

ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO



Experiência 5

Relatório

Turma: 02

2/05/2023

Nome	Número USP
Natanael Magalhães Cardoso	8914122
Tomas Gorescu Caldeira	9300881

5.5.1 - For Loop

*Code the following C code in assembly. The arrays *a* and *b* are located in memory at 0x4000 and 0x5000 respectively. You may wish to type your code into the assembler to check for syntax.*

```
for (i=0; i<8; i++) {  
    a[i] = b[7-i];  
}
```

A listagem abaixo mostra a implementação do algoritmo

```
@ Descrição do algoritmo  
@ -----  
@ Implementação da um loop for que itera sobre  
@ um vetor e armazena seu valor em outro vetor  
@ de forma inversa. Equivalente ao algoritmo:  
@  
@ for (i=0; i<8; i++) { a[i] = b[7-i]; }  
@  
@  
@ Lista de Registradores  
@ -----  
@ r1 : posição base do vetor a  
@ r2 : posição base do vetor b  
@ r3 : contador  
@ r4 : tamanho da sequência  
@ r5 : registrador de trabalho  
@  
@  
@ Instruções de uso  
@ -----  
@ arm build ex_5_5_1.s (montagem)  
@ arm debug (depuração)  
  
.text  
.global main
```

```

main:
    LDR    r1, =a        @ r1 = a
    LDR    r2, =b        @ r2 = b
    MOV    r3, #0        @ i = 0
    MOV    r4, #5        @ r4 = 8 (tamanho da sequência)

loop:
    CMP    r3, #10       @ compara r3 com 10
    BGT    fim           @ encerra loop se i > 10
    RSB    r5, r3, #7     @ r5 = 7 - i
    LDR    r5, [r2, r5, LSL #2] @ r5 = b[7-i]
    STR    r5, [r1, r3, LSL #2] @ a[i] = r5 = b[7-i]
    ADD    r3, r3, #1
    B      loop

fim:
    SWI    0x123456

.data
a:
    .word 0, 0, 0, 0, 0, 0, 0, 0
b:
    .word 0, 1, 2, 3, 4, 5, 6, 7

.space 100 @ abrindo um espaço de 100 bytes para armazenar a array
.align 1   @ align to even bytes REQUIRED!!!

```

```

tomas@tomas-Latitude-3420: ~/LabDig/E5
Register group: general
r0      0x1      1
r1      0x30028  196648
r2      0x30048  196680
r3      0xb      11
r4      0x5      5
r5      0x2      2
r6      0x2ff14  196372
r7      0x3ffff048  1073737800
r8      0x103ec  66540
r9      0x3fffed78  1073737080

14      STR      r5, [r1, r3, LSL #2]    @ a[i] = r5 = b[7-i]
15      ADD      r3, r3, #1
16      B        loop
17  fim:
B+> 18      SWI      0x123456
19
20      .data
21  a:
22      .word 0, 0, 0, 0, 0, 0, 0, 0
23  b:
24      .word 0, 1, 2, 3, 4, 5, 6, 7

remote Thread 1.6179 In: fim                                L18    PC: 0x10418
(gdb) b fim
Breakpoint 1 at 0x10418: file item-5.5.1.s, line 18.
(gdb) c
Continuing.

Breakpoint 1, fim () at item-5.5.1.s:18
(gdb) x/8d &a
0x30028:      7      6      5      4
0x30038:      3      2      1      0
(gdb) x/8d &b
0x30048:      0      1      2      3
0x30058:      4      5      6      7
(gdb)

```

5.5.2 Factorial Calculation

To take advantage of the idea of conditional execution, let's examine the algorithm for computing $n!$, where n is an integer, defined as:

For a given value of n , the algorithm iteratively multiplies a current product by a number one less than the number it used in the previous multiplication. The code would continue to loop until it is no longer necessary to perform a multiplication, first by subtracting one from the next multiplier value and stopping if it is equal to zero. Here we can use the concepts of:

- conditional execution to conditionally perform the multiplication
- saving of the current product into a temporary register

- *branch back to the start of the loop.*

In writing routines that have loops and branches, many programmers start with a nonzero value and count down, rather than up, because you can use the Z flag to quickly determine whether the loop count has been exhausted.

