2 Programação *Assembly*

Versão 2.0

Na programação *assembly*, usamos palavras especiais (mnemônicos) para representar as instruções do processador. Isto é necessário porque os processadores executam códigos binários. Cada código corresponde a uma operação. Citamos abaixo alguns códigos de 16 *bits* (em hexadecimal) e as operações que realizam.

4036 0014 → coloca no número 20 decimal em R6 (R6 = 20);
8326 → decrementa (subtrai 1) do conteúdo de R6;
1005 → "roda" para a direita o conteúdo de R5.

Esses códigos, cada um representado uma instrução, são denominados de Códigos de Operação (*Operational Code* em inglês) e abreviados por opcode. São também, muitas vezes, apelidados de Códigos de Máquina.

Os opcodes são extremamente inconvenientes para o ser humano. Imagine um programador tendo que consultar uma gigantesca tabela para cada operação que ele precise realizar. Tudo se complica quando se tem um programa completo com centenas desses códigos. A listagem abaixo é de um programa que calcula os primeiros 20 elementos da sequência de Fibonacci e os armazena a partir do endereço 0x2400. O programa está na memória a partir do endereço 0x4400, que é o início da memória Flash, onde são armazenados os programas a serem executados. Podemos dizer que é insano tentar entender programas analisando apenas os opcodes.

Listagem de um programa mostrando apenas os opcodes

```
004400: 4031 4400
004404: 40B2 5A80 015C
00440a: 4036 0014
00440e: 12B0 4414
004412: 3FFF
004414: 8326
004416: 4035 2400
00441a: 4385 0000
00441e: 5325
004420: 4395 0000
004424: 5325
004426: 4595 FFFE 0000
00442c: 5595 FFFC 0000
004432: 8316
004434: 23F7
004436: 4130
```

Apresentamos abaixo o mesmo programa (mesmos opcodes), mas agora com uma “dica” sobre as operações que estão sendo realizadas. Essa “dica” sobre cada operação recebe o nome de instrução.

Listagem do programa acima, mas agora mostrando os opcodes e as instruções

```
RESET       004400: 4031 4400       MOV    #0x4400,SP
            004404: 40B2 5A80 015C  MOV    #0x5A80,&Watchdog_Timer_WDTCTL
            00440a: 4036 0014       MOV    #0x0014,R6
            00440e: 12B0 4414       CALL   #FIB
            004412: 3FFF            JMP    (0x4412)
FIB         004414: 8326            DECD   R6
            004416: 4035 2400       MOV    #0x2400,R5
            00441a: 4385 0000       CLR    0x0000(R5)
            00441e: 5325            INCD   R5
            004420: 4395 0000       MOV    #1,0x0000(R5)
LOOP        004424: 5325            INCD   R5
            004426: 4595 FFFE 0000  MOV    0xFFFE(R5),0x0000(R5)
            00442c: 5595 FFFC 0000  ADD    0xFFFC(R5),0x0000(R5)
            004432: 8316            DEC    R6
            004434: 23F7            JNE    (LOOP)
            004436: 4130            RET
```

2.0. Quero Programar em Assembly com o MSP430 e Não Pretendo Ler Todo Este Capítulo

Sob este título, é apresentado o mínimo para ser programar em Assembly com o MPS430. Para evitar confusão como o restante do capítulo, as figuras aqui apresentadas não são numeradas e, algumas vezes, são uma mera repetição.

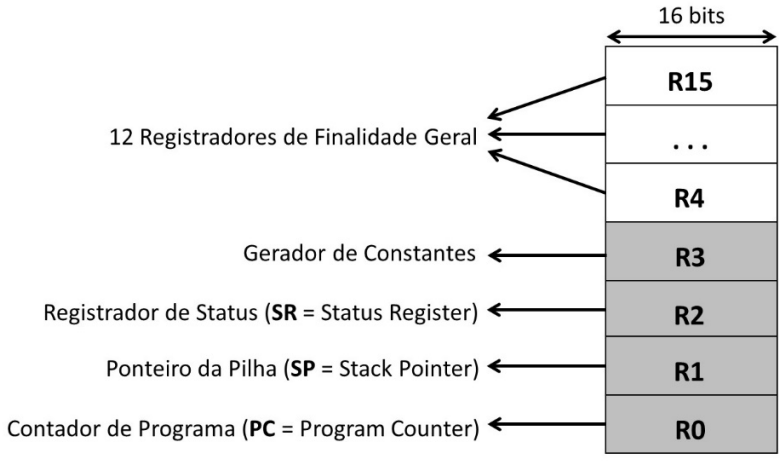
Toda instrução do MSP430 tem o formato `instr src,dst`, onde `src` representa a origem (*source*) do dado a ser operado e `dst` representa o destino (*destination*), o local onde será armazenado o resultado da operação. A seguir está uma tabela com os modos de endereçamento, ou seja, as formas de se indicar `src` e `dst`. Veja que há restrições para o campo `dst`. Nem todas as instruções fazem uso dos campos `src` ou `dst`.

Resumo dos modos de endereçamento do MSP430

Nr	Nome do Modo	Modo	Operando	src	dst
1	Registrador	`Rn`	Rn	Sim	Sim
2	Indexado	`X(Rn)`	(X+Rn)	Sim	Sim
3	Simbólico	`X`	(X+PC)	Sim	Sim
4	Absoluto	`&ADR`	(ADR)	Sim	Sim
5	Indireto via Reg.	`@Rn`	(Rn)	Sim	**Não**
6	Indireto via Reg. com autoincr.	`@Rn+`	(Rn) e Rn++	Sim	**Não**
7	Imediato	`#N`	N	Sim	**Não**

Na coluna operando, o parêntesis indica acesso à memória.

O conjunto de instruções do MSP430 está apresentado na Tabela 2.2, logo mais adiante. As instruções fazem referências à memória e aos registradores. A figura abaixo apresenta o conjunto de registradores do MSP430.



Registradores do MSP430.

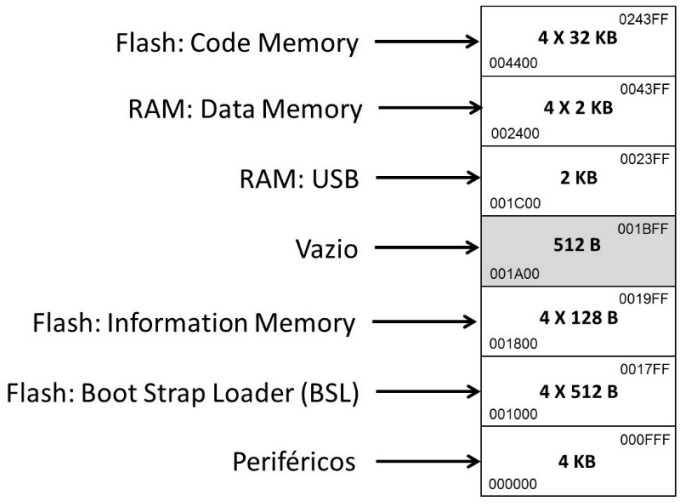
O Registrador de *Status* (R2), apresentado abaixo, merece destaque, pois ele contém as *flags* importantes para a programação assembly.

- C = Carry
- Z = Zero
- N = Negativo e
- V = *Overflow* (para operações em complemento a 2).

15 ... 9	8	7	6	5	4	3	2	1	0
Reservados	V	SCG 1	SCG 0	OSC OFF	CPU OFF	GIE	N	Z	C

Detalhamento do Registrador de Status (R2 = SR).

O mapa de memória está apresentado na figura a seguir. Note que este mapa é específico para a versão F559. A memória de programa (*flash*) inicia em 0x4400. No modelo de programação em 16 *bits* de endereço, que é o nosso caso, seu limite está em 0xFFFF. Já no modo de endereçamento de 20 *bits*, ela pode chegar a 0x243FF. A memória de dados ocupa a faixa de 0x2400 até 0x43FF. Usualmente este espaço é destinado aos dados do usuário é à pilha. Os dados ocupam a memória de forma crescente a partir de 0x2400 e a pilha cresce no sentido inverso, a partir de 0x43FF.



Mapa de memória do MSP430 F5529.

O MSP430 é uma CPU *Little-endian*, o que significa que os dados com tamanho superior a 8 *bits* são arrumados na memória a partir do *byte* menos significativo.

Terminamos assim este tópico. Os exercícios resolvidos apresentados no final deste capítulo irão ajudar bastante o desenvolvimento das habilidades de programação assembly com o MSP430. Entretanto, o leitor é fortemente recomendado a estudar este capítulo na sua íntegra.

2.1. Programação em Assembly

Para tornar possível o trabalho do programador, as operações que um processador pode realizar são organizadas em instruções (instruções *assembly*). Cada instrução tem um campo denominado de mnemônico que sugere o que a instrução faz e um segundo campo denominado de operandos, que indica os operandos envolvidos na execução da instrução. A tabela abaixo apresenta o mnemônico e os operandos de algumas instruções. Note que algumas instruções não usam o campo de operando, como é o caso da instrução `RET` que faz o retorno de uma sub-rotina.

Tabela 2.1. Exemplos de algumas instruções do MSP430

Instrução	Mnemônico	Operandos	Resultado
`MOV #20,R6`	`MOV`	`#20,R6`	`R6 = 20`
`DECD R6`	`DECD`	`R6`	`R6 = R6 - 2`
`INCD R5`	`INCD`	`R5`	`R5 = R5 + 2`
`RET`	`RET`	-	`Retorno de sub-rotina`

Um programa em *assembly* é apenas a reunião de instruções *assembly* na ordem conveniente para realizar as operações desejadas pelo programador. Cada instrução *assembly* (mnemônico mais operando) corresponde a um *opcode*. Para facilitar sua vida, o programador faz uso de um programa, denominado Montador (*Assembler* em inglês), que faz a tradução das instruções usadas pelo programador para os códigos de operação. A tabela mais adiante apresenta o programa fonte que calcula os primeiros 20 elementos da sequência de Fibonacci e o arquivo executável (*opcodes*) entregue pelo montador. Note que a instrução (`MOV.W #0x5A80,&Watchdog_Timer_WDTCTL`) foi abreviada para caber nesta tabela. Assim, o programador escreve seu programa em *assembly* e o montador faz o trabalho braçal para transformar cada instrução no *opcode* correspondente.

É interessante comentar que há um abuso quando se diz “Linguagem *Assembly*”. O termo linguagem, em computação, pressupõe a existência de átomos da linguagem, um

conjunto de regras de sintaxe e uma semântica específica para cada declaração da linguagem. Por exemplo, a “Linguagem C” possui todos esses requisitos. O mesmo não acontece quando se programa em *assembly*. A “Linguagem *Assembly*” é apenas uma arrumação interessante de mnemônicos e operandos para evitar que o usuário use diretamente os *opcodes*.

Tabela 2.2. Exemplo de programa fonte e sua tradução realizada pelo montador

Programa Fonte			Montador	Opcodes
RESET:	MOV	#0x4400,SP		004400: 4031 4400
	MOV	#0x5A80,&Wa...		004404: 40B2 5A80 015C
	MOV	#0x0014,R6		00440a: 4036 0014
	CALL	#FIB		00440e: 12B0 4414
	JMP	$	➔➔➔	004412: 3FFF
FIB:	DECD	R6		004414: 8326
	MOV	#0x2400,R5		004416: 4035 2400
	CLR	0(R5)		00441a: 4385 0000
	INCD	R5	➔➔➔	00441e: 5325
	MOV	#1,0(R5)		004420: 4395 0000
LOOP:	INCD	R5		004424: 5325
	MOV	-2(R5),0(R5)		004426: 4595 FFFE 0000
	ADD	-4(R5),0(R5)	➔➔➔	00442c: 5595 FFFC 0000
	DEC	R6		004432: 8316
	JNE	LOOP		004434: 23F7
	RET			004436: 4130

Por isso, o termo **Montador** (*Assembler*) é usado para designar o programa que "monta" os *opcodes* correspondentes às instruções em *assembly*. O termo **Compilador** é usado para designar o programa que analisa o código escrito em uma linguagem qualquer e o transforma em um código executável. Um **Compilador** separa os átomos de uma linguagem (analisador léxico), confere a regras de sintaxe (analisador sintático) e gera o executável (analisador semântico).

Assim, por exemplo, se diz **Montar um programa *assembly*** e **Compilar um programa em C**. É errado dizer "Compilar um programa *assembly*".

O programa *assembly* tem um aspecto típico, composto por 4 campos como pode ser visto na listagem abaixo:

Campo 1: *labels* (sempre na primeira coluna)
Campo 2: mnemônicos
Campo 3: operandos
Campo 4: comentários (tudo que vem depois do ";")

O campo ***label*** faz referência ao endereço da instrução que está à direita. É uma forma de indicar ao montador o destino dos saltos e desvios. É possível usar *label* para indicar campos de dados. O *label* deve começar na primeira coluna do editor. Tudo que é colocado nesta primeira coluna é encarado com *label*. Cuidado para não colocar instruções nessa primeira coluna.

Os campos mnemônico (mne) e operandos já foram discutidos. O montador entende como comentário tudo que estiver entre um ";" e o final da linha. Ele se destina a dar

legibilidade aos programas. Aos escrever seus comentários, indique o que a instrução significa para o programa. Evite comentar o que a instrução faz. Por exemplo, para instrução `INCD R5`, não se deve colocar o comentário "incrementar R5", mas sim "avançar ponteiro", que é o que faz sentido para o programa.

Use "TAB" para separar os campos. Isso facilita muito a legibilidade dos programas.

```
-label-     mne    --operandos----------  ;----comentários
RESET       mov    #__STACK_END,SP        ;Initialize stackpointer
StopWDT     mov    #WDTPW|WDTHOLD,&WDTCTL ;Stop watchdog timer
M01E07:     MOV    #20,R6                 ;QTD DE NÚMEROS
            CALL   #FIB
            JMP    $                      ;Loop infinito
            ;
FIB:        DECD   R6                     ;APENAS QTD-2 NÚMEROS
            MOV    #0X2400,R5
            MOV    #0,0(R5)               ;Primeiro Nr
            INCD   R5                     ;avançar ponteiro
            MOV    #1,0(R5)               ;Segundo Nr
LOOP:       INCD   R5                     ;avançar ponteiro
            MOV    -2(R5),0(R5)
            ADD    -4(R5),0(R5)
            DEC    R6                     ;contador-1
            JNZ    LOOP                   ;continua?
            RET
```

2.2. Instruções do MSP430

O MSP430 possui 51 instruções, que estão listadas na Tabela 2.2. Dessas instruções, 27 instruções são nativas e 24 instruções são emuladas. Por instrução emulada se entende instrução que foi criada para facilitar o trabalho do programador. Na hora da montagem, cada a instrução emulada é convertida em uma das 27 instruções nativas. Entretanto, o programador não precisa ser preocupar com isso. A título de clareza, as instruções nativas foram numeradas de 1 até 27. As 24 instruções não numeradas são as emuladas.

Nesta tabela, a coluna rotulada de *Word/Byte* indica quais instruções fazem uso do modificador ".B", que seleciona operação em 8 *bits*, ou do modificador ".W" , que seleciona operação em 16 *bits*. No caso de omissão, o montador supõe a operação em 16 *bits*. Para algumas instruções, o uso do ".B" ou do ".W" não faz sentido. Por exemplo, a instrução `SETC`, que coloca em 1 o *flag* Carry (C=1).

Além disso, na extrema direita estão 4 colunas que mostram com um "*" as *flags* que podem ser alterados em função da operação realizada. O sinal "-" é usado para sinalizar que as *flags* permanecem no estado anterior.

Para indicar as operações realizadas pelas instruções foram usados os símbolos listados na Tabela 2.3 (inspirados na Linguagem C). Lembre-se de que o Contador de Programa (PC = R0) indica o endereço da próxima instrução a ser executada, de que o Ponteiro da Pilha (SP = R1) indica sempre o Topo da Pilha (TOS) e de que o Registrador de *Status* (SR = R2) contém as flags V, N, Z e C.

Tabela 2.2. Conjunto de Instruções, parte 1

Ord	Word *Byte*	Mnemônico	Oper.	Descrição	Flags (SR)			
					V	N	Z	C
	W/B	ADC	dst	dst + C → dst	*	*	*	*
1	W/B	ADD	src,dst	src + dst → dst	*	*	*	*
2	W/B	ADDC	src,dst	src + dst + C → dst	*	*	*	*
3	W/B	AND	src,dst	src & dst → dst	0	*	*	*
4	W/B	BIC	src,dst	~src & dst → dst	-	-	-	-
5	W/B	BIS	src,dst	src | dst → dst	-	-	-	-
6	W/B	BIT	src,dst	src & dst	0	*	*	*
	-	BR	dst	dst → PC	-	-	-	-
7	-	CALL	dst	push PC+2, dst → PC	-	-	-	-
	W/B	CLR	dst	0 → dst	-	-	-	-
	-	CLRC	–	C = 0	-	-	-	0
	-	CLRN	–	N = 0	-	0	-	-
	-	CLRZ	–	Z = 0	-	-	0	-
8	W/B	CMP	src,dst	dst - src	*	*	*	*
	W/B	DADC	dst	BCD: dst + C → dst	*	*	*	*
9	W/B	DADD	src,dst	BCD: src + dst + C → dst	*	*	*	*
	W/B	DEC	dst	dst - 1 → dst	*	*	*	*
	W/B	DECD	dst	dst - 2 → dst	*	*	*	*
	-	DINT	–	GIE = 0	-	-	-	-
	-	EINT	–	GIE = 1	-	-	-	-
	W/B	INC	dst	dst + 1 → dst	*	*	*	*
	W/B	INCD	dst	dst + 2 → dst	*	*	*	*
	W/B	INV	dst	~dst → dst	*	*	*	*
10	-	JC (JHS)	label	se C=1, salta	-	-	-	-
11	-	JEQ (JZ)	label	se Z=1, salta	-	-	-	-
12	-	JGE	label	se N^V=0, salta	-	-	-	-
13	-	JL	label	se N^V=1, salta	-	-	-	-
14	-	JMP	label	salta	-	-	-	-
15	-	JN	label	se N=1, salta	-	-	-	-
16	-	JNC (JLO)	label	se C=0, salta	-	-	-	-
17	-	JNE (JNZ)	label	se N=0, salta	-	-	-	-

Tabela 2.2. (continuação) Conjunto de Instruções, parte 2

Ord	Word *Byte*	Mnemônico	Oper.	Descrição	Flags (SR)			
					V	N	Z	C
18	W/B	MOV	src,dst	src → dst	-	-	-	-
	-	NOP	–	-	-	-	-	-
	W/B	POP	dst	TOS → dst, SP +2 → SP	-	-	-	-
19	W/B	PUSH	src	SP – 2 → SP, src → TOS	-	-	-	-
20	-	RETI	–	TOS → SR, SP + 2 → SP TOS → PC, SP + 2 → SP	*	*	*	*
	-	RET	–	TOS → PC, SP + 2 → SP	-	-	-	-
	W/B	RLA	dst	C << dst << 0	*	*	*	*
	W/B	RLC	dst	C << dst << C	*	*	*	*
21	W/B	RRA	dst	MSb >> MSb >> dst >> C	0	*	*	*
22	W/B	RRC	dst	C >> dst >> C	*	*	*	*
	W/B	SBC	dst	dst – C → dst	*	*	*	*
	-	SETC	–	C = 1	-	-	-	1
	-	SETN	–	N = 1	-	1	-	-
	-	SETZ	–	Z = 1	-	-	1	-
23	W/B	SUB	src,dst	dst + ~src + 1 → dst	*	*	*	*
24	W/B	SUBC	src,dst	dst + ~src + C → dst	*	*	*	*
25	-	SWPB	dst	MSB <> LSB	-	-	-	-
26	-	SXT	dst	bit7 → bit8, ..., bit15	0	*	*	*
	W/B	TST	dst	dst - 0	*	*	*	*
27	W/B	XOR	src,dst	src ^ dst → dst	*	*	*	*

Tabela 2.3. Símbolos usados para apresentar o Conjunto de Instruções

Símbolo	Significado
dst	destination
src	source
PC	Program Counter
SP	Stack Pointer
TOS	Top of Stack
Label	Endereço
*	Flag pode mudar
-	Flag inalterado
<<	Desloc. esquerda

Símbolo	Significado
~	Complemento a 1
&	AND *bit* a bit
|	OR *bit* a bit
^	XOR *bit* a bit
V	Overflow
N	Negative
Z	Zero
C	Carry
>>	Desloc. direita

Não é objetivo deste trabalho detalhar o que faz cada instrução. Esta tarefa é executada pelo **Apêndice I**. Entretanto, abordaremos aquelas que costumam gerar dúvidas. As

instruções de rotação se enquadram neste caso. A Figura 2.1 apresenta de forma gráfica a operação realizada para cada uma dessas instruções de rotação.

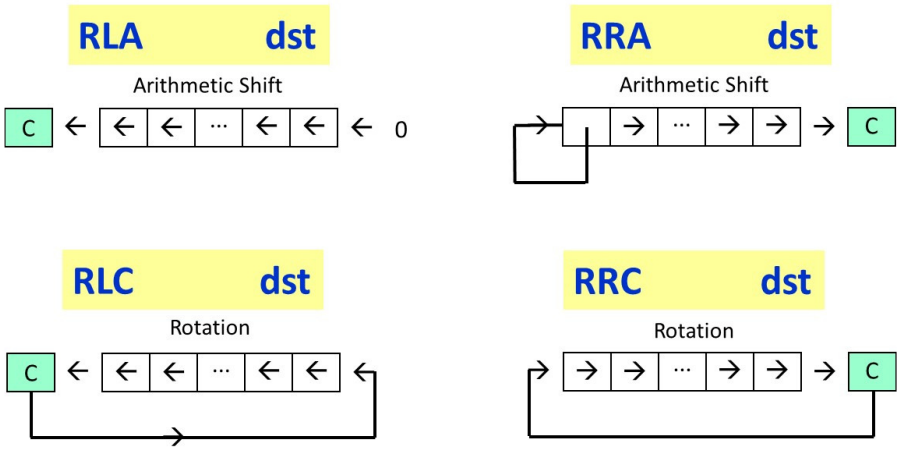


Figura 2.1. Detalhamento das instruções de rotação.

