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

Relatório 5 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 5 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. Experiência **3**

Exercício 5.5.1 3

Exercício 5.5.2 4

Exercício 5.5.3 5

Exercício 5.5.4 7

Exercício 5.5.5 9

1. Experiência

Exercício 5.5.1

Neste exercício, foi solicitado que se traduzisse o loop abaixo em C para assembly.

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

Código em C

Para isso fizemos o código em assembly abaixo. Nesse código, 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, que servirá como índice do vetor. Em seguida, já dentro do loop, fizemos uma comparação de r2 com 8, para, caso r2 seja maior ou igual, o loop se encerrar. Dentro do loop então também, 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.

```assembly
    .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
```

Código desenvolvido

Ao testar o código, podemos comprovar o funcionamento, ao ver os valores nas posições da memória do array_a e array_b antes e depois da execução do loop no programa.

```
(gdb) x/8 array_a

0x8244 <array_a>:       0x00000000      0x00000000      0x00000000      0x00000000
0x8254 <array_a+16>:    0x00000000      0x00000000      0x00000000      0x00000000
(gdb) x/8 array_b
0x8264 <array_b>:       0x00000001      0x00000002      0x00000003      0x00000004
0x8274 <array_b+16>:    0x00000005      0x00000006      0x00000007      0x00000008
```

Antes da execução do loop

```
(gdb) x/8 array_a
0x8244 <array_a>:       0x00000008      0x00000007      0x00000006      0x00000005
0x8254 <array_a+16>:    0x00000004      0x00000003      0x00000002      0x00000001
(gdb) x/8 array_b
0x8264 <array_b>:       0x00000001      0x00000002      0x00000003      0x00000004
0x8274 <array_b+16>:    0x00000005      0x00000006      0x00000007      0x00000008
```

Depois da execução do loop

Exercício 5.5.2

Para realizar o cálculo de um número fatorial, primeiramente, colocamos em r6 o valor n que deverá ser calculado o fatorial e fizemos uma cópia do valor dele em r5 (que será onde o resultado será armazenado) para iniciar os cálculos. Em seguida é realizado o loop principal, o qual é executado enquanto o valor em r4, que representa o n inicial, for maior que 0, e a cada iteração do loop calculamos e colocamos em r7 o valor de r4 * r5, que representa a multiplicação do n atual do loop pelo produto acumulado das multiplicações de iterações anteriores do loop. Por fim guardamos o resultado no registrador r5. O código pode ser visto abaixo.

```
    .text
    .globl main
main:
    MOV r6, #10 @ Valor n

    MOV r5, r6 @ Resultado

    MOV r4, r6 @ i = n
loop:
    SUBS r4, r4, #1
    MULNE r7, r4, r5 @ r7 = r5 * (n - 1)
    MOVNE r5, r7 @ r7 = r5
    BNE loop

    @ Depois do fim do loop
    SWI 0x123456
```

Código desenvolvido

Testando-se o código para n = 10, se obteve o valor de 3628800, no registrador r5, que era o valor esperado para 10!. O resultado pode ser visto abaixo.

```
r5              0x375f00 3628800
```

Resultado para n = 10

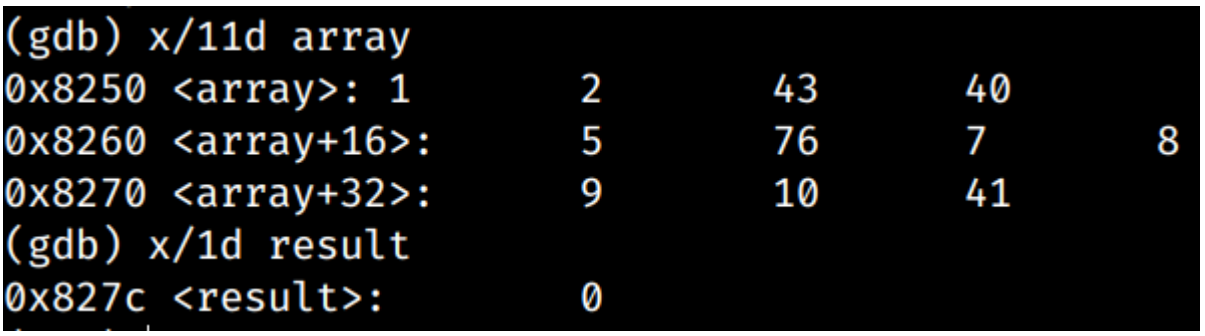
Exercício 5.5.3

Para calcular qual o valor máximo dentre todos os elementos de um array, devemos realizar um loop que é executado enquanto o iterador do loop (registrador r2) for menor que o tamanho do array. Dentro do loop realizamos um load em r4 do valor do array na posição do índice r2. Em seguida é comparado o valor de r4 com r7, sendo r7 o resultado temporário do maior valor do array e caso o valor de r4 for maior que r7, r7 recebe o valor de r4, em seguida o índice é incrementado de 1 e por fim fazemos um store de r7 no endereço da memória guardado por r0.

