Resumo AC2 2022

**Introdução**

**Microprocessador**: Um cpu basicamente

• Circuito integrado com um (ou mais) CPU (cores)

• Não tem memória interna (além do banco de registos)

• Os barramentos estão disponíveis no exterior

• Para obter um sistema completo é necessário acrescentar RAM, ROM e periféricos

• Pode operar a frequências elevadas (> 3GHz)

• Sistemas computacionais de uso geral

**Microcontrolador**:

• Circuito integrado inclui CPU, RAM, ROM, periféricos

• Frequência de funcionamento normalmente baixa (< 200 MHz)

• Baixo consumo de energia

• Disponibiliza uma grande variedade de periféricos e interfaces com o exterior

• Rapidez de resposta a eventos externos (Sistemas de Tempo Real)

• Utilizado em tarefas específicas (por exemplo controlo da velocidade de um motor)

**Sistema embebido:** Implementado com base num microcontrolador, pode fazer parte de um sistema mais complexo.

Microcontrolador tem 3 componentes fundamentais: cpu, memoria (ram e disco), e portos i/o.

Inclui periféricos assim como timers, adc, serial I/O (usb, i2c, spi …) e **tudo isto é interligado através de barramentos**.

**Desenvolver programa para microcontrolador**

**Edição** em C ou em assembly do microcontrolador.

**Geração do código** utilizando um **cross-compiler** 🡪 Compilador que corre no host e que gera o código executável para o microcontrolador.

**Transferência do código**:

* Programa-monitor (software)
  + Executa no arranque, comunicação com host por RS232.
  + Reside permanentemente no disco do microcontrolador.
  + Implementa funções de debug.
* Bootloader (software).
  + Reside permanentemente na memoria, não disponibiliza debug. Lê informação do porto de comunicação e escreve na memória Flash.
* In-Circuit Debugger (hardware)
  + Dispositivo externo proprietário, por exemplo, mete se o programa num CD e depois liga se o CD ao microcontrolador.
  + Isto é necessário para a transferência inicial de um bootloader/programa-monitor.

**Arquitetura de Harvard** implica a existência de dois espaços de endereçamento independentes: um para o programa e outro para dados.

O programa não pode ler dados da memoria de instruções e o cpu não pode ler instruções da memoria de dados.

**Bus Matrix** faz ligações ponto a ponto entre os módulos do microcontrolador aka cpu e ram ou cpu e memoria.

**I/O**

O sistema de I/O fornece mecanismos e recursos para comunicação entre o sistema computacional e o exterior. O dispositivo que assegura esta comunicação chama-se **periférico.**

É necessária uma interface (female port USB e.g) que forneça as adaptações entre as características intrínsecas do periférico e as do CPU/memória 🡪 **Módulo de I/O**

O I/O device (teclado) comunica então com o I/O Module formando um **periférico**.

**Módulo de I/O**

Este módulo assegura a compatibilização entre o sistema computacional e o dispositivo exterior.

O periférico liga-se ao sistema através dos barramentos, tal como qualquer outro dispositivo e.g memoria.

O módulo abstrai detalhes ao cpu.

A comunicação entre o cpu e o periférico é feita com operações de escrita e leitura, tal como um acesso a uma posição de memória, mas com I/O é possível que o valor associado a estes endereços de memoria mudem sem intervenção do cpu.

**Modelo de programação do periférico:** conjunto de registos (data, status e control. É comum um só registo incluir o control e status.) e a descrição de cada um deles (específicos a cada periférico)

**Comunicação entre cpu e dispositivos no geral**

O CPU toma sempre iniciativa. Envolve protocolos.

Apenas duas operações podem ser efetuadas: write (cpu para dispositivo) e read (dispositivo para cpu)

Para que o cpu **aceda** a um dispositivo, envolve:

1. utilizar o **barramento de endereços** para **especificar o endereço do dispositivo a aceder**.
2. utilizar **o barramento de controlo** para **sinalizar a operação a realizar**.
3. utilizar o **barramento de dados** para transferência de dados no sentido adequado.

**Seleção do dispositivo externo**

Operação de escrita (CPU 🡪 dispositivo)

* Apenas 1 dispositivo deve receber a informação que o CPU meteu no data bus.

Operação de leitura (CPU 🡨 dispositivo)

* Apenas 1 dispositivo pode estar ativo no data bus.
* Dispositivos inativos devem estar eletricamente desligados do data bus.
* Obrigatório **portas Tri-State** na ligação do dispositivo ao data bus.

Claro que num sistema computacional há vários circuitos ligados ao barramento de dados (I/O, memoria). Então o cpu tem de conseguir selecionar apenas um dos vários dispositivos.

Barramento simples liga apenas de A a B.

Barramento partilhado é tipo uma linha de comboio.

**Porta Tri-State** tem de entradas um Enable e um input bit e de saída o output. Se o enable estiver desligado existe alta impedância, ou seja, desliga-se eletricamente. Não é possível ter 2 enables no mesmo bus.

**Seleção do dispositivo externo**

Se CSn = 0, o dispositivo está inativo, em alta impedância, não é possível realizar escrita/leitura.

Se CSn = 1, o dispositivo está ativo e é possível realizar escrita/leitura.

Para geração dos sinais de seleção (CSn): descodificação de endereços.

**Memory-mapped I/O**

- Às **unidades de I/O são atribuídos endereços** do espaço de endereçamento de memoria.

- A memoria e unidades de I/O coabitam no mesmo espaço de endereçamento. Uma parte deste espaço é reservado para periféricos.

- Cada dispositivo tem a sua dimensão, o seu endereço inicial e final e o nº bits do address bus.

Por exemplo uma memoria de 64k apresenta 16 bits de espaço de endereçamento (0xXXXX); Nessa memoria podemos armazenar memoria rom de 2kB, que ocupa então 2048 bytes, vai de e.g 0xF800 a 0xFFFF (2048 bytes de diff) e o número de bits do address bus é 11.

**Isolated I/O**

- Memória e periféricos em espaços de endereçamento separados. Neste caso é especificado o espaço de endereçamento destino pelo control bus.

