Lista 3 - Comp Prog
Alunos: Luiz Rodrigo Lacé Rodrigues (DRE:11804983)
Livia Barbosa Fonseca (DRE:118039721)

Questão 1) Faça a engenharia reversa baseando-se no código de montagem e preencha as reticências colocadas na parte apagada do código C. Você não pode acrescentar linha ou deixar de preencher qualquer linha. Os caracteres não apagados devem estar presentes na sua solução.

```
1 endbr32
Instrução que protege o processador de realizar mudanças de endereços
ao chamar funções. Assim, evitando a descontinuidade do fluxo correto
do programa.

2 pushl %edi
Aumenta a pilha em 4 bytes e salva em %esp o valor de %edi

3 pushl %esi
Aumenta a pilha em 4 bytes e Salva em %esp o valor de %esi

4 pushl %ebx
Aumenta a pilha em 4 bytes e salva em %esp o valor de %ebx

Os três registradores acima devem ser salvos pela rotina que está sendo
chamada devido a convenção caller-callee

5 movl 16(%esp), %esi
Neste momento temos %esp = %ebx, %esp+4 = %esi, %esp+8 = %edi, %esp+12
= RIP rot. Logo %esp+16 = primeiro argumento (lista[]), será copiado
para %esi

6 movl 20(%esp), %edi
Neste momento temos %esp = %ebx, %esp+4 = %esi, %esp+8 = %edi, %esp+12
= RIP rot, %esp+16 = primeiro argumento. Logo %esp+20 = segundo
argumento da rot,(x), que será copiado para o %edi.

7 movzbl (%esi), %eax
```

Salva no registrador %eax o valor do endereço guardado em %esi, que será o primeiro char da lista[] e estende com zeros à esquerda para 32 bits. Resumidamente, pegamos lista[0].

8 testb %al, %al
Faz um "and" do byte menos significativo em %eax com ele mesmo. Estamos fazendo o teste com lista[0] (teste inicial do for).

9 je .L5
Se o ZF (zero flag) estiver setado (se o and anterior foi igual a zero) desvia para o endereço .L5 para finalizar.

10 leal 1(%esi), %edx
Como %esi guarda um vetor, temos o incremento de 1 nesse endereço. O resultado é guardado em %edx. Esse registrador terá uma referência ao endereço da lista[].

11 movl $0, %ecx
Salva o valor 0 em %ecx. Inicializando a variável i = 0.

12 jmp .L4
Faz um salto incondicional ("jump") para o endereço .L4

.L8:
13 movb %al, (%esi,%ecx)
Copia o valor de %al para o endereço %esi+i (primeira posição do vetor lista [%esi+i] = %al. Teremos lista[j] = lista[i].

14 leal 1(%ecx), %ecx
Incrementa o valor de %ecx em um e guarda em %ecx. (Pós incremento de j na mesma linha da operação anterior).

.L3:
15 addl $1, %edx
Soma 1 (tamanho de um char) no valor de %edx passando para a próxima posição do vetor. Estamos incrementando o índice, ou seja, i++.

16 movzbl -1(%edx), %eax
Copia o valor, em byte, que está que em %edx decrementado de 1, estende com zeros à esquerda para 32 bits e passa para %eax.

17 testb %al, %al
Faz um "and" entre o %al e ele mesmo.

```
18 je .L2
```
Se o ZF (zero flag) estiver setado (se o "and" anterior foi igual a zero) pula para o endereço .L2. Se for igual a 0 (NULL), desvia do loop for.

```
.L4:
19 movsbl %al, %ebx
```
Fazemos uma cópia do valor de %al (char da iteração atual do nosso vetor) e movemos para o registrador %ebx, estendendo o sinal à esquerda para 32 bits.

Nesse momento, estamos nos preparando para realizar o if entre x e o valor da posição atual do vetor lista.

```
20 cmpl %edi, %ebx
```
Faz o valor em %ebx (char da iteração atual convertido para 32 bits) menos o valor %edi (argumento x da função) e seta as flags de condição Equivale à linha (if lista[i] =! x).

```
21 jne .L8
```
Se o ZF (zero flag) não estiver setado(resultado anterior diferente de zero) pula para o endereço .L8. Se lista[i] =! vamos para .L8.

```
22 jmp .L3
```
Faz um salto incondicional ("jump") para o endereço .L3. Isso é, caso a comparação anterior seja igual a zero. Se (if lista[i] =! x) não for atendido vamos para .L3.

```
.L5:
23 movl $0, %ecx
```
Passa o valor imediato 0 para o registrador %ecx. (Variável i do código em c vai receber i = 0 , pois tentamos entrar no ciclo for e realizamos a inicialização da variável).

```
.L2:
24 movb $0, (%esi,%ecx)
```
Passa o valor 0 para o registrador %esi + ecx. Armazena o caractere final de linha no vetor, ou seja, lista[i] = '\0'.

```
25 popl %ebx
```
Decrementa a pilha, guardando em %ebx o valor que estava no topo da pilha, que era o seu valor no começo da rotina. (Restaura registrador)

```
26 popl %esi
Decrementa a pilha, guardando em %esi o valor que estava no topo da
pilha, que era o seu valor no começo da rotina. (Restaura registrador)

27 popl %edi
Decrementa a pilha, guardando em %edi o valor que estava no topo da
pilha, que era o seu valor no começo da rotina.(Restaura registrador)

26 ret
Como o topo da pilha aponta para RIP rot, devolve o controle a quem
chamou rot. (Retorna)
```

a) (10) Comente cada linha do código de montagem, procurando não dizer o que a instrução faz mas associar a instrução ao código C que foi apagado. Ao resolver este item, deixe para comentar após ter reconstruído o programa C pois fará mais sentido mostrar o significado das instruções em termos das variáveis do código C, o que é a engenharia reversa pretendida.

b) (10) Complete o código C e descreva o que a rotina faz.

```c
void rot (char lista[], int x){
   int i, j;
   for(i = j = 0; lista[i]!='\0';i++){//percorrendo o vetor lista até
encontrar o seu final
       if (lista[i] != x){//Analisa se o caractere é diferente de x
           lista[j++] = lista[i];//Faz uma cópia para lista eliminando
o caractere x da lista
       }
   }
   lista[j]='\0';//Coloca o caractere final de linha no fim do vetor
}
```

Questão 2)

```
rot:
1 endbr32
Instrução que protege o processador de realizar mudanças de endereços
ao chamar funções. Assim, evitando a descontinuidade do fluxo correto
do programa.

2 movl 8(%esp), %eax
Pega apenas o primeiro parâmetro da função (n) e passa para o
registrador %eax.

3 addl $3, %eax
Soma o valor 3 ao valor que estava em %eax (n+3). Indicando que n=-3 é
o valor mais baixo dos casos do switch.

4 cmpl $8, %eax
Compara o valor de %eax (n+3) com 8, isso é, (n+3=<8) indicando que o
maior valor será 5.

5 ja .L2
Desvia para default (.L2) se for maior que 5.

6 notrack jmp *.L4(,%eax,4)
Calcula para que posição de .L4 o switch irá executar. .L4+ %eax * 4

.L4:
7 .long .L9
Índice 0, caso = -3
8 .long .L7
Índice 1, caso = -2
9 .long .L2
Índice 2, caso = -1 (default)
10 .long .L2
Índice 3, caso = 0 (default)
11 .long .L6
Índice 4, caso = 1
12 .long .L2
Índice 5, caso = 2 (default)
13 .long .L2
Índice 6, caso = 3 (default)
14 .long .L5
Índice 7, caso = 4
15 .long .L3
```

```
Índice 8, caso = 5
16 .text
Indica sessão de código executável

/// Caso = -1, caso = 0, caso = 2, caso = 3///
.L2:
17 movl $-1, %eax
Passa o valor -1 para o registrador %eax. (n =-1)

.L8:
18 imull %eax, %eax
Realiza uma multiplicação entre o registrador %eax e ele mesmo.(n^2)
Faz (eax*eax)
19 ret
Retorna (eax*eax) (return n^2)

///caso = -2
.L7:
20 movl $1, 4(%esp)
Passa o valor 1 para o endereço %esp+4 (x)

///caso = 1
.L6:
21 movl 4(%esp), %eax
Passa para %eax o valor na posição do endereço %esp+4 (x)
22 addl $1, %eax
Adiciona 1 ao valor de x
23 jmp .L8
Pula incondicionalmente para .L8

///caso = 4
.L5:
24 addl $2, 4(%esp)
Adiciona o valor 2 em %esp+4 (x) = x+2

///caso = 5
.L3:
25 movl 4(%esp), %eax
Passa para %eax o valor de %esp + 4, que é x+2.
26 addl %eax, %eax
Faz eax + eax
27 jmp .L8
Pula incondicionalmente para .L8
```

```
///caso = -3
.L9:
28 movl $0, %eax
Salva o valor 0 no registrador %eax (x=0)
29 jmp .L8
Pula incondicionalmente para .L8
```

a) (4) As linhas 3 e 4 são fundamentais para descobrir os valores extremos envolvidos nos cases do switch. Indique o que cada linha está fazendo, uma por uma, e justifique o que ela está testando e mostrando como são reconhecidos os valores extremos do switch.

```
3 addl $3, %eax
Como o nosso "n" está em %eax, teremos o incremento de 3 em "n".
Sabemos que esse valor é importante, pois ele indicará o valor mais
baixo dos nossos casos do switch. Logo, teremos que o valor mais baixo
será n=-3.

4 cmpl $8, %eax
Aqui estamos comparando o valor de %eax (n+3) com 8, isso é, n+3-8.
Logo, teremos que o maior valor dos casos switch será 5.E seta os flags
de condições de acordo com o resultado comparativo.
```

