

Laboratório de Arquitetura de Computadores

Guião 7

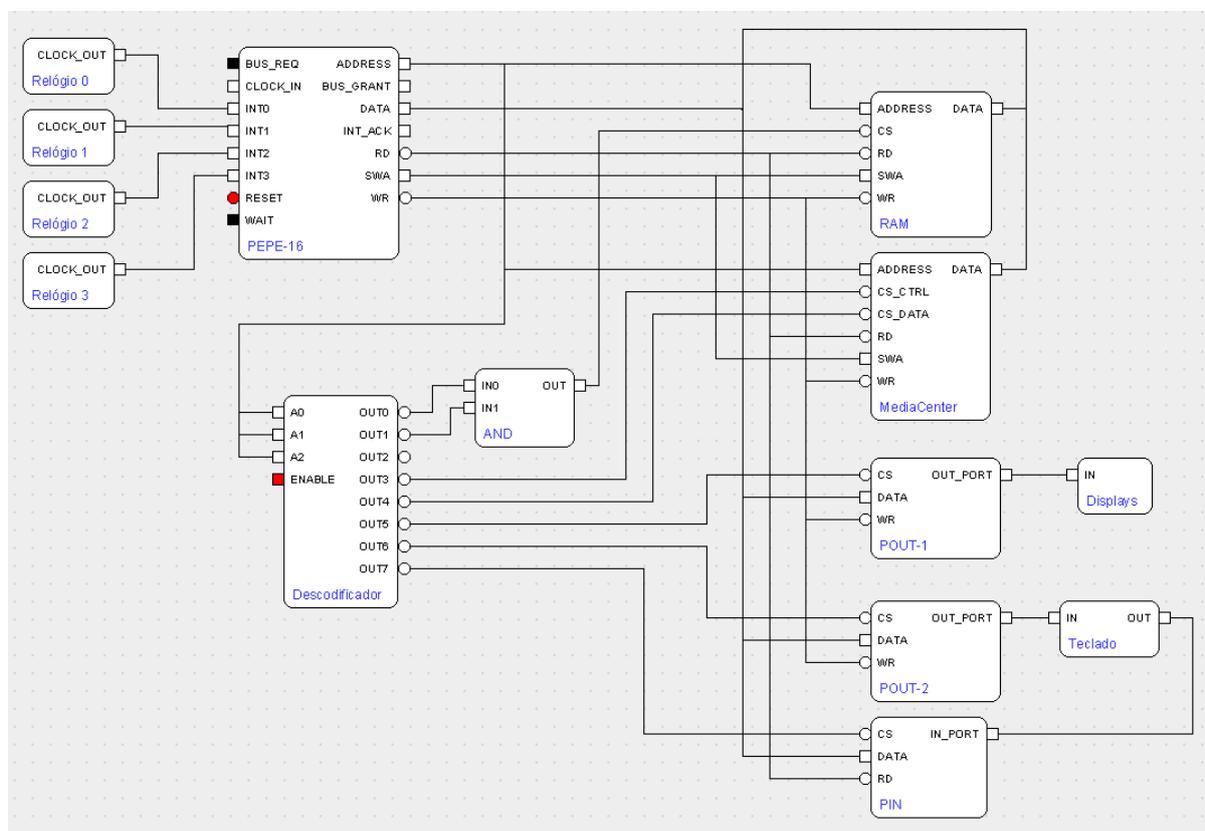
Programação de aplicações de tempo real

1 Objetivos

Com este guião pretende-se que o leitor pratique a utilização de programação de aplicações com múltiplas atividades concorrentes, bem como a sua relação com as interrupções e com periféricos com interação com o utilizador.

2 O circuito de simulação

Use o circuito contido no ficheiro **lab7.cir**. Este circuito é igual ao que já foi usado no Guião de laboratório 6, com um ecrã, quatro relógios, dois displays e um teclado.



3 O problema e possíveis soluções

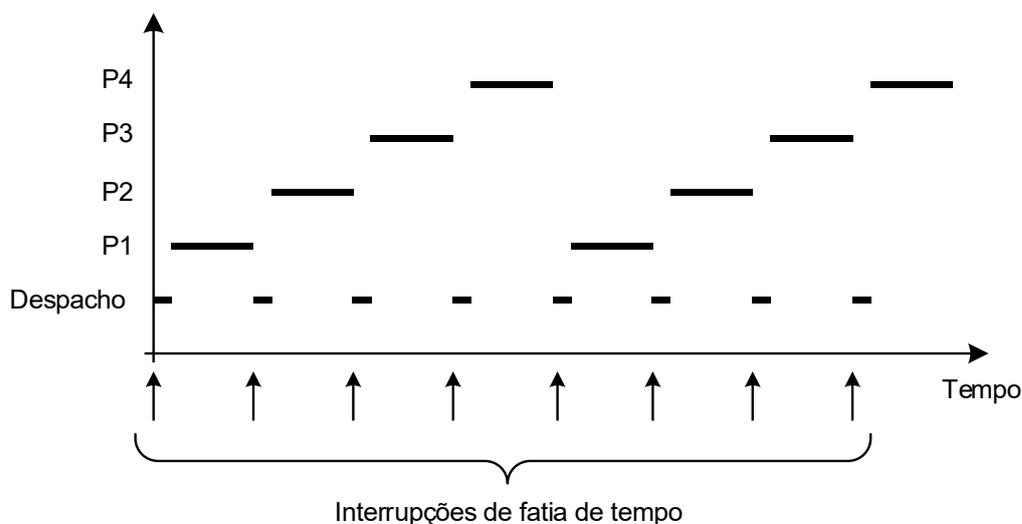
Uma aplicação interativa (um jogo, por exemplo) inclui normalmente várias atividades que se executam, aparentemente, de forma simultânea. Há bonecos e evoluir no ecrã, seja de forma autónoma (temporizada) ou por controlo do utilizador (através de um teclado, por exemplo), displays a contar, etc.

Tudo isto dá vida e interatividade à aplicação, mas como nada sucede por acaso tudo tem de ser previsto e programado. Há atividades que devem executar tão rápido quanto possível, enquanto outras devem obedecer a temporizações específicas, além de que uma não deve interferir com o funcionamento das restantes.

É usual designar por processo cada atividade que executa de forma “simultânea” (concorrente) com as restantes.

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) e não todos realmente em simultâneo.

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 (a parte do sistema operativo que faz isso designa-se Despacho).

O ideal é que cada processo possa ser programado de forma totalmente independente dos restantes (para não ter que se pensar em todas as atividades ao mesmo tempo, o que é demasiado complexo). No entanto, na prática eles precisam de interagir, de alguma forma, para implementar a funcionalidade da aplicação completa.

Como é que se consegue tudo isto? Depende muito do suporte que o sistema oferecer para processos.

