CAPÍTULO IV

OPERAÇÕES ARITMÉTICAS BÁSICAS ENVOLVENDO NÚMEROS REAIS

IV.1 - O Padrão IEEE

Para os números reais do PRIMATA foi escolhido o padrão IEEE (*Institute of Electrical and Electronic Engineers*) para aritmética de ponto flutuante [Coon80]. Esta escolha se deve ao fato de o padrão IEEE ser reconhecido mundialmente e aceito como referência. Isto traz como maior vantagem a compatibilidade com outros sistemas.

O padrão IEEE é bastante completo , o que o torna complexo e lento. Além disto, ele é relativamente recente. Por causa destes fatores, no passado, a maior parte dos fabricantes criou um padrão próprio, mais rápido e melhor adaptado às características da sua arquitetura. Porém, atualmente, a maioria dos fabricantes oferece a opção de utilizar o padrão IEEE (ou algo bem próximo a ele), na representação e nas operações sobre números reais, cabendo ao usuário optar entre a padronização e uma representação proprietária mais rápida.

Mesmo entre os fabricantes que oferecem o padrão IEEE, muitos não cumprem integralmente a norma, ou seja, criam algumas exceções à regra, gerando um "pseudo-padrão". Como será visto mais adiante, neste trabalho também optou-se por criar algumas exceções que serão explicadas e justificadas posteriormente. Cabe ressaltar que a não implementação do padrão completo é uma opção por um melhor desempenho e não uma restrição da arquitetura.

O padrão IEEE oferece uma série de opções de: formato de armazenamento, arredondamento, "denormalização", tratamento de exceções, etc. Para este trabalho foram escolhidas sempre as opções de implementação mais simples e apenas estas serão apresentadas aqui.

Seja x um número real finito qualquer. Então, x pode ser representado em ponto-flutuante por $m \cdot r^e$, onde m é a mantissa, r é o radical e e é o expoente. A representação computacional restringe o universo de números através da limitação do número de *bits* utilizados no armazenamento da mantissa e do expoente. Quanto mais *bits* forem utilizados no armazenamento da mantissa, maior será a precisão da representação. Por outro lado, quanto mais *bits* forem utilizados no armazenamento do expoente, maior será o intervalo de valores

representáveis. O radical normalmente é um valor fixo, característico do formato utilizado, não precisando portanto ser armazenado. O seu valor afeta tanto a precisão quanto o intervalo de representação. Quanto maior o radical, maior será o intervalo de representação e maior será a perda de *bits* significativos num caso de alinhamento, necessário por exemplo na soma e na normalização, como será visto mais adiante. Para o padrão IEEE o radical é fixo e igual a 2 (dois).

O padrão IEEE permite uma série de formatos de armazenamento de números reais: básicos (*single*, *double*, *quad*) e estendidos (*single-extended* e *double-extended*). A diferença entre eles se encontra basicamente no número de *bits* utilizados no armazenamento de cada componente do número real. A norma indica que a implementação mínima deve permitir pelo menos o formato *single*. Sendo assim, este trabalho só implementará este formato e, a partir de agora, somente as suas características serão apresentadas.

Sendo o radical um valor fixo, apenas a mantissa e o expoente precisam ser armazenados. Existem várias formas de representá-los, como por exemplo: sinal e magnitude, complemento a um e complemento a dois. Para o expoente, o padrão IEEE não utiliza nenhuma destas representações, mas sim a "notação por excesso". Por esta notação, para uma representação com *a bits*, o expoente *e* é acrescido de 2^{a-1}, resultando portanto em $(e+2^{a-1})$. Desta forma, os números negativos ficam representados por valores positivos ou nulos e o intervalo de representação é o mesmo da representação em complemento a dois. A representação do expoente usando a "notação por excesso" é denominada *E*.

Já a mantissa é representada em sinal e magnitude. O sinal é denominado *S* e é nulo se a mantissa for positiva ou igual a um se a mantissa for negativa. Quanto ao posicionamento da vírgula, assim como na notação científica, a vírgula vem sempre após o primeiro algarismo, no caso, após o primeiro *bit*. Sendo assim, a magnitude da mantissa pode ser representada por *L*,*F*, onde *L* é a parte inteira (composta por apenas um *bit*) e *F* é a parte fracionária da mantissa. Como *L* é composto por apenas um *bit*, pode-se concluir também que a mantissa tem magnitude sempre menor do que dois.

A fim de melhorar a precisão da representação, a mantissa deve ser alinhada de tal forma que o *bit* mais significativo seja sempre igual a um. Desta forma, os *bits* não significativos à esquerda do número são eliminados e, conseqüentemente, o número de *bits* significativos da mantissa é maximizado. Um número representado desta forma é denominado normalizado. Em resumo, pode-se dizer que, para os números normalizados, L=1. Logo, $1 \le |m| < 2$. Sendo assim, sabendo-se que um número é normalizado, torna-se desnecessário o armazenamento da parte inteira da mantissa (*L*).

O valor zero não pode ser normalizado, assim como alguns valores muito pequenos. Por isto, o padrão IEEE também possui a representação "denormalizada". Os números "denormalizados" se caracterizam por terem a parte inteira da magnitude da mantissa nula, ou seja, eles só contêm a parte fracionária. Em resumo, pode-se dizer que, para os números denormalizados, L=0. Logo, $0 \le |m| < 1$. Sendo assim, sabendo-se que um número é normalizado, torna-se desnecessário o armazenamento da parte inteira da mantissa (L).

Para saber se um número é normalizado ou denormalizado no padrão IEEE basta análisar o valor do expoente. O valor mínimo do expoente é reservado para os números denormalizados. O valor máximo do expoente é reservado para representar valores infinitos e valores não numéricos (resultados de operações inválidas). Para os demais valores do expoente os números são considerados normalizados. Desta forma, todos os problemas da representação ficam resolvidos.

Um número real x é armazenado no formato *single* em 32 *bits*, como é mostrado na Figura IV.1. O primeiro *bit* representa o sinal da mantissa e é denominado S. Os 8 *bits* seguintes contém o expoente representado em "notação por excesso", denominado E. Por fim, os 23 *bits* restantes contém a parte fracionária da magnitude da mantissa, denominada F. Este formato tem como propriedade especial uma ordenação igual a ordenação de inteiros representados em sinal e magnitude. Esta característica facilita enormemente uma operação de comparação ou ordenação de números reais.

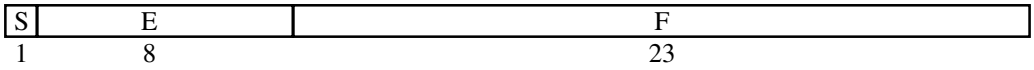

Figura IV.1 - Formato *single* do padrão IEEE

Se o expoente indicar um número normalizado, denormalizado ou nulo, ou seja, $E < 255$, pode-se dizer que: $x = (-1)^S \times 2^{E-127} \times (L, F)$, onde $L = 0$ se $E = 0$ e $L = 1$ caso contrário. O zero é um caso especial de número denormalizado. O zero pode ser positivo ou negativo, de acordo com o sinal armazenado em S. Da mesma forma, valores infinitos também podem ser positivos ou negativos, de acordo com o sinal armazenado em S. Um valor infinito é representado pelo valor máximo do expoente ($E = 255$) e por uma mantissa com parte fracionária nula ($F = 0$). O padrão IEEE contém dois modos de operação para os números infinitos, denominados *affine* e *projective*. O primeiro leva em consideração o sinal do número infinito, enquanto que o segundo ignora o sinal nas operações aritméticas envolvendo valores infinitos. Os algoritmos de operações aritméticas que serão apresentados mais adiante trabalham no modo *projective*, que é o modo padrão (*default*). Cabe ressaltar que os valores nulos ou infinitos podem não ser absolutos, mas apenas uma deficiência da representação, incapaz de representar valores muito pequenos ou muito grandes, devido às limitações do armazenamento. Sendo assim, valores muito pequenos acabam sendo aproximados para zero e

valores muito grandes acabam sendo aproximados para infinito. Os números normalizados podem variar em magnitude de $2^{-126} \times 1,000\cdots00$ até $2^{+127} \times 1,111\cdots11$.

O valor máximo do expoente ($E = 255$) acompanhado de uma mantissa com parte fracionária não nula ($F > 0$) é reservado para a representação de símbolos não numéricos (NaNs), decorrentes de operações inválidas como: divisão por zero, multiplicação de zero por infinito, etc. A Tabela IV.1 apresenta um resumo da interpretação do formato *single*.

