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

Relatório 4 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 4 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

Item B.1

Para observar o valor no registrador sp após a execução do programa, primeiramente atribuímos ao endereço 0x24 os valores de teste 0x06, 0xFC, 0x03 e 0xFF, e para cada um rodamos as seguintes instruções: LDRSB, LDRSH, LDR e LDRB. A partir do gdb podemos observar os seguintes resultados:

Valor no endereço 0x24	Valor carregado no stack pointer com **LDRSB** (signed byte)
0x06	0x06
0xFC	0xFFFFFFFC
0x03	0x03
0xFF	0xFFFFFFFF

Valor no endereço 0x24	Valor carregado no stack pointer com **LDRSH** (signed halfword)
0x06	0x06
0xFC	0x00FC
0x03	0x03
0xFF	0x00FF

Valor no endereço 0x24	Valor carregado no stack pointer com **LDR** (word)
0x06	0x06
0xFC	0xFC
0x03	0x03
0xFF	0xFF

Valor no endereço 0x24	Valor carregado no stack pointer com **LDRB** (unsigned byte)
0x06	0x06
0xFC	0xFC
0x03	0x03
0xFF	0xFF

Item B.2

O modo de cada instrução é o seguinte:

STR r6, [r4,#4] ➔ Pré-indexada

LDR r3, [r12], #6 ➔ Pós-indexada

LDRB r4, [r3,r2]! ➔ Pré-indexada

LDRSH r12, [r6] ➔ Pós-indexada

Item B.3

Para verificar as posições das memórias solicitadas, primeiramente, carregamos os valores em algumas posições pré determinadas da memória, como pode ser visto no código abaixo e então utilizamos o comando `x/1 posição da memória' para ver os valores na memória.

Com o comando STRB pré-indexado carregamos um byte na memória na posição guardada pelo registrador r3 somada com o valor em r4, sem alterar o registrador com o endereço base.

Com o comando LDRB pré-indexado carregamos um byte no r8 da memória na posição guardada em r3 somado com o valor de r4 multiplicado por 8.

Com o comando LDR pós-indexado carregamos uma palavra no r7 da posição de memória de endereço 0x4000 e modificamos o valor de r3 para 0x4020.

Com o comando STRB pós-indexado carregamos um byte no r6 da posição de memória de endereço 0x4020 e modificamos o endereço de r3 para r3 + r4/4 = 0x4028.

```
16      .text
17      .globl main
18  main:
19      LDR r3, =0×4000
20      LDR r4, =0×20
21      LDR r9, =0×FA
22
23      LDR r5, =0×EAE
24      LDR r0, =0×4100
25      STR r5, [r0]
26
27      LDR r10, =0×CACA
28      STR r10, [r3]
29
30      STRB r9, [r3, r4]         @ [[r3] + [r4] = 0×4020] = r9, r3 = 0×4000
31      LDRB r8, [r3, r4, LSL #3] @ r8 = [[r3] + [r4 * 8] = 0×4100], r3 = 0×4000
32      LDR  r7, [r3], r4         @ r7 = [0×4000], r3 = [0×4020]
33      STRB r6, [r3], r4, ASR #2 @ [0×4020] = r6, r3 = r3 + r4/4 = 0×4028
34
35      SWI 0×123456
```

Item B.4

Só é possível fazer o uso do shift no final da operação de load, quando se está trabalhando com o load de palavras ou bytes sem sinal. Não é possível utilizar o shift no final para transferências de bytes com sinal ou halfwords, pois o formato não é suportado.

Item C

Antes de realizar o loop criamos 2 ponteiros (r0 e r1) que apontavam para a primeira posição dos vetores array_a e array_b. Então antes de iniciar o loop em si, definimos o registrador r2 como 0. Na segunda parte criamos o loop que compara r2 com 8 e caso seja maior ou igual o loop se encerra. Dentro do loop definimos que o registrador r3 deve receber o valor de 7 - r2, depois que r4 deve receber o valor da posição de array_b somado com r3 multiplicado por 4 (é feita a multiplicação por 4, por se tratar de vetores de palavras que têm tamanho de 4 bytes) e por fim guardamos na posição de array_a somado com r2, também multiplicado por 4, o valor de r4. O código pode ser visto na próxima página.

```
    .text
    .globl main
main:
    ADR r0, array_a @ r0 recebe o endereço na memória de array_a
    ADR r1, array_b @ r1 recebe o endereço na memória de array_b

    MOV r2, #0 @ i = 0
loop:
    CMP r2, #8 @ i < 8 ?
    BGE done @ se i ≥ 8 → acaba o loop

    RSB r3, r2, #7 @ r3 = 7 - r2
    LDR r4, [r1, r3, LSL #2] @ r4 = array_b[7 - i]; // r4 = [r1 + r3 * 4] // r3 * 4, pois cada palavra tem 4 bytes
    STR r4, [r0, r2, LSL #2] @ array_a[i] = r4; // [r0 + r2 * 4] = r4 // r2 * 4, pois cada palavra tem 4 bytes

    ADD r2, r2, #1 @ i++
    B loop
done:
    SWI 0x123456

array_a:
    .word 0, 0, 0, 0, 0, 0, 0, 0

array_b:
    .word 1, 2, 3, 4, 5, 6, 7, 8
```

