

Laboratório de Introdução à Arquitetura de Computadores

IST - LEIC

2022/2023

Programação em Assembly

Guião 2

8 a 12 de maio de 2023

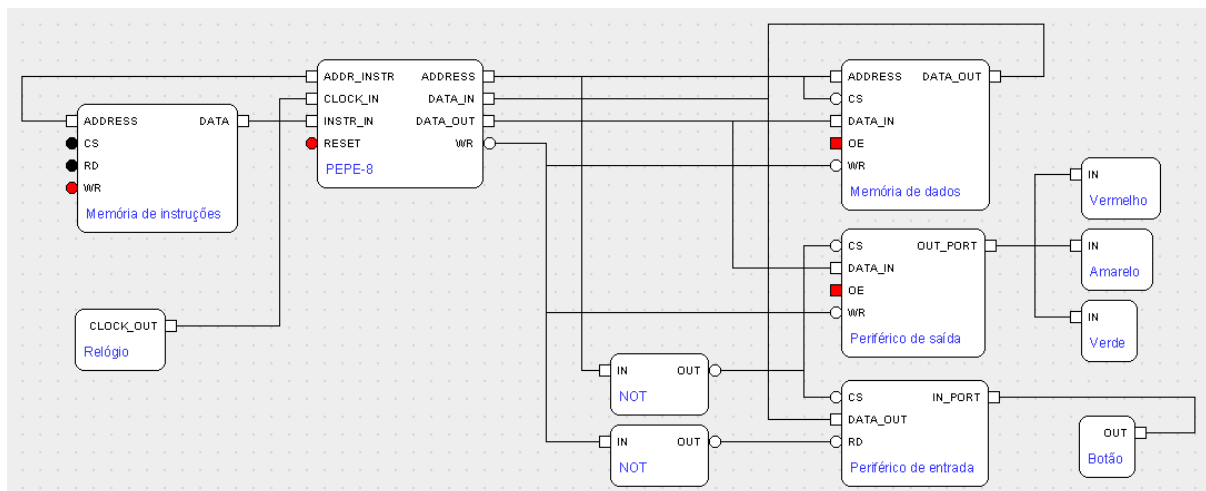
(Segunda metade da semana 2)

1 – Objetivos

Com este trabalho pretende-se que os alunos se familiarizem com as interfaces dos microprocessadores PEPE (Processador Especial Para Ensino) no simulador, por meio de um conjunto de instruções simples.

2 – PEPE-8

Comecemos pelo PEPE-8, processador de 8 bits extremamente simples (e também limitado).



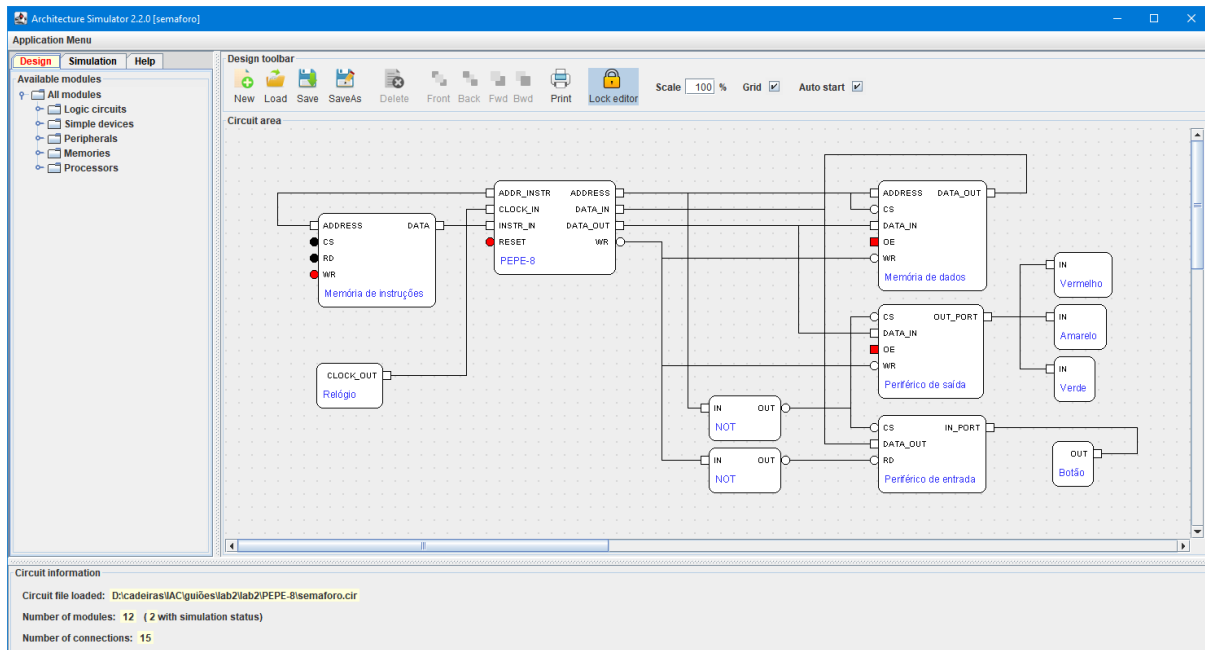
O circuito seguinte tem, essencialmente:

- O PEPE-8 (que tem memória de instruções e dados separadas);
- Memória de instruções, onde o programa é carregado;
- Memória de dados, onde o programa pode armazenar e ler valores;
- Periférico de saída, que liga a 3 leds (que vão fazer um semáforo);
- Periférico de entrada, que liga a um botão (para um peão forçar vermelho no semáforo);
- Um relógio (para temporizar a execução das instruções: relógio externo ao PEPE-8).

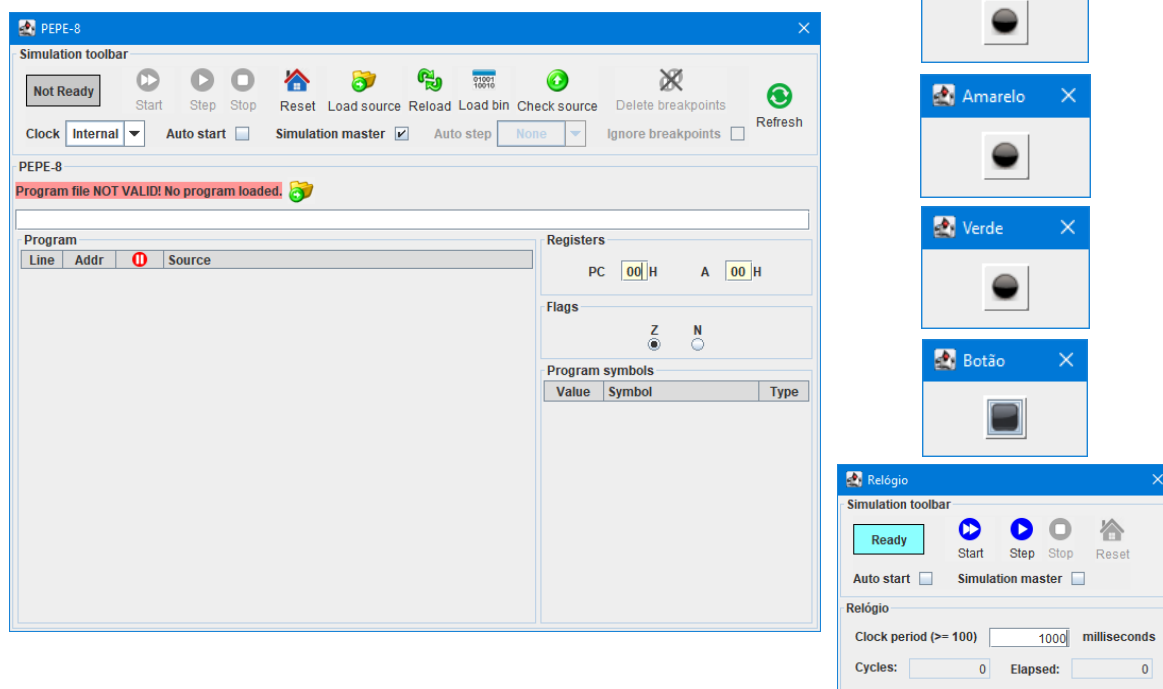
Inicie o simulador e carregue o ficheiro **semaforo.cir** (a extensão indica que é um circuito), que contém a descrição do circuito descrito pela figura anterior. Use o botão **Load** (📂) na “Design toolbar”. Os ficheiros do PEPE-8 estão dentro do diretório com o mesmo nome.


