PC Assembly Language

Paul A. Carter

24 de maio de 2004

Capítulo 1

Introdução

1.1 Sistemas Numéricos

A Memória em um computador consiste de números. A memória de um computador não armazena esses números na base decimal (base 10). Devido ao fato de simplificar o hardware, computadores armazenam todas as informações no formato binário (base 2). Primeiramente vamos relembrar o sistema decimal.

1.1.1 Decimal

Números de base 10 são compostos de 10 dígitos possíveis (0-9). Cada dígito de um número tem uma potência de 10 associada com ele, baseada na sua posição. Por exemplo:

$$234 = 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

1.1.2 Binário

Números de base 2 são compostos de 2 dígitos possíveis (0 e 1). Cada dígito de um número tem uma potência de 2 associada com ele, baseada na sua posição. (Um dígito binário é chamado de bit.) Por exemplo:

$$\begin{aligned} 11001_2 &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 16 + 8 + 1 \\ &= 25 \end{aligned}$$

Isso mostra como um número binário pode ser convertido em um número decimal. A tabela 1.1 mostra como os números são representados no formato binário.

A Figura 1.1 Mostra como dígitos binários individuais (*i.e.*, bits) são somados. Aqui há um exemplo:

Decimal	Binário		Decimal	Binário
0	0000		8	1000
1	0001		9	1001
2	0010		10	1010
3	0011		11	1011
4	0100		12	1100
5	0101		13	1101
6	0110		14	1110
7	0111		15	1111

Tabela 1.1: Números de 0 a 15 em binário

Sem pré-deslocamento				pré-Deslocamento			
$\begin{array}{r} 0 \\ +0 \\ \hline 0 \end{array}$	$\begin{array}{r} 0 \\ +1 \\ \hline 1 \end{array}$	$\begin{array}{r} 1 \\ +0 \\ \hline 1 \end{array}$	$\begin{array}{r} 1 \\ +1 \\ \hline 0 \end{array}$	$\begin{array}{r} 0 \\ +0 \\ \hline 1 \end{array}$	$\begin{array}{r} 0 \\ +1 \\ \hline 0 \end{array}$	$\begin{array}{r} 1 \\ +0 \\ \hline 0 \end{array}$	$\begin{array}{r} 1 \\ +1 \\ \hline 1 \end{array}$
			d		d	d	d

Figura 1.1: Adição binária (d significa *deslocamento*)

$$\begin{array}{r} 11011_2 \\ +10001_2 \\ \hline 101100_2 \end{array}$$

Considerando a seguinte divisão decimal:

$$1234 \div 10 = 123\ r\ 4$$

observa-se que essa divisão remove o dígito mais a direita e move os outros dígitos decimais uma posição para a direita. Dividir por dois realiza uma operação semelhante, mas para números no sistema binário. Considere a seguinte divisão binária[1]:

$$1101_2 \div 10_2 = 110_2\ r\ 1$$

Esse fator pode ser usado para converter um número decimal para o número equivalente em binário como mostra a figura 1.2. Esse método encontra o dígito mais a direita que é chamado de *bit menos significativo* (lsb, em inglês). O dígito mais a esquerda é chamado de *bit mais significativo* (msb, em inglês). A unidade básica de memória é composta por 8 bits e é chamada de *byte*.

[1] A notação 2 é usada para indicar que o número é representado em binário, não em decimal

Decimal	Binary
$25 \div 2 = 12\ r\ 1$	$11001 \div 10 = 1100\ r\ 1$
$12 \div 2 = 6\ r\ 0$	$1100 \div 10 = 110\ r\ 0$
$6 \div 2 = 3\ r\ 0$	$110 \div 10 = 11\ r\ 0$
$3 \div 2 = 1\ r\ 1$	$11 \div 10 = 1\ r\ 1$
$1 \div 2 = 0\ r\ 1$	$1 \div 10 = 0\ r\ 1$

Assim $25_{10} = 11001_2$

Figura 1.2: Conversão Decimal

1.1.3 Hexadecimal

Números hexadecimais usam a base 16. Números Hexadecimais podem ser usados para abreviar números binários. Hexadecimais tem 16 dígitos possíveis. Isso é um problema tendo em vista que não existem símbolos para representar esses dígitos adicionais que vem depois do 9. Por convenção, letras são usadas para representar esses dígitos extras. Os 16 dígitos hexadecimais são 0-9 e depois A, B, C, D, E e F. O dígito A corresponde ao número 10 decimal, B ao 11, etc. Cada dígito de um número hexadecimal tem uma potência de 16 associada com ele. Exemplo:

$$\begin{aligned} 2\mathrm{BD}_{16} &= 2 \times 16^2 + 11 \times 16^1 + 13 \times 16^0 \\ &= 512 + 176 + 13 \\ &= 701 \end{aligned}$$

Para converter de decimal para hexadecimal, use a mesma idéia que foi usada na conversão binária, porém divida por 16. Veja o exemplo na figura 1.3.

A razão pela qual o sistema hexadecimal é útil é que a conversão entre hexadecimais e binários é simples. Números binários ficam grandes rapidamente. Hexadecimais oferecem uma maneira muito mais compacta de de representar números binários.

Para converter um número hexadecimal em binário, simplesmente converta cada dígito hexadecimal em um número binário de 4 dígitos. Por exemplo, $24\mathrm{D}_{16}$ é convertido para $0010\ 0100\ 1101_2$. Note que os zeros precedentes são importantes! Se o zero precedente não for usado, o resultado estará errado. Converter de binário para hexadecimal também é fácil. O processo é feito de maneira inversa. Converta cada segmento de 4-bits de

$$\begin{aligned} 589 \div 16 &= 36\ r\ 13 \\ 36 \div 16 &= 2\ r\ 4 \\ 2 \div 16 &= 0\ r\ 2 \end{aligned}$$

Assim $589 = 24D_{16}$

Figura 1.3:

um binário para hexadecimal. Comece pela direita, não pela esquerda. Isso garante que o procedimento usa os segmentos de 4-bits corretos[2]. Exemplo:

110	0000	0101	1010	0111	1110_2
6	0	5	A	7	E_{16}

Um número de 4-bits é chamado de *nibble*. Assim, cada dígito hexadecimal corresponde a um nibble. Dois nibbles compõem um byte que pode ser representado por num número hex de 2 dígitos. A faixa de valores de um byte vai de 0 até 11111111 em binário, 0 até FF em hexadecimal e de 0 até 255 em decimal.

1.2 A organização de um computador

1.2.1 Memória

A memória é medida em unidades de kilobytes ($2^{10} = 1,024$ bytes), megabytes ($2^{20} = 1,048,576$ bytes) e gigabytes ($2^{30} = 1,073,741,824$ bytes).

