Lista 4 - Comp Prog
Alunos: Luiz Rodrigo Lacé Rodrigues (DRE:11804983)
Livia Barbosa Fonseca (DRE:118039721)

Questão 1 (35 pontos) No código C abaixo, a ópera recebe uma estrutura como argumento e retorna outra estrutura. A função calcula chama ópera e retorna um inteiro.

```
typedef struct { short int a1; short int a2;} st1;
typedef struct { int soma; int dif;} st2;
st2 opera (st1 s1){
      st2 guarda;
      guarda.soma = s1.a1 + s1.a2;
      guarda.dif = s1.a1 - s1.a2;
      return guarda;}
int calcula (short int x, short int y){
     st1 s1;
     st2 s2;
     s1.a1 = x;
     s1.a2 = y;
     s2 = opera(s1);
     return s2.soma * s2.dif;}
```

a) (10) Comente cada linha do código acima, associando-a ao código C. Identifique os elementos das estruturas questão sendo acessados e/ou manipulados. Justifique cada instrução.

```
opera:
1 endbr32
Instrução de proteção dos processadores. Com esse comando, esse
endereço é validado e pode ser usado como  um desvio. Assim pode ser
usado para que fluxos não controlados sejam detectados.
2 pushl %ebp
Incrementamos a pilha em 4 bytes e armazenamos no topo da pilha o frame
pointer atual.
3 movl %esp, %ebp
Aqui estamos criando o registro de ativação, isso mostra que o código
foi compilado sem otimização.
4 subl $16, %esp
Vamos abrir 4 espaços de 4 bytes na pilha para que a variável local
(guarda) possa ser alocada.
5 movzwl 12(%ebp), %eax
Copiamos o valor que está no endereço %ebp + 12, Ou seja, estamos
copiando o valor de s1.a1 para o registrador %eax.
6 movswl %ax, %edx
Estamos salvando s1.a1 no registrador %edx, ou seja agora %edx = s1.a1
```

7 movzwl 14(%ebp), %eax
Copiamos o valor que está no endereço %ebp + 14, Ou seja, estamos copiando o valor de s1.a2 para o registrador %eax.
8 cwtl
Aqui convertemos de word para doubleword no registrador %eax. Sabendo que as estruturas a1 e a2 são short int e sua soma será guardada em uma variável do tipo int (guarda.soma) teremos que realizar essa transformação.
9 addl %edx, %eax
Estamos fazendo a soma de s1.a1 + s2.a2, e guardando o resultado em %eax
10 movl %eax, -8(%ebp)
Vamos salvar o resultado anterior na pilha, equivalente a guarda.soma = s1.a1 + s1.a2 no código C.
11 movzwl 12(%ebp), %eax
Vamos passar s1.a1 para o registrador %eax.
12 movswl %ax, %edx
Passamos s1.a1 para o registrador %edx
13 movzwl 14(%ebp), %eax
Passamos s1.a2 para o registrador %eax
14 cwtl
Vamos converter de word para doubleword no registrador %eax. Sabendo que as estruturas a1 e a2 são short int e sua soma será guardada em uma variável do tipo int (guarda.dif) teremos que realizar essa transformação.
15 subl %eax, %edx
Vamos fazer a subtração de %edx com %eax, ou seja s1.a1 - s1.a2, no código C, e salvamos o resultado em %edx.
16 movl %edx, %eax
Vamos copiar o valor de %edx (s1.a1 - s1.a2) para %eax.
17 movl %eax, -4(%ebp)
Vamos salvar na pilha o resultado da subtração anterior, equivalente a guarda.dif = s1.a1 - s1.a2, no código C.
18 movl 8(%ebp), %ecx
Salva o endereço para a estrutura s2 da função calcula no registrador %ecx.
19 movl -8(%ebp), %eax
Salvamos o valor da soma (guarda.soma) em %eax.
20 movl -4(%ebp), %edx
Salva o valor da diferença (guarda.dif) em %edx.
21 movl %eax, (%ecx)
A posição de memória de endereço %ecx vai receber o valor de %eax, que é igual a s2.soma, da função opera, receber o valor de guarda.soma

```
22 movl %edx, 4(%ecx)
```
Aqui temos que %ecx + 4 vai receber o valor de %edx, ou seja, s2.dif (da função calcula)vai receber o valor de guarda.dif.
```
23 movl 8(%ebp), %eax
```
Vamos salvar em %eax a posição de memória da estrutura de s2, visto que esse registrador será retornado na função.
```
24 leave
```
Estamos preparando a pilha para retornar à função calcula. Nesse momento, %ebp volta a possuir o endereço da base da rotina calcula.
```
25 ret $4
```
Aqui estamos retornando %eax para a função calcula e limpando uma posição de 4 bytes do topo da pilha. Essa instrução vai ser a "return guarda;" no código c.

```
calcula:
26 endbr32
```
Instrução de proteção dos processadores. Com esse comando, esse endereço é validado e pode ser usado como um desvio. Assim pode ser usado para que fluxos não controlados sejam detectados.
```
27 pushl %ebp
```
Incrementa a pilha em 4 bytes e armazena no topo da pilha o frame pointer atual.
```
28 movl %esp, %ebp
```
Nesse momento, estamos criando o registro de ativação, ou seja, estamos fazendo a base possuir o endereço do topo.
```
29 subl $40, %esp
```
Abrimos 10 espaços de 4 bytes na pilha para que as variáveis s1 e s2 possam ser alocadas.
```
30 movl 8(%ebp), %edx
```
Guardamos o argumento x em %edx,estendido para long
```
31 movl 12(%ebp), %eax
```
Guardamos o argumento y em %eax, estendido para long.
```
32 movw %dx, -28(%ebp)
```
Guardamos o valor de x no registrador %ebp-28, agora como word.
```
33 movw %ax, -32(%ebp)
```
Guardamos o valor de y no registrador %ebp-32, agora como word.
```
34 movl %gs:20, %eax
```
Copia o canary(protetor de pilha) para o registrador %eax.
```
35 movl %eax, -12(%ebp)
```
Copia o protetor de pilha %eax para a pilha.
```
36 xorl %eax, %eax
```
Aqui iremos limpar o registrador %eax, ou seja, %eax = 0.
```
37 movzwl -28(%ebp), %eax
```