NOTA – Em alternativa, também pode fazer *drag & drop* do ficheiro para a zona de edição do circuito (“Circuit area”) da janela do simulador.

O circuito deverá aparecer no simulador.



Passa para “Simulation”. Devem aparecer logo as janelas de simulação dos vários componentes (sempre que um circuito é guardado depois de abrir estas janelas, da próxima vez que se passar para “Simulation” as janelas abrem de novo).



Carregue no botão **Load source** () do PEPE-8 e abra o ficheiro **semaforo.asm**, cujo conteúdo é o indicado a seguir:

NOTA – Em alternativa, também pode fazer *drag & drop* do ficheiro para a zona do lado esquerdo da janela de simulação do PEPE-8, com o título “Program”.

```
; *****
; * IST-UL
; * Modulo:      semaforo.asm
; * Descrição: Exemplo de um controlador de semáforos no PEPE-8.
; *****


;constantes de dados
vermelho      EQU 01H      ;valor do vermelho (lâmpada liga ao bit 0)
amarelo       EQU 02H      ;valor do amarelo (lâmpada liga ao bit 1)
verde         EQU 04H      ;valor do verde (lâmpada liga ao bit 2)

;constantes de endereços
semaforo EQU 80H ;endereço 128 (periférico de saída)

;programa
inicio:
    LD verde      ;Carrega o registo A com o valor para semáforo verde
    ST [semaforo] ;Atualiza o periférico de saída
semVerde:
    NOP           ;faz um compasso de espera
    NOP           ;faz um compasso de espera
    NOP           ;faz um compasso de espera
    LD amarelo    ;Carrega o registo A com o valor para semáforo amarelo
    ST [semaforo] ;Atualiza o periférico de saída
semAmar:
    NOP           ;faz um compasso de espera
    LD vermelho   ;Carrega o registo A com o valor para semáforo vermelho
    ST [semaforo] ;Atualiza o periférico de saída
semVerm:
    NOP           ;faz um compasso de espera
    NOP           ;faz um compasso de espera
    NOP           ;faz um compasso de espera
    NOP           ;faz um compasso de espera
    JMP inicio    ;vai fazer mais uma ronda
```

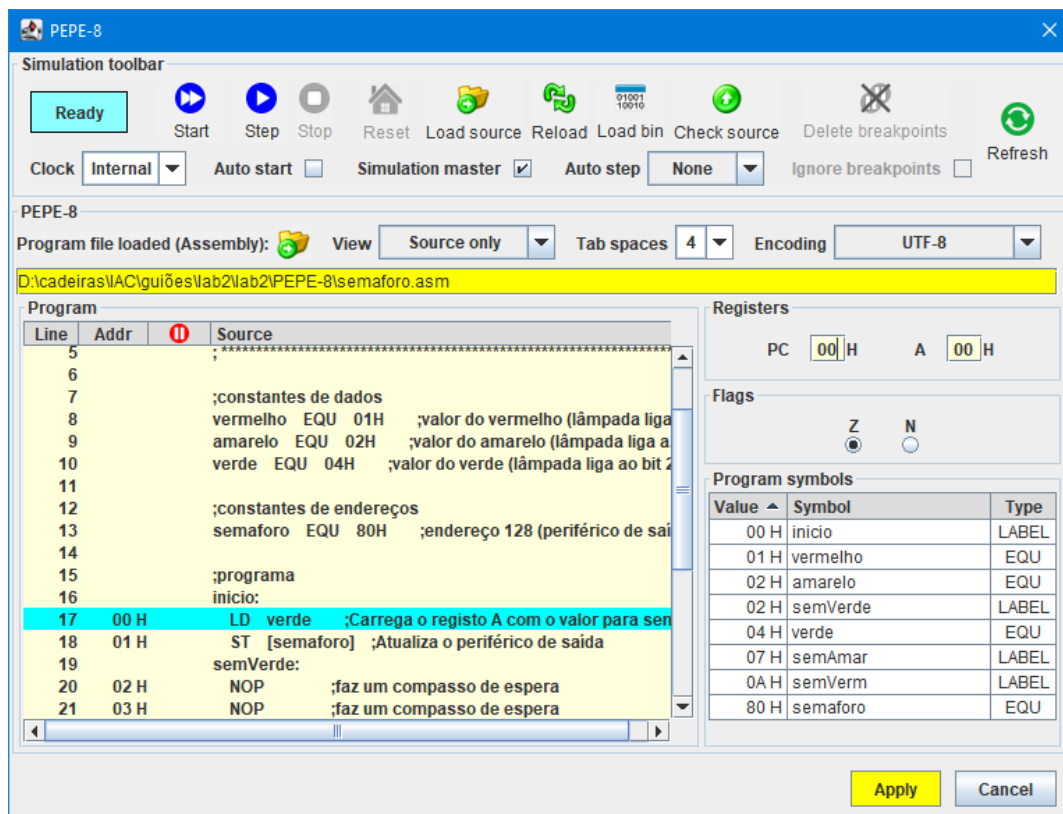
Este programa vai circulando pelas 3 cores do semáforo, em ciclo infinito. Os NOPs (No-Operation) são instruções que não fazem nada e constituem uma tentativa de gastar tempo, para marcar as temporizações das várias cores.

Na janela do PEPE-8 pode ver o programa deste ficheiro. A formatação pode não ser a mesma, devido ao número de espaços dos tabs. Pode tentar variar o “Tab spaces”.

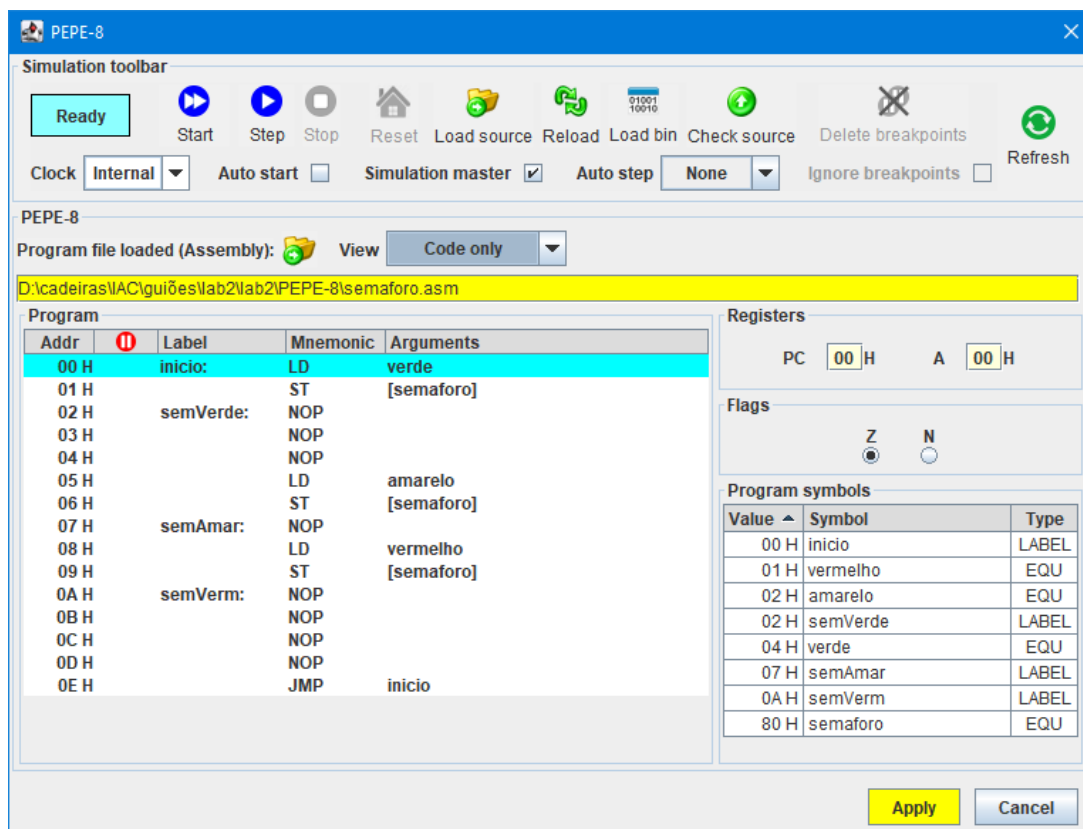
Se quiser alterar um programa, tem de usar um editor de texto, como por exemplo o NotePad++ para PC ou o Brackets para Mac. Não é possível alterar o programa diretamente na janela do PEPE-8. Depois de alterar um programa e guardar o ficheiro, pode usar o botão **Reload** () para carregar a nova versão.