A unidade básica de memória é o byte. Um computador com 32 megabytes de memória pode armazenar cerca de 32 milhões de bytes de informação. Cada byte na memória é rotulado por um número único conhecido como seu endereço como mostra a figura 1.4.

Endereço	0	1	2	3	4	5	6	7
Memória	2A	45	B8	20	8F	CD	12	2E

Figura 1.4: Endereços de memória

Geralmente a memória é usada em quantidades bem maiores que um simples byte. Na arquitetura de um PC, nomes foram dados a essas seções de memória maiores como mostra a figura 1.2.

[2]Se não estiver claro porque o ponto de início faz diferença, tente converter o exemplo começando da esquerda.

word	2 bytes
double word	4 bytes
quad word	8 bytes
paragraph	16 bytes

Tabela 1.2: Unidades de memória

Todos os dados em memória são numéricos. Caracteres são armazenados usando um *código de caracteres* que associa números a caracteres. Um dos códigos de caracteres mais comum é conhecido como *ASCII* (American Standard Code for Information Interchange). Um novo e mais completo código que suplementa o ASCII é o Unicode. A principal diferença entre esses dois códigos é que o primeiro usa um byte para codificar um caractere e o Unicode usa dois bytes (também conhecido como *word*) para codificar esse mesmo caractere. Por exemplo, ASCII reserva o byte 41_{16} (65_{10}) para o caractere maiúsculo A; Unicode reserva a *word* 0041_{16}. Como o ASCII usa somente um byte, ele é limitado a somente 256 caracteres diferentes [3]. Unicode extende os valores ASCII para *words* e permite que muito mais caracteres possam ser representados. Isso é importante para representar caracteres de todas as línguas do mundo.

1.2.2 A CPU

A unidade central de processamento (CPU, em inglês) é o dispositivo físico que executa isntruções. As instruções que uma CPU executa são geralmente bem simples. As instruções podem exigir que os dados usados por elas estejam em locais especiais na própria CPU chamados *registradores*. A CPU acessa dados em registradores muito mais rápido que na memória. Contudo, o número de registradores na CPU é limitado, assim programadores precisam ter o cuidado de manter somente o dados atualmente usados nos registradores.

As instruções que uma CPU é capaz de interpretar estão em *linguagem de máquina*. Programas de máquina tem uma estrutura bem mais básica do que linguagens de alto nível. As instruções de máquina são codificadas como fileiras de números, não como formatos de texto amigáveis. Uma CPU precisa ser capaz de decodificar o propósito de uma instrução muito rapidamente e executá-la de maneira eficiente. As linguagens de máquina são desenvolvidas com um objetivo em mente, não para serem facilmente decifradas por humanos. Programas escritos em outras linguagens precisam ser convertidos na linguagem de máquina nativa de uma CPU para serem

[3]De fato, ASCII usa somente os 7-bits mais baixos assim tem somente 128 valores diferentes disponíveis.

executados em um computador. Um *compilador* é um programa que traduz programas escritos em uma linguagem de programação para a linguagem de máquina de uma determinada arquitetura. Em geral, todo tipo de CPU tem sua própria linguagem de máquina. Esse é um dos motivos que impedem que um programa de Mac seja executado em um computador da linha IBM PC.

GHz *significa gigahertz ou um bilhão de ciclos por segundo. Um clock de 1.5 Ghz gera 1.5 bilhões de pulsos de clock por segundo.*

Computadores usam um *clock* para sincronizar a execução das instruções. O clock pulsa em uma frequência fixa (conhecida como *velocidade de clock*). Quando você compra um computador de 1.5 GHz, 1.5 GHz é a frequência desse clock. Esse clock não registra minutos ou segundos. Ele simplesmente pulsa em um ritmo constante. A eletrônica empregada na CPU usa esses pulsos para executar as operações corretamente, da mesma maneira que as batidas de um metrônomo auxiliam um músico a manter o ritmo. O número de pulsos (ou *ciclos* como são geralmente chamados) que uma instrução requer dependem da geração e modelo da CPU. O número de ciclos depende das instruções anteriores e outros fatores.

1.2.3 A família 80x86 de CPUs

Os PC's do padrão IBM contém uma CPU da família Intel 80x86 (ou clone de um). Todas as CPU's nessa família contém algumas características comuns incluindo uma linguagem de máquina base. Contudo, os menbros mais recentes possuem características mais avançadas.

8088,8086: Essas CPU's do ponto de vista de programação são idênticas. Eram usadas nos primeiros PC's. Elas disponibilizam vários registradores de 16 bits: AX, BX, CX, DX, SI, DI, BP, SP, CS, DS, SS, ES, IP, FLAGS. Elas suportam até um megabyte de memória e operam somente no *modo real*. Nesse mode, um programa pode acessar qualquer endereço de memória, até mesmo o de outros programas! Isso torna muito difícil a depuração de erros e a segurança. Além disso, a memória de um programa tem que ser dividida em *segmentos*. Cada segmento não pode ser maior do que 64K.

80286: Essa CPU foi usada nos PC's da linha AT. Ela inclui algumas novas instruções na linguagem de máquina base do 8088/86. Contudo, sua principal nova característica é *modo protegido de 16-bits*. Nesse modo, é possível acessar até 16 megabytes e impedir que programas acessem a memória uns dos outros. Mas os programas continuam sendo divididos em segmentos que não podem ser maiores que 64K.

80386: Essa CPU é um grande avanço do 80286. Ela aumenta muitos dos registradores para armazenar 32-bits (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP) e inclui dois novos registradores de 16-bits: FS

AX

AH	AL

Figura 1.5: O registrador AX

e GS. Ela também possui um novo *modo protegido de 32-bits*. Nesse modo, podem ser acessados até 4 gigabytes. Os programas continuam sendo divididos em segmentos, mas agora cada segmento pode ter até 4 gigabytes de tamanho!

80486/Pentium/Pentium Pro: Esses membros da família 80x86 acrescentam algumas poucas novas características. Eles basicamente aceleram a velocidade de execução das instruções.

Pentium MMX: Esse processador inclui as instruções MMX (MultiMedia eXtentions) ao Pentium. Essas instruções aceleram operações gráficas comuns.

Pentium II: Esse é um processador Pentium Pro com instruções MMX. (O Pentium III é essencialmente um Pentium II mais rápido.)

1.2.4 Os registradores de 16-bits

As CPU 8086 originais tinham quatro registradores de 16-bits de uso geral: AX, BX, CX e DX. Cada um desses registradores podia ser decomposto em registradores AH e AL como mostra a figura 1.5. O registrador AH contém os 8 bits mais altos de AX e o AL contém os 8 bits mais baixos de AX. Frequentemente AH e AL são usados como registradores independentes de 8 bits. Contudo, é importante ter em mente que eles não são independentes de AX. Alterar o valor de AX faz com que o valor de AH e Al sejam alterados e *vice versa*. Os registradores de uso geral são usados em muitas movimentações de dados e instruções aritméticas.

