

# ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO



## ***Experiência 2***

Relatório

Turma: 02

*28/03/2023*

<b>Nome</b>	<b>Número USP</b>
Natanael Magalhães Cardoso	8914122
Tomas Gorescu Caldeira	9300881

## Tabela de Conteúdo

<b>1. Comparação entre ADD e ADDS</b>	<b>3</b>
1.1. Instrução ADD	3
1.2. Instrução ADDS	5
<b>2. Exercícios do capítulo 2</b>	<b>6</b>
2.1. Exercício 2.4.1	6
2.2. Exercício 2.4.2	7
2.3. Exercício 2.4.2 - Diferentes formas de visualizar dados	10
<b>3. Soma com sinal e sem sinal</b>	<b>11</b>
<b>4. Multiplicação</b>	<b>15</b>
4.1. Multiplicação com MULS	15
4.2. Multiplicação com UMULLS e SMULLS	18
<b>5. Multiplicação pelo número 32</b>	<b>21</b>
<b>6. Register-Swap Algorithm</b>	<b>23</b>
<b>7. Conclusão</b>	<b>25</b>

# 1. Comparação entre ADD e ADDS

## 1.1. Instrução ADD

A instrução ADD da arquitetura ARM foi testada com o programa descrito na Listagem 1. O programa foi montado e depurado com sucesso. As Fig. 1 e Fig. 2 mostram uma comparação do estado dos registradores antes e depois da execução da instrução ADD, respectivamente. É notado que, após a execução do comando, o registrador r0 possui valor 35, que é a soma do valor anterior de r0 (15) com o valor do registrador r1 (20). Também é observado que não há modificações no registrador CPSR, que continua com o mesmo valor após a execução da instrução ADD.

Listagem 1: Programa usado para testar a instrução ADD do ARM

```
@item-2-2.s
.text
.globl main
main:
    MOV r0, #15      @ carrega valor no primeiro registrador
    MOV r1, #20      @ carrega valor no segundo registrador
    BL firstfunc     @ desvia para funcao, coloca o enderenco
                    @ de retorno em R14 ou LR (link register).
    MOV r0, #0x0     @ zera registrador
    MOV r7, #0x1     @ exit(0)
    SWI 0x0          @ system call
firstfunc:
    ADD r0, r0, r1   @ calcula soma r0 = r0 + r1
    MOV pc, lr       @ retorna da funcao
```

```

natan@asus: ~/repos/poli/lab-processadores/exp02
Register group: general
r0      0xf      15      r1      0x14      20
r2      0x407ffe5c 1082130012 r3      0x103ec 66540
r4      0x3ff90000 1073283072 r5      0x1      1
r6      0x2ff14   196372 r7      0x3ffff048 1073737800
r8      0x103ec   66540   r9      0x3fffd78 1073737080
r10     0x0      0       r11     0x2ff14   196372
r12     0x3ff90000 1073283072 sp      0x407ffce0 0x407ffce0
lr      0x103f8   66552   pc      0x10404 0x10404 <firstfunc>
cpsr    0x60000010 1610612752 fpscr   0x0      0

item-2-2.s
10  MOV r7, #0x1 @ exit(0)
11  SWI 0x0 @ system call
12  firstfunc:
> 13  ADD r0, r0, r1 @ calcula soma r0 = r0 + r1
14  MOV pc, lr @ retorna da funcao
15
16
17
18

remote Thread 1.181887 In: firstfunc L13 PC: 0x10404
(gdb) b main
Ponto de parada 1 at 0x103ec: file item-2-2.s, line 5.
(gdb) c
Continuing.

Breakpoint 1, main () at item-2-2.s:5
(gdb) s
(gdb) s
(gdb) s
firstfunc () at item-2-2.s:13
(gdb)

```

Figure 1: Depuração do programa da Listagem 1 mostrando os valores dos registradores imediatamente antes da instrução ADD.

```

natan@asus: ~/repos/poli/lab-processadores/exp02
Register group: general
r0      0x23      35      r1      0x14      20
r2      0x407ffe5c 1082130012 r3      0x103ec 66540
r4      0x3ff90000 1073283072 r5      0x1      1
r6      0x2ff14   196372 r7      0x3ffff048 1073737800
r8      0x103ec   66540   r9      0x3fffd78 1073737080
r10     0x0      0       r11     0x2ff14   196372
r12     0x3ff90000 1073283072 sp      0x407ffce0 0x407ffce0
lr      0x103f8   66552   pc      0x10408 0x10408 <firstfunc+
cpsr    0x60000010 1610612752 fpscr   0x0      0

item-2-2.s
10  MOV r7, #0x1 @ exit(0)
11  SWI 0x0 @ system call
12  firstfunc:
> 13  ADD r0, r0, r1 @ calcula soma r0 = r0 + r1
14  MOV pc, lr @ retorna da funcao
15
16
17
18

remote Thread 1.181887 In: firstfunc L14 PC: 0x10408
(gdb) b main
Ponto de parada 1 at 0x103ec: file item-2-2.s, line 5.
(gdb) c
Continuing.

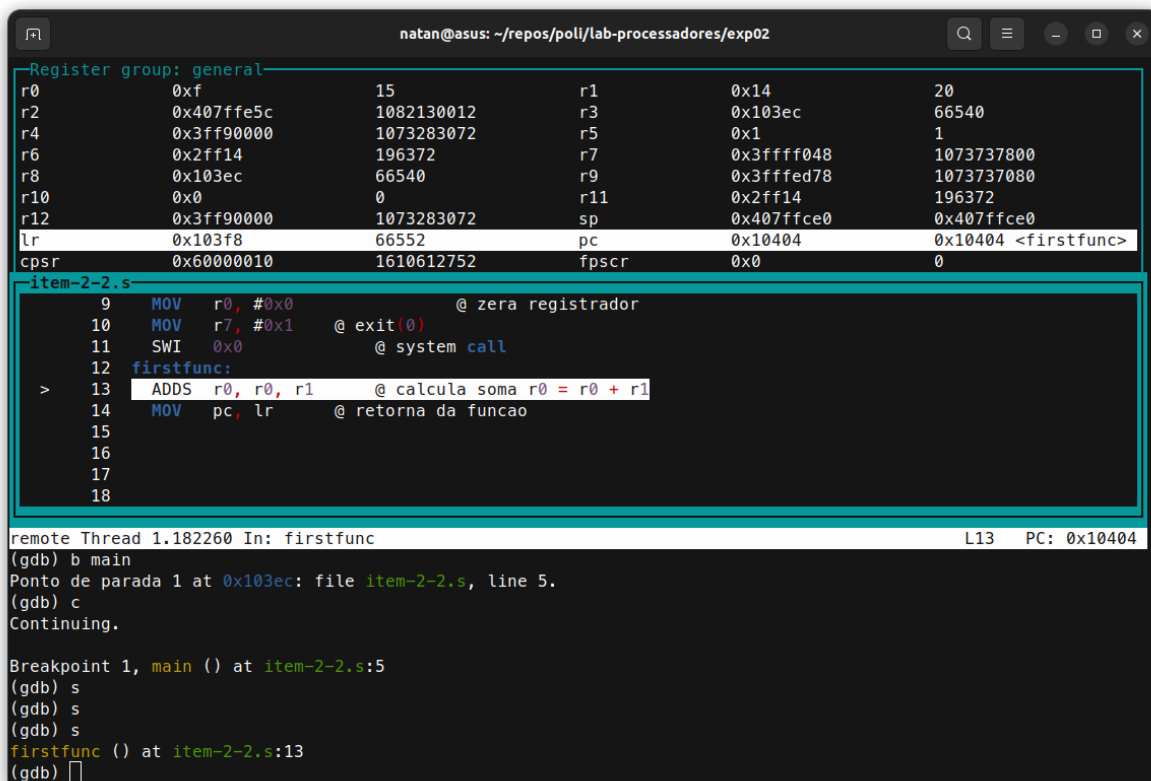
Breakpoint 1, main () at item-2-2.s:5
(gdb) s
(gdb) s
(gdb) s
firstfunc () at item-2-2.s:13
(gdb) s
(reverse-i-search)`':

```

Figure 2: Depuração do programa da Listagem 1 mostrando os valores dos registradores logo após a instrução ADD.