**Descodificação de endereços**

Acho que um dispositivo é selecionado quando a info do address bus (os bits mais significativos, falo deles mais à frente) == address atribuída ao dispositivo, esta operação de comparação faz se no descodificador, antes de chegar ao device ofc.

Descodificação total: Para uma dada posição de memoria existe apenas um endereço possível para acesso; Todos os bits relevantes são descodificados.

**Decodificação parcial:** **Vários endereços** possíveis para aceder à **mesma posição** de memoria; Apenas alguns bits são descodificados; circuitos mais simples e menores atrasos

Exemplos de descodificações com os seguintes dispositivos dentro de uma memoria de 64k com 16 bits de espaço de endereçamento

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dispositivo | Dimensão | Endereço Inicial | Endereço Final | Nr bits addr bus |
| RAM, 1k x 8 | 1024 | 0x0000 | 0x03FF | 10 |
| Porto de saída | 1 | 0x4100 | 0x4100 | 0 |

Porto de saída

* Descodificação total:

0x4100 = 0100 0001 0000 0000

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

* Decodificação parcial:

Não usar, e.g, os dois bits menos significativos

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

Gama de ativação do CS fica [0x4100, 0x4103]

Memória RAM

Tem 10 bits para address, portanto o resto dos 6 ficam como bits mais significativos e servem para descodificação.

Temos que garantir de certa forma que a ram está mapeada a partir do endereço 0x0000.

* Descodificação total:

Utilizamos todos os bits disponíveis (6 digamos a 000000), portanto um endereço só é valido para aceder à memoria se tiver os 6 MSBits a 0. CS=A15\.A14\.A13\.A12\.A11\.A10\

Com esta opção temos 1 endereço (único) para aceder a cada posição de memoria.

A memoria ocupa 1k na mesma.

* Descodificação parcial:

Usar A15, A14, A13 e A12 e ignorar A11 e A10, e.g. 0000xx CS=A15\.A14\.A13\.A12\

Com esta configuração vai ser possível aceder à memoria desde que os bits de 15 a 12 sejam 0.

Com esta opção a memoria ocupa, artificialmente, 4k endereços do espaço de endereçamento (4 endereços possíveis para aceder a cada posição de memoria 🡪 podemos aceder ao primeiro byte com 000000[0] 01, 10 e 11.

Zona ocupada é contígua.

* Descodificação parcial (outra opção):

Usar A13, A12, A11 e A10 e ignorar A15 e A14, e.g. xx0000 CS=A13\.A12\.A11\.A10\

Precisa de ter os bits de 13 a 10 a 0.

A memoria ocupa 4k como o de cima, mas a zona ocupada não é contígua e pode haver conflitos.

**Exercício de descodificação:**

Escrever a equação lógica do descodificador de endereços para uma memória de 4 kBytes, **mapeada num espaço de endereçamento de 16 bits** que respeite os seguintes requisitos:

- Endereço inicial: 0x6000; descodificação total.

Dentro de uma memória cujo espaço de endereçamento é de 16 bits, temos la dentro uma memoria de 4kBytes e o seu endereço inicial começa em 0x6000.

Para endereçar uma memoria de 4kbytes é preciso 2^2(= 4) \* 2^10(= k) bits. 12 bits então. Ficamos com 4 para fazer descodificação total.

**Descodificação total difere da parcial no sentido que usa todos os bits disponíveis.**

Digamos que os 4 MSbits ficam a 0110, logo A15 precisa de ser 0, A14 1, A13 1, A12 0.

Então para aceder ao início da memoria é 0110[0].

Só temos uma maneira para cada posição, portanto vai ocupar 4kbytes na mesma.

Resposta: Lógica positiva: CS = A15\.A14.A13.A12\

Lógica negativa: CS = A15+A14\+A13\+A12

**Gerador de sinais de seleção programável**

De volta ao exemplo da memoria de 10 bits addr e 6 descodificação, também se pode pegar nos 10 bits e fazer partições. Por isso ficava tipo 6|4|6. Não sei para que serve isto. Slides 34 aulas 2 e 3. Skip nisto.

**Estrutura portos I/O**

Um **porto de saída (LAT)** de 32 bits tem de entrada Clock, CS, WR e data bus (32 bits) e tem apenas uma saída de 32 bits. Instrução sw do MIPS.

O porto armazena, na transição ativa do clock, informação (nos **flip-flops**) proveniente do cpu se os sinais CS e WR estiverem a 1. **PARA LAT USAM SE FLIP-FLOPS.**

O sinal CS é gerado pelo descodificador e fica ativo se o endereço gerado pelo CPU coincidir com o endereço atribuído ao porto.

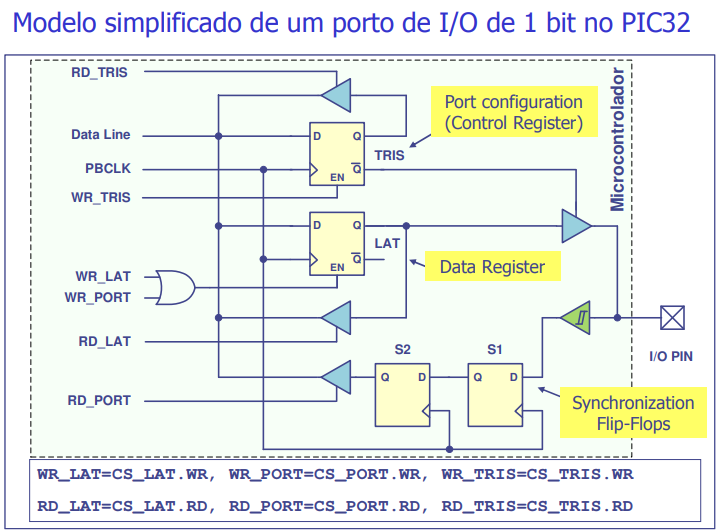
Um **porto de entrada (PORT)** de 32 bits (não armazena nada, no geral) tem como entrada os 32 bits do exterior, o CS e o RD e tem como saída os 32 bits do data bus. (Saidas vão para o CPU). Instrução lw do MIPS. **PARA PORT USAM SE TRI-STATE.**

As saídas têm portas **tri-state** que só se ativam quando se ativam os sinais CS e RD.

