

## Aula prática N.º 1

### Objetivos

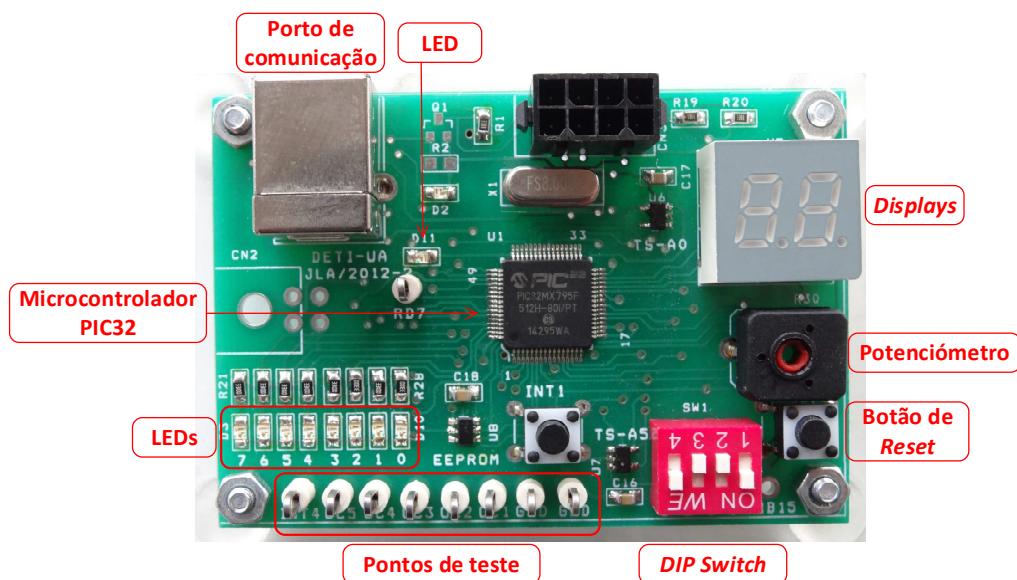
- Conhecer o processo de criação de um programa escrito em *assembly* para correr na placa DETPIC32: compilação, transferência e execução.
- Utilizar os *system calls* disponibilizados na placa DETPIC32.
- Rever os conceitos associados à manipulação de *arrays* de caracteres.

### Introdução

A realização de trabalhos práticos envolvendo o microcontrolador DETPIC32 requer a utilização de um conjunto de ferramentas de desenvolvimento que permitem editar código, compilar, transferir programas para a placa e interagir com o sistema em execução.

Para a edição do código fonte, podem ser usados diferentes editores de texto, como o VS Code, o gedit, o GVim ou o geany, entre outros.

Na fase de compilação, recorre-se a um *cross-compiler / cross-assembler*, concretamente o gcc com *back-end* para MIPS (**gcc-pic32**), responsável por gerar ficheiros essenciais, como o **".hex"**, que contém o programa final a ser carregado no microcontrolador, e o **".map"**, que descreve a utilização da memória e os símbolos do programa.



**Figura 1. Placa DETPIC32 a usar nas aulas práticas.**

Para além destas ferramentas genéricas, existem ainda utilitários desenvolvidos especificamente para a placa DETPIC32.

- O **bootloader**, previamente gravado na memória *boot flash* do PIC32, permite receber programas através de um porto de comunicação e armazená-los na memória *flash*.
- Associado ao *bootloader*, o **ldpic32** é o programa que realiza a transferência do ficheiro **".hex"** para a placa.
- A comunicação entre o utilizador e o microcontrolador é facilitada pelo **pterm**, um programa terminal que permite ler dados introduzidos através do teclado e mostrar informações e valores no ecrã, possibilitando assim a interação direta com o sistema e o acompanhamento em tempo real do seu comportamento (ver anexo 1 para uma explicação mais detalhada).

- Por fim, o **hex2asm** realiza o *disassemble* do ficheiro ".hex", convertendo o código binário em instruções Assembly legíveis. Para tornar o resultado mais comprehensível, o programa recorre ao ficheiro ".map" associado, permitindo dessa forma identificar de forma clara as secções de memória e os símbolos mais relevantes, facilitando a análise da estrutura do programa. Este programa é particularmente útil para detetar a origem de exceções em programas escritos em C, permitindo ao utilizador identificar a instrução que gerou a exceção e, desse modo, corrigir o programa.