## 1.2. Instrução ADDS

O programa da Listagem 1 foi modificado, com a alteração da instrução ADD para ADDS, para comparação da diferença de comportamento entre as duas instruções. Isso foi feito a partir da depuração e análise dos registradores, em especial o CPSR. As Fig. 3 e Fig. 4 mostram o estado dos registradores imediatamente antes e logo após a execução da instrução ADDS. Diferentemente do observado na Seção 1.1, aqui há modificação do valor do registrador CPSR, que passou a armazenar a palavra **00000000000000000000000010000**. Os primeiros quatro bits em negrito representam as flags NZCV contendo valores de 0 para ambos os bits. Isso está em conformidade com o esperado, pois a soma não resulta em valor negativo e nem nulo, também não produz carry e nem overflow.



```
natan@asus: ~/repos/poli/lab-processadores/exp02

Register group: general
r0      0xf      15      r1      0x14      20
r2      0x407ffe5c 1082130012 r3      0x103ec 66540
r4      0x3ff90000 1073283072 r5      0x1      1
r6      0x2ff14    196372  r7      0x3ffff048 1073737800
r8      0x103ec    66540   r9      0x3fffd78 1073737080
r10     0x0        0       r11     0x2ff14    196372
r12     0x3ff90000 1073283072 sp      0x407ffce0 0x407ffce0
lr      0x103f8    66552   pc      0x10404 0x10404 <firstfunc>
cpsr    0x6000010 1610612752 fpscr   0x0        0

item-2-2.s
 9      MOV     r0, #0x0      @ zera registrador
10      MOV     r7, #0x1      @ exit(0)
11      SWI     0x0          @ system call
12      firstfunc:
> 13      ADDS   r0, r0, r1    @ calcula soma r0 = r0 + r1
14      MOV     pc, lr        @ retorna da funcao
15
16
17
18

remote Thread 1.182260 In: firstfunc L13 PC: 0x10404
(gdb) b main
Ponto de parada 1 at 0x103ec: file item-2-2.s, line 5.
(gdb) c
Continuing.

Breakpoint 1, main () at item-2-2.s:5
(gdb) s
(gdb) s
(gdb) s
firstfunc () at item-2-2.s:13
(gdb) 
```

Figure 3: Depuração mostrando os valores dos registradores imediatamente antes da instrução ADDS.

```
natan@asus: ~/repos/poli/lab-processadores/exp02

Register group: general
r0      0x23      35      r1      0x14      20
r2      0x407ffe5c 1082130012 r3      0x103ec   66540
r4      0x3ff90000 1073283072 r5      0x1      1
r6      0x2ff14    196372     r7      0x3ffff048 1073737800
r8      0x103ec    66540     r9      0x3fffd78 1073737080
r10     0x0        0          r11     0x2ff14    196372
r12     0x3ff90000 1073283072 sp      0x407ffce0 0x407ffce0
lr      0x103f8    66552     pc      0x10408 0x10408 <firstfunc+
cpsr    0x10      16      fpscr   0x0      0

item-2-2.s
9      MOV    r0, #0x0      @ zera registrador
10     MOV    r7, #0x1      @ exit(0)
11     SWI     0x0          @ system call
12     firstfunc:
13     ADDS   r0, r0, r1     @ calcula soma r0 = r0 + r1
> 14     MOV    pc, lr       @ retorna da funcao
15
16
17
18

remote Thread 1.182260 In: firstfunc L14 PC: 0x10408
Ponto de parada 1 at 0x103ec: file item-2-2.s, line 5.
(gdb) c
Continuing.

Breakpoint 1, main () at item-2-2.s:5
(gdb) s
(gdb) s
(gdb) s
firstfunc () at item-2-2.s:13
(gdb) s
(gdb) 
```

Figure 4: Depuração mostrando os valores dos registradores logo após a instrução ADDS.

## 2. Exercícios do capítulo 2

### 2.1. Exercício 2.4.1

Usamos o código do exercício 2.2 da aula passada para representar como se faz o “Compiling, making, debugging, and running”, apresentado no texto do exercício. O comando `arm build` compila e monta o arquivo, o comando `arm debug` permite a execução e depuração.

