

Assembly do IA-32 em ambiente Linux

Trabalho para Casa: TPC4 Guião 3

Alberto José Proença

Metodologia

Este documento é simultaneamente o enunciado de TPC4 e o Guião3 de uma sessão laboratorial:

- o 1º exercício segue a estrutura típica de um **TPC**; i.e., é para ser integralmente resolvido em casa e a folha de resolução deverá ser entregue antes do início da aula;
- os restantes exercícios fazem parte do **Guião** da sessão laboratorial da semana de 3-Abr-06, e deverão ser estudados e preparados previamente a essa sessão; o texto do guião com alguns exercícios resolvidos deverá acompanhar a/o estudante na sessão laboratorial.

Temática

A lista de exercícios/trabalhos propostos em TPC4 / Guião3 analisa e complementa aspectos relacionados com o nível ISA do IA32, leccionados em aulas teóricas e referente ao conjunto de instruções para controlo do fluxo de execução, nomeadamente: **instruções de salto e suporte às estruturas de controlo em C** (ver sumários na página da disciplina na Web).

O texto anexo “**Introdução ao GDB debugger**”, contém informação pertinente ao funcionamento da sessão laboratorial, e é uma sinopse ultra-compacta do manual; a versão integral está disponível no site da GNU, e também em modo local (ver localizações no sumário desta aula).

Exercícios do TPC

Controlo do fluxo de execução de instruções

1. Nos seguintes excertos de programas desmontados do binário (*disassembled binary*), alguns ítems de informação foram substituídos por x's.

Notas: (i) no *assembly* da GNU, a especificação de um endereço em modo absoluto em hexadecimal contém o prefixo `*0x`, enquanto a especificação em modo relativo se faz em hexadecimal sem qualquer prefixo; (ii) não esquecer que o IA32 é *little endian*.

Responda às seguintes questões.

- a) ^(A) Qual o endereço destino especificado na instrução `jge`?

```
8048d1c: 7d 9e          jge XXXXXXXX
8048d1e: eb 24          jmp 8048d44
```

- b) ^(A) Qual o endereço em que se encontra o início da instrução `jmp`?

```
XXXXXXXX: eb 54          jmp 8047c42
XXXXXXXX: c7 45 f8 10    mov $0x10,0xffffffff8(%ebp)
```

- c) ^(R) Nesta alínea, o endereço da instrução de salto é especificado no modo relativo ao IP/PC, em 4 *bytes*, codificado em complemento para 2.

Qual o endereço especificado na instrução `jmp`?

```
8048902: e9 c2 10 00 00 jmp XXXXXXXX
8048907: 90          nop
```

- d) ^(R) Nesta alínea o código contém várias referências a endereços em instruções de salto,

cujos valores se encontram na gama $8043xxx_{16}$. Contudo, a sua codificação em binário segue regras distintas (absoluto/relativo, 1/4 bytes, ...). Calcule os endereços em falta para cada um dos 3 casos, e explicita a respectiva regra de codificação.

```

8043563: e9 XX XX XX XX      jmp 80436c1
8043568: 89 c2                mov %eax,%edx
804356a: 83 fa ff            cmp $0xffffffff,%edx
804356d: 74 XX                je 8043548
804356f: 89 d3                mov %edx,%ebx
8043571: ff 24 XX XX XX XX  jmp *0x8043580

```

MCC – Arquitectura de Computadores, 2004/05 TPC5

Nº	Nome:	Turma/Grupo/Nº:
----	-------	-----------------

Resolução dos exercícios

1. Controlo do fluxo de execução de instruções

- a) 8048d1c: 7d 9e jge XXXXXXXX _____
- b) XXXXXXXX: eb 54 jmp 8047c42 _____
- c) 8048902: e9 c2 10 00 00 jmp XXXXXXXX _____

Guião 3

A lista de exercícios que se segue é para ser parcialmente **feita em casa antes da sessão laboratorial** e estes estarão assinalados com uma caixa cinza (alguns poderão/deverão ser resolvidos após a sessão laboratorial, mas terão tal indicação).

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) ^(A) Considerando que os argumentos x , y , e n , passados para a função, se encontram respectivamente 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:

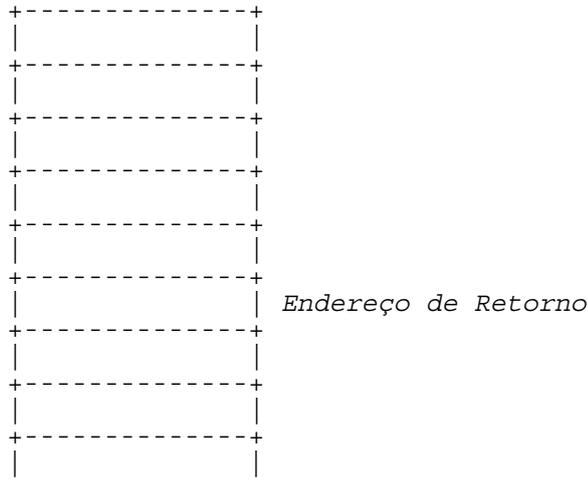
- i. ^(A) Construa em C um programa simples (`main`) que use a função `while_loop`, e que não faça mais do que inicializar numericamente um conjunto de valores que irá depois passar como argumento para a função.
(Sugestão: use variáveis com designações diferentes das usadas na função)