A leitura é assíncrona, não é necessário clock como no porto de saída.

Cada um dos bits de cada um dos portos da PIC32 pode ser configurado como entrada ou saída.

Um porto de I/O de n bits é um conjunto de n portos de I/O de 1 bit.



Os portos de entrada podem ter problemas de meta-estabilidade, porque o sinal externo é assíncrono relativamente ao CPU. Para ajudar neste problema, o sinal externo é sincronizado através de 2 flip-flops.

Acho que se pode escrever num porto de entrada

**Transferência de informação entre memória e I/O**

Como é que os periféricos (I/O Module + I/O device) comunicam com a memoria?

Primeiro efetua-se o pedido ao disco, segundo espera-se pela informação (**Latência**), por último transfere-se a informação do disco para a memoria.

**Técnicas de transferência de informação**

1. O cpu inicia e controla tudo

* Programmed I/O

O cpu toma a iniciativa e controla a transferência de informação, aguardando se necessário (**POLLING**).

* **Interrupt** driven I/O

O periférico diz ao cpu que está pronto para trocar informação e o cpu inicia e controla a operação.

1. O cpu não se envolve na transferência

* I/O por acesso direto à memoria (**DMA**)

Um dispositivo fora do cpu (DMA) assegura a transferência entre memoria e periférico.

O cpu apenas configura inicialmente o periférico e o dma e no fim da transferência o dma sinaliza o cpu a dizer que a transferência terminou.

Por polling é uma merda, o cpu ta sempre à espera num loop a ver se o periférico está disponível para a troca de informação.

Por interrupção, o periférico diz ao cpu quando está disponível para trocas.

Exemplo simplificado:

1. Cpu quer informação do periférico y, portanto manda-lhe pedido.
2. Cpu continua com as suas cenas
3. Assim que disponível, o periférico gera um pedido de interrupção ao cpu pelo canal Ireq
4. O cpu atende a interrupção, suspende o que está a fazer, faz a rotina de serviço da interrupção e no fim retoma o que tava a fazer.

Antes de entrarmos na rotina de serviço, é **preciso salvaguardar registos**.

**Exceções** (erros tipo overflow ou opcode invalido …) e Interrupções são cenas que alteram o normal fluxo do programa não sendo jumps nem branches.

Logo, uma instrução que tenha gerado uma exceção não vai ser terminada, damos logo skip para a rotina de tratamento de exceções. Com interrupções, o cpu executa uma instrução e no fim checka se há interrupções a executar (antes de seguir para a próxima instrução)

Dá para desativar interrupções se o cpu tiver a fazer um set de instruções importantes (**secção crítica**) e não quiser parar a meio.

**Processamento de interrupções pelo CPU**

1. Identifica-se a fonte da interrupção e obtém-se o endereço da rotina.
2. Salvaguarda dos registos e assim. (**Prólogo**)
3. Desativa-se interrupções (para não fazer interrupção dentro de interrupção).
4. Saltar para o endereço da rotina e executar tudo até ao fim (instrução de retorno).
5. Antes da instrução de retorno, repor as cenas salvaguardadas e reativar interrupções. (**Epilogo**)

Claro que se gasta tempo no prologo e epilogo, isto chama-se **overhead.**

**Organização do sistema de Interrupções**

Vamos ter muitos periféricos a fazer interrupções, logo é preciso organização.

Opção 1: **Múltiplas linhas de interrupção (hardware)**

O cpu tem uma entrada Ireq para cada periférico (e RSI), portanto a fonte é identificada automaticamente, mas há limitações de número máximo de periféricos que podem gerar interrupções.

Cada linha tem um prioridade fixa.

Opção 2: **Identificação da fonte de interrupção por software**

Apenas 1 Ireq no cpu (e 1 RSI).

A RSI lê o registo de status de todos os periféricos (em ordem 🡪 prioridade) até que encontre um que tenha gerado a interrupção.

Opção 3: **Interrupções vetorizadas (hardware)**

1 Ireq, id feita por hardware, cada periférico tem um id único(vetor), cada vetor tem uma RSI.

O vetor é usado como índice de uma tabela que contem endereço para RSI.

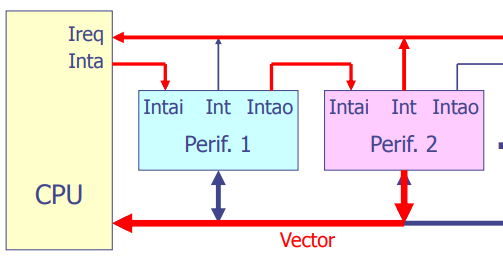
Aprofundando as interrupções vetorizadas…

A identificação da fonte é feita por **hardware** por **interrupt acknowledge cycle**

Periféricos podem estar organizados por **daisy chain (sequência ordenada)**

Procedimento por daisy chain:

1. Cpu deteta Ireq a 1, portanto ativa o Interrupt Acknowledge (Inta)
2. O Inta percorre a cadeia (com o Intao) até ao periférico que gerou a interrupção
3. O periférico que gerou a interrupção coloca o seu vetor no data bus e bloqueia a propagação do Inta (com o Intao).
4. Cpu lê o vetor e usa como índice da tabela que contem os endereços das RSI.

Foi o Perif. 2 a gerir interrupção, então desativa o Intao para interromper a propagação do intai.

**Tabela de vetores**

**- Organização**

* Tabela inicializada com os endereços de todas as RSI:

O cpu acede à tabela usando como índice o vetor.

* Tabela inicializada com instruções de salto para as RSI:

Vetor usado como offset (mesma cena +/-).

Exemplo:

Vetor = 2; Endereço base = 0x9D000200  
espaçamento (entre instruções na tabela) = 32 (e.g)  
Endereço tabela = vetor \* 0x20 (32 em hex) + 0x9D000200 = 0x9D000240

No PIC32 pode-se utilizar single-vector mode (único vetor, faz -se id por software) ou multi-vector mode (vetorizadas).

