

Provavelmente o grande obstáculo que a maioria dos novatos encontram quando tentam aprender a linguagem assembly é o constante uso dos sistemas numéricos binários e hexadecimais. Muitos programadores acham que números hexadecimais (ou hex¹) representam a prova absoluta que nunca foi a intenção de Deus fazer qualquer homem na Terra trabalhar com a linguagem assembly. Enquanto, a verdade é que, apesar de números hexadecimais serem um pouco diferente do qual você pode ter usado, as vantagens deles excedem as desvantagens por uma margem grande. Contudo, compreender estes sistemas numéricos é importante porque seu uso simplifica outros tópicos complexos incluindo álgebra booleana, projeto de lógica e representação numérica sinalizada, código de caracteres e pacote de dados.

1.0 Visão Geral do Capítulo

Este capítulo discute vários conceitos importantes incluindo os sistemas numéricos binários e hexadecimais, organização de dados binários (bits, nibble, bytes, words, e double words), sistemas numéricos sinalizados e não sinalizados, operações aritméticas, lógicas, trocas, e rotações em valores binários, campos bits e pacote de dados, e o conjunto de caracteres ASCII. Isto é a matéria-prima e o resto deste texto depende da compreensão destes conceitos. Se você já estiver familiarizado com estes termos provenientes de outros cursos ou estudo, você deve pelo menos folhear este material antes de proceder ao próximo capítulo. Se você não está familiarizado com este material, ou somente vagamente familiarizado, você deve estudá-lo cuidadosamente antes de proceder. *Todo do material neste capítulo é importante! Não pule-o!*

1.1 Sistemas Numéricos

A maioria dos sistemas de computadores modernos não representam valores numéricos usando o sistema decimal. Ao invés disso, eles usam o sistema numérico binário. Para entender as limitações aritméticas do computador, você tem que entender como os computadores representam os números.

1.1.1 Uma Revisão do Sistema Decimal

Com certeza você usa o sistema numérico decimal (base 10) há muito tempo. Quando você vê um número como “123”, você não pensa no valor 123; normalmente, você gera uma imagem mental de quantos elementos este valor representa. Na realidade, porém, o número 123 representa:

$$1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$

ou

$$100 + 20 + 3$$

Cada dígito que aparece à esquerda do ponto decimal representa um valor entre zero e nove vezes uma potência crescente de dez. Os dígitos que aparecem à direita do ponto decimal representam um valor entre zero e nove vezes uma potência negativa crescente de dez. Por

1. Hexadecimal é freqüentemente abreviado como hex embora, tecnicamente falando, hex significa base seis, e não base dezesseis.

exemplo, o valor 123.456 significa:

$$1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2} + 6 \cdot 10^{-3}$$

ou

$$100 + 20 + 3 + 0.4 + 0.05 + 0.006$$

1.1.2 O Sistema Numérico Binário

A maioria dos sistemas de computadores modernos (inclusive o IBM PC) funciona usando lógica binária. O computador representa valores usando dois níveis de voltagem (normalmente 0v e +5v). Com estes dois níveis nós podemos representar exatamente dois valores diferentes. Estes valores podem ser qualquer dois valores diferentes, mas por convenção nós usaremos os valores zero e um. Estes dois valores, coincidentemente, correspondem aos dois dígitos usados pelo sistema numérico binário. Desde então, há uma correspondência entre os níveis lógicos usados pelos 80x86 e os dois dígitos usados no sistema numérico binário, não deveria ser surpresa o porquê dos IBM PC's empregarem o sistema numérico binário.

O sistema numérico binário funciona igual o sistema numérico decimal, com duas exceções: o binário somente permite dígitos 0 e 1 (em vez de 0-9), e o sistema binário usa potências de dois em vez de potências de dez. Então, é muito fácil converter um número binário para decimal. Para cada "1" na seqüência binária, some 2^n onde "n" é a posição do dígito binário. Por exemplo, o valor binário 11001010_2 representa:

$$1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

$$=$$

$$128 + 64 + 8 + 2$$

$$=$$

$$202_{10}$$

Converter decimal para binário é ligeiramente mais difícil. Você tem que achar essas potências de dois nas quais, quando somadas, produzem o resultado decimal. O método mais fácil é trabalhar da maior potência de dois até 2^0 . Considere o valor decimal 1359:

- $2^{10} = 1024$, $2^{11} = 2048$. Então 1024 é a maior potência de dois menor que 1359. Subtraia 1024 de 1359 e comece o valor binário na esquerda com o dígito "1". Binário = "1", o resultado decimal é $1359 - 1024 = 335$.
- A próxima mais baixa potência de dois ($2^9 = 512$) é maior que o resultado acima, então acrescente um "0" no fim da seqüência binária. Binário = "10", o resultado Decimal ainda é 335.
- A próxima mais baixa potência de dois é 256 (2^8). Subtraia 256 de 335 e acrescente o dígito "1" no fim do número binário. Binário = "101", o resultado decimal é 79.
- 128 (2^7) é maior que 79, então ponha um "0" no fim da seqüência binária. Binário = "1010", e resultado decimal permanece 79.
- A próxima mais baixa potência de dois ($2^6 = 64$) é menor que 79, então subtraia 64 e acrescente "1" no fim da seqüência binária. Binário = "10101", o resultado decimal é 15.
- 15 é menor que a próxima potência de dois ($2^5 = 32$) então simplesmente acrescente um "0" no fim da seqüência binária. Binário = "101010", e o resultado decimal ainda é 15.
- 16 (2^4) é maior que o resto, então acrescente um "0" no fim da seqüência binária. Binário = "1010100", o resultado decimal é 15.

- 2^3 (oito) é menor que 15, então ponha outro dígito “1” no fim da seqüência binária. Binário = “10101001”, o resultado decimal é 7.
- 2^2 é menor que sete, então subtraia quatro de sete e acrescente outro “1” à seqüência binária. Binário = “101010011”, o resultado decimal é 3.
- 2^1 é menor que três, então acrescente “1” ao fim da seqüência binária e subtraia dois do valor decimal. Binário = “1010100111”, o resultado decimal é agora 1.
- Finalmente, o resultado decimal é 1, que é 2^0 , acrescente “1” no fim da seqüência binária. O resultado binário final é “10101001111”.

Números binários, embora eles tenham pouca importância em linguagens de alto nível, aparecem em todos lugares em programas de linguagem assembly.

1.1.3 Formatos Binários

No mais puro senso, todo número binário contém um número infinito de dígitos (ou bits que é a abreviação de dígitos binários ou binary digits). Por exemplo, nós podemos representar o número cinco por:

```

101          00000101          0000000000101          ...
0000000000000101

```

Qualquer de zeros nos primeiros bits pode preceder o número binário sem mudar seu valor.

Nós adotaremos a convenção que ignora qualquer zero inicial. Por exemplo, 101_2 representa o número cinco. Já que o 80x86 trabalha com grupos de oito bits, nós acharemos muito mais fácil completar com zero todo número binário para que se torne algum múltiplo de quatro ou oito bits. Então, seguindo esta convenção, nós representaríamos o número cinco como 0101_2 ou 00000101_2 .

Nos Estados Unidos, a maioria das pessoas separa cada três dígitos com uma vírgula para fazer números maiores mais fácil de ler. Por exemplo, 1,023,435,208 é muito mais fácil de ler e compreender do que 1023435208. Nós adotaremos uma convenção semelhante neste texto para números binários. Nós separaremos cada grupo de quatro bits binários com um espaço. Por exemplo, o valor 101011110110010 binário será escrito 1010 1111 1011 0010.

Nós freqüentemente faremos pacotes de vários valores em um mesmo número binário. Uma forma da instrução MOV no 80x86 (veja apêndice D) usa a codificação binária 1011 0rrr dddd dddd para fazer um pacote de três elementos em 16 bits: um código de operação de cinco bits (10110), um campo de registro de três bits (rrr), e um valor imediato de oito bits (dddd dddd). Por conveniência, nós associaremos um valor numérico a cada posição do bit. Nós associaremos cada bit como segue:

- 1) o bit mais à direita em um número binário é zero da posição do bit.
- 2) cada bit à esquerda determina o próximo número bit sucessivo.

Um valor binário de oito bits usa bits zero a sete:

$$X_7 \ X_6 \ X_5 \ X_4 \ X_3 \ X_2 \ X_1 \ X_0$$

Um valor binário de 16 bits usa bits de zero a quinze:

$$X_{15} X_{14} X_{13} X_{12} X_{11} X_{10} X_9 X_8 X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0$$

O bit zero normalmente é chamado de bit de *Baixa Ordem* (B.O) (ou L.O. de *Low Order*) . O bit mais à esquerda é chamado de bit de *Alta Ordem* (A.O) (ou H.O. de *High Order*) . Nós iremos nos referir aos bits intermediários pelos respectivos números da posição do bit .

1.2 Organização de Dados

Em matemática pura um valor pode assumir um número arbitrário de bits. Por outro lado, computadores geralmente trabalham com algum número específico de bits. Coleções comuns são bits únicos, grupos de quatro bits (chamados *nibbles*), grupos de oito bits (chamados *bytes*), grupos de 16 bits (chamados *words*), e outros mais. Os tamanhos não são arbitrários. Há uma boa razão para estes valores particulares. Esta seção descreverá os grupos de bit comumente usado nos chips Intel 80x86 .

1.2.1 Bits

A menor “unidade” de dados em um computador binário é um *bit*. Como um bit é capaz de representar somente dois valores diferentes (zero ou um) você pode ter a impressão de que há um número muito pequeno de elementos que você pode representar com um único bit. Não é verdade! Há um número infinito de elementos que você pode representar com um único bit.

Com um único bit, você pode representar qualquer dois elementos distintos. Exemplos incluem zero ou um, verdadeiro ou falso, ligado ou desligado, masculino ou feminino, e certo ou errado. Porém, você *não* está limitado a representar tipos de dados binários (quer dizer, esses objetos que têm somente dois valores distintos). Você pode usar um único bit para representar os números 723 e 1,245. Ou talvez 6,254 e 5. Você também pode usar um único bit para representar as cores vermelho e azul. Você pode representar dois objetos até mesmo sem relação entre eles com um único bit. Por exemplo, você pode representar a cor vermelho e o número 3,256 com um único bit. Você pode representar *qualquer* dois valores diferentes com um único bit. Porém, você pode representar *somente* dois valores diferentes com um único bit.

Para confundir ainda mais as coisas, bits diferentes podem representar coisas diferentes. Por exemplo, um bit poderia ser usado para representar os valores zero e um, enquanto um bit adjacente poderia ser usado para representar os valores verdadeiro e falso. Como você pode dizer o que o bit representa olhando para ele? A resposta, claro, que você não pode dizer. Mas isto ilustra a idéia completa atrás de estruturas de dados de computador: *dados é o que você define que ele seja*. Se você usa um bit para representar um valor booleano (verdadeiro/falso) então aquele bit (por definição) representa verdadeiro ou falso. Para o bit ter qualquer significado verdadeiro, você deve ser consistente. Quer dizer, se você estiver usando um bit para representar verdadeiro ou falso em um certo ponto em seu programa, você não deve usar o valor de verdadeiro/falso armazenado naquele bit para representar vermelho ou azul mais tarde.

Já que a maioria dos elementos que você vai modelar requer mais de dois valores diferentes, o bit sozinho não é o tipo de dado mais popular que você usará. Contudo, já que tudo consiste em grupos de bits, os bits farão um papel importante em seus programas. Claro que, há vários tipos de dados que requer dois valores distintos, e vai parecer que bits são importantes para eles. Porém, você logo verá que bits individuais são difíceis manipular, assim nós usaremos freqüentemente outros tipos de dados para representar valores booleanos.

1.2.2 Nibbles

Um *nibble* é uma coleção de quatro bits. Não seria uma estrutura de dados particularmente interessante com exceção de dois itens: números BCD (*binary coded decimal*) e números hexadecimais. Usa-se quatro bits para representar um único BCD ou dígito hexadecimal. Com um nibble, nós podemos representar até 16 valores distintos. No caso de números hexadecimais, os valores 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, e F são representados com quatro bits (veja “O Sistema Numérico Hexadecimal”). O BCD usa dez dígitos diferentes (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) e requer quatro bits. Na realidade, qualquer dezesseis valores distintos pode ser representado com um nibble, mas dígitos hexadecimais e BCD são os elementos primários que nós podemos representar com um único nibble.

