Análise do Código - Laboratórios de Informática II

André Gonçalves A75625 | Ricardo Pereira A74185 | Renato Silva A75310

Maio 2015

1 Introdução

Como projecto da disciplina Laboratórios Informática II, tivemos de programar em *C* um conjunto de funções que permitem criação de um jogo de *Batalha Naval*. O jogo, *Batalha Naval*, é conhecido a nível mundial, no entanto a versão que vamos implementar é para um jogador apenas. Consiste em encontrar os locais dos *barcos*, num tabuleiro, de forma a que satisfaça as regras do jogo. A frota num tabuleiro, vai variando com o tamanho do tabuleiro, ou seja, quanto maior o tabuleiro, mais barcos exitem, e maior é o barco maior da frota.

O jogo é feito num tabuleiro, em que o jogador sabe o número de segmentos de bracos existentes em cada coluna e em cada linha do tabuleiro. Apartir destas informações o jogador tem que completar o tabuleiro de forma a que os barcos estam completos, e o número de segmentos em cada linha e coluna seja certo.

Para que fosse possível a implementação do jogo em C, criamos um intrepretador que vai possuir vários comandos, de forma a poder gerir o jogo e jogar.

Na última etapa do projecto foi nos porpôsto, a implementação de um comando R, que quando aplicado ao jogo, resolvia o tabuleiro dado. Caso o tabuleiro era inválido, deixava no mesmo estado, não alterando nada.

Para além disso, como tarefa de análise de código, foi nos propôsto, a análise de um código em assembley, de um ficheiro ("comp.c"), da função *contar_segs*. Com conhecimento que fomos adequirindo na disciplina Sistemas de Computação, e prática em análisar código assembley, análisamos o código detalhadamente, criando uma tabela de alocação dos registos e a documentação do repectivo código, relacionando com o código da função *contar_segs* em C.

Para nosso auxílio na análise do código, ultilizaremos o GDB (GNU Debugger), para desmontar o código Assembly, e para podemos realizar certos breakpoints no programa, para conseguir detectar e perceber o que cada registo possuir, porquê e para quê.

Na análise, iremos explicar cada intrução uma a uma, de forma a que o código Assembly, seja depois claramente perceptível.

2 Implementação do Comando

2.1 Comando *R*

A função *resolver* começa por aplicar as 4 estratégias defenidas no nosso programa ao tabuleiro, e aplica estas 4 estratçegias enquanto que houver mudanças no tabuleiro e/ou o tabuleiro estiver resolvido.

Após isto chama se a funçao *aleatorio* que cria um array com o número de combinações de jogadas por linhas e outro com as linhas correspondentes, ordena este primeiro array, para que o programa possa saber quais as linhas que tem o menor número de combinações e assim começar por tentar resolver essas linhas, após isso, o programa ira chamar a função *aleoaux* que irá aplicar a combinação correpondente (variável play) e devolve o tabuleiro com esta combinação utilizada. Após isto é aplicado em loop as estratégias enquanto que estas modifiquem o tabuleiro, e depois, se o tabuleiro ficar resolvido, devolve o tabuleiro. Caso o tabuleiro ficar inválido, retorna o tabuleiro antes das alterações. Caso o tabuleiro ainda não estiver resolvido, mas também não for inválido aplica-se por recursividade a função aleatório para que esta possa tentar resolver outra linha do tabuleiro e assim resolver o tabuleiro.

2.2 Testes

Para averiguar a qualidade e funcionamento do código elaborado, recorremos à experimentação por testes. Após a conclusão do um comando é sempre efetuada pelo grupo vários testes em várias situações diferentes para verificar a funcionalidade do código e para que se possam corriguir eventuais falhas em casos anormais. Como forma de auílio e de perceção da validade do nosso código, foi-nos dados inúmeros testes para verificar o funcionamento dos comandos. Por fim, corremos o código na máquina vitual, fornecida pelos docentes da cadeira.