Possíveis soluções:

- Sem nenhum suporte para processos:
 1. **Concorrência baseada em interrupções.** O programa principal tem um ciclo infinito, onde trata, em sequência, as diversas atividades sem temporização de tempo real (**CALL** a uma ou mais rotinas), e cada rotina de interrupção implementa cada uma das atividades concorrentes cuja execução deva ocorrer em instantes específicos do tempo (temporização de tempo real);
 2. **Rotinas cooperativas.** O programa principal tem um ciclo infinito, onde trata, em sequência, cada uma de todas as atividades (**CALL** a cada uma das rotinas que implementam essas atividades), com ou sem temporização de tempo real. Cada rotina de interrupção deve apenas assinalar que ocorreu (alterando uma variável em memória, por exemplo), e devem ser as rotinas que implementam as funcionalidades temporizadas a detetar essa alteração e tratar essa ocorrência. Estas rotinas não podem ter ciclos bloqueantes (potencialmente infinitos), pois não permitiriam que as restantes rotinas fossem também executadas. Daí a designação “rotinas cooperativas”, pois elas próprias têm de ser bem-comportadas e colaborar na execução do conjunto global das atividades da aplicação. Quando invocadas, fazem qualquer coisa mas devem retornar logo que possível, sendo de novo invocadas na próxima iteração do ciclo do programa principal;
- Com (algum) suporte para processos:
 3. **Processos cooperativos.** O sistema já permite criar processos, que executam de forma autónoma e independente, e que até já podem ter ciclos bloqueantes (potencialmente infinitos), desde que tenham forma de indicar em que ponto do ciclo é que o sistema pode mudar para outro processo. Ou seja, os processos controlam onde é que podem largar o processador, passando-o a outro processo, mas não se podem esquecer de o indicar. Um ciclo potencialmente infinito que não inclua esta indicação impede os restantes de executar. Por isso, ainda são cooperativos. Tipicamente os processos são criados no programa principal e depois correm de forma autónoma;
 4. **Processos verdadeiros.** Agora já existe um sistema operativo, que muda automaticamente de processo quando chegar ao fim a sua fatia de tempo de execução, o que pode ocorrer em qualquer ponto do conjunto das suas instruções. Um processo não tem de indicar nenhum ponto onde a comutação de processo possa ocorrer e pode ser programado assumindo que tem o processador inteiramente para si.

O simulador suporta apenas as três primeiras soluções, isto é, não tem sistema operativo.

A solução 1 já foi ilustrada através dos exemplos do guião de laboratório 6 e não é uma boa prática, pois coloca mecanismos de baixo nível (interrupções) a executar tarefas de alto nível (atividades da aplicação). 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).

As soluções 2 e 3 são ilustradas por este guião.

4 Rotinas cooperativas (solução 2)

4.1 Rotinas cooperativas e interrupções

Cada rotina que implemente uma atividade concorrente tem de cooperar com as outras, sabendo que tem de executar apenas um bocadinho e retornar logo que possível, para dar oportunidade às restantes para executarem também, à vez.

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

- Uma rotina cooperativa não deve ser bloqueante (não deve ter 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 atividades concorrentes;
- As rotinas de interrupção mais não devem fazer do que assinalar os respetivos eventos (por exemplo, alterando uma dada variável em memória para um valor que indique que ocorreu uma interrupção);
- Uma rotina cooperativa que esteja interessada em saber se essa interrupção ocorreu deve repetidamente (sempre que for executada) ler essa variável e, caso detete que houve interrupção, implementar a funcionalidade correspondente, 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 **lab7-cooperativas-barras-displays.asm**. Este programa tem uma funcionalidade idêntica à do programa **lab6-quatro-barras-displays.asm**, mas agora com uma boa estrutura de rotinas cooperativas e interrupções.

Note os seguintes aspetos:

- O ciclo no programa principal invoca 5 rotinas cooperativas: uma para avançar a contagem nos displays e quatro para fazer descer as barras;
- As rotinas que fazem descer as barras são todas iguais, com exceção da coluna em que fazem descer a barra. Por isso, a mesma rotina (**anima_barra**) é chamada várias vezes, com **R3** como parâmetro para indicar o número da barra, que multiplicado por 8 dá o número da coluna em que cada rotina deve descer a sua barra;
- Como o número da barra 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). A rotina cooperativa respetiva 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.

Passe para “Simulation” e carregue no PEPE-16 o programa *assembly lab7-cooperativas-barras-displays.asm*, carregando em **Load source**  ou com *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-16. 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 **lab6-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-16.

Poderá até ficar ainda pior se aumentar a constante **DELAY** para o dobro, por exemplo. Faça isso, guarde o ficheiro, volte a carregá-lo no PEPE-16, carregando no botão **Reload** (↻) e execute-o de novo com o botão **Start** (▶). Piorou?

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 das várias rotinas cooperativas. 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 todas as rotinas cooperativas 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 **lab7-cooperativas-barras-displays-OK.asm** implementa esta solução. Carregue-o no PEPE-16 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 tem de ser bastante mais pequena do que antes, pois agora uma iteração do contador de atraso tem de dar a volta a todas as rotinas cooperativas, e portanto demora muito mais tempo do que um simples subtrair de um registo.

4.2 As rotinas devem ser cooperativas e não bloqueantes

Mas a situação pode ser ainda 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 lab7-cooperativas-teclado-bloqueante.asm**. Este programa acrescenta mais uma rotina (**teclado**) ao programa **lab7-cooperativas-barras-displays-OK.asm**, visto na secção anterior. O código da rotina 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 lab7-cooperativas-teclado-bloqueante.asm*, carregando em **Load source** (👉) 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-16.

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 a rotina **teclado** é invocada (porque só retorna quando se carrega e larga uma tecla).

É por esta razão que nenhuma rotina cooperativa pode ser bloqueante (ter ciclos potencialmente infinitos no seu interior), pois as restantes não são executadas.

Termine o programa, carregando no botão **Stop** (⏹) do PEPE-16.

O programa **lab7-cooperativas-teclado-OK.asm** resolve o problema e até permite controlar se os displays sobem (nenhuma tecla carregada) ou descem (com uma tecla da 4ª linha carregada).

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

- A rotina **teclado** já não é bloqueante e já pode ser apelidada de cooperativa. Lê as colunas da 4ª linha e coloca (na variável **coluna_carregada**) ou 0 ou o valor da coluna da tecla (1, 2, 4 ou 8), consoante não haja ou haja, respetivamente, uma tecla carregada. Em seguida, retorna. O ciclo no programa principal garante que será invocada novamente dentro de pouco tempo, depois de executar as restantes rotinas;
- A rotina **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 rotinas cooperativas;
- Como o teclado já não é bloqueante, todas as rotinas cooperativas correm, haja ou não uma tecla carregada.

Para verificar, carregue o programa *assembly lab7-cooperativas-teclado-OK.asm*, carregando em **Load source** (👉) 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-16.

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

Termine o programa, carregando no botão **Stop** (⏹) do PEPE-16.

Para as rotinas cooperativas, o truque fundamental é não haver ciclos potencialmente infinitos internos às rotinas. Cada rotina deve executar apenas uma iteração do ciclo que eventualmente for necessário, e regressar. Será invocada de novo na próxima iteração do ciclo do programa principal.

Ou seja, os ciclos potencialmente infinitos devem ser **externos** às rotinas cooperativas, **não internos**.

5 Processos cooperativos (solução 3)

5.1 Vantagens face às rotinas cooperativas

As rotinas cooperativas têm algumas limitações e desvantagens, nomeadamente:

1. Cada rotina tem de saber que está inserida no ciclo do programa principal, pelo que em cada invocação tem de fazer algo mas retornar logo que possível (não pode ter ciclos internos longos ou mesmo potencialmente infinitos);
2. É difícil uma rotina cooperativa manter estado interno entre invocações sucessivas (quando é invocada, entra sempre pelo mesmo sítio);
3. Têm de estar sempre a testar se ocorreram eventos que lhes interessem (tecla carregada ou interrupção), o que consome bastante tempo de processador no computador que está a executar o simulador;
4. Os registos são partilhados com todas as rotinas restantes, pelo que têm de recorrer bastante a variáveis em memória.

O simulador oferece suporte para processos cooperativos (solução 3 da secção 3), não tão elaborado como um sistema operativo, mas com maior suporte para atividades concorrentes do que simples rotinas cooperativas (solução 2 da secção 3).

