



Experiência 2

Programming Basics (cap2) + Data

Processing Operations (cap3).

Disciplina : PCS3432 - Laboratório de Processadores
Prof: Jorge Kinoshita
Membros:
José Otávio Brochado Colombini - 9795060
Filipe Penna Cerávolo Soares - 10774009

Introdução	3
Objetivo	3
Planejamento	3
Item-2-2	3
Relatório	4
Exercícios 2.4 da apostila	4
Exercícios 3.10 da apostila	4
Apêndice	4

1.Introdução

Neste experimento estudaremos as operações básicas do ARM através de testes com códigos assembly utilizando o gnu-arm. Verificando os flags e suas manipulações.

2.Objetivo

Realizaremos os exercícios 2.4.1 ao 2.4.3 e 3.10.1 ao 3.10.4

3.Planejamento

Para esta experiência foi necessário instalar e configurar para que esteja operacional o docker disponibilizado para executar os programas para ARM. Além da execução do item-2-2.

a. Item-2-2

O código original na apostila ARM contava com uma operação ADD. Quando era executado o código com tal operação, a soma $20 + 15$, não atualizava as flags do CPSR, que mantinha como '1' a flag Z. Ao mudar para ADDS (que permite a atualização do status do programa no CPSR), as flags foram atualizadas corretamente e então $Z = '0'$.

4. Relatório

a. Exercícios 2.4 da apostila

- i. Copy the code from Building a program on page 2-3 into CodeWarrior.

There are separate functions in CodeWarrior to compile, make, debug and run a program. Experiment with all four and describe what each does.

Para transformar o código assembly para que seja executado utilizamos o comando 'arm-elf-gcc' e para rodá-lo com o debugger utilizamos 'arm-elf-gdb', por fim colocamos como alvo o simulador e carregamos o código para ser executado.

- ii. Debug the code from Building a program on page 2-3. Instead of running the code, step all the way through the code using both the step method and the step in method. What is the difference between the two methods of stepping through the assembly code?

A pergunta se refere a diferença entre step e next no arm-elf-gdb. Enquanto step executa apenas uma linha, seja da main ou de uma função, o next executa o comando até que a próxima linha da main seja alcançada.

Ou seja, todas as linhas de chamada de função são realizadas quando o comando next ("n") é utilizado. Adicionalmente, o apontador de next se perde em algumas estruturas de código. Por isso, como boa prática, sugere-se a adoção do step ("s").

Utilizar a instrução SWI implica um erro, na medida em que o target sim gera um erro para a simulação. Como boa prática, devemos colocar um breakpoint ao final do código da main.

- iii. Sometimes it is very useful to view registers in different formats to check results more efficiently. Run the code from Building a program on page 2-3. Upon completion, view the different formats of r0 and record your results. Specifically, view the data in hexadecimal, decimal, octal, binary, and ASCII.

Dentro do ambiente do gdb, podemos exibir os valores dos registradores de algumas formas diferentes no console de comando. No manual gdb temos as possibilidades de exibição com hexadecimal (p/x), decimal (p/d), octal (p/o), binário (p/t), ASCII (p/c), entre outros.

Na imagem abaixo um exemplo utilizando o código do item-2-2.s

```
(gdb) p/x $r0
$1 = 0x18
(gdb) p/d
$2 = 24
(gdb) p/o
$3 = 030
(gdb) p/t
$4 = 11000
(gdb) p/c
$5 = 24 '\030'
```

b. Exercícios 3.10 da apostila

- i. For the following values of A and B, predict the values of the N, Z, V and C flags produced by performing the operation $A + B$. Load these values into two ARM registers and modify the program created in Building a program on page 2-3 to perform an addition of the two registers. Using the debugger, record the flags after each addition and compare those results with your predictions. When the data values are

signed numbers, what do the flags mean? Does their meaning change when the data values are unsigned numbers?

$$\begin{array}{rcl} 0xFFFF0000 & 0xFFFFFFFF & 0x67654321 & (A) \\ + 0x87654321 & + 0x12345678 & + 0x23110000 & (B) \end{array}$$

Para a primeira operação se espera-se um N e um C, pois não ultrapassará o limite da representação de números negativos. Para a segunda, espera-se um C, pois é um o sinal negativo de 0xFFFFFFFF sendo retirado. Enquanto para a última espera-se um V e um N, pois $0x2 + 0x6$ será pelo menos $0x8$, ou seja extrapolará a representação positiva se tornando uma representação negativa.