Estamos passando o valor do registrador %ebp-28 (x) para o registrador %eax.

38 movw %ax, -24(%ebp)

Estamos passando o valor de x para %ebp-24, agora somente como os 2 bytes do word.

39 movzwl -32(%ebp), %eax

Estamos passando o valor do registrador %ebp-32 (y) para o registrador %eax.

40 movw %ax, -22(%ebp)

Estamos passando o valor de y para %ebp-22, agora somente como os 2 bytes do word

41 leal -20(%ebp), %eax

O registrador %eax recebe o conteúdo do endereço de memória %ebp-20, que é o primeiro byte da estrutura st2 s2.

42 pushl -24(%ebp)

Vamos colocar %ebp - 24 no topo da pilha (conteúdo de x e y, visto que são dois word, 2 + 2 = 4).

43 pushl %eax

Vamos colocar o valor de %eax s2 no topo da pilha.

44 call opera

Iremos chamar a função "opera". Corresponde a linha "s2 = opera(s1);" no código em c.

45 addl $4, %esp

Vamos liberar 1 espaço de 4 bytes da pilha.

46 movl -20(%ebp), %edx

O registrador %edx receberá o que está em %ebp - 20, ou seja, o primeiro parâmetro da estrutura st2 s2, que é s2.soma, no código C.

47 movl -16(%ebp), %eax

O registrador %eax receberá o que está em %ebp - 16, ou seja, o segundo parâmetro da estrutura st2 s2, que é s2.dif, no código C

48 imull %edx, %eax

Vamos fazer a multiplicação completa de 64bits com sinal de %edx com %eax e %edx, guardando os primeiros 4 bytes em %eax. Equivale a multiplicação s2.soma * s2.dif no código C.

49 movl -12(%ebp), %ecx

Copia o conteúdo de %ebp-12 (valor do canary) para %ecx

50 xorl %gs:20, %ecx

Realiza um xor entre o valor original do canary e o valor que foi passado para a pilha

51 je .L5

Compara os valores anteriores. Se eles forem iguais há um desvio incondicional para .L5, caso o xor anterior dê 0, o programa é desviado para .L5

```
52 call __stack_chk_fail
Chama uma rotina para tratar da corrupção da pilha, caso o ZF não for
igual a zero.
.L5:
53 leave
Estamos preparando a pilha para retornar à função que chamou calcula.
Nesse momento, %ebp volta a possuir o endereço da base da rotina que
chamou o calcula.
54 ret
Iremos realizar o retorno para a função chamadora de calcula. Essa
instrução corresponde à "return s2.soma * s2.dif;" em código c.
```

b) (5) Assuma que main chama calcula. Preencha o desenho da pilha imediatamente antes de executar calcula, mostrando os argumentos, a base de main, e o RIP para retorno a main. Indique o endereço alinhado em x16, pelo padrão do Linux. A pilha cresce de cima para baixo em direção a endereços menores. Use mais linhas se julgar necessário.

