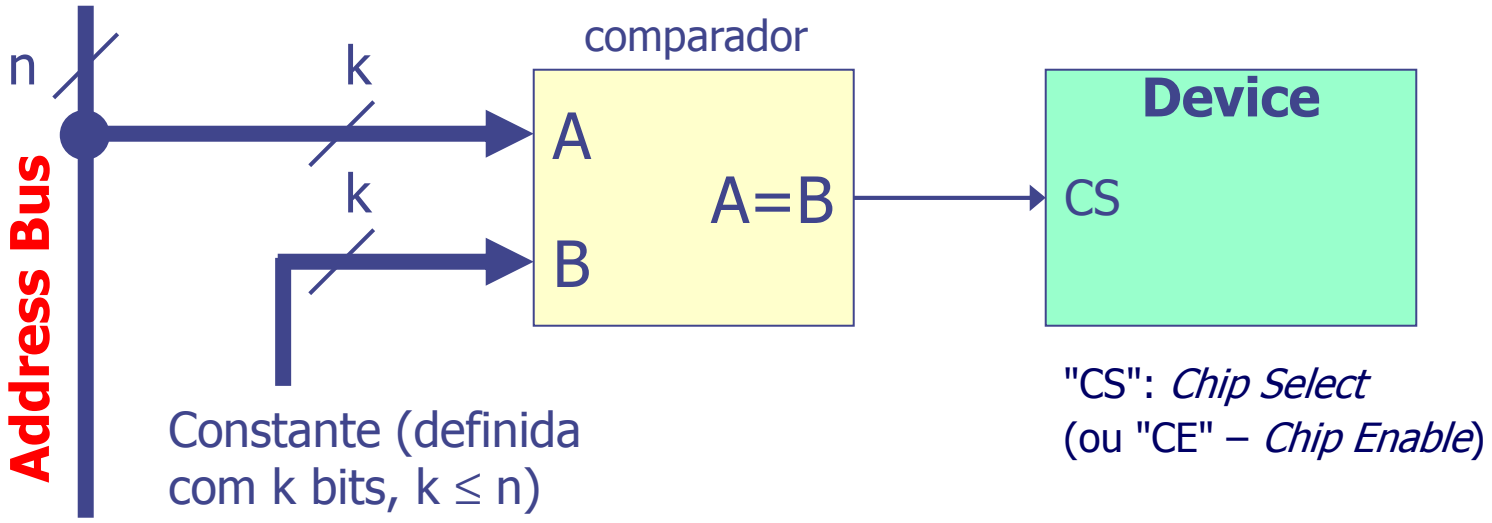
Endereçamento das unidades de I/O

- I/O Isolado



- 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\:
  - IO/M\=1 -> acesso ao espaço de endereçamento de I/O
  - IO/M\=0 -> 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, 0x8FFF]

Descodificação de endereços

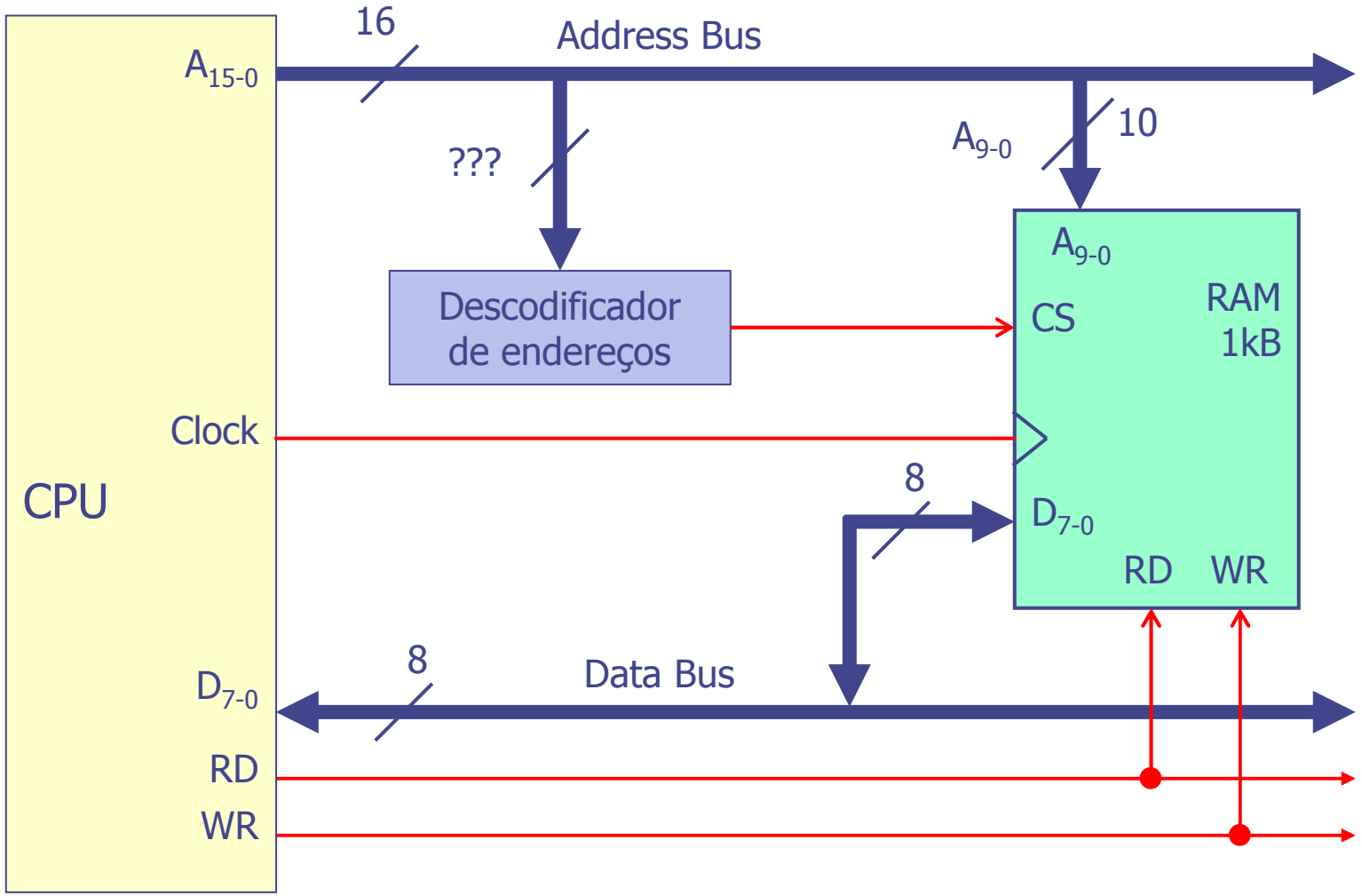
- Supondo um CPU com um espaço de endereçamento de 16 bits, memória *byte-addressable*, e um barramento de dados de 8 bits
  - 16 bits (A_{15}-A_0) –> (2^{16}=64 k)
  - 8 bits (D_7-D_0)

- **Exemplo 1**: ligação de uma memória RAM de 1 kByte ao CPU
  - 1 kByte (1k x 8) - 10 bits de endereço (2^{10}=1k)

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

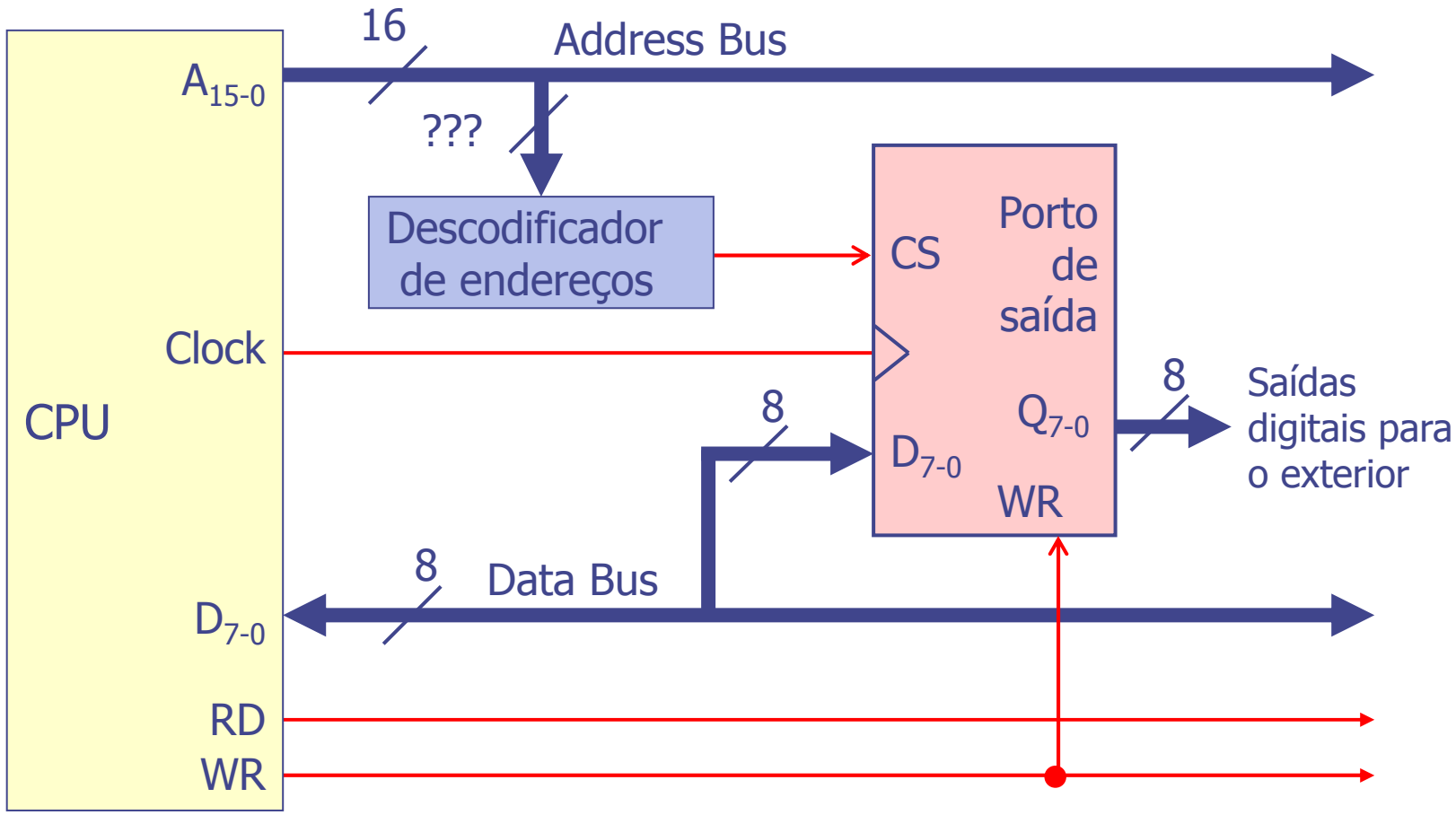
Descodificação de endereços

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



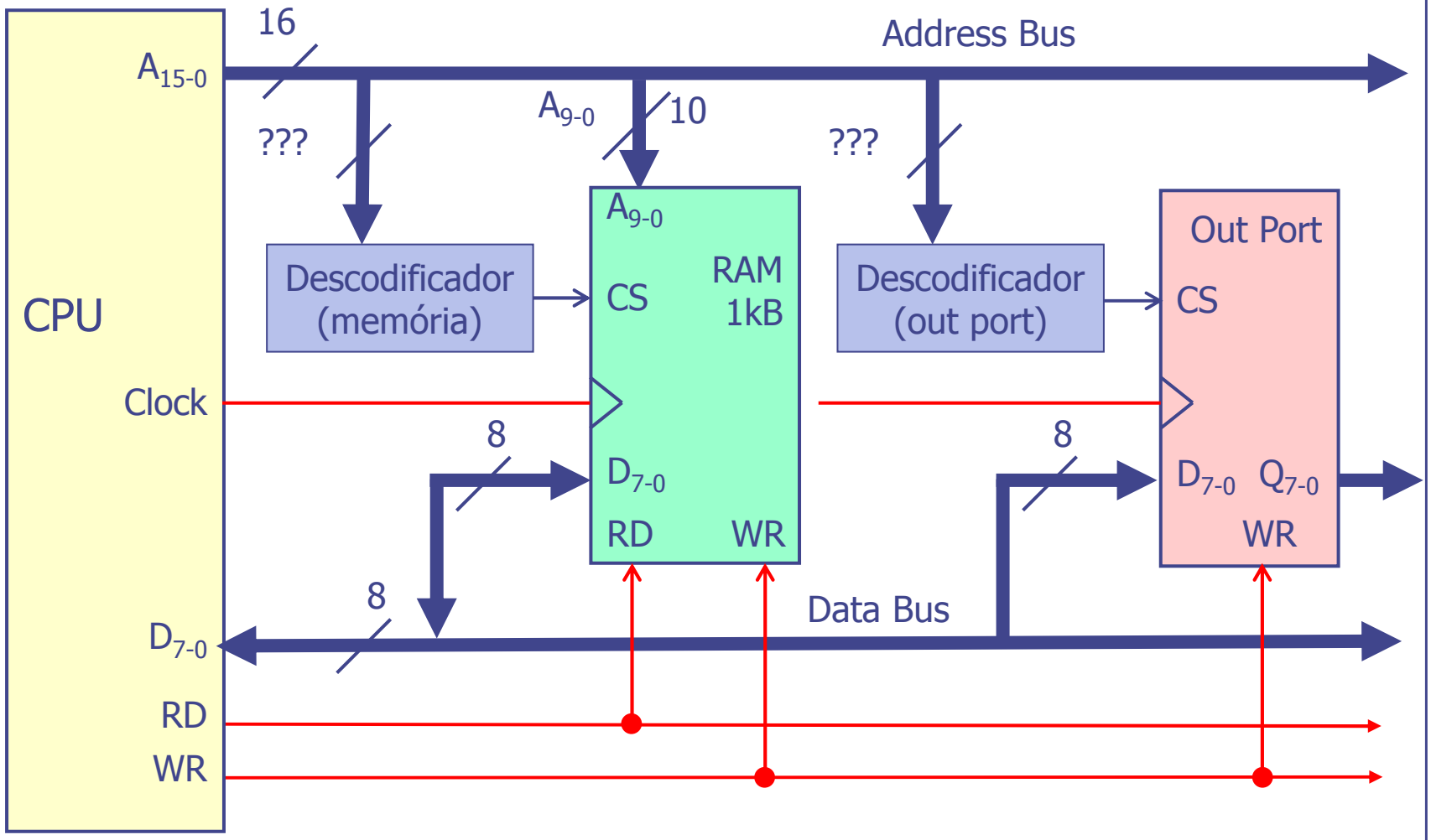
Descodificação de endereços

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



Descodificação de endereços

- Ligação 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 *address bus***
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 *address bus*
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
```

$An\backslash \Leftrightarrow \overline{An}$

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

- Porto de saída – **descodificação parcial**:
  - Não usar, por exemplo, os dois bits menos significativos

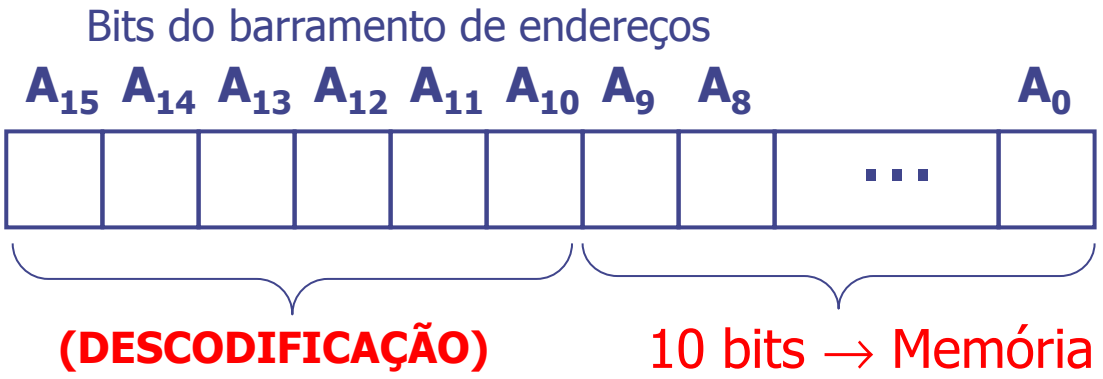
```
CS = A15\.A14.A13\.A12\.A11\.A10\.A9\.A8.
     A7\.A6\.A5\.A4\.A3\.A2\
```

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

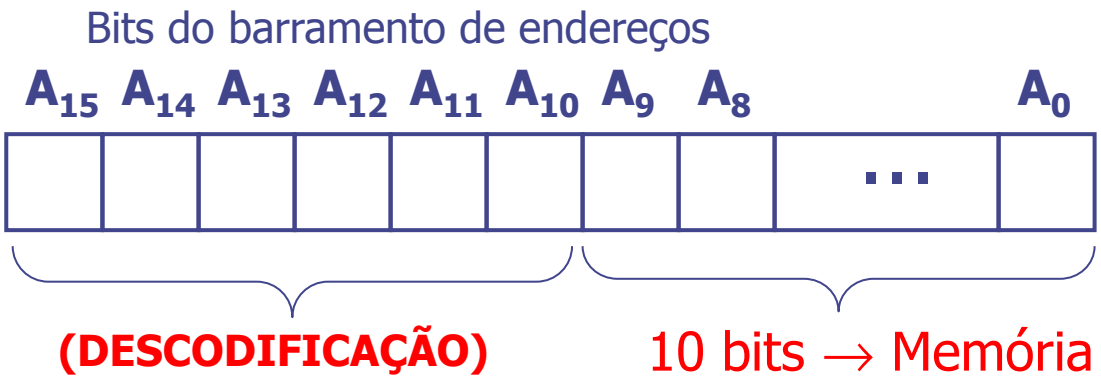
Descodificação de endereços

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

- Memória RAM



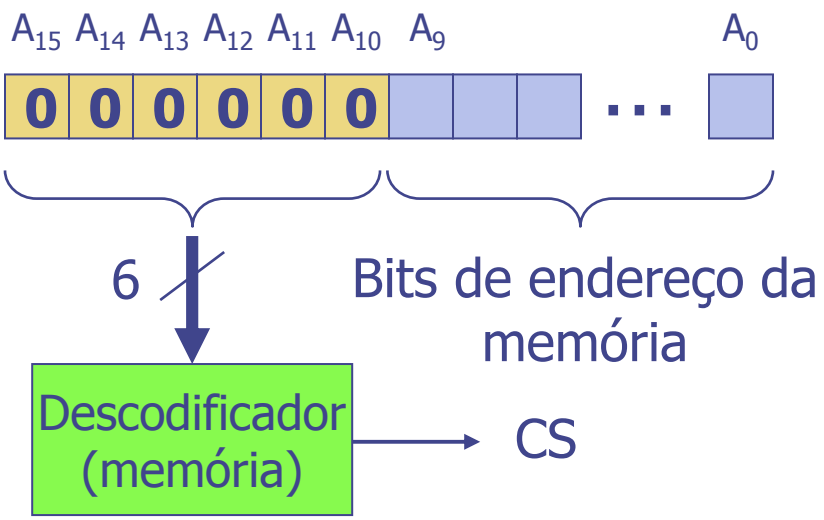
Descodificação de endereços

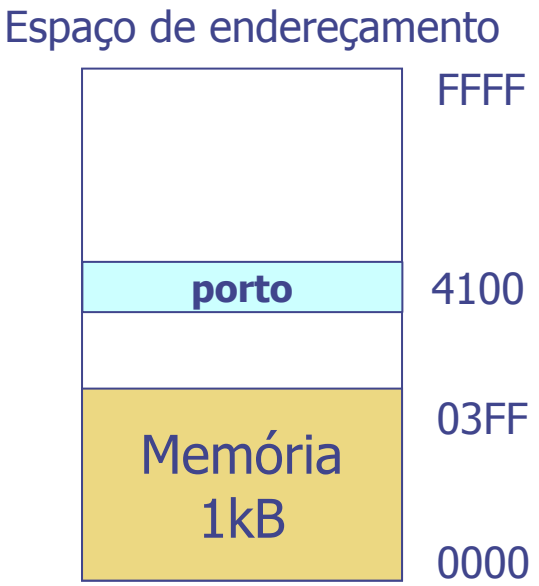


- Para garantir que a memória de 1kB está mapeada a partir do endereço 0x0000 (na gama 0x0000-0x03FF), há várias soluções possíveis; vamos analisar as seguintes 3:
  1) **descodificação total** – usar os 6 bits A15 a A10 (e.g. **000000**)
  2) **descodificação parcial** – usar A15, A14, A13 e A12 e ignorar 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



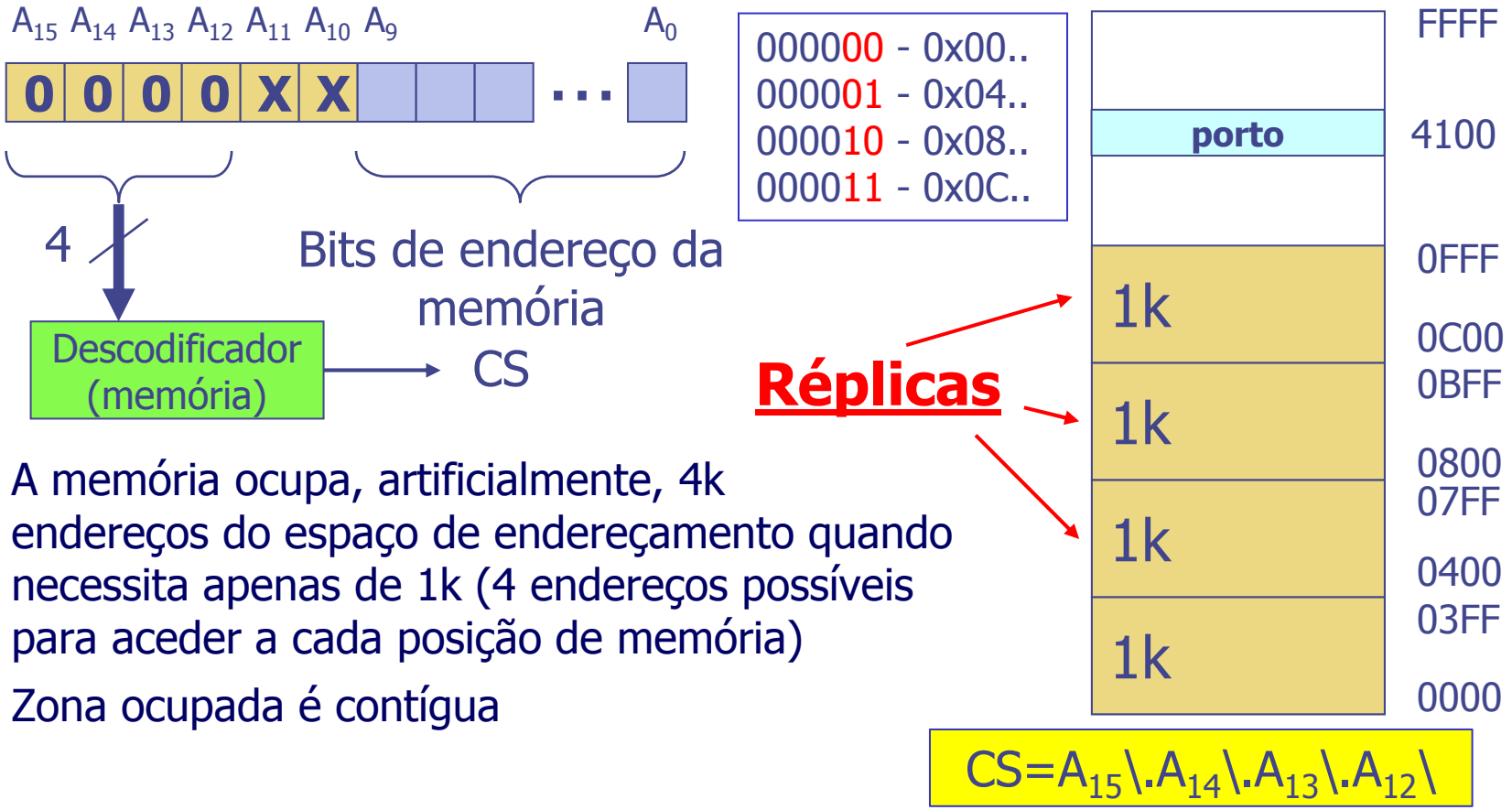


$CS = A_{15}\backslash.A_{14}\backslash.A_{13}\backslash.A_{12}\backslash.A_{11}\backslash.A_{10}\backslash$

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

Descodificação de endereços – descodificação parcial

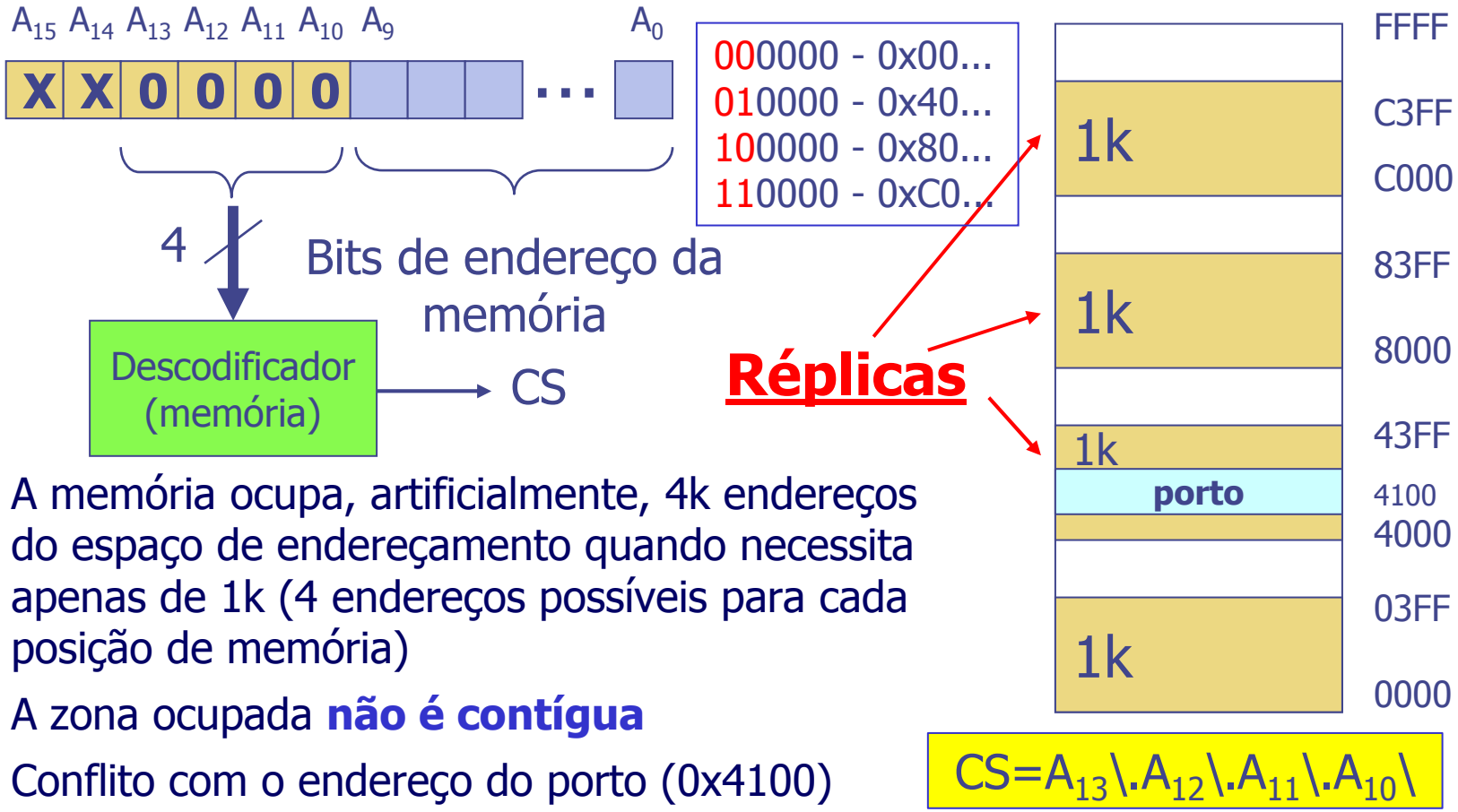
- Solução 2 – usar A15, A14, A13 e A12 e ignorar A11 e A10, e.g. **0000xx**. Isto significa que um endereço válido para aceder à memória não depende do valor dos bits A11 e A10, mas tem que ter os bits A15 a A12 a 0.



- A memória ocupa, artificialmente, 4k endereços do espaço de endereçamento quando necessita apenas de 1k (4 endereços possíveis para aceder a cada posição de memória)
- Zona ocupada é contígua

Descodificação de endereços – descodificação parcial

- Solução 3 – usar A13, A12, A11 e A10 e ignorar A15 e A14, e.g. **xx0000**. Isto significa que um endereço válido para aceder à memória não depende do valor dos bits A15 e A14, mas tem que ter os bits A13 a A10 a 0

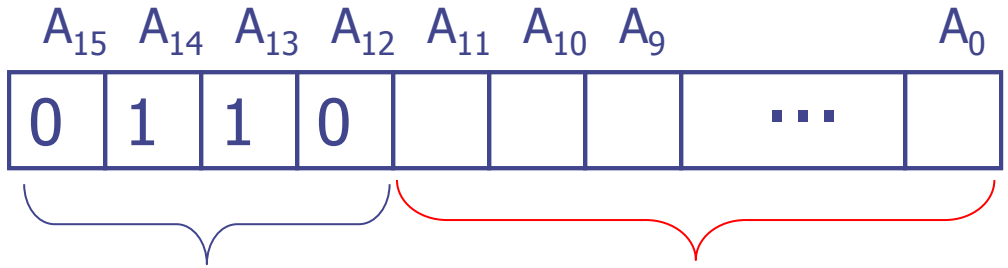


- A memória ocupa, artificialmente, 4k endereços do espaço de endereçamento quando necessita apenas de 1k (4 endereços possíveis para cada posição de memória)
- A zona ocupada **não é contígua**
- Conflito com o endereço do porto (0x4100)

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 kByte = 2^{12} (2^{12} - 1 = 0x0FFF)



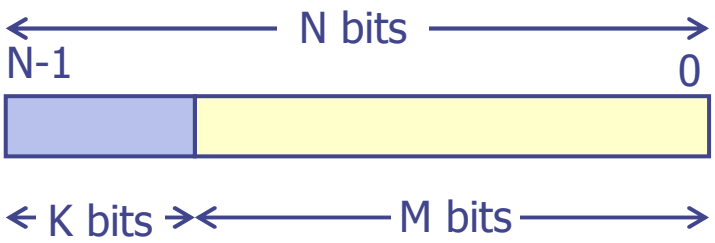
4 bits → Descodificação 12 bits → Memória