**DMA**

Um cpu para realizar operações de memoria, gasta tempo nas interrupções e dentro da RSI, gasta tempo nas instruções de ler words e armazenar words.

A solução para isto é o **Direct Memory Access** ou DMA e consiste na transferência de informação do periférico diretamente para a memória (ou o contrário), **sem intervenção do processador na transferência. Só tem que dizer ao DMAC a operação.**

É o DMAC (DMA Controller) que efetua a transferência.

Durante a mesma, este tem a habilidade de controlar os barramentos como se fosse um cpu.

Quando o DMAC acaba a transferência (que é feita por **hardware**), gera uma **interrupção**.

**Funcionamento de transferência de dados com DMAC**

1. Lê uma palavra do source address
2. Escreve essa palavra no destination address
3. Incrementa source e destination address assim como o número de palavras transferidas
4. Continua até transferir o bloco.

Para isto o DMAC precisa de acesso aos barramentos todos, então o cpu não pode estar a usá-los para evitar conflitos. Portanto, só pode haver em cada instante um **bus master**

Logo, antes da transferência, o DMAC **pede autorização ao cpu** para ser bus master e **espera até ter confirmação**. Faz o funcionamento de cima e no **fim retira o pedido (busReq) para ser bus master**.

Para se tornar bus master, o DMAC ativa o busReq e espera pelo busGrant ativo (pelo cpu).

**DMA – Modos de operação**

1. **Bloco**

* O DMAC assume o controlo dos buses até todos os dados serem transmitidos.

1. **Burst**

* O DMAC transfere até atingir o número de words pré-programado ou até chegar ao fim de info do periférico.
* Caso não seja transferido tudo, o DMAC liberta os bus e na próxima vez que seja bus master continua onde ficou.

1. **Cycle Stealing**

* O DMAC aproveita os ciclos que não usados pelo CPU, ou seja, quando este não pretende aceder à memoria.
* O DMAC assume controlo dos buses durante 1 bus cycle e liberta-os de seguida levando a **transferências parciais**.
* Transferência lenta, mas **não traz impacto** ao desempenho do cpu.

**DMA – Transferências**

Modo bloco:

1. O CPU manda um comando a um disco: leitura de um setor, número de words
2. O CPU programa o DMAC: source address, destination address, número de words
3. CPU continua com as suas tarefas
4. Quando o disco tiver lido a info pedida para a sua zona de memoria, ativa o sinal Dreq do DMAC (diz ao DMAC que está pronto para transferir)
5. O DMAC ativa o sinal BusReq, fica à espera do BusGrant
6. O CPU assim que possível ativa o busgrant e o DMAC ativa o Dack e o disco desativa o Dreq.
7. O DMAC começa a transferência: lê do source address para um registo interno, escreve no destino o que tá no registo interno e incrementa as suas cenas.
8. No fim da transferência, o DMAC desativa o Dack, desativa o busreq e ativa o sinal de interrupção
9. O cpu vê o busreq a 0 e então desativa também o busgrant.
10. Cpu, assim que possa, atende a interrupção do DMAC.

Modo cycle stealing:

1. DMAC torna-se bus master
2. Lê palavra do source address
3. **Liberta barramentos**
4. Espera um tempo fixo (e.g 1 clock)
5. Torna-se bus master
6. Escreve 1 palavra no destino
7. **Liberta** **barramentos**
8. Incrementa as suas cenas
9. Repete até transferir tudo e no fim gera interrupção

São **2 bus cycles por palavra**

**DMAC Dedicado**

**1** bus cycle por palavra

**1**

**Timers**

Contam até um número

PBCLK / 2^16 / fOut = PRx 🡪 PRx arredondar para cima 1,2,4,8,16,32,64,256

PBCLK / fOut / PRx = Valor a meter no timer.

PBCLK = 20e6; fOut = Frequência pretendida; PRx = constante

**Watchdog Timer**

Tem como função monitorizar a operação do cpu e em caso de falha **forçar o seu reinício**

Por exemplo, se o cpu não atuou a entrada de reset do watchdog timer ao fim de um tempo pré-determinado, o watchdog força o reset do cpu.

E.g o watchdog timer conta até 5, o cpu dá reset ao watchdog em t = 4, então tem que dar reset outra vez no máximo em t = 9 senão o watchdog vai reiniciar o cpu.

**PWM** consegue manter a frequência enquanto altera o duty cycle.

Na detpic32 os timers tipo B podem ser agrupados 2 a 2 para formar 2 timers de 32 bits

A **resolução** de **PWM** é log2(PRx+1)

**Barramentos**

Barramentos servem para a **interligação** dos blocos (cpu, memoria, I/O) num sistema de computação

Aos buses estão ligados **Bus Masters** (pode iniciar e controlar transferências) ou **Bus slaves** (só responde a pedidos de transferências)

**Barramento paralelo** 🡪 os dados são transmitidos em paralelo através de **múltiplas linhas**

Inclui data bus, address bus e control bus

Transmissão paralela implica dificuldades assim como ruído, interferência mútua e elevado custo.

**Barramento série** 🡪 transmite-se 1 bit de cada vez a cada ciclo de clock

Transmissão série implica simplicidade e diminuição de custos

Tipos de transmissão série: **Simplex** (One-Way), **Half-duplex** (nos dois sentidos, mas um de cada vez) e **Full-duplex** (nos dois sentidos simultaneamente, mas são usadas duas linhas)

É necessária a sincronização entre transmissor e recetor.

Esta é alcançada através da utilização do mesmo clock nos dois dispositivos ou através de clocks independentes que terão que estar sincronizados durante a transmissão

**Sincronização entre transmissor e recetor**

* **Transmissão Síncrona**

O clock é transmitido explicitamente através de um **sinal adicional** ou implicitamente **através da linha de dados**.

Os clocks devem se manter sincronizados

* **Transmissão Assíncrona**

Não há clocks, é necessário acrescentar bits para sinalizar o **start** e o **stop** da transmissão

**Técnicas de sincronização de relógio**

* **Relógio explícito do transmissor**

O transmissor **manda** dados por uma linha e clock por **outra**.

