

# Laboratório de Introdução à Arquitetura de Computadores

IST - Taguspark

2020/2021

## Programação cooperativa

### Guião 6

23 a 27 de novembro 2020

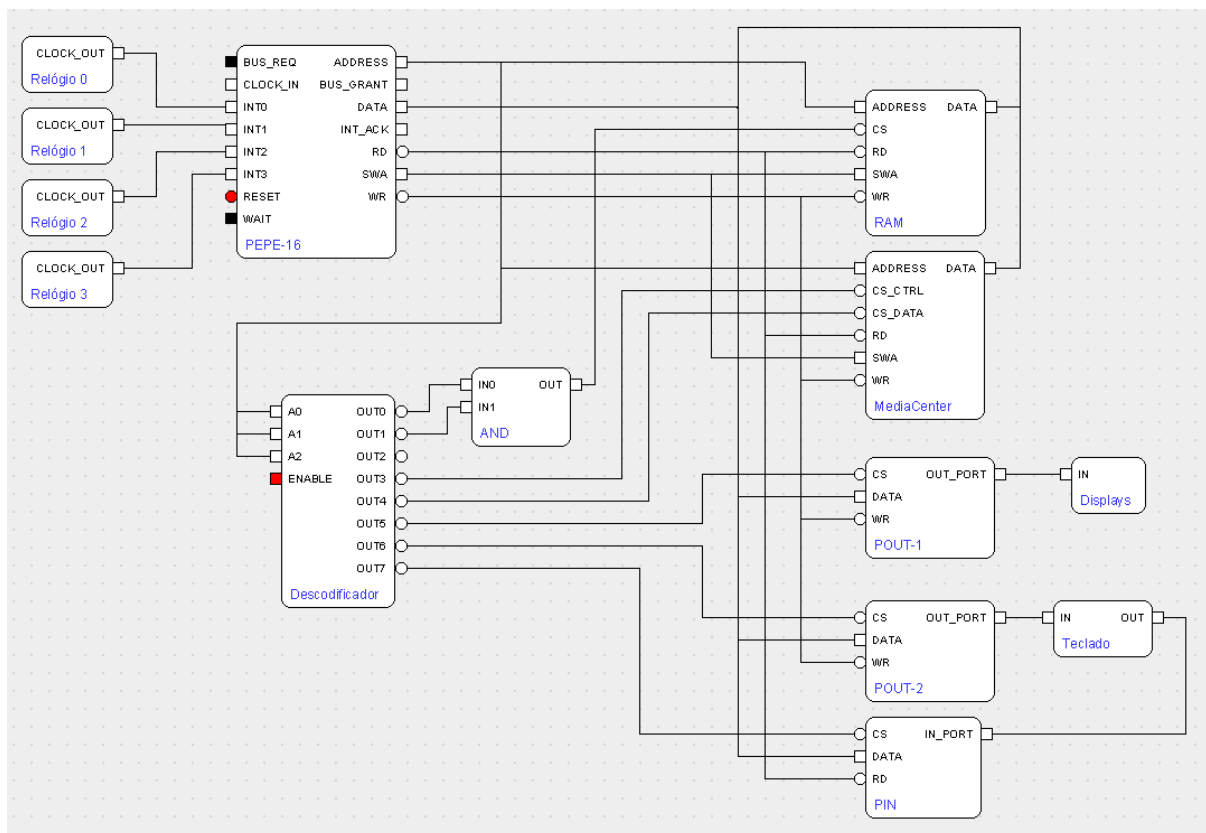
(Semana 8)

#### 1 – Objetivos

Com este trabalho pretende-se que os alunos pratiquem a utilização de programação com processos cooperativos, bem como a sua relação com as interrupções.

#### 2 – O circuito de simulação

Use o circuito contido no ficheiro **lab6.cir**. Este circuito é igual ao que já foi usado no Guião de laboratório 5, com um ecrã, quatro relógios, dois displays e um teclado.



