Módulo 1: Programação Assembly (v 2.1)

Este módulo é composto por 21 exercícios e pelo Problema 1. Os exercícios servem para praticar os conhecimentos vistos na teoria e para treinar habilidades de programação e depuração de código. Serão selecionados 3 exercícios para receberem vistos pelo professor. No final do módulo, é proposto um problema que sumariza o conhecimento do módulo como um todo. Tanto os exercícios e a solução do problema devem ser apresentados para o professor. A nota só será validada após o upload do código fonte na plataforma de ensino.

OBJETIVO: Usar a solução de pequenos programas para desenvolver o entendimento e o uso do assembly do MSP430.

Entendendo saltos em assembly: O registro R2, também chamado de Status Register (SR), armazena 4 bits correspondentes à última operação executada. Os 3 mais simples são os seguintes:

- **C** de Carry: Indica se houve carry na soma
- **Z** de Zero: Todos os bits do resultado são iguais a zero, ou seja, o resultado é zero.
- **N** de Negative: O bit mais significativo do resultado é igual a 1

Flags	Saltos se flag = 1		Saltos se flag = 0	
C (Carry)	JC (Jump if Carry)	Salta se houve o carry na última operação	JNC	Salta se não houve carry na operação passada
Z (Zero)	JZ (Jump if Zero)	Salta se o último resultado foi zero	JNZ	Salta se o último resultado foi diferente de zero
N (Negative)	JN (Jump if Negative)	Salta se o último resultado foi negativo	-	-

Veja um exemplo simples: Vamos verificar se um número em R4 é par ou ímpar, se for ímpar, vamos somar 1 para torná-lo par.

```
;-------------------------------------------------------------------------------
verifica:
            and         #0x01, R4           ; Verifica se o LSB de R4 é 1
            jz          par                 ; Se o resultado for zero,
impar:                                      ; o número é par
            inc         R4                  ; Se for ímpar, some 1 p/ torná-lo
par:                                        ; par
;-------------------------------------------------------------------------------
```

Perceba que o salto é colocado estrategicamente logo após a instrução AND entre R4 e a constante 0x01. O resultado dessa operação irá modificar as flags do registro R2 e permitirá que o salto seja referente ao resultado da instrução AND. Em suma, o programa irá para o rótulo “par” se R4 for igual a zero após a operação AND, caso contrário irá continuar a executar a instrução logo abaixo do salto.

Para conveniência, segue a lista simplificada de instruções do MSP430.