As vantagens face às rotinas cooperativas podem exprimir-se da seguinte forma:

1. O programa principal cria os processos, mas não tem um ciclo para os invocar. Uma vez criados, executam autonomamente. Não retornam, apenas têm de indicar em que ponto podem suspender a sua execução para passar a outro processo;
2. Quando um processo retoma a execução (depois de dar a vez aos outros), recomeça automaticamente no ponto em que tinha sido suspenso, pelo que um processo pode incluir ciclos longos, potencialmente bloqueantes. Desta forma, é fácil manter o estado interno do processo entre execuções sucessivas (não tem de sair de um ciclo);
3. Cada processo tem uma cópia integral dos 16 registos do processador (o **SP** é inicialmente carregado com o valor indicado na diretiva **PROCESS**). Cada processo tem também uma cópia do **PC**, inicialmente com o valor do endereço da primeira instrução do processo. Isto permite que cada processo tenha todos os registos só para si. Quando um processo deixa de executar para dar vez a outro, todos os seus registos são guardados, sendo recuperados quando voltar a ser a sua vez de executar;
4. Existem variáveis de comunicação entre processos. Quando um processo quiser ser avisado de uma ocorrência (tecla carregada, interrupção, etc.), lê uma dessas variáveis, o que faz bloquear esse processo (deixa de estar executável e não consome tempo do processador que executa o simulador). Quando essa variável for escrita (pelo processo teclado, por uma rotina de interrupção, etc.) os processos bloqueados nessa variável são automaticamente desbloqueados e prosseguem a sua execução.

Estes processos são designados “cooperativos” porque têm de indicar, para cada um dos seus ciclos internos, qual o ponto onde o simulador pode mudar desse processo para outro. Se o programador se esquecer de o fazer, um processo pode bloquear todos os outros.

5.2 Diretivas de suporte aos processos cooperativos

Para o suporte de processos cooperativos, o simulador oferece as seguintes diretivas (apenas no *assembly* do PEPE-16. O PEPE-8 não suporta processos):

- **PROCESS** *valor-inicial-do-SP*. Deve preceder o label da rotina que implementa o processo. Quando se fizer um **CALL** a esta rotina, em vez de invocar realmente a rotina, cria o processo, que depois será executado automaticamente e autonomamente. Esta diretiva precisa que se indique qual o valor com que o **SP** deste processo deve ser inicializado (o que é feito automaticamente, para o programador não se esquecer). Cada processo tem de ter a sua própria pilha, independente das restantes, que deve ser declarada com a diretiva **STACK**, como já tem sido usado no programa principal;
- **YIELD**. Onde esta diretiva aparecer, dentro da rotina que implementa o processo, o simulador pode comutar deste para outro processo. Tipicamente, usa-se dentro de ciclos potencialmente bloqueantes. Desta forma, o processo pode dar a vez a outros sem bloquear o processamento, mesmo com ciclos potencialmente bloqueantes;
- **LOCK** *valor-inicial*. É semelhante a **WORD** (reserva uma variável em memória, de 16 bits), mas uma leitura (com **MOV**) bloqueia o processo que a faz e uma escrita (com **MOV**) desbloqueia todos os processos que nela estiverem bloqueados, que leem o valor lá escrito pelo processo que desbloqueia. O **LOCK** pode assim ser usado para comunicação entre os processos. O *valor-inicial* não é relevante (uma leitura bloqueia), mas pode ser útil para, por inspeção direta da memória, mostrar que o **LOCK** ainda não foi escrito (se nenhum processo escrever o *valor-inicial*).

Destas diretivas, apenas **LOCK** gasta espaço de endereçamento, pois corresponde a uma variável. As restantes são meras indicações para o simulador.

De notar que o programa principal também é um processo, de criação automática (mal se corre o programa), e pode ser usado para implementar uma das atividades da aplicação.

5.3 Exemplo simples de uso de processos cooperativos

O programa **lab7-processos-boneco-displays-teclado.asm** ilustra o funcionamento dos processos e a sua interação com interrupções e periféricos (teclado).

O movimento de um boneco no ecrã é temporizado por uma interrupção e os displays sobem ou descem pelo meio do teclado. Há 3 processos:

- **Programa principal**, que trata dos displays, verificando que tecla foi carregada. Se for **C** incrementa os displays de uma unidade, se for **D** decrementa;
- **Teclado**, que deteta uma tecla na 4ª linha do teclado;
- **Boneco**, que trata do movimento do boneco no ecrã sempre que há uma interrupção.

Para verificar o funcionamento, carregue o programa **lab7-processos-boneco-displays-teclado.asm**, carregando em **Load source** (🟢) ou *drag & drop*.

Abra o ecrã, os displays, o teclado e o painel do relógio 0. Execute o programa, carregando no botão **Start** (▶) do PEPE-16.

Verifique que o boneco se move (ao ritmo do Relógio 0) e que os displays aumentam ou diminuem o seu valor, quando a tecla **C** ou **D**, respetivamente, é carregada.

Termine o programa, carregando no botão **Stop** (■) do PEPE-16.

Notas importantes sobre este programa, que deve verificar:

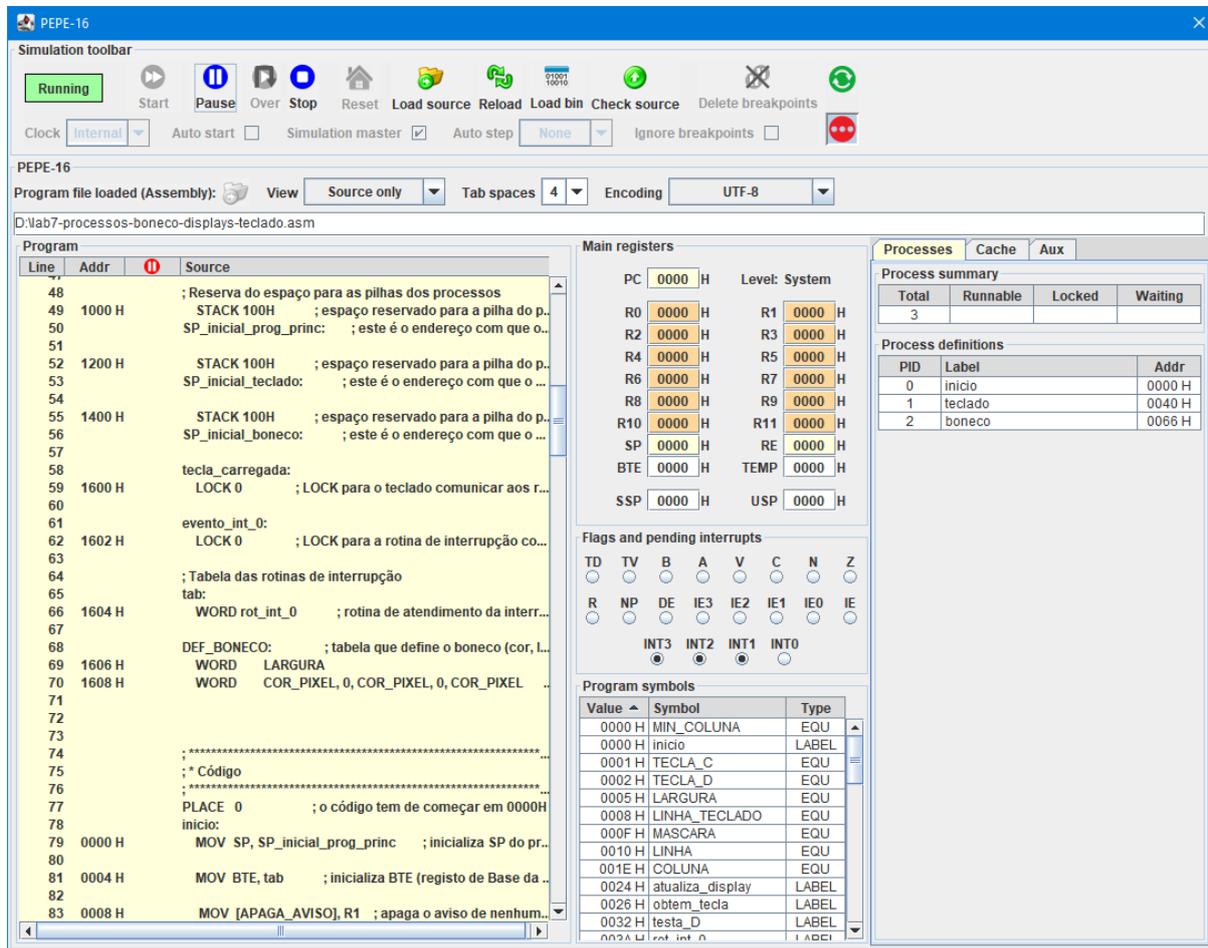
- Declaração dos três **STACKs**, um para cada processo, incluindo o do programa principal;
- Declaração dos dois **LOCKs**, `tecla_carregada` e `evento_int_0`;
- A criação dos dois processos além do programa principal (com **CALL boneco** e **CALL teclado**). Estes **CALLs** não invocam as rotinas. Criam processos que depois executam de forma autónoma, sem necessidade de os invocar;
- A rotina de interrupção, que apenas escreve no **LOCK evento_int_0**, que o processo **boneco** depois usa para saber quando mover o boneco;
- A diretiva **PROCESS**, antes do **boneco** e do **teclado**, que declara os processos respetivos. Esta diretiva inclui o endereço de inicialização do **SP**;
- O facto de nenhum dos três processos terminar (com **RET**). Ao contrário das rotinas cooperativas, que tinham sempre de retornar, os processos cooperativos até são geralmente constituídos por ciclos infinitos, internos ao processo. Se se quiser que um processo termine, a dada altura, basta deixá-lo executar um **RET**. Não retorna, pois também não foi invocado. Em vez disso, termina;
- O facto de não haver nem **PUSHs** nem **POP**s nos processos **boneco** e **teclado**. São “rotinas” de topo, isto é, não foram invocadas como rotinas normais e têm uma cópia exclusiva dos registos do processador. Logo, não precisam de guardar os registos;
- Os dois **YIELDS** no processo teclado, um em cada ciclo bloqueante. Isto é fundamental, para que, caso haja outros processos, o simulador possa comutar de processo nos pontos onde os **YIELDS** aparecem, de modo a que os ciclos bloqueantes não bloqueiem realmente os restantes processos. Depois de dar a volta pelos outros processos, o teclado será executado novamente, recomeçando onde interrompeu;
- Os processos do programa principal e o **boneco** têm ciclos internos infinitos, mas não têm **YIELDS**. Não precisam, pois leem **LOCKs**. Quando um processo lê um **LOCK** e se bloqueia, o simulador muda para outro processo. Logo, estes ciclos são infinitos, mas não são bloqueantes.

