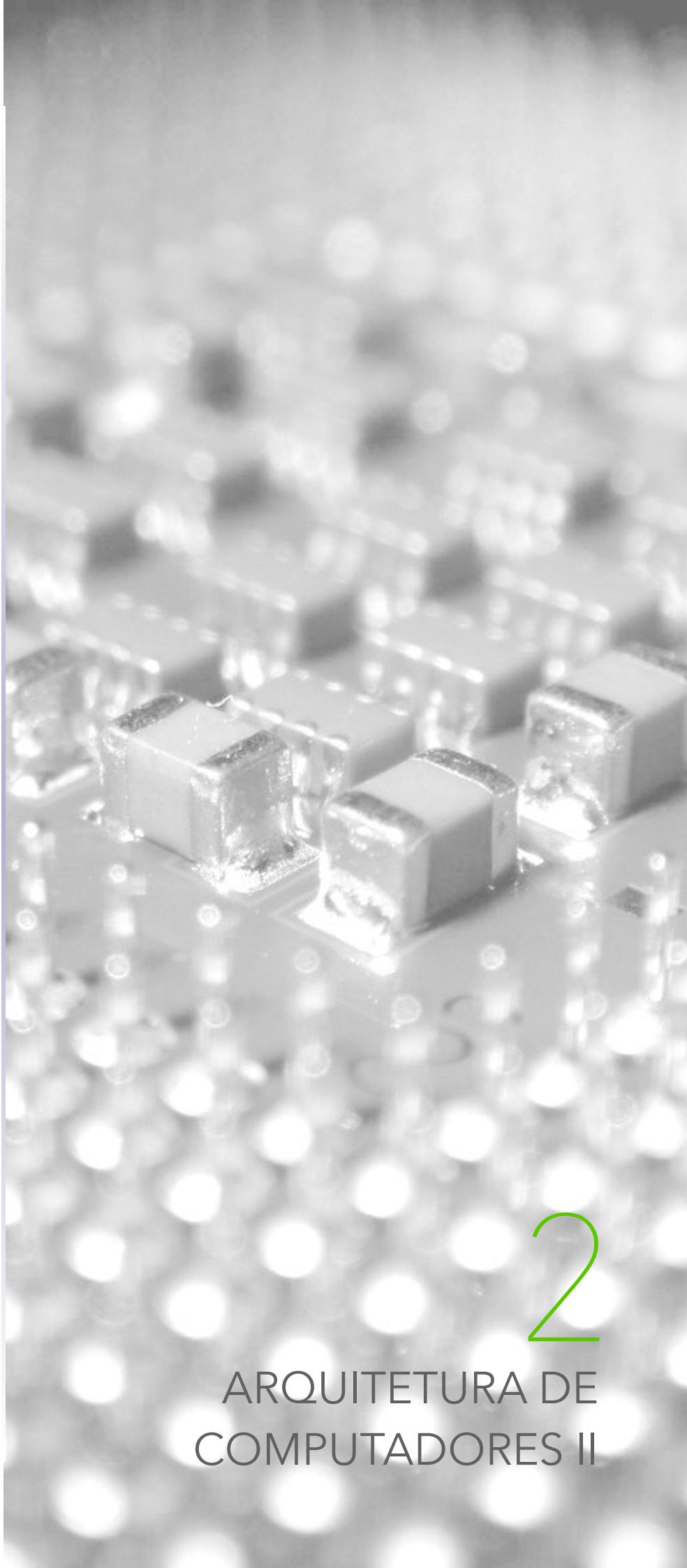


Often the great scientists, by turning the problem around a bit, changed a defect to an asset. For example, many scientists when they found they couldn't do a problem finally began to study why not. They then turned it around the other way and said, "But of course, this is what it is" and got an important result.

Richard Hamming



2

ARQUITETURA DE COMPUTADORES II



universidade de aveiro
teoria poesisis praxis

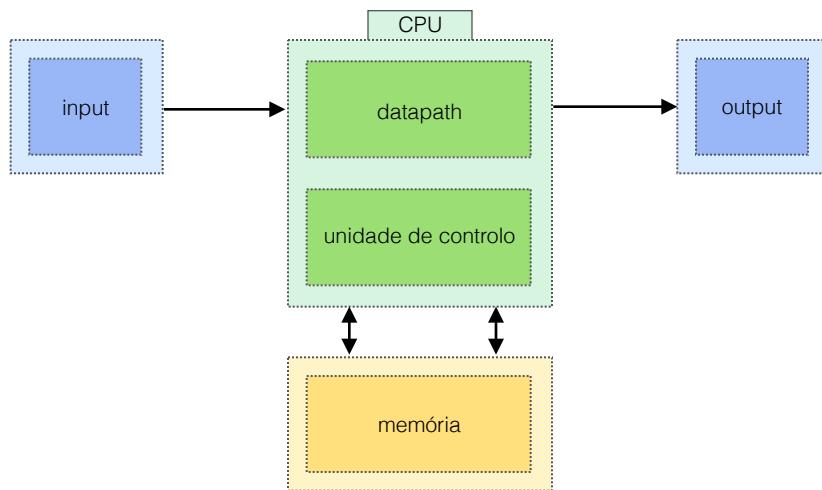
Atenção!

Todo o conteúdo deste documento pode conter alguns erros de sintaxe, científicos, entre outros... **Não estudes apenas a partir desta fonte.** Este documento apenas serve de apoio à leitura de outros livros, tendo nele contido todo o programa da disciplina de Arquitetura de Computadores II, tal como foi lecionada, no ano letivo de 2014/2015, na Universidade de Aveiro. Este documento foi realizado por Rui Lopes.

mais informações em ruieduardofalopes.wix.com/apontamentos

Através dos conhecimentos obtidos em Arquitetura de Computadores I (a2s1) surge a disciplina de Arquitetura de Computadores II (a2s2) como complemento de continuidade do estudo das unidades de processamento baseado em arquiteturas ao estilo MIPS. Esta disciplina provém da necessidade: de compreender a organização dos sistemas de entradas/saídas de um sistema de computação (em inglês *input/output*, também abreviado de I/O) e a sua programação; de adquirir alguma familiaridade com a arquitetura e programação de micro-controladores - estes essenciais para a conceção de sistemas embebidos; de conhecer a estrutura e a tecnologia dos principais periféricos e suas infraestruturas de interligação; e conhecer e compreender, tal como utilizar, microprocessadores e micro-controladores no contexto de **sistemas de tempo real** - sistemas que dão resposta em tempos muito curtos e íteis).

Sendo assim, na disciplina de Arquitetura de Computadores I (a2s1) estudámos a integração e interligação do processador, desenvolvendo conhecimentos a nível do datapath e das suas unidades de controlo. Agora cabe-nos estudar a restante envolvente, responsável pela interação com o mundo exterior - memória, e sistema de entradas/saídas (Figura 1).



sistemas de tempo real

figura 1
enquadramento da disciplina

De forma a que possa haver uma integração e uma partilha de informação (sob várias formas) entre todos os componentes da Figura 1, existem conjuntos designados de ligações, denominadas de **bus**, que ligam todos os blocos de componentes de um determinado sistema de computação. Na Figura 2 podemos ver uma representação simplificada da funcionalidade de um bus.

bus

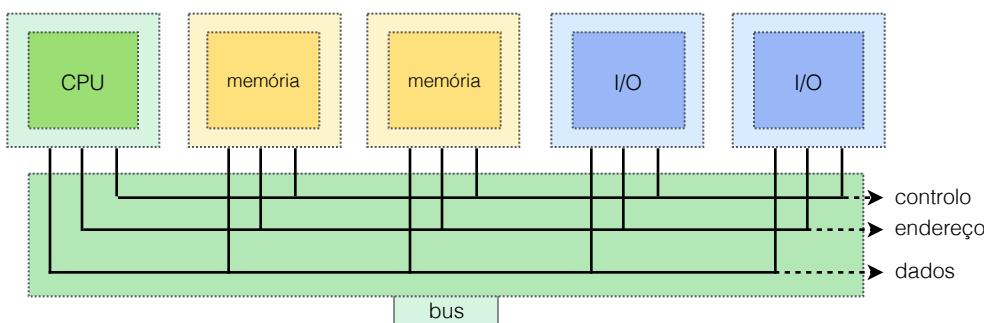


figura 2
bus

1. Introdução aos Sistemas Embebidos

Como referimos atrás, a arquitetura e programação de micro-controladores é essencial para a conceção de sistemas embebidos. Mas o que são os **sistemas embebidos**? Estes sistemas (em inglês, originalmente, *embedded systems*) são sistemas de computação com uma função específica, dedicada, integradas num sistema maior e mais complexo, essencial à ligação com o mundo exterior. Em comparação com computadores pessoais, os sistemas embebidos são computadores incluídos como parte de um sistema maior que contém outros componentes, elétricos ou mecânicos, que num todo desempenham uma função em específico, fixa.

sistemas embebidos

Micro-controladores

Atrás referimos o termo **micro-controladores**, mas não revelámos o seu conceito. Muitas vezes abreviado de μC , os micro-controladores são pequenos computadores que são integrados em chips. Para se tornarem computadores integrados significa que o chip terá de conter um processador, memória e periféricos de entrada/saída, tal como mostra a Figura 1. Este tipo de componentes é extremamente importante ainda por cima no contexto de sistemas embebidos, pois são estes os computadores dedicados para a instalação e utilização neste tipo de sistemas.

micro-controladores

Ao longo da história temos vindo também, tal como no resto do mundo da informática, a evoluir em termos de micro-controladores. Por esta mesma razão há uns anos se utilizavam, para pequena escala, micro-controladores de 4-bits. Hoje em dia, para a mesma escala, utilizam-se antes μC de 8-bits. Para escalas superiores temos também os micro-controladores de 16-bits e para escalas muito elevadas e exigentes existem μC de 32-bits, como o **PIC-32**, micro-controlador a usar nas aulas práticas desta disciplina, no decorrer do semestre.

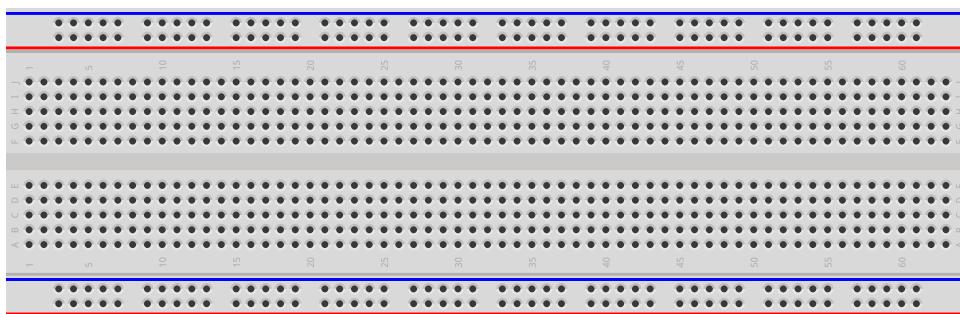
PIC-32

Alguns fatores de desempenho para computadores e micro-controladores são a velocidade de processamento, tal como a rapidez de resposta a eventos externos (sistemas de tempo real - em inglês **RTS**, de *Real-Time Systems*), o consumo de potência, a minimização das necessidades de memória, entre outros, como também analisámos na disciplina de Arquitetura de Computadores I (a2s1).

RTS

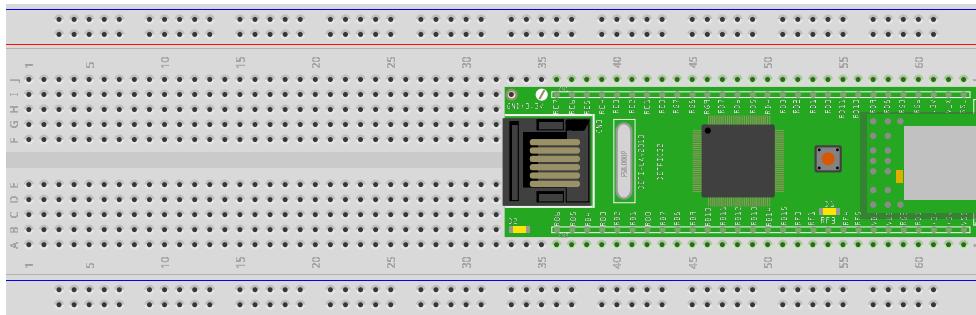
Nas aulas práticas desta disciplina iremos precisar de uma placa de suporte para a criação e teste dos circuitos a experimentar. A placa que usaremos será uma **placa branca** (em inglês **breadboard**) que serve para a prototipagem de circuitos. Na Figura 3 podes ver o aspeto de uma placa como esta.

placa branca, breadboard



O micro-controlador PIC-32 deverá ser instalado nesta placa. A Figura 4 mostra uma possível instalação do circuito DETPIC32, componente o qual requisitarás no teu departamento (específico da Universidade de Aveiro).

figura 3
breadboard



A placa DETPIC32 deve ficar então posicionada numa das extremidades da breadboard, pelo que o espaço restante servirá, mais à frente, para a adição de outros demais componentes. De forma a termos acesso a todas as ferramentas, devemos também instalar o pacote distribuído nas aulas **pic32-32.tgz** (ou **pic32-64.tgz** para computadores de 64 bits). Para tal, estando no diretório do arquivo **tgz**, devemos executar o seguinte comando no terminal (Código 1).

```
$ sudo tar xzvf pic32-32.tgz -C /opt
```

Após a execução do Código 1, temos que abrir o ficheiro oculto **.bashrc**, localizado no diretório home (**~**) e adicionar as linhas de código de Código 2, no fundo do ficheiro.

```
if [ -d /opt/pic32mx/bin ] ; then
    export PATH=$PATH:/opt/pic32mx/bin
fi
```

Agora, em princípio, já teremos acesso a todas as ferramentas de software necessário para trabalharmos com a placa DETPIC32. Para testar, experimentemos executar o comando **pcompile** (Código 3). Se aparecer a mensagem de erro que é apresentada no Código 3, tudo indica que o software está bem instalado.

```
$ pcompile
usage: /opt/pic32mx/bin/pcompile c_or_asm_file [c_or_asm_file ...]
```

Primeira execução na DETPIC32

Como já foi referido, a placa DETPIC32 contém um micro-controlador PIC32, que tem arquitetura MIPS. Dado isto, experimentemos criar um programa e tentemos executá-lo sob a placa DETPIC32. Vejamos o Código 4.

```
.equ PRINT_STR, 8
.data
msg: .ascii "AC2 - DETPIC32 primer\n"
.text
.globl main
main: la $a0, msg
ori $v0, $0, PRINT_STR
syscall
ori $v0, $0, 0
jr $ra
```

Para executar o Código 4 na placa, primeiro temos de compilar o ficheiro. Para tal, gravando o Código 4 como **teste.s** (a extensão ***.s** é de código assembly),

figura 4
breadboard com DETPIC32

pic32-32.tgz

código 1
instalação de ferramentas (1)

código 2
instalação de ferramentas (2)

código 3
teste de instalação

código 4
exemplo de código assembly

5 ARQUITETURA DE COMPUTADORES II

podemos executar o Código 5, onde adicionamos `teste.s` como argumento ao ficheiro binário `pcompile`, compilador para o PIC32.

```
$ pcompile teste.s
```

código 5
compilação PIC32

O que acontece depois da execução de Código 5 é a criação de quatro ficheiros com o mesmo nome, mas com diferentes extensões e conteúdos. O passo a seguir agora, é transportar o código para a memória de instruções do PIC32. Para tal, usamos o comando `ldpic32` como é possível verificar pelo Código 6, ligando antes a placa ao computador, por intermédio de um cabo USB.

```
$ ldpic32 -w teste.hex
```

código 6
carregamento para PIC32

Após iniciar o comando, é pedido para carregar no botão reset da placa DETPIC32, botão esse localizado junto ao PIC32, como mostra a Figura 5.

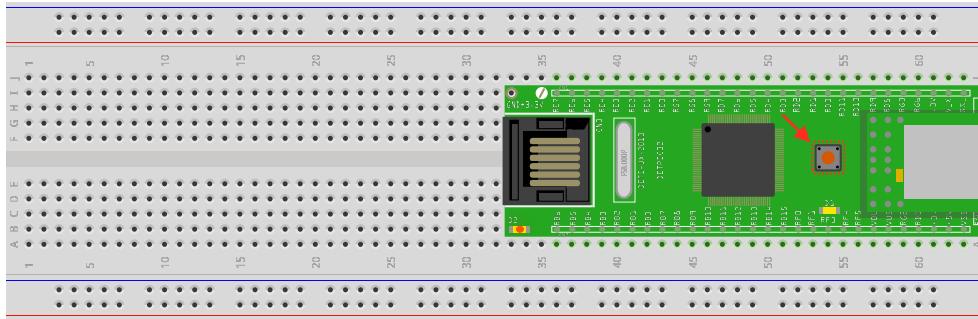


figura 5
botão reset do DETPIC32

De seguida, por último, só resta mesmo abrir o emulador de consola do PIC32, através do comando `pterm`, sobre o qual executamos o programa que acabámos de carregar.

2. Sistema de Entradas/Saídas

A placa que utilizaremos ao longo da disciplina, como já devemos saber, contém um micro-controlador PIC32, o que, no fundo, se trata de um **sistema de computação**. Um sistema de computação é um sistema que nele contém um ou mais processadores, memórias, módulos de entrada/saída e vias de ligação (usualmente denominadas de buses). Vejamos um pouco as características da unidade em estudo.

sistema de computação

Características e módulos I/O do PIC32

O processador da unidade de PIC32 é de arquitetura MIPS, que executa a uma frequência de 200MHz e com cinco níveis de pipeline, incluindo uma cache para instruções e outra para dados. A **cache** é uma memória rápida e acessível pelo **processador** a quase qualquer instante, isto é, em **menos ciclos de relógio** que as **memórias casuais**. Em termos de memória central, podemos verificar que coexistem três blocos fundamentais, sendo eles: um correspondente a 2Mb de **memória FLASH** (**memória não-volátil**), suficiente para manter o **programa preservado** num determinado estado, mesmo **após encerramento**; uma pequena memória de **prefetch**, como **memória SRAM** (**RAM estático**) de instrução; e um “segundo nível de cache”, suficiente para armazenar o código do programa e os dados com que os programas trabalham, de 512Kb, também por SRAM.

cache

memória FLASH

prefetch

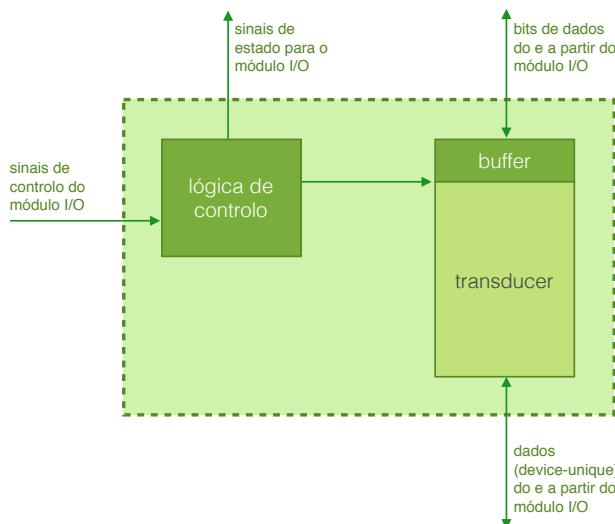
memória SRAM

Para além destes componentes, o módulo PIC32 também é habilitado de **módulos de entrada/saída**, também usualmente denominados pelo seu nome em inglês **input/output modules**, ou simplesmente, **I/O modules**. Estes blocos diferenciados de entradas e saídas estão divididos entre dois grandes grupos: um primeiro grupo, ligado ao **bus que interliga o processador e a memória**, isto é, ao bus mais rápido, e um segundo grupo, **ligado a um bus secundário ou periféricos**. Do primeiro grupo fazem parte módulos que suportam periféricos rápidos, com os quais é necessário assegurar uma taxa elevada de transferência de dados. Já do segundo grupo fazem parte módulos com taxas de transferência de dados mais baixas. Alguns destes componentes do segundo grupo são o periférico **ADC** (sigla inglesa para *analog to digital converter*), entre muitos outros. Todos os módulos do **primeiro grupo** são mais sofisticados que os do segundo, dado terem uma maior autonomia de funcionamento que se reflete na **capacidade de comunicação direta com a memória** - contêm **DMA** (sigla inglesa de *direct memory access*).

Mas como é que se faz a ligação de componentes externos a sistemas de computação? Dada a grande variedade de sistemas de operação, é totalmente impraticável incorporar no processador a lógica necessária para controlar uma gama diversificada de periféricos. É aqui que entra o conceito de módulo I/O, sendo ele um componente que por um lado liga-se a um bus e por outro lado a um ou mais periféricos, tendo, necessariamente, de ter a mesma especificidade, isto é, ser do mesmo tipo. Os dados, sendo assim, têm de ser endereçados para o módulo de I/O e ter ligações que permitem o seu fluxo entre o sistema processador-memória e o módulo I/O em questão.

Podemos então definir um **periférico**, como um dispositivo externo ligado a um **módulo de entrada/saída**. Eles podem agrupar-se em três categorias distintas: **legíveis por pessoas** (todos os equipamentos que comunicam de forma perceptível com o utilizador), **legíveis para máquinas** (todos os equipamentos que comunicam apenas de forma perceptível por máquinas) e de **comunicação** (todos os equipamentos integrados em telemática - comunicações remotas ou locais).

De forma algo ilustrativa e elucidativa, eis uma representação do conteúdo de um periférico, na Figura 6.



Na Figura 6 podemos ver uma versão simplificada do conteúdo de um periférico e **interface com módulo I/O**. Nele estão representados um **buffer** (em português, tampão) e um **transdutor**. Um **buffer** é um componente que armazena

módulos de entrada/saída

input/output modules,

I/O modules

ADC

DMA

periférico

figura 6

**periférico e interface com
módulo I/O**

buffer

transdutor

temporariamente os dados transferidos entre o módulo I/O e a sua envolvente. O transdutor é um componente encarregue da conversão dos sinais elétricos em outras formas de sinal.

A interface do módulo de I/O tem **sinais de controlo**, que determinam funções a executar pelos determinados periféricos, **uma entrada** (leitura) de envio de dados para o módulo e uma **saída** (escrita) para receber dados pelo módulo de I/O, **sinais de estado**, que indicam o **estado do periférico**, e **dados**, isto é, **conjuntos de bits** a ser enviados ou recebidos do módulo de I/O. Já o **periférico** contém uma determinada **lógica de controlo**, que é **responsável** pela **operação** do periférico segundo as diretivas do módulo I/O, um buffer e um transdutor, como verificado pela Figura 6.

sinais de controlo
entrada
saída
sinais de estado, dados
lógica de controlo

Teclado/discos magnéticos

Como é que se processa a informação entre o processador e o teclado ou discos magnéticos? Vejamos o caso do teclado primeiro.

Para poder haver uma comunicação e um processamento da interação do utilizador com o computador via teclado, é necessário antes haver uma unidade de informação detalhada. Sendo assim, concebeu-se o **caráter como um código de 7 bits e um bit de paridade**, conceito designado de **código ASCII**. O código ASCII tem assim 127 caracteres representáveis, incluindo tanto o alfabeto como números e alguns caracteres especiais (pontuação, espaço, entre outros...). Nesta **comunicação, cabe assim a um transdutor a geração de um código**, de forma a que quando o utilizador prima uma determinada tecla do seu teclado ele vá traduzir o sinal gerado num conjunto de bits correspondentes ao código do carácter escrito que é transmitido ao módulo de I/O.

código ASCII

Já no caso do disco magnético, uma unidade de disco contém eletrónica para comunicar dados, e sinais de controlo e de estado com o módulo I/O. Através de um transdutor, há a conversão da informação armazenada magneticamente na superfície do disco em bits, que coloca no buffer (e vice-versa). Restante eletrónica controla o movimento das cabeças de leitura/escrita, reposicionando-as na zona de leitura.

Funções do módulo I/O

Os **módulo de I/O** fazem o **controlo** e a **temporização do fluxo de dados** entre o **sistema** e os **periféricos**, a **comunicação com o processador**, a **comunicação com o periférico**, o **buffering de dados na memória local** e a **possível deteção de erros**. Por exemplo, numa transferência comum de uma gama de dados de um periférico para o computador (tarefa de leitura), o **processador** num **primeira fase** interroga o módulo de I/O verificando o **estado do periférico**; numa **segunda fase** o módulo de I/O **transmite** resposta do **estado ao processador**; numa **terceira fase**, se o periférico estiver operacional e pronto a transmitir, o **processador** **transmite** o comando de **leitura** ao módulo de I/O; numa **quarta fase** o módulo de I/O **obtém** o dado do **periférico**; e **finalmente** o dado é **transmitido** para o **processador**.

De forma simplificada e reduzida, na Figura 7 podemos ver uma representação da estrutura de um módulo de I/O com a interface do bus do sistema à esquerda e com a interface do periférico à direita.

Em suma, o módulo de I/O permite que o processador veja um modelo simplificado de um conjunto de periféricos, isto é, permite que o processador se abstraia dos detalhes de funcionamento de cada periférico.