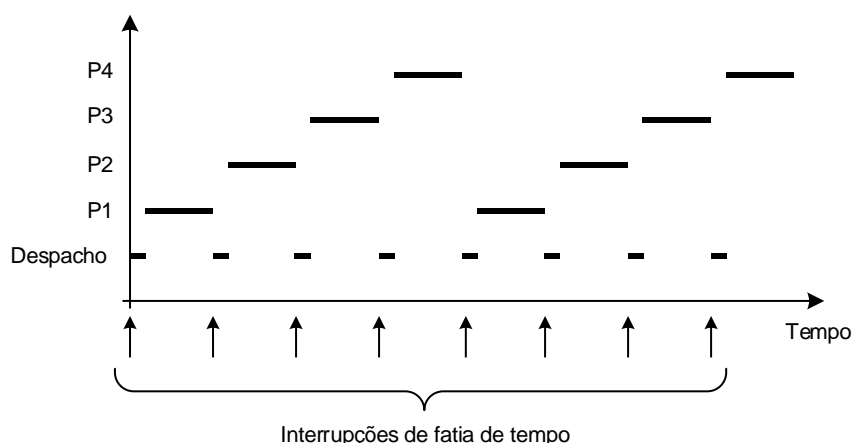
### 3 – Execução do trabalho de laboratório

#### 3.1 – Processos cooperativos

Um processador consegue executar várias atividades, aparentemente de forma simultânea. Cada atividade é designada por processo.

Na realidade, o processador só consegue executar um dado processo em cada instante (a instrução que estiver a executar só pode pertencer a um processo).

Para dar a ideia de vários processos em execução simultânea, cada processo é dividido em pequenos troços de execução e o processador vai circulando pelos vários processos, executando apenas um troço de cada vez. Fazendo esta ronda pelos vários processos de forma muito rápida, macroscopicamente parece que estão todos a ser executados ao mesmo tempo.



Num computador com um sistema operativo (programa que faz a ronda pelos vários processos sem intervenção destes, como por exemplo Windows, MacOS ou Linux), cada processo é programado como se tivesse o processador totalmente só para ele, sem ter em conta que há outros processos que também têm de executar as suas instruções. Para dar oportunidade a todos os processos de serem executados, o processo que estiver a ser executado é interrompido ao fim de algum tempo (fatia de tempo, ou *time-slice*), para passar ao processo seguinte (despacho).

Dado que o PEPE não dispõe de um sistema operativo, cada processo tem de cooperar com os outros, sabendo que tem de executar apenas um bocadinho e retornar para dar oportunidade aos restantes processos para executarem também, à vez.

Nomeadamente, os processos não podem ter ciclos bloqueantes (potencialmente infinitos), pois não permitiriam que os restantes processos fossem também executados. Daí a designação processos cooperativos, pois eles próprios têm de ser bem-comportados e colaborar na execução do conjunto global dos processos.

#### 3.2 – Rotinas de interrupção não são processos

O Guião de laboratório 5 usou o programa *assembly lab5-quatro-barras-displays.asm* (também aqui incluído para facilidade de referência) para ilustrar várias atividades em simultâneo.

O programa principal faz os displays contar e as interrupções permitem animar a descida das quatro barras. Se necessário para relembrar os detalhes, leia a secção 3.6 do Guião de laboratório 5 e execute novamente este programa.

Na realidade, neste exemplo apenas o programa principal pode ser designado por processo. As descidas das barras não passam de atividades esporádicas, que executam apenas como resposta a uma interrupção e não como processos (atividades continuadas).

O mecanismo das interrupções é de muito baixo nível e deve usar-se apenas para gerar temporizações ou lidar diretamente com os periféricos (*device drivers*), nunca para executar atividades continuadas que fazem parte da funcionalidade do programa (e que devem constituir processos).

Por simplicidade, no Guião de laboratório 5 (programa **lab5-quatro-barras-displays.asm**) são as quatro rotinas de interrupção que fazem diretamente descer as quatro barras, mas tal não é uma boa prática.