b) (4) Explique o que a linha 5 faz e mostre como ela completa os valores possíveis no comando switch.

```
5 ja .L2
Nessa linha temos um desvio para os casos default (.L2) se a comparação
feita na linha 4 for maior que 5 (n>5) e menor do que -3 (n<-3).

Porém quando n = -1,0,2,3 também entraremos no caso default, pois não
teremos esses casos no comando switch, visto que quando temos esses
valores eles desviam para .L2, que é o default.
```

c) (4) Explique o notrack na linha 6, o que a instrução faz e explique a razão do conteúdo de memória nas linhas 7 a 16. Associe cada uma das linhas 7 a 16 aos valores dos cases respectivos.

O prefixo notrack desabilita a proteção endbr e é útil para realizar desvios indiretos de forma segura (dependendo do valor do registrador), visto que o destino não pode ter suas informações alteradas.
A necessidade do notrack vem do fato que o processador gera bloqueios para desvios indiretos não verificados quando compilamos o programa como uma forma de proteção.

Na linha 6, *.L4(,%edx,4) é onde haverá um cálculo para o desvio o qual será usado como parâmetro para o jmp. O desvio consiste em calcular qual index da tabela de desvio será usado. A diretiva .L4 é usada pois é nesse endereço onde está definida a tabela de desvios.

Como visto nas linhas 3 e 4 do código de montagem, teremos os casos do switch variando entre 5, -3 (maior e menor valor entre os casos). Logo, é necessário armazenar todos os "valores" de 5 a -3. As opções que não se encontram no switch case são consideradas default.

```
.L4:
7 .long .L9
Índice 0, caso = -3
8 .long .L7
Índice 1, caso = -2
9 .long .L2
Índice 2, caso = -1 (default)
10 .long .L2
Índice 3, caso = 0 (default)
11 .long .L6
Índice 4, caso = 1
12 .long .L2
Índice 5, caso = 2 (default)
13 .long .L2
Índice 6, caso = 3 (default)
14 .long .L5
Índice 7, caso = 4
15 .long .L3
Índice 8, caso = 5
16 .text
Indica sessão de código executável
```

d) (8) Escreva abaixo o código C completo da rotina.

```
int rot(int x, int n){
   switch(n){
       case -3:
           x = 0;
           break;
       case -2:
           x = 1;
            //não possui break e executará case 1
       case 1:
           x += 1; //soma um em x
           break;
        case 4:
```

```
            x+=2; //soma 2 ao valor de x
             //não possui break e executará case 5.
        case 5:
            x+=x; // Soma x+x
            break;
        default:
            x = -1;
    }
    return (x*x);
}
```

Questão 3)Seja o código C abaixo, onde a dimensão das matrizes foi apagada.

```
#define N ...
typedef int matrix[N][N];
int prod (matrix A, matrix B, int i , int k){
    int j;
    int result = 0;
    for (j=0;j<N;j++)
        result += A[i][j] * B[j][k];
        return result;
}
```

a) (8) O código compilado acima com -m32 -fno-PIC -S -O2 gera o código de montagem abaixo. Comente cada linha, sem exceção, justifique sua existência e associe ao código C. Identifique o uso de ponteiros para acesso facilitado aos elementos das matrizes.

```
prod:
1 endbr32
Instrução que protege o processador de realizar mudanças de endereços
ao chamar funções. Assim, evitando a descontinuidade do fluxo correto
do programa.

2 pushl %esi
Incrementamos a pilha em 4 bytes e armazenamos no topo o registrador
%esi.

3 pushl %ebx
Incrementamos a pilha em 4 bytes e armazenamos no topo o registrador
%ebx.
4 movl 20(%esp), %eax
```

Depois dos pushes anteriores vamos ter que %esp = ebx, %esp+4 = %esi, %esp + 8 = RIP prod, %esp+12 = endereço da matrix A[0][0], %esp+16 = endereço da matrix b[0][0] e finalmente %esp+20 = i, que é o nosso terceiro parâmetro da função. Assim estamos passando o valor de i para o registrador %eax.

5 xorl %ebx, %ebx
Vamos fazer um "xor" de %ebx com ele mesmo, ou seja, vamos zerar o valor, visto que a operação xor com números iguais resulta em zero. Nesse comando é o que está acontecendo na linha (int result = 0;)

6 movl 12(%esp), %edx
Aqui estamos copiando o que está em %esp+12 (ponteiro de A) para o registrador %edx.

7 movl 24(%esp), %esi
Estamos copiando o quarto argumento da função(int k) para o resgistrador %esi.