A combinação destas ferramentas assegura todas as etapas do ciclo de desenvolvimento para a placa DETPIC32, desde a escrita do código até à análise detalhada da sua execução.

## Trabalho a realizar

### Parte I

- Utilizando um editor de texto, edite e grave o programa de demonstração *assembly* que é apresentado de seguida. Para facilitar a organização dos ficheiros dos vários programas que irão ser feitos ao longo do semestre, sugere-se que seja criado um diretório por aula prática. Para este exemplo o ficheiro poder-se-á chamar “**prog1.s**” (usa-se a extensão “**.s**” para ficheiros *assembly*) e deve ser colocado no diretório “**aula1**”.

```
#    int main(void)
#    {
#        printStr("AC2 - Aulas praticas\n");      // system call
#        return 0;
#    }
        .equ      PRINT_STR, 8
        .data
msg:   .asciiz   "AC2 - Aulas praticas\n"
        .text
        .globl main
main:  la      $a0,msg
        li      $v0,PRINT_STR
        syscall           # printStr("AC2 - Aulas praticas\n");
        li      $v0,0          # return 0;
        jr      $ra
```

- Compile o programa *assembly* anterior. Para isso abra um terminal e, no diretório onde guardou o ficheiro com o código fonte (**prog1.s**), execute o comando:

**pcompile prog1.s**

- O comando da linha anterior produz os seguintes ficheiros: “**prog1.o**”, “**prog1.elf**”, “**prog1.map**” e “**prog1.hex**”, sendo os dois primeiros ficheiros binários e os restantes de texto.
  - Observe o conteúdo do ficheiro “**prog1.hex**”; para isso abra-o com um editor de texto (gedit, gvim, geany, vscode, ...).
  - Execute, em linha de comando, o programa **hex2asm**:

**hex2asm prog1.hex** (produz o ficheiro “**prog1.hex.s**”)

De seguida abra, com um editor de texto, o ficheiro “**prog1.hex.s**”

- Identifique no ficheiro “**prog1.hex.s**” os endereços correspondentes aos *labels* **main** e **msg** do programa que editou.

- Transfira o programa “**prog1.hex**” para o microcontrolador da placa DETPIC32, realizando os seguintes passos:

- ligue a placa à porta USB do PC e execute o seguinte comando:

**ldpic32 prog1.hex** (a extensão **.hex** pode ser omitida)

- prima o botão de *reset* da placa DETPIC32 e aguarde que a transferência se processe

- Execute o programa transferido e observe o resultado; para isso:

- execute, em linha de comando, o programa **pterm**<sup>1</sup>
- prima novamente o botão de *reset* para iniciar a execução do programa.

---

<sup>1</sup> Os 3 comandos normalmente usados (**pcompile**, **ldpic32** e **pterm**) podem ser encadeados numa única linha de comando, do seguinte modo (usando como exemplo o ficheiro “**prog1.s**”):

**pcompile prog1.s && ldpic32 prog1 && pterm**

## Parte II

Os programas que se apresentam de seguida exercitam a utilização dos *system calls* disponíveis na placa DETPIC32. Verifique os *system calls* que pode usar, consultando a Tabela de Instruções Assembly do MIPS (versão para AC2), disponível no moodle, ou analisando o ficheiro `/opt/pic32mx/include/detpic32.h`. Analise a forma como cada um dos *system calls* deve ser invocado.

1. Identifique a funcionalidade de cada um dos programas que se seguem e traduza-os para *assembly* do MIPS, usando as convenções de passagem de parâmetros e salvaguarda de registos que estudou em AC1. Compile cada um dos programas *assembly*, usando o `pcompile`. Transfira o resultado da compilação (ficheiros `".hex"`) para a placa DETPIC32 (usando o `ldpic32`) e verifique o respetivo funcionamento.

- a) *System calls getChar()* e *putChar()*.

