

TPC6 e Guião Laboratorial

Análise e resolução dos exercícios

Este documento apresenta uma proposta de análise do problema que vai para além da resolução propriamente dita do TPC6. Está estruturado em várias secções:

1. Apresenta o código *assembly* resultante do exercício 1.a) com a respetiva anotação para ajudar a identificar a alocação de registos pelas variáveis feita pelo compilador, conforme pedido em 1.b)
2. Apresenta o código do `main` em C, conforme sugerido em 1.c) (i)
3. Faz a validação da utilização dos registos (objetivo do exercício 1.c)), mostrando quando os registos são inicializados (sugestão 1.c) (ii)), onde deverão ser inseridos os pontos de paragem (sugestão 1.c) (iii)) e confirmando depois a utilização dos registos conforme esperado (sugestão 1.c) (iv))
4. Apresenta a estrutura da *stack frame* da função, começando pela estimativa da sua estrutura (pedido em 1.e) (i)) e confirmando posteriormente a estrutura e conteúdos estimados (pedido em 1.e) (ii))
5. Discute com detalhe a estrutura de controlo do ciclo `while` (pedido em 1.f))
6. Apresenta uma proposta de uma versão de código C com `goto` (pedido em 1.g))

Em anexo é ainda apresentada uma versão do código *assembly* pedido em 1.a) mas com uma explicação mais detalhada de cada instrução.

1. Anotação do código em *assembly* do ciclo `while`

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

Eis o código *assembly* que o comando `gcc -S -O2` irá gerar na máquina remota, anotado manualmente, por vezes com alguma informação excessiva para um melhor esclarecimento:

```
while_loop:
    pushl    %ebp                ; salvaguarda na stack o frame pointer da função chamadora (%ebp)
    movl     %esp, %ebp          ; define o valor do frame pointer desta função: %esp -> %ebp
    movl     16(%ebp), %edx       ; vai buscar o 3º argumento à stack e coloca-o em registo: n -> %edx
    testl    %edx, %edx          ; testa o valor de n (fazendo n<AND>n, que é =n), alterando só as flags;
    pushl    %ebx                ; salvaguarda o conteúdo de %ebx na stack, pois vai precisar deste registo
    movl     12(%ebp), %eax       ; vai buscar o 2º argumento à stack e coloca-o em registo: y -> %eax
    movl     8(%ebp), %ebx        ; vai buscar o 1º argumento à stack e coloca-o em registo: x -> %ebx
    jle      .L3                 ; salta para o fim do while se as flags indicarem que n ≤ 0 (less or equal)
    movl     %edx, %ecx          ; cria uma variável temp em %ecx para calcular 16*n: temp= n
    sall     $4, %ecx            ; um shift aritmético de 4 bits para a esquerda de temp é equiv a temp=n*2^4
    cmpl     %ecx, %eax          ; compara y com 16*n, i.e., altera as flags com a operação y-16*n
    jge      .L3                 ; salta p/ fim do while se as flags indicarem que y ≥ 16*n (greater or equal)
    .p2align 2, , 3              ; pad-to-align: acrescentar instruções no-ops para que o endereço da próxima
                                ; instrução seja múltiplo de 4; compilação com -O2 pode ir mais longe
    .L6:
    addl     %edx, %ebx          ; aqui começa o corpo do ciclo while com x += n
    imull    %edx, %eax          ; y *= n
    decl     %edx                ; n--
    subl     $16, %ecx           ; p/ evitar calcular de novo 16*n o compilador optou por temp=temp-16
```

```

    testl    %edx, %edx        ; estas 3 instruções repetem o cálculo da condição do ciclo while
    jle      .L3
    cmpl     %ecx, %eax        ; na próxima instrução, em vez de saltar para fora do ciclo se  $y \geq 16*n$ 
    jl       .L6               ; salta para o início do ciclo while se ainda se mantiver  $y < 16*n$ 
.L3:
    movl     %ebx, %eax        ; coloca o valor a devolver (x) no registo convencionado para tal (%eax)
    popl     %ebx              ; recupera o conteúdo de %ebx, que tinha sido salvaguardado no início
    leave    %ebx               ; coloca o %esp a apontar para o mesmo endereço na stack que o frame
                                ; pointer (em %ebp) e recupera o frame pointer da função chamadora
    ret                               ; regressa à função chamadora, recuperando o IP que está no topo da pilha

```

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.