Esses são dois registradores de 16-bits: SI e DI. Eles são frequentemente usados como ponteiros, mas podem ser usados da mesma maneira que os registradores de uso geral. Contudo, eles não podem ser decompostos em registradores de 8-bits.

Os registradores de 16-bits BP e SP são usados para apontar para os dados na pilha de linguagem de máquina. Isso será discutido posteriormente.

Os registradores de 16-bits CS, DS, SS e ES são *registradores de segmento*. Eles indicam qual memória é usada por diferentes partes de um programa. CS significa Segmento de Código (Code Segment, em inglês), DS significa Segmento de Dados (Data Segment, em inglês), SS significa Segmento de

Pilha (Stack Segment, em inglês) e ES significa Segmento Extra (Extra Segment, em inglês). ES é usado como um registrador temporário de segmentos. Os detalhes desse registrador estão na sessão 1.2.6 e 1.2.7.

O registrador Ponteiro de Instruções (IP - Instruction Pointer) é usado com o registrador CS para indicar o endereço da próxima instrução a ser executada pela CPU. Normalmente, a medida que uma instrução é executada, IP é alterado para apontar para o endereço da próxima instrução em memória.

O registrador FLAGS armazena informações importantes sobre os resultados de uma instrução anterior. Esses resultados são armazenados como bits individuais no registrador. Por exemplo, o bit Z é 1 se o resultado da instrução anterior foi zero e 0 caso contrário. Nem todas as instruções alteram os bits no registrador FLAGS. Consulte a tabela no apêndice para ver como instruções individuais afetam o registrador FLAGS.

1.2.5 80386 registradores de 32-bits

O 80386 e processadores posteriores tem registradores extendidos. Por exemplo, o registrador de 16-bits AX é extendido para ser 32-bits. Para manter compatibilidade, AX continua se referindo ao registrador de 16-bits e EAX é usado para referenciar um registrador de 32-bits. AX são os 16-bits mais baixos de EAX da mesma maneira que AL são os 8-bits mais baixos de AX (e EAX). Não existe maneira de acessar os 16-bits mais altos de EAX diretamente.

Os registradores de segmento continuam sendo de 16-bits no 80386. Há também novos registradores de segmento: FS e GS. Não existe significado para suas siglas. Eles são registradores temporários de segmentos adicionais (assim como ES).

Uma das definições do termo *word* refere-se ao tamanho dos dados nos registradores da CPU. Na família 80x86, esse termo se tornou um pouco confuso. Na tabela 1.2, observamos que *word* é definido com 2 bytes (ou 16 bits). Foi dado esse significado quando o 8086 foi criado. Quando o 80386 foi desenvolvido, foi decidido que essa definição de *word* seria mantida, ainda que o tamanho do registrador tenha mudado.

1.2.6 Modo Real

De onde vinha aquela famosa limitação do DOS de 640K? A BIOS requisitava parte de 1M da memória disponível para seu código e dispositivos de hardware como vídeo.

No modo real, a memória é limitada a somente 1 Megabyte (2^{20} bytes). Os endereços de memória válidos vão de 00000 até FFFFF. Esses endereços requerem um número de 20-bits. Obviamente, um número de 20-bits não cabe em nenhum dos registradores de 16-bits de máquinas 8086. A Intel resolveu esse problema, usando dois valores de 16-bits para determinar um endereço. O primeiro valor de 16-bits é chamado de *seletor* (selector, em inglês) e o segundo valor de 16-bits é chamado de *offset*. O endereço físico referenciado pelo par *seletor:offset* é computado pela fórmula

$$16 * \text{selector} + \text{offset}$$

Multiplicar por 16 em hexadecimal é simples, simplesmente adicione um 0 a direita do número. Por exemplo, o endereço físico referenciado por 047c:0048 é dado por:

$$\begin{array}{r} 047C0 \\ +0048 \\ \hline 04808 \end{array}$$

Em consequência, o valor do seletor é o número do parágrafo. (Veja a tabela 1.2).

Endereços de memória real segmentados tem disvantagens:

- Um simples valor de um seletor pode somente referenciar 64K de memória (o limite de um offset de 16-bits). O que acontece se um programa tem mais que 64K de código? Um simples valor em CS não pode ser usado para a execução completa de um programa. O programa precisa ser dividido em seções (chamados *segmentos*) menores que 64K em tamanho. Quando a execução move de um segmento para outro, o valor de CS deve ser alterado. Problemas similares ocorrem com grandes quantidades de dados e o registrador DS. Isso pode ser bem complicado!

- Cada byte na memória não tem um único endereço segmentado. O endereço físico 04808 pode ser referenciado por 047C:0048, 047D:0038, 047E:0028 ou 047B:0058. Isso pode complicar a comparação de endereços segmentados.

1.2.7 O Modo protegido de 16-bits

No modo protegido de 16-bits da linha 80286, os valores de seletores são interpretados de maneira totalmente diferente do que no modo real. No modo real, o valor de um seletor é um *índice* dentro de uma *tabela descritora*. Em ambos os modos, os programas são divididos em segmentos. No modo real, esses segmentos estão em posições fixas na memória física e o valor de um seletor designa o número do parágrafo no início do segmento. No modo protegido, os segmentos não estão em posições fixas na memória física. De fato, eles não tem que estar na memória!

O modo protegido usa uma técnica chamada *memória virtual*. A idéia básica de um sistema de memória virtual é a de manter somente os dados e códigos em memória dos programas que estão sendo usados no momento.

Outros dados e códigos são temporariamente armazenados em disco até que eles sejam necessários novamente. No modo protegido de 16-bits, segmentos são movidos entre a memória e o disco quando necessário. Quando um segmento é devolvido do disco para a memória, é muito comum que ele seja colocado em um endereço de memória diferente daquele em que estava antes de ser transferido para o disco. Tudo isso é feito de forma transparente pelo sistema operacional. O programa não precisa ser escrito de maneira diferente para trabalhar com memória virtual.

No modo protegido, cada segmento é associado a uma entrada na tabela de descritores. Essa entrada tem todas as informações que o sistema precisa saber sobre o segmento. Essas informações incluem: está atualmente em memória; se está em memória, onde é; permissões de acesso (*e.g.*, somente leitura). O índice de uma entrada do segmento é o valor do seletor que está armazenado nos registradores e segmento.

Um famoso colunista de PCs chamou o 286 de "cérebro morto."

Uma grande disvantagem do modo protegido de 16-bits é que os offsets continuam sendo de 16 bits. Como consequencia disso, o tamanho dos segmentos continua limitado a 64K. Isso torna o uso de grandes vetores problemático!