- ii. (A) Complete o ficheiro `while_loop.c` com o programa `main` que elaborou, e crie um executável usando o comando `gcc -Wall -O2 -g`.
- iii. (A) 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; escreva aqui essas instruções em *assembly* e respectiva localização:
- iv. (A) Invocando o *debugger* (com `gdb <nome_fich_executável>`), **insira pontos de paragem** (*breakpoints*) imediatamente antes da execução dessas instruções; explicita aqui os comandos usados (e registre o nº de *breakpoint* atribuído a cada endereço):
- v. (A) Valide o conjunto de atribuições aos registos, **preenchendo esta tabela sem executar qualquer código** (apenas com base na análise do código). 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`; de notar que o `gdb` apenas aceita especificação de registos de 32 bits).

Registo	Variável	Break1	Break2	Break3	Break4	Break5
	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 deve fazer depois para confirmar esse valor.

- d) ^(A/R) Considerando que a *stack* cresce para cima, pretende-se construir o diagrama da *stack frame* da função `while_loop` com o máx de indicações (endereços e conteúdos; pode ser em hexadecimal), logo após a execução da instrução antes do 2º *breakpoint*. Comente cada um dos conteúdos da *stack frame* (por ex., "endereço de retorno").
Construa assim esse diagrama: (i) estimando os valores antes da execução do código, e (ii) confirmando posteriormente esses valores, usando o depurador durante a execução do código.



- e) ^(A/R) 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) ^(R) 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)

Ciclo For

2. No *site* da disciplina encontra-se disponível um ficheiro executável – mais concretamente em <http://gec.di.uminho.pt/mcc/ac0506/Labs/m-contaN>; faça o seu *download* para a sua máquina. O ficheiro contém um programa que calcula o somatório dos dígitos numa cadeia de caracteres, a partir de uma dada posição (tirando partido do facto de que o valor em hexadecimal do código ASCII do símbolo "0" é 0x30). Este ficheiro foi obtido após a execução do comando `gcc -Wall -O2 -I. contaN.c m-contaN.c -o m-contaN`, a partir da consola de um sistema Linux. Contudo, após a execução desse comando, o ficheiro `contaN.c` ficou danificado... O ficheiro `m-contaN.c` contém o seguinte:

```
#include <stdio.h>
#include <stdlib.h>

int contaN(char *s, int c);

int main()
{
    char cadeia[50];
    int c;

    printf("Introduza a cadeia de caracteres -->\n");
    scanf("%s",cadeia );
    printf("Qual a posição inicial na cadeia de caracteres -->\n");
    scanf("%d",&c );
    printf("O somatório dos digitos na cadeia é -->%d\n",contaN(cadeia,c));
    exit(0);
}
```

Por outro lado, desconfia-se que a estrutura da função que estava em `contaN.c` era do tipo:

```
int i;
int result;
???
for ( ??? ; s[i] != ??? ; ???)
    if (s[i] >= '0' && ??? )
        result += ??? ;
return result;
```

- a) ^(A) Teste o funcionamento do programa a partir da consola usando como entrada de dados uma cadeia de caracteres contendo alguns algarismos em decimal (ex.: "1239aaswe67899") e um inteiro para a posição inicial na cadeia de caracteres.
- b) ^(R) Execute de novo o mesmo programa através do `gdb`. Use os comandos disponíveis para examinar código, de forma a visualizar o código simbólico ("desmontado" ou *disassembled*) correspondente à função (e apenas este). Escreva aqui o que obteve:

c) ^(R) Anote cuidadosamente o código visualizado na alínea anterior tendo em consideração que o resultado da função é devolvido no registo `%eax`.

Identifique no código:

- os registos que são atribuídos às variáveis locais `result` (_____) e `i` (_____)
- os registos que são usados com os argumentos da função _____
- a condição de teste do ciclo `for` _____
- o modo como a variável `i` é actualizada _____
- o código decimal correspondentes aos dígitos representados em *ASCII* _____
- a expressão em C que actualiza o valor de `result` no ciclo _____

d) ^(R) Com base no resultado das alíneas anteriores, recupere o ficheiro `contaN.c`.
(Para fazer depois da sessão laboratorial)

```
int i;
int result ;

for (      ; s[i] !=      ;      )
    if (s[i] >= '0' &&      )
        result +=      ;
return result;
```

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 IA32.

Command	Effect
Starting and Stopping	
<i>quit</i>	Exit GDB
<i>run</i>	Run your program (give command line argum. here)
<i>kill</i>	Stop your program
Breakpoints	
<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
Execution	
<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
Examining code	
<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
Examining data	
<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 <code>555</code>
<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>
Useful information	
<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.