```
0110000000000000   (0x6000)
0110111111111111   (0x6FFF)
```

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

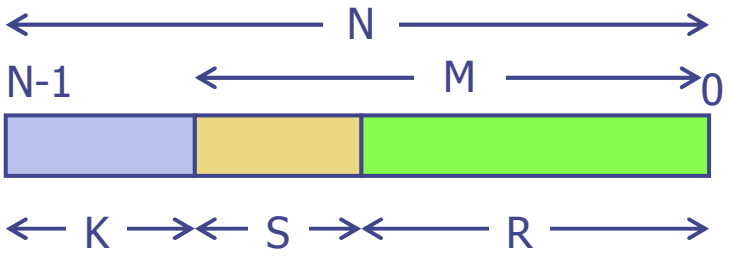
Gerador de sinais de seleção programável

- 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

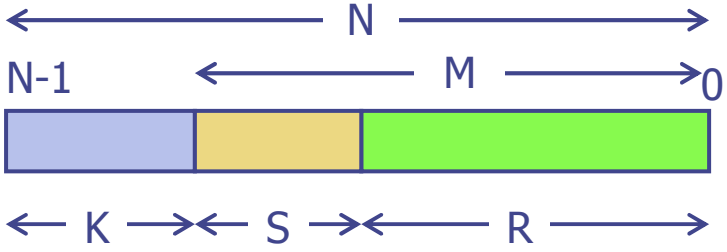
- O mesmo método pode ser aplicado para a sub-divisão dos M bits da gama descodificada: S bits usados para descodificação, R bits usados para endereçamento



- 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

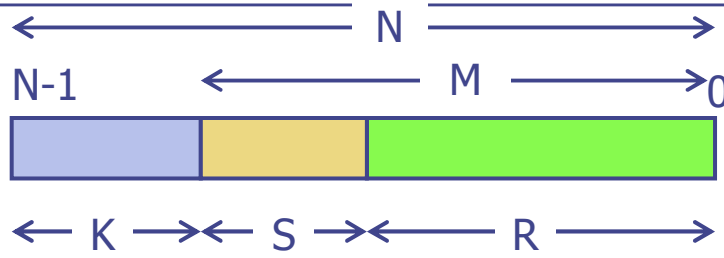
Gerador de sinais de seleção programável - exemplo

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

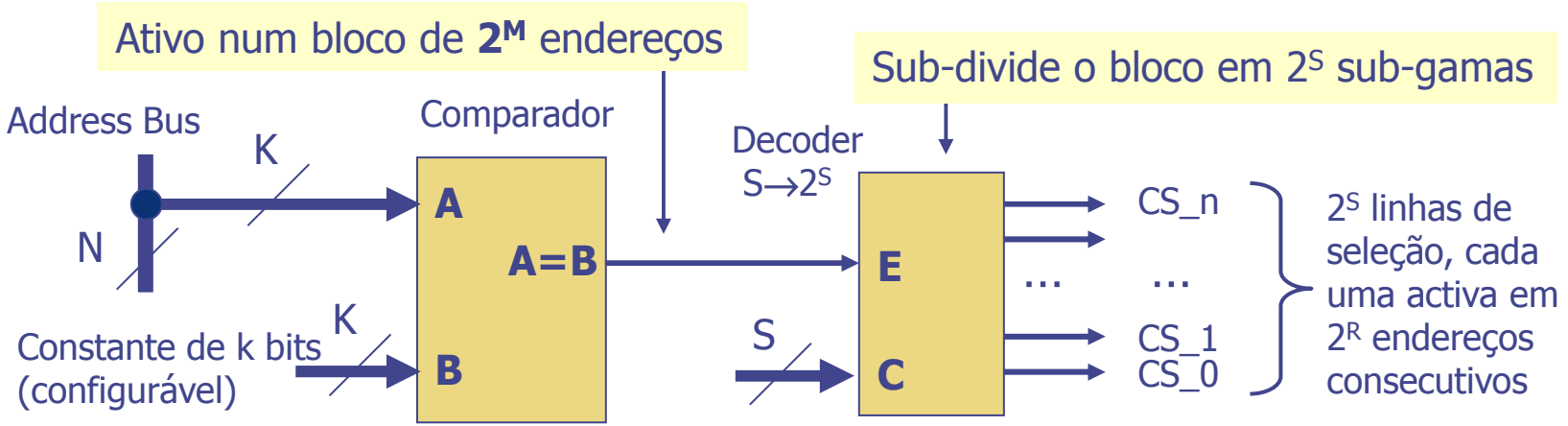


- M=5 (K=3), S2=2 e R=3
  - N = 8 – espaço de endereçamento com 2^8 = 256 endereços
  - M = 5 – gama descodificada com 2^5 = 32 endereços
  - S = 2 – número de sub-gamas 2^2 = 4
  - R = 3 – dimensão da sub-gama: 2^3 = 8 endereços
- A gama de 2^5 endereços foi sub-dividida em 2^2 gamas iguais, de 2^3 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



- 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



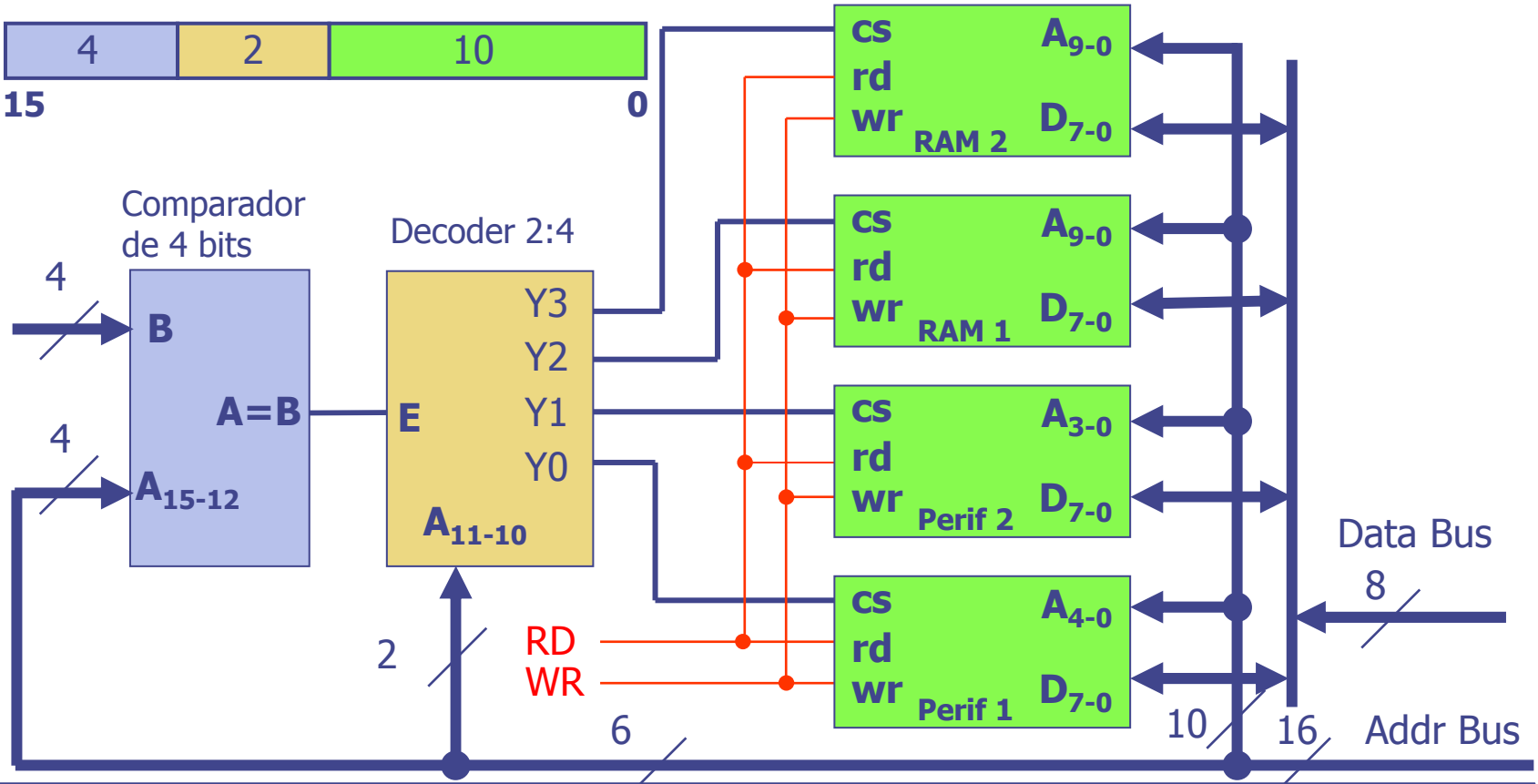
- **Exemplo**: N=16, K=4, S=2 (R=10); constante de comparação: 0010_2
  - Bloco descodificado pelo comparador: 0x2000 a 0x2FFF (i.e. 2^{12}=4K)
  - Nº de sub-gamas descodificadas: 2^2 (4 linhas de seleção, CS_0 a CS_3)
  - Dimensão de cada sub-gama: 2^{10} = 1024
- **Pergunta**: Qual das linhas CS_x é ativada pelo endereço 0x27C5?

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 registos internos
- Assuma que este descodificador pode usar e 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

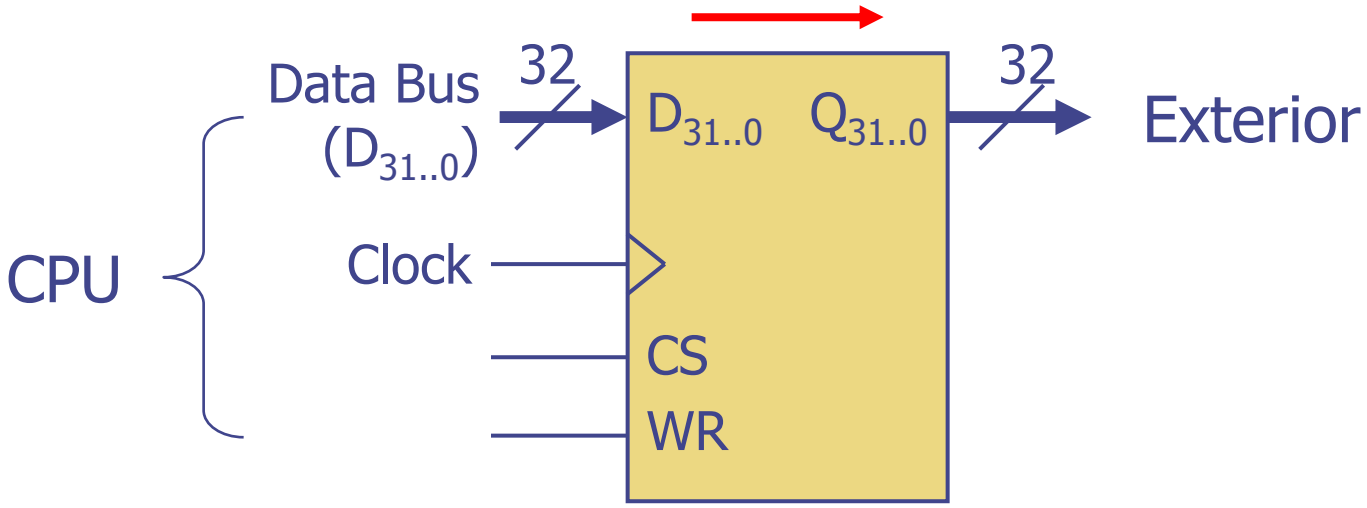
Gerador de sinais de seleção programável - exercício



- 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

Exemplos de portos de E/S – porto de saída de 32 bits

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

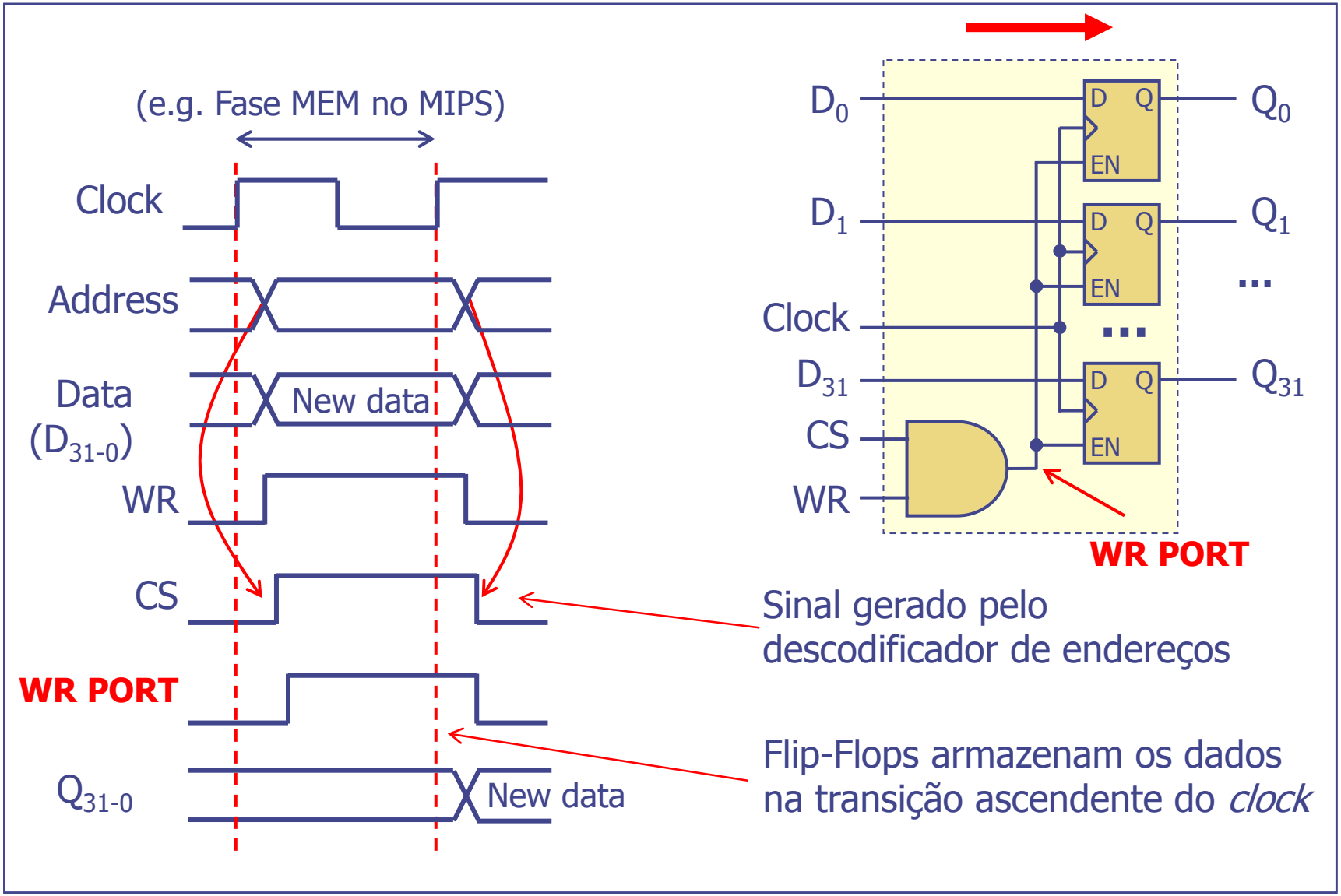


- O porto armazena informação proveniente do CPU, transferida durante uma operação de escrita na memória (estágio MEM nas instruções "**`sw`**", no caso do MIPS)
- A escrita no porto é feita na transição ativa do relógio se os sinais "**`CS`**" e "**`WR`**" estiverem ambos ativos
- O sinal "**`CS`**" 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)

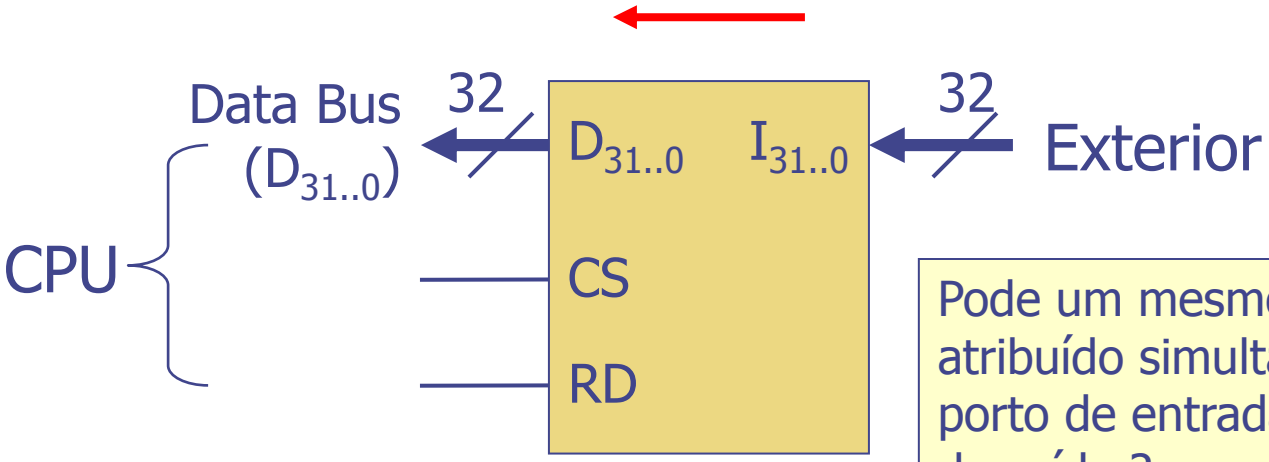
```
entity OutPort is
   port(clk, wr, cs  : in  std_logic;
        dataIn       : in  std_logic_vector(31 downto 0);
        dataOut      : out std_logic_vector(31 downto 0));
end OutPort;
architecture behav of OutPort is
begin
   process(clk)
   begin
      if(rising_edge(clk)) then
         if(cs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

Porto de saída de 32 bits



Exemplos de portos de E/S – porto de entrada de 32 bits

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



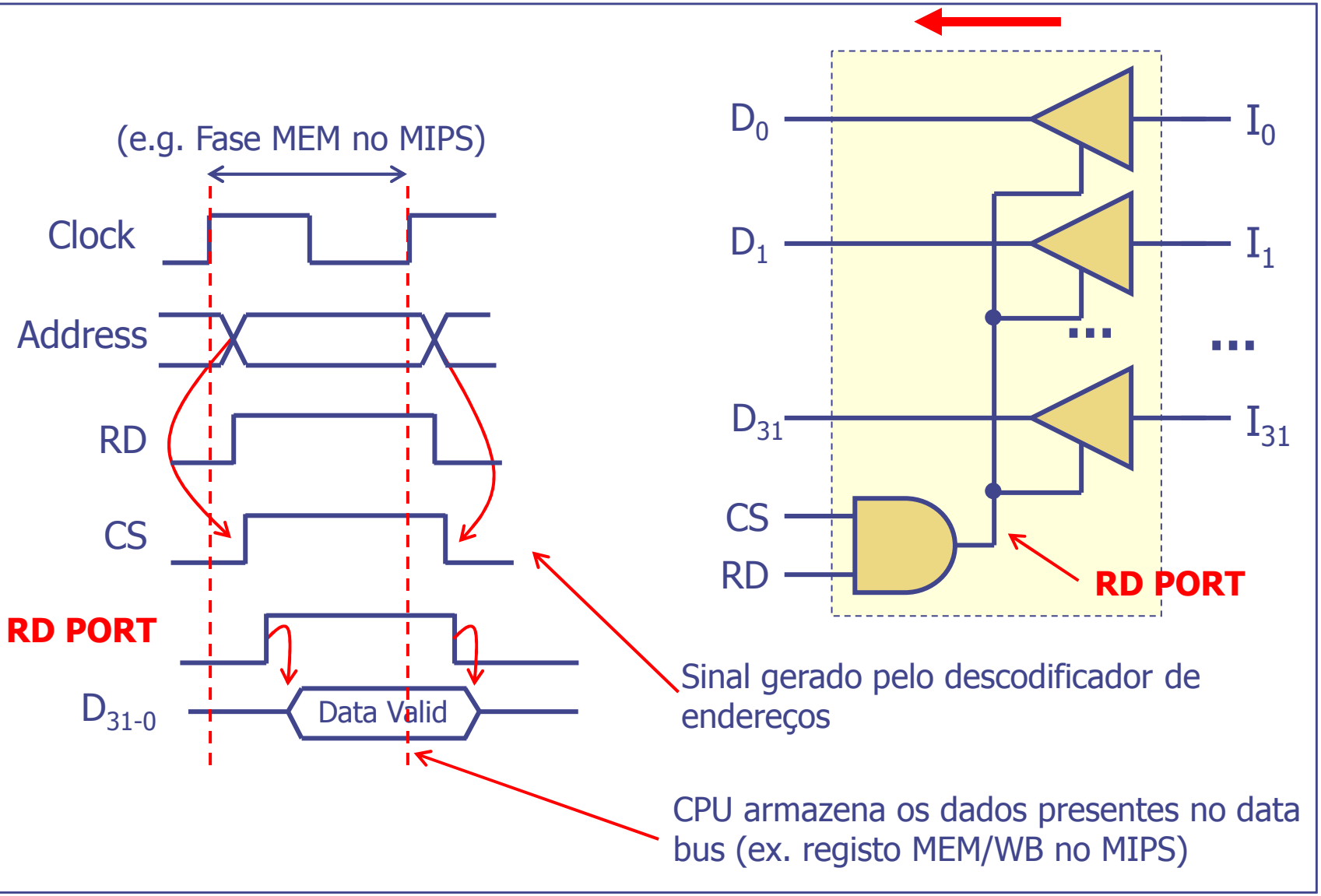
Pode um mesmo endereço estar atribuído simultaneamente a um porto de entrada e a um porto de saída ?

