



ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E SISTEMAS
DIGITAIS

Relatório 2 de Laboratório de Processador

Turma 02

No. USP

Lucas Haug

10773565

Renzo Armando dos Santos
Abensur

10772414

São Paulo

2021

Lucas Haug
Renzo Armando dos Santos Abensur

Relatório 2 de Laboratório de Processador

Relatório apresentado como requisito para avaliação na disciplina PCS3432 - Laboratório de Processadores, no curso de Engenharia Elétrica oferecido pela Escola Politécnica da Universidade de São Paulo.

São Paulo

2021

SUMÁRIO

| | |
|------------------------|----------|
| 1. Planejamento | 3 |
| 2. Item 2.4 | 3 |
| Exercícios 2.4.1 | 3 |
| Exercícios 2.4.2 | 5 |
| Exercícios 2.4.3 | 7 |
| 3. Item 3.10 | 7 |
| Exercícios 3.10.1 | 7 |
| Exercícios 3.10.2 | 10 |
| Exercícios 3.10.3 | 12 |
| Exercícios 3.10.4 | 12 |

1. Planejamento

1. Clonar <https://github.com/EpicEric/gcc-arm>.
2. Rodar o script `run_docker.sh`.
3. Rodar `gcc item-2-2.s`.
4. Rodar `gdb a.out`.
5. Apertar enter para iniciar.
6. Colocar break points:
 - a. b main.
 - b. b 9.
7. Rodar o programa com `r`.
8. Rodar passo a passo com `s` ou deixar o programa rodar tudo até o breakpoint com `c`.
9. Testar com ADDS ao invés de ADD.

ADD

| Register group: general | | | | | | | | | | | |
|-------------------------|----------|---------|------|------------|------------|-----|------------|---------|-----|--------|-------|
| r0 | 0x18 | 24 | r1 | 0x20026 | 131110 | r2 | 0xffffffff | -1 | r3 | 0xa9c8 | 43464 |
| r4 | 0x1 | 1 | r5 | 0x1ffff8 | 2097144 | r6 | 0x0 | 0 | r7 | 0x0 | 0 |
| r8 | 0x0 | 0 | r9 | 0x0 | 0 | r10 | 0x200100 | 2097408 | r11 | 0x0 | 0 |
| r12 | 0x1fffcc | 2097100 | sp | 0x1ffff8 | 2097144 | lr | 0x8224 | 33316 | pc | 0x822c | 33324 |
| fps | 0x0 | 0 | cpsr | 0x60000013 | 1610612755 | | | | | | |

ADDS

| Register group: general | | | | | | | | | | | |
|-------------------------|----------|---------|------|----------|---------|-----|------------|---------|-----|--------|-------|
| r0 | 0x18 | 24 | r1 | 0x20026 | 131110 | r2 | 0xffffffff | -1 | r3 | 0xa9c8 | 43464 |
| r4 | 0x1 | 1 | r5 | 0x1ffff8 | 2097144 | r6 | 0x0 | 0 | r7 | 0x0 | 0 |
| r8 | 0x0 | 0 | r9 | 0x0 | 0 | r10 | 0x200100 | 2097408 | r11 | 0x0 | 0 |
| r12 | 0x1fffcc | 2097100 | sp | 0x1ffff8 | 2097144 | lr | 0x8224 | 33316 | pc | 0x822c | 33324 |
| fps | 0x0 | 0 | cpsr | 0x13 | 19 | | | | | | |