Instrução (.B/.W)	Args	Descrição	Bits de Status			
			V	N	Z	C
Formato 1 (2 operandos)						
MOV	src,dst	src → dst	-	-	-	-
ADD	src,dst	src + dst → dst	*	*	*	*
ADDC	src,dst	src + dst + C → dst	*	*	*	*
SUB	src,dst	dst + not(src) + 1 → dst	*	*	*	*
SUBC	src,dst	dst + not(src) + C → dst	*	*	*	*
CMP	src,dst	dst - src	*	*	*	*
DADD	src,dst	src + dst + C → dst (decimally)	*	*	*	*
BIT	src,dst	src .and. dst (bit test)	0	*	*	$\bar{z}$
BIC	src,dst	not(src) .and. dst → dst (bit clear)	-	-	-	-
BIS	src,dst	src .or. dst → dst (bit set)	-	-	-	-
XOR	src,dst	src .xor. dst → dst	*	*	*	$\bar{z}$
AND	src,dst	src .and. dst → dst	0	*	*	$\bar{z}$
Formato 2 (1 operando)						
RRC	dst	C_{n-1} → MSB →.....LSB → C	0	*	*	*
RRA	dst	MSB → MSB →....LSB → C	0	*	*	*
PUSH	dst	SP - 2 → SP, src → SP	-	-	-	-
SWPB	dst	bit 15...bit 8 ↔ bit 7...bit 0	-	-	-	-
CALL	dst	PC+2→ TOS ; #addr →PC	-	-	-	-
RETI	dst		*	*	*	*
SXT	dst	Extend sign bits (B/W/A)	0	*	*	$\bar{z}$
Saltos (Jumps)						
JNE,JNZ	label	Jump if zero is reset	-	-	-	-
JEQ,JZ	label	Jump if zero/equal	-	-	-	-
JNC	label	Jump if carry is reset	-	-	-	-
JC	label	Jump if carry is set	-	-	-	-
JN	label	Jump if negative set	-	-	-	-
JGE	label	Jump if (N xor V) = 0	-	-	-	-
JL	label	Jump if (N xor V) = 1	-	-	-	-
JMP	label	Jump unconditionally	-	-	-	-
Instruções Emuladas						
Instrução	Args	Instrução real	Descrição	V	N	Z	C
ADC	dst	ADDC #0,dst	Add carry to dst	*	*	*	*
SBC	dst	SUBC #0,dst	Subtract carry from dst	*	*	*	*
BR	dst	MOV dst,PC	Branch	-	-	-	-
CLR	dst	MOV #0,dst	Clear dst	-	-	-	-
TST	dst	CMP(.B) #0,dst	Test dst (compare with 0)	0	*	*	1
INC(D)	dst	ADD #[1/2],dst	Increment by 1 (by 2)	*	*	*	*
DEC(D)	dst	SUB #[1/2],dst	Decrement by 1 (by 2)	*	*	*	*
INV	dst	XOR #–1,dst	Invert DST	*	*	*	*
NOP		MOV R3,R3	No operation	-	-	-	-
POP	dst	MOV @SP+,dst	Pop operand from stack	-	-	-	-
RET		MOV @SP+,PC	Return from subroutine	-	-	-	-
RL[A/C]	dst	ADD(C) dst,dst	C ← MSB← ... ← LSB ← [0/C_{n-1}]	*	*	*	*
SET[C/N/Z]		BIS #[1/4/2],SR	Set [Carry/Negative/Zero] bit	-	[1]	[1]	[1]
CLR[C/N/Z]		BIC #[1/4/2],SR	Clear [Carry/Neg/Zero] bit	-	[0]	[0]	[0]

Exercício 1: *Condição simples*
Crie uma soma saturada entre os registros R4 e R5. Se o resultado da soma gerar carry, trave o resultado no máximo representável em 16 bits, ou seja, 0xFFFF. Teste pelo menos dois casos para verificar as duas condições de salto.

Exercício 2: *Laço de execução*
Usando um laço, faça um programa que multiplique R4 por R5. Limite o tamanho do registro para 1 byte. O resultado de uma multiplicação de dois bytes deve caber em 16 bits. Vamos usar o algoritmo de somas sucessivas. Para isso, basta acumular o valor de R4, R5 vezes. Use a instrução DEC ou SUB para ir decrementando R5 de 1 a cada iteração. Use um salto JNZ para manter a iteração enquanto R5 não chega em zero.

Exercício 3: Múltiplas condições
Em assembly, pode parecer que os saltos permitem ramificar o programa em dois setores por vez. Entretanto, note que os saltos não alteram as flags e podem ser posicionados em sequência para gerar ramificações maiores. Realize uma soma entre R4 e R5 e verifique se o resultado é positivo, zero ou negativo. Se for positivo, some 1, se for negativo, subtraia 1 e se for zero, não mude o resultado. Note a necessidade de acrescentar um salto incondicional ao final de cada bloco para evitar passar pelos demais.

Uso de sub-rotinas

Daqui em diante, a solução dos exercícios sempre será uma sub-rotina. Sub-rotinas permitem que o código seja reutilizado e deixa o programa mais fácil de dar manutenção. Uma sub-rotina é chamada usando a instrução CALL #subrot e é delimitada entre o seu rótulo inicial e a instrução final RET. Utilizaremos os registros R12 a R15 para passagem de parâmetros (tanto de entrada como de saída). Os registros de R4 a R11 são de uso genérico e seus valores anteriores devem ser salvos na pilha (usando as instruções PUSH e POP) sempre que esses registros forem usados dentro das sub-rotinas. O programa deve ter a seguinte organização:

```
;-------------------------------------------------------------------------
; Rotina principal (que usa a sub-rotina)

            mov    #vetor, R12  ; Passagem de parâmetros
            mov    #10, R13     ; Ex: Vetor de 10 elementos

            call   #subrot      ; Chama sub-rotina
            jmp    $            ; Trava a execução ao retornar da sub-rotina
;-------------------------------------------------------------------------
; Sub-rotina
subrot:
            push   R4           ; Salva o valor
            push   R5           ; anterior de R4 e R5
            ...                 ; Algoritmo da sub-rotina
            pop    R5           ; Restaura o valor dos
            pop    R4           ; registros previamente salvos
            ret                 ; e retorna
;-------------------------------------------------------------------------
```

As sub-rotinas podem estar em arquivos diferentes da main desde que o símbolo de entrada (nome da sub-rotina) seja definido usando a pseudo-instrução ".def" e referenciado no programa principal usando ".ref".

Exercício 4:
Transforme o exercício 2 (multiplicação de números) numa sub-rotina. Vamos chamar a sub-rotina de **mult8** que tem como parâmetros de entrada:

- R12 → Operando A (8 bits)
- R13 → Operando B (8 bits)

E retorna:

- R12 → Resultado (16 bits)

Teste a sua sub-rotina p/ os casos extremos e para casos intermediários. Exemplo: zero vezes alguma coisa, máximo vezes máximo e X vezes Y.

Declarando vetores na memória

Para este módulo usaremos vetores. Vamos adotar o mesmo padrão usado pela GNU, ou seja, os elementos são organizados sequencialmente a partir de um endereço de origem. Para definir um vetor, precisamos do endereço de início e seu tamanho. Os elementos do vetor podem ser inteiros de 8 bits, ou seja, bytes (nBytes=1), inteiros de 16 bits (nBytes =2) ou inteiros de 32 bits (nBytes =4). A variável nBytes denota o número de bytes de cada elemento unitário.

Forma geral:

Endereço	Dados
Vetor	E_0
Vetor + 1* nBytes	E_1
Vetor + 2* nBytes	E_2

Exemplo de 8 bits

Endereço	Dados
0x2400	4
0x2401	6
0x2402	9

Exemplo de 16 bits:

Endereço	Dados
0x2400	10439
0x2402	40112
0x2404	59622

Uma sequência de letras (string) é formatada ligeiramente diferente. Ela é formada apenas por bytes, seguindo a tabela ASCII e terminada pelo byte de valor 0x00.

Exemplo da string "Hello World" contendo 12 bytes (incluindo o terminador)

H	e	l	l	o		W	o	r	l	d	\0
0x48	0x65	0x6C	0x6C	0x6F	0x20	0x57	0x6F	0x72	0x6C	0x64	0x00

Em assembly, usamos a pseudo-instrução `.data` para indicar o início da RAM. Podemos preencher a RAM de diferentes formas usando os modificadores *byte, word* e *cstring*. Note que letras são aceitas como bytes pois o montador usa a tabela ASCII.

```
;-------------------------------------------------------------------------------
; Dados na memória RAM (a partir do endereço 0x2400)
            .data                           ; Tudo a seguir vai p/ a RAM
v1:         .byte     4, 'A', 0x4F          ; Vetor de bytes (8 bits)
v2:         .word     15000, -1, 0xABCD     ; Vetor de words (16 bits)
str1:       .byte    "SisMic",0             ; Terminação de string manual
str2:       .cstring "2022/2"               ; Terminação de string automatica
;-------------------------------------------------------------------------------
```

Exercício 5:
Escreva a sub-rotina **FIB**, que armazena na memória do MSP a partir da posição 0x2400 os primeiros 20 números da sequência de Fibonacci. Use representação de 16 bits sem sinal.

Exercício 6:
Escreva a sub-rotina **FIB16**, que retorna em R12 o maior número da sequência de Fibonacci a "caber" dentro da representação de 16 bits.

Exercício 7:
Escreva a sub-rotina **FIB32**, que armazena em R13 (MSWord) e R12 (LSWord) o maior número da sequência de Fibonacci a "caber" dentro da representação de 32 bits.