Estes módulos são programáveis, pelo que agora precisamos de verificar como se comportam em termos de inclusão na arquitetura MIPS, de forma a que os possamos usar na placa DETPIC32.

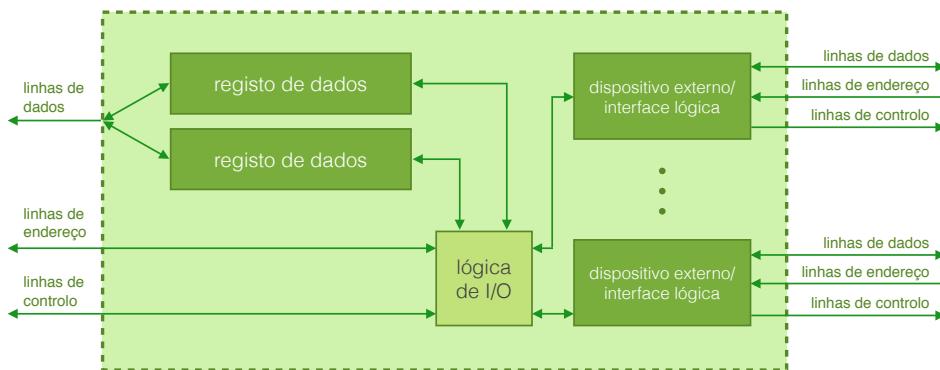


figura 7
estrutura de módulo I/O

Modelo de programação do módulo I/O

Um módulo I/O tem de ter informação de controlo, de estado e um registo de dados que lhe permita a transferência de dados entre o computador e um periférico. Nem sempre estes registos são independentes e muito frequentemente os campos de estado e de controlo são partilhados. Assim, a comunicação entre o processador e um módulo I/O requer que o primeiro defina o modo de operação dos periféricos, indique a operação a efetuar - escrevendo no registo de controlo (que pode ser mais do que um no módulo) -, saiba o estado do periférico - lendo o registo de controlo -, envie os dados (escrita) e leia outros pelo periférico (leitura).

Para identificar os registos de entradas/saídas existem diferentes alternativas para a seleção. De forma global, a respeito de arquiteturas, existem duas possíveis alternativas. Por discutirmos “seleção”, podemos verificar que há a necessidade de fazer a indicação de um determinado endereço. Uma forma de aplicar uma seleção é pela definição de um espaço de endereçamento designado especificamente para as entradas/saídas. A esta alternativa dá-se o nome de **I/O isolado** (em inglês, *isolated I/O*). Neste sistema existe um repertório de instruções específicas para lidar com periféricos. Uma outra alternativa é ter registos de I/O mapeados na própria memória, numa zona específica para periféricos. Esta é a organização usada pelos processadores de arquitetura MIPS. Isto significa que as operações sobre registos de entrada/saída são iguais às operações que são passíveis de ser efetuadas sobre registos de memória - como as instruções de *load* e *store*. Assim, em MIPS, usa-se a instrução de *load* para ler de um periférico e a instrução de *store* para a escrita sobre o mesmo. A esta organização damos o nome de **memory-mapped I/O** e pode ser usada para processadores com organização de I/O isolado, reorganizando a memória com um novo espaço de endereços partilhados com registos I/O¹.

Mas como é que o PIC32 partilha o espaço de endereçamento? Numa zona da memória denominada de **SFR** (acrónimo de *special function registers*), situada entre os endereços 0xBF80000 e 0xBF8FFFFF, existem várias zonas limitadas e próprias para os diversos módulos de entradas/saídas, como o **PORTA** até ao **PORTG**, **ADC**, entre outros...

Na descodificação de endereços, cada módulo tem um descodificador que verifica se existe um determinado endereço no enquadramento do endereçamento de um dado módulo.

modo de operação

I/O isolado

memory-mapped I/O

SFR

¹ esta organização é vulgar atualmente nos processadores Intel, pelo que permite uma maior gama de instruções e diversidade de operações. No caso de I/O isolado existem comandos específicos e especiais para I/O, como os comandos *in* e *out*, a assumirem os mesmos significados que *load* e *store*, do MIPS.

Comandos de I/O

Existem, ao todo, quatro tipos de comandos que o módulo de I/O pode executar quando este é endereçado pelo processador: os comandos de controlo, os comandos de estado, os comandos de leitura e os comandos de escrita. Os de controlo são responsáveis pela ativação do periférico e pela indicação de uma determinada tarefa ao periférico. Por sua vez, os comandos de estado servem para executar um teste de várias condições de estado associadas às tarefas designadas para execução. Já os comandos de leitura permitem a leitura de dados pelo periférico, enquanto que os comandos de escrita permitem a escrita de dados no periférico.

Técnicas de I/O

Várias técnicas de I/O foram criadas ao longo dos anos de forma a que se conseguisse aumentar o rendimento obtido da execução de um processador. Hoje em dia, algumas delas requerem mecanismos denominados de interrupção. Uma transferência de dados do módulo I/O para a memória através do processador pode não conter mecanismos de interrupção com I/O programado (polling), mas também pode ser interrompido por uma técnica de interrupt-driven I/O. A interrupção, aqui, é provocada, sempre, por um periférico, o qual lança um pedido ao processador, pelo que este responde estagnando o programa em si, executando uma rotina de tratamento de interrupção (também denominada de serviço à interrupção). A vantagem deste método está no facto do processador deixar de ter de esperar pelo periférico executar todas as suas tarefas designadas, podendo assim ir executando as suas próprias tarefas até que o periférico termine a tarefa e assinale esse evento ao processador.

Neste mecanismo de interrupção há ainda que, quando o processador está pronto para receber nova informação, terminando uma leitura, ele é interrompido, lê o dado do registo do periférico em causa e aloca na memória para prévio processamento. Mas isto ainda pode causar atrasos a nível de processamento, pelo que se criou uma transferência direta à memória, já referida anteriormente, mais sofisticada (com capacidade de endereçamento à memória), denominada de DMA - que inclui interrupção. A Figura 8, Figura 9 e Figura 10 mostram o fluxograma de cada um dos diferentes tipos de transferência de dados abordados.

polling

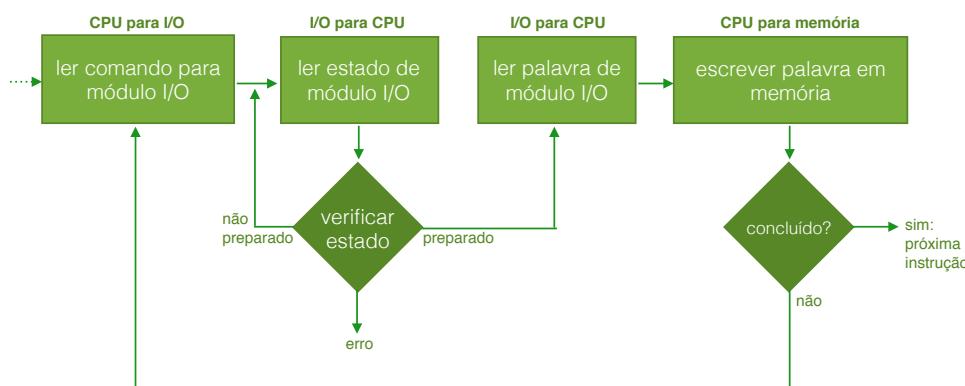
interrupt-driven I/O

rotina de tratamento

DMA

figura 8

fluxograma por polling



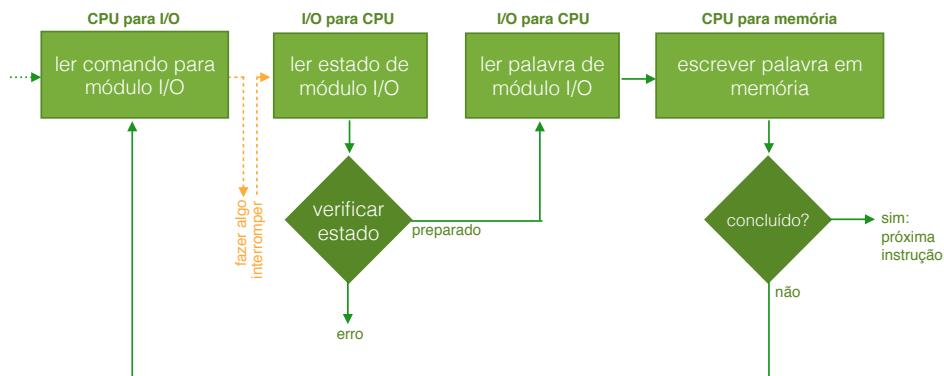


figura 9
fluxograma por interrupção

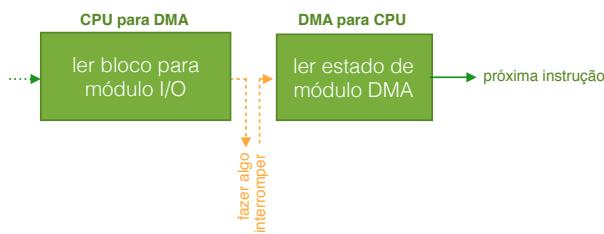


figura 10
fluxograma por DMA

Vejamos assim o caso de módulo de I/O programado e consideremos uma escrita num periférico. A invocação pelo programa, da rotina de comunicação com o periféricos exige que se siga a Figura 11.

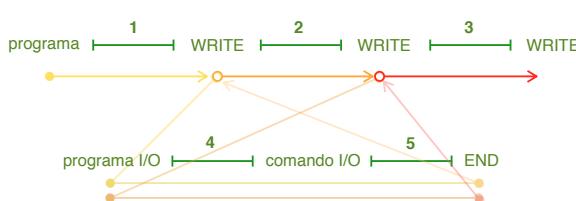


figura 11
rotina de comunicação com periféricos

Na Figura 11 as secções 1, 2, e 3 do programa nunca envolvem I/O. Já na quarta secção há uma preparação da operação de I/O, sendo que é verificado o estado do dispositivo, é transferido o dado para o módulo de I/O e são preparados os parâmetros para o comando de I/O. Emitindo este comando de I/O, aguarda-se que o dispositivo o execute (por teste de estado), o que completa a operação na secção 5. Este procedimento (técnica) é demorada, dada a espera pela execução.

No PIC32 a programação I/O é feita através dos seus conjuntos de portas (RB é porta B, RD é porta D, ...). Todas elas são programáveis, pino-por-pino. Mas como é que as podem programar, sendo que há portas com especificidade de entrada e outras com especificidade de saída?

Cada um dos pinos de comunicação com o exterior de um micro-controlador PIC32 tem a lógica explícita na Figura 12.

Na Figura 12 temos um flip-flop denominado de **TRIS**, que é um registo de controlo (configuração), o qual define se a determinada porta tem especificidade de entrada ou de saída, sendo de entrada se $\text{TRIS} = 1$ ($\text{I/O PIN} = \text{Z}$) ou saída se $\text{TRIS} = 0$ ($\text{I/O PIN} = \text{LAT}$). A designação **LAT** é respetiva a um outro flip-flop com o

TRIS

LAT

mesmo nome, que é um **registro de dados** que serve de **entrada** se **LAT** tiver o valor lido do pin de entrada/saída e serve de **saída** se **LAT** tiver o valor **escrito** no pin de entrada/saída. Estas portas são todas configuráveis ao nível do bit, pelo que fornecem máxima flexibilidade.

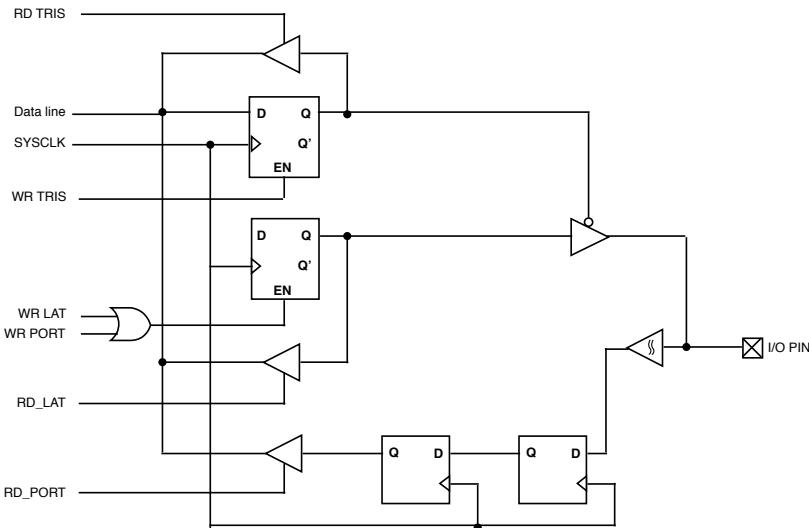


figura 12
pino I/O da PIC32

Os sinais **PORT** fazem parte de um **registro de dados**, sendo que **RD_PORT** lê o **I/O PIN** com 2 níveis de atraso (**causados** pelos **flip-flops**).

Para programar uma destas portas, designadas de **TRISx**, **LATx** ou **PORTx** (sendo x B, C, D, F ou G), carregam-se as suas identificações nos registos mapeados no espaço de endereçamento da memória. O **acesso aos registos** faz-se tendo definido um **deslocamento igual à distância do registo I/O ao registo-base** (zona dos **SFRs**).

Assim, **primeiro** há que **configurar a porta**, isto é, escrevendo no registo **TRISx**. Por exemplo, podemos configurar os bits 0 e 2 de **TRISE** como porta de saída (sem alterar a configuração de outros bits). Sendo o endereço relativo de **TRISE**, igual a **0x6100**, fazemos como se pode verificar no Código 7.

```

lui      $t1, SFR_BASE_HI
lw       $t2, TRISE($t1)      # ler configuração de PORTE
andi    $t2, $t2, 0xFFFA      # bits 0 e 2 de $t2
sw       $t2, TRISE($t1)      # pinos 0 e 2 de PORTE configurados
                                # como de saída; restantes bits inalterados

```

código 7
configuração da porta

Comunicando de seguida com a porta, há que **ler as portas de entrada** e **atribuir valor às portas de saída**.

Como já verificámos, o processo até agora descrito é muito moroso, porque o processado tem de esperar muito tempo para que o módulo de **I/O** esteja **preparado** (**não há busy-wait**). Havendo, o **processador** envia o comando de **I/O** e continua a fazer **tempo útil**, até que o módulo de **I/O** termina a execução, **avisa** o **processador** e este **retoma o processo**. No MIPS a esta técnica dá-se o nome de **tratamento de exceções**.

Os processos de interrupção podem provir de três categorias, em diferentes arquiteturas. No processador MIPS as interrupções são de apenas um tipo de exceção. Em geral, as três arquiteturas têm efeito sobre o hardware, software ou sobre o processador num todo, pelo que as interrupções provenientes de hardware (vulgarmente designadas por **IRQ** - sigla inglesa de *Interrupt Request*) são **enviadas** do **hardware** para o **processador**. Por outro lado, as **interrupções** levantadas por **software** podem **interromper** o **carregamento** pelo **processador**, de **instruções**. Já no último, se

busy-wait

tratamento de exceções

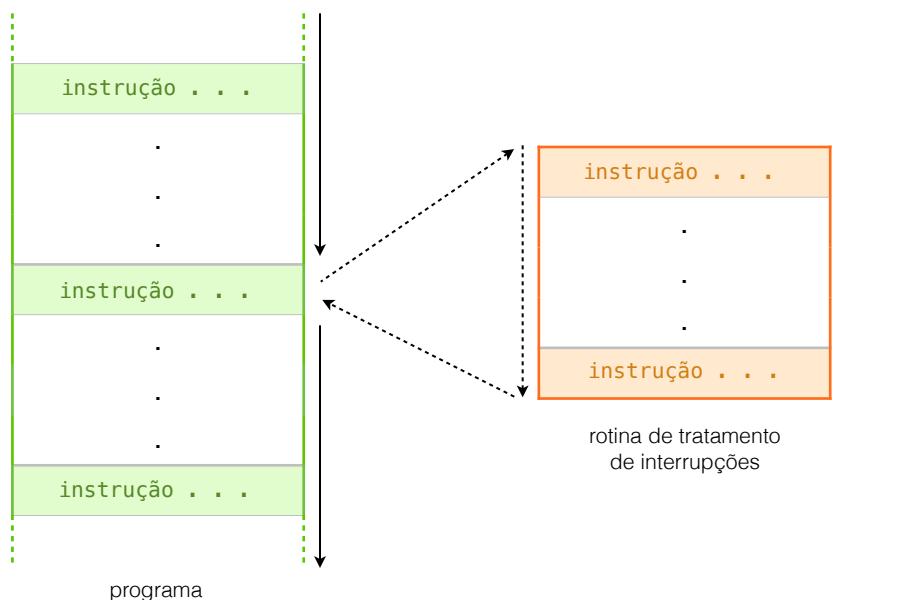
IRQ

uma **interrupção** for proveniente do próprio **processador**, por exemplo, por causa de uma **instrução ilegal** (instrução que não consta do repertório de instruções da máquina), é gerada uma **exceção**, a qual desencadeia mecanismos para tratar a má execução desse programa. Esta última interrupção (armadilha) é **enviada** para o próprio processador, o que o faz lançar a **exceção**, gravar o **estado** da **thread** atual, executar o **mechanismo** de tratamento de **exceções** e **retomar** a execução da **thread**.

thread

Uma leitura executada por um módulo de I/O com **interrupção** deve assim começar com o **processador** a **enviar** o **comando de leitura** ao periférico, prosseguindo os **métodos de execução**. Para tal, quando o **periférico** gerar uma **exceção** por ter terminado a **execução** do comando, o **processador** termina a **execução** da **instrução** atual e **executa** a **rotina de tratamento de interrupções** (*interrupt handler*), a qual identifica a **origem da interrupção**, o **estado do periférico**, lê o novo dado e **armazena-o** em **memória**. De seguida, é o **processador** que **retoma** a **execução** do programa, prosseguindo atividade.

De forma representada, uma invocação da rotina de tratamento é um mecanismo do hardware, onde o processador não interage, tendo a forma da Figura 13.

**figura 13****rotina de tratamento de interrupções**

O **processamento de uma interrupção**, interpretando a Figura 13, parte de **execuções do hardware** e termina com uma **instrução do software**, que **retoma** o valor da **instrução seguinte** a ser executada. De forma mais detalhada, primeiro o controlador do dispositivo ou outro hardware necessita de lançar uma interrupção. Dada esta ação, o processador deve terminar a sua execução atual e tomar conhecimento da interrupção, pelo que deve guardar o **valor do registo de program counter** num registo **EPC** (sigla inglesa de *exception program counter* - registo que não estando ligado ao modelo de programação do sistema, mas antes no coprocessador de controlo (C0)) e **carregar** um **novo** valor para o **program counter**, o qual deverá ser calculado com **base na interrupção**. Neste ponto, em **software** guarda-se o **produto** do processamento de informação, pelo que se deve **executar** um **processamento da interrupção**, **restaurando**, quando terminado o processamento, o **nível de informação** processado anteriormente e **restaurar** o **valor do registo program counter**, de forma a que se possa **voltar** ao programa **main**. No ISA do MIPS, a instrução que permite este último restauro é a **eret** (*exception return*).

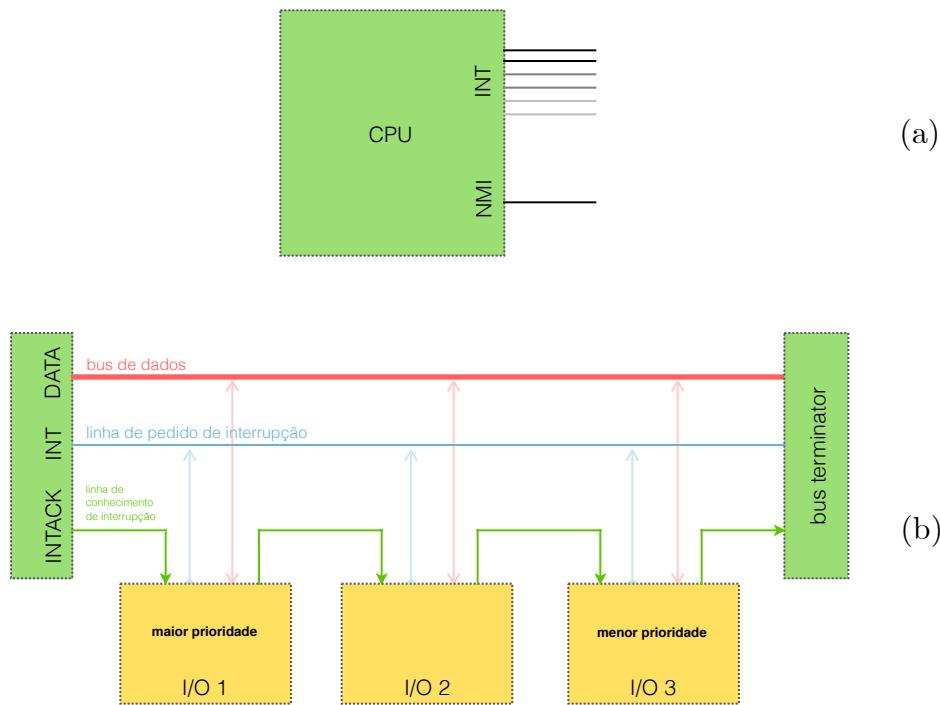
EPC

13 ARQUITETURA DE COMPUTADORES II

Existem assim duas questões importantes a ser feitas: primeiro, como é que o processador determina que dispositivo que formulou o pedido de informação?; e segundo, como é que o processador organiza, por prioridades, os pedidos formulados?

Quanto à primeira questão podemos afirmar que uma de três situações podem acontecer: ou cada dispositivo tem uma ou duas linhas de pedido de interrupção próprias (solução não muito viável, dado o rendimento e densidade de linhas - Figura 14 (a)), ou há uma identificação por software (técnica denominada de **software poll**), na qual a rotina de tratamento de interrupções trata cada módulo de I/O para identificar qual pediu a interrupção (uma vez identificado, o processador salta para a rotina específica de tratamento de interrupções desse módulo), ou ainda, por **Daisy chain (poll vetorizada por hardware)**.

Quanto às prioridades há duas soluções possíveis: primeiro, podemos impedir as interrupções (através de um processo denominado de **interrupt disable**) enquanto está a ser processada uma interrupção (não tendo em conta a necessidade de garantir tempos de registo); segundo, definir prioridades para as interrupções, permitindo que a rotina de tratamento de uma interrupção seja interrompida para atender um pedido de interrupção de um dispositivo com maior prioridade - Daisy Chain (Figura 14 (b)).



software poll

Daisy Chain, poll vetorizada

por hardware
interrupt disable

figura 14
dispositivos por interrupção

Então, o procedimento de tratamento de interrupções é realizado segundo uma condição de prioridade, o que evita a situação que se pretende retratar na Figura 15 (a), de forma a conduzir os processos para um tipo de execução de programas semelhante ao retratado na Figura 15 (b).

No caso da Figura 15 (b) podemos verificar que existe o tratamento de interrupções com prioridade, pelo que a ordem de execução permite indicar que o tratamento de Y tem uma prioridade mais elevada que o tratamento de X. Na prática, a Figura 16 mostra um exemplo mais aplicado ao conhecimento do âmbito geral de periféricos.

As interrupções têm de ter assim sistemas de controlo a si adjacentes. Os controladores de interrupções são responsáveis pela ativação ou inativação dos pedidos

de interrupção de cada periférico, definindo também os níveis de prioridade de atendimento dos pedidos de interrupção dos vários periféricos.

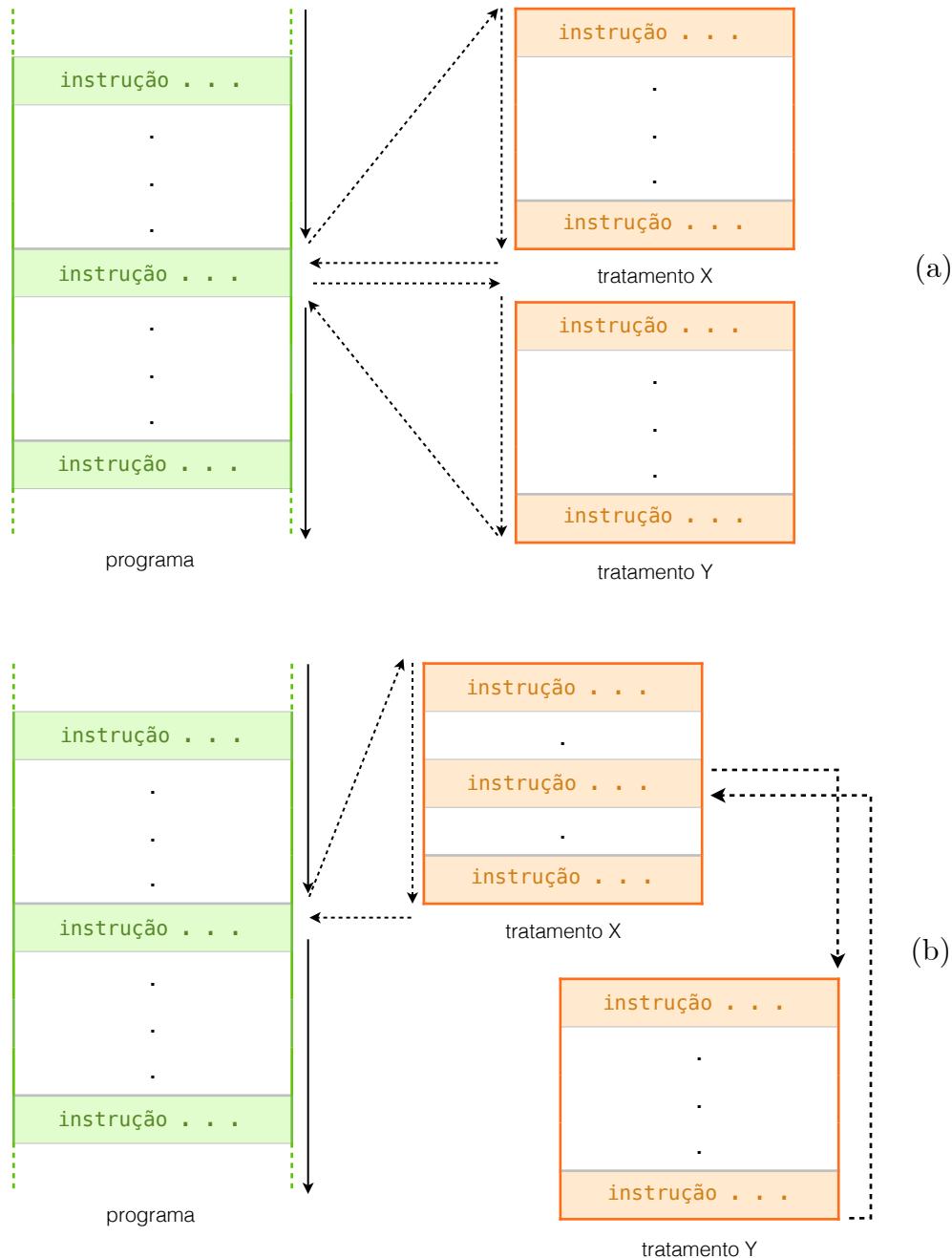


figura 15
execução com e sem
interrupções

Em termos de procedimento, um determinado periférico x pede interrupção: se ITx estiver ativo, o controlador de interrupções transmite um pedido de interrupção ao CPU, este, terminando a instrução em curso e, se a prioridade do $ITRequest$ for maior que o nível de prioridade de execução atual, há **IT acknowledge**. De seguida, é efetuada uma salvaguarda do *program counter* e é feita uma transferência de controlo para o **IT Handler** (rotina de tratamento de interrupções). Consoante os casos, o *IT Handler* pode identificar o periférico que fez o pedido de interrupção (sendo um **non-vectored**

IT acknowledge

15 ARQUITETURA DE COMPUTADORES II

IT) ou o controlador de interrupções fornece o vetor IT - controlo diretamente transferido para o *IT Handler* do periférico que pediu interrupção.

non-vectored IT

Interrupções no MIPS

Num processador MIPS as interrupções são geridas via **coprocessador 0** (designado usualmente pela sua sigla C0). Neste coprocessador, presente em qualquer processador de arquitetura MIPS, executam-se a configuração do CPU (se é *little endian* ou *big endian*), o controlo da cache, o controlo da **MMU** (sigla de *Memory Management Unit*), *timers*, controladores de eventos, deteção de erros de paridade e, claro está, o controlo de exceções/interrupções.

coprocessador 0

MMU

Interrupções e exceções

Então mas afinal qual é a diferença entre uma interrupção e uma exceção? Uma **interrupção** é o resultado de uma operação normal (pedido de um módulo de I/O) - única condição de execução que ocorre independentemente da execução da sequência de instruções do CPU. Por outro lado, uma **exceção** é o resultado de uma condição de erro: pode ocorrer em circunstâncias de erro de endereço de memória, instrução ilegal (instrução que não consta do painel de instruções, isto é, do ISA, ou que seja privilegiada), erro de paridade na leitura da memória ou com *system calls* ou *traps* (como as instruções de *syscall* ou *break* causam exceções).

erro

No entanto, tendo estas diferenças, a resposta é semelhante em ambos os casos, pelo que a sequência das instruções do programa em execução é interrompida para executar a rotina que trata da interrupção ou execução em causa (*interrupt/exception handler*).

Na ocorrência de uma execução/interrupção o *program counter*, que contém o endereço da instrução a executar quando o tratamento terminar é salvaguardado no registo **EPC** (registo de *Exception Program Counter*), presente no coprocessador C0. Também em C0, no seu registo **Cause** é registada a causa da interrupção ou exceção, passando o processador para um modo diferente - o **modo kernel**. Assim, o endereço do **handler** é calculado com base no conteúdo do registo **Ebase**. No fim, o processador retoma a execução a partir do valor salvaguardado anteriormente de *program counter*.

EPC

Cause

kernel

Neste último parágrafo enunciámos por várias vezes registos que ainda não foram abordados, específicos do coprocessador 0 (C0). Na Figura 16 encontra-se uma tabela que contém uma pequena lista de registos de C0 que estão relacionados com exceções e interrupções.

numero de registo	nome	função
8	BadVAddr	endereço de exceção ocorrida no acesso à memória
9	Count	contagem de ciclos do processador
11	Compare	controlo das interrupções do timer interno do CPU
12	Status (SR)	controlo e estado do processador
	IntCtl	controlo e estado do sistema de interrupções
	SRSCtrl	controlo e estado do <i>shadow register set</i> (SRS)
	SRSMMap	mapeamento das interrupções vetorizadas num SRS
13	Cause	indica causa da última exceção/interrupção
14	EPC	conteúdo do program counter na exceção mais recente
15	Ebase	endereço-base dos vetores de interrupção

figura 16

lista de alguns registos do C0

O registo de controlo e estado do processador (Status (SR)) contém quatro sinais fundamentais ao seu funcionamento: tendo um **ativador de interrupção** (*interrupt enable*), na posição 0, ativo-alto, para **permitir** ou **inibir** globalmente **interrupções**, descreve no sinal *IPL<2:0>* (sigla de *Interrupt Priority Level*), nos bits de 12 a 10, o **nível de prioridade da interrupção** - sendo que o processador avalia previamente se a prioridade do pedido de interrupção é maior do que o valor de IPL -, afere um NMI (no bit 19, sigla de *Non-Maskable Interrupt*) e define um nível de exceção, no bit 1 - sinal denominado de *EXL* -, condição quando ocorre uma interrupção/exceção, o que coloca o CPU em modo *kernel* e inibe as interrupções, permitindo ao software definir uma nova *interrupt mask* e o nível de privilégio do CPU.

O registo Cause é um registo que indica a causa de uma interrupção ou exceção. Contendo dois sinais importantes, o primeiro, denominado de IP7-0 (sigla de *Interrupt Pending*) destaca os pedidos de interrupção pendentes - onde os bits 15-10 (IP7-2) indicam os pedidos externos e numa operação AND com a *Interrupt Mask* do registo anterior, se definem os pedidos de interrupção ativos - e o segundo, TI, no bit 30 (sigla de *Timer Interrupt*), que declara uma interrupção causada pelo *timer* do CPU.

O registo Ebase (em extenso: *ExceptionBase*) tem, nos seus bits 29 a 12, o endereço-base dos vetores de interrupção. As prioridades na arquitetura MIPS são processadas via um controlador de interrupções externo (denominado de **EIC**), sendo que no PIC32 são definidas pelo **Priority Interrupt Controller**.

Por fim, os registos de Count e de Compare são os registos responsáveis pelas chamadas de sistema que usaremos nas aulas práticas - do **Core Timer**. Enquanto que o Count é **incrementado a cada dois ciclos de relógio**, o Compare, em **conjunto** com o Count, permite **implementar um timer**, mantendo um **valor estável** que só muda quando é **escrito**. Quando o valor de Count iguala o valor de Compare é gerada uma **interrupção**.

Interrupções no PIC32

Como já vimos, no PIC32 as interrupções são pré-processadas pelo **Priority Interrupt Controller** tendo, no fim, que apresentar ao CPU as interrupções por ordem de prioridade. O PIC32MX, micro-controlador que estamos a usar nas nossas aulas práticas, contém 96 possíveis fontes de interrupção, suportando 5 fontes de interrupção externas, com 7 níveis de prioridade, cada um com a possibilidade de se definir mais 4 sub-níveis de prioridade. Este micro-controlador inclui um **shadow register set**, que é um **segundo conjunto de registos gerais \$r0, ..., \$r31**, preferencialmente usados pelo CPU aquando de uma interrupção com a mais alta prioridade (nível 7), o que dispensa a salvaguarda do conteúdo dos registos gerais no stack (que constitui o prólogo das rotinas de tratamento das restantes interrupções).

Cada fonte de interrupção tem a si associados 7 bits de controlo, agrupados em vários SFR's: o **interrupt enable** (_IE) - processado em registos **IECx** - onde cada bit do registo ativa/inativa as interrupções de um dispositivo em específico (ativo-alto); o **interrupt flag** (_IF) - processado em registos **IFSx** - onde cada bit **regista** os **pedidos de interrupção** de um dispositivo em específico (ativo-alto); o **interrupt priority** - processado em registos **IPCx** - permitindo **dois níveis de prioridade**. O *interrupt priority*, tendo dois níveis de prioridade estabelece um nível primeiro (**priority level** (_IP)) que se define por três bits para o qual se *IPx* for menor que o valor do IPL do estado de C0 os pedidos de interrupção são ignorados. Um segundo

EIC

Priority Interrupt Controller

shadow register set

interrupt enable

interrupt flag

interrupt priority

priority level

nível é estabelecido por **subpriority level** ($_IS$), dois bits que definem outros 4 níveis de prioridade num mesmo grupo de prioridade (mesmo $_IP$).

subpriority level

Funcionamento dos controladores de interrupções

Todos os pedidos de interrupção são testados nas **transições positivas** do **SYSCLK** e **registados** nos registos **IFS x** . Como também já vimos, os pedidos de interrupção são ignorados se o correspondente bit $IECx = 0$, atuando este como máscara da *flag* de interrupção. Se as **interrupções** estiverem **ativas**, os **pedidos de interrupção** são **codificados** num número de **vetor**, estando a cada um destes associado um **nível de prioridade** determinado pelo valor do campo **IP x** do respetivo registo IPC - em dois modos de funcionamento: **single-vector** e **multi-vector**.

single-vector, multi-vector

Na conclusão de todos estes procedimentos, o controlador de interrupções seleciona o pedido com maior nível de prioridade entre os pendentes e apresenta ao CPU o número de vetor associado e o nível de prioridade do pedido.

Em relação à resposta do CPU aos pedidos de interrupção, o processador analisa a informação sobre o vetor de interrupção apresentado entre os níveis E (de execute) e M (de memory access) do pipeline. Neste estado, se o **nível de prioridade do vetor** for **maior** que o **nível de prioridade atual**, a **interrupção é servida**, caso contrário o **pedido fica pendente** até que a condição acima se verifique. Neste ponto o valor contido no registo **exception program counter** fica com o valor do **program counter**, o bit 1 do registo de estado (**EXL**) fica com o valor **1**, o que **desativa interrupções**. É assim que o **program counter** recebe o valor do endereço do vetor do serviço à interrupção. O CPU só retomará a execução do programa interrompido aquando da passagem à execução da instrução **ret**.

Em suma, é conveniente rever os diversos registos mapeados em memória, sendo eles: **INTCON** - Interrupt Control; **INTSTAT** - Interrupt Status; **IPTMR** - - Interrupt Proximity Timer; **IFS x** - Interrupt Flag Status; **IECx** - Interrupt Enable Control; **IPC x** - Interrupt Priority Control; **OFF x** - Interrupt Vector Address Offset.

De forma a **verificar** a presença de uma **interrupção** há que **verificar**, por sua vez, o **estado** da *flag* **IFS x** . Estes registos são 3 de 32 bits que contêm as notificações para cada dispositivo em caso de interrupção, sendo que denotam o valor de '**0**' quando **não houve interrupção** e '**1**' quando **houve interrupção**.

Para saber os locais para os quais são permitidas **interrupções** pode-se aceder aos 3 registos **IECx** que fornecem o valor '**1**' para quando dada fonte é **ativa** por **interrupções** ou o valor '**0**' quando dada fonte é **inativa** por **interrupções**.

Um outro conjunto de registos importante é o **IPC x** , sendo que este gera as prioridades de atendimento às interrupções, sendo que $IPx<0:2>$ são relativos aos bits de prioridade (avaliados de 0 a 7) e $ISx<1:0>$ são relativos aos bits de subprioridade (avaliados de 0 a 3).

Limitações de entradas e saídas sob interrupção

A ativação de interrupções nos módulos de entradas/saídas, *per si*, também incluem um conjunto de limitações, embora pequeno. Primeiramente há que considerar que as taxas de transferência de informação de/para os periféricos é limitada pela velocidade com que o processador pode atender e servir os dispositivos. Por conseguinte, também há de haver um maior consumo de tempo do processador, pelo que por cada transferência o processador terá de executar um conjunto de instruções.

Módulo DMA

Quando é necessário transferir grandes quantidades de dados, como HDD ou SSD, é mais eficiente usar outra técnica que as interrupções - a esta técnica denominamos de **acesso direto à memória (DMA)**. Esta técnica é suportada por módulos próprios, tal como o representado na Figura 17.

acesso direto à memória

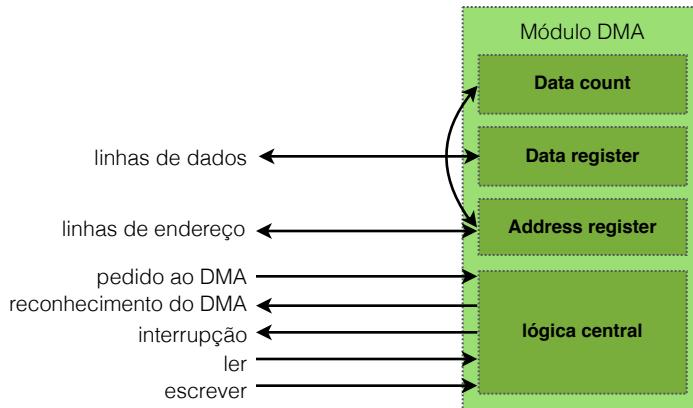


figura 17
módulo DMA

A operação no módulo DMA é feita de forma sequencial por via de três grandes passos de execução. Quando o processador pretende ler ou escrever um bloco de dados, este envia um comando ao DMA onde indica o tipo de operação a concretizar (se é leitura ou escrita), o endereço do periférico, o endereço base de memória para ler ou escrever os dados (no address register) e o número de palavras a registar (escrito no data count). De seguida o processador prossegue a sua execução, enquanto que o módulo DMA transfere o bloco, palavra² a palavra, diretamente para ou da memória sem intervenção do processador. Finalmente, o módulo DMA envia uma interrupção ao processador quando termina a transferência.

Como podemos ver em todo este processo, comparativamente à técnica de interrupções, o tempo consumido do processador foi muito restrito, pelo que apenas se ficou no início e no fim da operação total.

Por conseguinte deste processo todo, temos que o nosso módulo DMA tem 3 possibilidades de configuração: numa primeira configuração podemos ter uma transferência a consumir apenas 2 ciclos de bus (do DMA para o módulo de I/O e do DMA para a memória (vice-versa para ambas)); numa segunda configuração podemos ter o módulo DMA integrado, *per si*, no módulo de I/O ou ter módulos de I/O diretamente ligados ao DMA, o que permite haver um só ciclo de bus por transferência; e finalmente um sistema com bus de I/O. Esta última configuração permite que todos os módulos de I/O estejam ligados ao bus de I/O e que o módulo de DMA seja a ponte entre os 2 bases do conjunto total.

No micro-controlador que usaremos nas aulas práticas (PIC-32) há uma configuração resultante da fusão da segunda e da terceira enumeradas atrás.

3. Buses

No início desta disciplina revimos os conceitos que abordámos na disciplina de Arquitetura de Computadores I (a2s1), onde vimos e respondemos de modo muito breve à questão "como é que se interligam o processador, a memória e os dispositivos

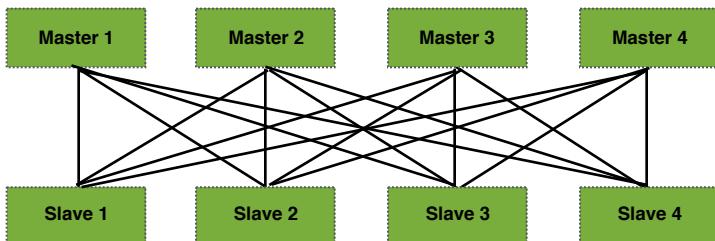
² Recorde-se que por "palavra" pretende-se enunciar word, isto é, uma sequência de 32 bits.

de entrada e saída?". Precisamente na Figura 1 e na Figura 2 deste mesmo documento retrata-se a resposta correta a tal questão - o bus.

O bus e os seus diferentes tipos

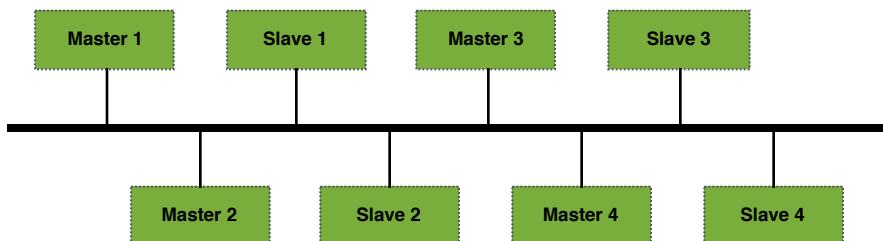
O **bus** é então, assim, um **barramento de dados** que permite criar uma estrutura de interligação entre **todas as partes constituintes** de um **sistema de computação**. Tais estruturas de ligação podem ser de dois tipos, estes, ligações ponto-a-ponto ou vias de comunicação (linhas) comuns partilhadas pelos diferentes componentes do sistema de computação.

Vendo cada um destes dois últimos tipos em separado, e começando pela ligação ponto-a-ponto, temos a Figura 18 que representa tal paradigma.



A cada ligação entre dois componentes através de linhas dedicadas para o efeito (exclusivamente para essa ligação) dá-se o nome de **full-crossbar** ou **matrix bus**.

Um outro tipo de ligação, como dito anteriormente, é a de bus **partilhado** (em inglês **shared bus**). Neste tipo de bus (modelo) os vários componentes do sistema estão interligados por linhas partilhadas entre todos eles, pelo que os sinais transmitidos por um dos dispositivos estão disponíveis para receção por qualquer um dos dispositivos ligados aos bus. Este esquema é possível de ser analisado na Figura 19.



Há assim **dois** tipos de componentes ligados aos bus: **master** e **slave**. Um componente **master** é um componente dirigente (como o processador) que coordena as ações, criando diretivas de execução. Por sua vez, um componente **slave** apenas recebe diretivas (como a **memória**). Mas como é que podemos classificar os módulos de I/O? Até agora só falámos dos dispositivos com capacidade de **DMA**, como dispositivos passíveis de serem **master**.

Alguns dos módulos ligados ao bus são: a **memória** - contém **n** posições, cada uma com um **endereço** $[0, 1, \dots, n - 1]$, tendo uma palavra a capacidade de ser lida ou escrita, com **dois** sinais de controlo (**read, write**) que indicam a operação; o **processador** - **lê instruções e dados** e **escreve dados depois de processados**, usando os sinais de controlo para controlar a operação do sistema de computação (recebe **pedidos de interrupção**); e os módulos de I/O - com **r portas**, podendo cada módulo suportar **r** periféricos, onde cada um destes é **uma porta** e tem um **endereço** $[0, 1, \dots, r - 1]$,

figura 18
ligação ponto-a-ponto

full-crossbar, matrix bus
shared bus

figura 19
shared bus

master
slave

regidos por operações de leitura e escrita, com uma linha externa de dados (dados de entrada e saída) e sinais de interrupção.

Usando buses obtemos uma maior versatilidade, pois é mais fácil adicionar novos dispositivos e os periféricos intuitivamente podem ser transferidos de um computador para outro que use o mesmo bus standard. Da mesma forma, é de baixo custo, dado que um único conjunto de linhas é partilhado para diferentes comunicações.

Por outro lado, os buses também têm as suas desvantagens. O uso destes cria um estrangulamento na comunicação, dado que a largura de banda é bem capaz de limitar a taxa de transferência do módulo I/O. Do mesmo modo, a velocidade de operação do bus tem de ser limitada pelo comprimento atual do bus, pelo número de dispositivos a ele conectados e pela necessidade de suportar uma gama de dispositivos com diversas latência (tempo de resposta) e taxas de transferência de dados diversas.

Especificidade dos buses

Qualquer que seja o bus que possamos referir, este terá que ter requisitos mínimos comportamentais de forma a que possa realmente ser um bus. Desta forma, qualquer que seja o barramento só o é se conseguir efetuar uma transferência da memória para o processador (processador lê uma instrução ou um dado em memória), do processador para a memória (processador escreve um dado na memória), de um módulo de I/O para o processador (processador lê um dado do periférico através de um módulo de I/O), do processador para um módulo de I/O (processador envia dados para o periférico) e entre um módulo de I/O e a memória (periférico troca dados diretamente com a memória sem intervenção do processador (DMA)).

Claramente, qualquer bus também deve estar bem definido e bem documentado de forma a permitir que seja ligada lógica cliente sem grandes problemas, tal como também se deve evitar a imposição de restrições desnecessárias ao desempenho do sistema e permitir a sua expansão no futuro a um custo razoável.

Bus de dados, de endereço e de controlo

Como vimos na Figura 2, os buses têm especificidades muito próprias e de três linhas importantes, uma delas é o bus de dados.

As linhas de dados são assim as ligações que permitem a transferência de dados entre os blocos componentes do sistema. A largura de banda, isto é, o número de linhas do bus de dados pode ser de 32, 64, 128 ou mais e é um fator determinante para o desempenho global do sistema.

Os outros dois buses que vimos anteriormente eram o bus de endereço e o bus de controlo. O bus de endereço é usado principalmente com o intuito de especificar a origem ou o destino dos dados no bus de dados, pelo que se o processador ler uma palavra da memória, este coloca o endereço da palavra nas linhas de endereço. Em termos de largura o bus de endereço, este determina a capacidade máxima possível de memória do sistema. Como também já verificámos, este bus também é usado pelos módulos de I/O, pelo que os bits mais significativos são usados para selecionar o módulo de I/O com o qual se quer comunicar e os bits menos significativos para selecionar a porta específica do módulo.

Em relação ao bus de controlo, este é usado, tal como o termo o indica, para controlar o acesso e uso das linhas de dados e de endereços. Como estas últimas estão partilhadas com todos os componentes, deve existir um método de controlo sem a sua utilização direta. É assim que se criam os sinais de controlo que transmitem, entre os

bus de dados

largura de banda

bus de endereço

bus de controlo

módulos do sistema, informação proveniente de comandos e de temporização. Os sinais de temporização indicam a validade dos dados e dos endereços no bus e os sinais de comando especificam quais as operações a realizar.

Os bus, como já referimos, em suma, têm várias características a si intrínsecas: têm uma **largura** - número de linhas de endereço e de dados; um **tipo - dedicado**, se houver linhas distintas para endereços e dados, ou **multiplexado**, se endereços e dados usarem as mesmas linhas; têm um **tipo de comunicação**: síncrona ou assíncrona; **métodos de arbitragem**; e **tipos de transferência** de dados suportados.

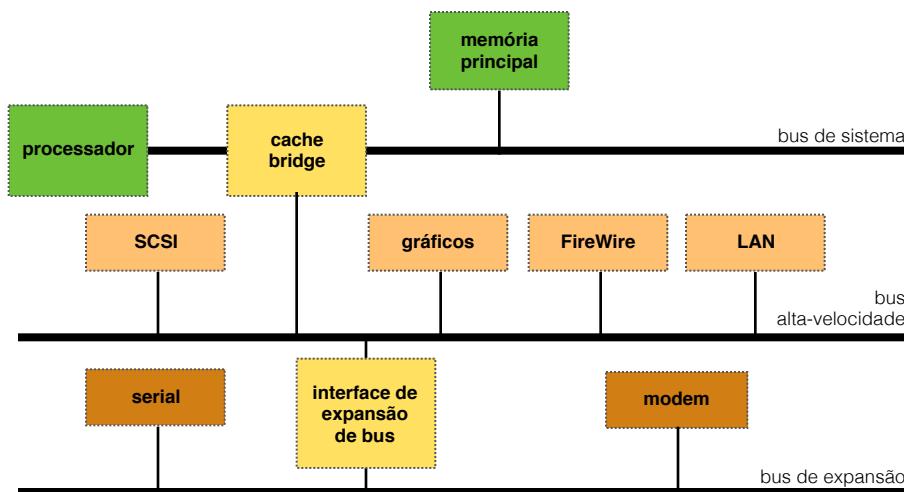
Dadas as possibilidades de configurações dos buses, de forma a que os sistemas possam funcionar o mais em **uníssono** possível, há que existir determinados **standards** que globalizassem as tecnologias de bus.

Ter um único bus pode ter duas grandes desvantagens, dado que o seu **desempenho** é menor quanto mais dispositivos a ele estiverem ligados e a usá-lo e porque o próprio bus pode-se tornar num **estrangulamento** do sistema quando as **taxas** de transferência de **informação** entre o **conjunto** de dispositivos ligados ao bus se aproxima da capacidade máxima deste. Uma forma de resolver isto é criando uma **hierarquia de buses**. Na Figura 20 podemos ver uma representação de uma hierarquia de buses.

largura, tipo, dedicado
multiplexado
tipo de comunicação
métodos de arbitragem,
tipos de transferência
standards

hierarquia de buses

figura 20
hierarquia de buses



Na Figura 20 temos um **bloco** que faz a ligação entre o **bus de sistema** e o **bus de alta-velocidade**, chamado de "cache bridge". Uma **bridge** (ponte em inglês) é um componente que **liga dois buses**, atuando como **slave** de **um** dos seus **lados** e como **master** do outro.

bridge

As regras que determinam o **formato** e a **transmissão de dados** através do bus dá-se o nome de **bus protocol** (protocolo de bus, em português) e é onde é definido se um determinado bus é **paralelo** em **série**. Um **bus paralelo** é um bus no qual os dados são **transmitidos** em **paralelo**, o que é bom, dado que se torna **muito rápido**, mas, em **simultâneo**, é **mau** porque torna-se **caro** efetuar uma transmissão à distância, dado que pode dar-se o caso de interferência entre várias linhas a alta frequência. Por outro lado, um **bus série** é um bus no qual os **dados são transmitidos em série**, o que por si se torna **barato**, no ponto de vista da **transmissão à distância**, dado que não é possível que ocorram interferências, mas é consideravelmente **mais lento** que o processo anterior.

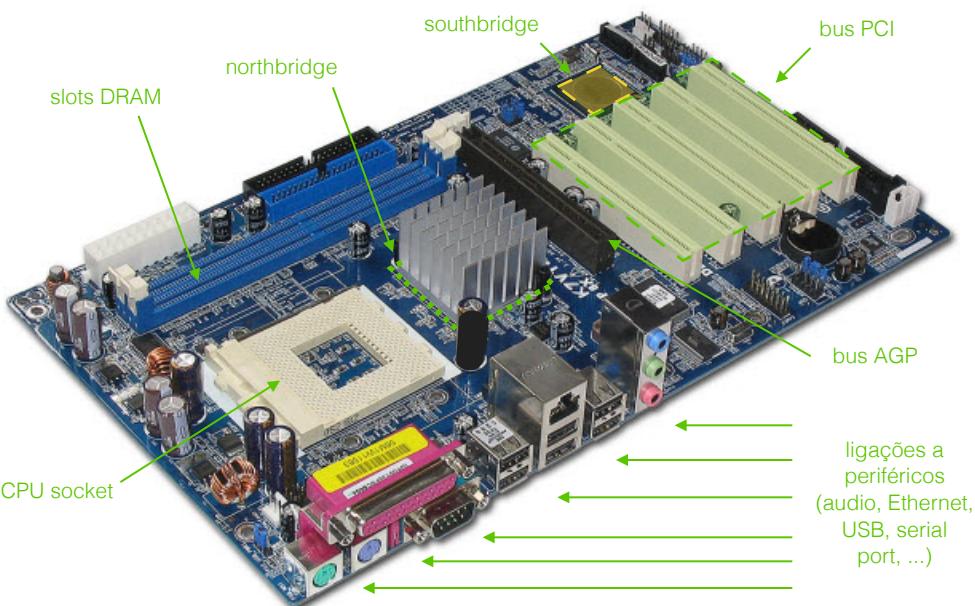
bus protocol
bus paralelo

bus série

Os **buses** podem também ter **vários tipos**, como já foi referido atrás. Três possíveis tipos de implementação são o **processor-memory bus**, o **I/O bus** e o **backplane**.

bus. Um **processor-memory bus** é um bus que é específico do processador, dado que este se liga diretamente ao processador, só necessita de ser compatível com o sistema de memória (o que maximiza a largura de banda das comunicações memória-processador), é curto e muito rápido e é otimizado para a transferência de blocos da cache. Um **I/O bus** é um possível standard de indústria, o qual se liga ao bus processador-memória ou ao bus *backplane*, sendo usualmente mais comprido, mas mais lento, e permite a ligação de dispositivos muito diversos. Finalmente, o **backplane bus** é um bus que se caracteriza como uma estrutura de ligação no *chassis*, isto é, liga diferentes PCB's.

Os computadores pessoais atuais incluem diversos buses ligados por bridges. Num conceito de organização da arquitetura de computadores foram especificadas duas partes como partes integrantes da ligação de buses até ao processador central. A essas duas partes deu-se o nome de **northbridge** e **southbridge**. O **northbridge** estava relacionado com o bus que se ligava ao processador central, enquanto que o **southbridge** fazia respeito à base de controlo dos módulos de I/O. Podendo, os buses, operar em simultâneo, a partir de 2008, passou-se a implementar nos computadores pessoais a tecnologia de **Platform Controller Hub** (PCH), na qual o northbridge está integrada (intrinsecamente) no processador central (CPU). Na Figura 21 podemos ver uma **motherboard** de um computador pessoal.



Como podemos reparar, pela Figura 21, os computadores pessoais são constituídos de uma grande diversidade de buses. Num primeiro plano (**back-side bus**) temos uma ligação entre o **CPU** e a **L2 cache** (assunto a estudar mais à frente). Num segundo plano (**front-side bus**) temos uma ligação entre o **CPU** e o **northbridge** chipset. Na ligação com a memória temos o **memory bus**, que liga o **northbridge** com a memória principal. Outras ligações são a plataformas geridas pelos buses PCIe (PCI Express) ou AGP e ligam o **northbridge** ao GPU. Em relação a periféricos, estes são geridos pelos buses PCI, FireWire, USB, entre outros, que ligam a **motherboard** aos dispositivos periféricos.

Vamos assim ver, um por um, alguns dos buses mais importantes que podemos encontrar nos nossos computadores pessoais e no micro-controlador PIC-32.

processor-memory bus

I/O bus

backplane bus

northbridge

southbridge

Platform Controller Hub

motherboard

figura 21

motherboard

back-side bus

front-side bus

memory bus

23 ARQUITETURA DE COMPUTADORES II

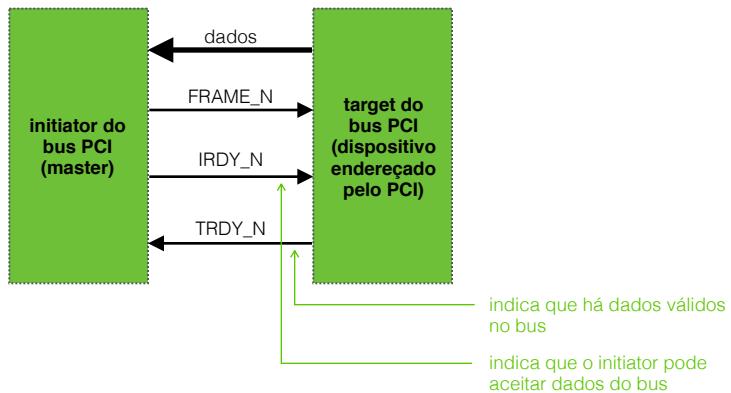
PCI

Um dos buses mais importantes dos nossos computadores pessoais é o **bus PCI**. Sigla de *Peripheral Component Interconnect* é um bus síncrono com um relógio de 33 MHz ou 66 MHz, multiplexado com linhas comuns (entre 32 a 64 linhas de endereço/dados), que é possível ser encontrado em ambas versões de 32 ou 64 bits, de arbitragem centralizada e que serve de bus de ligação a periféricos nos nossos computadores pessoais.

As transferências de dados no bus PCI é feita de dois modos diferentes. Uma possibilidade é a execução de uma transferência **item-a-item**, onde são levados até ao PCI pares (endereço, dado) que depois, este, colocará o "dado" no respetivo "endereço". Uma alternativa é o método **burst**. Com este método é possível apontar inicialmente para um endereço e quaisquer dados que sejam levados até ao PCI serão sempre contíguos aos inseridos a partir do endereço fornecido.

No bus PCI quem inicia um comando (**initiator**) é master e quem recebe o comando (**target**) é slave. Dado isto, estamos em perfeitas condições de verificar como é que se comporta o bus PCI numa operação de leitura e numa operação de escrita.

Numa operação de leitura através do bus PCI, na Figura 22, encontramos representado o respetivo diagrama de blocos.



bus PCI

item-a-item

burst

initiator

target

figura 22
diagrama de blocos PCI
para leitura

Para completar a Figura 22, a respeito do significado dos sinais de controlo, o PCI tem como estrutura pinos gerais do sistema - que são linhas de relógio (*CLK*) e de reset (*RST#*) -, pinos de endereço e de dados - que são linhas multiplexadas para endereços e dados (*AD[31..0]*) e linhas multiplexadas de comando ou *byte enable* (*C[3..0]* ou *BE[3..0]*) - e pinos de controlo da interface - como são o caso o *FRAME#* (indicam o início e a duração da transação), o *IRDY#* (significando *initiator ready*, que em leitura indica que o master está preparado para aceitar dados, e em escrita indica que há novo dado válido em *AD*), o *TRDY#* (significando *target ready*, que em escrita indica que o slave está preparado para aceitar dados, e em leitura indica que há novo dado válido em *AD*) e *DEVSEL#* (significando *device select*, que é ativado pelo target quando este reconhece o seu endereço, indicando ao master atual (*initiator*) que foi selecionado um dispositivo).

Em termos de diagrama temporal podemos ver a Figura 23. Detalhadamente, a seguinte lista ordenada explica o significado de cada evento:

- **passo 1** - master que obteve o controlo do bus (*initiator*) aciona *FRAME* que se mantém até à última fase da transferência e coloca o endereço do dispositivo nas linhas *AD*;

- **passo 2** - o dispositivo alvo (*target*) reconhece o endereço nas linhas *AD*;
- **passo 3** - o *initiator* retira-se do bus *AD* para preparar a sua utilização pelo *target* (ocorre um **turn-around cycle**) e ativa *IRDY* para indicar quando é que este se encontra pronto a receber o dado;
- **passo 4** - o *target* ativa *DEVSEL* para indicar que reconheceu o seu endereço, pelo que coloca o dado no bus *AD* e ativa *TRDY* para o assinalar;
- **passo 5** - o *initiator* lê o dado no início do quarto ciclo e muda as linhas *BE* em preparação para a leitura seguinte;
- **passo 6** - *target* fica com estado "not ready" para enviar dado, isto é, *TRDY* fica desativado (ocorre um **wait state**). Num quinto ciclo o *target* coloca o segundo dado em *AD*;
- **passo 7** - num sexto ciclo, *target* coloca dado em *AD*, mas *initiator* fica com o estado "not ready", pelo que *IRDY* fica desativado;
- **passo 8** - *initiator* desativa *FRAME* sinalizando que a leitura é a última, ativando de seguida *IRDY*, lendo o terceiro dado no início do oitavo ciclo;
- **passo 9** - finalmente, *initiator* desativa *IRDY*, colocando o bus no estado inativo.

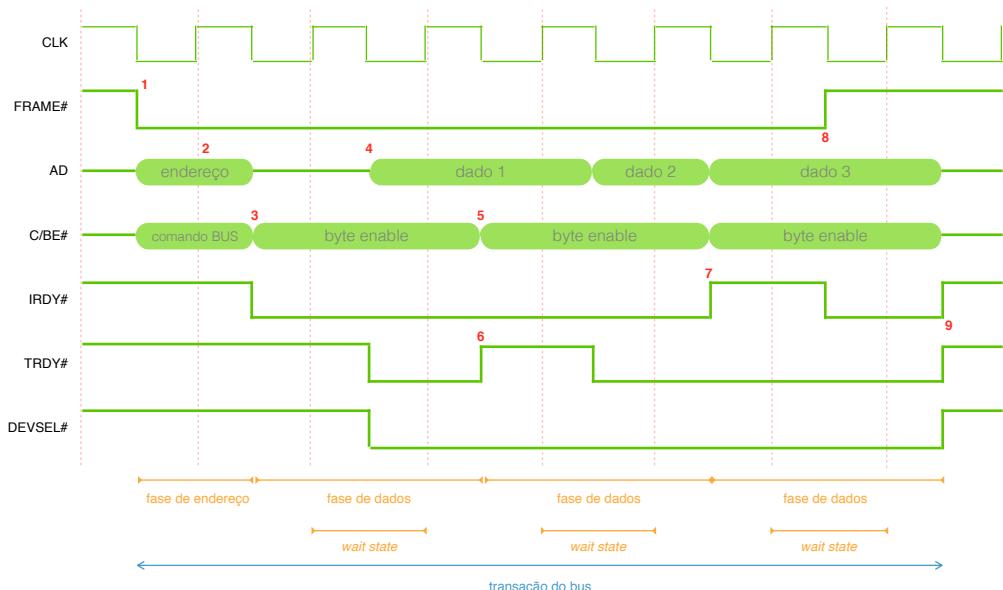


figura 23
diagrama temporal em
leitura no PCI

Numa operação de escrita, o bus PCI comporta-se de forma muito semelhante e algo análoga. Na Figura 24 podemos ver o diagrama de blocos que o exemplifica.

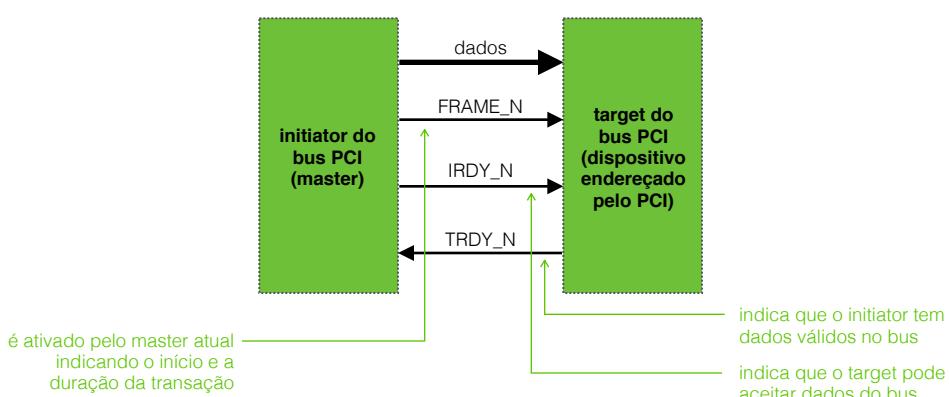


figura 24
diagrama de blocos PCI
para escrita

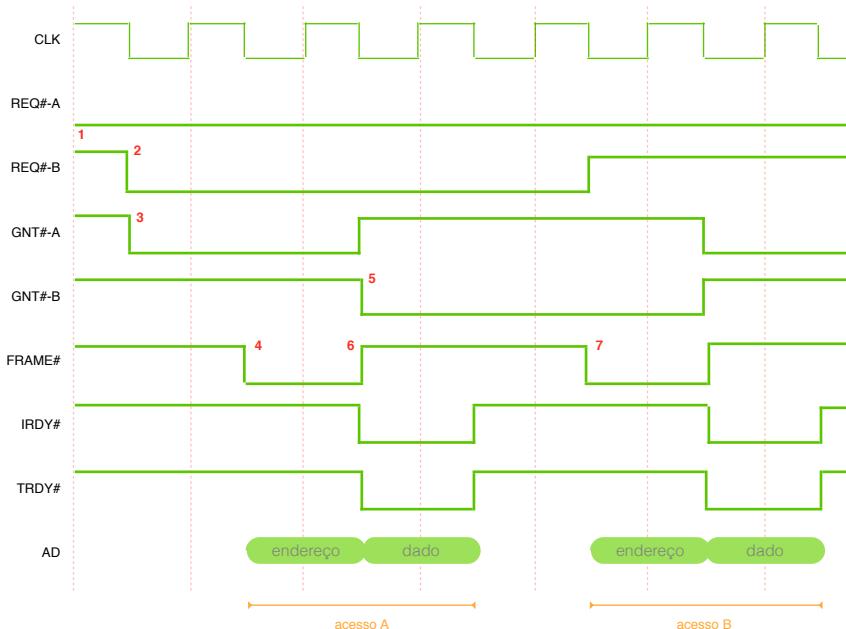
25 ARQUITETURA DE COMPUTADORES II

De acordo com várias possíveis combinações de $C/BE[3:0]$, vários tipos de ciclos do bus podem ocorrer. Uma tabela que contenha essa informação pode ser encontrada na documentação que normalmente precede o bus.

Uma outra questão muito importante de estudar é a **arbitragem**. Num bus PCI a arbitragem (controlo de operações) é feita de modo **central** (é centralizada), pelo que a especificação do bus não impõe qualquer algoritmo de arbitragem. Sendo assim, poder-se-á utilizar qualquer algoritmo, desde o **Round-Robin** a **FCFS** ou a uma questão gerida por **prioridade**.

Em termos de diagrama temporal, podemos ver a arbitragem conforme o representado na Figura 25, sendo que a sua interpretação se encontra representada na lista abaixo:

- **passo 1** - A pede utilização do bus;
- **passo 2** - durante o primeiro ciclo, B pede a utilização do bus;
- **passo 3** - bus é atribuído primeiramente a A ($GNT\#-A$);
- **passo 4** - no início do segundo ciclo A verifica que lhe foi atribuído o bus, pelo que ativa *FRAME* e coloca endereço-alvo em *AD*;
- **passo 5** - árbitro examina as linhas *REQ* no início do terceiro ciclo e concede o controlo do bus a B ($GNT\#-B$);
- **passo 6** - A desativa o seu *FRAME*, pelo que indica que está em curso a sua última transferência, colocando o dado em *AD* e ativando *IRDY*. O dado é assim lido no início do quarto ciclo;
- **passo 7** - B verifica que *FRAME* e *IRDY* estão desativados e toma controlo do bus.



arbitragem

central

Round-Robin, FCFS
prioridade

figura 25
diagrama temporal de
arbitragem

PCIe

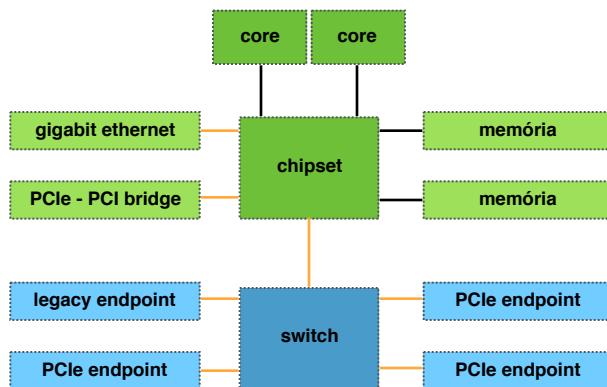
Com a evolução da tecnologia começaram a surgir problemas no transporte de dados pelos buses, dado que surgem limitações de nível físico no aumento da frequência de relógio com o uso de buses partilhados. A utilização de buses com ligações ponto-a-ponto começa a ser mais comum porque há menos latências e maiores

velocidades de transferência. A limitação física no bus partilhado provém da necessidade de fazer sincronização e arbitragem em tempo útil.

Entre buses em paralelo e em série, os paralelos, por si, têm algumas desvantagens em termos do aumento da taxa de transferência e do aumento do número de linhas como **timing skew** (quando um mesmo sinal de relógio chega a componentes diversos a tempos diferentes), consumo de potência ou interferências eletromagnéticas como **crosstalk**. Já os buses série não têm *timing skew* nem *crosstalk*, pelo que é permitida a execução de tarefas a frequências mais elevadas que o normal. Através de ligações ponto-a-ponto (ou de *hubs*) criam-se assim buses série como a **SATA** (*Serial ATA*) - usados nas ligações de unidades de disco rígido -, **USB** (*Universal Serial Bus*) ou **FireWire**.

Nasce assim o **PCIe**, sigla inglesa para **PCI Express**, bus cujas ligações são ligações ponto-a-ponto (há ligações diretas entre cada dois componentes, o que elimina a necessidade de arbitragem), com grande capacidade de suportar aplicações como **Gigabit Ethernet** ou **Data Streams** (como o *video-on-demand* (VOD) ou distribuição de áudio, ambas exigindo respostas em modo **real-time**). A comunicação tem de ser assim **encapsulada** (formatada) em pacotes ao invés de ser em sequências diretas de bits.

Na Figura 26 podemos encontrar a configuração típica de um PCIe, sob a forma de um diagrama de blocos (a laranja são ligações pelo PCIe).



Os dados enviados em pacotes, pelo PCIe, resultam de vários protocolos denominados de **TLP** (*Transmission Layer Packets*) e **DLLP** (*Data Link Layer Packets*). Estes protocolos estão organizados, tal como os nomes o indicam, em camadas - **OSI** (*Open Systems Interconnection*). Em termos do **TLP**, sabemos que as transações no bus são transferências de pacotes de dados, processo este que origina no dispositivo que envia e termina no dispositivo que recebe o pacote. Um pacote deste espécime, semelhante ao representado na Figura 27, contém várias secções. De dentro para fora, temos uma primeira parte a que corresponde o TLP, no qual se distingue um **header** (onde se descreve o tipo de pacote e inclui a informação necessária para o receptor processar o pacote), uma secção de dados - **data** - (que pode possuir dados até 4096 bytes) e um campo **ECRC**, opcional (que permite a destinatário verificar se há erros nos campos *header* e *data*). Continuando temos a parte da responsabilidade do DLLP, a qual contém um campo *Sequence Number* (no topo) e um campo *LCRC* (em baixo). Finalmente, já da responsabilidade final da camada física do modelo OSI, são adicionados *frames STP* no topo e a baixo de todas as outras secções.

timing skew

crosstalk

SATA

USB

FireWire

PCIe, PCI Express

Gigabit Ethernet, Data

Streams, real-time

encapsulada

figura 26
configuração de PCIe

OSI, TLP

header

data

ECRC

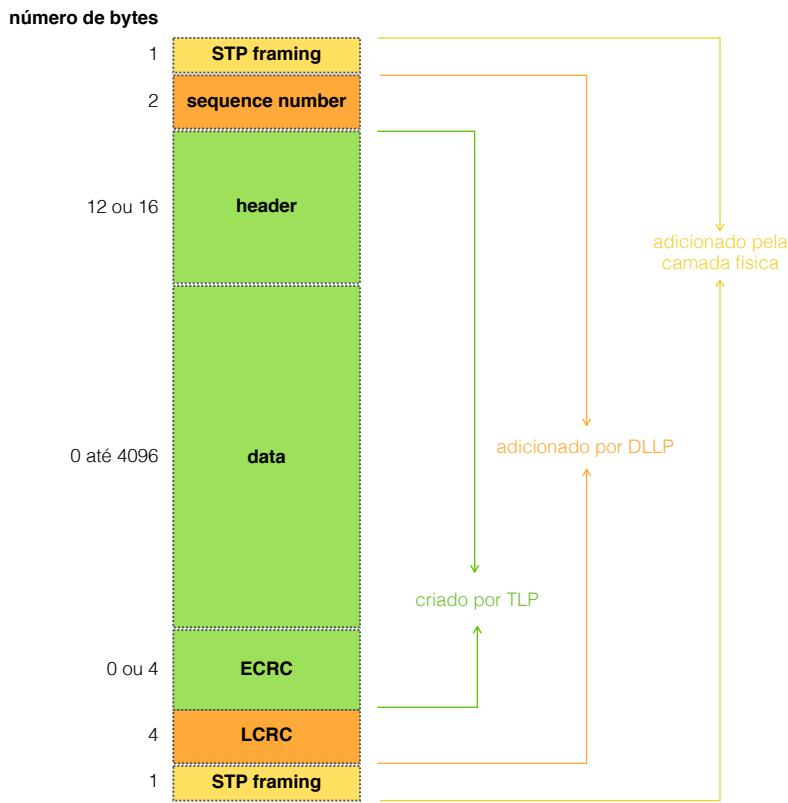


figura 27
pacote PCIe

4. Comunicação Série

Com a integração dos buses foi permitida uma melhor **comunicação** entre os diversos componentes que compõem um sistema computacional. Contudo um problema ainda se coloca: sobre que forma é que um determinado construtor cria a arquitetura de um dado componente, em termos de comunicação?