Nas instruções, pode usar minúsculas em vez de maiúsculas.

A barra azul indica a instrução que será executada a seguir. Pode também ver-se os números de linha e os endereços em que as instruções estão.



O *assembler* compila o código-fonte (o programa, que é texto) e gera código-máquina (números binários). Na janela de interface do PEPE-8 aparece o código-fonte por omissão, mas é possível também ver apenas o código-máquina (convertido de novo para instruções assembly, mas sem comentários, pois esses não passam para o código-máquina). Basta seleccionar a View “Code only”:



Na janela do PEPE-8 é ainda possível ver:

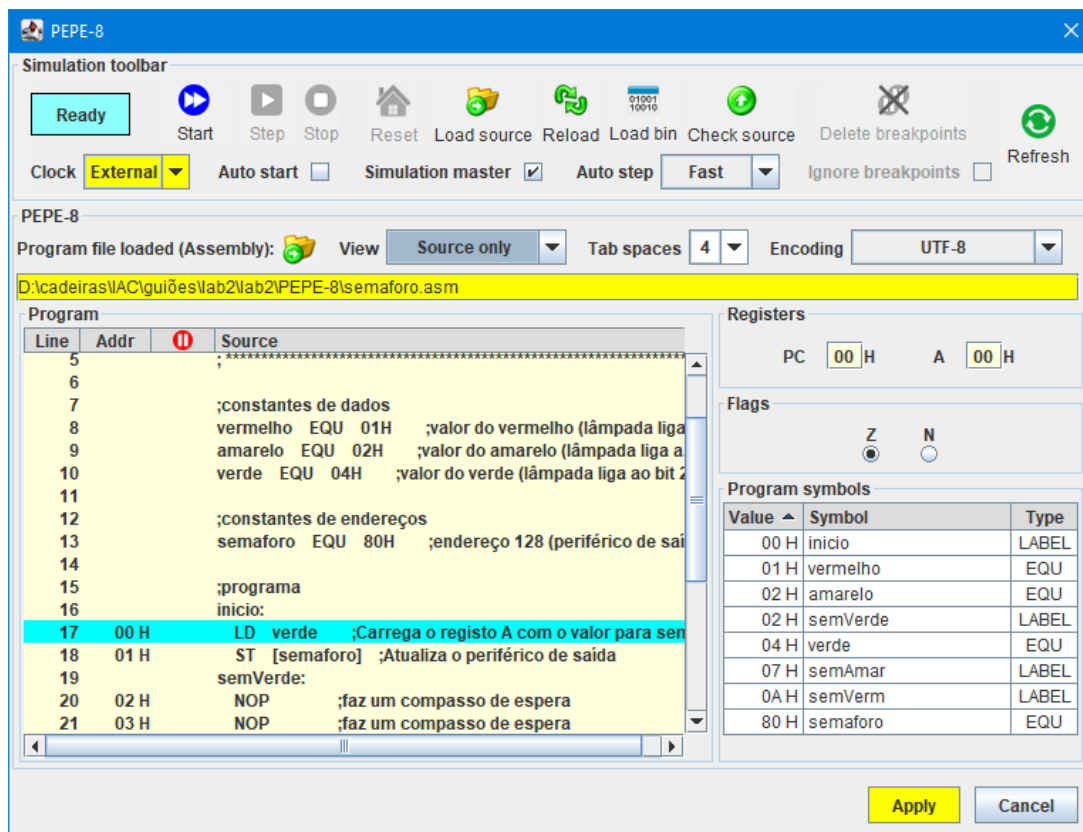
- O valor do registo PC (Program Counter), que indica o endereço da próxima instrução a executar;
- O valor do registo A;
- O valor dos bits de estado (*flags*) Z e N, que se estiverem seleccionados indicam se o registo A tem um valor zero ou negativo, respetivamente;
- A tabela de símbolos, que lista os símbolos (EQUs e labels) usados no programa e os respetivos valores.

Os amarelos indicam que houve uma alteração qualquer face ao ficheiro do circuito, quando este foi carregado. Neste caso, carregou-se um programa (não havia nenhum anteriormente).

Se fizer clique no botão “Apply”, este ficheiro de programa fica assumido, em vez de ser considerado uma alteração pendente (a amarelo). Se guardar agora o circuito (em modo “Design”), da próxima vez que o carregar este programa já é carregado automaticamente.

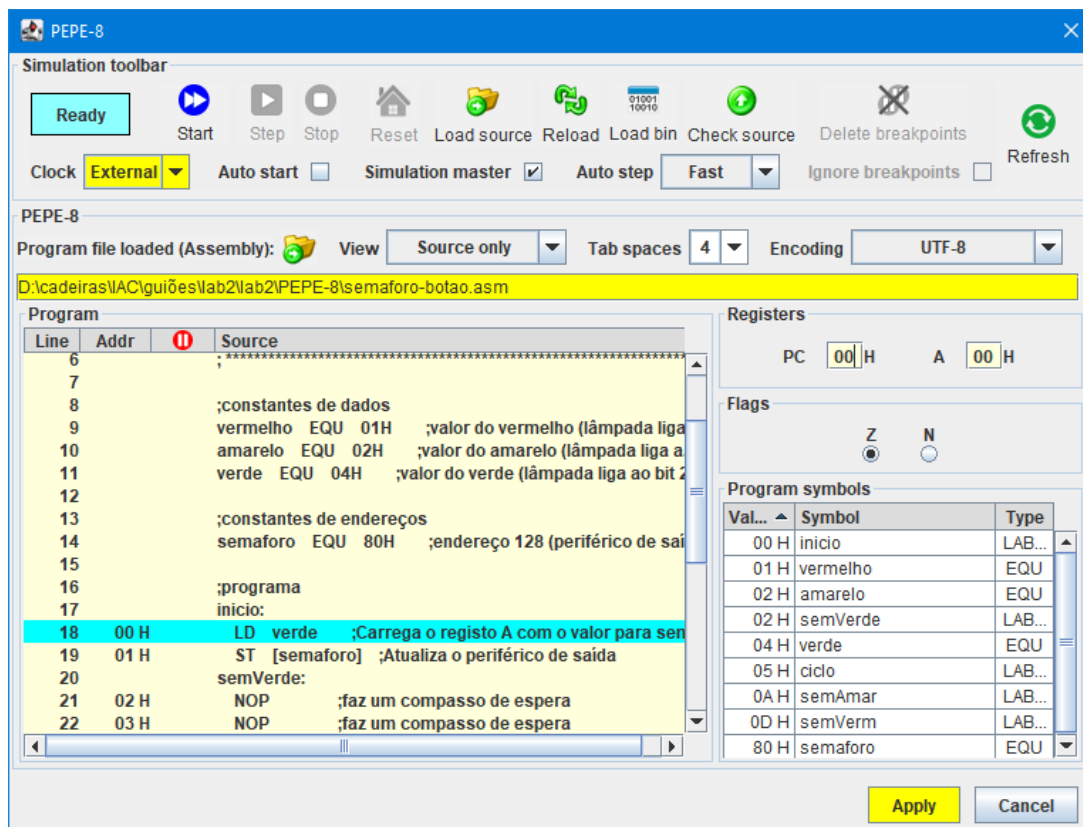
Pretende-se que:

- Execute as instruções uma a uma (passo a passo), com o botão **Step** (▶);
- Note que cada vez que carrega no botão **Step** (▶) o estado do processador passa brevemente por **STEPPING** (violeta), enquanto está a executar uma instrução, mas depois fica em **PAUSED** (amarelo);
- Após a execução de cada instrução (a barra azul passa a indicar a próxima instrução), verifique as alterações no registo A e se a instrução fez o esperado, nomeadamente nos ST (*store*, escreve na memória ou periférico), que escreve a cor do semáforo nos leds;
- Verifique a evolução do registo PC (*Program Counter*) e dos bits de estado, nomeadamente o bit **Z**;
- Verifique o sequenciamento das instruções e o salto incondicional no fim (**JMP**);
- O PEPE-8 também é capaz de executar as instruções passo a passo de forma automática. No **Auto step**, selecione uma velocidade média (“Fast”, por exemplo). O botão de **Step** fica diferente (▶), para indicar *step* automático. Carregue nesse botão e verifique que o PEPE-8 percorre automaticamente as várias instruções, uma a uma. Pode fazer pausa, carregando no botão **Pause** (⏏). Pode também seleccionar outras velocidades;
- Execute agora o programa em modo contínuo, à velocidade máxima que o simulador permite, carregando no botão **Resume** (▶). Repare que os leds piscam de forma muito rápida, o que quer dizer que o processador executa instruções muito rápido e os NOPs não conseguem uma temporização adequada. Para tal, temos de usar um relógio externo ao processador (relógio ligado ao processador que se vê na primeira figura deste guião);
- Termine a execução do programa, fazendo clique no botão **Stop** (■) do PEPE-8, e clique no botão **Reset** (🏠), o que faz o programa voltar ao seu estado inicial;
- Na janela do PEPE-8, selecione o clock “External” (figura seguinte). Mais uma vez, pode carregar em **Apply** para as indicações amarelas desaparecerem.
- Carregue no botão **Start** (▶) do PEPE-8 e igualmente no botão **Start** (▶) da janela do relógio. Verifique que agora o semáforo evolui com uma temporização mais adequada. O relógio externo tem um período de 1000 milissegundos (1 segundo), o que significa que o PEPE-8 executa uma instrução por segundo. Os NOPs já mantêm a temporização. Pode colocar um período mais curto, para ser mais rápido.



Este exemplo já ilustra como um processador muito simples já pode fazer coisas úteis, mas ainda é possível aumentar a funcionalidade.

Termine a execução do programa, fazendo clique no botão **Stop** (■) do PEPE-8, e carregue agora o programa **semaforo-botao.asm**.



Este programa é quase igual ao anterior, mas depois de colocar o semáforo a verde vai testando o botão e, enquanto não for pressionado, mantém o verde. Quando se carrega no botão, sai do verde e faz então um ciclo completo, até se bloquear de novo no verde.

Execute o programa (mantendo o relógio em “External”) e teste esta nova funcionalidade do botão, mas note que tem de manter o botão carregado durante algum tempo, até o processador executar as instruções que detetam esta situação (nomeadamente, até o led amarelo acender, altura em que pode largar o botão). Não se esqueça de fazer também **Start** (👉) no relógio.

Este guião inclui ainda um outro programa (ficheiro conta-uns.asm), que pode executar em single-step e verificar como funciona.

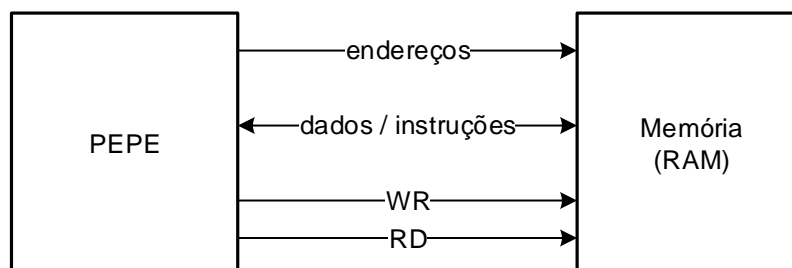
NOTA – O single-step só funciona com o relógio interno!

3 – PEPE-16

O PEPE-8 é um processador demasiado simples para uma utilização prática, pelo que a partir de agora utilizaremos o PEPE-16, mais poderoso.

3.1 – O circuito de simulação

Neste guião está em causa executar instruções que envolvam apenas os recursos internos do PEPE-16 e memória (periféricos vão aparecer no guião seguinte). Por conseguinte, o circuito a usar é muito simples e está representado de forma simplificada na figura seguinte.



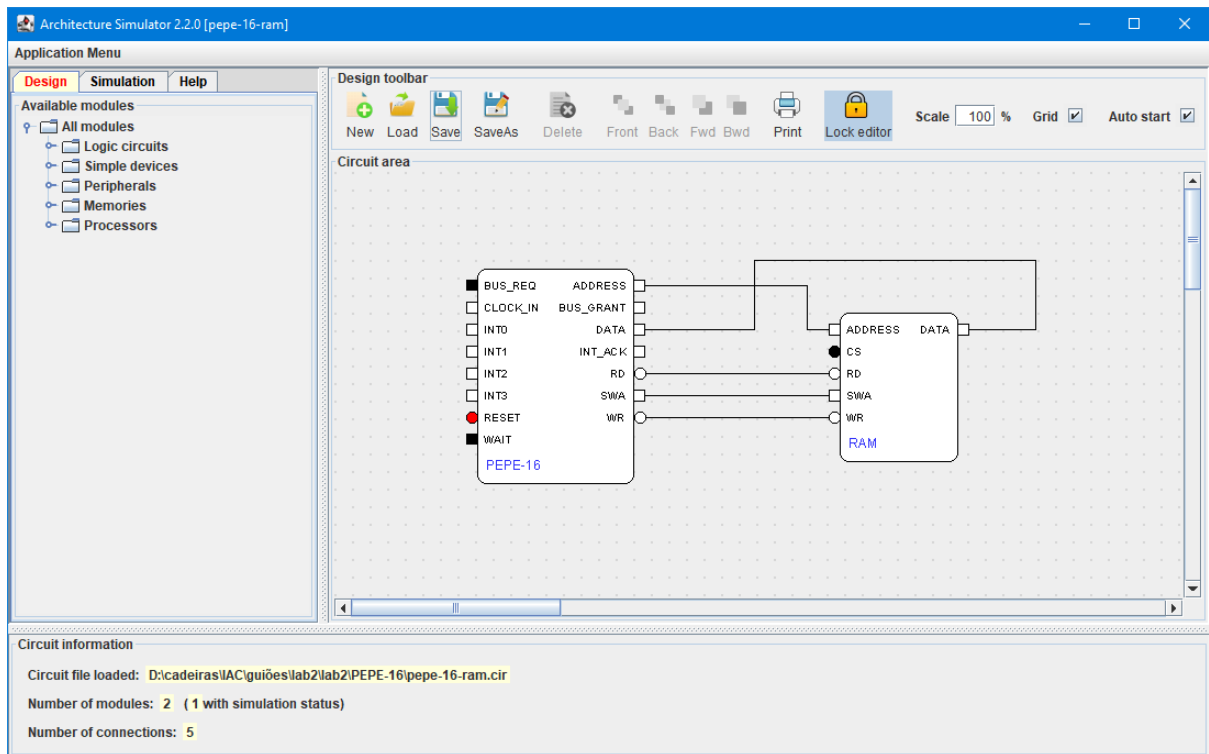
O relógio do PEPE-16 é gerado internamente, por omissão. Embora possa ser configurado para ligar a um relógio exterior, tal como o PEPE-8, qualquer programa não trivial ficaria, no entanto, ridiculamente lento.

