Trabalho prático N.º 4

Objetivos

- Configurar e usar os portos de I/O do PIC32 em linguagem C.
- Implementar um sistema de visualização com dois *displays* de 7 segmentos.

Introdução

A configuração e utilização dos portos de I/O do PIC32, em linguagem C, fica bastante facilitada se se utilizarem estruturas de dados para a definição de cada um dos bits dos registos a que se pode aceder. Por exemplo, para o registo **TRIS** associado ao porto **RE**, pode ser declarada uma estrutura com 8 campos (o número de bits real do porto **RE** no PIC32MX795F512H), cada um deles com a dimensão de 1 bit[1]:

```
typedef struct {
  unsigned int TRISE0 : 1;      // 1-bit field (least significant bit)
  unsigned int TRISE1 : 1;      // ...
  unsigned int TRISE2 : 1;      // ...
  unsigned int TRISE3 : 1;      // ...
  unsigned int TRISE4 : 1;      // ...
  unsigned int TRISE5 : 1;      // ...
  unsigned int TRISE6 : 1;      // ...
  unsigned int TRISE7 : 1;      // 1-bit field (most significant bit)
} __TRISEbits_t;
```

A partir desta declaração pode ser criada uma instância da estrutura, por exemplo, **TRISEbits**:

```
__TRISEbits_t TRISEbits; // TRISEbits é uma instância de __TRISEbits_t
```

O acesso a um bit específico da estrutura pode então ser feito através do nome da instância seguido do nome do membro, separados pelo caracter "." (e.g. **TRISEbits.TRISE7**). Por exemplo, a configuração dos bits 2 e 5 do porto **E** (**RE2** e **RE5**) como entrada e saída, respetivamente, pode ser feita com as duas seguintes instruções em linguagem C:

```
TRISEbits.TRISE2 = 1;    // RE2 configured as input
TRISEbits.TRISE5 = 0;    // RE5 configured as output
```

Seguindo esta metodologia, podem ser declaradas estruturas que representem todos os registos necessários para a leitura, a escrita e a configuração de um porto. Tomando ainda como exemplo o porto **E**, para além do registo **TRIS**, temos ainda os registos **LAT** (constituído pelos bits **LATE7** a **LATE0**) e **PORT** (constituído pelos bits **RE7** a **RE0**):

```
typedef struct {
  unsigned int RE0 : 1;
  unsigned int RE1 : 1;
  unsigned int RE2 : 1;
  unsigned int RE3 : 1;
  unsigned int RE4 : 1;
  unsigned int RE5 : 1;
  unsigned int RE6 : 1;
  unsigned int RE7 : 1;
} __PORTEbits_t;
```

```
typedef struct {
  unsigned int LATE0 : 1;
  unsigned int LATE1 : 1;
  unsigned int LATE2 : 1;
  unsigned int LATE3 : 1;
  unsigned int LATE4 : 1;
  unsigned int LATE5 : 1;
  unsigned int LATE6 : 1;
  unsigned int LATE7 : 1;
} __LATEbits_t;
```

Sendo a instanciação destas estruturas:

```
__PORTEbits_t PORTEbits;
__LATEbits_t  LATEbits;
```

[1] A forma como são declaradas as estruturas de dados que definem campos do tipo bit depende do compilador usado (a que se apresenta é a usada pelo compilador usado nas aulas práticas, pic32-gcc).

Do mesmo modo que se fez anteriormente para o registo **TRISE**, pode-se referenciar, de forma isolada, um porto I/O de 1 bit: usando a instância **PORTEbits** pode-se ler o valor de um porto de entrada; usando a instância **LATEbits** pode-se aceder ao flip-flop **LAT** do porto **E**, para ler ou para escrever. Por exemplo, para atribuir ao porto de saída **RB4**, o valor presente no porto de entrada **RE2** pode-se fazer:

```
LATBbits.LATB4 = PORTEbits.RE2; // atribui ao porto RB4 o valor lido do
                                // porto RE2
```