8 leal (%eax,%eax,4), %eax
Estamos passando para o registrador %eax 4%eax+%eax, como %eax guardava o valor do nosso terceiro argumento (int i), vamos ter que que agora %eax = 5i

9 leal (%edx,%eax,4), %eax
Estamos guardando o valor de (5i*4) + A em %eax, logo %eax = A + 20i (= &A[i][0]). Aqui teremos o início do "for" no código c.

10 leal 0(,%esi,4), %ecx
Estamos guardando o valor de 4*k no registrador %ecx.

11 addl 16(%esp), %ecx
Como %esp+16 = matrix B[0][0], logo estamos somando isso ao registrador %ecx e guardando o resultado nele, agora temos %ecx = B + 4*k (=B[0][k])

12 leal 20(%eax), %esi
Estamos armazenando o que temos em %eax somado ao valor imediato 20 no registrador %esi, logo vamos ter %esi = (A[0][0] + 20i) + 20, estamos fazendo i++ no codigo c

.L2:

13 movl (%eax), %edx
Vamos copiar o valor que está em A + 20i e colocar no registrador %edx

14 imull (%ecx), %edx
Aqui vamos fazer a multiplicação completa de 64bits com sinal do valor que está no endereço B + 4k e o valor que está no endereço de %edx que é A+20i e guarda o resultado no registador %edx.
No código C temos que essa multiplicação está na linha
(result += A[i][j] * B[j][k];)

15 addl $4, %eax
Aqui iremos somar 4 ao valor que está no registrador %eax (A + 20i). Resultando em (A + 20i + 4) (Incremento de 4 bytes, que é o tamanho de 1 int).Essa operação vai ser equivalente a "passar" para a próxima coluna j do vetor A. Equivale ao incremento do j no loop (for (j=0;j<N;j++))

16 addl $20, %ecx
Aqui iremos somar 20 ao valor que está no registrador %ecx (B + 4*k). Logo teremos (B + 4*k + 20). Como cada int tem tamanho de 4 bytes, estamos avançando 5 ints em B, ou seja, estamos incrementando o valor da linha de B. Aqui ainda estamos referenciando ao for comentado anteriormente, mas com a abstração que aqui estamos passando para a próxima linha.

17 addl %edx, %ebx
Iremos somar o valor de %edx (B + 4k)*(A+20i) com o valor no registrador %ebx. Isso significa acrescentar o resultado da multiplicação no %ebx. Esse comando faz referência a soma da multiplicação para a variável result na linha
(result += A[i][j] * B[j][k];)

18 cmpl %esi, %eax
Aqui subtraímos %eax (A + 20i + 4) de %esi (A + 20i + 20), e setamos as flags de condição. Isso vai servir para o saber se vamos fazer o desvio para voltar para o início do loop. Como o for é referente a dimensão das matrizes, só vamos sair dele quanto quando %eax = (A + 20i + 20)

19 jne .L2
Desvia para o endereço .L2 se a comparação da linha anterior for diferente de zero. Ou seja, se %esi for diferente de %eax. Aqui estamos repetindo o processo anterior até o final das matrizes.
20 movl %ebx, %eax

```
Aqui estamos armazenando o valor do registrador %ebx (result), depois
de todas as iterações, no %eax. Esse comando faz a preparação do
retorno da rotina (return result).

21 popl %ebx
Aqui realizamos o decremento da pilha, guardando em %ebx o valor que
estava no topo da pilha.

22 popl %esi
Realizamos o decremento da pilha, guardando em %esi o valor que estava
no topo da pilha.

23 ret
Nesse momento realizamos o retorno da função, devolvendo o controle a
quem chamou a rotina. Esse comando equivale a linha return result.
```

b) (2) Qual a dimensão das matrizes?

Na linha 9 temos leal (%edx,%eax,4), %eax, onde guardamos o valor de (5i*4) em %eax, logo %eax = A+20i. Como sabemos, pelo codigo C que A[i][j], ou seja vamos variar a linha de A conforme variarmos o valor de i, e para isso é preciso passar por todas as j colunas. Como cada int “custa” 4bytes, quando dividirmos 20/4 achamos que cada linha dessa matriz é composta por 5 colunas. E como a matriz tem o mesmo número N para linhas e colunas, a dimensão dessa matriz será matriz[5][5]

c) (10) Agora foi feita a compilação com otimização -O3. Comente cada linha, sem exceção, mostrando o que ela está de fato calculando em relação ao produto matricial. Justifique a existência de cada linha, associando ao código C.

```
prod:
1 endbr32
Instrução que protege o processador de realizar mudanças de endereços
ao chamar funções. Assim, evitando a descontinuidade do fluxo correto
do programa.
2 pushl %edi
Incrementamos a pilha em 4 bytes e armazenamos no topo o registrador
%edi


3 pushl %esi
```

