Arquitetura e Organização de Computadores

Exercícios - Parte II

António José Araújo

João Canas Ferreira

Bruno Lima

Daniel Granhão

Helder Avelar

Mestrado Integrado em Engenharia Informática e Computação

Novembro de 2019

Conteúdo

1 Linguagem *assembly*

1.1 Exercícios resolvidos

Exercício 1
Para as seguintes expressões aritméticas (números inteiros de 64 bits), especifique um mapeamento de variáveis para registos e o fragmento de código *assembly* ARM que as implementa.

a) `f = g - (f + 5)`

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 31 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 (escolha arbitrária):

`X0`: `f` `X1`: `g`

O fragmento de código que realiza os cálculos desejados é:

```
add     X0, X0, 5               // Calcula f = f + 5
sub     X0, X1, X0              // Calcula f = g - f
```

Após esta sequência de duas instruções, `X0` contém o novo valor associado a `f`. O cálculo da primeira parte da expressão (instrução `add`) pode guardar o resultado intermédio no registo `X0`, porque a segunda instrução estabelece o valor final correto.

b) Possível atribuição de variáveis a registos:

`X0`: `f` `X6`: endereço base de `A`

Como cada elemento de uma sequência de inteiros tem 8 bytes, o elemento de índice 12 da sequência `A` está guardado a partir da posição de memória cujo endereço é dado por:

$$\texttt{endereço base de A} + 12 \times 8$$

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

```
ldur    X0, [X6, #96]           // Carrega valor da posição X6+96
add     X0, X0, #17             // Soma-lhe o valor 17
```

Exercício 2

Assuma as seguintes condições iniciais:

`X0 = 0x00000000BEADFEED` `X1 = 0x00000000DEADFADE`

a) Determine o valor de `X2` após a execução da seguinte sequência de instruções:

```
lsl     X2, X0, 4
orr     X2, X2, X1
```

b) Determine o valor de `X2` após a execução da seguinte sequência de instruções:

```
lsr     X2, X0, 3
and     X2, X2, 0x00000000FFFFFFEF
```

Em binário, os valores iniciais dos registos são:

$$\texttt{X0} = 0\ldots0\ 1011\ 1110\ 1010\ 1101\ 1111\ 1110\ 1110\ 1101_2$$
$$\texttt{X1} = 0\ldots0\ 1101\ 1110\ 1010\ 1101\ 1111\ 1010\ 1101\ 1110_2$$

a) A primeira instrução desloca o valor de `X0` quatro bits para a esquerda. Nos 4 bits menos significativos são introduzidos zeros. O resultado da operação é guardado em `X2`; o registo `X0` fica inalterado. O valor de `X2` depois da execução da primeira instrução é:

$$\texttt{X2} = 0\ldots0\ 1011\ 1110\ 1010\ 1101\ 1111\ 1110\ 1110\ 1101\ 0000_2$$

A instrução `orr` calcula a função ou-inclusivo de cada bit de `X2` com o bit de `X0` situado na mesma posição. O resultado é guardado em `X2`. O resultado da operação `orr` é 1 sempre que pelo menos um dos operandos seja 1. Logo:

$$\texttt{X2} = 0\ldots0\ 1011\ 1111\ 1110\ 1111\ 1111\ 1111\ 1110\ 1101\ 1110_2 = \texttt{0x0000000DFEFFFEDE}$$

b) A instrução `lsr` desloca o valor de `X0` três posições para a direita, introduzindo zeros pela esquerda; os 3 bits menos significativos perdem-se. O valor de `X2` depois da execução da primeira instrução é:

$$\texttt{X2} = 000\ 0\ 0\ldots0\ 0001\ 0111\ 1101\ 0101\ 1011\ 1111\ 1101\ 1101_2$$

A instrução `and` calcula a função e-lógico de cada bit de `X2` com o bit correspondente da constante `0x00000000FFFFFFEF`

$$\texttt{00000000FFFFFFEF}_{16} = 0\ldots0\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 1111_2$$

O resultado é guardado em `X2`. 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 `X2` e pela constante indicada. O valor final de `X2` é:

$$\texttt{X2} = 0\ldots0\ 0001\ 0111\ 1101\ 0101\ 1011\ 1111\ 1100\ 1101_2 = \texttt{0x0000000017D5BFCD}$$