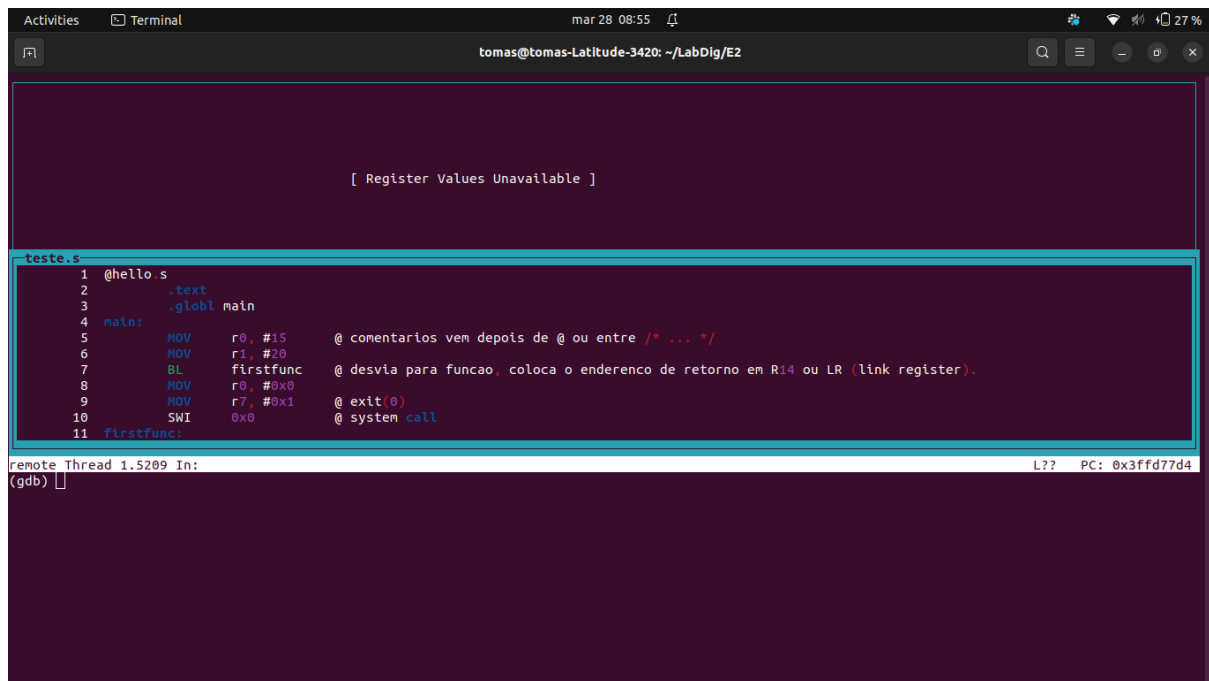


Figure 5: Nessa figura é possível ver o modo de depuração

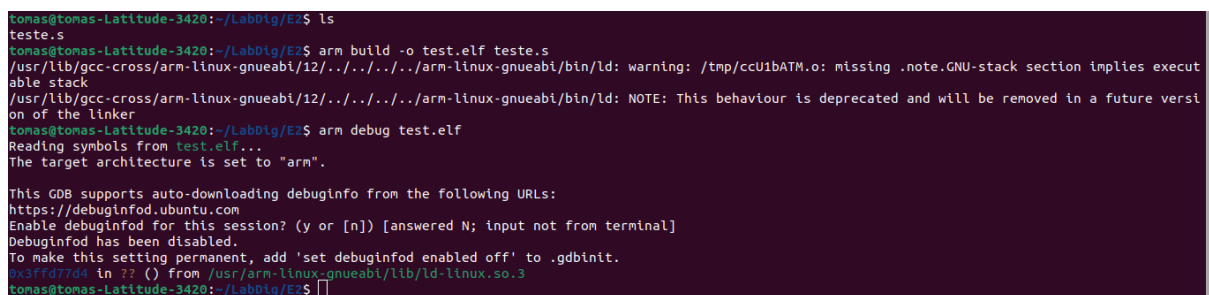


Figure 6: Nessa figura é possível ver os comandos mencionados na explicação do exercício.

## 2.2. Exercício 2.4.2

Para estudar a diferença entre o comando next e o comando step, iteramos sobre o programa da aula passada, do item 2.2, dando um next e um step a cada instrução, até o final do programa.

No caso do next, pudemos observar que usando o comando next iteramos sobre as instruções da main, sem “entrar” na firstfunc, conforme o print abaixo:

```
Activities Terminal mar 28 09:33 tomas@tomas-Latitude-3420: ~/LabDlg/E2

-Register group: general-
r0 0xf 15 r1 0x14 20
r2 0x4080005c 1082130524 r3 0x103ec 66540
r4 0x3ffa0000 1073348608 r5 0x1 1
r6 0x2ff14 196372 r7 0x3ffff048 1073737800
r8 0x103ec 66540 r9 0x3fffd78 1073737080
r10 0x0 0 r11 0x2ff14 196372
r12 0x3ffa0000 1073348608 sp 0x407ffee0 0x407ffee0
lr 0x3fe2e3f8 1071834104 pc 0x103f4 0x103f4 <main+8>
cpsr 0x60000010 1610612752 fpscr 0x0 0
fpsid 0x410430f0 1090793712 fpexc 0x40000000 1073741824

teste.s
1 @hello.s
2 .text
3 .globl main
4 main:
5     MOV    r0, #15    @ comentarios vem depois de @ ou entre /* ... */
6     MOV    r1, #20
7     BL     firstfunc  @ desvia para funcao, coloca o enderenco de retorno em R14 ou LR (link register).
8
9     MOV    r0, #0x0
10    MOV    r7, #0x1    @ exit(0)
11    SWI     0x0        @ system call
12 firstFunc:

remote Thread 1.6794 In: main L7 PC: 0x103f4
(gdb) b main
Breakpoint 1 at 0x103ec: file teste.s, line 5.
(gdb) c
Continuing.

Breakpoint 1, main () at teste.s:5
(gdb) n
(gdb) n
(gdb) n
(gdb) 
```

Figure 7: Instrução imediatamente anterior ao desvio para firstfunc usando next

```
Activities Terminal mar 28 09:33 tomas@tomas-Latitude-3420: ~/LabDlg/E2

-Register group: general-
r0 0x23 35 r1 0x14 20
r2 0x4080005c 1082130524 r3 0x103ec 66540
r4 0x3ffa0000 1073348608 r5 0x1 1
r6 0x2ff14 196372 r7 0x3ffff048 1073737800
r8 0x103ec 66540 r9 0x3fffd78 1073737080
r10 0x0 0 r11 0x2ff14 196372
r12 0x3ffa0000 1073348608 sp 0x407ffee0 0x407ffee0
lr 0x103f8 66552 pc 0x103f8 0x103f8 <main+12>
cpsr 0x10 16 fpscr 0x0 0
fpsid 0x410430f0 1090793712 fpexc 0x40000000 1073741824

teste.s
1 @hello.s
2 .text
3 .globl main
4 main:
5     MOV    r0, #15    @ comentarios vem depois de @ ou entre /* ... */
6     MOV    r1, #20
7     BL     firstfunc  @ desvia para funcao, coloca o enderenco de retorno em R14 ou LR (link register).
8     MOV    r0, #0x0
9     MOV    r7, #0x1    @ exit(0)
10    SWI     0x0        @ system call
11 firstFunc:

remote Thread 1.6794 In: main L8 PC: 0x103f8
(gdb) b main
Breakpoint 1 at 0x103ec: file teste.s, line 5.
(gdb) c
Continuing.

Breakpoint 1, main () at teste.s:5
(gdb) n
(gdb) n
(gdb) n
(gdb) 
```

Figure 8: Instrução imediatamente posterior ao desvio para firstfunc usando next, mostrando que o next não entra em branches

Repetindo o procedimento com a instrução step, podemos observar que o depurador segue o desvio de BL e podemos depurar também a firstfunc.





## 2.3. Exercício 2.4.2 - Diferentes formas de visualizar dados

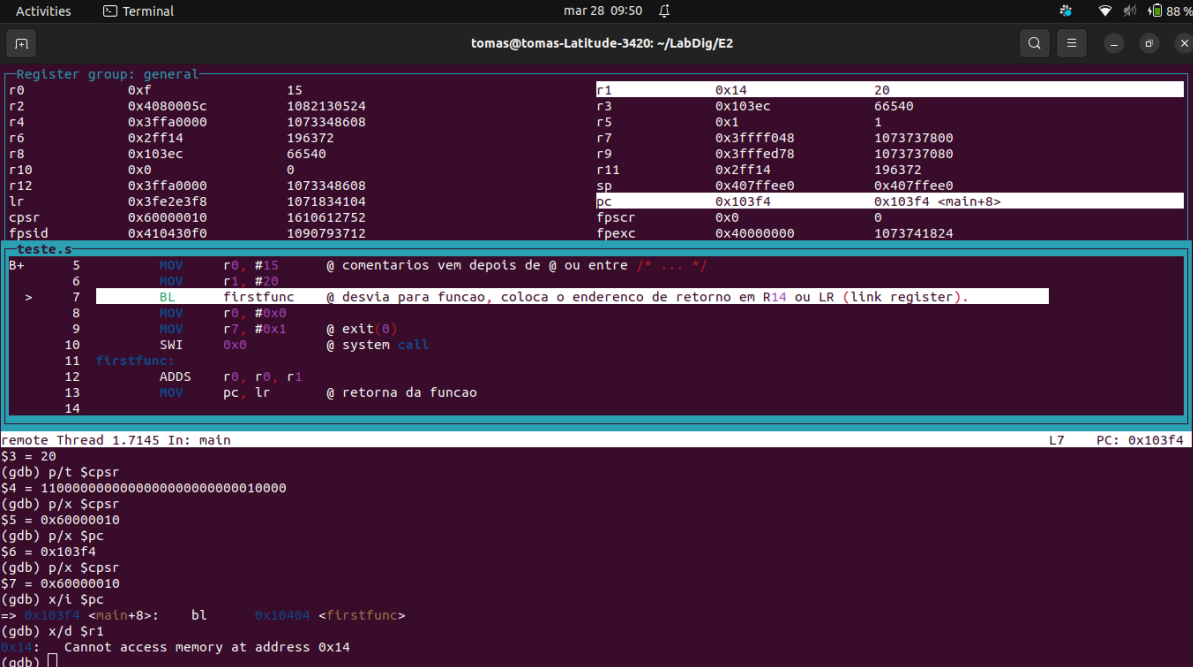
Também a título de simplificação, podemos usar o mesmo código do item 2-2 para analisar as diferentes formas o conteúdo dos registradores e da memória.