3 Código Assembly

O código Assembly, é obtido através do gcc da máquina virtual fornecida pela disciplina Sistemas de Computação. A sua obtenção foi através da compilação do ficheiro *comp.c*, com a flag -O2, a otimizar o código. Posteriormente, através do gdb, desmontamos o ficheiro excutável, obtendo este código Assembly:

```
0x08048400 <contar_segs+0>:     push %ebp
0x08048401 <contar_segs+1>:     mov %esp,%ebp
0x08048403 <contar_segs+3>:     push %edi
0x08048404 <contar_segs+4>:     push %esi
0x08048405 <contar_segs+5>:     push %ebx
0x08048406 <contar_segs+6>:     sub $0xc,%esp
0x08048409 <contar_segs+9>:     xor %edi,%edi
0x0804840b <contar_segs+11>:    xor %ecx,%ecx
0x0804840d <contar_segs+13>:    xor %edx,%edx
0x0804840f <contar_segs+15>:    cmpb $0x0,0x2720(%ebp)
0x08048416 <contar_segs+22>:    mov 0x2724(%ebp),%eax
0x0804841c <contar_segs+28>:    movl $0x0,-0x10(%ebp)
0x08048423 <contar_segs+35>:    movl $0x0,-0x14(%ebp)
0x0804842a <contar_segs+42>:    je 0x8048486<contar_segs+134>
0x0804842c <contar_segs+44>:    lea -0x1(%eax),%ecx
0x0804842f <contar_segs+47>:    movl $0x1,-0x10(%ebp)
0x08048436 <contar_segs+54>:    mov 0x2718(%ebp),%eax
0x0804843c <contar_segs+60>:    test %eax,%eax
0x0804843e <contar_segs+62>:    jle 0x804847b<contar_segs+123>
0x08048440 <contar_segs+64>:    mov %eax,%esi
0x08048442 <contar_segs+66>:    lea 0x0(,%edx,4),%eax
0x08048449 <contar_segs+73>:    add %edx,%eax
0x0804844b <contar_segs+75>:    lea (%ecx,%ecx,4),%ebx
0x0804844e <contar_segs+78>:    mov %eax,-0x18(%ebp)
0x08048451 <contar_segs+81>:    lea 0x0(%esi),%esi
0x08048454 <contar_segs+84>:    lea (%ebx,%ebx,4),%eax
0x08048457 <contar_segs+87>:    lea 0x8(%ebp,%eax,4),%eax
0x0804845b <contar_segs+91>:    sub $0xc,%esp
0x0804845e <contar_segs+94>:    movsbl (%edi,%eax,1),%eax
0x08048462 <contar_segs+98>:    push %eax
0x08048463 <contar_segs+99>:    call 0x80483e4<e_seg>
0x08048468 <contar_segs+104>:   add $0x10,%esp
0x0804846b <contar_segs+107>:   test %al,%al
0x0804846d <contar_segs+109>:   je 0x8048472<contar_segs+114>
0x0804846f <contar_segs+111>:   incl -0x14(%ebp)
0x08048472 <contar_segs+114>:   add -0x10(%ebp),%edi
0x08048475 <contar_segs+117>:   add -0x18(%ebp),%ebx
0x08048478 <contar_segs+120>:   dec %esi
0x08048479 <contar_segs+121>:   jne 0x8048454<contar_segs+84>
0x0804847b <contar_segs+123>:   mov -0x14(%ebp),%eax
0x0804847e <contar_segs+126>:   lea -0xc(%ebp),%esp
0x08048481 <contar_segs+129>:   pop %ebx
0x08048482 <contar_segs+130>:   pop %esi
0x08048483 <contar_segs+131>:   pop %edi
0x08048484 <contar_segs+132>:   leave
0x08048485 <contar_segs+133>:   ret
0x08048486 <contar_segs+134>:   lea -0x1(%eax),%edi
0x08048489 <contar_segs+137>:   mov $0x1,%edx
0x0804848e <contar_segs+142>:   mov 0x271c(%ebp),%eax
0x08048494 <contar_segs+148>:   jmp 0x804843c<contar_segs+60>
```

4 Tabela de alocação dos registos

Registos, Memória	(Variáveis,etc.)					
%eax	num	tam	indice(variação da linha)	endereço da linha	t.tab[y][x]	e_segs
%edi	x					
%ecx	y					
%edx	dy					
%ebx	valor com base do indice	y(com base no indice)				
%esi	tam					
-0x10(%ebp)	dx					
-0x14(%ebp)	count					
0x2720(%ebp)	lin					
0x2724(%ebp)	num					
0x2718(%ebp)	t.lins					
0x271c(%ebp)	t.cols					
———	i é retirado pela otimização da flag -O2					

Tabela 1: Alocação dos Registos no decorrer do programa

5 Variável *tab*

A variável tab localiza-se no endereço 0xbfff9930. A variável tab ocupa 10000 bytes de espaço, porque a variável tab uma matriz de 100 por 100 de char's. Como cada posição da matriz ocupa 1 byte, por ser um char, a matriz completa ocupa $100 * 100 = 10000$ bytes, aproximadamente 9.8 Kbytes. É de notar que mesmo que se defina a matriz apenas 10 por 10, a matriz é a de 100 por 100, mas apenas é alterado o estado em 10 por 10, isto é, usa-se uma porção mais pequna da matriz, mas no entanto ocupa os mesmo espaço, porque a matriz usada é sempre a 100 por 100.