Exercício 3

Assuma as seguintes condições iniciais:

$$\texttt{X0} = \text{FFFF FFFF FFFF FFFF}_{16}$$

$$\texttt{X1} = 0\ldots0\ 0011\ 1111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000_2$$

Determine o valor de `X1` após a execução do fragmento seguinte:

```
        cmp     X0, X1
        bge     ELSE
        b       DONE
ELSE:   add     X1, X1, 2
DONE:   ...
```

A primeira instrução compara os conteúdos de `X0` e `X1` alterando o valor das *flags* (registo NZVC) de acordo com o resultado da comparação. A operação realizada é equivalente a:

```
X0 - X1
```

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 `X0` é maior ou igual a `X1` (a condição `ge` interpreta os valores dos registos como sendo números em complemento para 2). Como neste caso o conteúdo de `X0` é negativo e o conteúdo de `X1` 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 `X1` não foi alterado, o seu valor são 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 `X0` e que devolve no registo `X0` o valor 1 caso seja par, ou 0 no caso contrário.

```
            mov   X10, 0x00000000FF000000
            mov   X1, 3
            mov   X2, 0
loop:       stur  X0, [X10]
            bl    par                // Chama rotina par
            cmp   X0, 0
            beq   step
            add   X2, X2, 1
step:       ldur  X0, [X10]
            add   X0, X0, 1
            adds  X1, X1, -1
            bne   loop
```

a) Considerando que antes da execução do programa o registo `X0` possui o valor 8, indique o conteúdo dos registos `X0` e `X2` após a execução do programa.

b) Tendo em consideração a descrição que foi realizada, implemente a rotina `par`.

a) Em algumas situações é útil preservar o valor do registo `X0` (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 0x00000000FF000000 para armazenar o conteúdo do registo `X0`.

O programa realiza 3 iterações, incrementando em cada uma delas o valor `X0`, que inicialmente é 8. Em cada iteração é determinada a paridade do valor em `X0` invocando a rotina `par`. Caso o valor em `X0` seja par o registo `X2` é incrementado. No final da execução do programa `X2`=2, correspondendo aos 2 números pares encontrados (8 e 10), e `X0`=11, correspondendo ao resultado da adição de uma unidade em cada iteração ao valor de `X0`.

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

```
par:        add   X0, X0, 1
            and   X0, X0, 1
            ret
```

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`→`X0`
- endereço-base de `SEQ`→`X1`
- número de elementos de `SEQ`→`X2`
- resultado→`X0`

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`→`X7`
- número de elementos de `SEQ1`→`X8`
- endereço-base de `SEQ2`→`X9`
- número de elementos de `SEQ2`→`X10`
- resultado→`X5`

a) O fragmento consiste num ciclo em que se varia o registo `X1` 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 `X3` 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 `X3` não é alterado, mantendo o valor inicial estabelecido na primeira instrução.

Na última linha o valor de `X3` é transferido para o registo `X0` de forma a deixar o resultado em `X0` tal como é pedido no enunciado.

```
1          mov     X3, 0      // resultado a Zero
2  prox:   cmp     X2, 0      // terminar?
3          beq     fim
4          ldur    x4, [x1]    // obter elemento
5          cmp     x4, X0      // elemento = N?
6          bne     seg
7          mov     X3, 1      // encontrado, resultado a 1
8          b       fim
9  seg:    add     x1, x1, 8 // atualizar endereço
10         sub     x2, x2, 1 // ajustar numero de elementos
11         b       prox
12 fim:    mov     x0, x3      // colocar resultado em X0
```

b) Para transformar o fragmento numa sub-rotina é necessário alterá-lo para corresponder às convenções de invocação: os argumentos nos registos `X0`–`X7` e o resultado nos registos `X0` e `X1`.

A atribuição de registos passa a ser a seguinte:

- `N`→`X0`
- endereço-base de `SEQ`→`X1`
- número de elementos de `SEQ`→`X2`
- resultado→`X0`

```
1  pesq:   mov     X3, 0       // resultado a Zero
2  L1:     cmp     X2, 0       // terminar?
3          beq     L3
4          ldur    X4, [X1]     // obter elemento
5          cmp     X4, X0       // elemento = N?
6          bne     L2
7          mov     X3, 1       // encontrado, resultado a 1
8          b       L3
9  L2:     add     X1, X1, 8   // atualizar endereço
10         sub     X2, X2, 1   // ajustar numero de elementos
11         b       L1
12 L3:     mov     X0, X3       // colocar resultado em X0
13         ret
```

A ultima instrução da sub-rotina (`ret`), 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 `X0`, endereço-base de `SEQ2` em `X1` e número de elementos de `SEQ2` em `X2`).