Para que os componentes possam efetuar comunicações o mais próximo possível do uníssono, há que criar normas de construção - **standards**. Foram assim criadas duas alternativas: a primeira alternativa é a **comunicação em paralelo**. Numa comunicação em paralelo, tal como o nome o indica, há a transmissão de múltiplos bits (tipicamente um ou mais bytes) através de múltiplas vias de ligação. Este tipo de comunicação é usado no PIC32 *Parallel Master Port* (e era usado, embora já obsoleto, na comunicação com impressoras com a porta *Centronics* (Figura 28)) - é usada para distâncias curtas. Uma outra alternativa é a **comunicação série**. Numa comunicação série, por outro lado, temos uma transmissão bit-a-bit. Exemplos de aplicação de comunicação série são a RS-232C (porta série dos computadores pessoais), a I²C (circuitos inter-integrados), SPI (*Serial Peripheral Interface*) ou a USB (*Universal Serial Bus*) - Figura 28.

comunicação

standards

comunicação em paralelo

comunicação série



centronics



USB



RS-232C

figura 28
comunicação série

Dentro da comunicação em série podemos definir três tipos de comunicação: síncrona, assíncrona, isócrona. Uma comunicação em série **síncrona** significa que a transmissão de dados é acompanhada da transmissão do sinal de relógio (caso do SPI) ou o sinal de relógio é recuperado da informação transmitida (caso do USB, I²C ou CAN). Uma comunicação **assíncrona** significa que não é transmitido o sinal de relógio, pelo que também não há de haver recuperação deste (como é o caso do RS-232C). Por último, uma comunicação **isócrona** significa que o tempo entre dados sucessivos é igual à unidade de tempo básica do sistema de transmissão (ciclo de relógio) ou a um seu múltiplo - como é o caso de *data streaming*.

Transmissão síncrona

Numa transmissão síncrona, tal como já foi referido, os relógios do transmissor e do receptor têm de se manter sincronizados. Quando o relógio não é transmitido, este deve ser extraído dos dados transmitidos a partir das transições de sinal na linha de dados.

Na transmissão síncrona, para manter esta característica de sincronicidade dos relógios, a mensagem deve consistir num grupo de bits que formam um bloco de dados, em lugar da transmissão separada de cada carater como iremos ver na transmissão assíncrona. Este **bloco de dados** completo é assim transmitido com carateres especiais no início e no final, formando todo o conjunto, uma unidade de informação.

síncrona

assíncrona

isócrona

Transmissão assíncrona

Numa transmissão assíncrona, contrariamente às transmissões síncronas, efetuam-se passagens carater-a-carater. Para transmitir um dado (um carater) é então necessário acrescentar bits para sinalizar o princípio e o fim da transmissão do mesmo. Usam-se assim artefactos como o **start bit** - sendo, por exemplo, um bit com valor lógico '0', indica que se está a iniciar uma transmissão de um carater - ou um **stop bit** - - podendo ser um ou dois bits, ambos com valor lógico '1' (por exemplo), indicam o fim da transmissão do carater.

start bit

stop bit

Quando não se está a transmitir, a linha assíncrona encontra-se num estado definido como **mark state**, permanecendo com o valor lógico '1', de forma a se poder distinguir da linha desligada. A partir do momento que se efetua a transição entre os valores lógicos '1' e '0', respetivamente, é assinalado o início da transmissão.

mark state

A transmissão de um carater pode ainda contar com um **bit de paridade** por questões de erros. O bit de paridade é uma forma leve de verificar se houve erros na transmissão, pelo que este avalia se a quantidade de valores lógicos '1' na sequência bits é par (caso seja '1') ou ímpar (caso seja '0').

bit de paridade

Interface RS-232

A **interface RS-232** surgiu em 1969 como um standard para comunicação em série assíncrona. Esta comunicação era efetuada entre um **Equipamento Terminal de Dados** (DTE) - em inglês *Data Terminal Equipment* - como é exemplo o nosso computador pessoal, e um **Equipamento de Comunicação de Dados** (DCE) - em inglês *Data Communications Equipment* - como é exemplo um **modem** (*modulator-demodulator*). Esta interface permite que seja estabelecida uma comunicação bidirecional e **full-duplex**, isto é, permite que se recebam dados e que se envie outros dados em simultâneo.

interface RS-232

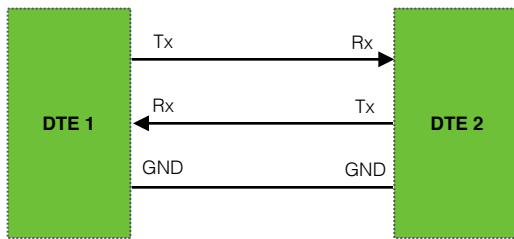
Equipamento Terminal de Dados, Equipamento de Comunicação de Dados, modem, full-duplex

Inicialmente tinha-se como objetivo de uma interface RS-232 apenas a ligação de DTE's a modems, mas a utilidade de tal standard estendeu-se muito para além do seu objetivo inicial. Contudo, como aparecimento do USB (que vamos referir mais à frente), os computadores pessoais deixaram de disponibilizar portas RS-232.

Por ser um modo de comunicação série muito fácil de implementar e de programar, este ainda continua a ser muito usado em micro-controladores. Apareceram assim no mercado conversores de USB para RS-232 que permitem a ligação a computadores pessoais de equipamentos que implementam RS-232.

Mas a interface RS-232 não é totalmente sã, em termos de implementação. A norma RS-232 tem alguns problemas: a nível físico, os níveis lógicos são codificados com tensões simétricas (por exemplo +5V e -5V); o consumo de energia é elevado; a sinalização é **single-ended** (constituído por dois cabos paralelos, um transporta um sinal que varia (sinal de transmissão) e o outro transporta a massa) o que pode criar fenómenos de *crosstalk*, dada a baixa imunidade ao ruído, e impõe limitações na velocidade e distância; apenas suporta ligações ponto-a-ponto; e a norma era suficientemente vaga para permitir implementações proprietárias que, na prática, dificultavam ou mesmo impossibilitavam a interligação entre equipamentos de fabricantes diferentes.

Na sua forma mais simples, a implementação da norma RS-232 requer apenas a utilização de duas linhas de comunicação e uma linha de massa, tal como já foi referido anteriormente e como podemos verificar na Figura 29.



single-ended

figura 29
implementação de RS-232

Podem ser usadas linhas adicionais para protocolar a troca de informação entre os dois equipamentos, através de um protocolo de **handshake**. É o caso de **RTS** (*request to send*), **CTS** (*clear to send*), **DTR** (*data terminal ready*), **DSR** (*data set ready*) ou **DCD** (*data carrier detected*).

**handshake, RTS
CTS, DTR, DSR
DCD**

Estrutura de uma trama RS-232

Tal como foi referido atrás, dado que estamos a tratar de uma comunicação série assíncrona, é natural que os parâmetros estabelecidos para tal comunicação na secção §§Transmissão assíncrona, se mantenham. No caso da trama RS-232 esta deve ser constituída por um *start bit* (com o valor lógico '0'), um ou dois *stop bits* (sendo que estes devem coincidir com o estado de linha inativa (com o valor lógico '1') - - *idle* - e devem proporcionar um intervalo de tempo de guarda mínimo entre o envio consecutivo de dois valores) e, de forma opcional, um bit de paridade.

Em relação ao bit de paridade é conveniente frisar que há a possibilidade de se definir, por vias do controlador de comunicação, se se pretende confirmar uma paridade par (em inglês *even*) ou uma paridade ímpar (em inglês *odd*). O que aqui acontece é que para o caso da paridade par, o controlador terá que efetuar uma operação XOR bitwise de todos os bits da sequência. Pelo contrário, na paridade ímpar, ter-se-á que efetuar uma operação XNOR bitwise.

Vejamos assim um exemplo de uma transmissão de dados (caracteres). Estabeleçamos como parâmetros de comunicação 7 bits de dados, 2 *stop bits* e paridade ímpar. A trama gerada pelo controlador de comunicação série RS-232 para transmitir o valor 0x43 é a representada na Figura 30.



figura 30
trama gerada pelo RS-232

A trama codificada em níveis da interface RS-232, isto é, à saída do *driver* de linha é a representada na Figura 31.

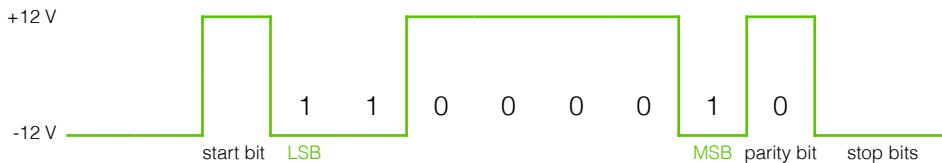


figura 31
trama codificada pelo
RS-232

Numa ligação física RS-232 os bits da trama são codificados em **NRZ-L** (*non-return-to-zero level*). Com esta codificação, o nível lógico '0' assume uma codificação com uma tensão positiva (na gama de +3V a +25V) e o nível lógico '1' assume uma codificação com uma tensão negativa (na gama de -3V a -25V). A codificação e descodificação da trama com estes níveis de tensão é assegurada por circuitos eletrónicos designados por **drivers de linha**.

Um aspecto importante a referir é um **baudrate**. O baudrate é, em termos globais, o número de símbolos transmitidos por segundo. A cada símbolo pode corresponder um ou mais bits de dados. A taxa de transmissão de dados bruta (em inglês **gross bit rate**) corresponde ao numero de bits transmitidos por segundo (**bps**). Assim, o baudrate não deverá ser confundido com *gross bit rate*.

No caso da interface RS-232, a cada símbolo está associado um único bit (duas tensões distintas), pelo que o baudrate e o *gross bit rate* coincidem.

No exemplo que considerámos na Figura 30 e na Figura 31 o número de bits a serem transmitidos são 11. Considerando um baudrate de 57600 bps a transmissão completa de uma trama demora aproximadamente 191 µs ($11/57600 \approx 191 \mu s$). O bit rate líquido é $(7 \times 57600) / 11 = 36655$ bps. Já o *word rate* é $1 / 191 \mu s = 57600 / 11 = 5236$ palavras por segundo (palavras de 7 bits).

NRZ-L

drivers de linha
baudrate

gross bit rate, bps

Receção de dados e sincronização na interface RS-232

Sendo que estamos a trabalhar de forma assíncrona, a sincronização do relógio terá de ser feita de forma implícita, isto é, ter-se-á de descobrir uma forma de tornar "síncronos" os relógios independentes do transmissor e do receptor. Isto é resolvido através da configuração do transmissor e do receptor de forma igual - com o mesmo número de bits de dados, tipo de paridade, número de *stop bits* e baudrate. Uma das razões pela qual isto deve ser assim é porque o relógio do receptor é sincronizado no início da receção de cada nova trama.

Para uma boa receção de dados o receptor deve sincronizar-se pelo flanco descendente (do valor lógico '1' para o valor lógico '0') da linha (*start bit*) e, idealmente, fazer as leitura a meio do intervalo reservado a cada bit de dados. Esta aplicação pode ser verificada na Figura 32, onde se tenta representar um exemplo da receção do valor 0x43, configurado a 7 bits de dados, paridade par e 2 *stop bits*.

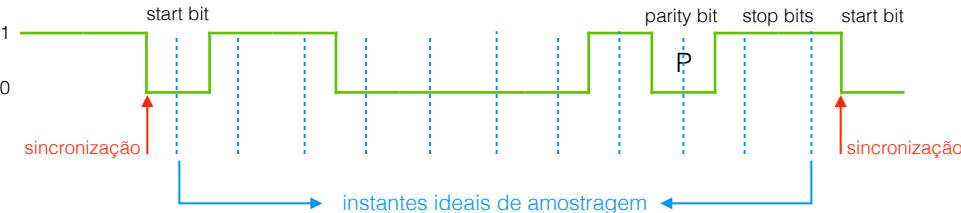


figura 32
recepção sincronizada

Entre instantes de sincronização o desvio dos relógios depende da estabilidade ou da precisão dos relógios do transmissor e do receptor. Caso a receção não seja corretamente efetuada devido a um desvio da frequência dos relógios do transmissor e do receptor, a curto ou a longo prazo, é possível que ocorram **erros tanto de paridade** (o bit de paridade devia ser '0' e é '1', por exemplo) ou de **framing** (é **detectado um nível lógico '0'** quando era esperado um **'stop bit'**).

Os erros nos instantes de amostragem podem ter duas causas distintas: é **erro de fase** (Δ_1), isto é, é um erro cometido ao determinar o instante inicial de sincronização; é **erro provocado por desvio de frequência**, pelo que a **frequência** dos relógios do transmissor e do receptor não são exatamente **iguais**. Este segundo erro é **cumulativo** e **proporcional ao comprimento da trama**.

É habitual usar-se, na receção, um relógio local (por exemplo $LCLK$) cuja frequência, idealmente, seja igual a $n \times f_{TCLK}$, onde f_{TCLK} é a frequência do relógio de transmissão ($f_{TCLK} = 1 / T_{bit}$) e n é habitualmente um valor entre 4, 16 ou 64, sendo este calculado através da expressão k_{sample}/r_{div} , sendo k_{sample} o fator de amostragem e r_{div} o fator de divisão do relógio. Este relógio $LCLK$ é tipicamente utilizado para incrementar um contador e não é, habitualmente, sincronizado com o sinal da linha, pelo que se impõe um erro de fase (que é sempre inferior a um período).

Considerando um fator de sobreamostragem (n) de 16, o tempo de bit equivale a 16 períodos do relógio local ($16T_{LCLK}$). Usando um contador de 4 bits como divisor de frequência a 16 a sincronização é trivial: basta **fazer o reset** quando é **detetado o start bit**. As transições ascendentes do bit 3 do contador definem assim os instantes de amostragem.

Em termos do desvio máximo de frequência entre o emissor e o receptor temos que é comum considerar-se como zona segura de amostragem do bit, num pior caso (quando temos cabos longos com efeito capacitivo pronunciado, velocidades de transmissão elevadas, ...) $\pm 25\%$ do tempo de bit, em torno do meio e, num caso ideal (quando temos cabos curtos e de acordo com as especificações, tal como velocidades moderadas, ...) $\pm 37.5\%$ do tempo de bit, em torno do meio.

O **erro de frequência**, como já foi referido, é **cumulativo** e **diretamente proporcional ao comprimento da trama**. Sendo assim, é necessário garantir que o último bit é amostrado dentro da zona segura. Em termos do relógio local (supondo que o nosso n é agora 16) temos, no caso da trama mais longa (8 bits de dados, com bit de paridade), formando $10.5 \times 16 = 168$ períodos. Dado isto, a máxima discrepância que poderá ser tolerada entre os relógios do transmissor e do receptor é $\Delta T = \pm 3 / 168 \approx \pm 1.8\%$, no pior caso, a $\Delta T = 5 / 168 \approx \pm 3.0\%$, num caso ideal.

Por exemplo, se tivermos uma taxa de transmissão de 115200 bps, com 8 bits de dados, bit de paridade e um $n = 16$, para que a comunicação se processe sem erros, o relógio do receptor deverá ter, no caso mais desfavorável, uma frequência de $T_{LCLK} = (1 \pm 0.018) / (16 \times 115200)$, logo $f_{LCLK} \in [1810609, 1876986]$ Hz.

Interface SPI

Como já foi referido anteriormente, a **interface SPI**, acrônimo de *Serial Peripheral Interface* (em português Interface Periférica em Série) foi definida, inicialmente pela Motorola®. O SPI é utilizado para comunicar com uma grande variedade de dispositivos como sensores de temperatura, pressão, ..., cartões de memória (como os MMC ou SD), circuitos de memória, ADC's, DAC's, *displays* LCD, comunicação entre o corpo de máquinas fotográficas e as lentes, entre micro-controladores, entre outros. Fornece, em suma, ligações a curtas distâncias.

Esta interface possui uma arquitetura do tipo *master-slave* com ligação ponto-a-ponto, contendo uma comunicação **bidirecional full-duplex**, funcionando em modo **data exchange**, pelo que por cada bit que é enviado para o receptor, também é recebido um. Isto é, ao fim de *N* ciclos de relógio o transmissor enviou uma palavra de *N* bits e recebeu uma palavra com a mesma dimensão. Sendo fácil de implementar em hardware e software, com ausência de *line drivers*, tem uma **comunicação síncrona** (recebe um relógio explícito do master). Tal relógio é gerado pelo master que disponibiliza para todos os *slaves*. Não é exigida precisão ao relógio, dado que os bits vão sendo transferidos a cada transição do relógio, o que permite utilizar um oscilador de baixo custo no *master* (não é necessário um cristal de quartzo).

Tendo uma arquitetura *master-slave*, o sistema só pode ter um *master*, pelo que é o único que tem a capacidade de controlar o relógio, no sistema. Como já vimos, um *master* pode estar ligado a vários *slaves*, mas no entanto, apenas um *slave* é selecionado pelo *master* para comunicar de cada vez, sendo que é, como seria de esperar, o *master* que inicia e controla a transferência de dados.

interface SPI

data exchange

comunicação síncrona

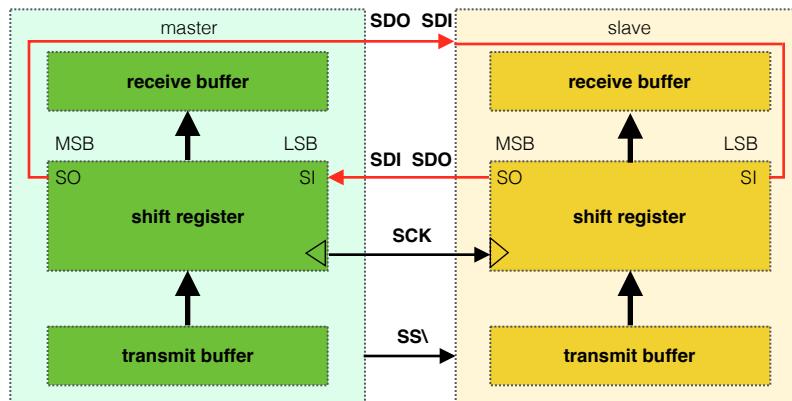


figura 33
comunicação no SPI

A comunicação no SPI, tal como podemos verificar pela Figura 33, envolve dois **registos de deslocamento** (um no master, outro no slave) ligados em anel. Primeiramente a informação é transferida do *transmit buffer* para o registo de deslocamento, depois a informação é transferida para o outro registo de deslocamento, um bit de cada vez, pelo que finalmente a informação é transferida do registo de deslocamento para o *receive buffer*.

A sinalização, conforme a representada na Figura 33, depende do tipo de função: para dados temos **SDO** (*serial data out*) e **SDI** (*serial data in*) e para controlo temos **SS\>** (*slave select*), cujo sinal é ativado pelo master para selecionar o *slave* com que vai comunicar, e **SCK** (*serial clock*).

O sinal de relógio tem um duty-cycle de 50%, pelo que o *master* utiliza uma transição do relógio para colocar um bit de informação na linha e o *slave*, por sua vez, usa a outra transição do relógio para o armazenar.

Em suma, em termos de operação temos que o *master* configura o relógio para uma frequência igual ou inferior à suportada pelo *slave* com quem vai comunicar. Sendo assim, o *master* ativa a linha *SS* do *slave* com quem vai comunicar. Em cada ciclo do relógio (por exemplo numa transição ascendente) o *master* coloca na sua linha *SDO* um bit de informação que é lido pelo *slave* na transição de relógio oposta seguinte. O *slave*, dado isto, coloca na sua linha *SDO* um bit de informação que é lido pelo *master* na transição de relógio oposta seguinte. O *master* termina assim por desativar a linha *SS* e "para" o relógio, permanecendo, este, estável, sobre o nível lógico '1', por exemplo - só existe relógio durante uma transferência. No final de tudo, o *master* e o *slave* trocaram o conteúdo dos seus registos de deslocamento.

Os sinais de seleção *SS* são totalmente independentes. Em cada instante apenas um *SSx* está ativo, isto é, apenas um *slave* está selecionado. Os sinais *SDO* dos *slaves* selecionados ficam assim em alta-impedância. O que acontece é que o sinal de *chip select* é comum com o *SDO/SDI* ligados em cascata. Sendo assim, a saída de dados de cada *slave* encontra-se ligada à entrada de dados dos seguintes. Os *slaves* podem ser assim vistos como um único dispositivo de maior dimensão.

As transferências no SPI podem ser de vários tipos. Como já foi visto, o SPI funciona em modo *data exchange*, pelo que o processo de comunicação envolve a troca do conteúdo dos registos de deslocamento do *master* e do *slave*. Cabe então aos dispositivos envolvidos na comunicação decidir qual a validade da informação. Assim, podem considerar-se os seguintes cenários de transferência: **bidirecional** - são transferidos dados válidos em ambos os sentidos (*master* para *slave* e vice-versa); *master* para *slave* (operação de escrita) - *master* transfere dados pretendidos para o *slave* e ignora/descarta os dados recebidos; *slave* para *master* (operação de leitura) - o *master* desencadeia transferência quando pretende ler dados do *slave*. O *slave*, por sua vez, ignora/descarta os dados recebidos.

Em termos de configuração, antes de iniciar a transferência (qualquer que seja) há algumas configurações que, normalmente, são efetuadas no *master* para adequar os parâmetros que definem a comunicação às características do *slave* com quem vai comunicar. Sendo assim há que configurar primeiramente a frequência do relógio (baudrate) e só depois especificar qual o flanco do relógio é que deve ser usado para a transmissão (a receção é efetuada no flanco oposto). Esta configuração é feita em função das características do *slave* com o qual o *master* vai comunicar: a transmissão pode ser efetuada no flanco ascendente (consequentemente, a receção é efetuada no flanco descendente) ou a transmissão pode ser efetuada no flanco descendente (consequentemente, a receção é efetuada no flanco ascendente).

Interface SPI no PIC32

No PIC32MX795F512H, micro-controlador que estudamos nas aulas práticas, há três dispositivos de comunicação SPI disponíveis. Cada um dos dispositivos pode ser configurado para funcionar como *master* ou como *slave*. Tendo o comprimento de palavra configurável entre 8, 16 e 32 bits, tem registos de deslocamento separados para receção e para transmissão e os *buffers* para receção e para transmissão são baseados no conceito FIFO (fila) - têm 16 posições se o comprimento de palavra for 8 bits, 8 posições se o comprimento de palavra for 16 bits ou 4 posições se o comprimento de palavra for 32 bits. Estes dispositivos podem ser configurados para gerar interrupções em função da ocupação das filas.

Interface USB

A **interface USB** (*Universal Serial Bus*) permite a ligação de periféricos a computadores de um modo estandardizado, com uma norma protocolar - dispositivos interagem diretamente com o sistema de operação do computador sem intervenção do utilizador (interação **plug and play**). Esta interface permite a ligação e a retirada de qualquer periférico a qualquer momento, sem necessidade de desligar o sistema (tecnologia **hot-swap**). Do mesmo modo, pode-se fornecer energia aos periféricos através do mesmo cabo utilizado para comunicação de dados.

Uma grande vantagem do USB prende-se ao facto deste permitir ligar periféricos com necessidade de taxas de transferência de dados muito mais rápidas que as da interface, por exemplo, RS-232. É precisamente essa a função do USB, substituir então as interfaces RS-232, dadas as suas facilidades de configuração e compatibilidade. No entanto, o USB tem uma pequena desvantagem, que é o facto de exigir software de comunicação de um maior nível de complexidade.

Os computadores pessoais de há 20 anos atrás (sem prejuízo) tinham uma arquitetura, a nível de periféricos, que permitiam que cada periférico tivesse uma porta específica de forma a poderem ser instalados (Figura 34).



interface USB

plug and play

hot-swap

figura 34
portas num PC



figura 35
porta USB tipo C num
MacBook Air

O USB foi então criado por um conjunto de sete empresas, entre as quais a CompaqTM, DEC, IBM[®], Intel[®], Microsoft[®], NEC[®] e Nortel[®]. Todas as empresas tinham como objetivo, neste projeto, facilitar a ligação de periféricos substituindo a diversidade de portas por um único tipo de conector, simplificar a configuração por software de todos os dispositivos ligados ao PC e permitir maiores velocidades de transferência de dados. Os primeiros circuitos integrados que suportaram USB foram produzidos pela Intel em 1995.

O USB teve uma longa evolução e ainda hoje permanece sobre essa rotina. Iniciando, basicamente, com uma versão 1.1, em 1998, permite ligações *half-duplex* a baixa velocidade de 1.5 Mbps e a *full-speed* de 12 Mbps. Prossegue com uma versão 2.0, denominada de *high speed*, com taxa de transferência máxima de 480 Mbps, a comunicação *half-duplex*, em 2000. Em 2008 é criada a versão 3.0 suportando 4.8 Gbps,

numa tecnologia *SuperSpeed*, a comunicação *full-duplex* e em 2013 cria-se o USB 3.1, com taxa máxima de 10 Gbps, tecnologia com o nome de *SuperSpeedPlus*.

O USB permite ligações até 127 dispositivos por porta, sendo que o comprimento máximo da ligação é de 5 metros para USB 2.0.

Um cabo USB tem quatro linhas, delas, uma de +5V, para dispositivos que consomem pouca potência, podendo ser alimentados por USB, outra sendo a massa, e duas sendo de dados (*D+* e *D-*).

As transferências de dados realizam-se entre o software no *host* e um *endpoint* específico no dispositivo. Cada *endpoint* é uma ligação *simplex* que suporta transferências apenas num sentido. Os *endpoints* estão agrupados em **interfaces**. Cada interface está associado a uma única função dispositivo (excetuando o *endpoint zero*, usado para a configuração do dispositivo).

