Módulo 1: Programação Assembly

Os exercícios deste módulo servem 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 como 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.

Declarando variáveis e vetores na memória

Variáveis e vetores são estruturas de dados que são colocadas na memória RAM a partir do endereço 0x2400. 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, inteiros de 16 bits (1 word = 2 bytes) ou inteiros de 32 bits (4 bytes). Abaixo está representado dois exemplos de vetores mostrando que o endereço dos elementos do vetor avança de maneira dependente do tamanho do elemento que o vetor guarda. A variável nBytes abaixo 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	14
0x2401	6
0x2402	250

Exemplo de 16 bits

Endereço	Dados
0x2400	10000
0x2402	20000
0x2404	60000

Uma sequência de letras (string) é formatada ligeiramente diferente na memória. Ela possui apenas bytes que representam letras seguindo a tabela ASCII. A tabela ASCII pode ser facilmente obtida na internet. Uma string sempre é terminada pelo byte de valor 0x00. Veja um 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
variavel:   .word    32768,
vetor1:     .byte    4, 'A', 0x4F           ; Vetor de bytes (8 bits)
vetor2:     .word    15000, -1, 0xABCD      ; Vetor de words (16 bits)
string1:    .byte   "SisMic",0              ; Terminação de string manual
string2:    .cstring "2022/2"               ; Terminação de string automatica
;-------------------------------------------------------------------------------
```

Uma instrução no assembly do MSP430 possui até dois argumentos. Os argumentos são as variáveis que queremos operar. No exemplo abaixo, o registro R4 é inicializado com o valor 512 e em seguida adicionamos 64 de forma acumulativa no mesmo registro. O operando da direita é o destino. *A lista de todas as instruções está na tabela 1.*

```
MOV.W   #512, R4  (use .B para escrever bytes e
ADD.W   #64,  R4   .W para escrever words)
```

Podemos acessar dados que estejam nos registradores do processador (R4 a R15) ou regiões da memória. O MSP430 oferece 7 modos de endereçamento. Note que nem todos podem ser usados no operando da direita, ou seja, no destino.

Modo de endereçamento	Sintaxe	src	dst
1) Registro	`Rn`	Sim	Sim
2) Simbólico	`LB`	Sim	Sim
3) Absoluto	`&LB`	Sim	Sim
4) Indexado	`i(Rn)`	Sim	Sim
5) Indireto	`@Rn`	Sim	**Não**
6) Indireto com autoincremento	`@Rn+`	Sim	**Não**
7) Imediato	`#CTE`	Sim	**Não**

Os modos indexado, indireto e indireto com autoincremento usam um registro como ponteiro para uma região da memória. Eles recuperam uma informação que estiver no endereço apontado pelo registro Rn. No caso do modo indexado, o endereço final é o valor do registrador somado do índice. O registro não é modificado. Exemplos: 0(Rn) é a posição apontada por Rn, 1(Rn) é a próxima. Valores negativos também podem ser usados, como -1(Rn). No modo indireto com autoincremento o valor do registro é modificado após a instrução ser executada. O incremento vale 1 para instruções de bytes (.B) e vale 2 para instruções de words (.W).

Exercício 1 (guiado): Complete os espaços em branco

a) Usando o modo imediato (#cte), inicialize o registrador R4 para a constante 0x2400.

```
MOV #0x2400, R4
```

Dica: Visualize a memória do MSP430 no Code Composer clicando no menu "View" → "Memory". Essa janela só fica disponível no modo de depuração. Use a visualização "8-Bit Hex TI-Style" e navegue para o endereço 0x2400 para ver o seu vetor.

a) Use agora o modo de endereçamento indexado "`i(Rn)`" para gravar o byte 0xFE na posição apontada por R4. Use o modificador "**`.B`**" para acessar bytes.

```
MOV.B #0xFE,  0(R4)
```

b) Tente usar o modo indireto (@Rn) para fazer a mesma coisa. Qual é o erro que aparece?