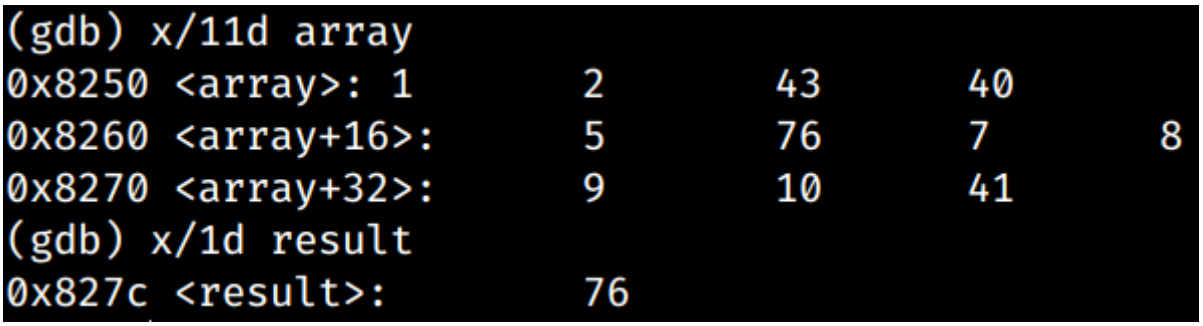
```
    .text
    .globl main
main:
    ADR r0, result @ r0 recebe o endereço na memória de 'result'
    ADR r1, array @ r1 recebe o endereço na memória de 'array'
    MOV r5, #11 @ Tamanho do array

    MOV r7, #0 @ Resultado temporário

    MOV r2, #0 @ i = 0
loop:
    CMP r2, r5 @ i < r5 ?
    BGE done @ se i ≥ r5 → acaba o loop

    LDR r4, [r1, r2, LSL #2] @ r4 = array[r2]; // r4 = [r1 + r2 * 4]

    CMP r4, r7 @ r4 < r7 ?
    MOVGT r7, r4 @ se r4 > r7, r7 = r4

    ADD r2, r2, #1 @ i++
    B loop
done:
    STR r7, [r0] @ result = r7

    SWI 0x123456

array:
    .word 1, 2, 43, 40, 5, 76, 7, 8, 9, 10, 41

result:
    .word 0
```

Código desenvolvido

Ao testar o código, podemos comprovar o funcionamento, ao ver os valores nas posições da memória do array e result antes e depois da execução do loop no programa.



Antes da execução do loop



Depois da execução do loop

Exercício 5.5.4

Para realizar o reconhecimento de uma sequência Y em uma entrada X de 32 bits desenvolvemos o código abaixo. No código inicialmente definimos o registrador r1 para guardar a entrada X de 32 bits, o registrador r2 para guardar quando se identifica a sequência Y na entrada X, o registrador r8 para a sequência Y de até 32 bits e por fim o registrador r9 para o tamanho da sequência Y.

A partir desses registradores foi criado um loop que é executado enquanto o índice r4 for menor ou igual ao r3, sendo que r3 representa 32 bits subtraído da quantidade de bits da sequência Y.

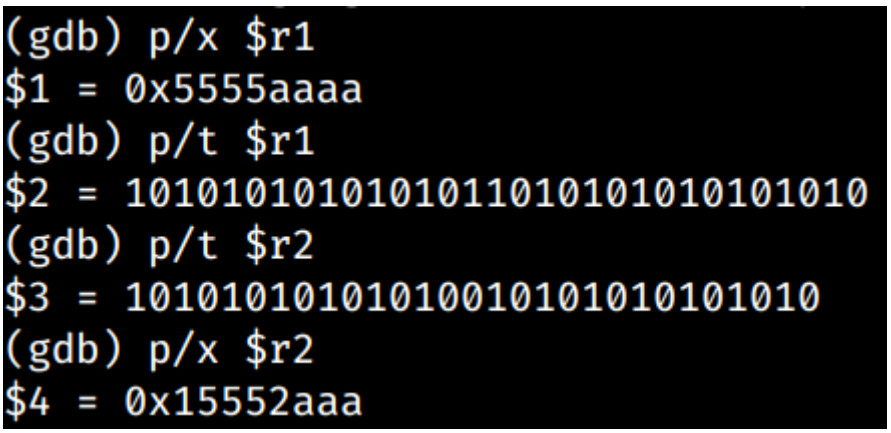
Dentro do loop são executados 2 shift, primeiramente um shift para esquerda de r4 posições para eliminar os números à esquerda da entrada X que não serão utilizados para comparação nesse loop, em seguida é realizado um shift para a direita de r3 posições para eliminar os bits da direita da entrada X que não serão utilizados para comparação nesse loop, dessa forma guardamos em r6 uma