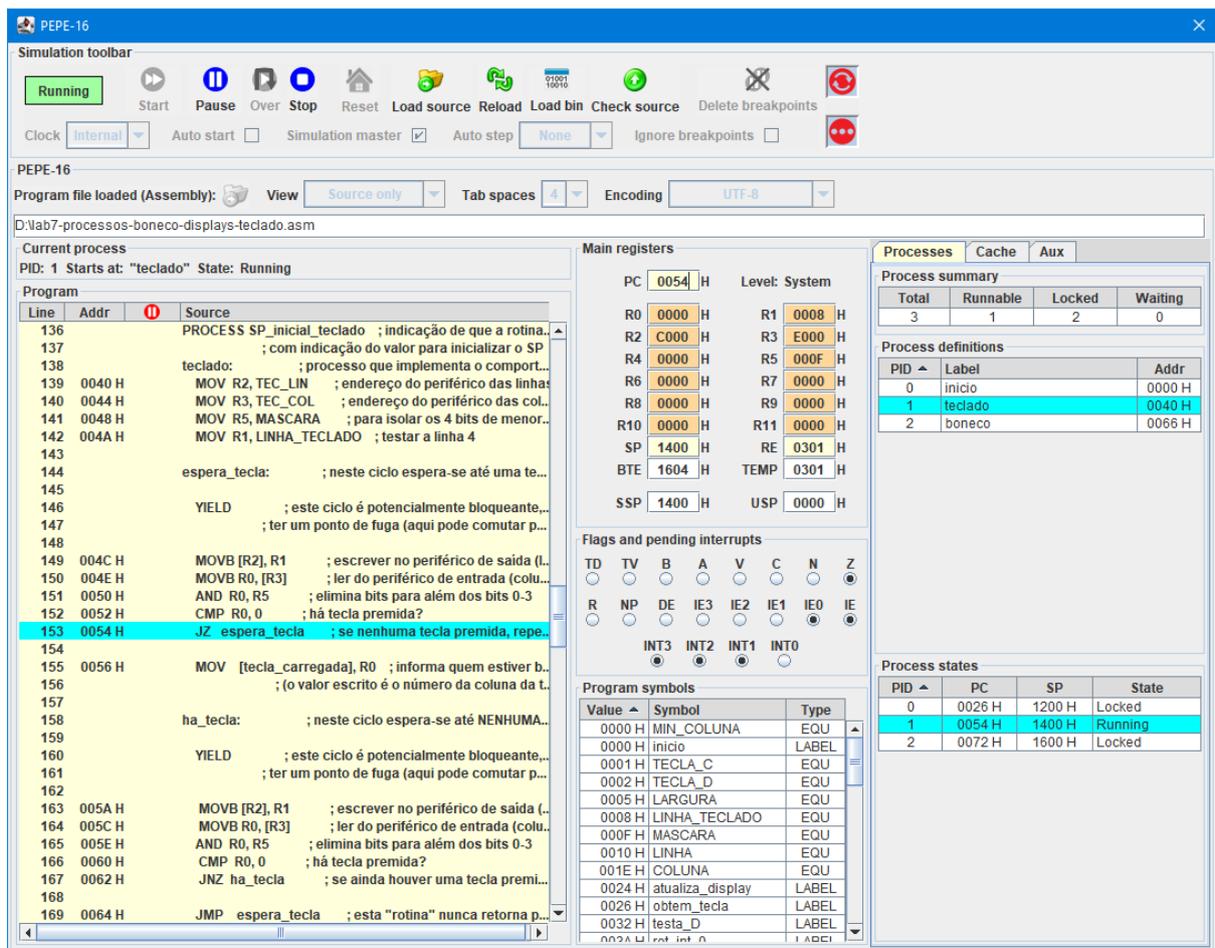
É possível visualizar informação sobre os processos existentes e o respetivo estado, se carregar no botão **More** (⋯) da interface do PEPE-16. Abre-se um painel adicional com indicação do PID (process ID, ou número do processo) e início (*label* e endereço) de cada processo, para além do número total de processos criados.

Se carregar no botão **Refresh** (↻), aparece também informação sobre o estado atual dos processos (atualizado de segundo a segundo). Mesmo sem o painel adicional aberto, com o

refrescamento ativo aparece informação sobre o processo corrente por cima do código do programa.

As figuras seguintes ilustram estes aspetos.





5.4 Teclado em modo “contínuo” versus “tecla a tecla”

O programa **lab7-processos-boneco-displays-teclado.asm** usa o teclado em modo de “tecla a tecla”. Quando se carrega numa tecla executa algo (aumentar ou diminuir o display) e, para executar de novo, tem de se largar a tecla e carregar de novo.

Isto é consequência dos dois ciclos, um para esperar que se carregue numa tecla e outro para que se liberte a tecla.

E se quisermos combinar este funcionamento com o modo “contínuo”, em que fica a executar algo repetidamente enquanto uma dada tecla está carregada?

Com os **LOCKS**, é fácil. O programa **lab7-processos-boneco-teclado-contínuo.asm** é quase igual ao anterior, mas com a diferença de que agora o boneco move-se continuamente enquanto se carrega na tecla **E**, à velocidade máxima que o simulador permite (em vez de ser temporizado pela interrupção). Os displays ainda funcionam da mesma forma, tecla a tecla.