A variável tab, está estruturada na memória da seguinte forma:

Memória

0xbfff9930 ->	char	10000 bytes (char tab[100] [100])
	char	
	char	
	char	
	. . .	
0xbfffc040 ->	int	4 bytes (int lins)
0xbfffc044 ->	int	4 bytes (int cols)

Tabela 2: Esquema da organização da Memória

6 Indexação da Matriz

A indexação da Matriz, utlilizada para representar o tabuleiro do jogo, é feita através de uma série de cálculos iniciais, para obter um valor que determina a linha, e posteriormente, obtem-se facilmente a posição na matriz que se prentende. A indexação é feita da seguinte forma:

Inicialmente, realiza-se uma série de intruções, que servem para o cálculo do endereço da variável t.tab[y][x]. A instrução *lea* calcula o valor de um índice que depende do valor de dy. Caso dy seja 1, o valor do indice será 4, caso seja 0, o valor do indice será 0. Com a próxima instrução *add*, conclui-se o cálculo do índice, que caso dy seja 1, o indice passa a ter o valor 5, caso seja 0, continua com o valor 0. O índice é 0 ou 5, dependendo do valor de dy. Este indice, será ultilizado numa instrução mais abaixo, onde se soma a y o indice, fazendo y (linha) variar. Com o decorrer das instruções e apenas perto do final do ciclo for, é que é percetivel o siginficado da variação da linha, com o indice. Se tomarmos o indice igual a 0, na instrução *<contar_segs+117>: add -0x18(%ebp),%ebx*, o valor do indice irá ser somado a %ebp que corresponde a y, e como o valor somado é 0, o valor de y mantém-se. Logo o ciclo for mantém sempre o mesmo valor da linha. Caso o indice tome o valor de 5, por cada interação do ciclo for, é somado mais 5. Numa análise rápida, não varia sentido. Mas se verificarmos, para passar para a linha seguinte é necessário somar 5. Por exemplo, linha 1 correponde o valor 5, linha 2 corresponde o valor 10, assim sucessivamente. Por isso, o cálculo de linha*5.

```
0x08048442 <contar_segs+66>:    lea 0x0(,%edx,4),%eax        //Cálculo para o valor do indice
0x08048449 <contar_segs+73>:    add %edx,%eax                //Cálculo final do indíce
```

Posteriormente, a próxima instrução começa a realizar cálculos, a patrir do valor de y, valor que representa o nº da linha . Este começa a cálcular a partir da linha dada. O cálculo é o seguinte :

$linha + linha * 4 = linha * 5$

De seguida realiza-se um *mov* para guardar o valor do índice em memória e por fim realiza-se um intrução que codifica equivalentemente a uma instrução nop, ou seja, não realiza nada.

```
0x0804844b <contar_segs+75>:    lea (%ecx,%ecx,4),%ebx       //Cálculo inicial do endereço da linha
0x0804844e <contar_segs+78>:    mov %eax,-0x18(%ebp)         //Guarda a variação da linha
0x08048451 <contar_segs+81>:    lea 0x0(%esi),%esi           //Equivale a um nop
```

Nestas duas instruções realiza-se os dois ultimos cálculos. O segundo *lea*, cálcula a linha na matriz. Os cálculos são:

$(linha * 5) + ((linha * 5) * 4) = (linha * 5) * 5 = (linha * 25)$

$\%ebp + 8 + (linha * 25) * 4 = \%ebp + 8 + (linha * 100)$

Como podemos ver, a série de cálculos levou-nos a obter linha*100, porque como a mariz é armazenada na memória, num conjunto de array's seguidos, para poder andar linha em linha, temos que andar 100 em 100 células de memória. Por exemplo, caso a linha seja 3 temos 300, obtido da fórmula linha*100. Esse valor somado ao %ebp mais o valor de 8, obtemos o endereço da linha 3.

