

# Assembly do IA-32 em ambiente Linux

## TPC6 e Guião laboratorial

Alberto José Proença

### Objectivo e notas

A lista de exercícios/tarefas propostos no TPC6 / Guião laboratorial analisa o **suporte a estruturas de controlo e a funções em C**, no IA-32, com recurso a um depurador (*debugger*). Os exercícios para serem resolvidos e entregues antes da aula TP estão assinalados com uma caixa cinza, e repetem-se na última folha. Recomenda-se o uso do mesmo servidor que foi usado na sessão laboratorial anterior, para se garantir coerência na análise e discussão dos resultados.

O texto de “**Introdução ao GDB debugger**”, no fim deste guião, contém informação pertinente ao funcionamento desta sessão laboratorial, e é uma sinopse ultra-compacta do manual; a versão integral está disponível no site da GNU, e recomenda-se ainda a consulta dos documentos disponibilizados nas notas de apoio da disciplina (na Web), por se referirem a versões mais compatíveis com as ferramentas instaladas no servidor.

### Ciclo *While*

1. Coloque a seguinte função em C num ficheiro com o nome `while_loop.c`, e execute apenas a sua compilação para *assembly*, usando o comando `gcc -O2 -S while_loop.c`.

```
1 int while_loop(int x, int y, int n)
2 {
3     while ((n > 0) & (y < n)) { /* Repare no uso do operador '&' */
4         x += n;
5         y *= n;
6         n--;
7     }
8     return x;
9 }
```

- a) <sup>(A)</sup> Considerando que os argumentos passados para a função `x`, `y`, e `n`, se encontram respetivamente à distância 8, 12 e 16 do endereço especificado em `%ebp`, **preencha a tabela de utilização de registos** (semelhante ao exemplo da série Fibonacci); considere também a utilização de registos para variáveis temporárias (não visíveis no código C).

Registo	Variável	Atribuição inicial
	x	
	n	
	y	

b) **Confirme esta utilização dos registos**, directamente no computador.

**Sugestão de resolução:** (i) escrever o código do `main`, (ii) inseri-lo no ficheiro que contém a função, (iii) procurar no código *assembly* as instruções que alterem registos pela 1ª vez, (iv) inserir pontos de paragem logo a seguir a elas, e (v) executar o código de modo a parar nesses locais e assim confirmar os valores nos registos. Detalhes destas 5 tarefas:

- i. (A) **Construa em C** um programa simples (`main`) que use a função `while_loop`, e que não faça mais do que inicializar numericamente um conjunto de valores que irá depois passar como argumento para a função (experimente 4, 2 e 3, respetivamente).  
(Sugestão: use variáveis com designações diferentes das usadas na função)
- ii. (A) **Complete o ficheiro** `while_loop.c` com o programa `main` que elaborou e **crie um executável** pronto para ser depurado, usando o comando `gcc -Wall -O2 -g`.
- iii. (A) Desmonte o executável com o comando `objdump -d`, analise o código *assembly* e **identifique** em `while_loop`, a **1ª instrução** (e respectiva **localização**) **logo a seguir** a: (i) leitura de cada um dos argumentos da *stack* (nota: se o código gerado pelo compilador efectuar esta leitura em 3 instruções consecutivas, basta então identificar apenas a instrução que se segue à última leitura) e (ii) utilização pela 1ª vez de cada um dos registos de 8 bits (para quê?); **escreva aqui** essas instruções em *assembly* e sua **localização em memória** (lista de endereços de memória):
- iv. (A) Invocando o *debugger* (com `gdb <nome_fich_executável>`), **insira pontos de paragem** (*breakpoints*) nesses endereços, antes da execução das instruções; explicita aqui os comandos usados (e registe o nº de *breakpoint* atribuído a cada endereço):
- v. (A) Estime os valores atribuídos aos registos, **preenchendo esta tabela sem executar qualquer código** (apenas com base na análise do código *assembly*).  
Depois, **confirme esses valores** executando o programa dentro do *debugger* e, após cada paragem num *breakpoint*, visualizando o conteúdo dos registos com `print $reg`, ou com `info registers` (nota1: o `gdb` apenas aceita especificação de registos de 32 bits; nota2: no IA-32 os registos de 8 bits são parte dos registos de 32 bits).

Registo	Variável	Break1	Break_	Break_	Break_	Break_
	x					
	n					
	y					

- c) (R/B) Com base nos argumentos passados para a função `while_loop` (no `main`), é possível estimar quantas vezes o *loop* é executado na função. Para confirmar esse valor, uma técnica é introduzir um *breakpoint* na instrução de salto condicional de regresso ao início do *loop*. Indique o que fazer depois para **confirmar o nº de execuções** do *loop*.
- d) (A/R) Considerando que a *stack* cresce para cima, pretende-se construir o diagrama da *stack frame* da função `while_loop` logo após a execução da instrução antes do 2º *breakpoint*, com o máx. de indicações (endereços e conteúdos, ver 1ª linha da figura). Comente cada um dos conteúdos da *stack frame* (por ex., "endereço de regresso").