Incrementamos a pilha em 4 bytes e armazenamos no topo o registrador %esi

4 pushl %ebx

Incrementamos a pilha em 4 bytes e armazenamos no topo o registrador %ebx

5 movl 24(%esp), %eax

Nesse momento vamos ter %esp = %ebx, %esp +4 = %esi, %esp+8 = %edi, %esp+12 = RIP prod, %esp +16 = primeiro parâmetro da função(endereço de A[0][0]), %esp+20 = segundo parametro da função (endereço de B[0][0]) e finalmente %esp+24 = valor do endereço do terceiro parametro (int i). Estamos passando o valor de i para o registrador %eax

6 movl 28(%esp), %ecx

Seguindo raciocínio anterior, vamos copiar o valor do quarto parâmetro da função(int k) para o registrador %ecx.

7 movl 20(%esp), %ebx

Copiamos o valor de %esp+20, que é um ponteiro para a matriz B no registrador %ecx.

8 leal (%eax,%eax,4), %edx

Nesse momento temos que %eax = i, então estamos passando para o registrador %edx 4i+i, ou seja %edx = 5i.

9 movl 16(%esp), %eax

Vamos passar valor de %esp+16, que é o ponteiro para a matriz A, para o registrador %eax

10 leal 0(,%ecx,4), %esi

Vamos agora passar o 4*%ecx para o registrador %esi, agora %esi = 4k

11 movl (%ebx,%ecx,4), %ecx

Vamos passar agora para o registrador %ecx %ebx+4%ecx,ou seja vamos ter que %ecx = B+4k, referente a B[0][k]

12 leal (%eax,%edx,4), %edx

Vamos passar para o registrador %edx 4*%edx + eax, ou seja, %edx = 4*(5i) + A = A + 20i. Que é referente ao A[i][0] no código C.

13 movl 20(%ebx,%esi), %eax

Vamos passar para o registrador %eax 20+%ebx+esi, ou seja, %eax = B + 4*k + 20, referente a B[1][k] no código C.

14 imull (%edx), %ecx

Vamos multiplicar %edx com %ecx (completa de 64 bits) e guardar o resultado no registrador %ecx, ou seja, %ecx = (A + 20i) * (B + 4k). Nesse momento estamos fazendo a seguinte multiplicação no código c "A[i][0] * B[0][k]". No código c temos j=0

15 imull 4(%edx), %eax

Vamos realizar uma multiplicação completa de 64 bits entre o valor do registrador %edx + 4 e %eax e guardar o resultado em %eax. Teremos %edx = A + 20i + 4 multiplicando com %eax = B + 4*k + 20, ou seja, estamos

fazendo uma multiplicação do tipo "A[i][1] * B[1][k]" no código C, referente a iteração com j=1.

16 leal (%eax,%ecx), %eax

Vamos realizar uma soma entre os registradores %eax e %ecx. Logo estamos fazendo "A[i][1] * B[1][k] + A[i][0] * B[0][k]" Ou seja, estamos somando as duas multiplicações feitas anteriormente, que é referente a passar o somatório para a variável result.

17 movl 40(%ebx,%esi), %ecx

Vamos armazenar o valor de %ebx + 40 + %esi em %ecx. Estamos fazendo a seguinte operação B + 4k + 40 = B[2][k] e passando para o registrador %ecx. Agora %ecx = B[2][k].

18 imull 8(%edx), %ecx

Vamos realizar uma multiplicação entre %edx + 8 e %ecx (completa de 64 bits). Teremos (A + 20i + 8) * (B + 4k + 40) = A[i][2] * B[2][k], que no código C é referente a iteração com j = 2.

19 addl %eax, %ecx

Vamos realizar a soma entre %eax e %ecx, ou seja, %ecx = A[i][1] * B[1][k] + A[i][0] * B[0][k] + A[i][2] * B[2][k]., referente ao somatório da variável result.

20 movl 60(%ebx,%esi), %eax

Vamos armazenar o valor de %ebx + %esi + 60, isso é B+4k+60 = B[3][k] no registrador %eax.

21 imull 12(%edx), %eax

Vamos realizar uma multiplicação completa de 64 bits entre %edx + 12 e %eax, ou seja, (A + 20i + 12) * B[3][k] = A[i][3] * B[3][k] e guardando o resultado em %eax, referente a iteração com j = 3 no código C.

22 addl %eax, %ecx

Vamos realizar uma soma entre %eax e %ecx, ou seja, agora %ecx = A[i][3] * B[3][k] + A[i][1] * B[1][k] + A[i][0] * B[0][k] + A[i][2] * B[2][k], referente ao somatório da variável result.

23 movl 80(%ebx,%esi), %eax

Vamos armazenar o valor de %ebx + %esi + 80 em %eax, ou seja, B + 4k + 80 (B[4][k])

24 imull 16(%edx), %eax

Vamos realizar uma multiplicação completa de 64 bits entre %edx + 16 e %eax, ou seja, A+20i + 16 * B[4][k], A[i][4] * B[4][k] e guardar o resultado em %eax, referente a iteração com j=4 no código C.