Em suma, numa transferência de dados por USB, cada comunicação é feita entre o *host* e um dispositivo (um **endpoint** do dispositivo - *buffer* do dispositivo que armazena dados recebidos ou dados a serem transmitidos) - este responde às comunicações do *host*. Para enviar ou receber dados, o *host* inicia uma transferência. Cada transferência (**IRP** - *I/O Request Packet*) usa um formato definido para enviar dados, endereço, bits de deteção de erros e informação de estado e de controlo (o formato varia entre o tipo de transferência (se é *in* ou se é *out*)). Cada *endpoint* tem um número (um endereço), um sentido (*in* ou *out*), e tem definido o número máximo de bytes que pode enviar ou receber numa transação. No caso de uma operação *in*, o *endpoint* é um produtor de dados e no caso contrário, o *endpoint* é um consumidor de dados.

Alguns outros conceitos definem o que é a interface, sendo esta o conjunto dos *endpoints in* e *out* para implementar transferências de dados viáveis, um **pipe**, como sendo a ligação entre um *endpoint* num dispositivo e o software em execução no *host*, e um **default control pipe**, sendo este o *pipe* associado aos dois *endpoints zero* (*in* e *out*). Este último é criado sempre que um dispositivo é ligado e recebeu sinal *bus reset*, sendo usado pelo software de sistema para identificar o dispositivo e configurá-lo.

Então, quando é ligado um dispositivo é ligado, o *host* questiona sempre todos os dispositivos ligados ao bus e atribui a cada um deles um endereço. A este processo damos o nome de **enumeration**. Este processo possui oito passos de execução, esses, descritos a seguir:

- **get_port_status** - o *host* deteta a ligação de um novo dispositivo;
- **set_port_feature** - *hub* envia sinal *reset* ao dispositivo e *sets* (define) o bit *PORT_ENABLE* do registo *PORT_CHANGE*;
- **clear_port_feature** - o *host* limpa a *flag* no registo *STATUS_CHANGE*. O dispositivo encontra-se num estado *default*, passando a responder a solicitações do *host* ao seu endereço 0;
- **get_device_descriptor** - o *host* interroga o dispositivo que responde enviando o seu *device descriptor*;
- **set_address** - o *host* atribui um endereço ao dispositivo. Todos os pedidos subsequentes serão feitos a esse novo endereço - o dispositivo está agora no estado *addressed*;
- **get_configuration_descriptor** - o *device driver* procura o descritor da configuração do dispositivo;
- **select_device_driver** - o PC determina qual o *device driver* que suporta o novo dispositivo. Se o *device driver* não estiver em memória, carrega-o;

interfaces

endpoint

IRP

pipe

default control pipe

enumeration

- **set_configuration** - o dispositivo está agora configurado e operacional, pelo que transita para o estado *configured*.

Tal como referido nos passos anteriores, um **device descriptor** é um objeto de forma tabular que descreve o conteúdo e as funções de um determinado periférico USB. Um dispositivo pode ter várias configurações possíveis e os dispositivos mais simples têm uma configuração e uma só interface.

No contexto do USB os dados são transferidos por pacotes. A formatação e interpretação dos dados transportados numa transação no bus é da responsabilidade do software cliente e da função pretendida. O USB disponibiliza diferentes tipos de transferência para corresponder de modo mais ajustado aos requisitos do software cliente e da função que utiliza o *pipe*. Cada tipo de transferência determina várias características do fluxo de dados: formato imposto pelo USB, direção da comunicação, tamanho dos pacotes e código de deteção de erros (por **CRC**).

A transferência de dados também deve ser especificada por tipo de transferência. Existem quatro tipos de transferência de dados possíveis via USB:

- **transferências de controlo** - usadas para configurar um dispositivo quando é ligado;
- **transferências bulk data** - transferência em bloco de volumes significativos de dados sem grandes exigências temporais (caso das impressoras, *scanners*, discos, ...);
- **transferências de dados de interrupção** - transferências de poucos dados a ser feitas num dado intervalo de tempo (caso do rato, teclado, ...);
- **transferências de dados isócrona** - transferências com um tempo máximo de latência (em tempo real) (caso de microfones, câmaras, ...).

As transferências no bus processam-se em **frames**. O *host* envia um sinal **SOF** (*start of frame*) a cada 125 μ s (para *high-speed* buses) ou a cada 1ms (para *full-speed* buses). Cada *frame* contém 1500 bytes. Até 90% da largura de banda disponível é usada pelas *frames*. Se se tentar ligar novos dispositivos que levem a exceder o limite dos 90% a sua ligação é recusada. Os 10% restantes são usados para transferências *bulk* e comandos.

Interface I²C

A **interface I²C** (abreviatura de *Inter-Integrated Circuit*) foi desenvolvida pela *Philips® Semiconductors* (agora *NXP® Semiconductors*) teve a sua primeira versão em 1992 (atualmente encontra-se na versão 6 - junho de 2015). De acordo com a sua criadora define-se como um bus de duas linhas bidireccionais simples para um controlo inter-integrado de circuitos eficiente. Tendo sido inicialmente desenvolvido para o controlo de subsistemas de TV, é implementável em hardware ou software.

Esta interface requer arbitragem, pelo que as suas transações são *master-slave* com opção de haver mais que um *master* (*multi-master*).

As taxas de transmissão são agrupadas em quatro modos distintos: em modo *standard* até 100 Kbps; em modo rápido (*fast*) até 400 Kbps; em modo *fast plus* até 1 Mbps; e em modo *high speed* até 3.4 Mbps.

Antes de avançarmos é conveniente tomar noção de alguns conceitos básicos no paradigma da interface I²C. Um primeiro conceito é o de **transmitter** - dispositivo que envia dados para o barramento. Na resposta, encontramos um **receiver** -

device descriptor

CRC

frames, SOF

interface I²C

**transmitter
receiver**

dispositivo que recebe dados do barramento. Quem controla o início de transferência ou o fim é o **master** - o dispositivo que inicia a transferência, gerando o sinal de relógio e terminando a transferência. Endereçado por este encontramos o **slave**. Nesta interface, como já foi referido antes, há a possibilidade de haver vários *master*, pelo que se diz que é **multi-master** - mais do que um *master* pode tentar, ao mesmo tempo, controlar o barramento sem corromper a comunicação em curso. Dado isto é necessário que haja um mecanismo de **arbitragem** - procedimento para assegurar que, se mais do que um *master* tentar, simultaneamente, controlar o barramento, apenas a um é permitido continuar, sem perturbação da comunicação iniciada pelo *master* vencedor. Para tal acontecer, todos os dispositivos *master* devem estar com detalhes de **sincronização** - procedimento para sincronização dos sinais de relógio de dois ou mais dispositivos.

Então, a transferência numa interface I²C, como também já foi referido, é bidirecional, *half-duplex* e orientada ao byte. O barramento de comunicação apenas necessita de dois fios: *serial data line* (SDA) e um *serial clock line* (SCL). Tendo um barramento *multi-master* devemos ter mecanismos de deteção de colisões e arbitragem, de modo a evitar corrupção de informação, se dois ou mais *master* iniciarem simultaneamente uma transferência, dado que estas envolvem sempre uma relação *master/slave*. Os *masters* podem ser transmissores ou receptores. Cada dispositivo ligado ao barramento é endereçável por software usando um endereço único previamente atribuído. Cada endereço possui 7 bits, pelo que há 112 endereços disponíveis (outros são reservados) e ainda existe uma possível extensão para 10 bits.

Na Figura 36 podemos ver um exemplo de interligação num bus I²C.

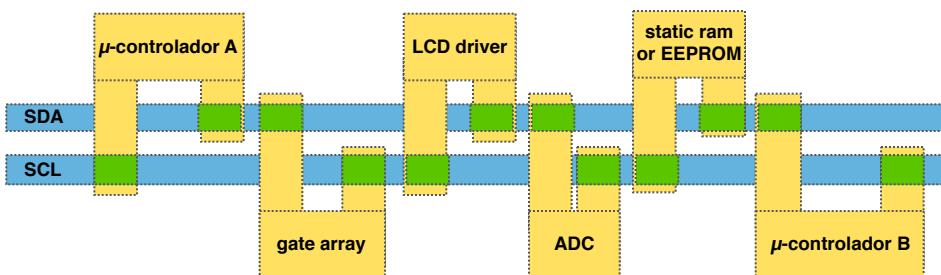


figura 36
interligação no bus I²C

Em termos de sinalização, a interface I²C, numa transferência de dados, marca um período de relógio por bit de dados. Dado isto, só quando a linha SCL estiver com o valor lógico '1' é que é permitido afirmar que há dados em SDA válidos. Quando SCL estiver com o valor lógico '0', os dados em SDA podem ser alterados.

As linhas funcionam numa ligação **wired AND**, pelo que permitem usar as lógicas de **bit recessivo** '1' e **bit dominante** '0' para várias sinalizações. A saída do dispositivo é *wire ANDed* com o sinal do barramento, onde na ausência de bit dominante, a linha respetiva está no nível lógico '1'.

Como dito, o *master* nesta interface é ocupado pelo controlo da linha SCL, pela gestão dos inícios e fins de transferências de dados e pelo controlo do endereçamento dos outros dispositivos. Já o *slave* é o dispositivo que é endereçado pelo *master* e que condiciona o estado da linha SCL. Dado que o *master* pode tomar o significado de transmissor ou de receptor, quando este é transmissor, envia dados para um *slave* receptor, e quando é receptor, recebe dados de um *slave* transmissor.

Num endereçamento proveniente do *master*, o primeiro byte transmitido contém sete bits do endereço mais um que especifica a qualidade da operação (escrita ou leitura). Se esse bit for '0', o *master* toma-se como transmissor e escreve dados na linha SDA, caso contrário é receptor e lê dados da linha SDA.

master

slave

multi-master

arbitragem

sincronização

wired AND

bit recessivo, bit dominante

Cada *slave* lê o endereço da linha SDA. Se o endereço coincidir com o seu próprio endereço, comuta para o estado de transmissor se o bit excepcional for '1', ou comuta para o estado de recetor se o bit excepcional for '0'.

As transações no I²C são delimitadas por **símbolos** (ou condições): condições *start* e *stop*. A condição de **start** existe quando há uma transição do valor lógico '1' para o valor lógico '0' na linha SDA e a linha SCL está com o valor lógico '1', em simultâneo. Por outro lado, a condição de **stop** existe quando há uma transição do valor lógico '0' para o valor lógico '1' na linha SDA e a linha SCL está com o valor lógico '1', em simultâneo. O barramento está no estado **ocupado** após um *start*, até ao próximo *stop* e **livre** após um *stop* e até ao próximo *start*.

Assim, dadas as condições, o *master* envia um *start* e de seguida envia o endereço do *slave* (em 7 bits) juntamente com o bit excepcional que indica a operação. O *slave* endereçado deve efetuar o **acknowledge** (conhecimento) no *slot* seguinte. De seguida, o transmissor (quer seja o *master* ou o *slave*) envia o byte de dados aguardando um **acknowledge** (conhecimento) do recetor, no final. Este ciclo de 9 bits repete-se para cada byte de dados que se pretenda transferir.

A transferência é orientada ao byte, sendo transmitido, em primeiro lugar, o bit mais significativo (MSB). Após o oitavo bit (LSB), o recetor tem de gerar um **acknowledge**, na linha SDA, sob a forma de um bit dominante. Na situação em que o *master* é recetor gera **acknowledge**, sinalizando dessa forma o *slave* de que vai continuar com operações de leitura, ou o *not acknowledge*, sinalizando o *slave* de que o byte recebido constitui o fim da transferência. Neste último caso o *master* envia logo de seguida um *stop*.

O *slave*, no entanto, pode forçar o alargamento da transferência mantendo um bit dominante na linha SCL (fenómeno de **clock stretching**) - sinal *wait*, do *slave*.

Para que haja coordenação na execução dos vários *masters*, há que haver um mecanismo de sincronização e de arbitragem. O mecanismo de sincronização, que coexiste na linha SCL, especifica que na transição de '1' para '0' todos os *master* reiniciam os seus relógios e mantêm o seu relógio a '0' até o tempo a '0' ter chegado ao fim. Quando um *master* termina a contagem do tempo a '0' do seu relógio este deve libertar a linha SCL (permitindo que esta passe a '1'). Caso a linha SCL se mantiver a '0', então deve comutar para um estado *wait*, ficando a aguardar que a linha SCL passe a '1'. Logo que a linha SCL passe a '1', inicia-se a contagem do tempo a '1' do seu relógio. O primeiro *master* a terminar o seu tempo a '1' força a linha SCL a '0'.

Em termos de arbitragem, esta é feita na linha SDA através de transições de bit dominante / bit recessivo. Por cada novo bit enviado para a linha SDA, os *master* leem o que ficou nessa linha, pelo que o processo de arbitragem é perdido quando um *master* lê da linha nível lógico '0' (dominante), tendo escrito nível lógico '1' (recessivo). O *master* que perde o processo de arbitragem retira-se, libertando a SDA (comutando de imediato para modo *slave*), tentando de novo quando o barramento passar ao estado *idle* (livre), este, caracterizado pelas linhas SDA e SCL, ambas, com o nível lógico '1'.

A interface I²C pode ser encontrada, dada a sua simplicidade, versatilidade e economia de recursos, em sensores, DAC's (*digital to analog converter*), ADC's (*analog to digital converter*), memórias externas de micro-controladores, controlo de subsistemas em eletrónica de consumo (como os ajustes dos parâmetros de imagem e som nas televisões, monitores, ...), controlo de subsistemas em terminais de telemóvel, monitorização de hardware (como temperatura de CPU e velocidade da ventoinha em *motherboards* - os MacBook, da Apple® têm um vasto conjunto de I²C para controlo de temperatura e ventoinhas) e interface com *real-time clocks*.

símbolos

start

stop

ocupado

livre

acknowledge

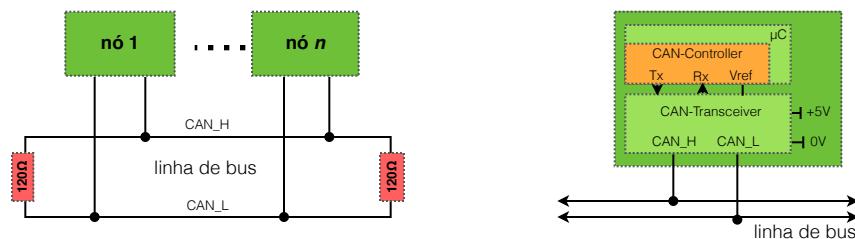
clock stretching

Barramento CAN

Desenvolvido em 1991 pela Bosch®, para simplificar as calagens nos automóveis, o **barramento CAN** (acrónimo de *Controller Area Network*) utiliza comunicação diferencial em par entrancado, com taxas de transmissão até 1 Mbps. Tendo um número máximo de nós no barramento de 40, é adequado a aplicações de segurança crítica, dada a sua elevada robustez, fator que lhe permite ter uma capacidade de detetar diferentes tipos de erros, ter uma tolerância a interferência eletromagnética e ter uma baixa probabilidade de não deteção de um erro de transmissão. Atualmente este barramento é utilizado numa panóplia de aplicações, como na comunicação entre subsistemas de um automóvel, aplicações industriais, domótica, robótica ou equipamentos médicos, entre outros.

O CAN é um barramento *multi-master*, pelo que será necessário um mecanismo de arbitragem, dado que qualquer nó do barramento pode produzir informação e iniciar uma transmissão. As comunicações são estabelecidas de forma bidirecional e *half-duplex*, sendo a informação produzida e encapsulada em tramas, e transmitida em **broadcast** - um transmissor pode enviar informação para todos os nós ao mesmo tempo.

No CAN uma mensagem tem uma identificação (ID) única. Esse ID determina a prioridade da mensagem e, consequentemente, a prioridade no acesso ao barramento. Na Figura 37 podemos ver, à esquerda, a topologia da rede CAN, onde se exibe a comunicação diferencial de par entrancado e, à direita, a estrutura de um nó.



barramento CAN

broadcast

figura 37
comunicação e nó CAN

Na receção, o *transceiver* discrimina o valor lógico pela diferença de tensão entre CAN_H e CAN_L e o resultado é enviado através da linha Rx para o controlador CAN. Por outro lado, na receção, o *transceiver* transforma o nível lógico presente na linha Tx em duas tensões e coloca-as nas linhas CAN_H e CAN_L.

O *CAN-Controller* (controlador CAN), que se representa na Figura 37, à direita, está representado, com mais detalhe, na Figura 38.

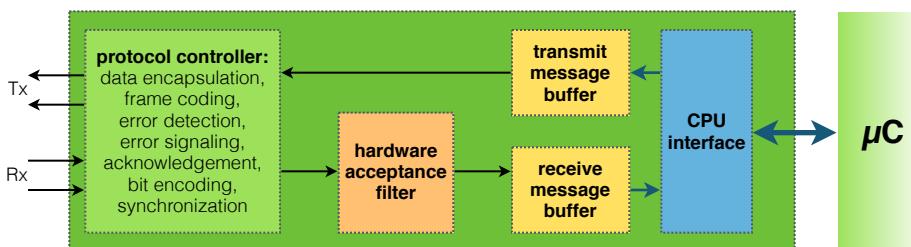


figura 38
CAN-Controller

O controlador CAN implementa em hardware o protocolo. A interface do CPU assegura, tipicamente, a comunicação com o CPU de um micro-controlador (registos de controlo, estado e dados - *buffers*). Já o **hardware acceptance filter** filtra as mensagens recebidas com base no seu ID. Por programação é possível especificar quais os ID's das

hardware acceptance filter

mensagens que serão disponibilizadas ao micro-controlador. Este mecanismo de filtragem, ao descartar mensagens não desejadas, reduz a carga computacional no micro-controlador.

Em suma, a comunicação no bus CAN é feita com um relógio implícito, pelo que a transmissão é orientada ao bit. Tendo um barramento *multi-master*, significa que vários nós trocam mensagens encapsuladas em tramas. Cria-se assim, também, um paradigma produtor-consumidor ou de transmissão em *broadcast*, onde há identificação do conteúdo da mensagem, não existindo identificação do nó de origem ou de destino.

A codificação das mensagens é efetuada através do método **NRZ** (*non-return-to-zero*) e por uma outra técnica, denominada de **bit stuffing**, que basicamente provoca a inserção de um bit de polaridade oposta a cada 5 bits, o que garante tempo máximo entre transições da linha.

Existem diferentes tipos de tramas no barramento CAN. O tipo de trama de dados é, tal como o nome indica, usada no envio de dados de um nó para o(s) consumidor(es) - pode transportar informação acerca de sensores, destes para atuadores. O tipo de trama de erro é usado para reportar um erro detectado (a trama de erro sobrepõe-se a qualquer comunicação, invalidando uma transmissão em curso). Já o tipo de trama **overload** é usado para atrasar o envio da próxima trama (é enviada por um nó em situação de sobrecarga, que não teve tempo para processar a última trama enviada). Por fim, um último tipo é a de *remote transmission request*, tipo o qual é enviado por um nó consumidor a solicitar (ao produtor) a transmissão de uma trama de dados específica.

Em termos de formato da trama, na Figura 39 podemos ver um exemplo genérico.

bus idle	SOF	arbitration field	control field	data field	CRC field	ACK field	EOF	intermission
1 bit	12 bits	6 bits		0 a 8 bytes	16 bits	2 bits	7 bits	3 bits

O campo SOF, quando está a '0' (bit dominante), significa que é o início de uma trama e é usado para a sincronização do relógio dos nós receptores. Já o campo de arbitragem é onde é especificada a arbitragem entre diferentes *master* que podem iniciar a transmissão das suas tramas em simultâneo e onde vem a identificação da mensagem. O campo de controlo especifica o tipo de trama e o número de bytes de dados que transporta. O campo de dados transporta os dados e o campo de CRC serve para a deteção de erros (através do algoritmo de CRC), sendo que o produtor e consumidor calculam a sequência de CRC com base nos bits transmitidos - o produtor transmite a sequência calculada e o consumidor só tem de comparar com a calculada localmente. O campo de *acknowledgement* é onde é realizada a validação da trama e o campo EOF (*end of frame*) é onde se sinaliza o fim da trama (com 7 bits recessivos ('1')).

Device drivers e UART

O número de periféricos existentes é muito vasto. Hoje em dia, a um computador pessoal de secretaria (ou portátil) mantemos constantemente ligados um teclado, um rato, uma placa gráfica, uma placa de som, uma placa de rede, discos rígidos, uma impressora, câmera de vídeo, entre outros... Estes periféricos apresentam todos características distintas entre eles. Entre eles, as operações suportadas são de leitura, escrita ou ambas, e a largura de banda varia entre alguns bytes por segundo a megabytes por segundo. Mas outros detalhes como a representação da informação (que varia entre ASCII, Unicode, Little/Big Endian, ...), o modo de acesso (por caractere, por

NRZ

bit stuffing

overload

figura 39

formato da trama CAN

bloco, ...) e os recursos (portos (I/O, *memory-mapped*), interrupções, DMA, ...) são diferentes. Em termos de implementação também diferentes dispositivos de uma dada classe podem ser baseados em implementações distintas - diferentes fabricantes ou modelos - com reflexos profundos na sua operação interna.

As aplicações e sistemas operativos não podem conhecer todos os tipos de dispositivos passados, atuais e futuros com um nível de detalhe suficiente para realizar o seu controlo a baixo nível. Criou-se assim uma camada de abstração que permitisse o acesso ao dispositivo de forma independente da sua implementação - **device driver**. Um *device driver* é assim um programa que permite a outro programa (aplicação ou sistema operativo) interagir com um dado dispositivo de hardware, implementando a camada de abstração e lidando com as particularidades de cada dispositivo controlado.

O sistema operativo de um dado computador especifica classes de dispositivos e, para cada classe, uma interface que estabelece como é realizado o acesso a esses dispositivos. A função do *device driver* é traduzir as chamadas realizadas pela aplicação ou sistema operativo em ações específicas do dispositivo. Alguns exemplos de classes de dispositivos são os de "interface com o utilizador", "armazenamento em massa", "comunicação", entre outros.

O acesso, por parte das aplicações, a um *device driver* é diferente num sistema embebido e num sistema computacional de uso geral (com um sistema operativo típico, como o Linux, Mac OS X® ou Windows™). As aplicações em sistemas embebidos acedem, tipicamente, de forma direta aos *device drivers*, enquanto que aplicações que correm sobre sistemas operativos comuns acedem a funções do próprio sistema (*system calls*), pelo que o **kernel**, por si, acede aos *device drivers*.