O valor de $n*16$, utilizado numa das condições do ciclo, é pré-calculado e guardado em %ecx; em cada iteração do ciclo este valor é atualizado subtraindo 16, evitando-se assim a multiplicação $n*16$, uma vez que n é decrementado de 1 unidade.

Utilização dos Registos		
Variável	Registo	Atribuição inicial
x	%ebx	valor recebido do 1º arg (x)
y	%eax	valor recebido do 2º arg (y)
n	%edx	valor recebido do 3º arg (n)
temp (=n*16)	%ecx	o valor recebido do 3º arg (n) e logo a seguir $16*n$

2. Código do main em C

```

1  int main() {
2      int a=4, b=3, c=2, x;
3      x = while_loop (a, b, c);
4      printf ("%d\n", x);
5      return x;
6  }

```

3. Validação da utilização dos registos

Para confirmar esta utilização dos registos, procede-se conforme sugerido em 1.c) do guião:

- Coloca-se no mesmo ficheiro fonte onde estava o código do ciclo, um programa simples (main) que chame a função while_loop, passando como argumentos para a função os valores 4, 2 e 3, respetivamente.
- Compila-se com os switches indicados e desmonta-se o executável com o comando `objdump -d`.
- Analisando a parte do código desmontado com a função while_loop procura-se a 1ª instrução a seguir à inicialização de cada uma das variáveis e regista-se o endereço onde se encontram essas instruções (e ainda após temp passar a ser $16*n$);
- Invocando o debugger (gdb), introduzem-se pontos de paragem (breakpoints) nesses endereços, os quais irão depois permitir que a execução do código seja interrompida nesses endereços para se verificar o conteúdo dos registos.
- Com os comandos apropriados do gdb manda-se executar o programa (comando `run`) e ele vai sendo interrompido de cada vez que encontra um breakpoint.
- Uma vez a execução do programa interrompida num breakpoint, pode-se então validar o conteúdo dos registos (por ex., com `info reg`), confirmando as estimativas que foram indicadas atrás.
- Após cada breakpoint continuar a execução do código (comando `cont`) até terminar a execução do programa.

Código desmontado da função while_loop com indicação dos endereços para os breakpoints:

Onde colocar breakpoints

```

0x08048354 <while_loop>:
0x08048354:    55                push    %ebp
0x08048355:    89 e5            mov     %esp,%ebp
0x08048357:    8b 55 10        mov     0x10(%ebp),%edx
0x0804835a:    85 d2            test    %edx,%edx
0x0804835c:    53              push    %ebx
0x0804835d:    8b 45 0c        mov     0xc(%ebp),%eax
0x08048360:    8b 5d 08        mov     0x8(%ebp),%ebx
0x08048363:    7e 1c            jle     0x8048381<while_loop+45>
0x08048365:    89 d1            mov     %edx,%ecx
0x08048367:    c1 e1 04        shl     $0x4,%ecx
0x0804836a:    39 c8            cmp     %ecx,%eax
0x0804836c:    7d 13            jge     0x8048381<while_loop+45>
0x0804836e:    89 f6            mov     %esi,%esi
0x08048370:    01 d3            add     %edx,%ebx
0x08048372:    0f af c2        imul    %edx,%eax
0x08048375:    4a              dec     %edx
0x08048376:    83 e9 10        sub     $0x10,%ecx
0x08048379:    85 d2            test    %edx,%edx
0x0804837b:    7e 04            jle     0x8048381<while_loop+45>
0x0804837d:    39 c8            cmp     %ecx,%eax
0x0804837f:    7c ef            jl      0x8048370<while_loop+28>
0x08048381:    89 d8            mov     %ebx,%eax
0x08048383:    5b              pop     %ebx
0x08048384:    c9              leave
0x08048385:    c3              ret

```

Inicialização de registos com as variáveis x, y, n e temp

Instrução nova inserida aqui pelo gcc ao criar o executável, que não faz nada de útil, apenas força que o início do corpo do ciclo (que está na instrução seguinte) comece num endereço que é múltiplo de 16

Variável	Registo	Break1	Break2	Break3	Break4
x	%ebx	lixo	4	4	4
y	%eax	lixo	2	2	2
n	%edx	3	3	3	3
temp (=n*16)	%ecx	lixo	lixo	3	48

4. A stack frame da função

Pretende-se neste exercício que se preencham 3 tipos de informação:

- À esquerda do desenho da *stack*, os endereços do início de algumas "caixas".
- No interior das "caixas" o valor numérico que lá deveria estar (pode ser em hexadecimal)
- À direita das "caixas", uma explicação do valor que se encontra na respetiva "caixa".

