

TPC7 e Guião Laboratorial

Resolução dos exercícios

1 Ciclo *for*

Uma forma de se analisar o código de um ficheiro executável (e para o qual não se tenha acesso ao ficheiro fonte em HLL que lhe deu origem) consiste em (i) desmontar o ficheiro binário para a versão *assembly* e depois (ii) tentar inverter o processo de compilação e produzir código C que pareça “natural” a um programador de C. Por exemplo, não queremos código com instruções *goto*, uma vez que estas são raramente usadas em C; e muito provavelmente não usaríamos também aqui a instrução *do-while*.

Este exercício obriga-nos a pensar no processo inverso da compilação num dado enquadramento: no modo como os ciclos *for* são traduzidos.

a) Rotina...

b) Ver alínea seguinte...

c) A partir do ficheiro executável que foi disponibilizado, `m_contaN`, é possível desmontá-lo para *assembly*, localizar a parte de código desmontado correspondente à função `contaN` e ainda distinguir as partes de inicialização e término da função, e do corpo da função (a parte pertinente neste exercício).

O código desmontado da função deverá ter um aspeto semelhante ao seguinte; este código inclui já uma anotação introduzida manualmente, bem como as etiquetas (*labels*):

contaN:

```

0x080483f4 <contaN+0>: push    %ebp                ; inicializa função: salvag/ frame pointer anterior
0x080483f5 <contaN+1>: mov     %esp, %ebp                ; cria novo frame pointer (FP)
0x080483f7 <contaN+3>: push    %esi                ; salvaguarda registo %esi
0x080483f8 <contaN+4>: push    %ebx                ; salvaguarda registo %ebx
0x080483f9 <contaN+5>: mov     0x8(%ebp), %esi      ; %esi = apontador para início da string s (1º arg)
0x080483fc <contaN+8>: mov     0xc(%ebp), %ecx      ; inicializa com c a var controlo de ciclo (em %ecx)
0x080483ff <contaN+11>: mov     (%ecx, %esi, 1), %dl ; %dl = carácter na posição c da string, s[c]
0x08048402 <contaN+14>: xor     %ebx, %ebx          ; inicializa a var local, o somatório result: %ebx=0
0x08048404 <contaN+16>: test    %dl, %dl            ; s[c] = char-fim-da-string? (null em ASCII)
0x08048406 <contaN+18>: je      0x8048420<contaN+34> ; je ou jz: salta p/ fim da função se s[c] for null
.L7:
0x08048408 <contaN+20>: lea     -0x30(%edx), %eax     ; %eax=%edx (contém %dl) -48 (ASCII char "0")
0x0804840b <contaN+23>: cmp     $0x9, %al           ; compara %al com 9
0x0804840d <contaN+25>: ja      0x8048416<contaN+34> ; salta se valor de %al acima de 9, i.e., não é dígito
0x0804840f <contaN+27>: movsb   %dl, %eax           ; %eax = char s[c] estendido c/ sinal para 32bits
0x08048412 <contaN+30>: lea     -0x30(%eax, %ebx), %ebx ; result=result+ dígito em s[c]
.L4:
0x08048416 <contaN+34>: inc     %ecx                ; increm var controlo de ciclo (posição c na string s)
0x08048417 <contaN+35>: mov     (%ecx, %esi, 1), %al ; %al = carácter na posição c da string, s[c]
0x0804841a <contaN+38>: test    %al, %al            ; s[c] = char-fim-da-string? (null em ASCII)
0x0804841c <contaN+40>: mov     %al, %dl            ; %dl = s[c] (não altera bits de condição)
0x0804841e <contaN+42>: jne     0x8048408<contaN+20> ; salta p/ início ciclo (L7) se s[c] não é fim-da-string
.L9:
0x08048420 <contaN+44>: mov     %ebx, %eax          ; %eax=result (Σ a devolver pela função)
0x08048422 <contaN+46>: popl    %ebx                ; término da função: recupera registo %ebx
0x08048423 <contaN+47>: popl    %esi                ; recupera registo %esi
0x08048424 <contaN+48>: leave   ; recupera SP e FP-anterior
0x08048425 <contaN+49>: ret     ; recupera IP e regressa