Na tradução para *assembly*, o compilador gera sequências do tipo "Read/Modify/Write", de modo a preservar o valor dos bits que não se pretendem alterar. Para o exemplo anterior, o compilador produz, tipicamente, a seguinte sequência de instruções (admitindo que o registo **$t0** já foi inicializado com os 16 bits mais significativos do endereço dos portos):

```
lw    $t2, PORTE($t0)
andi  $t2, $t2, 0x0004
sll   $t2, $t2, 2
lw    $t3, LATB($t0)
andi  $t3, $t3, 0x0FFEF
or    $t3, $t3, $t2
sw    $t3, LATB($t0)
```

A Figura 1 apresenta o diagrama de blocos de um porto de I/O de 1 bit no PIC32. Nesse esquema, para além dos registos **TRIS** e **LAT**, destacam-se ainda os dois flip-flops S1 e S2 presentes no caminho do porto para efeitos de leitura. Esses *flip-flops*, em conjunto, formam um circuito sincronizador que visa resolver os possíveis problemas causados por meta-estabilidade decorrentes do facto de o sinal externo ser assíncrono relativamente ao *clock* do CPU. Estes dois *flip-flops* impõem um atraso de, até, dois ciclos de relógio na propagação do sinal externo até ao barramento de dados do CPU ("data line").

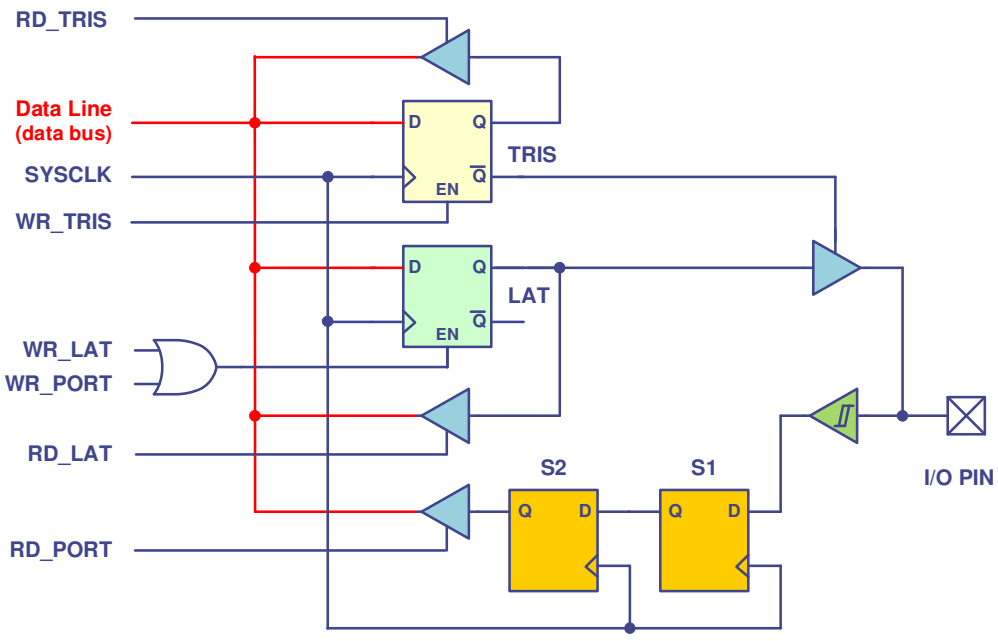

Figura 1. Diagrama de blocos simplificado de um porto de I/O no PIC32.

Para a manipulação dos valores a enviar para os portos configurados como saída devem sempre usar-se os registos **LATx** (ver explicação fornecida em anexo).

Exemplos:

a) Atribuição do valor '1' ao bit 3 do porto B:

```
LATBbits.LATB3 = 1;
```

b) Leitura do porto **RE2** (bit 2 do porto E) e escrita do seu valor, negado, no bit 5 do porto B:

```
LATBbits.LATB5 = !PORTEbits.RE2;
```

c) Inversão do valor de um porto de saída (por exemplo bit 0 do porto D):

```
LATDbits.LATD0 = !LATDbits.LATD0;
```

A forma como as estruturas de dados estão organizadas permite também o acesso a um dado registo (para ler ou escrever) tratando-o como uma variável de tipo inteiro, i.e., 32 bits (a descrição da estrutura feita acima não contempla esta possibilidade). Por exemplo, a configuração dos portos **RE3** a **RE1** como saída, e do porto **RE0** como entrada pode-se fazer do seguinte modo:

```
TRISE = (TRISE & 0xFFF0) | 0x0001; // RE3 a RE1 configurados como saídas
                                   // RE0 configurado como entrada
```

