Conceitos fundamentais e Software

Carlos A. Ruggiero e Gonzalo Travieso

Maio de 2012

1 Objetivo

O objetivo desta prática é demonstrar os conceitos fundamentais de arquitetura de computadores como apresentado nas aulas teóricas. Também, o aluno deverá verificar como um programa escrito em linguagem de alto nível é transformado em linguagem de máquina que o computador possa executar diretamente.

2 Conceitos fundamentais

2.1 Arquitetura e Linguagem Montadora

A arquitetura, como definida por Amdahl, é o conjunto das características de um computador como vistas por um programador em linguagem montadora (ou *assembly language,* em inglês). Assim, cada arquitetura tem associada uma linguagem montadora. O *modelo* de von Neumann, que é o nome dado ao conjunto de arquiteturas que seguem os princípios gerais sugeridos por von Neumann em 1945/1946 (ver capítulo 1 e figura 1.1 da apostila teórica), é o único estudado nesta disciplina e todos os exemplos aqui apresentados pertencem a esse modelo.

A linguagem montadora é uma representação, mais simples de ser entendida e utilizada pelo ser humano, da linguagem binária (seqüência de uns e zeros) que o computador consegue entender diretamente. Para se passar da linguagem montadora para a binária, utiliza-se um *tradutor* conhecido como montador (ou *assembler,* em inglês). O código objeto gerado pelo montador é finalmente tranformado em executável pelo linkeditor.

Para se entender o funcionamento de um arquitetura, é interessante utilizar-se de ainda outro programa, o depurador (ou *debugger,* em inglês). O GDB (*Gnu Project Debugger*) e o DDD (*Data Display Debugger*, que é um *frontend* para o GDB) serão utilizados nesta e nas práticas subsequentes. Os depurados permitem-nos visualizar a execução de um programa, com grande nível de detalhamento, incluindo execução instrução por instrução, exame do conteúdo de registradores e de posições de memória, etc, o que permite que o aluno tenha uma boa idéia do funcionamento da arquitetura.

2.2 A arquitetura x86

A empresa americana Intel foi uma das pioneiras no projeto, implementação e comercialização de microprocessadores monolíticos, ou seja, unidades centrais de processamento (UCPs) completas contidas em uma única pastilha de circuito integrado. O seu processador 8088 foi escolhido pela empresa IBM (na época a maior empresa de computação do mundo) para ser a base do IBM PC (*Personal Computer*) que fez enorme sucesso no início da década de 80 e popularizou a utilização de computadores pessoais. Na verdade, o 8088 era uma simplificação (barramento de 8 bits ao invés de 16) do 8086 que assim se tornaria um padrão de arquitetura seguido e utilizado até hoje em bilhões de computadores pelo mundo todo. Muitos sucessores do 8086 foram comercializados pela Intel, tais como o 80286, 80386, 80486, Pentium, Pentium 2, etc até os modernos core i5 e core i7 atuais. O nome genérico dado pela Intel a esta arquitetura tão bem sucedidadacomercialmente é **x86**.

A arquitetura do 8086 não é particularmente atraente, e, na verdade, nada tem de muito impressionante, mas a sua popularidade justifica seu estudo. O aluno não deve se preocupar em entender a linguagem montadora em detalhes, pois isso será visto melhor na disciplina Arquiteturas II. Deve porém, tentar verificar os conceitos das arquiteturas de Von Neumann expostos nas aulas teóricas.

2.2.1 Registradores

A figura a seguir apresenta os registradores de uso geral do 8086.

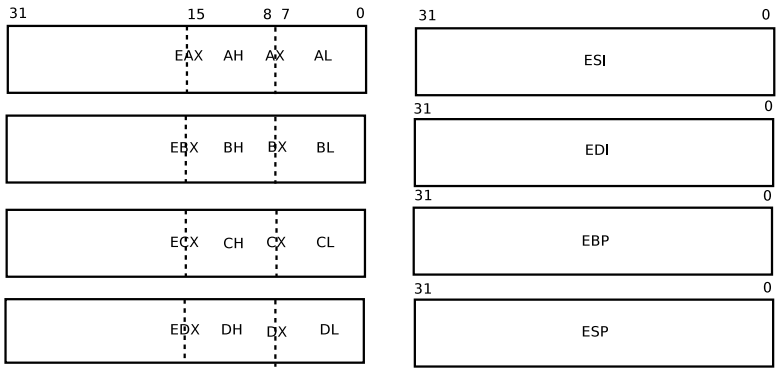

Figure 1: Registradores da arquitetura x86 (32 bits)

Os registradores apresentados são todos de 32 bits (registradores de 32 bits só foram introduzidos no 386). O registrador EAX pode ser acessado como 32 bits (EAX), 16 bits menos significativos (AX), 8 bits menos significativos (AL) e 8 bits entre o bit 8 e o 15 (AH).