No entanto, se quiseremos podemos reformular as fórmulas anteriores, tendo em conta o indice para a variaçao da linha.

$(\%ebx) + ((\%ebx) * 4) = (\%ebx) * 5$

$\%ebp + 8 + (\%ebx * 5) * 4 = \%ebp + 8 + (\%ebx * 20)$

Como por exemplo, a linha 2 corresponde a 10, temos 10*20 = 200. Assim obtémos os mesmos valores.

```
0x08048454 <contar_segs+84>:    lea (%ebx,%ebx,4),%eax       //Cálculo Aux. para endereço da linha
0x08048457 <contar_segs+87>:    lea 0x8(%ebp,%eax,4),%eax    //Cálculo final do endereço da linha
```

Realiza-se uma subtração ao valor de %esp, para aumentar mais 12 bytes. Depois realiza-se uma instrução *movsbl*, que como já tem o endereço da linha calculado, apenas soma-se o valor de x (%edi) e obtem a posição e o char pretendido. Por exemplo, se pretedemos a posição linha 3, colunha 10, com o endereço obtido anteriormente, apenas soma-se o valor de 10, e obtém-se a posição.

```
0x0804845b  <contar_segs+91>:    sub $0xc,%esp                  //Sobe o stack pointer mais 12 bytes
0x0804845e  <contar_segs+94>:    movsbl (%edi,%eax,1),%eax      //Buscar t.tab[y][x] à memória
```

7 Análise do código Assembly

Para analisar o código em assembly, vamos explicar instrução a instrução, relacionando com o código em C. De forma a que seja mais lúcido, e mais clara a explicação, diviriemos o código em várias partes.

Como podemos ver, para inicar uma função é necessário salvaguardar resgistos, para não se perder informação, isto é, valores de variáveis, etc. O *push %ebp*, é para salvaguardar o registo, do base pointer da stack frame da função chamadora. Tendo salvaguardado o registo, iguala-se o valor de %ebp com o %esp para poder contruir a stack frame da actual função, ignorando a stack frame da função chamadora, que fica salvaguardada. Como já referimos, é necessário salvaguradar registos, e por isso é visível a execução de três *push* seguidos.

```
0x08048400 <contar_segs+0>:    push %ebp          //Salvaguarda o resgisto
0x08048401 <contar_segs+1>:    mov %esp,%ebp      //Iguala o %ebp com o %esp
0x08048403 <contar_segs+3>:    push %edi          //Salvaguarda o resgisto
0x08048404 <contar_segs+4>:    push %esi          //Salvaguarda o resgisto
0x08048405 <contar_segs+5>:    push %ebx          //Salvaguarda o resgisto
```

Depois de ter salvaguardado os registos, o valor do resgisto %esp é alterado, ou seja, é subtraído, de foram a criar a stack frame da função. Posteriormente é efectuado três instruções indênticas. A instrução *xor*, quando é aplicada com os mesmo resgistos, dá o valor de 0. Esta é uma forma mais rentável de inicializar uma variável a 0. Este facto, provém da otimização do código através da flag -O2. Como é visível, as três isntruções correspondem a inicialização de três variáveis a 0, ou seja, x=0,y=0 e dy=0.

```
0x08048406 <contar_segs+6>:    sub $0xc,%esp      //Aumenta 12 bytes na stack
0x08048409 <contar_segs+9>:    xor %edi,%edi      //Iguala a 0(x=0)
0x0804840b <contar_segs+11>:   xor %ecx,%ecx      //Iguala a 0(y=0)
0x0804840d <contar_segs+13>:   xor %edx,%edx      //Iguala a 0(dy=0)
```

A instrução *cmpb*, vai comparar o valor em memória (0x2720(%ebp)) com a constante 0. Esta comparação esta relacionada com o if(lin), ou seja, compara o valor de lin, que só pode ser 0 ou 1, com a constante 0. De seguida, são executados três instruções *mov*, duas para atribuir o valor 0 às variáveis dx e count, e uma para buscar o valor da variável num à memória.

```
0x0804840f <contar_segs+15>:   cmpb $0x0,0x2720(%ebp)   //if(lin)
0x08048416 <contar_segs+22>:   mov 0x2724(%ebp),%eax    //Buscar o valor de num
0x0804841c <contar_segs+28>:   movl $0x0,-0x10(%ebp)    //dx=0
0x08048423 <contar_segs+35>:   movl $0x0,-0x14(%ebp)    //count=0
```