Fill in the blanks in the following code segment. Then run the code on the evaluation board by including the necessary header information and compiler directives. Using a starting value of 10 for n, demonstrate the result to your lab instructor and print out the register bank before and after the program runs.

```
factorial MOV r6,#0xA ; load 10 into r6
MOV r4,r6 ; copy n into a temp register
loop SUBS _____ ; decrement next multiplier
MULNE _____ ; perform multiply
MOVNE _____ ; save off product for another loop
BNE loop ; go again if not complete
```

Nesta implementação optamos por não usar a instrução MOVNE, pois, como não havia nenhuma instrução acerca de manter o valor de r6 inalterado, fomos armazenando o resultado do fatorial no próprio r6.

Nesse caso, a execução condicional permite não ter um branch direto para o “fim” na condição de parada.

```
@ Descrição do algoritmo
@ -----
@ Algoritmo calcula fatorial de um número (n!)
@
@
@ Lista de Registradores
@ -----
@ r6 : valor de n; ao final da execução, ele
@ armazena o valor de n!
@ r4 : registrador de trabalho
@
@
@ Instruções de uso
@ -----
@ arm build ex_5_5_2.s (montagem)
@ arm debug (depuração)

.text
```

```
.global main

main:
factorial:
    MOV r6, #0xA      @ load 10 into r6
    MOV r4, r6        @ copy n into a temp register

loop:
    SUBS r4, r4, #1    @ decrement next multiplier
    MULNE r6, r6, r4   @ perform multiply
    BNE  loop         @ go again if not complete

fim:
    SWI 0x123456
```

```
tomas@tomas-Latitude-3420: ~/LabDig/E5
Register group: general
r0      0x1      1
r1      0x40800054 1082130516
r2      0x4080005c 1082130524
r3      0x103ec  66540
r4      0x0      0
r5      0x1      1
r6      0x375f00  3628800
r7      0x3ffff048 1073737800
r8      0x103ec  66540
r9      0x3fffed78 1073737080

    9      SUBS r4, r4, #1      @ decrement next multiplier
   10      MULNE r6, r6, r4    @ perform multiply
   11      BNE loop           @ go again if not complete
   12  fim:
B+>  13      SWI 0x123456
   14
   15
   16
   17
   18
   19

remote Thread 1.7184 In: fim          L13  PC: 0x10400
(gdb) b fim
Breakpoint 1 at 0x10400: file item-5.5.2.s, line 13.
(gdb) c
Continuing.

Breakpoint 1, fim () at item-5.5.2.s:13
(gdb) p/d $r6
$1 = 3628800
(gdb) 
```

5.5.3 Find maximum value

In this exercise, you are to find the largest integer in a series of 32-bit unsigned integers. The length of the series is determined by the value in register r5. The maximum value is stored in the memory location 0x5000 at the end of the routine. The data values begin at memory location 0x5006. Choose 11 or more integers to use. Use as much conditional

execution as possible when writing the code. Demonstrate the program to your lab instructor and print out the memory space starting at 0x5000 before and after the program runs. Be sure to include enough memory space to show all of your 32-bit integer values.

Tratamos esse problema como um quase idêntico aos de iterações sobre as posições de um *array*, mas com o diferencial que, ao ler uma nova posição, comparamos seu valor com o do registrador que contém o máximo, nesse caso o r4, e se a posição lida for maior que r4, mover esse valor para r4.

```
@ Descrição do algoritmo
@ -----
@ Encontra o máximo valor de um array de inteiros
@ 32-bits sem sinal e armazena o valor encontrado
@ no registrador r4
@
@
@ Lista de Registradores
@ -----
@ r1 : endereço base do vetor a
@ r2 : contador
@ r4 : máximo valor encontrado
@ r5 : tamanho do vetor a
@
@
@ Instruções de uso
@ -----
@ arm build ex_5_5_3.s (montagem)
@ arm debug (depuração)
```

```
.text
```

```
.global main
```

```
main:
```

```
    LDR r1, =a      @ r1 = a
    MOV r5, #8      @ r5 = 8 (tamanho da sequência)
    MOV r2, #0      @ r2 = 0
    LDR r4, [r1]    @ r4 = a[0]
```



```

loop:
    CMP    r2, #10           @ compara r3 com 10
    BEQ    fim               @ sai do loop se contador = 10
    LDR    r3, [r1, r2, LSL #2] @ r3 recebe a[i]
    CMP    r3, r4            @ compara a[i] com o maior valor atual
    MOVGT  r4, r3            @ r4 = a[i], se r4 > a[i]
    ADD    r2, r2, #1        @ incrementa contador
    B      loop              @ retorna ao loop

fim:
    SWI    0x123456          @ termina execução
    .data
a:
    .word 5, 2, 3, 1, 8, 4, 2, 15

.space 100 @ abrindo um espaço de 100 bytes para armazenar a array
.align 1   @ align to even bytes REQUIRED!!!