2.2.2 Hello World e conceitos fundamentais

Considere o programa em linguagem montadora da arquitetura x86 a seguir:

```
section .data                           ;seção de dados

msg     db      "Oi gente!",0xa         ;cadeia de carateres  na memória
len     equ     $ - msg                 ;comprimento da cadeia

section .text                           ;seção de código

   global _start                        ;Entrada para o linkeditor

_start:

; Chamada do sistema (sys_write no Linux) para escrever uma cadeia de caracteres.

       mov     eax,4                    ;Qual chamada? sys_write.
       mov     ebx,1                    ;Onde escrever? - Primeiro argumento
       mov     ecx,msg                  ;Qual mensagem? - Segundo argumento
       mov     edx,len                  ;Taman. da mensagem? - Terc argumento
       int     0x80                     ;Vai!

;Chamada do sistema - acabar (sys_exit)
```

```
mov     eax,1                   ;Qual chamada? sys_exit.
mov     ebx,0                   ;Código de retorno - único argumento
int     0x80                    ;Vai!
```

Execute as seguintes operações em seu computador (depois do comando, sempre pressione o <**enter**>):

1. Em um terminal de linha de comando, e com um editor de texto, criar um arquivo chamado hello.s e digitar o programa em linguagem montadora mostrado.

2. Transformar o programa em código objeto através do montador: fazer **nasm -o hello.o -f elf hello.s**. Verifique a criação de um novo arquivo chamado **hello.o**.

3. Gerar o código executável chamando o linkeditor: **ld -o hello hello.o**

4. Executar o programa com **./hello**. (1) O que acontece?

5. Executar o programa dentro do depurador GDB. Fazer: **gdb hello**. Note que aparece a mensagem *no debugging symbols found* antes de aparecer o *prompt* do gdb. Nesta forma, a depuração é possível mas não é eficiente pois muitos recursos ficam indisponíveis. Para gerar o executável mantendo os símbolos é preciso montar e linkeditar com a opção **-g**

6. Montar e linkeditar com a opção **-g**. Fazer: **nasm -g -o hello.o -f elf hello.s** e depois: **ld -g -o hello hello.o**.

7. Confirme que o executável continua produzindo a mesma saída (ver ítem 4 acima).

8. Faça novamente: **gdb hello**. Note que agora, a mensagem de símbolos não achados não aparece.

9. Ainda dentro do gdb, faça: **list 1,25**. (2) O que acontece? Tente ajustar o tamanho da janela do terminal para ver todo o seu programa em uma única tela. Faça **help list** se quiser entender mais sobre como controlar as listagens.

10. Faça: **break 15** (ou, simplesmente **b 15**). Esse comando diz para o gdb parar quando passar pela linha 15 (a linha da instrução).

11. Execute o comando **run** para rodar o programa hello. (3) O que ocorre? (4) Anote o valor em hexadecimal que apareceu.

12. Faça: **stepi**. Se quiser mais informações, faça **help stepi**. (5) O que ocorre? (6) Anote o novo número em hexadecimal. (7)Você sabe o que é esse número?

13. (8) Tente descobrir porque o número anotado aumentou em 5 bytes.

14. Faça: **info reg**. (9) O que foi impresso no terminal? Não se esqueça, se precisar de ajuda faça **help info reg**.

15. (10) Tente explicar os valores associados a eax, ebx e eip ((11) o que você acha que é o eip?). (12) Assim o que faz a instrução **mov?** (13) Qual a direção da cópia na sintaxe desse montador (o nasm usa a sintaxe intel)? Em outros montadores, (por exemplo o Gnu Assembler ou gas, que usa a sintaxe AT&T) a direção é a oposta.

16. Faça **help x** e tente entender o que faz o comando. Depois faça: **x/5xb $pc**. O comando **x** mostra as posições de memória endereçadas pelo **$pc** que é outro nome para o eip. O gdb exige o uso do **$** diante dos nomes dos registradores. (14) Que números aparecem(anote)? (15) O que é isso?