Com a instrução de comparação, exectuada anteriormente, segue-se uma instrução de salto condicional, ou seja, consuante os valores obtidos na comparação, a instrução de salto verifica se salta para o endereço ou não. A condição para o salto, é ser igual a 0, ou seja, o valor de lin tem que ser igual a 0. Caso seja igual, o %eip passa a ter o endereço 0x8048486, continuando a executar o código desse endereço. Caso seja diferente, o salto é ignorado, e segue-se a ordem normal de instruções. No caso de não efectuar o salto, é executado uma instrução *lea*, que passa o valor de -0x1(%eax) para %ecx. Como %eax corresponde à variável num e %ecx à variável y, a instrução corresponde a y=num-1. Por fim neste bloco de código, é executado duas instruções *mov*, a primeira para atribuir o valor da constante 1 à variável dx, e segunda para passar o valor de t.lins, que está em memória (0x2718(%ebp)) para a variável tam. É de referir que caso seja igual a 0, passa do ramo do if para o else, sendo o código acado de analisar o do ramo do if.

```
0x0804842a <contar_segs+42>:   je 0x8048486<contar_segs+134>   //Salto condicional(=0)
0x0804842c <contar_segs+44>:   lea -0x1(%eax),%ecx             //y=num-1
0x0804842f <contar_segs+47>:   movl $0x1,-0x10(%ebp)           //dx=1
0x08048436 <contar_segs+54>:   mov 0x2718(%ebp),%eax           //tam=t.lins
```

No caso de ser igual, o salto é dirigido para o bloco de código abaixo. Neste caso, é visível equivalências com o código anterior. Este parte do código, diz respeito ao ramo do else. A instrução *lea*, que passa o valor de -0x1(%eax) para %edx. Como %eax corresponde à variável num e %edx à variável x, a instrução corresponde a x=num-1. Depois é executado duas instruções *mov*, a primeira para atribuir o valor da constante 1 à variável dy (%edx), e segunda para passar o valor de t.cols, que está em memória (0x271c(%ebp)) para a variável tam. Por fim, temos uma instrução de salto incondicional *jmp*, que salta para o endereço depois da estrutura if e else.

```
0x08048486 <contar_segs+134>:    lea -0x1(%eax),%edi                  //x=num-1
0x08048489 <contar_segs+137>:    mov $0x1,%edx                        //dy=1
0x0804848e <contar_segs+142>:    mov 0x271c(%ebp),%eax                //tam=t.cols
0x08048494 <contar_segs+148>:    jmp 0x804843c<contar_segs+60>        //Salto icondicional
```

Depois da estrutura if e else, entramos um ciclo for. Para tal, com a otimização da flag -O2, a variável i, foi retirada, e por tanto a condição que rege o ciclo, é uma pouco diferente a que está presente no código C. A intrução *test*, compara dois resgistos iguais (%eax, que possui a variável tam), para verificar se o valor em %eax é positivo, negativo ou igual a 0. Como era de esperar, ao fim de uma comparação, viria uma intrução de salto condicional. Este salto condicional ,*jle*, verifica se o valor é menor ou igual a 0. Desta forma, caso o valor seja menor ou igual a 0, salta para o endereço 0x804847b, ou seja, sai do ciclo, e continua o programa. Caso seja maior, prosegue as instruções normalmente. Como referi anteriormente, a condição que rege o ciclo, é o valor de tam, se é menor ou igual a 0, e sai do ciclo, ou se é maior, e continuar no ciclo. Como é claro, com a ausência da variável i, existe outra forma que controlar, o número de ciclos. Esta parte será explicada mais a baixo. Continuando a análise, a próxima instrução é um *mov*, que passa o valor de tam para o registo %esi. Após esta instrução, vamos nos deparar com uma série de intruções, que servem para o cálculo do endereço da variável t.tab[y][x]. A instrução *lea* calcula o valor de um índice que depende do valor de dy. Caso dy seja 1, ou 0. Com a próxima instrução *add*, conclui-se o cálculo do índice. Resumidamente, com as das intruções anteriores, podemos obter o índice para o cálculo da linha. O índice é 0 ou 5, dependendo do valor de dy. Posteriormente, a próxima instrução começa a calcular as componentes para o endereço de t.tab[y][x]. De seguida realiza-se um *mov* para guardar o valor do índice em memória e por fim realiza-se um intrução que codifica equivalentemente a uma instrução nop, ou seja, não realiza nada.

```
0x0804843c <contar_segs+60>:     test %eax,%eax                       //tam (verifica se é +, - ou 0)
0x0804843e <contar_segs+62>:     jle 0x804847b<contar_segs+123>       //Salto condicional (<=0)
0x08048440 <contar_segs+64>:     mov %eax,%esi                        //Passa o valor de tam para %esi
0x08048442 <contar_segs+66>:     lea 0x0(,%edx,4),%eax                //Cálculo para o valor do indice
0x08048449 <contar_segs+73>:     add %edx,%eax                        //Cálculo final do indíce
0x0804844b <contar_segs+75>:     lea (%ecx,%ecx,4),%ebx               //Cálculo inicial do endereço da linha
0x0804844e <contar_segs+78>:     mov %eax,-0x18(%ebp)                 //Guarda a variação da linha
0x08048451 <contar_segs+81>:     lea 0x0(%esi),%esi                   //Equivale a um nop
```