Cada "caixa" não é mais que um bloco de 32 bits armazenado em 4 células de 1 *byte* cada, em que o conteúdo da célula com menor endereço é o *byte* mais à direita de um valor de 32 bits (*little endian*).

A resolução completa implica uma análise detalhada do código gerado pelo `gcc`, do conteúdo de alguns registos e da localização em que deverá ser executado no PC, entre outros aspetos. Por exemplo, no `gdb` é possível visualizar o código da função `main` usando o comando `disas main`:

```

0x08048388 <main+0>:    push    %ebp
0x08048389 <main+1>:    mov     %esp,%ebp
0x0804838b <main+3>:    sub     $0x8,%esp
0x0804838e <main+6>:    and     $0xfffffffff0,%esp
0x08048391 <main+9>:    push    %eax
0x08048392 <main+10>:   push    $0x3
0x08048394 <main+12>:   push    $0x2

```

```

0x08048396 <main+14>:    push    $0x4
0x08048398 <main+16>:    call   0x8048354 <while_loop>
0x0804839d <main+21>:    leave
0x0804839e <main+22>:    ret

```

Neste caso, o valor do endereço de regresso que deverá estar na *stack* deverá ser o endereço da instrução na *main* imediatamente a seguir à invocação da função (após a instrução **call**), ou seja 0x0804839d (note que os endereços podem variar ligeiramente no seu caso, dependendo da forma como o código C está escrito).

Esta é a estimativa do conteúdo da *stack frame* associada à função `while_loop`, quando é invocada com os argumentos sugeridos e antes de se confirmar com o `gdb`:

	+-----+ +-----+ +-----+	Não há variáveis locais em memória...
valor de %esp →	+-----+ +-----+ +-----+	Registo %ebx salvaguardado pela função
valor de %ebp →	+-----+ +-----+ +-----+	Apontador para moldura/quadro anterior
	+-----+ +-----+ +-----+	Endereço de regresso da função
	+-----+ +-----+ +-----+	Valor do 1º argumento (sugerido: 4)
	+-----+ +-----+ +-----+	Valor do 2º argumento (sugerido: 2)
	+-----+ +-----+ +-----+	Valor do 3º argumento (sugerido: 3)

O valor dos registos salvaguardados, incluindo o apontador para o quadro (*frame pointer*) da *main* na *stack*, i.e., tudo o que está com ?? na figura) pode ser obtido no `gdb`, parando a execução do código logo na 1ª instrução da função (coloque aí um *breakpoint*), e analisando o conteúdo desses registos (que ainda não foram colocados na *stack*).

Para confirmar os conteúdos destas 24 posições de memória na *stack*, coloca-se outro *breakpoint* no fim da parte de arranque da função, i.e., após a salvaguarda do registo `%ebx`; quando o programa parar aí pode-se então usar um dos comandos para examinar dados do *debugger* e visualizar o conteúdo das 24 células com início no topo da pilha (valor em `%esp`), quer *byte a byte* (dá para ver o funcionamento *little endian*), quer em 6 blocos de 4 bytes.

Sugestão de procedimento a seguir, passo a passo:

(i) Escrevendo `disas while_loop` tem-se acesso novamente ao código da função:

```

0x08048354 <while_loop+0>:    push    %ebp
0x08048355 <while_loop+1>:    mov     %esp,%ebp
0x08048357 <while_loop+3>:    mov     0x10(%ebp),%edx
0x0804835a <while_loop+6>:    test    %edx,%edx
0x0804835c <while_loop+8>:    push    %ebx
0x0804835d <while_loop+9>:    mov     0xc(%ebp),%eax
0x08048360 <while_loop+12>:   mov     0x8(%ebp),%ebx
0x08048363 <while_loop+15>:   jle     0x8048381 <while_loop+45>
0x08048365 <while_loop+17>:   mov     %edx,%ecx
0x08048367 <while_loop+19>:   shl     $0x4,%ecx
0x0804836a <while_loop+22>:   cmp     %ecx,%eax
0x0804836c <while_loop+24>:   jge     0x8048381 <while_loop+45>
0x0804836e <while_loop+26>:   xchg    %ax,%ax
0x08048370 <while_loop+28>:   add     %edx,%ebx
0x08048372 <while_loop+30>:   imul    %edx,%eax
0x08048375 <while_loop+33>:   dec     %edx
0x08048376 <while_loop+34>:   sub     $0x10,%ecx
0x08048379 <while_loop+37>:   test    %edx,%edx
0x0804837b <while_loop+39>:   jle     0x8048381 <while_loop+45>

```

```

0x0804837d <while_loop+41>:    cmp     %ecx,%eax
0x0804837f <while_loop+43>:    jl      0x08048370 <while_loop+28>
0x08048381 <while_loop+45>:    mov     %ebx,%eax
0x08048383 <while_loop+47>:    pop     %ebx
0x08048384 <while_loop+48>:    leave
0x08048385 <while_loop+49>:    ret

```

(ii) Coloca-se um *breakpoint* após `push %ebx` (assinalado em cima):

```
break *0x0804835d
```

(iii) Executa-se o programa (`run`) e ele parará no *breakpoint*; visualiza-se o conteúdo dos registos escrevendo `info reg`; para responder integralmente a esta questão precisamos de saber o valor do apontador para o topo da pilha (registo `%esp`) e o apontador para o quadro da função, o *frame pointer* (registo `%ebp`).

(iv) Examinam-se 6 "palavras" na memória a partir do topo da pilha (endereço em `%esp`):

```

x/6wx $esp
0xbfffe874:      0x007d3ff4  0xbfffe898  0x0804839d  0x00000004
0xbfffe884:      0x00000002  0x00000003

```

(v) Preenche-se agora a figura com estes valores, conforme pedido:

0xbfffe874 →	+	-----	+	Não há variáveis locais em memória...
	+		+	
	+	00 7d 3f f4	+	Registo <code>%ebx</code> salvaguardado pela função
0xbfffe878 →	+	bf ff e8 98	+	Apontador para moldura/quadro anterior
	+		+	
	+	08 04 83 9d	+	Endereço de regresso da função
	+		+	
	+	00 00 00 04	+	Valor do 1º argumento (sugerido: 4)
	+		+	
	+	00 00 00 02	+	Valor do 2º argumento (sugerido: 2)
	+		+	
	+	00 00 00 03	+	Valor do 3º argumento (sugerido: 3)
	+	-----	+	

5. Estrutura do ciclo `while`

A expressão de teste é implementada em dois blocos. O primeiro bloco verifica se o ciclo deve ser executado na 1ª iteração:

```

testl    %edx, %edx    ; testa se n é 0; n <AND> n só é zero se n for zero...
jle      .L3           ; salta para fim do ciclo se n <= 0
movl     %edx, %ecx    ; temp = n
sall     $4, %ecx      ; temp = n*16
cmpl     %ecx, %eax    ; compara y com n*16
jge      .L3           ; salta para fim do ciclo se y >= 16

```

O segundo bloco, no fim do corpo do ciclo, verifica se este deve iterar:

```

decl     %edx          ; n--
subl     $16, %ecx     ; temp = temp-16 (=n*16)
testl    %edx, %edx    ; testa se n é 0
jle      .L3           ; salta para o fim do ciclo se n <= 0
cmpl     %ecx, %eax    ; compara y com n*16
jl       .L6           ; salta para o fim do ciclo se y < n*16