Tabela IV.1 - Formato *single* do padrão IEEE

E	*F*	*x* (valor)	*x* (classificação)
0	0	± 0	zero
0	>0	$(-1)^S \times 2^{E-127} \times (0, F)$	número denormalizado
1..254	qualquer	$(-1)^S \times 2^{E-127} \times (1, F)$	número normalizado
255	0	$\pm \infty$	infinito
255	>0	NaN	símbolo não numérico

Quanto ao arredondamento, a norma apresenta vários modos: arredondamento para o valor mais próximo (*round to nearest*), arredondamento em direção a zero, arredondamento em direção a infinito (positivo) e arredondamento em direção a infinito (negativo). Uma implementação dentro do padrão necessita suportar apenas o primeiro modo ou todos os quatro modos. Optou-se pela implementação mais simples, ou seja, apenas o arredondamento para o valor mais próximo.

Durante as operações sobre números reais, algumas condições excepcionais podem ocorrer. O padrão classifica as possíveis exceções em: operação inválida, *underflow*, *overflow*, divisão por zero e resultado inexato. Uma das possibilidades de resposta previstas nas normas no caso de ocorrência de uma exceção é o uso de *traps*, capazes de desviar o fluxo de instruções para uma determinada rotina de tratamento de exceções definida pelo usuário. Esta não é a melhor solução para o caso específico das arquiteturas SIMD, visto que elas possuem um único fluxo de instruções para todos os processadores e as exceções são características individuais dos processadores, já que eles operam sobre dados diferentes. Outra possibilidade de resposta a uma exceção é atribuir um resultado específico (dependente da exceção) e continuar o processamento. Esta foi a solução adotada por este trabalho e é também o padrão definido como resposta a uma exceção.

Uma exceção do tipo operação inválida pode surgir de diversos pequenos erros não freqüentes nem importantes. Alguns exemplos de operações inválidas são: raíz quadrada de um número negativo, multiplicação de zero por infinito, subtração de infinito (positivo) por infinito (positivo). A resposta a uma exceção deste tipo é o símbolo não numérico (NaN), caracterizado por um expoente com valor máximo ($E = 255$) acompanhado de uma mantissa

com parte fracionária não nula ($F > 0$). As normas deixam a cargo de cada implementação a escolha do uso do campo não nulo *F*. Este trabalho optou por utilizar este campo para armazenar um código de identificação da ocorrência que acarretou numa operação inválida.

Uma exceção do tipo divisão por zero ocorre, como o próprio nome já diz, quando é executada a divisão de um número finito não nulo pelo número zero. A resposta padrão a uma ocorrência deste tipo é o valor infinito. Quanto ao sinal da resposta, ele é positivo se os dois operandos tiverem o mesmo sinal e negativo se os dois operandos tiverem sinais contrários.

O *underflow* ocorre quando o expoente de um resultado é menor do que o valor mínimo para o intervalo de números normalizados (no caso do formato *single*, $e < -126$ (ou $E < 1$)). No caso de ocorrência desta exceção, o resultado é denormalizado através de deslocamentos da sua mantissa para a direita. Para cada deslocamento de um *bit*, o expoente é incrementado em uma unidade. Este processo se repete até que o expoente seja igual ao valor mínimo armazenável no campo *E*, ou seja, $E = 0$. Num caso extremo, o expoente pode ser tão pequeno que o processo de denormalização acarreta em uma mantissa com campo significativo nulo (*bits* armazenáveis nulos). Neste caso, apesar de o resultado não ser realmente nulo, a única forma de representá-lo no formato utilizado é através do arredondamento para o valor zero. A notação de números denormalizados foi criada exatamente para diminuir a possibilidade de uma ocorrência de *underflow* resultar em um arredondamento para zero.

Por outro extremo, o *overflow* ocorre quando o expoente de um resultado é maior do que o valor máximo estabelecido para o intervalo de números normalizados (no caso do formato *single*, $e > 127$ (ou $E > 254$)). No caso de ocorrência desta exceção, apesar de o número ser finito, a única forma de representá-lo é através do arredondamento para infinito.

IV.2 - Soluções Básicas de Implementação

O padrão IEEE exige uma série de tratamentos de exceções para todas as operações aritméticas, além de algumas rotinas condicionais características de cada operação. Sendo assim, um algoritmo de aritmética de ponto flutuante típico possui uma série de desvios condicionais para pequenas rotinas. Estes desvios condicionais são dependentes dos operandos, o que é uma característica desfavorável a sua implementação numa arquitetura do tipo SIMD. Numa arquitetura SIMD o fluxo de instruções é único, mas os operandos são distintos. Logo, um único EP pode ser responsável pelo desvio para o processamento de uma rotina. Durante este processamento, os outros EPs ficam ociosos aguardando o fim da rotina. Desta forma, é como se o algoritmo levasse sempre em consideração o "pior caso". Isto acarreta num tempo de execução grande e fixo, já que ele "passa" por todas as rotinas condicionais.

Já numa arquitetura convencional o tempo de execução é variável, dependente dos operandos. Apenas as rotinas necessárias são executadas. Como é impossível que todas as rotinas sejam necessárias, o algoritmo tende a ter um desempenho melhor do que o de uma arquitetura SIMD. Para diminuir esta vantagem das arquiteturas convencionais, ao elaborar os algoritmos, procurou-se minimizar, sempre que possível, as rotinas condicionais.

IV.2.1 - Modificações na Arquitetura do Elemento Processador

Na análise das operações aritméticas de ponto-flutuante, optou-se por não considerar todas as arquiteturas de EPs propostas para as operações aritméticas sobre inteiros. Em virtude da enorme vantagem apresentada pelo EP de 4 *bits* com banco de registradores (EP4BR) sobre os outros EPs apresentados, optou-se por analisar e aprimorar apenas esta arquitetura. Algumas modificações foram feitas em relação à proposta inicial.

Boa parte do tempo gasto na execução de algoritmos de ponto-flutuante é utilizado em testes. No início de todos os algoritmos, por exemplo, os expoentes e mantissas dos operandos são testados, a fim de identificar se um dos operandos é nulo, infinito ou um símbolo não numérico. Como o elemento processador opera dados de 4 *bits* de cada vez, os testes envolvendo expoentes ou mantissas são subdivididos em comparações de 4 *bits* cada. As principais comparações são: igualdade com zero, desigualdade com zero e igualdade com 0xF (todos os *bits* iguais a 1). Os resultados de cada comparação são computados de forma a gerar o resultado final do teste. Da maneira como foi proposto o EP4BR no capítulo anterior, tanto as comparações quanto o acúmulo dos resultados parciais são realizados de forma não trivial. Na arquitetura proposta, para testar se um número *x* de *n bits* é igual a zero, pode-se utilizar o algoritmo abaixo:

$C = 1, B = X_0$
para *i* variando de 1 até $n/4 - 1$
    $(C,) = 0 + \overline{B} + C, B = X_i$
$C(C,) = 0 + \overline{B} + C$
$K = (0 + 1111 + C)_3$

Ao final do teste o resultado encontra-se armazenado no registrador K. Este algoritmo consome $n/4 + 2$ ciclos na sua execução. Para melhorar o desempenho dos testes e aumentar a legibilidade dos algoritmos, duas modificações foram feitas na arquitetura. A primeira modificação é o acréscimo de um dispositivo de teste de igualdade com zero, ou seja, o operador OU-inclusivo. Esta porta OU, denominada NZ (não zero ou *not zero*), tem como entrada o conteúdo do barramento principal de 4 *bits* e a sua saída é ligada ao barramento principal de 1 *bit*. Esta modificação tem por objetivo facilitar as comparações. A outra modificação refere-se à computação dos diversos resultados parciais. A fim de facilitar esta

tarefa, foi acrescentada uma unidade lógica de 1 *bit*, denominada LK, com duas entradas: uma associada ao barramento principal de 1 *bit* e a outra ligada à saída do registrador de máscara K. O resultado da operação lógica alimenta o registrador de máscara. Com estas duas modificações simples, o algoritmo de comparação do número *x* com zero, apresentado anteriormente para a arquitetura original, fica:

$K = \overline{NZ(X_0)}$
para *i* variando de 1 até $n/4-1$
    $K = K$ and $\overline{NZ(X_i)}$

Este novo algoritmo, além de ser dois ciclos mais rápido do que o anterior, é bem mais claro, tornando a programação mais legível. Além disso, a unidade lógica torna dispensável o inversor do *bit* mais significativo do barramento principal de 4 *bits*, acrescentado à arquitetura durante a análise da divisão de inteiros, apresentada no capítulo anterior.

