



Arquitetura e Organização de Computadores

Exercícios - Parte II

António José Araújo

João Canas Ferreira

Bruno Lima

Mestrado Integrado em Engenharia Informática e Computação

Novembro de 2018

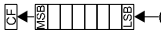




Conteúdo

1	Linguagem <i>assembly</i>	1	1.3 Exercícios propostos	8
1.1	Sumário das instruções ARM . .	1	Soluções dos exercícios propostos	11
1.2	Exercícios resolvidos	1	1 Linguagem <i>assembly</i>	11

1 Linguagem *assembly*

1.1 Sumário das instruções ARM

As instruções ARMv7 utilizadas em AOCO estão indicadas na tabela seguinte.

Categoria	Instrução	Mnemónica	Significado	Flags			
				N	Z	C	V
Aritmética	Add	ADD{S}{cond} dest, op1, op2	dest = op1 + op2	±	±	±	±
	Subtract	SUB{S}{cond} dest, op1, op2	dest = op1 - op2	±	±	±	±
	Add with Carry	ADC{S}{cond} dest, op1, op2	dest = op1 + op2 + carry	±	±	±	±
	Subtract with Carry	SBC{S}{cond} dest, op1, op2	dest = op1 - op2 - 1 + carry	±	±	±	±
	Reverse Subtract	RSB{S}{cond} dest, op1, op2	dest = op2 - op1	±	±	±	±
	Reverse Subtract with Carry	RSC{S}{cond} dest, op1, op2	dest = op2 - op1 - 1 + carry	±	±	±	±
Lógica	Bitwise And	AND{S}{cond} dest, op1, op2	dest = AND(op1, op2)	±	±	x	x
	Bitwise Exclusive Or	EOR{S}{cond} dest, op1, op2	dest = XOR(op1, op2)	±	±	x	x
	Bitwise Clear	BIC{S}{cond} dest, op1, op2	dest = AND(op1, NOT(op2))	±	±	x	x
	Bitwise Or	ORR{S}{cond} dest, op1, op2	dest = OR(op1, op2)	±	±	x	x
	Logical Shift Left	LSL{S}{cond} dest, op1, op2		±	±	±	x
	Logical Shift Right	LSR{S}{cond} dest, op1, op2		±	±	±	x
	Arithmetic Shift Right	ASR{S}{cond} dest, op1, op2		±	±	±	x
	Rotate Right	ROR{S}{cond} dest, op1, op2		±	±	±	x
Transferência de dados	Rotate Right and Extend	RRX{S}{cond} dest, op1		±	±	±	x
	Move	MOV{S}{cond} dest, op1	dest = op1	±	±	x	x
	Move Negated	MVN{S}{cond} dest, op1	dest = NOT(op1)	±	±	x	x
	Address Load	ADR{cond} dest, expression		x	x	x	x
	LDR Pseudo-Instruction	LDR{cond} dest, =expression		x	x	x	x
	Load Register	LDR{B}{cond} dest, [source {, OFFSET}]	dest = Mem[source + OFFSET]	x	x	x	x
Comparações e saltos	Store Register	STR{B}{cond} source, [dest {, OFFSET}]	Mem[dest + OFFSET] = source	x	x	x	x
	Compare	CMP{cond} op1, op2	op1 - op2	±	±	±	±
	Compare Negated	CMN{cond} op1, op2	op1 + op2	±	±	±	±
	Test Bit(s) Set	TST{cond} op1, op2	AND(op1, op2)	±	±	x	x
	Test Equals	TEQ{cond} op1, op2	XOR(op1, op2)	±	±	x	x
	Branch	B{cond} target		x	x	x	x
Comparações e saltos	Branch with Link	BL{cond} target		x	x	x	x

1.2 Exercícios resolvidos

Exercício 1

Para as seguintes expressões aritméticas (números inteiros de 32 bits), especifique um mapeamento de variáveis para registos e o fragmento de código *assembly* ARM que as implementa.

$$\text{a) } f = g - (f + 5)$$

$$\text{b) } f = A[12] + 17$$

O primeiro passo neste tipo de problemas é escolher uma atribuição de variáveis a registos. Cada variável é atribuída a um registo. Como a arquitetura ARM possui 12 registos de uso geral, trata-se de uma tarefa simples porque, neste caso, se pode usar um registo diferente para cada variável.