25 popl %ebx

Vamos decrementar a pilha e guardar em %ebx o valor que está no topo da pilha.

26 popl %esi

Vamos decrementar a pilha e guardar em %esi o valor que está no topo da pilha.

```
27 popl %edi
Vamos decrementar a pilha e guardar em %edi o valor que está no topo da
pilha.
28 addl %ecx, %eax
Vamos somar o conteúdo do registrador %ecx ao %eax. Agora %eax =
A[i][3] * B[3][k] + A[i][1] * B[1][k] + A[i][0] * B[0][k] + A[i][2] *
B[2][k] + A[i][4] * B[4][k], referente ao somatório total de das
iterações j =0 até j = 4 com os resultados sendo passados para a
variável result.
29 ret
Devolve o controle a quem chamou a rotina. Aqui temos a linha (return
result) do código em c.
```

d) Vamos calcular os custos dos dois códigos de montagem, usando m para o custo de execução de uma instrução que acessa a memória e por n o custo de execução de uma instrução que não acessa a memória.Para o código da otimização -O2, indique os tipos de cada uma das linhas. Considerando que algumas linhas serão executadas mais de uma vez, calcule o custo total de execução do código.
Para o código da otimização -O3, indique os tipos de cada uma das linhas. Calcule o custo total de execução do código. Tente justificar a otimização feita pelo GCC, pensando na eficiência da execução. Analise e conclua a razão da utilização -O3 ser mais eficiente, considerando todos os argumentos possíveis.

Com otimização -O2:

```
prod:
1 endbr32 //Não acessa
2 pushl %esi // Acessa
3 pushl %ebx //Acessa
4 movl 20(%esp), %eax //Acessa
5 xorl %ebx, %ebx //Não acessa
6 movl 12(%esp), %edx //Acessa
7 movl 24(%esp), %esi //Acessa
8 leal (%eax,%eax,4), %eax //Não acessa
9 leal (%edx,%eax,4), %eax //Não acessa
10 leal 0(,%esi,4), %ecx //Não acessa
11 addl 16(%esp), %ecx //Acessa
12 leal 20(%eax), %esi //Não acessa
.L2:
```

```
13 movl (%eax), %edx //Acessa
14 imull (%ecx), %edx //Acessa
15 addl $4, %eax //Não acessa
16 addl $20, %ecx // Não acessa
17 addl %edx, %ebx // Não acessa
18 cmpl %esi, %eax //Não acessa
19 jne .L2 //Não acessa
20 movl %ebx, %eax //Não Acessa
21 popl %ebx //Acessa
22 popl %esi //Acessa
23 ret //Acessa
```

(9+ 10(loop))m +n(7+ 25(loop))= 19m +32n

Com Otimização -O3:

```
prod:
1 endbr32 //Não acessa
2 pushl %edi //Acessa
3 pushl %esi //Acessa
4 pushl %ebx //Acessa
5 movl 24(%esp), %eax //Acessa
6 movl 28(%esp), %ecx //Acessa
7 movl 20(%esp), %ebx //Acessa
8 leal (%eax,%eax,4), %edx//Não acessa
9 movl 16(%esp), %eax //Acessa
10 leal 0(,%ecx,4), %esi //Não acessa
11 movl (%ebx,%ecx,4), %ecx //Acessa
12 leal (%eax,%edx,4), %edx //Não acessa
13 movl 20(%ebx,%esi), %eax //Acessa
14 imull (%edx), %ecx //Acessa
15 imull 4(%edx), %eax //Acessa
16 leal (%eax,%ecx), %eax //Não acessa
17 movl 40(%ebx,%esi), %ecx //Acessa
18 imull 8(%edx), %ecx //Acessa
19 addl %eax, %ecx //Não acessa
20 movl 60(%ebx,%esi), %eax //Acessa
21 imull 12(%edx), %eax //Acessa
22 addl %eax, %ecx //Não acessa
23 movl 80(%ebx,%esi), %eax  //Acessa
24 imull 16(%edx), %eax //Acessa
25 popl %ebx  //Acessa
26 popl %esi  //Acessa
27 popl %edi  //Acessa
28 addl %ecx, %eax //Não acessa
```

```
29 ret //Acessa
```

Custos dos códigos de montagem:
Otimização -O2: 19m +32n, 51 instruções sendo executadas
Otimização -O3: 21m + 8n, 29 instruções sendo executadas

Na compilação com otimização -O3, o GCC eliminou a multiplicação que ocorria no loop e passou a multiplicar de forma eficiente para a matriz A com 5 colunas e a matriz B com 5 linhas. Dessa forma, não há mais comparações, nem incrementos separados das instruções que acessam a memória.