1.2.3 Bytes

Sem dúvida, a estrutura de dados mais importante usada pelo microprocessador 80x86 é o byte. Um byte consiste em oito bits e é o menor dado endereçável (elemento de dados) no microprocessador 80x86. A memória principal e endereços de I/O nos 80x86 são todos endereços byte. Isto significa que o menor elemento que pode ser acessado individualmente por um 80x86 é um valor de oito bits. Para acessar qualquer coisa menor, requer que você leia o byte que contém o dado e descartasse os bits não desejados. Os bits em um byte são numerados normalmente de zero até sete usando a convenção da Figura 1.1.

O bit “0” é o bit de *baixa ordem* ou bit *menos significativo*, o bit 7 é o bit de *alta ordem* ou bit *mais significativo* do byte. Nós vamos nos referir a todos os outros bits pelo número da posição deles.



Figura 1.1: Numeração dos Bits em um Byte.

Note que um byte também contém exatamente dois *nibbles* (veja Figura 1.2).

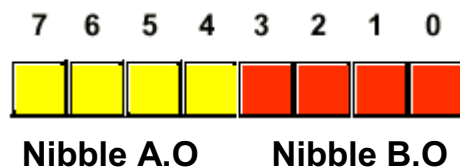


Figura 1.2: Dois Nibbles em um Byte.

Os bits 0..3 formam o *nibble de baixa ordem*, bits 4 ..7 formam o *nibble de alta ordem*. Como um byte contém exatamente dois nibbles, valores bytes requer dois dígitos hexadecimais.

Considerando que um byte contém oito bits, ele pode representar 2^8 , ou 256, valores diferentes. Geralmente, nós usaremos um byte para representar valores numéricos na faixa de 0..255, números sinalizados na faixa de -128 ..+127 (veja “Números Sinalizados e Não Sinalizados”), código de caracteres ASCII/IBM, e outros tipos de dados especiais que requerem não mais que 256 valores diferentes. Muitos tipos de dados têm menos que 256 elementos assim oito bits são normalmente suficientes.

Como os 80x86 são máquinas de byte endereçável (veja “Layout e Acesso de Memória”), mostra ser mais eficiente manipular um byte inteiro do que um bit ou nibble individual. Por isto, a maioria dos programadores usam um byte inteiro para representar tipos de dados que requerem não mais que 256 elementos, mesmo se oito bits for suficiente. Por exemplo, nós representaremos freqüentemente os valores booleanos verdadeiro e falso por 00000001_2 e 00000000_2 (respectivamente).

Provavelmente o uso mais importante para um byte acontece em código de caracteres. Caracteres digitados no teclado, exibidos na tela, e impresso na impressora, todos têm valores numéricos. Para permitir comunicar com o resto do mundo, o IBM PC usa uma variante do conjunto de caracteres ASCII (veja “O Conjunto de caracteres ASCII”). há 128 códigos definidos no conjunto de caracteres ASCII. O IBM usa os 128 valores restantes possíveis para código de character estendido incluindo caracteres europeus, símbolos gráficos, letras gregas, e símbolos de matemática. Veja o Apêndice A para os assuntos caracteres/códigos.

1.2.4 Words

Um word é um grupo de 16 bits. Nós numeraremos os bits em um word de zero até quinze. A numeração dos bits aparece na Figura 1.3.



Figura 1.3: Bits em um Word

Como o byte, O bit "0" é o bit de baixa ordem e bit 15 é o bit de alta ordem. Quando for referenciar os outros bits em um word use o número da posição do bit.

Note que um word contém exatamente dois bytes. Os Bits 0 até 7 formam o byte de baixa ordem, os bits 8 até 15 formam o byte de alta ordem (veja Figura 1.4).

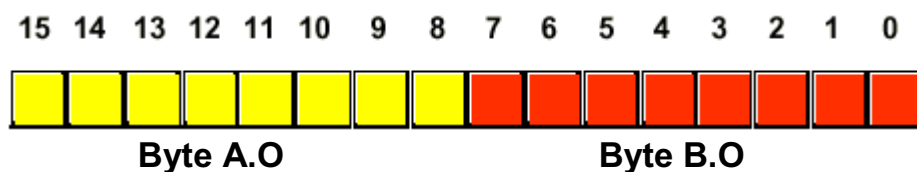


Figura 1.4: Os Dois Bytes em um Word

Naturalmente, um word pode ser dividido em quatro nibbles como mostrado em Figura 1.5.

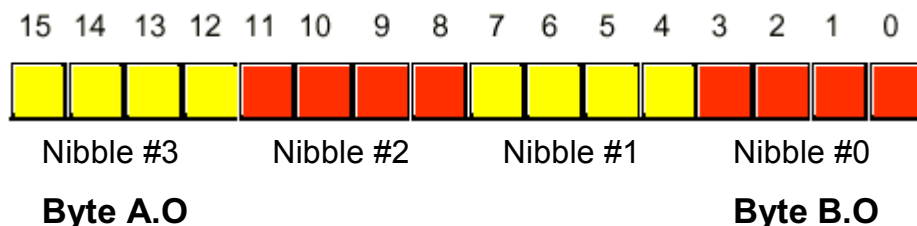


Figura 1.5: Nibbles em um Word

O nibble zero é o nibble de baixa ordem no word e nibble três é o nibble de alta ordem do word. Os outros dois nibble são “nibble um” ou “nibble dois.”

Com 16 bits, você pode representar 2^{16} (65,536) valores diferentes. Estes poderiam ser os valores na faixa de 0 ..65,535 (ou, como normalmente é o caso, -32,768 ..+32,767) ou qualquer outro tipo de dados sem mais de 65,536 valores. Os três usos principais para words são valores de inteiros, offsets, e valores de segmento (veja “Layout de Memória e Acesso” na para uma descrição de segmentos e offsets).

Words podem representar valores inteiros na faixa de 0 ..65,535 ou -32,768 ..32,767. Valores numéricos não sinalizados são representados pelo valor binário que corresponde aos bits no word. Valores numéricos sinalizados usam as duas formas complementares para valores numéricos (veja “Números Sinalizado e Não Sinalizados”). Valores de segmento que sempre são 16 bits longos constituem o *endereço de parágrafo* de um código, dados, extra, ou segmento de pilha em memória.

1.2.5 Double Words

Um double word é exatamente o que seu nome insinua, um par de words. Então, um double word é uma quantidade de 32 bits longos como mostrado na Figura 1.6.

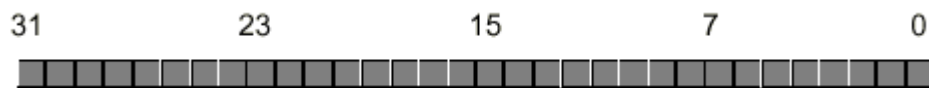


Figura 1.6: Número de Bits em um Double Word

Naturalmente, este double word pode ser dividido em um word de alta ordem e um word de baixa ordem, ou quatro bytes diferentes, ou oito nibble diferentes (veja Figura 1.7).

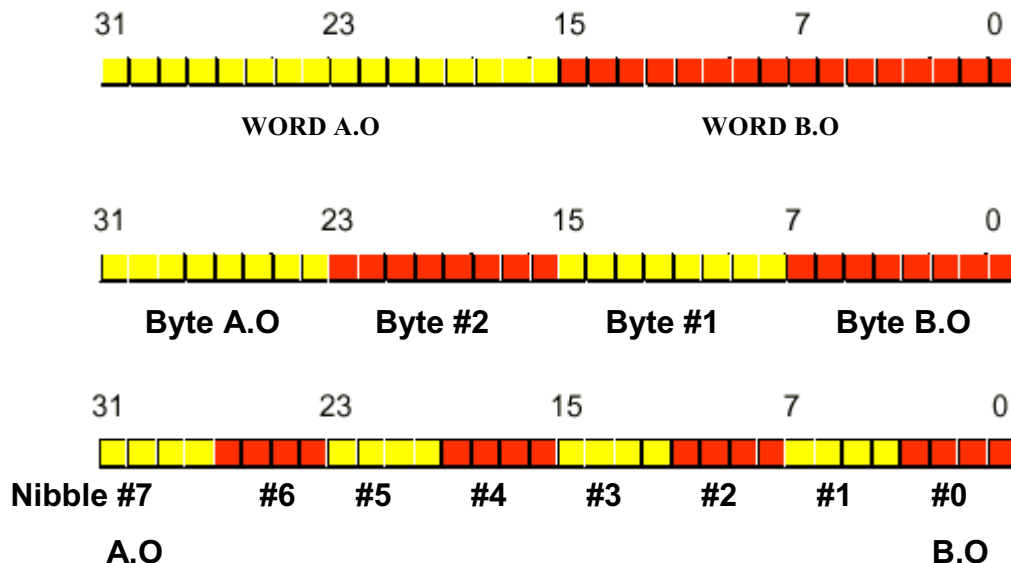


Figura 1.7: Nibbles, Bytes e Words em um Double Word

Double words podem representar todos os tipos de coisas diferentes. O primeiro na lista é um endereço segmentado. Outro item comum representado com um double word é um valor inteiro

de 32 bits (permite números não sinalizados na faixa de 0..4,294,967,295 ou números sinalizados na faixa de -2,147,483,648..2,147,483,647). Valores de ponto flutuantes de 32 bit também se encaixam em um double word. Na maioria das vezes, nós usaremos double word para organizar endereços segmentados.

1.3 O Sistema Numérico Hexadecimal

Um problema grande com o sistema binário é verbosidade. Representar o valor 202_{10} requer oito dígitos binários. A versão decimal requer somente três dígitos decimais e, então, representa números muito mais compacto que o sistema numérico binário. Este fato não foi esquecido pelos engenheiros que projetaram sistemas de computador binários. Ao lidar com valores grandes, os números binários ficam rapidamente difíceis de controlar. Infelizmente, o computador pensa em binário, sendo assim a maioria das vezes é conveniente usar o sistema numérico binário. Embora nós podemos converter entre decimal e binário, a conversão não é uma tarefa trivial. O sistema numérico hexadecimal (base 16) resolve estes problemas. Números hexadecimais oferecem as duas características para as que nós estamos procurando: eles são muito compactos, e são simples converte-los para binário e vice-versa. Por causa disto, a maioria dos sistemas de computadores de hoje usam o sistema numérico hexadecimal². Como a raiz (base) de um número hexadecimal é 16, cada dígito hexadecimal à esquerda do ponto hexadecimal representa algum valor vezes uma potência sucessiva de 16. Por exemplo, o número 1234_{16} é igual a:

$$1 \cdot 16^3 + 2 \cdot 16^2 + 3 \cdot 16^1 + 4 \cdot 16^0$$

ou

$$4096 + 512 + 48 + 4 = 4660_{10}.$$

Cada dígito hexadecimal pode representar um de dezesseis valores entre 0 e 15_{10} . Mas somente existem dez dígitos decimais, nós precisamos inventar seis dígitos adicionais para representar os valores na faixa 10_{10} até 15_{10} . De preferência então criar novos símbolos para estes dígitos, nós usaremos as letras de A até F. Os seguintes exemplos são todos de números hexadecimais válidos:

1234_{16} $DEAD_{16}$ $BEEF_{16}$ $0AFB_{16}$ $FEED_{16}$ $DEAF_{16}$

Considerando que nós precisaremos freqüentemente entrar com números hexadecimais no sistema de computador, nós precisaremos de um mecanismo diferente por representar números hexadecimais. Afinal de contas, na maioria dos sistemas de computador você não pode entrar em uma subscrição para denotar a raiz do valor associado. Nós adotaremos as seguintes convenções:

- Todos os valores numéricos (sem levar em consideração a raiz deles) começam com um dígito decimal.
- Todo valor hexadecimal termina com a letra “h”, por exemplo, $123A4h^3$.
- Todo valor binário termina com a letra “b.”
- Números decimais podem ter um sufixo “t” ou “d”.

