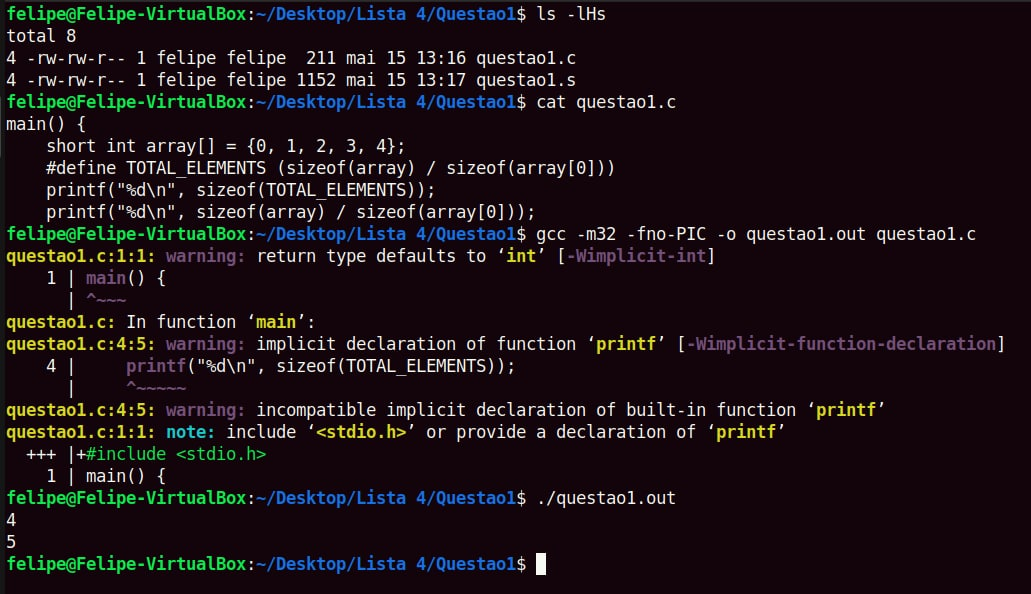
Lista 4 – Felipe Melo – Thalles Nonato

DRE Felipe: 119093752

DRE Thalles: 119058809

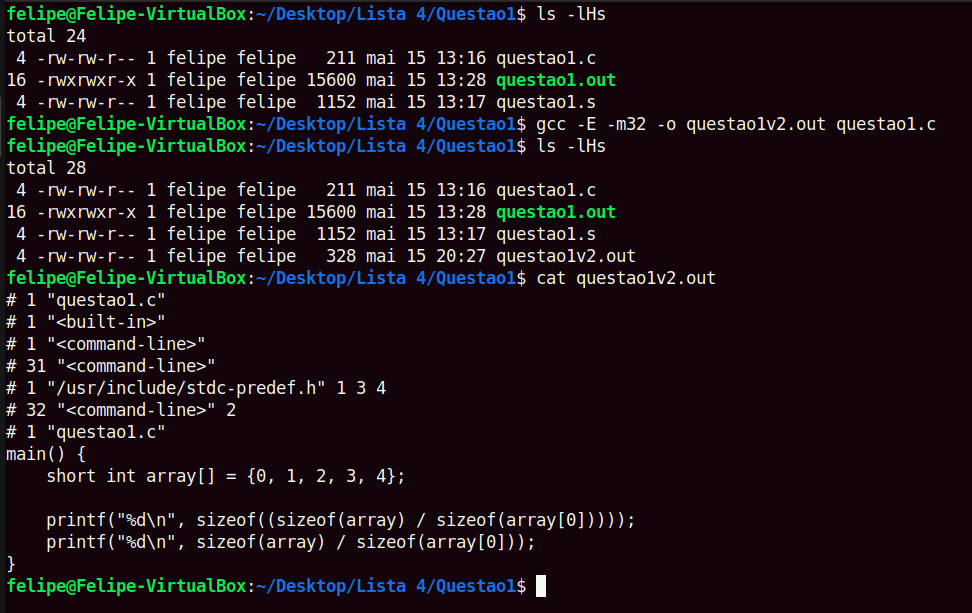
**Questão 1) a) Compile o código com a opção -m32 e gere um executável. Procure explicar a saída impressa. Seja detalhado, justificando cada valor impresso:**



O primeiro valor impresso é 4. Isso pois, apesar de a expressão sizeof(array) / sizeof(array[0]) resultar em 5, graças ao comando #define, ocorreu um cast desse valor para int, ocupando 4 bytes.