- 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**

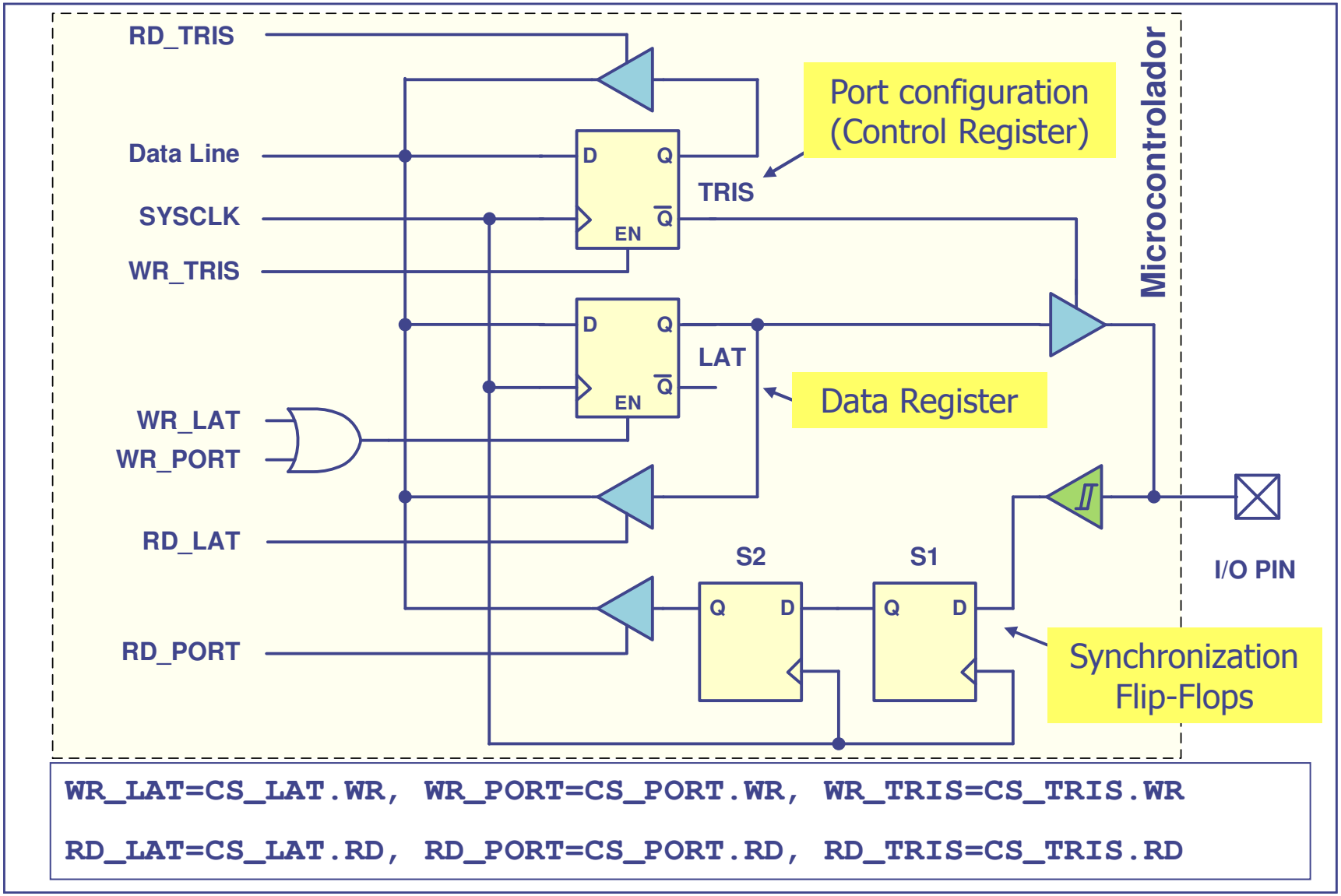
Portos de I/O no PIC32

- 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`**$_2$
  - portos 0, 2, 4 e 6 configurados como saída
  - portos 1, 3, 5 e 7 configurados como entrada

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

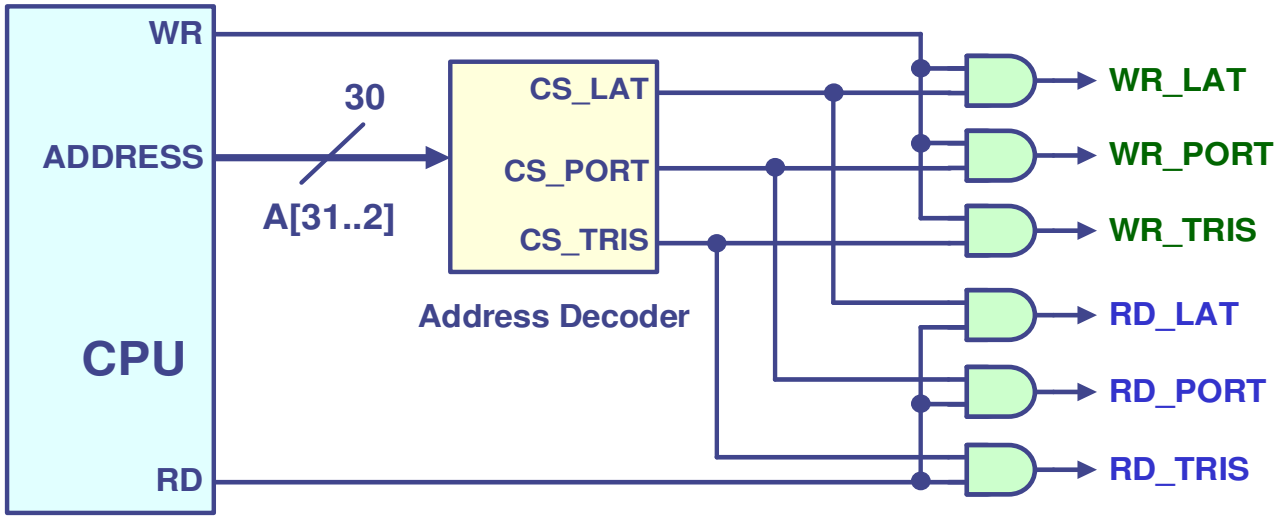


Portos de I/O no PIC32

- Registo TRISx (TRISB, TRISC, …) agrupa todos os flip-flop TRIS dos portos de I/O de 1 bit; permite a configuração individual de cada um dos bits do porto
- Registo LATx (LATB, LATC, …) é o registo de dados e agrupa todos os flip-flops LAT dos portos de I/O de 1 bit
- 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*. 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

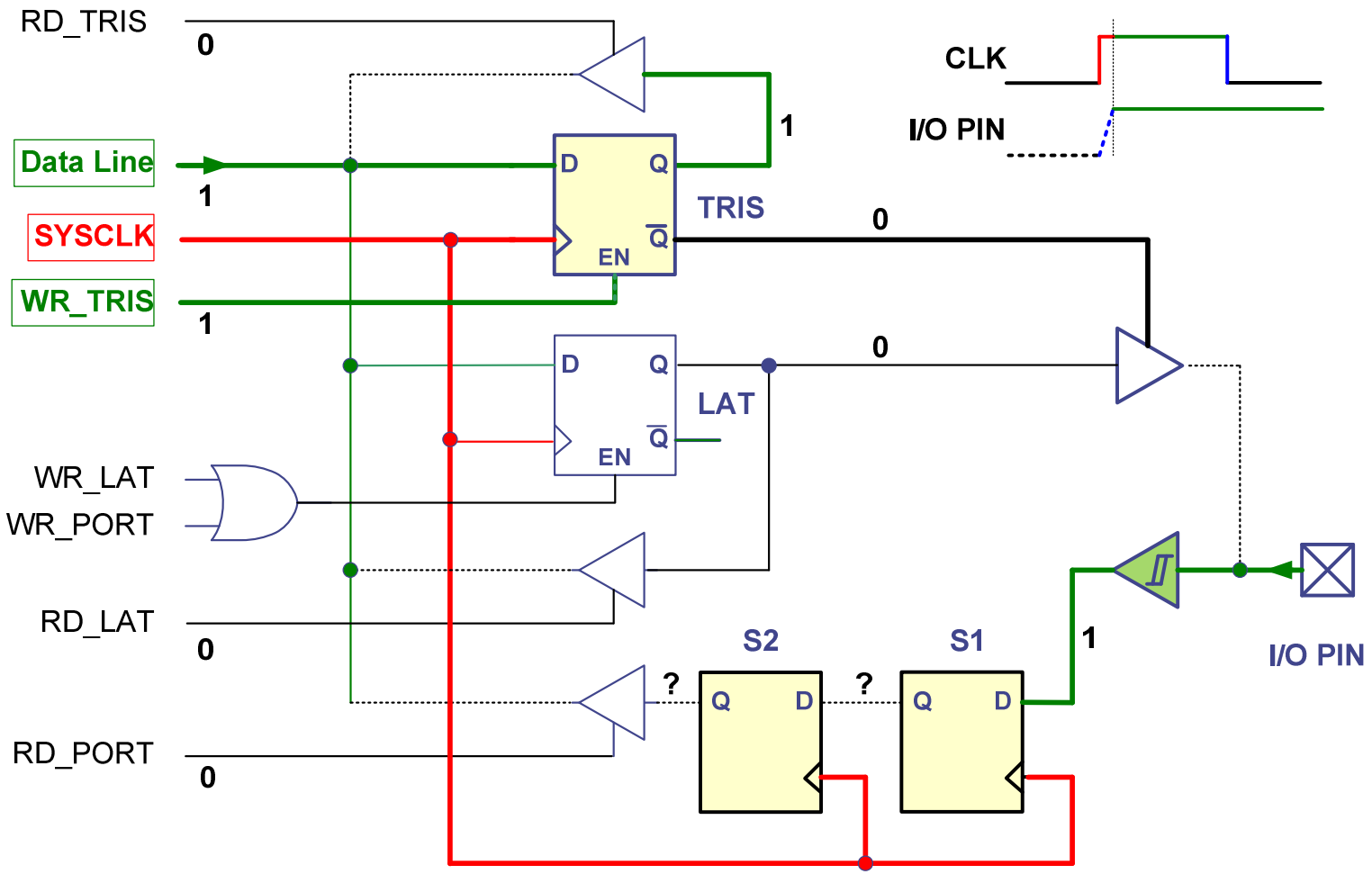
- A escrita no porto é feita no endereço referenciado pelo identificador **LATx**, em que x é a letra que identifica o porto; a leitura do porto é feita do endereço referenciado por **PORTx**
- Os portos estão mapeados no espaço de endereçamento unificado do PIC32 (ver aula 2), em endereços definidos 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:
  - Endereço de **`TRISB`**: **`0xBF886040`**
  - Endereço de **`PORTB`**: **`0xBF886050`**
  - Endereço de **`LATB`**: **`0xBF886060`**

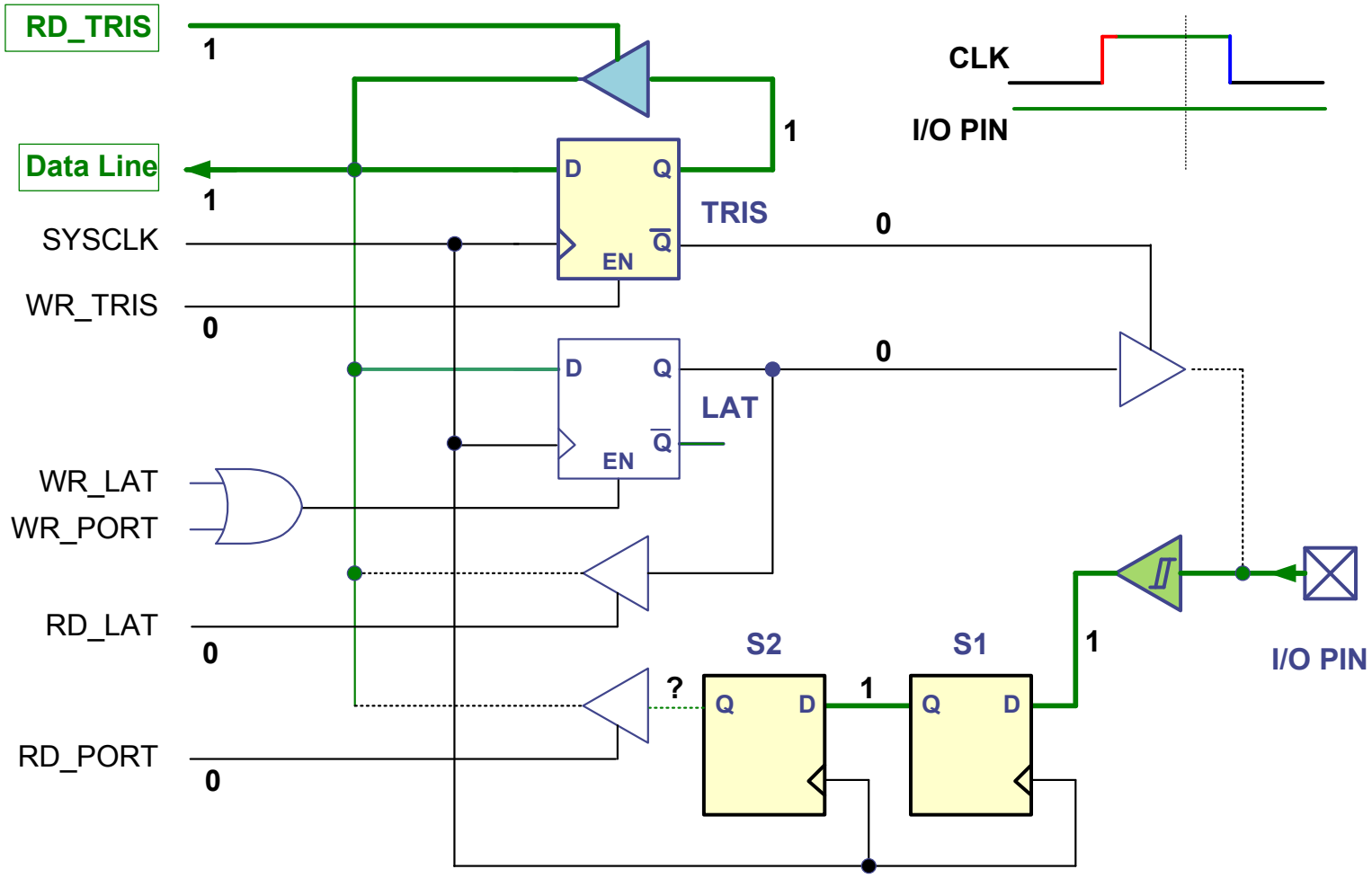
Modelo simplificado de um porto de I/O de 1 bit no PIC32

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



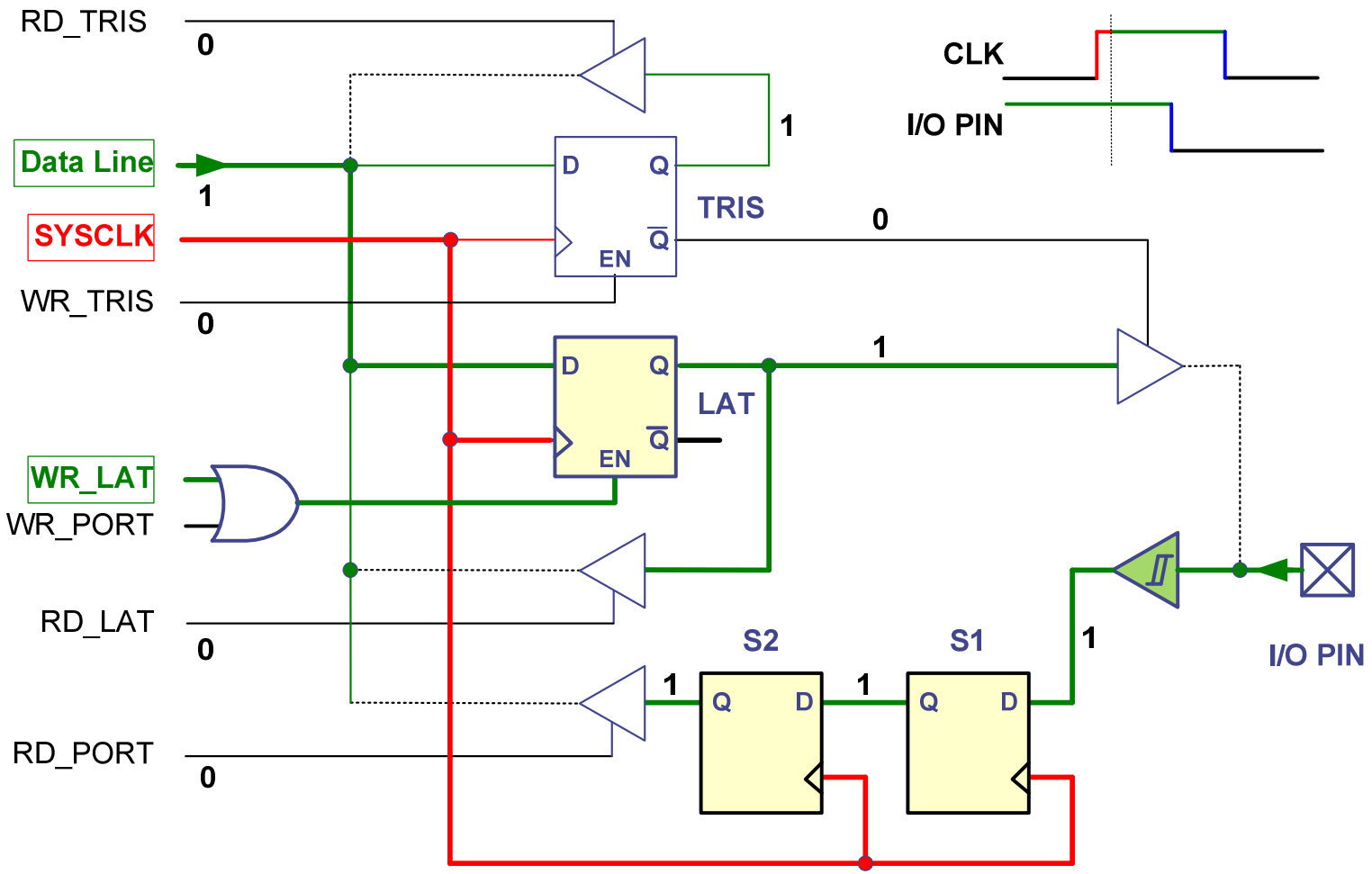
Modelo simplificado de um porto de I/O de 1 bit no PIC32

Ler o valor do registo TRIS



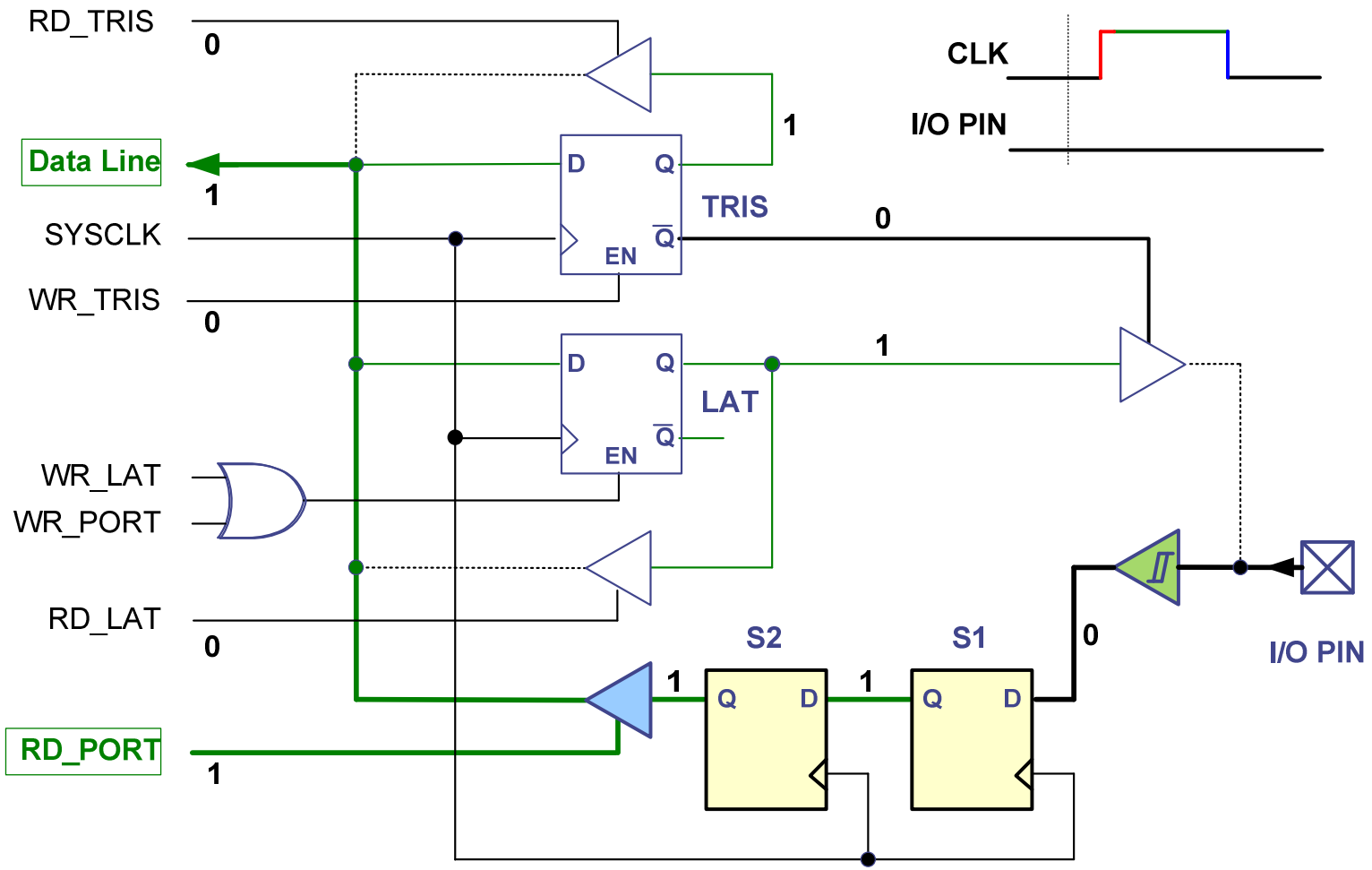
Modelo simplificado de um porto de I/O de 1 bit no PIC32

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



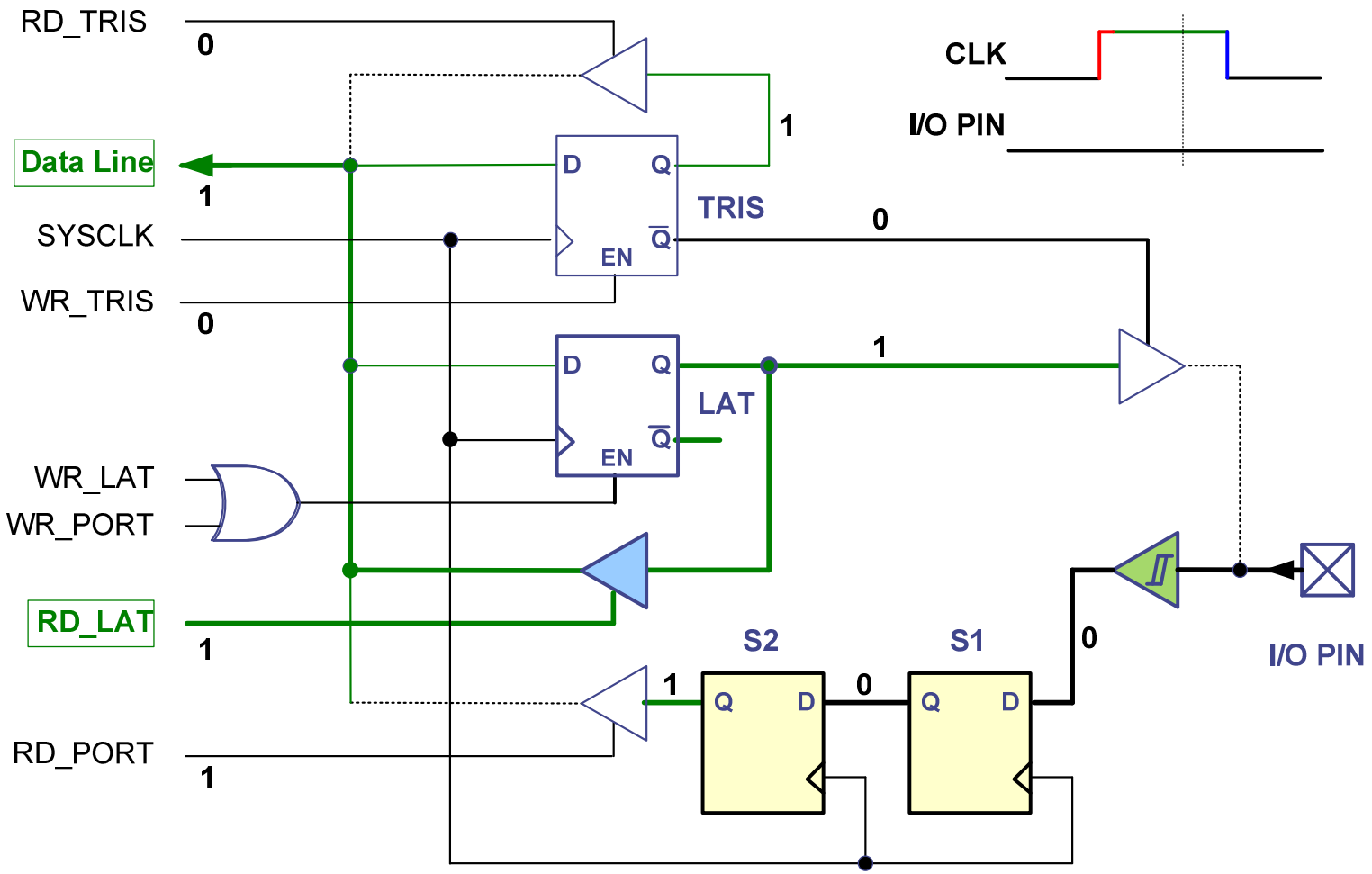
Modelo simplificado de um porto de I/O de 1 bit no PIC32

