TPC4 e Guião 3

Resolução dos exercícios

1 Controlo do fluxo de execução de instruções

Este exercício obriga a olhar com atenção o código "desmontado" e a pensar nos modos de codificação dos endereços-alvo nas instruções de salto. Vai obrigar também a alguma aritmética com valores hexadecimais...

a) **(A)** A instrução `jge` tem como endereço-alvo um valor (com sinal) relativo ao PC/IP (depois de ele ter sido incrementado para apontar para a próxima instrução); como o valor é negativo (o *byte* com `0x9e`), o endereço-alvo pode ser calculado de 2 modos: (`0x8048d1c + 2) - 0x62` (complemento para 2 de `0x9e`) ou (`0x8048d1c + 2) + 0xffffff9e` (extensão do *byte* `0x9e` para 32 bits, e desprezando o bit de *carry*). Como o código "desmontado" mostra, esse valor é `0x8048cbc`.

```
8048d1c: 7d 9e              jge 8048cbc
```

b) **(A)** De acordo com a notação produzida pelo *disassembler*, o endereço-alvo da instrução `jmp` é o endereço absoluto `0x8047c42`. De acordo com a codificação binária, este endereço deverá ser o valor relativo ao PC/IP que se encontra `0x54` *bytes* adiante da instrução `mov`. Subtraindo estes valores, chegamos ao endereço `0x8047bee`, tal como confirmado pelo código desmontado.

```
0x8047bec: eb 54            jmp 8047c42
0x8047bee: c7 45 f8 10      mov $0x10,0xfffffff8(%ebp)
```

c) **(R)** O endereço-alvo está à distância `0x10c2` relativo a `0x8048907`. Adicionando esses valores temos o endereço `0x80499c9`

```
8048902: e9 c2 10 00 00   jmp 80499c9
```

d) **(R)** Há 3 instruções de salto a completar neste exercício:

(i) um `jmp` para um endereço especificado em modo directo, que irá ser codificado em binário com um valor relativo ao PC/IP (em *little endian*); cálculo a fazer: subtrair ao endereço destino `0x80436c1` o endereço da instrução seguinte, `0x8043568` (dá um valor positivo, maior que `0xff` ou `127`; logo o compilador opta por representá-lo com 4 *bytes* e não 2: `0x159`):

```
8043563: e9 59 01 00 00     jmp 80436c1
```

(ii) um salto condicional, `je`, também para um endereço especificado em modo directo e relativo ao PC/IP, ocupando neste caso apenas 1 *byte*; cálculo a fazer: o mesmo, i.e., subtrair ao endereço destino `0x8043548` o endereço da instrução seguinte, `0x804356f` (dá um valor negativo, já em complemento para 2, mas representável com apenas 1 *byte*: `d9`):

```
804356d: 74 d9              je  8043548
```

(iii) um `jmp` para um endereço especificado em modo indirecto; i.e., a localização na memória onde se encontra o endereço-alvo da instrução de salto, vem especificada como se fosse um operando (duma instrução de `mov` ou duma operação aritmética/lógica) em memória, e, neste caso, encontra-se explicitamente codificado nos últimos 4 *bytes* da instrução (na ordem inversa, por ser *little endian*):

```
8043571: ff 24 80 35 04 08 jmp *0x8043580
```

2 Ciclo *While*

O código gerado na compilação de ciclos pode ser complicado de analisar, devido aos diferentes tipos de optimização do código do ciclo que o compilador poderá optar, para além da dificuldade em mapear variáveis do programa a registos do CPU. Para adquirimos alguma técnica, nada como começar com um ciclo relativamente simples.

O ficheiro em *assembly* que o comando `gcc -S -O2` irá gerar poderá conter algo do tipo:

```
07    pushl   %ebp
08    movl    %esp,%ebp
09    movl    16(%ebp),%ecx
10    pushl   %esi
11    movl    8(%ebp),%esi
12    testl   %ecx,%ecx
13    pushl   %ebx
14    movl    12(%ebp),%ebx
15    setg    %al
16    cmpl    %ecx,%ebx
17    setl    %dl
18    andl    %edx,%eax
19    testb   $1,%al
20    je      .L7
21    .align  16
22 .L5:
23    imull   %ecx,%ebx
24    addl    %ecx,%esi
25    decl    %ecx
26    testl   %ecx,%ecx
27    setg    %al
28    cmpl    %ecx,%ebx
29    setl    %dl
30    andl    %edx,%eax
31    testb   $1,%al
32    jne     .L5
33 .L7
34    popl    %ebx
35    movl    %esi, %eax
36    popl    %esi
37    popl    %ebp
38    ret
```