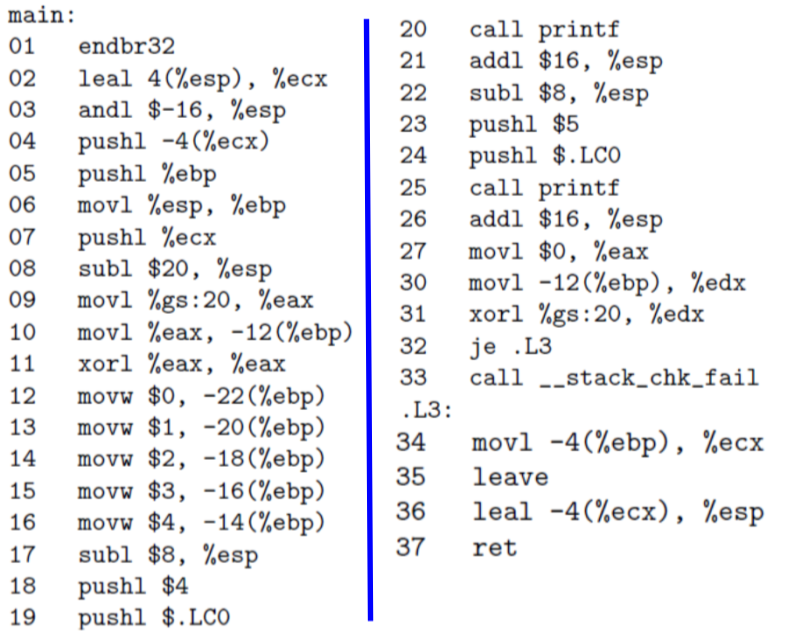
Já no segundo printf, o programa imprime o valor 5 pois se trata da razão entre o tamanho total do nosso array do tipo short int e o tamanho de um elemento isolado do array (array[0]). Dessa forma, teremos 10 / 2 = 5.

**Questão 1) b) Compile, sem includes como dado, usando as instruções -E -m32 e liste como resposta o que foi obtido, verificando se houve alguma substituição de sizeof no pré-processamento:**



Não houve substituição.

**Questão 1) c) Justifique cada linha no código gerado, associando ao código C mostrando ou a procedimento explicado:**

****

L2: copia o primeiro argumento argc para %ecx

L3: faz o topo da pilha se mover para o próximo endereço múltiplo de 16 após zerar os 4 bits menores do topo

L4: salva o RIP na pilha

L5 e L6: cria registro de ativação

L7: copia argc para o topo da pilha

L8: adiciona 5 posições na pilha

L9: prepara teste de corrupção

L10: copia %eax para (%ebp – 12)

L11: zera o registrador %eax

L12: copia o valor do índice 0 do array para (%ebp - 22)

L13: copia o valor do índice 1 do array para (%ebp - 20)

L14: copia o valor do índice 2 do array para (%ebp - 18)

L15: copia o valor do índice 3 do array para (%ebp - 16)

L16: copia o valor do índice 4 do array para (%ebp - 14)