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



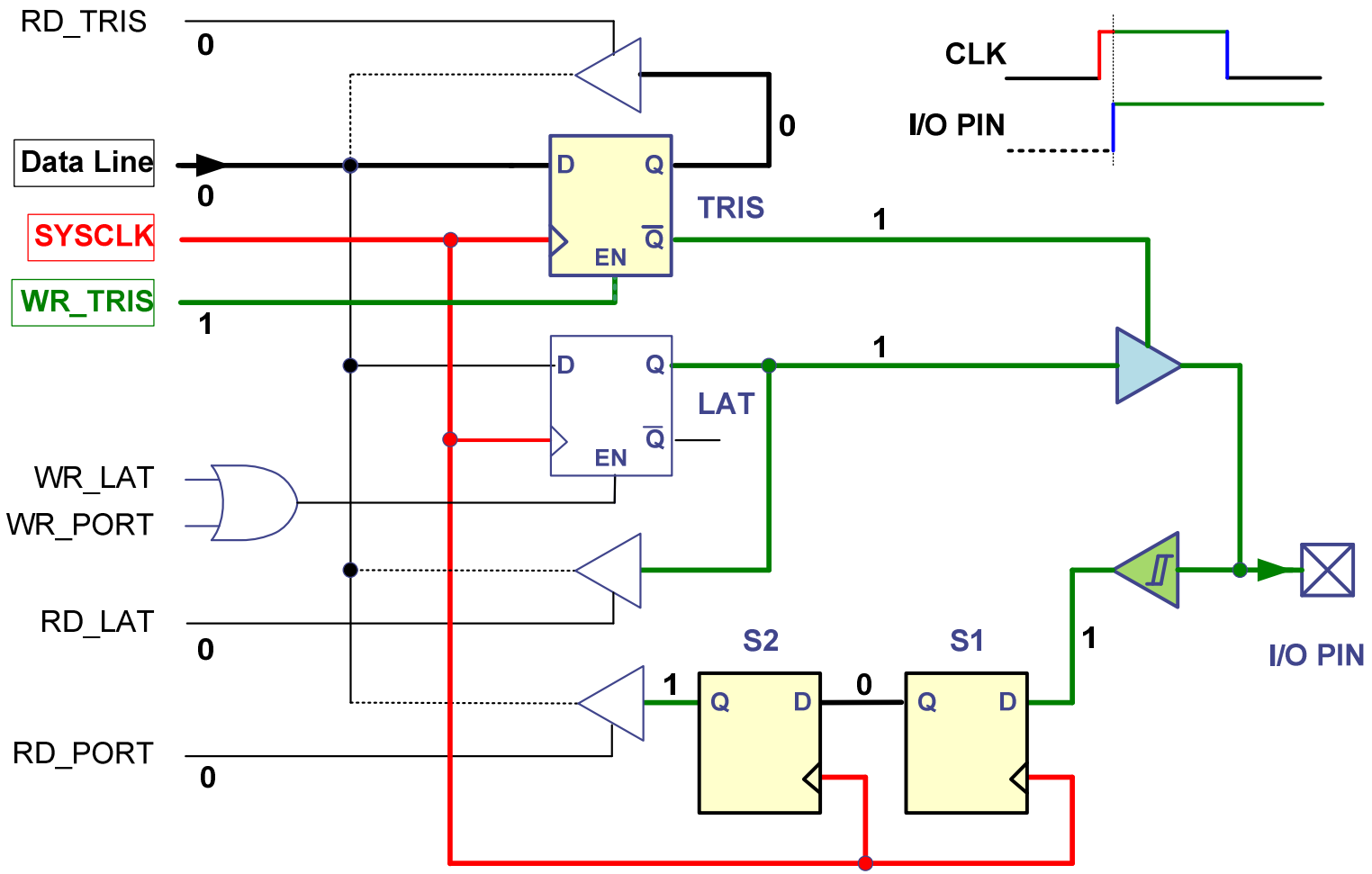
Modelo simplificado de um porto de I/O de 1 bit no PIC32

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



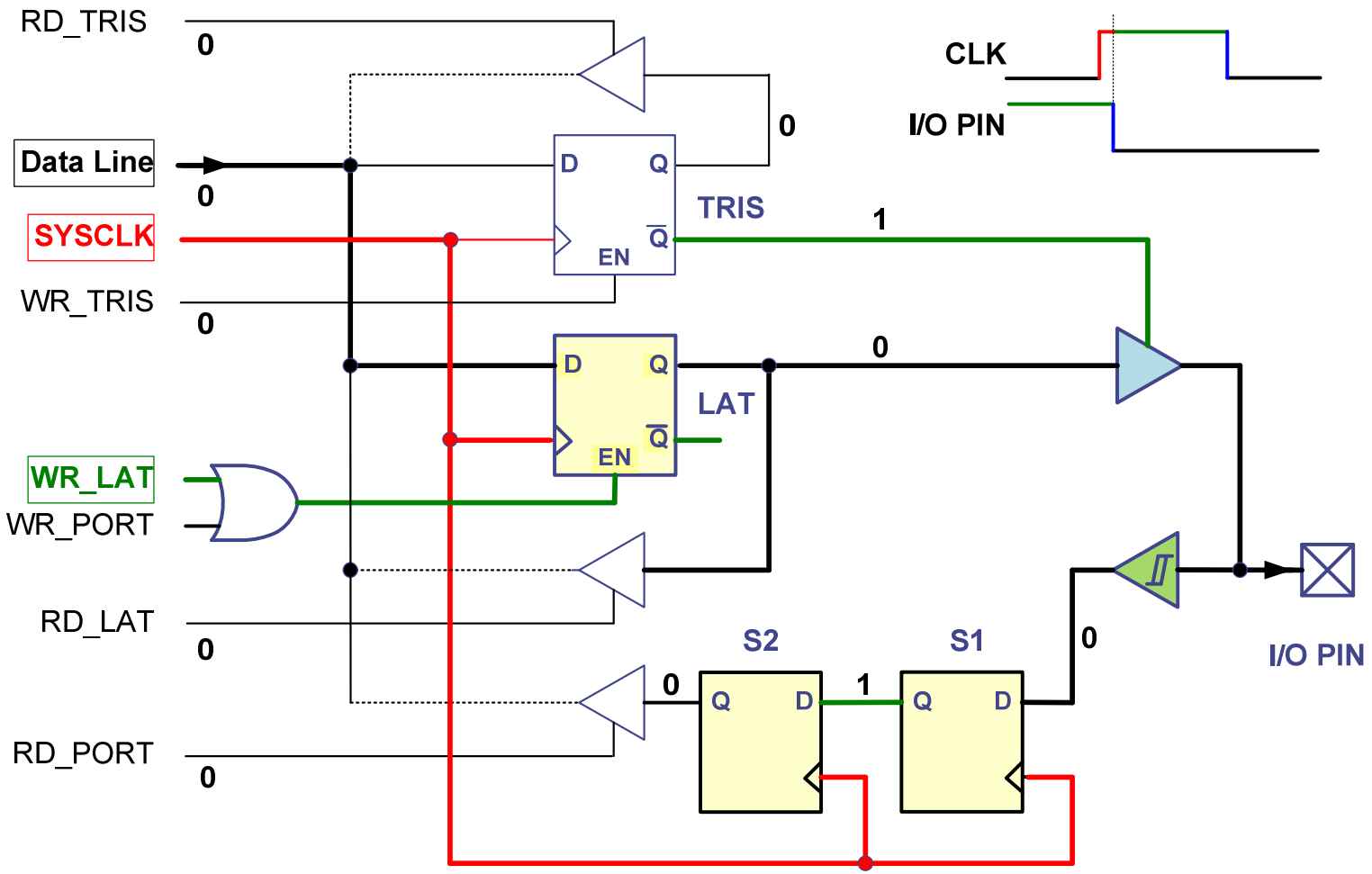
Modelo simplificado de um porto de I/O de 1 bit no PIC32

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

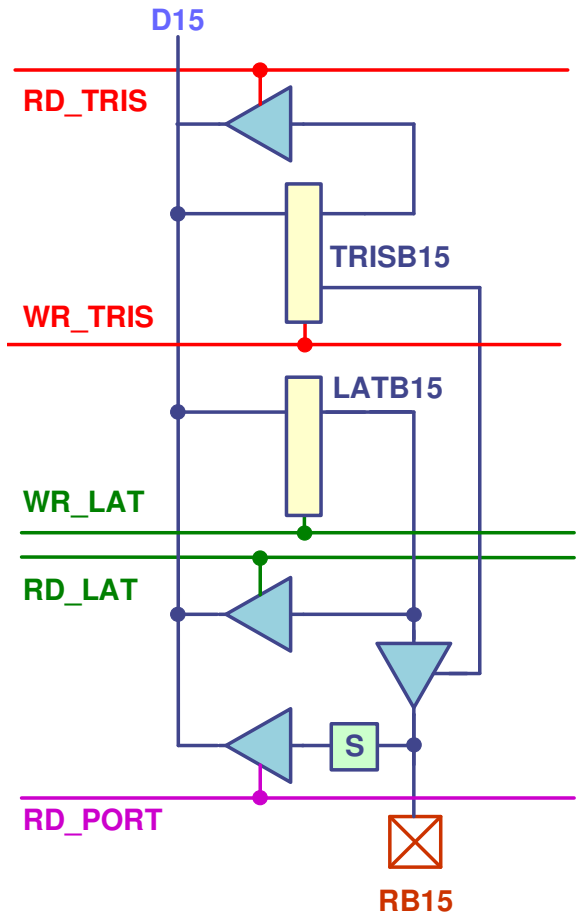


Modelo simplificado de um porto de I/O de 1 bit no PIC32

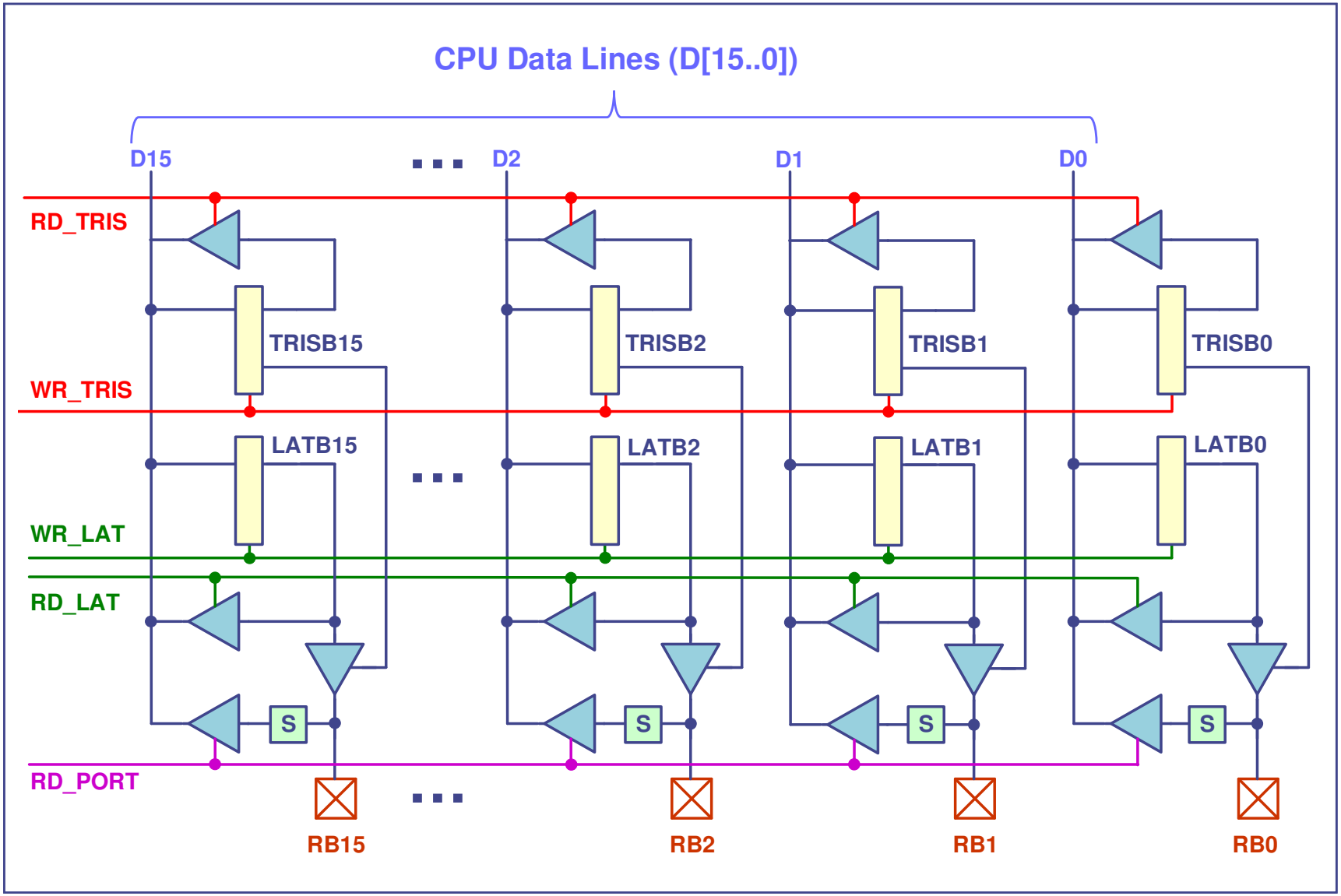
Alterar o valor do porto de saída – Escrita do valor '0' no LAT



Modelo simplificado de um porto de I/O no PIC32

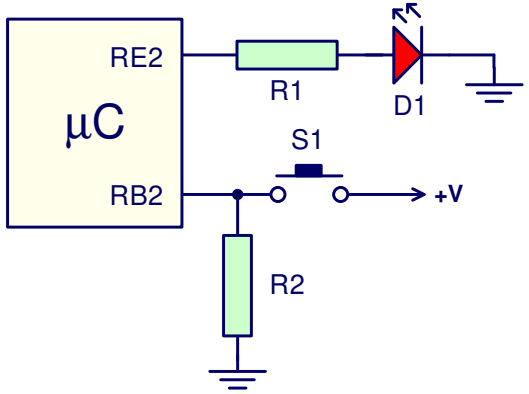


Modelo simplificado de um porto de I/O no PIC32

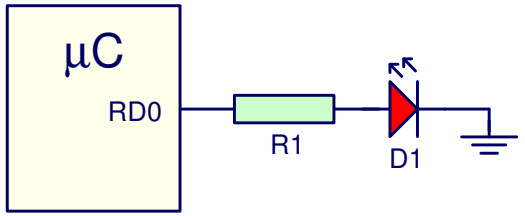


Exemplo de configuração/utilização dos portos

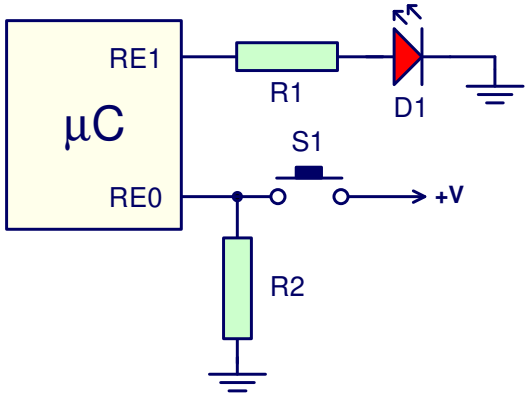
- Exemplo 1:
  - Acender o LED D1 enquanto o *switch* S1 estiver premido; LED ligado ao porto RE2 e *switch* ligado ao porto RB2



- 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  SFR_BASE_HI,0xBF88

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

.equ  TRISD,0x60C0    # TRISD address: 0xBF8860C0
.equ  PORTD,0x60D0    # TRISD 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,SFR_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 $t1,$t1,0xFFFB    # 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 $t2,$t2,0xFFFB    # 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,SFR_BASE_HI   # 16 MSbits of port addresses
        lw   $t1,TRISD($t0)     # Read TRISD register
        andi $t1,$t1,0xFFFE     # 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 $t1,$t1,0xFFFE     # 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)

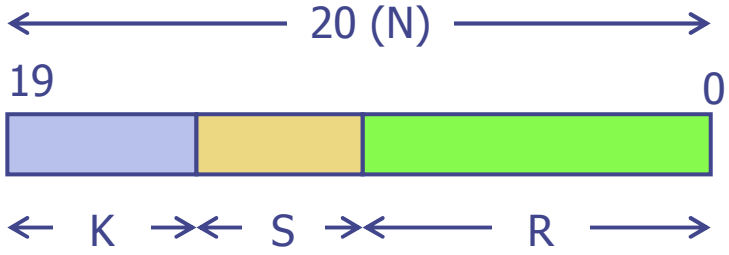
```
        .text
        .globl  main
main:   lui  $t0,SFR_BASE_HI   # 16 MSbits of port addresses
        lw   $t1,TRISE($t0)     # Read TRISE register
        ori  $t1,$t1,0x0001     # bit0 = 1 (IN)
        andi $t1,$t1,0xFFFD     # bit1 = 0 (OUT)
        sw   $t1,TRISE($t0)     # TRISE configured
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 $t2,$t2,0xFFFD     # Reset bit 1
        or   $t2,$t2,$t1        # Merge data
        sw   $t2,LATE($t0)      # Write LATE register
        j    loop
```

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, respetivamente.
   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\ = A15 + A13\
   a) Indique as gamas do espaço de endereçamento de 16 bits ocupadas pela memória.
   b) Indique os endereços possíveis para aceder à 15ª 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.

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.

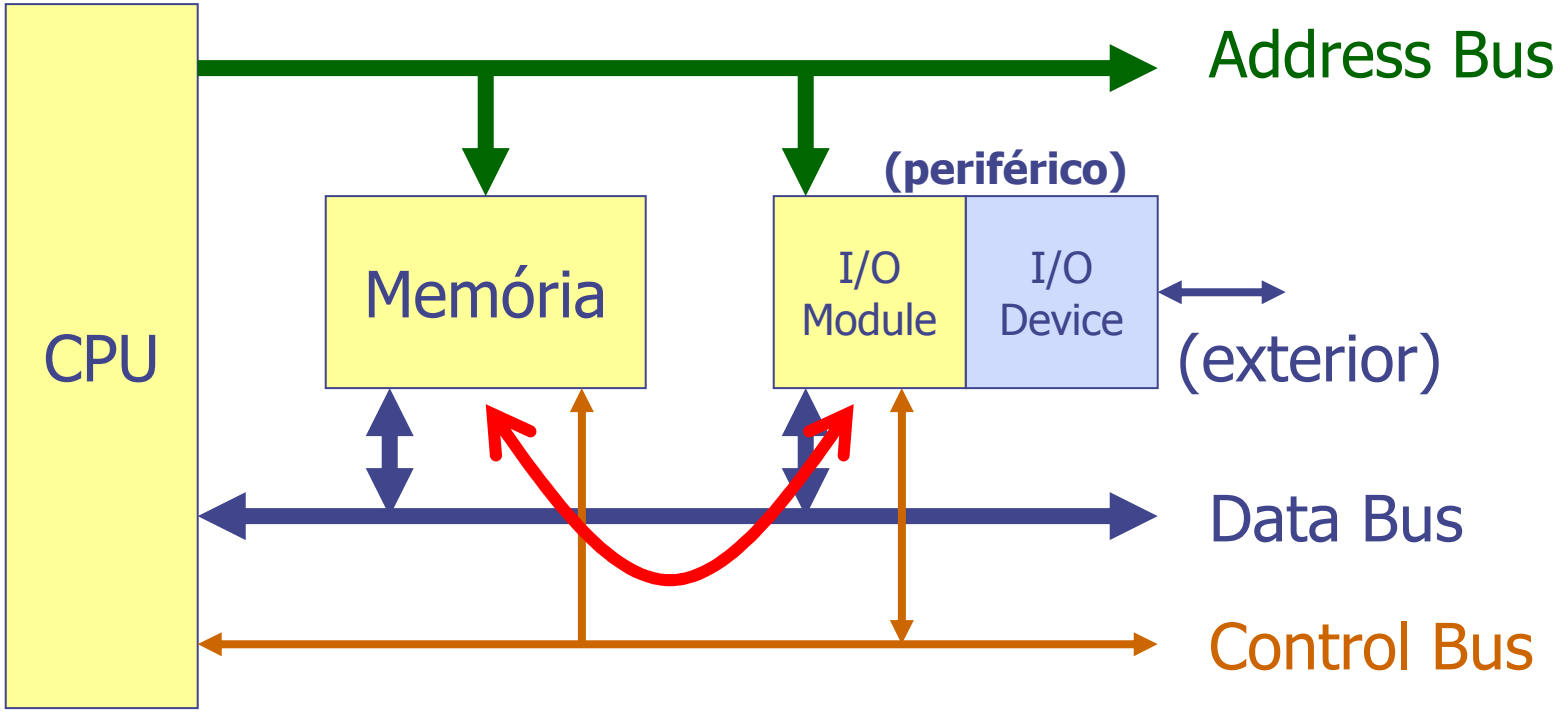
   a) configure o porto RB para executar corretamente a tarefa descrita
   b) efetue a leitura do porto indicado
   c) execute um ciclo de espera de 10ms
   d) efetue a transformação da informação lida para preparar o processo de escrita naquela porto
   e) efetue, no byte mais significativo, o valor resultante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

Aulas 6, 7 e 8

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

José Luís Azevedo, Arnaldo Oliveira, Tomás Silva, Bernardo Cunha

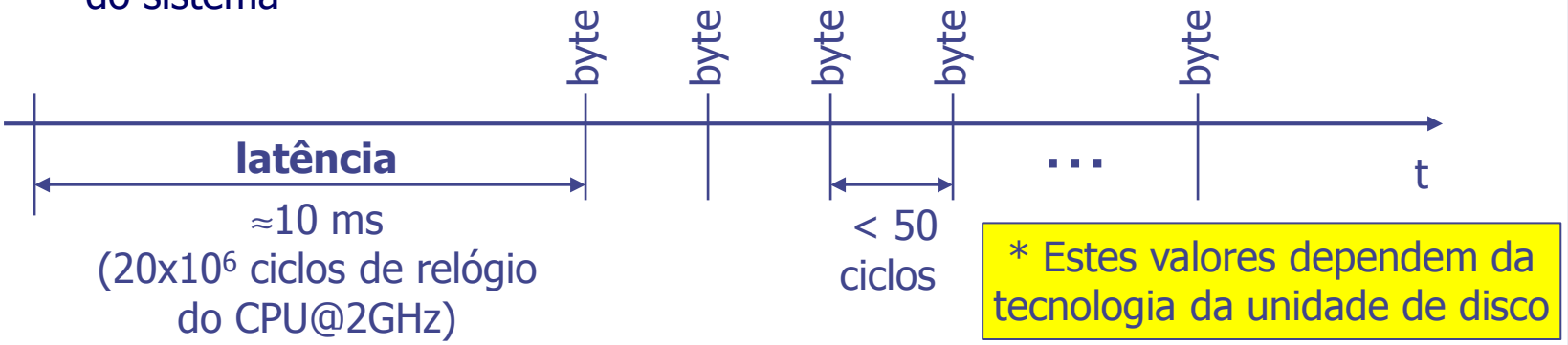
Transferência de informação entre memória e I/O



- CPU – processamento
- Memória – armazenamento (instruções e dados)
- Unidades de I/O – comunicação com dispositivos externos
- Que métodos podem ser usados para transferir informação entre os periféricos e a memória?