Utilizamos o seguinte código:

```
1      .text
2      .globl main
3  main:
4      BL firstfunc
5      BL secondfunc
6      BL thirdfunc
7      SWI 0x123456
8  firstfunc:
9      LDR    r0, =0xFFFF0000
10     LDR    r1, =0x87654321
11     ADDS   r0, r0, r1
12     MOV    pc, lr
13  secondfunc:
14     LDR    r0, =0xFFFFFFFF
15     LDR    r1, =0x12345678
16     ADDS   r0, r0, r1
17     MOV    pc, lr
18  thirdfunc:
19     LDR    r0, =0x67654321
20     LDR    r1, =0x23110000
21     ADDS   r0, r0, r1
22     MOV    pc, lr
```

Para a primeira operação obtivemos as flags $0xa = '1010'$ (Negativo e Carry como esperado)

cpsr

0xa0000013

Na operação 2 temos apenas a flag de carry out. Desta forma o número não se apresenta mais como negativo (como deveria ser).

l**cpsr**

0x20000013

536870931

Na operação 3 temos um overflow de números positivos e consequentemente ele se torna negativo.

l**cpsr**

0x90000013

-1879048173

Dessa forma, obtivemos a seguinte relação de resultados:

A	B	N	Z	C	V	cpsr
0xFFFF0000	0x87654321	VERDADEIRO	FALSO	VERDADEIRO	FALSO	0xa0000013
0xFFFFFFFF	0x12345678	FALSO	FALSO	VERDADEIRO	FALSO	0x20000013
0x67654321	0x23110000	VERDADEIRO	FALSO	FALSO	VERDADEIRO	0x90000013

Observe que todos os resultados previstos foram realizados.

Quando temos signed numbers as flags possuem seu significado original (zero, negativo, overflow (transbordo), carry out). Para unsigned numbers, como a representação é maior em módulo que complemento de 2, o overflow para de ter sentido e ele deve ser observado para flag Carry out e a flag negativo não possui valor.

- ii. Change the ADD instruction in the example code from Building a program on page 2-3 to a MULS. Also change one of the operand registers so that the source registers are different from the destination register, as the convention for multiplication instructions requires. Put 0xFFFFFFFF and 0x80000000 into the source registers. Now rerun your program and check the result.

```

josecolombini@josecolombini-Alienware-15-R2: ~/git/Labpros/gcc-arm
Register group: general
r0      0x80000000 -2147483648
r2      0x80000000 -2147483648
r4      0x1        1
r6      0x0        0
r8      0x0        0
r10     0x200100 2097408
r12     0x1ffffc 2097100
lr      0x8224   33316
fps     0x0        0
r1      0xffffffff -1
r3      0xa9bc   43452
r5      0x1ffff8 2097144
r7      0x0        0
r9      0x0        0
r11     0x0        0
sp      0x1ffff8 2097144
pc      0x822c   33324
cpsr    0xa000013 -1610612717

4      ldr    r1, =0xffffffff
5      ldr    r2, =0x80000000
6      BL     firstfunc
7      SWI    0x123456
8      firstfunc:
9      MULS   r0, r1, r2
> 10     MOV   pc, lr
11
12
13
14

sim process 42 In: firstfunc                               Line: 10   PC: 0x822c

```

1. Does your result make sense? Why or why not?

Não faz sentido, como podemos ver r2 e r1 são negativos por complemento de 2 e o resultado em r0 continua negativo (0x80000000) e pela flag a (negativo e carry out), mas deveria ser positivo

2. Assuming that these two numbers are signed integers, is it possible to overflow in this case?

Sim, pois a multiplicação de dois números de 32 bits necessariamente precisaria de 64 bits para ser representado sem possibilidade de overflow. Portanto, existe possibilidade de overflow por meio da multiplicação.

3. Why is there a need for two separate long multiply instructions, UMULL and SMULL? Give an example to support your answer.

Pois a representação de números signed e unsigned são distintas e a operação deles deve se adequar às suas particularidades.