2. Experiência

Item 1

Neste item, transformamos o código abaixo em assembly:

$$x = array[5] + y$$

Para isso utilizamos r0 como x e r1 como y, além de alocar um vetor de 25 posições na memória. Então resolvemos o problema das duas formas solicitadas, pós-indexada e pré-indexada.

a) Pós-indexado

Para realizar a soma desejada colocamos o endereço do array no registrador r2, depois adicionamos a este endereço 20 posições, pois queremos a quinta posição do vetor e em cada posição temos uma palavra, ou seja 4 bytes. Dessa forma com o pós-indexamento guardamos em r0 o valor que se encontra no endereço guardado por r3 e por fim somamos r0 com o registrador r1. Para verificar os valores na memória utilizamos `x/6 posição` para verificar 6 posições de memória.

```
    .text
    .globl main
main:
    MOV r0, #0 @ x
    MOV r1, #3 @ y
    ADR r2, array @ r2 recebe o endereço da array

    ADD r3, r2, #20 @ 20 = 5 * 4, 5 pois eé a quinta posição e 4 pois cada palavra tem 4 bytes
    LDR r0, [r3] @ r0 = [r3]
    ADD r0, r0, r1 @ r0 = r0 + r1

    SWI 0x123456

array:
    .word 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

b) Pré-indexado

Para realizar a soma desejada colocamos o endereço do array no registrador r2, depois com o pré-indexamento guardamos em r0 o valor que se encontra no endereço guardado por r2 mais 20 posições, pois queremos a quinta posição do vetor e em cada posição temos uma palavra, ou seja 4 bytes. Por fim, somamos r0 com o registrador r1. Para verificar os valores na memória utilizamos `x/6 posição` para verificar 6 posições de memória.

```
    .text
    .globl main
main:
    MOV r0, #0 @ x
    MOV r1, #3 @ y
    ADR r2, array @ r2 recebe o endereço do array

    LDR r0, [r2, #20] @ r0 = array[5 * 4] // 5 * 4, pois cada palavra tem 4 bytes
    ADD r0, r0, r1 @ r0 = r0 + r1

    SWI 0x123456

array:
    .word 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Item 2

Neste item, transformamos o código abaixo em assembly:

array[10] = array[5] + y

Para isso utilizamos r1 como y, além de alocar um vetor de 25 posições na memória. Então resolvemos o problema das duas formas solicitadas, pós-indexada e pré-indexada.

a) Pós-indexado

Para realizar a soma desejada colocamos o endereço do array no registrador r2, depois adicionamos a este endereço 20 posições, pois queremos a quinta posição do vetor e em cada posição temos uma palavra, ou seja 4 bytes. Dessa forma com o pós-indexamento guardamos em r3 o valor que se encontra no

endereço guardado por r0 e somamos 20 posições ao endereço no r0; Então realizamos a soma de r3 com r1 e guardamos em r3. Por fim utilizamos a instrução store para guardar o valor de r3 na posição guardada por r0 (posição 10 do vetor, uma vez que o r0 já foi modificado quando foi utilizado o load pós-indexado). Para verificar os valores na memória utilizamos `x/11 posição` para verificar 11 posições de memória.

```
    .text
    .globl main
main:
    MOV r1, #3 @ y
    ADR r2, array @ r2 recebe o endereço na memória de array

    ADD r0, r2, #20 @ r0 = array + 5 * 4
    LDR r3, [r0], #20 @ r3 = array[5 * 4] // 5 * 4, pois cada palavra tem 4 bytes // r0 = &array[5] + 5 * 4
    ADD r3, r3, r1 @ r3 = r3 + r1 //r3 = array[5] + y
    STR r3, [r0] @ array[10] = r3

    SWI 0x123456

array:
    .word 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

b) Pré-indexado

Para realizar a soma desejada colocamos o endereço do array no registrador r2. Depois, com a instrução de load pré-indexada, guardamos em r3 o valor que se encontra no endereço r2, somado de 20 posições, pois queremos a quinta posição do vetor e em cada posição temos uma palavra, ou seja 4 bytes. Em seguida realizamos a soma de r3 com r1 e guardamos em r3. Por fim, realizamos o store de r3 na posição guardada por r0 somada de 40 posições, pois queremos a décima posição e em cada posição temos uma palavra. Para verificar os valores na memória utilizamos `x/11 posição` para verificar 11 posições de memória.

