CI063 - Máquinas Programáveis

Bruno Müller Junior

26 de Fevereiro de 2010

Prefácio

Este texto corresponde ao material de apoio para a disciplina CI063-Máquinas Programáveis, do Curso de Ciência da Computação da UFPR.

Conteúdo

Capítulo 1

A Disciplina

A disciplina CI063-Máquinas Programáveis é ministrada no primeiro período do curso. A disciplina é basicamente uma introdução de alguns aspectos relacionados com o funcionamento de computadores, como sistema de representação de informação (representação de números e caracteres em computador), linguagem assembly e álgebra booleana.

Formalmente, a disciplina tem as seguintes características:
Aulas Práticas: 2 horas
Aulas Teóricas: 2 horas
Carga horária: 60 horas
Ementa: Programação em linguagem de máquina. Elementos de arquitetura de computadores. Noções de periféricos.
Objetivos: Fornecer ao aluno conhecimentos básicos sobre Sistemas Computacionais. Fornecer conhecimento elementar sobre a organização de um computador e o relacionamento entre a máquina, sua linguagem de baixo nível, e as linguagens de alto nível usadas para programá-la. Sedimentar a compreensão, através da programação em linguagem de máquina, dos conceitos básicos de programação: variáveis, atribuição, decisão, iteração.
Pré-requisitos: Não há.
Programa:

1. Organização de um computador: processador, memória, periféricos.
2. Funcionamento do sistema operacional, compilador e aplicativos.
3. Representação de dados, sistemas de numeração, conversão de bases.
4. Representação digital de dados: tipos de dados, armazenamento.
5. Álgebra de Boole, funções lógicas.
6. Definição de linguagem de programação.
7. Modelo do processador, registradores, operadores, instruções.

8. Linguagem de máquina; conjunto de instruções; acesso à memória.
9. Linguagem de máquina; conjunto de instruções; aritmética.
10. Linguagem de máquina; conjunto de instruções; controle de fluxo.
11. Linguagem de máquina; modos de endereçamento.
12. Implementação de construções de alto-nível em linguagem de máquina.
13. Exemplos de programas em linguagem de máquina.

Além disso, como é uma disciplina de primeiro período, inicia explicando o funcionamento da Universidade e das diferenças entre a abordagem didática do segundo e do terceiro graus.

Capítulo 2

Organização da Disciplina

Para atingir os objetivos, a disciplina está organizada em partes:

02 aulas: Introdução à universidade (organização, procedimentos, etc..).

02 aulas: Os componentes clássicos do computador. Uma breve introdução ao cinco componentes clássicos conforme visto nos primeiros capítulos de [DJ97]. Não será objeto de avaliação.

Primeira Parte: 10 aulas: Representação de informação no computador. Trata de como são representados internamente os números naturais, inteiros, reais e caracteres. Referências: [DJ97, Mon01, Tan99].

Segunda Parte: 10 aulas: Programação Assembly. A idéia é dar uma visão geral de programação assembly, como uso de registradores, cálculo de expressões, rótulos, acesso à memória, entre outros. Para tal, será usado o simulador SPIM, e isso já ajudará nas disciplinas que mais adiante lidarão com arquitetura de computadores. Referência [DJ97, Tan99].

Terceira Parte: 06 aulas: Introdução à Álgebra Booleana. Referências: Livros de álgebra booleana, arquitetura de computadores ou circuitos digitais que tratam de lógica combinacional. Vários livros (inclusive alguns via internet) contém este conteúdo. Para as aulas e para o presente texto, foram adotados [Kat94, MC00].

As quatro primeiras aulas não estão contidas neste texto, e serão apresentadas unicamente em aula. Não serão objeto de avaliação.

Parte I

Representação da Informação

Um computador armazena a informações em impulsos elétricos (ou magnéticos). Estes impulsos são normalmente associados a dois valores: “0” para desligado ou “1” para ligado. A entidade de armazenamento destas informações é chamado de “bit” (ou seja, um bit é uma entidade que pode assumir exatamente um entre dois valores : 0 ou 1).

Combinações de bits são usados para representar toda e qualquer informação em um computador. As máquinas são projetadas para trabalhar com grupos de bits, e não com bits individuais. Atualmente é comum encontrar máquinas que trabalham em grupos de 32 bits, 64 bits ou ainda mais.

O termo “representação de informação”, indica o problema de como fazer com que combinações de bits representem alguma informação do nosso mundo. Por exemplo, como mostrar o número 33 (um dos infinitos números naturais que usamos em nossa vida cotidiana) dentro de um computador. É este mecanismo que possibilita o mapeamento de aspectos do mundo real dentro de um computador.

Esta parte do texto está organizada da seguinte forma: O capítulo 3 explica como os números são representados internamente (naturais, inteiros e reais) enquanto que o capítulo 4 explica como os caracteres são representados (tabela ASCII, etc.).

Capítulo 3

Representação de números

Toda e qualquer representação do mundo externo que deve ser armazenada em computador deve ser "mapeada" para uma representação interna no computador que será possivelmente diferente da representação original. Por exemplo, o número natural $(33)_{10}$ (lê-se 33 na base 10, ou decimal) é representado internamente como $(0000\ 0000\ 0010\ 0001)_2$ ou simplesmente $(0010\ 0001)_2$ (lê-se zero, zero, um, zero, zero, zero, zero, zero, um, na base dois, ou binário).

Esta seção aborda vários assuntos pertinentes à representação de números, dividindo-os em três grandes grupos: naturais, inteiros e reais.

A representação de números naturais é apresentada na seção 3.1 enquanto que a representação de números inteiros é apresentada na seção 3.2. Limites e overflow[1], de inteiros e de naturais são vistos na seção 3.3.

A representação de números reais é apresentada na seção 3.5, e os limites e overflow para números reais é apresentado na seção 3.5.3.

3.1 Representação de números naturais (sem sinal)

Esta seção apresentará três sistemas de representação de números: o decimal (seção 3.1.1), o binário (seção 3.1.2 e o hexadecimal (seção 3.1.3). O sistema de numeração decimal não precisaria ser apresentado, uma vez que já é conhecido por ser usado na vida diária. A característica deste sistema de numeração é que ele utiliza dez símbolos diferentes. Esta quantidade de dígitos foi definido a partir do número de dedos dos seres humanos, e por isso, cada símbolo é também conhecido como "dígito". Em outros sistemas de numeração diferente do decimal também é comum utilizar-se o termo "dígito" para representar os símbolos.

Um computador não tem dez dedos. Na verdade, tem somente dois símbolos diferentes (aqui convencionados como `0` e `1`), o que dá origem ao sistema binário, que será estudado na seção 3.1.2. Apesar de ser usado internamente no computador, é difícil de ser reconhecido por seres humanos em função de serem necessários muitos bits para

[1] overflow é o nome dado ao que ocorre quando o resultado de uma operação aritmética excede o limite de representação daquele tipo de valor no computador

representar alguma informação. Por isso, normalmente agrupam os dígitos binários em grupos de quatro, e cada conjunto de quatro bits é representado por um símbolo diferente. Como são ao todo 16 representações diferentes, este sistema de numeração é chamado sistema hexadecimal, que será visto na seção 3.1.3.

Por fim, a seção 3.1.4 apresenta algoritmos para a conversão de quantidades representadas em números naturais entre quaisquer bases, em especial entre as bases binária, decimal e hexadecimal.

3.1.1 O sistema de numeração decimal

O sistema de numeração decimal é conhecido por todos. Apesar disso, ele aparece aqui para demonstrar como um número decimal pode ser decomposto e como pode ser representado em potências de 10. As seções que descrevem o sistema binário e hexadecimal fazem o mesmo (para aquelas bases, claro) e podem ser assimilados mais facilmente por associação ao modelo decimal.

Considere o número $(752,876)_{10}$. Este número pode ser decomposto da forma descrita na tabela 3.1

Termo	Alternativa	total
7×100	$= 7 \times 10^2$	$= 700,000$
5×10	$= 5 \times 10^1$	$= 50,000$
2×1	$= 2 \times 10^0$	$= 2,000$
8×0.1	$= 8 \times 10^{-1}$	$= 0,800$
7×0.01	$= 7 \times 10^{-2}$	$= 0,070$
6×0.001	$= 6 \times 10^{-3}$	$= 0.006$
		$= 752,876$

Tabela 3.1: Decomposição do número $(752,876)_{10}$ em potências de dez.

Se considerarmos que cada dígito está em uma posição no número e que o dígito imediatamente à esquerda da vírgula é o zero-ésimo dígito, tem-se que o dígito 2 está na posição zero, o dígito (5) é o dígito está na posição um e assim por diante. De maneira análoga, os dígitos à direita da vírgula também podem ser indicados pela sua posição. O dígito 8 está na posição -1, o dígito 7 está na posição -2 e assim por diante.

A tabela 3.2 completa o raciocínio.

7	5	2	,	8	7	6	dígito
2	1	0		-1	-2	-3	posição

Tabela 3.2: Posição dos dígitos na composição de um número

A partir desta tabela, é mais fácil de observar que os dígitos que compõe o número podem ser retratados da seguinte forma:

$$7 \times 10^2 + 5 \times 10^1 + 2 \times 10^0 + 8 \times 10^{-1} + 7 \times 10^{-2} + 6 \times 10^{-3} \tag{3.1}$$

A equação 3.1 utiliza a relação posicional entre os dígitos para representar uma quantidade no sistema decimal (veja a coluna central da tabela 3.1. Esta equação pode também representada como mostrado na equação 3.2 (que se lê *somatório para `i` variando de -3 até 2 de d_i multiplicado por 10 elevado à i-ésima potência*). Esta equação é a "forma fechada" da equação 3.1.

$$Q = \sum_{i=-3}^{i=2} d_i \times 10^i, \quad onde\, d_i = [0,1,\ldots,9] \tag{3.2}$$

$$Q = \sum_{i=n}^{i=m} d_i \times 10^i, \quad onde\, d_i = [0,1,\ldots,9] \tag{3.3}$$

A equação 3.3 é a generalização da representação de quantidades no sistema decimal. A partir destas equações, pode-se concluir que um número está na base decimal quando:

1. cada dígito (d_i) da equação é multiplicado por 10 elevado a i-ésima potência;
2. a existência de dez dígitos. Isto é representado na equação com $d_i = [0,1,\ldots,9]$.

Estas duas observações serão a base para apresentar os demais sistemas de numeração. Nas seções que seguem, os modelos são apresentados e em seguida relacionados com a equação 3.3.

Apesar deste texto se concentrar nas bases decimal, binária e hexadecimal, o mesmo princípio pode ser adotado para representar um número, por exemplo, na base 7 (uma base que agradaria tanto a matemáticos quanto a esotéricos).

3.1.2 O sistema de numeração binário

É importante ressaltar que o objetivo de qualquer sistema de numeração é representar quantidades, e que uma mesma quantidade pode ser representada em qualquer sistema de numeração.

O que será visto agora é como representar uma quantidade em um sistema de numeração que não é o sistema decimal. Esta seção trata especificamente de representação de quantidade no sistema binário. Porém, antes de iniciar, é imprescindível entender que:

uma quantidade pode ser representada em qualquer sistema de numeração, e há uma forma de converter a representação para outro sistema de numeração.

Quando uma quantidade é representada em binário, é importante lembrar que somente dois dígitos são utilizados e que convencionalmente utiliza-se os símbolos `0` e `1`[2]. Utilizando estes dois dígitos, é possível enumerar os números em binário da seguinte forma:

Quantidade	Decimal	binário
	0	000
|	1	001
||	2	010
|||	3	011
||||	4	100
|||||	5	101

A primeira linha indica que a quantidade "vazio" corresponde ao símbolo `0` em decimal e ao símbolo `0` em binário. De maneira análoga, a quantidade de um elemento corresponde ao símbolo `1` tanto em decimal quanto em binário.

As coisas começam a ficar interessantes para representar uma quantidade de dois elementos. Neste caso, precisamos de mais de um símbolo binário. O problema é análogo ao que ocorre para representar uma quantidade maior do que nove no sistema decimal. A solução adotada para o sistema binário é a mesma para o sistema decimal: acrescenta-se maior quantidade de símbolos. Intuitivamente, é possível perceber que ao somar

$$(01)_2 + (01)_2 = (10)_2$$

.

Observe a notação adotada. Para evitar confusões, usamos o subscrito para indicar a base do número entre parênteses. Assim, $(10)_2$ indica um número na base binária, enquanto que $(10)_{10}$ indica um número na base decimal[3]

A equação geral para representar uma quantidade na representação binária é a seguinte:

$$Q = \sum_{i=n}^{i=m} d_i \times 2^i,$$
$$onde\, d_i = [0,1]$$

A fórmula adota ainda o sistema de numeração decimal (o dígito dois está representado como 2, que não existe em binário), e ajuda a converter quantidades representadas

[2] A rigor, qualquer par de símbolos poderiam ser utilizados, porém o uso destes dois é o que foi adotado há mais de 50 anos, e é fácil de encontrá-los em teclados. Muitos não gostam desta escolha, uma vez que podem confundir com o sistema decimal. Porém, apesar disso, é improvável ocorram mudanças no futuro.

[3] Esta confusão potencial deu origem a uma "piada" que correu o mundo. Esta "piada" foi mandada por e-mail, e dizia: "Só existem 10 tipos de pessoas: os que conhecem números binários e os que não conhecem". Apesar de evidentemente hilariante, muitos não a entenderam.

em binário para decimal. Como exemplo disso, considere o número $(1100)_2$. Para convertê-lo para decimal, basta utilizar a seguinte equação:

1×2^3	1×8	8
1×2^2	1×4	4
0×2^1	0×2	0
0×2^0	0×1	0
		= 8+4=12

Ou seja, $(1100)_2 = (12)_{10}$.

Para converter um número decimal para binário, basta somar as potências de dois que compõem o número decimal. Por exemplo, os número $(14)_{10}$ e $(9)_{10}$ podem ser convertidos em somatório de potências de dois da seguinte forma:

$$(14)_{10} = 8+4+2 = 2^3+2^2+2^1 = (1110)_2$$
$$(09)_{10} = 8+1 = 2^3+2^0 = (1001)_2$$

O processo descrito acima é a base para a conversão entre quantidades, e é normalmente usada quando o número é pequeno. Quando o número é grande, um outro mecanismo é utilizado. Este outro mecanismo será visto na seção 3.1.4.2.

3.1.2.1 Exercícios

1. converta o número $(0011\ 1100)_2$ para decimal.
2. Qual o maior número decimal representável em uma máquina de 32 bits? (Dica: $(1111)_2 = (10000)_2 - 1$).
3. Qual o maior número decimal representável em uma máquina de *N* bits?
4. Converta os números decimais abaixo em binário.

Decimal	Binário
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

3.1.3 O sistema de numeração hexadecimal

Conhecer o sistema binário, em princípio, já seria o suficiente para representar em nosso mundo as informações (neste caso, números naturais) internas do computador. Porém, informações contidas em 32 bits utilizam 32 caracteres para serem representadas, o que a (maioria) dos seres humanos não são capazes de absorver.

Por isso foi criada uma outra representação para números binários, o sistema hexadecimal. Na verdade é simplesmente um "esquema de agrupamento" de números binários de quatro em quatro. Como cada quatro bits são capazes de representar 16 informações diferentes, este sistama de numeração é chamado hexadecimal. Os primeiros 16 símbolos desta representação são mostrados na tabela 3.3.

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

Tabela 3.3: Os primeiros 16 números decimais, binário e hexadecimal.

Como a quantidade de dígitos hexadecimais é maior do que a quantidade de dígitos decimais, foram usadas letras para preencher os dígitos que faltavam. Alternativas como letras gregas (α, β, ...) entre outras, foram sugeridas, mas a opção por letras (A, B, ... , F) é a alternativa mais viável pois estão presentes nos teclados e são simples de representar em vídeo..

Via de regra, os dígitos usado em qualquer sistema de numeração começam com os dígitos do nosso sistema de numeração ($0, 1, \ldots 9$) e se forem necessários mais dígitos, usa-se letras[4]

[4]Apesar de possível, é improvável ser necessária uma representação que precisasse de mais do que 16 dígitos. O problema maior seria se este sistema usasse mais dígitos do que números e letras, por exemplo para a base 70. Neste caso, quais símbolos utilizar? Por outro lado, é bastante improvável que tal sistema de numeração seja necessário em um futuro próximo, e por isso não é necessário perder o sono por causa disto.

Usando esta tabela, é possível converter qualquer número binário em hexadecimal. Por exemplo, considere o número $(10110101110010)_2$.

1. Separe o número de quatro em quatro, da direita para a esquerda: $(10110101110010)_2 = (10\ 1101\ 0111\ 0010)_2$.

2. Preencha o conjunto mais à esquerda com zeros à frente (em todos os sistemas de numeração, o zero à esquerda é um zero à esquerda): $(10110101110010)_2 = (0010\ 1101\ 0111\ 0010)_2$.

3. Substitua cada conjunto de quatro bits pelo símbolo hexadecimal correspondente: $(0010\ 1101\ 0111\ 0010)_2 = (2D72)_{16}$.

Usando o processo inverso, converte-se números em hexadecimal para binário.

A equação 3.4 é apropriada para representar quantidades em hexadecimal, e segue os modelos anteriores com as respectivas mudanças.

$$Q = \sum_{i=n}^{i=m} d_i \times 16^i,$$
$$onde\, d_i = [0, 1, \ldots, F]$$

Usando esta equação, também é simples converter números para decimal. Como exemplo, considere o número $(12)_{16}$

$$(12)_{16} = 1 \times 16^1 + 2 \times 16^0 = 16 + 2 = (18)_{10} \tag{3.4}$$

3.1.3.1 Exercícios

1. Converta $(22)_{16}$ para decimal e para binário.

2. Converta $(0100\ 0111)_2$ de decimal para hexadecimal.

3. Converta $(130)_{10}$ para binário e para hexadecimal.

3.1.4 Conversões entre as bases

Algumas regras para conversão de bases já foram apresentadas informalmente. Primeiramente vamos formalizar estas regras para então citar as regras “automáticas”. A ordem de apresentação dos temas segue o modelo apresentado na figura 3.1.

3.1.4.1 Regras Práticas

As regras de conversão apresentadas informalmente nas seções anteriores serão apresentadas em maiores detalhes aqui.

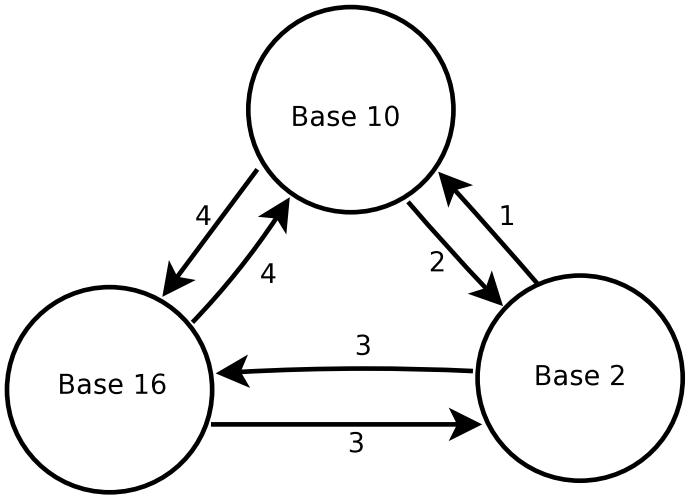


Figura 3.1: Conversões a serem estudadas

7	6	5	4	3	2	1	0		-1	-2	-3	-4
1	0	0	1	1	1	0	0	,	0	1	1	1
2^7	0	0	2^4	2^3	2^2	0	0	,	0	2^{-2}	2^{-3}	2^{-4}

Tabela 3.4: Posição dos dígitos na conversão de um número binário para decimal

3.1.4.1.1 Base 2 → Base 10 Para converter, basta anotar as potências e somá-las. Por exemplo, considere o número $(10011100,0111)_2$. A conversão segue a tabela 3.4.

A primeira linha da tabela indica as posições dos dígitos binários, a segunda contém os dígitos do número binário que deseja-se converter. Finalmente, a terceira linha contém os valores das potências de dois que são multiplicadas por `1`. Aqueles que são multiplicados por zero, tem o resultado zero (por exemplo, a potência `6` seria $0 \times 2^6 = 0$), e por isso foram anotadas como zero.

Em seguida, basta somar todos os campos que não são multiplicados por zero.

$$2^7 + 2^4 + 2^3 + 2^2 + 2^{-2} + 2^{-3} + 2^{-4} =$$

$$128 + 16 + 8 + 4 + 0,25 + 0,125 + 0,0625 =$$

$$156,4375$$

ou seja,
$(10011100,0111)_2 = (156,4375)_{10}$

3.1.4.1.2 Base 10 → Base 2 A idéia aqui é começar com um número binário todo zerado, e ir acrescentando zeros da esquerda para a direita. Por exemplo, suponha o número N que vimos anteriormente $N = (156,4375)_{10}$.

1. início:

7	6	5	4	3	2	1	0	-1	-2	-3	-4
0	0	0	0	0	0	0	0	0	0	0	0

2. Qual o maior número n tal que 2^n é menor do que o número decimal que se deseja converter. Em outras palavras: $n = \lfloor \log_2(N) \rfloor$. Neste caso, sabe-se que $256 < N < 128$, e que por isso $n = 128 = 2^7$. O número binário resultante é:

7	6	5	4	3	2	1	0	-1	-2	-3	-4
1	0	0	0	0	0	0	0	0	0	0	0

3. O número binário obtido até o momento corresponde a 128 em decimal. Uma vez que já foi encontrado o primeiro dígito binário do número binário desejado, basta subtrair este número de N (ou seja, $N = (156,4375)_{10} - n = (156,4375)_{10} - (128)_{10} = (28,375)_{10}$). Este novo número é menor do que 128, e por isso "cabe" nos bits à esquerda do bit 7. Sendo assim, basta iniciar o processo novamente com $N = (28,375)_{10}$ para encontrar os bits que faltam.

4. Repetir o processo até que $N = 0$.

Este processo funciona porque há uma relação entre os dígitos de um número binário e as somas de números decimais (veja 3.2). Assim, ao preencher os dígitos de um número binário da esquerda para a direita, o resultado sempre será um número decimal que "cabe" no lado direito do número binário.

IMPORTANTE: Este processo vale para números inteiros e naturais (os números que não tem parte fracionária). Porém, para números reais (onde há parte fracionária), isso **pode não funcionar**. Funciona para números cuja parte fracionária pode ser representada por somas de potências de dois, como $0,5$, $0,25$, $0,125$, $0,75$, etc.. Porém, existem números reais que não podem podem ser representados assim, e a soma de potências de dois resulta em uma aproximação do número desejado. Como exemplo considere ($(0,1)_{10}$), que em binário corresponde a uma dízima periódica ($(0,1)_{10} = (0,0001\ 1001\ 1001\ 1001\ldots$).

A representação binária de $(0,1)_{10}$ tende a $(0,1)_{10}$ quando a quantidade de bits usados tende ao infinito. Porém, como o número de dígitos é finito, a representação binária de $(0,1)_{10}$ acaba sendo uma aproximação, ou seja: $(0,0001\ 1001\ 1001\ 1001\cdots = (0,09999\ldots)_{10} \approx (0,1)_{10}$.

> O modelo apresentado nesta seção para representar **números reais** não é usado na prática, mas é uma boa introdução ao problema que será visto em maiores detalhes na seção 3.5.

3.1.4.1.3 Base 2 → Base 16 / Base 16 → Base 2 O processo aqui já foi descrito anteriormente. Basta agrupar os dígitos de quatro em quatro (da direita para a esquerda) e usar a tabela 3.3.

Exemplo (32 bits): $(00010010101000001001110001100111)_2 \rightarrow (?)_{16}$
$(0001\ 0010\ 1010\ 0000\ 1001\ 1100\ 0110\ 0111)_2 \rightarrow (12A09C67)_{16}$

O processo inverso também é simples: cada símbolo hexadecimal deve ser substituído por quatro dígitos binários.

3.1.4.1.4 Base 16 → Base 10 / Base 10 → Base 16 Nesta e em outras conversões, e mais apropriado usar as regras formais (próxima seção). Porém, uma forma simples de fazer esta conversão é usar a base binária como intermediário, ou seja:

$(X)_{16} \rightarrow (?)_{10} \Leftrightarrow (X)_{16} \rightarrow (?)_2 \rightarrow (?)_{10}$

ou

$(X)_{10} \rightarrow (?)_{16} \Leftrightarrow (X)_{10} \rightarrow (?)_2 \rightarrow (?)_{16}$

Uma outra forma de obter este resultado é adotar o mecanismo indicado na seção 3.1.4.1.

3.1.4.2 Regras Formais

Esta seção apresenta meios "automáticos" para fazer a conversão entre bases, e podem ser facilmente implementadas em computador.

Ao invés de se usar somente as conversões entre as bases decimal, binária e hexadecimal, a abordagem adotada permite converter qualquer quantidade representada em uma base para qualquer outra base. O método descrito a seguir está detalhado em [Knu97].

Os mecanismos de conversão são basicamente algébricos, e exigem cálculos. Porém, se estamos lidando com sistemas de numeração diferente do decimal, como fazer cálculos em decimal? Por exemplo, se quisermos converter $(65)_7 \rightarrow (?)_{11}$, como fazer os cálculos se nenhuma das bases é a base decimal?

Algumas raras pessoas serão capazes de fazer os cálculos na base 7. Outras, também raras, saberão fazer os cálculos na base 11. Porém, não deve-se considerar que estes casos irão corresponder à situação normal, e acreditamos que a maior parte das pessoas irão preferir fazer os cálculos na base 10.

Os algoritmos apresentados nesta seção permitem que todos os cálculos sejam feitos na base decimal a custo de algum trabalho "extra". No caso que citamos acima ($(65)_7 \rightarrow (?)_{11}$), será necessário usar a base decimal como intermediária, ou seja, $(65)_7 \rightarrow (?)_{10} \rightarrow (?)_{11}$). Na primeira parte da conversão ($(65)_7 \rightarrow (?)_{10}$), desejamos usar a base decimal para os cálculos (a base destino), enquanto que na segunda parte da conversão, $(?)_{10} \rightarrow (?)_{11}$), desejamos usar a base decimal para os cálculos (a base origem).

Os dois algoritmos que serão mostrados a seguir mostram como fazer a conversão de forma a fazer os cálculos na base origem ou na base destino.

3.1.4.2.1 Converter da base `b` para a base `B` usando a aritmética de `b`, a base origem Dado um número natural $(u)_b$, obtém-se o seu equivalente $(U)_B$ (composto pelos dígitos $(U_n U_{n-1} U_{n-2} \ldots U_1 U_0)$ da seguinte forma:

$$\begin{aligned} U_0 &= u \, mod \, B \\ U_1 &= \lfloor u \, div \, B \rfloor \, mod \, B \\ U_2 &= \lfloor \lfloor u \, div \, B \rfloor \, div \, B \rfloor \, mod \, B \end{aligned} \tag{3.5}$$

Onde, *mod* é o resto da divisão, e $\lfloor X \rfloor$ corresponde a X arredondado para baixo (ignora o que vem depois da vírgula).

Como primeiro exemplo, considere a conversão $(44)_{10} \rightarrow (?)_2$. O número binário resultante será composto por vários dígitos, por exemplo, se forem oito dígitos, teremos: $(b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0)_2$. Segundo a equação 3.6, teremos:

$$\begin{aligned} b_0 &= 44 \, mod \, 2 = 0 \\ b_1 &= \lfloor 44 \, div \, 2 \rfloor \, mod \, 2 = 0 \\ b_2 &= \lfloor \lfloor 44 \, div \, 2 \rfloor \, div \, 2 \rfloor \, mod \, 2 = 1 \\ &\ldots \end{aligned}$$

cujo resultado é:

7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	0

Ou seja, $(44)_{10} \rightarrow (00101100)_2$. A comprovação se este resultado está correto pode ser obtido através das regras informais:

$$(00101100)_2 \rightarrow 2^5 + 2^3 + 2^2 + 2^1 = 32 + 8 + 4 = (44)_{10}$$

Para deixar mais claro este processo, vamos mostrá-lo de duas outras formas. A primeira é na forma do algoritmo 1.

```
1  {
2     i := 0 ;
3     q := 0 ;
4     B := 2 ;
5     n := 44 ;
6      Enquanto ( n > 0 ), faça {
7        q := n mod B ;
8        vetor[i] := q ;
9        i := i+1 ;
10       n := n div B ;
11    }
12 }
```

Algoritmo 1: Algoritmo de conversão de base b para B usando aritmética de b

Pelo algoritmo acima, basta substituir B pelo divisor desejado. É interessante observar que o resto da divisão de um número por B é um dígito entre $[0 \ldots B-1]$, como sugerido pela equação 3.3.

Uma outra forma de ver o mesmo processo é construir uma cachoeira de divisões por B.

$$
\begin{array}{rl|l}
44 & 2 & \\
\hline
0_0 & 22 & 2 \\
\hline
 & 0_1 \quad 11 & 2 \\
\hline
 & \quad 1_2 \quad 5 & 2 \\
\hline
 & \quad\quad 1_3 \quad 2 & 2 \\
\hline
 & \quad\quad\quad 0_4 \quad 1 & 2 \\
\hline
 & \quad\quad\quad\quad 1_5 \quad 0 &
\end{array}
$$

O processo de divisões por dois termina quando o quociente chega a zero. O primeiro resto é o dígito binário que vai na posição 2^0. Para ressaltar este fato, indicamos cada resto com um subscrito, assim, o primeiro resto obtido foi 0_0, o segundo foi 0_1 e assim por diante.

Após colocar todos os restos na posição certa, teremos o seguinte:

7	6	5	4	3	2	1	0
		1_5	0_4	1_3	1_2	0_1	0_0

Como já foi dito, agrupam-se os dígitos de quatro em quatro bits, mas para isso é necessário preencher os bits 6 e 7. Quando os preenchermos com zeros, teremos o seguinte:

7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	0

que é o resultado correto. Para ter certeza, basta converter o número binário em decimal novamente:

$(00101100)_2 = (1 \times 2^5 + 1 \times 2^3 + 1 \times 2^2)_{10} = (32 + 8 + 4)_{10} = (44)_{10}$

Considere agora outra conversão: $(44)_{10} \rightarrow (?)_{16}$. Mais uma vez, a base origem é a base decimal e por isso o processo descrito nesta seção é a mais apropriada (para quem deseja fazer as contas em decimal, é claro).

Aqui teremos:

$$
\begin{aligned}
b_0 &= 44 \, mod \, 16 = 12 \\
b_1 &= \lfloor 44 \, div \, 16 \rfloor \, mod \, 16 = 2 \\
&\ldots
\end{aligned}
$$

O único cuidado aqui é que o primeiro resto é 12, que equivale a $(C)_{16}$.

Logo, $(44)_{10} \rightarrow (2C)_{16}$

3.1.4.2.2 Converter da base b para a base B usando a aritmética de B, a base destino Dado um número natural u, cuja representação na base b é $(u_n u_{n-1} u_{n-2} \ldots u_1 u_0)_b$, pode-se obter o número equivalente na base B, utilizando a aritmética de B, resolvendo o polinômio

$$u_n \times b^n + u_{n-1} \times b^{n-1} + \cdots + u_1 \times b^1 + u_0 \times b^0$$

, que é equivalente a

$$(u_n \times b + u_{n-1}) \times b + u_{n-2}) \times b + \times) \times b + u_1) \times b$$

```
1  {
2     i := n ;
3     r := vetor[i] ;
4     i := i-1 ;
5     b := 2 ;
6      Enquanto ( i >= 0 ), faça {
7        r := r*b + vetor[i];
8        i := i-1 ;
9     }
10 }
```

Algoritmo 2: Algoritmo de conversão de base b para B usando aritmética de B

Assim como na seção anterior, apresentamos um algoritmo (algoritmo 2) de conversão desta seção para auxiliar na compreensão.

Neste algoritmo, *r* é o resultado parcial do processo (ao sair do laço, é o resultado final), *b* é a base destino, `vetor[i]` indica os dígitos resultantes da conversão. Observe que cada dígito deste vetor está no intervalo [0 .. b-1].

Exemplos:

1. $(101100)_2 \rightarrow (?)_{10}$

$$\begin{aligned}
&(((((1\times 2+0)\times 2+1)\times 2+1)\times 2+0)\times 2+0)\\
&((((((2+0)\times 2+1)\times 2+1)\times 2+0)\times 2+0)\\
&((((2\times 2+1)\times 2+1)\times 2+0)\times 2+0)\\
&((((4+1)\times 2+1)\times 2+0)\times 2+0)\\
&((((5)\times 2+1)\times 2+0)\times 2+0)\\
&(((10+1)\times 2+0)\times 2+0)\\
&((11\times 2+0)\times 2+0)\\
&((22+0)\times 2+0)\\
&((22)\times 2+0)=44
\end{aligned}$$

2. $(2C)_{16} \rightarrow (?)_{10}$.

$$\begin{aligned}
(2\times 16)+12 &=\\
32+12 &= 44
\end{aligned}$$

3.1.4.3 Exercícios

Resolva os exercícios de conversão usando as regras práticas e as regras formais. Normalmente as regras práticas geram resultados mais rápidos quando os números são

pequenos e as regras formais geram resultados mais rápidos quando os números são grandes.

1. $(12345)_{10} \rightarrow (?)_{10}$. Este exercício é interessante para verificar porque os dois processos funcionam.

2. $(0101\ 1001\ 1100\ 1010)_2 \rightarrow (?)_{10}$

3. $(0110\ 0101\ 1010\ 1110)_2 \rightarrow (?)_{16}$

4. $(59CA)_{16} \rightarrow (?)_{10}$

5. $(FAAC)_{16} \rightarrow (?)_2$

6. $(2940)_{10} \rightarrow (?)_{16}$

7. $(3253)_{10} \rightarrow (?)_2$

8. O professor Pardal acha que seria mais conveniente representar os números na base oito (afinal, todos em Patópolis tem oito dedos). Já o professor Ludovico acha que seria mais interessante usar a base 11 devido às várias propriedades dos números primos. Em uma discussão, o professor Pardal afirmou que $(66337)_{11} > (273753)_8$, enquanto que o professor Ludovico afirmou que $(66337)_{11} < (273753)_8$. Quem tem razão?

9. O pato Donald estava passando por ali naquele momento, e entrou na conversa, afirmando que $(66337)_{11} = (66337)_8 = (66337)_{10}$, e muito se surpreendeu que os dois estudiosos não soubessem disso. Donald tem razão? Escreva um texto que seja capaz de fazer o pato Donald entender o seu erro de raciocínio.

10. Implemente os dois algoritmos apresentados. O programa deve ler três números do teclado (n- o número a ser convertido, b - a base origem e B - a base destino) e imprimir o número N_B, equivalente de n_b. Exemplos de funcionamento:

```
> converte 10011100 2 10
156
> converte 156 10 2
10011100
> converte 156 7 10
90
> converte 156 10 16
9C
> converte 156 7 11
82
```

11. Preencha a tabela abaixo assumindo que os números são naturais.

Binário	Decimal	Hexadecimal
0100 0001 0000 1111		
1001 1100 1111 0101		
	33212	
	10011	
		DEF0
		1111

3.2 Representação de números inteiros (com sinal)

É intuitivo tentar representar o sinal como um novo símbolo (no número -12, o símbolo `"-"` é que indica que o número é negativo), mas isso não é possível em binário porque os únicos símbolos existentes são `0` e `1`.

Sendo assim, a solução é adotar alguma convenção que indique que um número é negativo usando somente os símbolos `0` e `1`. O problema é que estes são os mesmos símbolos que são usados para representar dígitos, o que pode causar confusão.

A solução adotada para representar números inteiros negativos é chamado de "complemento de dois", porém esta não foi a única alternativa analisada. Para efeito histórico, serão apresentados dois dos mecanismos sugeridos e o porque deles não terem sido adotados, para só então apresentar o mecanismo de complemento de dois, que é o mecanismo efetivamente adotado em todos os computadores modernos.

Para efeito de padronização, consideramos que as máquinas-alvo deste texto são máquinas de 32 bits (ou seja, todos os números binários são representados com trinta e dois símbolos). como os números são grandes, por vezes mostraremos somente os quatro ou oito bits mais à direita, e por vezes somente escreveremos os quatro primeiros e os quatro últimos dígitos.

Sinal Magnitude Este mecanismo usa o bit mais à esquerda como sinal. Se este bit for `0`, o número tem sinal positivo, e se for `1`, o sinal é negativo. Assim, $(+1)_{10} = (0000\ldots0001)_2$, e $(-1)_{10} = (1000\ldots0001)_2$.

O inconveniente desta representação é que temos dois "zeros", o $(+0)_{10} = (0000\ldots0000)_2$ e o $(-0)_{10} = 1000\ldots0000_2$.

Complemento de um Aqui o bit mais à esquerda também indica o sinal. A diferença entre um número positivo e negativo é que o negativo é igual ao positivo com todos os bits invertidos. Por exemplo, $(+1)_{10} = (0000\ldots0001)_2$, e $(-1)_{10} = (1111\ldots1110)_2$.

Ambos os mecanismos tem o problema de representarem dois "zeros", além dificultarem a implementação de certas operações aritméticas em hardware. O mecanismo de complemento de dois, além de não ter o problema do $+0$ e -0, apresenta vantagens significativas na implementação em hardware, utiliza menor quantidade de circuitos e por conseqüência é mais simples, barato e rápido.

A idéia do complemento de dois é simples: o bit mais à esquerda (bit mais significativo) é multiplicado por -2^n e todos os demais são multiplicados por potências positivas. Exemplo:

$$(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100)_2 =$$
$$1 \times -2^{-31} + 1 \times 2^{-30} + 1 \times 2^{-29} + 1 \times 2^{-28} + \cdots + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 =$$
$$(-2.147.483.648)_{10} + (2.147.483.644)_{10} = (5)_{10}$$

Considere a tabela 3.5, que contém números em uma máquina de quatro bits. Esta tabela permite analisar comparativamente as representações de números inteiros e os números naturais nesta máquina.

Binário	Natural	Inteiro	Hexadecimal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	-8	8
1001	9	-7	9
1010	10	-6	A
1011	11	-5	B
1100	12	-4	C
1101	13	-3	D
1110	14	-2	E
1111	15	-1	F

Tabela 3.5: Relação entre números binários, naturais, inteiros e hexadecimal

Analisando a tabela, percebe-se que:

- o maior número positivo é sucedido pelo menor número negativo. Isto implica dizer que a comparação entre dois números inteiros é mais complicada do que a comparação entre dois números naturais. Na comparação entre dois números naturais, basta comparar os bits dos dois números da esquerda para a direita. O primeiro bit diferente indica que os números são diferentes. O maior deles é o que contém o "1". Em inteiros isso não funciona.
- Um mesmo símbolo hexadecimal (ou binário) pode corresponder a valores decimais diferentes (por exemplo: $A = -6 = 10$). Por isso, é necessário explicitar qual das duas representações está sendo usada. Ou seja, $A = -6$ se for complemento de dois (ou seja, número inteiro) e $A = 10$ se não for complemento de dois (ou seja, número natural).

3.2.1 Inversão do Sinal

A prática mostra que é importante saber inverter o sinal de um número representado em binário. Na notação decimal é simples, basta inserir ou retirar o símbolo "-", porém a coisa não é tão simples assim em binário.

Analise a tabela 3.5 novamente. Veja que $(+1)_{10} = (0001)_2$ enquanto que $(-1)_{10} = (1111)_2$. Compare $+2$ e -2 e assim por diante. Há um padrão: o número negativo corresponde à inversão de todos os bits do número positivo acrescido de $(1)_2$. Esta observação permite delinear o processo de inversão do sinal que será visto a seguir.

Como primeiro exemplo, considere a conversão de 2_{10} para -2_{10}.

1. Número original: $(2)_{10} = (0010)_2$
2. Inverte bits e soma 1: $(1101)_2 + (0001)_2 = (1110)_2$
3. Resultado final: $(1110)_2 = (-2)_{10}$

O mesmo processo vale para tornar um número positivo em negativo e um número negativo em positivo. Para demonstrar isso, verifique a conversão -2_{10} para 2_{10}.

1. Número original: $(-2)_{10} = (1110)_2$
2. Inverte bits e soma 1: $(0001)_2 + (0001)_2 = (0010)_2$
3. Resultado final: $(0010)_2 = (2)_{10}$

3.2.2 Aumento ou diminuição do número de bits

Uma "palavra" (*word*) é um nome usado para representar uma quantidade de bits. Quando se diz que uma máquina tem palavra de 32 bits, normalmente isso significa que a unidade de armazenamento e de transmissão de dados é de 32 bits, e também que cada registrador contém 32 bits.

Porém, por vezes é necessário trocar dados de uma máquina com uma palavra maior para uma máquina com palavra menor e vice-versa.

Por exemplo, se um dado de uma máquina de 64 bits for enviado para uma máquina de 32 bits, simplesmente serão ignorados os 32 bits de mais alta ordem (bits 32, 33, 34, até 63). Se estes bits estiverem zerados, não há perda de informação, mas se não estiverem zerados, o haverá perda de informação.

Por outro lado, por vezes, é necessário enviar dados de uma máquina de menos bits para uma máquina de mais bits (por exemplo, de uma máquina de 32 bits para uma máquina de 64 bits). Este processo é bastante simples, pois basta preencher os bits vazios à esquerda com o bit mais significativo da origem.

Nos exemplos abaixo, são convertidos os números 2 e -2 de 16 para 32 bits.

(**0**000 0000 0000 0010) → (**0000 0000 0000 0000 0**000 0000 0000 0010)
(**1**111 1111 1111 1110) → (**1111 1111 1111 1111 1**111 1111 1111 1110)

No exemplo acima, os bits “estendidos” estão grifados.

Não é necessário explicar porque este processo funciona quando o bit de mais alta ordem é zero. Afinal, um zero à esquerda é um zero a esquerda em qualquer sistema de numeração.

Porém, porque funciona com o 1 pode parecer mágico (talvez mais correto definir como matemágico). Para explicar isso, considere novamente um número inteiro representado em quatro bits, o número $(-1)_{10} = (1111)_2$. Se este número for estendido para cinco bits, a regra diz que o resultado em binário será $(11111)_2$. Isso significa dizer que $(1111)_2 = (11111)_2$ (considerando que o primeiro é -1 representado em uma máquina de quatro dígitos e o segundo é -1 representado em uma máquina de cinco dígitos.

Esta igualdade pode ser comprovada algebricamente da seguinte forma:

$$\begin{aligned}
(1111)_2 &= (11111)_2 \\
-2^3 + 2^2 + 2^1 + 2^0 &= -2^4 + 2^3 + 2^2 + 2^1 + 2^0 \\
-2^3 &= -2^4 + 2^3 \\
-2^3 - 2^3 &= -2^4 \\
2 * (-2^3) &= -2^4 \\
-2^4 &= -2^4
\end{aligned}$$

3.2.3 Operações Aritméticas em Naturais e Inteiros

Antes de abordar como efetuar operações aritméticas em binário, é interessante fazer uma análise sobre como é feita soma em decimal. A soma em hexadecimal segue o mesmo modelo aqui indicado.

Quando dois números de um dígito são somados, digamos $u + v$, a soma pode resultar em um número que resulta em um único dígito, mas se a soma for maior do que o total de dígitos representáveis no sistema de numeração adotado (por exemplo, $8 + 2 = 10$), então ocorre o que se chama de “vai um”. Basicamente, aparece um dígito à esquerda que é somado à casa das dezenas.

O processo de soma em binário ocorre exatamente da mesma forma. Soma-se todos os dígitos da direita para a esquerda, e pode ocorrer o “vai um”, que deve ser somado ao dígito imediatamente à esquerda.

Considere o que ocorre na soma de $(7+6)_{10}$, que em binário fica $(0111+0110)_2$.

(Carry)	$(1)_4$	$(1)_3$	$(1)_2$	$(0)_1$	
...	0	0	1	1	1
...	0	0	1	1	0
...	0	$(0)_4$1	$(1)_3$1	$(1)_2$0	$(0)_1$1

A soma se procede da direita para a esquerda:

1. $(1+0=01)_2$. Esta operação corresponde aos bits da coluna mais à direita. O resultado pode ser representado em um único dígito binário, porém para visualizar melhor o processo, todas as somas foram apresentadas em dois dígitos. Assim, o resultado desta soma é $(0)_1$1. O dígito mais à esquerda ($(0)_1$) é colocado no topo

da coluna imediatamente à esquerda, e corresponde ao "vai um". Isso completa a primeira parte da soma.

2. $((0)_1 + 1 + 1 = 10)_2$. Esta é a segunda coluna da direita para a esquerda. Observe que o valor $(0)_1$ foi somado junto com os outros dígitos desta coluna. O resultado é $(10)_2$, porém como o objetivo é mostra o "vai um", o primeiro dígito foi destacado como $(1)_20$

3. $((1)_2 + 1 + 1 = 11)_2$. Terceira coluna. Aqui o "vai um" ajudou na construção do resultado.

4. e assim por diante.

Observe que este processo é exatamente o mesmo que a soma entre números decimais, porém como os únicos dígitos são zero e um, alguns pode achar diferente.

O mesmo processo vale para a soma em hexadecimal. Como exercício, verifique que $(004A + 00AA = 00F4)$.

Um detalhe importante é que se os números estiverem representados em complemento de dois, o resultado do processo de soma visto aqui também irá gerar o resultado correto. .

Para verificar isso, considere o exemplo: $(7-6)_{10}$ abaixo. Neste exemplo, os números binários estão representados em oito bits para poupar espaço (se houver interesse em estender para 32 bits, basta adotar o mecanismo de aumentar o número de bits como visto na seção 3.2.2.

1. $((7-6)_{10}) = (7 + (-6))_{10})$

2. $(7)_{10} = (0000\ 0111)_2$

3. $(-6)_{10} = (1111\ 1010)_2$

4. $(0000\ 0111)_2 + (1111\ 1010)_2) = (0000\ 0001)_2$

É interessante, e talvez confuso, que $(0001)_2 + (1010)_2 = (1011)_2$ corresponde ao resultado correto se os números forem inteiros ou naturais. Se os números binários forem representações de números inteiros, a soma corresponde a $(1)_{10} + (-6)_{10} = (-5)_{10}$. Se os número binários forem representações de números naturais, então $(1)_{10} + (9)_{10} = (10)_{10}$. O problema é: como saber qual situação é qual.

Lembre que o computador armazena informações do mundo real, mas os representa internamente de forma diferente. Como o computador é capaz de efetuar operações mais rapidamente do que um ser humano, sua utilidade se restringe ao ganho de tempo em efetuar a operação. Não é responsabilidade do computador saber se o resultado é coerente é o programador.

Por isso, dada a operação de soma citada acima, o computador simplesmente irá efetuar a operação. O significado da operação é responsabilidade do programador.

3.2.3.1 Exercícios

1. Preencha a tabela abaixo assumindo que os números são inteiros (complemento de dois).

Binário	Decimal	Hexadecimal
0100 0001 0000 1111		
1001 1100 1111 0101		
	-32312	
	19911	
		DEF0
		1FF1

3.3 Limites e Overflow para Naturais e Inteiros

Considere um número natural de 32 bits. O menor número natural representável em 32 bits é zero (que corresponde a 32 bits iguais a zero). O maior número representável em 32 bits é $2^{32} - 1-$=4.294.967.295 (que corresponde a 32 bits iguais a um. Isso significa os números naturais representáveis em um computador só podem representar valores na faixa [0 ... 4.294.967.295]. Porém, o conjunto de números naturais contém uma quantidade infinita de números, e isso pode causar alguns problemas para representar algumas situações do mundo real em computador. Por exemplo, a quantidade de pessoas no mundo não é possível de ser representada em um natural de 32 bits.

Alguns poderiam dizer: para este problema, pode-se usar um natural de 64 bits (na pior das hipóteses, é possível emular um natural de 64 bits, ou mais, em uma máquina de 32 bits por software, porém isso acarreta perda sensível de desempenho). Porém, não é difícil de imaginar que existem quantidades que não poderiam ser representados em 64 bits. Não importa a quantidade de bits que é usada para representar a informação, sempre é possível encontrar um problema que não pode ser representado em um computador (na pior das hipóteses, contar o número de grãos de areia ou estrelas no céu :o).

Isso nos conduz a uma dedução óbvia, porém importante:

> O computador é uma entidade finita, e por isso existem limites para o que ele pode representar.

Felizmente, os problemas que são influenciados esta limitação não ocorrem com freqüência na prática.

Considere que um programa tem uma variável `i` declarada como inteiro sem sinal em uma máquina de 32 bits. Isto implica dizer que esta variável pode receber valores entre [0 ... 4.294.967.295]. Porém, suponha que algum programador desavisado escreva o seguinte comando: `i:=4294967296`. Este valor excede o valor máximo da variável, e ao ser compilado, ocorrerá uma mensagem indicando o ocorrido, e talvez indicando que o valor foi truncado (ou seja, que os bits acima do bit 31 serão descartados). O número $(4294967296)_{10}$ equivale a um `1` seguindo de 32 zeros. Por isso, se o programador imprimir esta variável, o valor que aparecerá no resultado será zero. Verifique.

Agora, considere os números inteiros.

A faixa de valores inteiros representadas dentro de um computador incluem números negativos. Em máquinas de 32 bits, esta faixa é a seguinte: [-2.147.483.648 . . . -1, 0, 1, . . . 2.147.483.647]. Observe que há um número negativo a mais do que positivo. Isso ocorre por causa do zero que não é nem positivo nem negativo. Como a quantidade de números existentes é par, o zero causa um desbalanceamento, e por isso deveria haver ou um positivo a mais ou um um negativo a mais. Em complemento de dois, há um negativo a mais, apesar de alguns textos dizerem simplesmente que convencionou-se que o zero é positivo (apesar disso ser uma heresia para os matemáticos).

Considerando o mesmo exemplo acima, se a variável `j` for declarada como inteiro, as atribuições de valores menores do que -2.147.483.648 e maiores do que 2.147.483.647 serão truncadas. Este truncamento também segue uma lógica. Por exemplo em, `j:=2147483648`, o valor de `j` será truncado para -2147483648.

Isso mostra que, quando mapeados em computador, todo e qualquer conjunto infinito de números (naturais, inteiros, reais - que serão vistos adiante -, etc.) tem um "menor número" e um "maior número", ou seja, tem limites.

Apesar das atribuições acima já darem uma idéia do problema, existe uma outra situação em que o limite também é ultrapassado: com o resultado de operações aritméticas. Por exemplo, quando uma variável tiver um valor positivo máximo, e for somado um a ele. O resultado não caberá no número de bits especificado, e o resultado final estará errado.

Por exemplo, para números inteiros, a operação `j:=2147483647 + 1` não ocasiona nenhum aviso durante a compilação. O resultado desta operação não é zero, mas sim o menor número negativo. Ou seja, após a atribuição acima, o valor de `j` será -2147483648 (confira!).

Quando o resultado de uma operação aritmética realizada pelo processador gerar um resultado que não "cabe" no tamanho da palavra, o processador gera um aviso do ocorrido.

O ato de o resultado da operação aritmética gerar um resultado que ultrapassa o limite de representação é chamado de "overflow", e só é detectado em tempo de execução.

Quando se trata de variáveis declaradas como números naturais, o processador avisa que ocorreu overflow quando o valor resultante da soma de dois números é menor do que qualquer um deles (na verdade o processador olha somente o que seria o valor do 33^o bit. Confira que ele sempre será igual a 1 se o resultado exceder o limite de representação.

Porém, quando se trata de números inteiros, as condições são um pouco diferentes.

Como exemplo, considere a soma $(1111\ldots1111)_2 + (0000\ldots0001)_2$. Se vistos como números inteiros, esta soma corresponde a $(+1)_{10} + (-1)_{10}$. Já sabemos que o resultado será $(0000\ldots0000)_2$, que é exatamente o que era esperado. Assim, diferente da soma de números naturais, esta soma não ocasionou overflow, e o resultado é o correto.

A pergunta que fica no ar é como o processador sabe qual regra usar? E a resposta é que o programador indica qual a operação que o processador deve executar. Por exemplo "Some considerando que os números são inteiros" ou "some considerando que os números são naturais". Não cabe aqui uma explicação detalhada sobre isso,

mas basta dizer que existe uma instrução para a soma com naturais (por exemplo `addu` (add unsigned) e uma para a soma com inteiros (por exemplo `add` (add integer).

Quando a operação for `addu`, o processador usa uma regra. Quando for `add`, usa outra regra.

O overflow de números inteiros ocorre de outra forma. Para simplificar a explicação, considere novamente uma máquina de quatro bits, cujos valores estão representados na tabela 3.5.

A soma de qualquer número positivo com negativo vai resultar em um número válido. Porém, somas entre dois positivos ou dois negativos podem causar problemas. Por exemplo, a soma $(0111)_2 + (0001)_2 = (1000)_2$, que em decimal seria $(7)_{10} + (1)_{10} = (-8)_{10}$, resultado incoerente, que indica a ocorrência de overflow.

A partir deste exemplo, é possível definir a regra para detectar a ocorrência de overflow em inteiros, que baseia-se na análise do sinal do resultado. A tabela a seguir indica todos os casos de overflow para inteiros.

Operação	**Operando A**	**Operando B**	**Resultado**
A+B	≥ 0	≥ 0	< 0
A+B	< 0	< 0	≥ 0
A-B	≥ 0	< 0	< 0
A-B	< 0	≥ 0	≥ 0

As operações citadas (`add` e `addu`) são somente duas operações de soma. Existem várias outras que não serão objetos deste texto.

3.3.0.1 Exercícios

1. Em uma máquina de N bits, qual a faixa de valores inteiros que podem ser representados?

2. Em uma máquina de N bits, qual a faixa de valores naturais que podem ser representados?

3. Considere que no futuro, torna-se viável utilizar um meio de armazenamento em computador que pode representar mais do que dois símbolos, ou seja, que o bit possa assumir não só dois valores, mas sim B valores diferentes. Por exemplo, se $B = 3$, então cada bit pode representar três informações diferentes (0, 1 e 2).

   (a) Em uma máquina de $N = 16$ destes bits, com $B = 3$, qual o maior número natural representável? E para $B = 4$?

   (b) E em uma máquina de $N = 16$ destes bits?

   (c) Qual o "ganho" deste bit comparado com o bit de dois valores? Indique uma fórmula que representa este ganho para N e B quaisquer.

4. Não foi explicado como fazer a multiplicação de dois números binários. Considerando que o processo é exatamente igual à multiplicação de decimais, preencha:

   (a) $(0000\ 0110)_2 \times (0000\ 0010)_2 = (\quad)_{10}$

(b) $(1111\ 1111)_2 \times (1111\ 1111)_2 = (\quad)_{10}$
(c) $(1111\ 1111)_2 \times (0000\ 0001)_2 = (\quad)_{10}$
(d) $(0000\ 1111)_2 \times (0000\ 1111)_2 = (\quad)_{10}$
(e) $(1000\ 1111)_2 \times (0000\ 1111)_2 = (\quad)_{10}$

5. Com relação à questão acima, como se detecta overlflow para a multiplicação de naturais e para a multiplicação de inteiros?

6. [Mon01] Quais os valores de x e y (em binário) na equação abaixo? Faça as contas em binário.

$$x+y = (0010\ 1000) x-y = (0000\ 1010)$$

7. [Mon01] Por um defeito de fábrica (ou maldade do fabricante), o computador de bordo e o odômetro de um carro mostra valores hexadecimais. No início de uma viagem, ele marcava $(A3FF)_{16}$. Ao chegar ao destino, o computador de bordo indicava que haviam passado $(A)_{16}$ horas, e o odômetro apontava $(A83C)_{16}$ kilômetros. Qual a distância percorrida? Qual a velocidade média da viagem? Respostas em hexa e em decimal.

8. Apresente o resultado das somas abaixo em complemento de dois. Indique a ocorrência de overflow com um asterisco à frente da resposta.

	Binário	Decimal	Hexa
$(11000011 + 00001111)_2$			
$(3214 + 1292)_{10}$			
$(F91F - ABCD)_{16}$			

9. No texto, foi dito que o valor de `j` após a atribuição `j:=2147483648` será -2147483648. Explique porque isso ocorre usando um modelo de 32 bits.

3.4 Operações Lógicas

As operações lógicas operam considerando que os parâmtros são "variáveis lógicas" (ou booleanas). Uma variável booleana é aquela que pode assumir somente dois valores: verdadeiro e falso. Quando mapeados para dentro do computador, estes dois valores são associados a 1 (verdadeiro) e 0 (falso).

A primeira idéia é usar um único bit para representar uma variável booleana, porém na prática isso não representa vantagem, uma vez que uma máquina de N bits opera com transferências de N bits por vez da memória para a CPU. Sendo assim, uma variável booleana (por exemplo na linguagem Pascal), ocupa 32 bits e trinta e um deles são zero. Somente o bit mais à direita (o menos significativo) é alterado.

Em outras linguagens, é possível usar o fato de que uma palavra tem potencialmente trinta e duas variáveis booleanas. A CPU tem instruções que permitem operações booleanas em palavras.

Esta seção aborda estas instruções, e para as explicações abaixo, considere a existência de duas palavras de 32 bits que correspondem às variáveis A e B.

3.4.1 And

A operação de “and” faz o “e lógico” bit a bit entre dois valores, como exemplificado abaixo:

	0000 0000 0000 0000 0000 1010 0010 0110
and	0000 0000 0000 0000 0000 1100 0000 0000
	0000 0000 0000 0000 0000 1000 0000 0000

Como o and é realizado bit a bit, no valor resultante, somente os bits em que as duas origens são 1 é que foram ativadas. Observe que o resultado pode ser analisado como se os todos os valores fossem números naturais. Neste caso, teríamos

$$A = (1010\,0010\,0110)_2 = (2598)_{10}$$
$$B = (1100\,0000\,0000)_2 = (1536)_{10}$$
$$Res = (1000\,0000\,0000)_2 = (2048)_{10}$$

Analisando como se fosse uma operação aritmética entre números naturais, o resultado não faz nenhum sentido. A CPU permite este tipo de operação booleana sobre números naturais. O programador o responsável por usar o resultado para algo construtivo. Se alguém encontrar um uso para o resultado desta operação lógica sobre números naturais, me avise por favor.

3.4.2 Or

A operação “or” faz o “ou lógico” bit a bit entre dois valores, como exemplificado abaixo:

	0000 0000 0000 0000 0000 1010 0010 0110
or	0000 0000 0000 0000 0000 1100 0000 0000
	0000 0000 0000 0000 0000 1110 0010 0110

Observe a diferença entre os resultados daqui e do AND.

3.4.3 Outros

As operações “and” e “or” podem ser resumidas nas tabelas-verdade abaixo:

	A	B	X		A	B	X
X := A AND B	0	0	0	X := A OR B	0	0	0
	0	1	0		0	1	1
	1	0	0		1	0	1
	1	1	1		1	1	1

Existem outras muitas operações, mas aqui apresentaremos somente mais uma, a operação “xor”.

	A	B	X
X := A XOR B	0	0	0
	0	1	1
	1	0	1
	1	1	0

A operação “xor” retorna 1 se os bits de entrada forem diferentes, e zero se forem iguais. É muito utilizada na confecção de circuitos lógicos (por exemplo para construir um somador seqüencial).

3.5 Representação de números reais (ponto flutuante)

A representação de números reais em computador foi uma questão desafiadora quando a computação ainda estava engatinhando. Desde o princípio percebeu-se que não haveria forma de representar todos os números reais.

Há uma quantidade infinita de números reais, e assim como nos números inteiros e reais, era visível que haveria dificuldade de representar números que ultrapassam determinado limite na faixa dos positivos e dos negativos.

Além disso, há uma quantidade infinita de números entre dois números reais quaisquer, um problema que não ocorre nos naturais e nem nos inteiros.

Logo, em qualquer solução adotada, seria necessário ter em mente que nem todos os números poderiam ser representados. A seção 3.1.4 foi uma introdução ao problema e apresentou uma solução inviável para ser implementada.

Várias soluções mais eficientes foram propostas, e várias companhias que fabricavam computadores criaram modelos próprios para representar os números reais, e com isso, uma miríade de representações diferentes surgiu (pode-se dizer, grosso modo, que cada empresa tinha a sua forma de representar números reais).

Era evidente que um padrão seria bem vindo. Por isso, a IEEE[5] organizou um grupo de trabalho para desenvolver um modelo de representação de números reais em computador. A versão preliminar deste trabalho foi divulgada em 1981, e a primeira versão foi divulgada em 1985. De lá para cá, praticamente todos os computadores do mundo adotaram este modelo.

O formato foi projetado para representar a faixa de números reais mais usado na prática e visa a eficiência de implementação em hardware, e esta seção concentra-se nos conceitos envolvidos com este formato. Informações mais detalhadas, assim como as questões matemáticas envolvidas com a validade do modelo podem ser encontrados facilmente na internet.

Esta seção está organizada da seguinte forma. A seção 3.5.1 revê a notação científica e como ela foi incorporada ao modelo IEEE-754. A seção 3.5.2 mostra como converter um número decimal para a notação IEEE-754 e vice-versa. A seção 3.5.3 mostra como se lida com limites e overflow e por fim, a seção 3.5.4 aborda assuntos interessantes, como valores especiais e a versão estendida do padrão.

3.5.1 A notação científica

A notação científica é aquela em que todos os números são representados no formato $D.N \times P^E$, onde `D` é um dígito diferente de zero, `N` é um conjunto de dígitos, `P` é a potência e `E` é o expoente.

Exemplos:

[5] *Institute of Electrical and Electronics Engineers*, um instituto sem fins lucrativos para o avanço das tecnologias relacionados com eletricidade.

Número Real	Notação Científica
$(0,000000001)_{10}$	$(1,0)_{10} \times 10^{-9}$
$(3.155.760.000)_{10}$	$(3,15576)_{10} \times 10^{9}$
$(0,1)_2$	$(1)_2 \times 2^{-1}$

O terceiro exemplo está na base binária, e é exatamente esta a representação adotada no modelo IEEE-754. Considere o modelo abaixo:

$$(1,xxx\ldots xxx)_2 \times 2^{(yyy\ldots yyy)_2} \tag{3.6}$$

Este modelo representa um número na base binária usando a notação científica. A idéia central do padrão IEEE-754 é representar os valores de $(xxx\ldots xxx)_2$ e $(yyy\ldots yyy)_2$ em 32 e 64 bits. Observe que $x = \{0,1\}$ e $y = \{0,1\}$, ou seja, cada dígito é binário.

Ponto Flutuante em precisão simples: No número ponto flutuante representado em 32 bits (chamado ponto flutuante de precisão simples), os valores de x e y são mapeados como indicado na figura 3.2

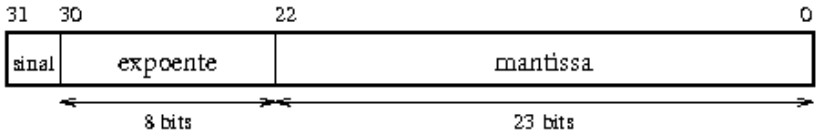


Figura 3.2: Formato ponto flutuante precisão simples

O bit 31 indica o sinal. $S = 0$ indica um número positivo, $S = 1$ indica um número negativo. O expoente é representado em oito bits, e corresponde aos $(yyy\ldots yyy)_2$ do modelo da equação 3.6, enquanto que a mantissa é composta por 23 bits, e corresponde aos $(xxx\ldots xxx)_2$ da equação 3.6.

Ponto Flutuante em precisão simples: No número ponto flutuante representado em 64 bits (são duas palavras de 32 bits consecutivos), os valores de x e y são mapeados como indicado na figura 3.3.

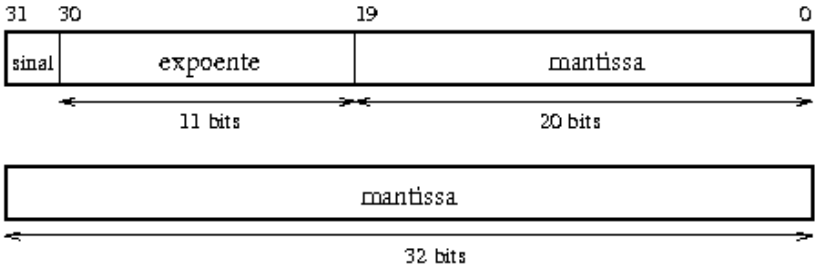


Figura 3.3: Formato ponto flutuante precisão dupla

O bit 31 indica o sinal. $S = 0$ indica um número positivo, $S = 1$ indica um número negativo. O expoente é representado em 11 bits, e corresponde aos $(yyy\ldots yyy)_2$ do modelo da equação 3.6, enquanto que a mantissa é composta por 52 bits, e corresponde aos $(xxx\ldots xxx)_2$ da equação 3.6.

A equação genérica para números em ponto flutuante é apresentada na equação 3.8, mas alguns livros apresentam a equação 3.9.

Observe que a mantissa corresponde aos dígitos que vem depois da vírgula (ou seja, $2^{-1}, 2^{-2}, \ldots$). A primeira impressão é que deveria haver alguma representação para o dígito que vem antes da vírgula, mas isso não é necessário. Este dígito deve ser diferente de zero, e em binário, isso corresponde a dizer que tem de ser igual a 1. Isto é deixado explícito na equação 3.9.

$$(-1)^S \times (1 + Mantissa) \times 2^{Expoente} \tag{3.7}$$

$$(-1)^S \times (1, Mantissa) \times 2^{Expoente} \tag{3.8}$$

$$\tag{3.9}$$

3.5.2 Conversões

A equação 3.8 é importante, uma vez que pode-se usá-la para converter um número ponto flutuante para decimal e vice-versa.

Exemplo 1 - Conversão do Expoente

Considere o seguinte número em binário:

$$B_1 = (z01100000zzzzzzzzzzzzzzzzzzzzzz)_2$$

Os dígitos *z* não são de interesse neste exemplo, que concentra-se no expoente.

O expoente está localizado entre os bits 30 e 23, e neste exemplo, corresponde a $(01100000)_2$. Se for convertido para decimal natural, temos que $(01100000)_2 = 2^5 + 2^6 = 32 + 64 = (96)_{10}$.

O expoente não está representado em complemento de dois, porém o padrão define que o expoente pode ser negativo.

Como já foi mencionado, o padrão foi criado para, entre outras coisas, otimizar o desempenho de operações sobre números que estiverem representados neste formato. Uma das operações mais utilizadas em computação é a comparação de números, e basta olhar para a tabela 3.5 para perceber que a comparação de números inteiros exige alguns "truques" para que a CPU saiba que $(-1)_{10} < (0)_{10}$, uma vez que as representações binárias sugerem o contrário. Estes "truques" geram custo computacional, resultando num desempenho aquém do desejado.

Quando foi criado o padrão, a operação de comparação foi levada em consideração, e procurou-se criar um modelo em que a simples comparação binária entre dois números pudesse ser utilizada[6]. Por esta razão, o expoente não é representado em complemento de dois.

A solução adotada é chamada de deslocamento (*bias*). A idéia é simples: o expoente que vai na equação 3.8, que denominaremos de "E" é obtido pela equação

$$E = e - bias$$

onde e é o expoente extraído dos bits 23-30 da representação binária do número em ponto flutuante, e no padrão IEEE-754, $bias = 127$. É importante destacar que a solução de representar números negativos com deslocamento é anterior à definição do padrão. Por isso, pode-se encontrar material que fala sobre um deslocamento com outro valor, como 126 (muito comum em livros de métodos numéricos). Porém, no padrão IEEE-754, o deslocamento foi fixado em 127 ($2^7 - 1$) para números ponto flutuante de 32 bits. Para 64 bits, o bias é 1023 ($2^{10} - 1$).

Assim, para 32 bits, as faixas de valores de E e e são:

$$0 < e < 255$$
$$-127 < E < 128$$

Voltando ao exemplo, $e = (96)_{10}$, e com isso, $E = e - 127 = 96 - 127 = -31$, que e o expoente desejado.

Exemplo 2 - Conversão da mantissa

Considere o seguinte número em binário:

$$B_2 = (zzzzzzzzz01000000000000000000000)_2$$

[6]Ou seja, compara-se os bits 31 de cada número binário. Se forem diferentes, aquele que tiver o 1 é o maior. Se forem iguais, compara os bits 30, e assim por diante

Os dígitos *z* não são de interesse neste exemplo, que concentra-se na mantissa.

A mantissa corresponde à parte que vem depois da vírgula, e na equação 3.8 é representada como $1+M$, porém também poderia ser representada como $1,M$ (equação 3.9).

Nesta última notação, o número binário do exemplo seria $1,01000000000000000000000$, ou seja, $1\times 2^0+1\times 2^{-2}=1+0,25=(1,25)_{10}$. A mantissa neste caso é $M=0,25$.

Não há segredo em obter o número decimal a partir da mantissa. É só lembrar que o primeiro dígito corresponde a 2^{-1}, o segundo a 2^{-2}, e assim por diante.

Exemplo 3 - Mantissa e expoente

$$B_3=(11000000001000000000000000000000)_2$$

Usando a equação 3.8 como base, extrai-se as seguintes informações:

$$\begin{aligned} S&=(1)_2=1\\ E&=(10000000)_2=(128-127)_{10}=(1)_{10}\\ M&=(01000000000000000000000)_2=2^{-2}=(0.25)_{10}\end{aligned}$$

Aplicando os resultados acima na equação 3.8, obtém-se o seguinte:

$$\begin{aligned} &(-1)^S\times(1+M)\times 2^E\\ &(-1)^1\times(1+0,25)\times 2^1=\\ &(-1)\times 1,25\times 2=\\ &-2,5\end{aligned}$$

Exemplo 4 - Mantissa e expoente

$$B_4=(01000010011000000000000000000000)_2$$

O primeiro passo é extrair as informações pertinentes:

$$\begin{aligned} S&=(0)_2=0\\ E&=(10000100)_2=128+4-127=(5)_{10}\\ M&=(11000000000000000000000)_2\\ &=+2^{-1}+2^{-2}=0,5+0,25=0,75\end{aligned}$$

Ao aplicar incógnitas na equação 3.8, teremos

$$
\begin{aligned}
(-1)^S \times (1+M) \times 2^E \\
(-1)^0 \times (1+0,75) \times 2^5 = \\
1 \times (1,75) \times 2^5 = \\
1,75 \times 32 = \\
(56)_{10}
\end{aligned}
$$

Exemplo 5 - Decimal para PF
Agora, considere o caminho inverso:

$$(-0,5)_{10} \rightarrow (?)_2$$

Para fazer esta conversão, o caminho mais simples é converter o número em questão direto para binário, no formato apresentado na seção 3.1, aquele formato que não é usado na prática, mas que será bastante útil aqui.

Usando aquele formato, pode-se encontrar uma forma de representação bastante próxima do formato normalizado.

$$(-0,5)_{10} = -(0,1000)_2 \times 2^0 = -(1,0)_2 \times 2^{-1}$$

O último passo normalizou o número binário. Agora, verifique a semelhança entre este formato e a equação 3.8:

$$
\begin{aligned}
-(1,0)_2 \times 2^{-1} \\
(-1)^S \times (1+M) \times 2^E
\end{aligned}
$$

Não é difícil concluir que:

$$
\begin{aligned}
S = 1 \\
E = -1 \\
E = e - 127 \\
e = E + 127 = -1 + 127 = (126)_{10} = (01111110)_2 \\
M = 0 = (00000000000000000000000)_2
\end{aligned}
$$

Colocando o resultado no formato ponto flutuante de 32 bits, tem-se:

$$(-0,5)_{10} = (10111111000000000000000000000000)_2$$

Exemplo 6 - Decimal para PF

$$(0,75)_{10} \rightarrow (?)_2$$

Passo 1: converter para binário normalizado.

$$(0,75)_{10} = (0,0110)_2 \times 2^0 = (1,1)_2 \times 2^{-2}$$

Passo 2: Comparar o número binário obtido com a equação 3.8:

$$\begin{aligned} (1,1)_2 \times 2^{-2} \\ (-1)^S \times 1 + M \times 2^E \end{aligned}$$

Passo 3: Extrair as incógnitas da equação:

$$\begin{aligned} S &= 0 \\ E &= -2 \\ E &= e - 127 \\ e = E + 127 = -2 + 127 &= (125)_{10} = (01111101)_2 \\ M = 0,1 &= (10000000000000000000000)_2 \end{aligned}$$

Passo 4: Por fim, colocar no formato PF binário:

$$(0,75)_{10} = (00111110110000000000000000000000)_2$$

3.5.3 Limites e Overflow para Ponto Flutuante

Como já foi destacado anteriormente, a quantidade de números reais é infinita, enquanto que a quantidade de números diferentes que podem ser representadas na notação ponto flutuante é finita (são $2^{32} = 4.294.967.296$, um pouco mais de quatro bilhões de representações diferentes em 32 bits e $2^{64} = 18.446.744.073.709.551.616$ em 64 bits)

A faixa de valores representáveis para ponto flutuante de precisão simples é indicado na figura 3.4

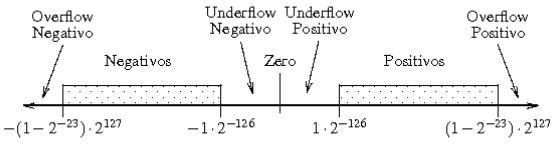


Figura 3.4: Faixa de valores válidos para ponto flutuante precisão simples

Como esperado, alguns números muito grandes não são representáveis, assim como outros muito pequenos. O termo *overflow* é aqui usado para indicar um número muito grande e o termo *underflow* é usado para indicar um número muito pequeno.

A figura destaca o underflow próximo a zero, e só. A primeira idéia é que entre quaisquer dois números pode ocorrer underflow, mas não é assim. Há uma questão técnica que diferencia underflow do `erro de representação`, que é aquele que ocorre entre dois números quaisquer.

Erro de Representação Entre quaisquer dois números reais existe uma infinidade de números. Nem todos podem ser representados, ocasionando "buracos", ou seja, números reais que não podem ser representados. Estes "buracos" são chamados de erros de representação.

Underflow É um caso especial de erro de representação que ocorre próximo ao zero, pois há um único grande buraco (a faixa de valores não representáveis é maior). Quando a faixa de valores é muito grande, o erro de representação indicado por esta faixa é chamado de underflow.

Um problema curioso, apesar de não ser comum, poderia ser ocasionado pelo seguinte código:

```
1 { var r,x : real
2    r := 1 ;
3     Enquanto ( r <> 0 ), faça {
4           ... r := r / 1.5 ;
5       }
6    }
```

Este código contém um erro sutil. Matematicamente, `r` jamais será igual a zero. Porém, é comum alguns programadores não pensarem neste tipo de detalhe, e implementarem código incorreto como este.

Quando `r` for suficientemente pequeno, digamos 1×2^{-126}, a operação `r:=r/1.5` resultará em um número na faixa de underflow positivo (veja que a faixa é grande), e poderá ser arredondado para 1×2^{-126} ("escorrega para cima"). Na próxima iteração, ocorrerá exatamente a mesma coisa de tal forma que `r` jamais será igual a zero. O laço segue indefinidamente.

Este erro não é nada fácil de encontrar.

Mais interessante é trocar `1.5` por `2`, quando o programa termina (matematicamente ele não deveria terminar também!).

Este programa entra em loop infinito sem motivo aparente. Este e outros programas que "escorregam" no underflow são bastante difíceis de detectar.

Para não recair neste problema, a solução é nunca comparar um número real com zero. Neste caso, o teste deveria ser:

```
Enquanto ( r >= epsilon ), faça {
```

Onde epsilon é o menor número positivo, ou seja, o primeiro número fora do underflow (em ponto flutuante de 32 bits, epislon=2^{-126})[7].

[7] As linguagens de programação normalmente dão suporte ao programador usar este número como cons-

Pode-se imaginar que entre quaisquer outros números este mesmo problema também pode ocorrer com igual probabilidade, mas não é assim. Entre quaisquer outros números, a faixa de números não representáveis é menor, e por isso é mais difícil o resultado ser arredondado sempre para o mesmo valor.

Na próxima seção, será visto que a faixa de underflow era maior, e que o padrão definiu uma nova classe de representação, chamada números denormalizados. O erro citado acima ocorre nesta nova classe.

3.5.4 Questões interessantes

Esta seção trata de três assuntos: (1) valores especiais (2) operações aritméticas do formato IEEE-754 e (3) representação estendida de 80 bits.

3.5.4.1 Valores Especiais

Em função da grande faixa de números não representáveis próximos a zero, o padrão IEEE-754 encontrou uma forma de aumentar a quantidade de números na faixa de underflow. Esta forma é chamada de número denormalizado, que é indicado quando o $e = E_{max} - 1$, o que em ponto flutuante de precisão simples significa $e = 255$. Este esforço não foi em vão, apesar de apesar de sua criação, a faixa de valores que está entre $[0..2^{-126}]$ não tem nenhum representante. Sem a criação dos números denormalizados, esta faixa seria maior.

Os números denormalizados valem tanto para ponto flutuante de precisão simples quanto ponto flutuante de precisão dupla. Generalizando, temos que um número denormalizado segue equação 3.10.

$$(-1)^S \times (0 + Mantissa) \times 2^{-126} \tag{3.10}$$

Observe que esta equação não contempla o valor do expoente, que é constante (2^{-126}). Este é um dos valores especiais da notação IEEE-754, porém existem outros. A tabela 3.6 apresenta todos os valores especiais do padrão.

Expoente	Mantissa	Significado
255	$\neq 0$	NaN (Not a Number)
255	0	$-1^S \times \infty$
0<e<255	$\neq 0$	$(-1)^S \times (1, Mantissa) \times 2^{e-127}$
0	$\neq 0$	$(-1)^S \times (0, Mantissa) \times 2^{-126}$
0	0	Zero

Tabela 3.6: Valores Especiais do padrão IEEE-754

tante. Por exemplo, na linguagem “C”, o cabeçalho `float.h` contém a constante `FLT_MIN`, que é o menor número ponto flutuante. O teste pode ser feito diretamente com esta constante, resolvendo o problema.

São cinco valores especiais. A primeira linhas, temos NaN, que é útil para representar resultados que não são números reais, como $\sqrt{-1}$. Na segunda linha, $+\infty$ e $-\infty$ são úteis para representar resultados de operações como $\frac{1}{0}$. A terceira linha mostra os números ponto flutuante normalizados enquanto que a quarta linha mostra os números ponto flutuante denormalizados. Finalmente, a quinta linha mostra o zero. Esta última linha pode surpreender, pois zero não deveria ser um número especial. Mas como não há como valores para o expoente, mantissa e sinal que resultem no número zero (verifique!), ele foi colocado como um valor especial.

3.5.4.2 Como a CPU lida com Operações Aritméticas

Na seção que tratou de limites para inteiros (seção 3.3), foi explicado que cada operação aritmética que a CPU executa contém parâmetros que ela assume que estão no formato apropriado para aquela instrução. Assim, uma instrução `addu` assume que os parâmetros são números naturais e uma instrução `add` assume que os parâmetros são inteiros.

Agora, considere o conjunto de números reais. A representação de reais foi vista nesta seção e é bem mais complicada do que a representação de naturais ou inteiros. As operações aritméticas são também mais complicadas.

Assim como no caso anterior, as operações sobre reais também requerem instruções específicas, que aqui será considerada como `add.s` (add FP single precision)[8]. Quando a CPU receber esta instrução para executar, ela considera que os operando estão no formato ponto flutuante. Se o programador se enganar e colocar parâmetros inteiros ou naturais, **a CPU não irá verificar**. Ela simplesmente irá executar a operação como se os números estivessem representados em ponto flutuante.

Nenhuma instrução verifica se os parâmetros são apropriados para aquela instrução (por exemplo, na soma de reais, CPU não confere se os parâmetros são números reais. Aliás, nem tem como fazer isso, pois todos os valores estão em binário sem indicação de tipo). Assim, se o programador colocar um número real em um parâmetro e um caractere em outro, e pedir para somá-los como inteiros sem sinal, a operação será realizada, e o resultado será ... inusitado.

Para muitos isso é surpreendente, uma vez que em nossa vida cotidiana, é comum fazer somas independente do tipo dos operandos. Porém em computação este cuidado é imprescindível, pois existem algumas diferenças. A mais simples de notar é que o processo de soma de reais é diferente do processo de soma de inteiros e naturais. Também há algumas diferenças entre a soma de números naturais e inteiros, não no processo, mas nas condições de erro.

> Nenhuma instrução verifica se a execução de uma instrução "faz sentido" ou não. Simplesmente a executa, e não se preocupa com as conseqüências que são responsabilidade do programador.

Por vezes, enganos como estes geram resultados surpreendentes e em outros casos causam cancelamento da execução do programa. As conseqüências são geralmente imprevisíveis.

[8] que é diferente de `add.d`, que soma ponto flutuante (FP=*floating point*) de precisão dupla

Por isso, é importante ter em mente que uma operação aritmética sobre inteiros exige instrução sobre inteiros, e que os operandos sejam inteiros. A CPU só garante que fará a soma como se os números fossem inteiros, gerando as excessões (overflow, por exemplo) para inteiros. Na CPU usada na segunda parte deste texto, não existe operação aritmética em que os operandos são misturados (soma inteiro com real gerando natural, por exemplo). Algumas CPUs antigas tinham algumas instruções assim, porém em função da complexidade de implementá-las, e da pequena utilização das mesmas, foram quase que inteiramente abolidas em CPUs mais modernas (em especial naquelas classificadas como RISC).

Como não há instrução para somar parâmetros de tipos diferentes, a responsabilidade da conversão fica por conta do programador. Como exemplo, considere que é necessário somar um número real com um número inteiro. Neste caso, o programador deverá primeiro converter um destes para o formato do outro. Exemplo: para somar $(1,2)_{PF} + (10)_{10}$, o processo poderia ser o seguinte:

1. $(1,2)_{PF} + (10)_{10}$
2. $(1,2)_{PF} + (10)_{PF}$ (converte $(10)_{10} \rightarrow (10)_{PF}$)
3. $= (11,2)_{PF}$.

Neste caso, o resultado está em notação ponto flutuante. O segundo passo poderia ter convertido $(1,2)_{PF} \rightarrow (1)_{10}$, porém aqui haveria perda de precisão.

3.5.5 Operações aritméticas em Números Reais

As operações aritméticas em números reais exigem mais cuidados do que as operações sobre naturais e sobre inteiros. O motivo é que os números reais são divididos em três partes, enquanto os demais são contidos em uma única parte.

A seção 3.5.5.1 será explica o funcionamento da operação de soma em ponto flutuante enquanto a seção 3.5.5.2 explica o funcionamento da multiplicação. As seções 3.5.5.3 e 3.5.6 mostram exemplos do tipo de erro de programação que pode ocorrer em função dos erros de arredondamento previstos nas operações aritméticas em ponto flutuante.

3.5.5.1 Soma de números reais

Como introdução ao assunto, considere a soma de números reais representados na base decimal, porém com restrição de espaço: a mantissa deve ser representada em três dígitos enquanto que o expoente em dois. O exemplo abaixo é igual ao contido em [DJ97].

Soma: $(9,999)_{10} \times 10^1 + (1,610)_{10} \times 10^{-1}$

O processo de soma nestes casos é conhecido, mas aqui iremos dividí-lo em quatro fases:

Fase 1 Alinhar o expoente (do menor):
$(1,610)_{10} \times 10^{-1} \rightarrow (0,016)_{10} \times 10^1$

Fase 2 Somar as mantissas:
$(09,999)_{10} \times 10^1$
$(00,016)_{10} \times 10^1$
$(10,015)_{10} \times 10^1$

Fase 3 Normalizar (se necessário):
$(10,015)_{10} \times 10^1 \rightarrow (1,0015)_{10} \times 10^2$

Fase 4 Arredondar (lembre-se que em nosso exemplo a mantissa ocupa somente três dígitos). O processo de arredondamento é o seguinte: se o quatro dígito após a vírgula for [0,1,2,3,4], os dígitos após o terceiro são ignorados. Se for [5,6,7,8,9], o arredondamento soma um ao terceiro dígito. Assim, temos:
$(1,0015)_{10} \times 10^2 \rightarrow (1,002)_{10} \times 10^2$

A descrição acima é muito próxima ao que ocorre realmente em uma CPU. Observe que a fase 1 "jogou fora" um dígito. A fase quatro fez um arredondamento, perdendo precisão. Isso também ocorre na prática em CPUs, porém normalmente não é perceptível e não causa maiores problemas.

A figura 3.5 é mais precisa sobre o processo:

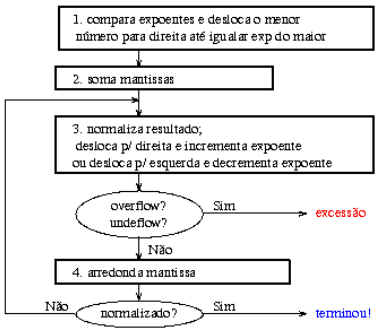


Figura 3.5: Adição em Ponto Flutuante

Nesta figura, cada fase está detalhada em um retângulo. A novidade é que entre as fases 3 e 4 há um teste que verifica se o resultado gerou overflow ou underflow. Se houve algum deles, a soma não se completa e o processo gera uma excessão que geralmente ocasiona o cancelamento da execução do programa.

Por fim, um exemplo completo. Considere a subtração $(0.5)_{10} - (0.4375)_{10}$, e que a representação binária usa três bits para a mantissa.

Prólogo: Converter para binário (PF)

$$(0.5)_{10} \rightarrow (0.1)_2 \rightarrow \mathbf{1.000 \times 2^{-1}}$$
$$(-0.4375)_{10} \rightarrow (-7/2^4) \rightarrow (-111)_2 \times 2^{-4} \rightarrow$$
$$\rightarrow -0.0111 \times 2^{-1} \rightarrow \mathbf{-1,110 \times 2^{-2}}$$

Ou seja, a soma a ser efetuada é: $(1.000)_2 \times 2^{-1} + (-1,110)_2 \times 2^{-2}$

Fase 1 Alinhar Expoentes:

$$(-1,110)_2 \times 2^{-2} \rightarrow (-0.111)_2 \times 2^{-1}$$

Fase 2 Somar Mantissas:

$$\begin{aligned}(+1.000)_2 \times 2^{-1}\\ (-0.111)_2 \times 2^{-1}\\ (= 0.001)_2 \times 2^{-1}\end{aligned}$$

Fase 3 Normalizar:

$$(0.001)_2 \times 2^{-1} \rightarrow (1.000)_2 \times 2^{-4}$$

Fase 4 Arredondar: (já está).

Resultado:

$$\begin{aligned}(1.000)_2 \times 2^{-1} + (-1,110)_2 \times 2^{-2} &= (1.000)_2 \times 2^{-4}\\ &= (0,5)_{10} - (0.4375)_{10} = (0.0625)_{10}\end{aligned}$$

3.5.5.2 Multiplicação de números reais

O processo de multiplicação de números reais representados em ponto flutuante envolve basicamente a soma dos expoentes e a multiplicação das mantissas como detalhado na figura 3.6.

Como exemplo de funcionamento, considere a multiplicação $(0.5)_{10} \times (0.4375)_{10}$. Cada fase abaixo corresponde aos retângulos numerados da figura 3.6.

Prólogo: Converter para binário

$$\begin{aligned}(0.5)_{10} &\rightarrow (0.1)_2 \rightarrow 1.000 \times 2^{-1}\\ (-0.4375)_{10} &\rightarrow -0.0111 \times 2^0 \rightarrow -1,110 \times 2^{-2}\end{aligned}$$

Ou seja, a operação a ser efetuada é: $(1.000)_2 \times 2^{-1} \times (-1,110)_2 \times 2^{-2}$

Fase 1 Soma expoentes, com deslocamento.

$$\begin{aligned}(-1+127) + (-2+127) - 127 &= 124\\ 124 &= 127 - 3\end{aligned}$$

O expoente resultante é $(-3)_{10}$. Observe que há um "atalho" para chegar a este resultado $(-1 + -2 = -3)$, porém não foi usado aqui. O motivo é destacar a

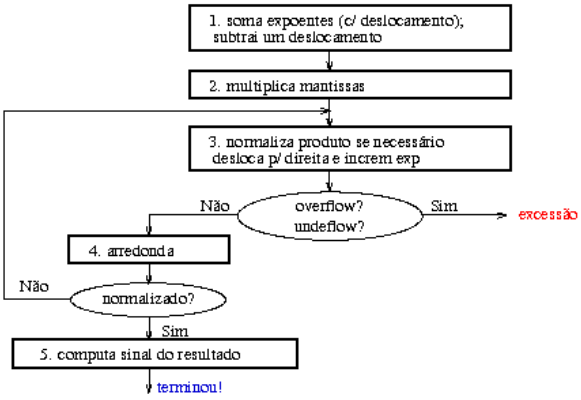


Figura 3.6: Multiplicação em ponto flutuante

representação interna do expoente com deslocamento. Internamente, o expoente $(-1)_{10}$ é representado como $(01111111)_2 = (127)_{10}$, e não destacar isso poderia gerar confusões.

Fase 2 Multiplica Mantissas:

$$\begin{aligned}(1.000)\\ \times(1.110)\\ (=1.110)\end{aligned}$$

Fase 3 Normalizar: (já está).

$$(1.110)_2 \times 2^{-3}$$

Fase 4 Arredondar: (já está).

Fase 5 Calcula o sinal: Positivo $\times$ Negativo = Negativo

Resultado

$$(1.000)_2 \times 2^{-1} \times (-1,110)_2 \times 2^{-2} = (-1.110)_2 \times 2^{-3}$$
$$(0,5)_{10} \times (0.4375)_{10} = (-0.21875)_{10}$$

3.5.5.3 O efeito de um erro

Observe mais uma vez a figura 3.6. Existem dois pontos de perda de precisão: na normalização e no arredondamento. Esta perda de precisão ocorre somente nos bits menos significativos da mantissa, e é normalmente inofensiva. Porém se a operação se repete, é possível que a perda de precisão seja "propagada" para os bits mais significativos. O mesmo problema ocorre com a soma.

Para exemplificar este problema, considere o a algoritmo 9, que foi "descoberto" por Flávio Miyazawa [Flá00].

```
1 {
2 var terco : real;
3 i : integer;
4    terco  := 1.0 / 3.0;
5     para (  i := 0  to 30  do {
6        terco  := 4.0 * terco - 1 ;
7        writeln (i, terco)  ;
8    }
9 }
```

Matematicamente, a variável `terco` sempre terá o valor $\frac{1}{3}$, pois:

$$4.0 * \frac{1}{3} - 1 = \frac{4.0 * 1 - 3}{3} = \frac{4.0 - 3}{3} = \frac{1}{3}$$

Porém, devido aos problemas de representação (a fração $\frac{1}{3}$ é uma aproximação em ponto flutuante, e quando representado como soma de potências de dois é somente uma aproximação), o resultado obtido é o seguinte:

```
00 0.333333
01 0.333333
02 0.333334
03 0.333336
04 0.333344
05 0.333374
06 0.333496
07 0.333984
08 0.335938
09 0.343750
10 0.375000
11 0.500000
12 1.000000
13 3.000000
14 11.000000
15 43.000000
16 171.000000
```

```
17 683.000000
18 2731.000000
19 10923.000000
20 43691.000000
21 174763.000000
22 699051.000000
23 2796203.000000
24 11184811.000000
25 44739244.000000
26 178956976.000000
27 715827904.000000
28 2863311616.000000
29 11453246464.000000
```

Após 30 iterações, o valor final de `terco` não é exatamente o que a matemática indica como o correto. Em algumas aplicações, o resultado gerado após a segunda iteração já seria inaceitável.

Se a variável `terco` for representado como um ponto flutuante de precisão dupla, o erro demora mais para aparecer, mas também aparece (`terco` fica negativo. Confira.).

Este erro é muito particular, e não há necessidade de ficar assustado com ele. O valor $\frac{1}{3} = (0,33333\ldots)$, ou seja, é uma dízima periódica em decimal (em decimal já é uma aproximação) que também gera uma dízima periódica em PF. A divisão também é por uma dízima. Isso faz com que em cada operação ocorra perda de informação. Visto desta forma, o resultado não é surpreendente.

3.5.6 Comparação

Um último detalhe a ser levantado está relacionado com as comparações. O código abaixo foi extraído de [Gol91].

```
1 {
2    double q;
3    q = 3.0/7.0;
4    if (q == 3.0/7.0)
5       printf(¨ Equal¨ );
6    else
7       printf(¨ Not Equal¨ );
8    return 0;
9 }
```

O código está em linguagem C. A variável `q` é uma variável real de precisão dupla. Ao ser compilado e executado, este programa irá imprimir `Equal`.

Porém, ao alterar o tipo da variável `q` para `float` (real de precisão simples), o resultado será `Not Equal`.

O motivo está em ações diferentes que o compilador irá adotar. Na linha de atribuição `q = 3.0/7.0;`, o compilador irá considerar que a divisão deve ser feita usando instruções de precisão dupla. Já a comparação `(q == 3.0/7.0)`, será quebrada em vários passos: (1) divisão de precisão dupla (2) converte resultado para ponto flutuante de precisão simples (3) compara.

Mais uma vez temos números que, quando representados em ponto flutuante, correspondem à uma dízima periódica, ou seja, um caso especial. Neste caso especial, a conversão de precisão dupla para simples perde precisão, ocasionando o erro.

Para solucionar este erro, é necessário dizer ao compilador que ele deve considerar que a divisão deve levar em conta que as duas constantes são de precisão simples como indicado abaixo:

```
1 {
2    float q;
3    q = 3.0/7.0;
4    if (q == (float) 3.0/ (float) 7.0)
5       printf(¨ Equal¨ );
6    else
7       printf(¨ Not Equal¨ );
8    return 0;
9 }
```

Em linguagem C, é possível fazer conversão de tipos. No caso, ao indicar que tanto a primeira quanto a segunda constante são do tipo ponto flutuante de precisão simples, a divisão adotada é a de precisão simples, e o resultado é igual ao volor contido em `q`.

Existem grande quantidade de material disponível em livros e na internet que apontam para estes casos particulares. Existem pesquisadores “especializados” em procurar e apontar os problemas do padrão IEEE-754.

Porém, isso não quer dizer que o padrão é ruim, ou que irá ser substituído, apesar de gerar alguns erros. Ele poderá ser atualizado (aliás, atualmente - 2008 - há um grupo trabalhando nisso), mas continuará sendo o padrão mundialmente adotado, pois os problemas aqui são **menores** do que em outros modelos propostos.

Afinal, os problemas apontados são casos particulares, que dificilmente ocorrem na prática cotidiana. De qualquer forma, levando em conta que as leis de Murphy podem afetar a todos que não se previnem, é bastante útil adotar as dicas aqui apontadas em seus programas.

O problema de perda de precisão nas operações sobre números reais gerou uma área especializada, chamada “Métodos Numéricos”, que procura encontrar algoritmos que minimizem os erros causados por estas operações. Por vezes estes algoritmos são até mais lentos do que os algoritmos “naturais”, porém como minimizam o erro, são preferíveis.

3.5.6.1 Representação de 80 bits

Aqui foram apresentados os formatos de 32 bits (precisão simples) e 64 bits (precisão dupla) do padrão IEEE-754. Além destes, foram propostos outros dois: precisão simples estendida (43 bits) e precisão dupla estendida (≥ 79 bits).

Durante a elaboração do projeto, um outro formato foi proposto (formato projetivo), porém para manter o padrão simples, ele não foi incorporado.

Apesar disto, a intel adotou este formato nos processadores da linha 80x87, co-processadores aritméticos. Este co-processador foi incorporado aos processadores da família 80x86, e é atualmente usado para operações em ponto flutuante destes processadores.

Na memória, os números reais são armazenados no formato de precisão simples ou dupla. Quando são carregados para a CPU, estes números são convertidos para 80 bits (precisão dupla estendida) e as operações são efetuadas em 80 bits. Quando os valores são armazenados na memória, eles são reconvertidos para precisão simples ou dupla.

Outras opções de implementação do projeto 80x87 foram também interessantes, porém acarretaram em perda de desempenho. Para explicações mais detalhadas sobre este tema, consulte [DJ96].

3.5.7 Exercícios

1. Implemente um programa que atribui um valor positivo a uma variável ponto flutuante (qualquer valor) e sucessivamente subtraia esta variável de 2^{-126}. A cada iteração imprima o valor da variável. Explique o ocorrido.O motivo é underflow ou é o arredondamento?

2. Explique porque o programa da seção 3.5.3 não entra em loop se a divisão for efetuada por 2.0 e não por 1.5.

3. Sejam x, y ez três valores quaisquer representados em notação ponto flutuante precisão simples. Alguém afirma que dependendo dos valores de x, y ez, é possível que o resultado da operação $(x+y)+z \neq x+(y+z)$. Explique se isso pode ocorrer e porque.

4. Converta para ponto flutuante:

   (a) $(0,125)_{10} \rightarrow (\quad)_{PF}$

   (b) $(0,4)_{10} \rightarrow (\quad)_{PF}$

5. Indique o que o padrão de bits abaixo representa se for visto como:

   1000 1111 1110 1111 1100 0000 0000 0000

   (a) número natural

   (b) número inteiro

   (c) número real (PF)

6. **Representação de Ponto Flutuante em 8 bits:**(Roberto Hexsel) Existem 256 valores diferentes que podem ser representados em ponto flutuante (PF) com 8 bits, e estes valores não são distribuídos uniformemente na reta dos Reais. A representação PF-8 é uma versão (muito) reduzida do Padrão IEEE 754.

   Nesta representação, há seis intervalos (expoentes 001..110) com 16 pontos em cada intervalo (0000..1111). O intervalo com expoente 111 é usado para representar infinito e NaN (*not a number*). O intervalo com expoente 000 é dito denormalizado. O deslocamento do expoente é 3.

s	exp	fração
1	1 0 0	1 0 0 1

   As figuras 3.7 e 3.8 mostram a representação dos números reais em 8 bits. Note que as figuras mostram somente os Reais positivos, e que o intervalo com expoente 000 é denormalizado.

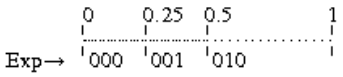


Figura 3.7: representação no intervalo $(0, 1)$

   (a) Preencha a tabela abaixo com os números representáveis em PF-8. Evidentemente, aqui não cabem todas as 256 possibilidades, e por isso é necessário completar a tabela em outra folha. Nesta tabela insira somente os casos especiais e os números decimais que são potências de dois.

   O campo 'expoente' deve conter o expoente deslocado (com o *bias* de 3). As 'magnitudes' são os números representados, o '*gap*' é o intervalo não-representável (vazio) entre dois números vizinhos na representação. Não esqueça das representações para $\pm\infty$ e NaN.

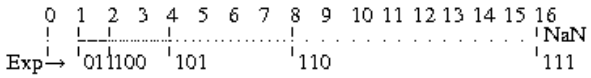


Figura 3.8: representação no intervalo $[1, 16)$

sinal	exp.	fração	Repr. Dec.	gap	comentário
0/1	000	0000	zero		caso especial

(b) Quais são os piores erros de arredondamento quando se faz cálculos nas faixas $[-8,8]$ e $[-1,1]$?

(c) Prove que as propriedades aritméticas abaixo são válidas, ou dê um contra-exemplo. $\oplus$ e $\otimes$ são a adição e o produto de números representados em PF-8. Evidentemente, os casos de overflow e underflow não podem ser usados como contra-exemplos. *With thanks to Jeff Sanders!*
a) monotonicidade c.r.a soma: $x \leq y \Rightarrow x \oplus z \leq y \oplus z$
b) associatividade c.r.a multiplicação: $x \otimes (y \otimes z) = (x \otimes y) \otimes z$
c) distributividade: $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$

(d) Represente as potências de 2 entre 4 e 1024 no formato `float` IEEE 754.

Capítulo 4

Representação de caracteres

A última representação de símbolos do universo que vivemos para computador abordada neste texto é a de representação de caracteres. Uma ótima referência para este tema é [Spo03].

O problema aqui é como representar símbolos como letras e dígitos para que a interação dos seres humanos com o computador seja mais natural.

Assim como os demais problemas de representação citados anteriormente, este problema teve várias propostas para resolução. A idéia básica é fazer com que cada caractere seja representado por um padrão binário diferente. É natural fazer uma escolha onde o padrão binário tenha sempre o mesmo número de bits, por exemplo oito bits. Se este padrão for adotado, há um total de $2^8 = 256$ representações diferentes. Com isso, é possível representar 256 símbolos diferentes.

A primeira "tabela de conversão" conhecida era chamada de EBCDIC, criada pela IBM na década de 60, porém atualmente ela só tem valor histórico.

Uma outra tabela de conversão, chamada tabela ASCII, foi proposta em 1961 e se tornou praticamente um padrão mundial. Esta tabela é apresentada na seção 4.1. A tabela ASCII é perfeita para a língua inglesa, mas não contém símbolos para, por exemplo os caracteres acentuados das línguas latinas, caracteres russos, chineses, entre outros. O Unicode (seção 4.2) surgiu para contemplar estes caracteres. Cada caractere em Unicode é representado em 16 bits, e uma versão mais "econômica" de tamanho variável (oito e dezesseis bits) surgiu a partir do Unicode, chamada UTF-8 que será analisada na seção 4.3.

4.1 ASCII

A tabela ASCII completa é apresentada no apêndice A. A idéia básica é que quando deseja-se imprimir um caractere, envia-se para o dispositivo de impressão o código do caractere a ser impresso. O dispositivo de impressão então procura este código na tabela interna dele, e coloca o caractere correspondente para visualização. É evidente que a tabela de quem envia o código e a tabela de quem recebe o código para impressão devem ser iguais.

Como exemplo de funcionamento, suponha que um computador usa esta tabela para imprimir caracteres no vídeo. Então, para imprimir o string "ERRO", o programa deve enviar a seguinte palavra $(0100\ 0101\ 0101\ 0010\ 0101\ 0010\ 0100\ 1111)_2$ ou $(4552524F)_{16}$ para que a mensagem seja impressa. Uma outra forma de ver o uso da tabela é quando salva em arquivo um texto criado em um editor de textos. Uma análise do arquivo em hexadecimal mostra os caracteres e códigos hexadecimais associados a cada símbolo.

Uma forma de fazer isso é através do programa `hexdump`. Abaixo é apresentado um trecho do parágrafo anterior salvo em um arquivo texto e impresso através deste programa.

```
> hexdump ArqHex.txt -C
00000000  71 75 61 6e 64 6f 20 73  61 6c 76 61 20 65 6d 20  |quando salva em |
00000010  61 72 71 75 69 76 6f 20  75 6d 20 74 65 78 74 6f  |arquivo um texto|
00000020  20 63 72 69 61 64 6f 20  65 6d 20 75 6d 20 65 64  | criado em um ed|
00000030  69 74 6f 72 20 64 65 20  74 65 78 74 6f 73 2e 20  |itor de textos. |
00000040  55 6d 61 0a 61 6e e1 6c  69 73 65 20 64 6f 20 61  |Uma.análise do a|
00000050  72 71 75 69 76 6f 20 65  6d 20 68 65 78 61 64 65  |rquivo em hexade|
00000060  63 69 6d 61 6c 20 6d 6f  73 74 72 61 20 6f 73 20  |cimal mostra os |
00000070  63 61 72 61 63 74 65 72  65 73 20 65 20 63 f3 64  |caracteres e cód|
00000080  69 67 6f 73 0a 68 65 78  61 64 65 63 69 6d 61 69  |igos.hexadecimai|
00000090  73 20 61 73 73 6f 63 69  61 64 6f 73 20 61 20 63  |s associados a c|
000000a0  61 64 61 20 73 ed 6d 62  6f 6c 6f 2e 0a           |ada símbolo..|
000000ad
```

A primeira coluna é a indicação posicional dos caracteres que serão impressos naquela linha. Assim, o caractere $(71)_{16} = (q)_{ASCII}$ é o caractere número zero. O caractere $(75)_{16}$ é o caractere um e assim por diante. A primeira linha tem dezesseis caracteres, por isso a primeira coluna da segunda linha contém $(00000010)_{16}$, que indica que o caractere $(61)_{16} = (a)_{ASCII}$ é o décimo sexto caractere ($(10)_{16} = (16)_{10}$).

A última coluna contém os caracteres gravados enquanto que a segunda coluna contém os códigos hexadecimais da tabela ASCII correspondentes. Alguns códigos especiais podem ser observados.

Os códigos não imprimíveis são representados com um ponto. Destes, há um que requer alguns comentários, o código $(0A)_{16}$, que corresponde a LF na tabela ASCII (Line Feed). Este símbolo, em linux, corresponde a duas ações: (1) retornar o cursor (ou carro de impressora) para a primeira coluna e (2) pular para a próxima linha. Em Windows, são necessários dois caracteres de controle para a mesma coisa: LF e CR (Carriage Return). Sendo assim, se alguém copiar um arquivo gravado em Windows para linux, haverá alguma incompatibilidade. Esta incompatibilidade pode ser eliminada com os comandos `dos2unix` e `unix2dos`.

É importante observar o que ocorre quando um número é impresso. Por exemplo para imprimir "12,43", são enviados os seguintes símbolos: $(3231342C0A33)_{16}$. Em outras palavras, para que variável numérica seja impressa na tela, é necessário convertê-la primeiro para um conjunto de caracteres e este conjunto de caracteres é que é enviado para a impressão.

De forma análoga, quando um símbolo é digitado no teclado, o hexadecimal ASCII é enviado para o computador, que decodifica qual é o símbolo digitado (de acordo com a configuração de teclado que foi indicada). Em seguida, analisa-se qual o caractere que deverá ser impresso para então imprimí-lo. Observe que para que isso funcione de

maneira adequada, é necessário indicar qual é o formato do teclado para que os símbolos sejam corretamente interpretados e também que seja conhecida qual o símbolo correspondente na tabela ASCII para a impressão.

Outra observação importante é que a tabela ASCII não representa símbolos latinos (símbolos acentuados) e outros símbolos que não fazem parte da língua inglesa. São vários grupos de línguas que não foram contemplados na tabela ASCII padrão.

Como há 128 códigos hexadecimais que não foram utilizados originalmente (entre $(80)_{16}$ e $(FF)_{16}$), cada grupo propôs um padrão para representação de seus símbolos naquele espaço. O problema é que todos propuseram isso de forma independente.

Somente quando a ISO[1] propôs um padrão para o uso daqueles 128 códigos é que a polêmica começou a se dissipar. O padrão chama-se `ISO 8859` (ou `ISO/IEC 8859`), que é contém um total de 16 especificações, das quais destacamos:

ISO 8859-1 (ISO 8859 parte 1) também conhecido como `latin-1`, é o mais usado. Contém padrões para línguas do oeste europeu, como o nórdico, holandês (parcial), alemão, francês, irlandês, espanhol, português entre outras.

ISO 8859-2 (ISO 8859 parte 2) também conhecido como `latin-2`. Línguas da Europa Central: polonês, tcheco, eslovaco, eslovênio, sérvio, húngaro entre outros.

ISO 8859-3 (ISO 8859 parte 3) também conhecido como `latin-3`. Línguas do sul da Europa: Turco, Maltês, etc..

Porém, se existem várias formas diferentes de utilizar os últimos 128 caracteres da tabela ASCII, a pergunta natural é "Como o sistema sabe qual usar?".

A resposta depende da aplicação que se usa. Por exemplo, em linux, é possível consultar qual a codificação que se está usando através da variável de ambiente `LANG`. Por exemplo, o comando

```
> echo $LANG
en_US.ISO-8859-1
```

indica que está sendo usada a tabela ISO-8859-1.

Em mensagens eletrônicas (e-mails), a tabela de codificação usada para compor a mensagem está normalmente indicada nas primeiras linhas de cada mensagem. Uma forma de fazer isso é que a mensagem contém as seguintes linhas no cabeçalho.

```
Content-Type: text/plain; charset=ISO-8859-1
```

O divertido é quando esta informação não está indicada e alguém tenta ler uma mensagem que foi escrita, digamos em hebreu, ou chinês, ou árabe, russo, etc.. Como exercício, procure as tabelas na internet (por exemplo na wikipedia) e compare os símbolos que correspondem ao código $(C8)_{16}$ em cada uma das 16 tabelas e copie abaixo:

[1] International Organization for Standardization

8859-1	8859-2	8859-3	8859-4	8859-5	8859-6	8859-7	8859-8
8859-9	8859-10	8859-11	8859-12	8859-13	8859-14	8859-15	8859-16

Se o programa que vai mostrar os caracteres na tela não sabe qual tabela usar, ele pode indicar o problema para que o usuário o resolva, ou simplesmente usar a tabela especificada como padrão pelo usuário e torcer para acertar (o que nem sempre ocorre).

Outra é mostrar um monte de pontos de interrogação como um aviso ao usuário do tipo "não sei qual tabela usar".

O ideal é que houvesse uma única tabela para representar todos os símbolos de todas as línguas. Duas tentativas neste sentido merecem destaque: o UNICODE (16 bits), e uma versão de tamanho variável do Unicode chamada UTF-8[2]. O UTF-8 usa oito bits para representar os primeiros 127 caracteres da tabela ASCII e para representar os demais caracteres (os acentuados, por exemplo), usa dezesseis bits Estes dois formatos serão apresentados na seqüência.

4.2 Unicode

O Unicode é o resultado de um esforço para traduzir todo e qualquer sistema de escrita existente no planeta em um único conjunto de símbolos.

O Consórcio Unicode define o Unicode da seguinte forma[3]:

```
O que é Unicode?

Unicode fornece um número único para cada caracter,
não importa a plataforma,
não importa o programa,
não importa a língua.
```

Como ele usa 16 bits, existe um total de $2^{16} = 65536$ representações diferentes potenciais. Na prática, este número é menor porque algumas faixas são reservadas como será visto a seguir. Porém, a grande quantidade de representações causa efeitos colaterais. Existem caracteres unicode para línguas fictícias, como Klingon, Élfico, entre outros. Afinal, havia espaço...

O primeiro aspecto a ressaltar é que assim como os demais formatos apresentados, Unicode não lida com formatações diferentes de um mesmo caractere, ou seja, **A**, *A*, A, são todos o mesmo símbolo. Porém, "a" e "A" são diferentes.

Assim como ocorre em ASCII, cada caractere tem um código associado a ele. Como são muitos códigos, e muitos caracteres, este texto tratará somente dos conceitos envolvidos com a codificação de caracteres em Unicode, e sua representação interna.

[2] UCS/Unicode Transformation Format

[3] http://www.unicode.org/standard/translations/portuguese.html

Serão usados alguns poucos caracteres como exemplo, e os interessados para descobrir os demais devem procurar a página do consórcio unicode.

Assim, a letra "A" está associado a $(0041)_{16}$, que será representado U+0041 daqui para frente. O simples fato de serem necessários dois bytes para representar um caractere já causa problemas, e isso teve de ser considerado na elaboração do padrão.

Em algumas máquinas, os bytes são armazendados invertidos, ou seja, $(0041)_{16}$ é armazenado $(4100)_{16}$[4]

Evidentemente isso é um problema, mas só quando a informação é transferida entre máquinas que armazenam bytes de forma diferente. Porém este é um caso comum, e por isso teve de ser tratado.

A solução foi usar o BOM (*Byte Order Mark*). Cada cadeia de caracteres Unicode é precedida pelos caracteres U+FEFF ou U+FFFE. O primeiro indica que a cadeia está indicada em "big endian" enquanto que a segunda implica em "little endian".

Porém é importante destacar que dependendo do protocolo Unicode adotado, o uso do BOM pode ser proibido ou obrigatório. Isso parece estranho, pois aparentemente o mais correto seria obrigar o uso em todas as cadeias de caracteres, mas foram detectados alguns casos em que isso é desnecessário e que a obrigação de uso do BOM causaria gasto inútil de espaço físico em memória ou disco. Como exemplo, considere o caso de uma cadeia de caracteres contida em um banco de dados que só será acessado pela mesma máquina que o armazenou. Aqui não é necessário o BOM.

4.3 UTF-8

Considere a codificação da cadeia "HELLO" [Spo03], que é representada em unicode da seguinte forma:

U+0048 U+0065 U+006C U+006C U+006F

Isso significa que a cadeia acima seria representada em memória da seguinte forma: $(00\ 48\ 00\ 65\ 00\ 6C\ 00\ 6C\ 00\ 6F)_{16}$

Bem, se o consórcio se preocupou em economizar espaço não tornando o BOM obrigatório, é evidente que eles se preocuparam com a quantidade de zeros que apareceram acima.

Se forem retirados os zeros do exemplo acima, teremos: $(48\ 65\ 6C\ 6C\ 6F)_{16}$. Consultando a tabela ASCII, veremos que $(48\ 65\ 6C\ 6C\ 6F)_{16} = HELLO_{ASCII}$. Isso não é por acaso.

O consórcio Unicode definiu mais de um padrão. O padrão que usa 16 bits para cada caractere é chamado de UTF-16 (Unicode Transformation Format - 16 bits). Existe também um padrão UTF-7, UTF-8, UTF-32 entre outros.

[4]Considere o caso dos bytes 0041. As máquinas que armazenam a informação na ordem em que o byte mais significativo fica no fim (ou seja, 4100) são chamados "big endian" enquanto que que terminam com o byte menos significativo são chamados de "little endian". Esse nome foi adotado por causa do livro "As viagens de Gulliver", onde havia uma guerra entre dois povos que discutiam se o local correto de se abrir um ovo cozido era no lado mais fino (que eles chamavam de little endian) ou se no lado mais largo (que eles chamavam de big endian).

Porém, merece destaque o padrão que usa 8 bits, chamado UTF-8, que tem a grande vantagem de ser igual ao que foi descrito na tabela ASCII para os usuários da língua inglesa (os primeiros 128 bits - [$(00)_{16}$ .. $(7F)_{16}$]).

O consórcio definiu que os outros 128 bits, ou seja a faixa [$(80)_{16}$ .. $(FF)_{16}$] correspondem ao que conhecemos como caracteres latinos, mais especificamente os mesmos caracteres estipulados no padrão ISO 8859-1 (latin-1).

Os demais padrões ISO 8859-2, ISO 8859-3, ... , ISO 8859-16 foram deslocados para outras faixas e só podem ser acessados com UTF-16. Como exemplo, as letras gregas estão na faixa [$(0370)_{16}$ .. $(03FF)_{16}$]. Todas as faixas e conjuntos de caracteres podem ser encontrados em na internet[5] ou nos livros publicados pelo consórcio Unicode.

É importante mencionar que o consórcio aumentou o número de caracteres latinos disponíveis, que podem ser encontrados também em outras faixas se for utilizar UTF-16.

Porém, um texto condificado como ASCII e um texto condificado como UTF-8 não são necessariamente compatíveis. Se o texto contiver somente caracteres na faixa [$(00)_{16}$ .. $(7F)_{16}$], serão compatíveis, porém se contiverem caracteres na faixa [$(80)_{16}$ .. $(FF)_{16}$] não serão. Os símbolos da última faixa serão representados por dois bytes, exatamente igual ao que Unicode codificaria.

A tabela 4.1 mostra a diferença de codificação entre alguns caracteres “latinos”.

Decimal	Caractere	ASCII	UTF-8
195	Ã	$(C3)_{16}$	$(C383)_{16}$
199	Ç	$(C7)_{16}$	$(C387)_{16}$
212	Ô	$(D4)_{16}$	$(C394)_{16}$
227	ã	$(E3)_{16}$	$(C3A3)_{16}$

Tabela 4.1: Comparaçao ASCII - UTF-8

Observe que o caractere $(C3)_{16}$ é um símbolo de controle, e não corresponde a nenhum caractere.

4.3.1 Conversões

Por vezes é interessante converter um arquivo que está no formato UTF-8 para ISO-8859-1 e vice-versa.

Em linux, usa-se o comando “iconv” como descrito abaixo:

```
> iconv --from-code=UTF-8 --to-code=ISO-8859-1 ArqUTF-8 > ArqISO

> iconv --from-code=ISO-8859-1 --to-code=UTF-8 ArqISO > ArqUTF-8
```

O “iconv” faz muito mais do que só esta conversão. Como exemplo, digite “iconv –list”.

[5] `http://www.unicode.org/charts/symbols.html`

```
> iconv --list
The following list contain all the coded character sets known.  This does
not necessarily mean that all combinations of these names can be used for
the FROM and TO command line parameters.  One coded character set can be
listed with several different names (aliases).

  437, 500, 500V1, 850, 851, 852, 855, 856, 857, 860, 861, 862, 863, 864, 865,
  866, 866NAV, 869, 874, 904, 1026, 1046, 1047, 8859_1, 8859_2, 8859_3, 8859_4,
  8859_5, 8859_6, 8859_7, 8859_8, 8859_9, 10646-1:1993, 10646-1:1993/UCS4,
  ANSI_X3.4-1968, ANSI_X3.4-1986, ANSI_X3.4, ANSI_X3.110-1983, ANSI_X3.110,
  ARABIC, ARABIC7, ARMSCII-8, ASCII, ASMO-708, ASMO_449, BALTIC, BIG-5,
  BIG-FIVE, BIG5-HKSCS, BIG5, BIG5HKSCS, BIGFIVE, BRF, BS_4730, CA, CN-BIG5,
  CN-GB, CN, CP-AR, CP-GR, CP-HU, CP037, CP038, CP273, CP274, CP275, CP278,
  CP280, CP281, CP282, CP284, CP285, CP290, CP297, CP367, CP420, CP423, CP424,
  CP437, CP500, CP737, CP775, CP803, CP813, CP819, CP850, CP851, CP852, CP855,
  CP856, CP857, CP860, CP861, CP862, CP863, CP864, CP865, CP866, CP866NAV,
  CP868, CP869, CP870, CP871, CP874, CP875, CP880, CP891, CP901, CP902, CP903,
  CP904, CP905, CP912, CP915, CP916, CP918, CP920, CP921, CP922, CP930, CP932,
  CP933, CP935, CP936, CP937, CP939, CP949, CP950, CP1004, CP1008, CP1025,
  CP1026, CP1046, CP1047, CP1070, CP1079, CP1081, CP1084, CP1089, CP1097,
  CP1112, CP1122, CP1123, CP1124, CP1125, CP1129, CP1130, CP1132, CP1133,
  CP1137, CP1140, CP1141, CP1142, CP1143, CP1144, CP1145, CP1146, CP1147,
  CP1148, CP1149, CP1153, CP1154, CP1155, CP1156, CP1157, CP1158, CP1160,
  CP1161, CP1162, CP1163, CP1164, CP1166, CP1167, CP1250, CP1251, CP1252,
  CP1253, CP1254, CP1255, CP1256, CP1257, CP1258, CP1282, CP1361, CP1364,
  CP1371, CP1388, CP1390, CP1399, CP4517, CP4899, CP4909, CP4971, CP5347,
  CP9030, CP9066, CP9448, CP10007, CP12712, CP16804, CPIBM861, CSA7-1, CSA7-2,
  CSASCII, CSA_T500-1983, CSA_T500, CSA_Z243.4-1985-1, CSA_Z243.4-1985-2,
  CSA_Z243.419851, CSA_Z243.419852, CSDECMCS, CSEBCDICATDE, CSEBCDICATDEA,
  CSEBCDICCAFR, CSEBCDICDKNO, CSEBCDICDKNOA, CSEBCDICES, CSEBCDICESA,
  CSEBCDICESS, CSEBCDICFISE, CSEBCDICFISEA, CSEBCDICFR, CSEBCDICIT, CSEBCDICPT,
  CSEBCDICUK, CSEBCDICUS, CSEUCKR, CSEUCPKDFMTJAPANESE, CSGB2312, CSHPROMAN8,
  CSIBM037, CSIBM038, CSIBM273, CSIBM274, CSIBM275, CSIBM277, CSIBM278,
  CSIBM280, CSIBM281, CSIBM284, CSIBM285, CSIBM290, CSIBM297, CSIBM420,
  CSIBM423, CSIBM424, CSIBM500, CSIBM803, CSIBM851, CSIBM855, CSIBM856,
  CSIBM857, CSIBM860, CSIBM863, CSIBM864, CSIBM865, CSIBM866, CSIBM868,
  CSIBM869, CSIBM870, CSIBM871, CSIBM880, CSIBM891, CSIBM901, CSIBM902,
  CSIBM903, CSIBM904, CSIBM905, CSIBM918, CSIBM921, CSIBM922, CSIBM930,
  CSIBM932, CSIBM933, CSIBM935, CSIBM937, CSIBM939, CSIBM943, CSIBM1008,
  CSIBM1025, CSIBM1026, CSIBM1097, CSIBM1112, CSIBM1122, CSIBM1123, CSIBM1124,
  CSIBM1129, CSIBM1130, CSIBM1132, CSIBM1133, CSIBM1137, CSIBM1140, CSIBM1141,
  CSIBM1142, CSIBM1143, CSIBM1144, CSIBM1145, CSIBM1146, CSIBM1147, CSIBM1148,
  CSIBM1149, CSIBM1153, CSIBM1154, CSIBM1155, CSIBM1156, CSIBM1157, CSIBM1158,
  CSIBM1160, CSIBM1161, CSIBM1163, CSIBM1164, CSIBM1166, CSIBM1167, CSIBM1364,
  CSIBM1371, CSIBM1388, CSIBM1390, CSIBM1399, CSIBM4517, CSIBM4899, CSIBM4909,
  CSIBM4971, CSIBM5347, CSIBM9030, CSIBM9066, CSIBM9448, CSIBM12712,
  CSIBM16804, CSIBM11621162, CSISO4UNITEDKINGDOM, CSISO10SWEDISH,
  CSISO11SWEDISHFORNAMES, CSISO14JISC6220RO, CSISO15ITALIAN, CSISO16PORTUGESE,
  CSISO17SPANISH, CSISO18GREEK7OLD, CSISO19LATINGREEK, CSISO21GERMAN,
  CSISO25FRENCH, CSISO27LATINGREEK1, CSISO49INIS, CSISO50INIS8,
  CSISO51INISCYRILLIC, CSISO58GB1988, CSISO60DANISHNORWEGIAN,
  CSISO60NORWEGIAN1, CSISO61NORWEGIAN2, CSISO69FRENCH, CSISO84PORTUGUESE2,
  CSISO85SPANISH2, CSISO86HUNGARIAN, CSISO88GREEK7, CSISO89ASMO449, CSISO90,
  CSISO92JISC62991984B, CSISO99NAPLPS, CSISO103T618BIT, CSISO111ECMACYRILLIC,
  CSISO121CANADIAN1, CSISO122CANADIAN2, CSISO139CSN369103, CSISO141JUSIB1002,
  CSISO143IECP271, CSISO150, CSISO150GREEKCCITT, CSISO151CUBA,
  CSISO153GOST1976874, CSISO646DANISH, CSISO2022CN, CSISO2022JP, CSISO2022JP2,
```

CSISO2022KR, CSISO2033, CSISO5427CYRILLIC, CSISO5427CYRILLIC1981,
CSISO5428GREEK, CSISO10367BOX, CSISOLATIN1, CSISOLATIN2, CSISOLATIN3,
CSISOLATIN4, CSISOLATIN5, CSISOLATIN6, CSISOLATINARABIC, CSISOLATINCYRILLIC,
CSISOLATINGREEK, CSISOLATINHEBREW, CSKOI8R, CSKSC5636, CSMACINTOSH,
CSNATSDANO, CSNATSSEFI, CSN_369103, CSPC8CODEPAGE437, CSPC775BALTIC,
CSPC850MULTILINGUAL, CSPC862LATINHEBREW, CSPCP852, CSSHIFTJIS, CSUCS4,
CSUNICODE, CSWINDOWS31J, CUBA, CWI-2, CWI, CYRILLIC, DE, DEC-MCS, DEC,
DECMCS, DIN_66003, DK, DS2089, DS_2089, E13B, EBCDIC-AT-DE-A, EBCDIC-AT-DE,
EBCDIC-BE, EBCDIC-BR, EBCDIC-CA-FR, EBCDIC-CP-AR1, EBCDIC-CP-AR2,
EBCDIC-CP-BE, EBCDIC-CP-CA, EBCDIC-CP-CH, EBCDIC-CP-DK, EBCDIC-CP-ES,
EBCDIC-CP-FI, EBCDIC-CP-FR, EBCDIC-CP-GB, EBCDIC-CP-GR, EBCDIC-CP-HE,
EBCDIC-CP-IS, EBCDIC-CP-IT, EBCDIC-CP-NL, EBCDIC-CP-NO, EBCDIC-CP-ROECE,
EBCDIC-CP-SE, EBCDIC-CP-TR, EBCDIC-CP-US, EBCDIC-CP-WT, EBCDIC-CP-YU,
EBCDIC-CYRILLIC, EBCDIC-DK-NO-A, EBCDIC-DK-NO, EBCDIC-ES-A, EBCDIC-ES-S,
EBCDIC-ES, EBCDIC-FI-SE-A, EBCDIC-FI-SE, EBCDIC-FR, EBCDIC-GREEK, EBCDIC-INT,
EBCDIC-INT1, EBCDIC-IS-FRISS, EBCDIC-IT, EBCDIC-JP-E, EBCDIC-JP-KANA,
EBCDIC-PT, EBCDIC-UK, EBCDIC-US, EBCDICATDE, EBCDICATDEA, EBCDICCAFR,
EBCDICDKNO, EBCDICDKNOA, EBCDICES, EBCDICESA, EBCDICESS, EBCDICFISE,
EBCDICFISEA, EBCDICFR, EBCDICISFRISS, EBCDICIT, EBCDICPT, EBCDICUK, EBCDICUS,
ECMA-114, ECMA-118, ECMA-128, ECMA-CYRILLIC, ECMACYRILLIC, ELOT_928, ES, ES2,
EUC-CN, EUC-JISX0213, EUC-JP-MS, EUC-JP, EUC-KR, EUC-TW, EUCCN, EUCJP-MS,
EUCJP-OPEN, EUCJP-WIN, EUCJP, EUCKR, EUCTW, FI, FR, GB, GB2312, GB13000,
GB18030, GBK, GB_1988-80, GB_198880, GEORGIAN-ACADEMY, GEORGIAN-PS,
GOST_19768-74, GOST_19768, GOST_1976874, GREEK-CCITT, GREEK, GREEK7-OLD,
GREEK7, GREEK7OLD, GREEK8, GREEKCCITT, HEBREW, HP-GREEK8, HP-ROMAN8,
HP-ROMAN9, HP-THAI8, HP-TURKISH8, HPGREEK8, HPROMAN8, HPROMAN9, HPTHAI8,
HPTURKISH8, HU, IBM-803, IBM-856, IBM-901, IBM-902, IBM-921, IBM-922,
IBM-930, IBM-932, IBM-933, IBM-935, IBM-937, IBM-939, IBM-943, IBM-1008,
IBM-1025, IBM-1046, IBM-1047, IBM-1097, IBM-1112, IBM-1122, IBM-1123,
IBM-1124, IBM-1129, IBM-1130, IBM-1132, IBM-1133, IBM-1137, IBM-1140,
IBM-1141, IBM-1142, IBM-1143, IBM-1144, IBM-1145, IBM-1146, IBM-1147,
IBM-1148, IBM-1149, IBM-1153, IBM-1154, IBM-1155, IBM-1156, IBM-1157,
IBM-1158, IBM-1160, IBM-1161, IBM-1162, IBM-1163, IBM-1164, IBM-1166,
IBM-1167, IBM-1364, IBM-1371, IBM-1388, IBM-1390, IBM-1399, IBM-4517,
IBM-4899, IBM-4909, IBM-4971, IBM-5347, IBM-9030, IBM-9066, IBM-9448,
IBM-12712, IBM-16804, IBM037, IBM038, IBM256, IBM273, IBM274, IBM275, IBM277,
IBM278, IBM280, IBM281, IBM284, IBM285, IBM290, IBM297, IBM367, IBM420,
IBM423, IBM424, IBM437, IBM500, IBM775, IBM803, IBM813, IBM819, IBM848,
IBM850, IBM851, IBM852, IBM855, IBM856, IBM857, IBM860, IBM861, IBM862,
IBM863, IBM864, IBM865, IBM866, IBM866NAV, IBM868, IBM869, IBM870, IBM871,
IBM874, IBM875, IBM880, IBM891, IBM901, IBM902, IBM903, IBM904, IBM905,
IBM912, IBM915, IBM916, IBM918, IBM920, IBM921, IBM922, IBM930, IBM932,
IBM933, IBM935, IBM937, IBM939, IBM943, IBM1004, IBM1008, IBM1025, IBM1026,
IBM1046, IBM1047, IBM1089, IBM1097, IBM1112, IBM1122, IBM1123, IBM1124,
IBM1129, IBM1130, IBM1132, IBM1133, IBM1137, IBM1140, IBM1141, IBM1142,
IBM1143, IBM1144, IBM1145, IBM1146, IBM1147, IBM1148, IBM1149, IBM1153,
IBM1154, IBM1155, IBM1156, IBM1157, IBM1158, IBM1160, IBM1161, IBM1162,
IBM1163, IBM1164, IBM1166, IBM1167, IBM1364, IBM1371, IBM1388, IBM1390,
IBM1399, IBM4517, IBM4899, IBM4909, IBM4971, IBM5347, IBM9030, IBM9066,
IBM9448, IBM12712, IBM16804, IEC_P27-1, IEC_P271, INIS-8, INIS-CYRILLIC,
INIS, INIS8, INISCYRILLIC, ISIRI-3342, ISIRI3342, ISO-2022-CN-EXT,
ISO-2022-CN, ISO-2022-JP-2, ISO-2022-JP-3, ISO-2022-JP, ISO-2022-KR,
ISO-8859-1, ISO-8859-2, ISO-8859-3, ISO-8859-4, ISO-8859-5, ISO-8859-6,
ISO-8859-7, ISO-8859-8, ISO-8859-9, ISO-8859-9E, ISO-8859-10, ISO-8859-11,
ISO-8859-13, ISO-8859-14, ISO-8859-15, ISO-8859-16, ISO-10646,
ISO-10646/UCS2, ISO-10646/UCS4, ISO-10646/UTF-8, ISO-10646/UTF8, ISO-CELTIC,
ISO-IR-4, ISO-IR-6, ISO-IR-8-1, ISO-IR-9-1, ISO-IR-10, ISO-IR-11, ISO-IR-14,

```
ISO-IR-15, ISO-IR-16, ISO-IR-17, ISO-IR-18, ISO-IR-19, ISO-IR-21, ISO-IR-25,
ISO-IR-27, ISO-IR-37, ISO-IR-49, ISO-IR-50, ISO-IR-51, ISO-IR-54, ISO-IR-55,
ISO-IR-57, ISO-IR-60, ISO-IR-61, ISO-IR-69, ISO-IR-84, ISO-IR-85, ISO-IR-86,
ISO-IR-88, ISO-IR-89, ISO-IR-90, ISO-IR-92, ISO-IR-98, ISO-IR-99, ISO-IR-100,
ISO-IR-101, ISO-IR-103, ISO-IR-109, ISO-IR-110, ISO-IR-111, ISO-IR-121,
ISO-IR-122, ISO-IR-126, ISO-IR-127, ISO-IR-138, ISO-IR-139, ISO-IR-141,
ISO-IR-143, ISO-IR-144, ISO-IR-148, ISO-IR-150, ISO-IR-151, ISO-IR-153,
ISO-IR-155, ISO-IR-156, ISO-IR-157, ISO-IR-166, ISO-IR-179, ISO-IR-193,
ISO-IR-197, ISO-IR-199, ISO-IR-203, ISO-IR-209, ISO-IR-226, ISO/TR_11548-1,
ISO646-CA, ISO646-CA2, ISO646-CN, ISO646-CU, ISO646-DE, ISO646-DK, ISO646-ES,
ISO646-ES2, ISO646-FI, ISO646-FR, ISO646-FR1, ISO646-GB, ISO646-HU,
ISO646-IT, ISO646-JP-OCR-B, ISO646-JP, ISO646-KR, ISO646-NO, ISO646-NO2,
ISO646-PT, ISO646-PT2, ISO646-SE, ISO646-SE2, ISO646-US, ISO646-YU,
ISO2022CN, ISO2022CNEXT, ISO2022JP, ISO2022JP2, ISO2022KR, ISO6937,
ISO8859-1, ISO8859-2, ISO8859-3, ISO8859-4, ISO8859-5, ISO8859-6, ISO8859-7,
ISO8859-8, ISO8859-9, ISO8859-9E, ISO8859-10, ISO8859-11, ISO8859-13,
ISO8859-14, ISO8859-15, ISO8859-16, ISO11548-1, ISO88591, ISO88592, ISO88593,
ISO88594, ISO88595, ISO88596, ISO88597, ISO88598, ISO88599, ISO88599E,
ISO885910, ISO885911, ISO885913, ISO885914, ISO885915, ISO885916,
ISO_646.IRV:1991, ISO_2033-1983, ISO_2033, ISO_5427-EXT, ISO_5427,
ISO_5427:1981, ISO_5427EXT, ISO_5428, ISO_5428:1980, ISO_6937-2,
ISO_6937-2:1983, ISO_6937, ISO_6937:1992, ISO_8859-1, ISO_8859-1:1987,
ISO_8859-2, ISO_8859-2:1987, ISO_8859-3, ISO_8859-3:1988, ISO_8859-4,
ISO_8859-4:1988, ISO_8859-5, ISO_8859-5:1988, ISO_8859-6, ISO_8859-6:1987,
ISO_8859-7, ISO_8859-7:1987, ISO_8859-7:2003, ISO_8859-8, ISO_8859-8:1988,
ISO_8859-9, ISO_8859-9:1989, ISO_8859-9E, ISO_8859-10, ISO_8859-10:1992,
ISO_8859-14, ISO_8859-14:1998, ISO_8859-15, ISO_8859-15:1998, ISO_8859-16,
ISO_8859-16:2001, ISO_9036, ISO_10367-BOX, ISO_10367BOX, ISO_11548-1,
ISO_69372, IT, JIS_C6220-1969-RO, JIS_C6229-1984-B, JIS_C62201969RO,
JIS_C62291984B, JOHAB, JP-OCR-B, JP, JS, JUS_I.B1.002, KOI-7, KOI-8, KOI8-R,
KOI8-RU, KOI8-T, KOI8-U, KOI8, KOI8R, KOI8U, KSC5636, L1, L2, L3, L4, L5, L6,
L7, L8, L10, LATIN-9, LATIN-GREEK-1, LATIN-GREEK, LATIN1, LATIN2, LATIN3,
LATIN4, LATIN5, LATIN6, LATIN7, LATIN8, LATIN9, LATIN10, LATINGREEK,
LATINGREEK1, MAC-CENTRALEUROPE, MAC-CYRILLIC, MAC-IS, MAC-SAMI, MAC-UK, MAC,
MACCYRILLIC, MACINTOSH, MACIS, MACUK, MACUKRAINIAN, MIK, MS-ANSI, MS-ARAB,
MS-CYRL, MS-EE, MS-GREEK, MS-HEBR, MS-MAC-CYRILLIC, MS-TURK, MS932, MS936,
MSCP949, MSCP1361, MSMACCYRILLIC, MSZ_7795.3, MS_KANJI, NAPLPS, NATS-DANO,
NATS-SEFI, NATSDANO, NATSSEFI, NC_NC0010, NC_NC00-10, NC_NC00-10:81,
NF_Z_62-010, NF_Z_62-010_(1973), NF_Z_62-010_1973, NF_Z_62010,
NF_Z_62010_1973, NO, NO2, NS_4551-1, NS_4551-2, NS_45511, NS_45512,
OS2LATIN1, OSF00010001, OSF00010002, OSF00010003, OSF00010004, OSF00010005,
OSF00010006, OSF00010007, OSF00010008, OSF00010009, OSF0001000A, OSF00010020,
OSF00010100, OSF00010101, OSF00010102, OSF00010104, OSF00010105, OSF00010106,
OSF00030010, OSF0004000A, OSF0005000A, OSF05010001, OSF100201A4, OSF100201A8,
OSF100201B5, OSF100201F4, OSF100203B5, OSF1002011C, OSF1002011D, OSF1002035D,
OSF1002035E, OSF1002035F, OSF1002036B, OSF1002037B, OSF10010001, OSF10010004,
OSF10010006, OSF10020025, OSF10020111, OSF10020115, OSF10020116, OSF10020118,
OSF10020122, OSF10020129, OSF10020352, OSF10020354, OSF10020357, OSF10020359,
OSF10020360, OSF10020364, OSF10020365, OSF10020366, OSF10020367, OSF10020370,
OSF10020387, OSF10020388, OSF10020396, OSF10020402, OSF10020417, PT, PT2,
PT154, R8, R9, RK1048, ROMAN8, ROMAN9, RUSCII, SE, SE2, SEN_850200_B,
SEN_850200_C, SHIFT-JIS, SHIFT_JIS, SHIFT_JISX0213, SJIS-OPEN, SJIS-WIN,
SJIS, SS636127, STRK1048-2002, ST_SEV_358-88, T.61-8BIT, T.61, T.618BIT,
TCVN-5712, TCVN, TCVN5712-1, TCVN5712-1:1993, THAI8, TIS-620, TIS620-0,
TIS620.2529-1, TIS620.2533-0, TIS620, TS-5881, TSCII, TURKISH8, UCS-2,
UCS-2BE, UCS-2LE, UCS-4, UCS-4BE, UCS-4LE, UCS2, UCS4, UHC, UJIS, UK,
UNICODE, UNICODEBIG, UNICODELITTLE, US-ASCII, US, UTF-7, UTF-8, UTF-16,
```

```
UTF-16BE, UTF-16LE, UTF-32, UTF-32BE, UTF-32LE, UTF7, UTF8, UTF16, UTF16BE,
UTF16LE, UTF32, UTF32BE, UTF32LE, VISCII, WCHAR_T, WIN-SAMI-2, WINBALTRIM,
WINDOWS-31J, WINDOWS-874, WINDOWS-936, WINDOWS-1250, WINDOWS-1251,
WINDOWS-1252, WINDOWS-1253, WINDOWS-1254, WINDOWS-1255, WINDOWS-1256,
WINDOWS-1257, WINDOWS-1258, WINSAMI2, WS2, YU
```

Parte II

Linguagem Assembly

Esta parte do texto lida mais detalhadamente com as instruções que uma CPU executa.

Considere a figura 4.1. Ela mostra esquematicamente uma arquitetura adotada em praticamente todos os computadores comerciais atuais.

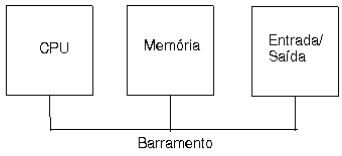


Figura 4.1: Arquitetura Esquemática

Este modelo é composto por quatro elementos:

CPU: a unidade de processamento de instruções. Toda e qualquer transformação nos dados ocorre neste elemento.

Memória: é a unidade de armazenamento de dados e de instruções. Cada "endereço" de memória contém dados ou instruções.

Entrada/Saída: Corresponde a todas as unidades de entrada e saída de dados, como disco, vídeo, teclado, mouse, etc..

Barramento: É o "meio de transporte" que leva informações de um elemento para outro. É composto por uma série de "fios" paralelos, por exemplo 32 fios em uma máquina de 32 bits. Cada fio ligado corresponde a 1, e desligado corresponde a 0.

Um programa em execução conterá suas instruções e seus dados na memória. A seqüência de passos para a execução de um programa é a seguinte:

1. A CPU requisita uma instrução que está na memória, e esta instrução é transportada da memória para a CPU.
2. Quando a instrução chega até a CPU, esta executa a instrução. Por exemplo, uma instrução de soma de dois números naturais chega pelo barramento, a CPU executa a soma entre dois números naturais.
3. O resultado da operação pode ser armazenado na memória novamente.
4. A CPU requisita a próxima instrução, e todo o processo recomeça.

Observe que a instrução que está na memória está em formato binário. Se esta instrução ocupar 32 bits, é possível transportá-la de uma única vez para a CPU. Se ocupar mais bits, por exemplo, 64 bits, são necessárias duas viagens. Quanto maior a instrução, mais demora para enviá-la para a CPU.

A CPU contém um pequeno número de entidades de armazenamento chamados registradores. O tempo de acesso a um registrador é **muito** menor do que o tempo de acesso à memória (afinal, está mais perto). Por esta razão máquinas com maior número de registradores são normalmente mais rápidas do que máquinas equivalentes com menor número de registradores.

Ao invés de adotar uma CPU verdadeira como objeto desta seção, será adotada uma CPU virtual, que tem a vantagem de ser mais simples, apesar de muito mais lenta. Esta CPU virtual é um simulador de uma CPU real, o MIPS. Esta CPU foi projetada por Hennessy e Patterson, e é detalhada em [DJ97].

O MIPS é um processador RISC (reduced instruction set computers), e contém um conjunto pequeno e regular de instruções. Este tipo de processador contrasta com os processadores CISC (complex instruction set computers), que contém maior número de instruções, e CPUs muito mais complicadas.

Os processadores RISC são normalmente mais rápidos, porém também são mais caros, motivo pelo qual é comum o uso processadores CISC como o intel 80x86.

O simulador do processador MIPS é chamado SPIM[6], e é facilmente encontrado para diversos sistemas operacionais.

Pelo custo (zero), e pela farta documentação, a linguagem assembly a ser estudada aqui é a linguagem definida para o MIPS/SPIM.

O capítulo 5 apresenta o simulador SPIM. O capítulo 6 apresenta o conjunto de instruções.

[6]Trocadilho sutil: SPIM = MIPS ao contrário

Capítulo 5

O simulador SPIM

O simulador pode ser obtido facilmente na internet para diversos sistemas operacionais. Apesar de, em princípio, não haver diferença de funcionamento entre eles, foi usada a versão linux para a elaboração deste texto.

O departamento de informática tem o SPIM instalado em todos os laboratórios. Existem dois programas que podem ser usados: o "xspim" (versão gráfica) e o "spim" versão texto.

Como primeira tarefa, abra um terminal e digite "spim". Deverá aparecer algo semelhante ao seguinte:

```
> spim
SPIM Version 7.3. of August 28, 2006
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/local/lib/exceptions.s
(spim)
```

O cursor aparecerá logo após o "(spim)" da última linha, aguardando ação do usuário.

O modelo de interação apresentado continua o exemplo. Após iniciar o simulador, o usuário digitou `load "teste.s"`, que faz o simulador ler o arquivo indicado. O comando "run" fez o programa "teste.s" executar e gerou o resultado "40".

Para sair do programa, o usuário digitou "exit".

```
> spim
SPIM Version 7.3. of August 28, 2006
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/local/lib/exceptions.s
(spim) load "teste.s"
(spim) run
40
(spim) exit
```

O programa spim é útil, porém normalmente é mais interessante usar o programa "xspim". Este programa é uma versão gráfica do spim. Quando o usuário digita `xspim`, aparece na tela algo semelhante com a figura 5.1.

Pode-se ver que existem quatro áreas distintas na figura. A parte superior indica os valores contidos em cada um dos registradores contidos na CPU. Entre os registradores, destacam-se `PC`, `EPC`, `R0(r0)`, `R1(at)`, etc..

Na parte central, os botões de interação do usuário. Através destes botões é possível carregar um programa assembly, executá-lo de uma vez, executá-lo passo a passo, etc..

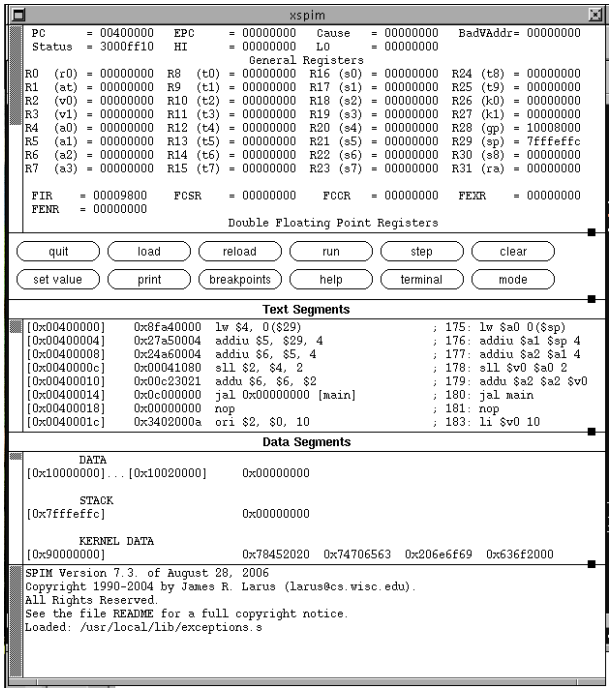


Figura 5.1: O programa xspim

A região anotada como `Text Segments` indica a instrução que está sendo executada. O termo *Text* neste contexto sempre indica região de instruções.

A instrução a ser executada pela CPU é indicada pelo registrador `PC`, que é igual a `0x00400000` (colocar `0x` na frente de um número indica que ele está em hexadecimal). A primeira linha desta região é:

```
[0x00400000]   0x8fa40000   lw $4, 0($29)   ; 175 lw $a0 0($sp)
```

A primeira coluna indica o endereço da memória. A segunda coluna indica o conteúdo daquele endereço de memória. Como esta região trata de instruções, o conteúdo do endereço `0x00400000` é uma instrução. Em SPIM, as instruções são regulares, e tem exatamente 32 ou 64 bits[1].

A instrução em questão, em hexadecimal é `0x8fa40000`, que em binário é `10001111101001000000000000000000`. Bem, nenhum dos dois é compreensível. Por isso, a terceira coluna indica o que esta instrução significa: `lw $4, 0($29)`, onde:

lw indica qual instrução a CPU deve executar. Neste caso, "load word". Esta instrução carrega o valor contido em um endereço de memória e o coloca em um registrador.

$4 é o primeiro operando da instrução. Neste caso, indica o registrador que é o destino da leitura. Neste caso, o registrador 4 (R4=a0).

0($29) é o segundo operando. Contém o endereço da memória que deve ser lido. Este endereço é a soma entre zero e o conteúdo do registrador 29 (R29=sp).

Por fim, a última coluna apresenta a mesma instrução em um formato diferente: `lw $a0 0($sp)`. Neste formato, ao invés de referenciar os registradores por números, usa os "apelidos".

A figura não mostra, mas após executar a instrução, o valor contido no registrador "at" será substituído pelo valor contido no endereço de memória `7fffeffc` (o valor de "sp").

A penúltima região, `Data Segments` indica a memória de dados. Neste caso, está dividida em três regiões (data, stack e kernel). Observe que o conteúdo do endereço `7fffeffc` é `0x00000000`.

A parte "DATA" mostra uma faixa de endereços de memória `[0x10000000]..[0x10020000]`. Todos os conteúdos estão zerados.

Já a parte "KERNEL" indica o conteúdo de alguns endereços de memória conforme a tabela abaixo:

Endereço	Conteúdo
0x90000000	0x78452020
0x90000004	0x74706563
0x90000008	0x206e6f69
0x9000000C	0x636f2000

Observe que os endereços mudam de quatro em quatro. É possível acessar bytes, quando os endereços mudariam de um em um. Porém, como estão agrupados palavras, e cada palavra contém quatro bytes, eles mudam de quatro em quatro. Isso está implícito em todos os endereços de memória da figura.

Finalmente, a última parte da figura mostra mensagens. Em caso de erros, eles são indicados ali.

O capítulo 6 inicia o estudo para elaborar programas em assembly.

[1] isso é característica de arquiteturas RISC. Arquiteturas CISC normalmente tem dezenas de formatos de instrução, com tamanho variável.

Capítulo 6

O conjunto de Instruções

O aprendizado de assembly é bastante árduo, um vez que o conjunto de instruções é vasto, e à primeira vista, desconexo. Por isso, a abordagem que é adotada neste texto apresenta primeiro programas Pascal, para então mostrar como construir um programa SPIM correspondente a estes programas.

A prática demonstrou que esta abordagem funciona muito bem, em especial com alunos que tiveram pouco ou nenhum contacto com programação (como é o caso do público-alvo deste texto).

Cada uma das próximas seções irá apresentar programas pascal e associá-los à determinadas construções assembly.

A seção 6.1 apresenta expressões aritméticas enquanto que a seção 6.2 apresenta expressões lógicas. A seção 6.3 mostra como construir laços (for, while, repeat) em assembly, enquanto que a seção 6.4 mostra como construir comandos condicionais (if).

A seção 6.5 apresenta comandos de leitura e impressão de valores. A seção 6.6 lida com dados na memória, e a seção 6.8 mostra como usar o co-processador aritmético e usar as instruções que fazer acesso a números reais no formato ponto flutuante IEEE-754.

O texto não tem a pretensão de apresentar todas as instruções assembly, mas somente um sub-conjunto introdutório. Um texto de referência é [Jam90]. Pode ser obtido gratuitamente e é indispensável.

6.1 Tradução de expressões aritméticas

Considere o algoritmo 3, que contém um programa pascal somente com variáveis inteiras e com atribuições.

```
 1 program exprAritm1;
 2 var a, b, c: integer;
 3 begin
 4     a:=1;
 5     b:=2;
 6     c:=3;
 7     a:=a+b;
 8     b:=b*c;
 9     c:=a mod 3
10 end.
```

Algoritmo 3: Programa exprAritm1.pas

O objetivo é obter um programa assembly equivalente. A primeira coisa a decidir é onde armazenar os dados. No programa, existem três variáveis inteiras, e cada uma delas será armazenado em um registrador.

Normalmente, estas variáveis seriam armazenadas na memória, porém, acesso à memória será visto somente na seção 6.6. Por isso, até chegar lá, todas as variáveis serão armazenadas diretamente em registradores.

Agora, é necessário escolher os registradores apropriados. Existem alguns registradores que não podem ser acessados por programas de usuário. A tabela 6.1 mostra quais os registradores mais importantes.

Apelido	Número	Uso
zero	R0	Constante zero
v0-v1	R2-R3	Cálculo de Expressões. Leitura, Escrita
a1-a3	R4-R7	Parâmetros. Cálculo de Expressões
t0-t7	R8-R15	Temporários. Cálculo de Expressões
s0-s7	R16-R23	Temporários. Cálculo de Expressões
t8-t9	R34-R25	Temporários. Cálculo de Expressões

Como pode ser visto na tabela, várias opções estão disponíveis. Neste primeiro exemplo serão usados os registradores t0, t1 e t2 para armazenar os valores das variáveis `a`, `b` e `c` respectivamente.

Neste ponto é mais simples mostrar o programa assembly equivalente e continuar a análise a partir daí. O algoritmo 4 é este programa.

```
1   .data
2   .text
3   .globl main
4 main:
5   li $t0, 1
6   li $t1, 2
7   li $t2, 3
8   add $t0, $t0, $t1
9   mul $t1, $t1, $t2
10  li $t4, 3
11  rem $t2, $t0, $t4
```

Algoritmo 4: Programa exprAritm1.s equivalente ao programa descrito no algoritmo 3

A linha 1 indica o início da seção de dados que estarão na memória. Como ainda não será abordada a questão de variáveis na memória, não há nenhuma entrada. Este linha pode inclusive ser omitida.

A linha 2 indica o início da seção de texto. Como já foi explicado, o termo "texto" corresponde a "instruções do programa". Estas instruções poderão ser vistas no xspim a partir do endereço `0x8fa40000`. Os endereços de memória variam de `0x00000000` até `0xFFFFFFFF`, ou seja, cada programa tem um espaço de endereçamento de 4Gbytes. Pode parecer estranho, pois nem todas as máquinas tem 4Gbytes de memória física, e mais estranho ainda que cada programa ocupa toda a memória física. A questão toda é que não se trata de memória física, mas de memória virtual. Este tópico será detalhado na seção 6.6.

A linha 3 indica o nome `main` (linha seguinte), é global. Se esta linha for omitida, o simulador acusará um erro.

A linha 4 indica o ponto de início de execução do programa. A execução inicia na próxima instrução (`li $t0, 1`).

As linha 5 até 11 contém o código do programa assembly. As instruções são descritas a seguir:

`li $t0, 1:` `li` são as iniciais para *load immediate*, ou carrega constante. Ao executar esta instrução, a CPU irá armazenar a constante indicada no segundo parâmetro no registrador indicado no primeiro parâmetro. Ou seja, `t0` ← 1.

`li $t1, 2:` `t1` ← 2

`li $t2, 3:` `t2` ← 3

`li $t2, 3:` `t2` ← 3

`add $t0, $t0, $t1:` Soma os dois últimos argumentos e joga o resultado da soma no primeiro. Ou seja, `t0` ← `t0` + `t1`.

...

A maneira mais interessante de entender o que está acontecendo é digitar o programa `exprAritm1.s` e simulá-lo no xspim. No spim, teremos:

```
01: > spim
02: SPIM Version 7.0 of July 7, 2004
03: Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
04: All Rights Reserved.
05: See the file README for a full copyright notice.
06: Loaded: /usr/local/lib/exceptions.s
07: (spim) load "exprAritm1Asm.s"
08: (spim) breakpoint main
09: (spim) run
10: Breakpoint encountered at 0x00400024
11: (spim) step
12: [0x00400024] 0x34080001  ori $8, $0, 1   ; 4: li   $t0, 1
13: (spim) print $8
14: Reg 8 = 0x00000001 (1)
15: (spim) print $9
16: Reg 9 = 0x00000000 (0)
17: (spim) step
18: [0x00400028] 0x34090002  ori $9, $0, 2   ; 5: li   $t1, 2
19: (spim) print $9
20: Reg 9 = 0x00000002 (2)
21: (spim) print $10
22: Reg 10 = 0x00000000 (0)
23: (spim) step
24: [0x0040002c] 0x340a0003  ori $10, $0, 3  ; 6: li   $t2, 3
25: (spim) print $10
26: Reg 10 = 0x00000003 (3)
27: (spim) step
28: [0x00400030] 0x01094020  add $8, $8, $9  ; 7: add  $t0, $t0, $t1
29: (spim) print $8
30: Reg 8 = 0x00000003 (3)
...
```

Foram inseridas linhas neste fragmento do resultado da execução do programa `exprAritm1.s` em spim para simplificar a descrição das ações do usuário e resultados do simulador.

Na linha `07`, o comando `load “exprAritm1Asm.s”` carrega o programa spim apresentado no algoritmo 4. O algoritmo foi digitado e salvo com o nome "exprAritm1Asm.s".

O comando da linha 07 diz ao simulador que crie um ponto de parada *breakpoint* associado com o nome “main”, presente programa spim. Não é obrigatório fazer isso. O usuário pode começar a execução diretamente com o comando `step`, porém os primeiros cinco comandos estão presentes no arquivo `/usr/local/lib/exceptions.s` (carregado na linha 06). Estes comandos iniciais preparam o simulador para receber um novo programa, porém para o nosso exemplo não são necessários. Como exercício, digite os `step` e acompanhe a execução.

A linha 08 diz ao simulador que ele deve inserir um ponto de parada *breakpoint* com nome `main`. Quando for iniciada a simulação, o simulador irá interromper a execução sempre que chegar a um ponto de parada. É possível definir vários pontos de parada.

A linha 09 (`run`) é um comando para iniciar a simulação. O ponto de parada foi encontrado no endereço `0x00400024`, e o simulador interrompeu a execução antes de executar a primeira instrução do algoritmo 4. O aviso de que o ponto de parada foi encontrado pode ser visto na linha 10.

Quando o simulador inicia a execução, todos os registradores de uso geral tem conteúdo igual a zero. É possível verificar o conteúdo de um registrador com o comando `print $<número do registrador>`.

Após encontrar o ponto de parada, o usuário digitou o comando `step`, que pede ao simulador que execute uma única instrução, a próxima instrução. Após o `step`, o simulador indicou que executou a instrução `ori $8, $0, 1 ; 4: li $t0, 1`. O comando `ori` é equivalente ao comando `li`. Para evitar de aumentar a complexidade da CPU, os projetistas resolveram mapear todos os comandos externos `li` para `ori`. Isto ocorre em várias instruções, algumas inclusive são mapeadas para dois ou mais instruções. A última coluna sempre mostra qual o comando "externo" e a coluna central sempre mostra para qual instrução interna foi mapeado.

Examine a linha 12. O comando `li` estava na posição de memória `0x00400024`, foi mapeado para `ori` e é representada internamente como `0x34080001`.

Na linha 13, o usuário digitou o comando `print $8`, e o simulador imprimiu o conteúdo do registrador número 08 (`t0`) na linha 14. São mostrados os valores em hexadecimal centro, e em decimal última coluna.

Os demais comandos mostram os valores dos registradores antes e depois de cada comando em assembly.

Este primeiro exemplo foi explicado em muitos detalhes, porém os próximos exemplos serão bem mais sucintos. Antes destes exemplos, porém, é útil observar a tabela 6.1, que contém algumas instruções assembly que ajudam a traduzir expressões aritméticas.

Comando	Ação	Explicação
`li $1, imm`	$1 ← imm	carrega constante
`move $1, $2`	$1 ← $2	copia valor
`add $1, $2, $3`	$1 ← $2 + $3	soma
`addi $1, $2, imm`	$1 ← $2 + imm	soma imediato
`sub $1, $2, $3`	$1 ← $2 - $3	subtração
`mul $1, $2, $3`	$1 ← $2 * $3	multiplicação
`div $1, $2, $3`	$1 ← $2 / $3	divisão
`rem $1, $2, $3`	$1 ← $2 mod $3	resto da divisão

Tabela 6.1: Comandos para uso em expressões aritméticas

Nesta tabela, `imm` corresponde a "imediato", que é o termo usado para referenciar uma constante numérica. Se o usuário escrever no seu programa `add $at, $t0, 4`, o simulador vai converter esta instrução para `addi $at, $t0, 4`. Confira!

É interessante que o comando `move` não opera como sugerido. "Mover" significa tirar de um lugar e colocar em outro. Porém, curiosamente o nome `move` aqui representa cópia.

Todas as instruções listadas acima tem sua "versão" onde o terceiro parâmetro é uma constante. Assim, existem as instruções `subi`, `muli`, `divi`, etc.. Para conhecer todas as instruções, consulte [Jam90].

A tabela 6.1 mostra que não há como escrever em assembly uma expressão pascal com mais de dois parâmetros. Por exemplo `a:=1+2+3;`. Nestes casos, é necessário usar registradores temporários. Assim, a expressão pascal citada deveria ser traduzida em vários comandos, como por exemplo:

```
li $t0, 1
add $t0, $t0, 2
add $t0, $t0, 3
```

Com isso em mente, traduza o algoritmo 5. Mapeie as variáveis `a`, `b`, `c`, `d` nos registradores `t0`, `t1`, `t2`, `t3`.

```
1  program exprAritm2;
2  var a, b, c, d: integer;
3  begin
4     a:=1;
5     b:=a+2;
6     c:=a+b+3;
7     a:=a+b*c-4;
8     b:=b/c-4;
9     d:=(a+b)/(b-c);
10 end.
```

Algoritmo 5: Programa exprAritm2.pas

6.1.1 Exercícios

1. Escreva um programa SPIM que calcula o resultado da expressão $(8-10+25)/3$ e coloca o resultado em `s0`. É possível fazê-lo em quatro linhas assembly.
2. Escreva um programa SPIM que calcula o resultado da expressão $(64+16)*(2-4)$ e coloca o resultado em `s0`. Qual o menor número de linhas para traduzir este programa?
3. traduza (se necessário), digite e execute passo a passo os algoritmos 3 e 5.
4. Na execução passo a passo destes algoritmos, verifique atentamente o valor do registrador `PC` após cada instrução. Ele muda de quanto em quantos bytes a cada comando?

6.2 Tradução de expressões lógicas

As expressões lógicas são aquelas utilizadas para variáveis lógicas, (ou booleanas), ou para o cálculo de expressões condicionais (if), repetitivas (while, for, repeat), entre outras.

Assim como as instruções assembly que lidam com expressões aritméticas, as instruções assembly que lidam com expressões lógicas tem três operandos. A tabela 6.2 apresenta as instruções lógicas mais utilizadas.

As primeiras instruções da tabela 6.2 podem ser associadas às instruções booleanas vistas na seção 3.4, porém as instruções do tipo `set if` são totalmente novas.

Estas instruções são utilizadas na tradução de expressões de comparação do tipo `a>b`. Se a expressão for verdadeira, coloca-se o valor 1 (que é associado a verdadeiro) no registrador `$1`, e caso contrário, é colocado o valor 0 (associado a falso) neste registrador. Ao final desta instrução, o registrador `$1` contém o resultado da comparação.

Como exemplo, considere o algoritmo 6, traduzido no algoritmo 7.

O algoritmo 7 mapeou a variável `a` para `s1` e a variável `b` para `s0`.

Observe que o comando `b:=TRUE;` foi traduzido como `li $s0, 1`.

Uma pergunta natural aqui é como o computador sabe se `s0` contém uma variável booleana ou uma variável inteira com sinal, ou sem sinal ou alguma outra coisa.

A resposta é que o computador **não sabe!**.

O programador poderia somar um a `s0`. A questão é que isso não faz muito sentido, pois se o programador associou `s0` a uma variável booleana, não faz sentido ele aplicar nela uma operação aritmética.

Comando	Ação	Explicação
`and $1, $2, $3`	$1 ← $2 and $3	and bit a bit
`or $1, $2, $3`	$1 ← $2 or $3	or bit a bit
`nor $1, $2, $3`	$1 ← $2 nor $3	nor bit a bit
`xor $1, $2, $3`	$1 ← $2 xor $3	xor bit a bit
`slt $1, $2, $3`	Se $2 < $3, então $1:=1 senão $1:=0;	*Set if less than*
`sgt $1, $2, $3`	Se $2 > $3, então $1:=1 senão $1:=0;	*Set if greater than*
`sle $1, $2, $3`	Se $2 <= $3, então $1:=1 senão $1:=0;	*Set if less or equal*
`sge $1, $2, $3`	Se $2 >= $3, então $1:=1 senão $1:=0;	*Set if greater of equal*
`seq $1, $2, $3`	Se $2 = $3, então $1:=1 senão $1:=0;	*Set if equal*
`sne $1, $2, $3`	Se $2 ≠ $3, então $1:=1 senão $1:=0;	*Set if not equal*

Tabela 6.2: Comandos para uso em expressões aritméticas

Do ponto de vista da CPU, não é feita nenhuma verificação de tipos. A CPU simplesmente executa o que o programa diz para ela fazer. Se o programa estiver inconsistente, a CPU não vai saber. Vai executar os comandos, que podem até gerar resultados consistentes (ou pelo menos aparentemente consistentes). Por outro lado, o mais comum é que estes programas inconsistentes gerem erros fatais de execução.

Por isso, ao escrever um programa assembly, lembre-se que estas inconsistências não serão detectadas, ao contrário de programas em linguagens de alto nível onde inconsistências como as levantadas acima são detectáveis em tempo de compilação.

6.3 Comandos iterativos

Comandos iterativos são aqueles que estão relacionados com a repetição de comandos. Na linguagem Pascal, os comandos iterativos são `while`, `for` e `repeat`.

Em linguagem assembly, não há nenhuma destas construções. Os comandos repetitivos são implementados através do uso de rótulos e de comandos de desvio.

A seção 6.3.1 apresenta a única construção de desvio do fluxo de execução que a linguagem assembly contém. As seções 6.3.2,6.3.3 e 6.3.4 descrevem como traduzir as construções `while`, `for` e `repeat` respectivamente para linguagem assembly.

6.3.1 Rótulos e Comandos de Desvio

Um rótulo nada mais é do que uma seqüência de letras e números iniciado por letra e terminado com dois pontos (:). Existem vários comandos de desvio, e o primeiro que irá ser abordado é o único comando para desvio incondicional (ou seja, desvia sempre).

Um exemplo de iteração em assembly é apresentado no algoritmo 8.

```
1 program exprBool1;
2 var a, b : boolean;
3 begin
4    b:=TRUE;
5    a:=( (5+4)*2 < 20) and b
6 end.
```

Algoritmo 6: Programa ExprBool1.pas

```
1    .data
2    .text
3    .globl main
4 main:
5    li $s0, 1
6    li $s1, 5
7    add $s1, $s1, 4
8    mul $s1, $s1, 2
9    slt $s1, $s1, 20
10   and $s1, $s1, $s0
```

Algoritmo 7: Programa ExprBool1.s equivalente ao programa descrito no algoritmo 6

Este algoritmo contém um rótulo (sugestivamente chamado `rotulo`, mas que poderia ser `r01`, `loop`, etc., mas que não poderia ser `01t`), e um comando de desvio `j` (jump). A idéia é que cada vez que o programa chegue no comando `j rotulo`, a próxima instrução seja a instrução após `rotulo:`.

Observe que isso afeta o fluxo de execução. Até aqui, a instrução a ser executada era sempre a da "linha de baixo".

O funcionamento de um desvio não é complicado. O fato mais importante a ser lembrado é que o registrador Program Counter (PC) sempre aponta para o endereço da próxima instrução.

Em instruções que não são de desvio, PC aponta para o endereço da instrução sendo executada, mais quatro. Assim, toda vez que a instrução do endereço `XXXX` está sendo executada, o valor de PC será `XXXX + 4`.

Quando a instrução é de desvio (como a instrução `j rotulo`), a regra acima também vale, porém a instrução altera o valor do PC para o endereço do rótulo.

Desta forma, o rótulo é um mnemônico para um endereço. `j rotulo` corresponde a `j endereçoDoRótulo`.

As instruções de desvio são apresentados na tabela 6.3. Esta tabela contém dois comandos de desvio condicional (desvia se registrador $1 é igual/diferente do registrador $2), e um comando de desvio incondicional (jump).

```
1    ...
2 rotulo:
3    ...
4    j rotulo
```

Algoritmo 8: Programa assembly com rótulo e desvio

Comando	Ação	Explicação
`beq $1, $2, L`	Se $1=$2, goto L	branch on equal
`bne $1, $2, L`	Se $1≠$2, goto L	branch on not equal
`j L`	goto L	jump

Tabela 6.3: Comandos para uso em desvios

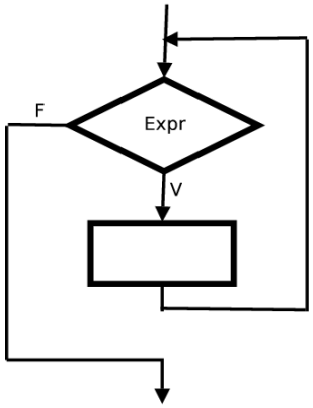


Figura 6.1: Fluxo de execução do comando while

Os iniciantes em programação de computadores são ensinados a evitar o uso de comandos “goto”, porém em assembly, estes são tudo o que resta. Não há outra opção.

6.3.2 Tradução do comando while

Considere agora o algoritmo 9. Este algoritmo apresenta uma iteração que ocorre através do comando `while` da linguagem Pascal.

Antes de apresentar o programa assembly equivalente, é necessário destacar o funcionamento do comando `while`. Este comando faz com que os comandos do while sejam executados de zero a *n* vezes (ou seja, é possível que os comandos não sejam executados nenhuma vez).

Ou seja, é necessário avaliar a expressão, verificar se ela é verdadeira, e se não for, desviar para o fim das repetições.

A figura 6.1 indica o fluxo de execução do comando while. O esquema de tradução do comando while, apresentado a seguir, é baseado nesta figura.

while (E) do	$\left\{ \begin{array}{l} RXX: \\ \quad traduz\ E \\ \quad \$t0 := E \\ \quad beq\ \$t0,\ \$0,\ RYY \end{array} \right.$
begin	
...	{*Traduz comandos*
end	$\left\{ \begin{array}{l} \quad j\ RXX \\ RYY: \end{array} \right.$

Este esquema de tradução é apropriado para a tradução de qualquer construção `while`. Ao encontrar o `while`, deve-se anotar o nome de um rótulo (aqui descrito como `RXX`, que representa o início do laço repetitivo), e as duas últimas linhas contém o desvio para o início do laço `j RXX` e o rótulo para o qual o fluxo deve ser desviado quando a expressão for falsa.

Ao lado do rótulo `RXX` está indicado que deve ser traduzida a expressão e o resultado deve ser atribuído ao registrador `$t0` (pode ser qualquer registrador, mas aqui será usado sempre este registrador).

O resultado da expressão é “verdadeiro” ou “falso”, que já foi visto na seção 3.4 são sempre mapeados em 32 bits para `1` e `0` respectivamente.

Este resultado é comparado com o registrador `$0`. Este registrador contém a constante zero (veja tabela 6.1). Ele foi sabiamente incluído no elenco de registradores para simplificar a comparação com “verdadeiro” ou “falso”. Neste caso, a comparação é se o resultado contido em `$t0` é falso (igual a zero). Se for, o fluxo deve ser desviado para o rótulo RYY. Caso contrário, a execução segue para o próximo comando.

Estes comandos correspondem à tradução dos comandos entre `begin` e `end` (que podem não existir, caso seja um único comando).

Após traduzir todos estes comandos, chega-se ao fim do laço (no caso descrito como `end`). Ao encontrar este comando, é necessário desviar o fluxo para o início da execução do laço novamente (`j RXX`). E o próximo assembly contém o rótulo para o qual o fluxo deve ser desviado ao fim da execução do laço.

Agora, um exemplo prático. O algoritmo 9 é um programa pascal que contém um laço. A tradução deste programa é apresentada no algoritmo 10.

```
1 program while1;
2 var a, b : integer;
3 begin
4    a:=1;
5    b:=5;
6    while (a<=b) do
7       a:=a+1;
8    a:=b;
9 end.
```

Algoritmo 9: Exemplo de iteração com o comando while.

Foram destacadas as linhas do algoritmo 10 que foram inseridas a partir do modelo apresentado. Analise atentamente a relação entre o modelo de tradução e o programa assembly resultante. Este programa não apresenta `begin` e `end`.

```
1    .data
2    .text
3    .globl main
4 main:
5    li $s0, 1
6    li $s1, 5
7  while:
8    sle $t0, $s0, $s1
9     beq $t0, $0, fim
10   add $s0, $s0, 1
11    j loop
12 fim:
13   move $s0, $s1
```

Algoritmo 10: Programa while.s equivalente ao programa descrito no algoritmo 9

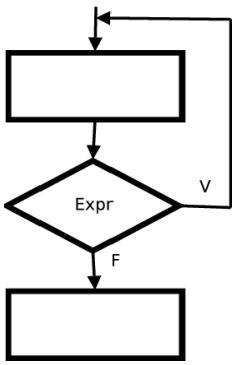


Figura 6.2: Fluxo de execução do comando repeat

6.3.3 Tradução do comando for

O comando `for` é praticamente igual ao comando `while`. A diferença é que:

1. o comando `for` tem uma forma de iniciar a variável que usada para testar o fim da iteração (nos exemplos, a variável `a`);
2. o comando `for` tem um forma de somar um a esta variável ao final de cada laço.

```
1 program for1;
2 var a, b : integer;
3 begin
4     b:=5;
5     for a:=1 to b do;
6 a:=b;
end.
```

Algoritmo 11: Exemplo de iteração com o comando for.

O algoritmo 11 é equivalente ao algoritmo 9. Tanto que a versão assembly deste algoritmo é exatamente igual (algoritmo 10).

6.3.4 Tradução do comando repeat

O comando `repeat` tem um fluxo de execução diferente do fluxo de execução dos comandos `while` e `for`. O comando `repeat` exige que o comando entre o `repeat` e o `until` seja executado pelo menos uma vez **antes** de verificar a expressão.

A figura 6.2 mostra o fluxo de execução de um comando `repeat`. Baseado no fluxo de execução apresentado na figura, é possível definir mais facilmente o esquema de tradução apresentado abaixo.

repeat RXX :
...
until (E)
{*Traduz comandos*
traduz E
$\$t0 := E$
beq $\$t0, \$0, RXX$

O esquema de tradução mostra que ao encontrar um comando `repeat`, ele é traduzido para um rótulo `RXX`. Em seguida, devem ser traduzidos os comandos um a um. Por fim, ao encontrar o `until (E)`,

calcula-se a expressão e se ela for falsa, o fluxo retorna para o rótulo `RXX`. Se a expressão for verdadeira, o fluxo segue adiante.

Como exercício, traduza o algoritmo 12 abaixo para SPIM:

```
1  program repeat;
2  var i, a : integer;
3  begin
4     i:=0;
5     a:=0;
6     repeat
7        a:=a + i*2;
8        i:=i+1;
9        until (i>10);
10    end.
```

Algoritmo 12: Programa que usa o comando `repeat`.

6.3.5 Exercícios

1. Em quais registradores foram mapeadas as variáveis do algoritmo 9?
2. Desenhe setas no algoritmo 10, indicando quais os possíveis fluxos de execução de cada comando.
3. Digite, se simule a execução do algoritmo 10 no spim ou xspim. Preste atenção ao valor de `PC` a cada passo, e como ele é mudado com as instruções `beq` e `j`.
4. Traduza os programas 13 e 14.

```
1  program while2;
2  var i, a : integer;
3  begin
4     i:=0;
5     a:=0;
6     while(i<=10) do
7     begin
8        a:=a + i*2;
9        i:=i+1;
10    end;
11 end.
```

Algoritmo 13: Programa que usa o comando `while`.

5. Os programas descritos nos algoritmos 12,13 e 14 são equivalentes, porém usam construções repetitivas diferentes.
   (a) Examine os programas assembly de cada um deles e indique as diferenças encontradas.
   (b) As construções diferentes exigiram expressões diferentes para analisar o fim do laço. Descreva estas diferenças.

```
1 program for2;
2 var i, a : integer;
3 begin
4    a:=0;
5    for i:=0 to 10 do
6       a:=a + i*2;
7    end.
```

Algoritmo 14: Programa que usa o comando `for`.

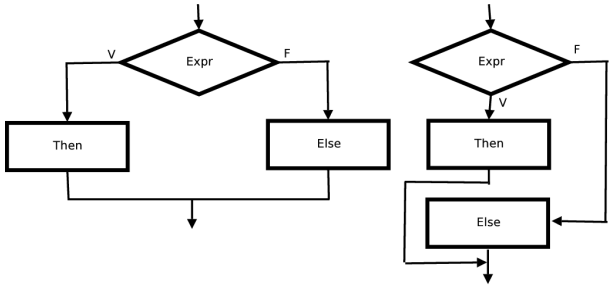


Figura 6.3: Fluxo de execução do comando if-then-else

6.4 Comandos condicionais

Assim como foi feito com as construções repetitivas, a tradução de comandos repetitivos será feita baseado no fluxo de execução do comando `if`. O fluxo "convencional" está apresentado do lado esquerdo da figura 6.3. Este fluxo é um pouco diferente, pois dois blocos são apresentados lado a lado (bloco do `then` e o bloco do `else`). Isto poderia sugerir, erroneamente, que o programa assembly deveria também ter duas colunas de código, e não é assim que deve ser.

O lado direito da figura 6.3 mostra o mesmo fluxo, porém este tem todos os blocos em uma única coluna, o que torna mais intuitivo esquema de tradução apresentado abaixo.

if (E) { *traduz E*; $t0 := E; *beq* $t0, $0, *RELSE*
then { *traduz comandos then*; *j RFIM*
else { *RELSE* : *traduz comandos else*
RFIM :

O esquema de tradução mostra que inicialmente traduz-se a expressão, e o resultado é colocado em um registrador (neste exemplo, `$t0`). Dependendo do resultado da expressão, o fluxo pode tomar duas alternativas:

1. se a expressão for verdadeira (`$t0=1`), o fluxo deve ir para o trecho contido no `then` e depois deve sair do `if` sem passar pelo `else`;
2. se a expressão for falsa (`$t0=0`), o fluxo deve pular os comandos contidos no `then` e ir direto para os comandos do `else`.

Para implementar este funcionamento, a instrução assembly

```
beq $t0, $0, RELSE
```

testa se o resultado da expressão é falso. Se for, desvia o fluxo para o rótulo `RELSE`. O que segue este rótulo são os comandos do else, e após executá-los, o fluxo segue normalmente para o que vier após o

comando `if`.

Se a expressão for verdadeira, o fluxo segue pelos comandos relacionados pelo `then` para, ao final, desviar incondicionalmente (`j RFIM`) para o rótulo que indica o fim do comando `if`.

Compare detalhadamente este esquema de tradução com a figura. Observe que os desvios de fluxo estão relacionados sempre com comandos de desvio em assembly e os pontos de entrada destes fluxos estão sempre relacionados com rótulos.

O algoritmo 15 utiliza o comando `if`. A tradução deste algoritmo é apresentado no algoritmo 16.

```
1 program if1;
2 var a, b : integer;
3 begin
4    a:=1;
5    if (a<5) then
6       b:=1;
7    else
8       b:=2;
9 end.
```

Algoritmo 15: Programa que usa o comando `if`.

```
1     .data
2     .text
3     .globl main
4  main:
5     li $s0, 1
6     slt $t0, $s0, 5
7     beq $t0, $0, else
8     li $s1, 1
9     j fim
10 else:
11    li $s1, 2
12 fim:
```

Algoritmo 16: Programa if.s equivalente ao programa descrito no algoritmo 15

Como exercício, traduza o algoritmo 17. Cuidado na hora de definir os rótulos. Uma sugestão é usar rótulos como `else1` e `else2` para referenciar o primeiro e segundo comando `if`.

6.5 Comandos de entrada e de saída

Até o momento, todos os programas apresentados não fazem leitura de dados e nem imprimem resultados. Os exemplos apresentados só são úteis para fins didáticos, uma vez que qualquer programa útil gera algum resultado.

Esta seção abordará a leitura e impressão de dados. Existe um comando SPIM para tal, e ele é usado para ler e imprimir qualquer tipo de dado.

Por uma questão de segurança, um programa não deveria ter acesso direto a nenhum dispositivo de entrada ou saída. Quando o usuário tem acesso direto a um dispositivo, pode inadvertidamente executar alguma operação que danifica o dispositivo. Em alguns casos, pequenos programas “intrusos”, também chamados de vírus, fazem isso sabendo exatamente o que estão fazendo.

```
1  program if2;
2  var a, b, p : integer;
3  begin
4     a:=5;
5     b:=4;
6     if (a>b) then
7        p:=a+b
8     else
9        if (a<2*b) then
10          p:=1
11       else
12          p:=2
13 end.
```

Algoritmo 17: Exercício com ifs aninhados.

Existe uma conjunto de sistemas operacionais que foram projetados para minimizar a possibilidade de ação destas operações. Linux é um dos sistemas operacionais que implementa isso. A idéia é que o sistema operacional dá uma "caixa de areia" para o programa em execução. Dentro desta caixa de areia, ele pode fazer o que quiser, que não vai danificar o "parquinho".

Porém, se o programa quiser acessar recursos fora desta caixa de areia, só poderá fazê-lo se pedir para alguém responsável buscar este recurso e colocá-lo dentro da caixa de areia. Nenhum programa pode sair dali de forma alguma.

Esta analogia ajuda a explicar o funcionamento das operações de entrada e saída. A idéia básica é que o programa não pode acessar nenhum dispositivo de entrada e saída diretamente. Ele deve sim pedir para que o sistema operacional faça esta operação para ele.

Desta forma, o programa deve interagir com o sistema operacional, e isto é feito de uma maneira bastante rígida. A idéia é que o programa indique o que deve ser feito colocando valores em alguns registradores específicos e depois deve chamar o sistema operacional através da instrução `syscall`.

Ao ser chamado, o sistema operacional verifica o que foi pedido olhando o conteúdo de alguns registradores e executa a operação solicitada. Quando terminar de fazer isso, devolve o controle ao programa.

Assim, para imprimir um valor inteiro, deve-se fazer o seguinte:

1. armazena a constante 1 no registrador `$vo`.
2. armazena o valor a ser impresso no registrador `$a0`.
3. instrução `syscall`.

A instrução `syscall` chamará o sistema operacional. Este examinará o valor em `$vo`. Neste caso, o valor será 1, o que indica que o sistema operacional deve executar uma operação de impressão de um número inteiro (mais especificamente do valor inteiro contido no registrador `$a0`. O sistema operacional traduzirá o conteúdo binário do registrador `$a0` em um conjunto de caracteres que representa aquele valor binário em decimal, e imprimirá o conjunto de caracteres na tela. Em seguida, retorna o controle de execução ao programa, para a linha que segue o comando `syscall`.

A constante armazenada em `$vo` é também chamado de "serviço". Existem vários serviços que o sistema operacional pode executar. Alguns estão listado na tabela 6.4.

Esta tabela apresenta somente três serviços: impressão de um número inteiro, leitura de um número inteiro e fim de programa. Existem outros que serão abordados posteriormente.

O algoritmo 18 faz a leitura e a impressão de um número inteiro. A tradução deste algoritmo é apresentada no algoritmo 19.

Serviço	$v0	Parâmetro	Resultado
Impr. Inteiro	1	$a0 ← num	
Lê Inteiro	5		$v0 ← num. lido
Fim Pgma	10		

Tabela 6.4: Alguns serviços do Sistema Operacional

```
1 program leImpr;
2 var a: integer;
3 begin
4    read(a);
5    write(a);
6 end.
```

Algoritmo 18: Programa que lê valor inteiro e o imprime.

As linhas 5 e 6 do algoritmo 19 são a tradução do comando `read`. O serviço número 5 solicita que o sistema operacional leia um número do teclado. Este número lido é um conjunto de dígitos que o sistema operacional converte para binário e este número binário é armazenado no registrador `$vo` (veja tabela 6.4).

É interessante simular a execução deste programa. Após o syscall da linha 6, verifique o valor contido em `$v0`.

A linha 7 copia o valor lido para o registrador `$a0`. Isto é necessário pois a impressão usa o registrador `$v0` para indicar o serviço e também porque o valor a ser impresso é o valor contido em `$a0`.

As linhas 7, 8 e 9 correspondem ao comando `write`. A linha 7 coloca o valor a ser impresso em `$a0`. A linha 8 coloca o serviço em `$ v0` e a linha 9 chama o sistema operacional.

Quando o sistema operacional é chamado, ele observa que o serviço é de impressão de número inteiro e converte o valor binário contido em `$a0` para um conjunto de caracteres e imprime este conjunto de caracteres.

Observe que se o programador errar o número do registrador, por exemplo, colocando o valor a ser impresso em `$a1` ao invés de `$a0`, o sistema operacional irá imprimir o conteúdo de `$a0`.

Por fim, as linhas 10 e 11 correspondem ao serviço de finalização de um programa. O serviço número 10 indica ao sistema operacional que o programa acabou. Após o `syscall` da linha 11, o programa pára a execução.

Até o momento, não foi usado este serviço por uma questão de simplicidade. Porém, a partir deste

```
1    .data
2    .text
3    .globl main
4  main:
5    li $v0, 5
6    syscall
7    move $a0, $v0
8    li $v0, 1
9    syscall
10   li $v0, 10
11   syscall
```

Algoritmo 19: Program spim equivalente ao apresentado no algoritmo 18

momento, será usado em todos os exemplos.

Existem outros serviços como leitura e impressão de caracteres, leitura e impressão de números ponto flutuante, entre outros, que ainda serão vistos neste texto.

6.5.1 Exercícios

1. Altere o algoritmo 17 para que ele leia os valores de `a` e `b` no início (ao invés de atribuir constantes) e ao final imprima o valor de b. Digite valores para `a` e `b` que façam o programa entrar em todas as ramificações dos `if`.

6.6 Acesso à memória

Até o momento, todos os programas apresentados utilizavam somente registradores. É interessante que um programa seja capaz de armazenar todas as suas variáveis em registradores, pois o acesso é mais rápido, porém, existem programas que lidam com grandes quantidades de dados, e neste caso, é impossível guardar todos estes dados em registradores o tempo todo.

No SPIM (e em praticamente todas as máquinas RISC), todas as instruções utilizam parâmetros que estão em registradores. Quando é necessário o uso de uma variável que não está em um registrador, primeiro deve-se mapear esta variável para um registrador para então executar a operação desejada sobre ele. Por fim, normalmente armazena-se o valor novamente na memória.

Assim, existem somente duas instruções de acesso à memória no SPIM. Uma para trazer um valor da memória para registrador (`load`) e uma para armazenar o valor contido em um registrador na memória (`store`).

Como são poucas instruções, poderia parecer que esta seção deveria ser bastante curta, porém ela não é, uma vez que é necessário explicar muitos aspectos de como a memória é vista de dentro de um programa assembly.

Comando	Ação	Explicação
`lw $1, cte($2)`	$1 ← Mem[$2+cte]	carrega palavra
`lw $1, cte`	$1 ← Mem[cte]	carrega palavra
`sw $1, cte($2)`	Mem[$2+cte] ← $1	armazena palavra
`sw $1, cte`	Mem[cte] ← $1	armazena palavra

Tabela 6.5: Comandos para acesso à memória

A tabela 6.5 contém as duas instruções. A idéia básica é que o endereço de memória cujo conteúdo se deseja carregar ou armazenar esteja em um registrador ou em uma combinação entre registrador e constante (`cte`). Esta constante é indicada como um rótulo dentro da seção `data` do programa assembly. Até o momento, esta seção não foi utilizada, apesar de aparecer nos programas anteriores.

Neste ponto é necessário explicar como a memória de um programa em execução está organizada. A figura 6.4 mostra como um programa SPIM está colocado na memória.

Um programa em execução no simulador SPIM ocupa sempre toda a memória (4Gbytes). O simulador não foi projetado para permitir que mais de um programa execute em um mesmo momento.

Observe que o programa ocupa 4Gbytes! Em sistemas linux (entre outros), cada programa ganha um espaço virtual de 4Gbytes. Se 10 programas estiverem em execução simultaneamente, cada um deles ocupa 4Gbytes de memória virtual (total de 40Gbytes de memória virtual). Isso não quer dizer que qualquer um deles ocupe muito espaço em memória física. Existe um mapeamento de memória virtual para física que usa somente o espaço de memória física que é necessário, e nada mais. Com isso, é possível de executar centenas de programas simultaneamente, todos com muita memória virtual, e pouca memória física. Maiores detalhes deste processo pode ser visto em livros de Sistemas Operacionais, no tópico “Memória Virtual”.

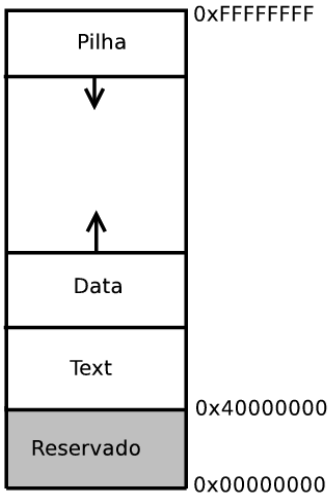


Figura 6.4: Um programa na memória

O que é importa aqui é que o programador saiba que seu programa ocupa um grande espaço de memória física, e que o programa é mapeado para determinados endereços.

Na figura 6.4, pode-se identificar várias faixas de endereços:

0x00000000 .. 0x40000000 Esta é uma área reservada onde normalmente é mapeado o programa Sistema Operacional. Não será usado aqui.

0x40000000 .. 0x???????? Esta faixa armazena as instruções do programa em execução e sobre este, a área de dados. A área de dados inicia após a última instrução da área `text`. No XSPIM, é possível observar qual o endereço da primeira instrução. O final desta faixa não é determinável.

0x???????? .. 0xFFFFFFFF esta é a faixa superior, usada para a pilha. Esta pilha não será usada aqui, mas é utilizada para implementar o funcionamento de funções e procedimentos.

Observe as duas setas. A área entre estas setas é grande, e pode ser alterada ao longo da execução do programa. Na parte de cima, a seta diz que a seção da pilha pode crescer para baixo. Na parte de baixo, a seta logo acima de `data` indica que esta parte pode crescer para cima (é aqui que são armazendados os dados dinâmicos). Porém, nenhum destes será abordado neste texto.

Como primeiro exemplo de uso de memória, veja o algoritmo 20, traduzido no algoritmo 21.

```
1 program mem1;
2 var a : integer;
3 begin
4    read(a);
5    while (a<10) do
6       a:=a+1;
7 end.
```

Algoritmo 20: Programa para acesso à memória.

A primeira novidade aparece na seção `data`, onde aparece a linha `A: .word 0` . Agora, `A` é um rótulo, que ocupa o espaço de uma palavra (32 bits - oito bytes) e que é iniciada com zero. Assim como os rótulos na seção de texto, este rótulo estará associado com um endereço da memória (a `cte` da tabela 6.5.

```
1    .data
2    A: .word 0
3    .text
4    .globl main
5  main:
6    li $v0, 5
7    syscall
8    move $s0, $v0
9     sw $s0, A
10 R00:
11    lw $s0, A
12   slt $t1, $s0, 10
13   beq $t1, $0, R01
14   add $s0, $s0, 1
15    sw $s0, A
16   j R00
17 R01:
18   li $v0, 10
19 syscall
```

Algoritmo 21: Programa equivalente ao programa descrito no algoritmo 20

Para o algoritmo 21, o acesso à memória é totalmente dispensável, pois como só tem uma única variável, ela poderia muito bem ser mapeada para um registrador. Se forem eliminadas as linhas em negrito, o programa funciona.

Porém, ele foi apresentado para apresentar as instruções de acesso à memória. Para isso, digite o programa e simule-o para responder as seguintes perguntas:

1. Qual o endereço de memória de **A**? É possível identificá-lo ao longo da execução, nas instruções `load` ou `store`. Sabendo este endereço, pergunta-se: quantos bytes ocupa o código executável da seção `text`?
2. Verifique o que ocorre no conteúdo deste endereço de memória após a instrução `sw A, $s0`. (no xspim é mais simples de ver).
3. Altere o programa assembly para conter duas variáveis adicionais, digamos `B` e `C`. A primeira deve vir antes de `A` e a segunda deve vir depois de `A`. Não esqueça de "declará-las" como `.word 0`. Simule o programa novamente. Explique a mudança no endereço de `A`. Qual o endereço de `B` e `C`?
4. Continuando o exercício anterior, se forem retirados os `.word 0`, explique porque o endereço de `A` volta a ser o que era. Qual o endereço de `B` e `C` agora?

6.6.1 Acesso a elementos de vetor

O acesso à memória passa a ser indispensável quando a quantidade de variáveis é maior do que o número de registradores. Um exemplo típico em que isto ocorre é em vetores.

Como exemplo, considere o algoritmo 22. Nele, há um vetor com onze elementos. O programa lê onze números e os coloca nos onze elementos dos vetores.

Para a tradução deste programa, a única novidade é a construção `a[i]`. Para explicar esta construção, é necessário primeiro explicar como um vetor de uma linguagem de alto nível é implementado em assembly.

A primeira coisa a se lembrar é que as únicas instruções de acesso à memória são `lw` e `sw`, e ambas precisam conhecer o endereço de cada um dos elementos para acessá-los. Por isso, é necessário primeiro compreender como os endereços são atribuídos a cada um dos elementos.

```
1  program vet1;
2  var a : array[0..10] of integer;
3  i : integer; begin
4     i:=0;
5     while (i<=10) do
6     begin
7        read(a[i]);
8        i:=i+1;
9     end;
10 end.
```

Algoritmo 22: Programa com vetor.

Este vetor ocupa 11 posições, e cada posição corresponde a um inteiro de 32 bits (quatro bytes). Cada elemento é armazenado "lado a lado" com os outros, ou seja, e elemento `a[0]` estiver no endereço X, o elemento `a[1]` estiver no endereço $X+4$, o elemento `a[2]` estará no endereço $X+8$, e assim por diante. A tabela 6.6 explica o mapeamento.

Pascal	Endereço
`a[0]`	$X+0$
`a[1]`	$X+4$
`a[2]`	$X+8$
`a[3]`	$X+12$
`a[4]`	$X+16$
...	...
`a[i]`	$X+i*4$

Tabela 6.6: Endereços de elementos de um vetor de inteiros

Se for conhecido o endereço do elemento `a[0]`, é possível determinar o endereço de qualquer dos outros elementos. Se o elemento procurado for o elemento `a[i]`, então o endereço do elemento pode ser obtido pela fórmula: $X+i*4$.

Em assembly existe uma forma de alocar espaço para um vetor, e a sintaxe dos comandos `sw` e `lw` possibilitam usar o endereço do elemento "base" da tabela `a[0]`. Isto pode ser visto no algoritmo 23, equivalente assembly do algoritmo 22.

Neste programa, está indicado em negrito os comandos até o momento não conhecidos. A linha 3 apresenta o comando `A: .space 44`. Isto indica que deve ser alocado um espaço de 44 bytes para o rótulo `A`. Através deste mecanismo, é possível alocar espaço para o vetor.

A variável `I` está mapeada no registrador `$s0`, e sempre que esta variável recebe alguma alteração no programa principal, esta alteração é também repassada ao endereço de memória reservado para `I` (linhas 9 e 17). Compare estas linhas com a tabela 6.5. Nestas linhas, não foram usados os parênteses.

Já o acesso aos elementos dos vetores é mais complicado, e neles serão usados os parênteses. A operação de armazenamento de dados em um vetor é apresentada nas linhas 14 e 15. Na linha 14, calcula-se quantos bytes o elemento `a[i]` está afastado de `A`. Esta distância é chamada de "deslocamento". Resumidamente, temos que o deslocamento é armazenado em `$t0`. Observe que o multiplicador é 4, uma vez que cada elemento é um inteiro. Outros tipos exigem outros multiplicadores. Se cada elemento fosse um caractere, o multiplicador deveria ser 1, e se cada elemento fosse um ponto flutuante de precisão dupla, o multiplicador deveria ser 8 e se fosse ponto flutuante de precisão simples, o multiplicador deveria ser 4, e assim por diante.

```
1     .data
2  I: .word 0
3   A: .space 44
4     .text
5     .globl main
6  main:
7     li $s0, 0
8  while:
9     lw $s0, I
10    sle $t1, $s0, 10
11    beq $t1, $0, fim
12    li $v0, 5
13    syscall
14    mul $t1, $s0, 4
15     sw $v0, A($t1)
16    add $s0, $s0, 1
17    sw $s0, I
18    j while
19 fim:
20    li $v0, 10
21    syscall
```

Algoritmo 23: Programa equivalente ao programa descrito no algoritmo 22

A linha 15 faz a operação de escrita na memória. A instrução é `sw $v0, A($t1)`, que conforme a tabela 6.5 significa que o valor contido em `$v0` deverá ser armazenado no endereço indicado por `A` (base) somado com o conteúdo de `$t1` (deslocamento).

É importante simular este programa no SPIM e verificar o endereço inicial de `A`, e que os armazenamentos dos valores lidos ocorrem corretamente nos endereços indicados na tabela 6.6.

6.6.2 Exercícios

1. Observe mais uma vez a figura 6.4. Foi dito que a área da seção ".data" inicia logo após a última instrução. Isto implica dizer que não é uma área que inicia sempre em um mesmo endereço. Para comprovar isso, anote o endereço da variável `I` do algoritmo 23. Em seguida, acrescente uma instrução, por exemplo, duplique a primeira linha do programa e simule novamente a execução. Qual o novo endereço de `I`?
2. Digite e simule a execução de todos os programas desta seção. Verifique os endereços das variáveis de memória.
3. Traduza e simule a execução do programa 24.
4. Traduza e simule a execução do programa 25.

6.7 Operações sobre caracteres

Os caracteres no SPIM seguem o padrão ASCII. O algoritmo 26 é um programa SPIM que lê um conjunto de caracteres (um vetor), altera-o e o imprime novamente.

As novidades deste programa são:

```
1  program vet2;
2  var a : array[0..10] of integer;
3      i : integer;
4  begin
5      for i:=0 to 10 do
6          read(a[i]);
7      for i:=0 to 10 do
8          write(a[i]);
9  end.
```

Algoritmo 24: Programa lê dados e imprime dados do vetor.

```
1  program vet3;
2  var a : array[0..7] of integer;
3      i : integer;
4  begin
5      for i:=0 to 7 do
6          read(a[i]);
7      for i:=1 to 6 do
8          if (a[i]>0) then
9              a[i]:=a[i-1]+a[i+1]
10         else
11             a[i]:=a[i-1]-a[i+1];
12     for i:=0 to 7 do
13         write(i, a[i]);
14 end.
```

Algoritmo 25: Programa com mais subscritos mais complexos.

1. A definição de `msg` é do tipo `asciiz`. Isto significa que após o último caractere ascii é inserido um 0_{16}.
2. A definição de `pl` é a cadeia `\n. Esta cadeia corresponde a pulo de linha.`
3. `A leitura da cadeia é feita nas linhas 8-11. O serviço é o de número 8, o ponto de início da impressão é dado por $a0, que contém o endereço de str ( la="load address"). O tamanho máximo da cadeia deve ser indicado em $a1, e neste caso é de 20 bytes (linha 10).`
4. `Como interessa ler um caractere (byte) de cada vez, é usada a operação lb=load byte. A sintaxe é a mesma de lw.`
5. `A impressão de uma cadeia corresponde ao serviço número 4. Este serviço converte os caracteres a partir do endereço $a0 até encontrar um caractere igual a zero. Observe que o syscall` **não** `sabe o limite da cadeia (que aqui é de 20 bytes). Ele vai simplesmente imprimindo os caracteres até encontrar um byte igual a zero.`

Não foi apresentado um programa Pascal para este programa SPIM, e o motivo é que Pascal e SPIM tratam strings de maneira diferente, especialmente no que tange a forma de indicar o tamanho do string.

Em Pascal, o tamanho de um string é armazenado antes do string, e cada acesso a um elemento do string passa antes pela verificação se o índice acessado está fora dos limites do string ou não. Isso é útil para

garantir que o acesso a um elemento é válido ou não em tempo de execução. Para verificar o tamanho do string, usa-se a função `length()`.

O SPIM adota um outro modelo, baseado na linguagem C. Neste modelo, o último caractere do string armazena um caractere especial, o "indicador de fim do string". Este caractere é $(0)_{16}$. Assim, para saber onde está o fim do string, basta procurar por uma posição cujo conteúdo é $(0)_{16} = (0)_{10}$. Como não há verificação se o elemento acessado está dentro dos limites do vetor, a execução é um pouco mais rápida do que em Pascal. O inconveniente é que se, por uma graça dos Deuses, este caractere for removido, o string continua indefinidamente. É divertido ver o que ocorre quando é pedido que seja impresso um string que "perdeu" o indicador de fim de string.

Como os dois modelo são incompatíveis, este programa não apresenta versão Pascal.

```
1   .data
2   str: .space 20
3   msg: .asciiz "Result: "
4   pl : .asciiz "\n"
5   .text
6   .globl main
7 main:
8   li $v0, 8
9   la $a0, str
10  li $a1, 20
11  syscall
12 ini:
13  lb $t1, str($t0)
14  beq $t1, $0, fim
15  add $t1, $t1, 1
16  sb $t1, str($t0)
17  add $t0, $t0, 1
18  j ini
19 fim:
20  li $v0, 4
21  la $a0, msg
22  syscall
23  li $v0, 4
24  la $a0, str
25  syscall
26  li $v0, 4
27  la $a0, pl
28  syscall
29  li $v0, 10
30  syscall
```

Algoritmo 26: Programa que lida com cadeias de caracteres

Quando se define uma "variável" na seção `.data`, existem diretivas que permitem dizer se ela ocupa um byte, dois, quatro, ou vários bytes. O texto se concentra em algumas poucas. Para ver todas as demais, veja [Jam90].

6.7.1 Exercícios

1. O que será impresso pelo programa indicado no algoritmo 27? Porque isso ocorre? Como corrigir?
2. escreva um programa SPIM que imprime os códigos decimais de toda a tabela ASCII. Em cada linha, imprima o decimal, pelo menos um caractere em branco, e o caractere ASCII correspondente.

```
1   .data
2   str1: .ascii "1234"
3   str2: .ascii "56789"
4   pl : .asciiz "\n"
5   .text
6   .globl main
7 main:
8   li $v0, 4
9   la $a0, str1
10  syscall
11  li $v0, 10
12  syscall
```

Algoritmo 27: Programa exemplo de problema

6.8 Operações sobre números reais

Números reais são representados em notação ponto flutuante. Lembre que a representação de um número em ponto flutuante é diferente da representação de número natural e inteiro. Além disso, as operações sobre reais são diferentes das operações sobre inteiros e naturais (compare o processo de soma de dois números ponto flutuante, da seção 3.5.5.1 com dois números naturais, da seção 3.2.3).

Por isso, não é natural utilizar o mesmo conjunto de registradores para armazenar reais, naturais e inteiros. Como as operações utilizam processos diferentes, usa-se uma CPU separada para tratar de números representados em ponto flutuante. Esta CPU separada é chamada normalmente de co-processador aritmético.

A medida que as CPUs diminuíram de tamanho, observou-se que era possível incluir o co-processador aritmético na pastilha da CPU principal. Assim, atualmente quando se compra uma CPU, compra-se também um co-processador aritmético na mesma pastilha.

A CPU do SPIM segue este modelo. A figura 6.5 mostra a sua arquitetura. Existe um processador principal (a CPU) e dois co-processadores:

1. co-processador de excessões. Este co-processador trata de excessões do tipo “divisão por zero”, “falha de página”, etc..
2. co-processador aritmético (também chamado de unidade de ponto flutuante[1]).

Como a figura mostra, tanto CPU, FPU (co-processador 1) e o co-processador 0 contém conjuntos internos de registradores e de unidades aritméticas (aquelas onde que faz as operações de soma, subtração, etc.). Cada uma, individualmente, tem acesso à memória e também tem acesso uma à outra através de barramento próprio.

A FPU tem um total de 32 registradores de ponto flutuante de precisão simples (32 bits), chamados `$f0`, `$f1`, `$f2`, … `$f31`.

Estes registradores podem ser combinados em 16 registradores de precisão dupla (64 bits), chamados `$f0`, `$f2`, `$f4`, … `$f30`. **Os registradores ímpares (`$f1`, `$f3`, etc.) não podem ser usados como de precisão dupla**.

[1] em inglês, *Floating Point Unit (FPU)*

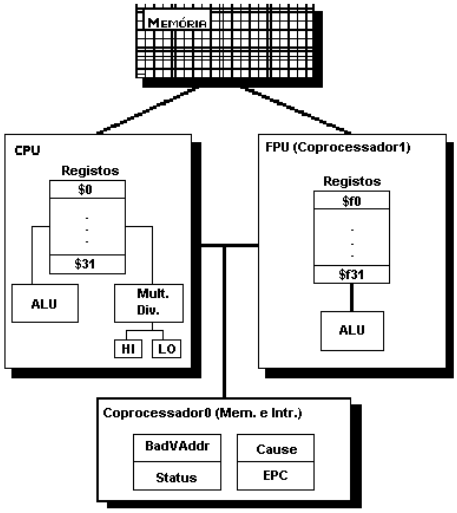


Figura 6.5: A CPU SPIM e seus co-processadores (extraído de [Jam90])

A semelhança nos nomes não é obra do acaso. Cada um dos registradores de precisão dupla é composto por dois registradores de precisão simples. Assim, o registrador de precisão dupla `$f0` é composto pelos registradores de precisão simples `$f0` e `$f1` (é como os 32 bits de um fossem concatenados com os 32 bits do outro). O ideal seria que os dois conjuntos de registradores fosses independentes, mas não são.

Por esta razão, quando o programador coloca uma constante no registrador de precisão dupla `$f0`, ele também está alterando o conteúdo dos registrados `$f0` e `$f1` de precisão simples. É importante estar atento a isso para não cometer erros.

Como a FPU é uma outra CPU, o conjunto de instruções dela é diferente do conjunto de instruções da CPU.

Para apresentar este novo conjunto de instruções, a abordagem será análoga ao que foi visto para a CPU principal. A diferença é que serão apresentadas várias instruções para só então apresentar alguns programas.

6.8.1 Instruções da FPU

A tabela 6.7 apresenta algumas das operações aritméticas que a FPU pode executar. Nesta tabela, `FRdest`, `FRorig` indicam registradores de ponto flutuante. Estas operações não trabalham com os registradores da tabela 6.1. Nesta tabela, FRd indica que o registrador destino é ponto flutuante, enquanto que FRo, indica que o registrador origem é um ponto flutuante. Quando houverem mais de um registrador origem, usa-se as abreviaturas FRo1 e FRo2.

A lista não é completa. Outras instruções usadas: `mul.s`, `mul.d`, `div.s`, `div.d`, etc.. Além destas, existem várias outras instruções que podem ser vistas em [Jam90].

Todas as demais instruções seguem o mesmo modelo apresentados na tabela, ou seja:

1. três operandos, sendo que o primeiro é o destino e os outros dois são a origem;
2. cada instrução tem uma versão para ponto flutuante de precisão simples (add.s, sub.s, mul.s, etc.), e uma versão para ponto flutuante de precisão dupla (add.d, sub.d, mul., etc.).

A tabela 6.8 mostra as instruções de acesso à memória. Observe que os registradores `$2` são registradores da CPU e não da FPU.

Comando	Ação	Explicação
`li.s FRd, imm`	`FRd` ← `cte`	carrega cte PF simples
`li.d FRd, imm`	`FRd` ← `cte`	carrega cte PF dupla
`add.s FRd, FRo1, FRo2`	`FRd` ← `FRo1 + FRo2`	soma PF simples
`add.d FRd, FRo1, FRo2`	`FRd` ← `FRo1 + FRo2`	soma PF dupla
`sub.s FRd, FRo1, FRo2`	`FRd` ← `FRo1 - FRo2`	subtrai PF simples
`sub.d FRd, FRo1, FRo2`	`FRd` ← `FRo1 - FRo2`	subtrai PF dupla

Tabela 6.7: Operações Aritméticas na FPU

Comando	Ação	Explicação
`l.s FRdest, cte($2)`	FRdest ← Mem[$2+cte]	carrega PF simples
`s.s FRorig, cte($2)`	Mem[$2+cte] ← FRorig	armazena PF simples
`l.d FRdest, cte($2)`	FRdest ← Mem[$2+cte]	carrega PF dupla
`s.d FRorig, cte($2)`	Mem[$2+cte] ← FRorig	armazena PF dupla

Tabela 6.8: Operações PF de acesso à memória

As instruções de comparação exigem alguma explicação, pois não seguem o modelo apresentado para a CPU.

Dentro de uma CPU é possível armazenar várias outras informações além daquelas contidas em registradores. Na FPU do SPIM existe um bit, chamado *floating point condition flag* (FPCF) que armazena o resultado da última comparação efetuada.

Assim, diferente da CPU, que usa um registrador de 32 bits para armazenar o resultado de uma comparação (conforme tabela 6.2), a FPU usa somente este bit para indicar se a última comparação efetuada foi verdadeira ou falsa.

A tabela 6.9 apresenta somente duas instruções de comparação para a FPU, aquela que verifica se o conteúdo de dois registradores é igual.

Comando	Ação	Explicação
`c.eq.d FR1, FR2`	Se FR1 = FR2, então FPCF:=1 senão FPCF:=0;	*Compare equal double*
`c.eq.s FR1, FR2`	Se FR1 = FR2, então FPCF:=1 senão FPCF:=0;	*Compare equal single*

Tabela 6.9: Operações PF de comparação

Todas as outras instruções de comparação seguem o modelo `c.OP.s FR1, FR2`, ou `c.OP.d FR1, FR2`. Neste modelo, `OP` corresponde às operações desejadas. Exemplos: `c.lt.s` (*less than*), `c.gt.s` (*greater than*), `c.le.s` (*less or equal*), `c.ge.s` (*greater or equal*), e suas equivalente para precisão dupla.

Como já foi visto nas operações equivalentes da CPU, o resultado de uma comparação pode ser usado tanto para avaliação de expressões booleanas quanto para desvios. Aqui também existem instruções específicas para desvios baseado no valor do FPCF.

Antes de abordar estas instruções, porém, veja novamente a figura 6.5. Lá constam dois co-processadores, e cada um deles tem um `condition flag`. O que vimos aqui era o *floating point condition flag*, ou *condition flag 1*. O outro *condition flag* corresponde ao co-processador 2.

As instruções de desvio analisam o valor do *condition flag* para determinar se o fluxo deve seguir ou deve mudar. As instruções de desvio só analisam se o valor é verdadeiro ou falso conforme descrito na tabela 6.10.

Comando	Explicação
`bc` *z* `t L`	branch if coprocessor *z* flag is true
`bc` *z* `f L`	branch if coprocessor *z* flag is false

Tabela 6.10: Operações PF de comparação

O *z* apresentado na tabela indica qual o *condition flag* usar. Para operações em ponto flutuante, o co-processador usado é o co-processador 1. Assim, as instruções adaptadas para PF são: `bc1t` e `bc1f`.

O próximo conjunto de instruções a ser visto é o de copia e conversões. Este conjunto está na tabela 6.11.

Comando	Explicação
`mov.s FRdest, FRorig`	copia PF simples
`mov.d FRdest, FRorig`	copia PF duplo
`cvt.s.d FRdest, FRorig`	converte PF simples para PF duplo
`cvt.d.s FRdest, FRorig`	converte PF duplo para PF simples
`cvt.w.d FRdest, FRorig`	converte inteiro para PF duplo
`cvt.w.s FRdest, FRorig`	converte inteiro para PF simples
`cvt.d.w FRdest, FRorig`	converte PF duplo para inteiro
`cvt.s.w FRdest, FRorig`	converte PF simples para inteiro
`mtc` *z* `$1, FRdest`	copia $1 para FRdest do co-processador *z*
`mfc` *z* `$1, FRorig`	copia FRorig do co-processador *z* para $1

Tabela 6.11: Operações PF para cópia

Entrada e saída ocorre de forma análoga ao que ocorre em inteiros: coloca-se o serviço desejado em `v0`, parâmetros em outros registradores, e a instrução `syscall`. A tabela completa das syscalls abordadas neste texto pode ser encontrada no apêndice B.

6.8.2 Exemplos:

O algoritmo 29 é a versão assembly do algoritmo 28.

Ao ser executado, este programa lê um número inteiro e um número real (PF de precisão simples). Em seguida, divide o número real por 2.0 o número de vezes indicado pelo primeiro argumento. Em cada iteração, imprime um número sequencial e o número real. A execução do programa no spim gera o seguinte resultado:

```
> spim
SPIM Version 7.3. of August 28, 2006
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
```

```
1  program ex1PF;
2     var n, i : integer;
3     x : real;
4  begin
5     read (x);
6     read (n);
7     for i:=1 to n do
8        begin
9        write(i);
10       write (' ');
11       writeln (x);
12       x:=x/2.0;
13    end
14 end.
```

Algoritmo 28: Programa exemplo Ponto flutuante

```
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/local/lib/exceptions.s
(spim) load "ex1PF.s"
(spim) run
3
8
0 3.00000000
1 1.50000000
2 0.75000000
3 0.37500000
4 0.18750000
5 0.09375000
6 0.04687500
7 0.02343750
8 0.01171875
(spim) quit
>
```

6.8.3 Exercícios

1. Altere o algoritmo 29 para que ele receba um terceiro argumento, o número real que é o divisor. Com isso, o programa passa a ter divisores variáveis, e não fixado em 2.0.
2. Como se declaram "variáveis" em ponto flutuante de precisão simples e dupla?
3. Traduza o programa `terco` (algoritmo 9). Crie uma versão com ponto flutuante de precisão simples, e uma de precisão dupla. Verifique nos registradores de ponto flutuante onde ocorre a perda de bits (o erro de arredondamento). Ele é maior na divisão ou na multiplicação?
4. Traduza o algoritmo 9. Releia a seção 3.5.6, e identifique o "erro" cometido pelo compilador. Como escrever o programa assembly que sempre responda "Equal"?
5. Escreva uma programa que lê um inteiro, converte-o para ponto flutuante de precisão simples e o imprime.
6. Escreva uma programa que lê um ponto flutuante de precisão simples, truca-o (em $f2) e o imprime. Aqui, o número não é enviado da FPU para a CPU.

7. Escreva uma programa que lê um ponto flutuante de precisão simples, converte-o para inteiro (em $t0) o imprime. Aqui o número é enviado da FPU para a CPU.
8. Escreva uma programa que lê um ponto flutuante de precisão simples, converte-o para ponto flutuante de precisão dupla e o imprime.
9. Escreva uma programa que lê um ponto flutuante de precisão dupla, converte-o para ponto flutuante de precisão simples e o imprime.
10. Dos exercícios de conversão acima, quais podem perder informação?
11. Escreva um programa assembly que lê um parâmetro inteiro n e n números em ponto flutuante de precisão dupla. Em seguida, o programa imprime o maior, o menor, a soma, a mediana e a média dos n números digitados.

```
1     .data
2  SPC : .asciiz
3  PL : .asciiz "\n"
4     .text
5  main:
6     li $v0, 6
7     syscall
8     li $v0, 5
9     syscall
10    move $t2, $v0
11    li $t1, 0
12 for:
13    slt $t0, $t2, $t1
14    bne $t0, $0, fimFor
15    move $a0, $t1
16    li $v0, 1
17    syscall
18    li $v0, 4
19    la $a0, SPC
20    syscall
21    li $v0, 2
22    mov.s $f12, $f0
23    syscall
24    li $v0, 4
25    la $a0, PL
26    syscall
27    li.s $f1, 2.0
28    div.s $f0,$f0,$f1
29    add $t1, $t1, 1
30    j for
31 fimFor:
32 li $v0, 10
33 syscall
```

Algoritmo 29: Programa assembly equivalente ao algoritmo 28

Parte III

Álgebra Booleana

Esta última parte da disciplina aborda um tema que será mais detadamente analisado em disciplinas que lidam com a descrição de portas lógicas, circuitos digitais entre outras.

Aqui será vista a álgebra booleana, um ramo da matemática que lida com funções cujo resultado é zero ou um.

Este ramo da matemática tem este nome em homenagem a George Boole, matemático inglês, que foi o primeiro a explicá-la como parte de um sistema de lógica em meados do século XIX. Mais especificamente, a álgebra booleana foi uma tentativa de utilizar técnicas algébricas para lidar com expressões no cálculo proposicional. Hoje, a álgebra booleana têm muitas aplicações na eletrónica. Foram pela primeira vez aplicadas a interruptores por Claude Shannon, no século XX.

Existem várias forma de abordar a álgebra boolena. Este texto é um misto da visão matemática e da visão de engenharia elétrica. O objetivo principal é capacitar o aluno para simplificar expressões booleanas (e em última instância, simplificar o projeto de circuitos digitais).

Este texto segue o modelo apresentado em livros de circuitos digitais, como [Kat94, MC00], porém qualquer livro que lide com o tema pode ser utilizado como apoio.

O tema está dividido da seguinte forma: a seção 6.9 apresenta as definições e termos usados ao longo do texto. A seção 6.10 apresenta as funções booleanas. A seção 6.11 descreve a simplificação de equações algébricas booleanas. A seção 6.12 apresenta as formas normais (mintermo e maxtermo) de expressões booleanas. A seção 6.13 apresenta os Mapas de Karnaugh, um mecanismo mais fácil para simplificar as funções booleanas.

6.9 Definições

Uma das aplicações de álgebra booleana é a construção de circuitos lógicos. Quando são usados neste contexto, é comum chamar a álgebra booleana de lógica binária.

A lógica binária lida com variáveis que podem assumir somente dois valores e com operações lógicas sobre estas variáveis. Os valores que as variáveis binárias podem assumir recebem nomes diferentes dependendo do contexto em que são usadas:

- na matemática, os valores são chamados V e F (para Verdadeiro e Falso).
- na construção de circuitos lógicos podem ser chamados de “com corrente elétrica” ou “sem corrente elétrica”.
- neste texto, serão chamados de 1 e 0. Porém tudo o que for visto aqui é aplicável nas outras áreas também.

As variáveis binárias serão representadas com letras maiúsculas (A, B, X, Y, etc.).

6.10 Funções Booleanas

As funções booleanas que serão tratadas neste texto são `and`, `or`, `not`. Porém existem outras como o `xor`, que não serão tratadas aqui. A seguir, são apresentadas as várias formas de se representar estas funções e quais os valores que elas geram como resultado.

6.10.1 A função AND

Algebricamente, existem três formas de representar a função booleana AND:

$$A = X\ AND\ Y$$
$$A = X \cdot Y$$
$$A = XY$$

Conforme descrito nas equações acima, a função booleana AND recebe como entrada duas variáveis lógicas (ou valores lógicos) e retorna um valor lógico. O valor retornado depende dos valores de entrada conforme apresentado na tabela 6.12.

Observe que a função AND se comporta como a multiplicação (daí o motivo das equações acima lembrarem multiplicação).

Uma outra forma de ver a função AND é como uma porta lógica usada para construir circuitos integrados. Neste contexto, ela é representada como indicado na figura 6.6

X	Y	A
0	0	0
0	1	0
1	0	0
1	1	1

Tabela 6.12: A tabela verdade da função AND

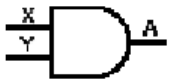


Figura 6.6: Porta lógica AND

6.10.2 A função OR

Existem duas formas de representar a função booleana OR:

$$A = X \ OR \ Y$$
$$A = X + Y$$

Conforme descrito nas equações acima, a função booleana OR recebe como entrada duas variáveis lógicas (ou valores lógicos) e retorna um valor lógico. O valor retornado depende dos valores de entrada conforme apresentado na tabela 6.13.

Observe que a função OR se comporta como a soma (daí o motivo das equações acima lembrarem multiplicação). A excessão é que $1+1 \neq 1$, mas vamos fazer de conta que é igual.

X	Y	A
0	0	0
0	1	1
1	0	1
1	1	1

Tabela 6.13: A tabela verdade da função OR

Uma outra forma de ver a função OR é como uma porta lógica usada para construir circuitos integrados. Neste contexto, ela é representada como indicado na figura 6.7

6.10.3 A função NOT

Existem várias formas de representar a função booleana NOT (também chamada de complemento). As duas principais são:

$$A = \overline{X}$$
$$A = X'$$

Figura 6.7: Porta lógica OR

Conforme descrito nas equações acima, a função booleana NOT (também chamado de complemento de função ou inversor) recebe como entrada uma variável lógica (ou valor lógico) e retorna um valor lógico. O valor retornado depende dos valores de entrada conforme apresentado na tabela 6.14.

Matematicamente, a função NOT pode ser vista como $A = 1 - X$.

X	A
1	0
0	1

Tabela 6.14: A tabela verdade da função NOT

Uma outra forma de ver a função NOT é como uma porta lógica usada para construir circuitos integrados. Neste contexto, ela é representada como indicado na figura 6.8.

Figura 6.8: Porta lógica NOT

6.11 Equações Algébricas

As funções booleanas apresentadas na seção 6.10 podem ser combinadas e usadas para construir circuitos lógicos. Porém, é importante usar a menor quantidade de portas lógicas para construir circuitos, pois quanto menor a quantidade, menor a temperatura do circuito e provavelmente mair rápido será o circuito.

Neste contexto, é imprescidível saber simplificar uma equação lógica. Esta seção aborda o processo de simplificação. Porém, antes de inicar a descrição de como simplificar, a seção 6.11.1 apresenta as identidades básicas. Estas identidades básicas merecem destaque pois são muito utilizadas para simplificar equações booleanas.

A seção 6.11.2 apresenta um mecanismo para demonstrar a igualdade (ou não) de equações algéricas e finalmente a seção 6.11.3 apresenta como manipular algebricamente equações booleanas.

6.11.1 Identidades Básicas

$$X + 0 = X \tag{6.1}$$
$$X + 1 = 1 \tag{6.2}$$
$$X + X = X \tag{6.3}$$
$$X + \overline{X} = 1 \tag{6.4}$$
$$\overline{\overline{X}} = X \tag{6.5}$$
$$X \cdot 1 = X \tag{6.6}$$
$$X \cdot 0 = 0 \tag{6.7}$$
$$X \cdot X = X \tag{6.8}$$
$$X \cdot \overline{X} = 0 \tag{6.9}$$
$$X + Y = Y + X \tag{6.10}$$
$$X + (Y + Z) = (X + Y) + Z \tag{6.11}$$
$$X(Y + Z) = XY + XZ \tag{6.12}$$
$$\overline{X + Y} = \overline{X} \cdot \overline{Y} \tag{6.13}$$
$$XY = YX \tag{6.14}$$
$$X(YZ) = (XY)Z \tag{6.15}$$
$$X + YZ = (X + Y)(X + Z) \tag{6.16}$$
$$\overline{XY} = \overline{X} + \overline{Y} \tag{6.17}$$

As nove identidades iniciais (equações 6.1 até 6.9) envolvem somente uma variável, e são facilmente comprovadas substituindo os valores das variáveis por `0` e `1`.

As oito equações seguintes envolvem duas ou três variáveis, e recebem nomes especiais.

Comutativas Equações 6.10 e 6.14.

Associativas Equações 6.11 e 6.15.

Distributivas Equações 6.12 e 6.16.

DeMorgan Equações 6.13 e 6.17.

As identidades comutativas, associativas e distributivas não precisam explicações, uma vez que já são conhecidas (bem, pelo menos era para serem). Ainda sim, a equação 6.16 está num formato um pouco diferente do habitual. Normalmente, este tipo de equação é apresentada "invertida", ou seja, $(X + Y)(X + Z) = X + YZ$. O formato apresentado torna-a uma equação até "diferente", porém é importante frisar que como é uma igualdade, tanto faz quem vai do lado direito da equação e quem vai do lado esquerdo. Como exemplo, sabemos que $X = X$. Porém, algumas identidades básicas também são iguais a X. Substituindo a equação 6.3 no X à esquerda e 6.8 no X à direita, temos que $X + X = X \cdot X$. Este processo é o inverso do habitual. Ao invés de "diminuir" o número de caracteres da equação, aumentamos. Em vários casos, isso será bastante freqüente. Aliás, Isto fica mais interessante quando se fala de equações de duas variáveis.

Uma última observação é que estas identidades se referem a equações com até três variáveis. Porém, é possível usá-las em equações com mais variáveis. Por exemplo, a expressão $(A+B)(A+CD)$ envolve quatro variáveis, e se assemelha à equação 6.16. Considere esta equação, e substitua X por A, Y por B e Z por CD. Após esta substituição, podemos ver que:

$$(A + B)(A + CD) = A + BCD$$

As identidades DeMorgan são as mais difíceis de aceitar. Nestes casos, é necessária uma prova matemática formal. As próximas seções apresentam mecanismos para provas formais.

Todas estas equações (ou identidades básicas) são verdadeiras. Algumas são mais simples, como as equações 6.1 e 6.2. Outras, porém, são mais difíceis de "acreditar". A questão é como ter certeza que estas equações, ou melhor, que qualquer equação booleana é ou não é verdadeira?

Neste texto, serão abordados três mecanismos: tabela verdade (seção 6.11.2), manipulação algébrica (seção 6.11.3) e mapa de Karnaugh (seção 6.13).

6.11.2 Tabelas Verdade

Para provar que uma igualdade é verdadeira usando tabela verdade, basta colocar uma tabela com todos os valores válidos para cada variável. Como cada variável só pode assumir os valores `0` e `1`, a tabela pode ser uma boa alternativa. O tamanho da tabela depende do número de variáveis. Com uma variável, a tabela tem duas linhas, com duas variáveis, quatro linhas, com n variáveis, 2^n linhas.

Como exemplo, considere a equação 6.13 ($\overline{X+Y} = \overline{X} \cdot \overline{Y}$). Esta equação tem duas variáveis, e por isso terá quatro linhas. Os dois lados da equação são mostrados nas tabelas 6.15 e 6.16.

As tabelas foram construídas da seguinte forma: primeiro, nas colunas da esquerda, são colocadas as variáveis e em cada linha da tabela recebe todas as combinações possíveis de valores destas variáveis. No exemplo, as variáveis X e Y podem assumir os valores 00 (primeira linha), 01 (segunda linha), 10 terceira linha) e 11 (quarta linha).

As demais colunas variam de acordo com a necessidade. Na tabela 6.15, pareceu mais claro primeiro fazer a operação $X+Y$ (terceira coluna) e depois negar o resultado na quarta coluna obtendo o que era desejado ($\overline{X+Y}$).

Na tabela 6.16, primeiro foi calculado o valor das variáveis X e Y negadas para só então calcular o resultado final de $\overline{X} \cdot \overline{Y}$.

Observe que é possível também combinar as duas tabelas em uma só aumentando o número de colunas. Por exemplo, as duas primeiras colunas com X e Y as duas seguintes com $\overline{X}$ e $\overline{Y}$. As coluna seguinte com $\overline{X+Y}$ e a última com $\overline{X} \cdot \overline{Y}$.

X	Y	X+Y	$\overline{X+Y}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

Tabela 6.15: A tabela verdade de $\overline{X+Y}$

X	Y	$\overline{X}$	$\overline{Y}$	$\overline{X} \cdot \overline{Y}$
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

Tabela 6.16: A tabela verdade de $\overline{X} \cdot \overline{Y}$

É importante destacar que as equações DeMorgan também podem ser estendidas para mais variáveis. A fórmula geral é:

$$\overline{X_1 + X_2 + \cdots + X_n} = \overline{X_1} \cdot \overline{X_2} \ldots \overline{X_n}$$

Como exercício, monte as tabelas verdade de todas as identidades básicas.

Este tipo de verificação é uma prova formal matemática, e pode ser comparada ao uso da força bruta (ou seja, todas as alternativas são testadas) e só é viável quando as variáveis são em pouca quantidade e binárias. Se as variáveis fossem inteiros, não seria possível provar usando a tabela (afinal, cada variável inteira pode assumir uma quantidade infinita de valores). Com mais variáveis, digamos 10, então o número de linhas ficaria muito grande ($2^{10} = 1024$), e também seria impraticável.

Nestes casos, é melhor trabalhar com manipulação algébrica.

6.11.3 Manipulação Algébrica

Considere seguinte equação 6.18. O seu circuito lógico é apresentado na figura **??**. Observe que o circuito é composto por três portas `AND`, duas portas `NOT` e uma porta `OR`. Apesar de ser pequena, esta equação (ou este circuito) pode ser simplificado. Os motivos já foram colocados anteriormente: menor custo e maior eficiência.

$$F = \overline{X}YZ + \overline{X}Y\overline{Z} + XZ \tag{6.18}$$

O processo de simplificação utiliza as identidades básicas como demonstrado nas equações 6.19 até 6.22.

$$\begin{aligned} F &= \overline{X}YZ + \overline{X}Y\overline{Z} + XZ && (6.19)\\ &= \overline{X}Y(Z + \overline{Z}) + XZ && (6.20)\\ &= \overline{X}Y \cdot 1 + XZ && (6.21)\\ &= \overline{X}Y + XZ && (6.22) \end{aligned}$$

A identidade básica para simplificar a equação 6.19 e obter a equação 6.20 foi a identidade básica 6.12. Para obter a equação 6.21 foi usada a identidade básica 6.4. Por fim, para obter a equação 6.22 foi usada a identidade básica 6.6.

As duas funções 6.18 e 6.22 geram os mesmos resultados para as mesmas entradas, mas a equação 6.22 usa menor quantidade de portas lógicas. Como exercício, construa as tabelas verdade para as duas funções e com isso prove que

$$\overline{X}YZ + \overline{X}Y\overline{Z} + XZ = \overline{X}Y + XZ$$

6.11.3.1 Outras Identidades

Existem várias outras identidades que podem ser usadas. Esta seção apresenta algumas destas identidades, e além disso, servem também como demonstração de como simplificar equações. Verifique quais identidades básicas foram utilizadas em cada caso.

$$X + XY = X(1 + Y) = X \tag{6.23}$$
$$XY + X\overline{Y} = X(Y + \overline{Y}) = X \tag{6.24}$$
$$X + \overline{X}Y = (X + \overline{X})(X + Y) = X + Y \tag{6.25}$$
$$X(X + Y) = X + XY = X \tag{6.26}$$
$$X(\overline{X} + Y) = X\overline{X} + XY = XY \tag{6.27}$$

Uma outra equação muito útil é o chamado “teorema do consenso” (equação 6.28). Ela indica que o último termo é redundante, e pode ser eliminado.

$$XY + \overline{X}Z + YZ = XY + \overline{X}Z \tag{6.28}$$

Para utilizar o teorema do consenso, veja que o último termo YZ tem Y associado ao X no primeiro termo, e Z associado ao $\overline{X}$ no segundo termo. Todas as vezes em que isso ocorre, é viável usar o teorema do consenso.

Não é fácil de entender porque há a igualdade, mas um bom caminho para começar é através da tabela verdade (que é deixado como exercício). Porém aqui apresentamos a prova através de manipulação algébrica.

$$\begin{aligned} XY + \overline{X}Z + YZ &= XY + \overline{X}Z + YZ(X + \overline{X})\\ &= XY + \overline{X}Z + XYZ + \overline{X}YZ\\ &= XY + XYZ + \overline{X}Z + \overline{X}YZ\\ &= XY(1 + Z) + \overline{X}Z(1 + Y)\\ &= XY + \overline{X}Z \end{aligned}$$

A manipulação não é intuitiva porque, mais uma vez, ao invés de “diminuir” o tamanho da equação, primeiro aumenta-se para só ao final conseguir as simplificações. Conseguir fazer este tipo de simplificação exige que o interessado faça muitos exercícios.

6.11.4 Complemento de Função

Seja F uma função booleana. Em alguns circuitos, assim como em algumas equações, é interessante obter o complemento da função, e obter a função $\overline{F}$. Para tal, deve-se aplicar DeMorgan quantas vezes for necessário.

Como exemplo, considere a função

$$F_1 = \overline{X}\,Y\overline{Z} + \overline{XY}Z$$

Para inverter uma função, usa-se DeMorgan quantas vezes quanto necessário.

$$\begin{aligned}\overline{F_1} &= \overline{\overline{X}\,Y\overline{Z} + \overline{XY}Z} \\ &= (\overline{\overline{X}\,Y\overline{Z}}) \cdot (\overline{\overline{XY}Z}) \\ &= (X + \overline{Y} + Z)(Z + Y + \overline{Z})\end{aligned}$$

Para reforçar, apresentamos um segundo exemplo.

$$\begin{aligned}F_2 &= X(\overline{Y} \cdot \overline{Z} + YZ) \\ \overline{F_2} &= \overline{X(\overline{Y} \cdot \overline{Z} + YZ)} \\ &= \overline{X} + \overline{(\overline{Y} \cdot \overline{Z} + YZ)} \\ &= \overline{X} + (\overline{\overline{YZ}}) \cdot (\overline{YZ}) \\ &= \overline{X} + (Y + Z)(\overline{Y} + \overline{Z})\end{aligned}$$

6.11.4.1 O Dual de uma função

Uma outra forma de se obter o complemento de uma função obter o seu dual. Para tal, aplica-se a "generalização de DeMorgan":

1. Trocar todos os AND por OR e todos os OR por AND na função F.
2. trocar todos os "1" por "0" e todos os "0" por "1" na função F.
3. complementar cada literal.

Exemplo:

$$\begin{aligned}F_3 &= \overline{X}Y\overline{Z} + \overline{X} \cdot \overline{Y}Z \\ &= (\overline{X}Y\overline{Z}) + (\overline{X} \cdot \overline{Y}Z) && (6.29) \\ Dual(F_3), Passo1 &= (\overline{X} + Y + \overline{Z})(\overline{X} + \overline{Y} + Z) && (6.30) \\ Dual(F_3), Passo2 &= (X + \overline{Y} + Z)(X + Y + \overline{Z}) && (6.31) \\ \overline{F_3} &= (X + \overline{Y} + Z)(X + Y + \overline{Z}) && (6.32)\end{aligned}$$

Observe o desenvolvimento da operação. A equação 6.29 é a função F_3 com parênteses. Ela não é necessária, mas é útil para obter a equação 6.30.

O trabalho começa na passagem da equação 6.29 para a equação 6.30. Nest passagem é aplicado o primeiro passo do dual, ou seja, troca de todos os AND por OR e de todos os OR por AND. O segundo passo do dual está indicado na equação 6.32, onde foi aplicado o complemento de cada literal. Esta equação é $\overline{F_3}$.

As tabelas verdades de F_3 e $\overline{F_3}$ são apresentadas a seguir:

X	Y	Z	$\overline{X}Y\overline{Z}$	$\overline{X} \cdot \overline{Y}Z$	F_3	$X + Y + \overline{Z}$	$X + \overline{Y} + Z$	$\overline{F_3}$
0	0	0	0	0	0	1	1	1
0	0	1	0	1	1	0	1	0
0	1	0	1	0	1	1	0	0
0	1	1	0	0	0	1	1	1
1	0	0	0	0	0	1	1	1
1	0	1	0	0	0	1	1	1
1	1	0	0	0	0	1	1	1
1	1	1	0	0	0	1	1	1

6.11.5 Exercícios

1. Demonstre as identidades abaixo usando tabelas verdade.
   (a) $\overline{XYZ} = \overline{X} + \overline{Y} + \overline{Z}$
   (b) $X + YZ = (X + Y)(X + Z)$
   (c) $\overline{X}Y + \overline{Y}Z + X\overline{Z} = X\overline{Y} + Y\overline{Z} + \overline{X}Z$
   (d) $XY + \overline{X}Z + YZ = XY + \overline{X}Z$
2. Demonstre as identidades abaixo usando manipulação algébrica
   (a) $\overline{X} \cdot \overline{Y} + \overline{X}Y + XY = \overline{X} + Y$
   (b) $\overline{A}B + \overline{B} \cdot \overline{C} + AB + \overline{B}C = 1$
   (c) $Y + \overline{X}Z + X\overline{Y} = X + Y + Z$
   (d) $\overline{X} \cdot \overline{Y} + \overline{Y}Z + XZ + XY + Y\overline{Z} = \overline{X} \cdot \overline{Y} + XZ + Y\overline{Z}$
   (e) $A\overline{B} + \overline{A} \cdot \overline{C} \cdot \overline{D} + \overline{A} \cdot \overline{B}D + \overline{ABCD} = \overline{B} + \overline{A} \cdot \overline{C} \cdot \overline{D}$
   (f) $XY + W\overline{Y} \cdot \overline{Z} + \overline{W}Y\overline{Z} + W\overline{X}\overline{Z} = XZ + W\overline{Y} \cdot \overline{Z} + WX\overline{Y} + \overline{W}XY + \overline{X}Y\overline{Z}$
   (g) $CD + A\overline{B} + AC + \overline{A} \cdot \overline{C} + \overline{A}B + \overline{C} \cdot \overline{D} = (\overline{A} + \overline{B} + C + \overline{D})(A + B + \overline{C} + D)$
3. Sabendo que $AB = 0$ e que $A + B = 1$, demonstre a identidade abaixo usando manipulação algébrica.

$$AC + \overline{A}B + BC = B + C$$

4. Simplifique as expressões booleanas abaixo para o menor número de literais.
   (a) $ABC + AB\overline{C} + \overline{A}B$
   (b) $(\overline{A + B})(\overline{A} + \overline{B})$
   (c) $\overline{A}BC + AC$
   (d) $BC + B(AD + A\overline{D})$
   (e) $(A + \overline{B} + A\overline{B})(AB + \overline{A}C + BC)$
   (f) $\overline{W}X(\overline{Z} + \overline{Y}Z) + X(W + \overline{W}YZ)$
5. Encontre os complementos das seguintes funções:
   (a) $A\overline{B} + \overline{A}B$
   (b) $(\overline{V}W + X)Y + \overline{Z}$
   (c) $WX(\overline{Y}Z + Y\overline{Z}) + \overline{W} \cdot \overline{X}(\overline{Y} + Z)(Y + \overline{Z})$
   (d) $(A + \overline{B} + C)(\overline{A} \cdot \overline{B} + C)(A + \overline{B} \cdot \overline{C})$
6. Simplifique as funções abaixo:
   (a) $f(W,X,Y,Z) = X + XYZ + \overline{X}YZ + \overline{X}Y + WX + \overline{W}X$
   (b) $f(X,Y,Z) = (X + Y)(\overline{X} + Y + Z)(\overline{X} + Y + \overline{Z})$
   (c) $f(A,B,C,D) = (AD + \overline{A}C)(\overline{B}(C + B\overline{D}))$

6.12 Formas Normais

Funções booleanas podem ser escritas de várias formas diferentes. Porém existem alguns formatos que são "padronizados". Estes formatos são chamados de formas normais.

As formas normais contém termos produto (por exemplo ABC , $X\overline{Y}Z$) e termos soma (por exemplo $A + B + C$, $X + \overline{Y} + Z$). Quando as funções são representadas nestas formas, é até mais simples desenhar o circuito.

Existem duas formas normais importantes a serem conhecidas: os mintermos, apresentados na seção 6.12.1 e os maxtermos, apresentados na seção 6.12.1.

6.12.1 Mintermos

Considere uma função booleana com *n* variáveis. Um mintermo é um produto contendo todas as *n* variáveis exatamente uma vez (complementada ou não).

Para duas variáveis, existem quatro mintermos. Por exemplo, para as variáveis X e Y, os quatro mintermos estão indicados na tabela 6.17.

Termo Produto	mintermo
$\overline{X} \cdot \overline{Y}$	m_0
$\overline{X} \cdot Y$	m_1
$X \cdot \overline{Y}$	m_2
$X \cdot Y$	m_3

Tabela 6.17: Mintermos para duas variáveis

A tabela 6.18 apresenta todos os mintermos para três variáveis.

Termo Produto	mintermo
$\overline{X} \cdot \overline{Y} \cdot \overline{Z}$	m_0
$\overline{X} \cdot \overline{Y} \cdot Z$	m_1
$\overline{X} \cdot Y \cdot \overline{Z}$	m_2
$\overline{X} \cdot Y \cdot Z$	m_3
$X \cdot \overline{Y} \cdot \overline{Z}$	m_4
$X \cdot \overline{Y} \cdot Z$	m_5
$X \cdot Y\overline{Z}$	m_6
$X \cdot Y \cdot Z$	m_7

Tabela 6.18: Mintermos para três variáveis

Os mintermos são utilizados também para representar funções, como pode ser visto a seguir:

$$\begin{aligned} F(X,Y,Z) &= \overline{X} \cdot \overline{Y} \cdot \overline{Z} + \overline{X}Y\overline{Z} + X\overline{Y}Z + XYZ \\ F(X,Y,Z) &= m_0 + m_2 + m_5 + m_7 \\ F(X,Y,Z) &= \Sigma m(0,2,5,7) \end{aligned}$$

O símbolo Σ é chamado "somatório". Neste caso, somatório indica operação `OR` entre os mintermos $m_0 + m_2 + m_5 + m_7$. Em outras palavras, um somatório de termos produto.

6.12.2 Maxtermos

Considere uma função booleana com *n* variáveis. Um maxtermo é uma soma contendo todas as variáveis extamente uma vez.

A tabela 6.19 apresenta todos os mintermos para três variáveis.

Observe que enquanto o mintermo m_0 tem todos os termos invertidos, o maxtermo M_0 tem todos os termos sem inversão. Isto ajuda na obtenção de duais. Por exemplo, considere a função $F(X,Y,Z) = \overline{X}YZ$.

Para obter o dual da função, temos:

$$\begin{aligned} F &= \overline{X}YZ = m_3 \\ \overline{F} &= \overline{\overline{X}YZ} \\ &= X + \overline{Y} + \overline{Z} \\ &= M_3 \end{aligned}$$

Termo Produto	mintermo
$X+Y+Z$	M_0
$X+Y+\overline{Z}$	M_1
$X+\overline{Y}+\overline{Z}$	M_2
$X+\overline{Y}+Z$	M_3
$\overline{X}+Y+Z$	M_4
$\overline{X}+Y+\overline{Z}$	M_5
$\overline{X}+\overline{Y}+Z$	M_6
$\overline{X}+\overline{Y}+\overline{Z}$	M_7

Tabela 6.19: Maxtermos para três variáveis

Em outras palavras, $M_j = \overline{m_j}$.
Agora, vamos examinar uma função com mais termos:

$$\begin{aligned}
F &= \Sigma(1,3,4,6) \\
F &= m_1 + m_3 + m_4 + m_6 \\
\overline{F} &= \overline{m_1 + m_3 + m_4 + m_6} \\
\overline{F} &= \overline{m_1} + \overline{m_3} + \overline{m_4} + \overline{m_6} \\
\overline{F} &= M_1 \cdot M_3 \cdot M_4 \cdot M_6 \\
\overline{F} &= (X+Y+\overline{Z}) \cdot (X+\overline{Y}+Z) \cdot (\overline{X}+Y+Z) \cdot (\overline{X}+\overline{Y}+Z) \\
\overline{F} &= M_1 \cdot M_3 \cdot M_4 \cdot M_6 \\
&= \Pi\, M(1,3,4,6)
\end{aligned}$$

Enquanto o símbolo Σ é utilizado para indicar um somatório, o símbolo Π é utilizado para indicar um produtório. No caso, um produtório de maxtermos, ou ainda um produto de somas.

6.12.2.1 Exercícios

1. Obtenha a tabela verdade das seguintes expressões, e as reescreva como soma-de-produtos e produto-de-somas.
   (a) (XY+Z)(Y+XZ)
   (b) $(\overline{A}+B)(\overline{B}+C)$
   (c) $WX\overline{Y}+WX\overline{Z}+WXZ+Y\overline{Z}$
2. Converta as expressões abaixo em soma-de-produtos e em produto-de-somas.
   (a) $(AB+C)(B+\overline{C}D)$
   (b) $\overline{X}+X(X+\overline{Y})(Y+\overline{Z})$
   (c) $(A+B\overline{C}+CD)(\overline{B}+EF)$

6.13 Mapa de Karnaugh

Uma outra forma de simplificar equações booleanas é o mapa de Karnaugh. A grande vantagem deste método quando comparado com a manipulação algébrica, é que aqui tudo ocorre "visualmente". Por isso mesmo é o método preferido.

A idéia é transformar uma equação algébrica em uma tabela onde as linhas e colunas representam uma variáveis. Como exemplo, considere a expressão XY.

O mapa de Karnaugh equivalente é o seguinte:

f(X,Y):

X \ Y	0	1
0	0	0
1	0	1

O mapa que segue pode ser compreendido mais facilmente como contendo todos os mintermos para duas variáveis, ou seja:

f(X,Y):

X \ Y	0	1
0	$\overline{X}\cdot\overline{Y}$	$\overline{X}\cdot Y$
1	$X\cdot\overline{Y}$	$X\cdot Y$

f(X,Y):

X \ Y	0	1
0	m_0	m_1
1	m_2	m_3

Como a expressão indicada era XY, e este é exatamente o mintermo m_3, todos os espaços destinados a outros mintermos foram assinalados como `0`, enquanto que o espaço destinado ao mintermo m_3 foi assinalado como `1`. Em outras palavras, um mapa de Karnaugh corresponde a uma expressão booleana, onde todos os espaços indicados com `0` não fazem parte da expressão e todos os espaços indicados com `1` fazem parte.

Cada espaço do mapa contém uma combinação de literais. A primeira linha contém sempre o literal $\overline{X}$, enquanto que a segunda linha contém o literal X. De maneira análiga, a primeira coluna contém sempre o literal $\overline{Y}$, enquanto que a segunda coluna contém sempre o literal Y.

Observe que isso está indicado no mapa. Os X e Y que estão fora do quadrado indicam os locais onde os valores são X e Y. Os valores de $\overline{X}$ e $\overline{Y}$ não estão indicados.

X \ Y	00	01
00	00	01
10	10	11

Não é difícil de perceber que o número de configurações possíveis para um map de Karnaugh são finitas. Para duas variáveis, 16 combinações - 2^k, onde k é o número de quadrados. Por exemplo, para duas variáveis, temos:

- um mapa com todos os quadrados iguais a zero;
- quatro mapas com três quadrados iguais a zero e um quadrado igual a um;
- seis mapas com dois quadrados iguais a um e dois iguais a zero;
- quatro mapas com quatro quadrados iguais a um e um quadrado igual a zero;
- um mapa com quatro quadrados iguais a um.

Algumas funções algébricas, porém, não são simples de colocar no mapa. Por exemplo, considere a expressão $F = X + Y$. Como ela não é composta de mintermos, parece que não pode ser representada em um mapa de Karnaugh. Porém, existe uma outra expressão, equivalente a ela, que só usa mintermos:

$$F = X + Y \tag{6.33}$$
$$F = X(Y + \overline{Y}) + Y(X + \overline{X}) \tag{6.34}$$
$$F = XY + X\overline{Y} + YX + \overline{X}Y \tag{6.35}$$
$$F = XY + X\overline{Y} + \overline{X}Y \tag{6.36}$$
$$F = m_3 + m_2 + m_1 \tag{6.37}$$

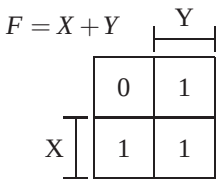
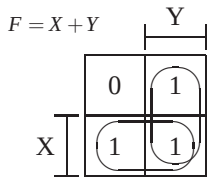


Figura 6.9: Mapar de Karnaugh para $F = X + Y$

Agora, fica fácil de entender o que representa o mapa de Karnaugh da figura 6.9

Observe que neste mapa, toda a linha X está ocupada, assim como toda a coluna Y. É por isso que podemos concluir que o mapa indica a expressão $F = X + Y$.

Normalmente é mais fácil de verificar isso anotando o mapa:



Neste mapa, observe que a entrada m_3 é usada duas vezes. Isto é válido, pois cada entrada não é exclusiva para uma anotação. Em outras palavras, cada quadrado livre pode ser combinado com quadrados já ocupados.

Para esclarecer o que ocorre quando são combinados quadrados já utilizados, imagine o mapa cheio de uns, o que corresponde a $F = \overline{XY}$, $\overline{X}Y$, $X\overline{Y}$, XY. A simplificação correta implica que $F = 1$ (ou seja, é verdadeiro independente dos valores de X e Y). Porém, dependendo da forma de anotar o mapa, podemos ter várias outras soluções, como $F = X + \overline{X}$, $F = Y + \overline{Y}$, entre outras. Porém, o resultado mais simplificado é $F = 1$.

6.13.1 Mapa de Karnaugh de três variáveis

Quando estendemos o mapa de Karnaugh para três variáveis, surge um problema: como fazer para representar três variáveis em um mapa com duas dimensões. A solução é colocar, ou nas linhas, ou nas colunas, uma combinação de duas variáveis, como indicado na figura 6.10.

A figura é composta de três partes:

1. Canto superior esquerdo: Estão indicados os mintermos.
2. Canto superior direito: Estão indicados as expressões dos mintermos.
3. Embaixo: a mesma coisa que no canto superior direito, só que em binário. Quando o valor é `1`, indica a variável. Quando é `0`, indica a variável invertida. Assim, o mintermo m_3 (canto superior esquerdo) está associado à expressão $\overline{X}YZ$, e também a 0 11 (zero indica $\overline{X}$, um indica Y e o último um indica Z.

Observe como o mapa de Karnaugh de três variáveis é organizado. A primeira linha contém sempre algum termo $\overline{X}$ enquanto que a segunda linha sempre contém algum termo X. Não há nenhuma entrada contendo $\overline{X}$ e X.

A primeira coluna contém somente termos $\overline{YZ}$, a segunda coluna contém somente termos $\overline{Y}Z$, a terceira somente termos YZ e finalmente a quarta linha contém somente termos $Y\overline{Z}$. Esta relação é mostrada na parte de baixo da figura 6.10, onde há um pequeno espaço entre o primeiro dígito e os outros dois. O primeiro dígito sempre representa a linha equanto que os outros dois sempre representam a coluna. Dependendo do tamanho da tabela (número de variáveis), a sua montagem é mais intuitiva através da representação binária

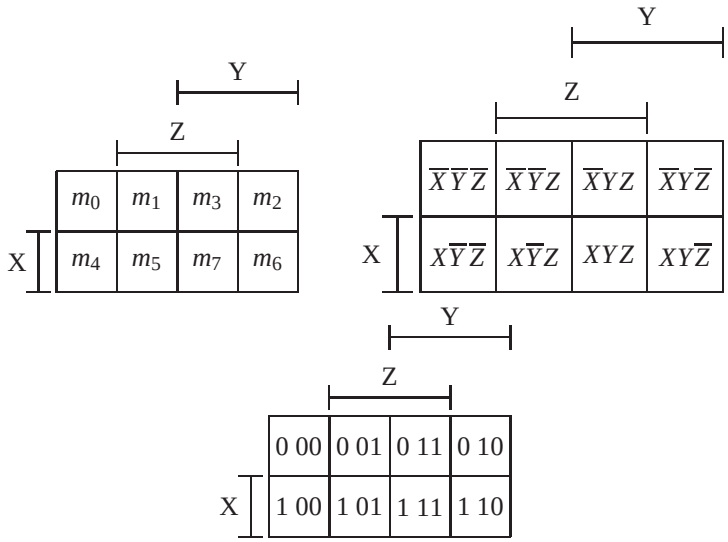


Figura 6.10: Mapa de Karnaugh para $F(XYZ)$

.

do que indicando quais são as variáveis. Porém, é necessário prestar atenção à ordem das colunas: 00, 01, 11, 10. Confira na figura 6.10.

A figura mostra algo incomum: a ordem em que aparecem os mintermos não é a ordem seqüencial (m_0,m_1,m_2,m_3,m_4,m_5,m_6,m_7). Isto complica a vida de muitas pessoas, mas a idéia é ajudar na simplificação (apesar de não parecer ajudar nada agora).

Observe que o desenho do mapa indica onde estão os valores de X (uma linha), Y (duas colunas) e Z (duas colunas).

Vamos agora abordar algumas questões práticas sobre mapas de Karnaugh de três variáveis antes de prosseguir. Para isso, considere o mapa de Karnaugh da figura 6.11. Este mapa pode ser indicado de várias formas diferentes, porém a mais simples é usar a notação dos minitermos. Este mapa tem os minitermos m_5 e m_7 ativos. Sendo assim, podemos dizer que ele é:

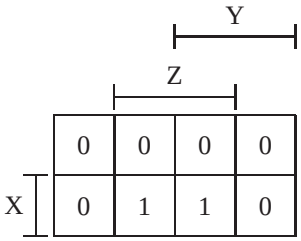


Figura 6.11: Mapa de Karnaugh para $F = X\overline{Y}Z + XYZ$

.

1. $F(X,Y,Z) = X\overline{Y}Z + XYZ$

2. $F(X,Y,Z) = m_5, m_7$
3. $F(X,Y,Z) = \Sigma m(5,7)$

6.13.1.1 Simplificação de funções booleanas usando o mapa

Para explicar como simplificar funções booleanas usando o mapa, considere a expressão booleana

$$F(X,Y,Z) = \Sigma m(2,3,4,5)$$

.

Levada ao mapa, esta expressão é indicada na figura 6.12.

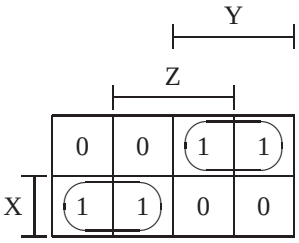


Figura 6.12: Mapa de Karnaugh para $F = \Sigma m(2,3,4,5)$

.

Observe que a expressão foi anotada com dois retângulos (meio ovais, mas faz de conta que são retângulos). Estas anotações são chamadas de retângulos (por motivos óbvios), e devem ser anotadas todas as vezes que forem encontrados `1` (aqui denominados de quadrados) juntos em linhas ou colunas.

Agora, veja que o retângulo da linha de cima corresponde à equação

$$\overline{X}YZ + \overline{X}Y\overline{Z} = \overline{X}Y(Z + \overline{Z}) = \overline{X}Y$$

Olhando para o mapa da figura 6.12, é possível deduzir o resultado olhando para as linhas e colunas. Como a primeira linha é a linha do $\overline{X}$, então o retângulo contém, em ambos os termos, o literal $\overline{X}$. Logo, ele não pode ser simplificado.

O retângulo também ocupa ambos os espaços destinados a Y, o que indica que Y não pode ser simplificado.

Por fim, observe que um dos termos do retângulo contém $\overline{Z}$ e o outro contém Z. Isto indica que este termo pode ser eliminado. Com isso, $F = \Sigma m(2,3) = \overline{X}Y$.

Baseado nas explicações acima, não é difícil de deduzir o resultado do outro retângulo: $F = \Sigma m(4,5) = X\overline{Y}$.

Logo, $F = \Sigma m(2,3,4,5) = X\overline{Y} + \overline{X}Y$

A idéia básica aqui é que, em uma mapa de Karnaugh com três variáveis, um retângulos compostos de dois quatrados indica que um dos literais pode ser simplificado.

Agora, serão apresentados alguns exemplos para reforçar o método:

- $F = \Sigma m(1,5) = \overline{Y}Z$ - figura 6.13
- $F = \Sigma m(1,3,5) = \overline{Y}Z + \overline{X}Z$ - figura 6.14

Se o mapa de Karnaugh tiver retângulos de quatro variáveis, então duas variáveis podem ser simplificadas. Para exemplificar, considere o mapa para $F = \Sigma m(0,1,2,3)$, (figura 6.15). A função contida neste mapa é

$$\begin{aligned} F = \Sigma m(0,1,2,3) &= \overline{X},\overline{Y}\cdot\overline{Z} + \overline{X}\cdot\overline{Y}Z + \overline{X}Y\overline{Z} + \overline{X}YZ \\ &= \overline{X}\cdot\overline{Y}(\overline{Z}+Z) + \overline{X}Y(\overline{Z}+Z) \\ &= \overline{X}\cdot\overline{Y} + \overline{X}Y \\ &= \overline{X}(\overline{Y}+Y) \\ &= \overline{X} \end{aligned}$$

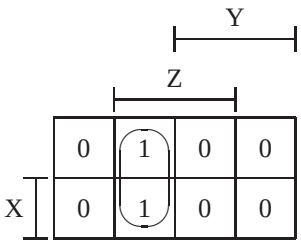


Figura 6.13: Mapa de Karnaugh para $F = \Sigma m(1,5)$

.

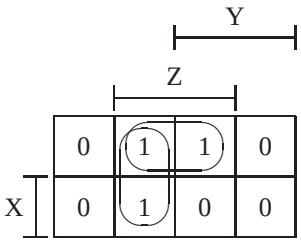


Figura 6.14: Mapa de Karnaugh para $F = \Sigma m(1,3,5)$

.

Observe novamente o mapa da figura 6.15. Deste mapa é possível deduzir o resultado simplesmente observando que a única variável que não "muda" é $\overline{X}$, ou seja, a primeira linha.

Mapas que correspondem às equações $F(XYZ) = Y$ e $F(XYZ) = Z$ são mostradas na figura 6.16. A partir desta figura, verifique como seriam os mapas para $F(XYZ) = \overline{X}$, $F(XYZ) = \overline{Y}$ e $F(XYZ) = \overline{Z}$.

Um retângulo de quatro quadrados também pode ser da forma da figura 6.17, onde os quatro quadrados não estão anotados em uma única linha.

Para simplificar este mapa sem usar manipulação algébrica, basta verificar que o único literal que não varia é $\overline{Z}$. Logo, $F = \Sigma m(0,1,4,5) = \overline{Z}$.

Porém, há uma situação "inusitada" no mapa de Karnaugh. Considere a função $F = \Sigma m(0,2)$, com mapa indicado na figura 6.18.

A primeira impressão é que não há simplificação possível, mas manipulando algebricamente, vemos que:

$$\begin{aligned} F = \Sigma m(0,2) &= \overline{X}\cdot\overline{Y}\cdot\overline{Z} + \overline{X}Y\overline{Z} \\ &= \overline{X}\cdot\overline{Z}(\overline{Y}+Y) \\ &= \overline{X}\cdot\overline{Z} \end{aligned}$$

É para conseguir detectar este tipo de simplificação que o mapa não usa os mintermos na ordem "normal". A idéia é que no mapa de Karnaugh de três variáveis, as posições referentes aos minitermos m_0 e m_2, m_4 e m_6 são vizinhos. Para visualizar isso, imagine que o mapa é uma folha solta, e que fazemos um cilindro ao unir as suas bordas. Cada posição deste cilindro tem um vizinho à esquerda e um à direita. Por exemplo, os vizinhos de m_0 são m_1 e m_2.

Assim, a simplificação do mapa da figura 6.18 é mostrada na figura 6.19

Uma última complicação do Mapa de Karnaugh é em que ordem proceder a simplificação. Como exemplo, vamos analisar novamente $F = \Sigma m(0,1,4,5)$. Na figura 6.17, colocamos direto um retângulo de

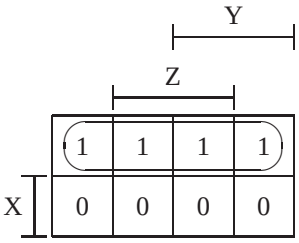


Figura 6.15: Mapa de Karnaugh para $F = \Sigma m(0,1,2,3)$

.

quatro quadrados e nem foi sugerido colocar dois retângulos de dois quadrados (tem dois jeitos de fazer isso). É que existe uma ordem para anotar o mapa:

1. Procure o retângulo com o maior número de quadrados e o anote. O número de quadrados em um retângulo **tem de** ser potência de dois (ou seja, 2, 4, 8, 16, etc).
2. Para cada quadrado que sobrou, combine-o com o maior número de quadrados não utilizados possível, reutilizando os quadrados já anotados.
3. Quando não sobrar nenhum quadrado, a simplificação terminou.

Esta é um "algoritmo guloso" (adivinhe porque ele recebe este nome).

Para exemplificar este processo, considere a função $F = \Sigma m(0,1,2,3,5)$. O processo de simplificação é mostrado na figura 6.20.

O lado esquerdo da figura mostra a primeira simplificação. Porém, como nem todos os quadrados foram usados, é necessária uma segunda aplicação do processo. Após esta segunda aplicação, temos o resultado indicado no lado direito da figura. Como nesta figura, todos os quadrados foram utilizados, então o processo termina.

Com estas simplificações, temos que
$F = \Sigma m(0,1,2,3,5) = \overline{X} + \overline{Y}Z$

Até o momento, lidamos somente com funções que tem as três variáveis em cada termo. Porém, como transportar a equação $F = \overline{X}Z + \overline{X}Y + X\overline{Y}Z + YZ$ para o mapa?

O termo $X\overline{Y}Z$ é fácil de transportar, pois ele é o minitermo m_5. Já os demais termos só tem duas variáveis. Porém, lembre-se que sempre é possível incluir a terceira variável da seguinte forma:

$$\begin{aligned} \overline{X}Z &= \overline{X}Z \cdot 1 \\ &= \overline{X}Z(Y + \overline{Y}) \\ &= \overline{X}YZ + \overline{X} \cdot \overline{Y}Z \\ &= m_3 + m_1 \end{aligned}$$

De maneira análoga, $\overline{X}Y = \Sigma m(2,3)$, e assim, $F = \overline{X}Z + \overline{X}Y + X\overline{Y}Z + YZ = \Sigma m(1,2,3,5,7)$ (figura 6.21).

Após a fase de montagem do mapa, a próxima fase é a simplificação mostrada na figura 6.22. Deste mapa, é fácil comprovar que $F = \overline{X}Z + \overline{X}Y + X\overline{Y}Z + YZ = Z + \overline{X}Y$.

Voltando agora ao algoritmo guloso, uma última observação é que a simplificação deve sempre tentar minimizar o número de retângulos. Um bom exemplo disso é a figura 6.23.

O lado esquerdo da figura 6.22 é o resultado mais apropriado pois tem menor número de termos. O lado direito tem um termo a mais. Aliás, esta função já foi apresentada anteriormente no teorema do consenso 6.28.

Pode-se chegar ambos os mapas aplicando o algoritmo guloso. Para chegar ao mapa esquerdo, inicie associando m_2 e m_3, e para chegar ao mapa esquerdo, inicie associando m_3 e m_7.

Neste caso, a solução é analisar novamente o mapa para ver se não tem algum termo "dispensável". Se tiver, reiniciar o processo deixando este termo para o final.

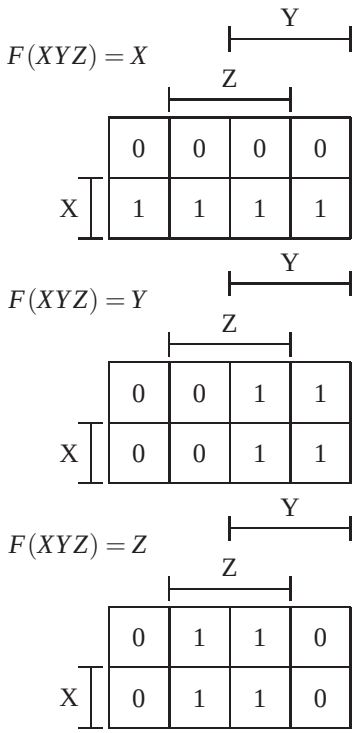


Figura 6.16: Mapa de Karnaugh que resultam em uma variável

.

6.13.1.2 Observações para MK de três variáveis

Em um Mapa de Karnagh com três variáveis:

- um quadrado indica um minitermo de três literais;
- um retângulo de dois quadrados indica um produto de dois literais;
- um retângulo de quadro quadrados indica um literal;
- um retângulo de oito quadrados encampa todo o mapa e é igual a verdadeiro (1 lógico).

6.13.1.3 Exercícios

1. Refaça todos os exercícios da seção 6.11.5 usando Mapas de Karnaugh. Nos exercícios de comparação, monte os mapas para os dois lados da equação.
2. Simplifique as funções booleanas abaixo usando o mapa.
   (a) $F(X,Y,Z) = \overline{X} \cdot \overline{Z} + Y\overline{Z} + XYZ$
   (b) $F(A,B,C) = \overline{A}B + \overline{B}C + \overline{A} \cdot \overline{B} \cdot \overline{C}$
   (c) $F(X,Y,Z) = \overline{A} \cdot \overline{B} + A\overline{C} + \overline{A}B\overline{C}$
   (d) $F(X,Y,Z) = \Sigma m(1,3,6,7)$
   (e) $F(X,Y,Z) = \Sigma m(3,5,6,7)$

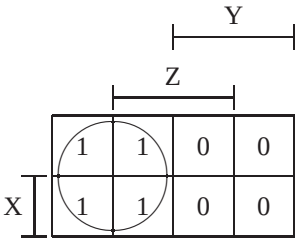


Figura 6.17: Mapa de Karnaugh para $F = \Sigma m(0,1,4,5)$

.

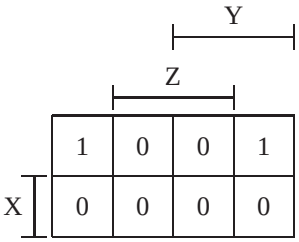


Figura 6.18: Mapa de Karnaugh para $F = \Sigma m(0,2)$

.

(f) $F(A,B,C) = \Sigma m(0,1,2,4,6)$

(g) $F(A,B,C) = \Sigma m(0,3,4,5,7)$

(h) $F(A,B,C) = \Sigma m(0,1,3,5)$

6.13.2 Mapa de Karnaugh de quatro variáveis

Um mapa de Karnaugh de quatro variáveis contém 16 mintermos, como apresentado na figura 6.24. Na figura 6.25, cada posição aparece com um número binário de quatro dígitos. Cada dígito representa uma das variáveis. Assim, o padrão `0110` corresponde a:

- primeira variável invertida;
- segunda variável sem inversão;
- terceira sem inversão;
- quarta invertida.

Como a função é F(W,X,Y,Z), então `0110` corresponde a $\overline{W}XY\overline{Z}$.

Agora, observe que os dois primeiros bits são sempre iguais em cada linha. A primeira linha, `00` (ou $\overline{WX}$), a segunda `01` (ou $\overline{W}X$) , a terceira `11` (ou WX) (quantos esperavam `10`?) e a quarta linha `10` (ou $W\overline{X}$).

As colunas também seguem um padrão. Na figura, os dois últimos dígitos sempre correspondem a Y e Z. Assim, na primeira coluna, temos `00` (ou $\overline{YZ}$). Na segunda coluna, `01` (ou $\overline{Y}Z$), na terceira coluna `11` (ou YZ) e finalmente na última coluna, `10` (ou $Y\overline{Z}$). Isso ajuda a montar a tabela.

É mais simples completar a tabela usando o mapa binário (figura 6.25 do que preencher a tabela com as variáveis. Assim como no mapa de Karnaugh de três variáveis, esta figura também separa as duas primeiras variáveis (W e X) das duas últimas (Y e Z).

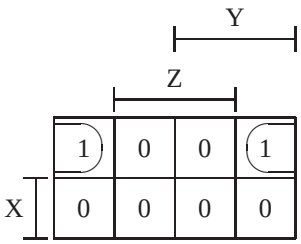


Figura 6.19: Anotação do Mapa de Karnaugh para $F = \Sigma m(0,2)$

.

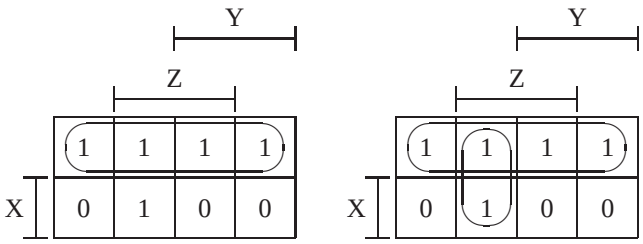


Figura 6.20: Passo a passo da anotação do Mapa de Karnaugh para $F = \Sigma m(0,1,2,3,5)$

.

O algoritmo de simplificação em um mapa de quatro variáveis é o mesmo algoritmo guloso apresentado na seção 6.13.1.1. Assim como no mapa de três variáveis, cada quadrado tem quatro vizinhos. Os quadrados das bordas também tem quatro vizinhos, só que aqui são um pouco mais difíceis de ver:

- Vizinhos de m_0: m_1, m_4, m_2, m_8
- Vizinhos de m_1: m_0, m_5, m_3, m_9
- ...

Agora, vamos analisar algumas simplificações em MK de quatro variáveis.

1. $F(W,X,Y,Z) = \Sigma\, m(0,1,2,4,5,6,8,9,12,13,14)$. Veja figura 6.26.

   (a) Seguindo o algoritmo guloso, precisamos encontrar o retângulo que tem o maior número de quadrados. Este retângulo contém oito quadrados e é mostrado na figura 6.27. Este retângulo corresponde à espressão $\overline{Y}$.

   (b) A partir da figura 6.27, procuramos agrupar os quadrados que não foram utilizados em retângulos que contenham o maior número de quadrados. No caso, temos três quadrados que são candidatos. A figura figura 6.28 anota o uso de m_2 e m_6, que resulta em $\overline{WZ}$.

   (c) A partir da figura 6.28, ainda há um quadrado sem uso (m_{14}). Este quadrado deve ser agrupado em um retângulo com o maior número de quadrados possível. Ele pode ser agrupado com quatro outros quadrados, como anotado na figura 6.29, que resulta em $X\overline{Z}$.

   (d) A figura 6.29 não apresenta mais quadrados não usados. Sendo assim, a expressão resultante após as simplificações é: $\overline{Y} + \overline{WZ} + X\overline{Z}$.

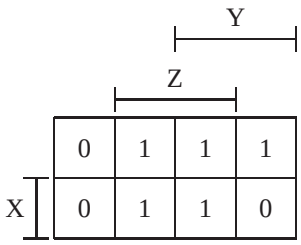


Figura 6.21: $F = \Sigma m(1,2,3,5,7)$

.

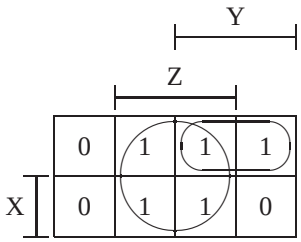


Figura 6.22: Simplificação de $F = \Sigma m(1,2,3,5,7)$

.

6.13.2.1 Observações para MK de quatro variáveis

Em um Mapa de Karnagh com quatro variáveis:

- um quadrado indica um minitermo de quatro literais;
- um retângulo de dois quadrados indica um produto de três literais;
- um retângulo de quadro quadrados indica um produto de dois literais;
- um retângulo de oito quadrados indica um produto de um literal;
- um retângulo de dezesseis quadrados encampa todo o mapa e é igual a verdadeiro (1 lógico).

É possível trabalhar com mapas de Karnagh para mais do que quatro variáveis, porém isso não será analisado aqui.

6.13.2.2 Exercícios

1. Simplifique as seguintes funções booleanas de quatro variáveis:
   (a) $F(A,B,C,D) = \Sigma\, m(1,5,9,12,13,15)$
   (b) $F(W,X,Y,Z) = \Sigma\, m(1,3,9,11,12,13,14,15)$
   (c) $F(A,B,C,D) = \Sigma\, m(0,2,4,5,6,7,8,10,13,15)$
   (d) $F(W,X,Y,Z) = \Sigma\, m(0,2,5,8,9,11,12,13)$
   (e) $F(A,B,C,D) = \Sigma\, m(3,4,6,7,9,12,13,14,15)$

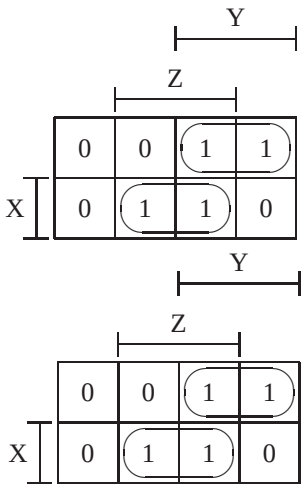


Figura 6.23: $F = \Sigma m(3,4,5,7)$

.

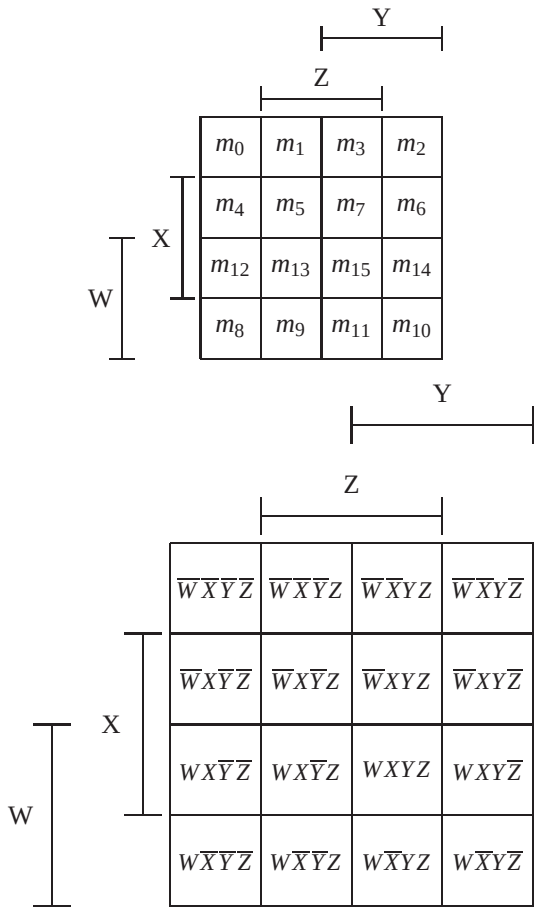


Figura 6.24: Mapas de Karnaugh para $F(WXYZ)$

.

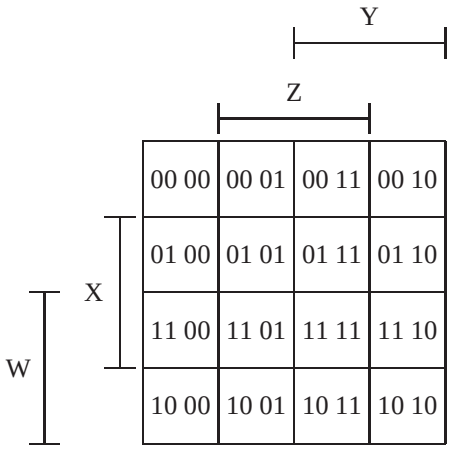


Figura 6.25: Representação binária das variáveis no mapa

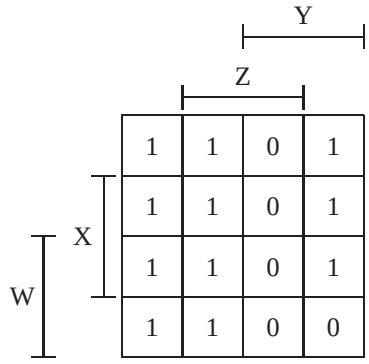


Figura 6.26: $F = \Sigma\, m(0,1,2,4,5,6,8,9,12,13,14)$.

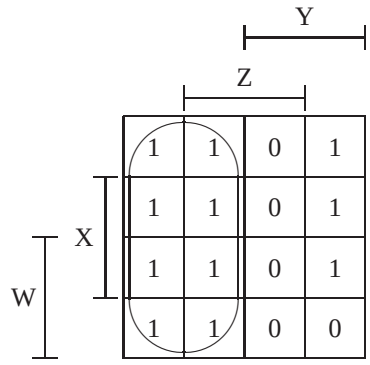


Figura 6.27: Anotação de $\overline{Y}$

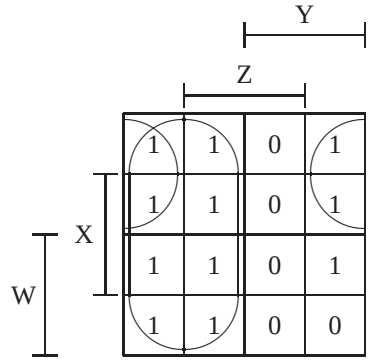


Figura 6.28: Anotação de $\overline{Y} + \overline{WZ}$

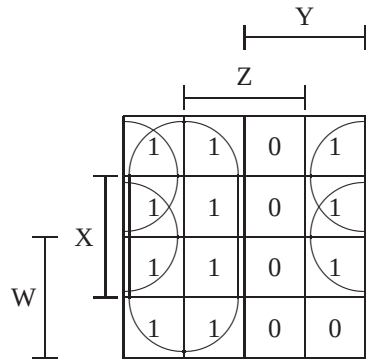


Figura 6.29: Anotação de $\overline{Y} + \overline{WZ} + X\overline{Z}$

Apêndice A

Tabela ASCII

A tabela ASCII (*American Standard Code for Information Interchange*, que se lê ascii e não asque dois.) foi definida em 1961 para resolver os problemas de representação dos caracteres da língua inglesa, e por isso não tem nenhum caractere acentuado.

Como existem 256 representações diferentes em oito bits, pode-se dividir as 256 representações em três grupos:

[$(00)_{16}$ **..** $(1F)_{16}$**,** $(127)_{16}$] caracteres de controle.

[$(20)_{16}$ **..** $(7E)_{16}$] caracteres “imprimíveis”.

[$(80)_{16}$ **..** $(FF)_{16}$] variável. Pode ser: ISO 8859-1 ou ISO 8859-2 ou ... ou ISO 8859-16.

Os caracteres entre $(00)_{16}$ e $(1F)_{16}$ são caracteres de controle, alguns já não usados atualmente por não terem mais sentido. Por exemplo, o caractere $(07)_{16}$ emite um sinal sonoro. Em alguns sistemas, já não emite som algum, porém em outros pode emitir um bipe.

A tabela ASCII completa pode ser vista na figura A.1 (extraída de [ISO]). Esta figura contém os caracteres latinos (latin1 = ISO 8859-1). A leitura é feita por colunas e linhas, ou seja, o caractere $(41)_{16} = (A)_{ASCII}$ está na quarta coluna, segunda linha.

Nesta tabela, as entradas destacadas correspondem a caracteres de controle. Observe que na parte associada ao ISO 8859-1, também há caracteres de controle ([$(80)_16$..$(9F)_{16}$]).

6.2 Code table

For each character in the set the code table (table 2) shows a graphic symbol at the position in the code table corresponding to the bit combination specified in table 1.

The shaded positions in the code table correspond to bit combinations that do not represent graphic characters. Their use is outside the scope of ISO/IEC 8859; it is specified in other International Standards, for example ISO/IEC 6429.

Table 2 – Code table of Latin alphabet No. 1

b_4 b_3 b_2 b_1		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
	b_8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
	b_7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	
	b_6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
	b_5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
0 0 0 0	00			SP	0	@	P	`	p			NBSP	°	À	Ð	à	ð	0
0 0 0 1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0 0 1 0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0 0 1 1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0 1 0 0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0 1 0 1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0 1 1 0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0 1 1 1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1 0 0 0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1 0 0 1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1 0 1 0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1 0 1 1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1 1 0 0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1 1 0 1	13			−	=	M	]	m	}			SHY	½	Í	Ý	í	ý	D
1 1 1 0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1 1 1 1	15			/	?	O	_	o				¯	¿	Ï	ß	ï	ÿ	F
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Hex

Figura A.1: Tabela ASCII+ISO 8859-1 (extraída de [ISO])

Apêndice B

Chamadas ao Sistema (syscalls)

A tabela B.1 contém as system calls abordadas neste texto. Versões mais completas (com `sbrk`, `open`, `close`, `read`, `write`, etc. não são de interesse neste texto, mas podem ser achada através de busca na internet.

Serviço	Código (`$v0`)	Argumentos	Resultado
print int	1	$a0=inteiro	
print float	2	$f12=PF simples	
print double	3	$f12=PF duplo	
print string	4	$a0=end. string	
read int	5		inteiro em $v0
read float	6		PF simples em $f0
read double	7		PF duplo em $f0
read string	8	$a0=buffer $a1=tamanho buffer	
exit	10		
print char	11	$a0 = char	
read char	12		char em $v0

Tabela B.1: Tabela de Chamadas ao sistema (syscalls) do SPIM

Bibliografia

[DJ96] David A. Patterson and John L. Hennessy. *Computer Organization and Design: A Quantitative Approach*. Morgan Kaufmann, 1996. *Livro adotado em vários cursos de pós-graduação.*

[DJ97] David A. Patterson and John L. Hennessy. Computer Organization and Design: The Hardware/Software Interface. *Morgan Kaufmann, 1997. Livro usado em pelo menos mais duas disciplinas do curso. Adotado em praticamente todos os grandes cursos de computação do mundo. Tem versão em português com o nome "Organização e Projeto de Computadores. A Interface Hardware/Software", LTC editora.*

[Flá00] Flávio Keidi Miyazawa. Algoritmos e Programação de Computadores. *Notas de Aula, 2000. http://www.ic.unicamp.br/ fkm/.*

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. Computing Surveys*, 1991.*

[ISO] 8-bit single-byte coded graphic character sets, part 1: Latin alphabet no. 1 (draft dated february 12, 1998, published april 15, 1998).

[Jam90] James R. Larus. Spim s20 a mips r200 simulator. 1990. Contém uma descrição do SPIM, em especial, lista todas as intruções assembly do processador.

[Kat94] Randy H. Katz. Conteporary Logic Design. *Benjamin Cunnings, 1994.*

[Knu97] Donald E. Knuth. Art of Computer Programming, Volume 1: Fundamental Algorithms. *Addison-Wesley Professional, 1997.*

[MC00] M. Morris Mano and Charles R. Kime. Logic and Computer Design Fundamentals. *Prentice-Hall, 2000.*

[Mon01] Mário A. Monteiro. Introdução à Organização de Computadores. *LTC, 2001. Contém quase todo o material sobre representação de informação em computador, em especial números inteiros, naturais e reais.*

[Spo03] Joel Spolsky. The absolute minimum every software developer absolutely, positively must know about unicode and character sets (no excuses!). Technical report, http://www.joelonsoftware.com/, 2003.

[Tan99] Andrew Tannenbaum. Organização Estruturada de Computadores. *LTC editora, 1999.*