A instrução `SWPB dst`, faz a troca dos *bytes* de uma palavra de 16 *bits*, conforme mostrado na Figura 2.2.

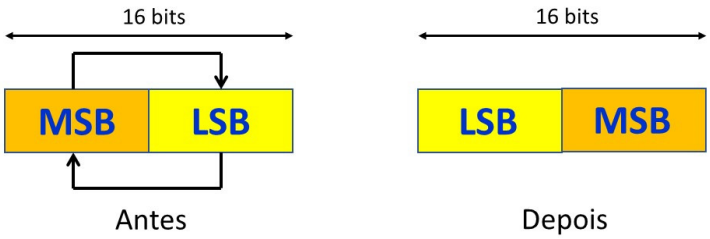


Figura 2.2. Detalhamento da execução da instrução `SWPB dst`.

Uma operação muito usada na programação é a comparação de dois números para saber se são iguais, diferentes ou qual deles é o maior. Este tipo de comparação pode ser feito considerando os números com sinal ou sem sinal. Para tal foi prevista a instrução `CMP src,dst`. Essa instrução faz a subtração `dst - src`, mas não guarda o resultado, apenas atualiza as flags V, N, Z e C. Julgando pelo estado destas *flags*, é possível decidir a comparação. A Tabela 2.4 abaixo apresenta as possibilidades quando se compara números sem sinal e com sinal (representados em complemento a 2).

Tabela 2.4. Comparação entre dois valores.
Considera-se que antes foi executada a instrução de comparação (`CMP src,dst`).

Sinal ?	Instrução	Salta se ...	Condição	Flag	Equivalente
-	JEQ	Equal	**dst** = src	Z = 1	JZ
	JNE	Not Equal	**dst** ≠ src	Z = 0	JNZ
Sem sinal	JHS	Higher or same	**dst** ≥ src	C = 1	JC
	JLO	Lower	**dst** ≤ src	C = 0	JNC
Com sinal	JGE	Greater or equal	**dst** ≥ src	N V = 0	JC
	JL	Less than	**dst** ≤ src	C = 0	JNC

2.3. Modos de Endereçamento

Na Tabela 2.2 foram apresentadas todas as instruções do processador MSP430. A grande maioria envolvia dois operandos, designados com **src** e **dst**. O acrônimo **src**, do inglês "*source*" indica a origem (fonte) do operando e o acrônimo **dst** indica o destino do operando. Podemos então dizer que:

> A instrução busca o dado a ser operado no local indicado como **src**, trabalha ou opera sobre este dado e armazena o resultado no local indicado como **dst**.

Algumas instruções não têm o campo de operandos. Por exemplo, as instruções `SETC` (faz a *flag* C = 1) ou `SETZ` (faz a *flag* Z = 1), já indicam no próprio mnemônico o operando. Outro bom exemplo é a instrução `DINT` (faz o *bit* GIE = 0), que desabilita todas as interrupções. Note também que a instrução de retorno de sub-rotina `RET`, não tem o campo de operandos. Isso porque estão implícitos os operandos envolvidos (PC e SP) e as operações realizadas com esses operandos são sempre as mesmas.

Como já foi afirmado, o MSP430 opera em 8 ou 16 *bits*. O tamanho do operando é indicado pelas letras ".W" ou ".B", colocada logo após o mnemônico. Se nada for indicado, o montador supõe que os operandos são de 16 *bits*. Veja os três exemplos a seguir:

- `MOV.B src,dst` → copia do "local de 8 *bits*" `src` para o "local de 8 *bits*" `dst`;
- `MOV.W src,dst` → copia do "local de 16 *bits*" `src` para o "local de 16 *bits*" `dst` e
- `MOV src,dst` → copia do "local de 16 *bits*" `src` para o "local de 16 *bits*" `dst`;

Visto esses detalhes, vamos agora à finalidade principal deste tópico, que é a de detalhar as maneiras (ou formas) de se indicar de onde vem os operando e para onde eles vão.

São 7 os modos de endereçamentos. Por simplicidade, todos os exemplos a seguir usarão a instrução `MOV src,dst`.

2.3.1. Modo Registrador (*Register Mode*)

Instrução	Operação
`MOV Rn,Rm`	Rn → Rm

Copia para Rm o conteúdo de Rn

Neste modo, o programador indica os registradores que serão envolvidos. Exemplos:

- `MOV.W R5,R6` → Copia o conteúdo (16 *bits*) de R5 para R6
- `MOV.B R5,R6` → Copia o LSB do conteúdo (8 *bits*) de R5 para o LSB de R6. Cuidado, neste tipo de endereçamento o MSB do `dst` (R6 no caso do exemplo) é zerado.

2.3.2. Modo Indexado (*Indexed Mode*)

Instrução	Explicação
`MOV X(Rn),Rm`	(X+Rn) → Rm

Copia para Rm o conteúdo da posição de memória cujo endereço é dado pela soma do conteúdo de Rn com o número X.

Neste modo, o programador, além de especificar os registradores envolvidos, indica um número a ser somando ao conteúdo do registrador fonte para gerar o endereço do operando fonte. A expressão "(X+Rn)" na coluna "explicação" é usada para indicar que se lê a posição de memória cujo endereço é dado pela soma do conteúdo de Rn com o número X. Exemplo:

- (considere R5=4) `MOV.W 8(R5),R6` → Copia para R6 a palavra de 16 *bits* que está no endereço 12 (8 + 4 = 12) da memória.

Este modo é muito usado para acessar vetores. Por exemplo, acessar o vetor que inicia no endereço `vet8` e usa R5 como indexador: `MOV.W vet8(R5),R6`.

2.3.3. Modo Simbólico (*Symbolic Mode*)

Instrução	Explicação
`MOV X,Rm`	(X+PC) → Rm

Copia para Rm o conteúdo da posição de memória cujo endereço é dado pela soma do conteúdo do Contador de Programa (PC) com o número X.

Este modo é semelhante ao anterior, mas não se indica o registrador fonte. Isto porque está implícito o uso do contador de programa (PC). Exemplo:

- (considere PC=100) `MOV.W 8,R6` → Copia o para R6 a palavra de 16 *bits* que está no endereço 108 (8 + 100 = 108) da memória.

2.3.4. Modo Absoluto (*Absolute Mode*)

Instrução	Explicação
`MOV &X,Rm`	(X) → Rm

Copia para Rm o conteúdo da posição de memória cujo endereço é dado pelo número X.

Continua semelhante ao modo anterior, mas surgiu o símbolo "&". Neste modo, o programador pode indicar de forma absoluta a posição de memória que ele deseja acessar.

2.3.4.a. Modo Relativo ao SP (SP-*Relative Mode*)

Instrução	Explicação
`MOV X(SP),Rm`	(X+SP) → Rm

Copia para Rm o conteúdo da posição de memória cujo endereço é dado pela soma do conteúdo do Ponteiro da Pilha (SP) com o número X.

O fabricante (Texas Instruments) não separa este modo, mas diversos autores e empresas o fazem, daí a letra "a" na numeração deste item. Este modo lembra bastante o indexado (item 2.3.2) onde o Rn foi substituído por SP. É muito usado para acessar posições na pilha. Exemplo:

- (considere SP=100) `MOV.W 8(SP),R6` → Copia para R6 a palavra de 16 *bits* que está no endereço 108 (8 + 100 = 108) da memória. É acessado o quarto elemento da pilha, pois cada elemento ocupa 2 *bytes*. Isto será visto mais adiante.

2.3.5. Modo Indireto via Registrador (*Indirect Register Mode*)

Instrução	Explicação
`MOV @Rn,Rm`	(Rn) → Rm

Copia para Rm o conteúdo da posição de memória cujo endereço é o conteúdo de Rn.

Agora, ao invés de indicar diretamente o endereço do operando, indicamos o registrador onde está esse endereço. Esse modo é muito usado para acessar posições em sequência na memória.

- (considere R5=100) `MOV.W @R5,R6` → Copia o para R6 a palavra de 16 *bits* de endereço 100.

2.3.6. Modo Indireto via Registrador com Autoincremento (*Indirect Autoincrement Register Mode*)

Instrução	Explicação
`MOV @Rn+,Rm`	(Rn) → Rm

Copia para Rm o conteúdo da posição de memória cujo endereço é o conteúdo de Rn e, em seguida, incrementa Rn.

Este modo é muito semelhante ao anterior, mas após a transferência, o conteúdo de Rn é incrementado. Se o acesso for de 8 *bits*, este incremento é de 1 (Rn = Rn + 1), caso seja um acesso de 16 *bits*, o incremento é de 2 (Rn = Rn + 2). Facilita ainda mais o acesso de posições sequenciais na memória.

- (considere R5=100) `MOV.W @R5+,R6` → Copia o para R6 a palavra de 16 *bits* de endereço 100 e depois incremente R5 que fica com conteúdo igual a 102.

2.3.7. Modo Imediato *(Immediate Mode)*

Instrução	Explicação
`MOV #N,Rm`	N → Rm

O número N é colocado em Rm.

Agora o programador indica de forma imediata o dado que ele quer operar. Exemplo.

- `MOV.W #1234,R6` → Faz R6 = 1234.

2.3.8. Resumo

A tabela abaixo resume os 7 modos de endereçamento do MSP. Lembramos que na coluna "Operando", o parêntesis é usado para indicar o acesso a uma posição da memória. Os modos de número 5, 6 e 7 não podem ser usados como destino.

Tabela 2.5. Resumo dos modos de endereçamento do MSP430

Nr	Nome do Modo	Modo	Operando
1	Registrador	`Rn`	Rn
2	Indexado	`X(Rn)`	(X+Rn)
3	Simbólico	`X`	(X+PC)
4	Absoluto	`&ADR`	(ADR)
5	Indireto via Reg.	`@Rn`	(Rn)
6	Indireto via Reg. com autoincr.	`@Rn+`	(Rn) e Rn++
7	Imediato	`#N`	N

Pergunta: De quantas diferentes maneiras podemos combinar os campos fonte e destino?
Resposta: De 35 diferentes maneiras, pois são 7 modos para o campo fonte e apenas 5 para o campo destino.

2.4. Algumas Particularidades do CCS

Neste tópico, vamos abordar algumas particularidades do montador disponível no CCS. Para nosso nível inicial de programação, estas particularidades não farão grandes diferenças. Por isso, recomendamos que o leitor apenas "passe os olhos" e retorne depois para verificar os itens específicos de que necessitar.

O conceito de um montador é claro e muito bem definido. Entretanto, cada um apresenta suas particularidades. Neste tópico abordaremos algumas particularidades para o programador se sentir confortável com o emprego do montador integrado do CCS. A referência usada foi o documento *slau131g-Assembler.pdf* (MSP430 Assembly Language Tools v 4.1, User's Guide).

- A linha de instrução dever ter menos de 400 caracteres. Os comentários podem ultrapassar este limite, mas serão truncados.
- Os 4 campos (*label* – mnemônico – operando – comentário) devem ser separados por espaços ou tabulações.
- Na primeira coluna deve estar um *label*, ou um caracter em branco, ou um asterisco, ou ainda um ponto-e-vírgula.
- Os *labels* devem começar na primeira coluna.
- Os comentários podem iniciar na primeira coluna, sendo indicados por ponto-e-vírgula ou asterisco.
- Os comentários fora da primeira coluna devem começar com ponto-e-vírgula.
- Os *labels* fazem distinção entre letras maiúsculas e minúsculas.
- São caracteres válidos para os *labels*: A-Z, a-z, 0-9, _, e $.

Constantes Aceitas pelo Montador do CCS

É preciso saber como especificar as constantes (números) para o montador. De acordo com o problema, é mais simples definir as constantes em decimal, em hexadecimal ou ainda em binário. Assim, vamos ver como indicá-las. A Tabela 2.6 apresenta alguns exemplos.

Inteiro Decimal: se nada for sinalizado, se supõe que seja uma constante decimal. As constantes decimais ficam na faixa: -2 147 483 648 até 4 294 967 295 (32 *bits* com sinal ou sem sinal).

Inteiro Hexadecimal: é uma cadeia de até 8 caracteres (32 *bits*), seguida por um "H" ou "h", ou ainda precedida por um "0x". O primeiro caracter de um número hexadecimal não pode ser uma letra, precisa ser um número na faixa 0 até 9. Ao invés de escrevermos F8H, escrevemos 0F8H.

Inteiro Octal: é uma cadeia de até 11 números (32 *bits*), seguida por um "Q" ou "q".

Inteiro Binário: é uma cadeia de até 32 caracteres (0 ou 1), seguida por um "B" ou "b".

Constante Caracter: é um caracter entre aspas simples. Seu valor é o código ASCII do caracter entre as aspas.

Constante String: é uma sequência de caracteres entre aspas duplas. Cada letra é substituída pelo seu código ASCII.

Constante Ponto-Flutuante: é uma cadeia de dígitos com um ponto decimal e um expoente opcional. É preciso conter o ponto decimal. São exemplos: 3.0, 3.14, 3.14e-5.

Tabela 2.6: Alguns exemplos de representação de números

Decimal	Hexadecimal	Octal	Binário
5	5H	05Q	101B
100	64H	144Q	01100100B
232	E8h	350q	11101000b
1000	3E8h	1750q	001111101000b
54321	0xD4E1	152061Q	1101010000110001B (loucura!)

Constantes Simbólicas

É possível usar símbolos para representar constantes numéricas. A indicação é feita com as diretivas `.set` ou `.equ`. Veja os exemplos abaixo.

```
ACLK  .set  32678
SMCLK .equ  32*ACLK     ;SMCLK = 1.048.576
PI    .set  3.14159
```

Operadores

Existe uma grande quantidade de operadores que o programador por usar para ajudar em sua programação. Não é nosso objetivo abordar todos os casos, entretanto a lista completa é apresentada abaixo.

Tabela 2.7. Operadores disponíveis (prec. representa a precedência)

Prec.	Op.	Descrição
1	+	Unário +
1	-	Unário -
1	~	Complemento 1

Prec.	Op.	Descrição
4	>>	Shift right
5	<	Menor que
5	<=	Menor ou igual a

1	!	NOT lógico
2	*	Multiplicação
2	/	Divisão
2	%	Módulo
3	+	Adição
3	-	Subtração
4	<<	Shift left

5	>	Maior que
5	>=	Maior ou igual a
6	=	Igual
6	!=	Diferente
7	&	AND bit-a-bit
8	^	XOR bit-a-bit
9	|	OR bit-a-bit

Apresentamos alguns exemplos.

```
ACLK  .set  1<<15        ;ACLK = 32.768
SMCLK .set  ACLK<<5      ;SMCLK = 1.048.576
2PI   .set  2*3.14159
K1    .set  1234Q
...
      MOV   #~K1+1,R5    ;R5 = -1234Q
```

Funções Matemáticas Incluídas

Várias funções matemáticas que podem ser úteis ao programador estão disponíveis no montador do CCS. A tabela a seguir apresenta a lista e uma breve descrição de cada uma delas. Caso necessite usar uma dessas funções, o leitor é recomendado fazer um teste para se certificar de que está correta sua interpretação da operação realizada pela função. É óbvio, mas não custa comentar que o argumento *expr* nunca pode ser um registrador.

Tabela 2.8. Funções matemáticas disponíveis

Função	Descrição
$acos(*expr*)	Arco cosseno
$asin(*expr*)	Arco seno
$atan(*expr*)	Arco tangente
$atan2(*expr*)	Arco tangente [-π,π]
$ceil(*expr*)	Arredonda para cima
$cos(*expr*)	Cosseno
$cosh(*expr*)	Cosseno hiperbólico
$cvf(*expr*)	Converte para *float*
$cvi(*expr*)	Converte para inteiro
$exp(*expr*)	e^*expr*
$fabs(*expr*)	Converte para float o valor absoluto de *expr*
$floor(*expr*)	Arredonda para baixo

$fmod(*expr1,expr2*)	Módulo de *expr1*/*expr2*
$int(*expr*)	Retorna 1 se *expr* for inteiro, 0 caso contrário
$ldexp(*expr1,expr2*)	*expr1 * 2^expr2*
$log(*expr*)	Log natural de *expr*
$log10(*expr*)	Log na base 10 de *expr*
$max(*expr1,expr2*)	Retorna o maior
$min(*expr1,expr2*)	Retorna o menor
$pow(*expr1,expr2*)	*expr1 ^ expr2*
$round(*expr*)	Arredondamento para o inteiro mais próximo
$sgn(*expr*)	Retorna o sinal de *expr*
$sin(*expr*)	Seno
$sinh(*expr*)	Seno hiperbólico
$sqrt(*expr*)	Raiz quadrada
$strtod(*str*)	Converte string str para ponto flutuante (prec. Dupla)
$tan(*expr*)	Tangentge
$tanh(*expr*)	Tangente hiperbólica
$trunc(*expr*)	Trunca

Diretivas para o Montador

Algumas diretivas permitem orientar o montador sobre o destino de trechos do código. Apresentamos apenas as que julgamos interessantes para nosso nível de programação.

- **.data** → indica ao montador que o destino do código é a seção "data", que é a seção de dados inicializados.
- **.text** → indica ao montador que o destino do código é a seção "text", que é a seção de códigos executáveis.
- **.space** → indica ao montador para reservar uma certa quantidade de *bytes* dentro de uma seção.

Apresentamos a seguir as diretivas que permitem inicializar valores na memória.

- ***.byte*** → inicializa com um ou mais *bytes*.
- **.char** → inicializa com um ou mais *bytes*.
- **.cstring** → inicializa com uma ou mais *strings*, terminadas com zero.
- **.double** → inicializa com um ou mais ponto flutuantes em precisão dupla.
- **.float** → inicializa com um ou mais ponto flutuantes.
- **.int** → inicializa com um ou mais inteiros de 16 bits.
- **.long** → inicializa com um ou mais inteiros de 32 bits.
- **.string** → inicializa com uma ou mais *strings*.
- **.word** → inicializa com um ou mais inteiros de 16 bits.

Alguns exemplos:

```
        .data
```

```
SEQ:  .byte       10, -1, "abc", 'a'      ;
VET:  .space      100                     ;reservou espaço de 100 bytes
      .word       100,42, -123
MSG1: .cstring    "Mensagem"              ;é terminada com 0
MSG2: .string     "Aviso"                 ;não é terminada com 0
```

2.5. Exercícios Resolvidos

A seguir são apresentados diversos exercício resolvidos. O leitor é convidado a executar todos os exercícios aqui apresentados usando o Code Composer Studio (CCS) e a placa **MSP430 F5529 launch pad**.

Nestes exercícios, os números em notação hexadecimal são precedidos por "0x". Por exemplo 0x50, corresponde ao número decimal 80. Se nada for indicado, o número é interpretado como decimal.

As soluções dos primeiros programas serão diretas, mais adiante passaremos a usar duas fases: a fase de inicialização e a fase de execução do programa solução.

> De nada adianta simplesmente ler os problemas apresentados e suas soluções. Seria como tentar aprender a andar de bicicleta apenas observando os ciclistas no parque.
> É preciso que o leitor tente esboçar sua solução e depois a compare com a solução apresentada. É necessário desenvolver a habilidade de *pensar em assembly*.

ER 2.1. Considerando R5 = 0x1234 e R6 = 0x4321, escreva um programa que armaze em R6 soma em 16 *bits* de R5 + R6 (R6 = R5 + R6).

Solução:
Basta usar a instrução `ADD.W R5,R6`. No programa listado abaixo, as duas primeiras instruções inicializam os registradores de acordo com o pedido pelo exercício e a terceira instrução realiza a soma, já armazenando o resultado em R6. A instrução seguinte `JMP $` é usada para prender o MSP em um laço infinito. O símbolo $ representa o endereço da própria instrução. Assim instrução ordena ao processador para saltar para o endereço da instrução de salto, ou seja, ele fica eternamente preso nesse laço. Esse truque é muito usado para quando se deseja impedir que o processador siga adiante.

Listagem da solução do ER 2.1

```
;ER 2.1
```

```
      MOV.W #0x1234,R5  ;inicializar R5
      MOV.W #0x4321,R6  ;inicializar R6
      ADD.W R5,R6       ;somar R6 = R5 + R6 =0x5555
      JMP   $           ;prender MSP
```

Resposta: R6 = 0x5555;

ER 2.2. Considerando R5 = 0x1234 e R6 = 0x 4321, armazenar em R6 soma em 8 *bits* de R5 + R6 (R6 = R5 + R6).

Solução:
A solução é muito semelhante à do exercício anterior, apenas a instrução de soma mudou para a versão de 8 *bits*: `ADD.B R5,R6`. Após executar o programa, veja que apenas o LSB de R6 recebeu o resultado da soma e que o MSB de R6 foi zerado.

Listagem da solução do ER 2.2

```
;ER 2.2
      MOV.W #0x1234,R5  ;inicializar R5
      MOV.W #0x4321,R6  ;inicializar R6
      ADD.B R5,R6       ;somar R6 = R5 + R6 =0x0055
      JMP   $           ;prender MSP
```

Resposta: R6 = 0x0055;

ER 2.3. Considerando R5 = 0xABCD e R6 = 0xDCBA, armazenar em R6 soma em 16 *bits* de R5 + R6 (R6 = R5 + R6).

Solução:
Este exercício tem a intenção de mostrar que a soma pode ultrapassar o tamanho da representação de seus elementos. No presente caso, o resultado correto é 0x18887, porém esse é um número de 17 *bits*. Assim, ao fazer a soma, a CPU armazena o resultado no destino (R6 = 0x8887) e faz *carry* = 1 para indicar que precisou de mais um bit.

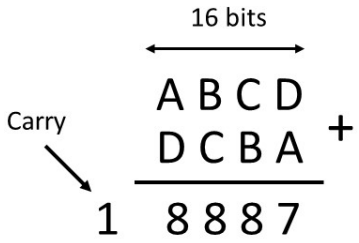


Figura 2.3. Soma de dois números de 16 bits, resultando num resultado de 17 bits.

Listagem da solução do ER 2.3

```
;ER 2.3
      MOV.W #0xABCD,R5  ;inicializar R5
      MOV.W #0xDCBA,R6  ;inicializar R6
      ADD.W R5,R6       ;somar R6 = R5 + R6 = 0x8887 e Cy = 1
      JMP   $           ;prender MSP
```

Resposta: R6 = 0x8887 e *Carry* = 1.

ER 2.4. Usaremos agora a memória RAM do MSP que inicia em 0x2400. É pedido para armazenar em R10 soma dos dois *bytes* que estão nas posições 0x2400 e 0x2401.

Solução:
Este exercício considera que existem dois *bytes* nas posições 0x2400 e 0x2401. Não sabemos seus valores, apenas seus endereços. O programa solução usa o endereçamento direto para copiar o conteúdo de 0x2400 para R10 e depois a instrução de soma endereçando a posição 0x2401. Note o uso do ".B" para indicar que as operações são com *bytes*.

É claro que para testar o programa precisaremos especificar o conteúdo das posições 0x2400 e 0x2401. Podemos pedir para o montador fazer isso usando a diretiva (pseudo-instrução) `.byte 0x33, 0x44`. Ela é chamada de pseudo-instrução porque não é uma instrução do MSP, mas sim uma orientação para o montador. Para indicar ao montador que queremos carregar os dados a partir do endereço 0x2400, colocamos antes a pseudo-instrução `.data`. Indicamos assim ao montador que deste ponto em diante, deve ser usada a memória de dados que começa em 0x2400. Para o ensaio desta solução estamos colocando na memória os números 0x33 e 0x44. A resposta, é claro, será 0x77. O leitor deve testar o programa com outros valores.

Particularidade do CCS: Precisamos fazer uma referência aos dados, do contrário o otimizador do CCS não os carrega na memória. Isto é feito com linha `DT: .byte 0x33, 0x44`, onde DT é uma constante cujo conteúdo é o endereço da primeira posição. A referência é conseguida com a linha `MOV.B &DT,R10`.

Listagem da solução do ER 2.4

```
;ER 2.4
      MOV.B &DT,R10     ;R10 = primeiro byte
      ADD.B &0x2401,R10 ;Somar R10 com o segundo byte
      JMP   $           ;Prender MSP

      .data
```

```
DT:   .byte 0x33, 0x44  ;Inicializar posições 0x2400 e 0x2401
```

Resposta: R6 = 0x77 e Carry = 0.

Observações sobre o Montador do CCS (mais detalhes na Seção 2.4):
Observação 1) O montador do CCS permite diversas facilidades com o uso de pseudo-instruções. A tabela a seguir apresenta algumas delas. Note que elas nunca podem ser colocadas na primeira coluna do texto do programa.

Tabela 2.6. Algumas pseudo-instruções usadas pelo CCS

Pseudo-Instr	Significado
.text	Define a seção de código (0x4400)
.data	Define a seção de dados (0x2400)
.bss	Define a seção de dados não inicializados
.*byte*	Carrega 8 *bits* na memória
.word	Carrega 16 *bits* na memória

Observação 2) O leitor deve ter notado que o CCS, quando prepara o ambiente para seu programa, insere as duas linhas abaixo:

- `RESET  mov.w #__STACK_END,SP` → esta instrução inicializa o ponteiro da pilha de forma a permitir o uso de sub-rotinas e a pilha.
- `StopWDT mov.w #WDTPW|WDTHOLD,&WDTCTL` → Esta instrução inibe o Cão de Guarda (*watchdog Timer*), que é um temporizador que se não for impedido, periodicamente, provoca o resete do processador. Ele será estudado mais adiante.

ER 2.5. Este exercício é semelhante ao anterior, mas opera com 16 *bits*. É pedido para armazenar em R10 soma das palavras de 16 *bits* que estão nas posições 0x2400 e 0x2402 (note os endereços pares).

Solução:
Para executar o programa solução, precisaremos armazenar duas palavras de 16 *bits* nas posições 0x2400 e 0x2402. Com a pseudo-instrução `.data` indicamos ao montador que a partir deste ponto, deve ser usada a memória de dados que começa em 0x2400. A pseudo-instrução `.word 0x2233, 0x5544` indica quais palavras de 16 *bits* devem ser armazenadas. Para o presente ensaio, estamos colocando na memória os números 0x2233 e 0x5544. A resposta, é claro, será 0x7777. O leitor deve testar o programa com outros valores.

Listagem da solução do ER 2.5

```
;ER 2.5
      MOV.W &DT,R10     ;R10 = primeira palavra
      ADD.W &0x2402,R10 ;Somar R10 com a segunda palavra
      JMP   $           ;Prender MSP

      .data                   ;Indicar uso da memória de dados
DT:   .word 0x2233, 0x5544    ;Inicializar posições de 16 bits
```

Resposta: R10 = 0x7777.

Observação
A partir de agora vamos estruturar melhor os exercícios e as soluções. Usaremos uma fase de inicialização e uma fase execução. A fase de inicialização apenas inicializa o ambiente de acordo com o pedido do exercício e a fase de execução será a chamada da sub-rotina que soluciona o que foi pedido. Muitas vezes a fase de inicialização será muito simples e conterá apenas a chamada da sub-rotina.

ER 2.6. Este exercício é idêntico ao anterior, mas pede a elaboração de uma sub-rotina. É pedido a sub-rotina de nome SOMA2, que armazena em R10 a soma das palavras de 16 *bits* que estão nas posições 0x2400 e 0x2402.

Solução:
Note que neste caso, a fase de inicialização apenas a chama a sub-rotina (`CALL #SOMA2`) e depois prende a execução (`JMP $`). O início da sub-rotina é indicado com o label `SOMA2:`, que deve ser colocado na primeira coluna de texto. É claro que para testar o programa, devemos armazenar duas palavras quaisquer de 16 *bits* nas posições 0x2400 e 0x2402 usando a pseudo-instrução `DT: .word 0x2233, 0x5544`. Note que a sub-rotina usou as referências `DT` e `DT+2` para acessar os dois endereços.

Listagem da solução do ER 2.6

```
;ER 2.6
            CALL  #SOMA2      ;Chamar sub-rotina
            JMP   $           ;Prender MSP
            ;
SOMA2:      MOV.W &DT,R10     ;R10 = primeira palavra
            ADD.W &DT+2,R10   ;Somar R10 com a segunda palavra
            RET               ;Retornar
            ;
            .data                   ;Indicar uso da memória de dados
DT:         .word 0x2233, 0x5544    ;Inicializar posições
```

Resposta: R10 = 0x7777.

ER 2.7. Escreva a sub-rotina SOMA, que armazena em R10 o somatório dos números de 8 *bits* que estão armazenados a partir do endereço 0x2400. A quantidade de *bytes* a serem somados é indicada por R6. Por simplicidade, vamos supor que o somatório não ultrapasse a representação em 8 *bits*.

Solução:
Toda sub-rotina deve deixar bem claro os parâmetros que recebe, os parâmetros que retorna e os recursos que utiliza. Vamos trabalhar esse conceito.

Sub-rotina SOMA:
Recebe: R6 = quantidade de números de 8 *bits*, a partir de 0x2400, a serem somados;
Retorna: R10 = resultado do somatório.

Recursos a serem usados pela sub-rotina:
R5 = ponteiro, inicializado com 0x2400;
R6 = contador, decrementado a cada iteração e
R10 = somatório, inicializado com zero.

A sub-rotina SOMA vai somando em R10 os *bytes* a partir de 0x2400. A cada soma efetuada, o contador R6 é decrementado. Quando este decremento de R6 resultar em zero, a sub-rotina retorna. Vamos usar um registrador, no caso o R5, para indicar o endereço das posições a serem somadas. Um registrador cujo conteúdo é interpretado com endereço recebe o nome de ponteiro e a forma de endereçamento é o indireto, indicado pelo símbolo "@".

Listagem subrotina `SOMA`*, solução do ER 2.7*

```
;ER 2.7: Sub-rotina SOMA
SOMA:       CLR.W R10          ;Zerar somatório
            MOV.W #DT,R5       ;Inicializar ponteiro
LOOP:       ADD.B @R5,R10      ;Somar um byte em R10
            INC.W R5           ;Avançar ponteiro uma posição
            DEC.W R6           ;Decrementar contador
            JNZ   LOOP         ;Repetir se contador diferente de zero
            RET                ;Retornar se contador igual a zero
```

Algumas instruções merecem atenção especial. A instrução `ADD.B @R5,R10` que soma em R10 o *byte* cujo endereço está indicado (apontado) por R5. Nessa instrução o modificador ".B" é importante para indicar que a soma é em 8 *bits*. A instrução seguinte, `INC.W R5`, incrementa o ponteiro de endereço e, por ser endereço, tem tamanho de 16 *bits* e deve fazer uso do modificador ".W". Essas duas instruções poderiam ser

substituídas pela instrução `ADD.B @R5+,R10` que após a transferência, incrementa o ponteiro R5.

A instrução `DEC.W R6` decrementa o contador R6. Como nada foi dito, se supõe que o contador seja de 16 *bits*, por isso o uso do ".W". Se bem que, os detalhistas podem pensar: se o somatório deve "caber" em 8 *bits*, então R6 não pode ser maior que 256. Não é bem verdade, já que se pode ter uma grande quantidade de números iguais a zero. Além disso, fica a pergunta: e se forem números com sinal? Vamos nos preocupar com isso mais adiante.

Logo após a instrução `DEC.W R6` vem o teste do resultado deste decremento. Isto é feito com a instrução `JNZ LOOP`. Se o resultado do decremento for diferente de zero, o processador salta para o *label* (endereço) `LOOP:`, e mais um laço é executado. Se o resultado for igual a zero, a sub-rotina termina (obrigatoriamente com um `RET`).

Para testar o funcionamento da sub-rotina SOMA, precisamos criar um pequeno ambiente, com um certo valor em R6 e números colocados na memória a partir de 0x2400. Um exemplo é sugerido na listagem abaixo.

Listagem de um ambiente para teste da subrotina `SOMA`

```
; Ambiente para testar sub-rotina SOMA
            MOV.W #5,R6       ;Somatório de 5 números
            CALL  #SOMA       ;Chamar a sub-rotina a ser testada
            JMP   $           ;Prender MSP
;
; Coloque aqui a sub-rotina SOMA listada acima
;
            .data             ;Indicar uso da memória de dados
DT:         .byte 1,3,5,7,11  ;Números a serem somados
```

Sugestão: tente executar a sub-rotina, mas agora usando os valores a seguir. Qual será o resultado?

```
      .byte 100,100,100,100,100     ;Números a serem somados
```

ER 2.8. Vamos repetir o ER 2.7, mas agora armazenando o somatório em um espaço de 16 *bits*, ou seja, usaremos os 16 *bits* de R10.

Solução:
Na solução do exercício anterior, usamos a instrução `ADD.B @R5,R10` para somar um *byte* ao R10. Ela precisa ser melhor entendida. Como indicamos operação de 8 *bits* (.B), o processador armazena o resultado da soma no LSB de R10 e, ao mesmo tempo, zera o MSB de R10.

Pode parecer que para fazer a soma em 16 *bits* basta usar o ".W" para indicar que a operação é de 16 *bits*, ficando assim: `ADD.W @R5,R10`. Isso vai dar muito errado! Ao buscar o operando apontado por R5, o processador vai buscar um número de 16 *bits*, ou seja, vai ler dois *bytes* seguidos e arrumá-los de acordo com a arquitetura *Little Endian*. A soma, sim, vai acontecer em 16 *bits*, porém com o operando errado. O leitor é convidado a fazer esse ensaio usando a solução do exercício anterior.

Precisamos, então, indicar que a leitura do operando é em 8 *bits* e que a soma deve acontecer em 16 *bits*. A maneira mais simples de fazer isso é usar um registrador auxiliar.

Sub-rotina SOMA:
Recebe: R6 = quantidade de números de 8 *bits*, a partir de 0x2400, a serem somados;
Retorna: R10 = resultado do somatório.

Recursos a serem usados pela sub-rotina:
R5 = ponteiro, inicializado com 0x2400;
R6 = contador, decrementado a cada iteração e
R7 = registrador auxiliar para realizar a soma em 16 *bits*
R10 = somatório, inicializado com zero.

Listagem subrotina `SOMA`*, solução do ER 2.8*

```
;ER 2.8: Sub-rotina SOMA
SOMA:       CLR.W R10          ;Zerar somatório
            MOV.W #DT,R5       ;Inicializar ponteiro
LOOP:       MOV.B @R5,R7       ;Armazenar byte em R7
            ADD.W R7,R10       ;Somar o byte lido em R10
            INC.W R5           ;Avançar ponteiro uma posição
            DEC.W R6           ;Decrementar contador
            JNZ   LOOP         ;Repetir se contador diferente de zero
            RET                ;Retornar se contador igual a zero
```

Para testar a sub-rotina, use o mesmo ambiente do exercício anterior. Carregue na memória uma sequência de números cuja soma ultrapasse 16 *bits*. Por exemplo:

```
      .byte 100,100,100,100,100     ;Números a serem somados
```

ER 2.9. Vamos propor um exercício muito semelhante ao anterior. Escreva a sub-rotina SOMA que calcula o somatório dos números de 16 *bits* que estão armazenados a partir do endereço 0x2400. A quantidade de números a serem somados é indicada por R6. O resultado do somatório deve ser calculado com 32 *bits*, usando os registradores R10 (LSWord) e R11 (MSWord).

Solução:

Este exercício vai mostrar que com uma ALU de 16 *bits*, podemos fazer somas de 32, 64, ou qualquer outra quantidade de *bits*. Um papel importante será desempenhado pela *flag Carry*. Devemos imaginar que os registradores R11 e R10 estão concatenados. Fazemos a soma em R10 e depois, se tivermos C = 1, somamos 1 em R11.

Sub-rotina SOMA:
Recebe: R6 = quantidade de palavras de 16 *bits*, a partir de 0x2400, a serem somados;
Retorna: [R11 R10] = resultado do somatório em 32 *bits*.

Recursos a serem usados pela sub-rotina:
R5 = ponteiro, inicializado com 0x2400;
R6 = contador, decrementado a cada iteração e
[R11 R10] = somatório em 32 *bits*, inicializado com zero.

Listagem subrotina `SOMA`*, solução do ER 2.9*

```
;ER 2.9: Sub-rotina SOMA
SOMA:       CLR.W R10          ;Zerar somatório LSW
            CLR.W R11          ;Zerar somatório MSW
            MOV.W #DT,R5       ;Inicializar ponteiro
LOOP:       ADD.W @R5,R10      ;Somar uma palavra em R10
            ADC   R11          ;Soma o Carry em R11
            INCD  R5           ;Avançar ponteiro para próxima palavra
            DEC.W R6           ;Decrementar contador
            JNZ   LOOP         ;Repetir se contador diferente de zero
            RET                ;Retornar se contador igual a zero
```

Merece comentário a instrução `INCD R5`, esse incremento avança o ponteiro em 2 posições, já que estamos trabalhando com palavras de 16 *bits*. Essa instrução poderia ser eliminada se usássemos a instrução de soma `ADD.W @R5+,R10`. Como é uma operação em 16 *bits*, o processador avança R5 duas posições (R5 = R5 + 2).

Listagem de um ambiente para teste da subrotina `SOMA`

```
; Ambiente para testar sub-rotina SOMA
            MOV.W #5,R6        ;Somatório de 5 números
            CALL  #SOMA        ;Chamar a sub-rotina a ser testada
            JMP   $            ;Prender MSP
;
; Coloque aqui a sub-rotina SOMA
;
            .data              ;Indicar uso da memória de dados
DT:         .word 0xABCD,0xEEEE,0x1111,0xBA98, 0xAAAA ;Nr a somar
```

Resposta: R11 R10 = 0x3110E.

ER 2.10. Para este exercício e os próximos, vamos sugerir uma forma de se arrumar um vetor na memória, como mostrado na Figura 2.4. Note que para indicar o vetor, a única referência necessária é a do endereço de início, pois a primeira posição já indica seu tamanho. Os elementos podem ser *bytes*, palavras de 16 *bits* (W16) ou palavras de 32 *bits* (W32). Para o caso de letras, os *bytes* serão indicados pela Tabela ASCII.

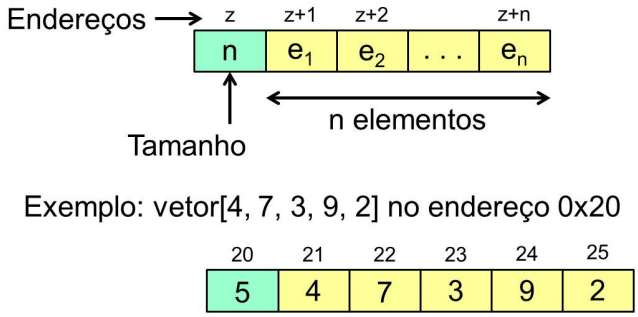


Figura 2.4. Sugestão de uma padronização para a formatação de um vetor na memória.

O exemplo abaixo indica como inicializar a memória de dados com o vetor da Figura 2.4. Note que este vetor é composto apenas por *bytes*.

```
            .data
; Declarar vetor com 5 elementos [4, 7, 3, 9, 2]
vetor:      .byte 0x05, 0x04, 0x07, 0x03, 0x09, 0x02
```

Pedido: Escreva a sub-rotina SUM8, que armazena em R10 o somatório dos elementos de um vetor composto por *bytes*, cujo endereço está em R5.

Solução:
Note que a única informação que a sub-rotina recebe é a de que o endereço do vetor está em R5. A sub-rotina deve usar R5 para ler o tamanho do vetor e depois ir incrementando-o para acessar sequencialmente os diversos elementos do vetor. A listagem abaixo apresenta a solução, que muito fácil de ser compreendida.

Sub-rotina SUM8:
Recebe: R5 = endereço de início do vetor;
Retorna: R10 = resultado do somatório em 8 *bits*.

Recursos a serem usados pela sub-rotina:
R5 = ponteiro, inicia apontando para o início do vetor;
R6 = contador, decrementado a cada iteração e

R10 = somatório em 8 *bits*, inicializado com zero.

Listagem subrotina `SUM8`*, solução do ER 2.10*

```
;ER 2.10: Sub-rotina SUM8
SUM8:       CLR.W R10          ;Zerar somatório
            MOV.B @R5+,R6      ;R6 = tamanho e incrementa R5
LOOP:       ADD.B @R5+,R10     ;Somar um byte R10
            DEC.B R6           ;Decrementar contador
            JNZ   LOOP         ;Repetir se contador diferente de zero
            RET                ;Retornar se contador igual a zero
```

Para testar a sub-rotina, é sugerido o ambiente abaixo. Note que o programador não precisa conhecer numericamente o endereço do vetor. Para isso, usamos o *label* `vetor1` que é uma constante cujo valor é 0x2400. Para ilustrar possibilidades, sugerimos 3 vetores diferentes (`vetor1`, `vetor1` e `vetor3`). Para calcular o somatório de qualquer um deles, basta alterar a instrução `MOV.W #vetor1,R5`.

Listagem de um ambiente para teste da subrotina `SUM8`

```
; Ambiente para testar sub-rotina SUM8
            MOV.W #vetor1,R5  ;Ponteiro = endereço do vetor
            CALL  #SUM8       ;Chamar a sub-rotina a ser testada
            JMP   $           ;Prender MSP
;
; Coloque aqui a sub-rotina SUM8
;
            .data ;Indicar uso da memória de dados
vetor1:     .byte 0x05, 0x04, 0x07, 0x03, 0x09, 0x02
vetor2:     .byte 7, 1, 2, 3, 4, 5, 6, 7
vetor3:     .byte 10, 1, 2, 3, 4, 5, 5, -4, -3, -2, -1
```

ER 2.11. Escreva a sub-rotina SUM16, que armazena em R10 o somatório dos elementos de um vetor de palavras de 16 *bits*, cujo endereço está em R5.

Solução:
A solução é muito semelhante à anterior. Compare as duas soluções.

Sub-rotina SUM16:
Recebe: R5 = endereço de início do vetor;
Retorna: R10 = resultado do somatório em 16 *bits*.

Recursos a serem usados pela sub-rotina:
R5 = ponteiro, inicia apontando para o início do vetor;
R6 = contador, decrementado a cada iteração e

R10 = somatório em 16 *bits*, inicializado com zero.

Listagem subrotina `SUM16`*, solução do ER 2.11*

```
;ER 2.11: Sub-rotina SUM16
SUM16:      CLR.W R10          ;Zerar somatório
            MOV.W @R5+,R6      ;R6 = tamanho e incrementa R5
LOOP:       ADD.W @R5+,R10     ;Somar uma palavra em R10
            DEC.W R6           ;Decrementar contador
            JNZ   LOOP         ;Repetir se contador diferente de zero
            RET                ;Retornar se contador igual a zero
```

Para testar a sub-rotina, é sugerido o ambiente abaixo, muito semelhante ao anterior. Note o uso da pseudo-instrução `.word` para indicar valores de 16 *bits*. Verifique o funcionamento da sub-rotina usando os outros vetores.

Listagem de um ambiente para teste da subrotina `SUM16`

```
; Ambiente para testar sub-rotina SUM16
            MOV.W #vetor1,R5  ;Ponteiro = endereço do vetor
            CALL  #SUM16            ;Chamar a sub-rotina a ser testada
            JMP   $           ;Prender MSP
;
; Coloque aqui a sub-rotina SUM16
;
            .data ;Indicar uso da memória de dados
vetor1:     .word 0x05, 0x04, 0x07, 0x03, 0x09, 0x02
vetor2:     .word 7, 1, 2, 3, 4, 5, 6, 7
vetor3:     .word 10, 1, 2, 3, 4, 5, 5, -4, -3, -2, -1
vetor4:     .word 6, 1234, 4567, 3, 5, -7654, 0
```

ER 2.12. Escreva a sub-rotina MAIOR8, que armazena em R10 o maior elemento de um vetor com números de 8 *bits* sem sinal, cujo endereço está em R5.

Solução:
Este exercício é interessante para mostrar como comparamos dois números. Para fazermos uma comparação precisamos de duas instruções. Primeiro, a instrução `CMP src,dst` que faz operação `dst - src`, mas não guarda o resultado, ou seja, os conteúdos de `src` e `dst` permanecem inalterados. Apenas as *flags* são alteradas. Em seguida usamos uma das instruções de decisão: `JEQ` (dst = src), `JNE` (dst ≠ src), `JHS` (dst ≥ src) ou `JLO` (dst < src).