endereço (em relação a %esp)	PILHA (4 bytes)	descrição
%ebp	OFP da rotina que chamou a rotina caller	Base da função que chamou calcula
...	...	...
%esp + 6	y	Segundo parâmetro de calcula
%esp + 4 (nx16)	x	Primeiro parâmetro de calcula
%esp	RIP CALCULA	End para o retorno à função que chamou calcula

endereço (em relação a %esp)	PILHA (4 bytes)	descrição
%ebp	OFP da rotina que chamou a rotina caller	Base da função que chamou calcula
...	...	...
%esp + 8	y	Segundo parâmetro de calcula
%esp + 4 (nx16)	x	Primeiro parâmetro de calcula
%esp	RIP CALCULA	End para o retorno à função que chamou calcula

c) (5) Agora complemente o desenho da pilha, a partir do início de execução da função calcula até imediatamente após a execução da linha 44, quando o controle é transferido para a função opera. Na coluna endereço, use a referência a %ebp(calcula), a base do registro de ativação da função. Marque todos os endereços x16. Na coluna descrição, use ”Após L?”para indicar qual instrução causou a alteração respectiva na pilha. Nesta coluna deve estar indicado o endereço das estruturas s1 e s2.

endereço (em relação a %esp)	PILHA (4 bytes)	descrição
%ebp	OFP da rotina que chamou a rotina caller	Base da função que chamou calcula
...	...	...
%ebp + 16	y	Segundo parâmetro de calcula
%ebp + 8 (nx16)	x	Primeiro parâmetro de calcula
%ebp+4	RIP CALCULA	End para o retorno à função que chamou calcula
%esp = %ebp	%ebp	Base de calcula (Após 28)
%ebp - 4		
%ebp - 8(x16)		
%ebp - 12	Canary	protetor de pilha (Após 35)
%ebp - 16		
%ebp - 20		
%ebp - 22	y	Salva y, apenas 2 bytes,Equivalente a s1.a2= y (Após 40)
%ebp - 24(x16)	x	Salva x,apenas 2 bytes,Equivalente a s1.a1 = x no código c(Após 38)
%ebp - 28	x	Salva x, como word,Equivalente a s1.a1 = x no código c (Após 32)
%ebp - 32	y	Salva y como word,Equivalente a s1.a2= y(Após 33)
%ebp - 36		
%esp - 40 (x16)		
%esp - 42	s1.a2=y	Equivalente a s1.a2= y no código c (Após 42)
%esp - 44	s1/s1.a1 = x	Salva a estrutura s1, ou seja x e y na pilha juntos, visto que são word(2bytes). Equivalente a s1.a1 = x no código c (Após 42)

%esp - 48	&s2	Ponteiro para a estrutura s2 (Após 43)
%esp - 52 = %esp	RIP opera	Endereço de retorno de opera (Após 44)

d) (5) Agora continue o desenho da pilha, a partir do início de execução da função opera até imediatamente após a execução da linha 25, quando o controle retorna para a função calcula. Na coluna endereço, use a referência a %ebp(opera), a base do registro de ativação da função. Marque todos os endereços x16. Na coluna descrição, use ”Após L?” para indicar qual instrução causou a alteração respectiva na pilha. Ao alterar o topo da pilha para retornar espaços, mantenha o conteúdo da memória (que ocorre na prática) e indique para onde está o topo apontando após a linha 25 ser executada.