Neste bloco de código, é a continuação do cálculo do endereço de t.tab[y][x]. Após de calcular o valor para %ebx, este sofre um novo cálculo, através de uma instrução *lea*. Por fim, o resultado obtido na última intrução, é aplicado nesta instrução *lea*, somando ao %ebp o valor obtido (após de ser multiplicado por 4 e somado 8). Este valor significa o número da linha que pretendemos. Como a matriz é armazenada em um conjunto de array's seguidos, o cálculo da linha, passa por um processo de cálculos diferentes, sendo que a maioria das intruções *lea*, serviram como uma forma de cálculos auxiliares. De seguida, realiza-se uma subtração ao valor de %esp, para aumentar mais 12 bytes. Depois realiza-se uma instrução *movsbl*, que como já tem o valor da linha calculado, apenas soma o valor de x (%edi) e obtem a posição e o char pretendido. Realiza-se um *push* para salvaguardar o valor de t.tab[y][x]. Por fim, com a instrução *call*, chama-se a função e_seg.

```
0x08048454 <contar_segs+84>:    lea (%ebx,%ebx,4),%eax          //Cálculo Aux. para endereço da linha
0x08048457 <contar_segs+87>:    lea 0x8(%ebp,%eax,4),%eax       //Cálculo final do endereço da linha
0x0804845b <contar_segs+91>:    sub $0xc,%esp                   //Sobe o stack pointer mais 12 bytes
0x0804845e <contar_segs+94>:    movsbl (%edi,%eax,1),%eax       //Buscar t.tab[y][x] à memória
0x08048462 <contar_segs+98>:    push %eax                       //Salvaguarda resgisto (t.tab[y][x])
0x08048463 <contar_segs+99>:    call 0x80483e4<e_seg>           //Chamada da funçao e_seg
```

Neste bloco de código, é executada a instrução *add*, para baixar o stack pointer (%esp). Depois, realiza-se um teste, com a instrução *test*. Como o valor que uma função retorna fica no registo %eax, o valor que é testado, é o valor retornado pela função e_seg. Como é de se esperar, a instrução seguinte, é uma instrução de salto. Nesta instrução, verifica se o valor é igual a 0 ou não, sendo que se for igual a 0, salta para o endereço 0x8048472, salta para a instrução *add*, ignorando a instrução *incl*. Comparado com o código em C, esta instrução, corresponde a condição if(e_seg(t.tab[y][x])), que caso a função retornar o valor 0, ignora o count++. Como ja referi indirectamente, a próxima instrução, incrementa a variável count, isto é, corresponde ao count++. De seguida, as duas instruções de soma, correspondem aos códigos, x+=dx e y+=dy. É de facíl reconhecimento os operadores das somas, sendo apenas preciso verificar os registos, e locais de memória em causa. Após estas instruções, é executada uma instrução, que não é visível no código em C. A instrução *dec*, decrementa a variável tam, em cada volta do ciclo for. Desta forma, em vez de trabalhar com a variável i, sendo preciso de a inicializar a 0 e de a incrementar em cada volta do ciclo, e ocupar registos ou memória, ao decrementar tam, poupamos estes procedimentos. Como a variável é inicializada a 0, a icrementação do i até ao valore de tam, e a decrementação de tam até ao valor de 0, é o mesmo nº de ciclos. Esta é uma forma de rentabilizar o código, obtido graças a otimização da flag -O2. Por fim, o salto condicional dirige-nos para o iníco do ciclo for.

```
0x08048468 <contar_segs+104>:   add $0x10,%esp                  //Desce o stack pointer 16 bytes
0x0804846b <contar_segs+107>:   test %al,%al                    //Resultado de e_seg(0/1)
0x0804846d <contar_segs+109>:   je 0x8048472<contar_segs+114>   //Salto condicional (=0)
0x0804846f <contar_segs+111>:   incl -0x14(%ebp)                //count++
0x08048472 <contar_segs+114>:   add -0x10(%ebp),%edi            //x+=dx
0x08048475 <contar_segs+117>:   add -0x18(%ebp),%ebx            //y+=dy
0x08048478 <contar_segs+120>:   dec %esi                        //tam- -
0x08048479 <contar_segs+121>:   jne 0x8048454<contar_segs+84>   //Salto condicional (!=0)
```