```

```

tomas@tomas-Latitude-3420: ~/LabDig/E5
Register group: general
r0      0x1      1
r1      0x30028  196648
r2      0xa      10
r3      0x0      0
r4      0xf      15
r5      0x8      8
r6      0x2ff14  196372
r7      0x3ffff048  1073737800
r8      0x103ec  66540
r9      0x3fffed78  1073737080

item-5.5.3.s
8      LDR      r4, [r1]          @ r4 = a[0]
9  loop:
10     CMP      r2, #10          @ compara r2 com 10
11     BEQ      fim
12     LDR      r3, [r1, r2, LSL #2] @ r3 recebe a[i]
13     CMP      r3, r4
14     MOVGT    r4, r3
15     ADD      r2, r2, #1
16     B        loop
17 fim:
B+> 18     SWI      0x123456

remote Thread 1.8773 In: fim L18 PC: 0x10418
(gdb) b fim
Breakpoint 1 at 0x10418: file item-5.5.3.s, line 18.
(gdb) c
Continuing.

Breakpoint 1, fim () at item-5.5.3.s:18
(gdb) x/8d &a
0x30028:      5      2      3      1
0x30038:      8      4      2      15
(gdb) p/d $r4
$1 = 15
(gdb)

```

5.5.4 Finite state machines: a nonresetting sequence recognizer

1. Consider an FSM with one input X and one output Z . The FSM asserts its output Z when it recognizes an input bit sequence of $b1011$. The machine keeps checking for the sequence and does not reset when it recognizes the sequence.

Here is an example input string X and its output Z :

$X = \dots 0010110110\dots$

$Z = \dots 0000010010 \dots$

Write ARM assembly to implement the sequence recognizer. Start with the initial input X in $r1$. Finish with the output Z in $r2$ at the end of the program.

2. Now write the code to recognize any sequence Y up to 32 bits. Start with the recognizing sequence Y in $r8$ and the size of Y in $r9$. For example, to recognize the sequence $Y = b0110110$, then $r8 = 0x36$ and $r9 = 0x7$ before program execution. Everything else should be the same as in Step 1. Make sure that your program works for every case, including the case when $r9 = 1$ or $r9 = 32$.

A ideia inicial para esta implementação era a de emular uma máquina de Mealy, com condições específicas para cada estado. Depois de começar a esboçar a máquina, percebemos que o requisito de que não “resetar” após reconhecer a sequência torna a resolução por esse caminho quase que “inviável”.

Para esse problema, basta isolar os fragmentos de tamanho igual a sequência a ser reconhecida, comparando-os.

```
@ Descrição do algoritmo
@ -----
@ Reconhece uma padrão genérico em uma sequência
@ de bits
@
@
@ Lista de Registradores
@ -----
@ r1 : entrada X
@ r2 : saída - posições onde o padrão foi encontrado
@ r3 : contador
@ r8 : padrão a ser detectado
@ r9 : número de bits do padrão
@
@
@ Instruções de uso
@ -----
@ arm build ex_5_5_4.s (montagem)
@ arm debug (depuração)
```