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 directo à memória (**DMA**)
     - Um dispositivo hardware externo ao CPU (DMA) faz a transferência de informação diretamente entre a memória e o periférico; o CPU não toma parte no processo de transferência
     - O CPU apenas configura inicialmente o periférico e o DMA; no final o DMA sinaliza o CPU que a transferência terminou

E/S Programada

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

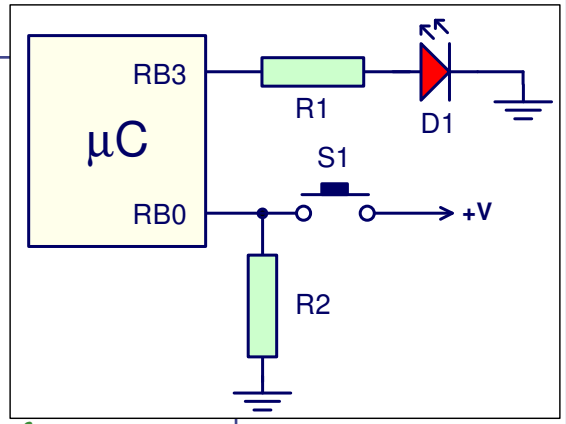
polling

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

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

E/S Programada (exemplo para o PIC32)

- Exemplo: comutar o estado do LED (ligado ao porto RB3) sempre que é detetada uma transição de 0 para 1 no porto RB0 (assumindo um sinal isento de *bouncing*).




```
        # config PIC32 ports
        lui   $t0,SFR_BASE_HI#
        lw    $t1,TRISB($t0)  #
        ori   $t1,$t1,0x0001  # RB0=1
        andi  $t1,$t1,0xFFF7  # RB3=0
        sw    $t1,TRISB($t0)  #
Wh0:    lw    $t1,PORTB($t0)  # while(1) {
        andi  $t2,$t1,1       #
        beq   $t2,$0,wh0      #   while(RB0==0);
        lw    $t3,LATB($t0)   #
        xori  $t3,$t3,0x0008  #
        sw    $t3,LATB($t0)   #   LATB3=!LATB3;
Wh1:    lw    $t1,PORTB($t0)  #
        andi  $t1,$t1,1       #
        bne   $t1,$0,wh1      #   while(RB0==1);
        j     wh0             # }
```

polling

polling

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 dada 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 ao CPU esse facto
- 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

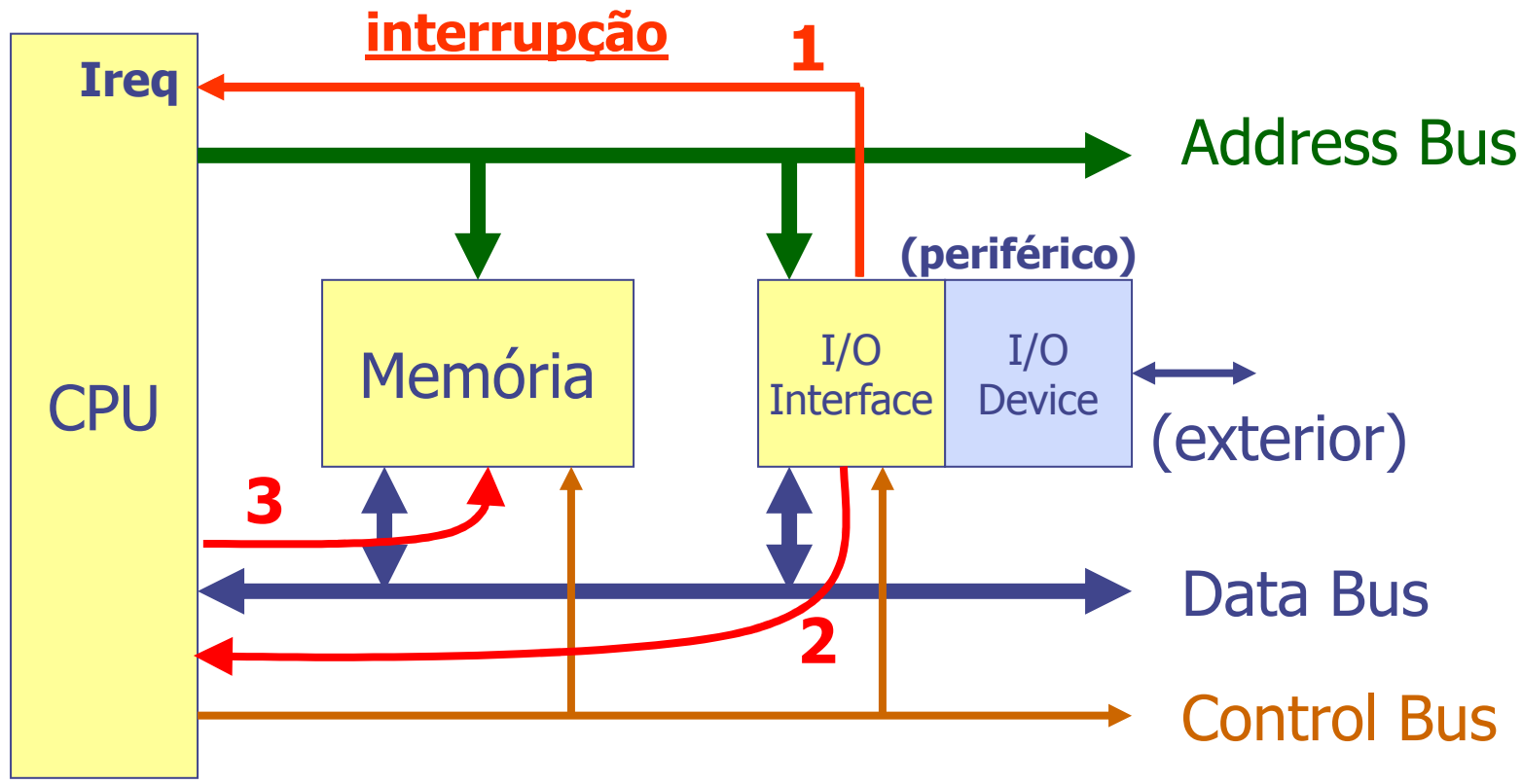
E/S por interrupção (exemplo de leitura)

```
// 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++
}
```

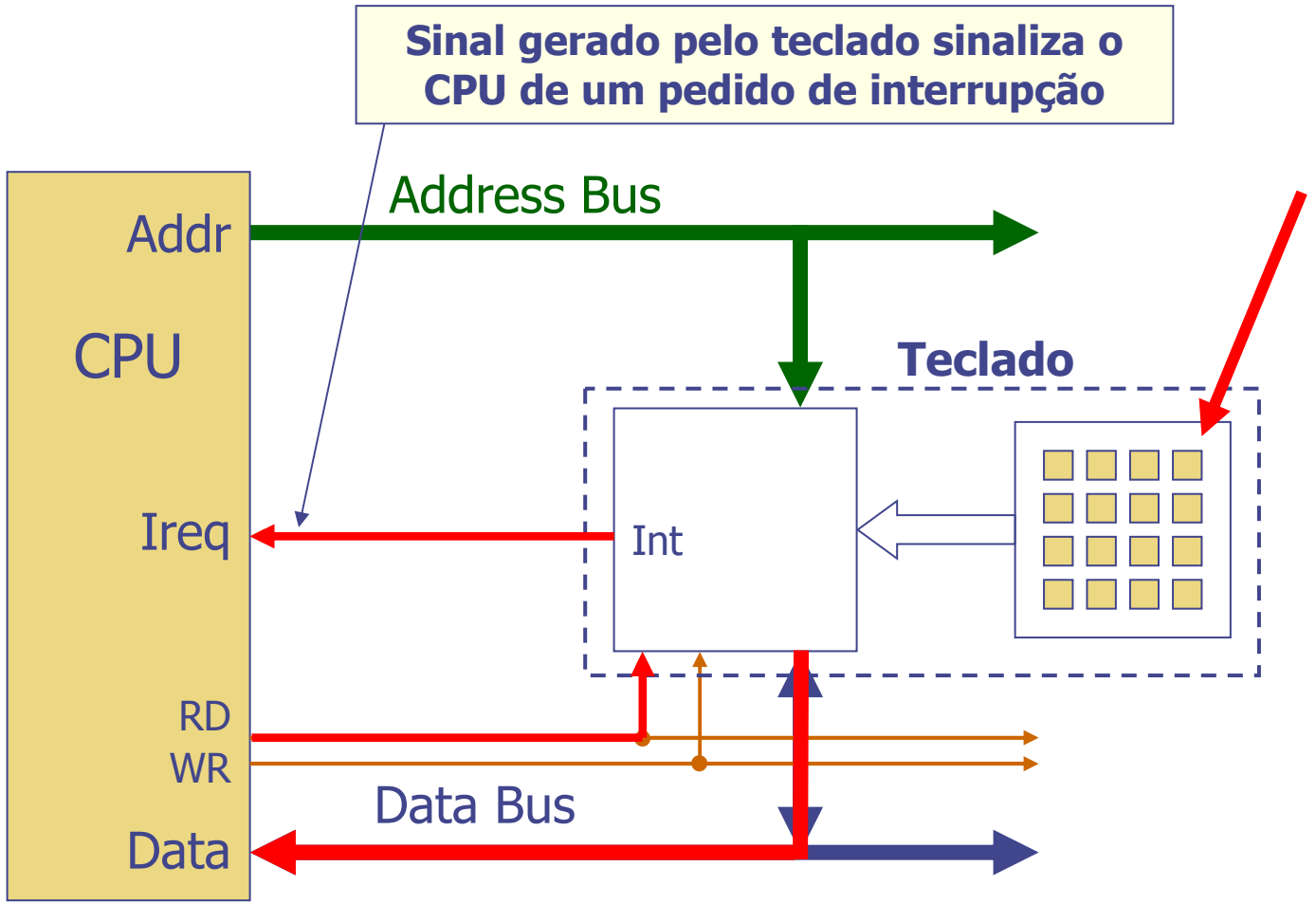
- **Não existe qualquer ciclo de espera**. O periférico gera o pedido de interrupção quando está pronto a transferir a informação
- O programa em execução **pode ser interrompido a qualquer momento**
- A Rotina de Serviço à Interrupção (RSI) tem que **salvaguardar o contexto** do programa (registos internos,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



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

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 a cada transição de 0 para 1 de um sinal de entrada

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

isr:# save program context
        (...)   # prólogo
        lui  $t0,SFR_BASE_HI
        lw   $t1,LATB($t0)   #
        xori $t1,$t1,0x0008 #
        sw   $t1,LATB($t0)   # RB3=!RB3;
        # restore program context
        (...)   # epílogo
        eret # exception return
```

Transição de 0 para 1 em RB0 inicia uma interrupção

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

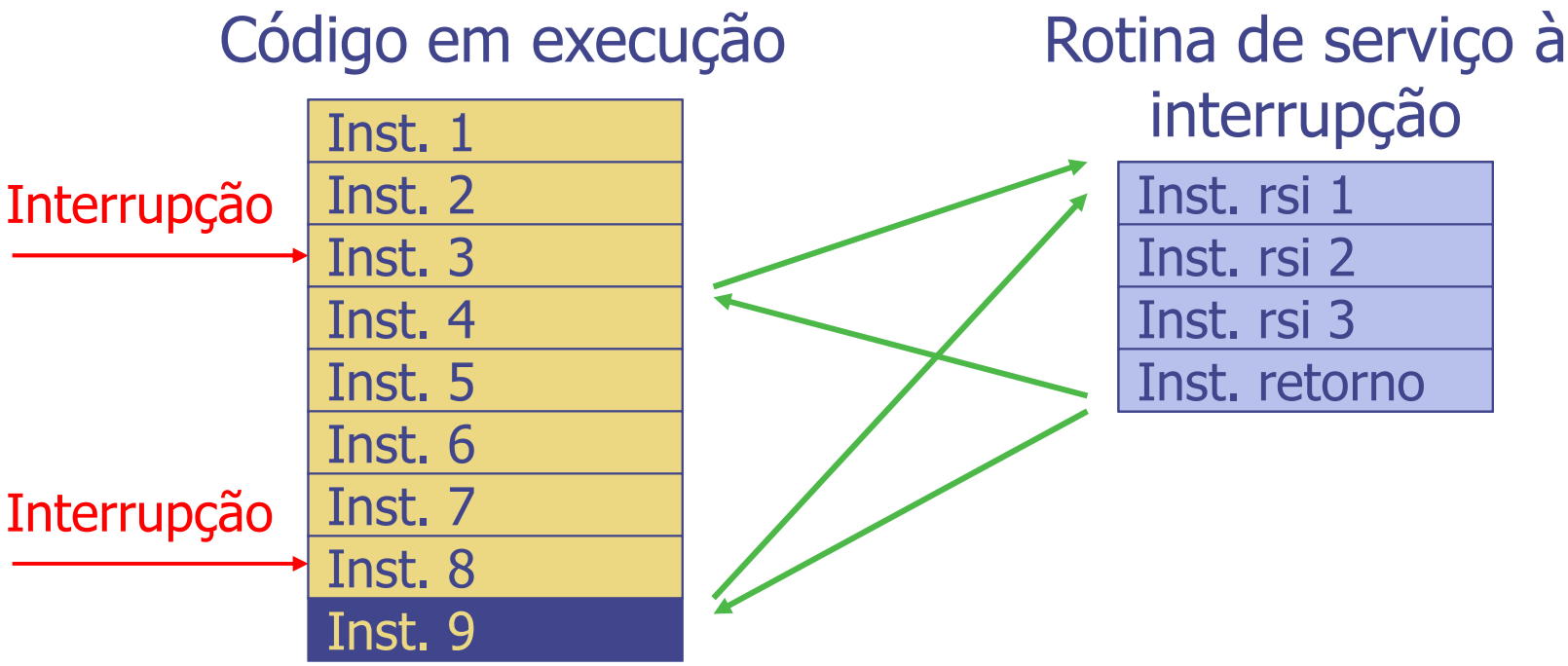
- Exemplos de dispositivos que podem gerar interrupções: teclado, rato, timers, dispositivos de comunicação, dispositivos de armazenamento, ...
- Exemplos de exceções:
  - Divisão por zero
  - *Overflow* numa operação aritmética
  - Tentativa de execução de uma instrução cujo OpCode é desconhecido
 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

Atendimento de interrupções e exceções

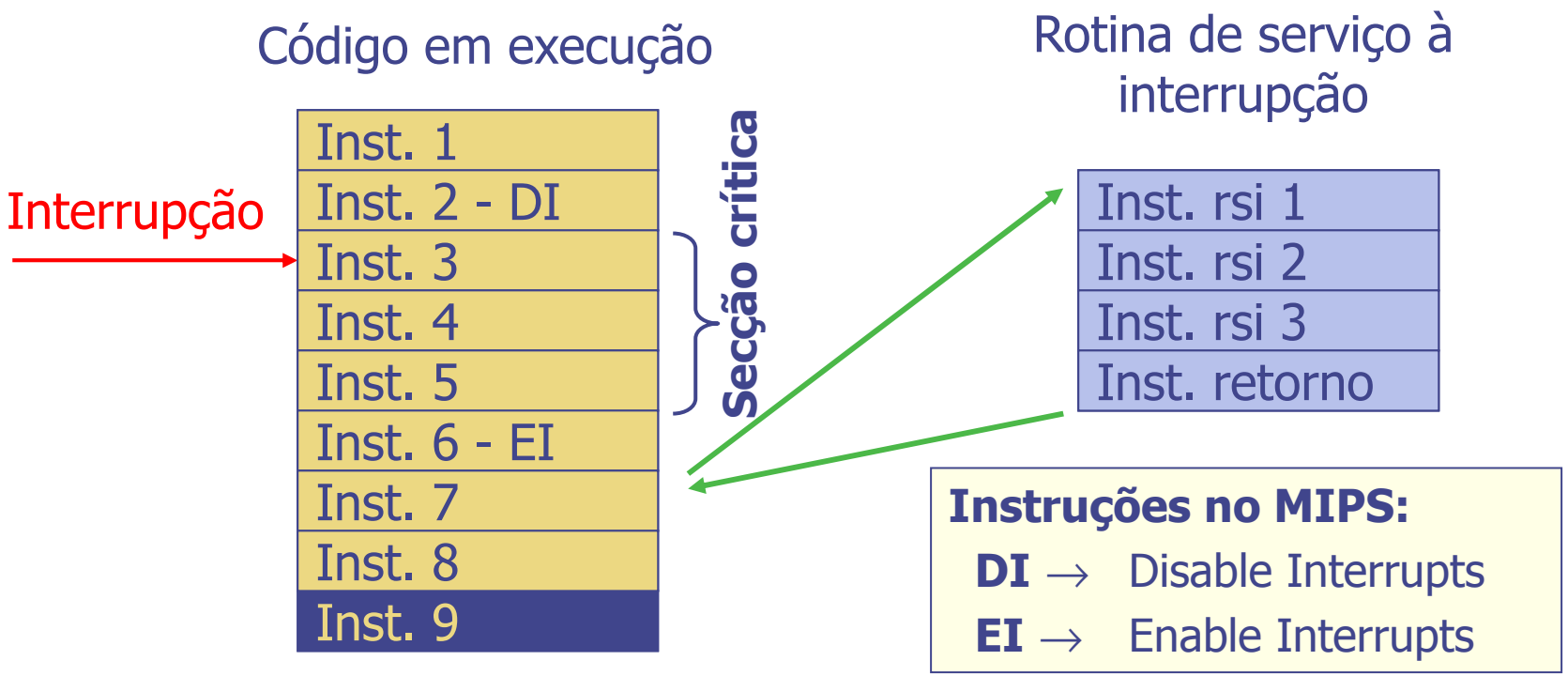
- **Exceções**: a instrução que gerou a exceção não termina, sendo a execução passada de imediato para a rotina de tratamento da exceção
- **Interrupções**: a passagem da execução para a rotina de tratamento da interrupção só acontece quando for concluída a instrução que está a ser executada no momento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

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

Processamento de interrupções



Ativação/desativação global das interrupções



- **Secção crítica**: uma sequência de instruções em que a ocorrência de uma interrupção interferiria de forma indesejada no funcionamento do sistema (exemplo: acesso a um recurso hw/sw partilhado)

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?

1)

Interrupt

```
(...)
la    $t0,var
lw    $t1,0($t0)
addi  $t1,$t1,1
sw    $t1,0($t0)
lw    $t2,0($t0)
```

2)

Interrupt

```
(...)
la    $t0,var
di
lw    $t1,0($t0)
addi  $t1,$t1,1
sw    $t1,0($t0)
ei
lw    $t2,0($t0)
```