Outra dificuldade encontrada na implementação dos algoritmos de ponto-flutuante refere-se às operações de armazenamento e, principalmente, deslocamento de dados, contidas em laços condicionais. Durante a execução destes laços, alguns EPs satisfarão a condição de execução, efetuando portanto as operações de armazenamento ou deslocamento em registradores. Ao mesmo tempo, os EPs que não satisfizerem a condição do laço deverão manter intactos os conteúdos dos seus registradores. Os armazenamentos condicionais ainda são adaptáveis à arquitetura proposta, através do uso da unidade lógica e do somador, apenas com pequena perda de legibilidade do algoritmo e, em alguns casos, com alguma perda de desempenho. Já os deslocamentos condicionais são mais problemáticos. A sua adptação à arquitetura proposta sempre envolve perda de desempenho. A implementação de um deslocamento condicional segue os seguintes passos: armazenamento do valor atual do dado, execução dos deslocamentos e restauração condicional (equivalente a um armazenamento condicional) do valor inicial.

Para resolver este problema, foi criado um sinal de habilitação do banco de registradores A que, associado ao valor do registrador de máscara K, permite a inibição do carregamento ou deslocamento de dados no banco de registradores. Esta modificação tem custo muito baixo, resultando apenas num aumento de mais um *bit* na palavra de controle do EP.

Mais três pequenas modificações foram feitas durante o desenvolvimento dos algoritmos de soma e multiplicação de ponto-flutuante. Nenhuma delas é essencial para a criação dos algoritmos, porém todas têm um custo muito baixo e cada uma delas é responsável por uma pequena melhora no desempenho. A primeira modificação é a criação de um barramento global de 4 *bits*, responsável por transmitir um mesmo dado para toda a Matriz de Elementos Processadores. Este barramento global é ligado à entrada do barramento principal de 4 *bits* de

todos os EPs. A sua utilidade não se restringe às operações de ponto-flutuante. A criação de um barramento global também facilita a execução de instruções com operandos imediatos.

Outra modificação foi o acréscimo de uma ligação entre a saída do registrador de *carry* (C) e a entrada do barramento principal de 1 *bit*. Com isto, fica mais fácil armazenar o resultado do *carry* de uma operação ou utilizá-lo em uma operação lógica. Por último, foi criada uma ligação entre o barramento principal de 1 *bit* e a entrada serial esquerda do registrador de deslocamento SH, tornando o uso deste registrador mais flexível. No caso das operações de ponto-flutuante esta ligação facilita a geração de variáveis auxiliares que armazenam o resultado de testes envolvendo os operandos. Estas variáveis serão apresentadas mais adiante.

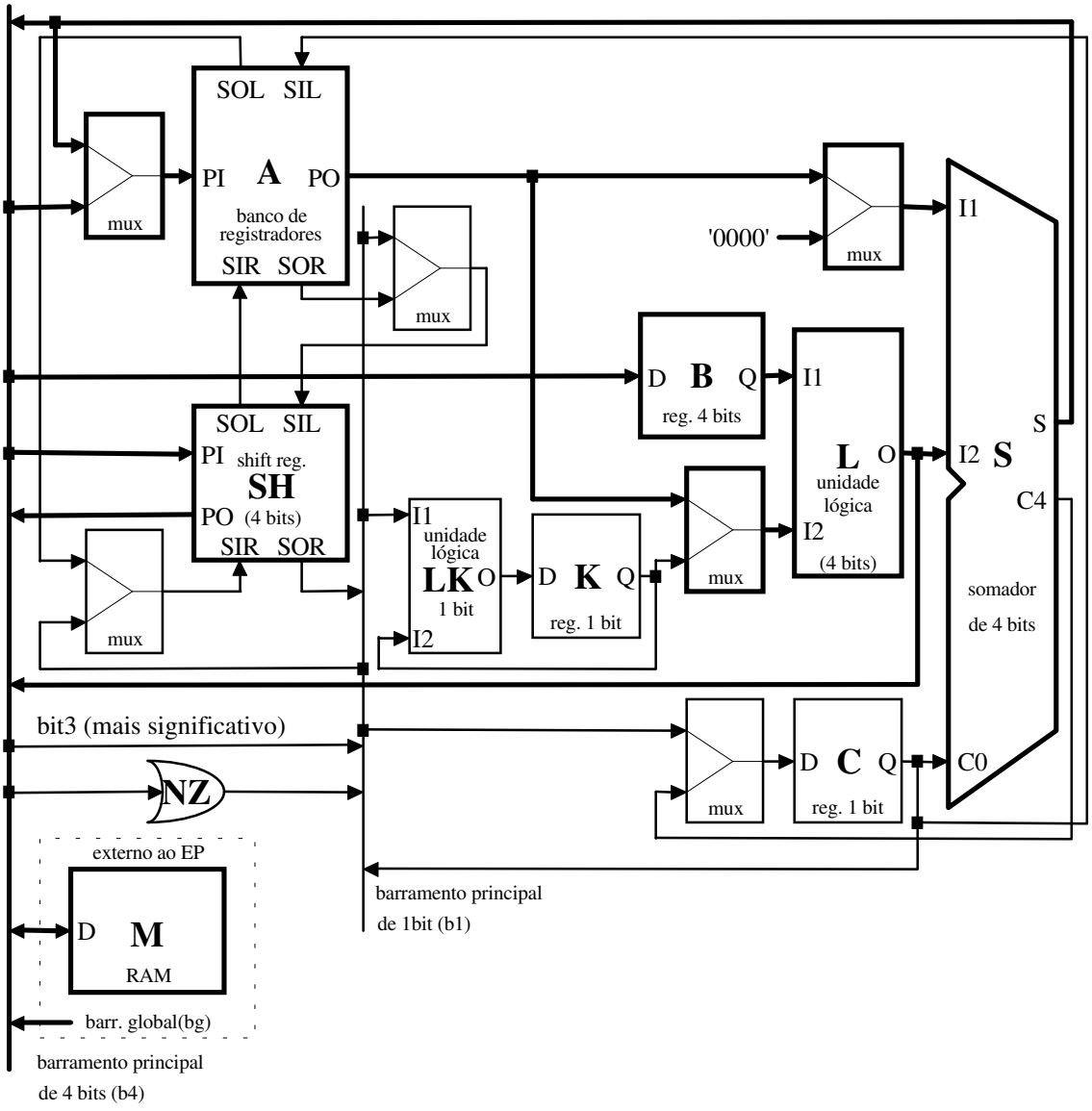

Figura IV.2 - EP de 4 bits com banco de registradores para operações de ponto-flutuante

Todas as alterações descritas foram feitas no EP com o objetivo de otimizar a implementação dos algoritmos de ponto-flutuante para a arquitetura EP4BR. O Elemento Processador resultante é apresentado na Figura IV.2.

IV.2.2 - Modificações Feitas em Relação ao Padrão IEEE

Antes de apresentar as modificações feitas no padrão, cabe introduzir alguns detalhes das especificações. As normas do padrão IEEE apresentam uma tabela de especificação de operações envolvendo dois operandos. Além disto, também é apresentado um formato para o armazenamento dos resultados parciais, ou seja, um formato de trabalho para o armazenamento dos resultados das operações antes dos testes, arredondamento, normalização, etc. Tanto a tabela quanto o formato são usados nas especificações de implementação das diversas operações aritméticas.

A Tabela IV.2 tem por objetivo facilitar as especificações de implementação dos casos envolvendo operandos especiais. Para esta tabela, *x* e *y* são os dois operandos, *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h* e *i* são os casos a serem especificados, *W* indica um número finito não nulo (normalizado ou denormalizado) e *M* indica a aplicação da regra de precedência do sistema (para a escolha do resultado entre *x* e *y*). As especificações das diversas operações aritméticas definem as diversas ações a serem tomadas para cada um dos nove casos da tabela. No sistema desenvolvido, não foi criada nenhuma regra de precedência para operações envolvendo dois operandos não numéricos. A escolha entre *x* e *y* vai depender da operação aritmética em questão.