Exercício 8:
Escreva a sub-rotina **sum8** que retorna o somatório de todos os bytes de um vetor. O resultado deve caber num registro de 16 bits. Use a instrução **JNZ** (Jump if Not Zero) para saltar para um rótulo enquanto itera pelos elementos do vetor. A subrotina tem como entrada:

- R12 → Endereço do vetor
- R13 → Número de elementos (bytes) do vetor

E retorna:

- R12 → A soma total

Teste o programa com um vetor de 10 bytes usando valores distribuídos entre 0 e 255. Você pode inicializar a memória com o vetor de teste usando a pseudo-instrução `.byte` na seção `.data` .

```
;-------------------------------------------------------------------------
            .data                           ; Início da RAM
vetor:      .byte    1,2,3,4,5,6,7,8,9,10   ; Vetor de bytes
```

Dica: Você pode visualizar a memória do MSP430 no Code Composer usando o navegador de memória enquanto estiver em depuração. Clique no menu "Window" → "Show View" → "Memory Browser". Use a visualização "8-Bit Hex TI-Style" e navegue para o endereço 0x2400 para ver o seu vetor.

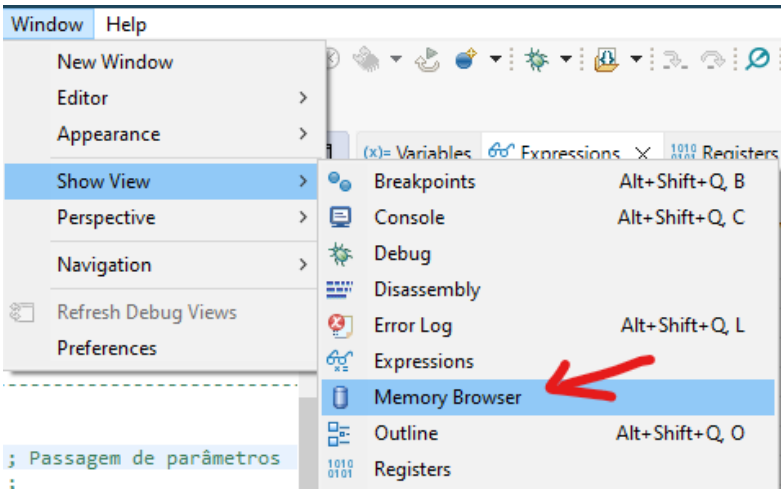


Figura 1: Abrindo o navegador de memória

Exercício 9:
Repita o exercício 1 para entradas de 16 bits. Escreva então a sub-rotina **sum16** que retorna o somatório de todos os números de 16 bits e retorna um valor de 32 bits. Use a instrução **ADDC** para levar em consideração o carry da soma na parte mais significativa. A rotina recebe como entrada:

- R12 → Endereço do vetor
- R13 → Tamanho do vetor

E retorna:

- R12 → Os 16 bits menos significativos (LSWord)
- R13 → Os 16 bits mais significativos da soma (MSWord)

Teste o seu programa com um vetor de inteiros de 16 bits, inicializando a memória com a pseudo-instrução `.word`

Exercício 10:
Escreva a rotina **sub8** que realiza a operação vetorial s = a – b entre vetores de bytes. A sub-rotina tem como entrada:

- R12 → Endereço do vetor de saída (s)
- R13 → Endereço do vetor a (positivo)
- R14 → Endereço do vetor b (negativo)
- R15 → Número de elementos dos vetores

O programa escreve direto na memória e não tem retorno. Para testar esse programa, inicialize o vetor de saída com zeros para facilitar a visualização na memória.

Exercício 11:
Escreva a sub-rotina **m2m4** que calcula a quantidade de múltiplos de 2 e de 4 que existem dentro de um vetor de bytes. Use a instrução **BIT** (Bit Test) em conjunto com **JC** ou **JNC** (Jump if [Not] Carry) para verificar se os bits de peso 1, 2 e 4 estão setados nos bytes analisados. A sub-rotina recebe como entrada:

- R12 → Endereço de início de um vetor de bytes
- R13 → O tamanho do vetor

e retorna:

- R12 → Quantidade de múltiplos de 2
- R13 → Quantidade de múltiplos de 4

Exercício 12:
Escreva a sub-rotina rot16 que rotaciona um valor de 16 bits seguindo as opções a seguir:

- Quantidade de bits rotacionados
- Direção: 0 p/ esquerda ou 1 p/ direita
- Tipo da rotação:
  - 0 p/ rotação lógica inserindo 0's,
  - 1 p/ rotação lógica inserindo 1's,
  - 2 p/ rotação aritmética
  - 4 p/ rotação circular

As opções são compactadas numa entrada onde cada nibble (4 bits) corresponde a uma opção, da seguinte forma:

Bits 15 a 12	Bits 11 a 8	Bits 7 a 4	Bits 3 a 0
Direção	Tipo da rotação	-	N rotações

Exemplos: Se o valor da opção for 0x1408 então se trata de uma rotação de 8 bits circular para a direita, se o valor for 0x000E se trata de uma rotação lógica de 14 bits para a esquerda inserindo 0's.