```
isr:  # save context
      (...)
      la    $a0,var
      lw    $a1,0($a0)
      addi  $a1,$a1,-1
      sw    $a1,0($a0)
      # restore context
      (...)
      eret
```

Resposta:

- Caso 1: $t2 = 6
- Caso 2: $t2 = 5

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 ← Endereço da RSI, i.e., salto para a 1ª instrução da RSI)
  5. Execução da RSI até encontrar a instrução de retorno
  6. Execução da instrução de retorno da RSI (e.g. eret, no MIPS)
     - Reposição do contexto salvaguardado (PC e flags) e reativação das interrupções => regresso ao programa interrompido, com a execução da instrução que teria sido executada se não tivesse acontecido a interrupção

Processamento de interrupções pela RSI

- Ações gerais que devem ser implementadas na Rotina de Serviço à Interrupção (software):
  1. Salvaguarda do contexto do programa que foi interrompido:
     registos internos do CPU → memória (*stack*) ("**PRÓLOGO**")
  2. .. ações associadas ao processamento da interrupção..
  3. Reposição do contexto do programa interrompido:
     registos internos do CPU ← memória (*stack*) ("**EPILOGO**")
  4. Conclusão da RSI com a instrução de retorno (do tipo "Return From Interrupt / exception" – "**`eret`**" no caso do MIPS)

- **Latência da interrupção**: define-se como o tempo que decorre desde a ocorrência do evento que desencadeia a interrupção até à execução da primeira instrução "útil" da Rotina de Serviço à Interrupção (pontos 1 a 4 do slide anterior mais eventual conclusão de secção crítica do código, mais a salvaguarda do contexto)

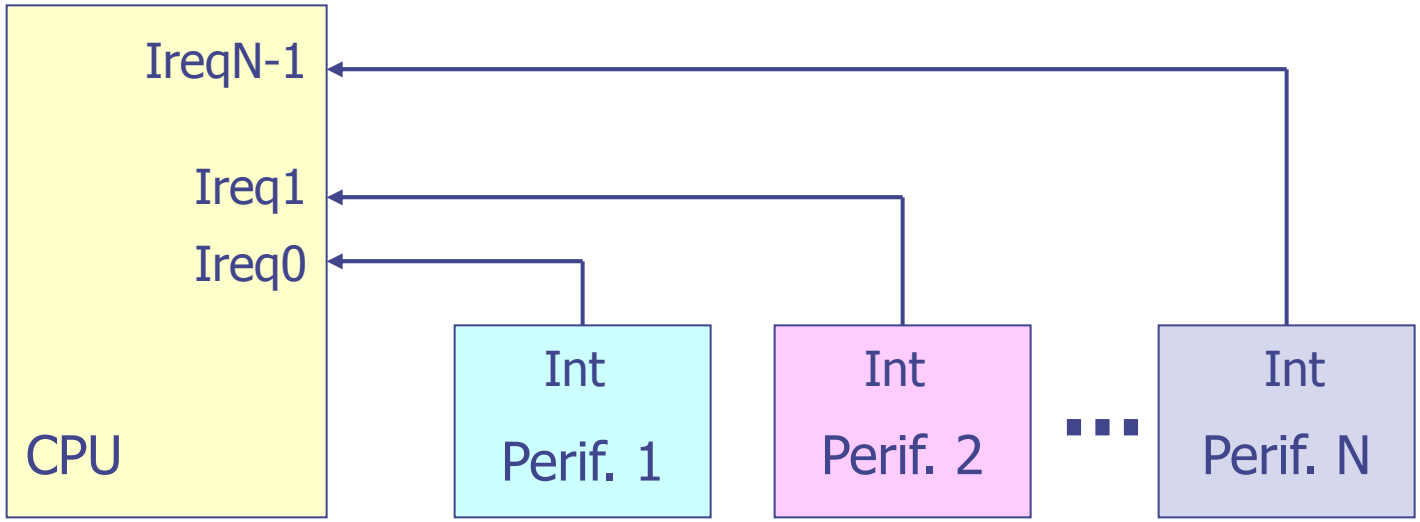
Overhead do método de transferência por interrupção

- O ***overhead*** global do método de transferência por interrupção é causado pela mudança de contexto:
  - A rotina de serviço à interrupção tem que, à entrada, **salvaguardar o contexto do programa interrompido**
  - Antes de abandonar a RSI tem que **repor o contexto salvaguardado**
  - A título de exemplo, estas 2 operações requerem, no MIPS 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

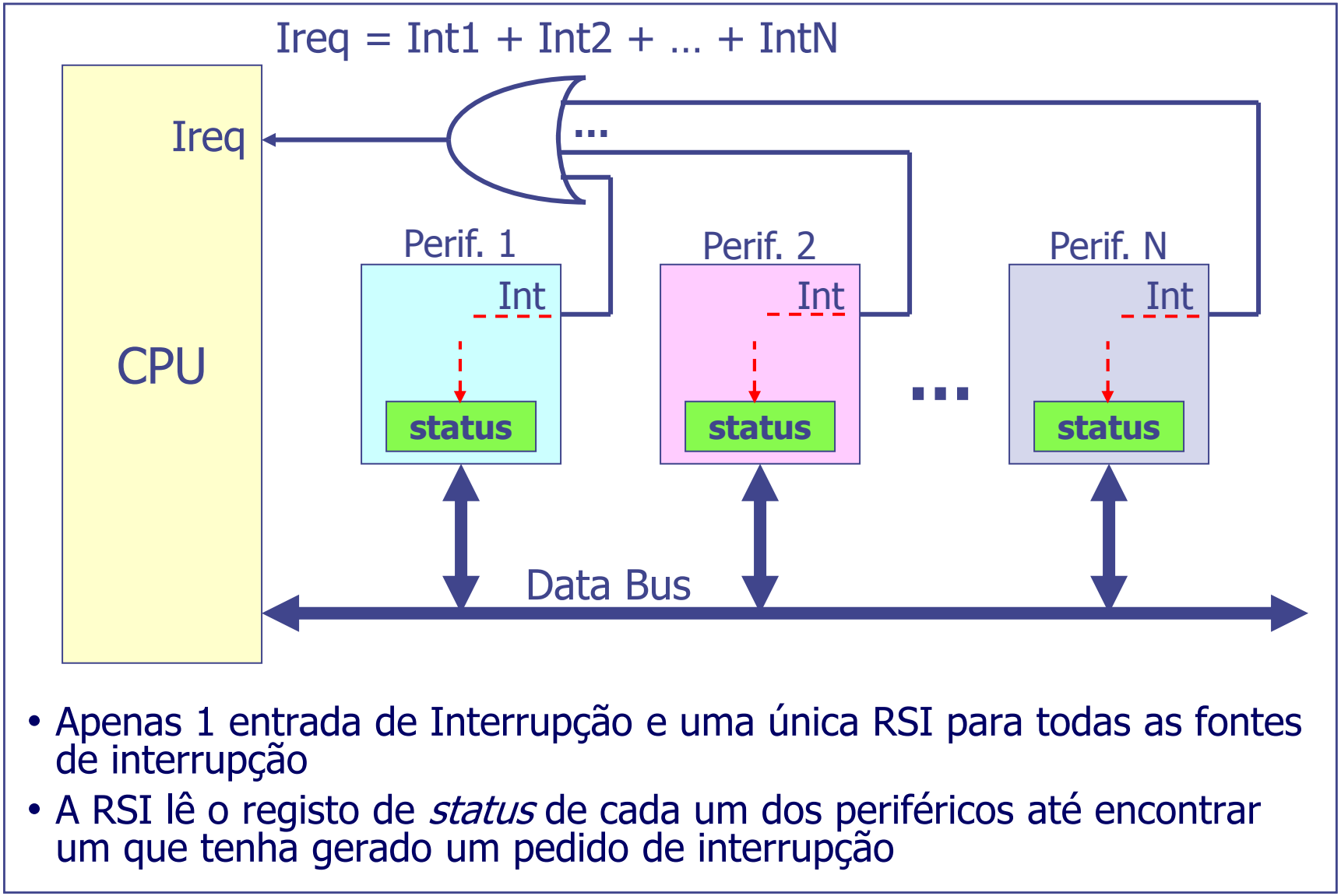
- Num sistema real é expectável que vários periféricos possam ter a capacidade de gerar interrupções
- Como organizar o sistema de interrupções para permitir a ligação de vários periféricos?
  - **Múltiplas linhas de interrupção**
  - **Identificação da fonte de interrupção por software**
  - **Interrupções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



- Apenas 1 entrada de Interrupção e uma única RSI para todas as fontes de interrupção
- A RSI lê o registo de *status* de cada um dos periféricos até encontrar um que tenha gerado um pedido de interrupção

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

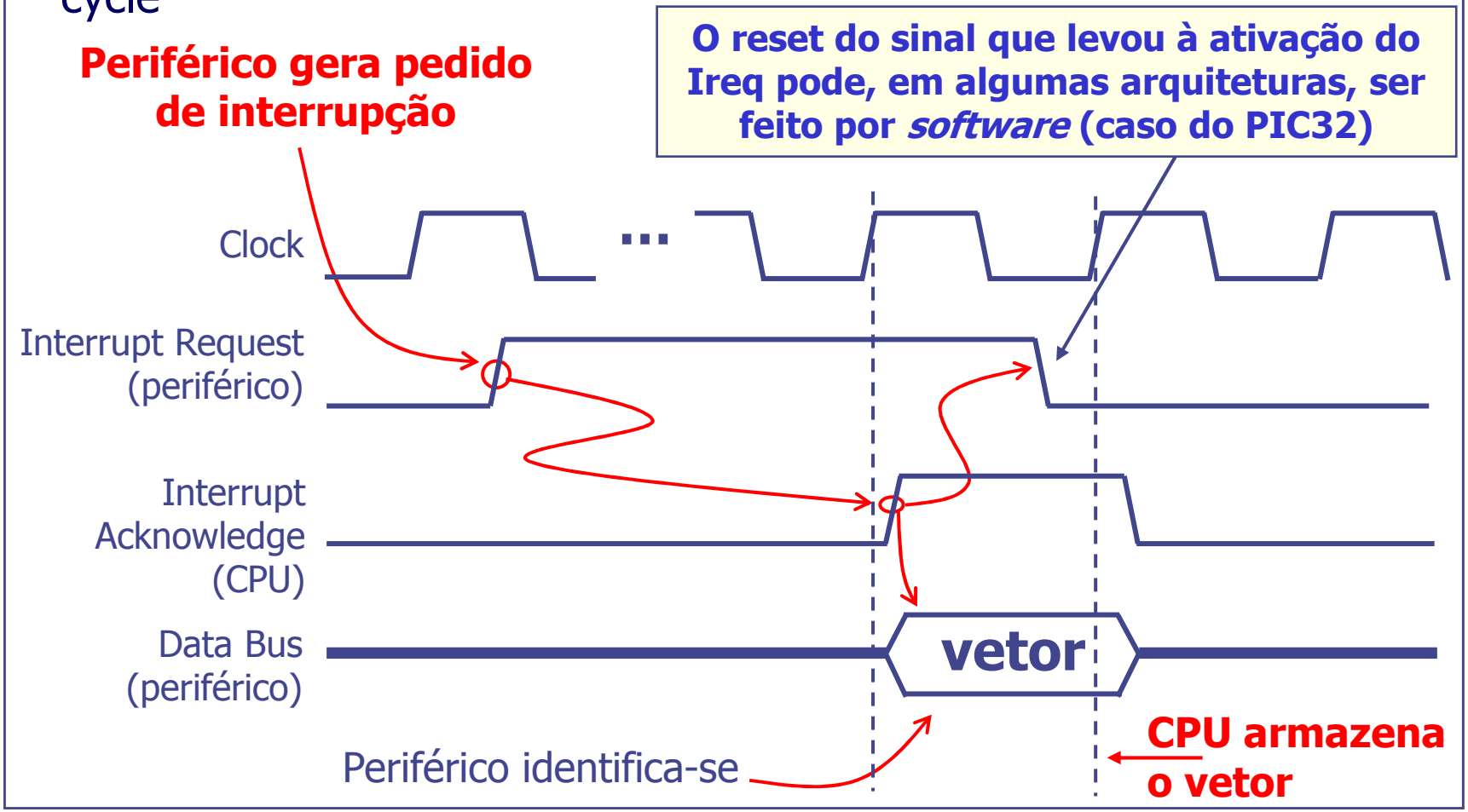
No caso de pedidos de interrupção simultâneos, a ordem pela qual os periféricos são "questionados" determina a prioridade no atendimento

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 o endereço de cada uma das RSI, ou instruções de salto incondicional para as RSI

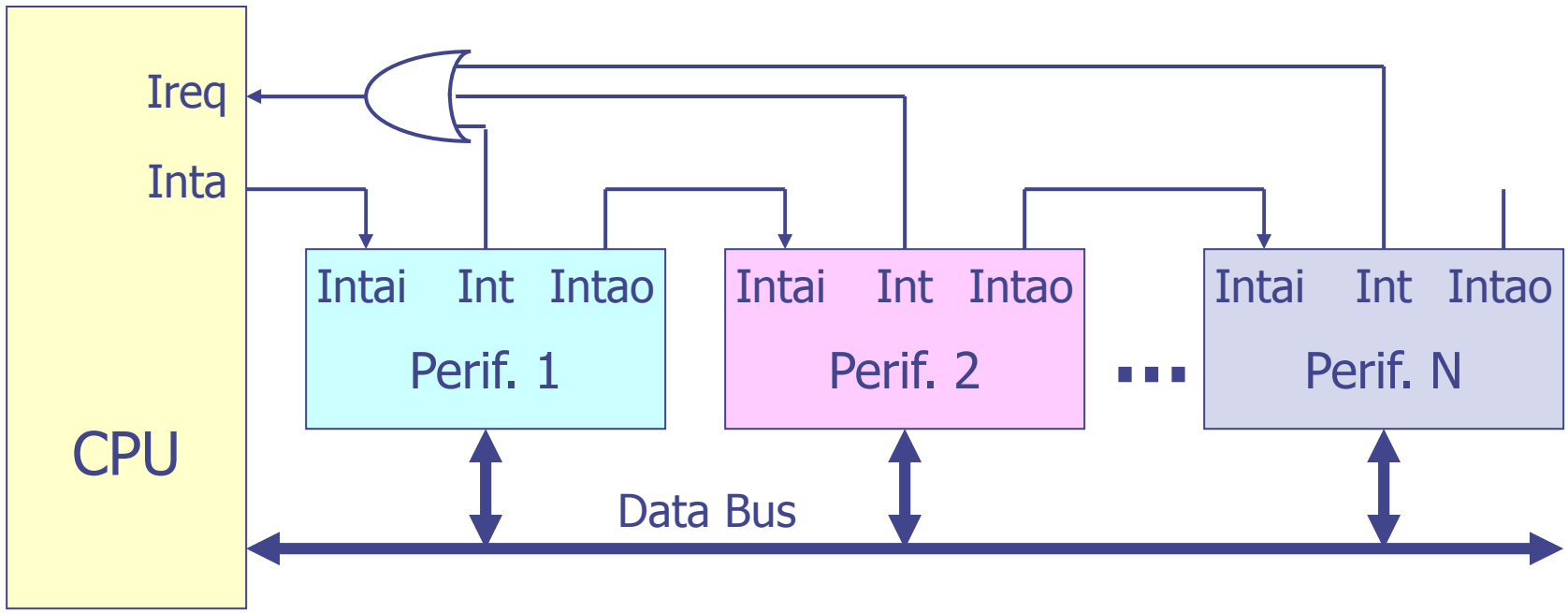
Interrupções vetorizadas

- 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*



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

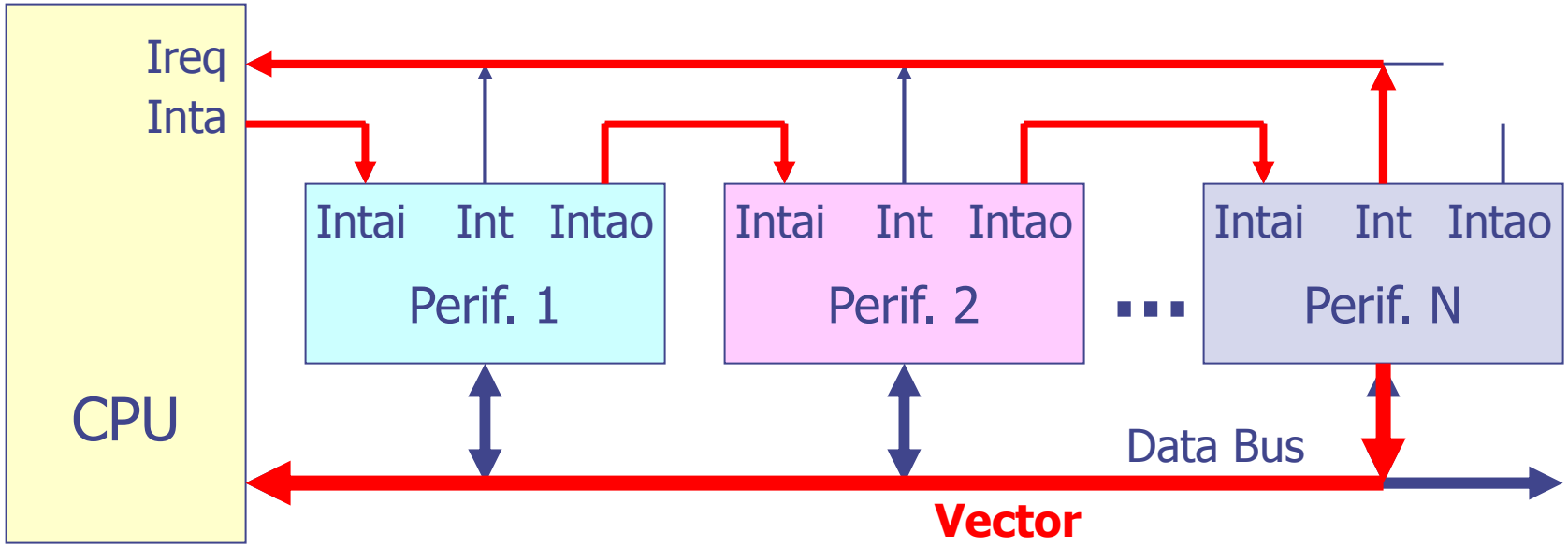
Intai/o - Interrupt Acknowledge in/out

Interrupções vetorizadas – *daisy chain*

- Genericamente, o procedimento de identificação da fonte de interrupção num esquema de interrupções vetorizadas em que os periféricos estão organizados numa cadeia *daisy chain* é o seguinte:
  1. Quando o CPU deteta o pedido de interrupção ("Ireq") e está em condições de o atender ativa o sinal "Interrupt Acknowledge" ("Inta")
  2. O sinal "Inta" percorre a cadeia até ao periférico que gerou a interrupção
  3. O periférico que gerou a interrupção coloca o seu identificador (vetor) no barramento de dados
  4. O CPU lê o vetor e usa-o como índice da tabela que contém os endereços das RSI ou instruções de salto para as RSI.

Interrupções vetorizadas – *daisy chain*

- Exemplo de identificação da fonte de interrupção

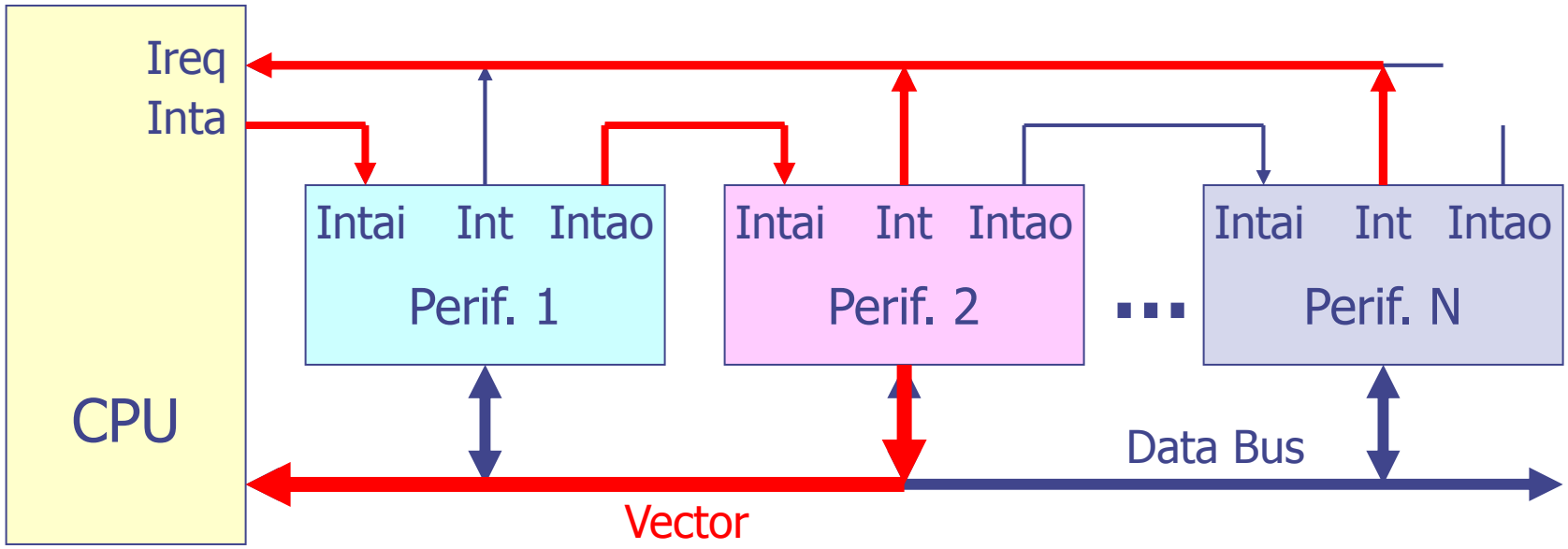


Ireq = Int1 + Int2 + … + IntN

Intai/Intao - Interrupt Acknowledge in/out

Interrupções vetorizadas – *daisy chain*

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

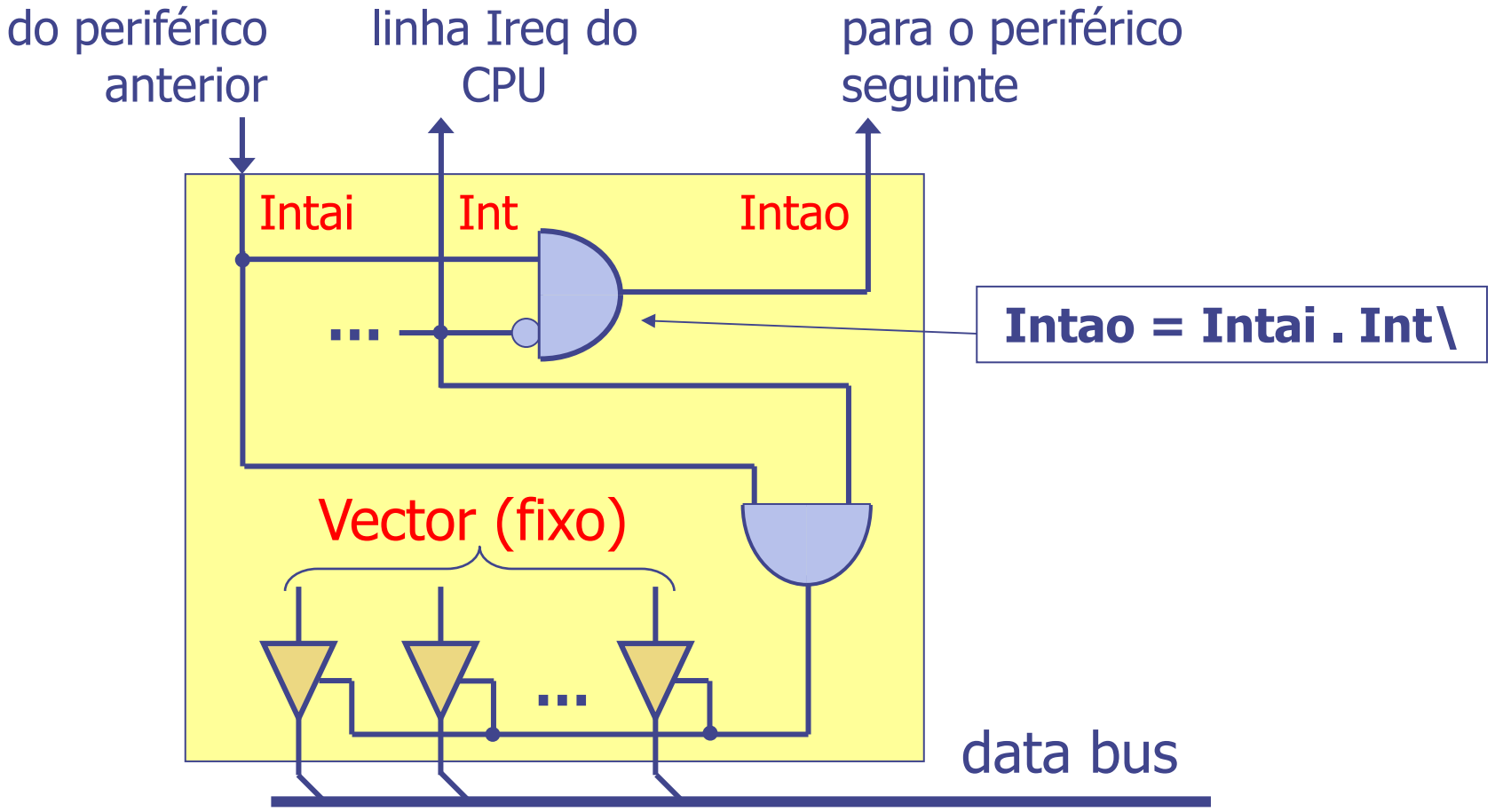


Ireq = Int1 + Int2 + … + IntN

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

Interrupções vetorizadas – *daisy chain*

- Estrutura típica do periférico (arbitragem e identificação)



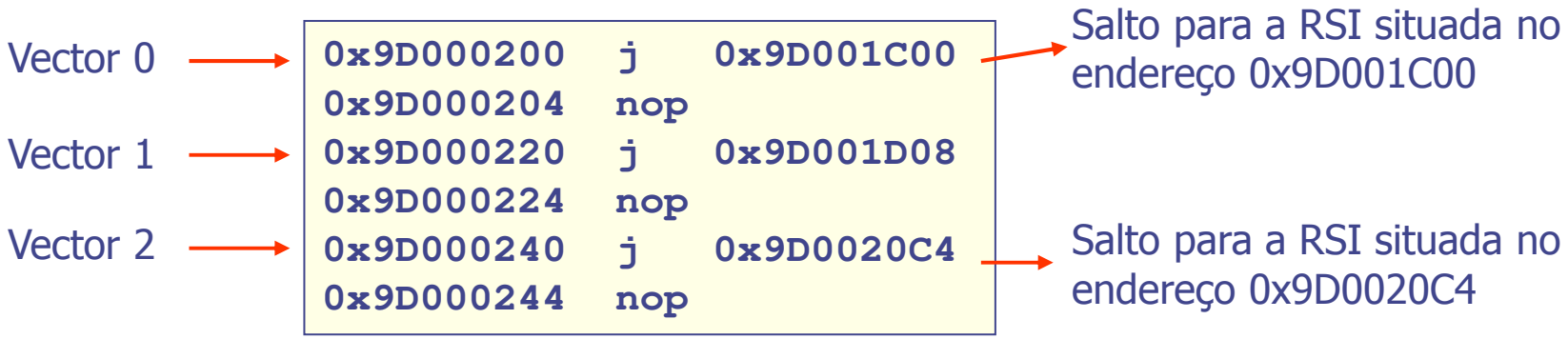
Interrupções vetorizadas – tabela de vetores

- Há duas formas de organizar a tabela de interrupções que associa um dado vetor a uma RSI
  1. A tabela é inicializada com os endereços de todas as RSI
  2. A tabela é inicializada com instruções de salto para as RSI

- **Tabela inicializada com os endereços de todas as RSI**: na fase inicial do processamento da interrupção o CPU acede à tabela, usando como índice o vetor
  - O valor lido da tabela é carregado no *Program Counter*
  - Este modelo é usado, por exemplo, na arquitetura Intel x86

Interrupções vetorizadas – tabela de vetores

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

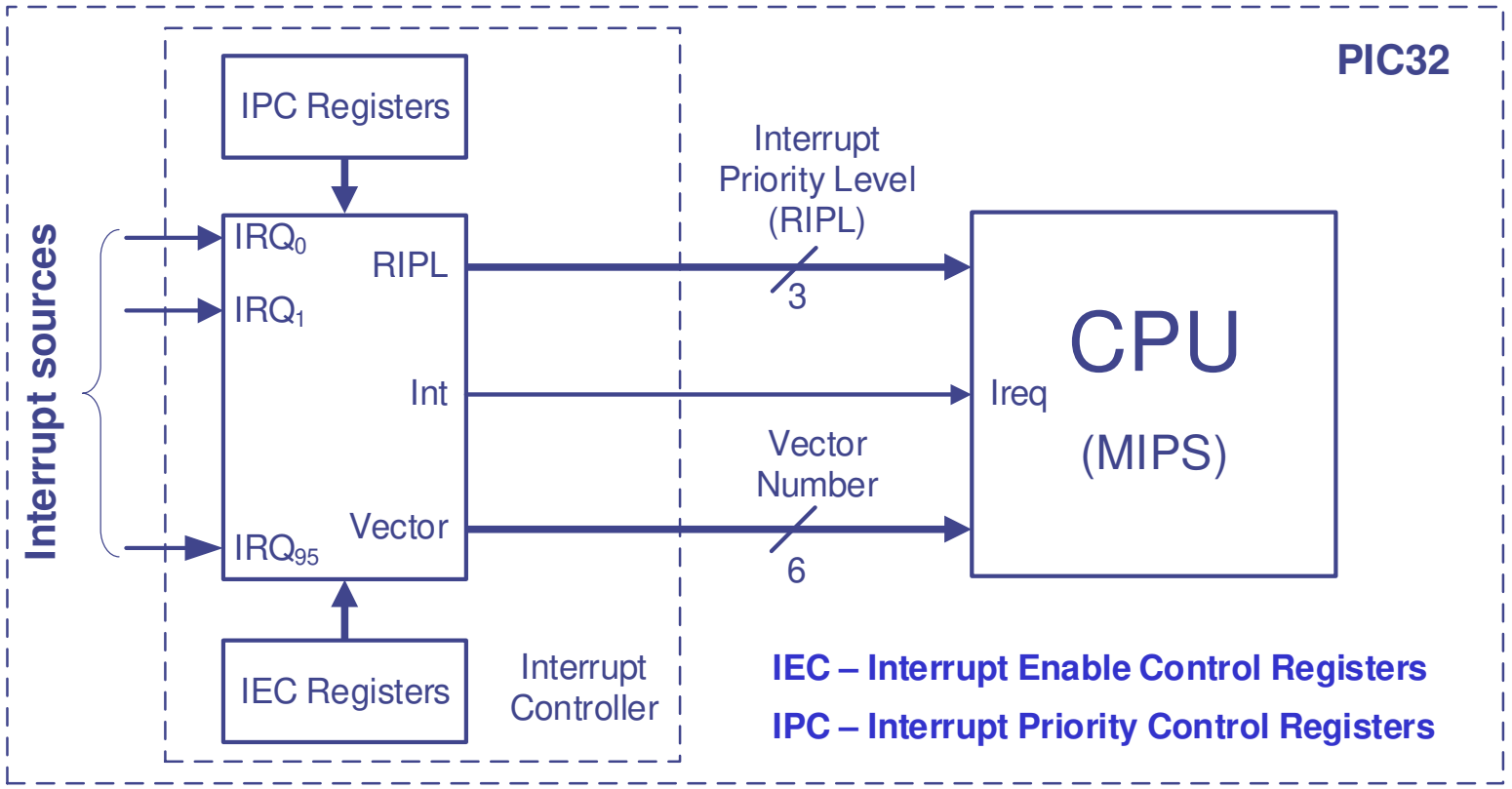


Interrupções no PIC32

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

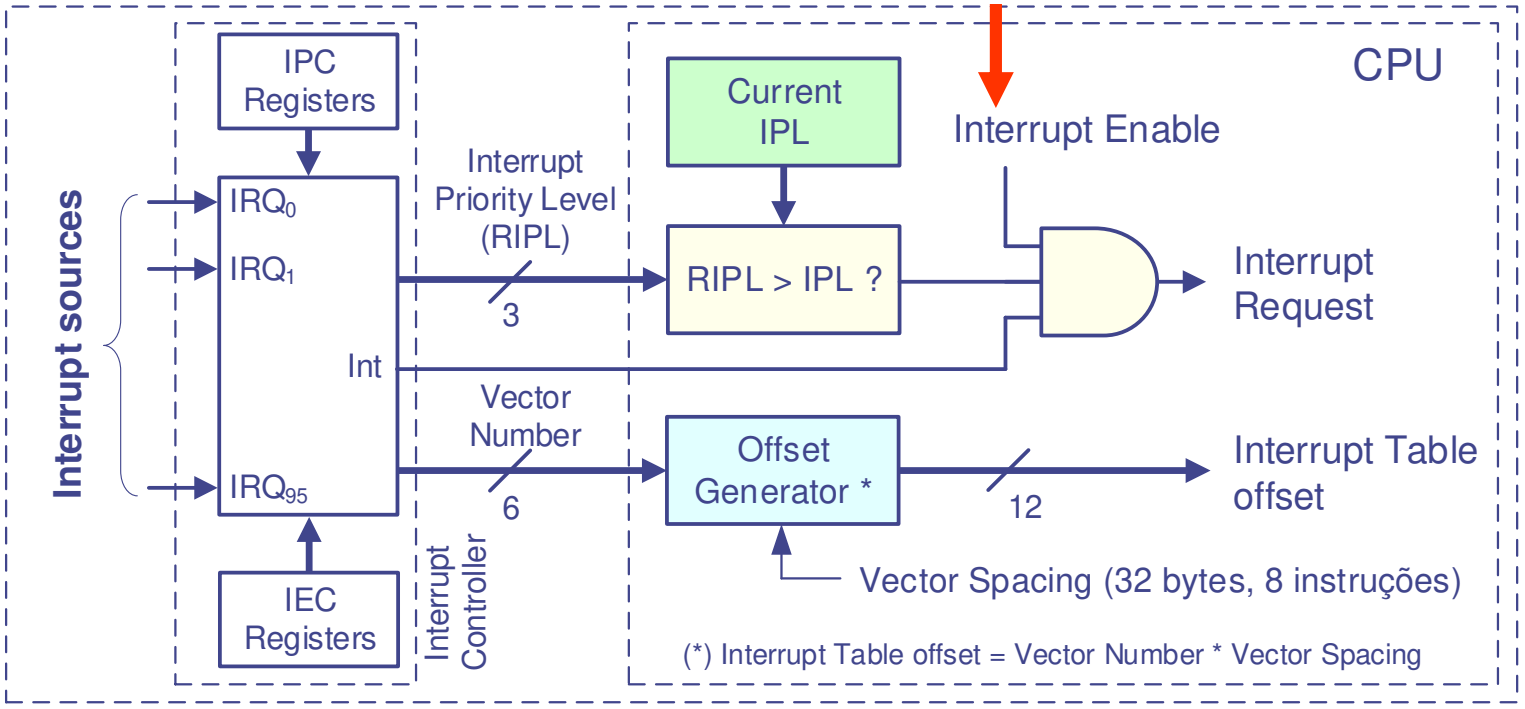
Interrupções no PIC32

- 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**
  - Registos através dos quais se pode habilitar / desativar (*enable* / *disable*) qualquer fonte de interrupção. Cada módulo do PIC32 que pode gerar interrupções usa 1 ou mais bits destes registos
- **IPC0**, **IPC1**, …, **IPC12** – Interrupt Priority **Control Registers**
  - Registos através dos quais se pode configurar, com 3 bits, a prioridade de cada uma das fontes de interrupção (0 a 7)
- **IFS0**, **IFS1**, **IFS2** – Interrupt Flag **Control** / **Status Registers**
  - Flags de sinalização da ocorrência de interrupções, de todas as fontes possíveis. Cada módulo do PIC32 que pode gerar interrupções usa 1 ou mais bits destes registos
- **INTCON** – Interrupt **Control Register**
  - Configura a polaridade da transição 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 em um dos registos **IECx** (Interrupt Enable Control Registers), através do qual se pode fazer o *enable* ou o *disable* de uma dada fonte de interrupção. O nome do bit é, normalmente, formado pela sigla identificativa da fonte, terminada com o sufixo –**IE** (e.g. **T1IE**, *Timer1 Interrupt Enable*)
- **Priority Level** – 3 bits definidos em um dos registos **IPCx** (Interrupt Priority Control Registers), designado com o sufixo –**IP** (e.g. **T1IP**, *Timer1 Interrupt Priority*)
  - 7 níveis de prioridade (1 a 7); a prioridade 0 significa fonte *disabled*
- **Interrupt Flag** – bit definido em um dos registos **IFSx** (Interrupt Flag Status Registers) e designado com o sufixo –**IF** (e.g. **T1IF**, *Timer1 Interrupt Flag*). Este bit é ativado automaticamente quando ocorre uma interrupção. A desativação é da responsabilidade do programador

Exemplo de uma tabela de vetores no PIC32