Tabela IV.2 - Casos envolvendo operandos especiais

		y			
	x op *y*	±0	*W*	±∞	NaN
x	±0	*a*	*b*	*c*	*y*
	W	*d*	*e*	*f*	*y*
	±∞	*g*	*h*	*i*	*y*
	NaN	*x*	*x*	*x*	*M*

O formato para os resultados parciais apresentado pelas normas é dado pela Figura IV.3, onde *sgn* é o sinal, *exp* é o expoente e os demais *bits* representam a mantissa. Destes, *V* é o *bit* de estouro (*overflow*), *N* é o *bit* mais significativo, *L* é o *bit* menos significativo, *G* e *R* são os dois *bits* depois de *L* e *S* é um OU lógico de todos os *bits* posteriores.

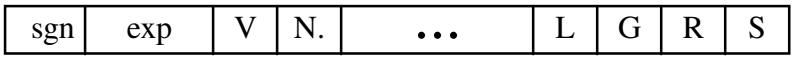

Figura IV.3 - Formato de armazenamento de resultados parciais

Como foi apresentado na seção IV.1, o padrão de arredondamento implementado por este trabalho foi o arredondamento para o mais próximo (*Round to Nearest*). A seguir é apresentada a definição deste padrão de arredondamento. Seja z o resultado com precisão infinita de uma operação aritmética. Sejam ainda $z1$ e $z2$ os números representáveis na precisão do formato IEEE mais próximos de z, tal que $z1 \leq z \leq z2$. Pelo padrão, z é arredondado para o número mais próximo a ele, entre os valores $z1$ e $z2$. Em caso de empate, ou seja, z está tão próximo de $z1$ quanto de $z2$, é escolhido aquele cujo *bit* menos significativo da mantissa é zero.

A dificuldade encontra-se exatamente na implementação da regra no caso de empate. Para testar o empate, três *bits* devem ser verificados: *G*, *R* e *S*. Aliás, a única função do *bit S* é auxiliar o arredondamento. Verificado o empate, ainda é preciso testar o *bit L*, a fim de examinar a necessidade de arredondamento para um número de magnitude maior. Todos estes testes, além da necessidade de manutenção constante do *bit S*, tornam o processo de arredondamento honeroso em termos de tempo de execução.

Pode-se resumir o padrão de arredondamento na necessidade ou não de incrementar a magnitude de z, após o truncamento do mesmo para o tamanho da precisão do destino. Por esta concepção, o algoritmo que descreve o arredondamento é:

```
se G == 0, então                          {z está mais próximo do menor número, em
magnitude}
    z = truncamento de z
senão
    se (R == 0) and (S == 0), então       {caso de empate}
        se L == 0, então
            z = truncamento de z
        senão
            z = (truncamento de z) + 1
    senão                                 {z está mais próximo do maior número, em magnitude}
        z = (truncamento de z) + 1
```

De forma mais resumida:

$$I = G \; and \; (L \; or \; R \; or \; S)$$
$$z = (\text{truncamento de } z) + I$$

Através de uma pequena modificação no padrão, é possível eliminar as quatro operações lógicas utilizadas no cálculo de *I* e a manutenção do *bit S* durante toda a execução da operação aritmética. Para isto, basta alterar a regra no caso de empate, optando por arredondar para o maior número em magnitude, entre $z1$ e $z2$. Com esta alteração, a execução do arredondamento fica reduzida a:

$$z = (\text{truncamento de } z) + G$$

Esta mesma modificação em relação ao padrão IEEE é adotada pelo MasPar [Batc80].

Uma questão ainda mais importante em termos de desempenho do EP refere-se ao formato denormalizado. O uso deste tipo de formato requer processos de normalização e denormalização de operandos e resultados durante a execução de operações aritméticas. Uma operação de normalização ou denormalização envolve deslocamentos de até 24 *bits* na mantissa e até 24 incrementos ou decrementos no expoente. Como já foi dito anteriormente, numa arquitetura SIMD, deve-se considerar o pior caso para calcular o tempo de execução. Sendo assim, operações de normalização e denormalização, conseqüências diretas da simples existência do formato denormalizado, acarretam num grande custo no desempenho de todas as operações aritméticas.

Além disto, deve-se considerar algumas desvantagens do formato denormalizado. O primeiro é a descontinuidade do intervalo de representação do formato *single*. o menor número em magnitude representável no formato normalizado é $2^{-126} \times 1{,}000\cdots00$ e o maior número em magnitude representável no formato denormalizado é $2^{-127} \times 0{,}111\cdots11$. Logo, um número contido, em magnitude, no intervalo $[2^{-127} \times 1{,}000\cdots00, 2^{-127} \times 1{,}111\cdots11]$ não é representável, embora números maiores ou menores do que ele o sejam. Por fim, uma operação envolvendo um operando denormalizado tem uma boa probabilidade de ter o mesmo resultado final que a mesma operação tendo como operando o valor zero.

Por todos estes motivos, eliminou-se o formato denormalizado nesta implementação. Da mesma forma que a alteração anterior, esta também é adotada pelo MasPar.

IV.2.3 - Considerações Gerais de Implementação

Para facilitar a apresentação dos algoritmos desenvolvidos, foram criadas algumas notações. Seja *x* um número representado em ponto-flutuante. Então:

- *Sx* é o campo *S* do formato *single*, ou seja, é o sinal da mantissa de *x*.
- *Ex* é o campo *E* do formato *single*, ou seja, é o expoente de *x* representado em notação por excesso.
- *Fx* é o campo *F* do formato *single*, ou seja, é a magnitude da parte fracionária da mantissa de *x*.
- *Mx* é a concatenação (1,*Fx*), ou seja, é a magnitude da mantissa de *x*, caso *x* seja um número normalizado.

Em todas as especificações de operações aritméticas apresentadas pelo padrão IEEE são definidas ações simples para a maioria dos casos envolvendo operandos especiais e apenas uma seqüência de ações mais complexa, normalmente para o caso *e* da Tabela IV.2. Ou seja, apenas

quando os dois operandos são números normalizados a operação é realmente executada. Para os demais casos, o resultado é quase imediato. Sendo assim, nos algoritmos desenvolvidos neste trabalho, inicialmente é calculado o resultado para os casos de solução simples e este resultado é armazenado na memória externa. A seguir é executada a seqüência de ações mais complexa que define a operação, independente do caso, gerando um novo resultado, armazenado nas variáveis auxiliares S', E' e M'. Desta forma, esta seqüência de ações, que já é complexa por si só, não precisa se preocupar se os operandos em questão necessitam deste processamento. Este problema é adiado para o momento do armazenamento. Se o caso em questão exige a execução da operação, o novo resultado é armazenado sobre o anterior. Caso contrário, o resultado anterior é preservado. Normalmente, durante o cálculo do novo resultado, é necessário o armazenamento de parte dos resultados parciais (S' e E') na memória externa, visto que a memória interna do EP é muito pequena.

A manipulação de dados no formato *single* não é uma tarefa fácil para um processador com registradores e memória com largura de 4 *bits*. Ao dividir um número *x*, representado no formato *single*, em palavras de 4 *bits*, da mesma forma como ele estará armazenado na memória, será encontrado o resultado apresentado na Figura IV.4.

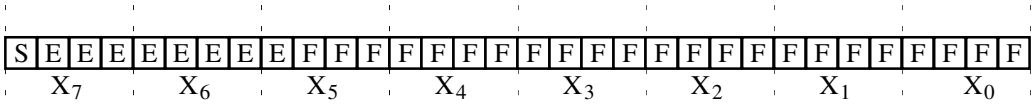

Figura IV.4 - Armazenamento do formato *single* numa memória com largura de 4 *bits*

Para manipular o expoente é necessária a leitura de três palavras (X_7, X_6 e X_5), embora o expoente seja representado por apenas 8 *bits*. A solução para a manipulação do expoente é carregar as três palavras mais significativas no banco de registradores A e a seguir deslocar o banco de registradores uma posição para a esquerda. Desta forma o expoente ficará alinhado em apenas dois registradores do banco.

Já para manipular a mantissa, é necessário substituir o *bit* mais significativo da palavra X_5 por zero (para trabalhar apenas com a parte fracionária) ou por um (para trabalhar com a mantissa completa). Nesta operação, o barramento global pode ser bastante útil. Para a substituição por zero basta executar um E lógico com a palavra '0111' e para a substituição por um basta executar um OU-inclusivo lógico com a palavra '1000'.

Durante a execução das operações aritméticas, normalmente o banco de registradores A é utilizado no armazenamento dos resultados parciais da mantissa. Comparando com o formato para este tipo de armazenamento sugerido pelo padrão IEEE: o *bit V* corresponde ao registrador de *carry* (C), o *bit N* corresponde ao *bit* mais significativo da palavra A[7], o *bit L* corresponde ao *bit* menos significativo da palavra A[2] e o *bit G* corresponde ao *bit* mais significativo da palavra A[1].