A *system call* `getChar()` é bloqueante, ou seja, só regressa ao programa chamador quando for premida uma tecla, retornando o respetivo código ASCII.

```
int main(void)
{
    char c;
    int cnt = 0;
    do
    {
        c = getChar();
        putChar( c );
        cnt++;
    } while( c != '\n' );
    printInt(cnt, 10);
    return 0;
}
```

Substitua a linha `putChar(c)` por `putChar(c+1)` e teste novamente o programa.

- b) *System call inkey()*.

A *system call* `inkey()` não é bloqueante, ou seja, se foi premida uma tecla devolve o respetivo código ASCII, mas se não foi premida qualquer tecla devolve o valor 0 (`0x00`).

```
int main(void)
{
    char c;
    int cnt = 0;
    while(1) {
        c = inkey();
        if( c == 'R' )
            cnt = 0;
        putChar('\r');
        printInt(cnt, 10 | 3 << 16); // ver nota 3 na página seguinte
        cnt = (cnt + 1) & 0xFF;
        wait(4);
    }
    return 0;
}

void wait(int ts)
{
    int i;
    for(i=0; i < 515000 * ts; i++); // wait approximately ts/10 seconds
}
```

- c) Teste dos *system calls* de leitura e impressão de inteiros.

```

int main(void)
{
    int value;
    while(1)
    {
        printStr("\nIntroduza um inteiro (sinal e módulo): ");
        value = readInt10();

        printStr("\nValor em base 10 (signed): ");
        printInt10(value);

        printStr("\nValor em base 2: ");
        printInt(value, 2);

        printStr("\nValor em base 2, formatado: ");
        printInt(value, 2 | 32 << 16);

        printStr("\nValor em base 16: ");
        printInt(value, 16);

        printStr("\nValor em base 10 (unsigned): ");
        printInt(value, 10);

        printStr("\nValor em base 10 (unsigned), formatado: ");
        printInt(value, 10 | 5 << 16); // ver nota 3
    }
    return 0;
}

```

## Notas

- O código *assembly* que escrever vai ser executado numa arquitetura *pipelined* de 5 fases com *delayed branches*. Ou seja, em todas as instruções que alteram o fluxo de execução (**breq**, **bne**, **j**, **jal**, **jr**, **jalr**) a instrução que vem imediatamente a seguir é sempre executada, independentemente do resultado (*taken/not taken*) da instrução de salto. Apesar disso, não é necessário ter em conta esse comportamento, uma vez que o *assembler* efetua, de forma automática, a reordenação das instruções de modo a preencher, sempre que possível, a *delayed slot*. Nos casos em que o *assembler* deteta que não pode reordenar as instruções devido a dependência(s) de dados, a *delayed slot* é preenchida com a instrução "**nop**". Este comportamento pode ser observado através da análise do ficheiro produzido pelo programa **hex2asm** (por exemplo "**prog1.hex.s**").
- Devido a limitações do compilador usado, nos programas escritos em *assembly* o *label* "**main**" deve ser o primeiro *label* do segmento de código, ou seja, o código das sub-rotinas deve vir a seguir ao código da função **main()**.
- O *system call* **printInt** permite especificar o número mínimo de dígitos com que o valor é impresso. Essa configuração é feita nos 16 bits mais significativos do registo usado para indicar a base da representação. Por exemplo, para a impressão do valor da variável **val** em binário com 4 bits, o valor a colocar no registo **\$a1** é **0x00040002**.

Em linguagem C: **printInt( val, 2 | 4 << 16 );**

Em *assembly*: **move \$a0,\$t0 # supondo que "val" reside em \$t0  
li \$a1,0x00040002  
li \$v0,6  
syscall**

- Em sistemas Linux, para visualizar a tabela ASCII, basta escrever o comando **ascii** na linha de comandos de um terminal.