endereço (em relação a %esp)	PILHA (4 bytes)	descrição
%ebp	OFP da rotina que chamou a rotina caller	Base da função que chamou calcula
...	...	...
%ebp + 16	y	Segundo parâmetro de calcula
%ebp + 8 (nx16)	x	Primeiro parâmetro de calcula
%ebp+4	RIP CALCULA	End para o retorno à função que chamou calcula
%esp = %ebp(calcula)	%ebp	Base de calcula (Após 28)
%ebp(calcula) - 4		
%ebp(calcula) - 8(x16)		
%ebp(calcula) - 12	Canary	protetor de pilha (Após 35)
%ebp(calcula) - 16		
%ebp(calcula) - 20		
%ebp(calcula) - 22	y	Salva y, apenas 2 bytes,Equivalente a s1.a2= y (Após 40)
%ebp(calcula) - 24(x16)	x	Salva x,apenas 2 bytes,Equivalente a s1.a1 = x no código c(Após 38)
%ebp(calcula) - 28	x	Salva x, como word,Equivalente a s1.a1 = x no código c (Após 32)
%ebp(calcula) - 32	y	Salva y como word,Equivalente a s1.a2= y(Após 33)
%ebp(calcula) - 36		
%ebp(calcula) - 40 (x16)		

%ebp(calcula) - 42	s1.a2=y	Equivalente a s1.a2= y no código c (Após 42)
%ebp(calcula) - 44	s1/s1.a1 = x	Salva o primeiro byte da estrutura s1, ou seja x e y na pilha juntos, visto que são word (2bytes). Equivalente a s1.a1 = x no código c (Após 42)
%ebp(calcula) - 48	&s2	Ponteiro para a estrutura s2 (Após 43)
%ebp(calcula) - 52	RIP opera	Endereço de retorno de opera (Após 44)
%ebp(opera) (x16)	OFP de calcula	Frame pointer de calcula (Após 3)
%ebp(opera)-4	guarda.dif	Guarda a diferença de s1.a1 e s2.a2 (Após 17)
%ebp(opera)-8	guarda.soma	Guarda a soma de s1.a1 e s2.a2 (Após 10)
%ebp(opera)-12		
%ebp(opera)-16 = %esp		

e) (5) Expresse a instrução na linha 25 em termos de uma sequência equivalente de código de montagem com operação na pilha (pushl/popl) e instrução sobre os registradores %eip e %esp.

Na linha 25 temos a instrução " ret $4" que tem a função de devolver o controle à função calcula. A equivalência com operações na pilha fica:

```
popl %eip
popl %ebp
```

Com esse conjunto de operações iremos retornar da função, devolver o controle a função calcula e liberar 1 espaço de 4 bytes da pilha (soma 4 ao %esp).

f) (5) Análise como a função opera monta a estrutura guarda na pilha da função calcula. Indique o que a função opera retorna em %eax. Descreva a estratégia para passar uma estrutura como argumento para uma função e como receber uma estrutura de retorno.

A rotina opera é uma função que retorna uma estrutura maior que 4 bytes. Assim, o retorno padrão,pelo %eax, de enviar por ele uma cópia do valor retornado, não acontece. Assim, ocorre uma estratégia diferente da normal para retornar essas estruturas maiores.

Primeiramente passamos um endereço "secreto" para a função opera quando ela é chamada (linha 43 no código de montagem), que é referente ao endereço de memória da

estrutura que receberá o retorno da função. Nesse caso estamos passando o endereço da estrutura “s2”.

Agora dentro da função, o endereço passado (&s2) é usado como referência de onde destinar os valores dos campos da estrutura “guarda”, ou seja, cada campo da estrutura “guarda” é copiado para os da estrutura passada como referência, nesse caso “s2”.

Logo, ocorre a manipulação da estrutura da função (opera) pela função (soma), como visto nas linhas 21 e 22 do código de montagem.

Como a montagem é feita dentro da pilha da função chamadora, só resta retornar a posição de memória da estrutura montada, fazendo com que a função chamadora saiba em que endereço está a estrutura em questão. Assim %eax recebe o endereço de memória da estrutura (linha 23) e é retornado em seguida (linha 25)

Questão 2

Procura-se resgatar as declarações perdidas de struct_a e a definição D, a partir do código de montagem referente ao fragmento de código C abaixo. Sabe-se que struct_a contém apenas os elementos idx e x[?].

```
typedef struct {
      int left;
      a_struct a[D];
      int right ;
      } b_struct;
void test(int i, b_struct *bp) {
      int n = bp->left + bp->right;
      a_struct *ap = &bp->a[i];
      ap->x[ap->idx] = n;}
```

a) (5) Faça a engenharia reversa, associando as linhas acima ao código C. Identifique o que está a sendo calculado, quais variáveis e ponteiros estão sendo manipulados.