17. Faça novamente **info reg**. Faça: **stepi**. Faça **info reg** novamente. (16) O que mudou do primeiro para o segundo **info reg**? (17) Você reconhece os números que você anotou no ítem anterior?

18. Repita o procedimento do ítem anterior. (18) E agora, o que mudou?

19. Faça novamente **stepi**. (19) O que ocorreu? A instrução que acaba de ser executada (*int 0x80*) executa uma chamada do sistema operacional requisitando a escrita na saída padrão. Reparar que a instrução *int* é uma instrução de desvio de fluxo (mais especificamente, uma instrução de chamada de subrotina). (20) Como os parâmetros são passados para o *kernel*?

20. Faça: **cont**. O programa continua sua execução até terminar.

21. Faça: **quit** para sair do gdb.

22. Alguns programadores preferem utilizar o **ddd** que é um *frontend* para o **gdb**.

23. Faça então: **ddd hello**. Uma janela deverá abrir exibindo o código fonte em linguagem montadora. Vá no menu *Status* e escolha a opção *registers*. Uma nova janela deverá ser aberta com a mensagem *The program has no registers now*. (21) O que quer dizer isso?

24. Do lado esquerdo da instrução *mov ebx,1* clique com o botão direito do mouse e escolha a opção *set breakpoint*. Note o surgimento de um sinal de *stop*. Clique em *run*. Note o aparecimento dos registradores na janela *registers*. Vá clicando no botão *stepi* e verifique a mudança dos valores dos registradores e de *eip*.

25. Saia do **ddd** e inicie novamente o **gdb** com **gdb hello**

26. Faça **b 18** e **run**. Examine os registradores. Agora, faça **set $edx= 6**. Execute **cont**. (22) O que ocorreu? (23) O que você acha que faz o **edx** neste programa?

27. Saia do **gdb** com **quit**.

28. Recompile o programa, agora com o comando **nasm -g -l hello.lis -o hello.o -f elf hello.s**. Examine, com um editor ou simplesmente com os comandos **more** ou **cat**, o arquivo hello.lis. Ele apresenta o código de máquina gerado (em hexadecimal, não em binário!) para o programa em linguagem montadora *hello.s*. O código de máquina está à esquerda do programa. Notar que todas as instruções **mov** apresentam 5 bytes enquanto a instrução *int* é de apenas 2 bytes.

29. Considere a primeira instrução, *mov eax,4*. O código gerado é B804000000. (24) O que é o o B8? (25) E o 04000000 não deveria ser 00000004? Poderia ser, e em muitas arquiteturas é assim, mas não na arquitetura x86. Leia o verbete *endianness* na Wikipedia se quiser entender melhor.

30. Note que o programa começa em 00000000 ((26) seriam os 8 zeros, as notas dos alunos?)

31. Execute novamente o linkeditor, com o comando **ld -g -M -o hello hello.o >hello.map**. Examine o arquivo *hello.map* e verifique o endereço da variável *_start*? (27) Este valor é compatível com aqueles anotados nos itens 11 e 12?

32. Crie um programa *hello2.s* igual a *hello.s* e adicione a instrução **mov [_start + 0x10], byte 0x35** logo após a instrução **mov edx,len** sem modificar mais nada. Execute o nasm e o ld como anteriormente mudando o *hello* por hello2. Execute **./**hello2**.** (28) O que ocorre? (29) Crie agora outro programa *hello3.s*, igual a hello2.s, somente alterando o 0x10 da instrução extra por 0x102c (a instrução fica **mov [_start + 0x102c], byte 0x35**). Repita o procedimento feito com *hello2.s*. (30) E agora, o que aconteceu? (31) Explique a causa de comportamentos tão diferentes com a mudança de apenas um número.

Depois de executar esses 32 passos, o aluno deverá ter fixado vários conceitos fundamentais em arquitetura de computadores, quais sejam:

1. Para se entender como um programa é efetivamente executado em um computador, deve-se entender a sua *arquitetura*. A linguagem montadora é a ferramenta para se atingir esse objetivo.

2. O conjunto de arquiteturas de von Neumann, chamado de *modelo de von Neumann* (o conceito de *modelo* será melhor estudado em Arquiteturas II) apresenta muitas similaridades entre si (só vimos a arquitetura x86 mas outras serão vistas posteriormente).