Em modo de depuração, com um breakpoint na main, podemos usar uma série de comando para observar essas dados:

- p/x (registrador): Mostra o conteúdo do registrador em hexadecimal.
- x/i (endereço da memória): Mostra o conteúdo do endereço da memória como instrução. Podemos usar um registrador que contém o endereço, ou o próprio endereço em hexa.
- x/d (endereço da memória): Mostra o conteúdo do endereço da memória em decimal. Também é possível usar um registrador que contenha o endereço.
- p/t (registrador): Mostra o conteúdo do registrador, ignorando os 0's a esquerda.

Para analisar a diferença entre usar p/t \$cpsr ou p/x \$cpsr, podemos recorrer a descrição das formas de acessar os dados descrita acima. Se usamos p/t, devemos ter a consciência de que o \$cpsr tem 32 bits e que estamos vendo apenas o menos significativo, que os bits não mostrados são todos 0. Isso é particularmente relevante porque os bits mais significativos do CSCR são os bits das flags NCZV.

Quando se usa o comando p/x, a saída é em hexadecimal. Daí é possível fazer uma conversão direta para uma palavra de 32 bits decimais e analisar o CSCR bit a bit.



The screenshot shows a GDB terminal window with the following content:

```
Activities Terminal mar 28 09:50 tomas@tomas-Latitude-3420: ~/LabDig/E2

-Register group: general-
r0 0xf 15 r1 0x14 20
r2 0x4080005c 1082130524 r3 0x103ec 66540
r4 0x3ffa0000 1073348608 r5 0x1 1
r6 0x2ff14 196372 r7 0x3ffff048 1073737800
r8 0x103ec 66540 r9 0x3fffd78 1073737080
r10 0x0 0 r11 0x2ff14 196372
r12 0x3ffa0000 1073348608 SP 0x407ffee0 0x407ffee0
lr 0x3fe2e3f8 1071834104 PC 0x103f4 0x103f4 <main+8>
cpsr 0x60000010 1610612752 fpscr 0x0 0
fpsid 0x410430f0 1090793712 fpexc 0x40000000 1073741824

-teste.s-
0+ 5 MOV r0, #15 @ comentaros vem depois de @ ou entre /* ... */
6 MOV r1, #20
7 BL firstfunc @ desvia para funcao, coloca o enderenco de retorno em R14 ou LR (link register).
8 MOV r0, #0x0
9 MOV r7, #0x1 @ exit(0)
10 SWI 0x0 @ system call
11 firstfunc:
12 ADDS r0, r0, r1
13 MOV pc, lr @ retorna da funcao
14

remote Thread 1.7145 In: main L7 PC: 0x103f4
S3 = 20
(gdb) p/t $cpsr
S4 = 11000000000000000000000000000000
(gdb) p/x $cpsr
S5 = 0x60000010
(gdb) p/x $pc
S6 = 0x103f4
(gdb) p/x $cpsr
S7 = 0x60000010
(gdb) x/i $pc
=> 0x103f4 <main+8>: bl 0x10404 <firstfunc>
(gdb) x/d $r1
0x14: Cannot access memory at address 0x14
(gdb)
```

Figure 11: Uso de diversos tipos de exibição do conteúdo dos registradores/memória

### 3. Soma com sinal e sem sinal

Esta seção é referente ao exercício 3.10.1 do Manual de Laboratório ARM. A Listagem 2 mostra o programa implementado para solução do problema.

Listagem 2: Programa para calcular a soma de com sinal e sem sinal de dois números

```
@ ex_3_10_1.s
@ Programa para calcular a soma entre dois números atualizando
@ as flags do registrador CPSR
.text
.globl main
main:
    LDR r0, =0xFFFFFFFF @ carrega valor no primeiro registrador
    LDR r1, =0x12345678 @ carrega valor no segundo registrador
    BL firstfunc @ chamada da função
    LDR r7, =0x1 @ exit(0)
    SWI 0x0 @ termina o programa
firstfunc:
    ADDS r2, r0, r1 @ calcula r2 = r0 + r1
    MOV pc, lr @ retorna da função
```

Três somas foram efetuadas neste exercício, os valores dos operandos de cada soma estão dispostos na Table 1. A Listagem 2 mostra os valores dos registradores r0 e r1 para a Operação 1, contudo, para as demais operações, os valores desses registradores são devidamente alterados para os valores dos operandos A e B, respectivamente, como mostra a Table 1.

Operando	Operação 1	Operação 2	Operação 3
A	0xFFFF0000	0xFFFFFFFF	0x67654321
B	0x87654321	0x12345678	0x23110000

Table 1: Valores de cada operando para cada uma das três somas calculadas

Os valores decimais dos operandos e o resultado da soma foram calculados considerando valores unsigned e signed. A Table 2 mostra esses valores, com destaque para as duas células assinaladas em azul, que apresentam um resultado superior a  $2^{32}$  e para a célula assinalada em vermelho, que apresenta um resultado maior que  $2^{31}$ .

#	Unsigned			Signed		
	A	B	A+B	A	B	A+B
1	4294901760	2271560481	6566462241	-65536	-2023406815	-2023472351
2	4294967295	305419896	4600387191	-1	305419896	305419895
3	1734689569	588316672	2323006241	1734689569	588316672	2323006241

Table 2: Valores, em decimal, de cada operando e da soma para cada operação, considerando os casos unsigned e signed

Os valores das flags NZCV da ULA foram obtidos a partir da depuração do registrador CPSR logo após a instrução ADDS. As Fig. 12, Fig. 13 e Fig. 14 mostram a tela do depurador e a Table 3 resume os valores esperados e obtidos das flags. A partir da comparação desta tabela com a Table 2, é possível notar que C=1 quando há overflow unsigned e V=1 quando há overflow signed.

Op.	Valor Esperado				Valor Obtido
	N	Z	C	V	NZCV
1	1	0	1	0	1010
2	0	0	1	0	0010
3	1	0	0	1	1001

Table 3: Valores esperados e obtidos das flags NZCV do registrador CPSR

```

natan@asus: ~/repos/poli/lab-processadores/exp02

Register group: general
r0      0xffff0000      -65536      r1      0x87654321      -2023406815
r2      0x87644321      -2023472351  r3      0x103ec        66540
r4      0x3ff90000      1073283072  r5      0x1          1
r6      0x2ff14         196372      r7      0x3ffff048     1073737800
r8      0x103ec         66540      r9      0x3fffed78     1073737080
r10     0x0             0           r11     0x2ff14        196372
r12     0x3ff90000      1073283072  sp      0x407ffce0     0x407ffce0
lr      0x103f8         66552      pc      0x10408        0x10408 <firstfunc+
cpsr    0xa0000010      -1610612720  fpscr   0x0           0

ex_3_10_1.s
8      LDR r0, =0x0
9      LDR r7, =0x1
10     SWI 0x0           @ termina o programa
11     firstfunc:
12     ADDS r2, r0, r1    @ calcula r2 = r0 + r1
> 13     MOV pc, lr       @ retorna da função
14
15
16
17

remote Thread 1.240006 In: firstfunc L13 PC: 0x10408
Ponto de parada 1 at 0x103ec: file ex_3_10_1.s, line 5.
(gdb) c
Continuing.

Breakpoint 1, main () at ex_3_10_1.s:5
(gdb) s
(gdb) s
(gdb) s
firstfunc () at ex_3_10_1.s:12
(gdb) s
(gdb)

```

Figure 12: Depuração da Operação 1 mostrando o estado dos registradores logo após a execução do comando ADDS

```

natan@asus: ~/repos/poli/lab-processadores/exp02

Register group: general
r0      0xffffffff      -1          r1      0x12345678      305419896
r2      0x12345677      305419895  r3      0x103ec        66540
r4      0x3ff90000      1073283072  r5      0x1          1
r6      0x2ff14         196372      r7      0x3ffff048     1073737800
r8      0x103ec         66540      r9      0x3fffed78     1073737080
r10     0x0             0           r11     0x2ff14        196372
r12     0x3ff90000      1073283072  sp      0x407ffce0     0x407ffce0
lr      0x103f8         66552      pc      0x10408        0x10408 <firstfunc+
cpsr    0x20000010      536870928  fpscr   0x0           0

ex_3_10_1.s
8      LDR r0, =0x0
9      LDR r1, =0x0
10     SWI 0x123456       @ terminate the program
11     firstfunc:
12     ADDS r2, r0, r1    @ calcula r2 = r0 + r1
> 13     MOV pc, lr       @ retorna da função
14
15
16
17

remote Thread 1.239278 In: firstfunc L13 PC: 0x10408
Ponto de parada 1 at 0x103ec: file ex_3_10_1.s, line 5.
(gdb) c
Continuing.

Breakpoint 1, main () at ex_3_10_1.s:5
(gdb) s
(gdb) s
(gdb) s
firstfunc () at ex_3_10_1.s:12
(gdb) s
(gdb)

```

Figure 13: Depuração da Operação 2 mostrando o estado dos registradores logo após a execução do comando ADDS

```

natan@asus: ~/repos/poli/lab-processadores/exp02
Register group: general
r0      0x67654321      1734689569      r1      0x23110000      588316672
r2      0x8a764321      -1971961055     r3      0x103ec        66540
r4      0x3ff90000      1073283072     r5      0x1          1
r6      0x2ff14         196372         r7      0x3ffff048    1073737800
r8      0x103ec         66540         r9      0x3fffd78     1073737080
r10     0x0             0              r11     0x2ff14       196372
r12     0x3ff90000      1073283072     sp      0x407ffce0     0x407ffce0
lr      0x103f8         66552         pc      0x10408       0x10408 <firstfunc+
cpsr    0x90000010     -1879048176    fpscr   0x0           0

ex_3_10_1.s
8      LDR r0, =0x0
9      LDR r7, =0x1
10     SWI 0x0          @ termina o programa
11     firstfunc:
12     ADDS r2, r0, r1   @ calcula r2 = r0 + r1
> 13     MOV pc, lr      @ retorna da função
14
15
16
17

remote Thread 1.239519 In: firstfunc L13 PC: 0x10408
Ponto de parada 1 at 0x103ec: file ex_3_10_1.s, line 5.
(gdb) c
Continuing.

Breakpoint 1, main () at ex_3_10_1.s:5
(gdb) s
(gdb) s
(gdb) s
firstfunc () at ex_3_10_1.s:12
(gdb) s
(gdb)

```

Figure 14: Depuração da Operação 3 mostrando o estado dos registradores logo após a execução do comando ADDS

A partir da comparação das Table 2 e Table 3, é possível verificar os seguintes comportamentos das flags de estado da ULA do registrador CPSR:

- N=1, se o bit 31 do resultado for igual a 1; N=0, caso contrário
- Z=1, se todos os bits do resultado forem 0; Z=0, caso contrário
- C=1, se houver carry; C=0, caso contrário. Esta flag também pode ser usada para detectar overflow para valores unsigned. C=1, se há overflow unsigned.
- V=1, se houver overflow; V=0, caso contrário. Esta flag é usada para detectar overflow em valores signed.

Também vale ressaltar que usamos o LDR ao invés da instrução MOV, porque a instrução MOV só aceita palavras de no máximo 8 bits. Já a instrução LDR (Load register from memory), aceita palavras de 32 bits. \*

\* Encontramos uma diferença entre o observado praticamente no uso do GDB e o encontrando no manual de instruções do ARM, que diz que o LDR aceita palavras de até 12 bits. Pensamos que essa diferença se dá porque o gdb deve usar algum tipo de pseudoinstrução no lugar do LDR.

## 4. Multiplicação

### 4.1. Multiplicação com MULS

Esta seção é referente ao exercício 3.10.2 do Manual de Laboratório ARM. A Listagem 3 mostra o programa implementado para solução do problema, que consiste na multiplicação do número 0xFFFFFFFF por 0x80000000. Os dois operandos são armazenados nos registradores r0 e r1 e o resultado da operação é armazenado no registrador r2. As Fig. 15 e Fig. 16 mostram a depuração do programa antes e depois da execução da instrução MULS, respectivamente.

Listagem 3: Programa para calcular o produto entre dois números

```
@ ex_3_10_2a.s
@ Cálculo do produto entre dois números, atualizando as
@ flags do registrador CPSR
    .text
    .globl main
main:
    LDR r0, =0xFFFFFFFF @ carrega valor no primeiro registrador
    LDR r1, =0x80000000 @ carrega valor no segundo registrador
    LDR r2, =0x0         @ zera o reg. que armazena o resultado
    BL  firstfunc        @ chamada da função
    LDR r7, =0x1         @ exit(0)
    SWI 0x0              @ termina o programa
firstfunc:
    MULS r2, r0, r1      @ calcula r2 = r0 * r1
    MOV pc, lr           @ retorna da função
```

```

natan@asus: ~/repos/poli/lab-processadores/exp02

Register group: general
r0      0xffffffff      -1          r1      0x80000000      -2147483648
r2      0x0              0            r3      0x103ec        66540
r4      0x3ff90000       1073283072  r5      0x1           1
r6      0x2ff14          196372      r7      0x3ffff048    1073737800
r8      0x103ec          66540      r9      0x3fffed78    1073737080
r10     0x0              0            r11     0x2ff14       196372
r12     0x3ff90000       1073283072  sp      0x407ffce0      0x407ffce0
lr      0x103fc          66556      pc      0x10408        0x10408 <firstfunc>
cpsr    0x60000010       1610612752  fpscr   0x0           0

ex_3_10_2a.s
9      LDR r0, =0x0
10     LDR r7, =0x1
11     SWI 0x0           @ termina o programa
12     firstfunc:
> 13     MULS r2, r0, r1   @ calcula r2 = r0 * r1
14     MOV pc, lr        @ retorna da função
15
16
17
18

remote Thread 1.294541 In: firstfunc                                L13   PC: 0x10408
Ponto de parada 1 at 0x103ec: file ex_3_10_2a.s, line 5.
(gdb) c
Continuing.

Breakpoint 1, main () at ex_3_10_2a.s:5
(gdb) s
(gdb) s
(gdb) s
(gdb) s
firstfunc () at ex_3_10_2a.s:13
(gdb)