* **Relógio explícito do recetor**

O recetor **recebe** dados por uma linha e **manda** clock por **outra**.

* **Relógio explícito mutuamente-sincronizado (“clock stretching”)**

Ambos os dispositivos têm o seu próprio clock. Então fazem uma porta (wired) **AND** entre estes 2 clocks e usam isso como clock.

* **Relógio codificado**

Relógio enviado **em conjunto** com os dados, o transmissor codifica o clock nos dados e o recetor descodifica o clock na entrada

* **Relógio implícito (Assíncrona)**

Relógios **independentes**/locais

Relógio do recetor sincronizado ocasionalmente com o do transmissor através da receção de símbolos específicos

Pode haver desvios

**Transmissão de dados**

**Transmissão orientada ao Byte**

Informação independente

É a operação indivisível do barramento

**Transmissão orientada ao bit**

Informação organizada em tramas

Tramas constituídas por **delimitadores** (símbolos de sincronização) seguido por uma sequência de bits de qualquer comprimento.

**Topologias**

**Ponto a ponto (half ou full-duplex)** em que o master só ta ligado a um slave

**Multiponto** **(half-duplex)** em que o master comunica com mais do que um slave com apenas um canal.

**SPI – Serial Peripheral Interface**

**Master-Slave** com ligação **ponto a ponto, full-duplex, síncrona (**clock explicito do master)

**Clock gerado pelo master** que o disponibiliza a todos os slaves

O sistema só pode então ter um master e este é o único a controlar o clock e a iniciar e controlar a transferência de dados.

Apenas **1 slave é selecionado pelo master** (porta SS), apesar deste estar ligado a vários slaves.

Como é full-duplex, o master tem uma linha (MOSI) para enviar dados e o Slave outra (MISO) para enviar dados.

Em cada ciclo de relógio o master manda 1 bit e o slave também **sempre**

Cada um tem um shift-register que é de onde saem e onde estão os bits

O clock tem **duty cycle de 50%**

Na transição negativa enviam os valores, na positiva armazenam os valores recebidos

**Operação**

1. O master ativa a linha **SSx** (select slave) do slave que quer
2. Ativa o clock
3. Em cada ciclo, na transição negativa, o master e o slave colocam 1 bit nas linhas respetivas de envio, na transição positiva armazenam o bit recebido e colocam no fim do shift-register.
4. No fim de toda a transferência, os dois **trocaram o conteúdo dos seus shift-registers**
5. O master desativa a SS e o clock

Só é possível ter um SSx ligado (só se pode selecionar **um slave de cada vez**)

Quantas mais linhas SS no master, mais slaves este pode ter, mas o microcontroladeiro pode gerar SS’s digitalmente se necessário.

Pode se fazer daisy chain com os slaves mas caguei

Os **dados são sempre trocados** entre master e slave independentemente de só ser preciso dados de um lado.

Se o master quiser escrever no slave, manda os seus dados e descarta o que recebe

Se quiser ler dados do slave, manda informação irrelevante para o slave (e.g 0’s)

Antes de iniciar a transferência, o master precisa de ser configurado com clk, duty cycle e se vai haver transmissão no rising clock ou então no falling clock

No pic32, cada modulo pode ser configurado como master ou slave

**I2C (Inter-Integrated Circuit)**

**Multi-Master**, **half-duplex**, **síncrono (explicito mutuamente sincronizado),** **orientado ao byte (é sempre mandada uma sequência com comprimento fixo)**, **apenas 2 fios** (🡪 tem 2 no nome), o master pode ser o recetor ou transmissor

Deteção de colisões e **arbitragem** para masters

Os dois fios são **SDA** (data) e **SCL** (clock). **Master e slave têm o seu próprio clk mas fazem um wired AND com o SCL.**

O master **inicia** a transferência, gera o clock e termina a transferência. O slave pode condicionar o clock (forçar a 0)

Vários masters podem tentar simultaneamente controlar o bus, mas é necessária **arbitragem** para deixar apenas um continuar sem causar perturbações

Existem bits recessivos (1) e bits dominantes (0)

O primeiro byte transmitido pelo master contém 7 bits (**slave address**) e 1 bit (**read ou write**)

Se o último bit for 1, o master é recetor, se for 0 é transmissor

**Cada slave vai ler o endereço** e se for igual, muda para estado transmissor ou recetor.

SDA só troca quando SCL = 1, se SDA trocar enquanto SCL = 1, isto sinaliza **condições** (geradas sempre pelo master)

Condição **START:** SDA vai de 1 para 0 enquanto SCL = 1

Condição **STOP:** SDA vai de 0 para 1 enquanto SCL = 0

**Processo de transferência de dados**

1. O master envia um start
2. O master envia o slave address e o bit de operação
3. O slave correspondente manda um ACK (bit dominante (0))
4. O transmissor manda 1 byte
5. O recetor manda um ACK
6. No final, o master envia um STOP

Então, quando o master é o recetor, se continuar a mandar ACK’s, isto quer dizer **que quer continuar a ler dados**. Se não quiser, manda um not ACK e depois um stop.

**Clock stretching –** O slave pode forçar o clock a 0 (se meter wait = 1) e alargar a transferência

Num barramento livre (i.e depois de um stop), múltiplos masters podem decidir começar transmissão ao mesmo tempo. É preciso arbitrar este processo. Quem **perder retira-se** e só tenta outra vez quando o bus tiver livre.

No processo de arbitragem é preciso primeiro sincronizar relógios

**Sincronização**

Quando SCL vai de 1 para 0, todos os masters metem os seus relógios a 0.

Quando o clock de um master termina o tempo de estar a 0, mete a 1, ou seja, deixa SCL passar a 1. Se SCL se manter a 0, isto quer dizer que ainda há masters a 0 e espera por todos.

Quando tiver tudo a 1, SCL passa a 1 e inicia a contagem do tempo a 1. O primeiro master a terminar o seu tempo a 1 força o SCL para 0.

De seguida, quando o bus ta livre e dois masters geram START, é preciso ver qual ganha.

Cada master verifica a SDA (quando SCL = 1) e se o valor coincide com o que enviou, fica master.