As entradas da subrotina são:
- R12 → Valor para rotacionar
- R13 → Opções: Direção/Tipo da rotação

E retorna
- R12 → O resultado da rotação

Use a instrução **AND** para filtrar e selecionar bits e as instruções **BIT** e **J(N)C** para controlar o fluxo do programa.

Exercício 13:
Escreva a sub-rotina **`menor`** que tem como entradas:
- R12 → Endereço de início de um vetor de bytes sem sinal
- R13 → Tamanho do vetor

e retorna:
- R12 → Menor elemento do vetor e
- R13 → Qual sua frequência (quantas vezes apareceu)

Teste o programa com um vetor de 10 bytes usando valores distribuídos entre 0 e 255.

Exercício 14:
Escreva a sub-rotina **maior16** que recebe
- R12 → Endereço de início de um vetor de palavras de 16 bits (words ou W16) sem sinal
- R13 → Tamanho do vetor

e retorna:
- R12 → maior elemento do vetor e
- R13 → qual sua frequência (quantas vezes apareceu)

Para testar, use o mesmo vetor do Exercício 4, mas agora seu programa irá interpretar cada elemento como sendo composto por 2 bytes. Assim, o tamanho do vetor deve cair para a metade, ou seja, para 5 elementos. Usando o navegador de memória na visualização "16-Bit Hex - TI Style", note a inversão dos bytes. Isso acontece pois o MSP430 usa organização *little endian*. O menor endereço corresponde ao byte menos significativo.

Exercício 15:
Escreva sub-rotina **EXTREMOS** que recebe em R5 o endereço de início de um vetor com palavras de 16 bits (W16) com sinal e retorna:
- **R6** → menor elemento, **R7** → maior elemento

Para este exercício, vamos formar o vetor usando o número de matrícula e o ano de nascimento de cada membro da equipe. Veja o exemplo para uma equipe com 2 alunos:
Aluno 1: matrícula = 12/1234567 e nasceu em 1990 → 121, 234, 567, -1990.
Aluno 2: matrícula = 11/786745 e nasceu em 1980 (está velho) → 117, 867, 45, -1980

```
            .data
; Declarar vetor com 8 elementos [121, 234, 567, -1990, 117, 867, 45, -1980]
vetor:      .word       8, 121, 234, 567, -1990, 117, 867, 45, -1980
```

Exercício 16:
Escreva a sub-rotina **SUM16** que armazena a soma (elemento a elemento) de dois vetores de 16 bits de mesmo tamanho.

R5 = 0x2400 → endereço do vetor 1;
R6 = 0x2410 → endereço do vetor 2;
R7 = 0x2420 → endereço do vetor soma.

```
            .data
; Declarar os vetores  com 7 elementos
Vetor1:     .word       7, 65000, 50054, 26472, 53000, 60606,   814, 41121
Vetor2:     .word       7,   226,  3400, 26472,   470,  1020, 44444, 12345
```

Exercício 17:
Escreva a sub-rotina **W16_ASC** que recebe em R6 um número (sem sinal) de 16 bits e escreve a partir do endereço 0x2400 o código ASCII correspondente ao valor hexadecimal de cada nibble (4 bits). A sugestão é para criar a sub-rotina **NIB_ASC**, que converte um nibble em ASCII e depois usar essa sub-rotina 4 vezes. Use R5 como ponteiro para escrita na memória. Veja o exemplo abaixo:

Recebe: R6 = 35243 (0x89AB em hexadecimal)
Retorna em 0x2400: 0x38, 0x39, 0x41, 0x42

Inicialize R6 com os **5 primeiros dígitos de seu número de matrícula**. Uma forma elegante de se declarar uma constante num programa é usando a diretiva ".set", como mostrado abaixo.

```
MATR        .set   35243
…
            mov    #MATR,R6
```

Exercício 18:
Escreva sub-rotina **ASC_W16** que faz a operação inversa do Exercício 5. Recebe R5 apontando para um endereço (0x2400) com quatro códigos ASCII e monta em R6 a palavra de 16 bits correspondente. Inicializar a memória com seus dados do programa anterior.