Ocorreu uma redução de 22 instruções se comparado com a otimização -O2

Questão 4)Seja dado o seguinte programa C que chama uma rotina recursiva,

```
void rotina (int n){
   if (n < 0) { //Estamos testando se n é negativo
      putchar('-'); //Se n for negativo, iremos imprimir um sinal "-"
      n = -n; //E converter esse número para um número positivo
   }
   if (n/10) rotina (n/10); //Chamamos recursivamente dividindo por 10
   putchar (n%10 + '0'); //Imprime o dígito decimal correspondente ao
                                    resto da divisão por 10
   }
int main () {rotina (...);}
```

a) (5) Explique e justifique a saída desta rotina recursiva, assumindo |n| = bk10^k + bk-1 10^k-1 + ... + b110+ b0. Mostre que instância da rotina imprime o quê.

Esta rotina imprime o valor decimal de uma representação inteira. Por exemplo, ao ser passado o inteiro n = -100 para a rotina:

Primeiro vamos verificar se o número é negativo na linha if(n<0), se for, imprimimos o símbolo traço (‘-’) e fazemos n valer -n, agora n = 100.

Passando para o segundo if (if(n/10)), vamos verificar se ainda temos um resultado inteiro para a divisão de n por 10 (que vai funcionar como um true/false), caso seja possível dividir por 10, isso é (n/10 > 0), vamos chamar a rotina novamente de forma recursiva passando agora (n/10).

Agora voltamos pro início da rotina com n = 10, como não é um número negativo não vamos entrar no primeiro if, mas ainda podemos dividir 10 por 10, visto que 10/10 = 1 e assim entramos no segundo if chamando novamente a recursividade da rotina, agora com n = 1.

Como n = 1, tentamos entrar no primeiro if, como é positivo, não entra, mas agora também não podemos dividir por 10, visto que 1/10 < 1, que o if reconhece como um false. partimos agora para o putchar onde vamos imprimir a soma do resto da divisão de n por 10 mais o 0 como um caracter. Como na tabela ASCII ‘0’ vale 48, então quando n%10 for igual a 1, vamos somá-lo a 48, resultando em 49, que é 1 na tabela ASCII, quando n%10 = 0, continuamos com 48 e imprimimos 0.

Aí vamos imprimir primeiramente o resto de 1 por 10, que é 1, e imprimimos 1 assim a rotina de n = 1 acaba, voltamos para a rotina de n= 10 e imprimimos a soma do resto de 10 por 10 que é 0 com ‘0’, ou seja, imprimimos 0 . E finalmente voltamos para a rotina de n=100 e imprimimos 100%10 + ‘0’, que é 0.
O resultado final que vamos imprimir será -100