A linha 8 processa o resultado da invocação (registo `X0`). Se o valor foi encontrado em `SEQ2`, o contador `X5` é incrementado de uma unidade.

```
1           mov     X5, 0       // inicializar contador
2  ciclo:   cmp     X8, 0       // mais elementos de SEQ1?
3           beq     stop         // terminar
4           ldur    X0, [X7]     // obter um elemento de SEQ1
5           mov     X1, X9       // onde pesquisar
6           mov     X2, X10      // nº de elementos a pesquisar
7           bl      pesq         // invocar sub-rotina
8           add     X5, X5, X0   // atualizar contador
9           add     X7, X7, 8   // próximo endereço
10          sub     X8, X8, 1   // decrementar nº de elementos
11          b       ciclo        // repetir
12 stop:    ...
```

1.2 Exercícios propostos

Exercício 6

Para as seguintes expressões aritméticas (números inteiros de 64 bits), especifique um mapeamento de variáveis para registos e o fragmento de código *assembly* ARMv8 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* ARMv8, indique um mapeamento entre registos e variáveis e determine as expressões simbólicas correspondentes.

a)
```
add     x0,x0,x1
add     x0,x0,x2
add     x0,x0,x3
add     x0,x0,x4
```

b)
```
ldur    x0,[x6, 8]
```

c) Assumir que `X6` contém o endereço-base da sequência `A[]`.
```
add     x6, x6, -40
lsl     x10, x1, 3
add     x6, x6, x10
ldur    x0, [x6, 16]
```

Exercício 8

Assuma as seguintes condições iniciais:

```
X0 = 0x5555555555555555          X1 = 0x0123456789ABCDEF
```

Determine o valor de `X2` após a execução das sequências de instruções seguintes.

a)
```
lsl     x2, x0, 4
orr     x2, x2, x1
```

b)
```
lsl     x2, x0, 4
and     x2, x2, -1
```

c)
```
lsr     x2, x0, 3
and     x2, x2, 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  X2, X1` é equivalente a `X2 ← |X1|`

Apresente uma sequência de instruções ARMv8 que realiza esta operação.

b) Repita a alínea anterior para a instrução `sgt`, em que `sgt  X1, X2, X3` é equivalente a `se X2 > X3 então X1 ← 1 senão X1 ← 0`.

Exercício 10

Considere o seguinte fragmento de código *assembly* ARMv8:

```
L1:     stur  X4,[X5]
        lsl   X4,X4,4
        add   X5,X5,8
        cmp   X4, 0
        b.ne  L1
```

Assuma os seguintes valores iniciais:

```
X4 = 0x12345678          X5 = 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 `X4`) está uma sequência de números inteiros (de 64 bits) diferentes de 0. A sequência é terminada por um zero.

Escreva um fragmento de código *assembly* que determina o número de valores da sequência. O valor final, 0, não entra para a contagem. O resultado deve ser guardado em `X0`.

Exercício 12

Considerar os seguintes fragmentos de código *assembly*.

Fragmento 1:
```
LOOP:   cmp     X1, 0
        b.eq    DONE
        add     X2, X2, 2
        add     X1, X1, -1
        b       LOOP
DONE:           ...
```

Fragmento 2:
```
LOOP:   mov     X3, 0xA
LOOP2:  add     X2, X2, 2
        adds    X3, X3, -1
        b.ne    LOOP2
        adds    X1, X1, -1
        b.ne    LOOP
DONE:   ...
```

a) Assumir a seguinte situação inicial: `X1=10` e `X2=0`. Determinar, para cada fragmento, o valor final de `X2`.

b) Assuma agora que `X1=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 (64 bits) têm o mesmo conteúdo. As sequências têm 100 elementos. Usar a seguinte atribuição de registos:

`endereço-base da sequência A` → `X4`
`endereço-base da sequência B` → `X5`

O resultado, a guardar em `X0`, deve ser 1, se as duas sequências tiverem o mesmo conteúdo ou 0 no caso contrário.