```
    .text
    .globl main
main:
    MOV r1, #3 @ y
    ADR r2, array @ r2 recebe o endereço do array

    LDR r3, [r2, #20] @ r3 = array[5 * 4] // 5 * 4, pois cada palavra tem 4 bytes
    ADD r3, r3, r1 @ r3 = r3 + r1 // r3 = array[5] + y
    STR r3, [r2, #40] @ array[10 * 4] = r3

    SWI 0x123456

array:
    .word 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Item 3

Para realizar o loop criamos inicialmente dois arrays e utilizamos os registradores r1 e r2 para receber os endereços desses arrays. Em seguida, criamos um loop com a condição de saída que compara o valor do registrador r3 (r3 representa o nosso i) com 10 e, caso o seu valor seja maior que 10, o loop acaba. Dentro do loop, o código guarda no registrador r5 o valor do array_b na posição i, para acessar essa posição é somado ao endereço base o valor de i (guardado em r3) multiplicado por 4. Em seguida foi adicionado ao r5 o valor de r5 somado com r4 (que representa o valor c da soma). Por fim realizamos um store que armazena no array_a, na posição i, o valor do registrador r5.

```
    .text
    .globl main
main:
    ADR r1, array_a @ r1 recebe o endereço na memória de array_a
    ADR r2, array_b @ r2 recebe o endereço na memória de array_b

    MOV r4, #5 @ c = 5

    MOV r3, #0 @ i = 0
loop:
    CMP r3, #10 @ i ⩽ 10 ?
    BGT done @ se i ≥ 10 → acaba o loop

    LDR r5, [r2, r3, LSL #2] @ r5 = array_b[i]; // r5 = [r2 + r3 * 4] // r3 * 4, pois cada palavra tem 4 bytes
    ADD r5, r5, r4 @ r5 = r5 + r4
    STR r5, [r1, r3, LSL #2] @ array_a[i] = r5; // [r1 + r3 * 4] = r5 // r3 * 4, pois cada palavra tem 4 bytes

    ADD r3, r3, #1 @ i++
    B loop
done:
    SWI 0x123456

array_a:
    .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

array_b:
    .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
```

Item 4

Neste item foi desenvolvido um loop para zerar as “s” posições de um vetor denominado array. Esse procedimento foi feito de duas formas, acessando as posições do vetor por meio de índices e utilizando ponteiros para cada posição da memória do vetor.

a) Com índices

Para zerar os bytes do array utilizando os índices, precisamos criar um loop sobre os valores de índices do array, acessando todos os elementos e zerando cada um deles. Para isso, primeiramente atribuímos ao registrador r1 o endereço do array e criamos um índice de verificação do loop no registrador r3. Em seguida, comparamos o valor do índice (r3) com o r2 (registrador que guarda o número de posições a serem zeradas do array) e, enquanto r2 for menor que r3, o loop é executado. Dentro do loop fazemos um store em que colocamos no endereço r1 + r3 (endereço da posição i do array) o valor do registrador r4, que no caso possui o valor de 0. Por fim, adicionamos 1 ao valor do índice, o r3.

```
    .text
    .globl main
main:
    ADR r1, array @ r1 recebe o endereço na memória do array

    MOV r2, #10

    MOV r3, #0 @ i = 0
loop:
    CMP r3, r2 @ i < s ?
    BGE done @ se i ≥ s → acaba o loop

    MOV r4, #0 @ Registrador auxiliar para zerar o array
    STRB r4, [r1, r3] @ array[i] = r4; // [r1 + r3 ] = r4

    ADD r3, r3, #1 @ i++
    B loop
done:
    SWI 0x123456

array:
    .byte 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

b) Com ponteiros

Para zerar os bytes do array utilizando ponteiros, precisamos criar um loop sobre os endereços que o array ocupa, acessando cada posição da memória e zerando cada uma delas. Para isso, primeiramente, atribuímos ao registrador r1 o endereço do array, definimos um registrador que recebe o endereço do começo do array, para ser utilizado como iterador, no caso o r3, e um registrador, o r4, para receber o endereço do final do array. Em seguida, comparamos o valor do endereço guardado no iterador (r3) com o r4, registrador que guarda o valor do endereço final, e enquanto r3 fosse menor que r4, o loop era executado. Já dentro do loop realizamos um store em que colocamos na posição da memória guardada por r3 o valor do registrador r5, que, no caso, possui o valor 0.