2. Item 2.4

Exercícios 2.4.1

Para compilar o arquivo se usa o comando `gcc nome_do_arquivo.s`, depois de compilado é gerado automaticamente o arquivo `a.out`. Para debug do arquivo se usa o comando `gdb a.out`, para não rodar o programa de uma vez coloca-se um breakpoint na main e nas linhas desejadas, no nosso caso `b main` e depois `b 18`, depois é só rodar o programa com `r` para o código rodar até o breakpoint de linha definida e `s` para ir seguindo o programa e ir parando de linha em linha. Para sair do método debug usar `q`.

```

1  @ gcc item-2-4-1.s
2  @ gdb a.out
3  @ b main
4  @ b 17
5  @ r
6  @ s
7  @ q
8
9      .text
10     .globl main
11     main:
12         MOV r0, #15
13         MOV r1, #20
14         BL firstfunc
15         MOV r0, #0x18
16         LDR r1, =0x20026
17         SWI 0x123456
18     firstfunc:
19         ADD r0, r0, r1
20         MOV pc, lr

```

Print do programa

```

Register group: general
r0      0x1      1      r1      0x1ffff8 2097144
r2      0xffffffff -1      r3      0xa9c8 43464
r4      0x1      1      r5      0x1ffff8 2097144
r6      0x0      0      r7      0x0      0
r8      0x0      0      r9      0x0      0
r10     0x200100 2097408 r11     0x0      0
r12     0x1ffffc 2097100 sp      0x1ffff8 2097144
lr      0x81fc 33276      pc      0x8218 33304
fps     0x0      0      cpsr    0x60000013 1610612755

```

```

B+> 12      MOV      r0, #15
13      MOV      r1, #20
14      BL       firstfunc
15      MOV      r0, #0x18
16      LDR      r1, =0x20026
b+ 17      SWI     0x123456
18      firstfunc:
19          ADD    r0, r0, r1
20          MOV    pc, lr
21
22
23
24
25
26
27

```

Registradores no início do programa

| | | | | | | | |
|-----|------------|---------|----|------|------------|---------|------------|
| r0 | 0x18 | 24 | | r1 | 0x20026 | 131110 | |
| r2 | 0xffffffff | | -1 | r3 | 0xa9c8 | 43464 | |
| r4 | 0x1 | 1 | | r5 | 0x1ffff8 | 2097144 | |
| r8 | 0x0 | 0 | | r9 | 0x0 | 0 | |
| r10 | 0x200100 | 2097408 | | r11 | 0x0 | 0 | |
| r10 | 0x200100 | 2097408 | | sp | 0x1ffff8 | 2097144 | |
| r12 | 0x1fffcc | 2097100 | | sp | 0x1ffff8 | 2097144 | |
| lr | 0x8224 | 33316 | | pc | 0x822c | 33324 | |
| | | | | cpsr | 0x60000013 | | 1610612755 |

| | | | |
|-----|----|-----|--------------|
| B+> | 12 | MOV | r0, #15 |
| B+> | 12 | MOV | r0, #15 |
| | 14 | BL | firstfunc |
| | 15 | MOV | r0, #0x18 |
| | 16 | LDR | r1, =0x20026 |
| b+> | 17 | SWI | 0x123456 |
| B+> | 17 | SWI | 0x123456 |
| | 19 | ADD | r0, r0, r1 |
| | 20 | MOV | pc, lr |
| | 21 | | |
| | 22 | | |
| | 23 | | |
| | 24 | | |
| | 25 | | |
| | 26 | | |
| | 27 | | |

Registradores no final do programa

Exercícios 2.4.2

```

1      @ gcc item-2-4-1.s
2      @ gdb a.out
3      @ b main
4      @ b 18
5      @ r
6      @ s
7      @ q
8
9      .text
10     .globl main
11     main:
12         MOV r0, #15
13         MOV r1, #20
14         BL firstfunc
15     teste:
16         MOV r0, #0x18
17         LDR r1, =0x20026
18         SWI 0x123456
19     firstfunc:
20         ADD r0, r0, r1
21         MOV pc, lr

```

Código com a adição da label “teste”