Código de montagem:

```
test:
1 endbr32
Instrução de proteção dos processadores. Com esse comando, esse
endereço é validado e pode ser usado como  um desvio. Assim pode ser
usado para que fluxos não controlados sejam detectados.
2 pushl %ebx
Aumenta a pilha em 4 bytes e colocar %ebx no topo da pilha
3 movl 8(%esp), %ecx
```

Copia o conteúdo de %esp+8 (i) para %ecx. %ecx = i

4 movl 12(%esp), %edx

Copia o conteúdo de %esp+12 (ponteiro para bp) para %edx, %edx = *bp

5 imull $14, %ecx, %eax

%eax = %ecx * $14 , %eax = 14*i

6 movswl 16(%edx,%eax), %ebx

%ebx = *bp + 14i + 16

7 leal 0(,%ecx,8), %eax

%eax = 8i

8 subl %ecx, %eax

%eax - i, %eax = 7i

9 addl %ebx, %eax

%eax = %eax + %ebx, %eax = [16 + *bp + 14i] + 7i.

Sabendo que os 4 primeiros bytes da estrutura são do atributo da estrutura b_struct que é bp->left, visto que é um inteiro.

Como também temos um vetor, o 14i indica que sizeof(a) = 14.

O 7i é um auxiliar para acessar o valor de x dentro da outra estrutura, que nesse caso é a a_struct

10 movl 60(%edx), %ecx

Temos que %ecx = 60 + *bp, ou seja, %ecx = bp->right, assim, obtemos o segundo elemento da estrutura.

Como podemos ver na instrução da linha 11, temos o primeiro elemento da estrutura b_struct (bp->left) somado com outro elemento da estrutura.

Ao observar o código C, encontramos a linha n = bp->left + bp->right.

Logo chegamos que a instrução da linha 10 está alocando em %ecx o valor de bp->right

11 addl (%edx), %ecx

Iremos obter %ecx = bp->left + bp->right, ou seja, estamos passando para o registrador %ecx o valor de n obtido na linha "int n = bp->left + bp->right;" do código em c.

12 movw %cx, 4(%edx,%eax,2)

[bp + 4 +([bp + (i14) + 16] + i7) * 2] = n

O registrador %ecx (variável n) irá receber os 2 últimos bytes de *(p + 2*(%eax) + 4). Referente a linha 4 da função test no código em c.

Aqui temos que ap->x[ap->idx] = n.

```
Como idx está sendo usado como um index e está em bp->a[i] + 12 e
sabendo que o tamanho de a é 14, logo se idx está em offset de 12
bytes, então sizeof(idx) = 2 e idx é então um short int

Como a instrução é movw, então o valor que está sendo guardado é dentro
do vetor x é um short int, logo temos que x é um vetor de short int

13 popl %ebx
Faz um pop em %ebx
14 ret
Retorno para a RIP da Main
```

b) (5) Encontre o valor de D, apresentando argumentos sólidos para a dimensão ao do vetor a.

Na letra a, sabemos que sizeof(a) = 14 e que o tipo de x é um short int. Podemos observar que na linha 10 obtemos bp-> right deslocando 60 espaços a partir de bp e lembrando que bp->left ocupa os 4 primeiros bytes de p. Assim, podemos dizer que D*sizeof(a) = 60 - 4, ou seja, D * 14 = 56 e D = 4.

c) (5) Determine sizeof(idx) e sizeof(a) com justificativas claras.

Na linha 9 temos a instrução %eax = [16 + *bp + 14i] + 7i., onde o acesso à memória busca idx para somar com 7i. Como idx está sendo usado como um index e está em bp->a[i] + 12 e sabendo que o tamanho de a é 14, logo se idx está em offset de 12 bytes, então sizeof(idx) = 2 e idx é então um short int . O termo 14i indica que sizeof(a)=14 bytes.

d) (5) Determine o tipo do vetor x e suas possíveis dimensões com justificativas claras.

Como estamos usando movw, sabemos que o valor que está sendo guardado dentro do vetor x é um short int.

Sabendo que idx está em bp-> a[i] + 12, sabemos que a tem tamanho 14, logo se idx está em um offset de 12 bytes, há 12 bytes antes dele(que é o nosso vetor x). Como o nosso vetor é do tipo short int,ou seja, cada número ocupa 2 bytes, então temos que 12/2 = 6. Então a dimensão de x é 6 (x[6]) .

e) (5) Identifique as possíveis declarações da estrutura struct a, sabendo que os únicos campos nesta estrutura são idx e o vetor x. Você tem que justificar os tipos das variáveis e a dimensão dos vetores de forma clara. Ao final, indique as declarações viáveis para struct_a.