Exemplos de números hexadecimais válidos:

2. Equipamentos digitais estão fora deste contexto. Eles ainda usam números octais. Um legado do dias quando produzirem máquinas de 12 bits.

3. Atualmente seguir valores hexadecimais com um “h” é uma convenção da Intel. Os assemblers 68000 e 65c816 usados no Macintosh e Apple II denotam números hexadecimais colocando ur prefixo com o símbolo “\$”.

1234h 0DEADh 0BEEFh 0AFBh 0FEEDh 0DEAFh

Como você pode ver, números hexadecimais são compactos e fáceis ler. Além disso, você pode facilmente converter entre hexadecimal e binário. Considere a tabela seguinte:

Tabela 1: Conversão Binário/Hexadecimal

Binário	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Esta tabela fornece toda a informação que você sempre precisará para converter qualquer número hexadecimal em um número binário ou vice-versa.

Para converter um número hexadecimal em um número binário, simplesmente substitua os quatro bits correspondentes para cada dígito hexadecimal no número. Por exemplo, para converter 0ABCDh em um valor binário, simplesmente converta cada dígito hexadecimal de acordo com a tabela acima:

0	A	B	C	D	Hexadecimal
0000	1010	1011	1100	1101	Binário

Para converter um número binário para o formato hexadecimal também é fácil. O primeiro passo é completar o número binário com zeros para ter certeza que há um múltiplo de quatro bits no número. Por exemplo, determinado o número binário 1011001010, o primeiro passo seria adicionar dois bits à esquerda do número de forma que ele possua 12 bits. O valor binário convertido é 001011001010. O próximo passo é separar o valor binário em grupos de quatro bits, por exemplo, 0010 1100 1010. Finalmente, olhe estes valores binários na tabela acima e substitua os dígitos hexadecimais pelos valores apropriados, por exemplo, 2CA. Compare com a dificuldade de conversão entre decimal e binário ou decimal e hexadecimal!

Como converter entre hexadecimal e binário é uma operação que você precisará executar inúmeras vezes, é bom perder alguns minutos e memorizar a tabela acima. Até mesmo se você tiver uma calculadora que faça a conversão para você, você achará conversão manual muito mais rápida e mais conveniente quando trabalhar com números binários e hex.

1.4 Operações Aritméticas em Números Binários e Hexadecimais

Há várias operações que nós podemos executar em números binários e hexadecimais. Por exemplo, nós podemos somar, subtrair, multiplicar, dividir, e executar outras operações aritméticas. Embora você não precise se tornar um perito nisto, você deve ser capaz de, rapidamente, executar estas operações usando um pedaço de papel e um lápis.. Há pouco tendo dito que você poderia executar estas operações manualmente, o modo correto para executar tais operações aritmético é com uma calculadora que faça para você. Há várias calculadoras no mercado; a tabela seguinte lista alguns dos fabricantes que produzem tais dispositivos:

Fabricantes de Calculadoras Hexadecimais:

- CASIO
- HEWLETT-PACKARD
- Sharp
- Texas Instruments

Esta lista é imensa. Outros fabricantes de calculadora provavelmente produzem bons dispositivos também. Os dispositivos da Hewlett-Packard são indiscutivelmente os melhores do grupo. Porém, eles são mais caros que os outros. A Sharp e Casio produzem unidades vedidas por um preço abaixo de \$50. Se você planeja fazer qualquer programação em linguagem assembly, possuir um destas calculadoras é essencial.

Outra alternativa para ter uma calculadora hexadecimal é obter um programa TSR (Terminate and Stay Resident, ou seja, Termine e Continue Residente) como SideKick™ que contém uma calculadora embutida. Porém, a menos que você já tenha um destes programas, ou caso você precise de algumas das outras características que eles ofereçam, tais programas não são particularmente bons já que eles custam mais que uma calculadora atual e não são conveniente para uso.

Para entender por que você deve gastar o dinheiro em uma calculadora, considere o seguinte problema aritmético:

$$\begin{array}{r} 9h \\ +1h \\ ---- \end{array}$$

Você provavelmente está tentado escrever a resposta “10h” como a solução para este problema. Mas isso não está correto! A resposta correta é dez, que são “0Ah”, e não dezesseis que é “10h”. Um problema semelhante existe com o problema aritmético:

$$\begin{array}{r} 10h \\ - 1h \\ ---- \end{array}$$

Você provavelmente está tentado responder “9h” embora a verdadeira resposta seja “0Fh”. Lembre-se, este problema está perguntando “qual é a diferença entre dezesseis e um?” A resposta, claro que, é quinze que é “0Fh.”

Até mesmo se os dois problemas acima não o aborrece, em uma situação de stress seu cérebro trocará para o modo decimal quando você estiver pensando em qualquer outra coisa e você produzirá o resultado incorreto. Moral da história – se você tem que fazer uma computação aritmética que usa números hexadecimais manualmente, vai perder seu tempo e deve ser cuidadoso com isto. Qualquer um que, converta-os para decimal, complete a operação em decimal, e converta-os de novo para hexadecimal

Você nunca deve executar computações aritméticas binárias. Já que números binários normalmente contêm seqüências longas de bits, há muito mais chances de você cometer um erro. Sempre converta números binários a hex, execute a operação em hex (preferivelmente com uma calculadora de hex) e converta o resultado de novo para binário, se necessário.

1.5 Operações Lógicas nos Bits

Há quatro operações lógicas principais que nós precisaremos executar em números hexadecimais e binários: AND, OR, XOR (OR-exclusivo), e NOT. Ao contrário das operações aritméticas, não é necessário uma calculadora hexadecimal executar estas operações. É freqüentemente mais fácil fazê-los manualmente do que usar um dispositivo eletrônico para computá-los. A operação lógica AND é uma operação diádica⁴ (isto significa que aceita exatamente dois elementos). Estes elementos são simples bits binários (base 2). A operação AND é:

0 AND 0 = 0
 0 AND 1 = 0
 1 AND 0 = 0
 1 AND 1 = 1

Um modo compacto para representar a operação lógica AND está numa tabela de verdade. Uma tabela de verdade tem a seguinte forma:

Tabela 2: Tabela de Verdade AND

AND	0	1
0	0	0
1	0	1

Isto é igual as tabelas de multiplicar que você viu na escola primária. A coluna da esquerda e a coluna no topo representam valores de entrada à operação AND. O valor localizado à interseção da fila e coluna (para um par particular de valores de entrada) é o resultado de lógico da operação AND entre esses dois valores. Em inglês, a operação lógica AND é, “Se o primeiro elemento é um e o segundo elemento é um, o resultado é um; caso contrário o resultado é zero.”

Um fato importante a ser notado sobre a operação lógica AND é que você pode usá-la para forçar um resultado zero. Se um dos operadores for zero, o resultado sempre é zero embora o outro elemento seja um. Na tabela de verdade acima, por exemplo, a coluna rotulada com uma entrada zero contém somente zeros e a linha somente rotulada com um zero contém resultados zero. Reciprocamente, se um elemento contiver 1, o resultado é exatamente o valor do segundo operador. Estas características da operação AND são muito importantes, particularmente ao trabalhar com seqüências de bit e nós quisermos forçar bits individuais na seqüência para zero. Nós investigaremos estes usos da operação lógica AND na próxima seção.

A operação lógica OR também é uma operação diádica. Sua definição é:

4. Muitos textos chamam isto de operação binária. O termodiádico significa mesma coisa e evita confusão com o sistema numérico binário.

$0 \text{ ou } 0 = 0$
 $0 \text{ ou } 1 = 1$
 $1 \text{ ou } 0 = 1$
 $1 \text{ ou } 1 = 1$

A tabela de verdade para a operação OR leva a seguinte forma:

Tabela 3: Tabela de Verdade OR

OR	0	1
0	0	1
1	1	1

Coloquialmente, a operação lógica OR é, “Se o primeiro elemento ou o segundo elemento (ou ambos) é um, o resultado é um; caso contrário o resultado é zero.” Isto também é conhecido como a operação *OR-inclusivo*.

Se um dos elementos para a operação lógica OR for um 1, o resultado sempre é 1 independente do valor do segundo elementos. Se um operador for zero, o resultado sempre é o valor do segundo elementos. Como a operação lógica AND, este é um efeito colateral importante da operação lógica OR que tornará bastante útil ao trabalhar com seqüências de bit (veja a próxima seção).

Note que há uma diferença entre esta forma da operação lógica OR inclusiva e o significado padrão do inglês. Considere a frase “eu vou para a loja ou eu vou para o parque.” Tal declaração insinua que a pessoa que falou vai para a loja ou para o parque mas não para ambos os lugares. Então, a versão inglesa do OR lógico é ligeiramente diferente que a operação de OR-inclusivo; realmente, é mais próximo à operação de OR-exclusivo. A operação lógica XOR (OR-exclusivo) também é uma operação diádica. Está definido como segue:

$0 \text{ xor } 0 = 0$
 $0 \text{ xor } 1 = 1$
 $1 \text{ xor } 0 = 1$
 $1 \text{ xor } 1 = 0$

A tabela de verdade para a operação XOR leva a seguinte forma:

Tabela 4: Tabela de Verdade XOR

XOR	0	1
0	0	1
1	1	0

Em inglês, a operação lógica XOR é, “Se o primeiro elemento ou o segundo elemento, mas não ambos, é um, o resultado é um; caso contrário o resultado é zero.” Note que a operação OR-exclusivo é mais próxima ao significado inglês da palavra “ou” que é a operação lógica OR.

5. Monádico significa que a operação tem somente um elemento.

Se um dos elementos para a operação lógica OR-exclusivo for um 1, o resultado sempre é o inverso do outro elemento; quer dizer, se um elemento for um, o resultado é zero se o outro elemento for um e o resultado é um se o outro elemento for zero. Se o primeiro elemento for zero, então o resultado é exatamente o valor do segundo elemento. Esta característica o permite inverter seqüências bits seletivamente.

A operação lógica NOT é uma operação monádica⁵ (isto significa que aceita somente um elemento). Ela é:

$$\begin{aligned}\text{NOT } 0 &= 1 \\ \text{NOT } 1 &= 0\end{aligned}$$

A tabela de verdade para a operação NOT tem a seguinte forma:

Tabela 5: Tabela de Verdade NOT

NOT	0	1
	1	0

1.6 Operações Lógicas em Números Binários e Seqüências de Bit

Como descrito na seção anterior, as funções lógicas somente trabalham com operadores de bit simples. Já que os 80x86 usam grupos de oito, dezesseis, ou trinta e dois bits, nós precisamos estender a definição destas funções para lidar com mais de dois bits. Funções lógicas nos 80x86 operam fundamentalmente *bit-a-bit* (ou *bitwise*). Dado dois valores, estas funções trabalham no bit zero que produzindo como resultado o bit zero. Eles trabalham em bit um nos valores de entrada produzindo o resultado bit um, etc. por exemplo, se você quiser computar o AND lógico dos seguintes dois números de oito bits, você executaria a operação lógica AND independentemente das outras em cada coluna:

```

1011 0101
1110 1110
-----
1010 0100

```

Esta forma de execução bit-a-bit pode ser aplicada facilmente como as outras operações lógicas.

Considerando que nós definimos operações lógicas em termos de valores binários, você achará muito mais fácil executar operações lógicas em valores binários do que em valores em outras bases. Então, se você quiser executar uma operação lógica em dois números hexadecimais, você deve primeiro convertê-los para binário. Isto se aplica à maioria das operações lógicas básicas em números binários (por exemplo, AND, OR, XOR, etc.).