```
MOV.B #0xFE, @R4
```

Perceba que o modo indexado não pode ser usado como destino

c) Usando o modo indexado, grave o byte 0xCA na próxima posição da memória (R4+1).

```
MOV.B #___, __(__)
```

d) Grave as words 0x1234 e 0x5678 nos endereços seguintes usando o modificador "**`.W`**". Perceba que os endereços de words andam de 2 em 2.

```
MOV.W #0x1234,  2(R4)
MOV.W #______, __(R4)
```

e) (opcional) O que acontece se você gravar uma word num endereço ímpar?

```
MOV.W #0xABCD, 5(R4)
```

Exercício 2 (guiado): Crie um novo projeto para este exercício. Inicialize a memória usando a pseudo-instrução "**`.data`**" com os **.bytes**: 0x12, 0x34, 0x56, 0x78, 0x9A, 0xBC, 0xDE (veja um exemplo na primeira página) a partir de um rótulo (label) **vetor**.

a) Inicialize o registro R4 com o endereço do vetor.

```
MOV.W   #vetor, R4
```

*Lembre-se que todo endereço tem 16 bits (**word**). Note como é mais fácil trabalhar com a referência nominal #vetor ao invés do número #0x2400.*

b) Use o modo de endereçamento indireto com autoincremento (`@Rn+`) para ler os dois primeiros **bytes** da memória e colocar nos registros R5 e R6.

```
MOV.B @R4+, R5
MOV.B @R4+, __
```

Veja como esse modo é prático para ler regiões sequenciais da memória.

c) Usando agora o modificador .W, escreva as duas próximas **words de 16-bits** do vetor nos registros R7 e R8.

```
MOV.W @R4+, R7
```

Note que o operador "+" é inteligente e depende do modificador .B ou .W. Note também que os bytes ficaram trocados. Qual é a organização de memória do MSP430? Dica: você pode ver o mesmo efeito trocando a visualização da memória de bytes para words.

d) Crie um rótulo "var8" na RAM (seção `.data`) e inicialize com o `.byte` 0x00. No programa, escreva o byte 0x33 usando o modo de endereçamento simbólico (`label`) ou absoluto (`&label`).

```
MOV__  #0x33, ___
```

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:
            bit         #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
;-------------------------------------------------------------------------------
```

A instrução BIT (Bit Test) é semelhante à instrução AND porém descarta o resultado. No exemplo, ela realiza a operação “E” lógica bit a bit entre uma máscara e o registro R4. A máscara com o valor 0x01 possui apenas o bit menos significativo (LSB) setado (em 1). Dessa forma, a operação lógica “E” resultará em 0x01 apenas se R4 for ímpar e 0x00 se for par. O salto JZ é colocado estrategicamente logo após a instrução BIT. O resultado da instrução anterior irá modificar as flags do registro R2 e permitirá que o salto seja referente ao resultado da instrução BIT. Em suma, o programa irá para o rótulo “par” se o LSB de R4 for zero, 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	Return from Interruption	*	*	*	*
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	-	-	-	-
JLO, JNC	label	Jump if carry is reset	-	-	-	-
JHS, 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]

Tabela 1 – Instruções do MSP430

Exercício 3: *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 4: *Laço de execução*
Usando um laço, faça um programa que multiplique R4 por R5. Limite o tamanho das entradas 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 5: 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 que usa R4 e R5

            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 6:
Transforme o exercício 4 (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.

Exercício 7:
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 8:
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 9:
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 10:
Escreva a sub-rotina **reduceSum8** 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 entre 0 e 255. Exemplo:

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

Exercício 11:
Repita o exercício 10 para entradas de 16 bits. Escreva então a sub-rotina **reduceSum16** 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 12:
Escreva a rotina **mapSub8** 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 13:
Escreva a sub-rotina **mapSum16** 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
vetor1:     .word    7, 65000, 50054, 26472, 53000, 60606,   814, 41121
vetor1:     .word    7,   226,  3400, 26472,   470,  1020, 44444, 12345
```

Exercício 14:
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 15:
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. *Dica: Crie subrotinas pequenas para cada tipo de rotação, depois junte tudo na rot16.*

Exercício 16:
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 17:
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 anterior, 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 18:
Escreva sub-rotina **extremos** que recebe

- R12 → Endereço de início de um vetor com palavras de 16 bits (W16) com sinal,
- R13 → Tamanho do vetor

e retorna:

- R12 → menor elemento,
- R13 → maior elemento

Teste seu algoritmo com vetores de words com valores positivos e negativos

Exercício 19:
Escreva a sub-rotina **W16_ASC** que recebe em

- R12 → um número sem sinal de 16 bits e
- R13 → o endereço do vetor de saída

e escreve a partir do endereço de R13 o código ASCII correspondente ao valor hexadecimal de cada nibble (4 bits). É sugerido 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: R12 = 0x89AB
Retorna em @R13: '8', '9', 'A', 'B' (ou seja, os bytes: 0x38, 0x39, 0x41, 0x42)

Exercício 20:
Escreva a sub-rotina **ASC_W16** que faz a operação inversa do exercício anterior. Recebe em

- R12 → endereço de vetor com quatro códigos ASCII

e retorna em

- R12 → a palavra de 16 bits correspondente.

Note que é necessário testar se os códigos são válidos de acordo com a Tabela ASCII (Números de 0x30 a 0x39 e letras de 0x41 a 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: R12 = 0x89AB e Carry = 1.

Caso de falha.
Recebe em 0x2400: 0x38, 0x3B, 0x41, 0x42
Retorna: R12 = "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 21:
Escreva uma subrotina 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 realiza N-1 varridas num vetor de N elementos. Cada varrida compara pares de elementos vizinhos e caso encontre um elemento menor à frente, troca os dois de posição. Repetimos esse procedimento até que o vetor esteja totalmente ordenado. Perceba que cada varrida coloca o maior elemento no final do vetor. Dessa forma, cada varrida só precisa ordenar um vetor menor, descartando o maior elemento da varrida anterior. Veja o exemplo a seguir do vetor [4, 7, 3, 5, 1] com 5 elementos.

A primeira varrida inicia comparando os elementos 4 e 7. Como 7 é maior, ele continua na direita.

4	7	3	5	1

Em seguida, comparamos 7 e 3. Como o 7 é maior, ele deve ir para a direita. Trocamos os dois números então:

4	7	3	5	1

4	3	7	5	1

Depois, comparamos o 7 com 5 e novamente percebemos que o 7 é maior e os dois devem ser trocados.

4	3	7	5	1

4	3	5	7	1

O mesmo procedimento se repete na última comparação e terminamos a nossa primeira varrida com o maior elemento na direita.

4	3	5	7	1

4	3	5	1	7

Agora vamos repetir esse procedimento até que o vetor esteja totalmente ordenado. Perceba que cada varrida trabalha com um vetor menor que a varrida anterior, já que o maior elemento da varrida já é colocado na direita.

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

Escreva sub-rotina **ORDENA** que recebe em R12 o endereço de início de um vetor de bytes (sem sinal) e em R13 o seu tamanho. O programa deve aplicar o algoritmo da bolha para ordenar o vetor. Teste seu programa com números que variam de 0 a 255.

Exercício 22:
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 em R12 um endereço apontando para uma string que representa um número em algarismos romanos (será uma a sequência de letras) e retorna também em R12 o número correspondente.

Exemplo: “MCDLXXIX” -> 1000 + (500-100) + (50+10+10) + (10-1) = 1479

Sugestão de solução: Transforme a string (vetor de bytes) num vetor de números (de 16-bits) com os pesos de cada número. Em seguida, para cada elemento do vetor, verifique se o próximo é maior. Se o próximo for maior, você deve subtrair o seu peso na soma final, se for menor ou igual, você deve somá-lo.

Exercício 23:
Apresente sub-rotina **ARAB_ROM** que recebe em R12 um número entre 1 e 3999 e o escreve numa string com algarismos romanos a partir da posição de memória apontada por R13. O fim da string deve ser indicado com o byte igual a zero (0x00).

Exercício 24:
Apresente a sub-rotina **MAT_TRANSP**, que recebe em R12 um ponteiro para uma matriz de words, em R13 o número de linhas da matriz e em R14 o número de colunas. A rotina deve escrever a transposta da matriz a partir da primeira posição de memória após o fim da matriz original. As matrizes devem estar escritas na memória linha por linha.

Por exemplo, considere a matriz de entrada

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Neste caso, a memória RAM deve ser inicializada como indicado na Tabela 24-1. **Após a execução da sub-rotina**, os endereços de memória mostrados na Tabela 1 devem continuar armazenando os mesmos valores e os endereços seguintes devem armazenar a matriz M^T, como indicado na Tabela 2. Note que as matrizes são salvas na memória da mesma forma que vetores, apenas interpretamos que essas sequências de dados representam matrizes.

Tabela 24-1 - Memória do MSP430 contendo a matriz de entrada

Endereço	0x2400	0x2402	0x2404	0x2406	0x2408	0x240A
Valor	1	2	3	4	5	6

Tabela 14-2 Memória do MSP430 contendo a matriz transposta

Endereço	0x240C	0x240E	0x2410	0x2412	0x2414	0x2416
Valor	1	4	2	5	3	6

O programa deve ter a seguinte estrutura:

```
    .cdecls "msp430.h"
    .global main
    .text
main:
    mov.w   #(WDTPW|WDTHOLD), &WDTCTL
    mov     #matriz, R12 ; Ponteiro para a matriz de entrada
    mov     #2, R13      ; Número de linhas da matriz
    mov     #3, R14      ; Número de colunas da matriz
    call    #MAT_TRANSP  ; Chamar sub-rotina
    jmp     $            ; Loop infinito
    nop
MAT_TRANSP:
    ; Seu código aqui

; Especificar a matriz de entrada na seção de dados
    .data
matriz: .word 1, 2, 3, 4, 5, 6
```

Note que a subrotina deve funcionar para **matrizes de qualquer tamanho** e que o programa tem que chamar a sub-rotina MAT_TRANSP, retornar para o bloco principal e então ficar preso no loop infinito.

Exercício 25:

Apresente a sub-rotina SUM_SUB, que recebe em R12 um ponteiro para uma matriz de words, em R13 o número de linhas da matriz e em R14 o número de colunas. A rotina deve salvar:

- a soma dos elementos da matriz exceto aqueles que estão na primeira linha e/ou na primeira coluna em R12;
- a soma dos elementos da matriz exceto aqueles que estão na primeira linha e/ou na última coluna em R13;
- a soma dos elementos da matriz exceto aqueles que estão na última linha e/ou na primeira coluna em R14;
- a soma dos elementos da matriz exceto aqueles que estão na última linha e/ou na última coluna em R15.

Por exemplo, considere a matriz de entrada

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Neste caso, teremos:

- R12 = 5 + 6 + 8 + 9 = 28.
- R13 = 4 + 5 + 7 + 8 = 24.
- R14 = 2 + 3 + 5 + 6 = 16.
- R15 = 1 + 2 + 4 + 5 = 12.

O programa deve ter a seguinte estrutura:

```
    .cdecls "msp430.h"
    .global main
    .text
main:
    mov.w   #(WDTPW|WDTHOLD), &WDTCTL
    mov     #matriz, R12 ; Ponteiro para a matriz de entrada
    mov     #3, R13      ; Número de linhas da matriz
    mov     #3, R14      ; Número de colunas da matriz
    call    #SUM_SUB     ; Chamar sub-rotina
    jmp     $            ; Loop infinito
    nop
SUM_SUB:
    ; Seu código aqui

; Especificar a matriz de entrada na seção de dados
    .data
matriz: .word 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Note que a sub-rotina deve funcionar para **matrizes de qualquer tamanho** (com pelo menos duas linhas e duas colunas) e que o programa tem que chamar a sub-rotina SUB_SUM, retornar para o bloco principal e então ficar preso no loop infinito.