Lista 2 - Comp Prog

Alunos: Luiz Rodrigo Lacé Rodrigues (DRE:11804983)
Livia Barbosa Fonseca (DRE:118039721)

Questão 1) Na lista 1 foi dado o seguinte programa C:

```
int main (){
        float x = 4.3;
        double z = 3.2, y;
        y = x - z;
        printf ("y = 4.3 - 3.2 = %10.13f \n", y);}
```

Compilando com “gcc -m32 -O2 -fno-PIC -S”, descartadas algumas linhas de diretivas, temos:

```
.LC1:
        .string "y = 4.3 - 3.2 = %10.13f \n"
main:
1 endbr32
2 leal 4(%esp), %ecx
3 andl $-16, %esp
4 pushl -4(%ecx)
5 pushl %ebp
6 movl %esp, %ebp
7 pushl %ecx
8 subl $8, %esp
9 pushl $1072798105
10 pushl $-858993460
11 pushl $.LC1
12 call printf
13 movl -4(%ebp), %ecx
14 addl $16, %esp
15 xorl %eax, %eax
16 leave
17 leal -4(%ecx), %esp
18 ret
```

a)Desenhe o conteúdo da pilha imediatamente após a execução da linha 12. Na coluna endereço, represente sempre a situação dos registradores ebp e esp no momento do desenho. Referencie os endereços das linhas da pilha sempre em relação à base de main após ela ser estabelecida. Na coluna comentários, preencha com “Após L?”, identificando a linha da instrução de montagem que gera o preenchimento daquela linha da pilha. Acrescente algum comentário que for pertinente. Marque com x16 os endereços que estiverem alinhados em múltiplos de 16.

Endereço	Conteúdo da Pilha <4 bytes>	Comentários
OFP		Base da pilha
...	...	...
%ecx		primeiro parâmetro da main (Apos L2)
%ecx - 4	RIP SÓ	End para retorno ao SO
		espaço para alinhamento
%esp+36 (n x 16)	...	múltiplo de 16 (Após L3)
%ebp + 4 = %esp+32	RIP SÓ	End para retorno ao SO (Após L4)
%ebp = %esp+28	OFP	Base da main (Após L6)
%ebp - 4 = %esp+24	%ecx	Endereço de %ecx (Após L7)
%ebp - 8 = %esp+20 (n x 16)		
%ebp - 16 = %esp+16		(Após L8)
%ebp - 20 = %esp+12	1072798105	Parte alta do double (Após L9)
%ebp - 24 = %esp +8	-858993460	Parte baixa do double (Após L10)
%ebp - 28 = %esp+4 (n x 16)	.LC1	End da lista de controle (Após L11)
%ebp - 32 = %esp	RIP main	Retorno para a função main (Após L12)

b)Qual a razão da existência da linha 7? Justifique, pois qualquer ação feita pelo GCC tem que ter uma justificativa plena.

A linha 7 (pushl %ecx) salva o valor de %ecx na pilha. No lugar de pegarmos o topo da pilha,usamos “leal 4(%esp),%ecx” para pegarmos uma referência para o primeiro argumento da função main(argc). Esse endereço nos permite acessar o topo da pilha com -4(%ecx), o qual nos retorna o RIP do SO.

Após o comando “andl $-16,%esp”, na linha 3, o topo da pilha apontará para o próximo end múltiplo de 16, assim perderemos a referência para o RIP do SO. Logo, salvamos o valor de %ecx na pilha para termos uma referência para o RIP SO (pushl -4(%ecx)). Principalmente, porque a linha 12, com a chamada do call printf, o registrador %exc é zerado.

Salvar o valor desse registrador na pilha nos permite restaurar %ecx e recuperar a referência para o RIP do SO.

c) Qual a razão da execução da linha 4? Justifique, pois qualquer ação feita pelo GCC tem que ter uma justificativa plena.

- O comando da linha 4 (pushl -4(%ecx)) insere no topo da pilha o RIP SO que foi perdido devido ao comando “andl $-16, %esp”.

d) Justifique a existência da linha 15.(xorl %eax, %eax)

- O comando xor(OU-EXCLUSIVO) de valores igual que são iguais (%eax e %eax), usamos para limpar o valor do registrador %eax. Assim a rotina main irá devolver, no ret, o valor de %eax que será igual a 0.

e) No código de montagem mostrado, existe alguma linha que possa ser suprimida, sendo desnecessária? Justifique.