```
#vector_7   (INT1, External Interrupt 1)
0x9D0002E0      0x0B40074D   j      0x9D001D34
0x9D0002E4      0x00000000   nop

#vector_8   (T2 - Timer2)
0x9D000300      0x0B4006C3   j      0x9D001B0C
0x9D000304      0x00000000   nop

#vector_19 (INT4 - External Interrupt 4)
0x9D000460      0x0B40077A   j      0x9D001DE8
0x9D000464      0x00000000   nop
```

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

Rotina de Serviço à Interrupção no PIC32

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

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

Interrupt Function

Vector Number

Function Name

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

Exemplo de programação com interrupções no PIC32

```
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_t2(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, Arnaldo Oliveira, Tomás Silva, Bernardo Cunha

Introdução

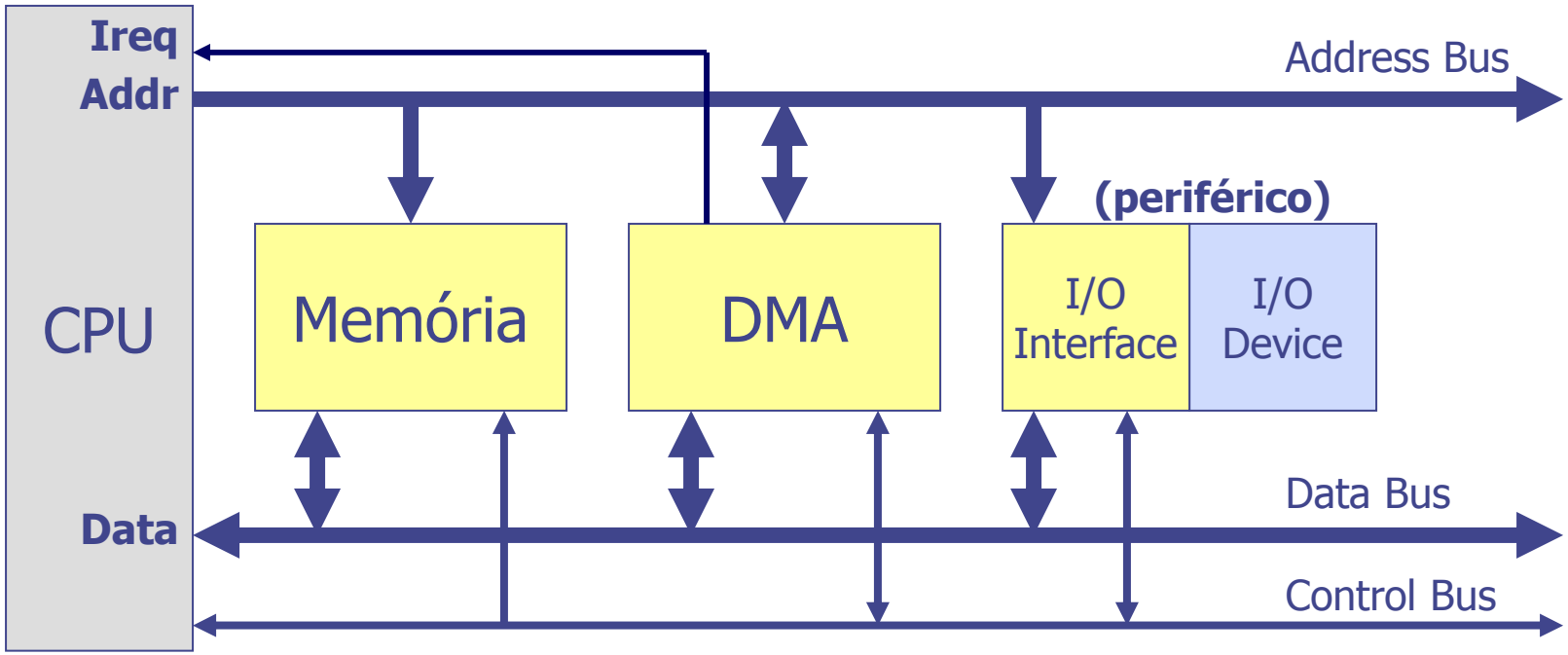
- O problema da (possível) longa latência apresentada pelos periféricos é adequadamente tratada pela técnica de E/S por interrupção
- Esta técnica não resolve, contudo, a questão da transferência a taxas elevadas, uma vez que o limite é sempre imposto pelo facto de o CPU ter que executar um programa para efetuar a transferência
- Transferir informação de um periférico para a memória implica, do ponto de vista temporal:
  - O tempo de latência de resposta à interrupção
  - 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 registos 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 o problema identificado é designada por **DMA** (do inglês *Direct Memory Access*) e consiste na transferência de informação do periférico diretamente para a memória, sem utilizar o 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?

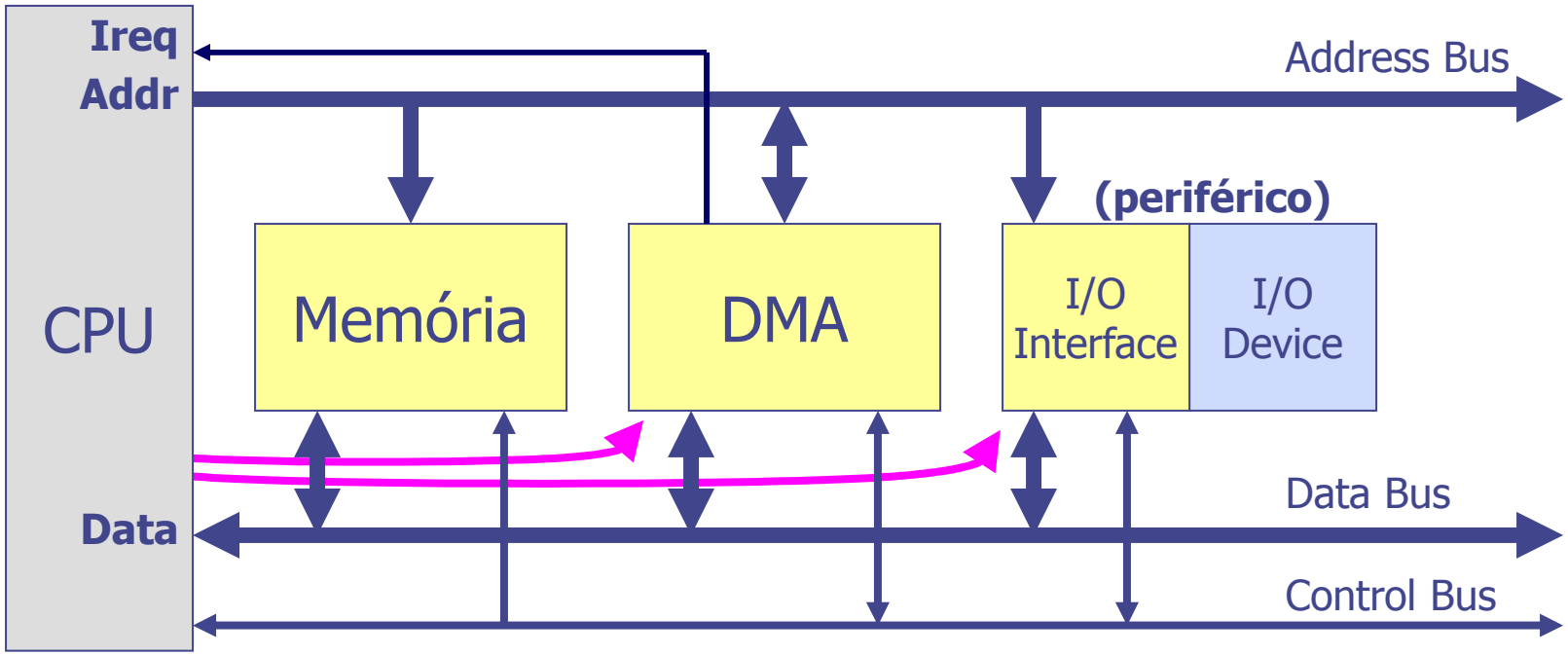
Transferência por Acesso Direto à Memória (DMA)

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



Transferência por Acesso Direto à Memória (DMA)

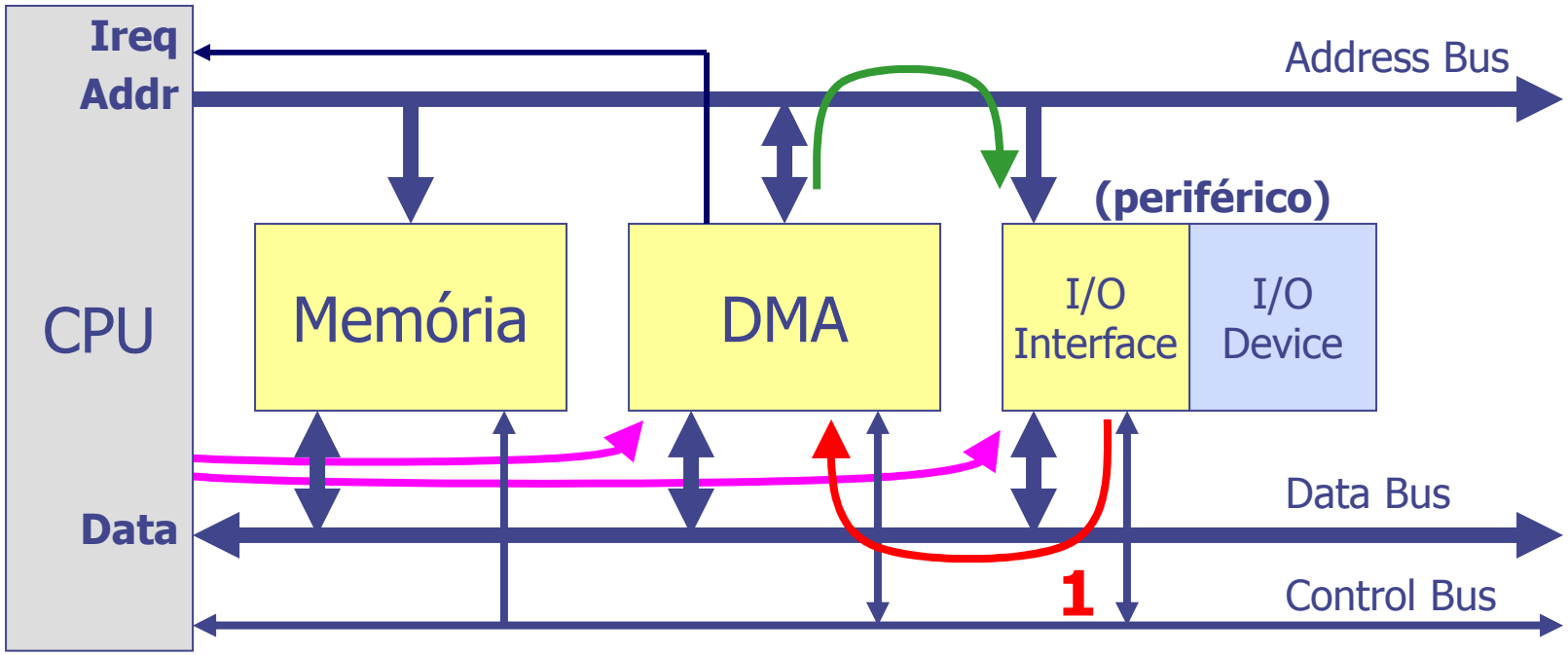
- Transferência de dados entre a memória e um periférico sem a intervenção do CPU. Um periférico especializado (DMA – DMA Controller) efetua essa transferência



- CPU programa o DMA e o periférico para efetuar uma transferência entre o mesmo e a memória

Transferência por Acesso Direto à Memória (DMA)

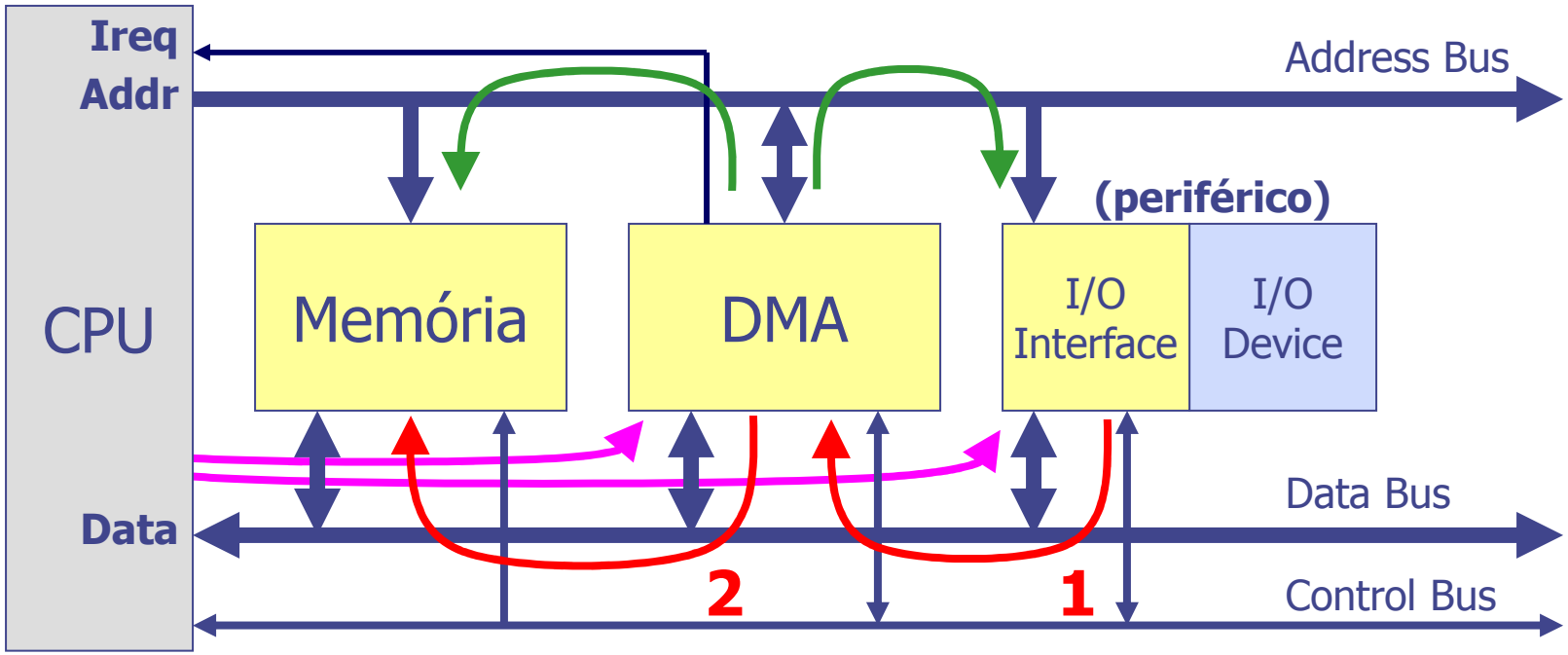
- Transferência de dados entre a memória e um periférico sem a intervenção do CPU. Um periférico especializado (DMA – DMA Controller) efetua essa transferência



- DMA lê um valor do periférico e armazena-o num registo interno

Transferência por Acesso Direto à Memória (DMA)

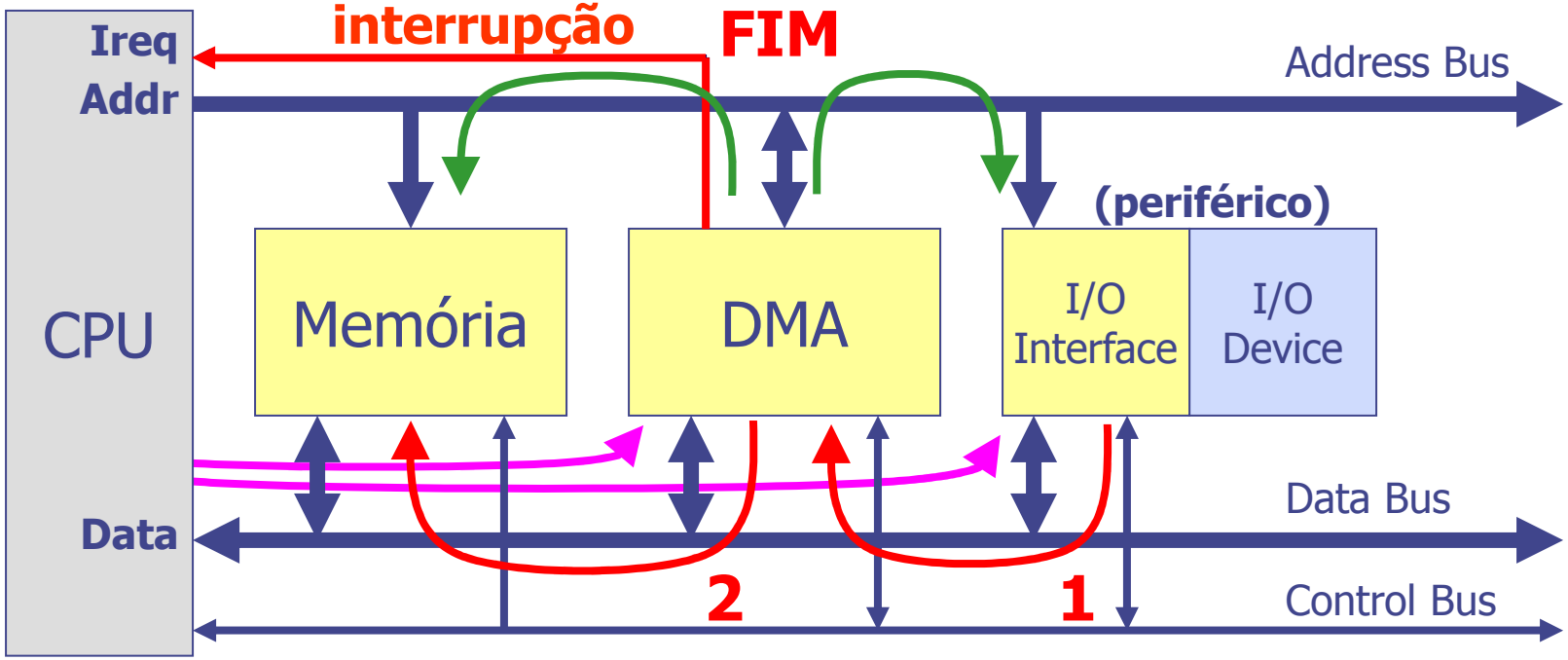
- Transferência de dados entre a memória e um periférico sem a intervenção do CPU. Um periférico especializado (DMA – DMA Controller) efetua essa transferência



- DMA copia o valor armazenado no seu registo interno para a memória

Transferência por Acesso Direto à Memória (DMA)

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



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

Controlador de DMA

- Um 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 DMA é um controlador especializado na transferência de dados (cópia), de forma autónoma, em hardware, após programação feita pelo CPU
- Durante a transferência, o DMA tem a capacidade de controlar os barramentos de endereços, dados e controlo, como se fosse um CPU

Controlador de DMA

- Para efetuar uma transferência de um bloco de dados de **n** palavras, o **DMA** realiza, no essencial, ao nível dos barramentos, as mesmas operações temporalmente sequenciais 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 DMA necessita de **controlar os barramentos** de **endereços** e de **dados** e ainda os sinais **rd** e **wr**
- Note-se que a capacidade do DMA para gerir os barramentos do sistema entra em conflito com capacidade idêntica do CPU

Controlador de DMA

- Apenas um dos dois dispositivos, CPU e DMA, 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 DMA necessita de aceder aos barramentos para realizar uma transferência, tem que primeiro ter autorização do CPU para ser o "bus master"
- O DMA 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 DMA é sempre temporário

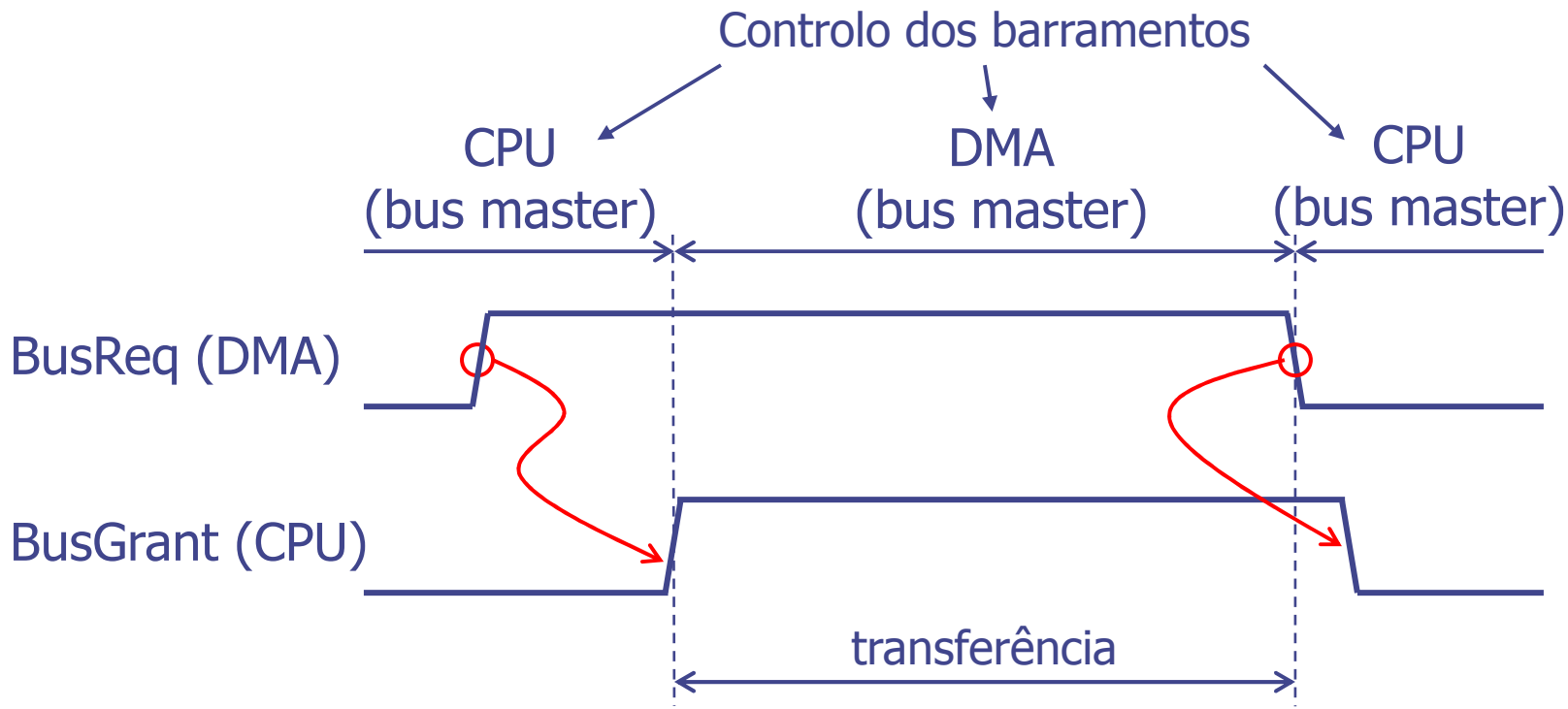
- Quais são então os passos que o DMA tem que seguir para fazer uma transferência de dados ?

Controlador de DMA – passos para uma transferência

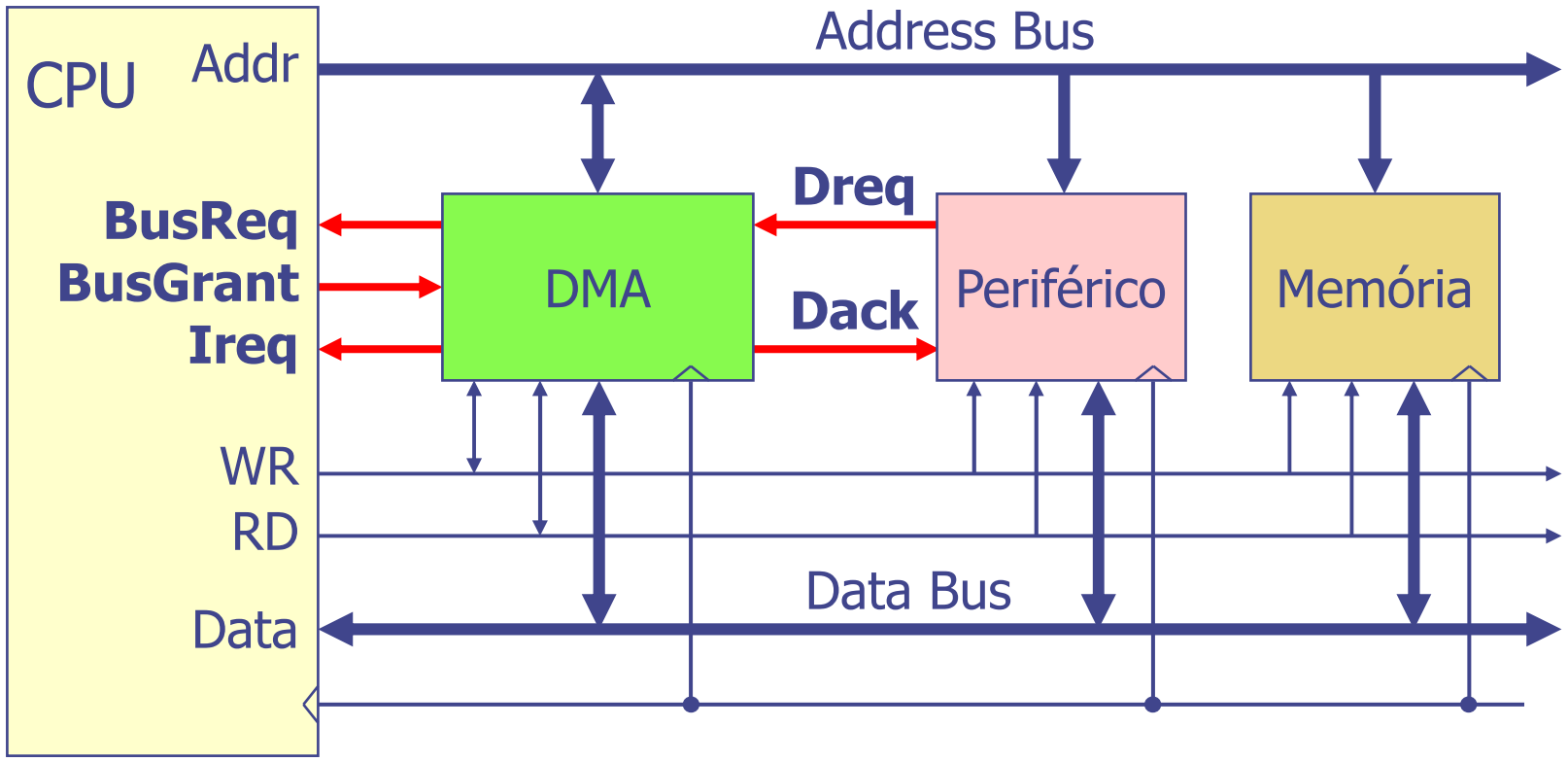
- DMA pede ao CPU para ser *bus master*
- Espera até ter a confirmação do CPU
- Efetua a transferência:
  - Lê uma palavra (*byte* ou *word*) do dispositivo fonte (do *source address*) para um registo interno
  - Escreve a palavra guardada no registo interno, no passo anterior, no dispositivo destino (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 DMA:
  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



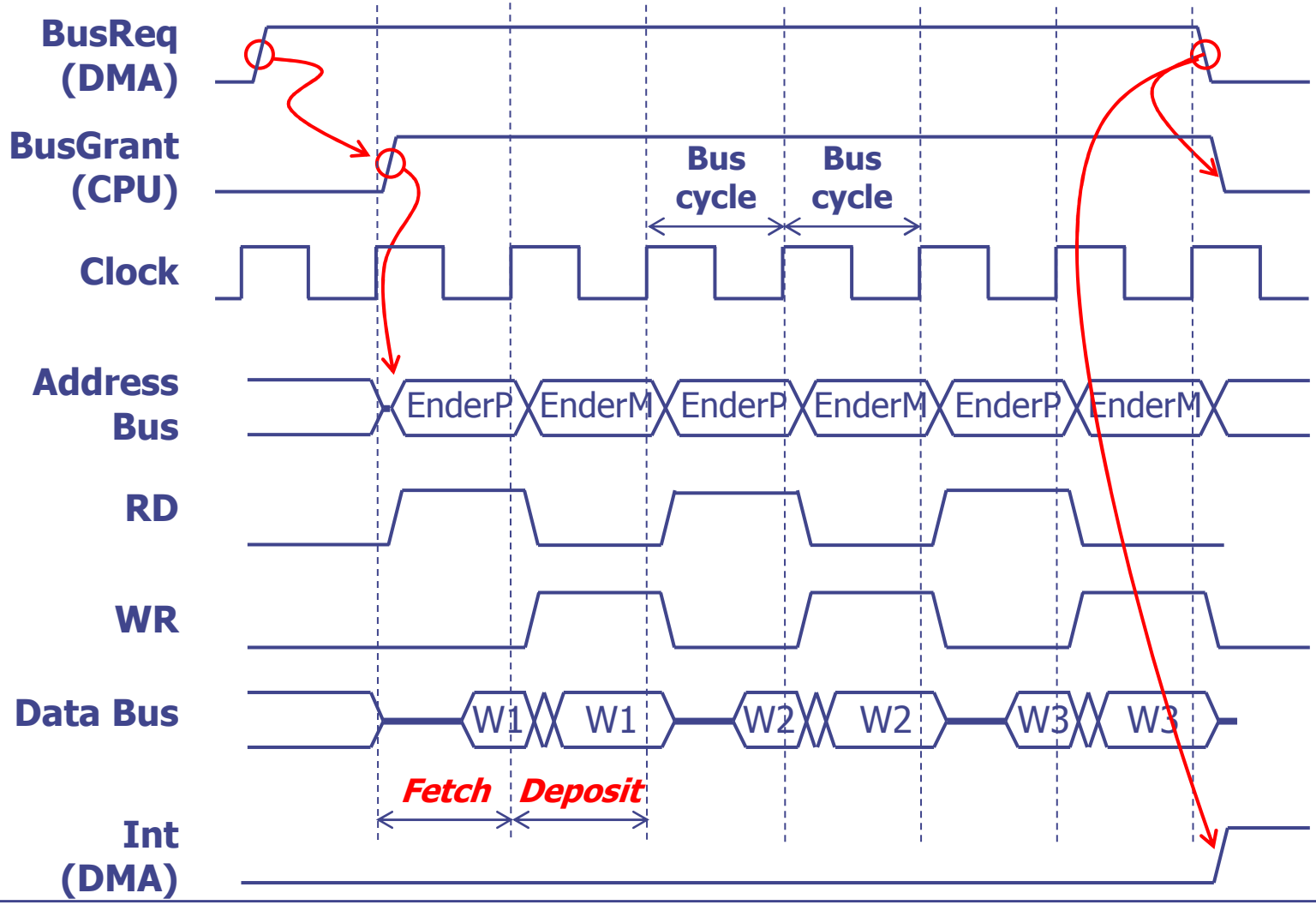
- Dreq – DMA Request
- Dack – DMA Acknowledge
- BusReq – Bus Request
- BusGrant – Bus Grant

DMA – Modos de operação

- **Bloco** – o DMA assume o controlo dos barramentos até todos os dados terem sido transferidos
- **Burst** (rajada)
  - O DMA 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 DMA a desativar o sinal BusReq e a libertar os barramentos
    - Logo que o periférico ative de novo o sinal Dreq o DMA volta a ativar o sinal BusReq e, logo que seja "bus master", continua no ponto onde interrompeu
- **Cycle Stealing**
  - O DMA 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 DMA no desempenho do CPU é nulo - o DMA 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



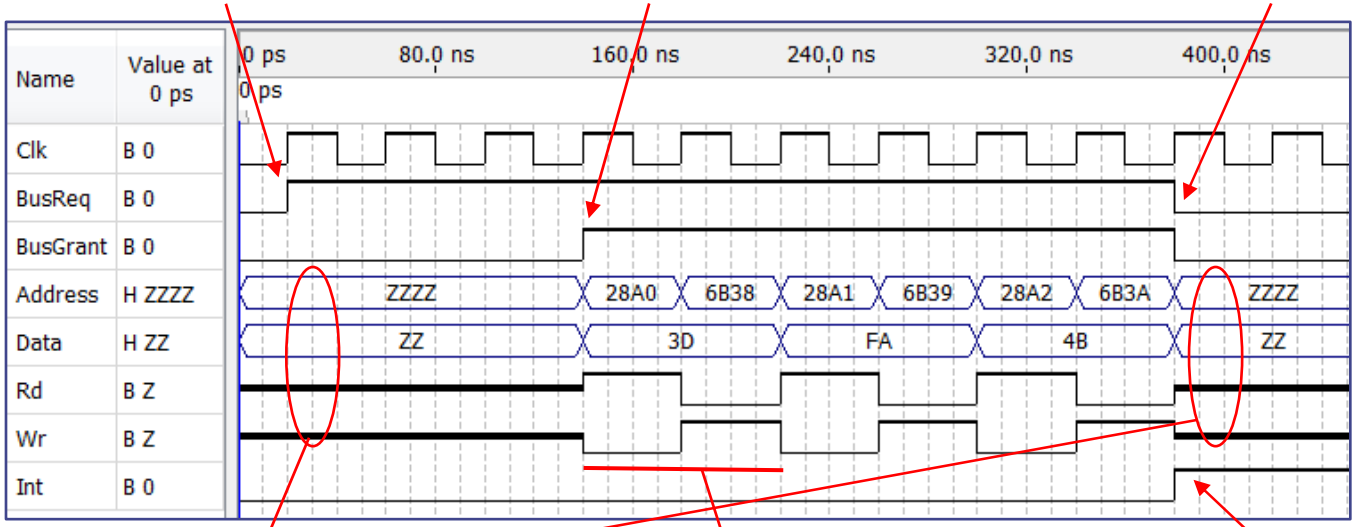
Modo Bloco (exemplo)

- Exemplo de uma transferência de 3 bytes:
  - Endereço de início na origem: 0x28A0, [0x28A0, 0x28A2]
  - Endereço de início no destino: 0x6B38, [0x6B38, 0x6B3A]

DMA ativa BusReq

CPU responde com BusGrant

DMA desativa BusReq

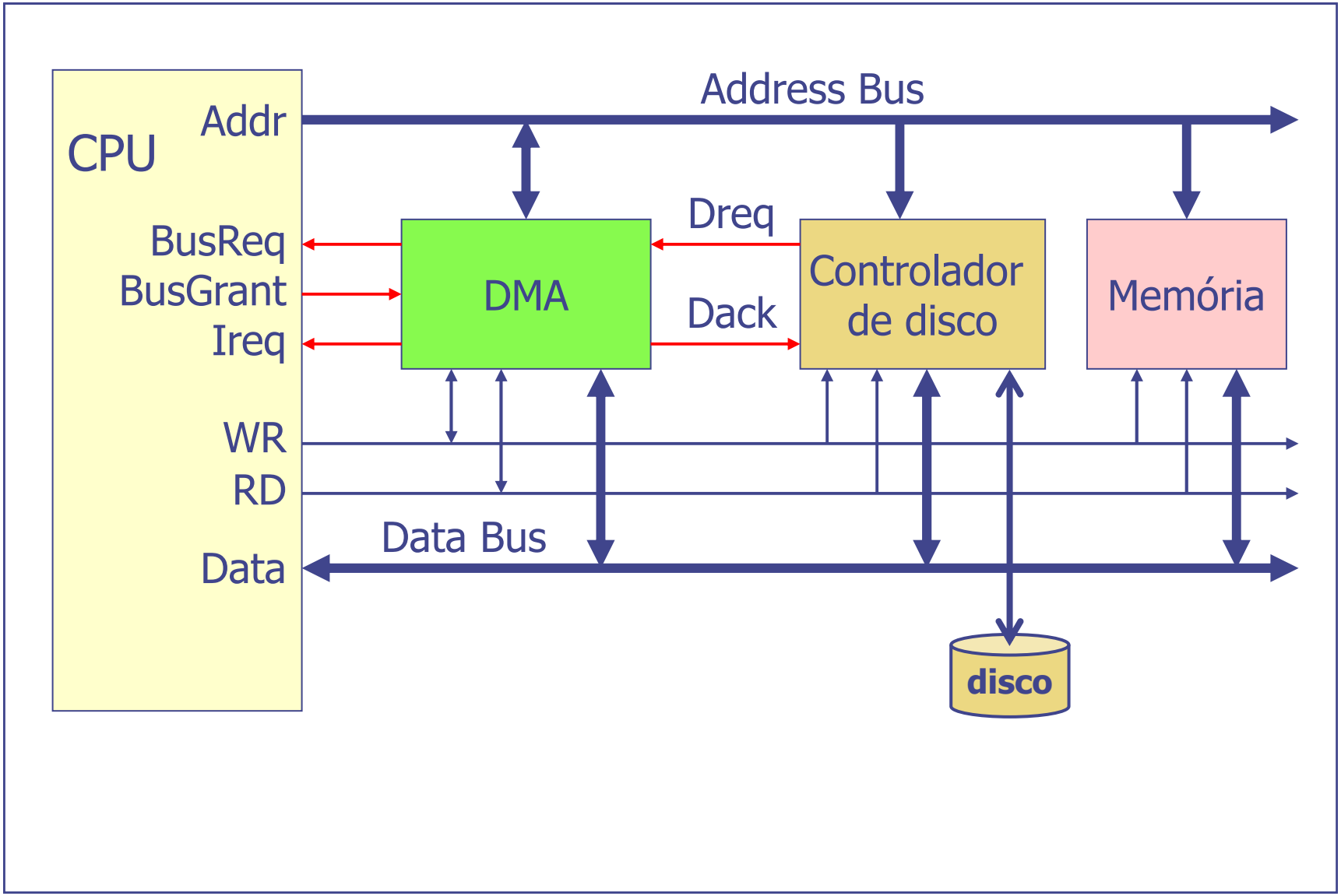


Barramentos do DMA em alta impedância

Transferência do 1º byte

DMA gera interrupção no fim da transferência

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 DMA: 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 DMA (sinalizando dessa forma o DMA de que a informação está pronta para ser transferida)
5. O DMA 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 DMA passou a ser o *bus master*)
7. O DMA ativa o sinal **Dack** e o DiskCtrl, em resposta, desativa o sinal **Dreq**

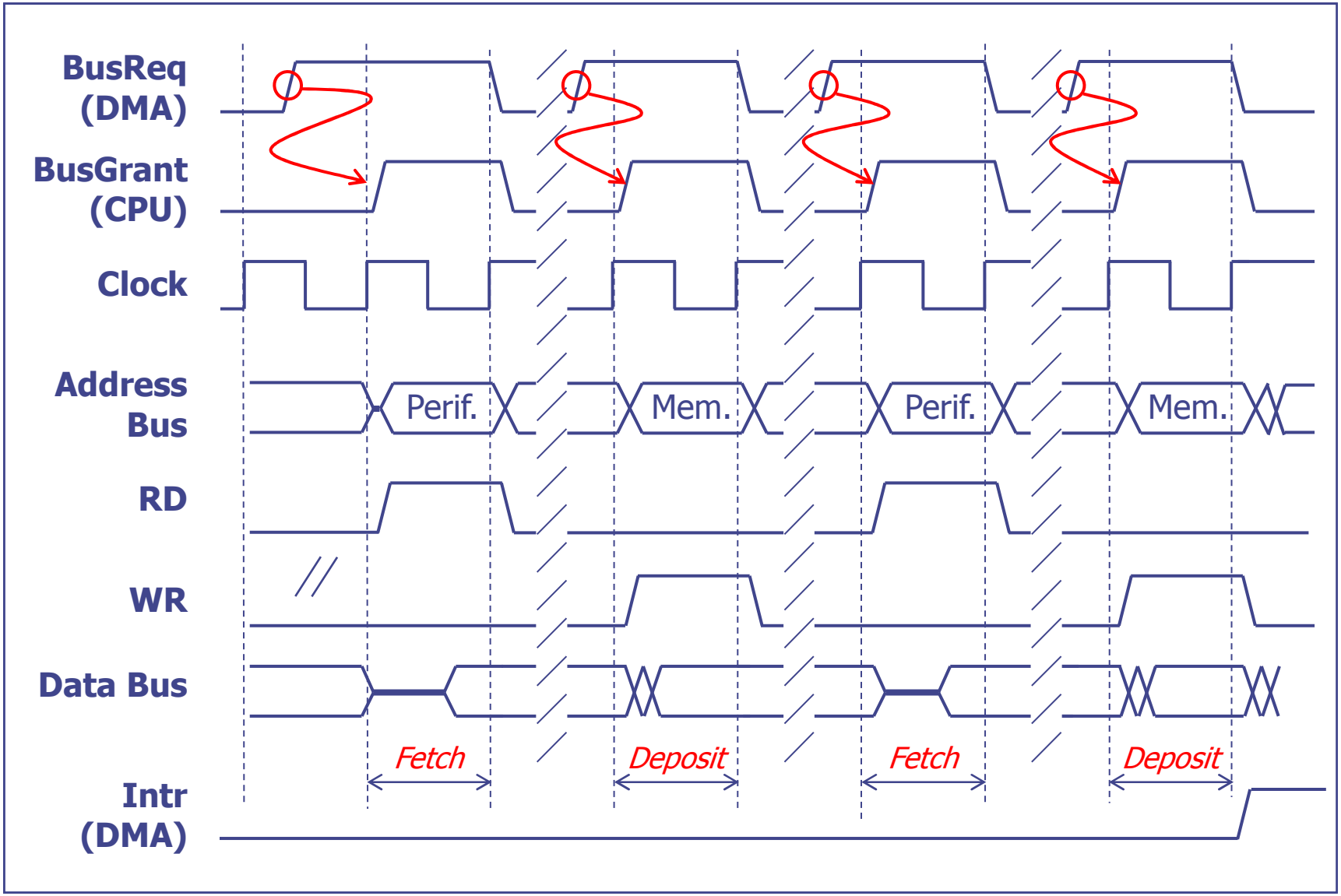
Modo Bloco – Exemplo de uma transferência por DMA

8. O DMA 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 DMA 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 DMA

Modo "Cycle-Stealing"

- Numa transferência em modo "*cycle-stealing*", o DMA 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)

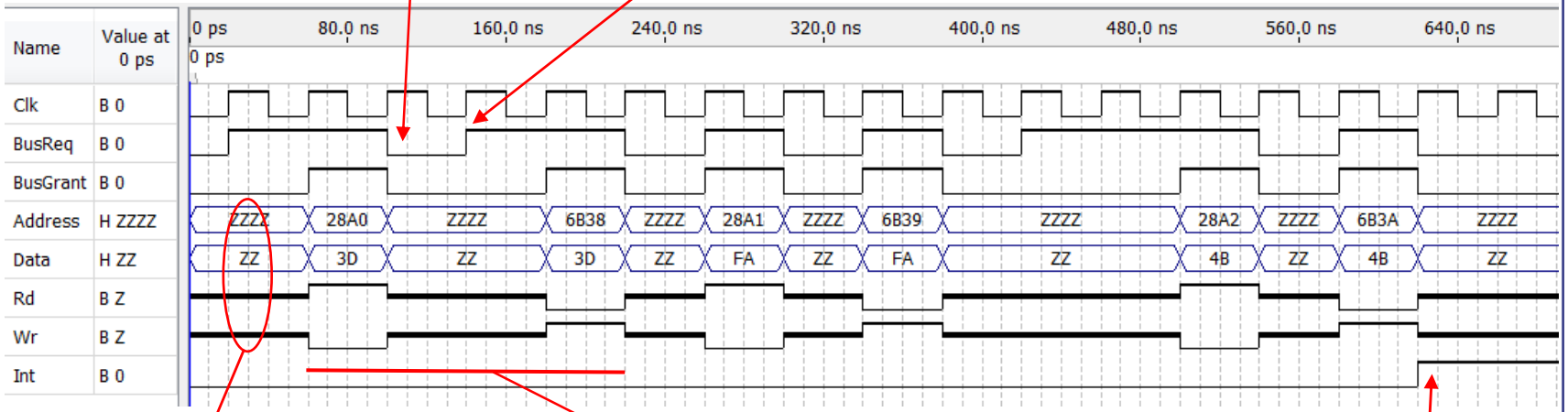


Modo "Cycle-Stealing" (exemplo)

- Exemplo de uma transferência de 3 bytes:
  - Endereço de início na origem: 0x28A0, [0x28A0, 0x28A2]
  - Endereço de início no destino: 0x6B38, [0x6B38, 0x6B3A]

DMA desativa BusReq

DMA reativa BusReq

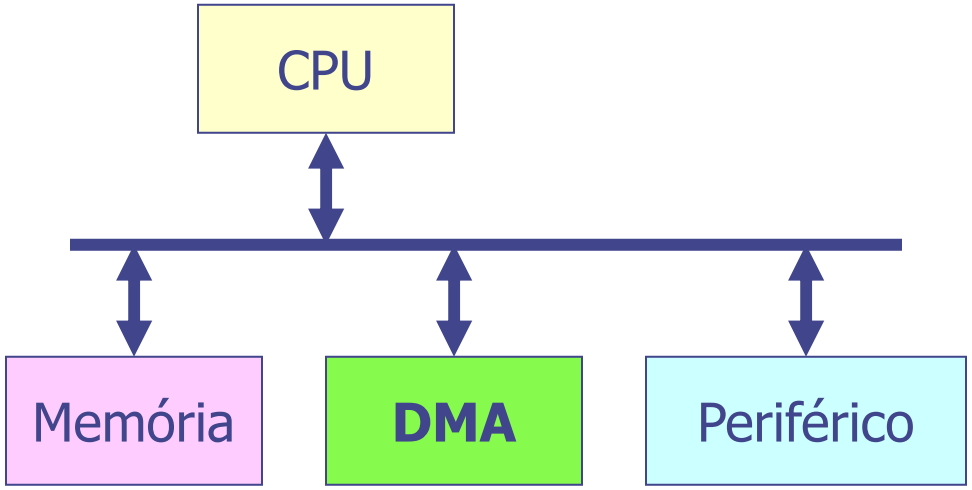


Barramentos do DMA em alta impedância

Transferência do 1º byte

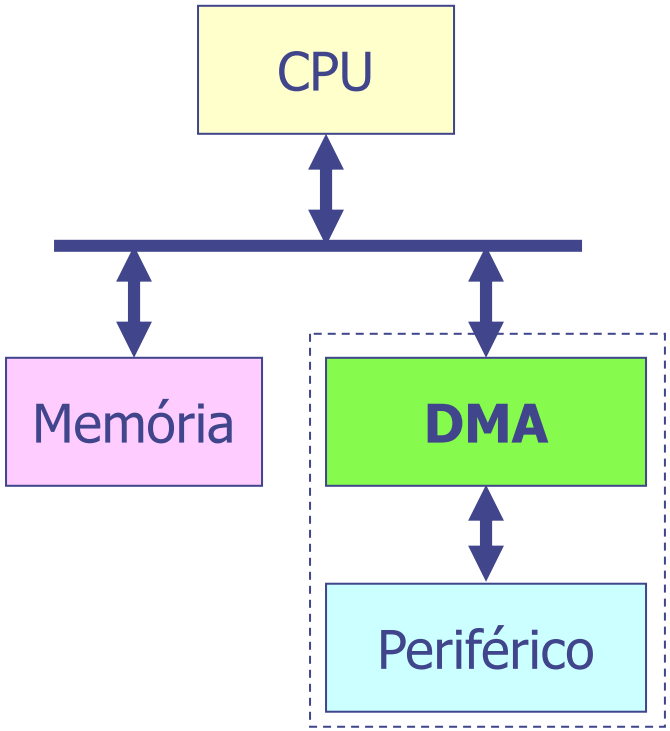
DMA gera interrupção no fim da transferência

Configurações – Canal de DMA



- O DMA 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

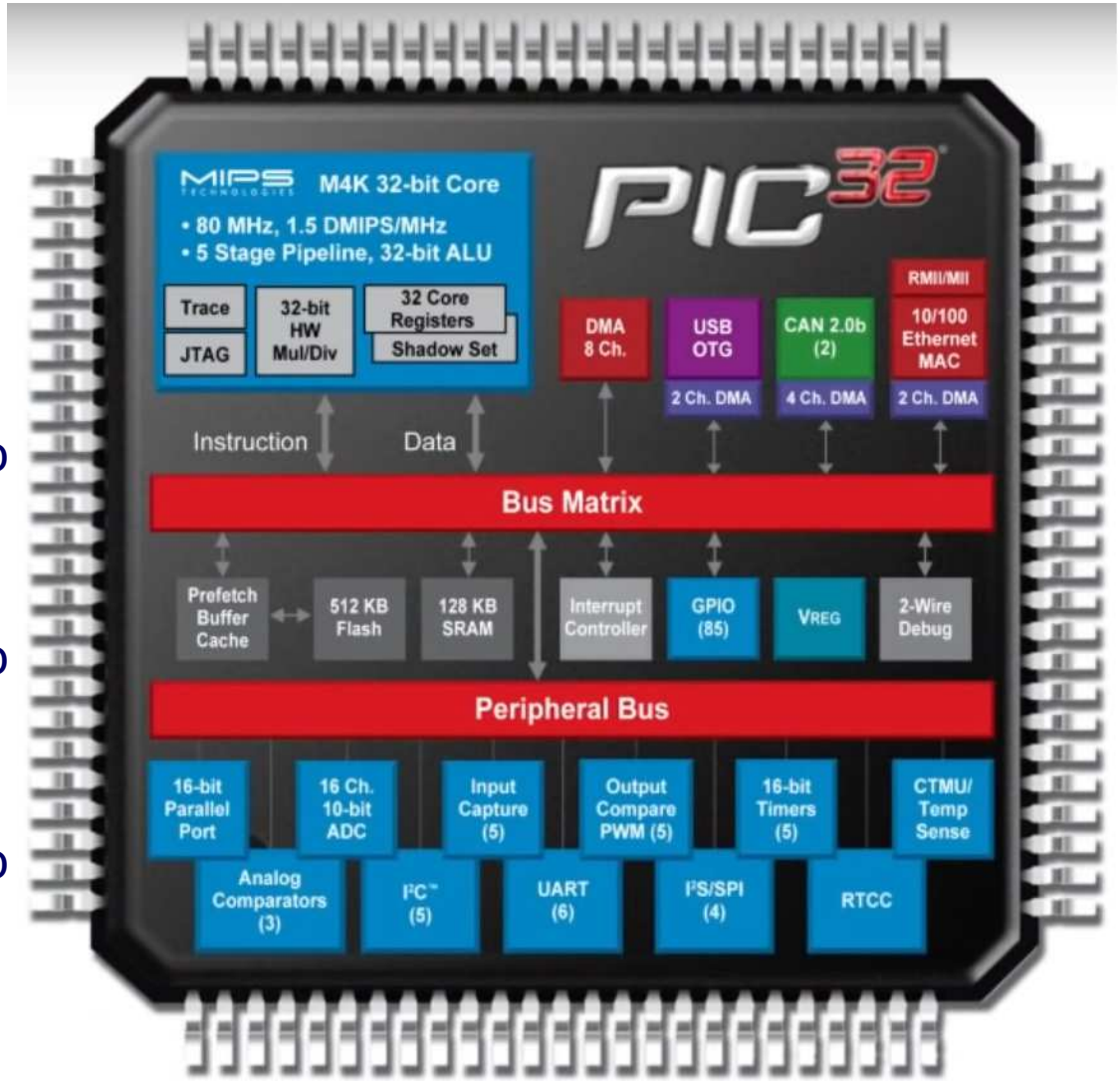
Configurações – DMA dedicado



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

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 DMA 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 DMA (expressa em Bytes/s), supondo um funcionamento em modo bloco.
- **Exercício 2** – Determine a taxa de transferência de pico desse DMA (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 DMA 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 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 DMA dedicado com as características referidas anteriormente.