a) Uma possível atribuição de registos a variáveis é a seguinte:

R0: *f* R1: *g*

O fragmento de código que realiza os cálculos desejados é:

```
add    R0, R0, #5           ; Calcula f = f + 5
sub    R0, R1, R0           ; Calcula f = g - f
```

Após esta sequência de duas instruções, R0 contém o novo valor associado a *f*. O cálculo da primeira parte da expressão (instrução `add`) pode guardar o resultado intermédio no registo R0, porque a segunda instrução estabelece o valor final correto.

b) Possível atribuição de variáveis a registos:

R0: *f* R6: endereço base de *A*

Como cada elemento de uma sequência de inteiros tem 4 bytes, o elemento de índice 12 da sequência *A* está guardado a partir da posição de memória cujo endereço é dado por:

$$\text{endereço base de } A + 12 \times 4$$

A primeira instrução deve ir buscar o valor guardado nessa posição.

```
ldr    R0, [R6, #48]        ; Carrega valor da posição R6+48
add    R0, R0, #17          ; Soma-lhe o valor 17
```

Exercício 2

Assuma as seguintes condições iniciais:

$R0 = 0XB\text{EADFEED}$ $R1 = 0x\text{DEADFADE}$

- a) Determine o valor de R2 após a execução da seguinte sequência de instruções:

```
lsl    R2, R0, #4
orr    R2, R2, R1
```

- b) Determine o valor de R2 após a execução da seguinte sequência de instruções:

```
lsr    R2, R0, #3
and    R2, R2, #0xFFFFFEFH
```

Em binário, os valores iniciais dos registos são:

$R0 = 1011\ 1110\ 1010\ 1101\ 1111\ 1110\ 1110\ 1101_2$
 $R1 = 1101\ 1110\ 1010\ 1101\ 1111\ 1010\ 1101\ 1110_2$

- a) A primeira instrução desloca o valor de R0 quatro bits para a esquerda. Os 4 bits mais significativos perdem-se; nos 4 menos significativos são introduzidos zeros. O resultado da operação é guardado em R2; o registo R0 fica inalterado. O valor de R2 depois da execução da primeira instrução é:

$R2 = 1110\ 1010\ 1101\ 1111\ 1110\ 1110\ 1101\ 0000_2$

A instrução **orr** calcula a função ou-inclusivo de cada bit de R2 com o bit de R0 situado na mesma posição. O resultado é guardado em R2. O resultado da operação **orr** é 1 sempre que pelo menos um dos operandos seja 1. Logo:

$R2 = 1111\ 1110\ 1111\ 1111\ 1111\ 1110\ 1101\ 1110_2 = 0x\text{FEFFFFEDE}$

- b) A instrução **lsr** desloca o valor de R0 três posições para a direita, introduzindo zeros pela esquerda. O valor de R2 depois da execução da primeira instrução é:

$R2 = 000\ 1\ 0111\ 1101\ 0101\ 1011\ 1111\ 1101_2$

A instrução **and** calcula a função e-lógico de cada bit de R2 com o bit correspondente da constante **0xFFFFFEFH**

$\text{FFFFFFEF}_{16} = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 1111_2$

O resultado é guardado em R2. O e-lógico de dois bits tem resultado 1 apenas se ambos os operandos forem 1. Neste caso, os operandos são dados pelo conteúdo de R2 e pela constante indicada. O valor final de R2 é:

$R2 = 0001\ 0111\ 1101\ 0101\ 1011\ 1111\ 1100\ 1101_2 = 0x\text{17D5BFCD}$

Exercício 3

Assuma as seguintes condições iniciais:

$$R0 = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2$$

$$R1 = 0011\ 1111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000_2$$

Determine o valor de R1 após a execução do fragmento seguinte:

```

                cmp      R0, R1
                bge      ELSE
                b        DONE
ELSE            add      R1, R1, #2
DONE           ...

```

A primeira instrução compara os conteúdos de R0 e R1 alterando o valor das *flags* (registro CPSR, *Current Processor Status Register*) de acordo com o resultado da comparação. A operação realizada é equivalente a:

$$R0 - R1$$

Neste caso os valores dos indicadores (*flags*) são alterados para N=1, Z=0, C=1 e V=0.