Este bloco código corresponde a parte final da função contar_segs. Em primeiro lugar, como count é o valor a retornar, tens que passar o seu para para o registo %eax, e é o que acontece na primeira instrução. De seguida, coloca o endereço a 12 bytes de distância do %ebp em %esp, que corresponde ao espaço dos valores para os três *pop* seguintes. E como inicialmente era necessário salvaguardar os registos, agora no fim da função é necessário, recuparar os valores. Para tal, é ultilizado a intrução *pop*, para recuparar os valores. Por fim, aparecem as duas instruções finais, *leave* e *ret*. O objectivo da instrução leave consiste na preparação da stack de forma a que o %esp aponte para o elemento da stack onde a anterior instrução call guardou o endereço de retorno. Desta forma, a instrução leave pode ser usada para preparar a stack para a operação de retorno, e a ret é que produz o return, colocando o %eip apontar para a próxima instrução da outra função.

```
0x0804847b  <contar_segs+123>:    mov -0x14(%ebp),%eax      //eax=count
0x0804847e  <contar_segs+126>:    lea -0xc(%ebp),%esp       //Coloca o %esp a 12 bytes de distância
0x08048481  <contar_segs+129>:    pop %ebx                  //Recupera registo
0x08048482  <contar_segs+130>:    pop %esi                  //Recupera registo
0x08048483  <contar_segs+131>:    pop %edi                  //Recupera registo
0x08048484  <contar_segs+132>:    leave
0x08048485  <contar_segs+133>:    ret
```

8 Conclusão

Como forma de introdução ao mundo da programação, a Unidade Curricular Laboratórios de Informática II permitiu-nos perceber as aplicações e ganhar alguma experiência com a linguagem imperativa *C*.

Após muito tempo despendido na realização do projecto, este levou-nos a concluir que quando programando, o trabalho diário, a tentiva de correção dos nossos erros, e o trabalho de equipa, forma a essência para a realização do nosso projecto.

A realização do comando R, passou pela junção de todo o trabalho feito até a altura. Muitos planos para a sua criação, muitas tentativas, e horas dependidas a verificar a sua validade e funcionalidade.

A análise de código, foi um forma de introdução ao mundo do código assembly. Devido a sua complexidade e a sua antiguidade, esta linguagem, fundamental para um programador, deu-nos uma melhor percepção, no que toca ao funcionamento do computador, processo de passagem do código C para Assembly, etc. A Unidade Curricular Laboratórios de Informática II permitiu-nos perceber as aplicações e ganhar alguma experiência com a linguagem imperativa *Assembly*.

Com isto, é de realçar que esta análise, tornou-se uma ajuda preciosa para a aprendizagem e solucionar certas dúvidas sobre o funcionamento do código Assembly. Muitas horas despendidas na tentiva de relacionar o código C com o Assembly, resgistos com variáveis, etc.

Em suma, foi um trabalho que nos incentivou à linguagem Assembly, no seu entendimento e funcionamento, que se tornou uma peça chave para a compreenção do código C em funionamento no computador, e que será muito útil para o nosso desempenho na Unidade Curricular Sistemas de Computação.

Em suma, foi um projecto que nos acomodou à programação, e com os problemas que viriam da sua experimentação, a calma nos momentos mais difíceis são aspectos chave os solucionar.