Carregue e execute este programa e verifique os dois modos de funcionamento do teclado. Note os dois **LOCKS** usados pelo teclado para reportar que uma tecla foi carregada, um para informar o processo do programa principal (teclas **C** e **D**, modo “tecla a tecla”) e outro para informar o processo **boneco**, em modo “contínuo”.

Note ainda que o boneco se move demasiado depressa. Para reduzir o ritmo, em vez de inserir um atraso que atrasaria não só o boneco como todos os processos, será melhor introduzir um contador no processo **boneco**. De cada vez que desbloqueia a leitura do **LOCK tecla_contínuo**,

incrementa um contador (num registo, por exemplo). Quando esse registo chegar a um dado valor (ou começar num valor, e ir diminuindo até chegar a 0), então move-se o boneco (e faz-se reset ao contador).

5.5 Múltiplas instâncias de um processo

E se houver vários processos idênticos, com o mesmo comportamento? Podem-se criar vários processos com base no mesmo código? Isto é, várias instâncias do mesmo processo?

Sim, tal como ilustrado pelo programa **lab7-processos-quatro-bonecos-displays-teclado.asm**. Este programa é uma extensão do programa **lab7-processos-boneco-displays-teclado.asm**, mas inclui agora quatro instâncias do processo **boneco** (mas cada uma move o boneco numa linha diferente), com o movimento dos quatro bonecos controlado pelas quatro interrupções do PEPE-16, mantendo o funcionamento do teclado e dos displays.

Edite o ficheiro deste programa e note os seguintes aspetos principais:

- O processo **boneco** recebe o **R11** como argumento, quando é criado, com o número da respetiva instância (0 a 3, para que as várias instâncias se possam distinguir);
- Na realidade, a criação de um processo faz uma cópia dos valores de todos os registos nessa altura (logo, qualquer registo pode ser considerado argumento), e o processo criado passa a ter uma cópia privada (não partilhada com nenhum outro processo) de todos os registos. É como se tivesse o processador todo só para si!
- Tudo o que diz respeito ao processo **boneco** está definido em tabelas com 4 elementos, um por cada instância. Assim, o processo **boneco** deve indexar essas tabelas usando o seu número de instância (**R11**), desta forma usando as estruturas de dados que lhe são específicas;
- Como cada instância tem os registos todos só para si, as tabelas estão a ser usadas apenas para obter valores iniciais, reservando-se registos específicos para manter os valores da linha, coluna, sentido de movimento, etc., ao longo de toda a vida do processo. Se se quisesse manter estes valores em memória, era apenas questão de aceder a tabelas da mesma forma (indexadas pelo número da instância), mas agora em escrita;
- Cada instância de **boneco** tem um **LOCK** específico, para receber as notificações da rotina de interrupção respetiva. Logo, estes **LOCKS** estão também organizados numa tabela de 4 elementos (**evento_int_bonecos**). De igual forma, cada rotina de interrupção acede também ao **LOCK** respetivo, na tabela;
- A diretiva **PROCESS**, que define **boneco** como um processo, tem de receber um valor inicial do **SP** (**SP_inicial_boneco**). Como este valor não pode ser igual para todas as instâncias (cada uma tem de ter a sua própria pilha), então deve reinicializar-se o **SP**, logo no início do código do processo (todas as instâncias o fazem). Declara-se uma zona de pilha suficiente para todas as instâncias do processo (com **STACK TAMANHO_PILHA * N_BONECOS**), que termina em **SP_inicial_boneco**, e em cada instância ajusta-se o **SP**, subtraindo ao valor inicial o n.º da instância vezes o tamanho da pilha de cada instância: **SP_inicial_boneco - (n.º instância * TAMANHO_PILHA)**. Desta forma, a instância 0 fica com o **SP** igual, mas as restantes vão ficando sucessivamente atrás, usando toda a zona declarada para as pilhas das instâncias do processo **boneco**.

Carregue e execute este programa e verifique que:

- Todos os bonecos (cada um é uma instância do processo **boneco**) têm o mesmo comportamento, mas dados diferentes (linhas, colunas, sentidos e temporizações) diferentes. Até podiam ter aspeto diferente, se houvesse uma tabela com os endereços das definições (cor e forma) de cada boneco;
- O teclado e os displays funcionam como dantes, sem alterações.

A figura seguinte ilustra as definições e o estado dos processos.

Note que todas as instâncias do processo boneco começam no mesmo endereço e têm o mesmo *label*, uma vez que o código é exatamente o mesmo. Por esse motivo, todas as que estão bloqueadas (*Locked*) têm o mesmo *Program Counter* (PC), que é onde leem a variável **LOCK** respetiva. Mas repare que os valores do *Stack Pointer* (SP) são diferentes.

IMPORTANTE – Note que todas as instâncias do boneco usam alegremente os “mesmos” registos, sem se preocupar com as restantes instâncias. Na realidade, cada uma delas tem a sua própria cópia independente dos registos do processador, pelo que não interferem umas com as outras.

Esta é uma das maiores vantagens do mecanismo dos processos, reduzindo muito a necessidade de guardar variáveis em memória (mais trabalhosas de aceder do que se estiverem sempre em registos).

The screenshot shows the PEPE-16 debugger interface. The main window displays the assembly code for the program 'PEPE-16'. The code is organized into sections: 'Program', 'Main registers', 'Flags and pending interrupts', and 'Program symbols'. The 'Program' section shows assembly instructions with their addresses and sources. The 'Main registers' section shows the current values of registers R0 through R11, SP, BTE, and SSP. The 'Flags and pending interrupts' section shows the status of various flags and interrupt lines. The 'Program symbols' section shows the symbols defined in the program, including labels and constants. On the right side, there are two panels: 'Processes' and 'Process definitions'. The 'Processes' panel shows a summary of the running processes, including their PID, label, PC, SP, and state. The 'Process definitions' panel shows the definitions for the processes, including their PID, label, and address.

Line	Addr	Source
309		*****
310		; APAGA_BONECO - Apaga um boneco na linha e coluna i...
311		; com a forma definida na tabela indicada.
312		; Argumentos: R1 - linha
313		; R2 - coluna
314		; R4 - tabela que define o boneco
315		*****
316		apaga_boneco:
317		apaga_boneco:
318	00C2 H	PUSH R2
319	00C4 H	PUSH R3
320	00C6 H	PUSH R4
321	00C8 H	PUSH R5
322	00CA H	MOV R5, [R4] ; obtém a largura do boneco
323	00CC H	ADD R4, 2 ; endereço da cor do 1º pixel (2 porqu...
324		apaga_pixels: ; desenha os pixels do boneco a pa...
325	00CE H	MOV R3, 0 ; cor para apagar o próximo pixel do ...
326	00D0 H	CALL escreve_pixel ; escreve cada pixel do boneco
327	00D2 H	ADD R4, 2 ; endereço da cor do próximo pixel (2...
328	00D4 H	ADD R2, 1 ; próxima coluna
329	00D6 H	SUB R5, 1 ; menos uma coluna para tratar
330	00D8 H	JNZ apaga_pixels ; continua até percorrer toda a la...
331	00DA H	POP R5
332	00DC H	POP R4
333	00DE H	POP R3
334	00E0 H	POP R2
335	00E2 H	RET
336		*****
337		*****
338		*****
339		; ESCRIVE_PIXEL - Escreve um pixel na linha e coluna ind...
340		; Argumentos: R1 - linha
341		; R2 - coluna
342		; R3 - cor do pixel (em formato ARGB de 16 bits)