L17: adiciona 2 posições na pilha

L18: salva o valor 4 no topo da pilha

L19: salva a string no topo da pilha

L20: chama a função printf

L21: diminui 4 posições da pilha

L22: adiciona 2 posições na pilha

L23: salva o valor 5 no topo da pilha

L24: salva a string no topo da pilha

L25: chama a função printf

L26: diminui 4 posições da pilha

L27: zera o registrador %eax

L30: copia (%ebp - 12) para %edx

L31: compara com valor original

L32: se for igual, não houve corrupção e faz um desvio para .L3 para retornar

L33: rotina para tratar corrupção da pilha

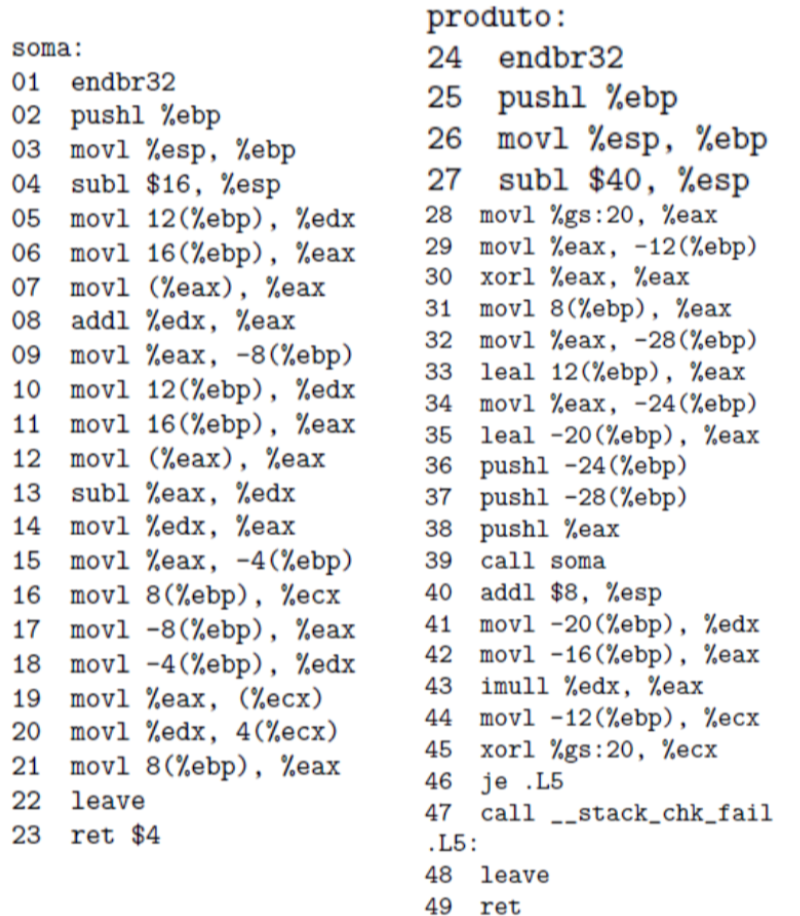
L34: copia (%ebp - 4) para %ecx

L35: copia %ebp para %esp e em seguida restaura o conteúdo anterior de %ebp na pilha

L36: faz o topo apontar para o RIP original

L37: transfere o controle para o endereço de retorno da função

**Questão 2) a) Comente cada linha do código, associando ao código C. Identifique os elementos das estruturas que estão sendo acessados e/ou manipulados. Justifique a ocorrência das instruções:**



L2 e L3: cria registro de ativação

L4: abre 4 posições na pilha

L5: copia s1.a para %edx

L6: copia s1.p para %eax

L7: faz \*s1.p para %eax

L8: soma s1.a e \*s1.p

L9: copia o resultado da soma para a variável resultado.soma

L10: copia s1.a para %edx

L11: copia s1.p para %eax

L12: pega o valor de s1.p, copiando \*s1.p para %eax

L13: subtrai s1.a e \*s1.p, armazenando o resultado em %edx (resultado.dif)

L14: copia resultado.dif para %eax

L15: salva resultado.dif na pilha

L16: copia (%ebp + 8) para %ecx

L17: copia resultado.soma para %eax

L18: copia (%ebp - 4) para %edx

L19: salva resultado.soma na pilha

L20: salva s1.a na pilha

L21: salva \*s1.p na pilha

L22: copia %ebp para %esp e em seguida restaura o conteúdo anterior de %ebp na pilha