A habilidade de forçar bits para zero ou para um usando as operações lógicas AND/OR e a habilidade para inverter os bits usando a operação lógica XOR é muito importante ao trabalhar

com seqüências de bits (por exemplo, números binários). Estas operações permitem você manipular seletivamente certos bits dentro de algum valor enquanto não afeta os outros bits. Por exemplo, se você tem um valor binário 'X' de oito bits e você quer garantir que os bits quatro até sete contenha zeros, você pode com o AND fazer o valor 'X' ser o valor binário 0000 1111. Esta operação lógica bitwise AND força os quatro bits de A.O. para zero e deixa os quatro bits de B.O. de 'X' inalterados. Igualmente, você pode forçar o bit de B.O. de 'X' para um e inverter o bit número dois de 'X' através da operação OR 'X' com 0000 0001 e através da operação OR-exclusivo 'X' com 0000 0100, respectivamente. Usando as operações lógicas AND, OR, e de XOR para manipular seqüências de bit desta maneira é saber como *maskar* seqüências de bit.

Nós usaremos o termo *maskar* porque nós podemos usar certos valores (um para AND, zero para OR/XOR) para 'desmaskar' certos bits da operação quando forçar os bits para zero, um, ou o inverso deles.

1.7 Números Sinalizados e Não Sinalizados

Até agora, nós tratamos números binários como valores não sinalizados. O número ...00000 binário representa zero, ...00001 representam um, ...00010 representam dois, e assim por diante até o infinito. E os números negativos? Foram dados valores sinalizados em torno das seções anteriores e nós mencionamos o sistema numérico complementar de dois, mas nós não discutimos como representar números negativos usando o sistema numérico binário. É sobre isso que esta seção fala!

Para representar números sinalizados usando o sistema numérico binário nós temos que colocar uma restrição em nossos números: eles têm que ser um número finito e ter um número fixo de bits. Até onde os 80x86 vão, isto não é bem de uma restrição, afinal de contas, os 80x86 podem endereçar somente um número finito de bits. Para nossos propósitos, nós vamos limitar o número de bits restritamente a oito, 16, 32, ou algum outro número pequeno de bits.

Com um número fixo de bits nós podemos representar somente um certo número de objetos. Por exemplo, com oito bits nós podemos representar somente 256 objetos diferentes. Valores negativos são objetos com propriedades, do mesmo jeito que os números positivos. Então, nós teremos que usar alguns dos 256 valores diferentes para representar números negativos. Em outras palavras, nós temos que usar alguns dos números positivos para representar números negativos. Para fazer as coisas direito, nós nomearemos a metade das possíveis combinações para os valores negativos e metade para os valores positivos. Assim nós podemos representar os valores negativos -128.. -1 e os valores 0 ..127 positivos com um único byte de oito bits⁶. Com um word de 16 bits nós podemos representar valores na faixa de -32,768 ..+32,767. Com um double word de 32 bits nós podemos representar valores na faixa de -147,483,648 ..+2,147,483,647. em geral, com n bits nós podemos representar os valores sinalizados na faixa de -2^{n-1} até $+2^{n-1}-1$.

Certo, então nós podemos representar valores negativos. Como faremos isto exatamente? Bem, há muitos modos, mas o microprocessador 80x86 usa notações complementares de dois. Nos sistemas complementares de dois, o bit de A.O de um número é um *bit de sinal*. Se o bit de A.O é zero, o número é positivo; se o bit de A.O é um, o número é negativo. Exemplos:

Para números 16 bits:

8000h é negativos porque o bit de A.O é um.

100h é positivos porque o bit de A.O é zero.

6. Tecnicamente, zero nem é positivo nem negativo. Por razões técnicas (devido ao hardware envolvido), nós consideraremos zero como o número positivo.

7FFFh é positivos.

0FFFFh é negativo.

0FFFh é positivo.

Se o bit de A.O é zero, então o número é positivo e é armazenado como um valor binário padrão. Se o bit de A.O é um, então o número é negativo e é armazenado na forma complementar de dois. Para converter um número positivo a seu negativo, forma complementar de dois, você usa o algoritmo seguinte:

1) inverta todos os bits no número, ou seja, aplique a função lógica NOT.

2) Some um ao resultado invertido.

Por exemplo, computar o número de oito bits equivalente a -5:

```
0000 0101 cinco (em binário).
1111 1010 inverte todos os bits.
1111 1011 soma um para obter resultado.
```

Se nós pegarmos menos cinco e executarmos a operação complementar de dois nele, nós adquirimos o valor original, 00000101, novamente, da mesma maneira que nós esperávamos:

```
1111 1011 complementar de dois para -5.
0000 0100 inverte todos os bits.
0000 0101 soma um para obter resultado (+5).
```

Os exemplos seguintes fornecem alguns valores sinalizados de 16 bits positivo e negativo:

```
7FFFh: +32767, o maior número 16 bits positivo.
8000h: -32768, o menor número 16 bits negativo.
4000h: +16,384.
```

Para converter os números acima para a contraparte negativa deles (ou seja, negá-los), faça o seguinte:

```
7FFFh: 0111 1111 1111 1111 +32,767t
        1000 0000 0000 0000 inverte todos os bits (8000h)
        1000 0000 0000 0001 soma um (8001h ou -32,767t)

8000h: 1000 0000 0000 0000 -32,768t
        0111 1111 1111 1111 inverte todos os bits (7FFFh)
        1000 0000 0000 0000 soma um (8000h ou -32768t)

4000h: 0100 0000 0000 0000 16,384t
        1011 1111 1111 1111 inverte todos os bits (BFFFh)
        1100 0000 0000 0000 soma um (0C000h ou -16,384t)
```

8000h invertido se torna 7FFFh. Depois de somar um nós obtemos 8000h! Espere, o que está acontecendo aqui? - (- 32,768) é -32,768 ? Claro que não. Mas o valor +32,768 não pode ser representado com um número sinalizado de 16 bits, então nós não podemos negar o menor valor negativo. Se você tentar fazer esta operação, o microprocessador 80x86 acusará um overflow aritmético sinalizado.

Por que se aborrecer com tal sistema numérico miserável? Por que não usar o bit de A.O como uma marca de sinal (flag), armazenando o positivo equivalente do número nos bits restantes? A resposta é o hardware. Como pode se notar, negar valores é o único trabalho tedioso. Com o sistema complementar de dois, a maioria das outras operações será tão fácil quanto o sistema binário. Por exemplo, suponha você execute a adição $5+(-5)$. O resultado é zero. Considere o que acontece quando nós somarmos estes dois valores no sistema complementar de dois:

```
00000101
11111011
-----
1 00000000
```

Nós terminamos com um carry no nono bit e todos os outros bits são zero. Como se nota, se nós colocarmos o carry fora do bit de A.O, somando dois valores sinalizados sempre produz o resultado correto ao usar o sistema numérico complementar de dois. Isto significa que nós podemos usar o mesmo hardware para adição e subtração sinalizada e não sinalizadas. Este não seria o caso com alguns outros sistemas numéricos.

Com exceção das perguntas ao término deste capítulo, você não precisará fazer a operação complementar de dois manualmente. O microprocessador 80x86 fornece uma instrução, NEG (negação) que executa esta operação para você. Além disso, todas as calculadoras hexadecimais executarão esta operação na tecla de mudança de sinal (+ / - ou CHS). Contudo, fazer um complementar de dois manualmente é fácil e você deve saber fazer isto.

Mais uma vez, você deve notar que os dados representados por um conjunto de bits binários depende completamente do contexto. O valor binário de oito bits 11000000b poderia representar um caracter IBM/ASCII, poderia representar o valor decimal 192 não sinalizado, ou poderia representar o valor decimal -64 sinalizado, etc. Como programador, é sua responsabilidade usar estes dados consistentemente.

1.8 Sinal e Extensão de Zero

Considerando que os formatos inteiros complementares de dois têm um tamanho fixo, um pequeno problema se desenvolve. O que acontece se você precisar converter um valor complementar de dois de oito bits para 16 bits? Este problema, e seu contrário (converter um valor de 16 bits para oito bits) pode ser realizado pelas operações de *extensão de sinal* e *contração*. Os 80x86 trabalham com tamanho fixo de valores, até mesmo ao processar números binários não sinalizados. Extensão de zero permite converter pequenos valores não sinalizados para valores não sinalizados maiores. Considere o valor "-64".

O valor complementar de dois de oito bits para este número é 0C0h. O número de 16 bits equivalente deste número é 0FFC0h. Agora considere o valor "+64". As versões oito e 16 bits deste valor são 40h e 0040h. A diferença entre os números de oito e 16 bits pode ser descrita pela regra: "Se o número é negativo, o byte de A.O. do número de 16 bits contém 0FFh; se o número é positivo, o byte de A.O. do número de 16 bits é zero."

Para sinalizar um valor estendido de algum número de bits para um número de bits maior é fácil, copie o bit de sinal em todos os bits adicionais no novo formato. Por exemplo, para fazer a extensão de sinal de um número de oito bits para um número de 16 bits, copie o bit sete do número de oito bits nos bits 8..15 do número de 16 bits. Para fazer a extensão de sinal de um número de 16 bits para um double word, simplesmente copie o bit 15 nos bits 16..31 do double word.

A extensão de sinal é requerida ao manipular valores sinalizados de comprimentos variados. Frequentemente você precisará acrescentar uma quantidade byte a uma quantidade

word. Você deve fazer a extensão de sinal de um byte para um word antes da operação acontecer. Outras operações (multiplicação e divisão, em particular) podem requerer uma extensão de sinal para 32 bits. Você não deve fazer a extensão de sinal de valores não sinalizados. Exemplos de extensão de sinal:

Oito Bits	Dezesseis Bits	Trinta e dois Bits
80h	FF80h	FFFFFFF80h
28h	0028h	00000028h
9Ah	FF9Ah	FFFFFFF9Ah
7Fh	007Fh	0000007Fh
---	1020h	00001020h
---	8088h	FFFF8088h

Para fazer a extensão de sinal de um byte não sinalizado você tem que fazer a extensão de sinal do valor zero. Extensão de zero é muito fácil – apenas ponha um zero no(s) byte(s) de A.O. do menor operador. Por exemplo, para fazer a extensão de zero o valor 82h para 16 bits você acrescenta um zero ao byte de A.O. que resulta 0082h.

Oito Bits	Dezesseis Bits	Trinta e dois Bits
80h	0080h	00000080h
28h	0028h	00000028h
9Ah	009Ah	0000009Ah
7Fh	007Fh	0000007Fh
---	1020h	00001020h
---	8088h	00008088h

A contração de sinal consiste em, converter um valor com algum número de bits para o valor idêntico com um número menor de bits, é um pouco mais problemático. Extensão de sinal nunca falha. Dado um bit- m de valor sinalizado você sempre pode convertê-lo a um número bit- n (onde $n > m$) usando extensão de sinal. Infelizmente, dado um número bit- n , você não pode sempre convertê-lo a um número bit- m se $m < n$. Por exemplo, considere o valor -448. Como um número hexadecimal de 16 bits, sua representação é 0FE40h. Infelizmente, a magnitude deste número é muito grande para ajustar em um valor de oito bits, então você não pode contrair o sinal dele para oito bits. Este é um exemplo de uma condição de overflow que acontece em conversão.

Para contrair o sinal corretamente de um valor para outro, você tem que olhar para o(s) byte(s) de A.O. que você quer descartar. Todos os bytes de A.O. que você deseja remover devem conter zero ou 0FFh. Se você encontrar qualquer outro valor, você não pode contrair-lo sem overflow. Finalmente, o bit de A.O. do valor do seu resultado tem que igualar *todo* bit que você removeu do número. Exemplos (16 bits para oito bits):

FF80h pode ser feito a contração de sinal para 80h
 0040h pode ser feito a contração de sinal para 40h
 FE40h não pode ser feito a contração de sinal para 8 bits.
 0100h não poder ser feito a contração de sinal para 8 bits.

1.9 Trocas e Rotações

Outro conjunto de operações lógicas na qual aplicam as seqüências de bit são as operações de *troca* e *rotação*. Estas duas categorias serão divididas mais adiante em *trocas esquerdas*, *rotações esquerdas*, *trocas direitas*, e *rotações direitas*. Estas operações são extremamente úteis a programadores de linguagem assembly.

A operação de troca esquerda move cada seqüência de bit uma posição à esquerda (veja Figura 1.8).