PC	Value	Symbol	Type
R0	0000 H		
R1	0014 H		
R2	0038 H		
R3	FF00 H		
R4	1C02 H		
R5	0005 H		
R6	0005 H		
R7	0001 H		
R8	0000 H		
R9	1C24 H		
R10	0004 H		
R11	0002 H		
SP	19F4 H		
RE	1F00 H		
BTE	1C1C H		
TEMP	000C H		
SSP	19F4 H		
USP	0000 H		

PID	Label	Addr
0	inicio	0000 H
1	teclado	0048 H
2	boneco	0070 H
3	boneco	0070 H
4	boneco	0070 H
5	boneco	0070 H

PID	PC	SP	State
0	0034 H	1200 H	Locked
1	0052 H	1400 H	Runnable
2	0094 H	1900 H	Locked
3	00EA H	19F4 H	Running
4	0094 H	1B00 H	Locked
5	0094 H	1C00 H	Locked

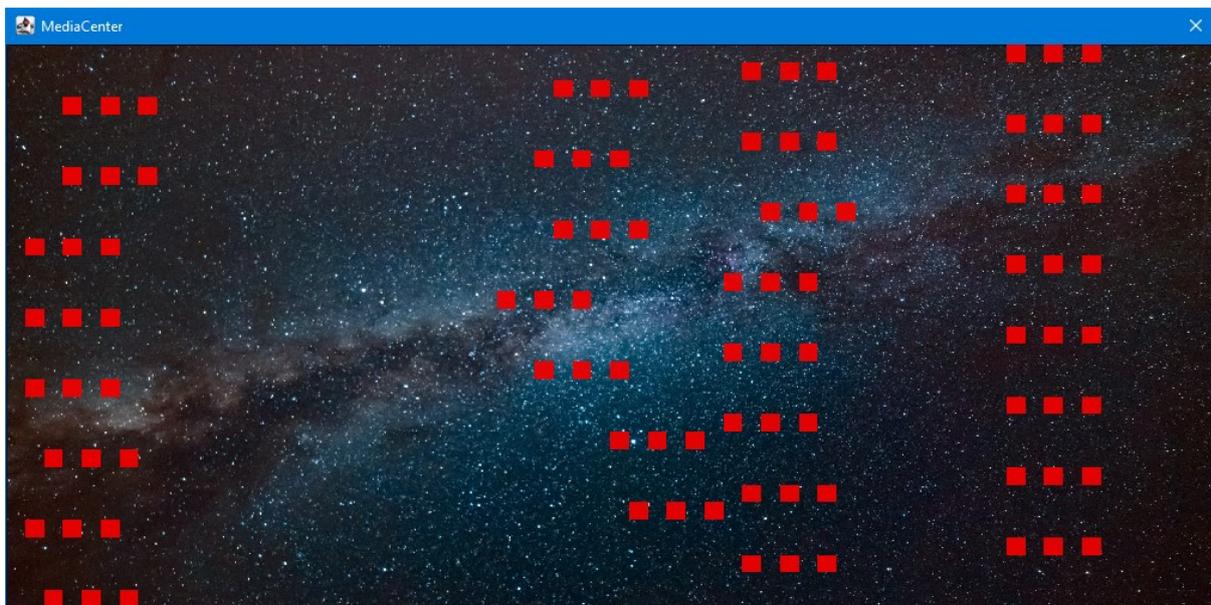
Mesmo assim, e embora o número de instâncias esteja definido por uma constante (**N_BONECOS**), as tabelas **linha_boneco** e **sentido_movimento**, que definem a linha e o sentido de movimento inicial de cada boneco, estão feitas para 4 bonecos, de forma rígida. Se quisermos ser mais flexíveis, estes valores têm de ser definidos mais por uma fórmula do que por uma tabela.

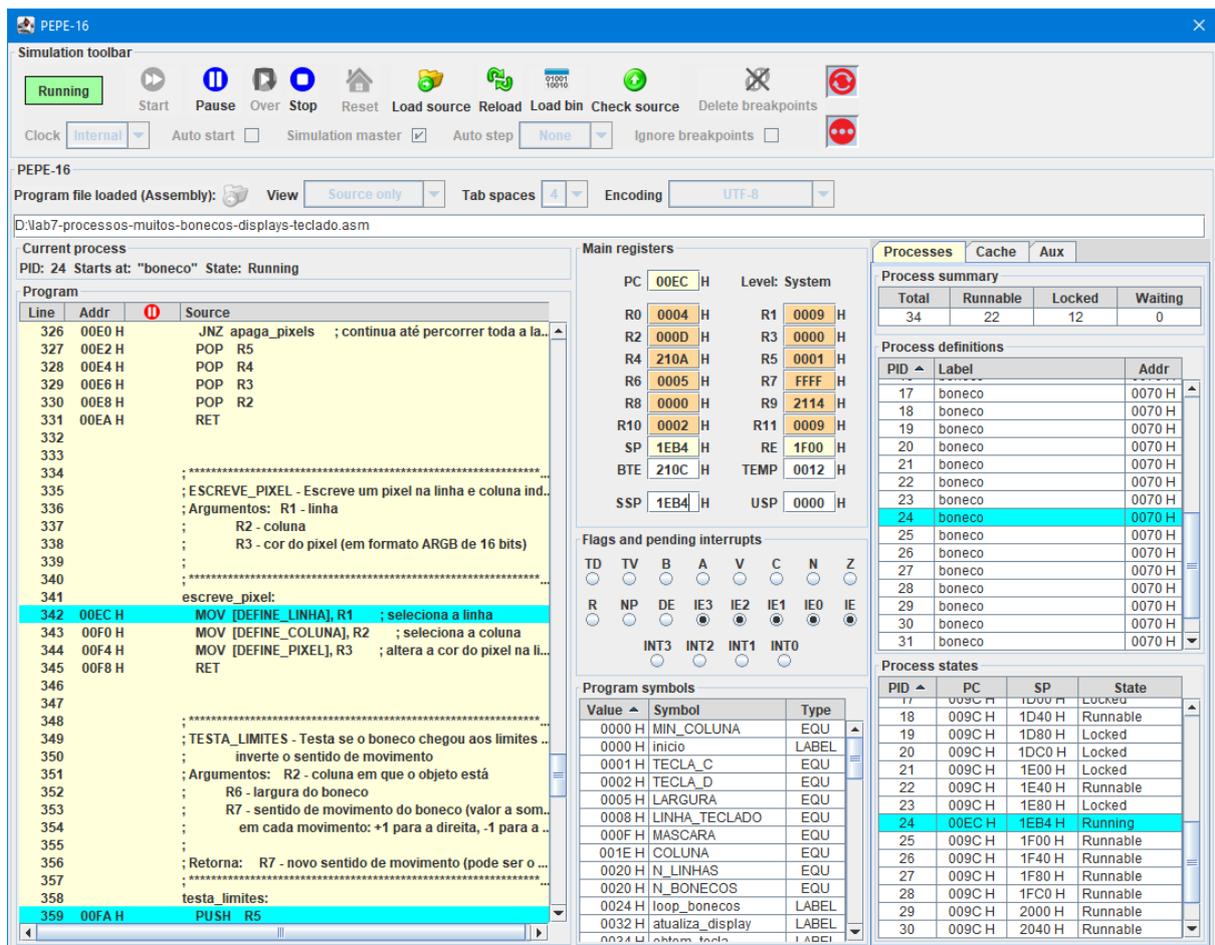
O programa **lab7-processos-N-bonecos-displays-teclado.asm** ilustra esta situação. Permite qualquer número N de instâncias (entre 1 e 32, o número de linhas do ecrã), em que a linha inicial de cada boneco é calculada de forma a distribuí-los pelas linhas e o sentido inicial de movimento é para a direita se o número da instância for par ou esquerda se for ímpar. Mais bonecos seriam possíveis, se se aumentasse o número de linhas no ecrã do MediaCenter.