Exercício 14

Escreva um fragmento de código *assembly* ARMv8 para determinar quantos números ímpares estão contidos numa sequência de 50 números inteiros (de 64 bits). Assuma que o endereço-base da sequência está contido no registo `X0`. O resultado deve ficar no registo `X7`.

Exercício 15

Pretende-se escrever um programa que permita realizar diversas tarefas envolvendo sequências de números inteiros (64 bits) 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.

Escreva um 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 as convenções de chamada e de 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 registos.

a) Atribuição: `f` → `x0`, `g` → `x1`, `j` → `x2`.

```
add     x0, x2, 2
add     x0, x1, x0
```

b) Atribuição: `a` → `x0`, `b` → `x1`, `d` → `x2`, `f` → `x3`, `k` → `x4`.

```
add     x4, x0, x1
sub     x4, x4, x3
add     x4, x4, x2
add     x4, x4, -30
```

c) Atribuição: `f` → `x0`, `g` → `x1`, `h` → `x2`, `A` → `x7`.

```
ldur    x0, [x7, 32]
add     x0, x0, x1
add     x0, x0, x2
```

d) Atribuição: `f` → `x0`, `g` → `x2`, `A` → `x6`, `B` → `x7`.

```
ldur    x5, [x7, 80]
lsl     x5, x5, 3
add     x5, x5, x6
ldur    x0, [x5]
sub     x0, x2, x0
```

e) Atribuição: `f` → `x0`, `g` → `x1`, `h` → `x2`, `k` → `x3`, `A` → `x6`.

```
add     x10, x2, 9
lsl     x10, x10, 3
add     x10, x6, x10
ldur    x0, [x10]
add     x0, x0, x3
sub     x0, x0, x1
```

f) Atribuição: `f` → `x0`, `g` → `x1`, `A` → `x6`, `B` → `x7`.

```
ldur    x10, [x7, 16]
add     x10, x10, 4
lsl     x10, x10, 3
add     x10, x6, x10
ldur    x10, [x10]
sub     x0, x1, x10
```

Exercício 7

a) Atribuição: `f` → `x0`, `g` → `x1`, `h` → `x2`, `i` → `x3`, `j` → `x4`

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

b) Atribuição: `f` → `x0`, `A` → `x6`

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

c) Atribuição: `f` → `x0`, `g` → `x1`, `A` → `x6`

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

Exercício 8

a) `0x55775577DDFFDDFF`

b) `0x5555555555555550`

c) `0x00000000000000AA`

Exercício 9

a)
```
        mov     X2, X1
        cmp     X1, XZR
        b.ge    pos
        sub     X2, XZR, X2
pos:    ...
```

Ou, tirando proveito da instrução CSNEG:

```
        cmp     X1, 0
        csneg   X2, X1, X1, ge
```

b)
```
        mov     X1, 0
        cmp     X2, X3
        b.le    pos
        mov     X1, 1
pos:    ...
```

Ou, tirando proveito da instrução CSINC:

```
        cmp     X2, X3
        csinc   X1, XZR, XZR, le
```

Exercício 10

A cada iteração do ciclo `L1` será guardado em memória (no endereço dado pelo conteúdo do registo `X5`) o valor do registo `X4`. Como em cada ciclo o valor do registo `X4` sofre um deslocamento de 4 bits para a esquerda, o seu valor será zero após 16 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	0x0000000012345678
0x7D8	0x0000000123456780
0x7E0	0x0000001234567800
0x7E8	0x0000012345678000
0x7F0	0x0000123456780000
0x7F8	0x0001234567800000
0x800	0x0012345678000000
0x808	0x0123456780000000
0x810	0x1234567800000000
0x818	0x2345678000000000
0x820	0x3456780000000000
0x828	0x4567800000000000
0x830	0x5678000000000000
0x838	0x6780000000000000
0x840	0x7800000000000000
0x848	0x8000000000000000