.text

```

.global main

main:
    LDR    r1, =0b0010110110 @ r1 = x
    LDR    r2, =0             @ r2 = z
    MOV    r3, #0             @ i = 0
    LDR    r8, =0x36          @ padrão a ser reconhecida
    LDR    r9, =0x7           @ tamanho em bits do padrão
    MOV    r4, #10            @ r4 = tamanho da sequência
    SUB    r7, r4, r9         @ r7 = tamanho sequência- tamanho do
reconhecido
    RSB    r5, r4, #32        @ r5 = 32 - tamanho da sequência
    RSB    r10, r9, #32       @ tamanho da palavra - tamanho da sequência
a ser reconhecida

loop:
    CMP    r3, r7             @ compara r3 com r4- condição de parada
    BGT    fim                @ encerra loop se maior
    ADD    r6, r5, r3         @ r6 = tamanho da palavra - tamanho da
sequência + i
    MOV    r6, r1, LSL r6     @ r6 = trecho de 4 bits a ser analisado
deslocado p/ esquerda
    MOV    r6, r6, LSR r10    @ desloca para direita para isolar segmento
de 4 bits

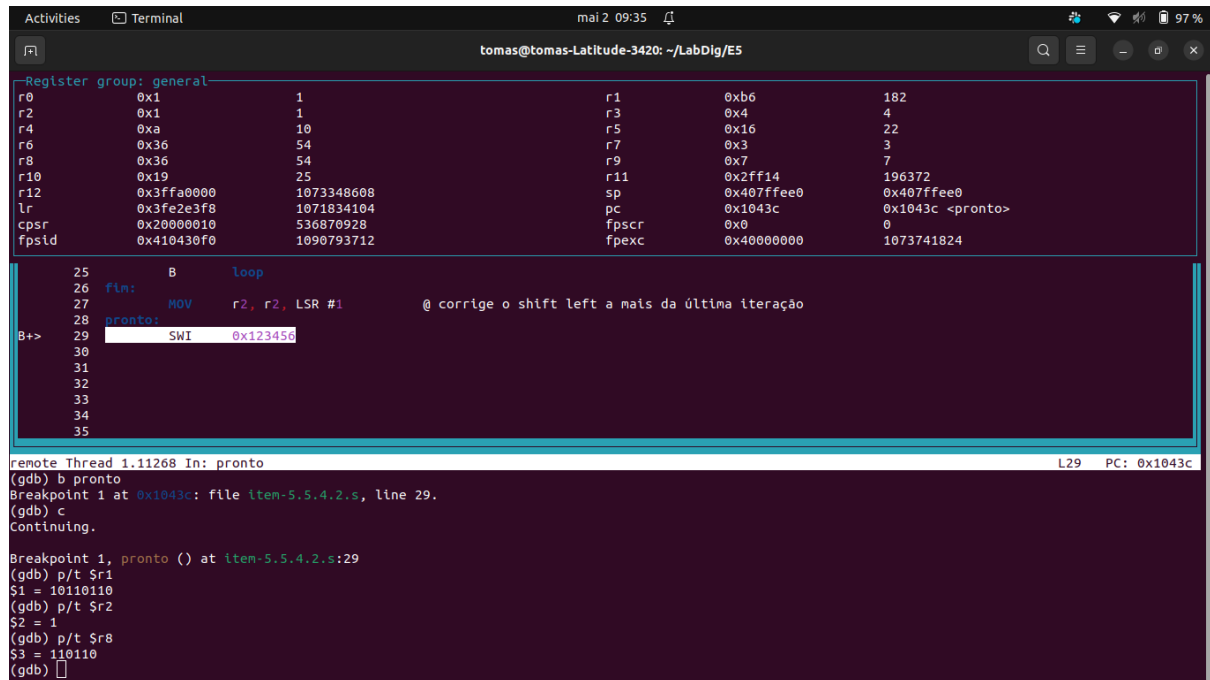
setado:
    @ label para ajudar a imprimir os valores
    CMP    r6, r8             @ compara r6 com sequência fornecida
    ADDEQ  r2, r2, #1         @ 1 em r2 se for igual
    MOV    r2, r2, LSL #1     @ também passa r2 para próximo bit
    ADD    r3, r3, #1         @ incrementa contador
    B loop

fim:
    MOV    r2, r2, LSR #1     @ corrige o shift left a mais da última
iteração

```

```
pronto:
    SWI    0x123456
```

A figura abaixo mostra a depuração do programa usando as entradas indicadas na apostila:



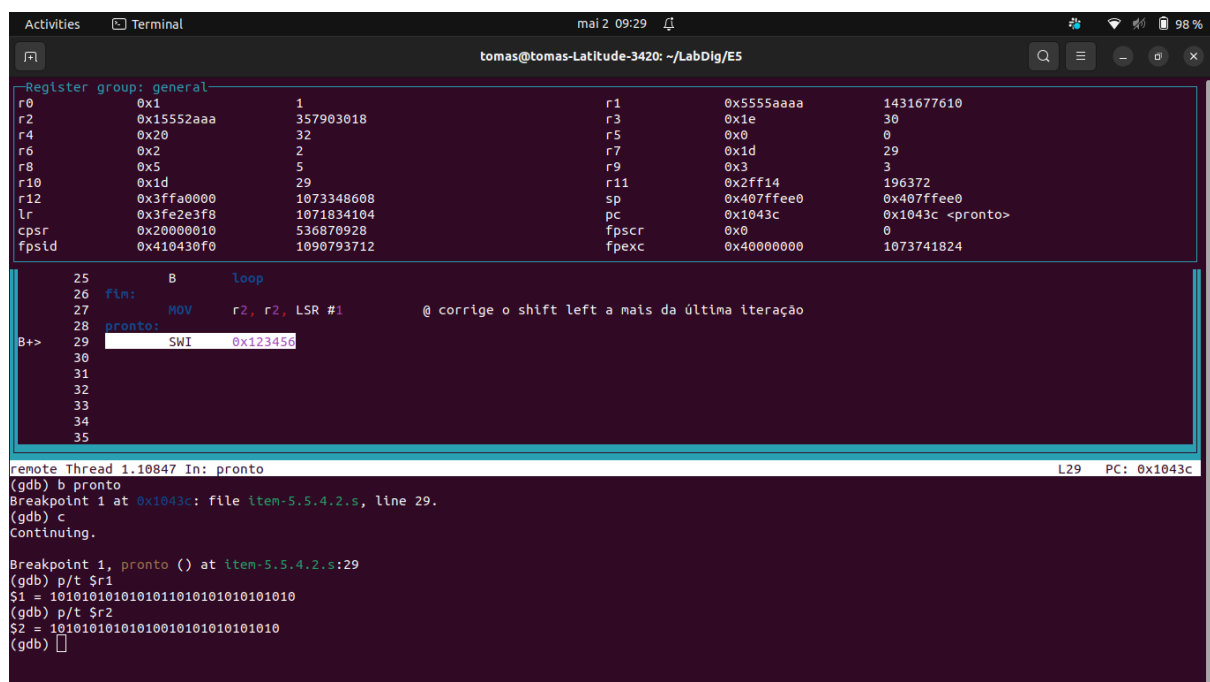
```
Register group: general
r0      0x1      1      r1      0xb6      182
r2      0x1      1      r3      0x4       4
r4      0xa      10     r5      0x16      22
r6      0x36     54     r7      0x3       3
r8      0x36     54     r9      0x7       7
r10     0x19     25     r11     0x2ff14    196372
r12     0x3ffa0000 1073348608 sp      0x407ffee0    0x407ffee0
lr      0x3fe2e3f8 1071834104 pc      0x1043c    0x1043c <pronto>
cpsr    0x20000010 536870928  fpscr   0x0       0
fpsid   0x410430f0 1090793712 fpexc   0x40000000 1073741824

25      B      loop
26      fn:
27      MOV     r2, r2, LSR #1      @ corrige o shift left a mais da última iteração
28      pronto:
29      SWI     0x123456
30
31
32
33
34
35

remote Thread 1.11268 In: pronto
(gdb) b pronto
Breakpoint 1 at 0x1043c: file item-5.5.4.2.s, line 29.
(gdb) c
Continuing.

Breakpoint 1, pronto () at item-5.5.4.2.s:29
(gdb) p/t $r1
$1 = 10110110
(gdb) p/t $r2
$2 = 1
(gdb) p/t $r8
$3 = 110110
(gdb) 
```

A figura abaixo mostra a depuração do programa usando as entradas indicados no roteiro do experimento (moodle):



```
Register group: general
r0      0x1      1      r1      0x5555aaaa 1431677610
r2      0x15552aaa 357903018  r3      0x1e      30
r4      0x20      32     r5      0x0       0
r6      0x2       2      r7      0x1d      29
r8      0x5       5      r9      0x3       3
r10     0x1d      29     r11     0x2ff14    196372
r12     0x3ffa0000 1073348608 sp      0x407ffee0    0x407ffee0
lr      0x3fe2e3f8 1071834104 pc      0x1043c    0x1043c <pronto>
cpsr    0x20000010 536870928  fpscr   0x0       0
fpsid   0x410430f0 1090793712 fpexc   0x40000000 1073741824

25      B      loop
26      fn:
27      MOV     r2, r2, LSR #1      @ corrige o shift left a mais da última iteração
28      pronto:
29      SWI     0x123456
30
31
32
33
34
35

remote Thread 1.10847 In: pronto
(gdb) b pronto
Breakpoint 1 at 0x1043c: file item-5.5.4.2.s, line 29.
(gdb) c
Continuing.

Breakpoint 1, pronto () at item-5.5.4.2.s:29
(gdb) p/t $r1
$1 = 10101010101010101010101010101010
(gdb) p/t $r2
$2 = 10101010101001010101010101010101
(gdb) 
```