```

De acordo com as anotações já introduzidas no código, as identificações pedidas na alínea **c)** são agora fáceis de resolver (não esquecer que se pedia para identificar no código desmontado em *assembly* e não no código C):

- sendo o `result` o somatório de dígitos a devolver, é só procurar a inicialização a zero de um registo e, se este não for o `%eax`, (neste caso é o `%ebx`) então procurar no fim uma instrução que copie o valor em `%ebx` para `%eax`; a variável `i` deveria ser inicializada com `c` e incrementada de uma unidade dentro do ciclo `for`, e isso acontece com o registo `%ecx`;
- sabendo que os 1º e 2º argumentos se encontram na *stack* à distância de, respetivamente, 8 e 12 células de memória do valor apontado pelo *frame pointer* (em `%ebp`), é fácil de ver para que registos foram copiados esses argumentos;
- o ciclo `for` deverá terminar quando o carácter lido for *null* (e também deverá ser garantido que o ciclo não é executado se o carácter inicial para análise da *string* `s` for também *null*); de notar que, tal como referido nos slides ISA_3, os compiladores transformam os ciclos `for` em `do_while` com testes da condição antes de entrar e dentro do ciclo; assim, o código *assembly* deverá conter 2 expressões de teste semelhantes: estão nas linhas `<contaN+16>` e `<contaN+18>` no teste antes de entrar no ciclo, e `<contaN+38>` e `<contaN+42>` dentro do ciclo `do_while`;
- atualização da variável `i`: procurar por uma instrução que faça o incremento de `%ecx`;
- consultando uma tabela ASCII fica-se a saber que os algarismos 0 a 9 são codificados por `0x30` a `0x39`, ou seja, de 48 a 57 (em decimal);
- a atualização do somatório (`result`, em `%ebx`) deveria ser feita pela expressão `result = result + (s[c] estendido c/ sinal para 32 bits) - 48`; como as instruções aritméticas do IA-32 não permitam especificar mais que 2 operandos, o compilador costuma aproveitar a expressividade da instrução `lea`.
Procurem no código onde uma adição como essa `(+=)` foi implementada com `lea...`

Pode-se ainda ver que:

- as expressões de teste do ciclo `for` têm como objetivo verificar se o carácter lido é o fim da cadeia ou não, i.e., se `s[c]=0`;
- há ainda uma outra expressão de teste associada a uma típica estrutura de `if...then` para decidir quando somar os caracteres (quando forem algarismos...); esta expressão tem como objetivo verificar se o carácter lido – o valor de 8 bits que está no registo `%al` – subtraído de 48 é menor ou igual a 9, o que corresponde a verificar se é o código ASCII de um algarismo ou não, i.e., se `s[c] >= '0' && s[c] <= '9'`; este duplo teste é feito de uma só vez ao se usar a instrução `ja` (*jump if above*) em vez de `jg` (*jump if greater*), porque se o carácter estiver na tabela ASCII antes do `'0'`, o resultado da subtração será um número negativo, codificado em complemento para 2, contendo um "1" no bit mais à esquerda, logo estará "acima" (maior-do-que, mas sem sinal) quando comparado com qualquer número positivo.

- d)** Com base no código anotado da alínea anterior (e comentários que se seguiram), e sabendo a estrutura habitual do código gerado por um compilador com um nível médio de otimização, é possível chegar-se ao seguinte código original em C:





```

1 int contaN(char *s, int c)
2 {
3     int i;
4     int result=0;
5     for (i = c; s[i]!='\0' ; i++)
6         if (s[i] >= '0' && s[i] <= '9')
7             result +=(s[i]-'0');
8     return result;
9 }
```

Adenda

Nota 1

O processo de submissão de trabalhos TPC7 foi feita com diversas incorreções, as quais irão ter penalizações na pontuação dos próximos trabalhos, começando no TPC9. Estas incluem:

- entrega do trabalho em mais que um ficheiro: desconta 50%; ex.:  
- entrega de ficheiro sem nº de aluno (o nº deve vir sem a letra "a") ou sem indicação do turno: desconta 50%; ex.: 
- entrega de ficheiro sem cumprir integralmente as regras na atribuição de nome ao ficheiro: desconta 30%; ex.: 
- entrega de trabalho num turno incorreto ou com indicação incorreta do turno: desconta 50%.