Os resultados parciais do expoente normalmente são armazenados numa variável auxiliar da memória com 12 *bits*. É abstraído o fato de o expoente estar representado em notação por excesso. Ele é tratado como uma variável inteira de 12 *bits* representada em complemento a dois, cujos valores válidos encontram-se no intervalo [0,255]. Durante a execução da operação, o algoritmo não se preocupa em verificar se a variável encontra-se neste intervalo. Isto só é feito nos testes de *underflow* e *overflow*. O uso da representação em complemento a dois tem por objetivo facilitar as operações sobre o expoente. Com este armazenamento, o teste de *underflow* ($E < 0$) resume-se na verificação do sinal do expoente (*bit* mais significativo). Já o teste de *overflow* ($E > 255$) resume-se na verificação da palavra mais significativa. O arredondamento também é uma tarefa bastante simples, considerando-se a modificação feita na sua padronização. Considere uma operação em processamento, onde os resultados parciais da mantissa e do expoente estão armazenados respectivamente no banco de registradores A e na variável auxiliar E'. Os procedimentos de arredondamento, teste de *underflow*, *overflow* são executados por:

$C = A[1]_3$
$(C, A[7..2]) = A[7..2] + C$ {arredondamento}
se $C == 1$, então
    $C = 0, A = A$ *shift right* $1, E' = E' + 1$
se $(E'_2)_3 == 1$, então {*underflow*}
    $E' = 0, A = 0$
se $E'_2 \neq 0$, então {*overflow*}
    $E' = 255, A = 0$

Para a análise da implementação de algoritmos de operações em ponto-flutuante para o EP de 4 *bits* com banco de registradores (EP4BR) foram escolhidas as operações de soma e multiplicação. A opção por estas duas operações visa mostrar que a maior dificuldade na implementação de algoritmos de ponto-flutuante não se encontra na execução da operação em si, mas sim no tratamento dos casos especiais, no alinhamento, na normalização, etc. A maior parte do tempo de processamento é gasto nestas etapas periféricas do cálculo, principalmente se analizarmos operações aparentemente simples como a soma.

IV.3 - Soma em Ponto-Flutuante

Ao contrário do que ocorre com os números inteiros, para os números em ponto-flutuante a soma não é uma operação trivial. Mesmo entre números normalizados, antes de efetuar a soma das mantissas é necessário alinhar as vírgulas, assim como numa operação realizada manualmente. Além disto, como o padrão IEEE representa a mantissa em sinal e magnitude, os sinais dos operandos devem ser analisados antes de somar ou subtrair as magnitudes das mantissas. Estas pequenas inconveniências tornam a operação de soma mais complexa do que a de multiplicação, embora não mais lenta.

Assim como para as demais operações, o padrão IEEE define o resultado da operação de soma em cada um dos nove casos de operandos especiais, como é mostrado abaixo. Considere a soma $z = x + y$.

caso ***a***: $z = +0$
casos ***c***, ***f***: $z = y$
casos ***g***, ***h***: $z = x$
caso ***i***: se os sinais de x e y são iguais, então
$z = x$
senão
$z = \text{NaN}$
caso ***b***: $z = y$
caso ***d***: $z = x$
caso ***e***: (1) Alinhar a vírgula dos dois operandos, denormalizando o de menor expoente, até que os dois expoentes sejam iguais.
Somar/subtrair as magnitudes.
(2) Em caso de soma de magnitudes:
Se $V == 1$, então
desloque a mantissa uma posição para a direita e incrementar o expoente.
(3) Em caso de subtração de magnitudes:
Normalizar o resultado.
Se não foi possível normalizar, então
$z = +0$
(4) Testar *underflow*, arredondar e testar *overflow*.

Analisando o primeiro passo do caso *e*, que é de longe o caso de solução mais complexa, identifica-se duas questões: alinhar os operandos e efetuar a soma das mantissas. O alinhamento se divide em duas etapas: identificação do operando de menor expoente e deslocamento da mantissa do mesmo. Por se tratarem de mantissas representadas em sinal e magnitude, antes de efetuar a soma é necessário comparar os sinais dos operandos. Em caso de sinais iguais, a manutenção do sinal acrescida de uma simples soma de magnitudes resolve o problema. Já em caso de sinais contrários, é preciso inicialmente identificar o operando de maior magnitude. O sinal deste prevalecerá. Resta então subtrair a magnitude do menor da magnitude do maior. O algoritmo que descreve estes procedimentos é apresentado abaixo:

se $Ex \geq Ey$, então
$Ez = Ex, My = My \; shift \; right \; (Ex - Ey)$
senão
$Ez = Ey, Mx = Mx \; shift \; right \; (Ey - Ex)$
se $Sx == Sy$, então
$Sz = Sx, Mz = Mx + My$
senão
se $Mx \geq My$
$Sz = Sx, Mz = Mx - My$
senão
$Sz = Sy, Mz = My - Mx$

Considerando-se que este algoritmo será processado por uma arquitetura SIMD, serão executados: três comparações (expoente, sinal e mantissa), duas atribuições e uma subtração de expoentes, dois deslocamentos de mantissas, três atribuições de sinal, uma soma e duas subtrações de mantissas. Esta multiplicidade de atribuições, deslocamentos e somas se deve às operações condicionais.

Com o objetivo de diminuir as operações condicionais, tentou-se pelo menos unificar os deslocamentos e as somas, tornando-os operações não condicionais. Observe que saber quem será deslocado depende exclusivamente de saber quem tem o menor expoente. Para definir se as magnitudes das mantissas serão somadas ou subtraídas basta comparar os sinais. Em caso de soma, a ordem dos operandos é indiferente. Já em caso de subtração, a ordem depende da comparação das mantissas após o alinhamento. Ora, se um operando tiver o expoente maior do que o outro, após o alinhamento obviamente a magnitude da sua mantissa também será maior. Já para expoentes iguais, é necessária a comparação das magnitudes das mantissas. Pode-se então resolver os problemas de alinhamento e soma de forma conjunta. O maior número em magnitude não será alinhado, seu expoente e seu sinal prevalecerão e ele será o primeiro operando da soma ou subtração. Já o menor número em magnitude será alinhado (caso os expoentes sejam diferentes) e ele será o segundo operando da soma ou subtração. Se os sinais forem iguais, os operandos serão somados, caso contrário serão subtraídos, como mostra o algoritmo apresentado abaixo:

$DE = Ex - Ey$
$DF = Fx - Fy$
se $DE > 0$ *or* $(DE == 0$ *and* $DF \geq 0)$, então
    $Sz = Sx, Ez = Ex, Mz = Mx, M' = My$
senão
    $Sz = Sy, Ez = Ey, Mz = My, M' = Mx$
$M' = M'$ *shift right* $|DE|$
se $Sx == Sy$, então
    $sinal = +1$
senão
    $sinal = -1$
$Mz = Mz + sinal \times M'$

Para este algoritmo são executadas: uma subtração de expoentes e outra de partes fracionárias de mantissas, quatro comparações e quatro atribuições de sinais, duas atribuições de expoente, quatro atribuições de mantissa, um cálculo de módulo, um deslocamento e uma soma ou subtração de mantissas. Foram eliminadas as multiplicidades de deslocamentos, somas e subtrações, porém as atribuições continuam duplicadas. Embora possa parecer que não houve um ganho significativo de desempenho com este novo algoritmo, uma análise dos demais casos de operandos especiais vai mostrar que a primeira parte do algoritmo apresentado (onde ocorrem as duplicidades) é capaz de resolver quase todos os casos.