Os restantes pinos de entrada do PEPE-16 e que não estão a ser usados estão forçados a 0 (preto) ou 1 (vermelho), consoante o valor por omissão necessário.

A memória tem um bus de dados de 16 bits, tal como o PEPE-16.

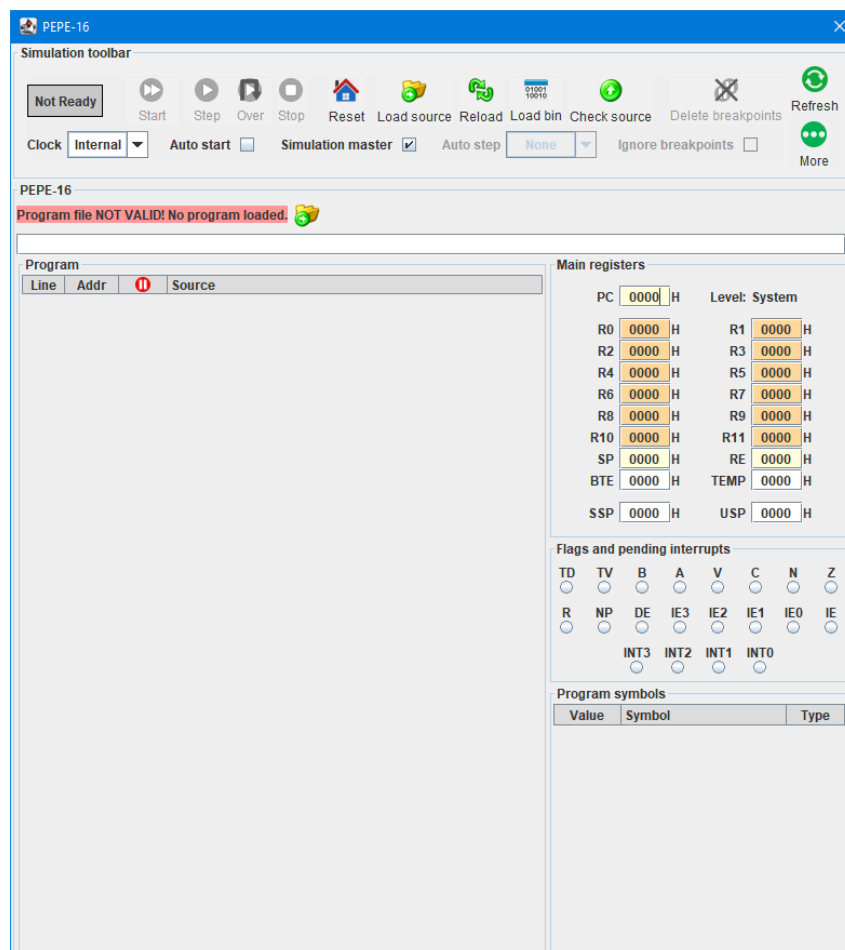
Carregue o ficheiro **pepe-16-ram.cir** (a extensão indica que é um circuito), que contém a descrição do circuito descrito pela figura anterior. Para tal, use o botão **Load** (📁) na “Design toolbar” ou faça *drag & drop* do ficheiro para a zona de edição do circuito (“Circuit area”) da janela do simulador.

Os ficheiros do PEPE-16 estão dentro do diretório com o mesmo nome.



Passe para “Simulation”.

Um clique no processador faz abrir a interface do PEPE-16, onde se pode ver o estado dos seus recursos internos (registos e bits de estado do processador).