O programa solução apresentado inicializa o contador R6 e coloca em R10 o menor elemento possível, que é o zero. Feito isso, vai em sequência comparando o conteúdo de

R10 com cada elemento do vetor. Se o elemento do vetor for maior que R10, esse elemento é copiado para R10 e passa a ser "maior provisório".

Sub-rotina MAIOR8:
Recebe: R5 = endereço de início do vetor;
Retorna: R10 = maior elemento do vetor.

Recursos a serem usados pela sub-rotina:
R5 = ponteiro, inicia apontando para o início do vetor;
R6 = contador, decrementado a cada iteração e
R10 = maior elemento provisório, inicializado com zero.

Listagem subrotina `MAIOR8`*, solução do ER 2.12*

```
;ER 2.12: Sub-rotina MAIOR
MAIOR8:     CLR.B R10          ;Maior elemento provisório = Zero
            MOV.B @R5+,R6      ;R6 = tamanho e incrementa R5
LOOP:       CMP.B @R5,R10      ;Comparar: R10 - @R5
            JHS   LB           ;Se R10 >= @R5, saltar para LB
            MOV.B @R5,R10      ;Senão, copiar novo maior para R10
LB:         INC.W R5           ;Avançar ponteiro
            DEC.B R6           ;Decrementar contador
            JNZ   LOOP         ;Repetir se contador diferente de zero
            RET                ;Retornar se contador igual a zero
```

O ambiente para teste é o mesmo sugerido para o ER 2.10. Neste ambiente estão definidos 3 vetores. Verifique o funcionamento da rotina com cada um dos três. Ao fazer uso do `vetor3`, a resposta não será a esperada. Por que? No próximo exercício veremos a razão dessa resposta inesperada.

ER 2.13. Escreva a sub-rotina MENOR8, que armazena em R10 o menor elemento de um vetor com números de 8 *bits* com sinal, cujo endereço está em R5.

Solução:
O pedido agora envolve números com sinal. O leitor é aconselhado a rever representação de números negativos usando Complemento a Dois, disponível no Apêndice A. Em 8 *bits* os números com sinal vão desde -128, -127, ..., 0, +1, +2, ..., +127.

Tudo fica muito semelhante à solução anterior, mas a decisão precisa ser feita com as instruções para comparação de números com sinal: `JGE` (dst ≥ src) ou `JL` (dst < src). Como procuramos pelo menor, inicializamos R10 com o maior número de 8 *bits* com sinal, que é o 127. Na solução abaixo estão ressaltadas em negrito as duas instruções que foram alteradas em relação à solução anterior. Para testar a sub-rotina, sugerimos o ambiente do ER 2.10, em especial o ensaio com o `vetor3`.

Sub-rotina MENOR:
Recebe: R5 = endereço de início do vetor;
Retorna: R10 = menor elemento do vetor.

Recursos a serem usados pela sub-rotina:
R5 = ponteiro, inicia apontando para o início do vetor;
R6 = contador, decrementado a cada iteração e
R10 = maior elemento provisório, inicializado com 127 (maior positivo em 8 *bits*).

Listagem subrotina `MENOR8`*, solução do ER 2.13*

```
;ER 2.13: Sub-rotina MENOR8
MENOR8:     MOV    #127,R10     ;Menor elemento provisório = 127
            MOV.B  @R5+,R6      ;R6 = tamanho e incrementa R5
LOOP:       CMP.B  @R5,R10      ;Comparar: R10 - @R5
            JL     LB           ;Se R10 < @R5, saltar para LB
            MOV.B  @R5,R10      ;Copiar novo maior para R10
LB:         INC.W  R5           ;Avançar ponteiro
            DEC.B  R6           ;Decrementar contador
            JNZ    LOOP         ;Repetir se contador diferente de zero
            RET                 ;Retornar se contador igual a zero
```

Observação: O ensaio com o `vetor3` deve resultar em R10 = -4, que corresponde ao 0xFC.

ER 2.14. Escreva a sub-rotina INV8 que inverte a ordem dos elementos de um vetor formado por números de 8 *bits*, cujo endereço está em R5. Por exemplo, se receber o vetor [4, 7, 3, 9, 2], deve retornar o vetor [2, 9, 3, 7, 4], como indicado na figura abaixo.

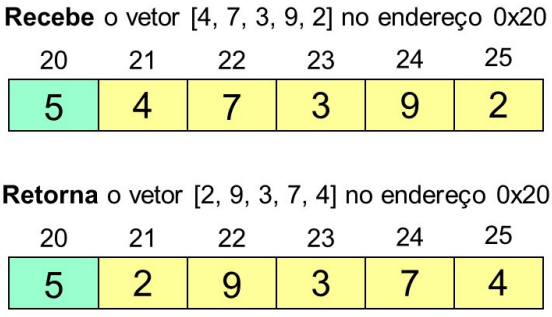


Figura 2.5. Ilustração da operação de inversão de um vetor.

Solução:
O leitor deve estar considerando que existem dois casos, em função do número de elementos do vetor ser par ou ímpar. É possível diversas soluções. Vamos adotar a que

usa dois ponteiros. Um ponteiro, chamado de crescente, é inicializado com o endereço do primeiro elemento e o outro, o decrescente, com o endereço do último elemento. A cada iteração, trocamos os elementos apontados por esses dois ponteiros, incrementamos o ponteiro crescente e decrementamos o ponteiro decrescente. Repetimos enquanto o primeiro ponteiro (crescente) for menor que o segundo (decrescente).

Sub-rotina INV8:
Recebe: R5 = endereço de início do vetor;
Retorna: Vetor invertido.

Recursos a serem usados pela sub-rotina:
R5 = ponteiro crescente;
R6 = ponteiro decrescente;
R10 e R11 = auxiliares para fazer a troca.

Listagem subrotina `INV8`*, solução do ER 2.14*

```
;ER 2.14: Sub-rotina INV8
INV8:       MOV.B @R5,R6      ;R6 = tamanho do vetor
            ADD.W R5,R6       ;R6 = endereço do último elemento
            INC.W R5          ;R5 = endereço do primeiro elemento
LOOP:       CMP.W R6,R5       ;Comparar os dois ponteiros (R5 - R6)
            JLO   SEGUE       ;Se R5 < R6, segue adiante
            RET               ;Senão (R5 >= R6), retorna
SEGUE:      MOV.B @R5,R10     ;Copiar em R10 elemento da parte inicial
            MOV.B @R6,R11     ;Copiar em R11 elemento da parte final
            MOV.B R11,0(R5)   ;Escrever R11 na parte inicial
            MOV.B R10,0(R6)   ;Escrever R10 na parte final
            INC.W R5          ;Incrementar ponteiro crescente
            DEC.W R6          ;Incrementar ponteiro decrescente
            JMP   LOOP        ;Repetir laço
```

A solução apresentada merece dois comentários. O primeiro é sobre a inicialização do ponteiro decrescente (R6). O leitor poderia ser tentado a usar a sequência listada abaixo, que elimina uma das instruções `INC.W R5`, que na listagem acima estão marcadas em negrito. A ideia é usar a instrução que faz o autoincremento. O problema é que a instrução `ADD.B @R5+,R6` por ser uma soma de 8 *bits*, vai zerar o MSB de R6. Note que R6 contém endereço e, por isso, precisa de 16 bits.

```
;Sub-rotina INV8
INV8:       MOV.W R5,R6       ;R5 = R6 = endereço do vetor
            ADD.B @R5+,R6     ;R6 = endereço último elemento e incr. R5
LOOP:       CMP.W R6,R5       ;Comparar os dois ponteiros (R5 - R6)
```

O segundo comentário, como o leitor deve ter percebido, é sobre o fato de a sub-rotina fazer a comparação entre os dois ponteiros antes de fazer a primeira transferência. Isto foi colocado para prever o caso de um vetor com um único elemento.

O ambiente para teste da sub-rotina é o mesmo sugerido para o ER 2.10.

ER 2.15. Escreva a sub-rotina FIBn que gera os "n" primeiros números da Sequência de Fibonacci e os armazena na memória RAM a partir da posição indicada por R5. O valor de "n" é indicado por R6. O leitor se lembra do programa apresentado logo no início deste capítulo?

Solução:
A Sequência de Fibonacci é uma sequência de números onde cada novo elemento é formado pela soma dos dois anteriores ($a_n = a_{n-1} + a_{n-2}$). Os dois primeiros elementos são "0" e "1". Por exemplo, os 10 primeiros elementos desta sequência são: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.

Sub-rotina FIBn:
Recebe: R6 = quantidade de elementos da Sequência da Fibonacci;
R5 = endereço para armazenar a sequência de Fibonacci.
Retorna: Sequência armazenada a partir do endereço indicado por R5.

Recursos a serem usados pela sub-rotina:
R5 = ponteiro indicando onde se deve armazenar os elementos e
R6 = contador, decrementado a cada iteração.

A primeira ideia do leitor, provavelmente, será a de ficar recuando o ponteiro R5 para "alcançar" os dois elementos anteriores e depois e avançando para armazenar o resultado da soma. Outra ideia provável é a de usar 3 ponteiros: dois ponteiros para os dois elementos anteriores e um terceiro ponteiro para o local de armazenamento. Essas ideias geram soluções viáveis, mas grandes e lentas. É preciso ainda lembrar que o endereçamento indireto (@Rn) não pode ser usado como destino. Apresentaremos uma solução simples que faz uso do Endereçamento Indexado.

Listagem subrotina `FIBn`*, solução do ER 2.15*

```
;ER 2.15: Sub-rotina FIBn
FIBn: MOV.B #0,0(R5)     ;Carregar primeiro elemento da seq.
      INC.W R5           ;Avançar ponteiro
      MOV.B #1,0(R5)     ;Carregar segundo elemento da seq.
      INC.W R5           ;Avançar ponteiro
      SUB.B #2,R6        ;Já temos 2 elementos
LOOP: MOV.B -2(R5),0(R5);Copiar antepenúltimo
      ADD.B -1(R5),0(R5);Somar com o último
```

```
      INC.W R5          ;Avançar ponteiro
      DEC.B R6          ;Decrementar contador
      JNZ   LOOP        ;Se diferente de zero, repetir
      RET               ;Senão, retornar
```

Para testar a sub-rotina, é sugerido um ambiente muito simples, mostrado abaixo.

Listagem de um ambiente para teste da subrotina `FIBn`

```
; Ambiente para testar sub-rotina FIBn
            MOV.W #0x2410,R5  ;Ponteiro = endereço do vetor
            MOV   #10,R6      ;Testar com 10 elementos
            CALL  #FIBn       ;Chamar a sub-rotina a ser testada
            JMP   $           ;Prender MSP
;
; Coloque aqui a sub-rotina FIBn
;
```

ER 2.16. Escreva a sub-rotina FIB8 que grave a partir do endereço apontado por R5 os números da Sequência de Fibonacci enquanto eles forem passíveis de representação em 8 *bits*.

Solução:
Vamos gerar a sequência enquanto os elementos "couberem" dentro da representação de 8 *bits*. O truque agora é determinar o momento de parar a sequência. Isso pode ser feito com a consulta do *Carry* após cada soma. Quando obtivermos *Carry* = 1, é porque o limite de 8 *bits* foi ultrapassado. A solução fica muito parecida com a do exercício anterior.

Sub-rotina FIB8:
Recebe: R5 = ponteiro para armazenar os elementos;
Retorna: Sequência armazenada a partir do endereço indicado.

Recursos a serem usados pela sub-rotina:
R5 = ponteiro indicando onde se deve armazenar os elementos;

Listagem subrotina `FIB8`*, solução do ER 2.16*

```
;ER 2.16: Sub-rotina FIB8
FIB8: MOV.B #0,0(R5)     ;Carregar primeiro elemento da seq.
      INC.W R5           ;Avançar ponteiro
      MOV.B #1,0(R5)     ;Carregar segundo elemento da seq.
LOOP: INC.W R5           ;Avançar ponteiro
      MOV.B -2(R5),0(R5);Copiar antepenúltimo
      ADD.B -1(R5),0(R5);Somar com o último
      JNC   LOOP         ;Se Cy = 0, repetir
```

```
      RET               ;Senão, retornar
```

A solução apresentada tem um pequeno defeito: ela escreve uma posição a mais. A soma é armazenada para depois se testar o *Carry*. Sugira uma forma de eliminar esta deficiência. O ambiente de teste da rotina é muito semelhante ao do exercício anterior.

Note que, em relação à solução do exercício anterior, foi preciso alterar a posição do *label* `LOOP` para que a operação de incremento do ponteiro `INC.W R5` não destruísse o *Carry* gerado pela soma. A solução abaixo está errada pois apresenta este defeito

```
...
LOOP: MOV.B -2(R5),0(R5);Copiar antepenúltimo
      ADD.B -1(R5),0(R5);Somar com o último
      INC.W R5           ;Avançar ponteiro (destrói o Cy da soma anterior)
      JNC   LOOP         ;Testar o Cy (testamos o Cy da INC.W R5)
```

Resposta: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233.

ER 2.17. Vamos agora trabalhar com a Tabela ASCII, apresentada no Apêndice B. Essa tabela designa códigos numéricos para os símbolos de impressão (ou de controle de impressão). Por exemplo, a letra “A” é representada pelo número 0x41 e a letra “B” pelo número 0x42. Este problema, fazendo uso da Tabela ASCII, pede a sub-rotina NIB_ASC que retorna o código ASCII corresponde ao valor hexadecimal do *nibble* (4 *bits*) que está em R5. Veja os exemplos abaixo.

Tabela 2.7. Exemplos de operação da sub-rotina NIB_ASC

Recebe (R5)		**Retorna (R5)**	
Decimal	Hexadecimal	Hexadecimal	ASCII
0	0x0	0x30	0
1	0x1	0x31	1
...	...	...	...
9	0x9	0x39	9
10	0xA	0x41	A
11	0xB	0x42	B
...	...	...	...
15	0xF	0x46	F

Solução:
Em palavras simples, a sub-rotina recebe um *nibble* em R5 e retorna no próprio R5 o código ASCII correspondente ao valor em hexadecimal deste *nibble*. Um ponto

interessante da Tabela ASCII é que os números de 0 até 9 e as letras de A até Z estão em sequência. Porém entre o número 9 e a letra A existem 7 códigos não usados e é isso que torna o exercício interessante. Ao observar a tabela acima vemos a solução seria: se R5 for menor ou igual a 9, somamos 0x30, senão somamos 0x37.

Note que o pedido indica que o *nibble* está armazenado em R5, ou seja, 0 ≤ R5 ≤ 15. Isto significa que os 4 *bits* mais à esquerda devem ser iguais a zero.

Sub-rotina NIB_ASC:
Recebe: R5 = *nibble*, no formato binário 0000 xxxx, onde xxxx representa o *nibble*.
Retorna: R5 =ASCII(*nibble*)

Recursos a serem usados pela sub-rotina:
R5 = operação para conversão

Listagem subrotina `NIB_ASC`*, solução do ER 2.17*

```
;ER 2.17: Sub-rotina NIB_ASC1 (Versão 1)
NIB_ASC:
      CMP.B #10,R5      ;Comparar (R5 - 10)
      JHS   LB          ;Se R5 >= 10, saltar
      ADD.B #0X30,R5    ;Se R5 < 10 => somar 0x30
      RET               ;Retornar
LB:   ADD.B #0X37,R5    ;Se R5 >= 10 => somar 0x37
      RET               ;Retornar
```

Apresentamos a seguir uma segunda versão do programa, que faz economia de uma instrução. Pode parecer pouco para o problema presente, mas a intenção é mostrar abordagens eficientes para a programação *assembly*. Logo no início somamos 0x30 e depois testamos o resultado para ver se devemos somar 7 (caso o resultado seja maior igual a 0x3A).

Listagem da segunda versão da subrotina `NIB_ASC`*, solução do ER 2.17*

```
;ER 2.17: Sub-rotina NIB_ASC2 (Versão 2)
NIB_ASC:
      ADD.B #0X30,R5    ;Somar 0x30
      CMP.B #0x3A,R5    ;Comparar (R5 - 0x3A)
      JLO   FIM         ;Se R5 < 0x3A => fim
      ADD.B #7,R5       ;Se R5 >= 0x3A, somar 7
FIM:  RET               ;Retornar
```

Vamos agora aproveitar uma terceira solução para apresentar o uso de tabelas. Em *assembly*, uma tabela nada mais é que uma "arrumação" de dados na memória. Note que na presente solução (ver abaixo), todos os resultados possíveis foram organizados na memória, a partir do endereço TAB_ASC. Assim, a sub-rotina se resume na consulta à tabela, feita com a instrução `MOV.B TAB_ASC(R5),R5`. Essa instrução acessa a posição de memória cujo endereço é dado pela soma de R5 (o *nibble* recebido) com o endereço indicado por TAB_ASC. O programa ficou rápido e aparentemente pequeno, apenas duas linhas. Na verdade, ele não é tão pequeno, pois para calcular seu tamanho devemos considerar também as 16 posições ocupados pela tabela. Note que a tabela está na área de códigos, já que não fizemos o uso da pseudo-instrução ".`data`".

Listagem da terceira versão da subrotina `NIB_ASC`*, solução do ER 2.17*

```
;ER 2.17: Sub-rotina NIB_ASC3 (Versão 3)
NIB_ASC:
            MOV.B TAB_ASC(R5),R5    ;Consultar a tabela
            RET                     ;Retornar
            .data
TAB_ASC:    .byte 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37
            .byte 0x38, 0x39, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46
```

Para testar a sub-rotina, é sugerido um ambiente muito simples, mostrado abaixo para o caso do *nibble* ser igual a 5.

Listagem de um ambiente para teste da subrotina `NIB_ASC`

```
; Ambiente para testar sub-rotina NIB_ASC
            MOV   #5,R5       ;R5 = um nibble, trocar para novos testes
            CALL  #NIB_ASC    ;Chamar a sub-rotina a ser testada
            JMP   $           ;Prender MSP
;
; Coloque aqui a sub-rotina NIB_ASC
;
```

ER 2.18. Escreva a sub-rotina *BYTE*_ASC que recebe um *byte* em R5 e retorna em R6 e em R5 os códigos ASCII correspondentes ao **nibble** menos significativo (LSN) e mais significativo (MSN), respectivamente. A tabela a seguir apresenta alguns exemplos. Use a sub-rotina elaborada no exercício anterior.

Tabela 2.8. Exemplos da operação da sub-rotina *BYTE*_ASC

Recebe (R5)		**Retorna**	
Decimal	Hexadecimal	R5	R6

59	0x3B	0x33	0x42
197	0xC5	0x43	0x35
0	0	0x30	0x30
255	0xFF	0x46	0x46

Solução:
A solução é muito simples. Basta, separar os *nibbles* e usar a sub-rotina NIB_ASC.

Sub-rotina *BYTE*_ASC:
Recebe: R5 = *byte*.
Retorna: R5 =ASCII(MSN) e R6 =ASCII(LSN)

Recursos a serem usados pela sub-rotina:
R5 e R6 = operação para conversão,
Uma posição da pilha e a sub-rotina NIB_ASC.

Listagem da subrotina `BYTE_ASC`*, solução do ER 2.18*

```
;ER 2.18: Sub-rotina BYTE_ASC
BYTE_ASC:
      PUSH.B      R5          ;Guardar na pilha uma cópia de R5
      AND.B       #0xF,R5     ;Isolar o LSN-Nibble menos signifivativo
      CALL        #NIB_ASC    ;Obter ASCII(LSN)
      MOV.B       R5,R6       ;Transferir ASCII(LSN) para R6
      POP.B       R5          ;Recuperar R5 original
      RRA.B       R5          ;1) Uma rotação para a direita
      RRA.B       R5          ;2) Uma rotação para a direita
      RRA.B       R5          ;3) Uma rotação para a direita
      RRA.B       R5          ;4) Uma rotação para a direita
      AND.B       #0xF,R5     ; Isolar o MSN-Nibble mais signifivativo
      CALL        #NIB_ASC    ;Obter ASCII(MSN)
      RET                     ;Retornar
```

A solução merece alguns comentários. Para poder usar a sub-rotina NIB_ASC, precisamos separar o *nibble* menos significativo (LSN) e depois o *nibble* mais significativo (MSN), que serão sempre alinhados pela direita. Os demais *bits* devem ser zero. Para isolar o LSB, basta fazer um AND do valor original com 0xF (0000 1111 em binário). Para isolar o MSB, é preciso rodar R5 para direita 4 vezes e só depois fazer a operação AND. É importante ressaltar o uso da pilha para armazenar, provisoriamente, uma cópia de R5, já que a primeira operação AND destrói do MSN.

Para testar a sub-rotina, é sugerido um ambiente muito simples, mostrado abaixo para o caso do *byte* ser igual a 0xB5. Teste com diversos valores.

Listagem de um ambiente para teste da subrotina `BYTE_ASC`

```
; Ambiente para testar sub-rotina BYTE_ASC
            MOV   #0xB5,R5    ;R5 = byte
            CALL  #BYTE_ASC   ;Chamar a sub-rotina a ser testada
            JMP   $           ;Prender MSP
;
; Coloque aqui a sub-rotina BYTE_ASC
;
; Coloque aqui uma das sub-rotinas NIB_ASC
;
```

Resposta: R5 = 0x42 e R6 = 0x35.

ER 2.19. Escreva a sub-rotina CONTA_1 que retorna em R6 a quantidade de *bits* iguais a 1 da palavra de 16 *bits* que está em R5. O conteúdo original de R5 e do *Carry* devem ser preservados.

Solução:
A solução é muito simples. Basta rodar R5 e testar o *Carry*. Como a rotação é circular, o valor de R5 e do *Carry* serão preservados. Teremos de executar 16 rotações. Ao final, dessas rotações, será necessário mais uma, para acertar o valor original de R5. Como R5 e *Carry* trabalham juntos, é como se fossem 17 *bits*.

Sub-rotina CONTA_1:
Recebe: R5 = palavra de 16 *bits*.
Retorna: R6 = quantidade de *bits* iguais a 1.

