Capítulo 6

Aritmética Computacional

As palavras de um computador são compostas por *bits* e podem representar números armazenados na memória. Estes números podem ter diferentes significados, como inteiros ou reais, serem positivos ou negativos. A manipulação dos números inclui operações de soma, subtração, multiplicação e divisão.

O objetivo deste capítulo é mostrar como o *hardware* implementa a representação dos números, os algoritmos adequados para operações aritméticas e sua implicação no conjunto de instruções da máquina.

6.1 Números com Sinal e Números sem Sinal

Os números podem ser representados em qualquer base. Porém, a base 2 é a mais adequada para os computadores porque tratam com somente dois valores 0 e 1. Estes valores são implementados facilmente através de circuitos elétricos.

Da aritmética temos que, em qualquer base, o valor do *i*-ésimo dígito *d* de um número é dado por: $\boldsymbol{d \times base^{i}}$, onde *i* começa em 0 e cresce da direita para a esquerda, de acordo com a posição ocupada pelo dígito. Por exemplo, o número 1011 na base dois é igual a:

$$(1x2^3) + (0x2^2) + (1x2^1) + (1x2^0) = (1x8) + (0x4) + (1x2) + (1x0) = 8 + 0 + 2 + 1 = 11.$$

Portanto, os *bits* são numerados como 0,1,2,3,... da direita para a esquerda em uma palavra. Utilizamos a expressão *bit* menos significativo para designar o *bit* 0, e a expressão *bit* mais significativo para designar o *bit* de mais alta ordem, como, por exemplo, o *bit* 31 numa palavra de 32 *bits*.

Como o tamanho de uma palavra manipulada por um computador tem tamanho limitado, os números que podem ser representados também têm tamanho limitado. Se o tamanho de uma palavra é igual a *n bits*, o maior número possível de ser representado é igual a 2^n. Se ao realizarmos operações sobre os números, elas gerarem resultados que não podem ser representados na quantidade de *bits* reservados ocorre o que denominados de *overflow* (números muito grandes) ou *underflow* (números muito pequenos). Tanto o *overflow* quanto o *underflow* geram exceções e são tratados pelo sistema operacional.

Os computadores manipulam tanto números positivos quanto números negativos, que são representados em **complemento a 2**. Nesta convenção os números que possuem 0s à esquerda são considerados positivos e os números com 1s à esquerda são considerados negativos. O complemento a 2 é obtido invertendo-se o número binário e depois somando 1 a este valor. Porém, uma regra simples para transformar um número binário em sua representação em complemento a 2 é a seguinte: 1) copie da direita para a esquerda todos os *bits* até encontrar o primeiro *bit* 1 inclusive e 2) inverta todos os demais *bits*. A Figura 6.1 ilustra um exemplo da obtenção de representação em complemento a 2 de um número binário com 4 dígitos.

```
 0110  =  6 na base 10                    usando a regra:
  1001  (número binário invertido)        0110  →  0110
+ 0001  (soma com 1)
  1010  (complemento a 2)
```

Figura 6.1. *Representação em complemento a 2.*

A representação em complemento a 2 tem a vantagem de representar números negativos sempre com o *bit* 1 em sua posição mais significativa. Assim, o *hardware* só precisa testar este *bit* para verificar se o número é positivo ou negativo. Este *bit* é conhecido como *bit* de sinal. Na Figura 6.2 está representada uma seqüência de números binários (8 dígitos) representados em complemento 2.

```
0000 0000 = 0
0000 0001 = 1
0000 0010 = 2
0000 0011 = 3
  .............
0111 1101 = 125
0111 1110 = 126
0111 1111 = 127
  .............
1000 0001 = -127   (primeiro número negativo)
1000 0010 = -126
1000 0011 = -125
  .............
1111 1101 = -3
1111 1110 = -2
1111 1111 = -1
```

Figura 6.2. *Seqüência de números binários representados em complemento a 2.*

Conversão entre Diferentes Bases

As bases octal e hexadecimal também são muito úteis em computação. A base octal é representada com 8 dígitos que variam entre 0 e 7. A base hexadecimal é composta por dígitos e letras da seguinte forma: 0 a 9 e as letras a, b, c, d, e e f. Na Figura 6.3 ilustramos como realizar a mudança da base binária para as demais bases de forma simples. Lembre-se de que precisamos de 3 ou 4 dígitos binários para a mudança de base octal e hexadecimal, respectivamente.