```

Figure 15: Depuração do programa mostrando o estado dos registradores imediatamente antes da execução da instrução MULS

```

natan@asus: ~/repos/poli/lab-processadores/exp02

Register group: general
r0      0xffffffff      -1          r1      0x80000000      -2147483648
r2      0x80000000       -2147483648  r3      0x103ec        66540
r4      0x3ff90000       1073283072  r5      0x1           1
r6      0x2ff14          196372      r7      0x3ffff048    1073737800
r8      0x103ec          66540      r9      0x3fffed78    1073737080
r10     0x0              0            r11     0x2ff14       196372
r12     0x3ff90000       1073283072  sp      0x407ffce0      0x407ffce0
lr      0x103fc          66556      pc      0x1040c        0x1040c <firstfunc+
cpsr    0xa0000010       -1610612720  fpscr   0x0           0

ex_3_10_2a.s
9      LDR r0, =0x0
10     LDR r7, =0x1
11     SWI 0x0           @ termina o programa
12     firstfunc:
13     MULS r2, r0, r1   @ calcula r2 = r0 * r1
> 14     MOV pc, lr        @ retorna da função
15
16
17
18

remote Thread 1.294541 In: firstfunc                                L14   PC: 0x1040c
(gdb) c
Continuing.

Breakpoint 1, main () at ex_3_10_2a.s:5
(gdb) s
(gdb) s
(gdb) s
(gdb) s
firstfunc () at ex_3_10_2a.s:13
(gdb) s
(gdb)

```

Figure 16: Depuração do programa mostrando o estado dos registradores logo após a execução da instrução MULS



Unsigned			Signed		
A	B	A x B	A	B	A x B
4294967295	2147483648	~9,22e18	-1	-2147483648	2147483648

Table 4: Valores esperados da operação

Na Table 4, que mostra os valores esperados em decimal para signed e unsigned, é observado que o resultado da operação unsigned é maior que  $2^{32}$  e da operação signed é maior que  $2^{32} - 1$ . Desta forma, há overflow para ambas as situações, pois o resultado da instrução MUL(S) é armazenado em um único registrador de 32 bits. Isso é notado no valor do registrador r2, que mostra um resultado incorreto para a operação.

Analisando o registrador CPSR, que possui valor 1010 para os 4 bits mais significativos, é possível ver que as flags também apresentam valores incorretos. As flags sinalizam para resultado negativo (incorreto) e não sinaliza para overflow (também incorreto).

Desta forma, nota-se que não é possível recuperar um valor confiante do resultado nem das flags de estado da ULA em operações de multiplicação envolvendo overflow. Uma forma de lidar com isso é aumentar o número de bits do resultado usando 2 registradores de 32 bits, ao invés de apenas 1, com o uso das instruções UMULLS e SMULLS. A Listagem 4 mostra o programa usando a instrução UMULLS. Os registradores r2 e r3 armazenam os 32 bits menos e mais significativos, respectivamente.

## 4.2. Multiplicação com UMULLS e SMULLS

Listagem 4: Multiplicação de dois números usando a instrução UMULLS

```
@ ex_3_10_2b.s
@ Multiplicação entre dois números usando dois registradores
@ para armazenar o resultado
.text
.globl main
main:
    LDR r0, =0xFFFFFFFF @ carrega valor no primeiro registrador
    LDR r1, =0x80000000 @ carrega valor no segundo registrador
    LDR r2, =0x0          @ zera o reg. que armazena o resultado
    BL firstfunc          @ chamada da função
    LDR r7, =0x1          @ exit(0)
    SWI 0x0              @ termina o programa
firstfunc:
    UMULLS r2, r3, r0, r1 @ calcula r3r2 = r0 * r1
```

**MOV pc, lr**

@ retorna da função

```
Register group: general
r0      0xffffffff -1 r1      0x80000000 -2147483648
r2      0x80000000 -2147483648 r3      0x7fffffff 2147483647
r4      0x3ff90000 1073283072 r5      0x1 1
r6      0x2ff14 196372 r7      0x3ffff048 1073737800
r8      0x103ec 66540 r9      0x3fffed78 1073737080
r10     0x0 0 r11     0x2ff14 196372
r12     0x3ff90000 1073283072 sp      0x407ffce0 0x407ffce0
lr      0x103fc 66556 pc      0x1040c 0x1040c <firstfunc+
cpsr    0x20000010 536870928 fpscr  0x0 0

ex_3_10_2b.s
9      LDR r0, =0x0
10     LDR r7, =0x1
11     SWI 0x0 @ termina o programa
12     firstfunc:
13     UMULLS r2, r3, r0, r1 @ calcula r3r2 = r0 * r1
14     MOV pc, lr @ retorna da função
15
16
17
18

remote Thread 1.296338 In: firstfunc L14 PC: 0x1040c
(gdb) c
Continuing.

Breakpoint 1, main () at ex_3_10_2b.s:5
(gdb) s
(gdb) s
(gdb) s
(gdb) s
firstfunc () at ex_3_10_2b.s:13
(gdb) s
(gdb) 
```

Figure 17: Depuração do programa mostrando o estado dos registradores logo após a execução da instrução UMULLS

```

natan@asus: ~/repos/poli
natan@asus:~/repos/poli$ hconv -n 32 7fffffff


|     |                                       |
|-----|---------------------------------------|
| HEX | 7FFFFFFF                              |
| BIN | 01111111111111111111111111111111 (32) |
| UNS | 2147483647                            |
| SIG | 2147483647                            |


natan@asus:~/repos/poli$ hconv -n 32 80000000


|     |                                       |
|-----|---------------------------------------|
| HEX | 80000000                              |
| BIN | 10000000000000000000000000000000 (32) |
| UNS | 2147483648                            |
| SIG | -2147483648                           |


natan@asus:~/repos/poli$ bconv 0111111111111111111111111111111110000000000000000
0000000000000000


|     |                                                                     |
|-----|---------------------------------------------------------------------|
| BIN | 01111111111111111111111111111111100000000000000000000000000000 (64) |
| HEX | 7FFFFFFF80000000                                                    |
| UNS | 9223372034707292160                                                 |
| SIG | 9223372034707292160                                                 |


natan@asus:~/repos/poli$

```

Figure 18: Concatenação dos valores dos registradores r3 e r2 e valor da multiplicação unsigned usando representação 64-bits

Observando o resultado mostrado na Fig. 18, observa-se que é obtido o valor correto da multiplicação unsigned concatenando os valores dos registradores r2 e r3, obtidos pela depuração (Fig. 17), em uma única palavra de 64 bits. Ademais, a Fig. 19 também sinaliza para um resultado correto da operação de multiplicação signed quando representada em 64-bits.

A instrução UMULL trata os operandos como inteiros sem sinal e realiza a multiplicação sem qualquer extensão de sinal ou manipulação de bit de sinal. Já a instrução SMULL leva em consideração o sinal dos operandos e realiza a multiplicação com extensão de sinal adequada e manipulação de bit de sinal.