Um caso útil de estudo é a realização de um *device driver* para uma **UART** RS-232. UART é acrónimo de *Universal Assynchronous Receiver Transmitter*. O princípio de operação desta unidade é haver um descolamento de dados entre uma UART e a aplicação realizada por meio de filas (FIFO) - uma de receção e outra para transmissão: a transmissão consiste em copiar os dados a enviar para o FIFO de transmissão do *device driver*; a receção consiste em ler os dados recebidos que residem no FIFO de receção do *device driver*. A transferência de dados entre as filas e a UART é realizada por interrupção (sem intervenção explícita da aplicação).

device driver

kernel
UART

5. Memória

Ao longo da evolução computacional um componente do qual se começou a sentir limitações físicas foi a **memória**. A pretensão de um utilizador comum é poder ter uma memória rápida de grande capacidade mas que custe o menos possível. Não existindo qualquer solução perfeita para este dilema, ao longo do tempo investiu-se mais na organização da memória num sistema computacional, criando um compromisso entre velocidade, capacidade, custo e consumo energético. Assim, uma possível solução seria criar a ilusão de uma memória rápida de grande capacidade através da utilização das várias tecnologias de memória disponíveis, segundo uma hierarquia. Foram assim criadas tecnologias de memória paralelas de forma a poder constituir uma hierarquia. Criou-se assim a SRAM (acrónimo de *Static RAM*), com tempo de acesso entre 0.5 e 2.5 ns e com custo entre \$2000 e \$5000 por gigabyte, a DRAM (acrónimo de *Dynamic RAM*) com tempo de acesso entre 50 e 70 ns e custo entre \$20 e \$75 por gigabyte, e os discos magnéticos com tempos de acesso entre 5 e 20 ms e custo entre \$0.20 e \$2 por gigabyte³. Dadas estas diferenças de custo e de tempo de acesso, é vantajoso construir o sistema de memórias como uma hierarquia

memória

onde se utilizem todas estas tecnologias. Só a DRAM, ao longo dos anos tem vindo a demonstrar um poder de desenvolvimento incrível. Na Figura 40 podemos verificar uma tabela onde se demonstra o desenvolvimento nos parâmetros discutidos anteriormente, sendo que de 1980 para 2000 podemos verificar que houve um aumento de 4000 vezes da capacidade e uma diminuição de 5 vezes o tempo de acesso. Na Figura 41 podemos ver um gráfico que resume a tabela da Figura 40.

ano	capacidade (Kb)	tempo de acesso (ns)	preço por Mb (\$)
1980	64	250	1500
1983	256	185	500
1985	1024	135	200
1989	4096	110	50
1992	16384	90	15
1996	65536	60	10
1998	131072	60	4
2000	262144	55	1
2002	524288	50	0.25
2004	1048576	45	0.10

figura 40
desenvolvimento de
memórias

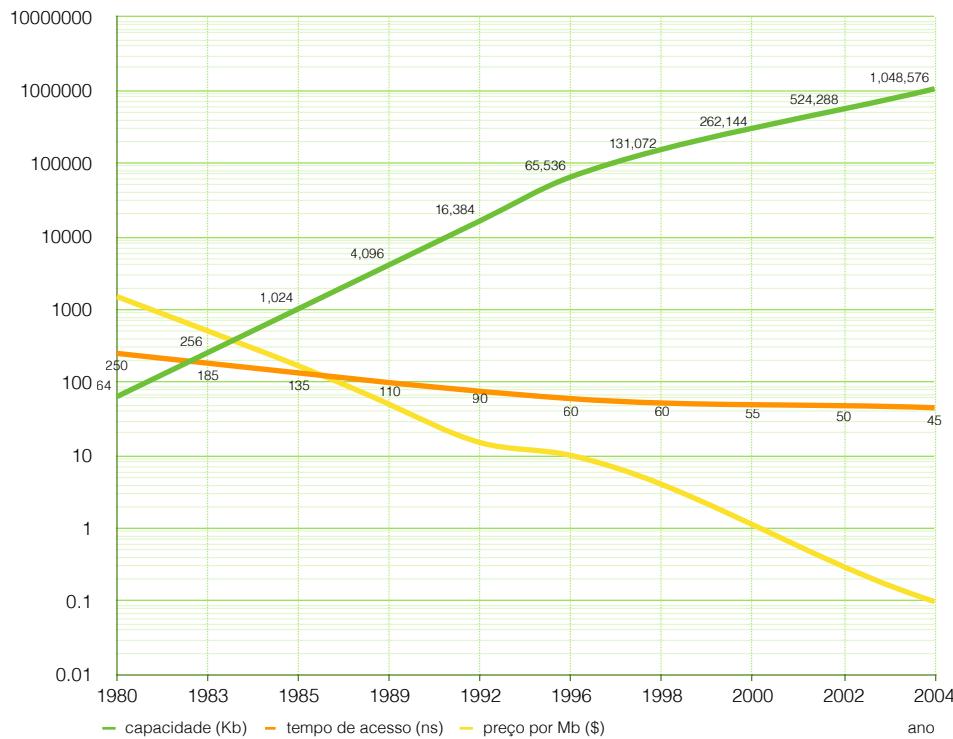


figura 41
desenvolvimento de
memórias

O processador deve ser alimentado de instruções e dados a uma taxa que não comprometa o desempenho do sistema. A diferença entre a velocidade do processador e da memória (DRAM) tem vindo a aumentar. Uma possível solução é guardar a informação mais vezes utilizada numa memória rápida (como a *static RAM*) de pequena dimensão, mais próxima do CPU, acedendo raramente à memória principal

(que por si é mais lenta) para obter os restantes dados (apenas quando necessário) e transferir blocos de informação da memória principal. A esta solução, que vamos aprofundar mais à frente, damos o nome de **cache**.

cache

Hierarquia de memória

Uma possível solução para a utilização da hierarquia de memória é expondo a hierarquia, isto é, utilizando alternativas de armazenamento como os registos internos do CPU, a memória rápida, memória principal e disco, cabendo ao programador usar eficiente e racionalmente estas alternativas de armazenamento. Esta solução é hoje em dia usada em processadores que equipam algumas televisões e a PlayStation® 3 (processador *Cell microprocessor*).

Uma outra possível solução é esconder a hierarquia. Fazer isto é usar um modelo de programação com um tipo de memória único e um espaço de endereçamento também único, pelo que a máquina deve saber gerir automaticamente o acesso à memória. Grande partes dos processadores dos dias de hoje utilizam esta técnica.

Já enunciarmos bastantes vezes o termo "hierarquia de memória" mas ainda não especificamos o seu significado. A **hierarquia de memória** é um conceito da organização da memória num sistema computacional que define níveis de memória. A informação nos níveis superiores é um subconjunto da dos níveis inferiores e a informação circula apenas entre níveis adjacentes. Um **bloco** define-se também como a quantidade de informação que circula entre níveis adjacentes. Na Figura 42 podemos ver a organização da hierarquia de memória com informação acerca da velocidade, tamanho (em bytes) e preço (por byte).

hierarquia de memória

bloco

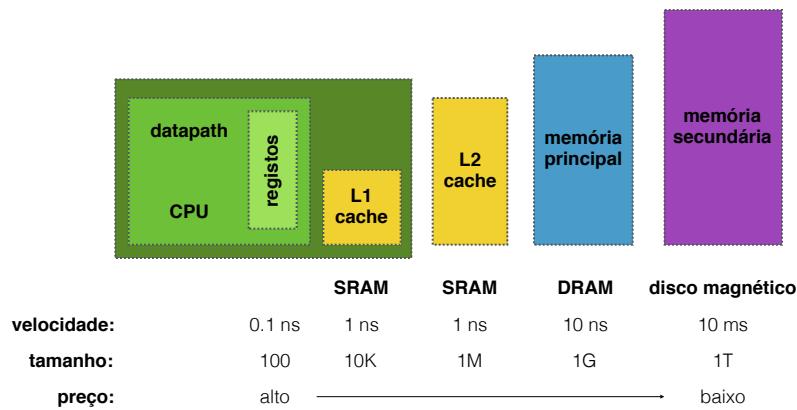


figura 42

hierarquia de memória

Memórias RAM e ROM

Tal como foi visto na disciplina de Laboratórios de Sistemas Digitais (a1s2) existem dois tipos de memória - RAM e ROM. As memórias **RAM** (acrónimo de *Random Access Memory*) são memórias voláteis cuja informação pode ser lida e escrita e cujo acesso é aleatório (o que significa que o tempo de acesso é o mesmo para qualquer posição de memória). As memórias **ROM** (acrónimo de *Read-Only Memory*), por outro lado, são memórias não voláteis que apenas podem ser lidas, com acesso aleatório.

RAM

As memórias ROM não são as únicas com tecnologia não volátil, pelo que há outras que possuem a mesma característica: as memórias ROM são programadas durante o processo de fabrico; as memórias **PROM** (acrónimo de *Programmable*

ROM

PROM

Read-Only Memory) são programáveis uma única vez; as memórias **EPROM** (acrónimo de *Erasable PROM*) podem ser escritas em segundos, sendo que são apagadas em minutos (ambas operações são realizadas em dispositivos especiais); as memórias **EEPROM** (acrónimo de *Electrically Erasable PROM*) têm características muito semelhantes às EPROM, só com a diferença de que as operações de apagar e escrever podem ser efetuadas pelo mesmo circuito (pelo próprio), sendo que o apagamento é feito bit-a-bit e a escrita é muito mais lenta que a leitura; finalmente a memória **flash EEPROM** tem também uma tecnologia muito próxima à EEPROM com a diferença de que a escrita pressupõe o prévio apagamento das zonas de memória a escrever, sendo que este é feito por blocos (por exemplo, blocos de 4kB) o que torna esta tecnologia mais rápida que a EEPROM. Nesta última tecnologia o apagamento e a escrita podem ser efetuados no mesmo equipamento que a memória e a escrita permanece muito mais lenta que a leitura.

Dentro das memórias do tipo RAM temos as que já especificámos anteriormente: SRAM e a DRAM. A memória **SRAM** (*Static Random Access Memory*) é uma memória muito rápida e cuja informação permanece estática até que a alimentação seja cortada. No entanto, esta memória peca por ter seis transístores por célula (implementação típica), ser de baixa densidade (elevada dissipação de potência) e ter um custo por bit elevado. Já as memórias **DRAM** (*Dynamic Random Access Memory*) têm implementações típicas de um transístor mais um condensador por célula, ter alta densidade (baixa dissipação de potência) e um custo por bit baixo. No entanto, estas memórias DRAM contêm informação que permanece apenas durante alguns milissegundos (necessita de *refresh* regular - daí a designação de *dynamic*) e é mais lenta (pelo menos uma ordem de grandeza) que a SRAM.

Organização de memória (SRAM)

Uma memória pode ser encarada como uma coleção de m registos de dimensão n ($m \times n$). Cada registo é formado por n células, cada uma delas capaz de armazenar um bit de dados. Através de um agrupamento de células-base pode formar-se uma memória de maior dimensão. Apesar de necessário especificar o tamanho das palavras ($x_1, x_4, x_8, x_{16}, 32, \dots$) e o número total de palavras que a memória pode armazenar ($tamanho_{palavra} \times numero_{palavras}$), podemos criar uma memória, por exemplo 4×4 , como se pode ver na Figura 44.

As células-base podem seguir uma implementação como a da Figura 43.

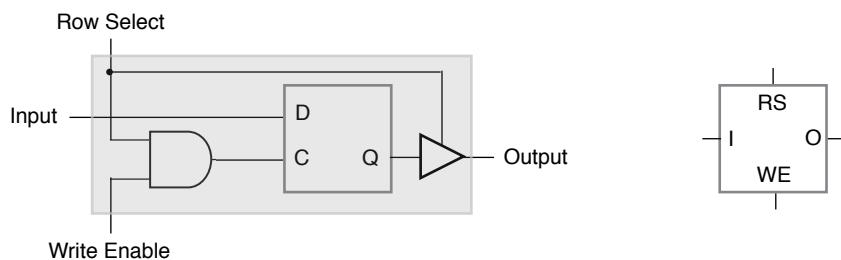


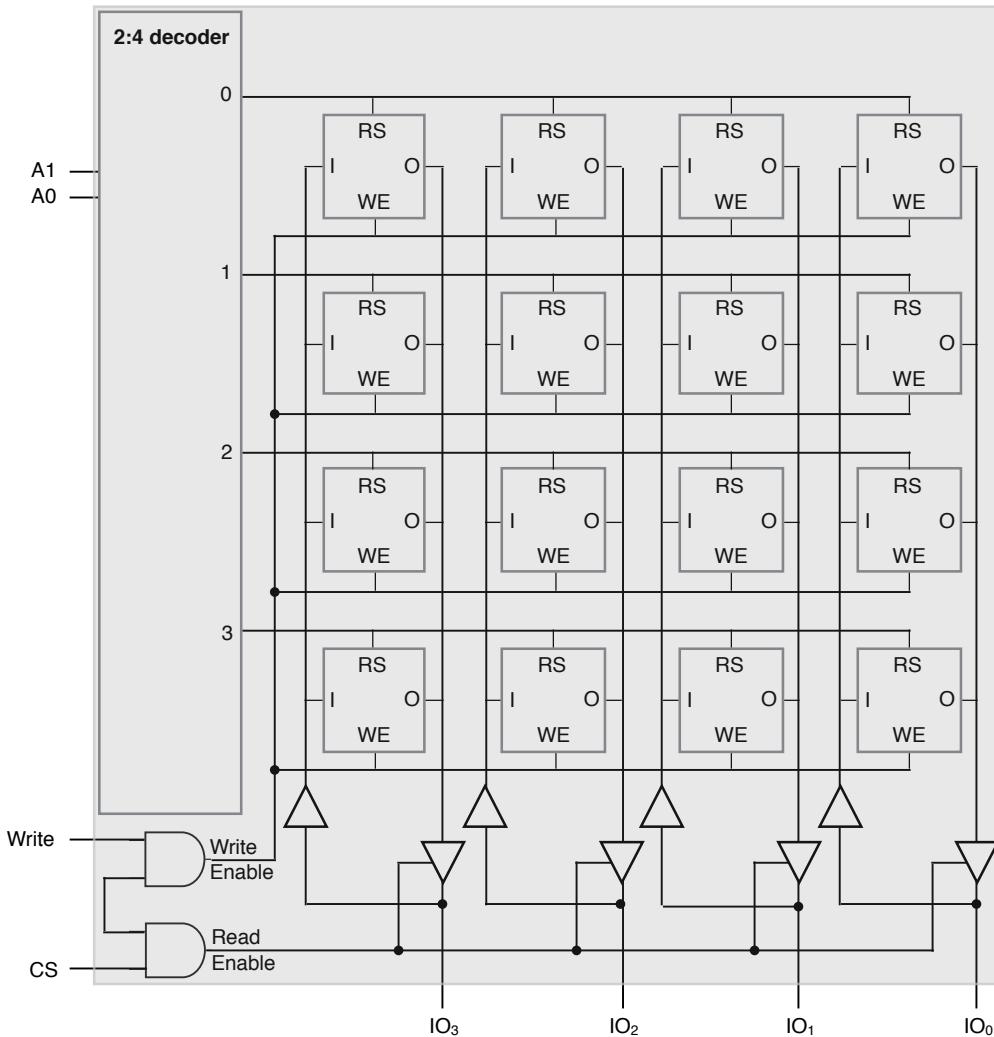
figura 43
célula-base de SRAM

Também é possível criar uma **organização em matriz**, sendo que nesse caso cada linha e cada coluna deve ter um descodificador (permite a escolha por passo).

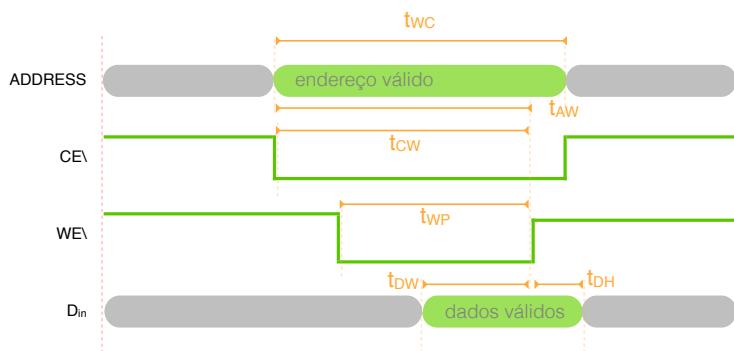
Uma RAM estática (SRAM), como já foi referido anteriormente, tem seis transístores por célula. Numa operação de escrita coloca-se o bit na entrada de dados do bloco e ativa-se a seleção. Numa operação de leitura ativa-se a seleção e o dado sai.

organização em matriz

45 ARQUITETURA DE COMPUTADORES II



Na Figura 45 podemos ver um diagrama temporal que reflete o ciclo de escrita e na Figura 46 o ciclo de leitura, com os respectivos parâmetros temporais.



Na Figura 47 podemos ver uma tabela que contém os valores indicativos (em nanosegundos) dos parâmetros associados a um ciclos de escrita de uma memória SRAM. Na Figura 48, podemos ver uma tabela que contém os valores indicativos (em nanosegundos) dos parâmetros associados a um ciclo de leitura de uma memória SRAM.

figura 44
possível SRAM

figura 45
diagrama temporal de
escrita

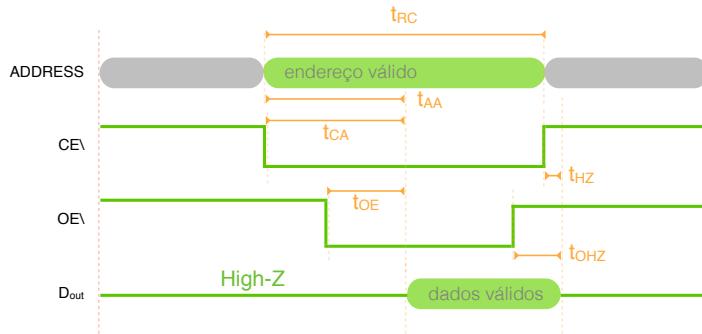


figura 46
diagrama temporal de
leitura

parâmetro	símbolo	mínimo	máximo
Write Cycle Time	t_{RC}	15	
Address Valid to End of Write	t_{AW}	10	
CE\ to End of Write	t_{CW}	10	
Write Pulse Width	t_{WP}	10	
Data Valid to End of Write	t_{DW}	7	
Data Hold Time	t_{DH}	0	

figura 47
parâmetros temporais de
escrita

parâmetro	símbolo	mínimo	máximo
Read Cycle Time	t_{RC}	15	
Address Access Time	t_{AA}		15
CE\ Access Time	t_{CA}		15
Output Enable to Output Valid	t_{OE}		7
CE\ to Output in High-Z	t_{HZ}		6
OE\ to Output in High-Z	t_{OHZ}		6

figura 48
parâmetros temporais de
leitura

É frequente ter-se necessidade de memórias com uma capacidade de armazenamento superior à capacidade individual dos circuitos disponíveis comercialmente. Nessa situação recorre-se à construção de módulos de memória que resultam do agrupamento de circuitos de acordo com o aumento pretendido. Assim, a construção de um módulo de memória pode envolver as duas fases seguintes, ou apenas uma delas, em função dos circuitos disponíveis e dos requisitos finais de armazenamento: aumento do comprimento de palavra ou aumento do número total de posições de memória.

Organização de memória (DRAM)

Como já referimos atrás, a célula de uma memória DRAM é constituída de um transístor e um condensador. O condensador é de uma capacidade muito pequena, da ordem dos femto-Farads (fF) - 1×10^{-15} F. A operação de leitura é destrutiva (descarrega o condensador). Na ausência de leitura, o condensador é lentamente descarregado, pelo que a informação permanece na célula durante apenas alguns milissegundos. Assim, para manter a informação é necessário ter um mecanismo de *refresh* periódico da carga do condensador. Na Figura 49 podemos ver uma representação de uma célula.

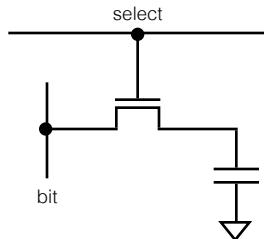


figura 49
célula de memória

Então, de forma rápida, para fazer uma escrita numa célula da Figura 49 temos de colocar o dado a escrever na linha *bit* e ativar a linha *select*. Para fazer uma leitura devemos pré-carregar a linha *bit* a VDD/2 e só depois ativar a linha *select* (o valor lógico é detetado pela diferença de tensão na linha *bit*) e fazer um restauro do valor da tensão no condensador (escrever o valor de volta). Para fazer uma atualização da célula, devemos repetir a operação de leitura.

As memórias DRAM estão organizadas em matriz, pelo que os endereços de linha e coluna são multiplicados no tempo. Esta multiplexagem obriga à utilização de dois sinais adicionais (multiplexagem com dois *strokes* independentes): **RAS** - *row address strobe*; **CAS** - *column address strobe*. A linha CAS também serve de *chip-select* e ambas RAS e CAS são sensíveis às transições.

Em termos de diagrama lógico temos um bloco lógico como o representado na Figura 50.

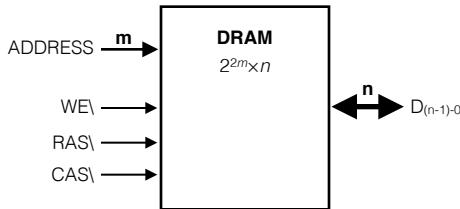


figura 50
bloco DRAM

A dimensão da DRAM é de $2^{2m} \times n$ se a matriz interna for quadrada. Em termos de funcionamento, $WE\backslash$ é um sinal de *write enable* (ativo-baixo), pelo que se estiver a '0' significa que a DRAM está programada para receber um valor e escrever em memória, e se estiver a '1' está programada para ler o valor. O local onde lê ou escreve é definido através da entrada *ADDRESS*, de m bits. Para validar o endereço na linha e na coluna na transição descendente temos os sinais de *RAS\backslash* e *CAS\backslash*, respetivamente. Na Figura 51 podemos ver o diagrama de blocos interno do bloco.

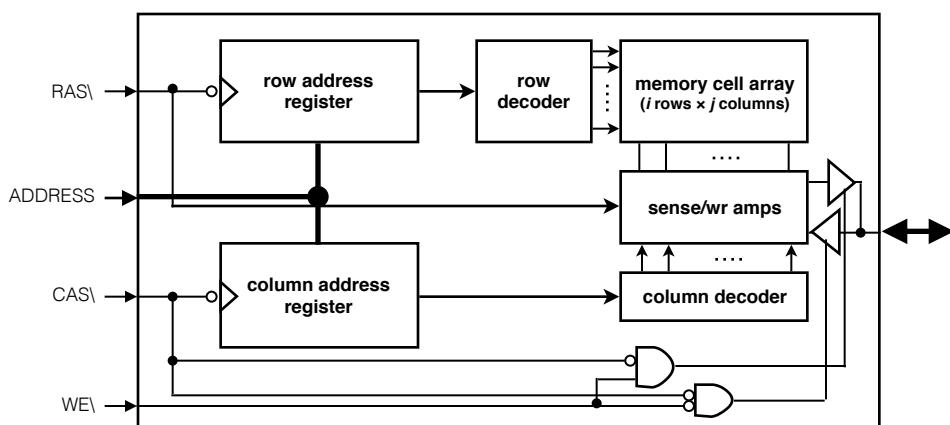


figura 51
arquitetura DRAM

Em termos de diagramas temporais, na Figura 52 temos um diagrama temporal típico de um ciclo de leitura de uma memória DRAM.

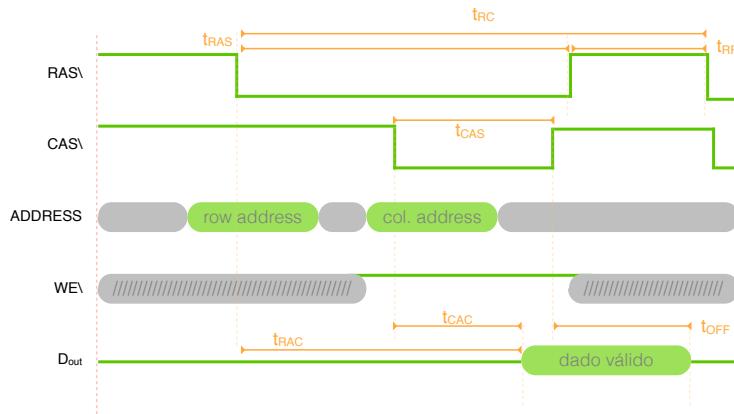


figura 52
diagrama temporal de um
ciclo de leitura

Na Figura 53 temos um diagrama temporal de um ciclo de escrita (*early write*) de uma memória DRAM.

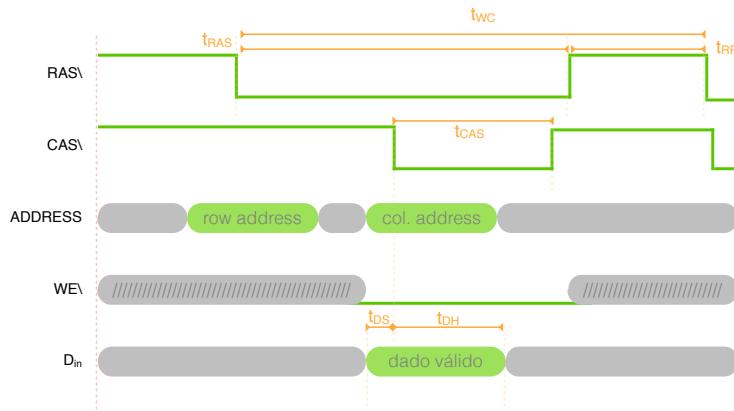


figura 53
diagrama temporal de um
ciclo de escrita

Na Figura 54 podemos ver os valores indicativos (em nanosegundos) dos parâmetros indicados nos diagramas temporais de leitura e escrita de uma memória DRAM com um tempo de acesso de 55 nanosegundos.

parâmetro	símbolo	mínimo	máximo
Read or Write Cycle Time	t_{RC}	100	
RAS\ Precharge Time	t_{RP}	45	
Page Mode Cycle Time	t_{PC}	35	
RAS\ Pulse Width	t_{RAS}	55	10000
CAS\ Pulse Width	t_{CAS}	28	10000
Data-in Setup Time	t_{DS}	5	
Data-in Hold Time	t_{DH}	14	
Output buffer turn-off delay	t_{OFF}		15
Access Time from RAS\	t_{RAC}		55
Access Time from CAS\	t_{CAC}		28

figura 54
parâmetros temporais

Tal como as memórias SRAM, nas DRAM também podemos gerar um aumento do comprimento das palavras, sendo que acrescentando p módulos, se o array de células de memória for quadrado, então o tamanho da RAM é $2^m \times (n \times p)$; ou um aumento do número total de posições de memória que, nas mesmas condições, teria um tamanho de $2^{m+k} \times n$ (por mais 2^k módulos).

Deteção e correção de erros nas memórias

Na utilização de memórias estamos sujeitos, por vezes, a efeitos ruidosos causados por **erros**. Estes erros, no contexto das memórias, podem ser de dois tipos: **hard failure** (erros que se caracterizam por causarem defeitos permanentes, pelo que as células de memória foram afetadas ou não podem armazenar informação de forma fiável (*stuck-at-zero* ou *stuck-at-one* ou simplesmente mudam erraticamente de estado)), o que pode ser causado por condições de utilização fora do especificado, defeitos de fabrico ou desgaste; e **soft error** (pequenos episódios de erros, não destrutivos, que alteram o conteúdo de uma ou mais células de memória, podendo ser causados por problemas na fonte de alimentação ou por partículas alfa).