Note que é necessário testar se os códigos são válidos de acordo com a Tabela ASCII (de 0x30 → 0x39 e de 0x41 → 0x46). Caso tenha sucesso, deve retornar o Carry em 1. Em caso de erro, retornar Carry em zero.

Caso de sucesso.
Recebe em 0x2400: 0x38, 0x39, 0x41, 0x42
Retorna: R6 = 0x89AB e Carry = 1.

Caso de falha.
Recebe em 0x2400: 0x38, 0x3B, 0x41, 0x42
Retorna: R6 = "don't care" e Carry = 0.

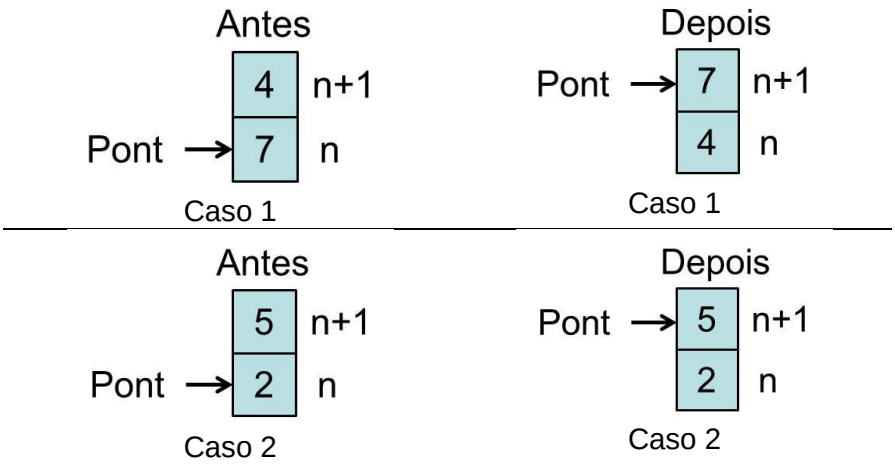
```
;-------------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------------
;
      mov   #MEMO,R5
      call  #ASC_W16    ;chamar sub-rotina
OK    jc    OK          ;travar execução com sucesso
NOK   jnc   NOK         ; travar execução com falha
;
ASC_W16:
      ...
      ret

;-------------------------------------------------------------------------------
; Segmento de dados inicializados (0x2400)
;-------------------------------------------------------------------------------
            .data
; Declarar 4 caracteres ASCII (0x38, 0x39, 0x41, 0x42)
MEMO:       .byte '8','9','A','B'
```

SUGESTÕES:

- Esboçar um fluxograma para o problema.
- Escreva os programas de forma fracionada. Faça uso de sub-rotinas. Coloque as sub-rotinas logo depois do programa principal.
- Documente as sub-rotinas, é provável que você as use em experimentos futuros.

Exercício 19:
Escrever sub-rotina que ordena, de forma crescente, um vetor. Um método muito conhecido para ordenar os elementos de um vetor é o método da BOLHA. Este método pede uma sub-rotina auxiliar que vamos chamar de ORD.
A sub-rotina ORD ordena de forma crescente duas posições de memória: a apontada pelo ponteiro e a seguinte. Veja os exemplos abaixo. No Caso 1, as duas posições foram trocadas. No Caso 2, a ordem já estava correta e nada foi feito. Note que em ambos os casos, o ponteiro terminou apontando para a segunda posição.