Se apenas uma instrução de multiplicação longa fosse fornecida, não seria possível realizar multiplicações longas em inteiros com e sem sinal de forma eficiente. Isso exigiria instruções extras para lidar com a extensão do sinal ou a manipulação do bit do sinal, levando a uma execução de código mais lenta e maior complexidade de código. Ao fornecer instruções UMULL e SMULL separadas, a arquitetura ARM pode executar ambos os tipos de multiplicações de forma eficiente e com menos complexidade de código.

```

natan@asus: ~/repos/poli/lab-processadores/exp02
Register group: general
r0 0xffffffff -1 r1 0x80000000 -2147483648
r2 0x80000000 -2147483648 r3 0x0 0
r4 0x3ff90000 1073283072 r5 0x1 1
r6 0x2ff14 196372 r7 0x3ffff048 1073737800
r8 0x103ec 66540 r9 0x3fffed78 1073737080
r10 0x0 0 r11 0x2ff14 196372
r12 0x3ff90000 1073283072 sp 0x407ffce0 0x407ffce0
lr 0x103fc 66556 pc 0x1040c 0x1040c <firstfunc+
cpsr 0x20000010 536870928 fpscr 0x0 0
ex_3_10_2b.s
9 LDR r0, =0x0
10 LDR r7, =0x1
11 SWI 0x0 @ termina o programa
12 firstfunc:
13 SMULLS r2, r3, r0, r1 @ calcula r3r2 = r0 * r1
> 14 MOV pc, lr @ retorna da função
15
16
17
18
remote Thread 1.296823 In: firstfunc L14 PC: 0x1040c
(gdb) c
Continuing

```

Figure 19: Depuração do programa mostrando o estado dos registradores logo após a execução da instrução SMULLS

```

natan@asus: ~/repos/poli
natan@asus:~/repos/poli$ hconv -n 32 0x0

```

HEX	0x0
BIN	00000000000000000000000000000000 (32)
UNS	0
SIG	0

```

natan@asus:~/repos/poli$ hconv -n 32 0x80000000

```

HEX	0x80000000
BIN	10000000000000000000000000000000 (32)
UNS	2147483648
SIG	-2147483648

```

natan@asus:~/repos/poli$ bconv 0000000000000000000000000000000010000000000000000
0000000000000000

```

BIN	00000000000000000000000000000000100000000000000000000000000000 (64)
HEX	80000000
UNS	2147483648
SIG	2147483648

```

natan@asus:~/repos/poli$

```

Figure 20: Concatenação dos valores dos registradores r3 e r2 e valor da multiplicação signed usando representação 64-bits

## 5. Multiplicação pelo número 32

Esta seção é referente ao exercício 3.10.3 do Manual de Laboratório ARM, que aborda o problema da multiplicação por múltiplo de  $2^n$ , que pode ser feita de forma eficiente com o uso de shift para esquerda, ao invés da instrução de multiplicação. A Listagem 5 mostra o programa implementado para resolução do problema.

Listagem 5: Multiplicação por 32 eficiente usando left-shift

```
@ ex_3_10_3.s
@ Algoritmo de multiplicação eficiente utilizando deslocamento
@ lógico para esquerda para multiplicar um número por 32
.text
.globl main
main:
    LDR r0, =0x95A      @ carrega valor no primeiro registrador
    LDR r2, =0x0        @ zera o reg. que armazena o resultado
    BL firstfunc        @ chamada da função
    LDR r7, =0x1        @ exit(0)
    SWI 0x0            @ termina o programa
firstfunc:
    LSL r2, r0, #5      @ calcula r2 = r0 * 2^5
    MOV pc, lr          @ retorna da função
```

A instrução LSL (Logical Shift Left), utilizada neste algoritmo, realiza o deslocamento lógico para a esquerda. No entanto, a arquitetura ARMv7 ainda dispõe de outras instruções de deslocamento, três delas estão listadas abaixo:

- LSL: realiza um deslocamento de bits para a esquerda, preenchendo os bits menos significativos com zeros. O deslocamento é lógico, o que significa que o bit mais à esquerda é perdido e um novo bit é inserido no bit menos significativo. Pode ser usado para fazer multiplicação por  $2^n$
- LSR: realiza um deslocamento de bits para direita, preenchendo os bits mais significativos com zeros. Como o deslocamento é lógico, o bit mais à direita é perdido e um novo bit é inserido no bit mais significativo. Pode ser usado para fazer divisão unsigned por  $2^n$
- ASR: realiza um deslocamento de bits para a direita, preenchendo os bits mais significativos com o bit mais significativo original (o sinal). Isso é útil para manter o sinal em operações com números com sinal. Pode ser usado para fazer divisão signed por  $2^n$

Assim, embora ambas as instruções realizem deslocamentos, elas operam de maneira diferente em relação aos bits que são deslocados e aos valores que são inseridos. Ainda existem outras instruções de deslocamento, como ROR e

Além disso, temos a adição do pós-fixado S, que atualiza as flags do CPSR. Apesar de não termos nenhuma operação condicional que faça uso desses valores, optamos por usá-la para seguir nos aprofundando no estudo do CPSR.

As Fig. 21 e Fig. 22 mostram a depuração antes e depois da execução do comando LSLs, respectivamente. Observando o valor do registrador r2, que armazena o resultado da operação, nota-se que a operação é executada corretamente,  $2394 * 32 = 76608$ .

The screenshot shows a GDB debugger window with the following content:

```

natan@asus: ~/repos/poli/lab-processadores/exp02

Register group: general
r0      0x95a      2394      r1      0x407ffe54  1082130004
r2      0x0        0          r3      0x103ec    66540
r4      0x3ff90000 1073283072 r5      0x1        1
r6      0x2ff14    196372    r7      0x3ffff048 1073737800
r8      0x103ec    66540    r9      0x3fffed78 1073737080
r10     0x0        0          r11     0x2ff14    196372
r12     0x3ff90000 1073283072 sp      0x407ffce0  0x407ffce0
lr      0x103f8    66552    pc      0x10404    0x10404 <firstfunc>
cpsr    0x60000010 1610612752 fpscr    0x0        0

ex_3_10_3.s
8      LDR r0, =0x0
9      LDR r7, =0x1
10     SWI 0x0      @ termina o programa
11     firstfunc:
> 12     LSLs r2, r0, #5 @ calcula r2 = r0 * 2^5
13     MOV pc, lr    @ retorna da função
14
15
16
17

remote Thread 1.297623 In: firstfunc L12 PC: 0x10404
(gdb) b main
Ponto de parada 1 at 0x103ec: file ex_3_10_3.s, line 5.
(gdb) c
Continuing.

Breakpoint 1, main () at ex_3_10_3.s:5
(gdb) s
(gdb) s
(gdb) s
firstfunc () at ex_3_10_3.s:12
(gdb)

```

Figure 21: Depuração do programa mostrando o estado dos registradores imediatamente antes da execução da instrução LSLs

```

natan@asus: ~/repos/poli/lab-processadores/exp02
Register group: general
r0      0x95a      2394      r1      0x407ffe54  1082130004
r2      0x12b40    76608     r3      0x103ec     66540
r4      0x3ff90000 1073283072 r5      0x1       1
r6      0x2ff14    196372    r7      0x3ffff048 1073737800
r8      0x103ec    66540     r9      0x3fffd78  1073737080
r10     0x0        0          r11     0x2ff14    196372
r12     0x3ff90000 1073283072 sp      0x407ffce0  0x407ffce0
lr      0x103f8    66552     pc      0x10408    0x10408 <firstfunc+
cpsr    0x10      16         fpscr   0x0       0

ex_3_10_3.s
8      LDR r0, =0x0
9      LDR r7, =0x1
10     SWI 0x0      @ termina o programa
11     firstfunc:
12     LSLS r2, r0, #5 @ calcula r2 = r0 * 2^5
> 13     MOV pc, lr   @ retorna da função
14
15
16
17

remote Thread 1.297623 In: firstfunc L13 PC: 0x10408
Ponto de parada 1 at 0x103ec: file ex_3_10_3.s, line 5.
(gdb) c
Continuing.

Breakpoint 1, main () at ex_3_10_3.s:5
(gdb) s
(gdb) s
(gdb) s
firstfunc () at ex_3_10_3.s:12
(gdb) s
(gdb)

```

Figure 22: Depuração do programa mostrando o estado dos registradores logo após a execução da instrução LSLS

## 6. Register-Swap Algorithm

Esta seção é referente ao exercício 3.10.4 do Manual de Laboratório ARM. A Listagem 6 mostra o programa implementado para resolução do problema.