O salto condicional (segunda instrução) é tomado se as *flags* N e V apresentarem o mesmo valor, o que se verifica quando R0 é maior ou igual a R1 (a condição **ge** interpreta os valores dos registos como sendo números em complemento para 2). Como neste caso o conteúdo de R0 é negativo e o conteúdo de R1 é positivo, o salto não será tomado.

Em consequência, a terceira instrução a ser executada é a de salto incondicional (a instrução **b**). Esta instrução leva o fluxo de execução a passar para o fim do fragmento apresentado (etiqueta **DONE**). A instrução **add** não é executada.

Como o conteúdo de R1 não foi alterado, o seu valor não sofre alteração.

Exercício 4

Apresenta-se a seguir um programa em *assembly*. Na quinta linha do programa é invocada uma sub-rotina, através da instrução *bl* (*branch with link*), designada por **par**. Esta sub-rotina, cujo código não é fornecido, tem como objetivo determinar se um número é par. Admita que a sub-rotina recebe esse número no registo R0 e que devolve no registo R0 o valor 1 caso seja par, ou 0 no caso contrário.

```

                                mov    R10, #0xFF000000
                                mov    R1, #3
                                mov    R2, #0
loop    str    R0, [R10]
                                bl     par                ; Chama rotina par
                                cmp    R0, #0
                                beq    step
                                add    R2, R2, #1
step    ldr    R0, [R10]
                                add    R0, R0, #1
                                adds   R1, R1, #-1
                                bne    loop
                                END

```

- Considerando que antes da execução do programa o registo R0 possui o valor 8, indique o conteúdo dos registos R0 e R2 após a execução do programa.
- Tendo em consideração a descrição que foi realizada, implemente a rotina **par**.

- Em algumas situações é útil preservar o valor do registo R0 (onde, segundo a convenção as sub-rotinas devem retornar o resultado), para isso uma solução possível é armazenar o conteúdo do registo numa posição de memória antes da invocação da sub-rotina e voltar a carregar esse valor após a execução da mesma. Neste caso foi utilizado o endereço de memória 0xFF000000 para armazenar o conteúdo do registo R0.

O programa realiza 3 iterações, incrementando em cada uma delas o valor R0 por forma a que o mesmo pode-se ser incrementado. de R0, que inicialmente é 8. Em cada iteração é determinada a paridade do valor em R0 invocando a rotina **par**. Caso o valor em R0 seja par o registo R2 é incrementado. No final da execução do programa R2=2, correspondendo aos 2 números pares encontrados (8 e 10), e R0=11, correspondendo ao resultado da adição de uma unidade em cada iteração ao valor de R0.

- O bit menos significativo de um número par é 0. A rotina seguinte baseia-se nesta propriedade.

```

par    add    R0, R0, #1
        and    R0, R0, #1
        mov    PC, LR

```

Exercício 5

- a) Escreva um fragmento de código *assembly* que determina se um dado número inteiro *N* está presente numa sequência *SEQ*. Assuma a seguinte atribuição de registos:
- $N \rightarrow R0$
 - endereço-base de *SEQ* $\rightarrow R1$
 - número de elementos de *SEQ* $\rightarrow R2$
 - resultado $\rightarrow R0$
- b) Converta o fragmento anterior numa sub-rotina chamada **pesq** (de “pesquisa”). Os argumentos da função seguem a ordem indicada na alínea anterior.
- c) Use a sub-rotina anterior num fragmento que determina quantos elementos de uma sequência *SEQ1* estão presentes noutra sequência *SEQ2*. Usar a seguinte atribuição de registos:
- endereço-base de *SEQ1* $\rightarrow R7$
 - número de elementos de *SEQ1* $\rightarrow R8$
 - endereço-base de *SEQ2* $\rightarrow R9$
 - número de elementos de *SEQ2* $\rightarrow R10$
 - resultado $\rightarrow R5$

- a) O fragmento consiste num ciclo em que se varia o registo *R1* de forma a conter o endereço de elementos sucessivos de *SEQ*. O número de iterações é, no máximo, igual ao número de elementos de *SEQ*, que é decrementado de uma unidade em cada iteração.

O ciclo pode terminar, quando se encontra um elemento igual ao procurado, nesse caso é colocado em *R3* o valor 1 e a instrução de salto para a etiqueta *fim* é executada.