Abstraindo o caso *e* e resumindo a definição do resultado dada pelo padrão IEEE para cada um dos demais casos, chega-se a Tabela IV.3:

Tabela IV.3 - Resultado da soma para os casos de operandos especiais

x \ y		±0 E=0,F=0	W E>0	±∞ E=255,F=0	NaN E=255,F>0
±0	E=0,F=0	**+0**	y	y	y
W	E>0	x		y	y
±∞	E=255,F=0	x	x	***x* ou NaN**	y
NaN	E=255,F>0	x	x	x	x ou y

Utilizando um algoritmo bem parecido com o apresentado para o caso *e*, apenas não efetuando a soma ou subtração de magnitudes de mantissas, obtém-se um resultado bem próximo do desejado, como mostra a Tabela IV.4, apresentada a seguir:

Tabela IV.4 - Resultado da soma utilizando o algoritmo inicial

x\y	±0	W	±∞	NaN
±0	***x***	y	y	y
W	x		y	y
±∞	x	x	***x***	y
NaN	x	x	x	x ou y

Há apenas possibilidade de erro nos casos *a* e *i*. Para chegar ao resultado correto, basta apenas acertar o sinal do zero no caso *a* e testar se os sinais são opostos no caso *i*. Acrescentando os testes que resolvem estes dois problemas, chega-se ao seguinte algoritmo:

$DE = Ex - Ey$
$DF = Fx - Fy$
se $DE > 0$ *or* $(DE == 0$ *and* $DF \geq 0)$, então
    $Sz = Sx, Ez = Ex, Fz = Fx, M' = My$
senão
    $Sz = Sy, Ez = Ey, Fz = Fy, M' = Mx$
M' = M' *shift right* $|DE|$
se $Sx == Sy$, então
    $sinal = +1$
senão
    $sinal = -1$
se $(Ex == 0)$ *and* $(Ey == 0)$, então
    $Sz = 0$
se $(Ex == 255)$ *and* $(Ey == 255)$ *and* $(Fx == 0)$ *and* $(Fy == 0)$ *and* $(\mathrm{Sx} \neq \mathrm{Sy})$, então
    Fz = código de soma de $+\infty$ com $-\infty$
$M' = (1, Fz) + sinal \times M'$

Resolvido o primeiro passo, resta então analisar os passos (2), (3) e (4) do caso *e*. O segundo passo é bastante simples. Ele pode ser resolvido por:

se $(sinal == +1)$ *and* $(M'_V == 1)$, então
    $M' = M'$ *shift right* 1
    $E' = Ez + 1$
senão
    $E' = Ez$

Ou, de maneira mais apropriada a uma arquitetura SIMD:

$K = (sinal == +1)$ *and* $(M'_V == 1)$
$M' = M'$ *shift right* K
$E' = Ez + K$

Já o terceiro passo, embora tenha uma estrutura simples, requer um maior cuidado, visto que ele envolve até 24 deslocamentos da mantissa e 24 decrementos do expoente. A maneira mais óbvia de resolvê-lo é:

para i variando de 1 até 24
    se $M'_N == 0$, então
        $M' = M'$ *shift left* 1
        $E' = E' - 1$

Esta solução, embora simples, não é nada eficiente. Ela executa 24 comparações, 24 deslocamentos de 1 *bit* e 24 subtrações de 12 *bits* (tamanho da variável auxiliar de armazenamento do expoente). Pode-se utilizar a facilidade oferecida pelo operador NZ para testar 4 *bits* de cada vez, ao invés de apenas um. Apenas para os últimos 4 *bits* é que o teste individual se fará necessário. Considerando-se que a variável auxiliar M' estará armazenada no banco de registradores A, com o *bit N* na posição $A[7]_3$, chega-se a:

para *i* variando de 1 até 5
    se $A[7] == 0$, então
        $A = A$ *shift left* 4
        $E' = E' - 4$
para *i* variando de 1 até 4
    se $A[7]_3 == 0$, então
        $A = A$ *shift left* 1
        $E' = E' - 1$

Com esta modificação passam a ser necessárias apenas 9 comparações, 5 deslocamentos de 4 *bits*, 4 deslocamentos de 1 *bit* e 9 subtrações de 12 *bits*. As comparações foram otimizadas, porém as subtrações ainda são um fator limitador de desempenho, visto que cada uma delas envolve 12 *bits*. O número máximo de deslocamentos (24) necessita de 5 *bits* para o seu armazenamento, o que levaria a operações de 8 *bits*, considerando-se que o processador trabalha com múltiplos de 4 *bits*. Isso permitiria totalizar o número de deslocamentos numa

variável auxiliar de 8 *bits* e só ao final efetuar uma subtração de 12 *bits*. Melhor do que isto, aproveitando a separação do algoritmo anterior em dois laços, pode-se utilizar duas variáveis auxiliares de 4 *bits*: uma para totalizar os deslocamentos de 4 *bits* e outra para totalizar os deslocamentos de 1 *bit*. Desta forma, o algoritmo fica:

$B1 = 0, B2 = 0$
para i variando de 1 até 5
    $K = \overline{NZ}(A[7])$
    se $K == 1,$ então $A = A$ *shift left* 4
    $B1 = B1 + K$
para i variando de 1 até 4
    $K = \overline{A[7]_3}$
    se $K == 1,$ então $A = A$ *shift left* 1
    $B2 = B2 + K$
$E' = E' - (B1$ *shift left* $2) - B2$

Por este algoritmo passam a ser necessárias duas atribuições de 4 bits, 9 comparações, 5 deslocamentos de 4 *bits*, 4 deslocamentos de 1 *bit*, um deslocamento de 2 *bits*, 9 somas de 4 *bits* e duas subtrações de 12 *bits*.

O quarto passo é comum a todos os algoritmos de ponto-flutuante e já foi analisado na seção IV.2.3. A última etapa do algoritmo consiste em sobrepor os resultados parciais armazenados na memória com E' e M' se for o caso *e*, ou deixá-los inalterados em se tratando dos demais casos. Reunindo todos os passos analisados, chega-se a base do algoritmo de soma:

{ tratamento dos casos especiais }
$DE = Ex - Ey$
$DF = Fx - Fy$
se $DE > 0$ *or* $(DE == 0$ *and* $DF \geq 0)$, então
    $Sz = Sx, Ez = Ex, Fz = Fx, M' = My$
senão
    $Sz = Sy, Ez = Ey, Fz = Fy, M' = Mx$
$M' = M'$ *shift right* $|DE|$
se $Sx == Sy$, então
    $sinal = +1$
senão
    $sinal = -1$
se $(Ex == 0)$ *and* $(Ey == 0)$, então
    $Sz = 0$
se $(Ex == 255)$ *and* $(Ey == 255)$ *and* $(Fx == 0)$ *and* $(Fy == 0)$ *and* $(\mathrm{Sx} \neq \mathrm{Sy})$, então
    $Fz =$ código de soma de $+\infty$ com $-\infty$
{ soma/subtração das mantissas }
$M' = (1, Fz) + sinal \times M'$
$K = (sinal == +1)$ *and* $(M'_V == 1)$
$M' = M'$ *shift right* K
$E' = Ez + K$

{ normalização }
$B1 = 0, B2 = 0$
para i variando de 1 até 5
    $K = (M'_{N..N-3} = 0)$
    se $K == 1$, então $M' = M'$ *shift left* 4
    $B1 = B1 + K$
para i variando de 1 até 4
    $K = \overline{M'_N}$
    se $K == 1$, então $A = A$ *shift left* 1
    $B2 = B2 + K$
$E' = E' - (B1$ *shift left* $2) - B2$
{ arredondamento }
$M'_{V..L} = M'_{V..L} + M'_G$
$K = M'_V$
$M' = M'$ *shift right* K
$E' = E' + K$
{ teste de *underflow* }
se $(E'_2)_3 == 1$, então
    $E' = 0, M' = 0$
{ teste de *overflow* }
se $E'_2 > 0$, então
    $E' = 255, A = 0$
{ armazenamento final }
se $Ex \neq 0$ *and* $Ey \neq 0$ *and* $Ex \neq 255$ *and* $Ey \neq 255$, então
    $Ez = E'$
    $Fz = F'_{N-1..L}$

Para a implementação deste algoritmo na arquitetura apresentada, foi necessária a utilização de algumas variáveis auxiliares armazenadas na memória externa. As variáveis *DE* e E' são um exemplo e podem ser armazenadas na mesma posição de memória, já que a segunda só é criada depois que a primeira não é mais necessária. A variável *DF* não precisa ser armazenada, pois apenas o seu sinal é utilizado e mesmo este tem uso apenas imediato, dispensando o armazenamento na memória externa.

Existem alguns resultados de testes que são utilizados em mais de uma ocasião ao longo do cálculo da soma, como por exemplo os testes de expoente nulo dos operandos. Para evitar repetir os testes, o que resultaria numa perda de desempenho, os resultados são armazenados na memória externa em duas variáveis de 4 *bits* denominadas *STATUS1* e *STATUS2*. Cada *bit* destas variáveis guarda o resultado de um teste. Esta mesma estratégia foi utilizada no desenvolvimento do algoritmo de multiplicação. A descrição das duas variáveis é apresentada na Figura IV.5, onde X representa um valor indefinido:

STATUS1	*Sx xor Sy*	X	X	*Fx==0 e Fy==0*
STATUS2	*Ex=255*	*Ey==255*	*Ex==0*	*Ey==0*

Figura IV.5 - Descrição das variáveis STATUS1 e STATUS2 na soma

O tempo gasto com testes de sinais, expoentes e mantissas dos operandos não é desprezível. Sendo assim, quando possível, estes testes são feitos em paralelo com outras operações, como no caso de ($Fx = 0$ e $Fy = 0$), que é calculado simultaneamente à subtração que gera *DF*.