Carregue e execute o programa é uma extensão do programa **lab7-processos-N-bonecos-displays-teclado.asm** e verifique o que são 32 bonecos em atividade simultânea no ecrã!

Note que, de 4 em 4, as instâncias respondem à mesma interrupção, pois só há 4 interrupções disponíveis. Note também que as instâncias ligadas aos relógios mais rápidos estão algo desfasadas, devido ao atraso provocado pelo tempo gasto no desenho dos muitos bonecos.

As figuras seguintes ilustram este exemplo.





5.6 Cuidados a ter com interrupções com processos cooperativos

Se uma interrupção ocorrer, a rotina respetiva usa a pilha do processo que estiver a executar na altura. Para assegurar a consistência do sistema de processos, é importante que a diretiva **YIELD** não seja usada nas rotinas de interrupção (até porque uma rotina de interrupção não é um processo!).

IMPORTANTE – Dado que cada processo fica com uma cópia integral dos registos, com os valores que esses registos tiverem na altura em que o processo é criado (incluindo o registo de estado), convém permitir as interrupções antes de criar os processos, senão só aqueles que as permitirem explicitamente é que poderão ser interrompidos (o que até poderá ser uma vantagem em certas aplicações).

5.7 Otimização do varrimento do teclado (diretiva **WAIT**)

Carregue e execute de novo programa **lab7-processos-boneco-displays-teclado.asm**.

No seu computador, veja a percentagem de tempo do computador que este programa ocupa (processo Java, no gestor de tarefas; a designação exata depende do sistema operativo).

Esta percentagem pode ser tão alta como 30% (ou mais)!

Tal deve-se ao facto de o PEPE-16 ter de estar continuamente a varrer o teclado, pois não sabe quando é que o utilizador vai carregar numa tecla. Isto é intensivo!

O simulador permite resolver este problema. Carregue e execute o programa contido no ficheiro **lab7-processos-wait-teclado.asm**, que é idêntico ao do ficheiro **lab7-processos-boneco-displays-teclado.asm**, apenas com a diferença de que a primeira diretiva **YIELD** no processo teclado (no ciclo que espera que uma tecla seja carregada, a seguir ao label **espera_tecla**) é substituída pela diretiva **WAIT**.

Esta diretiva faz o processador adormecer quando todos os restantes processos estão bloqueados (em **LOCKS**) e acordar automaticamente quando há um evento relevante (por exemplo, carregar numa tecla do teclado ou uma interrupção que esteja permitida).

Desta forma, não gasta tempo do computador que corre o simulador, exceto quando sucede algo que precisa da atenção do PEPE-16.

NOTA - Não adianta substituir o segundo **YIELD** por **WAIT**, pois nessa altura o **WAIT** comporta-se como **YIELD** (não pode adormecer o processador porque o utilizador está a carregar numa tecla).

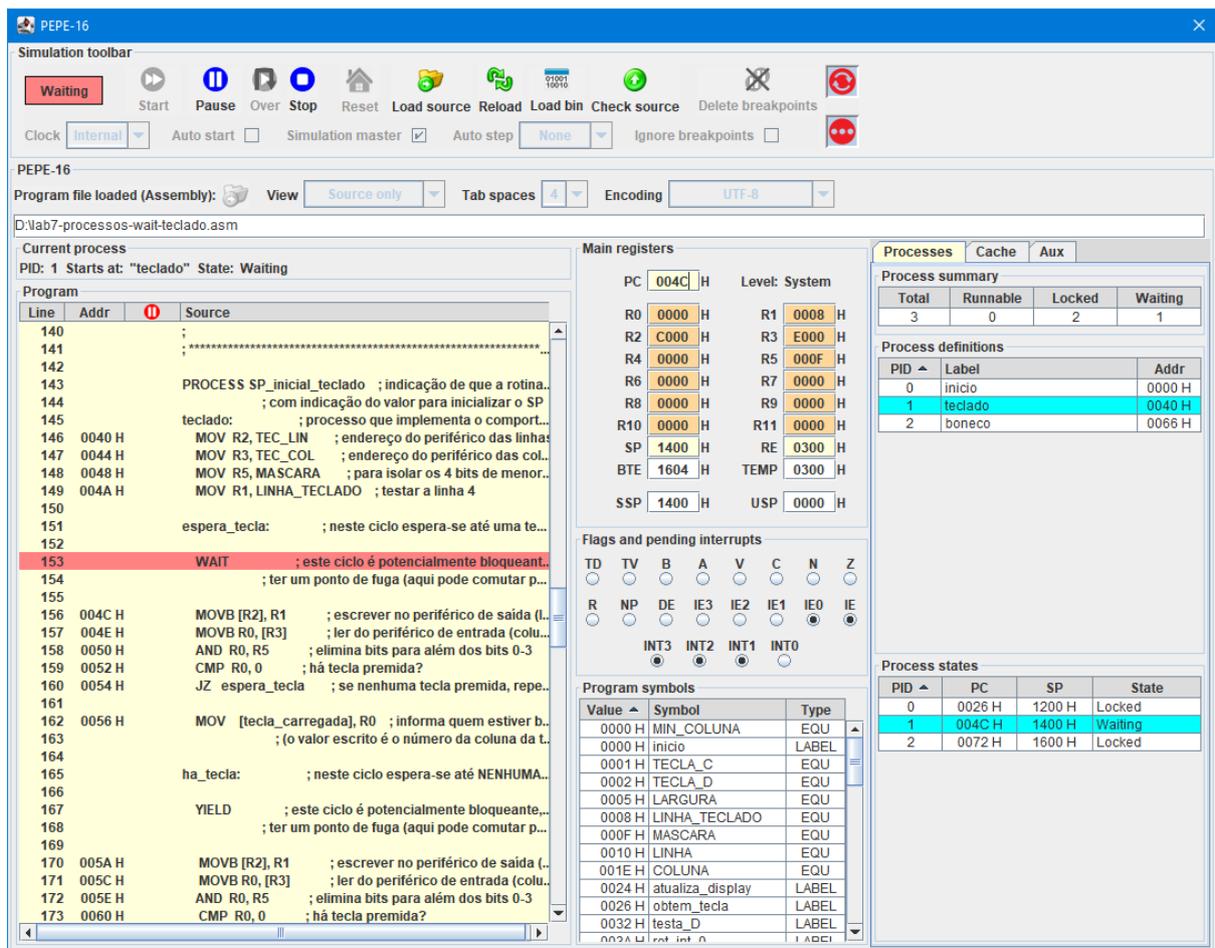
Carregue e execute o programa **lab7-processos-wait-teclado.asm** e verifique que:

- O estado de simulação do PEPE-16, que normalmente é verde (**Running**), está agora normalmente a vermelho (**Waiting**).
- O estado do PEPE-16 passa brevemente por **Running**, de forma periódica, ao ritmo das interrupções que animam o boneco. As interrupções também são acontecimentos relevantes que acordam o processador, para que sejam tratadas (desde que estejam permitidas);
- Se carregar (sem largar) na tecla **C** ou **D**, o estado passa a **Running** e o contador sobe ou desce por cada toque na tecla;
- Se largar a tecla, o estado volta a normalmente **Waiting**;
- A percentagem do processo de Java no computador é agora muito mais reduzida, e se parar todos os relógios nem carregar em nenhuma tecla reduz-se mesmo a zero!

A diretiva **WAIT** é apenas uma otimização, sem impacto na funcionalidade.

A figura seguinte ilustra este caso, em que se pode verificar que não há nenhum processo *Runnable*. O programa principal e o boneco estão *Locked* (à espera que a variável **LOCK** respetiva seja escrita, uma pelo processo teclado e a outra pela rotina de interrupção).