O nome de cada ficheiro deverá ser assim constituído:

<nº_aluno>_<PLxx>.pdf

Ex.: 83462_PL03.pdf

Adicionalmente espera-se que cada aluno coloque o seu nome escrito na 1ª página da resolução entregue.

Nota 2

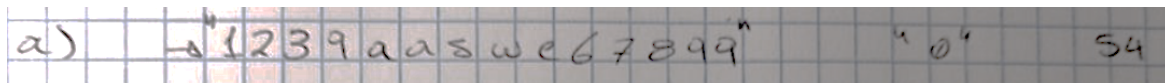
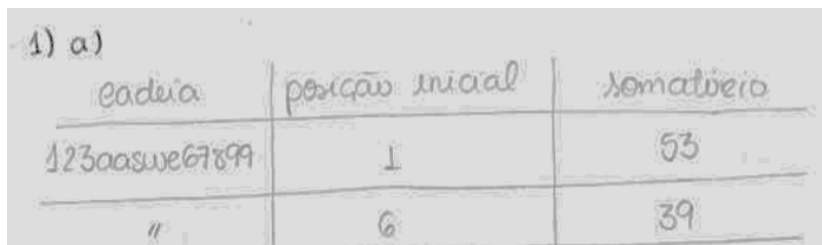
A introdução de comentários no código *assembly* (que se designa por anotação) tem regras consoante o *assembler* utilizado e o processador a analisar.

No caso do *assembler* da GNU e dos processadores x-86 (incluindo o IA-32), os comentários devem começar sempre com o carácter "#", embora o carácter ";" seja também aceite como separador.

Análise de trabalhos entregues

O principal objetivo da 1ª questão — em 1. a) — era avaliar até que ponto é que o aluno tinha ou não acedido de facto ao sistema remoto ou se limitou a copiar a versão *assembly* de colegas. Por isso se pedia que mostrasse toda a informação que aparecia no monitor, incluindo os comandos inseridos pelo aluno.

Estes exemplos em baixo passam a indicação ao docente que o aluno não acedeu ao servidor remoto, uma vez que não coincidem com o que de facto aparece no monitor...

cadeia	posição inicial	somatório
1239aaswe67899	1	53
"	6	39

O exercício proposto para 1.c) era essencialmente a anotação do código desmontado e a resposta a 6 questões concretas.

As anotações continuam a mostrar a falta de estudo de alguns estudantes, pois não fazem a menor ideia do que está a acontecer com o código; pelo menos é o que mostram os seguintes exemplos:

- `0x080483d4 <CONTIN+20>:lea 0xffffffff0(%edx),%eax →`
 GUARDA NO REGISTO %eax O CONTEUDO DE %edx

`lea 0xffffffff0(%edx),%eax - converte char em int.`

(continua) = 10 `lea 0xffffffff0(%edx),%eax → char em int;`
`cmp $0x9,%eax`
`ja 0x80483e2 <CONTIN+34>` } salta as 2 instruções seguintes if (`s[i]>9`);
 guarda os últimos 8 bits de %edx em %eax; `movsbl %dl,%eax`
`lea .0xffffffff0(%eax,%ebx,1),%ebx → result += s[i]-48`

Estes 3 exemplos mostram a ignorância relativa à instrução `lea`, apesar desta ter sido devidamente explicada nos slides 16 e 17 de ISA_2, ter tido um exercício no TPC5 sobre esta instrução, e ter sido discutida no fórum.

Encontram-se ainda incorreções na instrução `movsbl`: esta faz a conversão `char→int`, i.e., pega num valor de 8 *bits* com sinal (num registo de 8 *bits*) e converte-o para um valor de 32 *bits* com sinal, estendendo o *bit* do sinal e colocando o resultado num registo de 32 *bits*.