Do mesmo modo, se se pretender alterar os portos **RE3** e **RE2**, colocando-os a 1 e 0, respetivamente, sem alterar o valor de **RE1** (nem qualquer outro dos restantes), pode-se fazer[2]:

```
LATE = (LATE & 0xFFF3) | 0x0008;   // RE3=1; RE2=0; RE1 mantém o valor
```

Ficheiro detpic32.h

As declarações de todas as estruturas, bem como as respetivas instanciações, estão já feitas no ficheiro "**p32mx795f512h.h**" disponibilizado pelo fabricante, que é automaticamente incluído pelo ficheiro "**detpic32.h**". Logo, este último ficheiro deve ser incluído em todos os programas a escrever em linguagem C para a placa DETPIC32. Nesse ficheiro estão declaradas estruturas de dados para todos os registos de todos os portos do PIC32, bem como para todos os registos de todos os outros periféricos. Estão também feitas as necessárias associações entre os nomes das estruturas de dados que representam esses registos e os respetivos endereços de acesso.

Está igualmente definida no ficheiro "**detpic32.h**" a frequência de funcionamento do *core* MIPS da placa DETPIC32 (previamente configurada para 40MHz):

```
#define FREQ 40000000   // 40 MHz
```

É boa prática de programação usar o símbolo **FREQ** em vez de usar a constante **40000000** (ou usar **FREQ/2** em vez de **20000000**) diretamente no código C. Deste modo bastará recompilar o código se algum dia a frequência do *core* for alterada (a frequência máxima possível, na versão usada na placa DETPIC32, é 80MHz). O símbolo **PBCLK** (que é igual a **FREQ/2**), também está definido:

```
#define PBCLK  (FREQ / 2)
```

[2] O compilador gcc permite especificar constantes em binário, usando o prefixo 0b. Por exemplo, 0x13 é o mesmo que 0b10011. Em alguns casos, especificar as contantes em binário (desde que não tenham muitos bits!) pode tornar o programa mais fácil de entender.

Notas importantes:

- A escrita num porto configurado como entrada não tem qualquer consequência. O valor é escrito no *flip-flop* LAT associado ao porto mas não fica disponível no exterior uma vez que é barrado pela porta *tri-state* que se encontra na saída e que está em alta impedância (ver Figura 1).
- A configuração como saída de um porto que deveria estar configurado como entrada (e que tem um dispositivo de entrada associado) pode, em algumas circunstâncias, destruir esse porto. É, assim, muito importante que a configuração dos portos seja feita com grande cuidado.
- Após um *reset* (ou após *power-up*) os portos do PIC32 ficam todos configurados como entradas.

Trabalho a realizar

Parte I

O objetivo do programa seguinte é fazer o *toggle* do bit 14 do porto C, ao qual está ligado um LED na placa DETPIC32-IO[3], a uma frequência de 1 Hz (usando a função **`delay()`** já apresentada na aula anterior):

```
#include <detpic32.h>

int main(void)
{
  LATCbits.LATC14 = 0;    // The initial value should be set
                          //  before configuring the port as output
  TRISCbits.TRISC14 = 0;  // RC14 configured as output
  while(1)
  {
    delay(500);           // half period = 0.5s
    LATCbits.LATC14 = !LATCbits.LATC14;
  }
  return 0;
}
```

1. Edite, compile e teste o programa anterior. Note:
   - o nome do ficheiro não pode ter espaços ou carateres especiais
   - o nome do ficheiro tem que ter a extensão ".c" (exemplo: **`prog.c`**)
   - a compilação é feita através do comando: **`pcompile nome_ficheiro.c`**
2. Implemente, em linguagem C, um contador crescente, atualizado a uma frequência de 4Hz. O resultado deverá ser observando nos 4 LEDs já montados na sua placa e ligados aos portos **`RE0`** a **`RE3`**.