Recursos a serem usados pela sub-rotina:
R5 = valor original
R6 = quantidade de *bits* iguais a 1 da palavra que estava em R5
R7 = contador
Uma posição da pilha

A listagem abaixo apresenta a solução mais intuitiva, porém com erro. O leitor está convidado a identificar e explicar este erro. A segunda listagem apresenta a rotina já corrigida. Teste as rotinas com os parâmetros sugeridos na tabela abaixo.

Tabela 2.9. Exemplos de operação da sub-rotina CONTA_1

Recebe	**Retorna**	
R5	R5	R6
0	0	0

0x5555	0x5555	8
0xAAAA	0xAAAA	8
0xFFFF	0xFFFF	16

Listagem (com erro) da subrotina `CONTA_1`*, solução do ER 2.19*

```
;ER 2.19: Sub-rotina CONTA_1 (versão com erro)
CONTA_1:
      CLR.W       R6          ;Zerar contador de 1's
      MOV.W       #16,R7      ;Serão 16 testes, um para cada bit
LOOP: RRC.W       R5          ;Rodar R5 para direita, com Carry
      JNC         LB1         ;Se Cy=0, salta
      INC.W       R6          ;Se Cy=1, incrementa R6
LB1:  DEC.W       R7          ;Decrementar contador de testes
      JNZ         LOOP        ;Repetir laço se contador <> 0
      RRC.W       R5          ;Mais uma rotação para "acertar" R5
      RET                     ;Retornar
```

Listagem da subrotina `CONTA_1`*, solução do ER 2.19*

```
;ER 2.19: Sub-rotina CONTA_1 (versão correta)
CONTA_1:
      CLR.W       R6          ;Zerar contador de 1's
      MOV.W       #16,R7      ;Serão 16 testes, um para cada bit
      PUSH.W      R2
LOOP: POP.W       R2
      RRC.W       R5          ;Rodar R5 para direita, com Carry
      PUSH        R2
      JNC         LB1         ;Se Cy=0, salta
      INC.W       R6          ;Se Cy=1, incrementa R6
LB1:  DEC.W       R7          ;Decrementar contador de testes
      JNZ         LOOP        ;Repetir laço se contador <> 0
      POP.W       R2          ;
      RRC.W       R5          ;Mais uma rotação para "acertar" R5
      RET                     ;Retornar
```

O problema da primeira listagem, a que está com erro, é que a contagem será feita corretamente, porém o conteúdo de R5 e do *carry* serão destruídos (ao contrário do que pede o problema). A cada rotação, um *bit* de R5 vai para o *carry*. Porém o decremento do contador (`DEC.W R7`) destrói o valor deste *carry*. Na solução seguinte, o valor do registrador de *Status* (R2) onde está o *carry* é armazenado na pilha e recuperado antes de se fazer a rotação. Com isso, é preservada a integridade de R5 e do *carry*.

Para testar a sub-rotina, é sugerido um ambiente muito simples, mostrado abaixo para o caso do *byte* ser igual a 0xB5. Teste com diversos valores.

Listagem de um ambiente para teste da subrotina `CONTA_1`

```
; Ambiente para testar sub-rotina CONTA_1
            MOV   #0x5555,R5  ;R5 = byte  ;0x5555 tem oito 1s
            CALL  #CONTA_1    ;Chamar a sub-rotina a ser testada
            JMP   $           ;Prender MSP
;
; Coloque aqui a sub-rotina CONTA_1
;
```

ER 2.20. Escreva a sub-rotina ALEAT que armazena a partir do endereço indicado por R5 uma certa quantidade de palavras de 16 *bits* pseudo-aleatórias. A quantidade total de palavras é indicada por R6. Use R7 para construir um LFSR (*Linear Feedback Shift Register*) segundo o polinômio gerador abaixo e com o valor inicial de 0x5555:

$$X^{16} + X^{14} + X^{13} + X^{11} + 1$$

Observação: O método aqui indicado é eficiente para a geração de uma sequência de *bits* pseudo-aleatórios e não de palavras pseudo-aleatórias. Entretanto, a finalidade é didática e por isso será tolerado o emprego equivocado. (Está certo isso?)

Solução:
É de certa forma óbvio que usando-se exclusivamente técnicas de programação não é possível criar um verdadeiro gerador de números aleatórios. Por isso, tais técnicas são denominadas de pseudo-aleatórias. Um gerador muito comum é o construído com Registrador de Deslocamento com Realimentação Linear (LFSR - *Linear Feedback Shift Register*). Mais informações podem ser conseguidas no *link* abaixo, onde também se encontra o polinômio gerador usado neste exercício.
https://en.wikipedia.org/wiki/Linear-feedback_shift_register

O polinômio gerador é expresso em ambiente booleano, assim, a soma é substituída pela operação OU-Exclusiva e os expoentes indicam a posição do *bit*. Sem entrar em considerações que não são objetivo do presente texto, o gerador pseudo-aleatório para o polinômio acima pode ser descrito pela Figura 2.6. Essa figura simplifica bastante o problema: tudo o que precisamos fazer é rodar para a esquerda um registrador de 16 *bits* e fazer seu *bit* mais à direita (b0) igual ao resultado do XOR (b15, b13, b12, b10). A cada rotação "coletamos" um *bit* na saída. Assim, após 16 rotações teremos uma nova palavra pseudo-aleatória.

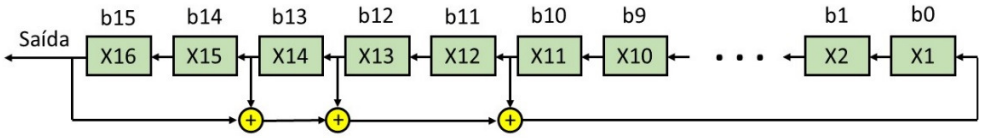


Figura 2.6. Ilustração de um LFSR para gerar números aleatórios, considerando o polinômio gerador acima citado. As somas "+" indicadas são realizadas com a operação OU-Exclusivo.

Os *bits* gerados passam a se repetir depois de 65.535 (2^{16} - 1) rotações. Por isso, é importante comentar que estamos cometendo um erro intencional. O LFSR é um gerador de *bits* aleatórios e não de palavras aleatórias. Ao fazermos agrupamentos de 16 *bits*, geraremos apenas 4.096 palavras dentre as 65.535 possíveis.

Sub-rotina ALEAT:
Recebe: R6 = quantidade de palavra de 16 *bits*;
R5 = ponteiro para armazenar as palavras geradas.
Retorna: números pseudo-aleatórios a partir do endereço apontado por R5

Recursos a serem usados pela sub-rotina:
R5 = ponteiro para armazenar as palavras pseudo-aleatórias
R6 = contador de palavras
R7 = registrador LFSR
R8 = contador para as 16 operações LFSR
R9 = realizar bit15 XOR bit13 XOR bit12 XOR bit10

Listagem da subrotina `ALEAT`*, solução do ER 2.20*

```
;ER 2.20:
;Constantes para auxiliar no entendimento
BIT15       .equ  0x8000      ;1000 0000 0000 0000
BIT13       .equ  0x2000      ;0010 0000 0000 0000
BIT12       .equ  0x1000      ;0001 0000 0000 0000
BIT10       .equ  0x400       ;0000 0100 0000 0000
;
; Sub-rotina para gerar números aleatórios e armazenar na memória
;
ALEAT:      MOV   R7,0(R5)    ;Conteúdo de R7 já é o primeiro número
            INCD  R5          ;Avançar ponteiro
            DEC   R6          ;Decrementar contador
            JNZ   LB1         ;Se contador=0, retornar
            RET
            CALL  #GER_PAL    ;Senão, gerar nove número
            JMP   ALEAT
;
; Sub-rotina para fazer 16 operações LFSR
```

```
;
GER_PAL:    MOV   #16,R8
LB1:        CALL  #XOR_BITS
            RLC   R7
            DEC   R8
            JNZ   LB1
            RET
;
; Sub-rotina para fazer o XOR com os bits indicados pelo polinômio
; Cy = b15 XOR b13 XOR b12 XOR b10
; Resultado é retornado no carry
; Usa R9 para fazer a conta
;
XOR_BITS:   MOV   #0,R9       ;1)
            BIT   #BIT15,R6   ;2) bit 15 = 0?
            JZ    XB1         ;3) Sim => R9=0
            MOV   #1,R9       ;4) Não => R9=1
XB1:        BIT   #BIT13,R6   ;5) bit 13 = 0?
            JZ    XB2         ;6) Sim => vai para o próximo
            XOR   #1,R9       ;7) Não => R9 = R9 XOR 1
XB2:        BIT   #BIT12,R6   ;8) bit 12 = 0?
            JZ    XB3         ;9) Sim => vai para o próximo
            XOR   #1,R9       ;10)Não => R9 = R9 XOR 1
XB3:        BIT   #BIT10,R6   ;11) bit 10 = 0?
            JZ    XB4         ;12) Sim => vai para o próximo
            XOR   #1,R9       ;13) Não => R9 = R9 XOR 1
XB4:        RRC   R9          ;14) Carry recebe resultado
            RET               ;15) Retornar
```

O leitor deve ter notado que a listagem acima não faz uso dos sufixos ".W" e ".B". No caso de omissão, o montador considera a opção de 16 *bits*. É interessante verificar se realmente todas as instruções precisavam ser de 16 *bits*. Enquanto não se ganha prática, é recomendado o uso explícito dos sufixos ".W" e ".B".

Essa sub-rotina pede por uma série de explicações. Iniciamos pelo nível mais baixo que é a sub-rotina XOR_BITS. Não é simples fazer a operação indicada na Figura 2.6, porque ela envolve diversos *bits* do mesmo registrador. A solução foi usar o registrador R9 como auxiliar. Na verdade, estamos usando apenas o *bit* 0 deste registrador. A sequência das operações está apresentada abaixo.

- linhas 1, 2, 3 e 4 → R9 = bit15 de R6
- linhas 5, 6 e 7 → R9 = R9 XOR bit13 de R6;
- linhas 8, 9 e 10 → R9 = R9 XOR bit12 de R6;
- linhas 11, 12 e 13 → R9 = R9 XOR bit10 de R6;
- linha 14 roda R9 de forma a que o *bit* mais à direita vá para o *carry*.

Para facilitar o entendimento da rotina, logo no início do programa foram definidas 4 constantes, uma para cada *bit* a ser operado (Elas já não deveriam estar disponíveis?). Elas foram declaradas usando `.equ` (*equate*) que é uma diretiva para definir constantes. O valor dessas constantes é obvio. Assim, ficou fácil usar a instrução de teste de *bits* `bit src,dst`. Essa instrução faz a operação `src AND dst`, mas não armazena o resultado, apenas atualiza as *flags*. Assim, a instrução de teste de bit, `bit #BIT15,R6` retorna zero se o bit15 de R6 for zero e retorna um caso contrário.

Pode não parecer muito simples, já que a Figura 2.6 apresenta a saída de *bits* na extremidade esquerda, mas a palavra que procuramos é o próprio conteúdo de R7, após 16 operações. É fácil de ver que a primeira palavra será o valor inicial que é 0x5555. Depois de 16 operações, formamos uma nova palavra em R7. E assim se segue.

Para testar a sub-rotina, é sugerido o ambiente abaixo.

Listagem de um ambiente para teste da subrotina `ALEAT`

```
; Ambiente para testar sub-rotina ALEAT
            MOV.W #0x2400,R5  ;Armazenar a partir de 0x2400
            MOV.W #8,R6       ;Gerar 8 palavras
            MOV.W #0x5555,R7  ;Inicializar LFSR
            CALL  #ALEAT      ;Chamar a sub-rotina a ser testada
            JMP   $           ;Prender MSP
;
; Coloque aqui a sub-rotina ALEAT
;
```

Resposta: 0x5555, 0x52EF, 0xF59C, 0x788A, 0x2ADD, 0x5FCF, 0xFA64, 0xF9E4.

ER 2.21. Em diversos casos, a fila circular é uma estrutura de dados muito útil. Ela é composta por um *buffer* de tamanho "n" e de três sub-rotinas:
POE → coloca um dado na fila circular
TIRA → retira um dado da fila circular e
INIC → inicializa os ponteiros da fila circular.
Proponha as rotinas para criar uma fila circular de tamanho TAM.

Solução:
A fila circular é uma estrutura muito útil quando se trabalha com sistemas embarcados. Ela trabalha como um "amortecedor" e permite que se receba dados e se retire dados de forma não sincronizada. Um exemplo de uso é no teclado do PC. A cada ação no teclado, a BIOS coloca os códigos correspondentes às teclas acionadas numa fila circular e o Windows, à medida que tem tempo, retira esses códigos da fila. Como ilustrado na Figura 2.7, a fila circular simplesmente consiste na alocação de um *buffer* em memória e no uso de dois ponteiros:

- **Pin** → ponteiro para colocar um elemento na fila e
- **Pout** → ponteiro para retirar um elemento da fila.

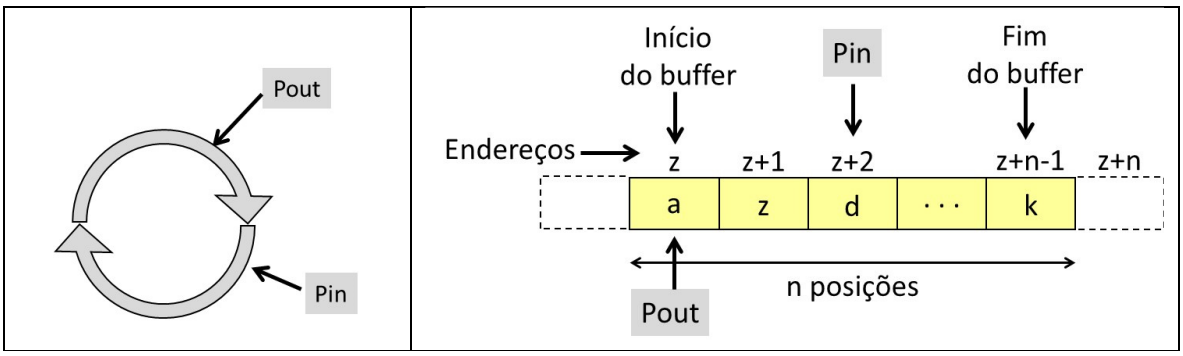


Figura 2.7. Ilustração de uma Fila Circular.

A fila circular é construída com um vetor de tamanho "n" e o controle dos ponteiros (Pin e Pout) é feito de forma a se ter a ilusão de que as duas extremidades do vetor estão unidas, como ilustrado no lado esquerdo da figura acima.

É claro que um ponteiro nunca pode ultrapassar o outro. Na figura abaixo temos a caracterização de dois estados importantes, que são o de fila vazia e o de fila cheia. Isto dispensa a necessidade de um contador para verificar se a fila está cheia. Por outro lado, se sacrifica uma posição.
Fila vazia: (Pin = Pout + 1) Pin está uma posição à direita de Pout e
Fila cheia: (Pin = Pout) Pin igual a Pout.

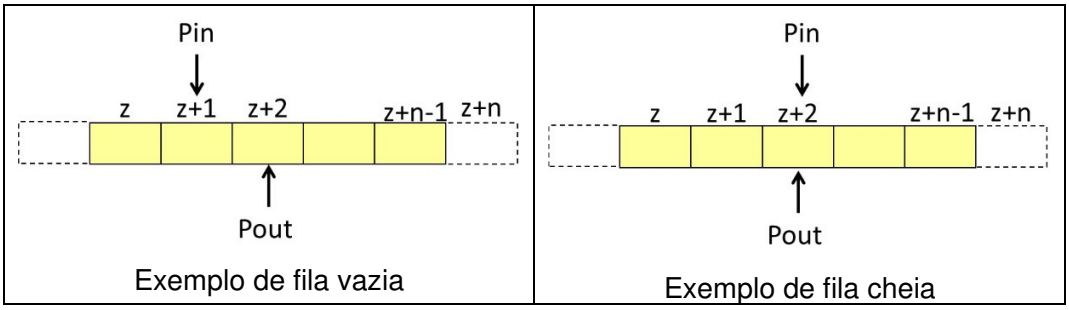


Figura 2.8. Posição relativa dos ponteiros caracteriza os estados de fila vazia e fila cheia.

A regra para operação dos ponteiros e da fila é simples:

- **Põe** → se Pin ≠ Pout, então escrever o dado na posição Pin e incrementar Pin;
- **Tira** → se Pout ≠ Pin+1, então retirar o dado da fila e incrementar Pout.

A cada incremento ou comparação é preciso verificar se o ponteiro ultrapassou o final do *buffer*, situação em que ele deve voltar para o início, para assim tornar a fila circular. Vamos propor as seguintes sub-rotinas.

INIC → Inicializar as PIN e POUT
Recebe: Nada
Retorna: POUT = início da fila e PIN = POUT+1 (fila vazia)

POE → Colocar um *byte* na fila circular
Recebe: R5 = *byte* a ser colocado na fila
Retorna: Carry = 1 se teve sucesso
Carry = 0 se não teve sucesso (fila cheia)
Usa: R6 para escrever na memória (R6 = PIN)

TIRA → Retirar um *byte* da fila circular
Recebe: Nada
Retorna: R5 = *byte* e *carry* = 1 se teve sucesso
carry = 0 se não teve sucesso (fila vazia)
Usa: R6 para ler da memória (R6 = POUT+1)

A Figura 2.9 apresenta o fluxograma das sub-rotinas POE e TIRA. Note que a cada incremento de um ponteiro (PIN ou POUT) é necessário verificar se ele chegou ao final da fila. Caso isso aconteça, é necessário voltá-lo para o início da fila.

Logo a seguir está a listagem das sub-rotinas e um trecho de programa para testar seu funcionamento. O leitor deverá fazer uso do CCS para ver a memória de dados e assim poder constatar o funcionamento dessa sub-rotinas.

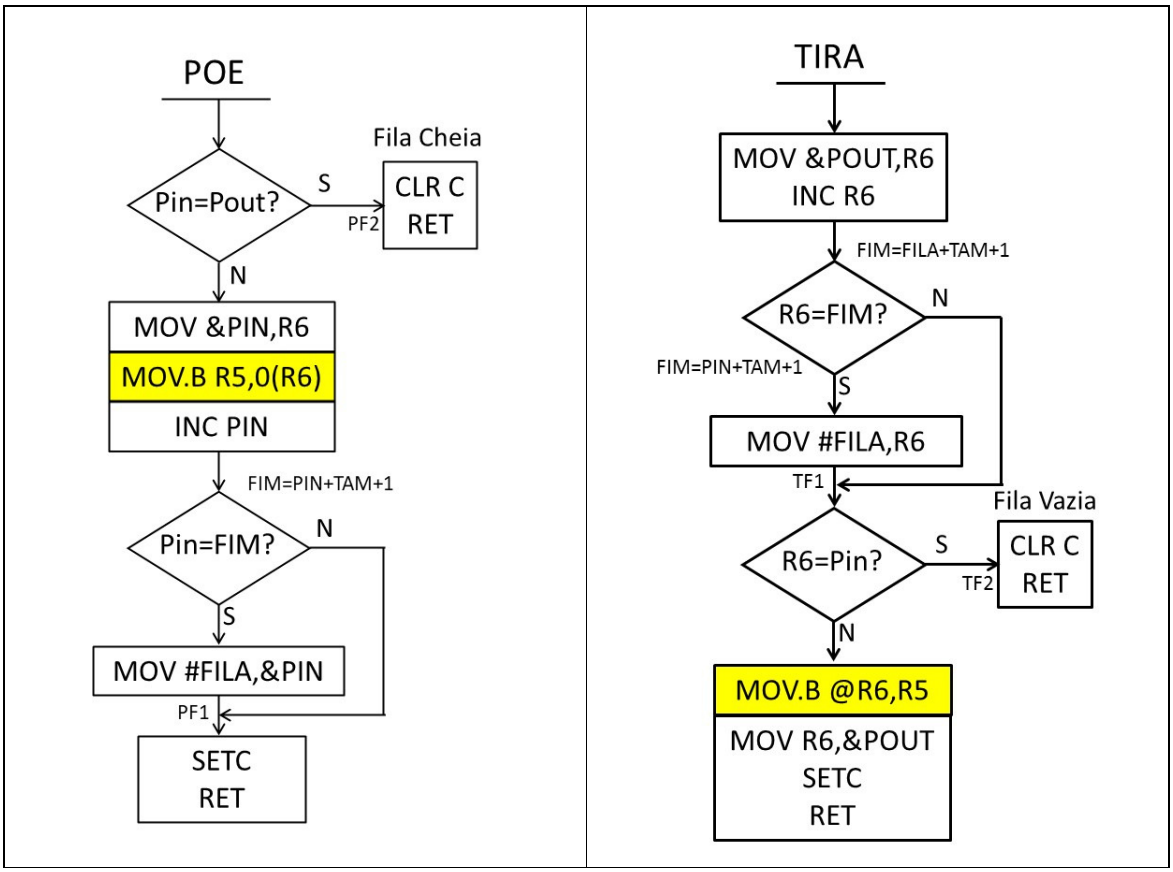


Figura 2.9. Fluxograma das sub-rotinas POE e TIRA.