3. A execução de um programa em uma arquitetura de von Neumann é fortemente controlada por um registrador fundamental que é o *contador de programa* ou *program counter*, na arquitetura x86 também conhecido como *instruction pointer*. Ele sempre aponta para a próxima instrução a ser executada. Será visto em disciplinas futuras que isso pode ser uma dificuldade considerável para se estender o modelo para execução paralela.

4. O funcionamento de uma instrução neste modelo é muito parecido com aquele proposto por von Neumann na década de 40, e não mudou praticamente nada. Para recordar, o aluno deve estudar as páginas 8 e 9 da apostila teórica.

5. Uma memória apenas existe neste modelo e tudo está armazenado lá: instruções, dados, pilha, espaços de alocação dinâmica, etc. Todos os tipos de dados se confundem, sendo representados por números binários: números inteiros com e sem sinal, números em ponto flutuante, caracteres, vetores, estruturas mais complexas, códigos de instruções, etc.

6. O acesso à memória é intenso, sendo que uma única instrução pode consultar a memória muitas vezes. Essa alta comunicação entre unidade central de processamento (UCP) e memória, que pode limitar o desempenho do computador, ficou conhecida como o **gargalo de von Neumann** e muito esforço tem sido feito para mitigá-lo.

2.3 A arquitetura MIPS

A arquitetura MIPS (Multiprocessor without Interlocked Pipelined Stages), juntamente com a arquitetura SPARC, foi uma das mais importantes da filosofia RISC, que ganhou grande importância a partir da década de 80. O MIPS surge como o projeto de um grupo de pesquisadores liderados por John Hennessy na Universidade de Stanford (John Hennessy é hoje o reitor daquela universidade). Hennessy, juntamente com David Paterson de Berkeley escreveu dois livros muito influentes na área de arquitetura de computadores.

2.3.1 Registradores

A versão de 32 bits (existe o MIPS64 de 64 bits) é composta por 32 registradores de 32 bits cada. Na linguagme montadora, eles podem ser referenciados por um $ seguido do número do registrador (de $0 a $31). Alguns desses registradores recebem nomes especiais, nomes estes que também podem ser usados no programa. Assim, o registrador 0 pode ser chamado de $zero pois contém o valor 0 fixo (ele só pode assumir este valor). O registrador $1 é o $at de *assembler temporary*. Os registradores $2 e $3 são também $v0 e $v1 (v de *value*), os $4 a $7 são também $a0 a $a3 e são utilizados na passagem de parâmetros, os de $8 a $15 são $t0 a $t7 (t de temporários), de $16 a $23 são $s0 a $s7 (s de salvos), $24 e $25 são mais dois temporários $t8 e $t9, $26 e $27 são $k0 e $k1, $28 é o $gp (*global pointer*), $29 é o $sp (*stack pointer*), $30 é o $fp (*frame pointer*) e $31 é o $ra (*return address).*

Somente as intruções de *load* e *store* operam com dados da memória. Todas as outras instruções operam com os registradores.

2.3.2 Hello World

Considere o programa a seguir:

```
    .text
    .globl __start
__start:

    .set noreorder
```

```
    .cpload $gp       # setup the pointer to global data
    .set reorder
                      # print sth. via sys_write
    li $a0, 1         # print to standard ouput
    la $a1, stradr    # set the string address
    lw $a2, strlen    # set the string length
    li $v0, 4004      # index of sys_write:
                      # __NR_write in /usr/include/asm/unistd.h
    syscall           # causes a system call trap.

                      # exit via sys_exit
    move $a0, $0      # exit status as 0
    li $v0, 4001      # index of sys_exit
                      # __NR_exit in /usr/include/asm/unistd.h
    syscall

   .rdata
stradr: .asciiz "hello, world!\n"
strlen: .word . - stradr  # current address - the string address
# end
```

Não temos disponível um computador que use uma UCP MIPS para que os alunos possam executar programas para essa arquitetura. Atualmente porém, existem bons emuladores para a maioria de arquiteturas que tiveram sucesso no mercado. Um desses emuladores é o QEMU, que executa em diversos sistemas operacionais rodando na arquitetura x86/x86-64. Ele é capaz de emular várias arquiteturas RISC tais como MIPS, SPARC, PowerPC, etc. O emulador é completo o suficiente para ser capaz de executar um sistema operacional como o Debian GNU/Linux dentro da máquina emulada. O computador hospedeiro (onde é executado o QEMU) é normalmente chamado de *host* enquanto a máquina emulada é o *guest.* Nesta seção, tanto o *host* quanto o *guest* executarão o debian GNU/Linux.