- A linha 14 (addl $16, %esp) pode ser suprimida. Como a próxima operação de pilha depois do addl seria o desmonte da pilha, igualando %esp a %ebp e decrementando o topo da pilha com %ebp recebendo o OFP do início do programa. Como não temos outras operações de pilha entre o “addl $16, %esp” e o leave, e a mesma já está alinhada a um múltiplo de 16 bytes, não há motivo para liberar o espaço ocupado pelos argumentos empilhados para a chamada de printf, visto que isso será feito no comando leave.

f) Normalmente após um leave é executado o ret. Por que no código acima, entre o leave e o ret existe a linha 17?

- O comando da linha 17 é “leal -4(%ecx),%esp”. O registrador %esp, que marca o topo da pilha, recebe o endereço de memória (%ecx-4). Tal endereço é a posição da pilha onde o RIP do SO é guardado. Assim, %esp apontará para a RIP SO, sendo necessário para a saída da função main e retorno ao SO.

g) Qual o significado das constantes 1072798105 e -858993460? Justifique. Pode usar o resultado da lista 1.

Sabendo que o double tem 8 bytes e estamos na arquitetura de 32 bits, precisamos dividir o double em até 32 bits cada.

Na hora da compilação do programa, quando chegamos no resultado de y = x - z, como z é um double, nosso resultado também será. Assim, quando formos passar o resultado de y para o printf, precisamos passar o número como dois parâmetros. A parte alta do double (1072798105) e a parte baixa (-858993460).

Sabemos disso, visto que, o resultado encontrado na lista 1 foi.

Como o resultado de y hexa é = 1,1 = 0x3FF19999CCCCCCCC e
1072798105 = 0x3FF19999
-858993460 = 0xCCCCCCCC

h) Considerando como m o custo de execução de uma instrução que acessa a memória física e n o custo de instrução que não acessa a memória, calcule a estimativa de custo do código de montagem.

Vamos analisar cada linha do código dado para ver quais instruções acessam ou não a memória:

```
1    endbr32
2    leal 4(%esp), %ecx //Acessa - referenciando %ecx em %esp+4 (?)
3    andl $-16, %esp  //Não acessa - só mexe na pilha
4    pushl -4(%ecx)  //Acessa - estamos colocando na pilha
5    pushl %ebp  //Acessa a memória - estamos colocando na pilha
6    movl %esp, %ebp //Acessa - temos que ler o que está em %esp
7    pushl %ecx //Acessa a memória - estamos salvando %ecx na pilha
8    subl $8, %esp //Não acessa - só mexe na pilha
9    pushl $1072798105 //Acessa - estamos salvando um número
10   pushl $-858993460 //Acessa - estamos salvando um número
11   pushl $.LC1 //Acessa - salva o que está em $.LC1
12   call printf //Acessa - Dá push no endereço da RIP main
13   movl -4(%ebp), %ecx //Acessa - temos que ler o que está em %ebp-4
14   addl $16, %esp //Não acessa - só mexe com o topo da pilha
15   xorl %eax, %eax //Não acessa - operação entre %eax
16   leave //Acessa - (operação movl + popl)
17   leal -4(%ecx), %esp //Acessa - referenciando %esp em %ecx-4 (?)
18   ret //Acessa - Retira do topo da pilha o endereço de retorno
```

Fazendo o cálculo temos:

Custo =13m + 4n

Onde, m é o custo de uma instrução que acessa a memória e n é o custo de uma que não acessa a memória.

Questão 2) Na lista 1 foi dado o seguinte programa:

```
int main (){
        float x = 4.3;
        double z = 3.2, y;
        y = x - z;
        printf ("y = 4.3 - 3.2 = %10.13f \n", y);}
```

Compilando com “gcc -m32 -fno-PIC -S”, descartadas algumas linhas de diretivas, temos:

```
.LC2:
        .string "y = 4.3 - 3.2 = %10.13f \n"
main:
1 endbr32
2 leal 4(%esp), %ecx
3 andl $-16, %esp
4 pushl -4(%ecx)
5 pushl %ebp
6 movl %esp, %ebp
7 pushl %ecx
8 subl $36, %esp
9 flds .LC0
10 fstps -28(%ebp)
11 fldl .LC1
12 fstpl -24(%ebp)
13 flds -28(%ebp)
14 fsubl -24(%ebp)
15 fstpl -16(%ebp)
16 subl $4, %esp
17 pushl -12(%ebp)
18 pushl -16(%ebp)
19 pushl $.LC2
20 call printf
21 addl $16, %esp
22 movl $0, %eax
23 movl -4(%ebp), %ecx
24 leave
25 leal -4(%ecx), %esp
26 ret
        .align 4
.LC0:
        .long 1082759578
        .align 8
.LC1:
        .long 2576980378
        .long 1074370969
```

a)Justifique as diretivas .align 4 e .align 8. O que representa as constantes nos endereços .LCO e .LC1?