```
    .text
    .globl main
main:
    ADR r1, array @ r1 recebe o endereço na memória do array

    MOV r2, #10 @ s = 10

    MOV r3, r1 @ r3 = r1 = &array[0], r3 = endereco para o comeco da array
    ADD r4, r3, r2 @ r4 = r3 + r2 = &array[s], r4 = endereco para o final da array
loop:
    CMP r3, r4 @ &array[0] < &array[s] ?
    BGE done @ se i ≥ s → acaba o loop

    MOV r5, #0 @ Registrador auxiliar para zerar o array
    STRB r5, [r3], #1 @ *p = r5,  p++

    B loop
done:
    SWI 0×123456

array:
    .byte 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Item 5

Neste item foi desenvolvido um programa para fazer os cálculos da sequência de fibonacci guardando todos os resultados na memória. Primeiramente fizemos utilizando um vetor de bytes e depois com um vetor de palavras.

a) Byte

Para fazer os cálculos da sequência de fibonacci guardando todos os resultados na memória, primeiramente fazemos store do valor de f(0) e f(1) nas duas primeiras posições do array, colocamos em r3 a quantidade n de números desejados na sequência e em r4 alocamos o índice atual de n, ou seja 2, já que f(0) e f(1) já foram guardado na memória. Em seguida, realizamos o loop principal que é executado enquanto o valor do índice r4 for menor ou igual ao valor do r3. Dentro do loop realizamos o load de um byte em r0 e r1 os valores do array nas posições n - 2 e n - 1 respectivamente, em seguida somamos os valores de r0 e r1 em r7 e fazemos um store de um byte do resultado na posição de índice r4 do array. Por fim, no final do loop, somamos 1 ao valor do índice.

```
    .text
    .globl main
main:
    MOV r0, #0 @ f(0) = 0
    MOV r1, #1 @ f(1) = 1

    ADR r2, array @ r0 recebe o endereço na memória do array

    STRB r0, [r2] @ array[0] = f(0)
    STRB r1, [r2, #1] @ array[1] = f(1)

    MOV r3, #12 @ Deve-se calcular os 12 primeiros números da sequência

    MOV r4, #2 @ n = 2
loop:
    CMP r4, r3 @ n < 12 ?
    BGT done @ se n ≥ 12 → acaba o loop

    SUB r6, r4, #2
    SUB r7, r4, #1
    LDRB r0, [r2, r6] @ r0 = f(n - 2)
    LDRB r1, [r2, r7] @ r1 = f(n - 1)
    ADD r7, r0, r1 @ r7 = f(n - 1) + f(n - 2)
    STRB r7, [r2, r4] @ array[r4] = r7

    ADD r4, r4, #1 @ n++
    B loop
done:
    SWI 0x123456

array:
    .byte 0,0,0,0,0,0,0,0,0,0,0,0
```

b) Word

O algoritmo para fazer o cálculo da sequência de fibonacci utilizando palavras é semelhante ao de bytes, porém ao invés das instruções STRB e LDRB se utiliza as instruções STR e LDR. Além disso, ao utilizar essas instruções é necessário multiplicar os valores de índices por 4, uma vez que cada palavra é formada por 4 bytes e o endereçamento da memória é feito por bytes.

```
    .text
    .globl main
main:
    MOV r0, #0 @ f(0) = 0
    MOV r1, #1 @ f(1) = 1

    ADR r2, array @ r0 recebe o endereço na memória de array

    STR r0, [r2] @ array[0] = f(0)
    STR r1, [r2, #4] @ array[1] = f(1)

    MOV r3, #65 @ Deve-se calcular os n primeiros números da sequência

    MOV r4, #2 @ n = 2
loop:
    CMP r4, r3 @ n < r3 ?
    BGT done @ se n ≥ r3 → acaba o loop

    SUB r6, r4, #2
    SUB r7, r4, #1
    LDR r0, [r2, r6, LSL #2] @ r0 = f(n - 2)
    LDR r1, [r2, r7, LSL #2] @ r1 = f(n - 1)
    ADD r7, r0, r1 @ r7 = f(n - 1) + f(n - 2)
    STR r7, [r2, r4, LSL #2] @ array[r4] = r7

    ADD r4, r4, #1 @ n++
    B loop
done:
    SWI 0x123456

array:
    .word 0,0,0,0,0,0,0,0,0,0,0,0
```