sequência de bits da entrada X do mesmo tamanho da sequência Y, que será comparada com a sequência Y definida em r8. Em seguida, realizamos uma comparação entre a sequência Y e a parte da entrada X em r6, dessa forma caso os valores sejam equivalentes colocamos 1 no registrador r7, caso contrário deixamos r7 como 0. Então, fizemos um shift para a esquerda em r2, para preservar a informação de reconhecimento da sequência em posições anteriores e somamos r2 com r7 para guardar se foi reconhecida a sequência na posição atual. Por fim adicionamos 1 no iterador em r4.

```
    .text
    .globl main
main:
    LDR r1, =0x5555AAAA @ Entrada X
    MOV r2, #0 @ Saída Z
    LDR r8, =0b101 @ Sequência Y
    MOV r9, #3 @ Tamanho da sequência Y

    RSB r3, r9, #32 @ r3 → máximo que se pode shiftar
    MOV r4, #0
loop:
    CMP r4, r3
    BGT done

    MOV r6, r1, LSL r4 @ Corta os bits que não vão ser comparados da esquerda
    MOV r6, r6, LSR r3 @ Corta os bits que não vão ser comparados da direita

    CMP r6, r8
    MOVEQ r7, #1 @ r2 = 1, se r6 == r8
    MOVNE r7, #0 @ r2 = 0, se r6 ≠ r8

    @ Guarda o resultado atual da saída em r2
    MOV r2, r2, LSL #1 @ r2 = r2 << 1
    ADD r2, r2, r7 @ r2 = r2 + r7

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

Código desenvolvido

Com isso, obtivemos o resultado abaixo, que mostra a entrada em r1 e a saída em r2, mostrando ambos em hexadecimal e em binário.



Resultado obtido

Exercício 5.5.5

Neste exercício, foi solicitado que se fizesse a inspeção da paridade de um número guardado em r0 e então guardasse 1 em r1 caso fosse paridade ìmpar e 0 caso contrário.

Para isso então se utilizou um loop para percorrer todos os bits do número em r0 e realizar a operação de “ou exclusivo” entre todos eles. Já para se conseguir pegar o valor de cada bit foi utilizada uma máscara, que começa com 0x80000000 e então é deslocada para a direita para conseguir pegar diferentes posições de bits. Dessa forma então se tem um loop com um iterador que vai de 31 a 0, para varrer todos os 32 bits de r0, sendo que para pegar cada um desses bits se faz um AND da máscara com r0. Então, tendo-se o valor do bit, se faz um shit para a direita para que esse bit fique na casa menos significativa, em seguida se faz um EOR desse bit com o resultado atual em r1, dessa forma se faz o EOR com todos os bits. Ainda dentro do loop, por fim, é deslocada a máscara para a direita e é subtraído 1 do valor em r3, que é o iterador. O código desenvolvido pode ser visto abaixo.

```
    .text
    .globl main
main:
    LDR r0, =0b1100000100001 @ Entrada de numero a ser verificada
    MOV r1, #0 @ Resultado da paridade de r0, 1 se ímpar e 0 se par

    LDR r2, =0x80000000 @ Máscara para pegar bit a bit

    MOV r3, #31
loop:
    CMP r3, #0
    BLT done

    AND r4, r0, r2 @ r4 = r0 && r2

    EOR r1, r1, r4, LSR r3 @ r1 = r1 xor (r4 >> r3)

    MOV r2, r2, LSR #1 @ Rotaciona a máscara um bit para a direita

    SUB r3, r3, #1
    B loop
done:
    SWI 0x123456
```

Código desenvolvido

Para testar o programa, se utilizou valores diferentes de r0, como 0b1100000100001 e 0b1100100100001, obtendo-se os resultados esperados para cada um, como pode ser visto nas imagens abaixo.

```
(gdb) p/t $r0
$1 = 1100000100001
(gdb) p/t $r1
$2 = 0
```

Resultado com número com paridade par

```
(gdb) p/t $r0
$1 = 1100100100001
(gdb) p/t $r1
$2 = 1
```

Resultado com número com paridade ímpar