Só quando uma tecla é carregada ou uma interrupção ocorre é que o processador acorda e sai brevemente de **Waiting**, passando por **Running**, altura em que o processo teclado e (se for o caso) um dos outros correm mais uma iteração até se bloquearem de novo.



6 Outros aspetos relevantes para uma aplicação completa

6.1 Start, pause, stop

Estas são funcionalidades que tipicamente devem existir, em particular em aplicações com sessões de execução, como por exemplo um jogo.

Tipicamente, são usadas 3 teclas do teclado para este efeito, em modo de “tecla a tecla” (só executa o comando uma vez por cada toque na tecla).

O programa em si está sempre a funcionar mal se executa. Tem 3 modos (ativo, em pausa e parado), correspondentes às 3 teclas. Uma variável (registo ou memória) pode ser usada para indicar qual o modo atual. A mesma tecla é tipicamente usada para colocar em pausa e recomeçar, mudando de ativo para pausa e vice-versa.

Soluções típicas (sugestões):

- Rotinas cooperativas. O ciclo principal executa sempre (em qualquer modo) a rotina do teclado e a que implementa este controlo (*start, pause, stop*), para que se possa mudar o modo em qualquer altura. A meio do ciclo testa-se qual o modo atual, e só se for ativo é que se invocam as restantes rotinas cooperativas, caso contrário volta-se logo ao início do ciclo;
- Processos cooperativos. No simulador não há forma de suspender ou parar um dado processo, pelo que tem de ser este a testar qual o modo atual da aplicação, pelo que este

modo deve ser guardado numa variável global que todos os processos possam ler. Se o modo atual não for ativo, por exemplo, saem do seu ciclo de processamento normal e podem bloquear-se num **LOCK**, que o processo de controlo da aplicação (o que controla as teclas de *start*, *pause* e *stop*) escreverá quando se voltar ao modo ativo. Desta forma, não têm de estar sempre a testar se o modo já é o ativo.

6.2 Mensagens no ecrã

Há alturas em que se quer exibir algo no ecrã, sobreposto ao que já lá está, sem esconder tudo. Um exemplo típico é colocar a mensagem “Paused” sobreposta ao ecrã da aplicação quando esta é colocada em pausa.

Para isto, o MediaCenter tem o comando número 35 (endereço 6046H, no circuito deste guião), que seleciona uma imagem como cenário frontal. Este é mostrado à frente dos painéis onde são desenhados os bonecos, pixel a pixel. Se for uma imagem opaca, esconde tudo.

O truque é definir uma imagem com o formato do ecrã do MediaCenter (2:1), com algo (texto ou um ícone) num retângulo com fundo transparente. Isto pode ser feito no PowerPoint, por exemplo, selecionando a cor do retângulo como cor transparente e guardando o ficheiro como imagem PNG (que suporta transparência).

6.3 Deteção de colisões

Uma aplicação como um jogo pode ter necessidade de detetar colisões entre bonecos.

Por simplicidade, recomenda-se uma solução segundo as ideias seguintes:

- Deteta-se colisão entre os dois retângulos que envolvem os bonecos (e não pixel a pixel, por exemplo). Não respeita a forma precisa do boneco, mas é simples e eficaz;
- Testam-se apenas as coordenadas de dois vértices diametralmente opostos em cada boneco (por exemplo, canto superior esquerdo e inferior direito);
- Testam-se as condições em que não pode haver colisão. Por exemplo, se a coluna do canto superior esquerdo do boneco B está para a direita do canto inferior direito do boneco A, não pode haver colisão, e assim sucessivamente para as restantes situações;
- Se os dois objetos em teste estiverem em alguma das situações de não-colisão, então não há colisão. Se não estiverem em nenhuma destas situações, há colisão;
- Tanto faz ser o boneco A a testar se colidiu com o B ou vice-versa. No caso de haver várias instâncias de um mesmo boneco A que possam colidir com outro B que só tem uma instância, por exemplo, será melhor cada instância do boneco A a testar se colidiu com o objeto B, pois assim cada boneco só tem de se preocupar com outro (e todos os A têm o mesmo comportamento, pois são instâncias do mesmo processo), em vez de ser o boneco B a ter de saber quantos bonecos A há e a ter de fazer um ciclo para testar se colidiu com cada um dos bonecos A.

6.4 Escolhas pseudo-aleatórias

Para aumentar a interatividade e dinamismo da aplicação, é usual esta incluir escolhas e comportamentos pseudo-aleatórios (com base em números cuja determinação é feita com algoritmos que constituem uma boa aproximação de uma escolha verdadeiramente aleatória).

Como o PEPE-16 não tem mecanismos para gerar valores aleatórios, usa-se um truque simples:

- A leitura de um periférico de entrada gera valores aleatórios nos bits que estejam “no ar”, ou seja, não ligados a algo que force um valor. Tal é o caso dos bits 7 a 4 do periférico **PIN** no circuito usado neste guião, cujos bits 3 a 0 ligam ao teclado, mas nada liga aos bits 7 a 4;
- Faz-se uma leitura do periférico, querendo-se aproveitar os bits aleatórios 7 a 4 e colocá-los nos bits de menor peso, com uma instrução de deslocamento à direita (instrução **SHR**);
- Para gerar um valor aleatório entre 0 e 7, por exemplo, faz-se de um deslocamento do valor lido para a direita (instrução **SHR**) de 5 bits, o que coloca os bits 7 a 5 (aleatórios) nos bits 2 a 0. Fica-se com 3 bits que dão um valor entre 0 e 7.

Outra hipótese, muito simples, mas que dada a existência de várias atividades pode funcionar surpreendente bem, é simplesmente ir circulando sequencialmente pelos vários valores possíveis (0 a 7, por exemplo), por meio de uma variável usada como um contador que dá a volta quando chega a um valor máximo.

Com várias coisas a acontecer “simultaneamente” e de forma rápida na interface da aplicação, a noção desta sequencialidade dilui-se e acaba por parecer mais aleatório do que sequencial.

6.5 Displays em decimal

O PEPE-16 só sabe funcionar em hexadecimal (melhor dizendo, em binário). Mas os displays, para utilizador ver, devem estar em decimal e não em hexadecimal.

O que se recomenda é deixar o PEPE-16 fazer as contas em hexadecimal, normalmente, mas quando se quer mostrar um dado valor nos displays ele deve ser convertido para decimal.

Mais corretamente, o que se pretende obter é um número hexadecimal cujos *nibbles* (conjuntos de 4 bits) sejam os dígitos do número decimal equivalente.

Por exemplo, o hexadecimal 26H corresponde a 38 decimal. É apenas mudança de base.

O que se pretende neste exemplo é obter o número 38H num registo, que depois é escrito nos displays. Estes exibem 38, e o utilizador lê-o como 38 decimal.

O truque para conseguir isto de forma genérica e simples é fazer um ciclo, em que em cada iteração se divide o número que se quer converter por 10 (decimal) e se obtém o resto da divisão (entre 0 e 9). Estes restos irão dar os dígitos decimais que se pretendem.

Sugere-se o seguinte algoritmo genérico, até o fator ser inferior a 10 (em pseudo-código, com as instruções DIV, divisão inteira, e MOD, resto da divisão inteira):

1. número = número MOD fator ; número: o valor a converter nesta iteração
; fator: uma potência de 10 (para obter os dígitos)
2. fator = fator DIV 10 ; prepara o próximo fator de divisão
3. se fator < 10, termina ; resultado fica com o valor pretendido
4. dígito = número DIV fator ; mais um dígito do valor decimal (0 a 9)
5. resultado = resultado SHL 4 ; desloca, para dar espaço ao novo dígito
6. resultado = resultado OR dígito ; vai compondo o resultado
7. volta ao passo 1

Para converter um número hexadecimal para um decimal com 3 dígitos, por exemplo, o valor inicial do fator deve ser 1000 (decimal).