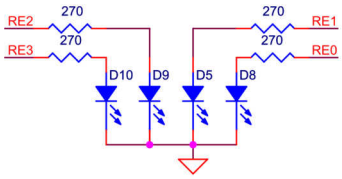

Figura 2. Ligação de 4 LEDs a portos do PIC32.

3. Pretende-se agora interagir com o sistema de visualização baseado em dois *displays* de 7 segmentos (Figura 3).

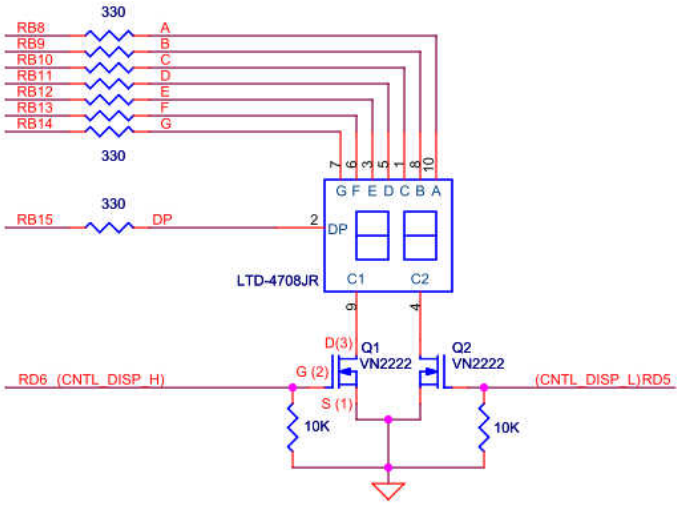

Figura 3. Ligação de dois *displays* de 7 segmentos ao porto B do PIC32.

[3] Na placa DETPIC32 deve usar o porto RD0.

Para isso, faça um programa que configure os portos **RB8** a **RB15**, **RD5** e **RD6** como saídas, que selecione apenas o "display low" (**RD5=1**, i.e. "**CNTL_DISP_L**"=1, e **RD6=0**) e que, em ciclo infinito, execute as seguintes tarefas:

- Ler um caracter do teclado e esperar que seja digitada uma letra entre 'a' e 'g' (ou 'A' e 'G'). Use o *system call* **getChar()**.
- Escrever no porto B a combinação binária que ative apenas o segmento do *display* correspondente ao caracter lido; note que a Figura 3 contém a informação de qual o porto que corresponde a cada segmento: por exemplo, o segmento A está ligado ao porto **RB8**.

Teste o programa para todos os segmentos e repita o procedimento para o "display high" (**RD6=1** e **RD5=0**).

4. Selecionando em sequência o "display low" e o "display high" envie para os portos **RB8** a **RB14**, em ciclo infinito e com uma frequência de 2 Hz, a sequência binária que ativa os segmentos do *display* pela ordem a, b, c, d, e, f, g, a, ...; o período de 0.5s deve ser obtido através da função **delay()**.

```
int main(void)
{
   unsigned char segment;
   LATDbits.LATD6 = 1; // display high active
   LATDbits.LATD5 = 0; // display low inactive
   // configure RB8-RB14 as outputs
   // configure RD5-RD6 as outputs
   while(1)
   {
      LATDbits.LATD6 = !LATDbits.LATD6;  //
      LATDbits.LATD5 = !LATDbits.LATD5;  // toggle display selection
      segment = 1;
      for(i=0; i < 7; i++)
      {
         // send "segment" value to display
         // wait 0.5 second
         segment = segment << 1;
      }
   }
   return 0;
}
```

5. A inicialização dos portos **RD6** e **RD5** (a '1' e a '0', respetivamente) e a inversão da seleção do *display* também podem ser feitas do modo que a seguir se indica. Analise as duas linhas de código e tire conclusões.

```
LATD = (LATD & 0xFF9F) | 0x0040; // display high active, low inactive
LATD = LATD ^ 0x0060;            // toggle display selection
```

6. Aumente a frequência para 10 Hz, 50 Hz e 100 Hz e observe, para cada uma destas frequências, o comportamento do sistema.
7. Construa a tabela que relaciona as combinações binárias de 4 bits (dígitos 0 a F) com o respetivo código de 7 segmentos, de acordo com o circuito montado no ponto anterior e com a definição gráfica dos dígitos apresentada na Figura 4.

```
display7Scodes[] = {0x3F, 0x06, 0x5B,...};
```

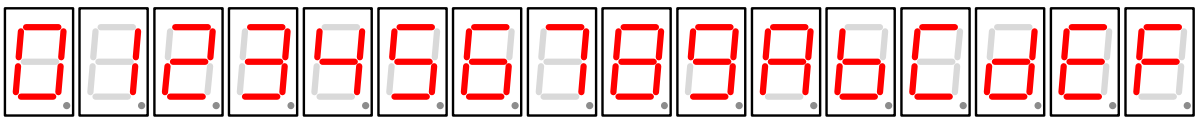

Figura 4. Representação dos dígitos de 0 a F no *display* de 7 segmentos.

8. Escreva um programa que leia o valor do *dip-switch* de 4 bits (Figura 5), faça a conversão para o código de 7 segmentos respetivo e escreva o resultado no *display* menos significativo (não se esqueça de configurar previamente os portos **`RB0`** a **`RB3`** como entradas).

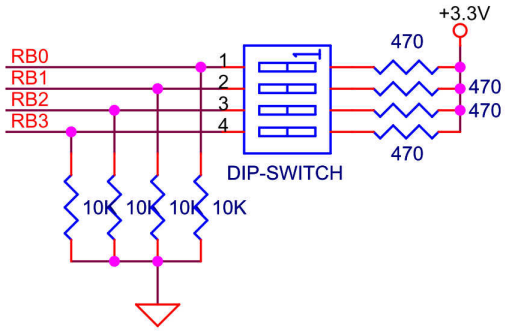

Figura 5. *Dip-switch* de 4 posições ligado a 4 bits do porto B.

```
int main(void)
{
   static const char display7Scodes[] = {0x3F, 0x06, 0x5B,...};
   // configure RB0 to RB3 as inputs
   // configure RB8 to RB14 and RD5 to RD6 as outputs
   // Select display low
   while(1)
   {
      // read dip-switch
      // convert to 7 segments code
      // send to display
   }
   return 0;
}
```

9. Altere o programa anterior de modo a mostrar o valor lido do *dip-switch* no *display* mais significativo.

Parte II

1. O programa desenvolvido nos pontos anteriores permite enviar 4 bits – um caracter hexadecimal – para um dos *displays*. Escreva agora uma função que envie um byte (8 bits) ou seja dois algarismos hexadecimais para os dois *displays*, fazendo corresponder os 4 bits menos significativos ao *display low* e os 4 bits mais significativos ao *display high*.

```
void send2displays(unsigned char value)
{
   static const char display7Scodes[] = {0x3F, 0x06,0x5b,...};
   // send digit_high (dh) to display_high: dh = value >> 4
   // send digit_low (dl) to display_low:   dl = value & 0x0F
}
```

2. Escreva um programa que implemente um contador binário de 8 bits. O contador deve ser incrementado com uma frequência de 5 Hz e o seu valor deve ser enviado, ao mesmo ritmo, para os *displays* através da função **`send2displays()`** escrita no ponto anterior. Utilize a função **`delay()`** para gerar um atraso de 200 ms e dessa forma determinar a frequência de incremento/visualização.
3. Como pode observar, o sistema de visualização apresenta um comportamento bastante deficiente, aparecendo um dos *displays* com um brilho muito reduzido (quase apagado).

Com a configuração usada, é necessário enviar de forma alternada os valores para os dois *displays*. Se o tempo de ativação dos dois displays não for igual, o brilho exibido por cada um deles será também diferente (o que está menos tempo ativo terá um brilho inferior).

a. Reescreva a função **send2displays()** de modo a que, sempre que for invocada, envie, de forma alternada, apenas um dos dois dígitos para o sistema de visualização. Isto é, em chamadas sucessivas à função, o comportamento deverá ser: enviar "dígit_low", enviar "dígit_high", enviar "digit_low", enviar "dígit_high", ...

```
void send2displays(unsigned char value)
{
   static const char display7Scodes[] = {0x3F, 0x06, 0x5b,...};
   static char displayFlag = 0; // static variable: doesn't loose its
                                // value between calls to function
   digit_low = value & 0x0F;
   digit_high = value >> 4;
   // if "displayFlag" is 0 then send digit_low to display_low
   // else send digit_high to didplay_high
   // toggle "displayFlag" variable
}
```

b. Será ainda necessário aumentar a frequência de trabalho do processo de visualização de modo a que o olho humano não detete a alternância na seleção dos *displays*. Assim, de modo a melhorar o desempenho do sistema de visualização, teremos que i) garantir que o tempo de ativação dos dois *displays* é o mesmo e ii) aumentar a frequência de refrescamento do sistema de visualização.

Reescreva o programa principal, tal como se esquematiza abaixo, de modo a invocar a função **send2displays()** com uma frequência de 20 Hz (i.e., a cada 50 ms), continuando a usar a função **delay()** para determinar as frequências de refrescamento (20 Hz) e de contagem (5 Hz).

```
int main(void)
{
   // declare variables
   // initialize ports
   counter = 0;
   while(1)
   {
      i = 0;
      do
      {
         send2displays( counter );
         // wait 50 ms
      } while(++i < 4);
      // increment counter (mod 256)
   }
   return 0;
}
```

ou:

```
int main(void)
{
   unsigned int i;
   ...
   counter = 0;
   i = 0;
   while(1)
   {
      i++;
      send2displays(counter);
      // wait 50 ms
      if(i % 4 == 0)
         // increment counter
   }
   return 0;
}
```

4. Com as alterações introduzidas no ponto anterior, o brilho de cada um dos dois *displays* ficou equilibrado. Continua, contudo, a notar-se a comutação entre os dois *displays,* efeito que é comum designar-se por *flicker*. De modo a diminuir, ou mesmo eliminar, o *flicker*, a frequência de refrescamento (*refresh rate*) tem que ser aumentada (no programa anterior era efetuada uma atualização dos *displays* a cada 50 ms, ou seja, o mesmo *display* é atualizado de 100 em 100 ms).

   Assim, mantendo a frequência de atualização do contador em 5Hz, altere o programa anterior de forma a aumentar a frequência de refrescamento para 50 Hz (20 ms) e depois para 100 Hz (10 ms). Observe os resultados num e noutro caso.

Parte III

1. Utilize o osciloscópio para visualizar os dois sinais de seleção dos *displays* ("**`CNTL_DISP_H`**" e "**`CNTL_DISP_L`**"). Meça o tempo de ativação desses sinais para as frequências de refrescamento de 50 e 100 Hz.
2. Mantendo a frequência de refrescamento em 100 Hz, altere o programa anterior de modo a incrementar o contador em módulo 60. A frequência de incremento deverá ser 1 Hz e os valores devem ser mostrados em **decimal**. A conversão para decimal pode ser feita, de forma simplificada e desde que o valor de entrada seja representável em decimal com dois dígitos (**`00-99`**), pela seguinte função:

```
unsigned char toBcd(unsigned char value)
{
   return ((value / 10) << 4) + (value % 10);
}
```

3. Altere o programa anterior de modo a que quando a contagem dá a volta (isto é, quando o valor do contador volta a zero), o valor **`00`** fique a piscar (meio segundo *ON*, meio segundo *OFF*) durante 5 segundos, antes da contagem ser retomada.

ANEXO: *Porquê usar LAT para escrever no porto e não PORT*

Os dois *flip-flops* do porto de entrada impõem um atraso de, até, dois ciclos de relógio na propagação do sinal externo até ao barramento de dados do CPU ("data line"). O mesmo atraso de 2 ciclos de relógio acontece na leitura do registo **LAT** de um porto configurado como saída, usando para acesso o endereço **PORT**. Assim, na situação em que o porto está configurado como saída, a leitura do registo **LAT** usando o endereço **PORT** é possível, mas o atraso de 2 ciclos de relógio impõe alguns cuidados na forma como se escreve o código. Vejamos o seguinte exemplo (que pressupõe que o porto **RE0** já está devidamente configurado como saída):

```
lw    $t0,PORTE($a0) # RD PORT
ori   $t0,0x0001
sw    $t0,PORTE($a0) # RE0 = 1
...                  # duas ou mais instruções
lw    $t0,PORTE($a0) # RD PORT
andi  $t0,0xFFFE
sw    $t0,PORTE($a0) # RE0 = 0
lw    $t1,PORTE($a0) # RD PORT: lê o valor 1, mas devia ler 0
```

Esta sequência de código escreve o valor 1 no porto **RE0**, a seguir escreve o valor 0 e, finalmente, lê o valor do porto **RE0** para o registo **$t1**. O valor lido para o registo **$t1** deveria ser 0 (i.e., o último valor escrito em **RE0**), mas será 1, ou seja, o valor que o porto apresentava antes da última operação de escrita.

Para que a última leitura do porto produza o resultado esperado, é necessário compensar o atraso, de dois ciclos de relógio, introduzido pelo *shift-register* (constituído pelos *flip-flops* S1 e S2), na leitura do valor à saída do registo **LAT** (não esquecer que o MIPS inicia a execução de uma nova instrução a cada ciclo de relógio). Ou seja, é necessário separar operações consecutivas de escrita e de leitura do porto de dois ciclos de relógio, o que pode ser feito através da introdução de duas instruções **nop**, tal como se apresenta de seguida:

```
lw    $t0,PORTE($a0) # RD PORT
ori   $t0,0x0001
sw    $t0,PORTE($a0) # RE0 = 1
...                  # duas ou mais instruções
lw    $t0,PORTE($a0) # RD PORT
andi  $t0,0xFFFE
sw    $t0,PORTE($a0) # RE0 = 0
nop                  # compensa o atraso de 2 ciclos
nop                  # de relógio introduzido pelo shift register
lw    $t1,PORTE($a0) # RD PORT
```

A alternativa à introdução das duas instruções **nop** é usar o registo **LAT** para a manipulação dos portos configurados como saída. Nesse caso o código ficaria:

```
lw    $t0,LATE($a0)  # RD LAT
ori   $t0,0x0001
sw    $t0,LATE($a0)  # RE0 = 1
...                  # zero ou mais instruções
lw    $t0,LATE($a0)  # RD LAT
andi  $t0,0xFFFE
sw    $t0,LATE($a0)  # RE0 = 0
lw    $t1,LATE($a0)  # RD LAT (o bit 0 de $t1 é 0)
```

Esta solução funciona porque o valor escrito em **LATE** num ciclo de relógio fica disponível para ser lido, através do endereço **LAT**, no ciclo de relógio seguinte, como se pode facilmente verificar no esquema da Figura 1.

Quando a programação é feita em linguagem C, e uma vez que o programador não controla a forma como o código é gerado, devem sempre usar-se os registos **LATx** para a manipulação dos valores em portos de saída.

Exemplo 1:

A sequência da esquerda usa o **PORTx** para manipular um porto de saída, mas o resultado final não é o esperado. A sequência da direita corrige o problema, ao acrescentar 2 **nop** entre a escrita do porto e a sua leitura.

```
#include <detpic32.h>

int main(void)
{
   PORTDbits.RD0 = 1;
   TRISDbits.TRISD0=0;    // output

   PORTDbits.RD0 = 0;
   PORTDbits.RD0 = !PORTDbits.RD0;
   return 0;
}
```

```
#include <detpic32.h>

int main(void)
{
   PORTDbits.RD0 = 1;
   TRISDbits.TRISD0=0;    // output

   PORTDbits.RD0 = 0;
   asm volatile("nop"); // inline
   asm volatile("nop"); // assembly
   PORTDbits.RD0 = !PORTDbits.RD0;
   return 0;
}
```

Verifique experimentalmente as duas sequências de código (o LED da placa está ligado ao porto **RD0** e pode ser usado para verificar o valor de saída do porto).

Exemplo 2:

A sequência de código seguinte usa **LATx** para manipular o porto de saída. O resultado é o esperado.

```
#include <detpic32.h>

int main(void)
{
   LATDbits.LATD0 = 1;
   TRISDbits.TRISD0 = 0;    // output

   LATDbits.LATD0 = 0;
   LATDbits.LATD0 = !LATDbits.LATD0;
   return 0;
}
```

Verifique experimentalmente o funcionamento desta sequência de código.