1. Coloque o disco da máquina virtual em um diretório conveniente. O arquivo se chama: *debian_squeeze_mips_standard.qcow2*. Coloque, no mesmo diretório o *kernel vmlinux-2.6.32-5-4kc-malta,* bem como o *script* de inicialização *run_mips_standard*. Verifique o conteúdo do *script* de inicialização. O emulador é chamado a emular uma máquina com processador MIPS e com 256 MB de memória. O disco *deb...qcow2* já contém uma distribuição instalada com os programas necessários para essa e para práticas futuras.

2. Execute o *script* com *sh run_mips_standard*. Você deverá ver uma nova janela se abrir e o sistema emulado começar a carregar o sistema operacional. Aguarde até que o sistema tenha carregado completamente e seja oferecido um *login*. Digite *user* como usuário e novamente *user* como senha (*password*).

3. Dentro do *guest* e com um editor de textos (*vi* e *emacs* estão disponíveis), crie um arquivo chamado *hello.s* com o programa acima. Monte o programa com **as -o hello.o hello.s**. (32) Esse montador é o mesmo nasm, ou gas, usado na seção anterior com a arquitetura x86? Chame o linkeditor com **ld -o hello hello.o**. Execute o programa executável gerado com **./hello**. (33) O que ocorre?

4. Execute: **as -al -o hello.o hello.s >hello.lis.** Examine o arquivo gerado *hello.lis* e responda: (34) Qual o tamanho mais comum para o comprimento da instrução? Compare o tamanho das instruções no **MIPS** e no **x86**. (35) O que você pode dizer? Execute a linkedição com **ld -g -M -o hello hello.o >hello.map**. Examine o arquivo *hello.map* e diga: (36) Qual o endereço de início do programa? (37) Qual o espaço de endereçamento?

5. Colocar simplesmente **-g** na linha de comando não gera os símbolos necessários para uma execução confortável do depurador. Assim, execute novamente a montagem e a linkedição,

agora porém com o *gcc*: **gcc -g -o hello hello.s**. Substitua antes, porém, o símbolo *__start* por *main*.

6. Execute **gdb hello**. No *prompt* do gdb, faça **list 1,25** para listar o programa. Faça **b 10** para criar um *breakpoint* na linha da instrução *la $a1,stradr*. Faça **run**, para executar o programa até o *break*. Examine os registradores ((38) como?). Execute instrução por instrução com **step** (não use *stepi* como anterioremente) e vá observando os valores dos registradores. Para entender melhor, considere que *li* é *load immediate*, *lw* é *load word* e *la* é *load address*.

7. Tente executar **ddd -display 10.0.2.2:0.0 hello**. Depois de um tempo, você deverá ver o depurador *ddd* abrindo uma nova janela no *host.* Crie um *breakpoint* na linha 10, e vá, como no passo anterior, verificando os registradores a cada instrução. Não se esqueça de abrir uma janela com os registradores.

3 Software

Existe uma relação forte entre as várias camadas de software, principalmente as de nível mais baixo com a arquitetura dos computadores.

O objetivo desta seção é mostrar, apesar de que, superficialmente, a relação existente entre um programa escrito em uma linguagem de alto nível, o sistema operacional utilizado no computador e a sua arquitetura. Alunos iniciantes em arquitetura tem grande dificuldade nessa área, já que a maioria foi treinada nas linguagens de alto nível que tendem a esconder (e essa é uma qualidade dessas linguagens!) os detalhes da arquitetura, organização e implementação do computador. Isso leva alguns a verem o funcionamento do computador como algo quase mágico, intransponível para os não iniciados.

Considere o programa simples abaixo, o conhecido *hello world* já tratado por nós mas agora escrito em ANSI C.

```
#include <stdio.h>

int main(void)
{
printf("Oi Tchurma\n");
return 0;
}
```

Com relação a este programa faça:

1. Utilizando um editor de texto, digite esse programa em um arquivo *hello.c*.

2. Compile o programa com **cc -o hello hello.c** e o execute com **./hello**. (39) O que acontece?

3. O compilador disponível nos computadores do laboratório é o Gnu C Compiler, mais conhecido como *gcc*. Compile agora o programa com: **gcc -v -o hello hello.c**. (40) Qual a saída? Todos os comandos chamados, com informações extras importantes são apresentados, o que pode ser um pouco assustador para o iniciante.