.align 4 e .align 8 asseguram que a próxima variável estará num endereço múltiplo de 4 e 8 respectivamente.

.LCO:
long 1082759578 = 0x4089999A = 4,3 em precisão simples (lista 1)

.LC1:
.long 2576980378 = 0x9999999A parte baixa de 3,2 (lista 1)
.long 1074370969 = 0x40099999 parte alta de 3,2 (lista 1)

b) Comente cada uma das linhas de 9 a 13, indicando o que elas causam. Indique o conteúdo de ST(0) após a execução de cada linha. Aponte as ineficiências de movimentação que pode detectar.

```
 9      flds .LC0
10      fstps -28(%ebp)
11      fldl .LC1
12      fstpl -24(%ebp)
13      flds -28(%ebp)
```

Linha 9-) flds .LC0 (load floating point) de precisão simples (32 bits,sufixo s)
Coloca o valor que está no .LCO (push) na pilha FPU. Então, temos que o valor decimal long 1082759578 (4,3 em precisão simples) será adicionado em ST(0).

Linha 10-) fstps -28(%ebp) (store float floating point and pop) de precisão simples (32bits,sufixo s)
Estamos movendo o conteúdo de ST(0),o decimal 1082759578(4,3 em precisão simples),para o endereço %ebp-28 no formato simples (sufixo s) e damos pop na pilha FPU (decrementamos o topo)

Linha 11-) fldl .LC1 (load float point) de precisão dupla (64bits, sufixo l)
Adiciona os valores que estão em .LC1 (push) na pilha FPU. Então teremos o valor de 3,2 salvo na pilha FPU. O conteúdo de ST(0) nesse momento é o decimal são as constantes de .LC1(2576980378 e 1074370969, ou seja 0x400999999999999A em hexa)

Linha 12-) fstpl -24(%ebp) (Store Float point and pop) de precisão dupla.
Vamos mover o conteúdo de ST(0) para a posição de memória %ebp-24 e dar um pop na FPU. O conteúdo de ST(0) agora não possui nada.

Linha 13) flds -28(%ebp) (Load Floating Point) de precisão simples

Damos um push novamente na FPU com o conteúdo referenciado no endereço %ebp-28, o decimal 1082759578 (0x4089999A = 4.3 em precisão simples). O conteúdo de ST(0) agora é o decimal 1082759578

Quanto as ineficiências bastaria carregar .LC0 da memória (linha 9), fazer a conta com esses dois números com a instrução fsubl .LC1 (só .LC0 está na FPU, mas podemos fazer a conta quando um número está em FPU e outro na memória. Ex.: linha 14) - e colocar o resultado na memória (dar pop em FPU), já que será passado como argumento para printf.

c) Comente cada uma das linhas de 14 e 15, indicando o que elas causam. Indique o conteúdo de ST(0) após a execução de cada linha.

```
14      fsubl -24(%ebp)
15      fstpl -16(%ebp)
```

Linha 14 -)
Subtrai o valor que está na posição de memória %ebp-24 (1074370969 = 0x40099999, parte alta de 3,2) do valor que está em ST(0) (1082759578 = 0x4089999A = 4,3)

Por se tratar de uma operação com sufixo l, subtrai o conteúdo de %ebp-24 0x40099999 agora em double 0x400999999999999A do conteúdo que está em ST(0) (0x4089999A) na FPU. Antes da subtração a mantissa de 0x4089999A é completada com zeros para 52 bits. Chegamos no resultado de 0x3FF19999CCCCCCCC e salvamos esse resultado em ST(0)

Linha 15 -)
Essa linha passa o conteúdo da parte baixa de 1.1 (0x3FF199990) para o endereço de %ebp - 16 e a parte alta de 1.1 (0x9999999A) para a pilha e faz um pop na pilha FPU, liberando a memória. Deixando ST(0) “vazia”.

d)

Endereço	Conteúdo da Pilha <4 bytes>	Comentários
OFP		Base da pilha
	...	
%esp +60 (nx16)	...	Alinhado em um múltiplo de 16 (Apos L3)
%esp + 56 = %ebp +4	RIP SO	Valor que estava no (%ecx - 4) (Apos L4)
%esp + 52 = %ebp	OFP	iguala a base da pilha ao topo (registro de ativação) (Apos L6)
%esp + 48= %ebp - 4	%ecx	guarda no topo da pilha o endereço de %ecx (Apos L7)
%esp + 44(nx16)= %ebp -8		
%esp + 40= %ebp -12	0x3FF199990	Parte alta de 1.1,resultado da subtração de 4.3 e 3.2 realizada na FPU, onde é passado pra memoria e a FPU sofre um pop(Apos L15)
%esp + 36= %ebp - 16	0xCCCCCCCC	Parte baixa de 1.1, resultado da subtração de 4.3 e 3.2 realizada na FPU, onde é passado pra memoria e a FPU sofre um pop (Apos L15)
%esp + 32= %ebp - 20	1074370969=0x40099999	parte alta de 3.2(precisão dupla)passado da ST(0) da FPU depois é dado um pop(precisão dupla(Apos L12)
%esp + 28(nx16)= %ebp -24	2576980378 = 0x9999999A	parte baixa de 3.2(precisão dupla)passado da ST(0) da FPU depois é dado um pop. (Apos L12)
%esp + 24= %ebp - 28	1082759578=0x4089999A	4.3 em precisão simples, passado da ST(0) da FPU depois é dado um pop. (Apos 10)
%esp + 20= %ebp -32		
%esp + 16= %ebp - 36		
%esp + 12 (nx16)= %ebp - 40	0x3FF199990	Push do endereço %ebp -12 (Parte alta de 1,1)(Apos L17)
%esp +8= %ebp - 44	0xCCCCCCCC	Push do endereço %ebp-16 (Parte baixa de 1,1)(Apos L18)
%esp+4= %ebp - 48	.LC2	Armazena o endereço de .LC2 que possui a string que será usada no printf (Apos L19)
%esp = %ebp - 52	RIP main	Retorno para a função main (Apos L20)

e) (5) Tente explicar a razão da Linha 16 e faça as observações pertinentes em relação a ineficiências no uso de memória e alinhamentos nas operações entre FPU x87 e a memória.

Linha 16 = subl $4, %esp
Cria um espaço na pilha para alinhar os parâmetros no topo da pilha para chamar o comando call printf. Entretanto, tais valores já estavam na pilha devido aos outros comandos de do cálculo da subtração realizada na FPU, o que demonstra uma ineficiência com o uso da memória.

Questão 3) Um valor inteiro foi apagado na rotina C abaixo.
int foo(unsigned int n) {unsigned int x; x = n/... ;return x;}
O código de montagem gerado pelo GCC com otimização -O1 é:
foo:
1 endbr32
2 movl $1374389535, %edx
3 movl %edx, %eax
4 mull 4(%esp)
5 movl %edx, %eax
6 shrl $3, %eax
7 ret
Comente cada linha, sob o ponto de vista de engenharia reversa, para descobrir o valor apagado. A justificativa tem que ser clara e todas as linhas têm que ser comentadas, sem exceção, de 1 a 7. Pode ser usada calculadora e eventuais conversões para binário e/ou hexadecimal, se necessárias, podem ser apresentadas sem o passo a passo.

```
1  endbr32
Essa instrução serve para proteger o processador para mudanças de
endereços quando realizamos chamadas de funções. Assim, conseguimos
manter o fluxo correto do programa.

2  movl $1374389535, %edx
Armazena o valor 1374389535 = 0x51EB851F para o registrador %edx

3  movl %edx, %eax
Copia para %eax o valor que estava em %edx(0x51EB851F)

4  mull 4(%esp)
Como %esp possui RIP foo,o endereço %esp+4 possui o endereço contendo o
o argumento n. Logo estamos fazendo a multiplicação completa e sem
sinal de 64 bits entre "n" e %eax (que nesse momento contém
0x51EB851F). Assim, obtendo n*0x51EB851F.(com 64 bits de saída em
%edx::%eax)

5  movl %edx, %eax
Move  o valor de %edx (0x51EB851F) para %eax, que tinha a parte baixa
da multiplicação anterior. Obtemos então n*0x51EB851F*2**-32

6  shrl $3, %eax
Desloca %eax ( 0x51EB851F ) 3 bits para a direita.
Ou seja, calculamos n*0x51EB851F*2**-32*2**-3= n*0x51EB851F*2**-35.
Finalmente chegamos em x = n/25, 25 era o nosso número faltante.

7  ret
Faz o retorno do valor contido em %eax (n/25)
```

Justificativa:
Temos que 0x51EB851F x 2^{-32} x 2^{-3} =

0x51EB851F = 0101 0001 1110 1011 1000 0101 0001 1111

0x51EB851F x 2^{-32} x 2^{-3} = 0.[0000 1010 0011 1101] 0111 0000 1010 0011

Temos que arredondar o valor, pois temos apenas 32 bits na representação unsigned int
0x51EB851F x 2^{-32} x 2^{-3} = 0.0000 1010 0011 1101 0111 0000 1010 0100

Fazendo o somatório das frações:

$$\frac{10}{16^2} + \frac{3}{16^3} + \frac{13}{16^4} + \frac{7}{16^5} + \frac{10}{16^7} + \frac{4}{16^8} \simeq \quad 0.04 = \frac{1}{25}$$

Com isso, descobrimos que x = $\frac{n}{25}$. Logo, a rotina está dividindo o parâmetro de chamada por 25 e retornando esse valor.

Questão 4) É dada uma rotina que recebe um valor inteiro com sinal e imprime o resultado da divisão, que pode ser negativo ou positivo. Determine o valor do divisor apagado na rotina C abaixo:

int foo (int n) {int x; x = n/... ; return x;}
O código de montagem gerado pelo GCC com otimização -O1 é:
foo:
1 endbr32
2 movl 4(%esp), %ecx
3 movl $-1307163959, %edx
4 movl %ecx, %eax
5 imull %edx
6 leal (%edx,%ecx), %eax
7 sarl $4, %eax
8 sarl $31, %ecx
9 subl %ecx, %eax
10 ret
Comente cada linha, sob o ponto de vista de engenharia reversa, para descobrir o valor apagado. Todas as linhas devem ser
justificadas. GCC insere instrução por uma razão que tem que ficar clara. Pode ser usada calculadora e eventuais conversões
para binário e/ou hexadecimal, se necessárias, podem ser apresentadas sem o passo a passo.

1 endbr32
Essa linha serve para proteger o processador para mudanças de endereços quando realizamos chamadas de funções. Assim, conseguimos manter o fluxo correto do programa.

2 movl 4(%esp), %ecx
Como %esp possui o RIP foo, %esp + 4 possui o endereço que contém o argumento n,nessa linha estamos copiando n para o registrador %ecx

3 movl $-1307163959, %edx
Armazena o valor (-1307163959) = (0xB21642C9) no registrador %edx.

4 movl %ecx, %eax
Salva no valor que está em %ecx para o registrador %eax. Fazendo %eax receber n.

5 imull %edx
Esse comando faz uma multiplicação com sinal completo de 64 bits entre os valores salvos em %edx e %eax. Estamos realizando uma multiplicação entre n e 0xB21642C9 (n* 0xB21642C9). Sendo assim, o sinal será estendido se caso for necessário e a parte alta do valor será salva em %edx e a parte baixa em %eax (a parte alta é 0xB21642C9*n* 2^{-32}).

```
6 leal (%edx,%ecx), %eax
```
O comando salva em %eax a soma entre %edx e %ecx (0xB21642C9*n*2^{-32}+n).

```
7 sarl $4, %eax
```
Realiza um shift aritmético de 4 bits à direita de n, do que está em %eax. Ou seja, temos agora em %eax (0xB21642C9*n*2^{-32}+n*2^{-4})

```
8 sarl $31, %ecx
```
Realiza um shift aritmético de 31 bits à direita de n. Assim, %ecx recebe (n*2^-31). Caso o número seja negativo, obtemos -1, caso seja positivo, obtemos 0.

```
9 subl %ecx, %eax
```
Guarda no registrador %eax o resultado da subtração entre %ecx e %eax, que é 0xB21642C9*n*2^{-32}+n*2^{-4}.Caso o número seja negativo, somamos 1 devido a divisão inteira negativa. Podemos colocar em evidência o n, assim o resultado final que será guardado no registrador %eax é n*(0xB21642C9*2^{-32}* 2^{-4}).
Como a variável x no código é signed, foi necessário o uso do shift aritmético. E também, como (0xB21642C9*2^{-32} *2^{-4}) equivale a multiplicar o n por 1/23
temos que %eax armazena o resultado de n/23 . Portanto, o inteiro removido foi 23.

```
10 ret
```
Retorna o valor em %eax e devolve o controle ao endereço de retorno que está em %esp.

Então temos que 0xB21642C9*2^{-32}*2^{-4} = 0.0000 1011 0010 0001 0110 0100 0010 1100 =

$$\frac{11}{16^2} + \frac{2}{16^3} + \frac{1}{16^4} + \frac{6}{16^5} + \frac{4}{16^6} + \frac{2}{16^7} + \frac{12}{16^8} \simeq \mathbf{0.04347826086}$$

Com isso, como 1/23 = 0.04347826086, descobrimos que x = $\frac{n}{23}$. Logo, a rotina está dividindo o parâmetro de chamada por 23 e retornando esse valor.