Exemplo: o número negativo -1, que é representado em binário, em complemento para 2, pelo conjunto de *bits* `1111 11112` passa para `1111 1111 1111 1111 1111 1111 1111 11112`.

De notar que a expressão `result += s[i]-'0';` vai exigir a conversão `char→int`, porque `result` é uma variável do tipo inteiro.

Outros exemplos de confusão, estes relacionados com a distinção entre um apontador para uma estrutura de dados (a *string* cadeia no nosso exemplo), e os valores (carateres) dessa *string*:

`mov 0x8(%ebp),%esi - guarda o array s em %esi`

`mov 0x8(%ebp),%esi → guarda a cadeia em %esi`

Ainda outros exemplos de anotação confusa:

`je 0x80483c9 <CONTIN+44>`
`lea 0xffffffff0(%edx),%eax - ciclo`

Este tipo de comentário conduz a uma confusão, já que pode ser interpretado como uma explicação para o que faz a instrução `lea` (e então estaria totalmente incorreto), quando provavelmente quem o redigiu pretendia simplesmente indicar que o ciclo começava nesta instrução...

Push `%ebp` ① coloca o "%ebp" no topo da stack
 mov `%esp,%ebp` ① copia o valor de "%esp" para "%ebp"
 mov `%eax,%eax` ①

De facto, a interpretação literal das instruções está correta, mas estes comentários em nada ajudam a interpretar o que realmente está a acontecer. Afinal, qual a finalidade destas instruções?

Vamos agora analisar um caso diferente e muito específico deste trabalho laboratorial.

cmp \$0x9, %al compara %al com 9
ja 0x20483e2 < contaN+34 > salta se valor > 9 (já é maior do que 9)

cmp \$0x9, %al
ja < contaN+34 >
if statement aver se i < 9
jump do if

Olhando para as duas instruções esta anotação afirma que o salto condicional `ja` é ativado se o valor em `%al` for maior do que 9.

No entanto, no código C estamos a avaliar se o caracter atual está entre 0 e 9, e não apenas se ele é maior do que 9.

Então, porque não foi usado um `jg` e depois feita uma comparação com 0?

Sabemos que, antes da execução destas instruções, o `%dl` tem o valor ASCII de um caracter (por exemplo '1', que segundo ASCII é representado em decimal por 49) e que é executada a instrução `lea -48(%edx), %eax`, que coloca em `%al` o valor decimal correspondente ao caracter em `%dl`, desde que este seja um número.

Assim, se o caracter em `%dl` for o 49 (ASCII para '1') `%al` terá o valor 1. Se o `%dl` tiver um valor menor do que 48 (ASCII para '0') ou maior do que 57 (ASCII para '9') `%al` irá conter um valor decimal negativo ou maior do que 9.

Se no salto condicional no código tivesse sido um `jg`, apenas os casos em que o valor seria superior a 9 é que eram contemplados.

Usando um `ja` a comparação está a ser feita ignorando o sinal tal como binário puro (relembro que os valores em decimal são representados usando complemento para 2), contemplando os casos em que `%al` é maior do que 9 mas também menor do que 0: um valor negativo em complemento para 2 terá sempre um binário $1xxxxx_2$ (aqui usando apenas 6 bits) que será sempre maior do que 9, que tem um binário 001001_2 , quando estes valores são interpretados como binário puro.

Assim, o `cmp` em conjunto com o `ja` são capazes de simultaneamente contemplar as 2 condições `c < 0` e `c > 9`, eliminando a necessidade de dois blocos de teste independentes.

Apresento agora alguns exemplos de respostas incorretas às questões concretas que foram colocadas, as quais, tendo ocorrido tantas vezes com a mesma resposta, sugerem que as pessoas não pensaram pelas cabeças delas:

Condição de teste do ciclo for: `s[i] != '10'`
o modo como a variável `i` é atualizada: `i++`

condição de teste ciclo for: quando `s[i] = '10'` sai

expressão em C que atualiza o valor de `result` no ciclo:
~~result = result + s[i] - '0'~~
`result += s[i] - '0'`

Acham que estas resoluções são respostas aos pedidos feitos?

A resposta apresenta código assembly?