4. Faça: **gcc -S hello.c**. Repare que é criado um arquivo *hello.s*. Examine esse arquivo. A sintaxe do programa montador gerado é diferente da vista anteriormente; aqui utiliza-se do formato AT&T. (41) O que você pode verificar aqui de diferente? Repare na nova instrução *call*: ela chama um rótulo que não é definido no programa: **puts.** (42) O que é isso?

5. Faça: **as -a -o hello.o hello.s >hello.lis**. Note como o programa compila normalmente, o que mostra que o código gerado pelo compilador C é realmente válido. Note a utilização de um novo programa montador: o *Gnu Assembler* ou *gas*, ou simplesmente *as*. Examine o arquivo hello.lis. (43) Existem símbolos não definidos?

6. Tente agora o linkeditor: **ld -o hello hello.o**. (44) O que ocorre? (45) E se você fizer: **ld -o hello hello.o -lc**? Quando se adiciona o *-l<alguma_coisa>* o linkeditor é instruído a procurar nos locais normais de bibliotecas pelo arquivo *libalguma_coisa.a*. Como foi passado *-lc* o arquivo procurado e "linkado" será o *libc.a* que é a biblioteca padrão da linguagem C. Execute: **ar -tv /usr/lib/libc.a** que mostra todos os módulos contidos em *libc.a*. (46) Qual é a função do linkeditor?

7. No programa em linguagem montadora *hello.s* não se vê chamada para o sistema operacional. Note que, nos programas em linguagem montadora vistos anteriormente, a instrução *int 0x80* fazia esse papel, mas ela não está presente aqui. (47) Será que os códigos gerados pela linguagem C não precisam do *kernel* para exibir mensagens na tela?

8. O programa executável gerado em 6 não roda corretamente. Isso ocorre porque o processo de geração de código da linguagem C é um pouco mais elaborado. Faça então: **gcc -o hello hello.o**. Este novo executável roda corretamente. Na verdade, é possível usar o *script* gcc para montar e linkar um programa, por exemplo com: **gcc -o hello hello.s**. Será que o *gcc* só funciona porque o arquivo original era em C e, de alguma forma, o programa *hello.s* gerado contém algum truque que ativa o compilador C por "baixo dos panos"? A resposta é não e para ver isso considere o programa abaixo:

```
.text
.globl main
main:
    pushl %ebp
    movl %esp, %ebp
    andl $-16, %esp
    subl $16, %esp
    movl $l1, (%esp)
    call puts
    movl $0, %eax
    leave
    ret
l1: .string "Oi Tchurma"
```

Este programa é trivialmente derivado do código em linguagem montadora gerado pelo compilador C para o *hello.c*. Várias linhas foram eliminadas, sendo mantido apenas o rótulo *main* que é o ponto de entrada de um programa em C. Inclusive, eliminamos a seção *.rodata* ((48) o que era isso?) e colocamos tudo na seção de código ((49) podemos fazer isso, ou seja, colocar os dados na seção de código?). A montagem e linkedição deste programa com o *gcc* gera um programa correto.

Consideremos agora um programa um pouco mais interessante (*pero no mucho*!). Este programa soma os dez digitos (de 0 a 9) de forma um tanto ingênua, colocando os valores em um vetor chamado *x*. Assim:

```
#include <stdio.h>
int sum, i, x[10];

int  main(void) {
  for (i=0; i<10; i++) x[i]=i;
  sum = 0;
  for (i=0; i<10; i++) sum = sum + x[i];
  printf("O resultado da soma é: %d\n", sum);
}
```

Com esse programa em um arquivo *somvec.c* faça:

1. Compile o código com **gcc -g -o somvec -Wa,-a,-ad somvec.c >somvec.lis** e examine o código em linguagem montadora produzido, juntamente com o código original em C no

arquivo *somvec.lis*. (50) O que é a instrução *jle .L5* e qual sua função no código? (51) Por que a instrução anterior a esta, em linguagem montadora, é *cmpl $9,%eax*?

2. Execute **./somvec**. (52) O que é impresso? Faça **nm somvec** que mostra todos os símbolos do programa com seus respectivos endereços. (53) Quais os endereços de *i, sum e x*?

3. Considere agora que o programador cometou um engano e, ao invés de *i<10*, ele digitou *i<=10*. (54) Qual a saída agora? (55) Está correta? (56) Por que a saída foi essa?