L23: transfere o controle para o endereço de retorno da função e incrementa o topo da pilha de 8

L25 e L26: cria registro de ativação

L27: abre 10 posições na pilha

L28: move canário para %eax

L29: salva o canário na pilha

L30: zera %eax

L31: copia x para %eax

L32: salva x na pilha

L33: copia y para %eax

L34: salva o endereço de y (s1.p) na pilha

L35: copia (%ebp - 20), ou seja, s2, para %eax

L36: coloca s1.p no topo da pilha (preparando para a chamada da função soma)

L37: coloca s1.a no topo da pilha (preparando para a chamada da função soma)

L38: coloca s1 no topo da pilha (preparando para a chamada da função soma)

L39: chama a função soma

L40: diminui 2 posições da pilha

L41: copia s2.soma para %edx

L42: copia s2.dif para %eax

L43: computa (s2.soma \* s2.dif) e armazena o resultado em %eax

L44: move o canário para %ecx

L45: compara com valor original

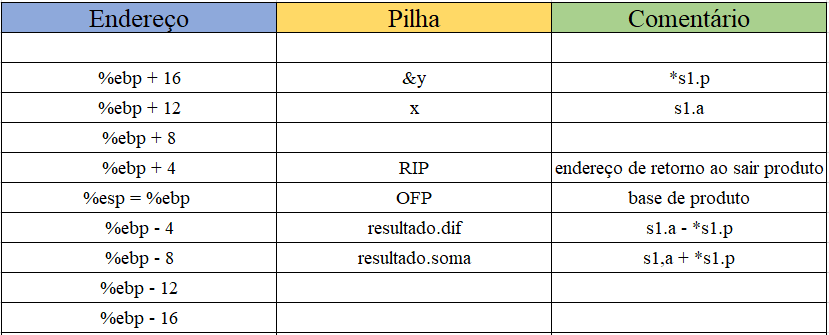
L46: se for igual, não houve corrupção e faz um desvio para .L5 para retornar

L47: rotina para tratar corrupção da pilha

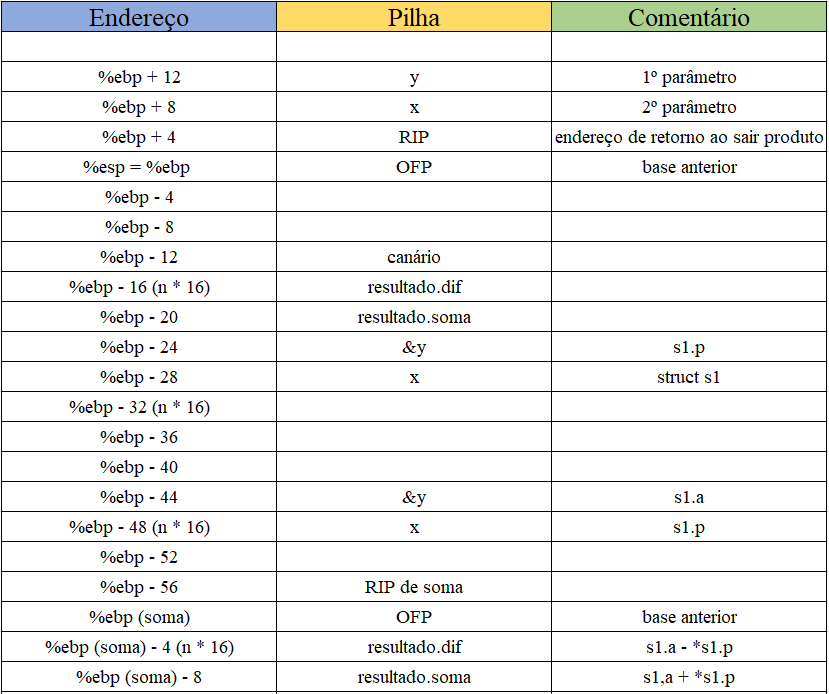
L48: copia %ebp para %esp e em seguida restaura o conteúdo anterior de %ebp na pilha