Se der igual nos 2 masters, a arbitragem continua.

**CAN (Controller Area Network)**

Comunicação diferencial em par entrançado (???)

Elevada robustez, muito seguro, é usado então em cenas importantes 🡪 carros, medicina…

**Broadcast, half-duplex, encapsulamento em tramas, multi-master, assíncrono (implícito), orientado ao bit, ID único de cada mensagem** que identifica natureza do conteúdo e prioridade.

Fácil de acrescentar nós (dispositivos) novos.

Os nós têm um CAN Controller e um Transceiver

O nível logico (que vem da linha Tx do controller até ao transceiver) é transformado em 2 tensões e as tensões são colocadas em duas linhas CAN\_H e CAN\_L. Na receção, no transceiver, faz-se a diferença de tensão entre estas 2 linhas e é enviado o resultado através da linha Rx para o controller

**Bit stuffing –** Por cada 5 bits iguais de seguida, é inserido 1 bit de polaridade oposta

Assegura sincronização

**Formato da trama**

Começo da frame 🡪 um start bit (0) que é utilizado para sincronizar clocks de recetores

11 bits para identificador que também serve como prioridade (baixo id, mais prioritário)

1 bit que é 0 numa transmissão de dados

Mais bits para controlo, dados, crc, ack, end of frame e 3 bits a 1 para intermission

**Tipos de tramas**

**Data Frame** – envio de dados de 1 produtor para 1 ou mais recetores. RTR = 0

**Remote Transmission Request Frame** – 1 recetor solicita ao produtor a transmissão de uma frame específica. RTR = 1

**Error Frame** – reportar um erro detetado (sobrepõe-se a transmissões em curso)

**Overload Frame** – Atraso o envio do próximo frame porque ta em sobrecarga e precisa de recuperar tempo.

**Erros**

**CRC Error** – o CRC calculado não coincide com o recebido

**ACK Error** – o produtor não recebe um 0 no campo ACK, ou seja, ninguém recebeu a mensagem

**Form Error** - se houver 0’s em campos que só devem ter 1’s (e.g fim da frame)

**Bit Error** – o produtor analisa cada bit transmitido, se ler algo que é oposto do que escreveu, gera erro

**Stuffing Error** – Não há stuffing bit onde devia haver

Um CAN Controller tem um filtro de mensagens onde verifica ID’s.

O CAN Controller lê todas as mensagens e coloca-as num **buffer temporário**. Se for uma mensagem valida e se passar no filtro, a mensagem é copiada para o buffer de receção

As máscaras fornecem filtros mais avançados ao programador

A arbitragem faz-se com o id (mais baixo ganha) e é baseada em bit recessivo / dominante

**RS-232C**

**Assíncrona, full-duplex, ponto a ponto**

Sem handshakes, só tem linha de Rx, Tx e ground

Sinal obtido com a diferença entre tensão num fio e ground

Apresenta parity bit (opcional), start (0) e stop bit(s) (1, ou seja, idle)

**Parity even** 🡪 conta-se os 1’s nos dados, se der ímpar adiciona-se um 1, else 0

**Parity odd** 🡪 se der par adiciona-se um 1, else 0

Sinais enviados por tensão (negativa é 1 e positiva é 0)

**Baudrate** 🡪 número de bits enviados por segundo.

1 start bit, 8 data bits, 1 parity, 2 stop bits 🡪 12 bits por transmissão

**Bit rate líquido** é só os data bits

O relógio do recetor é sincronizado a partir do (início do) start bit.

Para a transmissão correr bem, os dispositivos devem estar configurados com o mesmo nº de data bits, tipo de paridade, nº de stop bits e baudrate

Se o clock for muito diferente, irá haver desvios que podem causar **erros**

**Erro de paridade** parity bit diferente

**Erro de framing** detetado um 0 no stop bit

**Erro de fase** o transmissor reseta o clock assim que manda o start bit… o recetor vê o start bit e faz o mesmo, mas há um certo delay (cabo longo, ainda mais) até o bit ir do transmissor ao recetor.

Muita matemática para calcular máximo desvio i shleep

**Device drivers**

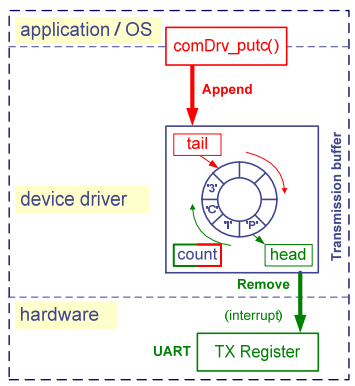
Existem periféricos muito variados (rato, disco, pen, camara…) e cada um apresenta características assim como, operações suportadas, bandwidth, se é byte-addressable…

Os sistemas operativos não vão conhecer todos os periféricos em detalhe, então criou-se uma camada de abstração que permite o acesso ao dispositivo de forma independente da sua implementação 🡪 **Device Driver**

Fornecimento sempre assegurado pelo fabricante, é um programa que permite a outro programa (OS) interagir com o hardware

O acesso a um device driver em sistemas embutidos é direto, mas em OS’s tipicamente há system calls/funções. O kernel do OS acede ao device driver.

Basicamente na aplicação faz-se read (8, r\_memory)

Se fossemos ler um carater da UART, fazíamos getc na aplicação e teríamos um FIFO como na figura

**Memória**

**RAM – Random Access Memory**

* Memoria volátil (informação perde-se no shutdown); pode ser lida e escrita
* O acesso é random, logo é igual para qualquer posição na memória

**ROM – Read Only Memory**

* Memoria não volátil; apenas pode ser lida
* Acesso também é random

**Memória não volátil (disco)**

**ROM** – programada durante o processo de fabrico

**PROM** – Programmable Read Only Memory: programável uma **única** vez

**EPROM** – Erasable PROM: escrita em segundos, apagamento em minutos (ambas efetuadas em dispositivos especiais)

**EEPROM** – Electrically Erasable PROM

• O apagamento e a escrita podem ser efetuados no circuito em que a memória está integrada

• O apagamento é feito byte a byte

• Escrita muito mais lenta que leitura

**Flash EEPROM** - (tecnologia semelhante à EEPROM)

• A escrita pressupõe o reset prévio das zonas de memória a escrever

• O reset é feito por blocos o que torna esta tecnologia mais rápida que a EEPROM

• O reset e a escrita podem ser efetuados no circuito em que a memória está integrada

Existem diferentes tecnologias de memoria, as mais rápidas custam mais (SRAM > DRAM > Flash > Disco)

É vantajoso construir o sistema de memoria como uma **hierarquia** onde se usa isto tudo

Memória é basicamente um **conjunto de registos de tamanho x**

Cada registo é **formado por x células e cada célula pode armazenar 1 bit**

Cada célula tem entrada sel para ser selecionada e **entradas negadas** wr\ e rd\

Podemos agrupar então células para se formar memoria maior, sabendo o número total de words e o word size.