a) **(A)** A análise do modo como os argumentos são recuperados no código da função dá-nos uma boa pista de como o `gcc` usa os registos no cálculo de expressões de teste.
Algumas versões do `gcc` geram código com diferenças significativas, como por ex., usarem os registos `%eax` e `%edx` em substituição de `%al` e `%dl` (o que requer colocação prévia a 0, uma vez que as instruções de `set` apenas alteram registos de 8-bits).

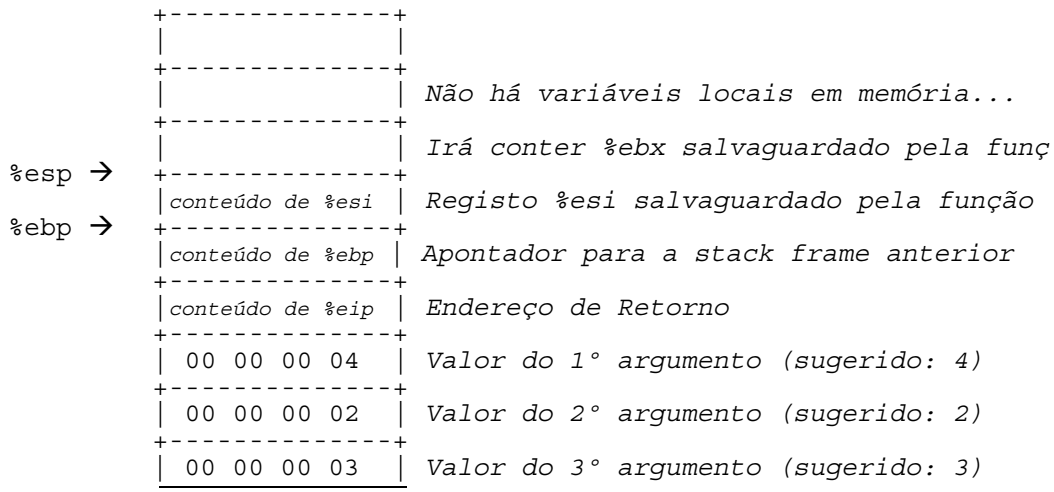
Utilização dos Registos		
Registo	Variável	Atribuição inicial
`%esi`	`x`	`valor recebido do 1° arg (x)`
`%ebx`	`y`	`valor recebido do 2° arg (y)`
`%ecx`	`n`	`valor recebido do 3° arg (n)`
`%al`	`temp1 (=n>0)`	`valor booleano da expressão (n > 0)`
`%dl`	`temp2 (=y<n)`	`valor booleano da expressão (y < n)`

Para confirmar esta utilização dos registos, proceda conforme sugerido na alínea **b)**.

b) **(A)** (Feito na aula; valores sugeridos para inicializar `x`, `y` e `n`: 4, 2, 3)

c) **(R/B)** (Feito na aula com os grupos que melhor se prepararam antes da sessão laboratorial)

d) **(A/R)** A resolução completa desta alínea vai depender do código gerado pelo `gcc` e da localização em que deverá ser executado no PC, entre outros aspectos. Assim, não faz sentido indicar aqui valores que digam respeito a endereços de memória, nem a conteúdos de registos que foram salvaguardados, pois serão distintos dos observados no laboratório; apenas se preencherá o diagrama com os valores estimados.
Nota 1: cada linha representa 4 células; conteúdo da célula com menor endereço: à direita.
Nota 2: o enunciado pede o diagrama da *stack frame* após o 2º *breakpoint*, assumindo que nessa altura a *stack frame* já estaria definida, o que não é o caso com o código gerado por esta versão do `gcc`, uma vez que a salvaguarda do registo `%ebx` (à responsabilidade desta função chamada) apenas ocorre 2 instruções após este *breakpoint*:



Os conteúdos actuais dos registos `%esp` e `%ebp` são obtidos da análise directa no `gdb`, após o 2º *breakpoint*; o dos registos salvaguardados tb poderão sê-lo, ou então analisando o conteúdo das células de memória nos endereços da *stack*.