Como falado nas questões anteriores, sabemos x é um vetor do tipo short int e que sua dimensão é 6 e que idx está em um offset de 12 bytes e que sizeof(a) = 14 então ele é um short int posicionado após o vetor x, considerando o alinhamento dentro da a_struct, temos que:

```
typedef struct{
   short int x[6];
   short int idx;
}a_struct;
```

Questão 3

Sabendo que o programador fez sua escolha de registradores, escreva o comando asm apagado que originou o código de montagem correspondente, usando a sintaxe (nominal ou posicional) que preferir:

```
int main() {
int x=5, z=3, y=1;
asm(...
);
printf("x =%d, y = %d\n", x,y);
return z;
}
```

```
main:
1 endbr32
Instrução de proteção dos processadores. Com esse comando, esse
endereço é validado e pode ser usado como  um desvio. Assim pode ser
usado para que fluxos não controlados sejam detectados.
2 leal 4(%esp), %ecx
3 andl $-16, %esp
4 pushl -4(%ecx)
5 pushl %ebp
6 movl %esp, %ebp
%esp = %ebp
7 pushl %edi
8 pushl %esi
9 pushl %ebx
10 pushl %ecx
11 subl $24, %esp
Abre 6 espaços
12 movl $5, -36(%ebp)
%ebp-36 = 5
13 movl $3, -32(%ebp)
%ebp-32 = 3
14 movl $1, -28(%ebp)
%ebp-28 = 1
15 movl -36(%ebp), %eax
```

```
%eax = 5
16 movl -32(%ebp), %edx
%edx = 3
17 movl %eax, %esi
%esi = 5
18 movl %edx, %edi
%edi = 3

#APP início do código asm
19 leal 5(%esi), %edx
%edx = 10
20 addl %edi, %edi
%edi = 3+3 = 6 (z+z)
21 movl %edi, %ecx
%ecx = 6 (z)
#NO_APP fim do código asm
22 movl %edi, %eax
%eax = 6 (z)
23 movl %esi, %ebx
%ebx = 5 (x)
24 movl %edx, -28(%ebp)
%ebp-28 = 10 // y é saída, pois está armazenando na pilha
25 movl %ebx, -36(%ebp)
%ebp-36 = 5 (x) //x é saída, pois está armazenando na pilha
26 movl %eax, -32(%ebp)
%ebp-32 = 6 (z)
27 subl $4, %esp
Abre 2 espaços
28 pushl -28(%ebp)
Push 10
29 pushl -36(%ebp)
Push 5
30 pushl $.LC0
31 call printf
32 addl $16, %esp
Diminui 4 espaços
33 movl -32(%ebp), %eax
%eax = 6 (z)
34 leal -16(%ebp), %esp

35 popl %ecx
36 popl %ebx
37 popl %esi
```

```
38 popl %edi
39 popl %ebp
40 leal -4(%ecx), %esp
41 ret
```

a) (5) Quais os registradores que GCC escolheu inicialmente para as variáveis x, y e z? Justifique.

O código c nos diz que: int x=5, z=3, y=1

No código de montagem temos que
12 movl $5, -36(%ebp)
%ebp-36 = 5
13 movl $3, -32(%ebp)
%ebp-32 = 3
14 movl $1, -28(%ebp)
%ebp- 28 = 1
15 movl -36(%ebp), %eax
%eax = 5
16 movl -32(%ebp), %edx
%edx = 3
17 movl %eax, %esi
%esi = 5
18 movl %edx, %edi
%edi = 3

Após essa sequência de comandos temos que:
%esi = x, %edi = z

Para y, o GCC não escolheu para nenhum registrador e seu conteúdo ficou em %ebp-28, visto que **y = 1 e %ebp-28 = 1**

b) (5) Quais os registradores escolhidos pelo programador no comando ASM para armazenar as variáveis x, y e z? Tem que justificar a dedução com argumentos sólidos.

No código c, vemos que a função printf tem como parâmetro x e y, respectivamente. No trecho #NO_APP, antes da chamada da função temos

```
28 pushl -28(%ebp)
29 pushl -36(%ebp)
30 pushl $.LC0
```

Sabendo que .LC0 está passando a string, temos que o conteúdo de x = %ebp-36 e y = %ebp-28 são passados como parâmetros da chamada da função printf.