Então uma memoria com **1k x 8 é 1k words de 8 bits cada word** 🡪 2^10 endereços

Celulas organizam em matriz e para aceder às mesmas, usa se um decoder

**RAM**

**SRAM – Static RAM**

Rápida e **volátil**, mas custa muito, necessita **6 transístores/célula** e baixa densidade

Para escrever um 1, mete-se na linha bit = 1 e bit\ = 0 e select = 1

Para ler, select = 1 e o valor logico é a diferença de tensão entre bit e bit\

Na organização interna, podem-se colocar células umas em cima das outras ((i x j) x k)

**Interface SRAM** aceita address, enable negado, escrita e leitura negado, write tem prioridade em read

**Cycle time:** tempo de acesso mais algo adicional

**Access time:** tempo para os dados ficarem disponíveis no output

**Taxa de transferência:** 1/cycle time

Pode-se aumentar a memoria adicionando mais posições (de words de tamanho igual)

32k8 🡪 256k8

**DRAM – Dynamic RAM**

**Custa** **pouco**, alta densidade, **1 transístor/célula**, mas a informação dura apenas 1 ms e dá **refresh** e é mais lenta que a SRAM

Refresh periódico **obrigatório**. **Leitura é destrutiva** e descarrega o condensador. Na ausência de leitura, o condensador vai **descarregando lentamente**

Para ler, ativa-se a linha bit e select

Para escrever, coloca-se dado na linha bit e ativa-se select

O refresh é tipo read

O endereço de acesso à memória é dividido em 2 partes: row e column address

Então o address bus é multiplexed, primeiro envia-se o row e depois a column

**RAS – Row Address Strobe**

**CAS – Column Address Strobe**

A interface é: entradas: addr, bit para escrita/leitura negado (0 = write), ras\ valida endereço da linha na transição descendente e cas\ valida endereço da coluna. De saída, dados

O CPU fala com o memory controller e este é que trata de tudo: gera os sinais para a interface, faz multiplexagem, faz o refresh…

Row buffer armazena temporariamente todos os bits de uma linha de células da matriz

No refresh não há troca de informação com o exterior e é feito linha a linha, ao mesmo tempo em todas as células de uma linha

O refresh **RAS** faz só refresh numa dada linha

**Fast Page Mode** adiciona sinais de temporização que permitem acessos repetidos ao row buffer

**Synchronous DRAM (SDRAM)** adiciona clk na interface DRAM, para facilitar sincronizações

**Double Data Rate DDR SDRAM** transfere dados em rising como em falling clk, duplicando a taxa de transferência.

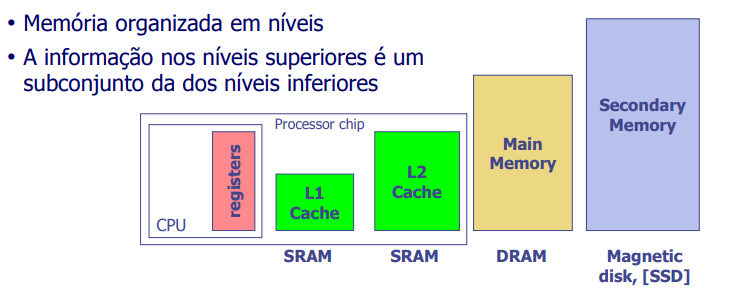
Melhoram apenas a bandwidth, não a latência.

**Organização e hierarquia da memória**

O cpu tem que ser alimentado de instruções e dados muito rapidamente, mas o cpu está a evoluir mais rápido (nas últimas décadas) que a memoria, pode-se criar eventuais bottlenecks

Como **solução**, guarda-se a **informação mais utilizada numa SRAM** de pequena capacidade 🡪 **CACHE**

Raramente aceder à memoria principal que é mais lenta para obter info em falta



É suposto aceder muitas vezes à cache e copiar informação para a mesma poucas vezes.

Então a probabilidade de a informação requisitada no momento estar nos níveis mais elevados da hierarquia tem que ser elevada

**Princípio da localidade:** Os programas acedem à memória usando endereços que se situam na vizinhança uns dos outros, ou seja, um programa acede a uma zona reduzida do espaço de endereçamento.

* **Spatial locality:** Se uma posição foi acedida, então é possível que os espaços contíguos também sejam. Exemplos: instruções de um programa, aceder a um array.

Quanto **maior** o bloco, **melhor**

* **Temporal locality:** Se uma posição foi acedida, então é possível que seja acedida outra vez no futuro. Exemplo: instruções de um while, variável de controlo.

Quantos **mais blocos** estiverem na memoria rápida, melhor

O nível superior (cache) é de pequena dimensão, mais rápido e contém os blocos mais recentemente utilizados pelo cpu.

Os pedidos de info são sempre dirigidos ao primeiro nível e só passa deste para a frente se a informação não estiver no nível

Se encontrar, dá-se um **hit** e manda a info para o cpu, else **miss** e encontra espaço na cache para um novo bloco, acede à memoria principal e lê o bloco que o cpu quer e o conteúdo é mandado para o cpu.

**Tempo médio de acesso à info:** hit ratio cache \* tempo de acesso cache + miss ratio cache \* hit ratio ram \* tempo acesso ram (+ cache) + miss ratio ram \* tempo acesso disco (+ cache + ram)

Quando o acesso a uma word falha, vai se ao nível seguinte, mas acede-se também às palavras adjacentes.