1.2.8 O modo protegido de 32-bits

O 80386 introduz o modo protegido de 32-bits. Há duas grandes diferenças entre os modo protegido do 286 e do 386:

1. Os offsets são expandidos para 32-bits. Isso faz com que um offset alcance até 4 bilhões. Assim, os segmentos podem ter tamanho de até 4 gigabytes.

2. Os segmentos podem ser divididos em pequenas unidades de 4K chamadas *páginas*. O sistema de memória virtual agora trabalha com páginas ao invés de segmentos. Isso significa que somente parte do segmento pode estar em memória em qualquer momento. No modo 16-bits do 286, o segmento inteiro ou está em memória ou não está. Isso não é prático com os grandes segmentos que o modo 32-bits suporta.

No Windows 3.x, o *modo padrão* refere-se ao modo protegido de 16-bits do 286 e o *modo avançado* refere-se ao modo de 32-bits. O Windows 9X, Windows NT/2000/XP, OS/2 e no Linux rodam todos em um modo paginado de 32-bits.

1.2.9 Interrupções

Algumas vezes o fluxo ordinário de um programa deve ser interrompido para processar eventos que requerem uma resposta. O hardware de um computador provê um mecanismo chamado *interrupções* para capturar esses eventos. Por exemplo, quando um mouse é movido o hardware do mouse interrompe o programa atual para capturar o movimento do mouse (para mover o cursor do mouse, *etc.*) interrupções fazem com que o controle seja transferido para um *capturador de interrupções.* Capturadores de interrupção são rotinas que processam a interrupção. Cada tipo de interrupção é associada a um número inteiro. No início da memória física, uma tabela de *vetores de interrupção* contém os endereços segmentados dos capturradores de interrupção. O número da interrupção é essencialmente um índice nessa tabela.

Interrupções externas são geradas fora da CPU. (O mouse é um exemplo disso.) Muitos dispositivos de E/S geram interrupções (*e.g.*, teclado, timer, unidades de disco, CD-ROM e placas de som). Interrupções internas são geradas dentro da CPU, tanto para um erro quanto para uma instrução de interrupção. Interrupções de erro também são chamadas de *traps.* Interrupções geradas por uma instrução de interrupção são chamadas de *interrupções de software.* O DOS usa esse tipo de interrupção para implementar a sua API (Application Programming Interface). Modernos sistemas operacionais (como o Windows e o UNIX) usam a interface baseada em C.[4]

Muitos gerenciadores de interrupção devolvem o controle para o programa interrompido quando eles finalizam a operação. Eles restauram todos os registradores para os mesmos valores que eles tinham antes da interrupção ocorrer. Dessa forma, o programa interrompido roda como se nada tivesse acontecido (exceto os ciclos de CPU que foram perdidos). Traps geralmente não retornam o controle. Geralmente elas abortam o programa.

1.3 A linguagem Assembly

1.3.1 A linguagem de máquina

Todo tipo de CPU entende seu próprio código de máquina. Instruções em linguagem de máquina são números armazenados como bytes em memória. Cada instrução tem um código numérico único chamado de *código de operação* ou *opcode* abreviando (essa abreviação é relativa ao termo original operation code). As instruções dos processadores 80x86 variam em tamanho. O opcode está sempre no início de uma instrução. Muitas instruções também incluem dados (*e.g.*, constantes ou endereços) usados pelas instruções.

Programar em linguagem de máquina diretamente é muito difícil. Decifrar o significado de instruções em código numérico é tedioso para humanos. Por exemplo, a instrução que manda somar os registradores EAX e EBX e armazenar o resultado em EAX é codificado pelo seguinte código hexadecimal:

[4] Contudo, eles podem usar uma interface de nível mais baixo no kernel.

03 C3

Essa é a dura realidade. Felizmente, um programa chamado *assembler* pode fazer esse tedioso trabalho para os programadores.

1.3.2 A linguagem Assembly

Um programa em linguagem assembly é armazenado como texto (assim como os programas em linguagem de alto nível). Cada instrução em assembly representa exatamente uma instrução de máquina. Por exemplo, a instrução de adição descrita acima poderia ser representada na linguagem assembly como:

```
add eax, ebx
```

Aqui o significado da instrução é *muito* mais claro que no código de máquina. A palavra `add` é um *mnemônico* para a instrução de adição. O formato geral de uma instrução assembly é:

mnemônico operador(es)

Um *assembler* é um programa que lê um arquivo texto com instruções em assembly e converte-as em código de máquina. *Compiladores* são programas que fazem conversões similares para linguagens de programação de alto nível. Um assembler é muito mais simples que um compilador. Todo comando em assembly corresponde diretamente a uma instrução em código de máquina. Os comandos de linguagens de alto nível são *muito* mais complexos e podem exigir muitas instruções de máquina.

Levou muitos anos para que os cientistas de computador imaginassem como escrever um compilador!

Outra importante diferença entre assembly e linguagens de alto nível e que a partir do momento que cada CPU tem sua própria linguagem de máquina, também terá sua própria linguagem assembly. Migrar programas assembly entre diferentes arquiteturas de computador é *muito* mais difícil que uma linguagem de alto nível.

Os exemplos desse livro usam o Netwide Assembler ou NASM para abreviar. Ele está disponível gratuitamente na Internet (veja no prefácio a URL). Os assemblers mais comuns são o Microsoft Assembler (MASM) e o Borland Assembler (TASM). Há algumas diferenças de sintaxe entre o MASM/TASM e o NASM.

1.3.3 Operadores de uma instrução

Instruções em código de máquina possuem um número variado de tipos de operadores; contudo, em geral, cada instrução por si só tem um número fixo de operadores (0 a 3). Operadores podem ser dos seguintes tipos:

registrador: Esses operadores referem-se diretamente ao conteúdo dos registradores da CPU.

memória: Se refere aos dados na memória. O endereço de um dado pode ser uma constante codificada na instrução ou pode ser computada usando valores dos registradores. Endereços são sempre offsets no iniício de um segmento.

imediato: Esses são volores fixos que são listados nas instruções. Eles são armazenados na própria instrução (no segmento de código), não no segmento de dados.

implícito: Esses operadores não são explicitamente exibidos. Por exemplo, a instrução de incremento adiciona 1 ao registrador ou memória. O 1 é implícito.

1.3.4 Instruções básicas

A instrução mais básica é a instrução `MOV`. Ela move dados de um local para o outro (como o operador de atribuição em uma linguagem de alto nível). Ela possui dois operadores:

```
mov dest, src
```

O dado especificado por *src* é copiado para *dest*. Uma restrição é que ambos os operadores não podem ser operadores de memória. Isso indica outra artimanha de assembly. Frequentemente há em alguma parte regras arbitrárias sobre como as várias instruções são usadas. Os operadores também devem ser do mesmo tamanho. O valor de AX não pode ser armazenado em BL.