Exercício 11

Uma possível solução:

```
1          mov       x0, 0
2  ciclo:  ldur      x1, [x4]
3          cmp       x1, 0
4          b.eq      fim
5          add       x4, x4, 8
6          add       x0, x0, 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   X6, 100      // dimensão das seq.
2              mov   X0, 1
3  proximo:    ldur  X1,[X4]      // extrai elemento de A
4              ldur  X2,[X5]      // extrai elemento de B
5              cmp   X1,X2        // compara os elementos extraidos
6              b.eq  continua     // continua se forem iguais
7              mov   X0,0         // termina se forem diferentes
8              b     fim
9  continua:   subs  X6,X6, 1
10             b.eq  fim
11             add   X4,X4, 8     // se não chegou ao fim
12             add   X5,X5, 8     // passar ao próximo elemento
13             b     proximo
14 fim:        ...
```

Exercício 14

Uma solução entre outras possíveis:

```
1          mov    X8, 50          // contador: X8
2          mov    X7, 0           // resultado a zero
3  ciclo:  ldur   X9, [X0]
4          ands   X10, X9, 1
5          b.eq   prox            // par
6          add    X7, X7, 1       // ímpar
7  prox:   add    X0, X0, 8       // próximo endereço
8          subs   X8, X8, 1       // decrementar contador
9          b.ne   ciclo
10         ...
```

Exercício 15

a) Parâmetros da sub-rotina:

- `X0`: endereço-base da sequência
- `X1`: número de elementos da sequência

```
1  soma:   mov   X2, 0        // coloca contador a 0
2  L1:     cmp   X1, 0        // verifica se chegou ao fim
3          b.eq  L2           // terminar
4          ldur  X3, [X0]     // obter um elemento
5          add   X2, X2, X3   // acumular
6          add   X0, X0, 8    // endereço do próximo elemento
7          add   X1, X1, -1   // ajustar nº de elementos
8          b     L1           // repetir (próximo elemento)
9  L2:     mov   X0, X2       // devolver resultado em X0
10         ret
```

b) Parâmetros da sub-rotina:

- `X0`: endereço-base da sequência

- `X1`: número de elementos da sequência

```
1  impar:  mov    X7, 0           // coloca contador a 0
2  L21:    cmp    X1, 0           // verifica se chegou ao fim
3          b.eq   L23             // terminar
4          ldur   X9, [X0]        // obter um elemento
5          ands   X10, X9, 1      // testar se é impar
6          beq    L22             // se par, não contabiliza
7          add    X7, X7, 1       // contabilizar
8  L22:    add    X0, X0, 8       // endereço do próximo elemento
9          sub    X1, X1, 1       // ajustar nº de elementos
10         b      L21             // repetir (próximo elemento)
11 L23:    mov    X0, X7          // devolver resultado em X0
12         ret
```

c) Parâmetros da sub-rotina:

- `X0`: endereço-base da sequência
- `X1`: número de elementos da sequência
- `X2`: valor de limiar

```
1  limiar: mov    X7, 0           // coloca contador a 0
2  L31:    cmp    X1, 0           // verifica se chegou ao fim
3          b.eq   L33             // terminar
4          ldur   X9, [X0]        // obter um elemento
5          cmp    X9, X2          // testar se é maior que o limiar
6          b.lt   L32             // se menor, não contabiliza
7          add    X7, X7, 1       // contabilizar
8  L32:    add    X0, X0, 8       // endereço do próximo elemento
9          sub    X1, X1, 1       // ajustar nº de elementos
10         b      L31             // repetir (próximo elemento)
11 L33:    mov    X0, X7          // devolver resultado em X0
12         ret
```

d) Parâmetros da sub-rotina:

- `X0`: endereço-base da sequência 1
- `X1`: endereço-base da sequência 2
- `X2`: número de elementos de cada sequência

```
1  iguais: mov   X7, 1           // coloca resultado a 1
2  L41:    cmp   X2, 0           // verifica se chegou ao fim
3          b.eq  L43             // terminar
4          ldur  X9, [X0]        // obter um elemento de cada sequência
5          ldur  X10,[X1]
6          cmp   X9, X10
7          b.ne  L42             // Se nao forem iguais termina
8          add   X0, X0, 8       // endereço do próximo elemento
9          add   X1, X1, 8
10         sub   X2, X2, 1       // ajustar nº de elementos
11         b     L41             // repetir (próximo elemento)
12 L42:    mov   X7, 0           // resultado a 0
13 L43:    mov   X0, X7          // devolver resultado em X0 (1 se iguais)
14         ret
```