Neste primeiro caso temos uma multiplicação signed de $0xffffffff = -1$ com ele mesmo, logo o resultado em r3 e r4 é 1.

```

josecolombini@josecolombini-Alienware-15-R2: ~/git/Labpros/gcc-arm
Register group: general
r0      0x1      1
r2      0xffffffff -1
r4      0x1      1
r6      0x0      0
r8      0x0      0
r10     0x200100 2097408
r12     0x1fffc 2097100
lr      0x8224 33316
fps     0x0      0
r1      0xffffffff -1
r3      0x0      0
r5      0x1ffff8 2097144
r7      0x0      0
r9      0x0      0
r11     0x0      0
sp      0x1ffff8 2097144
pc      0x822c 33324
cpsr    0x60000013 1610612755

Ex2E32.s
4      ldr      r1, =0xffffffff
5      ldr      r2, =0xffffffff
6      BL      firstfunc
7      SWI      0x123456
8      firstfunc:
9      SMULL    r4, r3, r1, r2
> 10     MOV     pc, lr
11
12

```

Neste segundo exemplo com unsigned temos a multiplicação de $0xFFFFFFFF = 4294967295$, quando elevado ao quadrado temos aproximadamente $1,8447e19$, que é o valor representado pela união $\{r3, r4\} = 0xffffffff00000001$.

```

josecolombini@josecolombini-Alienware-15-R2: ~/git/Labpros/gcc-arm
Register group: general
r0      0x1      1
r2      0xffffffff -1
r4      0x1      1
r6      0x0      0
r8      0x0      0
r10     0x200100 2097408
r12     0x1fffc 2097100
lr      0x8224 33316
fps     0x0      0
r1      0xffffffff -1
r3      0xffffffff00000001 -2
r5      0x1ffff8 2097144
r7      0x0      0
r9      0x0      0
r11     0x0      0
sp      0x1ffff8 2097144
pc      0x822c 33324
cpsr    0x60000013 1610612755

Ex2E32.s
4      ldr      r1, =0xffffffff
5      ldr      r2, =0xffffffff
6      BL      firstfunc
7      SWI      0x123456
8      firstfunc:
9      UMULL    r4, r3, r1, r2
> 10     MOV     pc, lr
11
12
13
14

```

- iii. Assume that you have a microprocessor that takes up to eight cycles to perform a multiplication. To save cycles in your program, construct an ARM instruction that performs a multiplication by 32 in a single cycle.

Para que tal operação seja feita basta realizar um shift left de 5 bits em uma operação de escrita, como visto no código a seguir com seu resultado:

```

josecolombini@josecolombini-Alienware-15-R2: ~/git/Labpros/gcc-arm
Register group: general
r0      0x1      1
r2      0x20     32
r4      0x1      1
r6      0x0      0
r8      0x0      0
r10     0x200100 2097408
r12     0x1ffffc 2097100
lr      0x81fc   33276
fps     0x0      0
r1      0x1      1
r3      0xa8b0   43184
r5      0x1ffff8 2097144
r7      0x0      0
r9      0x0      0
r11     0x0      0
sp      0x1ffff8 2097144
pc      0x8220   33312
cpsr    0x60000013 1610612755

4      ldr    r1, =0x00000001
5      MOV   r2, r1, LSL #5
6      SWI   0x123456
7
8
9
10
11
12
13
14

```

- iv. The EOR instruction is a fast way to swap the contents of two registers without using an intermediate storage location such as a memory location or another register. Suppose two values A and B are to be exchanged. The following algorithm could be used:

$$A = A \oplus B$$

$$B = A \oplus B$$

$$A = A \oplus B$$

Write the ARM code to implement the above algorithm, and test it with the values of A = 0xF631024C and B = 0x17539ABD. Show your instructor the contents before and after the program has run.

Utilizando o código abaixo:

```
ASM item-3-10-4.s
1      .text
2      .globl main
3      main:
4      LDR    r0, =0xF631024C
5      LDR    r1, =0x17539ABD
6      EOR    r0, r0, r1
7      EOR    r1, r0, r1
8      EOR    r0, r0, r1
9      SWI    0x123456
```

Obtivemos a seguinte resposta para os passos da linha 5 a 8 em ordem de execução:

Register group: general		
r0	0xf631024c	-164560308
r1	0x17539abd	391355069
r2	0xffffffff	-1
r3	0xa9c4	43460
r4	0x1	1
r5	0x1ffff8	2097144

Register group: general		
r0	0xe16298f1	-513632015
r1	0x17539abd	391355069
r2	0xffffffff	-1
r3	0xa9c4	43460
r4	0x1	1
r5	0x1ffff8	2097144

Register group: general		
r0	0xe16298f1	-513632015
r1	0xf631024c	-164560308
r2	0xffffffff	-1
r3	0xa9c4	43460
r4	0x1	1
r5	0x1ffff8	2097144

Register group: general		
r0	0x17539abd	391355069
r1	0xf631024c	-164560308
r2	0xffffffff	-1
r3	0xa9c4	43460
r4	0x1	1
r5	0x1ffff8	2097144

Pudemos observar que os valores dos registradores foram invertidos o que comprova a premissa inicial de que isso ocorreria.