Se o valor procurado não existir em *SEQ*, o ciclo é terminado porque o contador de elementos vem a 0. Neste caso, o valor de *R3* não é alterado, mantendo o valor inicial estabelecido na primeira instrução.

Na última linha o valor de *R3* é transferido para o registo *R0* de forma a deixar o resultado em *R0* tal como é pedido no enunciado.

```

1      mov      R3, #0      ; resultado a Zero
2 prox  cmp      R2, #0      ; terminar?
3      beq      fim
4      ldr      r4, [r1]    ; obter elemento
5      cmp      r4, R0      ; elemento = N?
6      bne      seg
7      mov      R3, #1      ; encontrado, resultado a 1
8      b        fim
9 seg   add      r1, r1, #4 ; atualizar endereço
10     sub      r2, r2, #1 ; ajustar numero de elementos
11     b        prox
12 fim   mov      r0, r3      ; colocar resultado em R0
```

- b) Para transformar o fragmento numa sub-rotina é necessário alterá-lo para corresponder às convenções de invocação: os argumentos nos registos *R0–R3* e o resultado em *R0*.

A atribuição de registos passa a ser a seguinte:

- $N \rightarrow R0$
- endereço-base de $SEQ \rightarrow R1$
- número de elementos de $SEQ \rightarrow R2$
- resultado $\rightarrow R0$

```

1  pesq    mov     R3, #0        ; resultado a Zero
2  L1      cmp     R2, #0        ; terminar?
3          beq     L3
4          ldr     R4, [R1]      ; obter elemento
5          cmp     R4, R0        ; elemento = N?
6          bne     L2
7          mov     R3, #1        ; encontrado, resultado a 1
8          b       L3
9  L2      add     R1, R1, #4    ; atualizar endereço
10         sub     R2, R2, #1    ; ajustar numero de elementos
11         b       L1
12  L3      mov     R0, R3        ; colocar resultado em R0
13         mov     PC, LR

```

A ultima instrução da sub-rotina `mov PC, LR`, tem com função fazer com que no final da execução da sub-rotina o programa retorne para o programa principal continuando a executar as suas instruções normalmente.

- c) O fragmento consiste num ciclo em que se “percorre” a sequência **SEQ1**. As linhas 2–3 verificam se existem elementos a processar. Em caso afirmativo, obtém-se o próximo elemento de memória; as linhas 9–10 procedem à atualização do contador de elementos e do endereço do próximo elemento.

A sub-rotina **pesq** é usada para procurar um dado elemento de **SEQ1** em **SEQ2**. As linhas 4–6 preparam a invocação da sub-rotina, colocando os argumentos nos registos apropriados (valor a procurar em **R0**, endereço-base de **SEQ2** em **R1** e número de elementos de **SEQ2** em **R2**).

A linha 8 processa o resultado da invocação (registo **R0**). Se o valor foi encontrado em **SEQ2**, o contador **R5** é incrementado de uma unidade.

```

1          mov     R5, #0        ; inicializar contador
2  ciclo   cmp     R8, #0        ; mais elementos de SEQ1?
3          beq     stop         ; terminar
4          ldr     R0, [R7]      ; obter um elemento de SEQ1
5          mov     R1, R9        ; onde pesquisar
6          mov     R2, R10       ; nº de elementos a pesquisar
7          bl      pesq         ; invocar sub-rotina
8          add     R5, R5, R0    ; atualizar contador
9          add     R7, R7, #4    ; próximo endereço
10         sub     R8, R8, #1    ; decrementar nº de elementos
11         b       ciclo        ; repetir
12  stop

```

1.3 Exercícios propostos

Exercício 6

Para as seguintes expressões aritméticas (números inteiros de 32 bits), especifique um mapeamento de variáveis para registos e o fragmento de código *assembly* ARMv7 que as implementa.

a) $f = g + (j + 2)$

b) $k = a + b - f + d - 30$

c) $f = g + h + A[4]$

d) $f = g - A[B[10]]$

e) $f = k - g + A[h + 9]$

f) $f = g - A[B[2] + 4]$

Exercício 7

Para os seguintes fragmentos de código *assembly* ARMv7, indique um mapeamento entre registos e variáveis e determine as expressões simbólicas correspondentes.

a) `add r0,r0,r1`
 `add r0,r0,r2`
 `add r0,r0,r3`
 `add r0,r0,r4`

b) `ldr r0,[r6,#4]`

c) Assumir que R6 contém o endereço-base da sequência A[].