## Exercícios adicionais

1. Utilização da system call **inkey()** na implementação de um contador up/down de 8 bits, módulo 256. O valor do contador é atualizado a cada 0.5s (aproximadamente) e é mostrado no ecrã, em decimal e em binário (com o system call **printInt()**). O estado "up" ou "down" do contador é assegurado por uma máquina de estados simples, com 2 estados, controlada pelas teclas '+' e '-'.

```
#define UP      1
#define DOWN    0

void wait(int);

int main(void)
{
    int state = UP;
    int cnt = 0;
    char c;
    do
    {
        putChar('\r');      // Carriage return character
        printInt( cnt, 10 | 3 << 16 );// 0x0003000A: decimal w/ 3 digits
        putChar('\t');      // Tab character
        printInt( cnt, 2 | 8 << 16 ); // 0x00080002: binary w/ 8 bits

        wait(5);           // wait 0.5s

        c = inkey();
        if( c == '+' )
            state = UP;
        else if( c == '-' )
            state = DOWN;

        if( state == UP )
            cnt = (cnt + 1) & 0xFF;      // Up counter MOD 256
        else
            cnt = (cnt - 1) & 0xFF;      // Down counter MOD 256
    } while( c != 'q' );
    return 0;
}

void wait(int ts)
{
    int i;
    for(i=0; i < 515000 * ts; i++); // wait approximately ts/10 seconds
}
```

- Traduza o programa para *assembly* do MIPS.
- Altere o código C do programa anterior de modo a adicionar a possibilidade de parar o contador (tecla '**s**') ou de reiniciar o seu valor (tecla '**R**'). Reflita essas alterações no programa *assembly* que escreveu na alínea anterior e teste o resultado na placa.

## 2. Contador com passo variável e *reset*

O objetivo deste exercício é implementar, em *assembly*, um contador cujo valor se mantém sempre no intervalo [0, 59], com comportamento circular (aritmética módulo 60).

O contador incrementa ou decrementa com um passo configurável pelo utilizador e pode ser colocado a zero a qualquer momento.

A frequência de incremento/decremento deve ser 5 Hz (aproximadamente).

No ecrã deve ser impresso, sempre na mesma linha, o valor atual do contador (formatado com 2 dígitos), o estado atual (*Up* ou *Down*) e o valor do passo de incremento/decremento.

Exemplo: **12 U 4**

O utilizador pode controlar o contador através do teclado, usando a *system call* **inkey()**:

- **Tecla '+'** → modo ***Up***: o contador incrementa pelo valor atual do passo.
- **Tecla '-'** → modo ***Down***: o contador decrementa pelo valor atual do passo.
- **Tecla 'R'** → coloca o contador a 0.
- **Teclas '1', '2', '4'** → definem o passo do contador para 1, 2 ou 4, respetivamente.

Sempre que o valor do contador ultrapasse os limites do intervalo [0, 59], deve ser aplicado o comportamento circular adequado.

Exemplos:

**Modo Up:** se o valor atual for **57** e o passo for **4**, então o próximo valor deve ser **1**:

$$(57 + 4) \% 60 = 1 \quad (4: \text{ valor do passo})$$

**Modo Down:** se o valor atual for **1** e o passo for **4**, então o próximo valor deve ser **57**:

$$(1 - 4 + 60) \% 60 = 57 \quad (4: \text{ valor do passo})$$

Para realizar este exercício pode usar como base o código apresentado no exercício 1.

O programa deve usar uma máquina de estados simples para gerir o modo de contagem(*up/down*) e o passo.

3. Manipulação de *strings*<sup>2</sup> e teste do *system call* **readStr()**.

```

#define SIZE 20

char *strcat(char *, char *);
char *strcpy(char *, char *);
int strlen(char *);

int main(void)
{
    static char str1[SIZE + 1];
    static char str2[SIZE + 1];
    static char str3[2 * SIZE + 1];

    printStr("Introduza 2 strings: ");
    readStr( str1, SIZE );
    readStr( str2, SIZE );
    printStr("Resultados:\n");
    printInt( strlen(str1), 10 );
    printInt( strlen(str2), 10 );
    strcpy(str3, str1);
    printStr( strcat(str3, str2) );
    printInt10( strcmp(str1, str2) );
    return 0;
}

// Returns the length of string "str" (excluding the null character)
int strlen(char *str)
{
    int len;
    for( len = 0; *str != '\0'; len++, str++ );
    return len;
}

// Copy the string pointed by "src" (including the null character) to
// destination (pointed by "dst")
char *strcpy(char *dst, char *src)
{
    char *p = dst;

    for( ; (*dst = *src) != '\0'; dst++, src++ );
    return p;
}

// Concatenates "dst" and "src" strings
// The result is stored in the "dst" string
char *strcat(char *dst, char *src)
{
    char *p = dst;

    for( ; *dst != '\0'; dst++ );
    strcpy( dst, src );
    return p;
}