Com o comando `step` o código roda linha por linha ignorando a label 'teste' que foi incluída e rodando linha por linha até o final do programa. Como pode ser visto na imagem abaixo, o programa conseguiu chegar no comando depois da label "teste".

```

Register group: general
r0      0x23      35
r2      0xffffffff -1
r4      0x1       1
r6      0x0       0
r8      0x0       0
r10     0x200100  2097408
r12     0x1ffffc  2097100
lr      0x8224    33316
fps     0x0       0
r1      0x14      20
r3      0xa9c8    43464
r5      0x1ffff8  2097144
r7      0x0       0
r9      0x0       0
r11     0x0       0
sp      0x1ffff8  2097144
pc      0x8224    33316
cpsr    0x60000013 1610612755

12      MOV     r0, #15
12      MOV     r0, #15
13      MOV     r1, #20nc
14      BL      firstfunc
> 15      MOV     r0, #0x18
17      LDR     r1, =0x20026
18      SWI     0x123456
b+ 19      firstfunc:
20      ADD     r0, r0, r1
20      ADD     r0, r0, r1
21      MOV     pc, lr
23

```

Código rodando com step

Já com o comando `next` o código roda linha por linha até chegar na label 'teste', ao chegar nesta label o código executa tudo que está dentro da label sem seguir passo a passo as rotinas nela definida., como pode ser visto abaixo que não se passou explicitamente pela instrução `LDR r1, =0x20026`, porém pelo valor no registrador r1 pode se ver que ela foi executada.

```

Register group: general
r0      0x18      24
r2      0xffffffff -1
r4      0x1       1
r6      0x0       0
r8      0x0       0
r10     0x200100  2097408
r12     0x1ffffc  2097100
lr      0x8224    33316
fps     0x0       0
r1      0x20026   131110
r3      0xa9c8    43464
r5      0x1ffff8  2097144
r7      0x0       0
r9      0x0       0
r11     0x0       0
sp      0x1ffff8  2097144
pc      0x822c    33324
cpsr    0x60000013 1610612755

12      MOV     r0, #15
13      MOV     r1, #20
14      BL      firstfunc
15      MOV     r0, #0x18
16      teste:
17      LDR     r1, =0x20026
B-> 18      SWI     0x123456
19      firstfunc:
20      ADD     r0, r0, r1
21      MOV     pc, lr
22
23

```

Código rodando com next

Exercícios 2.4.3

Para conseguir imprimir as saídas em diferentes formatos, foi utilizado a convenção:

```
Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),
t(binary), f(float), a(address), i(instruction), c(char) and s(string).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).
The specified number of objects of the specified size are printed
according to the format.
```

Com isso, utilizando o comando ``p`` para ver os valores do registrador `r0`:

```
(gdb) p/d $r0
$1 = 24
(gdb) p/x $r0
$2 = 0x18
(gdb) p/o $r0
$3 = 030
(gdb) p/t $r0
$4 = 11000
```

Esses como se pode ver correspondem todos aos mesmos valores em diferentes formatos.

3. Item 3.10

Exercícios 3.10.1

O registrador CPSR tem o seguinte formato:



Figure 1-3 ARM status register format

Onde N é de negativo, Z de zero, C de carry e V de overflow.

Realizando as contas propostas na apostila neste item obtivemos os seguintes resultados:

Conta 1 (0xFFFF0000 + 0x87654321):

Realizando a conta proposta com a calculadora o resultado obtido foi de 0x187644321 portanto podemos prever as seguintes flags: C e N. Se obteve a flag C, uma vez que o resultado obtido está representado em 33 bits, sendo esse um bit a mais, em relação aos 32 bits, o bit de carry out. Já a flag N foi obtida, pois o resultado da soma, representado em 32 bits, tem o seu bit mais significativo com o valor de 1, indicando que seria um número negativo.

| | | |
|-------------------|---|-----------|
| FFFF0000+87654321 | = | 187644321 |
| | | 187644321 |

Conta calculadora

Quando rodamos o programa abaixo e observamos o registrador CPSR obtivemos o valor de 0xA0000093, ou seja, o binário dos 4 primeiros bits dados por 1010 e portanto temos as flags Negativo e Carry em alto. O que condiz com a nossa previsão inicial.

```
.text
.globl main
main:
    LDR r0, =0xFFFF0000
    LDR r1, =0x87654321
    BL firstfunc
    SWI 0x123456
firstfunc:
    ADDS r0, r0, r1
    MOV pc, lr
```