Para testar o algoritmo de cálculo da sequência de fibonacci, foi utilizado os dois últimos algarismos do número USP dos integrantes, como pode ser visto abaixo:

```
0x8260 <array>: 0       1       1       2
0x8270 <array+16>:      3       5       8       13
0x8280 <array+32>:      21      34      55      89
0x8290 <atexit>:        144     233     377     610
0x82a0 <atexit+16>:     987     1597    2584    4181
0x82b0 <atexit+32>:     6765    10946   17711   28657
0x82c0 <atexit+48>:     46368   75025   121393  196418
0x82d0 <atexit+64>:     317811  514229  832040  1346269
0x82e0 <atexit+80>:     2178309 3524578 5702887 9227465
0x82f0 <atexit+96>:     14930352        24157817        39088169        63245986
0x8300 <atexit+112>:    102334155       165580141       267914296       433494437
0x8310 <atexit+128>:    701408733       1134903170      1836311903      -1323752223
0x8320 <$d>:    512559680       -811192543      -298632863      -1109825406
0x8330 <exit+12>:       -1408458269     1776683621      368225352       2144908973

0x8340 <exit+28>:       -1781832971     363076002       -1418756969     -1055680967
0x8350 <exit+44>:       1820529360      764848393       -1709589543     -944741150
0x8360 <exit+60>:       1640636603
```

Para número USP com final 65

```
(gdb) x/14d array
0x8260 <array>: 0       1       1       2
0x8270 <array+16>:      3       5       8       13
0x8280 <array+32>:      21      34      55      89
0x8290 <atexit>:        144     233
```

Para número USP com final 14

É possível observar que, como se está utilizando vetores de palavras de 32 bits, há um número limite que se consegue representar, então há um número máximo de elementos que se consegue representar na sequência utilizando-se somente números de 32 bits. Dessa forma só se consegue representar os números da sequência para valores de n menores ou iguais a 46, por isso que se pode ver que para o teste com número USP com final 65, a partir de n = 47 os valores guardados na memória não correspondem mais aos valores da sequência.

Item 6

Neste item realizamos os cálculos da sequência de fibonacci guardando somente o último resultado no registrador r0 e guardando no registrador r1 qual a ordem do termo desejado (n). Para isso, primeiramente colocamos em r2 e r3 os 2 primeiros valores da sequência de fibonacci, no caso 0 e 1 respectivamente, em seguida realizamos uma comparação entre o valor de n (registrador r1) com 1 e caso seja igual, colocamos o valor de r3 em r0. Para iniciarmos o loop, definimos r4, o n, com o valor de 2, uma vez que só faz sentido rodar o loop para valores de n maiores ou iguais a 2. Em seguida definimos que o loop principal é executado enquanto o valor do índice em r4 for menor ou igual ao valor do n máximo em r1. Dentro do loop realizamos a adição de f (n-2) com f (n-1), que estão armazenados nos registradores r3 e r4 respectivamente, e guardamos o resultado em r0. Em seguida colocamos os valores de f(n - 1) em f(n - 2) e o valor de f(n) em f(n-1), para isso é feito com r2 = r3 e r3 = r0, isso é necessário para que na próxima iteração do loop os valores de f(n - 1) e f(n - 2) estejam atualizados para o valor de n atualizado. Por fim, somamos 1 ao valor do índice r4. O código pode ser visto na próxima página.

```
    .text
    .globl main
main:
    MOV r0, #0 @ f(n) = 0
    MOV r1, #35 @ n = 6

    MOV r2, #0 @ f(0) = 0
    MOV r3, #1 @ f(1) = 1

    CMP r1, #1
    MOVEQ r0, r3

    MOV r4, #2 @ i = 2
loop:
    CMP r4, r1 @ i < n ?
    BGT done @ se i ≥ n → acaba o loop

    ADD r0, r2, r3 @ r0 = f(n - 1) + f(n - 2)

    MOV r2, r3 @ f(n - 2) = f(n - 1)
    MOV r3, r0 @ f(n - 1) = f(n)

    ADD r4, r4, #1 @ n++
    B loop
done:
    SWI 0x123456
```