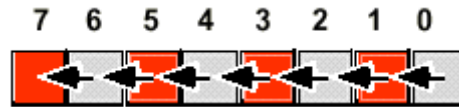


Figura 1.8: Operação de Rotação Esquerda

O bit zero passa para a posição um, o valor anterior na posição um passa para a posição dois, etc. Há, claro que, duas perguntas que naturalmente surgem: “O que acontece com o bit zero?” e “Pra onde vai o bit sete?” Bem, isso depende do contexto. Nós trocaremos o zero de valor no bit de B.O , e o valor anterior do bit sete será carregado pra fora desta operação.

Note que trocar um valor à esquerda é a mesma coisa como multiplicá-lo por sua raiz. Por exemplo, trocar um número decimal uma posição à esquerda (somando um zero à direita do número) efetivamente multiplica-o por dez (a raiz):

1234 SHL 1 = 12340 (SHL 1 = troca uma posição esquerda)

Como a raiz de um número binário é dois, trocando-o à esquerda multiplica-o por dois. Se você trocar à esquerda duas vezes um valor binário, você multiplica-o duas vezes por dois (isto é, você multiplica-o por quatro). Se você trocar um valor binário à esquerda três vezes, você multiplica-o por oito ($2 \times 2 \times 2$). Em geral, se você troca um valor à esquerda n vezes, você multiplica aquele valor por 2^n .

Uma operação de troca direita trabalha do mesmo modo, exceto que nós vamos mudar os dados para a direção oposta. O bit sete passa a bit seis, bit seis passa a bit cinco, bit cinco passa a bit quatro, etc. Durante uma troca direita, nós passaremos o bit zero para bit sete, e bit zero será o carregado para fora da operação (veja Figura 1.9).

Considerando que uma troca esquerda é equivalente a uma multiplicação por dois, não deve ser surpresa que uma troca direita é comparável a uma divisão por dois (ou, em geral, uma divisão pela raiz do número). Se você executar n trocas direitas, você dividirá aquele número por 2^n .



Figura 1.9: Operação de Troca Direita

Há um problema em relação a trocas direitas no que diz respeito a divisão: como descrito acima uma troca direita é apenas equivalente a uma divisão por dois *não sinalizada*. Por exemplo, se você troca a representação não sinalizada de 254 (0FEh) uma posição à direita, você obtém 127 (07Fh), exatamente o que você esperava. Porém, se você troca a representação binária de -2 (0FEh) uma posição à direita, você obtém 127 (07Fh) o que *não* está correto. Este problema acontece porque nós estamos trocando um bit zero em um bit sete. Se bit sete tinha 1 anteriormente, nós estamos mudando-o de um número negativo para um número positivo. Não é

7. Não há nenhuma necessidade para a troca aritmética esquerda. A troca esquerda padrão funciona tanto para números sinalizados e não sinalizados, não causando nenhum overflow.

uma coisa boa quando dividi-lo por dois.

Para usar a troca direita como um operador de divisão, nós temos que definir uma terceira operação de troca: *troca aritmética direita*⁷. Uma troca aritmética direita funciona como a operação de troca direita normal (uma *troca lógica direita*) com uma exceção: em vez de trocar um zero com um bit sete, uma troca aritmética direita deixa o bit sete sozinho, quer dizer, durante a operação de troca ele não modifica o valor de bit sete como mostra a Figura 1.10.

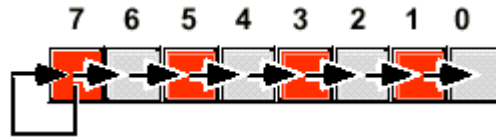


Figura 1.10: Operação Aritmética de Rotação Direita

Isto geralmente produz o resultado que você espera. Por exemplo, se você executa a troca aritmética direita em -2 (0FEh) você obtém -1 (0FFh). Porém, lembre-se de uma coisa sobre troca aritmética direita. Esta operação sempre arredonda os números para o *inteiro mais próximo menor ou igual ao resultado atual*. Baseado em experiências com linguagens de programação de alto nível e as regras padrões de truncção inteira, a maioria das pessoas dizem que isto significa que uma divisão sempre trunca para zero. Mas este simplesmente não é o caso. Por exemplo, se você aplica a operação de troca aritmética direita em -1 (0FFh), o resultado é -1, e não zero. -1 é menor que zero então a troca aritmética direita arredonda para menos. Isto não é um “bug” na troca aritmética direita. Isto é a divisão inteira de modo definido. A instrução de divisão de inteiro no 80x86 também produz este resultado.

Outro par de operações úteis são de *rotação esquerda* e *rotação direita*. Estas operações se comportam como as operações de trocas esquerda e direita com uma diferença principal: o bit trocado de um extremo vai para o outro.

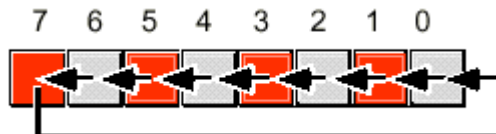


Figura 1.11: Operação de Rotação Esquerda

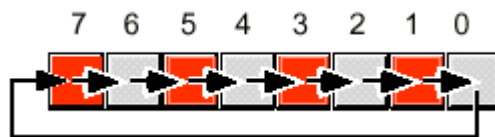


Figura 1.12: Operação de Rotação Direita

1.10 Campos Bit e Pacote de Dados

Embora os 80x86 operarem eficazmente em byte, word, e tipos de dados de double word, ocasionalmente você precisará trabalhar com um tipo de dados que usa algum número de tipos de bits, ocasionalmente você precisará trabalhar com um tipo de dados que usa algum número de bits diferente de oito, 16, ou 32. Por exemplo, considere uma data da forma “4/2/88”. Compreende três

valores numéricos para representar esta data: um mês, dia, e um valor ano. Meses, claro que, assume os valores 1..12. Necessitará de quatro bits pelo menos (máximo de dezesseis valores diferentes) para representar o mês. Dias variam entre 1..31. Assim usará cinco bits (máximo de 32 valores diferentes) representar a entrada de dia. O valor ano, assumindo que nós estamos trabalhando com valores na faixa de 0..99, requer sete bits (que pode ser usado para representar até 128 valores diferentes). Quatro mais cinco mais sete são 16 bits, ou dois bytes. Em outra palavra, nós podemos empacotar nosso dados de data em dois bytes ao invés dos três que seria necessários se nós usássemos um byte separado para cada valor do mês, dia, e ano. Isto economiza um byte de memória para cada data armazenada que poderia ser uma economia significativa se você precisar armazenar muitas datas. Os bits poderiam ser organizados como mostrado.



Figura 1.13: Formato de Pacote Data

MMMM representa os quatro bits que compõem o valor de mês, DDDDD representa os cinco bits que compõem o dia, e YYYYYYY é os sete bits que incluem o ano. Cada coleção de bits que representa um elemento de dados é um campo bit. 2 de abril de 1988 seria representado como 4158h:

0100 00010 1011000 = 0100 0001 0101 1000b ou 4158h
4 2 88

Embora valores empacotados são *eficiente em espaço* (quer dizer, muito eficiente em termos de uso de memória), eles computacionalmente são *ineficientes* (lento!). A razão? Usa-se instruções extras para desempacotar o dados nos vários campos de bit. Estas instruções extras levam tempo adicional para executar (e bytes adicionais para manter as instruções); consequentemente, você tem que considerar cuidadosamente se campos de pacote de dados o ajudará em qualquer coisa.

Os exemplos práticos de tipos de pacote de dado são muitos. Você poderia empacotar oito valores booleanos em um único byte, você poderia empacotar dois dígitos BCD em um byte, etc.

1.11 O Conjunto de Caracteres ASCII

O conjunto de caracteres ASCII (excluindo os caracteres estendidos definidos pela IBM) é dividido em quatro grupos de 32 caracteres. Os primeiros 32 caracteres, código ASCII que vai de 0 até 1Fh (31), formam um conjunto especial de caracteres não-imprimíveis chamado de caracteres de controle. Nós os chamamos caracteres de controle porque eles executam que várias operações de controle de impressão/exibição em vez de exibir símbolos. Exemplos incluem *retorno de carruagem* (carriage return) que posiciona o cursor ao lado esquerdo da linha de caracteres atual⁸. *Alimentação de linha*, ou *line feed*, (que move o cursor uma linha abaixo no dispositivo de saída), e *back space* (*volta um espaço*) (que move o cursor atrás uma posição à esquerda). Infelizmente, caracteres de controle diferentes executam operações diferentes em dispositivos de saída diferentes. Há pequena padronização entre dispositivos de saída. Para descobrir como um caracteres de controle afeta exatamente um dispositivo particular, você precisará consultar seu

8. Historicamente, o retorno de carruagem vem da carruagem de papel usada em máquinas de escreve. Um retorno de carruagem consiste em mover a carruagem fisicamente à direita totalmente de modo que próximo caráter digitado apareça ao lado da mão esquerda no papel.

manual.

O segundo grupo de 32 códigos de caracteres ASCII inclui vários símbolos de pontuação, caracteres especiais, e os dígitos numéricos. Os caracteres mais notáveis neste grupo incluem o caracter de espaço (o código ASCII 20h) e os dígitos numéricos (o código ASCII 30h ..39h). Note que os dígitos numéricos somente diferem dos valores numéricos deles no nibble de A.O. Subtraindo 30h do código ASCII para qualquer dígito em particular você pode obter o número equivalente daquele dígito.

O terceiro grupo de 32 caracteres ASCII é reservado para os caracteres alfabéticos maiúsculos. Os códigos ASCII para os caracteres "A".."Z" estão na faixa de 41h ..5Ah (65 ..90). Já que há somente 26 caracteres alfabéticos diferentes, o seis códigos restantes dão suporte à vários símbolos especiais.

O quarto, e último, grupo de 32 códigos de caracter ASCII é reservado para os símbolos alfabéticos minúsculos, e cinco símbolos especiais adicionais, e outro caráter de controle (o delete). Note que os símbolos de caracteres minúsculos usam o código ASCII 61h ..7Ah. Se você converter os códigos para caracteres maiúsculos e o minúsculo para binário, você notará que os símbolos maiúsculos diferem dos equivalentes minúsculos exatamente uma posição de bit. Por exemplo, considere o código de caracter para "E" e "e" na Figura 1.14.

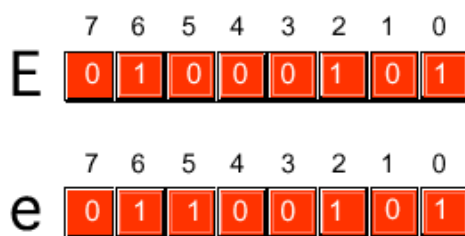


Figura 1.14: Código ASCII para "E" e "e"

O único lugar que estes dois códigos diferem é no bit cinco. Caracteres maiúsculos sempre contêm um zero no bit cinco; caracteres alfabéticos minúsculos sempre contêm 1 no bit cinco. Você pode usar este fato para converter rapidamente entre maiúscula e minúscula. Se você tem um caráter maiúsculo você pode forçá-lo para minúscula colocando um no bit cinco. Se você tem um caráter minúsculo e você deseja forçá-lo para maiúsculo, você pode fazer isso colocando um zero no bit cinco. Você pode mudar um caráter alfabético entre maiúsculo e minúsculo invertendo somente o bit cinco.

Realmente, os bits cinco e seis determinam em qual dos quatro grupos no conjunto de caracteres ASCII o caracter está:

Bit 6	Bit 5	Grupo
0	0	Caracteres de controle
0	1	Dígitos e pontuação
1	0	Maiúsculos e especiais
1	1	Minúsculos e especiais

Assim você pode, por exemplo, converter qualquer caráter maiúsculo ou minúsculo (ou especial correspondente) em seu caráter de controle equivalente colocando zero para os bits cinco e seis.

