Aula prática N.º 4

Objetivos

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

Introdução

A configuração e utilização dos portos de I/O, 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 pic32-gcc, usado nas aulas práticas).

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** foi previamente inicializado com o endereço-base dos portos: **0xBF880000**):

```
lw   $t2, PORTE($t0)   # Lê PORTE para $t2
andi $t2, $t2, 0x0004  # Isola bit2 (coloca restantes bits a 0)
sll  $t2, $t2, 2       # Desloca bit da posição 2 para a posição 4
lw   $t3, LATB($t0)    # Lê registo LATB para $t3
andi $t3, $t3, 0x0FFEF # Clear bit 4 (não altera bits restantes)
or   $t3, $t3, $t2     # Atualiza bit 4 com o valor lido de PORTE
sw   $t3, LATB($t0)    # Escreve resultado no registo LATB
```

Ficheiro detpic32.h

As declarações de todas as estruturas, bem como as respetivas instanciações, estão já feitas no ficheiro "**p32mx795f512h.h**" disponibilizado pelo fabricante[2]. Nesse ficheiro estão declaradas estruturas de dados, semelhantes às descritas na secção anterior, 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.

O ficheiro "**p32mx795f512h.h**" é automaticamente incluído pelo ficheiro "**detpic32.h**", pelo que apenas este último deve ser incluído em todos os programas a escrever em linguagem C para a placa DETPIC32. No ficheiro "**detpic32.h**" estão também declarados os protótipos das funções de chamada aos *system calls* e estão igualmente definidas as frequências de funcionamento do CPU MIPS e do barramento de periféricos do PIC32 (Peripheral Bus Clock), ambas previamente configuradas por hardware para 40MHz e 20 MHz, respetivamente:

```
#define FREQ 40000000      // CPU frequency: 40 MHz
#define PBCLK  (FREQ / 2)  // Peripheral Bus Clock
```

É boa prática de programação usar o símbolo **PBCLK** em vez de usar a constante **20000000** (ou usar **FREQ** em vez de **40000000**) diretamente no código C. Deste modo, se a frequência do *core* for alterada basta recompilar o código para que o programa fique funcional (a frequência máxima possível, na versão usada na placa DETPIC32, é 80MHz).

Portos de I/O no PIC32

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

[2] Note que a designação utilizada pelo fabricante para os campos da estrutura **PORTxbits** (como **RB0**, **RE0**, …) não segue a mesma lógica aplicada aos campos das estruturas **LATxbits** (como **LATB0**, **LATE0**, …) e **TRISxbits** (como **TRISB0**, **TRISE0**, …).

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").

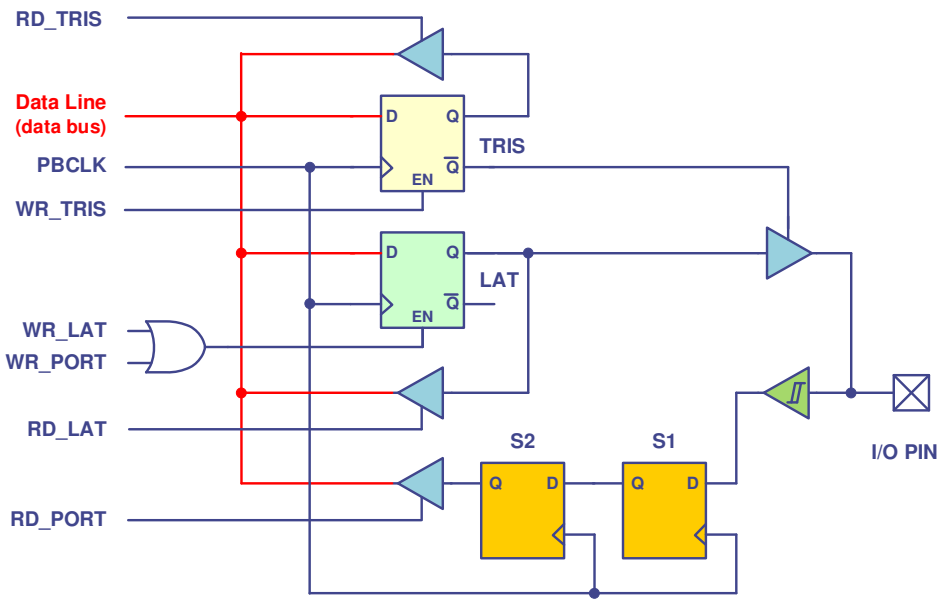

Figura 1. Diagrama de blocos simplificado de um porto de I/O no PIC32.

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

Exemplos:

a) Atribuição do valor lógico **`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, RE2 e RE1 configurados como saídas
                                  // RE0 configurado como entrada
```

Do mesmo modo, caso se pretenda alterar o valor dos portos **`RE3`** e **`RE2`**, colocando-os a **`1`** e **`0`**, respetivamente, sem alterar o valor dos restantes, pode-se fazer:

```
LATE = (LATE & 0xFFF3) | 0x0008;   // RE3=1; RE2=0
```

Este tipo de sequência não pode ser separado em duas operações independentes (*reset* dos bits a alterar e atualização); isso teria como consequência a escrita transitória do valor lógico **`0`** nos portos (**`RE2`** e **`RE3`** no exemplo anterior).

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.
- É obrigatório configurar, como entrada ou como saída, todos os portos do PIC32 que forem usados na aplicação.
- A escrita direta nos registos associados aos portos do PIC32 (configuração e dados) é obrigatoriamente feita com uma sequência do tipo "Read-Modify-Write". Nesse acesso, apenas os bits usados podem ser alterados.

Trabalho a realizar

Parte I

O objetivo do programa seguinte é piscar o bit 14 do porto C (**RC14**), ao qual está ligado um LED na placa DETPIC32, a uma frequência de 1 Hz. Para a temporização pode utilizar-se a função **delay()** já apresentada anteriormente:

```
#include <detpic32.h>

int main(void)
{
   // Configure port RC14 as output
   while(1)
   {
      // Wait 0.5s
      LATC = LATC ^ ????;  // Toggle RC14 port value
   }
   return 0;
}
```

1. Complete, e teste o programa anterior. Note:
   - o nome do ficheiro não pode ter espaços ou caracteres especiais
   - o nome do ficheiro tem que ter a extensão "**.c**" (por exemplo: **prog.c**)
   - a compilação é feita através do comando: **pcompile nome_ficheiro**
2. Implemente, em linguagem C, um contador crescente módulo 10, atualizado a uma frequência de 4.6Hz. O resultado deverá ser observando nos 4 LED ligados aos portos **RE6** a **RE3**.

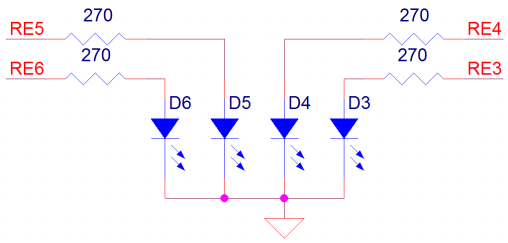

Figura 2. Ligação de 4 LED a portos do PIC32.

Estrutura do programa:

1) Configurar os portos **RE6–RE3** como saídas:

```
TRISE = TRISE & 0xFF87;
```

2) Inicializar a variável do contador:

```
counter = 0
```

3) Atualizar os portos de saída (**RE6–RE3**) com o valor da variável do contador:

```
LATE = (LATE & 0xFF87) | counter << 3;
```

4) Esperar 1/4,6 s[3]:

```
resetCoreTimer(); while( readCoreTimer() < 4347826 );
```

5) Incrementar a variável do contador, módulo 10:

```
counter = (counter + 1) % 10;
```

6) Repetir desde 3.

[3] Consulte o Anexo 1 - Considerações sobre o uso de operações em vírgula flutuante em microcontroladores.

3. Repita o exercício anterior para um contador decrescente módulo 10, atualizado a uma frequência de 2.7Hz.

   Para o decremento do contador (módulo 10) poderá fazer:

```
counter = counter > 0 ? counter - 1 : 9;
```

   ou

```
counter = (counter + 9) % 10;
```

Parte II

Pretende-se agora interagir com o sistema de visualização que está implementado na placa DETPIC32. Esse sistema de visualização é constituído por dois *displays* de 7 segmentos, que estão montados no mesmo componente físico.

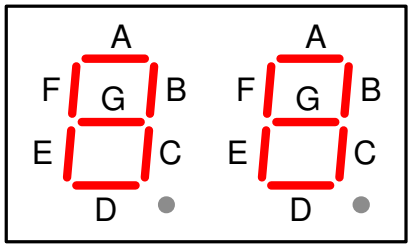

Figura 3. Designação de cada um dos segmentos de um *display* de 7 segmentos.

Apesar de o componente ser constituído por dois *displays*, existe apenas um conjunto de entradas "A" a "G", que liga internamente às entradas correspondentes dos dois *displays*, tal como esquematizado na Figura 4. A ativação de cada um dos *displays* é feita através das entradas CH e CL, que funcionam como um *enable*, ativo em lógica negativa. Ou seja, para que um *display* esteja ativo é necessário que o respetivo *enable* (CH ou CL) tenha o valor lógico 0.

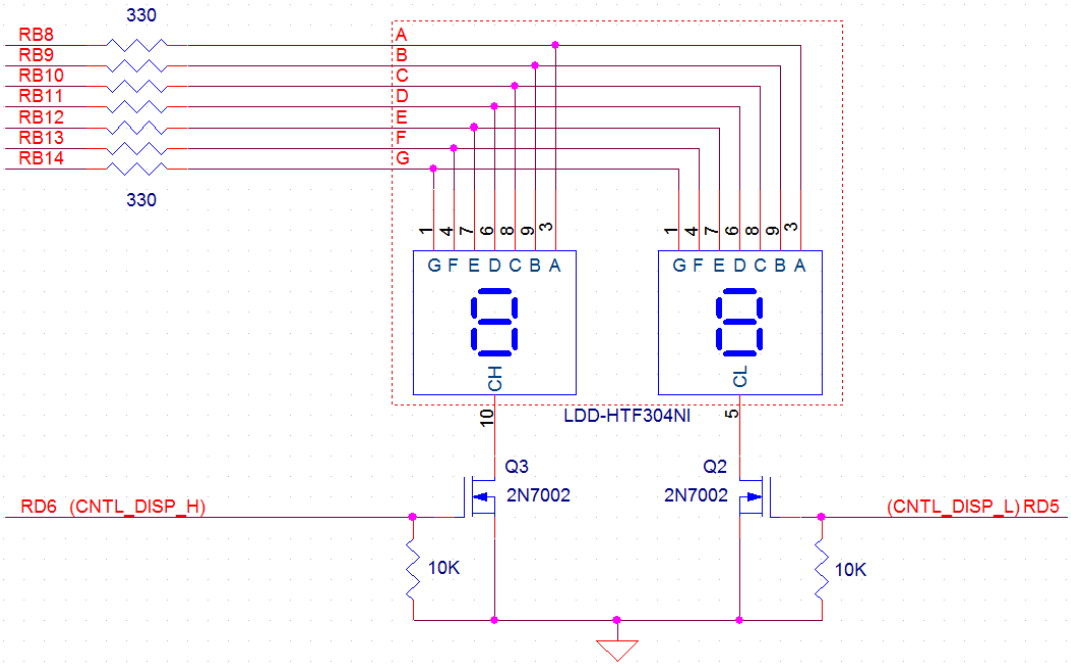

Figura 4. Ligação de um *display* de 7 segmentos duplo ao porto B do PIC32.

Os dois transístores (Q2 e Q3) funcionam como inversores, o que significa que para se ativar o *display* mais significativo é necessário atribuir o valor lógico 1 ao porto **`RD6`** (e 0 ao porto **`RD5`**); para se ativar o *display* menos significativo é necessário atribuir o valor lógico 1 ao porto **`RD5`** (e 0 ao porto **`RD6`**).

1. Escreva um programa que configure os portos **RB8** a **RB14**, **RD5** e **RD6** como saídas, que selecione apenas o *display* menos significativo (**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'. Use o *system call* **getChar()**.
   - Escrever no porto B a combinação binária que ligue apenas o segmento do *display* correspondente ao caracter lido (e apague todos os restantes); na Figura 4 pode ver o porto a que está ligado cada um dos segmentos (por exemplo, o segmento A está ligado ao porto **RB8)**.

     ```
     ...
     if(ch == 'a') LATB = (LATB & 0x80FF) | (1 << 8);
     if(ch == 'b') LATB = (LATB & 0x80FF) | ???;
     ...
     if(ch == 'g') LATB = (LATB & 0x80FF) | ???;
     ```

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

2. A atualização do porto B no exercício anterior pode ser escrita de uma forma mais compacta. Complete o código seguinte e teste-o.

   ```
   ...
   if(ch >= 'a' && ch <= 'g') {
      ch = ch – 'a';
      LATB = (LATB & 0x80FF) | 1 << (ch + ???);
   }
   ...
   ```

3. Selecionando em sequência o *display* menos significativo e o *display* mais significativo 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, ...; a frequência de 2 Hz deve ser controlada usando a função **delay()**.

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

4. Se, no exercício anterior, acedeu aos portos **RD6** e **RD5** usando estruturas (e.g. **LATDbits.LATD5=1**), faça a inicialização dos portos **RD6** e **RD5** (a 1 e a 0, respetivamente) e a inversão da seleção do *display* usando diretamente o registo, em conjunto com operações lógicas e máscaras (e.g. **LATD=(LATD & 0x????) | 0x????**).

5. Aumente a frequência para 10 Hz, 50 Hz e 100 Hz e observe, para cada uma destas frequências, o comportamento do sistema.

6. 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 5. Com os códigos de 7 segmentos de cada um dos 16 dígitos complete a declaração do *array* seguinte:

```
disp7Scodes[] = {0x3F, 0x06, 0x5B,...};
```

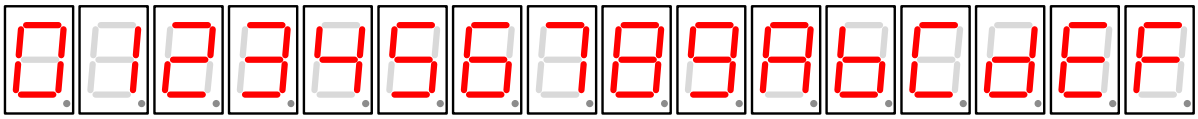

Figura 5. Representação dos dígitos de 0 a F no *display* de 7 segmentos.

7. Escreva um programa que leia o valor do *dip-switch* de 4 bits (ligado aos portos **RB3** a **RB0**, Figura 6), 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 **RB3** a **RB0** como entradas).

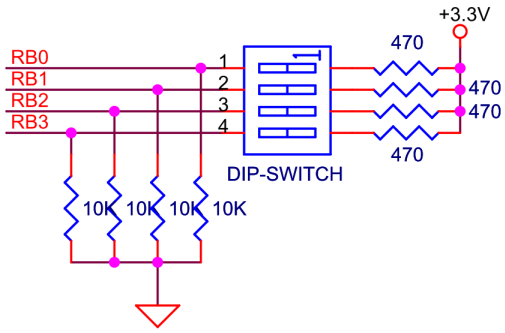

Figura 6. *Dip-switch* de 4 posições ligado a 4 bits do porto B.

```
int main(void)
{
   static const char disp7Scodes[] = {0x3F, 0x06, 0x5B,...};
   int dips, code;
   // configure RB0 to RB3 as inputs
   // configure RB8 to RB14 and RD5 to RD6 as outputs
   // Select display low
   while(1)
   {
      dips = PORTB & 0x????;        // read dip-switch (bits 3-0)
      code = disp7Scodes[???];      // convert to 7 segments code
      LATB = (LATB ...              // send code to display
   }
   return 0;
}
```

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

Elementos de apoio

- PIC32 Family Reference Manual, Section 12 – I/O Ports.
- PIC32MX5XX/6XX/7XX, Family Data Sheet, Pág. 159 a 164.
- *Slides* das aulas teóricas (aulas 2 a 4).

ANEXO 1 - *Considerações sobre o uso de operações em vírgula flutuante em microcontroladores*

Em microcontroladores de baixo custo e poder de cálculo limitado, tipicamente utilizados em sistemas embutidos, o uso de constantes fracionárias e/ou variáveis do tipo *float* ou *double* deve ser evitado, pelas seguintes razões:

1. **Desempenho**: A generalidade dos microcontroladores não dispõe de hardware dedicado para a realização de operações em vírgula flutuante. Por essa razão, operações envolvendo quantidades fracionárias são realizadas por software através de bibliotecas incluídas no processo de linkagem da aplicação final. Essas operações são, inevitavelmente, muito mais lentas do que operações sobre números inteiros, para as quais o microcontrolador dispõe, em geral, de hardware com capacidade para as realizar.

2. **Memória**: O processamento de operações com tipos de dados *float* ou *double*, requer mais memória do que operações sobre inteiros. Em microcontroladores com recursos limitados de memória isso pode ser um problema.

3. **Economia de energia**: Sistemas baseados em microcontroaldor são muitas vezes alimentados por bateria, ou dependentes de fontes de energia com capacidade limitada (e.g. *energy harvesting*). Devido à maior complexidade das operações em vírgula flutuante, o consumo de energia será mais elevado, baixando, consequentemente, a autonomia do sistema.

Por estas razões, em programas para microcontroladores (em especial para os de baixo poder de cálculo) devem ser utilizados, sempre que possível, tipos de dados inteiros otimizados para o hardware disponível. Maximiza-se, desse modo, o desempenho e a autonomia, diminuindo, simultaneamente, a necessidade de recursos de memória.

Na maioria dos casos, é possível substituir operações em vírgula flutuante por aritmética de vírgula fixa para representar números fracionários. Para isso usam-se inteiros em que um número fixo de bits/dígitos é usado para representar a parte fracionária e os restantes são usados para representar a parte inteira. Por exemplo, para representar números com duas casas decimais, os valores podem ser armazenados multiplicados por 100. Assim, 2.35 seria representado como 235, sendo os dois dígitos menos significativos usados para representar a parte decimal.

ANEXO 2: *Porquê usar LAT e não PORT para escrever num porto*

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, com um mínimo de dois ciclos de relógio, operações consecutivas de escrita e de leitura do porto, 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.

PDF criado em 24/02/2025