Aqui há um exemplo (ponto e vírgula inicia um comentário):

```
mov    eax, 3   ; armazena 3 no registrador EAX (3 é um operador imediato)
mov    bx, ax   ; armazena o valor de AX no registrador BX
```

A instrução `ADD` é usada para adicionar inteiros.

```
add    eax, 4   ; eax = eax + 4
add    al, ah   ; al = al + ah
```

A instrução `SUB` subtrai inteiros.

```
sub    bx, 10   ; bx = bx - 10
sub    ebx, edi ; ebx = ebx - edi
```

As instruções `INC` e `DEC` incrementam e decrementam valores em 1. A partir do momento que 1 é um operador implícito, o código de máquina para `INC` e `DEC` é menor que o equivalente para as instruções `ADD` e `SUB`.

```
inc    ecx      ; ecx++
dec    dl       ; dl--
```

1.3.5 Diretivas

Uma *diretiva* é um artefato do assembler, não da CPU. Elas são geralmente usadas tanto para instruir o assembler a fazer algo quanto para informar o assembler de alguma coisa. Elas não são traduzidas em código de máquina. Usos comuns de diretivas são:

- definir constantes
- definir memória para armazenar dados
- Grupos de memória em segmentos
- inclusão condicional de código fonte
- inclusão de outros arquivos

O código do NASM passa por um preprocessamento assim como em C. Ele tem muitos comandos de preprocessamento equivalentes ao C. Contudo, as diretivas de processamento no NASM começam com um % ao invés de # como em C.

A diretiva equu

A diretiva `equ` pode ser usada para definir um *símbolo*. Símbolos são constantes nomeadas que podem ser usadaas no programa assembly. O formato é:

```
símbolo equ valor
```

Valores de símbolos *não* podem ser redefinidos posteriormente.

A diretiva %define

Essa diretiva é similar a diretiva `#define` em C. Ela é comunmente usada para definirconstantes de macro da mesma maneira que em C.

```
%define SIZE 100
      mov    eax, SIZE
```

O código acima define uma macro denominada `SIZE` e mostra seu uso em uma instrução `MOV`. Macros são mais flexíveis que símbolos por duas razões: Macros podem ser redefinidas e podem ser mais do que simples constantes numéricas.

Unit	Letter
byte	B
word	W
double word	D
quad word	Q
ten bytes	T

Tabela 1.3: Letras para as diretivas `RESX` e `DX`

A diretiva data

As Diretivas Data são usadas em segmentos de dados para definir áreas em memória. Existem duas maneiras de reservar memória. A primeira maneira somente define área para dados; a segunda maneira define área e um valor inicial. O primeiro método usa uma das diretivas `RES`*X*. O *X* é substituido com uma letra que determina o tamanho do objeto (ou objetos) que irão ser armazenados. A tabela 1.3 mostra os valores possíveis.

O segundo método (que também define um valor inicial) usa uma das diretivas `D`*X*. As letras *X* são as mesmas como aquelas da diretiva `RES`*X*.

É muito comun marcar regiões de memória com *labels*. Labels facilitam referências a localizações de memória no código. Abaixo há vários exemplos:

```
L1    db     0        ; byte rotulado L1 com valor inicial 0
L2    dw     1000     ; word rotulado L2 com valor inicial 1000
L3    db     110101b  ; byte inicializado como binário 110101 (53 em decimal)
L4    db     12h      ; byte inicializado como hexadecimal 12 (18 em decimal)
L5    db     17o      ; byte inicializado como octagonal 17 (15 em decimal)
L6    dd     1A92h    ; double word inicializada como hexadecimal 1A92
L7    resb   1        ; 1 "uni inicializado" byte
L8    db     "A"      ; byte inicializado com o código ASCII de A (65)
```

Aspas duplas e aspas simples são tratadas da mesma forma. Definições de dados consecutivas são armazenadas sequencialmente em memória. Isso é, a word L2 é armazenada imediatamente depois de L1 em memória. Sequências de memória podem ser também definidas.

```
L9    db     0, 1, 2, 3               ; define 4 bytes
L10   db     "w", "o", "r", 'd', 0    ; define uma string C = "word"
L11   db     'word', 0                ; O mesmo que L10
```

Para grandes sequências, a diretiva do NASM `TIMES` é muito útil. Essa diretiva repete seu operador um número específico de vezes. Por exemplo,

```
L12   times 100 db 0                  ; equivalente a 100 (db 0)'s
L13   resw   100                      ; reserva área para 100 words
```

Lembre-se que labels podem ser usadas para referenciar dados no código. Há duas maneiras de se usar um label. Se uma label base é usada, ela é interpretada como um endereço (ou offset) de um dado. Se a label é colocada entre colchetes ([]), ela é interpretada como o dado no endereço. Em outras palavras, pode-se pensar em um label como um *ponteiro* para o dado e os colchetes referencia o ponteiro da mesma maneira que o asterisco em C. (MASM/TASM seguem padrões diferentes.) No modo de 32-bits, os endereços são de 32-bits. Alguns exemplos:

```
1   mov   al, [L1]     ; copia o byte em L1 para AL
2   mov   eax, L1      ; EAX = endereço do byte em L1
3   mov   [L1], ah     ; copia AH como byte em L1
4   mov   eax, [L6]    ; copia a double word em L6 para EAX
5   add   eax, [L6]    ; EAX = EAX + double word em L6
6   add   [L6], eax    ; double word em L6 += EAX
7   mov   al, [L6]     ; copia o primeiro byte da double word em L6 into AL
```

A linha 7 dos exemplos mostra uma importante propriedade do NASM. O assembler *não* armazena o tipo de dado que o label faz referência. Com isso o programador (ou programadora) deve fazer uso correto de um label. Adiante será comum armazenar endereços de dados em registradores e usar os registradores como uma variável ponteiro do C. Novamente, não é feita verificação se o ponteiro está sendo usado corretamente. Com isso, assembly é muito mais sujeito a erros que o C.

Considere as seguintes instruções:

```
mov   [L6], 1            ; armazena 1 em L6
```

Esse comando produz um `operation size not specified` (tamanho do operador não especificado). Por quê? Porque o assembler não sabe se o valor 1 deve ser armazenado como byte, word ou double word. Para consertar isso, adicione um especificador de tamanho.

```
mov   dword [L6], 1      ; armazena 1 em L6
```

Isso instrui o assembler a armazenar o valor 1 na double word que inicia em `L6`. Outros especificadores de tamanho são: `BYTE`, `WORD`. `QWORD` e `TWORD`.

