

Lista 3 – Felipe Melo – Thalles Nonato

DRE Felipe: 119093752

DRE Thalles: 119058809

Questão 1) a) O que é armazenado inicialmente em `%ecx` e em `%eax`?

Em `%ecx` é armazenado o parâmetro `n` e em `%eax` é armazenado o parâmetro `x`.

Questão 1) b) Qual valor é carregado em `%edx` na linha 4? Explique a razão deste valor:

É carregado o valor em $(\%ecx + 15)$, ou seja, $(n + 15)$. A razão para isso é para entrar adequadamente em cada case do switch. Para isso, nosso jump computa $(4 * |n + 15|)$, mas devemos dividir esse resultado por $\text{sizeof}(\text{int}) = 4$, resultando em $|n + 15|$, que nos levará para o respectivo statement do switch.

Questão 1) c) O que faz a linha 5? Explique a razão desta operação:

Compara subtraindo 30 do valor de $(\%edx = n + 15)$. Isso ocorre pois 30 é o intervalo de instruções .long do código Assembly. No código C, como o valor do nosso menor case é -15 e o maior é 15, o tamanho do intervalo $[-15, 15]$ é 30. No entanto, nem toda instrução nesse intervalo é usada e, por isso, o código Assembly desvia esses casos para o default (.L8).

Questão 1) d) Quais condições provocam o desvio para .L8? Explique a instrução `ja`:

Se $(|n + 15| > 30)$, ocorre o desvio para .L8. A instrução `ja` realiza um desvio caso a magnitude do termo $(n + 15)$ for maior que 30.



Questão 1) e) Explique a linha 7 e a necessidade do `notrack`:

`notrack` é um prefixo usado para diminuir o número de instruções `endbr` para realizar saltos indiretos que derivam o endereço de destino de ramificação de informações que não podem ser modificadas.

Questão 1) f) Escreva o código C completo da rotina `ex`, inclusive preenchendo o comando `switch` corretamente. O único `return` na rotina está mostrado no código C. Atenção e não insira `return` dentro do `switch`:

```
1 int ex(int x, int n)
2 {
3     switch(n)
4     {
5         case -15:           // 0 - 15 (.L7)
6             x *= 15;
7
8         case -10:           // 5 - 15 (.L6)
9             x *= x * (-10);
10            break;
11
12        case 0:              // 15 - 15 (.L9)
13            x = (x >> 31) + x;
14            break;
15
16        case 5:              // 20 - 15 (.L3)
17        case 10:             // 25 - 15 (.L3)
18        case 15:             // 30 - 15 (.L3)
19            x = (x * x);
20            break;
21
22        default:             // (.L8)
23            x = 0;
24
25    } x = x / 2;
26
27    return x;
28
29 }
```

Questão 2) a) Fazendo engenharia reversa no código de montagem, preencha as lacunas do código C. Todas as linhas devem ser preenchidas e nenhuma acrescentada. Reticências podem conter expressões e não apenas um valor, uma variável ou um operador. Atenção aos tipos dos parâmetros de `rot`:

```
1 void rot (int x, char *p) {
2     int i, j;
3     for (i = j = 0 ; p[i] != '\0' ; i++) { // percorre p até o fim
4         if (p[i] != x) // se p[i] for diferente de x
5             p[j++] = p[i]; // copia para p, deletando x
6         p[j] = '\0'; // finaliza com o caractere de término
7     }
8 }
```

Questão 2) b) Faça a engenharia reversa, comentando todas as linhas do código de montagem. É preciso associar ao C, e não indicar simplesmente o que a instrução faz, que sabemos. Justifique a presença de instrução no código de montagem se não estiver diretamente relacionada com o C:

```

rot:
1.  pushl  %edi
2.  pushl  %esi
3.  pushl  %ebx
4.  movl   16(%esp), %edi
5.  movl   20(%esp), %esi
6.  movzbl (%esi), %eax
7.  testb  %al, %al
8.  je     .L5
9.  leal   1(%esi), %edx
10. movl   $0, %ecx
11. jmp    .L4
.L3:
12. addl   $1, %edx
13. movzbl -1(%edx), %eax
14. testb  %al, %al
15. je     .L2
.L4:
16. movsbl %al, %ebx
17. cmpl   %edi, %ebx
18. je     .L3
19. movb   %al, (%esi, %ecx)
20. leal   1(%ecx), %ecx
21. jmp    .L3
.L5:
22. movl   $0, %ecx
.L2:
23. movb   $0, (%esi, %ecx)
24. popl   %ebx
25. popl   %esi
26. popl   %edi
27. ret

```

L1: salva %edi na pilha

L2: salva %esi na pilha

L3: salva %ebx na pilha

L4: pega o parâmetro x

L5: pega o parâmetro p

L6: copia o parâmetro p para %eax. Com essa instrução, sabemos que p é ponteiro para char, pois `sizeof(char) = 1` e a instrução move 1 byte para um long (movzbl).

L7: realiza um AND bit a bit de %eax com %eax

L8: se o último byte de %eax for zero, ou seja, se chegou ao fim da lista, desvia para .L5

L9: realiza (%esi + 1), ou seja, (p + 1), que é o próximo caractere

L10: realiza (j = 0)

L11: faz desvio incondicional para .L4

L12: incrementa (i = i + 1)

L13: pega p[i] após o incremento

L14: realiza um AND bit a bit de %eax com %eax

L15: se p[i] = '\0', desvia para .L2

L16: copia p[i] para %ebx

L17: compara x com p[i]

L18: se (x = p[i]), desvia para .L3

L19: copia de %al o caractere para p[j]

L20: incrementa (j = j + 1)

L21: faz desvio incondicional para .L3
 L22: zera a variável j
 L23: realiza $p[j] = 0$
 L24: restaura `%ebx`
 L25: restaura `%esi`
 L26: restaura `%edi`
 L27: retorna

Questão 3) a₁) Comente todas as linhas do código, associando ao código C:

```

calc:
.LFB0:
    1.  endbr32
    2.  pushl  %ebp
    3.  movl  %esp, %ebp
    4.  cmpl  $10, 8(%ebp)
    5.  jle   .L2
    6.  movl  8(%ebp), %eax
    7.  sall  $2, %eax
    8.  movl  %eax, n
    9.  jmp   .L3

.L2:
    10. incl  8(%ebp)
    11. movl  8(%ebp), %eax
    12. movl  %eax, n

.L3:
    13. movl  n, %eax
    14. popl  %ebp
    15. ret
  
```

L2 e L3: cria registro de ativação
 L4: compara x com 10, computando a expressão $(x - 10)$
 L5: se $(x \leq 10)$, desvia para .L2
 L6: copia o parâmetro x para o registrador `%eax`
 L7: realiza $x = 4 * x$
 L8: faz $(n = 4 * x)$
 L9: faz desvio incondicional para .L3
 L10: realiza `++x`
 L11: copia `++x` para `%eax`
 L12: copia x para n
 L13: copia o valor n para `%eax`, preparando o retorno da função
 L14: restaura a base `%ebp`
 L15: retorna n

Questão 3) a₂) Indique o custo (? m) + (? n) desta compilação. Comente sobre as ineficiências que observar:

$7m + 7n$. São 14 instruções, nas quais metade delas acessa a memória, o que diminui a velocidade de execução, tornando ineficiente.

Questão 3) b₁) Comente as linhas do código, associando ao código C:

calc:

.LFB0:

1. endbr32
2. movl 4(%esp), %edx
3. leal 1(%edx), %eax
4. cmpl \$10, %edx
5. jle .L3
6. leal 0(, %edx, 4), %eax

.L3

7. movl %eax, n
8. ret



L2: pega o parâmetro x e salva em %edx

L3: move ($x = x + 1$) para %eax

L4: compara x com 10, computando a expressão ($x - 10$)

L5: se ($x \leq 10$), desvia para .L3

L6: copia ($\%edx * 4$), ou seja, ($x * 4$), para %eax

L7: realiza $n = x$

L8: retorna

Questão 3) b₂) Indique o custo (? m) + (? n) desta compilação. Compare com o anterior sem otimização. Cite as principais alterações observadas que favorecem o desempenho:

$3m + 4n$. Além de acessar menos vezes a memória, possibilitando uma execução mais rápida, possui menos instruções.

Questão 4) a) Comente cada linha, justifique sua existência e associe ao código C. Identifique o uso de ponteiros para acesso facilitado aos elementos das matrizes:

```

1  fix_prod_ele:
2
3
4  .LFB0:
5
6  endbr32
7  pushl %esi
8  pushl %ebx
9  movl 20(%esp), %eax // pega i
10 xorl %ebx, %ebx // zera ebx
11 movl 12(%esp), %edx // pega A
12 movl 24(%esp), %esi // pega K
13 leal (%eax,%eax,2), %eax // (2*i + i) = 3i
14 leal (%edx,%eax,8), %eax // A + 24i // A[i][0]
15 leal 0(%esi,4), %ecx // grava 4*k no registrador
16 addl 16(%esp), %ecx // B + 4k
17 leal 24(%eax), %esi // grava A + 24i + 24 A[i+1][0]
18
19 .L2:
20 movl (%eax), %edx // A[i][0]
21 imull (%ecx), %edx // multiplica A[i][0] * B[0][k]
22 addl $4, %eax // A+24i + 4 // A[i][1]
23 addl $24, %ecx // B+ 4k + 24 // B[1][k]
24 addl %edx, %ebx // faz as somas das matrizes (A[i][0] * B[0][k]) com 0
25 cmpl %esi, %eax // compara A[i+1][0] com A[i][1]
26 jne .L2 // desvia para .L2 se as matrizes não forem iguais
27 movl %ebx, %eax // %eax = (A[i][5] * B[5][k]) + (A[i][4] * B[4][k]) + (A[i][3] * B[3][k]) + (A[i][2] * B[2][k]) + (A[i][0] * B[0][k]) + (A[i][1] * B[1][k])
28 popl %ebx // restaura %ebx
29 popl %esi // restaura %esi
30 ret // retorna eax
31
32

```


Questão 4) b) Feita a otimização -O3, comente cada linha, justifique sua existência e associe ao código C:

```

1  fix_prod_ele:
2  endbr32
3  pushl %edi // salva registrador
4  pushl %esi // salva registrador
5  pushl %ebx // salva registrador
6  movl 24(%esp), %eax // pega i
7  movl 28(%esp), %edi // pega k
8  movl 20(%esp), %ecx // pega B = B[0][0]
9  leal (%eax,%eax,2), %edx // grava 3i em edx
10 movl 16(%esp), %eax // pega A = A[0][0]
11 leal 0(%edi,4), %esi // grava 4k em esi
12 leal (%eax,%edx,8), %edx // A + 24i // A[i][0]
13 movl 24(%ecx,%esi), %eax // B + 4k + 24 = B[1][k]
14 imull 4(%edx), %eax // A+24i + 4 * B[1][k] = A[i][1] * B[1][k]
15 movl %eax, %ebx // %ebx grava A[i][1]
16 movl (%ecx,%edi,4), %eax // B+4k = B[0][k]
17 imull (%edx), %eax // A[i][0] * B[0][k]
18 addl %ebx, %eax // (A[i][0] * B[0][k]) + A[i][1]
19 movl 48(%ecx,%esi), %ebx // 4k + B + 48 = B[2][k]
20 imull 8(%edx), %ebx // (A+ 24i + 8) * B[2][k] / A[i][2] * B[2][k]
21 addl %ebx, %eax // (A[i][2] * B[2][k]) + (A[i][0] * B[0][k]) + A[i][1]
22 movl 72(%ecx,%esi), %ebx // B + 4k + 72 = B[3][k]
23 imull 12(%edx), %ebx // A+24i+12 * B[3][k] // A[i][3] * B[3][k]
24 addl %ebx, %eax // (A[i][3] * B[3][k]) + (A[i][2] * B[2][k]) + (A[i][0] * B[0][k]) + A[i][1]
25 movl 96(%ecx,%esi), %eax // B+4k + 96 = B[4][k]
26 imull 16(%edx), %eax // A+24i+16 * B[4][k] // A[i][4] * B[4][k]
27 addl %eax, %ebx // (A[i][4] * B[4][k]) + (A[i][3] * B[3][k]) + (A[i][2] * B[2][k]) + (A[i][0] * B[0][k]) + A[i][1]
28 movl 120(%ecx,%esi), %eax // B + 4k + 120 = B[5][k]
29 imull 20(%edx), %eax // A+24i+20 * B[5][k] // A[i][5] * B[5][k]
30 addl %ebx, %eax // (A[i][5] * B[5][k]) + (A[i][4] * B[4][k]) + (A[i][3] * B[3][k]) + (A[i][2] * B[2][k]) + (A[i][0] * B[0][k]) + A[i][1]
31 popl %ebx // restaura ebx
32 popl %esi // restaura esi
33 popl %edi // restaura edi
34 ret // retorna

```

Questão 4) c) Tente justificar a otimização feita pelo GCC, pensando na eficiência da execução. Analise e conclua as razões deste novo código ser mais eficiente do que o anterior. Compare os custos dos dois códigos (em função do custo definido na questão 1):

A otimização -O3 trata o código com mais linearidade, ou seja,  ao invés de buscar por .L2 diversas vezes no código, como na otimização -O2, ela é muito mais direta, possibilitando uma execução clara e objetiva. Além disso, o código com maior otimização, as operações fazem mais uso de registradores e, assim, são mais rápidas que acessar a memória diversas vezes como na otimização -O2.