Programa utilizado

cpsr 0xa0000093

Valor do registrador CPSR

Conta 2 (0xFFFFFFFF + 0x12345678):

Realizando a conta proposta com a calculadora o resultado obtido foi de 0x112345677, portanto, podemos prever as seguintes flags: C. Se obteve essas flags, uma vez que o resultado obtido está representado em 33 bits, sendo esse um bit a mais, em relação aos 32 bits, o bit de carry out.

| | | |
|-------------------|---|-----------|
| ffffffff+12345678 | = | 112345677 |
| | | 112345677 |

Calculadora

Quando rodamos o programa abaixo e observamos o registrador CPSR obtivemos o valor de 0x20000093, ou seja, o binário dos 4 primeiros bits dados por 0010 e portanto temos as flags Carry em alto. O que condiz com a nossa previsão inicial.

```
.text
.globl main
main:
    LDR r0, =0xFFFFFFFF
    LDR r1, =0x12345678
    BL firstfunc
    SWI 0x123456
firstfunc:
    ADDS r0, r0, r1
    MOV pc, lr
```

Programa utilizado

cpsr 0x20000093

Valor do registrador CPSR

Conta 3 (0x67654321 + 0x23110000):

Realizando a conta proposta com a calculadora, o resultado obtido foi de 0x8A764321, portanto, podemos prever as seguintes flags: V e N. A flag N foi obtida, pois o resultado da soma, representado em 32 bits, tem o seu bit mais significativo com o valor de 1, indicando que seria um número negativo. Já a flag V se deve ao fato de o resultado da soma ter extrapolado o valor máximo que se consegue representar em um número de 32 bits, isso pode ser verificado uma vez que se estava somando dois números positivos (com 0 no bit mais significativo) e se obteve um número negativo (com 1 no bit mais significativo).

| | | |
|-------------------|---|----------|
| 67654321+23110000 | = | 8A764321 |
| 8A764321 | | |

Calculadora

Quando rodamos o programa abaixo e observamos o registrador CPSR obtivemos o valor de 0x90000093, ou seja, o binário dos 4 primeiros bits dados por 1001 e portanto temos as flags Negativo e Overflow em altos. O que condiz com a nossa previsão inicial.

```

.text
.globl main
main:
    LDR r0, =0x67654321
    LDR r1, =0x23110000
    BL firstfunc
    SWI 0x123456
firstfunc:
    ADDS r0, r0, r1
    MOV pc, lr

```

Programa utilizado

cpsr 0x90000093

Valor do registrador CPSR

Exercícios 3.10.2

Para este exercício foi necessário fazer a multiplicação de 0xFFFFFFFF por 0x80000000 utilizando-se inicialmente o método MULS e observar o resultado no registrador r2. Utilizando-se da calculadora o valor esperado desta multiplicação é dado por 0x7FFFFFFF80000000.

| | | |
|------------------|---|------------------|
| fffffffx80000000 | = | 7FFFFFFF80000000 |
| 7FFFFFFF80000000 | | |

Conta realizada na calculadora

```

.text
.globl main
main:
    LDR r0, =0xFFFFFFFF
    LDR r1, =0x80000000
    BL firstfunc
    SWI 0x123456
firstfunc:
    MULS r2, r0, r1
    MOV pc, lr

```

Programa utilizado para fazer a conta

Contudo quando observado o resultado no registrador r2 a multiplicação com MULS armazenou em r2 apenas os 32 bits menos significativos, deixando de registrar os outros 32 bits mais significativos, no caso o 0x7FFFFFFF, uma vez que o resultado era grande demais para ser representado somente por 32 bits.