```

<sup>2</sup> A versão do *assembler* que está a ser usada nas aulas práticas não interpreta corretamente o carácter de terminação das *strings*, '\0'; em *assembly* use, em vez desse carácter, o valor 0 (que é o código ASCII correspondente a '\0').

```
// Compares two strings, character by character
// Returned value is:
//   < 0  string "str1" is "less than" string "str2" (first
//         non-matching character in str1 is lower, in ASCII, than
//         that of str2)
//   = 0  string "str1" is equal to string "str2"
//   > 0  string "str1" is "greater than" string "str2" (first
//         non-matching character in str1 is greater, in ASCII, than
//         that of str2)

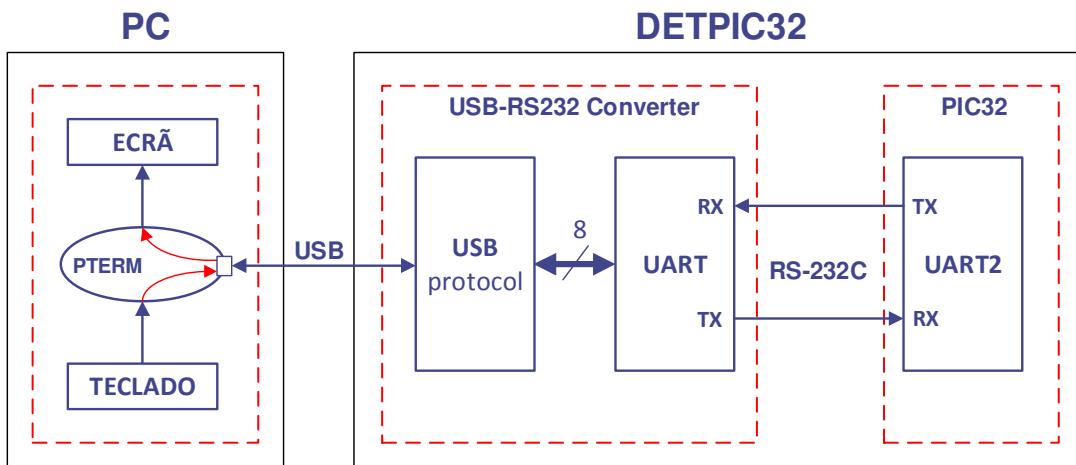
int strcmp(char *str1, char *str2)
{
    for( ; (*str1 == *str2) && (*str1 != '\0'); str1++, str2++ );
    return( *str1 - *str2 );
}
```

### ***Elementos de apoio***

- Tabela com o resumo do conjunto de instruções da arquitetura MIPS, na versão adaptada a Arquitetura de Computadores II (disponível no *moodle* de AC2).
- Slides das aulas teóricas de Arquitetura de Computadores I.
- David A. Patterson, John L. Hennessy, Computer Organization & Design – The Hardware/Software Interface, Morgan Kaufmann Publishers.

## Anexo 1

### Modelo de comunicação entre a placa DETPIC32 e o PC



**Figura 2. Diagrama de blocos simplificado do modelo de comunicação entre a placa DETPIC32 e o PC, usando RS-232C sobre USB.**

A Figura 2 descreve a arquitetura de comunicação entre um computador (PC) e a placa DETPIC32. O sistema permite que a porta USB funcione como se fosse uma porta série, simulando comunicação RS-232C entre o PC e o microcontrolador.

#### 1. O lado do PC

No computador, a interação com o utilizador é feita através da aplicação PTERM, designada por programa terminal. Utilizando esse programa, o utilizador introduz dados através do teclado, que são enviados para a porta de comunicação. Os dados recebidos dessa porta são apresentados no ecrã. O PTERM comunica via USB, que serve como meio físico para o transporte da informação.

#### 2. Conversão de protocolos

Como o microcontrolador PIC32 utiliza o protocolo RS-232C e o PC comunica via USB, existe um bloco intermédio na placa DETPIC32 designado por USB-RS232 Converter. Este módulo gera a receção e envio de pacotes de dados através do barramento USB e liga internamente a uma UART que implementa o protocolo RS232-C.