Vamos agora considerar um vetor com ***n*** elementos, como mostrado abaixo. Note que o vetor está na vertical. Se varrermos todo este vetor com a sub-rotina ORD, no topo do vetor deverá estar o maior elemento (Maior 1). Se varrermos novamente o vetor, exceto a última posição, selecionaremos o segundo maior elemento (MAIOR 2). Repetimos esse procedimento ***n-1*** vezes e o vetor será ordenado.

n	y	Maior 1	Maior 1	Maior 1	...	Maior 1
n-1	x	y	Maior 2	Maior 2	...	Maior 2
...	...	...	...	Maior 3	...	Maior 3
3	c	c	c	...	...	...
2	b	b	b	b	...	Menor 2
1	a	a	a	a	...	Menor 1
Seq.	**Original**	**Varrida 1**	**Varrida 2**	**Varrida 3**	**...**	**Varrida n-1**

Exemplo: vetor [4, 7, 3, 5, 1] com 5 elementos, na horizontal.

	Tamanho	Elementos do vetor				
Original	5	4	7	3	5	1
Varrida 1 (4 comparações)	5	4	3	5	1	**7**
Varrida 2 (3 comparações)	5	3	4	1	**5**	7
Varrida 3 (2 comparações)	5	3	1	**4**	5	7
Varrida 4 (1 comparação)	5	1	**3**	4	5	7

Escreva sub-rotina **ORDENA** que recebe em R5 o endereço de início de um vetor de bytes (sem sinal) e o ordena. Organize os registradores da forma abaixo:

Para este programa, declare um vetor de bytes formado pela **concatenação do código ASCII dos nomes completos de cada membro da equipe**. Note que o montador já converte as letras para o código ASCII correspondente, como mostrado abaixo. Use letras maiúsculas, omita os

espaços e não use acentos. Preste atenção ao tipo das aspas. Note que usamos a mesma formatação de vetor do Exercício 1.

Atenção: Não use a instrução swpb (swap bytes) pois ela opera em palavras de 16-bits e está sempre alinhada em endereços pares, ou seja, não vai funcionar em endereços ímpares.

O programa deve ter a seguinte organização:

```
;-------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------
            mov   #vetor,R5   ;incializar R5 com o endereço do vetor
            call  #ORDENA     ;chamar sub-rotina
            jmp   $           ;travar execução
            nop               ;exigido pelo montador
;
ORDENA:     ...
            ...
            Ret
            .data
; Declarar vetor com a concatenação dos nomes completos da equipe
vetor:      .byte 11, "JOAQUIMJOSE"
```

Exercício 20:
Neste exercício vamos operar com algarismos romanos. Para relembrar, indicamos o link abaixo:
https://www.somatematica.com.br/fundam/romanos.php

Escreva a sub-rotina ROM_ARAB, que recebe R6 apontando para um número representado com algarismos romanos (será uma a sequência de letras) e retorna em R5 o número correspondente. Note que o número representado com algarismos romanos é, na verdade, um vetor composto por bytes. Neste exercício, ao invés de usarmos a primeira posição para indicar o tamanho do vetor, vamos apenas marcar seu final com o byte zero (0x00).

Siga a estruturação indicada abaixo.

```
;-------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------
;
      mov   #NUM_ROM,R6 ;R6 aponta para o início do número
      call  #ROM_ARAB   ;chamar sub-rotina
      jmp   $           ;travar execução
;
ROM_ARAB:
      ...
      ret

;-------------------------------------------------------------------------
; Segmento de dados inicializados (0x2400)
;-------------------------------------------------------------------------
            .data
; Especificar o número romano, terminando com ZERO.
NUM_ROM:    .byte "MMXIX",0         ;2019
```

Exercício 21:
Apresente sub-rotina **ALG_ROM** que recebe em R5 um número entre 1 e 3999 e o escreve com algarismos romanos a partir da posição de memória apontada por R6. O fim desse número deve ser indicado como o byte igual a zero (0x00). O programa deve ter a seguinte organização:

```
;-------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------
NUM         .equ  2019        ;Indicar número a ser convertido
;
            mov   #NUM,R5     ;R5 = número a ser convertido
            mov   #RESP,R6    ;R6 = ponteiro para escrever a resposta
            call  #ALG_ROM    ;chamar sub-rotina
            jmp   $           ;travar execução
            nop               ;exigido pelo montador
;
ALG_ROM:    ...
            ret

            .data
; Local para armazenar a resposta (RESP = 0x2400)
RESP:       .byte       "RRRRRRRRRRRRRRRRR",0
```