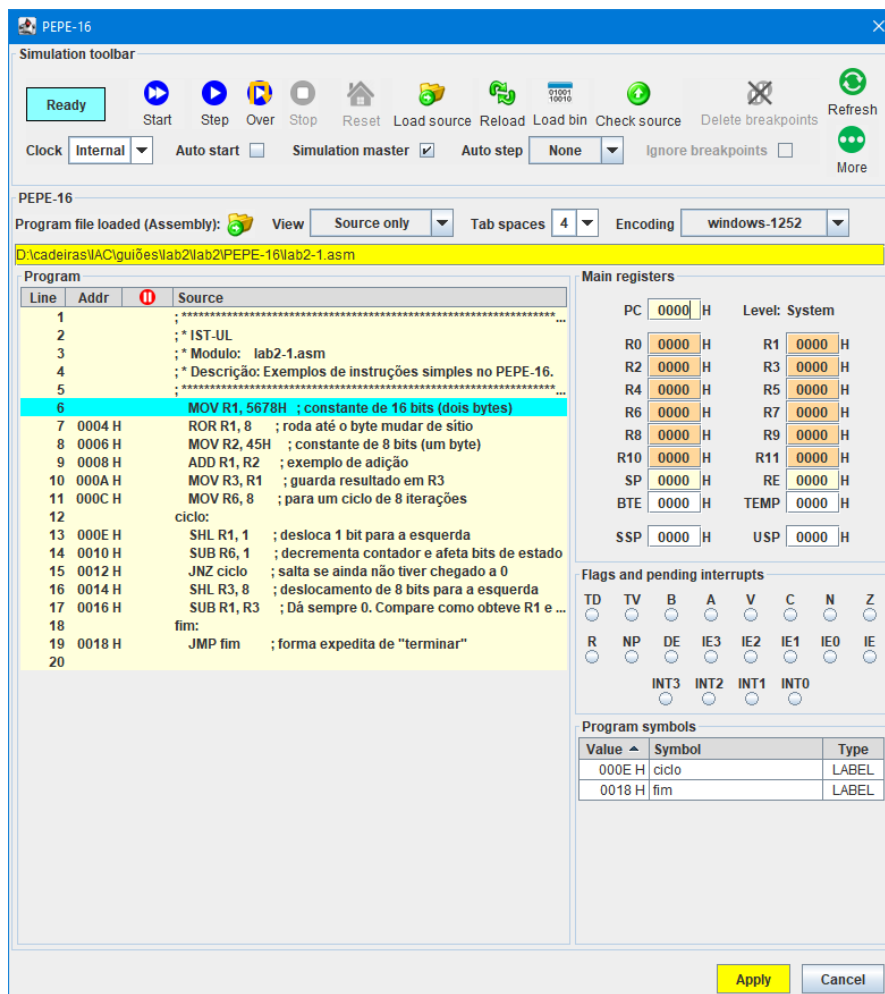


3.2 – Exemplos de instruções simples

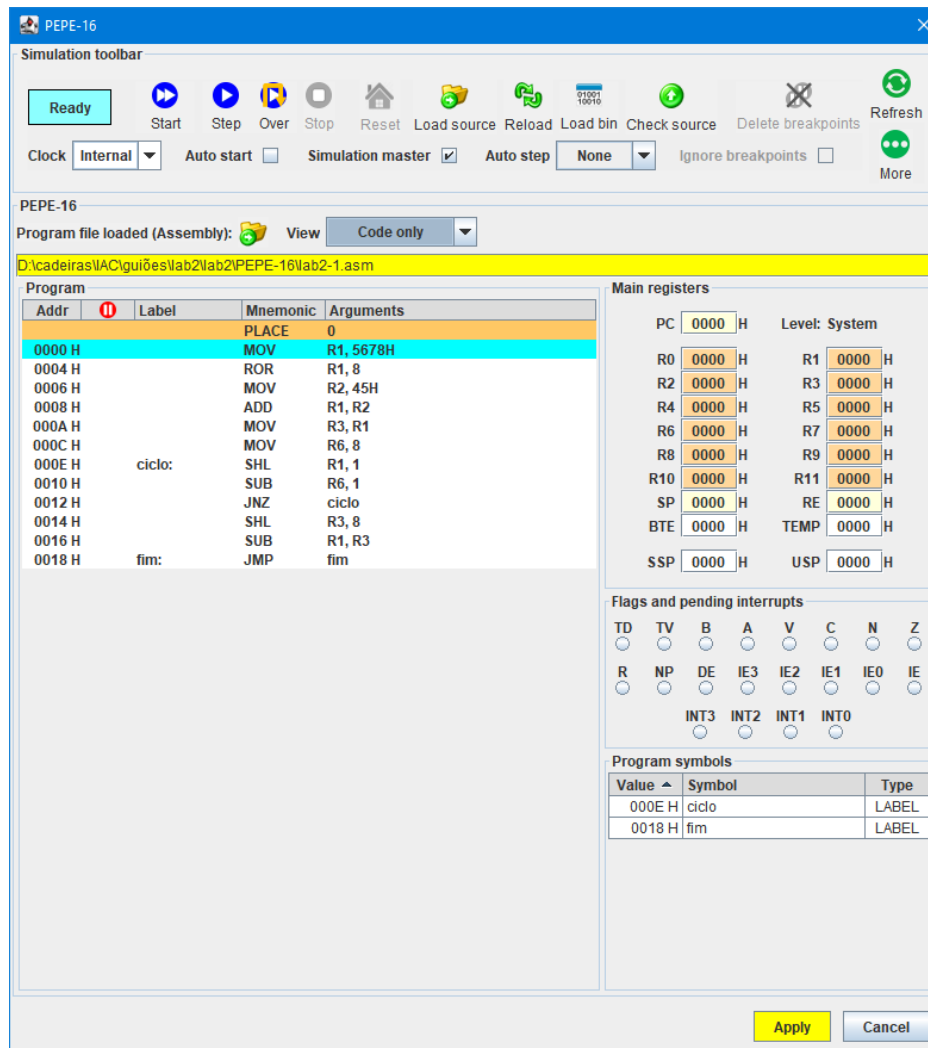
Carregue no botão **Load source** (📁) do PEPE-16 e abra o ficheiro **lab2-1.asm**, cujo conteúdo é o indicado a seguir:

NOTA – Em alternativa, também pode fazer *drag & drop* do ficheiro para a zona do lado esquerdo da janela de simulação do PEPE-16, com o título “Program”.

```
; *****
; * IST-UL
; * Modulo:    lab2-1.asm
; * Descrição: Exemplos de instruções simples no PEPE-16.
; *****
MOV R1, 5678H    ; constante de 16 bits (dois bytes)
ROR R1, 8        ; roda até o byte mudar de sítio
MOV R2, 45H      ; constante de 8 bits (um byte)
ADD R1, R2       ; exemplo de adição
MOV R3, R1       ; guarda resultado em R3
MOV R6, 8        ; para um ciclo de 8 iterações
ciclo:
SHL R1, 1        ; desloca 1 bit para a esquerda
SUB R6, 1        ; decrementa contador e afeta bits de estado
JNZ ciclo        ; salta se ainda não tiver chegado a 0
SHL R3, 8        ; deslocamento de 8 bits para a esquerda
SUB R1, R3       ; Dá sempre 0. Compare como obteve R1 e R3
fim:
JMP fim          ; forma expedita de "terminar"
```



Tal como no PEPE-8, é possível ver apenas o código gerado, na view “Code only”:



Note que a instrução **MOV R1, 5678H** gasta 4 bytes, ao contrário da generalidade de instruções que gasta apenas 2. Isto deve-se ao facto de internamente o assembler dividir esta instrução em duas, uma que carrega o byte 56H e outra o byte 78H. Tal é transparente para o programador, mas nota-se nos endereços das várias instruções. Não seria possível incluir na mesma instrução de 16 bits informação sobre qual a operação a executar (MOV), qual o registo destino (R1) e ainda uma constante de 16 bits! Assim, divide-se a instrução original em duas e cada uma trata de metade da constante.

A barra azul indica qual a instrução que vai ser executada a seguir.

Se fizer clique no botão “Apply”, este ficheiro fica assumido em vez de ser considerado uma alteração pendente (a amarelo).

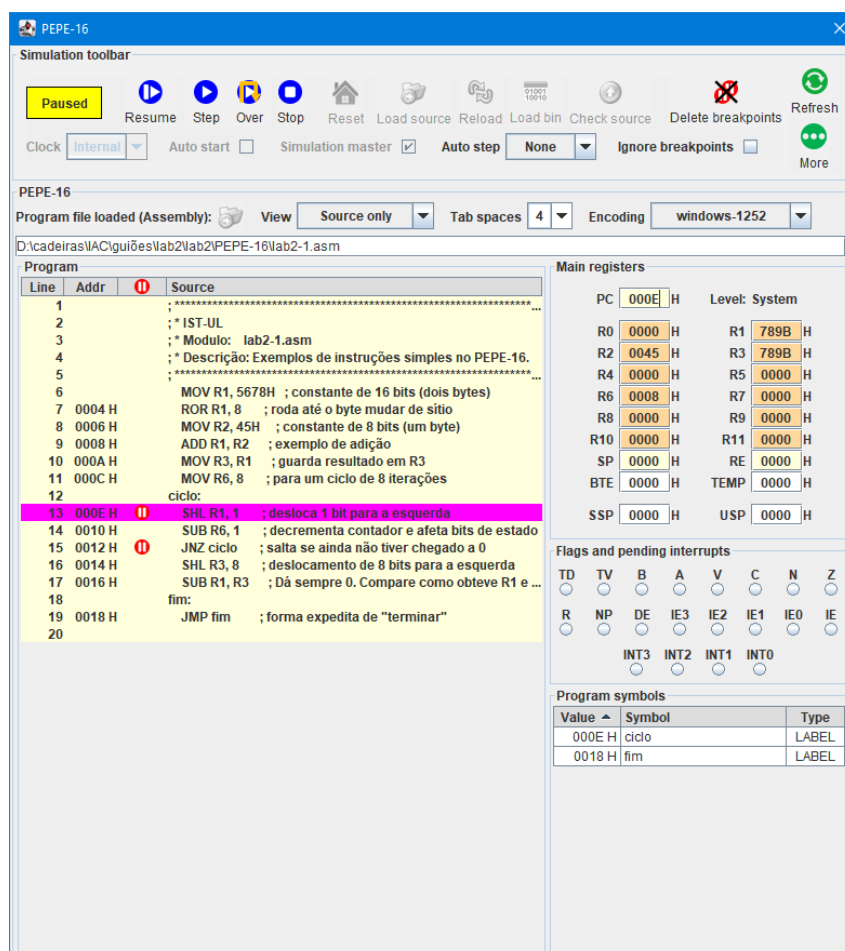
Pretende-se que:


- Execute as instruções uma a uma (passo a passo), com o botão **Step** (🔵). O botão **Over** (🟡) também funciona, mas as diferenças face ao **Step** só serão explicadas no guião de laboratório 4;
- Após a execução de cada instrução (a barra azul passa a indicar a próxima instrução), verifique as alterações nos registos e se a instrução fez o esperado;

- Verifique a evolução do registo PC (*Program Counter*) e dos bits de estado, nomeadamente o bit **Z**;
- Verifique o sequenciamento da execução das instruções e os saltos, condicionais e incondicionais (verifique que **JNZ ciclo** salta para **ciclo**, mas apenas até **R6** chegar a zero, e verifique que após chegar a **fim** não sai de lá);
- Termine a execução do programa, fazendo clique no botão **Stop** (■) do PEPE-16.