Considere, por um momento, os códigos de caracteres ASCII dos dígitos numéricos:

Caracter Dec Hex

"0"	48	30h
"1"	49	31h
"2"	50	32h
3	51	33h
4	52	34h
"5"	53	35h
"6"	54	36h
"7"	55	37h
"8"	56	38h
"9"	57	39h

As representações decimais destes códigos ASCII não são ilustrados. Porém, a representação hexadecimal destes códigos ASCII revelam algo muito importante—o nibble de B.O do código ASCII é o binário equivalente do número representado. Esvaziando (ou seja, configurando para zero) o nibble de A.O de um caracteres numérico, você pode converter aquele código de carácter à representação binária correspondente. Reciprocamente, você pode converter um valor binário na faixa de 0..9 para sua representação de carácter ASCII configurando somente o nibble de A.O. para três. Note que você pode usar a operação de AND-lógico para forçar os bits de A.O para zero; igualmente, você pode usar a operação lógica OR para forçar os bits de A.O para 0011 (três).

Note que você *não pode* converter uma seqüência de caracteres numéricos na representação binária equivalente deles simplesmente tirando o nibble de A.O de cada dígito na seqüência. Convertendo 123 (31h 32h 33h) desta maneira resulta três bytes: 010203h, não é o valor correto no qual é 7Bh. Convertendo uma seqüência de dígitos a um inteiro requer mais sofisticação que isto; a conversão acima funciona somente com dígitos únicos.

O bit sete no padrão ASCII é sempre zero. Isto significa que o conjunto de caracteres ASCII consome somente a metade dos possíveis código de carácter em um byte de oito bits. A IBM usa os 128 código de caracteres restantes com vários caracteres especiais que incluem caracteres internacionais (esses com acentos, etc.), símbolos de matemática, e caracteres de desenho de linha. Note que estes caracteres extras são uma extensão não padronizados ao conjunto de caracteres ASCII. Claro que, o nome a IBM tem influência uma considerável, quase todos os computadores pessoais modernos baseado nos 80x86 usam na exibição de vídeo o conjunto de carácter de IBM/ASCII estendido. A maioria das impressoras suportam o conjunto de caracteres da IBM.

Se você necessita trocar dados com outras máquinas que não são compatíveis com PC, você tem somente duas alternativas: migre para um padrão ASCII ou certifique-se que a máquina

alvo suporta o conjunto de caracteres estendidos do IBM-PC. Algumas máquinas, como o Macintosh Apple, não suporta nativamente o conjunto de caracteres estendidos do IBM-PC; porém você pode obter uma fonte de PC que permite-o exibir o conjunto de caracteres estendido. Outras máquinas (por exemplo, Amiga e Atari ST) tem capacidades semelhantes. Porém, os 128 caracteres no conjunto de caracteres ASCII padrão é o único que você deve contar em transferência de sistema a sistema.

Apesar do fato ser um “padrão”, simplesmente codificar seus dados usando padrão caracteres ASCII não garante para compatibilidade em outros sistemas. Enquanto a verdade é que um “A” em uma máquina é provável um “A” em outra máquina, mas é muito pequena a padronização do de máquinas com respeito ao uso dos caracteres de controle. Realmente, os 32 códigos de controle mais o delete, há somente quatro códigos de controle comumente— back space (BS), tab, enter (CR), e line feed (LF). Pior ainda, máquinas diferentes usam freqüentemente estes códigos de controle de modos diferentes. Fim de linha é um exemplo particularmente problemático. MS-DOS, CP/M, e outros sistemas marcam o fim de linha pela sucessão de dois caracteres CR/LF. O Macintosh Apple, Apple II, e muitos outros sistemas marcam o fim de linha por um único caracteres de CR. Sistemas de UNIX marcam o fim de uma linha com um único caráter de LF. Desnecessário dizer, que tentar trocar simples arquivos de texto entre tais sistemas podem ser uma experiência frustradora. Até mesmo se você usar o padrão de caracteres ASCII em todos seus arquivos nestes sistemas, você ainda precisará converter o dados ao trocar arquivos entre eles. Felizmente, tais conversões são bastante simples.

Apesar de algumas deficiências principais, dados ASCII é o padrão para troca de dados por sistemas de computador e programas. A maioria dos programas pode aceitar dados ASCII; igualmente a maioria dos programas pode processar dados ASCII. Considerando que você estará lidando com caracteres ASCII na linguagem assembly, seria sábio estudar o layout do conjunto de caracteres ASCII e memorizar alguns códigos chave (por exemplo, “0”, “o A”, “o a”, etc.).

1.12 Resumo

Sistemas de computadores mais modernos usam o sistema numérico binário para representar valores. Considerando que valores binários são de difícil manuseio, nós usaremos freqüentemente a representação hexadecimal para esses valores. Isto porque é muito fácil converter entre hexadecimal e binário, ao contrário da conversão entre os mais familiares sistemas decimais e binários. Um único dígito hexadecimal consome quatro dígitos binários (bits), e nós chamamos um grupo de quatro bits um nibble. Veja:

- “O Sistema Numérico Binário”
- “Formatos Binários”
- “O Sistema Numérico Hexadecimal”

Os 80x86 funcionam melhor com grupos de bits que são oito, 16, ou 32 bits longos. Nós chamamos o tamanho destes objetos de bytes, words, e double word, respectivamente. Com um byte, nós podemos representar qualquer um de 256 únicos valores. Com um word nós podemos representar um de 65,536 valores diferentes. Com um double word nós podemos representar mais de quatro bilhões valores diferentes. Freqüentemente nós representamos simplesmente valores inteiros (sinalizado ou não sinalizado) com bytes, words, e double word; porém nós representaremos freqüentemente outras quantidades. Veja:

- “Organização de Dados”
- “Bytes”
- “Words”
- “Double Word”

Para falar sobre bits específicos dentro de um nibble, byte, word, double word, ou outra estrutura, nós numeraremos os bits começando com zero (para o menos bit significativo) até $n-1$ (onde n é o número de bits no objeto). Nós também numeraremos nibble, bytes, e words em estruturas grandes de maneira semelhante. Veja:

- “Formatos Binários”

Há muitas operações que nós podemos executar em valores binários incluindo os de aritmética normal (+, -, *, e /) e as operações lógicas (AND, OR, XOR, NOT, Troca Esquerda, Troque Direita, Rotação Esquerda, e Rotação Direita). O AND Lógico, OR, XOR, e NOT são tipicamente definidos para operações de bits únicos. Nós podemos estender estes a para n bits executando operações de bitwise. As trocas e rotações sempre estão definidas para uma sequência de comprimento fixo de bits. Veja:

- “Operações Aritméticas em Números Binários e Hexadecimais”
- “Operações Lógicas em Bits”
- “Operações Lógicas em Números Binários e Sequências de Bits”
- “Troca e Rotações”

Há dois tipos de valores inteiros que nós podemos representar com sequências binárias nos 80x86: inteiros não sinalizados e inteiros sinalizados. Os 80x86 representam inteiros não sinalizados usando o formato binário padrão. Ele representa inteiros sinalizados usando o formato complementar de dois. Enquanto os inteiros não sinalizados podem ser de comprimento arbitrário, somente faz sentido falar sobre valores binários sinalizado comprimento fixo. Veja:

- “Números Sinalizados e Não Sinalizados”
- “Sinal e Extensão de Zero”

Pode não ser particularmente prático armazenar dados em grupos de oito, 16, ou 32 bits. Para conservar espaço você pode querer empacotar vários bits de dados no mesmo byte, word, ou double word. Isto reduz as exigências de armazenamento tendo que executar operações extras para empacotar e desempacotar os dados. Veja:

- “Campos Bits e Pacote de Dados”

Dados do tipo caracter é provavelmente o tipo de dados mais comum encontrado além dos valores inteiros. O IBM PC e compatíveis usam uma variante do conjunto de caracteres ASCII – o conjunto de caracteres estendidos IBM/ASCII. Os primeiros 128 destes caracteres são os caracteres ASCII padrão, 128 são caracteres especiais criados pela IBM para linguagens internacionais, matemáticas, e desenho de linha. Como o uso do conjunto de caracteres ASCII é tão comum em programas modernos, este conjunto de caracter é essencial. Veja:

- “O Conjunto de Caracteres ASCII”

1.13 Exercícios de Laboratório

Acompanhando este texto existe uma quantidade significativa de software. Este software é dividido em quatro categorias básicas: código fonte para exemplos que aparecem ao longo deste texto, a Biblioteca Padrão UCR para 80x86 para programadores de linguagem assembly, códigos de exemplo que você pode modificar vários exercícios de laboratório, e software de aplicação para apoio em vários exercícios de laboratório. Este software foi escrito usando a linguagem assembly, C++, Flex/Bison, e Delphi (Object Pascal). A maioria dos programas de aplicação inclui código fonte e também o código executável. Muitos dos softwares que acompanha este texto roda sob Windows 3.1, Windows 95, ou Windows NT. Porém, alguns softwares que manipula o hardware diretamente somente rodará sob DOS ou uma caixa de DOS em Windows 3.1. Este texto assume

que você está familiarizado com os sistemas operacionais DOS e Windows; se você conhecer pouco sobre DOS ou Windows, você deve recorrer a um texto apropriado nesses sistemas para detalhes adicionais.

1.13.1 Instalando o Software

O software que acompanha este texto geralmente é distribuído em CD-ROM⁹. Você pode usar a maioria fora do CD-ROM. Porém, para maior velocidade e conveniência você provavelmente achará melhor instalar o software em um disco rígido¹⁰. Para fazer isto, você precisará criar dois subdiretórios no diretório de raiz em seu disco rígido: ARTOFASM e STDLIB. O diretório de ARTOFASM conterá os arquivos específico para este livro texto, o diretório STDLIB conterá os arquivos associados com a Biblioteca Padrão UCR para programadores de linguagem assembly no 80x86. Uma vez criados estes dois subdiretórios, copie todos os arquivos e subdiretórios dos diretórios correspondentes no CD para seu disco rígido. A partir do DOS (ou uma janela de DOS), você pode usar o seguinte comando XCOPY para fazer isto:

```
xcopy r:\artofasm\*. * c:\artofasm /s
xcopy r:\stdlib\*. * c:\stdlib /s
```

Estes comandos assumem que seu CD-ROM é o drive R: e que você está instalando o software no disco rígido C:. Eles também assumem que você criou os subdiretórios ARTOFASM e STDLIB antes de executar os comandos XCOPY.

Para usar a Biblioteca Standard em projetos de programação, você precisará adicionar ou modificar duas linhas em seu arquivo AUTOEXEC.BAT. Se linhas semelhantes já não estão presentes, acrescente as seguintes duas linhas a seu arquivo AUTOEXEC.BAT:

```
set lib=c:\stdlib\lib
set include=c:\stdlib\include
```

Estes comandos dizem para o MASM (o Microsoft Macro Assembler) onde pode encontrar a biblioteca e pode incluir arquivos para a Biblioteca Padrão UCR. Sem estas linhas, o MASM informará um erro a toda hora que você usar as rotinas da biblioteca standard em seus programas.

Se já existe as linhas “set include = ...” e “set lib=...” em seu arquivo AUTOEXEC.BAT, você não deve substituir com as linhas acima. Ao invés, você deve juntar a seqüência “;c:\stdlib\lib” no fim da declaração existente “set lib=...” e “;c:\stdlib\include” no fim da declaração existente “set include=...” . Várias linguagens (como C++) também usam estas declarações “set”; se você substituí-los arbitrariamente com as declarações acima, seus programas em linguagem assembly trabalharão bem, mas qualquer tentativa para compilar um programa C++ (ou outra linguagem) pode falhar.

Se você esquecer de pôr estas linhas em seu arquivo AUTOEXEC.BAT, você pode temporariamente (até a próxima vez que você der o boot no sistema) executar estes comandos simplesmente digitando-os no prompt do DOS. Digitando “set” sozinho na linha de comando, você pode ver se estes comandos set estão atualmente ativos.

Se você não tiver um drive de CD-ROM, você pode obter o software associado a este livro pelo ftp anônimo em cs.ucr.edu. Olhe no subdiretório “/pub/pc/ibmpc”. os arquivos estão comprimidos no servidor de ftp. Um arquivo “README” descreve como descomprimir os dados.