e) **(A/R)** A expressão de teste aparece na linha 3 do código C. No código *assembly*, é implementado pelas instruções nas linhas 12 a 19 (o cálculo inicial da expressão de teste, fora do ciclo), bem como nas instruções de 26 a 31 (cálculo dentro do ciclo).
O bloco *body-statement* encontra-se nas linhas 4 a 6 no código C, e nas linhas 23 a 25 no código *assembly*.
Neste caso, o compilador não fez nenhuma optimização de destacar. Contudo, se substituir na expressão de teste o operador booleano `&` pelo operador de lógica proposicional `&&` (operações descritas no enunciado do TPC anterior), irá provavelmente encontrar uma optimização. Qual?

Eis um exemplo de código devidamente anotado (apenas do corpo da função):

```
// Inicialmente x, y, e n estão, respectivamente, à distância de 8, 12, e 16 células de %ebp
09    movl    16(%ebp),%ecx   ; coloca n em %ecx
11    movl    8(%ebp),%esi    ; coloca x em %esi
12    testl   %ecx,%ecx       ; testa n(=n&n), i.e., afecta essencial/ os bits  ZF e SF
14    movl    12(%ebp),%ebx   ; coloca y em %ebx  (não afecta nenhuma flag)
15    setg    %al             ; coloca em  %al  o valor lógico de  (n > 0)  (é 0 ou 1)
16    cmpl    %ecx,%ebx       ; calcula y-n  (afectando as flags ou bits de condição)
17    setl    %dl             ; coloca em  %dl  o valor lógico de  (y-n<0)  ou  (y<n)
18    andl    %edx,%eax       ; coloca em %al o valor lógico de  (n > 0) & (y < n)
19    testb   $1,%al          ; testa o bit menos significativo
20    je      .L7             ; se  = 0,  vai para  fim_do_ciclo
21    .align  16
```

```
22 .L5:                    ciclo:
23    imull    %ecx,%ebx      ; y *= n
24    addl     %ecx,%esi      ; x += n
25    decl     %ecx           ; n--
26    testl    %ecx,%ecx      ; testa n(=n&n), i.e., afecta essencial/ os bits  ZF e SF
27    setg     %al            ; coloca em  %al  o valor lógico de  (n > 0)
28    cmpl     %ecx,%ebx      ; calcula y-n  (afectando os bits de condição)
29    setl     %dl            ; coloca em  %dl  o valor lógico de  (y < n)
39    andl     %edx,%eax      ; coloca em %al  o resultado de  (n > 0) & (y < n)
31    testb    $1,%al         ; testa o bit menos significativo
32    jne      .L5            ; se  != 0,  vai para  ciclo
33 .L7:                    fim_do_ciclo:
35    movl     %esi,%eax      ; coloca em %eax o valor de retorno (x)
```

Note a forma um tanto ou quanto estranha de implementar a expressão de teste: aparentemente o compilador sabe que as duas condições de teste - `(n > 0)` e `(y < n)` - apenas podem tomar os valores de 0 ou 1, e daí apenas precisa de testar o bit menos significativo do resultado do `&`. O compilador poderia ter sido mais "esperto" e usado apenas a instrução `testb %dl,%al`, para efectuar a operação `&` (em substituição das instruções 18 e 19).

f) **(R)** Versão do tipo `goto` (em C) da função, com uma estrutura semelhante ao do código *assembly* (tal como foi feito para a série Fibonacci):

```
1  int while_loop_goto(int x, int y, int n)
2  {
3     if (!((n > 0) & (y < n))) goto done;
4     loop:
5        x += n;
6        y *= n;
7        n--;
8        if ((n > 0) & (y < n)) goto loop;
9     done:
10    return x;
11 }
```

3 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)** 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) **(A)** Rotina...

b) **(R)** Ver alínea seguinte…

c) **(R)** A partir do ficheiro executável que foi disponibilizado, `m_contaN`, é possível desmontá-lo para *assembly*, localizar a parte de código simbólico correspondente à função `contaN` e ainda distinguir as partes de incialização e término (da função) do corpo da função (a parte pertinente neste exercício).
O código simbólico da função deverá ter um aspecto semelhante ao seguinte (este código inclui já uma anotação introduzida manualmente):

```
4   contaN:
5     pushl   %ebp              ; inicialização da função:    salvaguarda frame pointer antigo
6     movl    %esp, %ebp        ;                             cria um novo frame pointer
7     pushl   %esi              ;                             salvaguarda registo %esi
8     pushl   %ebx              ;                             salvaguarda registo %ebx
9     movl    8(%ebp),%esi      ; coloca o apontador para início do array cadeia em %esi
10    movl    12(%ebp),%ecx     ; coloca o valor da variável c em %ecx
11    movb    (%ecx,%esi),%dl   ; coloca o caracter, na posição c da cadeia, em %dl
12    xorl    %ebx, %ebx        ; inicializa a zero %ebx (registo alocado à var local result)
13    testb   %dl, %dl          ; testa se %dl é 0 (caracter "null" em ASCII)
14    je      .L9               ; je equivale a jz, i.e., salta para o fim se é o fim da cadeia
15    .p2align 2,,3
16  .L7:                      ;ciclo
17    leal    -48(%edx), %eax   ; subtrai 48 (código ASCII de 0) a %dl e coloca em %al
18    cmpb    $9, %al           ; calcula (%al - 9)
19    ja      .L4               ; salta p/ .L4 se valor acima de 9 (i.e., não é dígito em ASCII)
20    movsbl  %dl,%eax          ; coloca o caracter lido, estendido para 32bits c/ sinal, em %eax
21    leal    -48(%eax,%ebx),%ebx       ; soma o dígito e acumula o valor em result
22  .L4:
23    incl    %ecx              ; actualiza o valor da posição na cadeia
24    movb    (%ecx,%esi),%al   ; coloca o caracter, na nova posição da cadeia, em %al
25    testb   %al, %al          ; novo teste: se %al é 0 ( caracter "null" em ASCII)
26    movb    %al, %dl          ; transfere o caracter para %dl (não afecta bits de condição)
27    jne     .L7               ; salta para o início do ciclo se não é o fim da cadeia
28  .L9:                      ;fim do ciclo
29    movl    %ebx, %eax        ; o somatório calculado (result) é colocado em  %eax
30    popl    %ebx              ; término da função: recupera registo  %ebx
31    popl    %esi              ;           recupera registo  %esi
32    leave                     ;           recupera stack pointer e  frame pointer antigo
33    ret                       ;           retorno da função
```

Pode-se ver que:

- o código da função calcula 3 expressões de teste e efectua os correspondentes saltos condicionais; a 1ª e a última expressões de teste são equivalentes (linhas 11 e 13 na 1ª, e 24 e 25 na última) o que sugerem uma ligação entre ambas; a 2ª está associada a uma típica estrutura de `if...then`;
- o código entre as linhas 16 e 27 corresponde a um ciclo; como existe um teste_e_salto antes deste ciclo – "salte para após o ciclo se o caracter lido for *null*" – e um outro complementar a este dentro do ciclo – "salte para trás se o caracter lido não for *null*" – tudo leva a crer que este ciclo corresponde a um ciclo `while` ou a um ciclo `for`;
- expressões de teste: a 1ª e a última pretendem verificar se o caracter lido é o fim da cadeia ou não, i.e., se `cadeia[c]=0` ; a 2ª pretende verificar se o caracter lido – o valor de 8 bits que está num registo de 32 – subtraído de 48 (o código ASCII de 0) é menor ou igual a 9, o que corresponde a verificar se é o código ASCII de um dígito ou não, i.e., se `cadeia[c]>='0'&&cadeia[c]<='9'`;
- ao encontrar a inicialização de uma variável antes do ciclo e a sua actualização dentro do ciclo, muito provavelmente estamos na presença de um ciclo `for`: de facto, o registo `%ecx` é inicializado com o valor de `c` antes do ciclo (linha 10), e incrementado de 1 unidade no interior do ciclo (linha 23); este registo deve conter uma variável local associada à iteração do ciclo, a que corresponde `i` (e então, nas expressões de teste referidas em cima substituir `c` por `i`)
- a variável que vai ser usada para calcular o resultado (um somatório), `result`, deve estar no registo `%ebx`: este registo é inicializado a 0 e é copiado para `%eax` no fim do ciclo, sendo este o registo usado para o valor de retorno duma função.

d) **(R)** 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 optimização, é possível chegar-se ao seguinte código original em C:

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