Faça agora o seguinte:



- Clique no botão **Reset** (🏠), o que faz o programa voltar ao seu estado inicial;
- Coloque um *breakpoint* (ponto de paragem) na instrução com a etiqueta **ciclo** (**SHL**), simplesmente fazendo clique na linha da instrução, que fica com o sinal (⏸), e outro na instrução **JNZ** (ver figura seguinte). Execute o programa em modo contínuo, carregando no botão **Start** (▶), e verifique que o programa suspende a sua execução no *breakpoint*, no mesmo estado que quando fez step. A barra a roxo indica que o PEPE-16 parou devido ao *breakpoint* nessa instrução. Para apagar um *breakpoint*, basta fazer clique na linha novamente. Se usar o botão direito do rato, tem mais opções. Pode ainda apagar ou ignorar os *breakpoints* todos (na *toolbar*);
- Analise o que é que o programa faz, em particular como é se chegou aos valores de **R1** de **R3** antes da instrução **SUB R1, R3**. O valor de **R1** após esta instrução é sempre zero, pois **R1** e **R3** resultam de duas formas alternativas de calcular o mesmo valor;
- Termine a execução do programa, carregando no botão **Stop** (■) do PEPE-16.








Carregue agora no PEPE-16 o ficheiro **lab2-2.asm** (ou pelo botão **Load source**, , ou por *drag & drop*), cujo conteúdo é o seguinte:

```
; *****
; * IST-UL
; * Modulo:    lab2-2.asm
; * Descrição: Exemplos de instruções simples no PEPE-16.
; *****
MOV R1, 4356H    ; constante de 16 bits (dois bytes)
MOV R2, 21H      ; constante de 8 bits (um byte)
SUB R1, R2       ; exemplo de subtração
MOV R5, 00FFH    ; máscara
AND R1, R5       ; elimina bits 8 a 15
XOR R1, R5       ; nega os bits 0 a 7
MOV R4, 0H       ; inicializa R4
ciclo:
MOV R6, 01H      ; máscara
AND R6, R1       ; força a 0 todos os bits exceto o de menor peso
JZ salta         ; salta se for 0
ADD R4, 1        ; se não der zero adiciona 1
salta:
SHR R1, 1        ; deslocamento de um bit a direita
JNZ ciclo        ; salta se ainda não tiver chegado a 0
fim:
JMP fim          ; forma expedita de "terminar"
```

Pretende-se que:

- Execute as instruções uma a uma (passo a passo), com o botão **Step** ();
- Após a execução de cada instrução, verifique as alterações nos registos e se a instrução fez o esperado, em particular antes da etiqueta **ciclo**;
- As instruções dentro do ciclo contam o número de bits a 1 do valor que **R1** tem quando o programa passa pela etiqueta **ciclo** pela primeira vez;
- Termine a execução do programa, carregando no botão **Stop** () do PEPE-16.

O PEPE-16 também é capaz de executar as instruções passo a passo de forma automática:

- Clique no botão **Reset** () , o que faz o programa voltar ao seu estado inicial;
- No **Auto step**, selecione uma velocidade média (“Fast”, por exemplo). O botão de **Step** fica diferente () , para indicar *step* automático. Carregue nesse botão e verifique que o PEPE-16 percorre automaticamente as várias instruções, uma a uma. Pode fazer pausa, carregando no botão **Pause** () , e retomar, carregando no botão **Resume** (). Pode também seleccionar outras velocidades;
- Termine a execução do programa, carregando no botão **Stop** () do PEPE-16.

3.3 – Exemplos de programas com funcionalidade específica

De seguida carregue, execute passo a passo (single-step, em “Simulation”) e tente perceber o funcionamento do programa 4.11 do livro, contido no ficheiro **programa4-11.asm**, e que calcula o fatorial de um dado número.

Os comentários permitem perceber melhor cada instrução e ilustram o estilo de programação em linguagem assembly.

O livro explica estes exemplos em maior detalhe. A tabela A.9 do livro contém informação sobre cada instrução do PEPE.

```
; *****
; * IST-UL
; * Modulo:      programa4-11.asm (do livro)
; * Descrição: Algoritmo para calcular o fatorial.
; *             Utilização dos registos:
; *             R1 - Produto dos vários fatores (valor do fatorial no fim)
; *             R2 - fator auxiliar que começa com N-1, depois N-2, ...
; *             ... até ser 2 (1 já não vale a pena).
; *
; * Nota :  Verifique como se declaram constantes
; *****

; *****
; * Constantes
; *****
N    EQU 6          ; número de que se pretende calcular o fatorial

; *****
; * Código
; *****

inicio:
    MOV R1, N      ; valor inicial do produto
    MOV R2, R1     ; valor auxiliar
maisUm:
    SUB R2, 1      ; decrementa fator
    MUL R1, R2     ; acumula produto de fatores
    JV  erro       ; se houve excesso, o fatorial tem de acabar aqui
    CMP R2, 2      ; verifica se o fator diminuiu até 2
    JGT maisUm     ; se ainda é maior do que 2, deve continuar
fim:
    JMP fim        ; acabou. R1 com o valor do fatorial
erro:
    JMP erro       ; termina com erro
```

Agora carregue, execute passo a passo (single-step, em “Simulation”) e tente perceber o funcionamento do programa 4.21 do livro, contido no ficheiro **programa4-21.asm**:

```
; *****
; * IST-UL
; * Modulo:      programa4-21.asm
; * Descrição: Algoritmo para contar '1s' numa constante.
; *      Utilização dos registos:
; *      R1 - valor cujo número de '1s' deve ser contado
; *      R2 - contador
; *
; * Nota : Verifique como se declaram constantes
; *****

; *****
; * Constantes
; *****
valor EQU 6AC5H ; valor cujos bits a 1 vão ser contados

; *****
; * Código
; *****
inicio:
    MOV R1, valor ; inicializa registo com o valor a analisar
    MOV R2, 0     ; inicializa contador de número de 1s
    MOV R3, 0
maisUm:
    CMP R1, 0     ; para atualizar os bits de estado
    JZ fim        ; se o valor já é zero, não há mais 1s para contar
    SHR R1, 1     ; retira o bit de menor peso (fica no bit C-Carry)
    ADDC R2, R3    ; soma mais 1 ao contador, se esse bit for 1
    JMP maisUm     ; vai analisar o próximo bit
fim:
    JMP fim        ; acabou. R2 tem o número de bits a 1 no valor
```