9. Também está disponível viaftp anônimo, embora sejam muitos associados a este texto.

10. Se você está usando este software no laboratório de sua escola, seu instrutor provavelmente instale este software nas máquinas do seu laboratório. Como regra geral, nunca instale softwares nas máquinas do laboratório de sua escola. Consulte o instrutor antes de instalar este software no laboratório.

O diretório STDLIB que você criou possui a biblioteca e arquivos fonte para a Biblioteca Padrão UCR para Programadores de linguagem assembly no 80x86. Este é um conjunto essencial de subrotinas na linguagem assembly na qual você pode chamar de *mímicos* muitas das rotinas C na biblioteca standard. Estas rotinas simplificam grandemente a escritura de um programa em linguagem assembly. Além disso, eles são domínio público assim você pode usá-los em qualquer programa que você escrever sem medo de restrições autorais.

O diretório ARTOFASM contém arquivos específicos para este texto. Dentro do diretório ARTOFASM verá você uma sucessão de subdiretórios nomeados ch1, ch2, ch3, etc. Este subdiretórios contém os arquivos associados com Capítulo Um, Capítulo Dois, e assim por diante. Dentro de alguns deste subdiretórios, você achará dois subdiretórios nomeados “DOS” e “WINDOWS”. Se estes subdiretórios estiverem presentes, eles separam esses arquivos que têm que rodar sob MS-Windows os que rodam sob DOS. *Muitos dos programas de DOS requerem* um ambiente “modo-real” e não rodará em uma janela DOS no Windows 95 ou Windows NT. Você precisará rodar estes software no MS-DOS. As aplicações Windows requerem um monitor colorido.

Existe freqüentemente um terceiro subdiretório apresentado em cada diretório de capítulo: SOURCES. Este subdiretório contém a listagem de fonte (quando apropriado ou possível) para o software daquele capítulo. A maioria dos softwares para este texto é escrito em linguagem assembly usando o MASM 6.x, C++ genérico, Turbo Pascal, ou Borland Delphi (Visual Object Pascal). Se você está interessado em ver como o software funciona, você pode olhar neste subdiretório.

Este texto assume que você já sabe executar programas de MS-DOS e o Windows e que você está familiarizado com a terminologia comum de DOS e Windows. Também assume que você sabe alguns comandos simples de MS-DOS como DIR, COPY, DEL, RENAME, e assim por diante. Se você for iniciante no Windows e DOS, você deve ver uma manual de referência apropriado nestes sistemas operacionais.

Os arquivos para os exercícios de laboratório do Capítulo Um aparecem no subdiretório ARTOFASM\CH1. Estes são todos programas Windows, assim você precisará estar rodando Windows 3.1, Windows 95, Windows NT, ou alguma versão posterior (e compatível) do Windows para rodar estes programas.

1.13.2 Exercícios de Conversão de dados

Neste exercício você usará o programa “convert.exe” encontrado no subdiretório ARTOFASM\CH1. Este programa exhibe e converte inteiros de 16 bits usando decimal sinalizado, decimal não sinalizado, hexadecimal, e notação binária. Quando você roda este programa abre uma janela com quatro *caixas de edição*. (uma para cada tipo de dados). Mudando o valor em uma das caixas de edição imediatamente é atualizado os valores nas outras caixas assim elas exibem as representações correspondentes delas para o valor novo. Se você cometer um erro na entrada de dados, o programa buzina e a caixa de edição fica vermelha até que você corrija o erro. Note que você pode usar o mouse, teclas de controle de cursor, e as teclas de edição (por exemplo, DEL e Backspace) para mudar os valores individuais dentro das caixas de edição.

Para este exercício e seu relatório de laboratório, você deve explorar a relação entre vários valores binários, hexadecimal, decimal não sinalizado, e sinalizado. Por exemplo, você deve entrar nos valores decimais não sinalizados como 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, e 32768 e deve fazer um comentário sobre os valores que aparecem nas outras caixas de texto.

O propósito primário deste exercício é se familiarizar com o equivalentes decimais de alguns valores binários e hexadecimais comuns. Em seu relatório de laboratório, por exemplo, você

deve explicar sobre o que tem de especial entre os equivalentes binários (e hexadecimais) dos números decimais acima.

Outro conjunto de experiências para tentar é escolher vários números binários que têm exatamente dois conjuntos de bits, por exemplo, 11, 110, 1100, 1 1000, 11 0000, etc. Certifique-se de comentar os resultados decimais e hexadecimais que estas entradas produzem.

Tente, entrar com vários números binários onde o B.O. em oito bits são todo o zero. Faça um comentário sobre os resultados em seu relatório de lab. Tente a mesma experiência com números hexadecimais usando zeros para o dígito de B.O. ou o dois dígitos de B.O.. Você também deve experimentar entrar com números negativos decimais sinalizados na caixa de texto; tente, usar valores como -1, -2, -3, -256, -1024, etc. Explique os resultados que você obtém usando seu conhecimento em sistema numérico complementar de dois.

Tente, até mesmo entrar com números ímpares em decimal não sinalizado. Descubra e descreva a diferença entre números pares e ímpares na representação binária deles. Tente, entrar com múltiplos de outros valores (por exemplo, de três: 3, 6, 9, 12, 15, 18, 21,...) e vê se você pode descobrir um padrão nos resultados binários.

Verifique a conversão hexadecimal <-> binária que este capítulo descreve. Em particular, entre o mesmo dígito hexadecimal em cada das quatro posições de um valor 16 bits e faça um comentário sobre a posição dos bits correspondentes na representação binária. Tente vários valores binários entrando com 1111, 11110, 111100, 1111000, e 11110000. Explique os resultados que você obtém e descreva por que você sempre deve estender valores binários então tamanho deles é um mesmo múltiplo de quatro antes de convertê-los.

Em seu relatório de laboratório, liste as ou invente você mesmo. Explique os resultados você espera e inclua os resultados atuais que o programa convert.exe produz. Explique qualquer percepção que você tem enquanto usa o programa convert.exe.

1.13.3 Exercícios de Operações Lógicas

O programa logical.exe é uma calculadora simples que computa várias funções lógicas. Lhe permite entrar com valores binários ou hexadecimais e então computa o resultado de alguma operação lógica nas entradas. A calculadora suporta o AND lógico diádico, OR, e XOR. Também suporta o NOT monádico, NEG (complemento de dois), SHL (troca esquerda), SHR (troca direita), ROL (rotação esquerda), e ROR (rotação direita).

Quando você roda o programa logical.exe ele exibe um conjunto de botões no lado esquerdo da janela. Estes botões o permitem selecionar o cálculo. Por exemplo, apertando o botão AND instrui a calculadora para computar a operação lógica AND entre os dois valores de entrada. Se você seleciona um operação monádica (única) como NOT, SHL, etc., então você pode entrar somente com um único valor; para as operações diádica, ambos os conjuntos de caixas de entrada de texto serão ativos.

O programa logical.exe permite entrar com valores binários ou hexadecimais. Note que este programa converte qualquer mudança automaticamente na janela de entrada de texto binária para hexadecimal e atualiza o valor na entrada da caixa de edição hex. Imediatamente é refletida qualquer mudança na caixa de texto de entrada hexadecimal na caixa de texto binária. Se você entrar com um valor ilegal em uma caixa de texto de entrada, o programa logical.exe mudará a caixa para vermelho até que você corrija o problema.

Para este exercício de laboratório, você deve explorar cada operação lógica bitwise. Crie várias experiências escolhendo alguns valores cuidadosamente, manualmente compute o resultado que você espera, e então faça a experiência usando o programa `logical.exe` para verificar seus resultados. Você deve especialmente experimentar as capacidades de mascaramento do operações de AND lógico, OR, e XOR. Tente as operações lógicas AND, OR, e XOR valores com valores diferentes como 000F, 00FF, 00F0, 0FFF, FF00, etc. Informe os resultados e faça um comentário sobre eles em seu relatório de laboratório.

Algumas experiências que você poderia querer tentar, além dessas que você inventar, incluem as seguintes:

- Invente uma máscara para converter o valor ASCII '0'..'9' para as contrapartes dos binários inteiros deles usando a operação lógica AND. Tente, entrar com os códigos ASCII de cada destes dígitos ao usar esta máscara. Descreva seus resultados. O que acontece se você entrar com dígito de códigos não-ASCII?
- Invente uma máscara para converter valores de inteiro na faixa de 0 ..9 para o código ASCII correspondente deles usando a operação lógica OR. Entre com cada um dos valores binários na faixa de 0 ..9 e descreva seus resultados. O que acontece se você entrar com valores fora da faixa de 0 ..9? em particular, o que acontece se você entra em valores fora da faixa de 0h..0fh?
- Invente uma máscara para determinar se um 16 bits valor de inteiro é positivo ou negativo usando a operação lógica AND. O resultado deve ser zero se o número for positivo (ou zero) e deve ser não-zero se o número for negativo. Entre com vários valores positivos e negativos para testar sua máscara. Explique como você poderia usar a operação AND para testar *qualquer* bit único para determinar se é zero ou um.
- Invente uma máscara para usar com a operação lógica XOR que produzirá o mesmo resultado no segundo operador aplicando o operador NOT lógico para o segundo operador.
- Verifique que os operadores SHL e SHR correspondem a uma multiplicação de inteiro por dois e uma divisão de inteiro de dois, respectivamente. O que acontece se você trocar dados dos bits de A.O. ou de B.O? A que isto corresponde em termos de multiplicação e divisão de inteiro?
- Aplique a operação ROL para um conjunto de números positivo e negativos. Baseado em suas observações na Seção 1.13.3, você pode dizer sobre o resultado quando você faz a rotação esquerda em um número negativo ou um número positivo?
- Aplique os operadores NEG e NOT para um valor. Discuta a semelhança e a diferença nos resultados deles. Descreva esta diferença baseado em seu conhecimento do sistema numérico complementar de dois.

1.13.4 Exercícios de Sinal e Extensão de Zero

O programa `signext.exe` aceita valores binários ou hexadecimais de oito bits para estudo do sinal e extensão de zero para 16 bits. Como o programa `logical.exe`, este programa o permite entrar tanto com um valor binário ou hexadecimal e zero imediato e sinal de valor estendido.

Para seu relatório de laboratório, proveja vários valores de entrada de oito bits e descreva os resultados que você espera. Rode estes valores através do programa `signext.exe` e verifique os resultados. Para cada experiência que você faça, tenha certeza de listar todos os resultados em seu relatório de lab. Tente valores como 0, 7fh, 80h, e 0ffh.

Enquanto fizer estas experiências, descubra quais dígitos hexadecimais que aparecem no nibble de A.O. produzindo números negativos de 16 bits e que produzem valores 16 bits positivos. Documente isto no seu relatório de lab.

Entre com conjuntos de valores como (1,10), (2,20), (3,30),..., (7,70), (8,80), (9,90), (A,A0),..., (F,F0). Explique os resultados que você entra no seu relatório de lab. Por que o sinal estendido de "F" faz-se com zeros enquanto o sinal estendido de "F0" faz-se com uns?

Explique em seu relatório de laboratório como sinalizar ou fazer extensão de zero com valores de 16 bits para a 32 valores de bit. Explique por que a extensão de zero ou extensão de sinal é útil.

1.13.5 Exercícios de Pacote de Dados

O programa packdata.exe usa o tipo de dados data que aparece neste capítulo (veja "Campos Bits e Pacote de Dados"). Ele permite introduzir um valor de data em binário ou decimal e empacota essa data em um único valor de 16 bits.

Quando você rodar este programa, aparecerá uma janela com seis caixas de entrada de dados: três para entrar com a data na forma decimal (mês, dia, ano) e três caixas de entrada de texto que o permite entrar com a data em forma binária. O valor do mês deve estar na faixa de 1 ..12, o valor do dia deve estar na faixa de 1..31, e o valor do ano deve estar na faixa de 0 ..99. Se você entrar com um valor fora desta faixa (ou outro valor ilegal), então o programa packdata.exe mudará a caixa de entrada de dados para vermelho até que você corrija o problema.