`add r6, r6, #-20`
 `lsl r10, r1, #2`
 `add r6, r6, r10`
 `ldr r0, [r6, #8]`

Exercício 8

Assuma as seguintes condições iniciais:

$$R0 = 0x55555555$$

$$R1 = 0x12345678$$

Determine o valor de R2 após a execução das sequências de instruções seguintes.

a) `lsl r2, r0, #4`
 `orr r2, r2, r1`

b) `lsl r2, r0, #4`
 `and r2, r2, #-1`

c) `lsr r2, r0, #3`
 `and r2, r2, #0x00EF`

Exercício 9

Os processadores RISC como o ARM implementam apenas instruções muito simples. Este exercício aborda exemplos de hipotéticas instruções mais complexas.

- a) Considere uma instrução hipotética **abs** que coloca num registo o valor absoluto de outro registo.

abs R2, R1 é equivalente a $R2 \leftarrow |R1|$

Apresente a mais curta sequência de instruções ARMv7 que realiza esta operação.

- b) Repita a alínea anterior para a instrução **sgt**, em que **sgt** R1, R2, R3 é equivalente a **se** R2 > R3 **então** R1 \leftarrow 1 **senão** R1 \leftarrow 0.

Exercício 10

Considere o seguinte fragmento de código *assembly* ARMv7:

```

L1      str    R4, [R5]
        lsl    R4, R4, #4
        add    R5, R5, #4
        cmp    R4, #0
        bne    L1

```

Assuma os seguintes valores iniciais:

R4 = 0x12345678 R5 = 0x7D0

Explique como é alterada a memória durante a execução do fragmento de código. Apresente numa tabela o endereço e o conteúdo das posições de memória alteradas pela execução do fragmento de código.

Exercício 11

Numa zona de memória (endereço-base em R4) está uma sequência de bytes diferentes de 0. A sequência de bytes é terminada por um byte a zero.

Escreva um fragmento de código *assembly* que determina o número de bytes da sequência. O byte final, 0, não entra para a contagem. O resultado deve ser guardado em R0.

Exercício 12

Considerar os seguintes fragmentos de código *assembly*.

```

Fragmento 1:  LOOP    cmp    R1, #0
               beq     DONE
               add     R2, R2, #2
               add     R1, R1, #-1
               b       LOOP
               DONE
               ...

```

```

Fragmento 2:  LOOP    mov     R3, #0xA
               LOOP2   add     R2, R2, #2
               adds    R3, R3, #-1
               bne     LOOP2
               adds    R1, R1, #-1
               bne     LOOP
               DONE
               ...

```

- a) Assumir a seguinte situação inicial: $R1=10$ e $R2=0$. Determinar, para cada fragmento, o valor final de $R2$.
- b) Assuma agora que $R1=N$ (com $N>0$). Determinar, para cada fragmento, o número de instruções executadas em função de N .

Exercício 13

Escrever um fragmento para determinar se duas sequências de números inteiros (32 bits) têm o mesmo conteúdo. As sequências têm 100 elementos. Usar a seguinte atribuição de registos:

endereço-base da sequência A \rightarrow R4
endereço-base da sequência B \rightarrow R5

O resultado, a guardar em $R0$, deve ser 1, se as duas sequências tiverem o mesmo conteúdo ou 0 no caso contrário.

Exercício 14

Escreva um fragmento de código *assembly* ARMv7 para determinar quantos números ímpares estão contidos numa sequência de 50 números inteiros (de 32 bits). Assuma que o endereço-base da sequência está contido no registo $R0$. O resultado deve ficar no registo $R7$.

Exercício 15

Pretende-se escrever um programa que permita realizar diversas tarefas envolvendo sequências de números inteiros em memória. Para que o programa resulte estruturado e o código seja facilmente reutilizado, o seu desenvolvimento deve basear-se na chamada de sub-rotinas, realizando cada uma destas sub-rotinas uma tarefa específica.

Escrever o programa que realiza as tarefas a seguir descritas usando uma sub-rotina para cada tarefa.

Nota: deve gerir a utilização de registos por forma a respeitar a convenções de chamada de chamada e retorno das sub-rotinas.

- a) Somar todos os elementos de uma sequência.
- b) Determinar o número de elementos ímpares da sequência.
- c) Determinar o número de elementos que são iguais ou superiores a um valor dado.
- d) Determinar se duas sequências com o mesmo comprimento são iguais.