L49: transfere o controle para o endereço de retorno da função

**Questão 2) b) Faça um desenho inicial da pilha no estado imediatamente antes de entrar na execução da função produto, mostrando os parâmetros que foram passados para a função, bem como o RIP de quem chamou produto. Mostre que você conhece a passagem de parâmetros para a rotina produto. Todos os endereços que forem x16 devem ser identificados:**

****

**Questão 2) c) Complete o desenho da pilha, usando o modelo ex12-editado, a partir do início de execução da função produto, passando pela execução da função soma, mostrando todo o conteúdo da pilha até o retorno de soma, após execução da L23. Ao alterar o topo da pilha para retornar espaços, mantenha o conteúdo da memória e apenas atualize o ponteiro para o topo da pilha. O desenho da pilha deverá apresentar simultaneamente os registros de ativação de produto e o de soma. Identifique no campo descrição a base de cada um dos registros de forma clara (e.g., %ebp de prod, %ebp de soma):**



**Questão 2) d) Quais os endereços das estruturas s1 e s2, manuseadas por produto? Indique isso no desenho da pilha e justifique:**

Verificamos que a estrutura s1, pelo desenho da pilha, estará no endereço (%ebp - 28) e a estrutura s2 em (%ebp - 20).

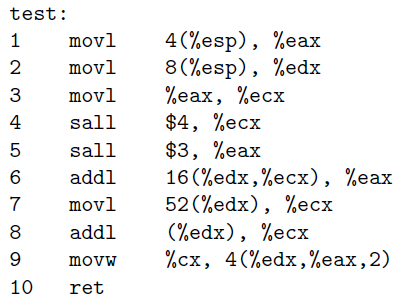
**Questão 2) e) O que a instrução na linha 23 faz? Pesquise e seja preciso:**

A instrução ret sem acompanhar nenhum valor apenas transfere o controle para o endereço de retorno da rotina, enquanto a instrução da linha 23 além disso também incrementa o topo da pilha de 8.

**Questão 2) f) Existe uma estratégia geral para o retorno de estruturas por uma função. Analise como a função soma sabe onde montar a estrutura resultado na pilha da função produto, o que a função soma retorna em %eax e descreva a estratégia. Dê argumentos sólidos baseados no exemplo acima.**

A função recebe um parâmetro camuflado, que é um ponteiro para a estrutura de retorno. Quando a função soma retorna, a produto consegue acessar a estrutura de retorno, pois ela está em seu registro de ativação.

**Questão 3) a) Faça a engenharia reversa, associando as linhas ao código C. Identifique o que está sendo calculado, quais variáveis e ponteiros estão sendo manipulados:**

****

L1: copia o argumento i para %eax

L2: copia o segundo argumento p para %edx

L3: copia i para %ecx

L4: computa 16\*i e salva o resultado em %ecx

L5: computa 8\*i e salva o resultado em %eax

L6: computa [(%ecx + %edx + 16) + %eax], ou seja, [(16i + p + 16) + 8i] = (24i + p + 16)

L7: copia (%edx + 52) , ou seja, (p + 52), que é p->right, em %ecx

L8: soma o conteúdo de %edx com %ecx

L9: copia n para q->x[p->idx]

L10: transfere o controle para o endereço de retorno da função

**Questão 3) b) Encontre o valor de, apresentando argumentos sólidos para a dimensão do vetor a:**

A linha 7 do código de montagem nos dá o limite superior do tamanho da nossa struct1 a[D] (p + 52). Considerando que a primeira instrução int left está em (p + 0) e que ela é do tipo inteiro, verificamos que ela varia de (p + 0) a (p + 4), onde começa a struct. A partir daí, ela está entre os limites (p + 4) a (p + 52), totalizando 48 bytes de tamanho. A linha 6 nos indica que

[8i + (16i + p + 16)] = [8i + (16i + p + 4 + 12)] = [8i + (16i + struct1 a[D] + 12)]

= [8i + (q->idx)].