#### 3. O microcontrolador (PIC32)

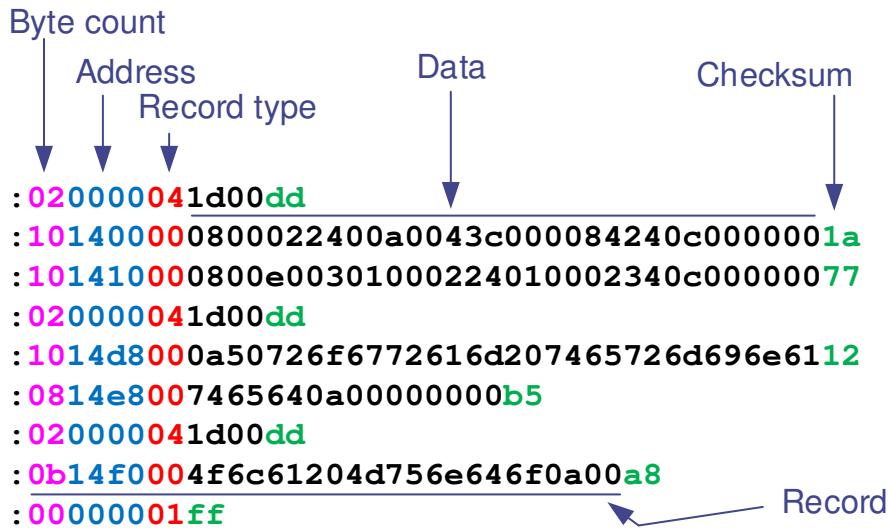
O destino (ou origem) da comunicação é o PIC32, especificamente o periférico que implementa o protocolo RS-232C (a UART2).

Na transmissão, o pino de TX do microcontrolador envia dados para o pino de receção (RX) do conversor. Na receção o pino de receção (RX) do microcontrolador recebe os dados provenientes do pino de transmissão (TX) do conversor.

Este sistema permite que o programador utilize funções simples de leitura e escrita série no PIC32 (como, por exemplo, as *system calls*) para trocar mensagens com o PC, usando USB como meio físico de interligação entre os dois sistemas.

## Anexo 2

### Formato Intel HEX



- Byte count:** número de bytes do campo de dados
- Address:** endereço de memória onde é armazenado o primeiro byte do campo de dados; o endereço efetivo é obtido em conjunto com um endereço base especificado anteriormente
- Record type:**
  - 00 - data record
  - 01 - end-of-file record
  - 04 - extended linear address record (especifica os 16 bits mais significativos do campo de endereço das linhas seguintes)
- Checksum:** complemento para dois dos 8 bits menos significativos resultantes da soma dos bytes do record; o *checksum* é usado para a deteção de possíveis erros na transmissão dos dados