1.3.6 Entrada e saída

Entrada e saída são atividades muito dependentes do sistema. Elas envolvem integração com o hardware do sistema. Linguagens de alto nível, como C, disponibilizam rotinas em bibliotecas padrão que tornam as interfaces de E/S simples e com programação uniformizada. Linguagens assembly

print_int	imprime na tela os valores inteiros armazenados em EAX
print_char	imprime na tela caracteres cujo valor ASCII está armazenado em AL
print_string	imprime na tela o conteúdo da strings no *endereço* armazenado em EAX. A string deve ser uma string do tipo C (*i.e.* com terminador nulo).
print_nl	imprime na tela um caractere de nova linha.
read_int	Lê um inteiro pelo teclado e armazena-o no registrador EAX.
read_char	Lê um caractere simples pelo teclado e armazena seu código ASCII no registrador EAX.

Tabela 1.4: Rotinas de E/S do assembly

não disponibilizam bibliotecas padrão. É necessário acessar o hardware diretamente (que é uma operação privilegiada no modo protegido) ou usar alguma rotina de baixo nível que o sistema operacional oferece.

É muito comum rotinas em assembler serem integradas com C. Uma vantagem disso é que o código assembly pode usar as bibliotecas de rotinas de E/S padrão do C. Contudo, é necessário conhecer as regras de passagem de informações entre as rotinas que o C usa. Essas regras são muito complexas para serem cobertas aqui. (Elas serão cobertas mais para frente!) Para simplificar a E/S, o autor desenvolveu suas próprias rotinas que ocultam as complexas regras do C e disponibilizam uma interface bem mais simples. A tabela 1.4 descreve as rotinas disponíveis. Todas as rotinas preservam os valores de todos os registradores, exceto as rotinas de leitura. Essas rotinas modificam o valor do registrador EAX. Para usar essas rotinas, é preciso incluir um arquivo com informações que o assembler precisa para usá-las. Para incluir um arquivo no NASM, use a diretiva de pré-processamento `%include`. A seguinte linha inclui um arquivo requerido pelas rotinas de E/S do autor[5]:

```
%include "asm_io.inc"
```

Para usar uma das rotinas de impressão, carregue o registrador EAX com o valor correto e use uma instrução `CALL` para invocá-lo. A instrução `CALL` é equivalente a chamada de uma função em linguagens de alto nível. Ela transfere a execução para outra seção de código, mas retorna para sua origem depois que a rotina termina. O programa exemplo abaixo mostra vários exemplos de chamadas dessas rotinas de E/S.

[5] O `asm_io.inc` (e o arquivo objeto `asm_io` que o `asm_io.inc` requer) estão na página de downloads de exemplos desse tutorial, `http://www.drpaulcarter.com/pcasm`

1.3.7 Depuração

As bibliotecas do autor também contém algumas rotinas úteis para depuração de programas. Essas rotinas de depuração mostram informações sobre o estado do computador sem modificar o estado. Essas rotinas são de fato *macros* que preservam o estado atual da CPU e então fazem a chamada da subrotina. As macros estão definidas no arquivo `asm_io.inc` discutido acima. Macros são usadas como instruções ordinárias. Operadores de macros são separados por vírgulas.

Há quatro rotinas de depuração denominadas `dump_regs`, `dump_mem`, `dump_stack` e `dump_math`; Elas mostram os valores dos registradores, memória, pilha e do co-processador matemático, respectivamente.

dump_regs Essa macro imprime o valor dos registradores (em hexadecimal) do computador para `stdout` (*i.e* a tela). Ela também mostra os bits "setados" no registrador FLAGS[6]. Por exemplo, se a flag zero é 1, *ZF* é exibida. Se é 0, não é exibida. Ela recebe um argumento inteiro simples que é impresso da mesma forma. Isso pode ser usado para distinguir a saída de diferentes comandos `sump_regs`.

dump_mem Esssa macro imprime o valor de uma região de memória (em hexadecimal) e também como caracteres ASCII. Ela recebe três argumentos separados por vírgula. O primeiro é um inteiro que é usado para rotular a saída (assim como o argumento de `dump_regs`). O segundo argumento é o endereço a ser mostrado. (Ele pode ser um label.) O último argumento é um número de 16-bytes em parágrafos, exibido depois do endereço. A memória exibida irá iniciar na primeira marca de parágrafo antes do endereço requisitado.

dump_stack Essa macro imprime o valor da pilha da CPU. (A pilha será coberta no capítulo 4.) A pilha é organizada como double words e essa rotina mostra elas dessa maneira. Ela recebe três argumentos separados por vírgula. O primeiro é um label do tipo integer (como `dump_regs`). O segundo é o número de double words que se deseja exibir *abaixo* do endereço que o registrador `EBP` armazena e o terceiro argumento é o número de double words que se deseja mostrar *acima* do endereço em `EBP`.

dump_math Essa macro imprime o valor dos registradores do co-processador matemático. Ele recebe um argumento que é usado para rotular a saída da mesma maneira que o argumento de `dump_regs` faz.

[6]Esse registrador é discutido no capítulo 2

```
int main()
{
  int ret_status;
  ret_status = asm_main();
  return ret_status;
}
```

Figura 1.6: `driver.c` código

1.4 Criando um Programa

Hoje em dia, não é comum criar um programa sozinho escrito totalmente na linguagem assembly. Assembly é geralmente usada para codificar certas rotinas críticas. Por quê? É *muito* mais fácil criar um programa em uma linguagem de alto nível do que em assembly. Além disso, o uso de assembly faz com que seja muito difícil portar um programa para outra plataforma. De fato, é raro usar assembly em tudo.

Sendo assim, por quê alguém deveria aprender assembly?

1. As vezes o código escrito em assembly pode ficar mais rápido e menor do que um código gerado por compilação.

2. Assembly permite o acesso direto a características de hardware do sistema que poderiam ser difíceis ou impossíveis de usar em linguagens de alto nível.

3. Aprender como programar em assembly ajuda a ganhar um conhecimento mais profundo sobre como um computador funciona.

4. Aprender como programar em assembly ajuda a ter uma visão melhor sobre como funcionam compiladores e linguagens de alto nível.

Esses últimos dois itens demonstram que aprender assembly pode ser útil mesmo que não se programe usando ele depois. De fato, o autor raramente programa em assembly, mas ele usa os conceitos aprendidos com assembly o tempo todo.