Além disso, sabemos que em C não há nenhum tipo primitivo de tamanho 12 bytes. Sendo assim, sabemos que anteriormente à idx havia um array. A linha 8 nos ajuda a descobrir o tamanho de idx, uma vez que ela consiste de um addl, o qual, por padrão, só pode operar com inteiros (4 bytes). Logo, sizeof(idx) = 4 e, analogamente, sizeof(x[]) = 12. Assim, D = 48 / (sizeof(x[]) + sizeof(idx)) = 48 / 16 = 3

**Questão 3) c) Determine sizeof(idx) e sizeof(a) com justificativas claras:**

Como explicado na alternativa B:

A linha 8 nos ajuda a descobrir o tamanho de idx, uma vez que ela consiste de um addl, o qual, por padrão, só pode operar com inteiros (4 bytes). Logo, sizeof(idx) = 4.

A linha 7 do código de montagem nos dá o limite superior do tamanho da nossa struct1 a[D] (p + 52). Considerando que a primeira instrução int left está em (p + 0) e que ela é do tipo inteiro, verificamos que ela varia de (p + 0) a (p + 4), onde começa a struct. A partir daí, ela está entre os limites (p + 4) a (p + 52), totalizando 48 bytes de tamanho.

**Questão 3) d) Determine o tipo do vetor x e sua dimensão com justificativas claras:**

Na linha 9, percebemos que, por se tratar da instrução movw, precisamos apenas de 2 bytes. Por isso, o vetor x, necessariamente deve conter dados de 2 bytes, ou seja, é do tipo short int. Além disso, pela análise feita no item A, sabemos que a struct1 a[3] possui 48 bytes, os quais representam o somatório de x[i] com idx.

3 \* [sizeof(idx) + i \* sizeof(x[])] = 48 🡪 3 \* [4 + 2i] = 48 🡪 i = 6.

Dessa forma, descobrimos que a dimensão do vetor é 6.

**Questão 3) e) Identifique as possíveis declarações da estrutura struct1, 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 struct1:**

Na struct1 a[D], como analisado acima, percebemos que seus campos idx e x[] se tratam, respectivamente, de dados do tipo int e short int, assim como o valor de D é 3, a dimensão de x[] é 6.

**Questão 4) a) Quais registradores o GCC escolheu inicialmente para as variáveis x e z? Justifique:**

%esi para a variável x e %ebx para z. Podemos ver claramente nas linhas 10 e 11 do código de montagem que são atribuídos 7 e 3, que são justamente os valores passados no trecho de código em C, aos registradores em questão.

**Questão 4) b) Quais os registradores escolhidos pelo programador no comando ASM para armazenar as variáveis x, y e z?**

%esi, %edx e %ebx, respectivamente. Não há mudança nos registradores que já possuem valores e, portanto, os registradores nos quais estavam salvos x e z (%esi e %ebx) continuam com esses valores. Portanto, y está armazenado em %edx, que é o único registrador dentro de #APP que não recebeu valores anteriormente.

**Questão 4) c) Dê argumentos sólidos para configurar a lista de saída do comando ASM:**

A lista de saída é composta pelos elementos que est ão na função printf (x e y, respectivamente).

**Questão 4) d) Dê argumentos sólidos para configurar a lista de entrada do comando ASM:**

A lista de entrada é composta pelos elementos que além de declarados, são também inicializados no programa (x e z, respectivamente).

**Questão 4) e) Escreva o comando ASM apagado, usando tanto notação posicional como a nominal:**

Notação posicional:

asm (“leal 5(%1), %0 \n\t”

“addl %3, %1 \n\t”

:“=d”(y), “=S”(x)

:“S”(x), “b”(z)

:

);

Notação nominal:

asm (“leal 5(%%[x]), %%[y] \n\t”

“addl %%[z], %%[x] \n\t”

:[y] “=d”(y), [x] “=S”(x)

:[x] “S”(x), [z] “b”(z)

:

);