Binário	Octal	Hexadecimal
9 8 7 6 5 4 3 2 1 0	1 000 111 101	10 0011 1101
1 0 0 0 1 1 1 1 0 1 = 573	1 0 7 5 = 1075	2 3 d = 23d

Figura 6.3. *Mudança de base.*

6.2 Adição e Subtração

Numa soma os *bits* são somados um a um da direita para a esquerda, com os *carries* sendo passados para o próximo *bit* à esquerda. A operação de subtração usa a adição. O subtraendo é simplesmente negado antes de ser somado ao minuendo. Lembre-se que a máquina trata com números representados em complemento a 2. A Figura 6.4 mostra as operações de soma (6+7) e subtração (7-6) *bit* a *bit* entre dois números representados com 4 dígitos binários.

Representação binária	Soma	Subtração
7 = 0 1 1 1	1 1 (vai um)	1 1 1 (vai um)
6 = 0 1 1 0	0 1 1 1	0 1 1 1
13 = 1 1 0 1 (soma)	+ 0 1 1 0	+ 1 0 1 0 (complemento a 2)
1 = 0 0 0 1 (subtração)	1 1 0 1	0 0 0 1

Figura 6.4. *Operações de soma e subtração (complemento a 2) com representação binária*.

Como citado anteriormente, tanto a soma como a subtração podem gerar *overflow* ou *underflow*, se o resultado obtido não puder ser representado pela quantidade de *bits* que formam uma palavra. Se somarmos ou subtrairmos dois números com sinais contrários, nunca ocorrerá *overflow* ou *underflow*. Isto porque operandos com sinais contrários nunca podem ser maior do que qualquer dos operandos.

O *overflow* ocorre quando somamos dois operandos positivos e obtemos um resultado negativo, ou vice-versa. Isto significa que utilizamos o *bit* de sinal, gerando um *carry*, para armazenar um valor pertencente ao resultado da operação. Raciocínio semelhante é realizado para detectar a ocorrência do *underflow* numa subtração. Neste caso, o *bit* de sinal também é usado para armazenar um valor pertencente ao resultado da operação.

Os projetistas de um sistema devem decidir onde tratar a ocorrência de *overflow* ou de und*erflow* em operações aritméticas. Elas podem ser tratadas tanto por *hardware* quanto por *software*. Pode existir a detecção por *hardware* que gera uma exceção, e que depois é tratada por *software*.

6.3 Operações Lógicas

Os computadores manipulam palavras, mas é muito útil, também, manipular campos de *bits* dentro de uma palavra ou mesmo *bits* individuais. O exame de caracteres individuais (8 *bits*) dentro de uma palavra é um bom exemplo dessa necessidade. Assim, as arquiteturas de conjuntos de instruções incluem instruções para manipulação de *bits*.

Um dos tipos de instrução utilizados são as de deslocamento de *bits*. As instruções podem deslocar *bits* tanto à direita quanto à esquerda. Todos os *bits* são movidos para o lado determinado e os *bits* que ficam vazios são preenchidos com 0s. Outras instruções lógicas muito úteis são que implementadas na unidade lógica e aritmética de um processador são as operações NOT, AND, OR e XOR. A Figura 6.5 mostra as operações lógicas, *bit* a *bit*, de deslocamento à direita, à esquerda, NOT, AND, OR e XOR.

Desl. à direita	Desl. à esquerda	NOT	AND	OR	XOR
1 1 0 1	1 1 0 1	1 1 0 1	1 1 0 1	1 1 0 1	1 1 0 1
0 1 1 0	1 0 1 0	0 0 1 0	0 1 0 1	0 1 0 1	0 1 0 1
			0 1 0 1	1 1 0 1	1 0 0 0

Figura 6.5. *Operações lógicas.*

6.4 Construção de uma Unidade Lógica Aritmética

A unidade lógica aritmética (ALU – *Arithmetic Logic Unit*) é o dispositivo que realiza as operações lógicas e aritméticas, definidas pelo conjunto de instruções, dentro do processador.

A ALU é construída basicamente por quatro blocos básicos de *hardware*: portas AND, portas OR, NOT (inversores) e multiplexadores.

As implementações de operações lógicas são as mais simples de serem realizadas, pois elas são mapeadas diretamente com componentes do *hardware*.

A próxima função a ser incluída é a adição. Supondo que temos apenas um *bit* para ser somado, necessitamos de um circuito com duas entradas para os operandos, uma saída para a soma resultante, uma entrada relativa ao *carry in* e uma saída para o *carry out*. A Figura 6.6 mostra este somador.

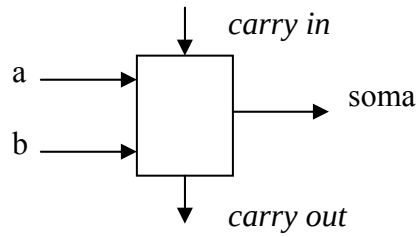

Figura 6.6. *Somador de um bit*

Podemos especificar as saídas soma e *carry out* através de equações lógicas, que podem ser implementadas a partir dos blocos de *hardware* mencionados anteriormente. A equação lógica para gerar o *bit carry out* é dada por:

$$\mathbf{CarryOut = (b.CarryIn) + (a.CarryIn) + (a.b).}$$

E, a equação lógica para gerar o *bit* soma é dada por:

$$\mathbf{Soma = (a \,.\, \overline{b} \,.\, \overline{CarryIn}) + (\overline{a} \,.\, b \,.\, \overline{CarryIn}) + (\overline{a} \,.\, \overline{b} \,.\, CarryIn) + (a \,.\, b \,.\, CarryIn).}$$

Para completar o projeto de uma ALU de *n bits* podemos conectar *n* somadores de um *bit*. Os *carry outs* gerados pelos *bits* menos significativos da operação podem ser propagados por toda a extensão do somador, gerando um *carry out* no *bit* mais significativo do resultado da operação. Este somador é denominado somador de *carry* propagado.

A operação de subtração pode ser realizada somando-se o minuendo com a negação do subtraendo. Este efeito é realizado acrescentando uma entrada complementada de **b** ao somador e ativando o *carry in* do *bit* menos significativo para um. O somador então calcula a + b + 1. Ao escolhermos a versão invertida de **b** obtemos:

$$\mathbf{a + \overline{b} + 1 = a + (\overline{b} + 1) = a + (-b) = a - b.}$$

A simplicidade do projeto do *hardware* de um somador para números de complemento a 2 demonstra porque esta representação tornou-se um padrão para operações aritméticas inteiras em computadores.

O problema com o somador de *carry* propagado está relacionado à velocidade de propagação do *carry,* que é realizada seqüencialmente. Num projeto de *hardware* a velocidade é um fator crítico. Para solucionar este problema existem diversos esquemas para antecipar o *carry*. Porém, nestes esquemas são utilizadas mais portas lógicas o que provoca um aumento no custo.

Um dos esquema para antecipar o *carry* é denominado *carry lookahead*. Os somadores que utilizam o esquema de *carry lookahead* baseiam sua implementação em

vários níveis de abstração. Utilizando a abreviação *ci* para representar o *i*-ésimo *bit* de *carry*, podemos escrever a equação do *carry* como:

$$ci = (bi \cdot ci) + (ai \cdot ci) + (ai \cdot bi) = (ai \cdot bi) + (ai + bi) \cdot ci$$

Os termos ***(ai . bi)*** e ***(ai + bi)*** são tradicionalmente chamados de **gerador** (gi) e **propagador** (pi), respectivamente. Usando estas relações para definir *ci + 1*, obtemos:

$$ci + 1 = gi + pi \cdot ci$$

Qualquer equação lógica pode ser implementada com uma lógica de dois níveis. Mesmo esta formulação mais simplificada pode gerar equações muito grandes e, portanto levar a circuitos lógicos relativamente grandes e caros, dependendo do número de *bits* a serem somados.

6.5 Multiplicação

Para realizar a multiplicação são necessários dois operandos, um multiplicando e um multiplicador para gerar um operando produto. O algoritmo da multiplicação diz que os dígitos do multiplicando devem ser multiplicados pelos dígitos do multiplicador um de cada vez, da direita para a esquerda, deslocando os produtos intermediários um dígitos à esquerda em relação ao imediatamente anterior.

Uma observação importante é que o número de *bits* do produto final ($n+m$) é maior do que o número de *bits* do multiplicando (n) ou do multiplicador (m). Além disso, a multiplicação também precisa tratar a ocorrência de *overflow*.

Considerando os dígitos binários 0 e 1, temos apenas duas possibilidades de escolha, a cada passo da multiplicação:

1. coloque uma cópia do multiplicando (multiplicando x 1) no lugar apropriado, se o dígito do multiplicador for igual a 1, ou
2. coloque 0 (multiplicando x 0) no lugar apropriado, se o dígito do multiplicador for igual a 0.

Assim, é necessário desenvolver um algoritmo em *hardware* que seja eficiente para realizar a multiplicação.

Um método elegante de multiplicar números com sinal recebeu o nome de **algoritmo de Booth**. Ele foi elaborado a partir da constatação de que com a capacidade de somar e de subtrair números existem várias maneiras de se calcular um produto. Por exemplo, podemos substituir um *string* de 1s no multiplicador por uma subtração quando encontramos o primeiro 1, e por uma soma ao encontrarmos o último 1 do *string*. Maiores detalhes podem ser vistos no livro texto.

Booth buscou atingir maior velocidade de processamento utilizando operações de deslocamento, que ainda hoje são operações mais rápidas do que operações de soma. Baseado nesta observação, se desejarmos maior velocidade na multiplicação de números inteiros por

uma potência de 2, basta que utilizemos operações de deslocamento indicando a quantidade de deslocamentos igual ao expoente.

A grande vantagem do algoritmo de Booth é tratar com facilidade os números com sinal. O raciocínio de Booth foi classificar os grupos de *bits* como início, meio e fim de um *string* de 1s. Naturalmente um *string* de 0s não precisa ser considerado.

Algoritmo da multiplicação

Este algoritmo precisa apenas de dois passos principais: o teste do produto e o seu deslocamento; pois os registradores Produto e Multiplicador podem combinados em um só. O algoritmo começa com o Multiplicador na metade à direta do registrador Produto, e 0 na metade à esquerda.

1. Testa se Produto é igual a 0 ou 1.
2. Produto = 0, passa ao item 4.
3. Produto = 1, soma o Multiplicando à metade esquerda do Produto e coloca o resultado na metade à esquerda do registrador Produto.
4. Desloca o registrador Produto 1 bit à direita.
5. Verifica se foram realizadas todas as repetições necessárias de acordo com o tamanho da palavra, se não volta ao item 1.
6. Fim.

6.6 Divisão

A divisão é a operação recíproca da multiplicação. Dentre as operações aritméticas é a que aparece menos freqüentemente nos códigos dos programas.

No algoritmo da divisão são utilizados dois operandos, o dividendo e o divisor, e produzidos dois resultados o quociente e o resto. A relação entre os componentes da divisão pode ser expressa da seguinte forma:

dividendo = quociente x divisor + resto,

onde o resto é sempre menor que o divisor.

Às vezes, os programas usam a divisão simplesmente para obter o resto, ignorando o quociente. Além disso, é necessário que seja detectada a divisão por zero, que é matematicamente inválida.

Algoritmo da divisão

Da mesma forma que foram combinados registradores na multiplicação, também na divisão são combinados dois registradores, o Resto e o Quociente. O algoritmo começa com o Resto na metade à esquerda do registrador Resto, e o Quociente na metade à direita.

1. Desloca o registrador Resto 1 bit à esquerda.
2. Subtrai o registrador Divisor da metade à esquerda do registrador Resto e armazena o resultado na metade esquerda do registrador Resto.
3. Testa se Resto é menor do que 0.
4. Resto < 0, restaura valor original com Divisor + metade esquerda do Resto, armazenando na metade esquerda do registrador Resto e deslocando 1 bit à esquerda, inserindo 0 no novo bit menos significativo, passa ao item 6.
5. Resto ≥ 0, desloca o registrador Resto 1 bit à esquerda, inserindo 1 no novo bit mais à direita.
6. Verifica se foram realizadas todas as repetições necessárias de acordo com o tamanho da palavra, se não volta ao item 1.
7. Desloca a metade à esquerda do registrador Resto 1 bit à direita, Fim.

6.7 Ponto Flutuante

Assim como os números decimais podem ser representados em notação científica normalizada os números binários também podem. A aritmética computacional que manipula os números binários em notação científica normalizada é denominada de aritmética de ponto flutuante.

Os projetistas do *hardware* devem encontrar um compromisso entre a mantissa e o expoente dos números em ponto flutuante. A relação entre e mantissa e o expoente é expressa do seguinte modo: o aumento do número de *bits* reservados à mantissa aumenta a precisão do número, enquanto o aumento do número de *bits* reservados ao expoente aumenta o intervalo de variação dos números representados.

Deve ser observado que as interrupções relativas ao *overflow* e ao *underflow* também ocorrem na representação em ponto flutuante. Porém, neste caso, *overflow* e o *underflow* ocorrem quando o expoente é muito grande ou muito pequeno, respectivamente, para ser armazenado no espaço reservado a ele.

Outra questão que os projetistas devem decidir é se vão ser utilizados os mesmos registradores tanto para números inteiros quanto para números de ponto flutuante. A adoção de registradores diferentes aumenta ligeiramente o número de instruções necessárias à execução do programa. O impacto maior está na criação de um conjunto de instruções de

transferência de dados para mover os dados entre os registradores de ponto flutuante e a memória. Os benefícios estão no fato de não precisar aumentar o tamanho do campo nas instruções para diferenciar os operandos e aumentar a banda passante dos registradores.

A partir de 1980 todos os computadores projetados adotam uma representação padrão para números em ponto flutuante denominada IEEE 754. A adoção deste padrão facilita a portabilidade de programas (precisão simples = 1 *bit* de sinal, 8 *bits* de expoente e 23 *bits* de mantissa + 1 implícito = 24, precisão dupla = 1 *bit* de sinal, 11 *bits* de expoente e 52 *bits* de mantissa + 1 implícito = 53).

A adição e a multiplicação com números de ponto flutuante, na sua essência, utilizam as operações inteiras correspondentes para operar as mantissas, mas é necessária uma manipulação extra nos expoentes e para a normalização do resultado.

Algoritmo da adição em ponto flutuante

1. Compare o expoente dos dois números. Desloque o menor número à direita até que seu expoente se iguale ao maior número.
2. Some as mantissas.
3. Normalize a soma, deslocando à direita e incrementando o expoente ou deslocando à esquerda e decrementando o expoente.
4. Teste se *overflow* ou *underflow*.
5. Sim, gera exceção.
6. Não, arredonde a mantissa para o número de *bits* apropriado.
7. Testa se resultado está normalizado.
8. Sim, Fim.
9. Não, retorna ao passo 3.

Algoritmo da multiplicação em ponto flutuante

1. Soma os expoentes com peso dos dois números, subtraindo o valor do peso da soma para obter o novo expoente.
2. Multiplique as mantissas.
3. Normalize o produto se necessário, deslocando à direita e incrementando o expoente.
4. Teste se *overflow* ou *underflow*.
5. Sim, gera exceção.
6. Não, arredonde a mantissa para o número de *bits* apropriado.
7. Testa se resultado está normalizado.
8. Não, retorna ao passo 3.
9. Não, faça o sinal do produto positivo se ambos os sinais dos operandos originais são os mesmos, caso contrário o sinal é negativo, Fim.