A fim de facilitar o armazenamento e a utilização dos resultados parciais guardados na memória externa, evitando o processo de separação do sinal, expoente e mantissa necessário no formato *single*, os resultados parciais utilizam outro formato de armazenamento. O sinal é armazenado numa variável auxiliar denominada S'. Ele está representado no *bit* mais significativo desta variável. Livre do sinal, o expoente é armazenado nas duas palavras mais significativas do resultado. Da mesma forma, livre do *bit* menos significativo do expoente, a mantissa é representada não só pela sua parte fracionária, mas também pela sua parte inteira, que para efeitos de armazenamento será sempre igual a um. Isto tem por objetivo facilitar a soma das mantissas, já que está operação envolve as partes inteiras e só é válida quando os dois operandos são números normalizados, isto é, tem parte inteira igual a um. O resultado só é convertido para o formato *single* ao final do algoritmo, no momento em que é executado o armazenamento final da soma.

Na maior parte do tempo, o banco de registradores A é utilizado para o armazenamento da variável M'. Isto gera um pequeno problema na operação de soma ou subtração de mantissas, mais precisamente no caso de ser necessária a subtração de mantissas. Ocorre que a arquitetura em questão só permite subtrair o conteúdo do banco de registradores A do conteúdo de uma posição qualquer da memória externa, mas não o inverso como é necessário. Para resolver este problema, no caso dos sinais dos operandos serem contrários, a subtração das mantissas é substituída por uma inversão de sinal de M' seguida de uma soma das mantissas. A inversão do sinal de M' é executada dentro do banco de registradores, antes mesmo do alinhamento. Por isto, durante o alinhamento, quando a mantissa é deslocada para a direita, deve-se tomar especial cuidado com o sinal, que deve ser preservado.

A questão do alinhamento também merece atenção em relação ao seu desempenho, visto que ela envolve uma grande série de deslocamentos, definida pela variável *DE*. Se fosse considerado o valor máximo possível de *DE*, poderiam ser executados 256 deslocamentos, o que seria um absurdo, já que todos os *bits* seriam eliminados nos primeiros 28 deslocamentos (tamanho da mantissa utilizada para o armazenamento de resultados parciais), tornando os demais 228 deslocamentos absolutamente supérfluos, esbanjando tempo de processamento do algoritmo de soma. A solução utilizada limitou em 31 o total de deslocamentos.

Para entender melhor o valor deste limite, considere por hipótese que *DE* é representado por apenas cinco *bits*. Neste caso, para resolver o alinhamento, seriam executados deslocamentos condicionais a cada um destes cinco *bits*: um deslocamento condicional ao *bit*

menos significativo, dois deslocamentos condicionais ao segundo *bit* menos significativo, quatro deslocamentos condicionais ao terceiro *bit*, oito deslocamentos condicionais ao quarto *bit* e dezesseis deslocamentos condicionais ao quinto *bit*. Esta solução resolveria o alinhamento para uma variável *DE* representada por cinco *bits*. Seriam executados no máximo 31 deslocamentos, já que o maior número representável por cinco *bits* é 31. Como a variável *DE* é representada por oito *bits*, resta considerar os três *bits* mais significativos. Se todos forem nulos, a solução apresentada resolve o problema. Caso contrário, *DE* é maior do que 31. Qualquer que seja o valor maior do que 31, a execução de pelo menos 28 deslocamentos já resolve o alinhamento. Sendo assim, inicialmente é verificado se *DE* é maior do que 31, através do teste dos três *bits* mais significativos desta variável. Em caso afirmativo, todos os 31 deslocamentos serão efetuados. Para isto, é executado um OU-inclusivo entre o resultado do teste e os quatro *bits* menos significativos de *DE*. Depois são executados deslocamentos condicionais a cada um destes quatro *bits*, resolvendo os 15 primeiros deslocamentos. Por último, são executados 16 deslocamentos *DE* for maior do que 16, ou seja, se pelo menos um dos quatro *bits* mais significativos de *DE* for diferente de zero. Neste caso, não é utilizada a operação de deslocamento condicional disponível para o banco de registradores, pois ela demanda 16 ciclos para a sua execução. Ao invés disto, cada registrador de 4 bits é transferido diretamente para a sua nova posição, num total de 11 ciclos. Abaixo é apresentado o algoritmo de alinhamento das mantissas. Considere que M' está armazenado no banco de registradores A e que o seu sinal está no registrador de carry C.

$K = NZ(DE_{7..5})$
$DE_{3..0} = DE_{3..0}$ *or* K
$K = DE_0$
se $K == 1,$ então$A = (C, A)$ *shift right* 1
$K = DE_1$
se $K == 1,$ então$A = (C, A)$ *shift right* 1
se $K == 1,$ então$A = (C, A)$ *shift right* 1
$K = DE_2$
para *i* variando de 1 até 4
    se $K == 1,$ então$A = (C, A)$ *shift right* 1
$K = DE_3$
para *i* variando de 1 até 8
    se $K == 1,$ então$A = (C, A)$ *shift right* 1
$K = NZ(DE_{7..4})$
se $K == 1,$ então$A[3..1] = A[7..5]$
se $K == 1,$ então$A[7..4] = C$

O algoritmo desenvolvido (Apêndice A) processa a soma em 219 ciclos. Destes, mais de um terço (77 ciclos) é gasto com alinhamentos. O cálculo de *DF*, *DE*, *STATUS1* e *STATUS2* requer 43 ciclos. O cálculo dos resultados parciais leva 31 ciclos e o segundo operando da soma de mantissas é obtido em 22 ciclos. A soma das mantissas propriamente dita é efetuada

em apenas 6 ciclos. As operações de arredondamento e teste de *underflow* e *overflow* levam 22 ciclos e a formatação e armazenamento final do resultado gastam 18 ciclos.

IV.4 - Multiplicação em Ponto-Flutuante

Os problemas encontrados no desenvolvimento do algoritmo de multiplicação em ponto-flutuante são mais simples do que os da soma em ponto-flutuante e da multiplicação de inteiros. Até mesmo a questão da combinação dos sinais encontrada na análise da multiplicação de inteiros representados em complemento a dois inexiste no caso dos números em ponto-flutuante, visto que a mantissa é representada em sinal e magnitude.

O padrão IEEE define os seguintes resultados para a operação de multiplicação $p = x \times y$ em cada um dos nove casos de operandos especiais:

$Sp = Sx$ xor Sy
casos ***a***, ***b***, ***d***: $p = 0$
casos ***c***, ***g***: $p = \text{NaN}$
casos ***f***, ***h***, ***i***: $p = \infty$
caso ***e***: (1) $Ep = Ex + Ey - 127$
Multiplicar as mantissas
(2) Se $V == 1$, então deslocar a mantissa uma posição para a direita e incrementar o expoente
(3) Testar *underflow*, arredondar e testar *overflow*.

Abstraindo a princípio o caso *e*, pode-se inicialmente calcular o resultado dos demais casos e armazená-lo na memória externa. Para facilitar a análise dos oito casos restantes foram criadas duas tabelas: a Tabela IV.5 contém o expoente dos resultados e a Tabela IV.6 contém a parte fracionária da mantissa dos resultados. Para a Tabela IV.6, considere *H* o código que representa uma multiplicação de zero por infinito.