b) (10) A compilação sem otimização gerou o código de montagem que se segue. Comente cada linha deste código e procure então associá-lo ao código C dado. Não descreva o que a instrução faz, pois isso é sabido. Tente entender o que ela representa para o código C e faça a engenharia reversa. Um conjunto de instruções de montagem podem ser necessárias para realizar uma operação em C.

```
rotina:
1 endbr32
Instrução que protege o processador de realizar mudanças de endereços
ao chamar funções. Assim, evitando a descontinuidade do fluxo correto
do programa.
2 pushl %ebp
Incrementamos a pilha em 4 bytes e armazenamos no topo o registrador
%ebp.
3 movl %esp, %ebp
Estamos fazendo %esp = %ebp, ou seja, estamos criando uma nova base.
4 subl $8, %esp
Estamos abrindo 2 espaços na pilha.
5 cmpl $0, 8(%ebp)
Estamos realizando uma comparação entre 0 e o conteúdo do registrador
%ebp + 8 (int n).
6 jns .L2
Se o resultado da operação anterior não for negativo, teremos um desvio
para .L2. Ou seja, se o int n não for negativo teremos o desvio da
função, referente ao "if (n < 0)" no código C.
7 subl $12, %esp
Abre 3 espaços na pilha
8 pushl $45
Coloca na pilha o símbolo negativo '-', (45 segundo a tabela ASCII)
9 call putchar
Imprime o símbolo '-'
10 addl $16, %esp
Devolve 4 espaço para a pilha
11 negl 8(%ebp)
Obtemos -n, onde o menor negativo dará 0x80000000,ou seja overflow.

.L2:
12 movl 8(%ebp), %eax
Salva n em %eax
13 addl $9, %eax
Soma 9 no valor de %eax e salva, agora %eax = n + 9
14 cmpl $18, %eax
Faz a comparação do valor 18 com %eax, n-9
```

```
15 jbe .L3
Desvia para .L3 caso n < 10, referente ao "if (n/10)" no código C.

16 movl 8(%ebp), %ecx
Passa o valor de n para o registrador %ecx
17 movl $1717986919, %edx
Estamos passando o a representação de 0x66666667 = 0,4 * 2 ** (32) para
o registrador %edx
18 movl %ecx, %eax
Passa o valor de n para %eax
19 imull %edx
Vamos multiplicar o que temos em %edx com %eax e guardar o resultado em
%eax, temos agora que %eax = n * 0x66666667 em 64 bits
20 sarl $2, %edx
Fazemos um shift aritmético para a direita no valor de %edx, temos
agora que n * 0x66666667 * 2 ** (-32)* 2 ** (-2) = n*0,4**(-2) = n*0,1.
Agora temos que %edx = n*0,1
21 movl %ecx, %eax
Passa n para %eax
22 sarl $31, %eax
Estende o MSB para %eax
23 subl %eax, %edx
Vamos somar 1 à magnitude em %edx, caso n seja o menor negativo
"possível"
24 movl %edx, %eax
Vamos passar o conteúdo de %edx para %eax, %eax = n*0,1
25 subl $12, %esp
Abrimos 3 espaços na pilha
26 pushl %eax
Vamos passar para o topo da pilha n/10 como parâmetro para a função que
será chamada.
27 call rotina
Chama a função "rotina"
28 addl $16, %esp
Iremos devolver 4 espaços na pilha

.L3:
29 movl 8(%ebp), %ecx
Vamos passar n para o registrador %ecx
30 movl $1717986919, %edx
Passa a representação de 0,4 *2**(32) para o registrador %edx
31 movl %ecx, %eax
Vamos passar n para o registrador %eax
```

```
32 imull %edx
Iremos multiplicar n por 0,4 *2**(32), ficando com %eax = n * 0,4
*2**(32), em 64 bits.
33 sarl $2, %edx
Iremos realizar um shift aritmético de 2 posições para a direita,
obtendo %edx = n*0,1
34 movl %ecx, %eax
Passa n para %eax
35 sarl $31, %eax
Iremos estender o MSB para o todo registrador eax
36 subl %eax, %edx
Estamos somando 1 à magnitude em %edx caso n seja o menor negativo
"possível"
37 movl %edx, %eax
Iremos passar o valor de %edx para %eax, %eax = n*0,1
38 sall $2, %eax
Iremos realizar um deslocamento aritmético de 2 bits à esquerda no
valor de %eax, ou seja estamos fazendo n*0,1*2**2 = n*0,4
39 addl %edx, %eax
Vamos adicionar %edx em %eax, agora %eax = n*0,1 + n*0,4 = n*0,5 =
5*n*0,1
40 addl %eax, %eax
Vamos somar %eax com %eax, %eax agora 2*(5*n*0,1) = 10 * n * 0,1
41 subl %eax, %ecx
Vamos subtrair o conteúdo de %eax de %ecx, %ecx agora n - 10 * n * 0,1
Agora temos que %ecx = n - 10*n/10, ou seja %ecx recebe n%10
42 movl %ecx, %edx
Vamos copiar o conteúdo de %ecx para %edx, %edx = n%10
43 leal 48(%edx), %eax
Vamos passar o conteúdo de %edx + 48 para %eax, sendo que 48 na tabela
ASCII representa 0. A parte “putchar ( n%10+ '0')” do código em c.
44 subl $12, %esp
Vamos abrir 3 espaços na pilha
45 pushl %eax
Vamos dar um push com o conteúdo de %eax para ser o parâmetro da
chamada da função
46 call putchar
Chama a função “putchar”, imprime n%10
47 addl $16, %esp
Devolve 4 espaços para a pilha
48 nop
Sem operação
49 leave
```

```
Estamos nos preparando para o retorno, iremos restaurar a base da pilha
e apontar o topo para a RIP.
50 ret
Iremos retornar a função
```

c) (10) Compilando e executando main () {rotina (0x80000000) é impresso --214748364(, que está claramente errado. Analise e encontre o que está errado no código de montagem e que precisa ser alterado. Apresente a justificativa de sua análise. Indique exatamente as modificações que devem ser feitas para que o programa rode corretamente e apresente a saída correta esperada que é -2147483648. Não delete ou acrescente linhas ao código. Altere apenas as instruções que foram necessárias, o mínimo possível.

Para a impressão de números negativos, a rotina converte n para - n, entretanto, quando tivermos o menor negativo inteiro possível, -2 ** 32 em hexadecimal, 0x80000000, quando executarmos o comando negl %ebx, vamos obter 0x80000000 de novo devido ao overflow

Podemos fazer outra verificação para que n = 0x80000000 não entre no primeiro if, assim não vamos ter dois traços.

Outro if/else para verificar se n%10 + ‘0’ == 40 (valor de ‘(‘ na tabela ASCII, caso seja somamos 16, obtendo 56 (valor de 8 na tabela ASCII), caso contrário simplesmente imprimimos n%10 + ‘0’