**Construa assim esse diagrama:** (i) estime os valores antes da execução do código, e (ii) confirme posteriormente esses valores, usando o depurador durante a execução do código (nota: neste diagrama, cada caixa representa um bloco de 32-bits em 4 células).

[illegible]

- e) <sup>(A/R)</sup> **Identifique** a expressão de teste e o corpo do ciclo `while` (*body-statement*) no bloco do código C, **e assinale** as linhas de código no programa em *assembly* que lhe são correspondentes. Que otimizações foram feitas pelo compilador?
- f) <sup>(R)</sup> Escreva uma versão do tipo `goto` (em C) da função, com uma estrutura semelhante ao do código *assembly* (tal como foi feito para a série Fibonacci).  
(Para fazer depois da sessão laboratorial)

## Anexo: Introdução ao GNU *debugger*

O GNU *debugger* GDB disponibiliza um conjunto de funcionalidades úteis na análise e avaliação do funcionamento de programas em linguagem máquina, durante a sua execução; permite ainda a execução controlada de um programa, com indicação explícita de quando interromper essa execução – através de *breakpoints*, ou em execução passo-a-passo – e possibilitando a análise do conteúdo de registos e de posições de memória, após cada interrupção.

Use o GDB para confirmar as tabelas de utilização de registos e o valor dos argumentos nos exercícios.

Nota: utilize primeiro `objdump` para obter uma versão “desmontada” do programa.

A tabela/figura seguinte (de CSAPP) ilustra a utilização de alguns dos comandos mais comuns para o IA-32.

Command	Effect
<b>Starting and Stopping</b>	
<i>quit</i>	Exit GDB
<i>run</i>	Run your program (give command line argum. here)
<i>kill</i>	Stop your program
<b>Breakpoints</b>	
<i>break sum</i>	Set breakpoint at entry to function <code>sum</code>
<i>break *0x80483c3</i>	Set breakpoint at address <code>0x80483c3</code>
<i>disable 3</i>	Disable breakpoint 3
<i>enable 2</i>	Enable breakpoint 2
<i>clear sum</i>	Clear any breakpoint at entry to function <code>sum</code>
<i>delete 1</i>	Delete breakpoint 1
<i>delete</i>	Delete all breakpoints
<b>Execution</b>	
<i>stepi</i>	Execute one instruction
<i>stepi 4</i>	Execute four instructions
<i>nexti</i>	Like <i>stepi</i> , but proceed through function calls
<i>continue</i>	Resume execution
<i>finish</i>	Run until current function returns
<b>Examining code</b>	
<i>disas</i>	Disassemble current function
<i>disas sum</i>	Disassemble function <code>sum</code>
<i>disas 0x80483b7</i>	Disassemble function around address <code>0x80483b7</code>
<i>disas 0x80483b7 0x80483c7</i>	Disassemble code within specified address range
<i>print /x \$eip</i>	Print program counter in hex
<b>Examining data</b>	
<i>print \$eax</i>	Print contents of <code>%eax</code> in decimal
<i>print /x \$eax</i>	Print contents of <code>%eax</code> in hex
<i>print /t \$eax</i>	Print contents of <code>%eax</code> in binary
<i>print 0x100</i>	Print decimal representation of <code>0x100</code>
<i>print /x 555</i>	Print hex representation of 555
<i>print /x (\$ebp+8)</i>	Print contents of <code>%ebp</code> plus 8 in hex
<i>print *(int *) 0xbffff890</i>	Print integer at address <code>0xbffff890</code>
<i>print *(int *) (\$ebp+8)</i>	Print integer at address <code>%ebp + 8</code>
<i>x/2w 0xbffff890</i>	Examine 2(4-byte) words starting at addr <code>0xbffff890</code>
<i>x/20b sum</i>	Examine first 20 bytes of function <code>sum</code>
<b>Useful information</b>	
<i>info frame</i>	Information about current stack frame
<i>info registers</i>	Values of all the registers
<i>help</i>	Get information about GDB

Figure 3.27: **Example GDB Commands.** These examples illustrate some of the ways GDB supports debugging of machine-level programs.

**Nº****Nome:****Turma:****Resolução dos exercícios****1. <sup>(A)</sup>Análise do código em *assembly***

```

1 int while_loop(int x, int y, int n)
2 {
3     while ((n > 0) & (y < n)) { /* Repare no uso do operador '&' */
4         x += n;
5         y *= n;
6         n--;
7     }
8     return x;
9 }

```

Código otimizado em *assembly*:

Registo	Variável	Atribuição inicial
	x	
	n	
	y	

Código C de um programa simples (*main*) que usa a função `while_loop`: