Assembly do IA-32 em ambiente Linux

Trabalho para Casa: **TPC5**

Alberto José Proença

Prazos

Entrega **impreterível**: semana de 04-Nov-02, na hora e local da sessão Teórico-Prática.

Não serão aceites trabalhos entregues depois deste prazo.

Metodologia

O material a entregar nesta semana é similar ao das semanas que se seguirão, e constará essencialmente da(s) folha(s) anexa(s) ao enunciado, preenchidas à mão (manuscrito), excepto quando indicado em contrário. A validação do conteúdo será feita nas sessões teórico-práticas da semana em que forem entregues os trabalhos.

Introdução

A lista de exercícios que se apresenta segue directamente o material apresentado nas aulas teóricas da semana 6 (ver sumários na página da disciplina na Web), requerendo os conceitos básicos adquiridos em aulas anteriores. O texto anexo "Introdução ao GDB *debugger*" contém informação pertinente ao funcionamento da sessão laboratorial.

Exercícios

Controlo do fluxo de execução de instruções

1. Nos seguintes excertos de programas desmontados do binário (*disassembled binary*), alguns ítens de informação foram substituidos por X's. Responda às seguintes questões.

a) Qual o endereço destino especificado na instrução `jbe`?

```
8048d1c: 76 da              jbe XXXXXXX
8048d1e: eb 24              jmp 8048d44
```

b) Qual o endereço em que se encontra o início da instrução mov?

```
XXXXXXX: eb 54              jmp 8048d44
XXXXXXX: c7 45 f8 10 00     mov $0x10,0xfffffff8(%ebp)
```

c) Nesta alínea, o endereço da instrução de salto é especificado no modo relativo ao IP/PC, em 4 bytes, codificado em complemento para 2. Os *bytes* são apresentados do menos para o mais significativo, de acordo com o modelo de representação *little endian*, característico do IA32. Qual o endereço especificado na instrução `jmp`?

```
8048902: e9 cb 00 00 00     jmp XXXXXXX
8048907: 90                 nop
```

d) Explique a relação que existe entre a anotação à direita, e a codificação dos *bytes* à esquerda. Ambas linhas fazem parte da codificação da instrução `jmp`?

```
80483f0: ff 25 e0 a2 04     jmp *0x804a2e0
80483f5: 08
```

Ciclo *Do-While*

2. Considere a seguinte função em C:

```
1 int dw_loop(int x, int y, int n)
2 {
3   do {
4    x += n;
5   y *= n;
6   n--;
7   } while ((n > 0) & (y < n)); /* Note use of bitwise '&' */
8   return x;
9 }
```

a) Mostre o código em *assembly* que seria gerado pelo comando

`gcc –O2 –S` *`file_name`*`.c`

b) 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).

c) Identifique a expressão de teste e o corpo da função (*body-statement*) no bloco do código C, e assinale as linhas de código no programa em *assembly* que lhe são correspondentes.

d) Acrescente comentários ao programa em *assembly* (idênticos ao feito nos exemplos apresentados nos acetatos das aulas).

Ciclo *While*

3. Considere a seguinte função em C:

```
1 int loop_while(int a, int b)
2 {
3   int i = 0;
4   int result = a;
5   while (i < 256) {
6    result += a;
7    a -= b;
8    i += b;
9   }
10  return result;
11 }
```

a) Mostre o código em *assembly* que seria gerado pelo comando

`gcc –O2 –S` *`file_name`*`.c`

b) Considerando que os argumentos `a,` e `b,` passados para a função, se encontram respectivamente à distância 8 e 12 do endereço especificado em `%ebp`, preencha a tabela de utilização de registos (semelhante ao exemplo da série Fibonacci).

c) Identifique a expressão de teste e o corpo da função (*body-statement*) no bloco do código C, e assinale as linhas de código no programa em *assembly* que lhe são correspondentes. Que optimizações foram feitas pelo compilador?

d) Acrescente comentários ao programa em *assembly* (idênticos ao feito nos exemplos apresentados nos acetatos das aulas).

e) 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).

Ciclo *For*

4. A seguinte porção de código *assembly*:

```
Initially x, y, and n are offsets 8, 12, and 16 from %ebp
1     movl    8(%ebp),%ebx
2     movl    16(%ebp),%edx
3     xorl    %eax,%eax
4     decl    %edx
5     js      .L4
6     movl    %ebx,%ecx
7     imull   12(%ebp),%ecx
8     .p2align 4,,7                    Inserted to optimize cache performance
9   .L6:
10    addl    %ecx,%eax
11    subl    %ebx,%edx
12    jns     .L6
13  .L4:
```

foi obtido pela compilação de código C que tinha a seguinte estrutura:

```
1 int loop(int x, int y, int n)
2 {
3   int result = 0;
4   int i;
5   for (i = ____; i ____ ; i = ___ ) {
6      result += _____ ;
7   }
8   return result;
9 }
```

Pretende-se completar o programa em C de modo a obter-se um programa equivalente ao obtido com o código *assembly*. De notar que o resultado da função é devolvido no registo `%eax`.

Para resolver este problema, sugere-se que

- arranje alguma "inspiração" para tentar acertar com os registos apropriados; deverá depois confirmar se a escolha efectuada faz sentido ou não;
- identifique/caracterize:
  - os registos que são alocados às variáveis `result` e `i`
  - o valor inicial de `i`
  - a condição a testar com `i`
  - o modo como a variável `i` é actualizada
  - a expressão em C que descreva o modo como a variável `result` é incrementada no ciclo.

a) O compilador retirou a expressão que incrementa a variável `result` do interior do ciclo. Porquê?

b) Complete as partes que faltam no código C.

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	
quit	Exit GDB
run	Run your program (give command line argum. here)
kill	Stop your program
Breakpoints	
break sum	Set breakpoint at entry to function sum
*break *0x80483c3*	Set breakpoint at address 0x80483c3
disable 3	Disable breakpoint 3
enable 2	Enable breakpoint 2
clear sum	Clear any breakpoint at entry to function sum
delete 1	Delete breakpoint 1
delete	Delete all breakpoints
Execution	
stepi	Execute one instruction
stepi 4	Execute four instructions
nexti	Like stepi, but proceed through function calls
continue	Resume execution
finish	Run until current function returns
Examining code	
disas	Disassemble current function
disas sum	Disassemble function sum
disas 0x80483b7	Disassemble function around address 0x80483b7
disas 0x80483b7 0x80483c7	Disassemble code within specified address range
print /x $eip	Print program counter in hex
Examining data	
print $eax	Print contents of %eax in decimal
print /x $eax	Print contents of %eax in hex
print /t $eax	Print contents of %eax in binary
print 0x100	Print decimal representation of 0x100
print /x 555	Print hex representation of 555
print /x ($ebp+8)	Print contents of %ebp plus 8 in hex
*print *(int *) 0xbffff890*	Print integer at address 0xbffff890
*print *(int *) ($ebp+8)*	Print integer at address %ebp + 8
x/2w 0xbffff890	Examine 2(4-byte) words starting at addr 0xbffff890
x/20b sum	Examine first 20 bytes of function sum
Useful information	
info frame	Information about current stack frame
info registers	Values of all the registers
help	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:** Seg1 - Seg2 - Ter

Resolução dos exercícios

1. Controlo do fluxo de execução de instruções

a) `8048d1c: 76 da jbe XXXXXXX` _____

b) `XXXXXXX: c7 45 f8 10 00  mov…`_____

c) `8048902: e9 cb 00 00 00  jmp XXXXXXX` _____

d)

2. Ciclo *Do-While*

Registo	Variável	Atribuição inicial
	x	
	n	
	y	

```
1 int dw_loop(int x, int y, int n)
2 {
3   do {
4    x += n;
5   y *= n;
6   n--;
7   } while ((n > 0) & (y < n)); /* Note use of bitwise '&' */
8   return x;
9 }
```

Nº	**Nome:**	**Turma:** Seg1 - Seg2 - Ter

3. Ciclo *While*

Registo	Variável	Atribuição inicial
	a	
	b	
	i	

```
1 int loop_while(int a, int b)
2 {
3     int i = 0;
4     int result = a;
5     while (i < 256) {
6        result += a;
7        a -= b;
8        i += b;
9     }
10    return result;
11 }
```

4. Ciclo *For*

a) Alocação dos registos às variáveis `result` (_____) e `i` (_____)

Valor inicial de `i` _____

Condição a testar com `i` _____

Modo de actualizar a variável `i` _____

Expressão que incrementa `result` _____

O compilador retirou a expressão que incrementa a variável `result` do interior do ciclo. Porquê?

b)

```
1 int loop(int x, int y, int n)
2 {
3   int result = 0;
4   int i;
5   for (i = ____; i ____ ; i = ___ ) {
6      result += _____ ;
7   }
8   return result;
9 }
```