Soluções dos exercícios propostos

1 Linguagem *assembly*

Exercício 6

É necessário definir uma atribuição arbitrária de variáveis a registros.

a) Atribuição: $f \rightarrow r0$, $g \rightarrow r1$, $j \rightarrow r2$.

```
add    r0, r2, #2
add    r0, r1, r0
```

b) Atribuição: $a \rightarrow r0$, $b \rightarrow r1$, $d \rightarrow r2$, $f \rightarrow r3$, $k \rightarrow r4$.

```
add    r4, r0, r1
sub    r4, r4, r3
add    r4, r4, r2
add    r4, r4, #-30
```

c) Atribuição: $f \rightarrow r0$, $g \rightarrow r1$, $h \rightarrow r2$, $A \rightarrow r7$.

```
ldr    r0, [r7, #16]
add    r0, r0, r1
add    r0, r0, r2
```

d) Atribuição: $f \rightarrow r0$, $g \rightarrow r2$, $A \rightarrow r6$, $B \rightarrow r7$.

```
ldr    r5, [r7, #40]
lsl    r5, r5, #2
add    r5, r5, r6
ldr    r0, [r5]
sub    r0, r2, r0
```

e) Atribuição: $f \rightarrow r0$, $g \rightarrow r1$, $h \rightarrow r2$, $k \rightarrow r3$, $A \rightarrow r6$.

```
add    r10, r2, #9
lsl    r10, r10, #2
add    r10, r6, r10
ldr    r0, [r10]
add    r0, r0, r3
sub    r0, r0, r1
```

f) Atribuição: $f \rightarrow r0$, $g \rightarrow r1$, $A \rightarrow r6$, $B \rightarrow r7$.

```
ldr    r10, [r7, #8]
add    r10, r10, #4
lsl    r10, r10, #2
add    r10, r6, r10
ldr    r10, [r10]
sub    r0, r1, r10
```

Exercício 7

a) Atribuição: $f \rightarrow r0$, $g \rightarrow r1$, $h \rightarrow r2$, $i \rightarrow r3$, $j \rightarrow r4$

A expressão correspondente é: $f = f + g + h + i + j$

b) Atribuição: $f \rightarrow r0$, $A \rightarrow r6$

A expressão correspondente é: $f = A[1]$

c) Atribuição: $f \rightarrow r0$, $g \rightarrow r1$, $A \rightarrow r6$

A expressão correspondente é: $f = A[g-3]$

Exercício 8

a) 0x57755778

b) 0x55555550

c) 0x000000AA

Exercício 9

```
a)      movs    R2, R1
        bpl     pos
        rsb     R2, R1, #0
pos
```

Ou, melhor ainda:

```
        movs    R2, R1
        rsbmi   R2, R1, #0
```

```
b)      mov     R1, #0
        cmp     R2, R3
        movgt   R1, #1
```

Exercício 10

A cada iteração do ciclo L1 será guardado em memória (no endereço dado pelo conteúdo do registo R5) o valor do registo R4. Como em cada ciclo o valor do registo R4 sofre um deslocamento de 4 bits para a esquerda, o seu valor será zero após 8 iterações e consequentemente a instrução de salto não é realizada, terminando o programa.

Tendo em conta os valores iniciais, a tabela com o conteúdo da memória é:

Endereço	Valor
0x7D0	0x12345678
0x7D4	0x23456780
0x7D8	0x34567800
0x7DC	0x45678000
0x7E0	0x56780000
0x7E4	0x67800000
0x7E8	0x78000000
0x7EC	0x80000000

Exercício 11

Uma possível solução:

```

1          mov      R0, #0
2  ciclo   ldrb      R1, [R4]
3          cmp      R1, #0
4          beq      fim
5          add      R4, R4, #1
6          add      R0, R0, #1
7          b        ciclo
8  fim     ...

```

Exercício 12

a) Fragmento 1: 20; fragmento 2: 200.

b) Fragmento 1: $5 \times N + 2$; fragmento 2: $(1 + 3 \times 10 + 2) \times N = 33 \times N$.

Exercício 13

Uma possível solução:

```

1          mov      R6, #100      ; dimensão das seq.
2          mov      R0, #1
3  proximo  ldr      R1, [R4]      ; extrai elemento de A
4          ldr      R2, [R5]      ; extrai elemento de B
5          cmp      R1, R2        ; compara os elementos extraídos
6          beq      continua      ; continua se forem iguais
7          mov      R0, #0        ; par diferente
8          b        fim
9  continua subs     R6, R6, #1
10         beq      fim
11         add      R4, R4, #4      ; se não chegou ao fim
12         add      R5, R5, #4      ; passar ao próximo elemento
13         b        proximo
14  fim     ...

```

Exercício 14

Uma solução entre outras possíveis:

```

1      mov     R8, #50          ; contador: R8
2      mov     R7, #0           ; resultado a zero
3  ciclo  ldr     R9, [R0]
4          ands   R10, r9, #1
5          beq    prox          ; par
6          add    R7, R7, #1     ; ímpar
7  prox   add    R0, R0, #4      ; próximo endereço
8          subs   R8, R8, #1     ; decrementar contador
9          bne    ciclo
10     ...

```

Exercício 15

a) Parâmetros da sub-rotina:

- R0: endereço-base da sequência
- R1: número de elementos da sequência

```

1  soma    mov     R2, #0        ;coloca contador a 0
2  L1      cmp     R1, #0        ;verifica se chegou ao fim
3          beq     L2           ;terminar
4          ldr     R3, [R0]      ;obter um elemento
5          add     R2, R2, R3    ;acumular
6          add     R0, R0, #4    ;endereço do próximo elemento
7          add     R1, R1, #-1   ;ajustar n° de elementos
8          b       L1           ;repetir (próximo elemento)
9  L2      mov     R0, R2        ;devolver resultado em R0
10         mov     PC, LR

```

b) Parâmetros da sub-rotina:

- R0: endereço-base da sequência
- R1: número de elementos da sequência

```

1  impar   mov     R7, #0        ;coloca contador a 0
2  L21     cmp     R1, #0        ;verifica se chegou ao fim
3          beq     L23          ;terminar
4          ldr     R9, [R0]      ;obter um elemento
5          ands   R10, R9, #1    ;testar se é ímpar
6          beq     L22          ;se par, não contabiliza
7          add     R7, R7, #1     ;contabilizar
8  L22     add     R0, R0, #4     ;endereço do próximo elemento
9          sub     R1, R1, #1     ;ajustar n° de elementos
10         b       L21          ;repetir (próximo elemento)
11  L23     mov     R0, R7        ;devolver resultado em R0
12         mov     PC, LR

```


c) Parâmetros da sub-rotina:

- R0: endereço-base da sequência
- R1: número de elementos da sequência
- R2: valor de limiar

```

1  limiar    mov     R7, #0           ;coloca contador a 0
2  L31      cmp     R1, #0           ;verifica se chegou ao fim
3          beq     L33              ;terminar
4          ldr     R9, [R0]          ;obter um elemento
5          cmp     R9, R2            ;testar se é maior que o limiar
6          blt     L32              ;se menor, não contabiliza
7          add     R7, R7, #1        ;contabilizar
8  L32      add     R0, R0, #4        ;endereço do próximo elemento
9          sub     R1, R1, #1        ;ajustar n° de elementos
10         b       L31              ;repetir (próximo elemento)
11  L33      mov     R0, R7           ;devolver resultado em R0
12         mov     PC, LR

```

d) Parâmetros da sub-rotina:

- R0: endereço-base da sequência 1
- R1: endereço-base da sequência 2
- R2: número de elementos de cada sequência

```

1  iguais    mov     R7, #1           ;coloca resultado a 1
2  L41      cmp     R2, #0           ;verifica se chegou ao fim
3          beq     L43              ;terminar
4          ldr     R9, [R0]          ;obter um elemento
5          ldr     R10, [R1]         ;obter outro elemento
6          cmp     R9, R10           ;testar se é maior que o limiar
7          bne     L42              ;Se nao forem iguais termina
8          add     R0, R0, #4        ;endereço do próximo elemento
9          add     R1, R1, #4
10         sub     R2, R2, #1        ;ajustar n° de elementos
11         b       L41              ;repetir (próximo elemento)
12  L42      mov     R7, #0           ;resultado a 0
13  L43      mov     R0, R7           ;devolver resultado em R0 (1 se iguais)
14         mov     PC, LR

```