#### Descodificação do exemplo:

:020000041d00dd

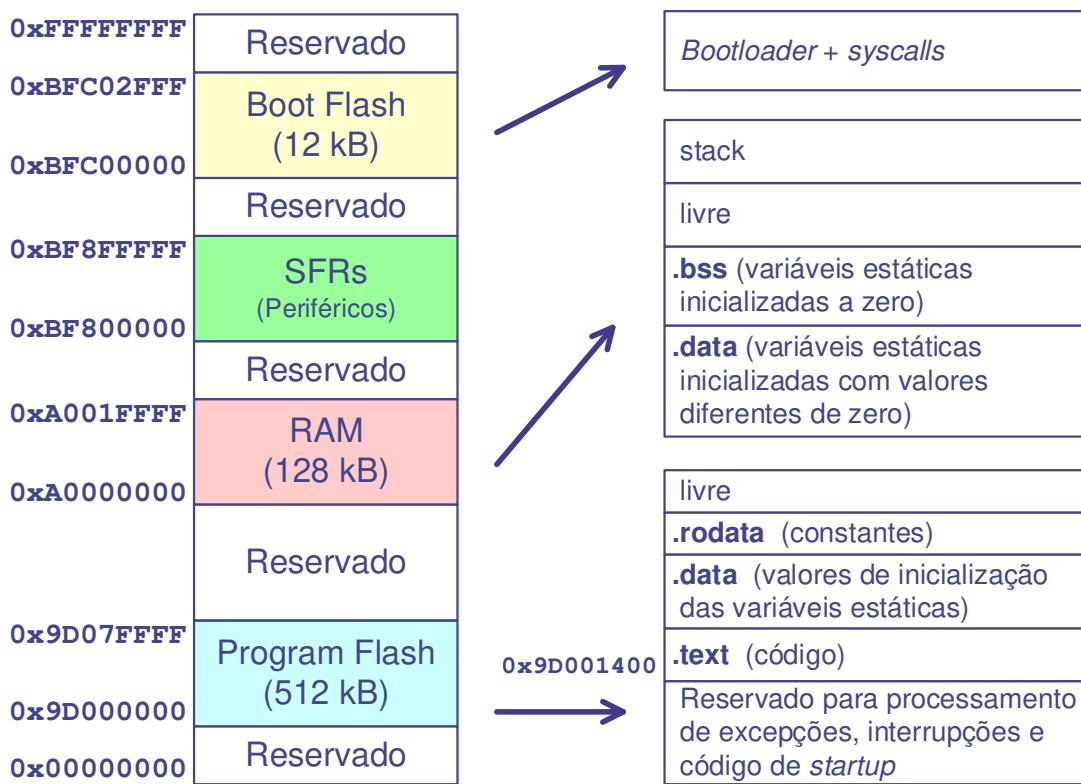
- 2 bytes no campo de dados
- record do tipo 4: o campo de dados contém os 16 bits mais significativos do endereço dos *records* seguintes, ou seja, **0x1D00**
- checksum: **0x100-trunc8 (02+00+00+04+1D+00)=0xDD**, ou seja, complemento para dois da soma, truncada a 8 bits, de todos os bytes do *record*

:1014000008000224...0c0000001a

- 16 bytes no campo de dados
- record do tipo 0: data record
- endereço do primeiro byte: **0x1D001400**
- checksum: **0x100-trunc8 (10+14+00+00+08+00+02+...+0C+00+00+00)=0x1A**

## Anexo 3

### Mapa de memória do PIC32



## Anexo 4

### Instalação das ferramentas PIC32

- 1) Descarregue do *moodle* de AC2 o *tarball* **pic32-64.tgz** (para sistemas de 64 bits)
- 2) Abra um terminal e execute o comando (note que o SO Linux é *case sensitive*):

```
sudo tar xzvf TARBALL -C /opt
```

onde **TARBALL** é o *path* completo do *tarball*; por exemplo, se descarregou o **pic32-64.tgz** para o diretório **Downloads** deve fazer:

```
sudo tar xzvf ~/Downloads/pic32-64.tgz -C /opt
```

- 3) Abra o ficheiro de texto **.bashrc**. Para isso, primeiro, abra a janela do gestor de ficheiros no diretório **home**. Caso o ficheiro não esteja visível, prima **Ctrl + H** para mostrar os ficheiros ocultos. Alternativamente, pode abrir um terminal e executar um dos seguintes comandos:

```
gedit ~/.bashrc
```

```
nano ~/.bashrc
```

```
vim ~/.bashrc
```

- 4) No final do ficheiro **.bashrc** adicione as seguintes linhas:

```
if [ -d /opt/pic32mx/bin ] ; then
    export PATH=$PATH:/opt/pic32mx/bin
fi
```

### Configuração do computador para comunicar com a placa DETPIC32

- 1) Remova o pacote **brltty** (em Ubuntu):

```
sudo apt remove brltty
```

**Se der erro a desinstalar**, tem que desativar o serviço e para isso deve executar a seguinte sequência de comandos:

```
systemctl stop brltty-udev.service
sudo systemctl mask brltty-udev.service
systemctl stop brltty.service
systemctl disable brltty.service
```

- 2) Adicione o utilizador ao grupo **dialout**; para isso abra um terminal e execute o comando (*case sensitive*):

```
sudo adduser $USER dialout
```

ou, em alternativa:

```
sudo usermod -aG dialout $USER
```

- 3) Faça *reboot* ao sistema.

**Nota:** Se pretender desinstalar as ferramentas pic32 abra um terminal e execute o comando:

```
sudo rm -rf /opt/pic32mx
```