# **APONTAMENTOS\_AC\_II\_2022\_2023**

**Introdução**

**Microprocessador** (basicamente um CPU):

* 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 que inclui CPU, RAM, ROM e periféricos;
* Frequência de funcionamento normalmente baixa (<200MHz);
* Baixo consumo de energia;
* Disponibiliza uma grande variedade de periféricos e interface 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, Memória (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 de 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 memória, 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;
  + Insto 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 memória de instruções e o CPU não pode ler instruções da memória de dados.

**Bus Matrix** faz ligações ponto a ponto entre os módulos do microcontrolador também conhecido como CPU e RAM ou CPU e memória.

**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 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. memória). 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 memória mudem sem intervenção do CPU.

**Modelo de programação do periférico:** conjunto de registos (data, status e control; NOTA: 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 em geral**

O CPU toma sempre a 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 um dispositivo deve receber a informação que o CPU meteu no DATA BUS.

Operação de leitura (CPU <- Dispositivo)

* Apenas um dispositivo pode estar ativo no DATA BUS;
* Dispositivos inativos devem estar eletricamente desligados do DATA BUS;
* Obrigatórias **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, memória). 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.

**Portas 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 memória;
* A memória e unidade 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 número de bits do address bus. Por exemplo, uma memória de 64k apresenta 16 bits de espaço de endereçamento (0xXXXX); nessa memória podemos armazenar memória ROM de 2kB, que ocupa então 2048 bytes, vai de e.g. 0XF800 a 0xFFFF (2048 bytes de diferença) 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 a frente) = address atribuída ao dispositivo, esta operação de comparação faz-se no descodificador, antes de chagar ao device oficialmente.

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

**Descodificação parcial**: **Vários endereços** possíveis para aceder à **mesma posição** de memória; apenas alguns bits são descodificados; circuitos mais simples e menores atrasos.

Exemplos de decodificações com os seguintes dispositivos dentro de uma memória de 64k com 16 bits de espaço de endereçamento.

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

**Porto de saída**

* Descodificação total:

0x4100 = 0100 0001 0000 0000

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

* Descodificação parcial:

Não usar, e.g., os 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 de 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 válido para aceder à memória 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 memória;

A memória 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 memória desde que os bits de 15 a 12 sejam 0;

Com esta opção de memória ocupa, 4k endereços do espaço de endereçamento (4 endereços possíveis para aceder a cada posição de memória -> 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 memória 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 4kB, **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 lá dentro uma memória de 4kB e o seu endereço inicial começa em 0x6000.

Para endereçar uma memória de 4kB 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 memória é o11o[0].

Só temos uma maneira para cada posição, portanto vai ocupar 4kB 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 memória de 10 bits address e 6 descodificação, também se pode pegar nos 10 bits e fazer partições. Por isso ficava tipo 6|4|6. {SLIDE 34 – AULA 2 E 3}.

**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. (Saídas 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.

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

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

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

**Técnicas de transferência de informação**

1. O CPU inicia e controla tudo
   1. Programmed I/O

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

* 1. **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
   1. I/O por acesso direto à memória (**DMA**)

Um dispositivo fora do CPU (DMA) assegura a transferência entre memória 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 está 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 estava a fazer.

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

**Exceções** (erros tipo overflow ou opcode inválido, …) e Interrupções são cenas que alteram o fluxo normal 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 verifica se há interrupções a executar (antes de seguir para a próxima instrução).

Dá para desativas as interrupções se o CPU estiver 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, repôs as cenas salvaguardadas e reativas interrupções (**epílogo**).

Claro que se gasta tempo no prólogo e epílogo, 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 uma 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 contém 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 de INTA (com o INTAO);
4. CPU lê o vetor e usa como índice da tabela que contém os endereços das RSI.



Foi o Perif. 2 a gerir a 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 hexadecimal) + 0x9D000200 = 0x9D000240

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

**DMA**

Um COU para realizar operações de memória, 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 de 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 coo o número de palavras transferidas;
4. Continua até transferir o bloco.

Para isto o DMAC precisa de acesso aos barramentos todos, netão 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 o bus master e **espera até ter confirmação**. Faz o funcionamento de cima e no **fum retira o pedido (busReq) para ser bus master**.

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

**DMA – Modos de operação**

1. **Bloco**
   1. O DMAC assume o controlo dos buses até todos os dados serem transmitidos.
2. **Burst**
   1. O DMAC transfere até atingir o número de words pré-programado ou até chegar ao fim de info do periférico;
   2. Caso não seja transferido tudo, o DMAC liberta os bus e na próxima vez que seja bus master continua onde ficou.
3. **Cycle Stealing**
   1. O DMAC aproveita os ciclos que não são usados pelo CPU, ou seja, quando este não pretende aceder à memória;
   2. O DMAC assume controlo dos buses durante 1 bus Cycle e liberta-os de seguida levando a **transferências parciais**;
   3. 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 memória, 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 está 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 uma 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

**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 = 2Oe6; fOut = Frequência pretendida; PRx = constante

**Watchdog Timer**

Tem como função monitorizar a operação do CPU e em caso de falha **forçar o seu reinicio**.

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 de 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, memória, 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 de estar sincronizados durante a transmissão.

**Sincronização entre transmissor e recetor**

* Transmissão Síncrona (Relógio explícito do transmissor, do recetor e mutuamente-sincronizado e relógio codificado (“self-clocking”))

O CLOCK é transmitido explicitamente através de um sinal adicional ou implicitamente através da linha de dades;

Os CLOCKS devem se manter sincronizados.

* Transmissão Assíncrona (Relógio implícito)

Não há CLOCKS, é necessário acrescentar bits para sinalizar o START e o STOP da transmissão.