1.4.1 O primeiro Programa

Os primeiros programs nesse texto irão todos começar a partir de uma simples rotina em C como mostra a figura 1.6. Ela simplesmente chama outra função chamada `asm_main`. Essa será a rotina que será escrita em assembly. Há muitas vantagens no uso dessa rotina em C. Primeiro, ela permite que o sistema em C configure o programa para rodar corretamente no modo protegido. Todos os segmentos e seus registradores de segmento serão inicializados pelo C. O código em assembly não precisa se preocupar com isso. Segundo, a biblioteca do C estará disponível para ser usada pelo código assembly. As rotinas de E/S do autor tiram proveito disso. Elas usam funções de E/S do C (`printf`, *etc.*). Abaixo é mostrado um programa assembly simples.

first.asm

```
; file: first.asm
; Primeiro programa em assembly. Esse programa solicita dois inteiros e imprime
; a sua soma
;
; Para criar um executável usando djgpp:
; nasm -f coff first.asm
; gcc -o first first.o driver.c asm_io.o

%include "asm_io.inc"
;
; Os dados inicializados são colocados no segmento .data
;
segment .data
;
; Essas labels referem-se as strings que serão usadas para saída
;
prompt1 db    "Digite um número: ", 0       ; não esqueça o terminador nulo
prompt2 db    "Digite outro número: ", 0
outmsg1 db    "Você entro com ", 0
outmsg2 db    " e ", 0
outmsg3 db    ", a soma deles é ", 0

;
; dados não inicializados são colocados no segmento .bss
;
segment .bss
;
; Essas labels referem-se as double words usadas para armazenar as entradas
;
input1  resd 1
input2  resd 1

;
; O código é colocado no segmento .text
;
```

```
segment .text
        global  _asm_main
_asm_main:
        enter   0,0               ; rotina de configuração
        pusha

        mov     eax, prompt1      ; imprime o aviso
        call    print_string

        call    read_int          ; lê o inteiro
        mov     [input1], eax     ; armazena em input1

        mov     eax, prompt2      ; imprime o aviso
        call    print_string

        call    read_int          ; lê o inteiro
        mov     [input2], eax     ; armazena em input2

        mov     eax, [input1]     ; eax = dword at input1
        add     eax, [input2]     ; eax += dword at input2
        mov     ebx, eax          ; ebx = eax

        dump_regs 1                ; imprime o valor dos registradores
        dump_mem  2, outmsg1, 1    ; imprime a memória
;
; As próximas saídas saídas imprimem o resultado em uma série de passos
;
        mov     eax, outmsg1
        call    print_string      ; imprime a primeira mensagem
        mov     eax, [input1]
        call    print_int         ; imprime input1
        mov     eax, outmsg2
        call    print_string      ; imprime a segunda mensagem
        mov     eax, [input2]
        call    print_int         ; imprime input2
        mov     eax, outmsg3
        call    print_string      ; imprime a terceira mensagem
        mov     eax, ebx
        call    print_int         ; imprime a soma (ebx)
        call    print_nl          ; imprime uma nova linha

        popa
```

```
78        mov     eax, 0            ; volta para o C
79        leave
80        ret
```

first.asm

A linha 13 do programa define uma seção do programa que especifica a memória a ser armazenada no segmento de dados (cujo nome é `.data`). Somente dados inicializados devem ser definidos nesse segmento. Das linhas 17 a 21, várias strings são declaradas. Elas serão impressas com a biblioteca C e assim precisam ser terminadas com um caractere nulo (em inglês, *NULL* com código ASCII 0). Lembre-se que existe uma grande diferença entre `0` e `'0'`.

Dados não inicializados devem ser declarados no segmento bss (chamado `.bss` na linha 26). Esse segmento recebeu esse nome de um operador de uma versão assembler baseada em UNIX que indicava "bloco iniciado por símbolo." Há também um segmento de pilha. Ele será discutido depois.

O segmento de código é chamado `.text` históricamente. É onde as instruções são colocadas. Note que o label da rotina principal (linha 38) possui um símbolo de underscore. Isso é parte das *convenção de chamadas em C*. Essa convenção determina as regras que o C usa quando compila um código. É muito importante conhecer essas convenções quando existe uma interface entre C e assembly. Mais adiante a convenção será completamente detalhada; contudo, por hora, precisamos somente saber que todos os símbolos do C (*i.e.*, funções e variáveis globais) tem um underscore como prefixo adicionado a eles pelo compilador C. (Essa regra é especificamente para DOS/Windows, os compiladores C Linux não prepõem nada para o nome dos símbolos do C.)

A diretiva `global` na linha 37 instrui o assembler a tornar a label `_asm_main` global. Diferente do C, as labels possuem um *escopo interno* por padrão. Isso significa que somente o código no mesmo módulo pode usar a label. A diretiva `global` da a label especificada (ou labels) um *escopo externo*. Esse tipo de label pode ser acessada por qualquer módulo do programa. O módulo `asm_io` declara as labels `print_int`, *et.al.* para serem globais. Esse é o motivo que torna possível usá-las no módulo `first.asm`.

1.4.2 Dependências do compilador

O código assembly acima é específico para o free GNU[7]-baseado no compilador C/C++ DJGPP.[8] Esse compilador pode ser baixado gratuitamente da internet. Ele requer um 386 ou maior e roda sobre o DOS, Windows 95/98 ou NT. Esse compilador usado arquivos objeto no formato COFF (Formato

[7]GNU é um projeto da Free Software Foundation (`http://www.fsf.org`)
[8]`http://www,delorie.com/djgpp`

Comum de arquivo objeto ou Common Object File Format, em inglês). Para montar nesse formato use a chave `-f coff` com o `nasm` (como mostrado nos comentários do código acima). A extensão o arquivo objeto resultante será `o`.

O compilador C do Linux é um compilador GNU também. Para converter o código acima permitindo que ele rode no Linux, simplesmente remova os prefixos de undercore nas linhas 37 e 38. Linux usa o formato ELF (Formato executável e "linkavel"ou Executable and Linkable Format, em inglês) para arquivos objeto. Use a chave `-f elf` para o Linux. O arquivo objeto gerado também recebe a extensão `o`.

Os arquivos de exemplo do compilador, disponíveis no web site do autor, foram modificados para trabalhar com o compiladro apropriado.

Borland C/C++ é outro compilador popular. Ele usa o formato OMF da Microsoft para arquivos objeto. Use a chave `-f obj` para compiladores da Borland. A extensão doarquivo será `obj`. O formato OMF usa diferentes diretivas `segment` do que nos outros formatos. O segmento de dados (linha 13) deve ser mudado para:

```
segment _DATA public align=4 class=DATA use32
```

The bss segment (line 26) must be changed to:

```
segment _BSS public align=4 class=BSS use32
```

The text segment (line 36) must be changed to:

```
segment _TEXT public align=1 class=CODE use32
```

Em adição uma nova linha deve ser incluída antes da linha 36:

```
group DGROUP _BSS _DATA
```

O compilador C/C++ da Microsoft pode tanto usar o formato OMF quanto o formato Win32 para arquivos objeto. (se o formato escolhido for o OMF, ele converte as informações para o formato Win32 internamente.) O formato Win32 permite que os segmentos sejam definidos da mesma maneira que no DJGPP e no Linux. Use a chave `-f win32` para que a saída seja nesse modo. A extensão do arquivos será `obj`.