### 3.3 – Processos e interrupções

A forma correta de implementar um conjunto de processos, com boa prática na sua relação com as interrupções, é a seguinte:

- Um processo é implementado por uma rotina não bloqueante (sem ciclos potencialmente infinitos);
- O programa principal deve conter um ciclo infinito (o único ciclo infinito no programa), que invoque por **CALL** todas as rotinas que correspondem a processos;
- As rotinas de interrupção mais não devem fazer do que assinalar os respetivos eventos, isto é, alterar uma dada variável para um valor que indique que ocorreu uma interrupção;
- O processo que estiver interessado em saber se essa interrupção ocorreu deve repetidamente (sempre que for executado) ler essa variável e, caso detete que houve interrupção, implementar a funcionalidade correspondente (por exemplo, descer a barra), alterando depois a variável para um valor que indique que não houve interrupção (consumindo desta forma o evento de interrupção).

Com o editor de texto abra o programa **lab6-processos-barras-displays.asm**. Este programa tem uma funcionalidade idêntica à do programa **lab5-quatro-barras-displays.asm**, mas agora com uma boa estrutura de processos e interrupções.

Note os seguintes aspetos:

- O ciclo no programa principal invoca 5 processos: um para avançar a contagem nos displays e quatro para fazer descer as barras;
- Os processos que fazem descer as barras são todos iguais, com exceção da coluna em que fazem descer a barra. Por isso, são implementados pela mesma rotina (**anima\_barra**), com **R3** como parâmetro para indicar a coluna em que cada um deve trabalhar;
- Como o número da coluna bate certo com o número da interrupção, é usado como índice para aceder a uma tabela de quatro variáveis (**evento\_int**), onde as rotinas de interrupção assinalam que houve interrupção (cada uma na sua componente). O processo respetivo deteta e consome esse evento;
- As rotinas de interrupção não sabem nada para além da variável onde assinalam que houve interrupção. Estão ao nível certo.

Carregue o programa *assembly* **lab6-processos-barras-displays.asm**, com **Load asm** (📁) ou *drag & drop*.

Abra o ecrã do MediaCenter, os displays de 7 segmentos e os painéis de todos os relógios.

Execute o programa, carregando no botão **Start** (▶) do PEPE. Note que os displays começam logo a contar e os relógios a gerar ciclos nos seus sinais de saída.

A funcionalidade devia ser a mesma que já tinha observado no programa **lab5-quatro-barras-displays.asm** (barras a descer, cada uma ao seu ritmo, 1, 2, 4 e 8 vezes por segundo, e os displays a contar, ao ritmo que a rotina de atraso e o desempenho do seu computador permitem). No entanto, a menos que o seu computador seja mesmo rápido, deverá observar que as barras dos relógios mais rápidos não descem ao ritmo esperado.

Termine o programa, carregando no botão **Stop** (⏏) do PEPE.

Poderá até ficar ainda pior se aumentar a constante **DELAY** para o dobro, por exemplo. Salve o ficheiro, volte a carregá-lo no PEPE, carregando no botão **Reload** (🔄) e execute-o de novo com o botão **Start** (▶).

Experimente agora alterar o valor da constante **DELAY** no programa para um valor bastante mais baixo, por exemplo 200H ou mesmo 100H. Execute de novo e verifique que as barras já funcionam bem, embora os displays mudem agora mais rápido.

**Qual era o problema?** A rotina **anima\_displays**, com a rotina **atraso**, atrasa demasiado o ciclo dos vários processos. O resultado é que as interrupções ocorrem ao ritmo dos relógios, mesmo os mais rápidos, mas as rotinas **anima\_barra** não são chamadas a um ritmo suficientemente rápido para apanhar todas as interrupções, e algumas perdem-se.

**Conclusão:** o tempo de uma volta a todos os processos não pode ser maior do que o período da interrupção mais rápida!

**Solução:** em vez de ter uma rotina **atraso** lenta, fazendo todo o ciclo de uma só vez, deve fazer apenas uma iteração de cada vez que for chamada, e guardar o valor do contador de atraso numa variável. Desta forma, é o ciclo principal do programa que, invocando repetidamente a rotina **anima\_displays**, faz com que o contador de atraso vá evoluindo.

O programa **lab6-processos-barras-displays-OK.asm** implementa esta solução. Carregue-o no PEPE e verifique que agora já pode mudar a constante **DELAY** à vontade, pondo os displays tão lentos quanto queira, sem interferir na velocidade de evolução das barras.

Note ainda que esta constante deve ser bastante mais pequena do que antes, pois agora uma iteração do contador de atraso tem de dar a volta a todos os processos, e portanto demora muito mais tempo do que um simples subtrair de um registo.

### 3.4 – Os processos devem ser cooperativos, não bloqueantes

Mas a situação pode ser pior, pois em vez de uma rotina lenta poderá haver uma mesmo bloqueante. Lembra-se do teclado, objeto de estudo no Guião de laboratório 3?

O programa **lab3.asm**, incluído aqui para facilidade de referência, permitiu detetar uma tecla na 4ª linha do teclado.

Com o editor de texto abra o programa *assembly* **lab6-processos-teclado-bloqueante.asm**. Este programa acrescenta mais um processo (**teclado**) ao programa **lab6-processos-barras-displays-OK.asm**, visto na secção anterior. O código da rotina que implementa este processo é basicamente o programa principal do ficheiro **lab3.asm**, que lê o teclado de forma

bloqueante. Tem ciclos potencialmente infinitos (até o utilizador carregar e libertar uma tecla).

Carregue o programa *assembly* **lab6-processos-teclado-bloqueante.asm**, carregando em **Load asm** (📁) ou *drag & drop*.

Abra o ecrã do MediaCenter, os displays de 7 segmentos, o teclado e os painéis de todos os relógios.

Execute o programa, carregando no botão **Start** (▶) do PEPE.

Vá carregando numa tecla da 4ª linha do teclado e depois largando e verifique que só se consegue que as barras desçam ao ritmo das teclas e não ao ritmo dos relógios!

As interrupções até estão a funcionar (note que os relógios continuam a contar ciclos), mas o ciclo principal é bloqueado quando o processo **teclado** é invocado (porque só retorna quando se carrega e larga uma tecla).

É por esta razão que nenhum processo pode ser bloqueante (ter ciclos potencialmente infinitos no seu interior), pois os restantes não são executados.

Termine o programa, carregando no botão **Stop** (⏏) do PEPE.

O programa **lab6-processos-teclado-OK.asm** resolve o problema e até permite controlar se os displays sobem (nenhuma tecla carregada) ou descem (uma tecla da 4ª linha carregada).

Com o editor de texto abra este programa e verifique que:

- O processo **teclado** já não é bloqueante. Lê as colunas da 4ª linha e coloca 0 ou o valor da coluna da tecla (1, 2, 4 ou 8) na variável **coluna\_carregada**, consoante não haja ou haja, respetivamente, uma tecla carregada. Em seguida, retorna. O ciclo principal dos processos garante que será invocado novamente dentro de pouco tempo, depois de executar os restantes processos;
- O processo **anima\_displays** incrementa os displays se não houver nenhuma tecla carregada. Enquanto carregar numa tecla da última linha, decrementa os displays;
- As variáveis em memória constituem uma boa forma de comunicação entre processos;
- Como o teclado já não é bloqueante, todos os processos correm, haja ou não uma tecla carregada.

Para verificar, carregue o programa *assembly* **lab6-processos-teclado-OK.asm**, carregando em **Load asm** (📁) ou *drag & drop*.

Abra o ecrã, os displays, o teclado e os painéis de todos os relógios.

Execute o programa, carregando no botão **Start** (▶) do PEPE.

Verifique que tudo funciona, e que os displays aumentam ou diminuem o seu valor, consoante não haja ou haja, respetivamente, uma tecla carregada.

Termine o programa, carregando no botão **Stop** (⏏) do PEPE.

### 3.5 – Processos que implementam máquinas de estados

Alguns processos têm de se comportar de forma diferente consoante o estado em que se encontram.

Suponha por exemplo que, em vez de mudar o sentido de evolução dos displays apenas enquanto carrega numa tecla, quer ter as duas situações estáveis (displays a aumentar ou a diminuir) carregando na tecla C ou D, respetivamente, ficando nessa situação após deixar de carregar na tecla.

A figura seguinte ilustra este funcionamento, através de um diagrama de 4 estados:

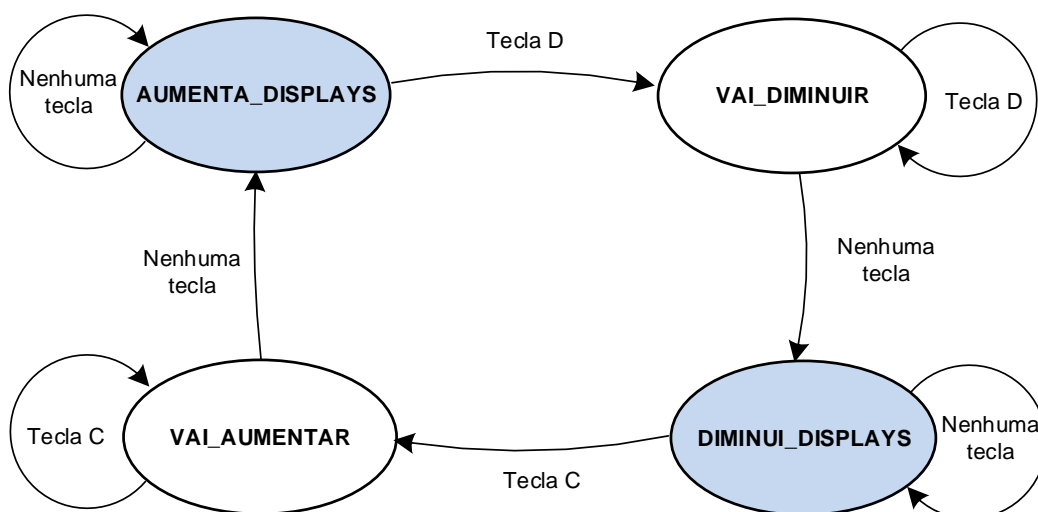
- **AUMENTA\_DISPLAYS** – Estado estável, que se mantém mesmo sem carregar em tecla nenhuma. O valor dos displays vai aumentando;
- **VAI\_DIMINUIR** – Estado transitório, para o qual se transita ao carregar na tecla D, transitando para o estado **DIMINUI\_DISPLAYS** quando se larga a tecla;
- **DIMINUI\_DISPLAYS** – Estado estável, que se mantém mesmo sem carregar em tecla nenhuma. O valor dos displays vai diminuindo;
- **VAI\_AUMENTAR** – Estado transitório, para o qual se transita ao carregar na tecla C, transitando para o estado **AUMENTA\_DISPLAYS** quando se larga a tecla.

A noção de estados de um processo permite-lhe ter memória. Por exemplo, na situação estável de não carregar em tecla nenhuma os displays podem estar a aumentar ou a diminuir, dependendo da evolução anterior dos estados.

Assim, é preciso alterar o processo **anima\_displays** para ter quatro estados.

A ideia base é a rotina **anima\_displays** ter quatro partes, uma que implementa cada estado. Quando a rotina é invocada (pelo ciclo do programa principal), esta verifica a variável **estado\_anima\_displays** para saber em que estado ficou na última vez que executou, e salta diretamente para a parte correspondente, sem executar as outras.

Em cada uma dessas partes, testa se houve alguma tecla carregada ou não, consoante o que fizer mudar o estado. Se for o caso, muda de estado alterando a variável **estado\_anima\_displays** para o valor do novo estado. Cada estado é identificado por uma constante (uma das quatro descritas em cima).



O programa seguinte ilustra o novo processo **anima\_displays** de forma simplificada (as instruções que cada estado executa foram omitidas).

```
        MOV  R0, estado_anima_displays
        MOV  R1, [R0]
aumentar:
        CMP  R1, AUMENTA_DISPLAYS
        JNZ  prepara_diminuir
; ... Executa estado AUMENTA_DISPLAYS

        JMP  sai_anima_displays

prepara_diminuir:
        CMP  R1, VAI_DIMINUIR
        JNZ  diminuir
; ... Executa estado VAI_DIMINUIR

        JMP  sai_anima_displays

diminuir:
        CMP  R1, DIMINUI_DISPLAYS
        JNZ  prepara_aumentar
; ... Executa estado DIMINUI_DISPLAYS

        JMP  sai_anima_displays

prepara_aumentar:
        CMP  R1, VAI_AUMENTAR
        JNZ  sai_anima_displays
; ... Executa estado VAI_AUMENTAR


        JMP  sai_anima_displays
```

O processo começa por ler a variável de estado **estado\_anima\_displays**, para saber em que estado está. Segue-se uma série de comparações, até o estado atual ser igual ao valor de comparação. São as instruções desse estado que o processo executa, e só desse (tem um **JMP** no fim dessas instruções para o fim do processo).


Em cada estado é feito o que deve ser executado (alterar os displays, nomeadamente) e depois a condição de saída (tecla carregada) é testada. Se a tecla carregada for a certa, o estado é mudado para o próximo, caso contrário o estado mantém-se.

Note-se que o processo é não bloqueante e cada estado só é executado uma vez em cada iteração do ciclo do programa principal. A mudança para um novo estado consiste apenas em mudar a variável de estado, **estado\_anima\_displays**, e a rotina do processo retorna logo em seguida. A execução do novo estado será feita apenas na próxima iteração do programa principal.

Com o editor de texto, verifique estas alterações no processo **anima\_displays**, no programa **lab6-processos-estados.asm**. Para facilitar, foi introduzida uma rotina auxiliar, **altera\_displays**, que altera o valor mostrado nos displays.


Carregue o programa *assembly lab6-processos-estados.asm*, carregando em **Load asm** () ou *drag & drop*.

Abra o ecrã, os displays, o teclado e os painéis de todos os relógios.

Execute o programa, carregando no botão **Start** () do PEPE.

Verifique que tudo funciona, e que os displays vão aumentando ou diminuindo o seu valor, consoante se carregue na tecla C ou D, respetivamente. Note que os displays param enquanto se está a carregar numa destas teclas, só recomeçando quando a tecla se larga. É de propósito, para a contagem se realizar numa situação estável (não estar a carregar em nenhuma tecla).

Mesmo quando os displays param (enquanto a tecla está carregada), as barras continuam a descer. O mecanismo dos processos está a funcionar!

Termine o programa, carregando no botão **Stop** () do PEPE.

#### 4 – Considerações em termos do Projeto

Mais do que funcionar, o objetivo do projeto deve ser produzir um programa bem estruturado. Isso implica separação de responsabilidades e de atividades, algo que os processos permitem.

Na sua versão final, pretende-se que o programa do projeto tenha as seguintes características:

- O programa principal deve ter apenas a estrutura de processos (lista de **CALLs**);
- Cada entidade relevante do projeto deve ter um processo dedicado ao seu tratamento. Se houver várias do mesmo tipo, como as barras que diferiam apenas na sua localização, a rotina do processo deve ser a mesma, passando-lhe um parâmetro para distinguir uma entidade das restantes “gémeas”;
- Não deve haver processos demasiado lentos (com ciclos de atraso) nem bloqueantes (nomeadamente, o teclado);
- O teclado e as rotinas de interrupção constituem software de baixo nível e devem saber detalhes das atividades de alto nível, que implementam a funcionalidade do sistema:
  - O teclado mais não deve fazer do que detetar uma tecla e colocar o valor dessa tecla numa variável. Os processos de mais alto nível devem ir ler essa variável e ver se devem reagir a essa tecla;
  - As rotinas de interrupção mais não devem fazer do que assinalar a ocorrência da interrupção respetiva, colocando um dado valor numa dada variável. Um processo que deva reagir a uma interrupção deve ir ler a variável respetiva, que deve depois ser colocada no estado inicial por esse processo, assinalando que já tratou da ocorrência (para não tratar da mesma mais do que uma vez).