3.4 – Exemplos de instruções de acesso à memória

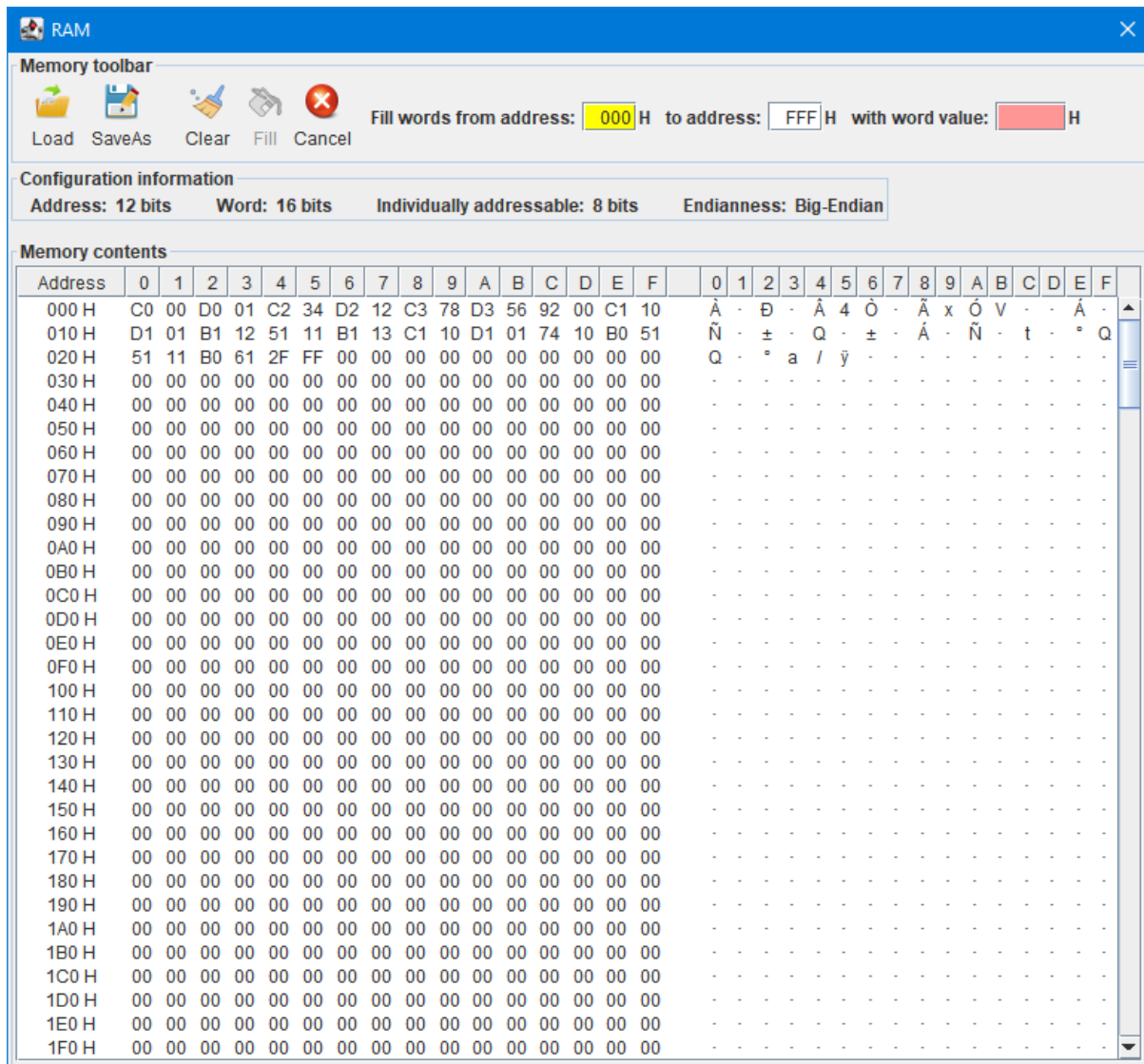
Verifique agora o funcionamento das instruções de acesso à memória (**MOV** e **MOVB**), carregando no PEPE-16 o ficheiro **lab2-3.asm**, cujo conteúdo é o seguinte:

```
; *****
; * IST-UL
; * Modulo:      lab2-3.asm
; * Descrição: Exemplos de instruções de acesso à memória,
; *      em word e em byte.
; *****
inicio:
    MOV R0, 0100H ; endereço da célula de memória a aceder em word
    MOV R2, 1234H ; constante de 16 bits (dois bytes)
    MOV R3, 5678H ; constante de 16 bits (dois bytes)
    MOV [R0], R2  ; escreve uma word (16 bits) na memória
    MOV R1, 0110H ; endereço da célula de memória a aceder em byte
    MOVB [R1], R2 ; escreve um byte (8 bits) na memória (0110H)
    ADD R1, 1     ; endereço 0111H
    MOVB [R1], R3 ; escreve um byte (8 bits) na memória (0111H)
    MOV R1, 0110H ; repõe endereço 0110H em R1
    MOV R4, [R1]  ; lê uma word (16 bits) da memória (endereço par)
    MOVB R5, [R1] ; lê um byte (8 bits) da memória (0110H)
    ADD R1, 1     ; endereço 0111H
    MOVB R6, [R1] ; lê um byte (8 bits) da memória (0111H)
fim:
    JMP fim        ; forma expedita de "terminar"
```

Este programa tem as suas instruções localizadas a partir do endereço 0000H (atribuído automaticamente pelo assembler do programa) e acede à memória nos endereços 0100H, 0110H e 0111H.

Os acessos em palavra (word, 16 bits, instrução **MOV**) só podem ser feitos a endereços pares, enquanto os acessos em byte (8 bits, instrução **MOVB**), podem ser feitos a qualquer endereço (par ou ímpar).

Faça clique na memória (RAM), o que faz abrir o respetivo painel de simulação:



Neste painel, note o seguinte sobre esta RAM:

- Tem 12 bits de endereço. Isto quer dizer que consegue endereçar 2^{12} (4096, ou 1000H) células diferentes;
- Tem 16 bits de palavra. Isto quer dizer que no máximo consegue ler ou escrever 16 bits ao mesmo tempo, em cada acesso;

- No entanto, consegue aceder individualmente a cada byte (8 bits). Por isso, suporta acessos de 16 bits (feitos pelas instruções **MOV** do PEPE-16) e acessos de 8 bits (feitos pelas instruções **MOVB** do PEPE-16);
- Porque cada byte tem de ter o seu próprio endereço (para poder ser acedido individualmente), a capacidade da RAM é 2^{12} bytes, ou 2^{11} palavras de 16 bits;
- A indicação “Big-Endian” quer dizer que num acesso de palavra (instruções **MOV**) o byte de maior peso da palavra é escrito no (ou lido do) endereço par, e o byte de menor peso da palavra é escrito no (ou lido do) endereço ímpar.

Cada linha do painel tem o conteúdo de 16 bytes (note o cabeçalho das colunas de 0 a F), pelo que o endereço em cada linha é múltiplo de 16 (dígito hexadecimal de menor peso a 0).

Na parte direita aparecem caracteres sempre que o byte correspondente tem um valor que indique uma letra, um dígito ou um sinal de pontuação.

Note que os bytes iniciais já estão com valores diferentes de 00H (valor com que todos os bytes da RAM são inicializados). Estes bytes são os do programa. O PEPE-16 começa a executar os programas a partir do endereço 0, pelo que é a partir daí que os programas têm de ser carregados em memória.

Execute agora o programa passo a passo, com o botão **Step** (▶). Após cada instrução, verifique o conteúdo dos registos envolvidos (no painel do PEPE-16) e o valor dos bytes nos endereços, 0100H, 0110H e 0111H da RAM (no seu painel), de forma a perceber o funcionamento das instruções **MOV** e **MOVB**.

A instrução **MOV [R0], R2** escreve não apenas no endereço 0100H (valor do **R0**) mas também no seguinte, 0101H, pois o acesso é em 16 bits (dois bytes) e cada endereço refere-se apenas a um byte.

As instruções **MOVB [R1], R2** e **MOVB [R1], R3** escrevem apenas no endereço indicado por R1, pois o acesso é apenas a um byte.

Note que só as instruções com parênteses retos acedem realmente à memória. Os restantes **MOV**s destinam-se apenas a carregar constantes nos registos.

NOTA – As instruções **MOV** suportam uma constante entre os parênteses retos, mas as instruções **MOVB** obrigam a que se use um registo entre os parênteses retos. Esta diferença deve-se a não ser possível suportar todas as codificações de instruções desejáveis com apenas 16 bits, por isso umas são mais “privilegiadas” do que outras.

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