1.4.3 Montando (Assembly) o código

O primeiro passo para montar o código. A partir da linha de comando, digite:

```
nasm -f object-format first.asm
```

aonde *object-format* é tanto *coff*, *elf*, *obj* ou *win32* dependendo do compilador C usado. (Lembre-se que o código fonte deve ser alterado tanto para Linux quanto para o Borland.)

1.4.4 Comnpilando o código em C

Compile o arquivo `driver.c` usando um compilador C. Para o DJGPP, use:

```
gcc -c driver.c
```

A chave `-c` indica somente compilação, não tentar linkar ainda. Essa mesma chave funciona nos compiladores Linux, Borland e Microsoft.

1.4.5 Linkando os arquivos objeto

Linkar é o processo de unir o código de máquina e os dados dos arquivos objeto com os arquivos de biblioteca para criar um arquivo executável. Como será mostrado abaixo, esse processo é complicado.

O código em C requer uma biblioteca padrão C e um *código de inicialização* especial para rodar. É *muito* mais fácil deixar o compilador chamar o linkador com os parâmetros corretos, do que tentar linkar diretamente. Por exemplo, para linkar o código do primeiro programa usando DJGPP, use:

```
gcc -o first driver.o first.o asm_io.o
```

Isso irá criar um exercutável chamado `first.exe` (ou simplesmente `first` no Linux).

Com o Borland, seria usado:

```
bcc32 first.obj driver.obj asm_io.obj
```

O Borland usa o nome do primeiro arquivo listado para determinar o nome do executável. Nos casos acima, o programa seria chamado de `first.exe`.

É possível combinar os passos de compilação e linkagem. Por exemplo,

```
gcc -o first driver.c first.o asm_io.o
```

Agora o `gcc` irá compilar o `driver.c` e depois linkar.

1.4.6 Entendo um arquivo de listagem em assembly

A chave `-l` *`listing-file`* pode ser usada para instruir o `nasm` a criar um arquivo de listagem com o nomew dado. Esse arquivo mostra como o código foi montado. Aqui é mostrado como as linhas 17 e 18 (no segmento de dados) aparecem no arquivo de listagem. (O número das linhas estão no arquivo de listagem; contudo note que o número da linha no arquivo fonte pode não corresponder ao número da linha no arquivo de listagem.)

```
48 00000000 456E7465722061206E-     prompt1 db    "Digite um número: ", 0
49 00000009 756D6265723A2000
50 00000011 456E74657220616E6F-     prompt2 db    "Digite outro número: ", 0
51 0000001A 74686572206E756D62-
52 00000023 65723A2000
```

A primeira coluna de cada linha é o número da linha e a segunda é o offset (em hexadecimal) do segmento de dados. A terceira coluna mostra a sequência de valores hexadecimais que serão armazenados[9]. Nesse caso o dados hexadecimais correspondem aos seus códigos ASCII. Finalmente, o texto do código fonte é exibido na linha. Os offsets listados na segunda coluna comunmente *não* são os verdadeiros offsets onde os dados serão armazenados no programa completo. Cada módulo pode definir suas próprias labels no segmento de dados (e em outros segmentos também). No passo de linkagem (veja a seção 1.4.5), todas essas definições de labels de segmentos de dados são combinadas para formar um segmento de dados. Os novos offsets são então computados pelo linkador.

Aqui está uma pequena seção (linhas 54 até 56 do arquivo fonte) do segmento de texto no arquivo de listagem:

```
94 0000002C A1[00000000]           mov     eax, [input1]
95 00000031 0305[04000000]         add     eax, [input2]
96 00000037 89C3                   mov     ebx, eax
```

A terceira coluna mostra o código de máquina gerado pelo assembly. Geralmente o código completo das instruções ainda não pode ser computado. Por exemplo, na linha 94 o offset (ou endereço) do `input1` não é conhecido até que o código seja linkado. O assembler pode computar o opcode para a instrução `mov` (que na listagem aparece como A1), mas ele escreve o offset entre colchetes porque o valor exato não foi computado ainda. Nesse caso, um offset temporário de 0 é usado porque o `input1` está na parte inicial do segmento bss definido nesse arquivo. Lembre-se que isso *não* significa que ele estará no início do segmento bss final do programa. Quando o código é linkado, o linkador irá inserir o offset correto na posição. Outras instruções, como na linha 96, não referenciam labels. Aqui o assembler pode computar o código de máquina completo.

Se olharmos com mais atenção a linha 95, algo parece muito estranho no offset entre conchetes no código de máquina. A label do `input2` está no offset 4 (como definido nesse arquivo); contudo, o offset que aparece não é 00000004, mas 04000000. Por quê? Processadores diferentes armazenam inteiros multibyte com diferentes métodos em memória. Há dois métodos populares de armazenar inteiros: *big endiam* and *little endian.* Big endian

[9]Essas sequências correspondem as mensagens do texto original em inglês

é o método que parece ser o mais natural. O maior (*i.e* mais significativo) byte é armazenado primeiro, depois o segundo maior, *etc.* Por exemplo, a dword 00000004 poderia ser armazenada nos 4 bytes 00 00 00 04. Mainframes da IBM, a maioria dos processadores RISC e Motorola usam o método big endian. Contudo, processadores Intel e similares usam o método little endian! Aqui o byte menos significativo é armazenado primeiro. Então, 00000004 é armazenado em memória como 04 00 00 00. Esse formato está inserido na CPU e não pode ser alterado. Normalmente, o programador não precisa se preocupar com o formato usado. Contudo, há cirscunstâncias onde ele é importante.

1. Quando dados binários são transferidos entre diferentes computadores (tanto através de arquivos quanto através de uma rede).

2. Quando dados binários são escritos na memória como inteiros multibyte e então lidos de volta como bytes individuais ou *vice versa.*

1.5 Arquivo Modelo

A figura 1.7 mostra um arquivo modelo que pode ser usado como ponto de partida para escrever programas em assembly.

skel.asm

```
%include "asm_io.inc"
segment .data
;
; dados inicializados são colocados no segmento de dados aqui
;

segment .bss
;
; dados não inicializados são colocados no segmento bss
;

segment .text
        global  _asm_main
_asm_main:
        enter   0,0               ; rotina de configuração
        pusha

;
; código é colocado no segmento text. Não modifique o código antes ou
; depois desse comentário
;

        popa
        mov     eax, 0            ; Volta para o C
        leave
        ret
```

skel.asm

Figura 1.7: Modelo de Programa