Nesse momento temos que:

%ebp - 28 = 10 e %ebp-36 = 5, logo x = 5 e y = 10.

Voltando na linha 19, temos

```
leal 5(%esi), %edx
```

Nesse momento temos %esi = 5, logo **%edx recebe 10,** que depois esse conteúdo é colocado em %ebp - 28 na linha 24 com **movl %edx, -28(%ebp),** antes de %ebp-28 receber um push para servir como parâmetro da função, logo descobrimos que o registrador escolhido para y é o **%edx**.

No trecho #APP temos

```
19 leal 5(%esi), %edx
%edx = 10 (y)
20 addl %edi, %edi
%edi = 3+3 = 6 (z = 2*z)
21 movl %edi, %ecx
%ecx = 6 (z)
```

Como o GCC tinha escolhido %edi para z, aqui temos que o seu valor está apenas multiplicando por 2. Depois temos que o resultado foi passado então para o registrador %ecx

Como não houve modificação no registrador onde estava **x**, então ele continua sendo referenciado pelo registrador **%esi.**

Resumindo:

%ecx recebe z
%esi recebe x
%edx recebe y

c) (5) Dê argumentos sólidos para configurar a lista de saída do comando ASM.

Do exercício anterior, sabemos que o programador escolheu %esi para receber x e %edx para receber y. Porém, se observarmos as seguintes linhas do código de montagem temos:

```
22 movl %edi, %eax
%eax = 6 (z)
23 movl %esi, %ebx
%ebx = 5 (x)
24 movl %edx, -28(%ebp)
%ebp-28 = 10 // y é saída, pois está armazenando na pilha
25 movl %ebx, -36(%ebp)
%ebp-36 = 5 (x) //x é saída, pois está armazenando na pilha
```

```
26 movl %eax, -32(%ebp)
%ebp-32 = 6 (z)
```

Podemos observar que na linha 24 estamos armazenando o valor contido em %edx na pilha, isso nos mostra que y (que estava salvo em %edx) é um operador de saída. Já na linha 23 temos que o registrador %ebx irá receber o valor do registrador %esi (o valor de x) e na linha 25 iremos colocar o valor do registrador %ebx na pilha, mostrando que x será um operador de saída.

Sabendo que o gcc escolheu o registrador %edi para armazenar o z, temos na linha 22 o valor de **%edi** sendo passado para o registrador %eax, logo, teremos z armazenado em %eax. Na linha 26 observamos que o valor contido em %eax é salvo na pilha, assim, teremos que z também será um operador de saída.

Com isso, temos que a lista de saída será

[x] “=b”(x), [y] “=d” (y), [z] “=a”(z) ou **“=b”(x), “=d” (y), “=a”(z)**

d) (5) Dê argumentos sólidos para configurar a lista de entrada do comando ASM.

Pelas linhas do código de montagem a seguir sabemos que só precisamos de %esi e %edx para que seja possível fazer a atribuição dos valores em %edx e %ecx:

19 leal 5(%esi), %edx
20 addl %edi, %edi
21 movl %edi, %ecx

Como não precisamos nem de %edx e nem de %ecx, apenas dos valores de **x** e **z** então nossa lista de entrada no comando asm é formada por:

“S” (x), [z] “D” (z) ou **“S” (x), “D” (z)**

e) (5) Escreva o comando ASM apagado, usando tanto a notação posicional como a nominal.

Aproveitando os resultados das questões anteriores quanto a lista de entrada e saída do comando ASM no código de montagem, chegamos que o código que foi apagado foi o seguinte:

Notação posicional:

```
asm(
      "leal 5(%0), %1 \n\t"       //y = x + 5
      "addl %2,%2  \n\t"       //z = z + z
      "movl %3,%2  \n\t"      //%ecx = z

      :"=d" (y), "=b"(x), "=a"(z) //operandos de saída
      :"S" (x), "D" (z)            //operandos de entrada
      :"%ecx"               //registradores modificados
);
```

Notação Nominal:

```
asm(
      "leal 5(%%[x]), %[y] \n\t"       //y = x + 5
      "addl %[z],%[z]      \n\t"       //z = z + z
      "movl %[z], %%ecx    \n\t"      //%ecx = z

      :[y] "=d" (y), [x] "=b"(x), [z] "=a"(z) //operandos de saída
      :"S" (x), "D" (z)            //operandos de entrada
      :"%ecx"              //registradores modificados
);
```