```

6. Versão do código com goto

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*16)) goto done;
4      loop:
5          x += n;
6          y *= n;
7          n--;
8          if ((n > 0) && (y < n*16)) goto loop;
9      done:
10     return x;
11 }
```

Anexo

Alguns comentários prévios:

- a anotação de código é para nos ser útil quando analisamos o código *assembly*, não deve ser usado apenas para explicar o que faz a instrução em *assembly*
- a anotação dá-nos uma indicação sobre o que está de facto a acontecer, se é algo relacionado com uma instrução do código fonte (em C, por exemplo) ou se é para implementar uma estrutura de controlo, ou ainda se é para o arranque ou término duma função (e o que é que de facto está a acontecer)
- o verbo “*push*” em inglês quer dizer “*empurrar*” e não “*puxar*” (a tradução desta é “*pull*”); assim, no código *assembly* é basicamente um “*empurrar para o topo da pilha*”, ou simplesmente “*empilhar*”
- a instrução de `test` (dentro do ciclo `while_loop`) vai operar com 2 operandos para alterar *flags*; vai, de facto, realizar a operação $n \text{ AND } n$ e sabe-se que $n \text{ AND } n = n$; por isso se diz muitas vezes que se está a testar o valor de `n`; e este teste é para se saber se `n` é zero, positivo, negativo...

A1. Explicação mais detalhada do código em *assembly* do ciclo `while`

Nova análise do código *assembly* da função `while_loop`, anotado (a ***bold***) e comentado:

```

pushl    %ebp                ; salvaguarda na stack o frame pointer da função
                                chamadora (que está ainda em %ebp)
                                %ebp aponta nesta altura para quadro da função chamadora na stack, mas é
                                necessário definir um novo apontador para o quadro na stack (frame pointer) da
                                função chamada (while_loop); guarda-se na stack o valor do frame pointer
                                que se encontra no registo %ebp, para o %ebp poder alojar o valor do frame
                                pointer da função chamada (while_loop).

movl     %esp, %ebp          ; define o valor do frame pointer desta função:
                                %esp -> %ebp

movl     16(%ebp), %edx       ; copia n (o 3º arg de while_loop) da stack p/ %edx
                                os argumentos da função while_loop foram colocados na stack da função
                                chamadora antes da execução desta função; assumindo argumentos de 4 bytes de
                                tamanho em IA-32, o primeiro encontra-se sempre à distância de 8 células do
```

		endereço no <i>frame pointer</i> (em <code>%ebp</code>) da função chamada (<code>while_loop</code>) durante a sua execução; o 2º argumento encontra-se a 12 células de <code>%ebp</code> , o 3º a 16, e assim por diante.
<code>testl %edx, %edx</code>		; testa <i>n</i> (faz & lógico de <code>%edx</code> consigo próprio) esta instrução implementa um & lógico entre os operandos, ativando as <i>flags</i> <code>ZF</code> , <code>CF</code> , <code>OF</code> , e <code>SF</code> de acordo com o resultado; quando os 2 operandos são iguais, tudo se passa como se o teste fosse sobre um dos operandos, já que o & lógico de um valor consigo mesmo dá sempre o próprio valor; as instruções de salto consultam estas <i>flags</i> para a tomada de decisão; portanto esta instrução vai então testar se <i>n</i> é zero, positivo ou negativo.
<code>pushl %ebx</code>		; salvaguarda conteúdo do <code>%ebx</code> garantir que os conteúdos dos registos <code>%eax</code> , <code>%ecx</code> , e <code>%edx</code> não são alterados quando se chama uma função é uma responsabilidade da função chamadora; se contiverem informação relevante para essa função ela terá de salvaguardar os seus conteúdos (na <i>stack</i>) antes de chamar a função <code>while_loop</code> de modo a que esta os possa usar, e depois terá de repor esses valores; a responsabilidade de salvaguardar os conteúdos dos registos <code>%ebx</code> , <code>%edi</code> , e <code>%esi</code> é da função chamada, caso ela necessite de usar esses registos (e neste caso a função necessita de usar 4 registos, os 3 que não tem de salvaguardar e mais este).
<code>movl 12(%ebp), %eax</code>		; copia <i>y</i> (o 2º arg de <code>while_loop</code>) da <i>stack</i> p/ <code>%eax</code> <code>%eax</code> toma o valor de <i>y</i> que foi passado como argumento.
<code>movl 8(%ebp), %ebx</code>		; copia <i>x</i> (o 1º arg de <code>while_loop</code>) da <i>stack</i> p/ <code>%ebx</code> a salvaguarda do conteúdo deste registo foi feita anteriormente para agora poder ser usado; <code>%ebx</code> toma o valor de <i>x</i> que foi passado como argumento.
<code>jle .L3</code>		; salta para depois do ciclo se $n \leq \text{zero}$ a instrução consulta as <i>flags</i> previamente ativadas pela instrução <code>testl</code> ; (de notar que as instruções de transferência de dados não alteram as <i>flags</i> , tais como os <code>movl</code> e <code>pushl</code> que estão entre o <code>testl</code> e este <code>jle</code>); neste caso, essa instrução em conjunto com esta de salto condicional estão a avaliar se o conteúdo de <code>%edx</code> é menor ou igual a zero; caso <i>n</i> seja menor ou igual a zero o controlo do programa salta para o fim de <code>while_loop</code> , em <code>.L3</code> (a primeira das duas condições é $n > 0$), não executando nenhuma iteração do corpo do ciclo.
<code>movl %edx, %ecx</code>		; coloca o valor de <i>n</i> numa variável temporária o valor de <i>n</i> é copiado para esta nova variável <code>temp</code> (em <code>%ecx</code>) de modo a ficar depois com $16 * n$.
<code>sall \$4, %ecx</code>		; desloca 4 bits p/ a esquerda (<i>shift left</i>) no valor binário de <code>temp</code>, equivalente a calcular $n * 2^4$ um deslocamento de 2 bits para a esquerda no nº binário 111_2 resulta em 11100_2 ; é equivalente a uma multiplicação de 2^2 pelo número em decimal; neste caso, está a ser feita uma multiplicação de 2^4 (16) pelo conteúdo de <code>%ecx</code> (que tem ainda o valor de <i>n</i>) que depois será útil para a avaliação de $y < 16 * n$; poder-se-ia usar <code>imull \$16, %ecx</code> , mas a instrução <code>sall</code> ocupa menos <i>bytes</i> em memória e <i>shifts</i> são mais rápidos de executar que multiplicações.
<code>cmpl %ecx, %eax</code>		; compara <i>y</i> com $16 * n$ fazendo $y - 16 * n$ e ativando as <i>flags</i>: as <i>flags</i> serão alteradas consoante o resultado da diferença é zero, negativo ou positivo.
<code>jge .L3</code>		; salta para fora do ciclo se o resultado da operação anterior for $\geq \text{zero}$

		esta instrução, em conjunto com o <code>cmpl</code> anterior, indica um salto para o fim do ciclo (em <code>L3</code>) caso <code>y</code> (em <code>%eax</code>) seja maior ou igual que <code>temp</code> (em <code>%ecx</code> , com $16*n$); caso $y > 16*n$ o controlo salta para <code>L3</code> , que se encontra no fim de <code>while_loop</code> , sem executar qualquer iteração do ciclo.
<code>.p2align 2,,3</code>		; etiqueta vista na resolução do TPC anterior.
<code>.L6:</code>		; etiqueta que indica o início do ciclo while etiqueta usada como referência para saltos condicionais; indica o início do corpo do ciclo <code>while</code> do código C.