O cpu é abstraído de tudo isto.

**Organização da cache**

Há 3 formas de organizar a cache: cache **totalmente associativa**, com **mapeamento direto** e **parcialmente associativa** (set associative)

Imaginemos uma memoria principal byte-addressable de 16 bits… Tem 64kB

Imaginemos uma memoria cache de 2kB em que cada linha tem 8 bytes… São 256 linhas e cada linha tem 8 bytes e espaço de endereçamento fica 1 byte acho

Assim a memoria principal pode ser vista como sendo constituída por, em vez de 64k linhas de 1 byte, 8k linhas de 8 bytes (0xXXXX)

Assim no endereço de 16 bits, os 13 bits (tag) (de 2^13=8k) mais significativos identificam o bloco e os 3 menos significativos identificam o byte no bloco (já que cada bloco tem 8 bytes)

**Mapeamento associativo**

Basicamente o cpu quer o conteúdo de uma address de 16 bits (13 bits para a tag e 3 para o byte) e essa tag é comparada com todas as entradas da tag memory. Quando se encontrar a tag, vemos se é valida (valid bit = 1) e vai se buscar a cache o bloco e escolhe se o byte com os 3 bits e já ta.

Qualquer bloco da memoria principal pode ser colocado em qualquer posição na cache, mas

A tag tem que ter todos os bits do número do bloco o que implica comparadores muito grandes, todas as entradas tag têm que ser analisadas 🡪 muitos comparadores, custa muito

**Mapeamento direto**

Cada bloco da memoria principal é associada uma linha da cache

É simples de implementar, mas vários blocos têm associada a mesma linha e num dado instante apenas um bloco de um dado grupo pode residir na cache, o que tem como consequência que alguns blocos podem ser substituídos e recarregados várias vezes, mesmo existindo

espaço na cache para guardar todos os blocos em uso

Para melhorar este último ponto, devia-se permitir o armazenamento simultâneo de mais que um bloco do mesmo grupo

**Mapeamento parcialmente associativo**

Semelhante à anterior, mas permite que mais que um bloco de um mesmo grupo possa ser carregado na cache

Uma cache com associatividade de 2 (2 vias) permite que 2 blocos do mesmo grupo possam estar ao mesmo tempo na cache

----INCOMPLETO----

Nos exercicios basicamente dao nos o tamanho da cache, dizem como ta organizada e querem os bits de offset, group e tag

Por exemplo uma cache de 64KB, com blocos a ocupar 8 Bytes, com associatividade de 2.

A cache tem 64K / 8 (2^6\*2^10 / 2^3) = 2^13 blocos.

Como tem associatividade de 2, divide se o numero de blocos, 2^13 por 2. Ficamos então com 2^12 **linhas.**

Portanto para ver que linha queremos, precisamos de 12 bits, para ver que byte queremos dentro da linha, precisamos de 3 bits e o campo mais significativo que é a tag precisa de x bits de endereçamento – 12 – 3.

**Políticas de substituição de blocos**

FIFO, quem tá la há mais tempo sai

LRU – Least Recently Used, o que ta la há mais tempo sem ser referenciado

LFU – Least Frequently Used, o menos acedido

Random – Não é muito pior que LRU

**Políticas de escrita**

**Write-through:** todas as escritas são feitas simultaneamente na cache e na memoria principal, se o endereço não existir na cache, atualiza só a memória principal **(write-no-allocate)** e, portanto, a memoria principal está sempre consistente.

**Write-back:** Valor escrito apenas na cache, novo valor escrito na memoria quando o bloco da cache é substituído (não há duplicados), **dirty bit** é ativado quando há uma escrita em qualquer endereço do bloco presente na linha da cache. Se o endereço não existe na cache, carrega o bloco para a cache e atualiza-o **(write-allocate)**.

**Memória virtual**

Num sistema apenas com memória física, os endereços gerados pelo cpu **apontam diretamente** para as posições de memoria que pretende aceder, não é preciso nenhum processo.

Este sistema tem problemas no caso de memória insuficiente, gestão de memória disponível, segurança em casos de programa A escrever na memoria do programa B…

Utiliza-se **memoria virtual**, uma abstração (não se usa em microcontroladores)

Esta aumenta a **eficiência na utilização da memória**, **segurança**, **transparência** **e** **partilha de memoria**.

**Utilização eficiente da memoria física**

* Vários programas podem estar em execução ao mesmo tempo
* A memoria pode estar fragmentada
* A memoria total necessária a executar todos os programas pode ser superior à memoria física disponível, mas apenas uma fração dessa quantidade está ativamente a ser usada num determinado instante de tempo
* Assim, na memoria principal apenas é necessário ter zonas ativas de todos os programas a correr (o resto pode ir para o disco)
* O sistema operativo pode reservar mais memória para um processo ou libertar memoria que não esteja a ser usada de acordo com as necessidades

**Segurança**

* Muitos processos partilham a mesma memoria física
* É preciso suar mecanismos de proteção para não haver overwrites
* Endereços usados pelos processos não são endereços da memoria física

**Transparência**

* Cada processo tem o seu espaço de endereçamento que pode usar na totalidade, de forma exclusiva
* Cada processo executa como se fosse o único a ocupar a memoria

**Partilha de memória:** Com a implementação de mecanismos de proteção, torna-se possível a partilha de zonas de memoria entre processos

Os endereços virtuais gerados pelo cpu são convertidos para endereços físicos através da **Page Table**

**Endereço virtual é gerado pelo cpu** e o espaço de endereçamento virtual é a coleção de todos os endereços virtuais

Endereço físico é um endereço da memoria principal/disco e o espaço de endereçamento físico é a coleção de todos os endereços físicos

Um endereço virtual é traduzido num físico quando o cpu acede à memoria

Na memoria virtual, blocos são páginas (Os espaços de endereçamento virtual e físico são divididos em blocos)

Cada processo tem então o seu próprio espaço de endereçamento virtual

Slide 401

INCOMPLETO