Listagem da solução do ER 2.21

```
;ER 2.21: Proposta de sub-rotinas para uma Fila Circular
;
; Constantes para a Fila Circular
TAM   .equ  20          ;Tamanho da Fila Circular
PIN   .equ  0x2400      ;Posição para guardar valor de Pin
POUT  .equ  PIN+2       ;Posição para guardar valor de Pout
FILA  .equ  POUT+2      ;Inicio da fila circular
;
; Rotina para teste da fila circular
FC:   CALL  #INIC       ;Inicializar ponteiros
      CLR.B R5          ;R5 = 0, gerador de dados
      ;
      ;Laço para encher a fila
FIL1: INC.B R5          ;Gerar novo dado
      CALL  #POE        ;Colocar na fila
      JC    FIL1        ;Se teve sucesso (Cy=1), repetir
      ;
      ;Laço para esvaziar a fila
```

```
FIL2: CALL  #TIRA       ;Retirar um dado
      JC    FIL2        ;Se teve sucesso (Cy=1), repetir
      JMP   $           ;Travar execução
;
; POE: Por na fila o byte que está em R5, usa R6
; Retorna Carry = 1 --> teve sucesso
; Retorna Carry = 0 --> falhou (fila cheia)
POE:  CMP   &PIN,&POUT  ;PIN=POUT --> fila cheia
      JZ    PF2         ;PIN=POUT --> Carry = Z = 0, retornar
      MOV   &PIN,R6     ;R6 = PIN, para acesso
      MOV.B R5,0(R6)    ;Colocar byte na fila
      INC   &PIN        ;Avançar ponteiro
      CMP   #FILA+TAM+1,&PIN ;PIN ultrapassou final da fila?
      JNZ   PF1         ;Não, seguir
      MOV   #FILA,&PIN  ;Sim, voltar PIN para o início da fila
PF1:  SETC              ;Cy = 1 --> sucesso
      RET               ;Retornar
PF2:  CLRC              ;Cy = 0 --> Fila cheia
      RET               ;Retornar
;
; TIRA: Retira um byte da fila e o coloca em R5, usa R6
; Retorna Carry = 1 --> teve sucesso
; Retorna Carry = 0 --> falhou (fila vazia)
TIRA: MOV   &POUT,R6    ;R6 = POUT
      INC   R6          ;R6= POUT+1
      CMP   #FILA+TAM+1,R6 ;R6 ultrapassou fim da fila?
      JNZ   TF1         ;Não: seguir
      MOV   #FILA,R6    ;Sim, R6 = início da fila
TF1:  CMP   &PIN,R6     ;POUT+1 = PIN? --> fila vazia
      JZ    TF2         ;Sim, Carry=0 e Z=0, saltar para retornar
      MOV.B @R6,R5      ;Não, ler byte da fila para R5
      MOV   R6,&POUT    ;POUT = POUT + 1 (ficou guardado em R6)
      SETC              ;Cy = 1  --> Teve sucesso
      RET               ;Retornar
TF2:  CLRC              ;Cy = 0 --> Indicar fila vazia
      RET               ;Retornar
;
; INIC: Inicializar ponteiros
INIC: MOV   #FILA+1,&PIN ;Começar com PIN=1  (posição 1 da fila)
      MOV   #FILA,&POUT  ;Começar com POUT=0 (posição 0 da fila)
            RET
```

ER 2.22. Este é um exercício que tem uma longa solução. A intenção é mostrar como se estrutura um problema mais sofisticado. Vamos ainda abordar o uso de multiplicador em *hardware* disponível nessa arquitetura.

Pedido: Use o multiplicador em hardware do MSP430 para escrever a sub-rotina MULT_AB, que faz a multiplicação de duas matrizes: C = A * B. As matrizes são compostas por palavras de 16 *bits* com sinal e armazenadas na memória de acordo com a sugestão proposta na Figura 2.10. Por simplicidade, vamos supor que os elementos da matriz resultado "caibam" dentro da representação de 16 *bits* com sinal. O Apêndice P apresenta uma descrição simplificada do multiplicador em *hardware* disponível no MSP430.

A sub-rotina MULTM recebe:

R5 → endereço da matriz A;
R6 → endereço da matriz B;
R7 → endereço da matriz C, onde será armazenado o resultado.

A sub-rotina MULTM retorna:

Carry = 0 se não foi possível fazer a multiplicação das matrizes ou
Carry = 1 se foi possível fazer a multiplicação e o produto armazenado no endereço indicado por R7.

Cada matriz está arrumada na memória segundo a forma indicada na figura abaixo. Note que, de cada matriz, apenas se precisa saber o primeiro endereço.

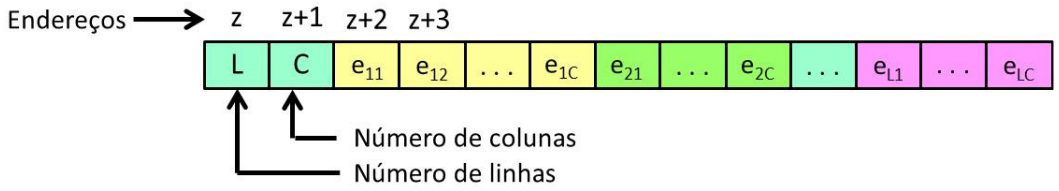


Figura 2.10. Proposta para armazenagem de uma matriz.

Solução:

Este problema é um excelente exemplo de como se deve escrever um programa longo. Em tais casos, o correto é dividi-lo em diversas sub-rotinas pequenas, muito bem documentadas. Pode ser que a quantidade de registradores disponíveis não seja suficiente, por isso, vamos ainda reservar posições em memória, ou seja, vamos criar uma série de variáveis, todas de 16 *bits*. A razão dessas variáveis é óbvia em função do problema a ser resolvido. Note que as variáveis foram criadas a partir do endereço arbitrário 0x2500. É decisão do programador a escolha do local onde ele deve alocar as variáveis.

Lista das variáveis:

ENDA → endereço do primeiro elemento da Matriz A
NLA → Número de linhas da Matriz A
NCA → Número de colunas da Matriz A

ENDB → endereço do primeiro elemento da Matriz B
NLB → Número de linhas da Matriz B
NCB → Número de colunas da Matriz B

ENDC → endereço do primeiro elemento da Matriz C
NLC → Número de linhas da Matriz C
NCC → Número de colunas da Matriz C

LIN → Contador de linhas da Matriz A
COL → Contador de colunas da Matriz B
PROD → Produto de uma linha por uma coluna

Lista das sub-rotinas

MULT_AB → Multiplica duas matrizes
Recebe: R5 = ponteiro para a Matriz A
R6 = ponteiro para a Matriz B;
R7 = ponteiro para a Matriz C;
Retorna: produto C = A x B

INIC → Inicializar as variáveis a serem usadas pela sub-rotina
Recebe: R5 = endereço da matriz A
R6 = endereço da matriz B
R7 = endereço da matriz C
Retorna: variáveis NLA, NCA, ENDA, NLB, NCB, ENDB, NLC, NCC, ENDC.
Comentário: Esta sub-rotina manipula os ponteiros de forma a inicializar as variáveis que vão facilitar o algoritmo de multiplicação.

MUL_LIN_COL → Multiplica uma linha de A por uma coluna de B
Recebe: LIN = número da linha
COL = número da coluna
Retorna: PROD = produto da linha LIN pela coluna COL.
Comentário: Essa sub-rotina retorna em PROD o produto da linha LIN pela coluna COL. Note que LIN e COL são números e não endereços. Por exemplo, para multiplicar a linha "m" pela coluna "n", fazemos LIN = "m" e COL = "n". Para descobrir o endereço de início da linha "m", usamos a sub-rotina CALC_LIN e para descobrir o endereço de início da coluna "n", usamos a sub-rotina CALC_COL.

CALC_LIN → Calcula endereço de início de uma linha da Matriz A
Recebe: R12 = número da linha (contador)
Retorna: R10 = endereço de início da linha
Comentário: Essa sub-rotina retorna em R10 o endereço de início da linha cujo número foi recebido em R12. Para poder multiplicar a linha "n", é preciso saber o endereço de início desta linha. Como o endereço do primeiro elemento da matriz está em ENDA, basta somar "n-1" vezes o número de colunas (n-1 passos). R10 = ENDA + (R12-1) x NCA. É preciso lembrar que trabalhamos em 16 *bits*, assim, é devemos dobrar o tamanho dos passos. Portanto, o correto é R10 = ENDA + 2 x (R12-1) x NCA. Por esse motivo a instrução de soma `ADD &NCA,R10` foi repetida.

CALC_COL → Calcula endereço do início de uma coluna da Matriz B
Recebe: R12 = número da coluna (contador)
Retorna: R11 = endereço de início da coluna
Comentário: Essa sub-rotina retorna em R11 o endereço de início da coluna da matriz B cujo número foi recebido em R12. Para poder multiplicar a coluna "n", é preciso saber o endereço de início desta coluna. Como o endereço do primeiro elemento da matriz está em ENDB, basta avançar "n-1" palavras de 16 *bits* (n-1 passos). R11 = ENDB + 2 x (R12-1). É preciso lembrar que trabalhamos em 16 *bits*, assim, é devemos dobrar o tamanho dos passos, daí o uso da instrução `INCD R11`.

MLC → Multiplica uma linha de A por uma coluna de B, usando os endereços
Recebe: R10 = endereço de início da linha
R11 = endereço de início da coluna
Retorna: PROD = produto
Usa: R9 = NCA = NLB = contador de iterações
Comentários: Esta sub-rotina já recebe R10 apontando para o primeiro elemento da linha e R11 apontando para o primeiro elemento da coluna. Usamos o recurso de Multiplica-e-Acumula com sinal (MACS) do multiplicador em *hardware* para fazer o produto (ver Apêndice P). Por isso, logo no início da sub-rotina zeramos o acumulador do multiplicador (RES0). Como o pedido simplifica a solução afirmando que os resultados sempre "cabem" em 16 *bits*, não usamos as demais porções do registrador de resultados (RES1, RES2 e RES3). Se os elementos das matrizes são palavras de 16 *bits*, os ponteiros devem avançar 2 *bytes*.

ARMZ → Armazenar o produto de linha de A por coluna de B na matriz C
Recebe: ENDC = endereço do primeiro elemento da Matriz C
LIN = número da linha
COL = número da coluna
PROD = produto de linha por coluna
Retorna: Matriz C(LIN,COL) = PROD
Comentário: Após o produto da linha LIN, pela coluna COL, é preciso armazenar o resultado na matriz C, na posição LIN x COL. Para isso, R8 recebe o endereço do primeiro elemento da matriz C, executa (LIN-1) passos de tamanho COL (passos duplos,

pois se usam elementos de 16 *bits*) e, para terminar, avança ainda (COL-1) passos duplos. No final das contas, temos R8 apontando para o endereço do elemento C(LIN,COL). Bastão então fazer a escrita do resultado que está em PROD.

Listagem da solução do ER 2.22

```
;-------------------------------------------------------------------
; ER 2.22: MULT_AB --> multiplica duas matrizes, C = A x B
; R5 = endereço da matriz A
; R6 = endereço da matriz B
; R7 = endereço da matriz C (matriz produto)
;-------------------------------------------------------------------
;
; Labels (constantes) para acesso à memória
; Cada endereço funcionará como uma variável
;
ENDA  .equ  0x2500      ;Endereço do primeiro elemento Matriz A
NLA   .equ  0x2502      ;Número de linhas  de A
NCA   .equ  0x2504      ;Número de colunas de A

ENDB  .equ  0x2506      ;Endereço do primeiro elemento Matriz B
NLB   .equ  0x2508      ;Número de linhas  de B
NCB   .equ  0x250A      ;Número de colunas de B

ENDC  .equ  0x250C      ;Endereço do primeiro elemento Matriz C
NLC   .equ  0x250E      ;Número de linhas  de C
NCC   .equ  0x2510      ;Número de colunas de C

LIN   .equ  0x2512      ;Contador de linhas A
COL   .equ  0x2514      ;Contador de colunas B
PROD  .equ  0x2516      ;Produto de uma linha por uma coluna
;
;-------------------------------------------------------------------
; MULT_AB --> Multiplica duas matrizes
; Recebe: R5 = ponteiro para a Matriz A
;         R6 = ponteiro para a Matriz B;
;         R7 = ponteiro para a Matriz C;
; Retorna: produto C = A x B
;-------------------------------------------------------------------
MULT_AB:    CALL  #INIC       ;Inicializar variáveis
            CMP   &NCA,&NLB   ;NLA = NCB? (Nr linhas = Nr colunas?)
            JZ    OK1         ;Se NLA = NCB OK
            CLRC              ;Se NLA <> NCB, carry=0 e retornar
            RET
;
OK1:        MOV   &NLA,-4(R7)       ;Nr linhas C = Nr Linhas A
            MOV   &NCB,-2(R7)       ;Nr colunas C = Nr colunas B
            MOV   #1,&LIN           ;Iniciar pela linha 1
LP1:        MOV   #1,&COL           ;e coluna 1
```

```
LP2:        CALL  #MUL_LIN_COL      ;Multiplicar LIN x COL
            INC   &COL
            CMP   &COL,&NCB
            JHS   LP2
            INC   &LIN
            CMP   &LIN,&NLA
            JHS   LP1
            SETC
            RET

;-------------------------------------------------------------------
; MUL_LIN_COL --> Multiplica uma linha de A por uma coluna de B
; Recebe: LIN = número da linha
;         COL = número da coluna
; Retorna: PROD = produto
;-------------------------------------------------------------------
MUL_LIN_COL:
            MOV   &LIN,R12    ;R12 = número da linha
            CALL  #CALC_LIN   ;R10=endereço de início da linha de A
            MOV   &COL,R12    ;R12 = número da coluna
            CALL  #CALC_COL   ;R11=endereço de início da coluna de B
            CALL  #MLC        ;Linha A x Coluna de B, usando endereços
            CALL  #ARMZ       ;Armazenar na matriz C
            RET

;-------------------------------------------------------------------
; CALC_LIN --> Calcula endereço de início de uma linha da Matriz A
; Recebe:  R12 = número da linha (contador)
; Retorna: R10 = endereço de início da linha
;-------------------------------------------------------------------
CALC_LIN:   MOV   &ENDA,R10   ;R10 = endereço de início da matriz A
LIN1:       DEC   R12         ;Decrementa contador
            JNZ   LIN2        ;Se contador <> 0, continuar
            RET
LIN2:       ADD   &NCA,R10    ;Passo igual ao tamanho da coluna de A
            ADD   &NCA,R10    ;Repete, matriz com elementos de 16 bits
            JMP   LIN1        ;Repete o laço

;-------------------------------------------------------------------
; CALC_COL --> Calcula endereço do início de uma coluna da Matriz B
; Recebe:  R12 = número da coluna
; Retorna: R11 = endereço de início da coluna
;-------------------------------------------------------------------
CALC_COL:   MOV   &ENDB,R11   ;R11 = endereço de início da matriz B
COL1:       DEC   R12         ;Decrementar contador
            JNZ   COL2        ;Se contador <> 0, continuar
            RET
COL2:       INCD  R11         ;Incremento duplo em R11 (16 bits)
            JMP   COL1        ;Repete o laço
```

```
;-------------------------------------------------------------------
; MLC --> Multiplica uma linha de A por uma coluna de B
; Recebe: R10 = endereço de início da linha
;         R11 = endereço de início da coluna
; Retorna: PROD = produto
; Usa: R9 = NCA = NLB = contador de iterações
;-------------------------------------------------------------------
MLC:        MOV   #0,RES0     ;Zerar acumulador dos resultados
            MOV   &NCA,R9     ;R9 = Qtd de iterações
MLC1:       MOV   @R10,&MACS  ;MACS = elemento da linha
            MOV   @R11,&OP2   ;OP2 = elemento da coluna
            INCD  R10         ;avançar ponteiro da linha
            ADD   &NCB,R11    ;avançar ponteiro da coluna
            ADD   &NCB,R11    ;avançar ponteiro da coluna
            DEC   R9          ;decrementar contador
            JNZ   MLC1        ;se contador <> 0, repetir
            MOV   RES0,&PROD  ;PROD =  resultado de lin x col
            RET               ;se contador = 0, retornar

;-------------------------------------------------------------------
; ARMZ --> Guardar produto de linha de A por coluna de B na matriz C
; Recebe: ENDC = endereço do primeiro elemento da Matriz C
;         LIN = número da linha
;         COL = número da coluna
;         PROD = produto de linha por coluna
; Retorna: Matriz C(LIN,COL) = PROD
;-------------------------------------------------------------------
ARMZ:       MOV   &ENDC,R8    ;R8 = Endereço primeiro elemento de C
            MOV   &LIN,R9     ;R9 = Número da linha do resultado
            MOV   &COL,R10    ;R10= Número da coluna do resultado
ARMZ1:      DEC   R9          ;Decrementa contador de linhas
            JZ    ARMZ2       ;Se contador = 0, segue adiante
            ADD   &NCC,R8     ;Avançar ponteiro (Nr de colunas)
            ADD   &NCC,R8     ;Avançar ponteiro (Nr de colunas)
            JMP   ARMZ1       ;Repetir o laço
ARMZ2:      DEC   R10         ;Decrementa contador de colunas
            JZ    ARMZ3       ;Se contador = 0, segue adiante
            INCD  R8          ;Avançar ponteiro (2 x Nr de linhas)
            JMP   ARMZ2       ;Repetir o laço
ARMZ3:      MOV   &PROD,0(R8) ;Armazer o resultado C(LIN,COL)=PROD
            RET

;-------------------------------------------------------------------
; INIC
; Inicializar as variáveis para realizar a multiplicação
; Inicializa: NLA, NCA, ENDA, NLB, NCB, ENDB, NLC, NCC, ENDC,
;-------------------------------------------------------------------
INIC:       MOV   @R5,&NLA          ;NLA = número de linhas de A
            INCD  R5                ;Avançar ponteiro
            MOV   @R5,&NCA          ;NCA = número de colunas de A
```

```
            INCD  R5                ;Avançar ponteiro
            MOV   R5,&ENDA          ;ENDA = end. primeiro elemento de A

            MOV   @R6,&NLB          ;NLB = número de linhas de B
            INCD  R6                ;Avançar ponteiro
            MOV   @R6,&NCB          ;NCB =  número de colunas de A
            INCD  R6                ;Avançar ponteiro
            MOV   R6,&ENDB          ;ENDB = end. primeiro elemento de B

            MOV   &NLA,&NLC         ;NLC = NLA (Nr de linhas de A)
            MOV   &NCB,&NCC         ;NCC = NCB (Nr de colunas de B)
            INCD  R7                ;Avançar ponteiro
            INCD  R7                ;Avançar ponteiro
            MOV   R7,&ENDC          ;ENDC = end. primeiro elemento de C

            RET
```

A maneira como foi reservado o espaço para as variáveis (ENDA, NLA, ...) é clara, porém não muito inteligente. Caso venha a ser necessário mudar o endereço dessas variáveis, será preciso editar todas as linhas. A sugestão abaixo é muito mais eficiente. Note que definimos a constante VARS (de variáveis) e toda a alocação toma esta constante como referência. No futuro, uma redefinição de VARS, faz a realocação de todas as variáveis. Aliás, esta é uma prática muito importante em qualquer programa. Use constantes no início do programa para definir parâmetros que são passíveis de mudanças futuras.

```
; Labels (constantes) para acesso à memória
; Cada endereço funcionará como uma variável
;
VARS  .equ  0x2500
ENDA  .equ  VARS+0      ;Endereço do primeiro elemento Matriz A
NLA   .equ  VARS+2      ;Número de linhas  de A
...
PROD  .equ  VARS+22     ;Produto de uma linha por uma coluna
```

As duas linhas apresentadas abaixo merecem comentário especial. Note que a primeira coisa que a sub-rotina MUL_AB faz é chamar a sub-rotina INIC que inicializa todas as variáveis. Essa sub-rotina incrementa R5, R6 e R7, de forma que eles fiquem apontando para o primeiro elemento de cada matriz. Assim, para escrever o número de linhas da matriz C (que é igual ao número de linhas da matriz A), precisamos retroceder R7 em 4 posições e para escrever o número de colunas da matriz C (que é igual ao número de colunas da matriz B), precisamos retroceder R7 em 2 posições.

```
OK1:        MOV   &NLA,-4(R7)       ;Nr linhas C = Nr Linhas A
            MOV   &NCB,-2(R7)       ;Nr colunas C = Nr colunas B
```

Para verificar o funcionamento da subtrotina, sugerimos o seguinte programa. Note que se o programa ficar parado do laço `ERRO: JNC $` é porque não foi possível multiplicar as duas matrizes.

Listagem de um ambiente para teste da subrotina `MULT_AB`

```
; Ambiente para testar sub-rotina MULT_AB

            MOV   #MATA,R5
            MOV   #MATB,R6
            MOV   #MATC,R7
            CALL  #MULT_AB
ERRO:       JNC   $                 ;ERRO caso pare aqui
OK:         JC    $                 ;SUCESSO caso pare aqui
            NOP;
;
; Coloque aqui a sub-rotina MULT_AB
;
      .data
; Declarar Matrizes
MATA: .word 3, 4, -1, 2, 3, -4, 5, 6, 7, 8, 9, 10, 11, 12
MATB: .word 4, 5, 1, -2, 3, 4, 5, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 6, 7, 8,
      .word 9, 10
MATC: .word 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
      .word 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
```

Os dados acima sugeridos correspondem ao produto das duas matrizes abaixo.

$$\begin{bmatrix} -1 & 2 & 3 & -4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \times \begin{bmatrix} 1 & -2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 6 & 7 & 8 & 9 & 10 \end{bmatrix} = \begin{bmatrix} -5 & -1 & -5 & -5 & -5 \\ 101 & 107 & 153 & 179 & 205 \\ 157 & 163 & 241 & 283 & 325 \end{bmatrix}$$

2.5. Exercícios Propostos

A seguir são propostos diversos exercícios. Alguns deles são meras ampliações de algum dos exercícios propostos, enquanto que outros são completamente novos. É fortemente recomendado que o leitor, após escrever seu programa, o simule usando o Code Composer Studio (CCS) e a placa **MSP430 F5529 launch pad**. Sempre separe sua solução (em geral, uma sub-rotina) do ambiente usado para a verificação. Em muitos casos, será necessário corrigir os erros descobertos durante a fase de verificação.

EP 2.1. Sempre operando com *bytes* (.B), escreva uma sub-rotina que armazene em R7 a soma dos *bytes* presentes em R5 e R6 (R7 = R5 + R6). Teste seu programa com R5 = 100 e R6 = 100 e depois com R5 = 150 e R6 = 150.

EP 2.2. Sempre operando com palavras de 16 *bits* (.W), escreva uma sub-rotina que armazene em R7 a soma das palavras de 16 *bits* presentes em R5 e R6 (R7 = R5 + R6). Teste seu programa com R5 = 20.000 e R6 = 20.000 e depois com R5 = 30.000 e R6 = 30.000.

EP 2.3. Considerando que R7 é um registrador de 16 *bits*, escreva uma sub-rotina que armazene em R7 a soma dos *bytes* presentes em R5 e R6 (R7 = R5 + R6). Teste seu programa com R5 = 100 e R6 = 100 e depois com R5 = 150 e R6 = 150. Note que a resposta da soma deve ser em 16 *bits*.

EP 2.4. Considerando a concatenação de R8(MSB) com R7(LSB), escreva uma sub-rotina que armazene em R8 e R7 a soma das palavras de 16 *bits* presentes em R5 e R6 (R8 R7 = R5 + R6). Teste seu programa com R5 = 20.000 e R6 = 20.000 e depois com R5 = 40.000 e R6 = 40.000.

EP 2.5. Sempre operando com *bytes* (.B), escreva uma sub-rotina que armazene em R7 o resultado de R6 subtraído de R5 (R7 = R5 - R6). Teste seu programa com R5 = 6 e R6 = 5 e depois com R5 = 5 e R6 = 6. Observe a representação de números negativos em Complemento a 2. Em caso de dúvidas, consulte o Apêndice A.

EP 2.6. Sempre operando com palavras de 16 *bits* (.W), escreva uma sub-rotina que armazene em R7 o resultado de R6 subtraído de R5 (R7 = R5 - R6). Teste seu programa com R5 = 6 e R6 = 5 e depois com R5 = 5 e R6 = 6. Qual a diferença para com os resultados apresentados no problema anterior?

EP 2.7. Este problema envolve operações aritméticas em 8 e 16 *bits*. Escreva uma sub-rotina que resolva a expressão: R7 = R7 + (R6 – R5). Dica: veja a instrução de extensão do sinal. Conteúdo dos registradores:

- R5 = 200 (8 *bits*);
- R6 = 100 (8 *bits*) e
- R7 = 1.100 (16 *bits*)

EP 2.8. Este problema envolve o entendimento das *flags* presentes no registrador de *status* (R2): V, N, Z e C. Sem executar o programa, tente preencher a tabela abaixo e depois a confira usando o CCS. Para os testes usaremos uma instrução que armazena em R7 o resultado de uma operação envolvendo R5 e R7:

Tabela 2.10. Teste para o entendimento dos flags do Registrador de Status

R5	R7	Operação	Resultado (16 *bits*)		V	N	Z	C
			Decimal	Hexa				
5	7	R7 = R7 + R5						

5	7	R7 = R7 - R5						
5	7	R7 = R5 - R7						
-5	7	R7 = R7 + R5						
-5	7	R7 = R7 - R5						
-5	7	R7 = R5 - R7						
5	-7	R7 = R7 + R5						
5	-7	R7 = R7 - R5						
5	-7	R7 = R5 - R7						
-5	-7	R7 = R5 - R7						
-5	-7	R7 = R5 - R7						
-5	-7	R7 = R5 - R7						

EP 2.9. Este problema envolve o entendimento da instrução de comparação (`CMP`), com sinal e sem sinal, e dos saltos que podem ser realizados logo em seguida. Sempre considerando que acabou de ser executada a instrução `CMP R5,R7`, tente preencher a tabela abaixo indicando com "1" quando o salto acontece e com "0" quando ele não acontece e também as *flags* de *status* V, N, Z e C. Depois, escreva um programa com os valores sugeridos na tabela e use o CCS para verificar suas respostas.

Tabela 2.11. Saltos condicionais e flags com a instrução `CMP R5,R7`

Regs		**Saltos Condicionais**						**Flags**			
R5	**R7**	**JEQ**	**JNE**	**JHS**	**JLO**	**JGE**	**JL**	**V**	**N**	**Z**	**C**
5	7										
7	5										
7	7										
-5	-7										
-7	-5										
5	7										
-5	7										
1-	32767										

EP 2.10. Uma das formas mais simples de se fazer multiplicação é através de somas sucessivas. Escreva a sub-rotina MULT_SS, para multiplicação em 16 *bits*, que calcula o seguinte produto: [R8 | R7] = R6 x R5. Considerando, de forma simplificada, que as

instruções de salto consumam 2 µs e as demais 1 µs, calcule o tempo máximo consumido pela sua sub-rotina.

EP 2.11. Uma forma mais eficiente para se fazer multiplicação é através de somas e descolamentos, como fazemos em decimal usando papel e lápis e mostrado na Figura 2.11. Escreva a sub-rotina MULT_SD, para multiplicação em 16 *bits*, que calcula o seguinte produto: [R8 | R7] = R6 x R5. Veja o exemplo abaixo do produto de 23 x 13 = 299. Dica: Pense em qual registrador deslocar. Que tal o registrador de resultado?

```
                        1   0   1   1   1       R6 = 23
                        X   1   1   0   1       R7 = 13
                    ---------------------------------
                        1   0   1   1   1
                    0   0   0   0   0
                1   0   1   1   1
            1   0   1   1   1
        ---------------------------------
    1   0   0   1   0   1   0   1   1       R8|R9 = 299
```

Figura 2.11. Proposta multiplicação em binário.

Considerando, de forma simplificada, que as instruções de salto consumam 2 µs e as demais 1 µs, calcule o tempo máximo consumido pela sua sub-rotina. Compare com a solução anterior.

EP 2.12. O MSP430 possui um multiplicador em *hardware*. Isto significa que ele é capaz de fazer produtos de forma muito rápida. Veja o Apêndice P. Escreva a sub-rotina MULT_HW, para multiplicação em 16 *bits*, que calcula o seguinte produto: [R8 | R7] = R6 x R5. Quanto tempo consome sua sub-rotina, considerando MCLK = 1 MHz? (Consulte o Apêndice P ou o manual do MSP430).

EP 2.13. A divisão é também um bom problema em *Assembly*. Vamos tratar apenas da divisão inteira, resultando em quociente e resto. Uma das formas mais simples de se fazer divisão é através de subtrações sucessivas. Escreva a sub-rotina DIV_SS, para divisão em 16 *bits* de R8 / R7. O quociente deve ser armazenado em R5 e o resto em R6. Não se esqueça do caso da divisão por zero. Considerando, de forma simplificada, que as instruções de salto consumam 2 µs e as demais 1 µs, calcule o tempo máximo consumido pela sua sub-rotina.

EP 2.14. Uma forma mais eficiente para se fazer divisão é através de subtrações e descolamentos, como fazemos em decimal usando papel e lápis e mostrado na Figura 2.12, onde se divide 23 por 4. Escreva a sub-rotina DIV_SD, para a divisão em 16 *bits*, que calcula R8 / R7. O quociente deve ser armazenado em R5 e o resto em R6. Não se esqueça do caso da divisão por zero. Considerando, de forma simplificada, que as instruções de salto consumam 2 µs e as demais 1 µs, calcule o tempo máximo consumido pela sua sub-rotina.

```
 1   0   1   1   1  |1   0   0          R8=23 / R7=4
 1   0   0          |    1   0   1      R5 = 5
-----------          ----------
         1   1   1
         1   0   0
         ---------
             1   1                     R6 = 3
```

Figura 2.12. Proposta para divisão em binário.

EP 2.15. Para este exercício e os próximos, vamos usar a representação de vetores sugerida no ER 2.10. Escreva a sub-rotina MAIOR16, que armazena em R10 o maior elemento de um vetor com números de 16 *bits* sem sinal, cujo endereço está em R5.

EP 2.16. Escreva a sub-rotina MAIOR16S, que armazena em R10 o maior elemento de um vetor com números de 16 *bits* com sinal, cujo endereço está em R5.

EP 2.17. Escreva a sub-rotina MENOR32, que armazena em R11(MSB) e R10(LSB) o maior elemento de um vetor com números de 32 *bits* sem sinal, cujo endereço está em R5.

EP 2.18. Escreva a sub-rotina MENOR32S, que armazena em R11(MSB) e R10(LSB) o maior elemento de um vetor com números de 32 *bits* com sinal, cujo endereço está em R5.

EP 2.19. Escreva a sub-rotina ORD8, que trabalha com números de 8 *bits* sem sinal e ordena de forma crescente duas posições de memória: a apontada por R5 e a seguinte. Veja os exemplos na figura abaixo. No Exemplo 1, as duas posições foram trocadas. No Exemplo 2, a ordem já estava correta e nada foi feito. Note que em ambos os casos, o ponteiro (R5) terminou apontando para a segunda posição.

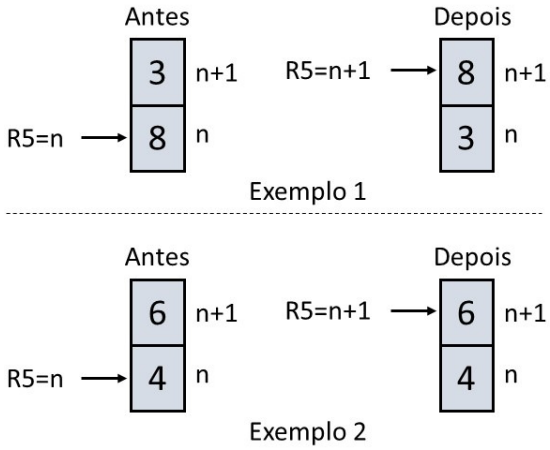


Figura 2.13. Exemplos de emprego da sub-rotina ORD8.

EP 2.20. Escreva a sub-rotina BOLHA8, que ordena de forma crescente o vetor com números de 8 *bits* sem sinal, cujo endereço está em R5. Use a sub-rotina ORD8, preparada no exercício anterior. Este tipo de ordenamento recebe o nome de "Método da Bolha".

EP 2.21. Escreva a sub-rotina BOLHA16S, que ordena de forma crescente o vetor com números de 16 *bits* com sinal, cujo endereço está em R5. Para auxiliar sua solução, escreva a sub-rotina ORD16S, análoga à do EP 2.19.

EP 2.22. Escreva a sub-rotina CRTL_FV que verifica se o vetor cujo endereço está em R5 contém o vetor cujo endereço está em R6. Retorna *carry* em 1 caso positivo e *carry* em 0 caso negativo. Esta sub-rotina lembra o comando de busca (*find*) "CTRL + F" usado em vários programas. Veja exemplos na figura abaixo.

Vetor 1	R5 →	6	3	8	4	7	2	9
Vetor 2	R6 →	4	7	2				

Exemplo 1: o Vetor 1 contém a sequência contida no Vetor 2, retornar *carry* = 1.

Vetor 1	R5 →	6	3	8	4	7	2	9
Vetor 2	R6 →	2	3	4				

Exemplo 2: o Vetor 1 não contém a sequência contida no Vetor 2, retornar *carry* = 0.

Vetor 1	R5 →	14	P	A	U	L	O		A	N	A	J	O	S	E	P
Vetor 2	R6 →	3	A	N	A											

Exemplo 3: O nome ANA está presente no Vetor 1, retornar *carry* = 1.

Figura 2.14. Exemplos de emprego da sub-rotina CTRL_FV.

EP 2.23. Quando se programa em linguagem C, as cadeias de caracteres (*strings*) são armazenadas em sequência na memória e seu final é marcado com um *byte* igual a zero ('\0'). Isso simplifica o uso, pois se necessita apenas do endereço inicial, já o que final é indicado pelo 0. Abaixo está o exemplo de uma pequena *string* que está carregada a partir do endereço 0x2400. É claro que cada letra é representada pelo seu código ASCII.

0x2400	0x2401	0x2402	0x2403	0x2404	0x2405	0x2406	0x2407	0x2408
A	S	S	E	M	B	L	Y	0

Figura 2.15. Exemplo da representação de uma string na convenção usada pela linguagem C.

Escreva a sub-rotina TAM_STR, que recebe em R5 o endereço de início de uma *string* e retorna em R10 seu tamanho. O zero, que marca o final da *string*, não deve ser contado.

EP 2.24. Escreva a sub-rotina INV_STR, que recebe em R5 o endereço de início de uma *string* e inverte a ordem de seus elementos. A figura abaixo apresenta o resultado caso a sub-rotina fosse aplicada no exemplo do exercício anterior.

0x2400	0x2401	0x2402	0x2403	0x2404	0x2405	0x2406	0x2407	0x2408
Y	L	B	M	E	S	S	A	0

Figura 2.16. Resultado da sub-rotina INV_STR aplicada ao exemplo da Figura 2.12.

EP 2.25. Escreva a sub-rotina CAT_STR, que concatena duas *strings* cujos endereços estão em R5 e R6. A *string* apontada por R6 deve ser adicionada ao final da *string* apontada por R5. Veja o exemplo abaixo.

Antes	n	n+1	n+2	n+3	n+4	n+5
R5→	T	U	R	B	O	0

Antes	m	m+1	m+2	m+3	m+4	m+5
R6→	L	E	N	T	O	0

Depois	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10
R5→	T	U	R	B	O	L	E	N	T	O	0

Figura 2.17. Exemplo da concatenação de duas strings.

EP 2.26. Considerando o exercício anterior, como nada se disse sobre os endereços contidos em R5 e R6, existe o perigo da nova *string* se sobrepor à *string* apontada por R6. Vamos então expandir o exercício anterior. Escreva a sub-rotina CAT_STRG, que concatena duas *strings* cujos endereços estão em R5 e R6 e armazena a *string* resultante a partir do endereço apontado por R7. Se supõe que R7 aponta para o início de uma porção de memória com espaço suficiente para receber a nova *string*.

EP 2.27. Diferente do que acontece quando se programa em C, os montadores *assembly*, usualmente, não fazem distinção entre letras maiúsculas e minúsculas. (No caso particular do CCS, esta distinção é feita apenas para o *labels*). Assim, antes de iniciar a montagem, o montador converte para letras maiúsculas todos os caracteres do arquivo fonte. Escreva a sub-rotina MAIUSC, que transforma em maiúscula todas as letras da *string* cujo endereço está em R5.

EP 2.28. Escreve a sub-rotina FREQ, que calcula a frequência de cada letra presente na *string* apontada por R5. Esta sub-rotina deve criar a partir da posição apontada por R6 um vetor de 26 posições indicando a frequência de cada letra. A primeira posição para a frequência da letra “A”, a segunda posição para a frequência de letra “B” e assim por diante. Não deve ser feita distinção entre maiúsculas e minúsculas.

EP 2.29. Vamos propor um sistema muito simples de cifragem "deslocando" as letras que compõem uma *string*. Com o deslocamento de 1, a letra "A" é substituída" pela letra "B", o "B" é substituído pelo "C" e assim por diante. Já com o deslocamento de 2, a letra "A" é substituída" pela letra "C", etc. Os caracteres numéricos devem seguir a mesma lógica. O deslocamento deve trabalhar de forma circular com a sequência de letras maiúsculas (A, B, ..., Y, Z, A, B, ..., Y, Z, A, ...), com as letras minúsculas (a, b, ..., y, z, a, b, ..., y, z, a, ...) e com a sequência de números (0, 1, ..., 9, 0, 1 ..., 9, 0, ...). Os espaços em branco e a pontuação não devem ser alterados. Veja os exemplos na figura abaixo.

Pedido: Escreva a sub-rotina CIFR que cifra a *string* apontada por R5 usando o deslocamento indicado por R6. Os valores de R5 e R6 devem ser preservados.

Exemplo 1:

Recebe		n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8
R6 = 1	R5 = n →	H	A	L		9	0	0	0	0

Retorna		n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8
R6 = 1	R5 = n →	I	B	M		0	1	1	1	0

Curiosidade: quem foi HAL 9000?

Exemplo 2:

Recebe		n	n+1	n+2	n+3	n+4	n+5	n+5	n+6	n+7	n+8
R6 = 3	R5 = n →	A	l	o		D	a	v	i	d	0

Retorna		n	n+1	n+2	n+3	n+4	n+5	n+5	n+6	n+7	n+8
R6 = 3	R5 = n →	D	o	r		F	d	y	l	f	0

Figura 2.18. Exemplos de cifragem por deslocamentos segundo a ordem alfabética.

EP 2.30. Este exercício é semelhante ao EP 2.22. Escreva a sub-rotina CRTL_FS que verifica se a *string* apontada por R5 contém a *string* apontada por R6. Retorna *Carry* em 1, caso positivo e *Carry* em 0, caso negativo. Esta sub-rotina lembra o comando de busca "CTRL + F" usado em vários programas.

EP 2.31. Eratóstenes foi um grande matemático e astrônomo grego, nascido em 276 A.C. Ele foi o primeiro a medir o raio da terra e também propôs um método para calcular números primos, que hoje em dia é chamado de "Crivo (ou peneira) de Eratóstenes". Vamos exemplificar o crivo calculando os números primos de 1 até 15, que está apresentado na Figura 2.19. Partirmos com uma lista com todos os números de 1 até 15. Sabemos que o número 1 é primo, o deixamos separado.

- No Passo 1 eliminamos os múltiplos de 2, que é o nosso segundo número primo (note que, apesar de não ser muito visível, o número 4 está tachado).
- No Passo 2, dos números sobreviventes, o número seguinte ao 2 é o número 3, que é primo. Eliminamos todos os múltiplos de 3.
- No Passo 3, dos números sobreviventes, o número seguinte ao 3 é o número 5, que é primo. Eliminamos todos os múltiplos de 5.
- E assim seguimos até o final.

Início	**1**	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	...
Passo 1	**1**	**2**	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~	11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	...
Passo 2	**1**	**2**	**3**	4	5	~~6~~	7	~~8~~	~~9~~	~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	...
Passo 3	**1**	**2**	**3**	4	**5**	~~6~~	7	~~8~~	~~9~~	~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	...

Figura 2.19. Ilustração do Crivo de Eratóstenes usado para calcular os números primos entre 1 e15.

Pedido: Escreva a sub-rotina PRIMOS que cria um vetor a partir da posição apontada por R5 com os números primos (em 16 *bits*) de 1 até o limite indicado por R6.

Dica: a partir do endereço apontado por R5, preencha cada posição de 16 *bits* com a palavra 1 até o limite indicado por R6. Aplique o crivo, marcando as posições dos múltiplos com 0. Ao final, faça a compactação do vetor (desloque para a esquerda os elementos sobreviventes).

EP 2.32. A dica sugerida no exercício anterior consome muita memória. Refaça o exercício anterior usando *bytes* para marcar com 0 ou 1 as posições. Cuidado, os primos continuam sendo de 16 *bits*.

EP 2.33. Ainda sobre o Crivo de Eratóstenes, a única coisa que precisamos é marcar posições com 0 (não ímpar) ou com 1 (ímpar). Usar um *byte* ou palavra de 16 *bits* para armazenar apenas essa informação é um grande desperdício. Uma maneira mais inteligente e econômica, porém, não tão rápida, é imaginar a memória (a partir da posição apontada por R5) como um grande vetor de *bits* numerados sequencialmente. Refaça o EP 2.31 usando os *bits* de memória para trabalhar com o crivo.
Dica: crie uma sub-rotina para acessar em sequência os *bits* na memória.

EP 2.34. Escreva a sub-rotina FIB16 que grava a partir do endereço apontado por R5 os números da Sequência de Fibonacci enquanto eles forem passíveis de representação em 16 *bits*. Considerado que dois primeiros elementos são "0" e "1", os números são gerados segundo a equação $a_n = a_{n-1} + a_{n-2}$.

EP 2.35. A sequência de Fibonacci pode ser resumida pela equação recursiva $a_n = a_{n-1} + a_{n-2}$, para $n > 2$. Vamos agora inventar a sequência Tribonacci, dada pela equação:

$$a_n = a_{n-1} + 2.a_{n-2} - a_{n-3} \text{ , para } n>3.$$

Usando 1, 2 e 3 como valores iniciais, escreva a sub-rotina TRIBn que gera os "n" primeiros números (em 16 *bits*) da Sequência de Tribonacci e os armazena na memória RAM a partir da posição indicada por R5. O valor de "n" é indicado por R6.

EP 2.36. Nos exercícios resolvidos (ER 2.17 e 2.18), preparamos rotinas que transformam um *byte* em dois caracteres ASCII, de acordo com valor hexadecimal de cada *nibble*.

Neste exercício e no próximo, vamos agora preparar rotinas que fazem a operação reversa. Escreva a sub-rotina ASC_NIB que retorna em R5 o *nibble* correspondente ao valor hexadecimal do código ASCII presente em R5. Além disso, indica com *carry* = 1 se teve sucesso, e com *carry* = 0 o insucesso.

Dica: A sub-rotina deve testar se o código ASCII que está em R5 é um código válido para ser interpretado como hexadecimal. São válidos os códigos 0x30, 0x31, ..., 0x39, 0x41, 0x42, ...,0x46.

EP 2.37. Escreva a sub-rotina ASC_*BYTE* que retorna em R5 o *byte* correspondente ao valor hexadecimal dos códigos ASCII presentes em R5 (LSB) e R6 (MSB). Além disso, indica com *carry* = 1 o sucesso e com *carry* = 0 o insucesso. Use a sub-rotina preparada no item anterior.

EP 2.38. Escreva a sub-rotina FAT64, que retorna o fatorial do número que está em R5, dentro do limite da representação de 64 *bits*. O resultado deve ser armazenado na concatenação de 4 registradores [R9 | R8 | R7 | R6]. Ao retornar, o *carry* = 1 indica sucesso e *carry* = 0 o insucesso (ultrapassou representação em 64 *bits*). Use o multiplicador em *hardware* (Veja Apêndice P).

EP 2.39. Escreva a sub-rotina P_ESC8, que retorna em R7 (16 *bits*) o produto escalar de dois vetores formados por números de 8 *bits* com sinal, cujos endereços estão em R5 e R6. Ao retornar, o *carry* = 1 indica sucesso e *carry* = 0 o insucesso.

EP 2.40. Escreva a sub-rotina P_ESC16, que retorna em [R8 | R7] (32 *bits*) o produto escalar de dois vetores formados por números de 16 *bits* com sinal, cujos endereços estão em R5 e R6. Ao retornar, o *carry* = 1 indica sucesso e *carry* = 0 o insucesso.

EP 2.41. Escreva a sub-rotina P_VET8, que retorna a partir do endereço indicado por R7 o produto vetorial de dois vetores formados por números de 8 *bits* com sinal, cujos endereços estão em R5 e R6. Ao retornar, o *carry* = 1 indica sucesso e *carry* = 0 o insucesso.

EP 2.42. Escreva a sub-rotina P_VET16, que retorna a partir do endereço indicado por R7 o produto vetorial de dois vetores formados por números de 16 *bits* com sinal, cujos endereços estão em R5 e R6. Ao retornar, o *carry* = 1 indica sucesso e *carry* = 0 o insucesso.

EP 2.43. Com a finalidade de gerar exercícios, podemos sugerir uma representação de 8 *bits* com sinal para números complexos usado apenas um registrador de 16 *bits*. Usamos o MSB para a parte real e o LSB para a parte imaginária.
Assim, podemos facilmente operar com complexos. O exemplo abaixo indica:

$$R7\ (36 + 45i) = R5\ (12 + 13i) + R6\ (24 + 32i)$$

R7		=	R6		+	R5	
9.261			6.176			3.085	
36	45		24	32		12	13

Note que 9.261 = 36*256 + 45, que 6.176 = 24*256 + 32 e que 3.085 = 12*256 + 13.

Figura 2.20. Operação com números complexos usando um registrador de 16 bits para representar as partes real e imaginária.

Pedido: Considerando, por simplicidade, que os resultados sejam passíveis de representação em inteiros de 8 *bits* com sinal, escreva sub-rotinas para as 4 operações aritméticas usando a representação para números complexos sugerida, de acordo com a descrição a seguir:

- SUMC: faz R7 = R6 + R5;
- SUBC: faz R7 = R6 – R5;
- MULC: faz R7 = R6 * R5 e
- DIVC: faz R7 = R6 / R5

Observação: a sub-rotina DIVC deve oferecer bastante dificuldade. Apresente sugestões para sua solução.

EP 2.44. Este exercício e os próximos fazem uso do Apêndice E que explica a máquina Enigma e apresenta alguns modelos simplificados. Aqui é pedido para construir a sub-rotina `ENG1` (Enigma 1), que é o Enigma usando apenas o Rotor I e o Refletor I da Tabela E.1. Note que o rotor não gira e que sua Configuração (*Ringstellung*) e sua Posição Inicial (*Grundstellung*) não foram alteradas. A Figura 2.21 apresenta sua ilustração.

Contatos
A-1 →
B-2 →
C-3 →
D-4 →
E-5 →
F-6 →

Letras	**Ordem**	**Rotor I**
A	01	03
B	02	05
C	03	01
D	04	02
E	05	04
F	06	06

Figura 2.21. Ilustração do uso do Rotor I. Na coluna da esquerda estão os contados com o corpo do Enigma. Note que a letra A (contato A-1) se conecta com a letra A do Rotor I e assim por diante.

Como mostrado abaixo, a mensagem a ser cifrada deve estar na memória de dados indicada pelo *label* `msg` com seu final marcado pelo *byte* zero. O resultado da cifragem deve ser armazenado no local indicado pelo *label* `gsm` e ter seu final também marcado pelo *byte* zero. Para conferir se o programa funciona corretamente, apresente o texto cifrado ao seu programa e ele deve resultar no texto em claro.

```
EP2p44:     MOV         #msg,R5     ;Ponteiro para mensagem em claro
            MOV         #gsm,R6     ;Ponteiro para mensagem cifrada
            CALL        #ENG1       ;Chamar sua sub-rotina
            JMP         $           ;Prender execução
;
; sua sub-rotina ENG1
;
            .data
msg:        .byte       "CABECAFEFACA",0  ;Mensagem em claro
gsm:        .byte       "XXXXXXXXXXXX",0  ;Mensagem cifrada
```

EP 2.45. Este exercício pede a sub-rotina `ENG2` (Enigma 2). O exercício é a repetição do anterior, mas agora vamos mudar apenas a Configuração (*Ringstellung*) do Rotor I. Conforme combinamos no Apêndice E, essa Configuração é indicada pela letra que coincide com a primeira posição (número 1) do anel interno numerado. Na Figura 2.21 (apresentada acima), o Rotor I está na configuração A. Vamos mudá-la para a Configuração C, ou seja, o anel de letras gira de forma que a letra C coincida com o número 1 (posição 1 do anel interno), como mostrado na Figura 2.22. Em consequência, a letra D vai coincidir com o número 2 e assim por diante. Refaça a cifragem usando o Rotor I na Configuração C.

Contatos	**Configuração A (*Ringstellung A)***			**Configuração C (*Ringstellung C)***		
	Letras	**Ordem**	**Rotor I**	**Letras**	**Ordem**	**Rotor I**
A-1 →	**A**	01	03	**C**	01	03
B-2 →	**B**	02	05	**D**	02	05
C-3 →	**C**	03	01	**E**	03	01
D-4 →	**D**	04	02	**F**	04	02
E-5 →	**E**	05	04	**A**	05	04
F-6 →	**F**	06	06	**B**	06	06

Figura 2.22. Ilustração da Configuração (Ringstellung) do Rotor I. Na esquerda está a Configuração A e na direita a Configuração C. Note que nesta Configuração C, a letra A (contato A-1) se conecta com a letra C do Rotor I e assim por diante.

EP 2.46. Este exercício pede a sub-rotina `ENG3` (Enigma 3). O exercício é a repetição do anterior, mas agora vamos mudar a Posição Inicial (*Grundstellung*) do Rotor I. Conforme combinamos no Apêndice E (Figura E.11), essa Posição Inicial é indicada pela letra do Rotor que aparece numa janela (aberta no corpo do Enigma) que exibe sua terceira posição, ou seja, a letra correspondente ao terceiro contato com o corpo do Enigma. Refaça a cifragem usando o Rotor I na Configuração C e na Posição Inicial A.

Contato	Configuração C Posição Inicial E (*Grundstellung E*)			Configuração C Posição Inicial A (*Grundstellung A*)		
	Letras	**Ordem**	**Rotor I**	**Letras**	**Ordem**	**Rotor I**
A-1 →	**C**	01	03	**E**	03	01
B-2 →	**D**	02	05	**F**	04	02
C-3 →	**E**	03	01	**A**	05	04
D-4 →	**F**	04	02	**B**	06	06
E-5 →	**A**	05	04	**C**	01	03
F-6 →	**B**	06	06	**D**	02	05

Figura 2.23. Ilustração da Posição Inicial (Grundstellung) do Rotor I. Na esquerda está a Posição Inicial E e na direita a Posição Inicial A. Note que no presente caso, com a Posição Inicial A, a letra A (contato A-1) se conecta com a letra E do Rotor I e assim por diante.

EP 2.47. Este exercício pede a sub-rotina `ENG4` (Enigma 4). O exercício é a repetição do anterior, mas agora o Rotor vai girar a cada letra cifrada. Com isso o mapeamento das letras será alterado a cada cifragem. Adotamos a seguinte convenção: primeiro cifra e depois gira. A Figura 2.24 ilustra a cifragem da primeira e da segunda letra. Então, refaça a cifragem usando o Rotor I na Configuração C e na Posição Inicial A, girando-o a cada letra cifrada.

Contato	Posição para a 1ª Letra			Posição para a 2ª Letra		
	Letras	**Ordem**	**Rotor I**	**Letras**	**Ordem**	**Rotor I**
A-1 →	↑ **E**	03	01	↑ **F**	04	02
B-2 →	↑ **F**	04	02	↑ **A**	05	04
C-3 →	↑ **A**	05	04	↑ **B**	06	06
D-4 →	↑ **B**	06	06	↑ **C**	01	03
E-5 →	↑ **C**	01	03	↑ **D**	02	05

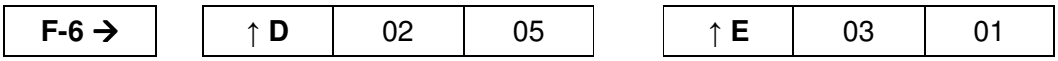


Figura 2.24. Ilustração da cifragem consecutiva de duas letras, considerando que o rotor gire no sentido indicado. Primeiro ele cifra e depois gira. Foi usado o Rotor I, na Configuração C (Ringstellung C) e Posição Inicial A (Grundstellung A).

EP 2.48. Vamos agora ao desafio do problema inverso, usando a força bruta. Com relação ao EP 2.44, considere que você tem a mensagem em claro e a mensagem cifrada. Escreva a sub-rotina `GNE1`, que descobre o mapeamento interno do Rotor usado e o escreve no vetor `MAP`.

```
EP2p48:     MOV         #msg,R5     ;Ponteiro para mensagem em claro
            MOV         #gsm,R6     ;Ponteiro para mensagem cifrada
            CALL        #GNE1       ;Chamar seu programa
            JMP         $           ;Prender execução
;
; sua sub-rotina GNE1
;
            .data
msg:        .byte       "CABECAFEFACA",0  ;Mensagem em claro
gsm:        .byte       "XXXXXXXXXXXX",0  ;Mensagem cifrada (EP 2.44)
            ; Ordem:    1 2 3 4 5 6
map:        .byte       0,0,0,0,0,0       ;Resposta
```

EP 2.49. Considerando o ambiente do EP 2.45, de posse da mensagem em claro e da mensagem cifrada, escreva sub-rotina `GNE2`, que descobre o mapeamento interno do Rotor usado e sua Configuração (*Ringstellung*).

EP 2.50. Considerando o ambiente do EP 2.46, de posse da mensagem em claro e da mensagem cifrada, escreva sub-rotina `GNE3`, que descobre o mapeamento interno do Rotor usado, sua Configuração (*Ringstellung*) e sua Posição Inicial (*Grundstellung*).

EP 2.51. Considerando o ambiente do EP 2.47 (o rotor gira), de posse da mensagem em claro e da mensagem cifrada, escreva sub-rotina `GNE4`, que descobre o mapeamento interno do Rotor usado, sua Configuração (*Ringstellung*) e sua Posição Inicial (*Grundstellung*). Concordo com você, leitor, vai ficando complicado!

EP 2.52. Considerando o ambiente do EP 2.47 (o rotor gira) e de posse apenas da mensagem cifrada, escreva sub-rotina `GNE5`, que descobre o mapeamento interno do Rotor usado, Configuração (*Ringstellung*) e sua Posição Inicial (*Grundstellung*). A única informação que você tem é que a mensagem em claro contém palavra "FACA", mas não sabe em que posição ela está. Novamente estamos de acordo: ficou ainda mais complicado!

EP 2.53. Este exercício pede a sub-rotina `ENG5` (Enigma 5), que gira o rotor a cada letra cifrada. Ela recebe a indicação do Rotor a ser usado (um dentre os 5 disponíveis), sua Configuração (*Ringstellung*), sua Posição Inicial (*Grundstellung*) e ainda a seleção do Refletor. Veja uma sugestão a seguir.

```
; Seleção dos parâmetros do Enigma
ROTOR:      .equ        3           ;Indicando Rotor 3
CONF:       .equ        'C'         ;Configuração
POS:        .equ        'E'         ;Posição Inicial
REF:        .equ        2           ;Refletor 2
;
EP2p53:     MOV         #msg,R5     ;Ponteiro para mensagem em claro
            MOV         #gsm,R6     ;Ponteiro para mensagem cifrada
            CALL        #ENG5       ;Chamar sua sub-rotina
            JMP         $           ;Prender execução
;
; sua sub-rotina ENG5
;
;Rotores com 6 posições
ROTORES:
RT1:  .byte 03, 05, 01, 02, 04, 06
RT2:  .byte 01, 05, 03, 04, 02, 06
RT3:  .byte 02, 05, 06, 01, 03, 04
RT4:  .byte 05, 02, 06, 01, 03, 04
RT5:  .byte 01, 04, 05, 03, 06, 02

;Refletores com 6 posições
REFLETORES:
RF1:  .byte 04, 05, 06, 01, 02, 03
RF2:  .byte 05, 06, 04, 03, 01, 02
RF3:  .byte 04, 03, 02, 01, 06, 05

            .data
msg:        .byte       "CABECAFEFACA",0  ;Mensagem em claro
gsm:        .byte       "XXXXXXXXXXXX",0  ;Mensagem cifrada
```

EP 2.54. Este exercício pede a sub-rotina `GNE5` (Inverso do Enigma 5). Ela considera que você conhece o mapeamento interno de todos os rotores e refletores. Porém, não sabe quais foram usados e nem sua Configuração (*Ringstellung*) e sua Posição Inicial (*Grundstellung*). Então, de posse da mensagem em claro e da mensagem cifrada, descubra todos esses parâmetros.

EP 2.55. Este exercício pede a sub-rotina `ENG6` (Enigma 6), que faz uso de 3 rotores girando. A cada volta completa de um rotor, o da direita anda uma posição (Figura E.12). A sub-rotina deve permitir a seleção dos rotores e sua disposição relativa, a Configuração

(*Ringstellung*) e a Posição Inicial (*Grundstellung*) de cada um deles e ainda a seleção do Refletor. Veja sugestão a seguir.

```
EP2p55:     MOV         #msg,R5     ;Ponteiro para mensagem em claro
            MOV         #gsm,R6     ;Ponteiro para mensagem cifrada
            CALL        #ENG6       ;Chamar sua sub-rotina
            JMP         $           ;Prender execução
;
; Configuração do Enigma
SEL_ROT:    .byte 2,1,4       ;Usar os Rotores 2:1:4
SEL_CONF:   .byte 'B','D','A' ;Configuração dos rotores acima
SEL_POS:    .byte 'C','B','F' ;Posição Inicial dos rotores acima
SEL_REF:    .byte 2           ;Refletor selecionado
;
; sua sub-rotina ENG5
;
;Rotores com 6 posições
ROTORES:
RT1:  .byte 03, 05, 01, 02, 04, 06
RT2:  .byte 01, 05, 03, 04, 02, 06
RT3:  .byte 02, 05, 06, 01, 03, 04
RT4:  .byte 05, 02, 06, 01, 03, 04
RT5:  .byte 01, 04, 05, 03, 06, 02
;
;Refletores com 6 posições
REFLETORES:
RF1:  .byte 04, 05, 06, 01, 02, 03
RF2:  .byte 05, 06, 04, 03, 01, 02
RF3:  .byte 04, 03, 02, 01, 06, 05

            .data
msg:        .byte       "CABECAFEFACA",0  ;Mensagem em claro
gsm:        .byte       "XXXXXXXXXXXX",0  ;Mensagem cifrada
```

EP 2.56. Este exercício pede a sub-rotina `GNE6` (Inverso do Enigma 6). Ela considera que você conhece o mapeamento interno de todos os rotores e refletores. Porém, não sabe quais foram usados e nem a Configuração (*Ringstellung*) e a Posição Inicial (*Grundstellung*) de cada um deles e ainda desconhece qual Refletor foi empregado. Então, de posse da mensagem em claro e da mensagem cifrada, descubra todos esses parâmetros.

EP 2.57. Este exercício pede a sub-rotina `ENG7` (Enigma 7), que é o ER 2.55, com a possibilidade de adicionar o Painel de Plugs (Figura E.15). Como temos apenas 6 letras, vamos permitir a troca entre 2 pares de letras. Por exemplo, trocar as letras A↔C e B↔F (vide listagem abaixo). As letras D e E permanecem inalteradas. Veja sugestão a seguir.

Note que as trocas realizadas pelo Painel de Plugs (`SEL_PP`) tem seu limite marcado com um byte igual a zero.

```
EP2p55:     MOV         #msg,R5     ;Ponteiro para mensagem em claro
            MOV         #gsm,R6     ;Ponteiro para mensagem cifrada
            CALL        #ENG7       ;Chamar sua subrotina
            JMP         $           ;Prender execução
;
; Configuração do Enigma
SEL_ROT:    .byte 2,1,4       ;Usar os Rotores 2:1:4
SEL_CONF:   .byte 'B','D','A' ;Configuração dos rotores acima
SEL_POS:    .byte 'B','D','A' ;Posição Inicial dos rotores acima
SEL_REF:    .byte 2           ;Refletor selecionado
SEL_PP:     .byte 'A','C','B','F',0 ;Painel de Plugs (A e C) (B e F)
;
; sua sub-rotina ENG6
;
;Rotores com 6 posições
ROTORES:
RT1:  .byte 03, 05, 01, 02, 04, 06
RT2:  .byte 01, 05, 03, 04, 02, 06
RT3:  .byte 02, 05, 06, 01, 03, 04
RT4:  .byte 05, 02, 06, 01, 03, 04
RT5:  .byte 01, 04, 05, 03, 06, 02
;
;Refletores com 6 posições
REFLETORES:
RF1:  .byte 04, 05, 06, 01, 02, 03
RF2:  .byte 05, 06, 04, 03, 01, 02
RF3:  .byte 04, 03, 02, 01, 06, 05

            .data
msg:        .byte       "CABECAFEFACA",0  ;Mensagem em claro
gsm:        .byte       "XXXXXXXXXXXX",0  ;Mensagem cifrada
```

EP 2.58. Este exercício pede a sub-rotina `GNE7` (Inverso do Enigma 7). Ela considera que você conhece o mapeamento interno de todos os rotores e refletores. Porém, não sabe quais rotores foram usados e nem a Configuração (*Ringstellung*) e a Posição Inicial (*Grundstellung*) de cada um deles, ignora qual Refletor foi empregado e ainda desconhece as conexões do Painel de Plugs (realmente, inferno completo). Então, de posse da mensagem em claro e da mensagem cifrada, descubra todos esses parâmetros. Ficou complicado? Quanto tempo sua rotina leva para descobrir essa configuração?

EP 2.59. Este exercício pede a sub-rotina `ENIGMA1`, que é a construção do Enigma Completo. Temos 3 rotores selecionados dentre 5, cada um com sua Configuração (*Ringstellung*) e sua Posição Inicial (*Grundstellung*), um Painel Refletor selecionado

dentre 3 e ainda a possibilidade de trocas de até 10 letras usando o Painel de Plugs. Lembre-se de que somente as letras são cifradas, os demais símbolos são ignorados

```
EP2p59:     MOV         #msg,R5     ;Ponteiro para mensagem em claro
            MOV         #gsm,R6     ;Ponteiro para mensagem cifrada
            CALL        #ENIGMA1    ;Chamar sua sub-rotina
            JMP         $           ;Prender execução
;
; Configuração do Enigma
SEL_ROT:    .byte 2,1,4       ;Usar os Rotores 2:1:4
SEL_CONF:   .byte 'H','R','D' ;Configuração dos rotores acima
SEL_POS:    .byte 'V','E','M' ;Posição Inicial dos rotores acima
SEL_REF:    .byte 2           ;Refletor selecionado
SEL_PP:     .byte 'A','G', 'K','B', 'M','R', 'D','U', 'H','W', 0
                  ;10 letras Trocadas pelo Painel de Plugs
;
; sua sub-rotina ENIGMA1
;
;Rotores com 26 posições
ROTORES:
RT1:  .byte 11, 19, 01, 08, 04, 03, 05, 09, 15, 18, 06, 23, 21
      .byte 26, 24, 02, 25, 14, 22, 20, 12, 07, 13, 16, 10, 17
RT2:  .byte 20, 10, 08, 24, 12, 26, 18, 17, 03, 25, 16, 11, 07
      .byte 15, 01, 09, 14, 05, 04, 19, 02, 23, 22, 21, 06, 13
RT3:  .byte 17, 22, 05, 01, 02, 13, 16, 15, 09, 26, 10, 20, 18
      .byte 07, 06, 21, 14, 25, 24, 11, 04, 23, 12, 08, 19, 03
RT4:  .byte 26, 05, 22, 23, 18, 15, 13, 09, 06, 16, 24, 07, 03
      .byte 19, 11, 17, 14, 04, 20, 21, 01, 25, 12, 08, 10, 02
RT5:  .byte 14, 06, 21, 08, 01, 16, 22, 10, 20, 15, 25, 19, 13
      .byte 07, 03, 12, 17, 09, 04, 02, 11, 24, 05, 26, 23, 18

;Refletores com 26 posições
REFLETORES:
RF1:  .byte 08, 18, 06, 20, 16, 03, 10, 01, 15, 07, 19, 17, 26
      .byte 24, 09, 05, 12, 02, 11, 04, 23, 25, 21, 14, 22, 13
RF2:  .byte 03, 14, 01, 09, 11, 20, 24, 15, 04, 23, 05, 21, 26
      .byte 02, 08, 18, 22, 16, 25, 06, 12, 17, 10, 07, 19, 13
RF3:  .byte 23, 06, 10, 13, 15, 02, 14, 11, 24, 03, 08, 22, 04
      .byte 07, 05, 25, 18, 17, 20, 19, 26, 12, 01, 09, 16, 21

            .data
; Sugestões de mensagens para cifrar
MSG1:       .byte "WAS ICH NICHT WEISS MACHT MICH NICHT HEISS",0
MSG2:       .byte "REDEN IST SILBER SCHWEIGEN IST GOLD",0
MSG3:       .byte "ALLER ANFANG IST SCHWER",0
MSG4:       .byte "MORGENSTUND HAT GOLD IM MUND",0
MSG5:       .byte "KOMMT ZEIT KOMMT RAT",0
MSG6:       .byte "IRREN IST MENSCHLICH",0
MSG7:       .byte "FREMD IST DER FREMDE NUR IN DER FREMDE",0
```

```
GSM:        .space 100  ;Espaço para mensagens cifradas
```

Curiosidade: a MSG7 é famosa. Quem é o autor e o que ela significa?

Observação: o link abaixo apresenta uma página do cryptomuseum.com, onde se pode ver uma mensagem em claro (Alemão) e depois em inglês junto com a cifragem correspondente.
https://www.cryptomuseum.com/crypto/enigma/msg/p1030681.htm

EP 2.60. Este exercício pede a sub-rotina `BOMBE`, que faz o inverso do EP 2.59. Esta sub-rotina conhece o mapeamento interno de todos os rotores e refletores. Porém, não sabe quais rotores foram usados, sua ordem relativa, e nem a Configuração (*Ringstellung*) e a Posição Inicial (*Grundstellung*) ou qual Refletor foi empregado e ainda desconhece as conexões do Painel de Plugs. Então, de posse da `MSG1` em claro e de sua cifragem, descubra todos esses parâmetros. Faça testes com as demais mensagens. Ficou complicado? Quanto tempo sua rotina leva para descobrir essa configuração? Como será que no século passado os britânicos quebravam esses códigos em menos de 24 horas?

EP 2.61. Propor um exercício com o DES. Buscar por um paper com uma boa descrição.