Escolha várias datas para suas experiências e converta estas datas manualmente para a forma binária empacotada de 16 bits (se você tiver dificuldade com a conversão decimal para binária, use o programa de conversão do primeiro exercícios neste laboratório). Então rode estas datas através do programa packdata.exe para verificar sua resposta. Certifique-se de incluir toda a saída do programa em seu relatório de lab.

A uma mínima descoberta, você deve incluir as seguintes datas em suas experiências:

2/4/68, 1/1/80, 8/16/64, 7/20/60, 11/2/72, 12/25/99, a Data de hoje, um aniversário (não necessariamente seu), em seu relatório de lab.

1.14 Questões

1) Converta os seguintes valores decimais para binário:

- | | | | | |
|-----------|-----------|-----------|----------|-----------|
| a) 128 | b) 4096 | c) 256 | d) 65536 | e) 254 |
| f) 9 | g) 1024 | h) 15 | i) 344 | j) 998 |
| k) 255 | l) 512 | m) 1023 | n) 2048 | o) 4095 |
| p) 8192 | q) 16,384 | r) 32,768 | s) 6,334 | t) 12,334 |
| u) 23,465 | v) 5,643 | w) 464 | x) 67 | y) 888 |

2) Converta os seguintes valores binários para decimal:

- | | | | | |
|--------------|--------------|--------------|--------------|--------------|
| a) 1001 1001 | b) 1001 1101 | c) 1100 0011 | d) 0000 1001 | e) 1111 1111 |
| f) 0000 1111 | g) 0111 1111 | h) 1010 0101 | i) 0100 0101 | j) 0101 1010 |
| k) 1111 0000 | l) 1011 1101 | m) 1100 0010 | n) 0111 1110 | o) 1110 1111 |

p) 0001 1000 q) 1001 111 1 r) 0100 0010 s) 1101 1100 t) 1111 0001
u) 0110 1001 v) 0101 1011 w) 1011 1001 x) 1110 0110 y) 1001 0111

3) Converta os valores binários do problema 2 para hexadecimal.

4) Converta os seguintes valores hexadecimais para binário:

a) 0ABCD	b) 1024	c) 0DEAD	d) 0ADD	e) 0BEEF
f) 8	g) 05AAF	h) 0FFFF	i) 0ACDB	j) 0CDBA
k) 0FEBA	l) 35	m) 0BA	n) 0ABA	o) 0BAD
p) 0DAB	q) 4321	r) 334	s) 45	t) 0E65
u) 0BEAD	v) 0ABE	w) 0DEAF	x) 0DAD	y) 9876

Execute as seguintes computações hexadecimais (mantenha o resultado em hex):

5) 1234 + 9876

6) 0FFF - 0F34

7) 100 - 1

8) 0FFE - 1

9) Qual é a importância de um nibble?

10) Quantos dígitos hexadecimais tem em:

a) um byte b) um word c) um double word

11) Quantos bits tem em um:

a) nibble b) byte c) word d) double word

12) Qual bit (número) é o bit de A.O em um:

a) nibble b) byte c) word d) double word

13) Que caracteres nós usamos como sufixo para números hexadecimais? Números binários? Números Decimais ?

14) Assumindo um formato complemento de dois de 16 bits, determine qual dos valores na questão 4 são positivos e qual são negativos.

15) Faça extensão de sinal em todos os valores na questão dois para dezesseis bits. Dê sua resposta em hex.

16) Execute a operação bitwise AND nos seguintes pares de valores hexadecimais. Forneça sua resposta em hex. (Sugestão: converta valores de hex para binário, faça a operação, então converta de novo a hex).

- | | | | |
|-----------------|----------------|-----------------|----------------|
| a) 0FF00, 0FF0 | b) 0F00F, 1234 | c) 4321, 1234 | d) 2341, 3241 |
| e) 0FFFF, 0EDCB | f) 1111, 5789 | g) 0FABA, 4322 | h) 5523, 0F572 |
| i) 2355, 7466 | j) 4765, 6543 | k) 0ABCD, 0EFDC | l) 0DDDD, 1234 |
| m) 0CCCC, 0ABCD | n) 0BBBB, 1234 | o) 0AAAA, 1234 | p) 0EEEE, 1248 |
| q) 8888, 1248 | r) 8086, 124F | s) 8086, 0CFA7 | t) 8765, 3456 |
| u) 7089, 0FEDC | v) 2435, 0BCDE | w) 6355, 0EFDC | x) 0CBA, 6884 |
| y) 0AC7, 365 | | | |

17) Execute a operação lógica OR nos pares de números anteriores.

18) Execute a operação lógica XOR nos pares de números anteriores.

19) Execute a operação lógica NOT em todos os valores na questão quatro. Assuma que todos os valores são 16 bits.

20) Execute a operação complementar de dois em todos os valores na questão quatro. Assuma valores de 16 bits.

21) Faça extensão de sinal nos seguintes valores hexadecimais de oito a dezesseis bits. Apresente sua resposta em hex.

- | | | | | |
|-------|-------|-------|-------|-------|
| a) FF | b) 82 | c) 12 | d) 56 | e) 98 |
| f) BF | g) 0F | h) 78 | i) 7F | j) F7 |
| k) 0E | l) AE | m) 45 | n) 93 | o) C0 |
| p) 8F | q) DA | r) 1D | s) 0D | t) DE |
| u) 54 | v) 45 | w) F0 | x) AD | y) DD |

22) Contraia o sinal dos seguintes valores de dezesseis bits para oito bits. Se você não puder executar a operação, explique por que.

- | | | | | |
|---------|---------|---------|---------|--------|
| a) FF00 | b) FF12 | c) FFF0 | d) 12 | e) 80 |
| f) FFFF | g) FF88 | h) FF7F | i) 7F | j) 2 |
| k) 8080 | l) 80FF | m) FF80 | n) FF | o) 8 |
| p) F | q) 1 | r) 834 | s) 34 | t) 23 |
| u) 67 | v) 89 | w) 98 | x) FF98 | y) F98 |

23) Faça extensão de sinal de 16 bits para 32 bits na questão 22.

24) Assumindo que os valores na questão 22 são valores 16 bits, execute a operação de troca esquerda neles.

25) Assumindo que os valores na questão 22 são valores 16 bits, execute a operação de troca direita neles.

26) Assumindo que os valores na questão 22 são valores 16 bits, execute a operação de rotação esquerda neles.

27) Assumindo que os valores na questão 22 são valores 16 bits, execute a operação de rotação direita neles.

28) Converta as datas seguintes para o formato empacotado descrito neste capítulo (veja “Campos Bits e Pacote de Dados”). Forneça seus valores como um número hex de 16 bits.

a) 1/1/92 b) 2/4/56 c) 6/19/60 d) 6/16/86 e) 1/1/99

29) Descreva como usar as operações lógicas e troca para extrair o campo dia do registro de data empacotado na questão 28. Quer dizer ,conclua com um valor inteiro de 16 bits na faixa de 0 ..31.

30) Suponha você tem um valor na faixa de 0 ..9. Explique como você poderia convertê-lo a um caráter ASCII usando as operações lógicas básicas .

31) A função de C++ seguinte localiza o primeiro bit configurado no parâmetro BitMap começando na bit inicial e indo até o bit de A.O. Se nenhum bit existir, retorna -1. Explique, com detalhes, como esta função trabalha.

```
int FindFirstSet(unsigned BitMap, unsigned start)
{
    unsigned Mask = (1 << start);
    while (Mask)
    {
        if (BitMap & Mask) return start;
        ++start;
        Mask <<= 1;
    }
    return -1;
}
```

32) A linguagem de programação C++ não especifica quantos bits há dentro um inteiro não sinalizado. Explique por que o código acima funcionará embora o número de bits seja um inteiro não sinalizado.

33) A seguinte função C++ é o complemento à função nas perguntas acima. Ela localiza o primeiro bit zero no parâmetro de BitMap. Explique, em detalhes, como isto funciona.

```
int FindFirstClr(unsigned BitMap, unsigned start)
{
    return FindFirstSet(~BitMap, start);
}
```

34) As duas funções seguintes configuram ou limpam (respectivamente) um bit particular e retorna o novo resultado. Explique, em detalhes, como estas funções funcionam.

```
unsigned SetBit(unsigned BitMap, unsigned position)
{
    return BitMap | (1 << position);
}
```



```

unsigned ClrBit(unsigned BitMap, unsigned position)
{
    return BitMap & ~(1 << position);
}

```

35) No código que aparece nas perguntas acima, explique o que acontece se o começo e os parâmetros posicionais contêm um valor maior ou igual ao número de bits em um inteiro não sinalizado.

1.15 Projetos de Programação

Os seguintes projetos de programação assumem você está usando C, C++, Turbo Pascal, Borland Pascal, Delphi, ou alguma outra linguagem de programação que suporta operações lógicas bitwise. Note que o C e C++ usam os operadores “&”, “|”, e “^” para AND lógico, OR, e XOR, respectivamente. Os produtos Borland Pascal permitem usar os operadores “and”, “or”, e “xor” em inteiros para executar operações lógicas bitwise. Todos os projetos seguintes esperam que você use estes operadores lógicos. Há outras soluções para estes problemas que não envolvem o uso de operações lógicas, **não empregue tal solução**. O propósito destes exercícios é apresentar às operações lógicas disponíveis em linguagens de alto nível. **Confira com seu instrutor qual linguagem você deve usar.**

As descrições seguintes descrevem as funções que você irá escrever. De qualquer forma, você precisará escrever um programa principal para chamar e testar cada uma das funções que você escrever como parte da tarefa.

1) Escreva as funções, *toupper* e *tolower* que pega um único caráter como o parâmetro e converte este caractere para maiúscula (se for minúscula) ou para minúscula (se for maiúscula) respectivamente. Use as operações lógicas para fazer a conversão. Usuários de Pascal podem precisar usar funções as `chr()` e `ord()` para realizar esta tarefa.

2) Escreva uma função “CharToInt” na qual você insere uma seqüência de caracteres e ela retorna o valor inteiro correspondente. *Não use uma rotina de biblioteca embutida como atoi (C) ou strtoint (Pascal) fazer esta conversão.* Você irá processar cada caractere inserido na seqüência de entrada, converta-a de caráter para um inteiro usando as operações lógicas, e guarde o resultado até que você chegue ao fim da seqüência. Um algoritmo fácil para esta tarefa é multiplicar o resultado guardado por 10 e então somar no próximo dígito. Repita isto até que você chegue ao fim da seqüência. Usuários de Pascal provavelmente vão precisar usar a função `ord()` nesta tarefa.

3) Escreva uma função *ToDate* que aceita três parâmetros, um valor mês, dia, e ano. Esta função deve retornar o valor de data empacotado de 16 bits usando o formato dado neste capítulo (veja “Campos Bits e Pacote de Dados”). Escreva três funções correspondentes *ExtractMonth*, *ExtractDay*, e *ExtractYear* que esperam um valor de data de 16 bits e retorna o valor do mês, dia, ou ano correspondente. A função *ToDate* deve automaticamente converter datas na faixa de 1900-1999 para a faixa de 0..99.

4) Escreva uma função “CntBits” que conta o número de bits em um valor inteiro de 16 bits. *Não use nenhuma função built-in na biblioteca de sua linguagem para contar estes bits para você.*

5) Escreva uma função “TestBit”. Esta função requer dois parâmetros inteiros de 16 bits. O primeiro parâmetro é um valor de 16 bits para teste; o segundo parâmetro é um valor na faixa de 0..15 que descreve qual bit é para testar. A função deve retornar verdadeiro se o bit correspondente contiver um, a função deve retornar falso se aquela posição do bit contém zero. A função sempre deve retornar falso se o segundo parâmetro for um valor fora da faixa de 0..15.

6) Pascal e C/C++ oferecem os operadores de troca esquerda e troca direita (SHL/SHR em Pascal, "<<" e ">>" em C/C++). Porém, eles não oferecem operadores de rotação direita e de rotação esquerda. Escreva um par de funções, ROL e ROR que execute as tarefas de rotação. Sugestão: use a função do exercício cinco para testar o bit de A.O. Então use a operação de troca correspondente e a operação lógica OR para executar a rotação.