Listagem 6: Algoritmo register-swap

```

@ ex_3_10_4.s
@ Implementação do algoritmo register-swap, que permuta os
@ valores de dois registradores sem o uso de um registrador
@ auxiliar
.text
.globl main
main:
    LDR r0, =0xF631024C @ carrega valor no primeiro registrador
    LDR r1, =0x17539ABD @ carrega valor no segundo registrador
    BL swap              @ chamada da função
    LDR r7, =0x1         @ exit(0)
    SWI 0x0             @ termina o programa

```

```

swap:
    EOR r0, r0, r1    @ calcula r0 = r0 eor r1
    EOR r1, r0, r1    @ calcula r0 = r1 eor r1
    EOR r0, r0, r1    @ calcula r0 = r0 eor r1
    MOV pc, lr        @ retorna da função

```

O algoritmo register-swap realiza a troca dos valores entre dois registradores sem o uso de um registrador auxiliar. As Fig. 23 e Fig. 24 mostram que foi possível alternar os valores entre os registradores r0 e r1 utilizando este algoritmo.

Essa é uma implementação do algoritmo XOR Swap. Estamos usando as instruções EOR, que equivalem a um XOR lógico. A instrução EOR r0, r1, r2 equivale a armazenar em r0 o resultado de um XOR lógico entre r1 e r2.

A permutação dos valores é dada da seguinte forma:

$$\begin{aligned}
 a' &= a \oplus b \\
 b &= a' \oplus b = a \oplus b \oplus b = a \\
 a &= a' \oplus b = a \oplus b \oplus a = b
 \end{aligned}$$

The screenshot shows a GDB window with the following content:

Register group: general

r0	0xf631024c	-164560308	r1	0x17539abd	391355069
r2	0x407ffe5c	1082130012	r3	0x103ec	66540
r4	0x3ff90000	1073283072	r5	0x1	1
r6	0x2ff14	196372	r7	0x3ffff048	1073737800
r8	0x103ec	66540	r9	0x3fffed78	1073737080
r10	0x0	0	r11	0x2ff14	196372
r12	0x3ff90000	1073283072	sp	0x407ffce0	0x407ffce0
lr	0x103f8	66552	pc	0x10404	0x10404 <swap>
cpsr	0x60000010	1610612752	fpscr	0x0	0

ex\_3\_10\_4.s

```

8    LDR r0, =0x0
9    LDR r7, =0x1    @ exit(0)
10   SWI 0x0        @ termina o programa
11   swap:
> 12   EOR r0, r0, r1 @ calcula r0 = r0 eor r1
13   EOR r1, r0, r1 @ calcula r0 = r1 eor r1
14   EOR r0, r0, r1 @ calcula r0 = r0 eor r1
15   MOV pc, lr      @ retorna da função
16
17

```

remote Thread 1.298323 In: swap L12 PC: 0x10404

(gdb) b main

Ponto de parada 1 at 0x103ec: file ex\_3\_10\_4.s, line 5.

(gdb) c

Continuing.

Breakpoint 1, main () at ex\_3\_10\_4.s:5

(gdb) s

(gdb) s

(gdb) s

swap () at ex\_3\_10\_4.s:12

(gdb)

Figure 23: Depuração do programa mostrando o estado dos registradores imediatamente antes da execução do algoritmo



```

natan@asus: ~/repos/poli/lab-processadores/exp02
Register group: general
r0      0x17539abd      391355069      r1      0xf631024c      -164560308
r2      0x407ffe5c      1082130012     r3      0x103ec        66540
r4      0x3ff90000      1073283072     r5      0x1           1
r6      0x2ff14         196372         r7      0x3ffff048     1073737800
r8      0x103ec         66540          r9      0x3fffd78      1073737080
r10     0x0             0              r11     0x2ff14        196372
r12     0x3ff90000      1073283072     sp      0x407ffce0      0x407ffce0
lr      0x103f8         66552          pc      0x10410        0x10410 <swap+12>
cpsr    0x60000010      1610612752     fpscr   0x0           0

ex_3_10_4.s
8      LDR r0, =0x0
9      LDR r7, =0x1      @ exit(0)
10     SWI 0x0           @ termina o programa
11     swap:
12     EOR r0, r0, r1     @ calcula r0 = r0 eor r1
13     EOR r1, r0, r1     @ calcula r0 = r1 eor r1
14     EOR r0, r0, r1     @ calcula r0 = r0 eor r1
15     MOV pc, lr        @ retorna da função
16
17

remote Thread 1.298323 In: swap L15 PC: 0x10410
Continuing.

Breakpoint 1, main () at ex_3_10_4.s:5
(gdb) s
(gdb) s
(gdb) s
swap () at ex_3_10_4.s:12
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb)

```

Figure 24: Depuração do programa mostrando o estado dos registradores logo após a execução do algoritmo

## 7. Conclusão

Neste experimento, foram abordadas algumas operações aritméticas e a recuperação das flags de controle NZCV do registrador CPSR. Os seguintes tópicos resumizam este experimento:

- A arquitetura ARM possui um registrador denominado Current Program Status Register (CPSR), que contém bits (flags) de estado e de controle do processador;
- Foi estudado o comportamento e o significado das flags de estado N, Z, C, V (negative, zero, carry e overflow) do registrador CPSR, que armazenam informações sobre o estado da última operação performada pela ULA;
- Muitas instruções da arquitetura ARM permitem a adição do sufixo “S” em seu nome para que haja a modificação das flags do registrador CPSR;
- Tanto números signed quanto unsigned podem ser somados com a mesma instrução ADDS e o overflow pode ser verificado para cada caso separadamente, C=1 (overflow unsigned) e V=1 (overflow signed);
- Não é possível recuperar um valor confiante do resultado nem das flags de estado da ULA em operações de multiplicação envolvendo overflow. Uma forma de lidar com isso é aumentar o número de bits do resultado usando 2 registradores de 32 bits, ao invés de apenas 1, com o uso das instruções UMULLS e SMULLS;
- Ao fornecer instruções UMULL e SMULL separadas, a arquitetura ARM pode executar ambos os tipos de multiplicações de forma eficiente e com menos complexidade de código, um vez que uma única instrução de multiplicação longa

exigiria instruções extras para lidar com a extensão do sinal ou a manipulação do bit do sinal;

- É possível fazer multiplicações de forma eficiente por múltiplos de  $2^n$  substituindo o uso da instrução de multiplicação pela instrução de deslocamento lógico para esquerda;
- É possível alternar os valores entre dois registradores sem o uso de registradores auxiliares com o algoritmo register-swap, feito, unicamente, com instruções EOR.