Quando estamos perante a análise de deteção de erros de 1 bit, a melhor forma de o fazer é usando um **bit de paridade**. Já para a deteção e correção há que usar algoritmos e códigos mais abrangentes, como o **código de Hamming** (ECC - *Error Correcting Code*).

O código de Hamming aplica-se da seguinte forma: consideremos que os nossos bits de dados (de informação) são os seguintes: 1011. Para criarmos o nosso código de Hamming necessitamos de gerar bits de paridade nos bits de posições 2^n , onde n é um número inteiro. Assim, geramos um código como 101?1???, onde "?" pretende designar cada bit de paridade acrescentado. Dando nome a cada bit de paridade (P_n) temos um código 101P₄1P₂P₁. Mas agora como é que podemos calcular o valor dos bits de paridade? P₁ é igual à operação "ou exclusivo" (XOR) de si próprio (?) e dos bits de maior índice, de uma em uma casa binária (uma sim, uma não), isto é, $P_1 = ? \oplus 1 \oplus \dots \oplus 1$. Já P₂ é igual à mesma operação de si próprio (?) e dos bits de maior índice, de duas em duas posições (duas sim, duas não). Assim, $P_2 = ? \oplus 1 \oplus 0 \oplus 1$. Por fim, P₄ é igual à mesma operação de si próprio (?) e dos bits de maior índice, de quatro em quatro posições (quatro sim, quatro não). Assim, $P_4 = ? \oplus 1 \oplus 0 \oplus 1$. Avaliando a paridade de cada P_n , temos que P₁ é ímpar (logo 1), P₂ é par (logo 0) e P₄ é par (logo 0). Assim, substituindo cada P_n no código inicial 101P₄1P₂P₁, temos que o nosso código de Hamming é 1010101.

Agora imaginemos que o código foi transmitido, mas que houve um erro, e, sem saber, recebemos o código 1110101. Como podemos reparar, na receção houve a troca de um bit. Neste caso, o que temos de fazer, é verificar o valor da paridade para cada bit de paridade e confirmar. Começando por P₁, temos que este é 1 - mas será que isso se confirma com o resto do código. Verificando a paridade (1110101) vemos que este é ímpar, logo é 1, o que se confirma (não há erros com este bit de paridade). Passando para P₂, temos que o bit de paridade é 0. Se verificarmos a paridade para este bit (1110101) vemos que este é ímpar (logo 1), contrariamente ao que o bit de paridade indica - há erro! Passamos para o bit seguinte e temos P₄ com o valor 0. Verificando (1110101) temos mais um erro, dado que $1 \neq 0$.

Para detetarmos e corrigirmos o erro na sequência anterior, basta converter para decimal a quantidade formada por justaposição dos bits de paridade e obtemos a posição que tem erro. Assim, P₁P₂P₄ é igual a $100_2 = 6_{10}$. A posição 6 no nosso código tem um erro (1110101), que nós conseguimos verificar comparando com a sequência original (1010101).

erros

hard failure

soft error

bit de paridade

código de Hamming

• **R. W. Hamming**

Memória cache

A hierarquia de memória combina uma memória adaptada à velocidade do processador (de pequena dimensão) com uma (ou mais), menos rápida, mas de maior dimensão. Uma vez que a memória com que o processador interage diretamente é de pequena dimensão, a eficiência da hierarquia resulta do facto de se mover informação para a memória rápida poucas vezes (antes de a substituir). Para se tirar partido deste esquema, a probabilidade de que a informação necessária esteja no nível mais elevado da hierarquia tem de ser elevada. Mas o que é que torna essa probabilidade elevada?

Um programa não acede, tipicamente, a todo o seu código (ou dados) ao mesmo tempo com igual probabilidade. Assim, um programa acede a uma zona reduzida do espaço de endereçamento numa dada fração do tempo (princípio da localidade). Há assim, dois tipos de **localidade**: localidade no espaço (*spatial locality*), na qual se existe um acesso a uma zona de memória, então é provável que as zonas contíguas sejam também acedidas, e a localidade no tempo (*temporal locality*), na qual se existe um acesso a uma zona de memória, então é provável que essa mesma zona seja acedida novamente num futuro próximo.

localidade

Para uma localidade espacial, a informação que o processador necessita de seguida, tem um elevada probabilidade de estar próxima da que consome agora, como as instruções de um programa, o processamento de um array, pelo que quanto maior for o bloco de informação existente na memória rápida, melhor. Já para uma localidade temporal, a informação que o processador consome agora tem uma elevada probabilidade de ser novamente necessária num curto espaço de tempo, como a variável de controlo de um ciclo ou as instruções de um ciclo, pelo que quantos mais blocos de informação estiverem na memória rápida, melhor.

Na hierarquia de memória definida por dois níveis temos então um nível superior, primário e rápido, de pequena dimensão e um nível inferior, secundário e mais lento, de maior dimensão, sendo que no nível superior ficam os blocos de memória mais recentemente utilizados.

Os pedidos de informação são sempre dirigidos ao nível primário, sendo o nível secundário envolvido apenas quando a informação pretendida não está nesse nível. Se os dados pretendidos se encontram num bloco de nível primário então existe um **hit**, caso contrário existe um **miss**. Na ocorrência de um *miss* é efetuado o acesso ao nível secundário para obter os dados, transferindo o bloco que contém a informação pretendida.

hit
miss

A **taxa de sucesso** (*hit ratio*) é dada pelo número de *hits* sobre o número total de acessos. Por conseguinte, a **taxa de insucesso** (*miss ratio*) é dada pela diferença de 1 com a taxa de sucesso. O **tempo de acesso** no caso de um *hit* designa-se por **hit time** e o tempo de substituir um bloco de nível superior e enviar os dados para o processador é designado de **miss penalty**. O **tempo médio de acesso** à informação é assim o resultado $t_a = hit\ ratio \times hit\ time + miss\ ratio \times miss\ penalty$.

taxa de sucesso
taxa de insucesso
tempo de acesso, hit time
miss penalty, tempo médio de acesso

A **cache** surge assim como um nível de memória que se encontra entre o CPU e a memória principal (as primeiras utilizações foram na década de '60). É comum os computadores recentes incluírem um ou mais níveis de cache. A cache é muitas vezes integrada no mesmo chip do processador.

Numa leitura da cache, esta recebe um endereço (*MAddr*) do CPU: se o bloco que contém o endereço estiver na cache, é enviado o conteúdo de *MAddr* diretamente para o CPU; caso contrário, há que encontrar espaço na cache para um novo bloco, acede-se à memória principal e lê-se o bloco que contém o endereço *MAddr*, lendo de seguida a cache e enviando para o CPU o conteúdo de *MAddr*.

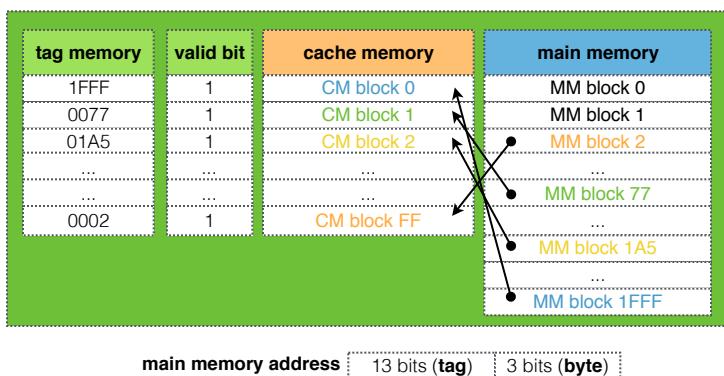
As operações da cache são transparentes para o programa. O programa emite um endereço e pede que seja feita uma operação de leitura ou de escrita. Esse pedido, uma vez feito, é satisfeito pelo sistema de memória, pelo que é desconhecido para o programa se é a cache ou a memória principal que está a satisfazer o pedido.

Na implementação da unidade de controlo da cache é necessário ter em atenção vários aspectos, como "como saber se um determinado está na cache?" (se sim, onde?), "onde colocar um novo bloco na cache?", entre outros.

Há basicamente três formas de organizar os sistemas de cache: ter uma cache totalmente associativa (*fully associative*), ter uma cache com mapeamento direto (*direct mapped*) ou ter uma cache parcialmente associativa (*set associative*). A partir deste ponto iremos abordar cada um dos três tipos, tendo em consideração um espaço de endereçamento de 16 bits (com memória principal de 64 kB) e uma memória cache com 256 posições de 8 bytes (2^3) cada, isto é, cada bloco tem uma dimensão de 8 bytes.

Tendo as nossas especificações-base como um espaço de endereçamento de 16 bits e uma memória cache com 256 posições de 8 bytes, com estes valores, a memória principal pode ser vista como sendo constituída de um conjunto de 2^{13} blocos contíguos, de 8 bytes cada ($2^{16}/2^3 = 2^{13}$): 8000 blocos (numerados de 0x0000 a 0x1FFF). Assim, no endereço de 16 bits, os 13 bits mais significativos identificam o bloco (**tag**) e os 3 bits menos significativos o **byte** dentro desse bloco.

Numa cache **totalmente associativa** seguimos um modelo semelhante ao apresentado na Figura 55.



tag, byte

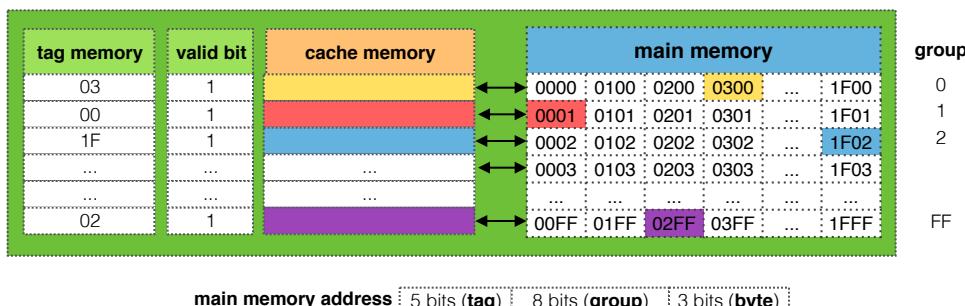
totalmente associativa

figura 55

cache totalmente associativa

Por vantagens deste modelo temos que qualquer bloco da memória principal pode ser colocado em qualquer posição da cache, no entanto, por inconvenientes temos que todas as entradas da memória *tag* têm de ser analisadas, de forma a verificar se um endereço se encontra na cache, e que há muitos comparadores, pelo que o custo de tal equipamento é elevado.

Numa cache com **mapeamento direto** segue-se um modelo muito estrito ao representado na Figura 56.



mapeamento direto

figura 56

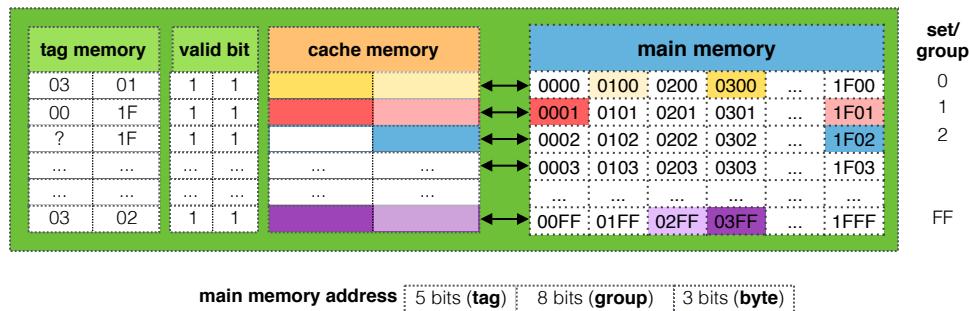
cache com mapeamento direto

Neste tipo de cache o endereço do bloco é obtido, a partir do endereço real, pelo cálculo $\text{endereço do bloco} = \text{endereço real} / \text{dimensão do bloco}$, sendo que o *endereço do bloco* é igual à concatenação da *tag* com o grupo. A posição da cache, a linha, associada a um dado endereço é dada por $\text{pos} = \text{endereço do bloco \% número de blocos da cache}$, sendo que pos é igual ao grupo. O tamanho do bloco e o número de blocos da cache são potências de 2.

Num mapeamento direto, então, a cada bloco da memória principal é associada uma linha da cache, sendo que o mesmo bloco é sempre colocado na mesma linha. Vários bloco têm associada a mesma linha. Uma grande vantagem deste método é que há uma maior simplicidade de implementação, embora, por outro lado, uma grande desvantagem é o facto de num dado instante, apenas um bloco de um dado grupo poder residir na cache, o que tem como consequência que alguns blocos podem ser substituídos e recarregados várias vezes, mesmo existindo espaço na cache para guardar todos os blocos em uso.

Uma melhoria deste último tipo de cache seria permitir o armazenamento simultâneo de mais que um bloco do mesmo grupo. A esta melhoria damos o nome da implementação da cache **parcialmente associativa**. Na Figura 57 podemos ver um exemplo do modelo conceptual.

parcialmente associativa



Em suma, neste esquema de cache parcialmente associativa é semelhante à cache com mapeamento direto, mas a primeira permite que mais que um bloco de um mesmo grupo possa estar na cache. Uma cache com associatividade de 2 permite que dois blocos do mesmo grupo possam estar simultaneamente na cache.

A divisão do endereço de memória é igual ao do mapeamento direto (o campo *group* é normalmente designado por *set* - conjunto), mas agora há n possíveis lugares onde um dado bloco de um mesmo grupo pode residir (para uma cache com associatividade de n). Os n possíveis lugares onde o bloco pode residir têm que ser procurados simultaneamente.

Existem várias **estratégias de sobreposição** de blocos na cache, na ocorrência de um *miss*. Elas são várias, mas eis algumas das mais utilizadas:

estratégias de sobreposição

- **LRU** (*Least Recently Used*) - é substituído o bloco da cache que está há mais tempo sem ser referenciado;
 - **LFU** (*Least Frequently Used*) - é substituído o bloco menos acedido;
 - **FIFO** (*First-In First-Out*) - é substituído o bloco que foi carregado há mais tempo;
 - **random** - substituição aleatória (testes indicam que não é muito diferente que LRU).
- | | |
|-------------|---------------|
| LRU | LFU |
| FIFO | random |

Em termos de **políticas de escrita** existem duas: seguindo a política de **write-through** todas as escritas são realizadas simultaneamente na cache e na memória principal, sendo que esta está sempre consistente, e caso um dado esteja ausente, atualiza-se apenas a memória principal (técnica de **write-no-allocate**); se seguirmos antes a política de **write-back**, então o valor é escrito apenas na cache, sendo que o novo valor é escrito na memória quando o bloco da cache é substituído. Nesta última é lançada uma *flag dirty bit*, ativada quando houver uma escrita em qualquer endereço do bloco presente na linha da cache. Esta política é muito mais complexa que a anterior, sendo que nesta, se um determinado dado estiver ausente na cache, é carregado o bloco para a cache e atualizando-o (técnica de **write-allocate**).

políticas de escrita
write-through

write-no-allocate
write-back

write-allocate

Memória externa

Por fim, o único conjunto de memórias que não estudámos foram as **memórias externas**. As memórias externas são, tal como o nome indica, as mais externas da hierarquia de memória, como podemos ver na Figura 58.

memórias externas

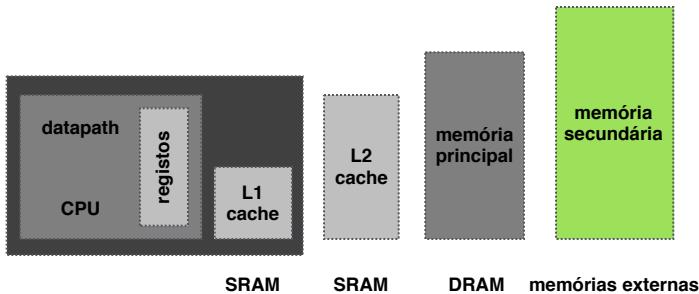


figura 58
memórias externas na
hierarquia

Como memórias externas podemos ter discos magnéticos ou *solid-state disks*. Começando pelos **discos magnéticos** (HDD) estes têm princípios puramente mecânicos. Durante muitos anos, a tecnologia do **magnetismo** dominou a área do armazenamento em massa. O exemplo mais comum, que ainda hoje se usa, são os discos magnéticos, nos quais existe um disco de superfície magnética que gira e guarda informação. Cabeças de escrita e leitura estão posicionadas por cima e/ou por baixo do disco, para quando este girar, cada cabeça faça um círculo, este, chamado de **faixa** (*track*). Ao reposicionar as cabeças de escrita/leitura diferentes faixas concêntricas são passíveis de serem acedidas. Em muitos casos, sistemas de armazenamento em disco consistem em vários discos montados num mesmo eixo, uns por cima dos outros, com determinados espaços entre eles, de modo a que caibam as cabeças de escrita/leitura, entre as superfícies. Em certos casos as cabeças movem-se em uníssono. Cada vez que estas são repositionadas, um novo conjunto de faixas - chamado de **cilindro** - torna-se acessível. Todos estes detalhes podem ser vistos na Figura 59.

discos magnéticos
magnetismo

faixa

cilindro

Sendo que uma faixa pode conter muito mais informação do que a que nós pretendemos manipular em simultâneo, cada faixa é dividida em pequenos arcos chamados **setores** onde a informação é gravada numa contínua sequência de bits. Todos os setores num disco contêm o mesmo número de bits (as capacidades típicas são entre 512 bytes a uns poucos kilobytes), e no mais simples disco, todas as faixas apresentam o mesmo número de setores. Por conseguinte, nesse disco, as informações gravadas na faixa mais exterior estão menos compactadas do que na mais interior, sendo que as primeiras são maiores que as últimas. Em discos de alta capacidade de armazenamento, as faixas mais afastadas do centro são as com maior probabilidade de

setores

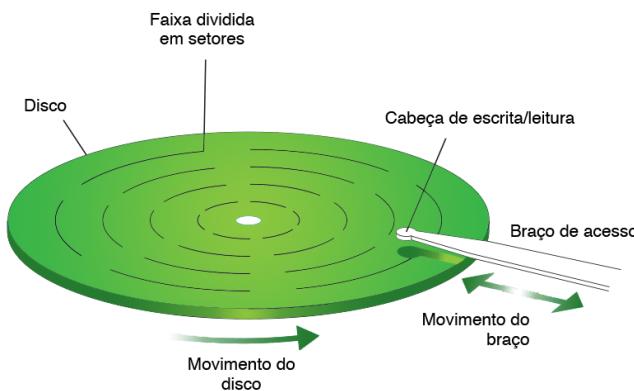


figura 59
disco magnético

ter mais setores que as interiores, capacidade adquirida pelo uso da técnica de **gravação de bit localizada** (*zoned-bit recording*). Usando esta técnica, várias faixas adjacentes são coletivamente conhecidas como zonas, sendo que cada disco contém cerca de dez zonas. Aqui, todas as faixas de uma dada zona têm o mesmo número de setores e cada zona tem mais setores por zona que as suas inferiores. Desta forma consegue-se maior rendimento no que toca ao aproveitamento de todo o espaço do disco. Independentemente dos detalhes, um sistema de armazenamento em disco consiste em vários setores individuais, cujos podem ser acedidos como sequências independentes de bits.

Diversas medidas são usadas para avaliar o desempenho de um disco:

- **Tempo de procura** (*seek time*): o tempo necessário para mover uma cabeça de escrita/leitura de uma faixa para outra;
- **Atraso de rotação ou tempo latente** (*rotation delay*): metade do tempo necessário para que o disco faça uma rotação completa, a qual é a média do tempo total para que uns dados desejados girem até às cabeças de escrita/leitura, desde o momento em que estas já se localizam por cima da faixa pretendida;
- **Tempo de acesso** (*access time*): a soma do tempo de procura e do atraso de rotação;
- **Taxa de transferência** (*transfer rate*): a taxa de velocidade a qual a informação pode ser transportada para ou do disco.

nota!! que no caso da gravação de bit localizada, o total de dados que passam as cabeças de escrita/leitura numa única rotação do disco é maior nas faixas exteriores que nas faixas interiores, por conseguinte a taxa de transferência também diferirá consoante a faixa em questão.

Um fator que limita o tempo de acesso e a taxa de transferência é a velocidade a que o disco roda. De modo a facilitar essa velocidade as cabeças de escrita/leitura não tocam nunca no disco, mas antes “flutuam” sobre a sua superfície. O espaço entre a cabeça e o disco é tão pequeno que um simples grão de pó pode travar o sistema, destruindo ambos - um fenómeno chamado de **head crash**. Tipicamente, estes discos vêm, de fábrica, selados, dadas essas possibilidades. Só assim é que os discos podem girar a velocidades de várias centenas por segundo, alcançando taxas de transferência que são medidas em Mb por segundo.

Sendo que os sistemas de armazenamento requerem movimento para as suas operações, estes ficam a perder quando comparados a circuitos eletrónicos. Tempos de

gravação de bit localizada

tempo de procura

atraso de rotação, tempo latente

tempo de acesso

taxa de transferência

nota

head crash

atraso, num circuito eletrónico, são medidos em nanossegundos ou menos, enquanto que tempos de procura, de latência e de acesso são medidos em milissegundos. Assim, o tempo necessário para receber algum dado de um sistema de armazenamento em disco pode parecer um “eternidade”, em comparação a um circuito eletrónico que aguarda uma resposta.

Se nós ligarmos múltiplas unidades de disco de forma a que essas possam ser vistas pelo sistema operativo como uma única unidade lógica, a isso, damos o nome de **RAID** (*Redundante Array of Independent Disks*). Nesta organização os dados são distribuídos pelas várias unidades físicas num esquema designado por **data stripping**. A capacidade redundante é usada para armazenar informação de paridade, para garantir recuperação dos dados no caso de uma unidade de disco avariar. Isto gera uma maior capacidade de transferência de dados, um maior número de operações de I/O processadas por unidade de tempo e uma maior fiabilidade devida a redundância.

Um outro tipo de memória externa é a **solid-state**. Este nome provém do facto destes serem criados com base em eletrónica construída com **semi-condutores**. Existem dois tipos de memória *solid-state* (memória **flash**): as memórias flash **NOR** têm como unidade básica o bit, fornecendo acesso aleatório a alta-velocidade e são usadas para preservar o código do sistema operativo de telemóveis e na BIOS, sobre a qual arranca a execução do WindowsTM; as memórias flash **NAND** têm como unidade básica 16 ou 32 bits, leem e escrevem em pequenos blocos e são usadas nos dispositivos de memória com conexão via USB, cartões de memória e SSD's, sendo que não fornece acesso aleatório, pelo que para a leitura só é possível analisar bloco-a-bloco.

Os SSD's são muito melhores que os anteriores e ainda usados HDD's dado que permitem maior número de operações I/O por segundo, tem uma maior robustez (é menos suscetível a choques e vibrações), tem um menor consumo de energia, são mais pequenos e menos ruidosos e são mais de uma ordem de grandeza mais rápida. No entanto, o desempenho das SSD têm tendência a piorar à medida que o dispositivo é usado, pelo que o bloco completo tem de ser lido da memória flash e colocado na RAM do *buffer*. Antes do bloco alterado poder ser escrito na memória flash, o bloco completo tem de ser apagado e o conteúdo completo do RAM *buffer* pode, assim, ser escrito na memória flash. Também depois de um certo número de operações de escrita a memória flash deixa de se poder utilizar. Para evitar que isto aconteça, pode-se incluir cache para diminuir o número de operações de escrita, usar algoritmos *wear-leveling* que tentam distribuir os *writes* uniformemente pelos blocos de células e técnicas para gerir os blocos danificados (análogo ao HDD). A maioria dos dispositivos flash estimam o tempo de vida útil que lhes resta, permitindo aos sistemas antecipar avarias e tomar medidas preventivas.

E assim terminamos o estudo da disciplina de Arquitetura de Computadores II (a2s2), programa o qual pode ser seguido por várias disciplinas como Sistemas de Operação (a3s1).

RAID

data stripping

solid-state

semi-condutores

flash, NOR

NAND

Apontamentos de Arquitetura de Computadores II

1ª edição - junho de 2015

ac2

Autor: Rui Lopes

Fontes bibliográficas: Computer Organization and Design: The Hardware/Software Interface, Patterson, David A., Hennessy, John L., Fourth Edition (2012); Computer Architecture: A Quantitative Approach, John L. Hennessy, David A. Patterson, Third Edition; Computer Science, An overview, BROOKSHEAR, James, Addison-Wesley, 2008.

Outros recursos: Notas das aulas de Arquitetura de Computadores II e Slides de Arquitetura de Computadores II (2012-2013).

Agradecimentos: professores António de Brito Ferrari e António Cruz.

Todas as ilustrações gráficas são obra de Rui Lopes e as imagens são provenientes das fontes bibliográficas divulgadas.



apontamentos

© Rui Lopes 2015 Copyright: Pela Creative Commons, não é permitida a cópia e a venda deste documento. Qualquer fraude será punida. Respeite os autores e as suas marcas. Original - This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/deed.en_US.