| | |
|----|------------|
| r2 | 0x80000000 |
|----|------------|

Resultado no registrador r2

Por outro lado, utilizando-se a instrução UMULL foi possível fazer a multiplicação registrando o resultado nos registradores r2 e r4, guardando assim o resultado completo corretamente em 64 bits, onde os 32 bits menos significativos foram guardados no registrador r2 e os mais significativos no r4.

```
.text
.globl main
main:
    LDR r0, =0xFFFFFFFF
    LDR r1, =0x80000000
    BL firstfunc
    SWI 0x123456
firstfunc:
    UMULL r2, r4, r0, r1
    MOV pc, lr
```

Programa utilizando o UMULL

| | |
|----|------------|
| r2 | 0x80000000 |
|----|------------|

| | |
|----|------------|
| r4 | 0x7fffffff |
|----|------------|

Resultado nos registradores r2 e r4

Por fim, utilizando-se a instrução SMULL e também armazenando o resultado nos registradores r2 e r4. Como pode-se notar o resultado obtido foi diferente do que se obteve utilizando o UMULL, isso deve ao fato de que o SMULL faz a multiplicação considerando o sinal dos números em complemento de dois.

```
.text
.globl main
main:
    LDR r0, =0xFFFFFFFF
    LDR r1, =0x80000000
    BL firstfunc
    SWI 0x123456
firstfunc:
    SMULL r2, r4, r0, r1
    MOV pc, lr
```

Programa utilizando o SMULL

| | |
|----|------------|
| r2 | 0x80000000 |
| r4 | 0x0 |

Resultado nos registradores r2 e r4

Exercícios 3.10.3

Neste exercício, para realizar a multiplicação por 32 bastas fazer um bitshift de 5 zeros para a esquerda do número desejado. Portanto, para testar este método fizemos a multiplicação do número 3 (registrador r0) por 32 e guardamos o resultado no registrador r1. Como esperado, o resultado no registrador r1 após a operação de bitshift resultou em 96, como apresentado abaixo.

```

8
9      .text
10     .globl main
11     main:
12         LDR r0, =0x3
13         BL  mult_by_32
14         SWI 0x123456
15     mult_by_32:
16         MOV r1, r0, LSL #5
17         MOV pc, lr
18

```

Programa para multiplicação por 32 utilizando Bit shift

| | | |
|----|------|----|
| r0 | 0x3 | 3 |
| r1 | 0x60 | 96 |

Resultado no registrador r1

Exercícios 3.10.4

Neste exercício utilizamos a instrução EOR para realizar o swap entre os registradores r0 e r1. Para isso, primeiramente, fizemos as seguintes operação:

$A = A \oplus B$

$B = A \oplus B$

$A = A \oplus B$

Assim pela lógica booleana temos que:

$$B = A \oplus B \Rightarrow B = A \oplus B \oplus B \Rightarrow B = A \oplus 1 \Rightarrow B = A$$

Além disso temos em:

$$A = A \oplus B \Rightarrow A = A \oplus B \oplus B \Rightarrow A = A \oplus B \oplus A \Rightarrow A = 1 \oplus B \Rightarrow A = B.$$

Desta forma é realizado o swap entre os registradores.

```
9      .text
10     .globl main
11     main:
12         LDR r0, =0xF631024C
13         LDR r1, =0x17539ABD
14         BL  firstfunc
15         SWI 0x123456
16     firstfunc:
17         EOR r0, r0, r1
18         EOR r1, r0, r1
19         EOR r0, r0, r1
20         MOV pc, lr
21
```

Programa utilizado para realizar o Swap

| | | | | | |
|----|------------|------------|----|------------|-----------|
| r0 | 0xf631024c | -164560308 | r1 | 0x17539abd | 391355069 |
|----|------------|------------|----|------------|-----------|

Valor inicial dos registradores

| | | | | | |
|----|------------|-----------|----|------------|------------|
| r0 | 0x17539abd | 391355069 | r1 | 0xf631024c | -164560308 |
|----|------------|-----------|----|------------|------------|

Valor final dos registradores