<code>addl %edx, %ebx</code>		; calcula $x+=n$ é feita a soma de <code>x</code> (em <code>%ebx</code>) com <code>n</code> (em <code>%edx</code>) guardando o resultado em <code>x</code> ; é equivalente à primeira instrução do corpo do ciclo $x+=n$;
<code>imull %edx, %eax</code>		; calcula $y*=n$ multiplicação de <code>n</code> (em <code>%edx</code>) com <code>y</code> (em <code>%eax</code>) guardando o resultado em <code>y</code> ; equivalente à expressão $y*=n$ no código C.
<code>decl %edx</code>		; calcula $n--$ decremento de <code>n</code> (em <code>%edx</code>), equivalente a $n--$.
<code>subl \$16, %ecx</code>		; calcula o novo $temp=16*(n-1)$ trocando a multiplicação pela subtração $temp=16*n-16$ esta instrução evita o uso da multiplicação novamente, já que o <code>n</code> foi decrementado e é necessário recalcular $16*n$; assim, apenas se subtrai 16 a $16*n$, que é equivalente a subtrair uma unidade a <code>n</code> e voltar a fazer $16*n$ com o novo <code>n</code> , quer usando um <i>shift</i> ou uma multiplicação; é mais rápido fazer uma subtração do que uma dessas duas instruções.
<code>testl %edx, %edx</code>		; testa n (faz & lógico de <code>%edx</code> consigo próprio) preparação para a avaliação de $n > 0$, semelhante ao processo feito nas instruções antes do ciclo; no entanto, essa avaliação inicial foi feita para determinar se é possível entrar dentro do corpo do ciclo <code>while</code> ; agora, esta avaliação verifica se é possível continuar dentro do ciclo na próxima iteração, sendo esta instrução repetida a cada iteração; poder-se-ia evitar a repetição destas avaliações com alguma reorganização do código, no entanto este é o padrão normalmente usado por esta versão do compilador da GNU.
<code>jle .L3</code>		; salta para fora do ciclo se $n \leq$ zero salto para fora do ciclo (em <code>L3</code>) caso <code>n</code> (em <code>%edx</code>) seja menor ou igual que zero, falhando a avaliação de $n > 0$ do ciclo <code>while</code> .
<code>cmpl %ecx, %eax</code>		; compara y com $16*n$ fazendo $y-16*n$ e ativando as flags comparação de <code>y</code> com $16*n$, já contemplando o facto de <code>n</code> ter sido decrementado; no entanto, a avaliação inicial idêntica a esta foi feita para determinar se era possível entrar dentro do corpo do ciclo <code>while</code> ; agora, esta avaliação verifica se é possível continuar dentro do ciclo na próxima iteração (em conjunto com as instruções que avaliam se $n > 0$), sendo esta instrução repetida a cada iteração; poder-se-ia evitar a repetição destas avaliações com alguma reorganização do código, no entanto este é o padrão normalmente usado por esta versão do compilador da GNU.
<code>j1 .L6</code>		; salta para o início do ciclo se o resultado da operação anterior for $<$ zero salto para o início do ciclo (em <code>L6</code>) caso a condição $y < 16*n$ seja válida; caso esta condição não se verifique não é possível continuar dentro do corpo do ciclo, sendo a instrução seguinte a próxima a ser executada.

.L3:	<p>; etiqueta que indica o início do bloco de código após o ciclo while</p> <p>corresponde ao endereço da primeira instrução após o ciclo <code>while</code>; na função <code>while_loop</code> corresponde à preparação para devolver o valor <code>x</code> (<code>return x</code> no código C), mas que envolve uma série de passos antes de finalizar a execução desta função.</p>
<code>movl %ebx, %eax</code>	<p>; coloca <code>x</code> (o valor a devolver) no registo <code>%eax</code></p> <p>por convenção no IA-32, os valores devolvidos por funções são sempre colocados no registo <code>%eax</code> antes do seu término; poder-se-ia ter alocado logo o registo <code>%eax</code> à variável <code>x</code>, que é a que temos de devolver (<code>return x</code> no código C), eliminando esta instrução final.</p>
<code>popl %ebx</code>	<p>; recupera o valor original de <code>%ebx</code></p> <p>como foi feita a salvaguarda do conteúdo deste registo na fase de arranque da função agora é necessário recuperar esse valor, de modo à função chamadora manter o conteúdo que tinha inicialmente atribuído a este registo, antes de chamar <code>while_loop</code>.</p>
<code>leave</code>	<p>; recupera o <i>frame pointer</i> da função chamadora</p> <p>agora que a função <code>while_loop</code> acabou a sua execução é necessário recuperar a <i>frame pointer</i> da função chamadora, que foi salvaguardada na <i>stack</i> logo na 1ª instrução desta função; basta garantir que o <code>%esp</code> está a apontar para o mesmo sítio que <code>%ebp</code> e fazer um <code>popl %ebp</code>, que é o processo implementado pela instrução <code>leave</code>.</p>
<code>ret</code>	<p>; regressa à função chamadora</p> <p>é necessário que a próxima instrução a ser executada (cuja localização na memória se encontra sempre em <code>%eip</code>) seja a instrução na função chamadora imediatamente a seguir à instrução <code>call</code> que chamou a função <code>while_loop</code>; o endereço dessa instrução foi colocado na <i>stack</i> na execução da instrução <code>call while_loop</code> (encontra-se neste momento no topo da pilha), e tem agora que ser colocado no <code>%eip</code>; esta instrução seria equivalente a um <code>popl %eip</code>.</p>