Tabela IV.5 - Expoente do resultado da multiplicação para os casos de operandos especiais

x \ y		±0 E=0,F=0	W E>0	±∞ E=255,F=0	NaN E=255,F>0
±0	E=0,F=0	E=0	E=0,	E=255	E=255
W	E>0	E=0		E=255	E=255
±∞	E=255,F=0	E=255	E=255	E=255	E=255
NaN	E=255,F>0	E=255	E=255	E=255	E=255

Analisando a Tabela IV.5, conclui-se facilmente que o cálculo parcial do expoente pode ser executado por:

$Ep = 0$
se $(Ex == 255)$ *or* $(Ey == 255)$, então
    $Ep = 255$

Ou, de maneira mais adequada a uma arquitetura SIMD:

$K = (Ex == 255)$ *or* $(Ey == 255)$
$Ep = K$

Tabela IV.6 - Mantissa do resultado da multiplicação para os casos de operandos especiais

x \ y		±0 E=0,F=0	W E>0	±∞ E=255,F=0	NaN E=255,F>0
±0	E=0,F=0	F=0	F=0	F=H	F=Fy
W	E>0	F=0		F=0	F=Fy
±∞	E=255,F=0	F=H	F=0	F=0	F=Fy
NaN	E=255,F>0	F=Fx	F=Fx	F=Fx	F=Fy

Já o cálculo parcial da mantissa não é tão simples, é necessária uma série de testes e existem quatro possibilidades de resultado diferentes. Portanto, este cálculo demanda mais tempo maior de execução. A cálculo da mantissa pode ser executado por:

$Fp = 0$
se $(Ex == 255)$ *and* $(Fx \neq 0)$, então
    $Fp = Fx$
se $(Ey == 255)$ *and* $(Fy \neq 0)$, então
    $Fp = Fy$
se
    $((Ex == 255)$ *and* $(Fx == 0)$ *and* $(Ey == 0))$ *or*
$((Ey == 255)$ *and* $(Fy == 0)$ *and* $(Ex == 0))$, então
    $Fp =$ código de multiplicação de 0 por ∞

Ou, de maneira mais adequada a uma arquitetura SIMD:

$K = (Ex == 255)$ *and* $(Fx \neq 0)$
    $Fp = Fx$ *and* K
$K = (Ey == 255)$ *and* $(Fy \neq 0)$
se $K == 1$, então $Fp = Fy$
$K = ((Ex == 255)$ *and* $(Fx == 0)$ *and* $(Ey == 0))$ *or*
    $((Ey == 255)$ *and* $(Fy == 0)$ *and* $(Ex == 0))$
se $K == 1$, então $Fp =$ código de multiplicação de 0 por ∞

Estas duas pequenas rotinas são suficientes para a obtenção do resultado parcial da multiplicação. Resolvida esta primeira etapa, resta analisar o caso *e*. Inicialmente é calculado o expoente, que é armazenado numa variável auxiliar de 12 *bits* denominada E'. Após o cálculo do expoente é necessário multiplicar as mantissas dos operandos. Porém, pelo padrão de armazenamento IEEE apenas a parte fracionária destas mantissas está guardada na memória.

Como no caso *e* a parte inteira dos dois operandos é sempre igual a um, não é muito difícil resolver este problema. Considere que a multiplicação das mantissas será efetuada através de 24 somas da mantissa de *x*, cada uma condicional a um *bit* da mantissa de *y*. A questão da parte inteira da mantissa do operando *y* é resolvida executando sempre a última soma, que é condicional a parte inteira da mantissa de *y*, já que esta é sempre igual a um. Já para obter a mantissa completa de *x*, a palavra mais significativa tem que ter o seu *bit* mais significativo (que armazena parte do expoente) igualado a um. A fim de não destruir o valor do operando, uma nova palavra mais significativa será calculada e armazenada em uma variável auxiliar denominada *D*. Em resumo, as somas serão efetuadas utilizando as cinco palavras menos significativas de *x* e a variável *D*. Sendo assim e considerando M' a variável auxiliar de armazenamento da mantissa do resultado da operação, o algoritmo de multiplicação das mantissas é:

$D = X_5$ or 1000 { iguala a parte inteira da mantissa de X a 1}
$SH = Y_0$
$K = SH_0, SH = SH \; shift \; right \; 1, C = 0$
$(C, M') = 0 + (D, X_{4..0}) \; and \; K + C$ { soma condicional ao *bit* 0 de Y_0 }
para *i* variando de 1 até 5 { para cada palavra da mantissa de *Y* }
    $K = SH_0, SH = SH \; shift \; right \; 1, M' = (C, M') \; shift \; right \; 1, C = 0$
    $(C, M') = M' + (D, X_{4..0}) \; and \; K + C$ { soma condicional ao *bit* 1 de Y_{i-1} }
    $K = SH_0, SH = SH \; shift \; right \; 1, M' = (C, M') \; shift \; right \; 1, C = 0$
    $(C, M') = M' + (D, X_{4..0}) \; and \; K + C$ { soma condicional ao *bit* 2 de Y_{i-1} }
    $K = SH_0, SH = Y_i, M' = (C, M') \; shift \; right \; 1, C = 0$
    $(C, M') = M' + (D, X_{4..0}) \; and \; K + C$ { soma condicional ao *bit* 3 de Y_{i-1} }
    $K = SH_0, SH = SH \; shift \; right \; 1, M' = (C, M') \; shift \; right \; 1, C = 0$
    $(C, M') = M' + (D, X_{4..0})$ and $K + C$ { soma condicional ao *bit* 0 de Y_i }
$K = SH_0, SH = SH \; shift \; right \; 1, M' = (C, M') \; shift \; right \; 1, C = 0$
$(C, M') = M' + (D, X_{4..0}) \; and \; K + C$ { soma condicional ao *bit* 1 de Y_5 }
$K = SH_0, SH = SH \; shift \; right \; 1, M' = (C, M') \; shift \; right \; 1, C = 0$
$(C, M') = M' + (D, X_{4..0}) \; and \; K + C$ { soma condicional ao *bit* 2 de Y_5 }
$M' = (C, M') \; shift \; right \; 1, C = 0$
$(C, M') = M' + (D, X_{4..0}) + C$ { soma incondicional (parte inteira da mantissa de *Y*)}

Após a multiplicação das mantissas fica encerrado o primeiro passo do caso *e*. O segundo passo é bastante simples e parecido com o mesmo passo do caso *e* da soma em ponto-flutuante, apresentada na seção anterior. O algoritmo que resolve este passo é:

$K = M'_V$
$M' = M' \; shift \; right \; K$
$E' = E' + K$

O terceiro passo é comum a todos os algoritmos de ponto-flutuante e já foi analisado na seção IV.2.3. A última etapa do algoritmo consiste em sobrepor os resultados parciais

armazenados na memória com E' e M' se for o caso *e*, ou deixá-los inalterados em se tratando dos demais casos. Reunindo todos os passos analisados, chega-se a base do algoritmo de multiplicação:

{ cálculo do sinal }
$Sp = Sx \; xor \; Sy$
{ tratamento dos casos especiais }
$K = (Ex == 255) \; or \; (Ey == 255)$
$Ep = K$
$K = (Ex == 255) \; and \; (Fx \neq 0)$
$Fp = Fx \; and \; K$
$K = (Ey == 255) \; and \; (Fy \neq 0)$
se $K == 1$, então $Fp = Fy$
$K = ((Ex == 255) \; and \; (Fx == 0) \; and \; (Ey == 0)) \; or$
$((Ey == 255) \; and \; (Fy == 0) \; and \; (Ex == 0))$
se $K == 1$, então $Fp =$ código de multiplicação de 0 por ∞
{ cálculo do expoente }
$E' = Ex + Ey - 127$
{ multiplicação das mantissas }
$D = X_5$ or 1000
$SH = Y_0$
$K = SH_0, SH = SH \; shift \; right \; 1, C = 0$
$(C, M') = 0 + (D, X_{4..0}) \; and \; K + C$
para *i* variando de 1 até 5
$K = SH_0, SH = SH \; shift \; right \; 1, M' = (C, M') \; shift \; right \; 1, C = 0$
$(C, M') = M' + (D, X_{4..0}) \; and \; K + C$
$K = SH_0, SH = SH \; shift \; right \; 1, M' = (C, M') \; shift \; right \; 1, C = 0$
$(C, M') = M' + (D, X_{4..0}) \; and \; K + C$
$K = SH_0, SH = Y_i, M' = (C, M') \; shift \; right \; 1, C = 0$
$(C, M') = M' + (D, X_{4..0}) \; and \; K + C$
$K = SH_0, SH = SH \; shift \; right \; 1, M' = (C, M') \; shift \; right \; 1, C = 0$
$(C, M') = M' + (D, X_{4..0})$ and $K + C$
$K = SH_0, SH = SH \; shift \; right \; 1, M' = (C, M') \; shift \; right \; 1, C = 0$
$(C, M') = M' + (D, X_{4..0}) \; and \; K + C$
$K = SH_0, SH = SH \; shift \; right \; 1, M' = (C, M') \; shift \; right \; 1, C = 0$
$(C, M') = M' + (D, X_{4..0}) \; and \; K + C$
$M' = (C, M') \; shift \; right \; 1, C = 0$
$(C, M') = M' + (D, X_{4..0}) + C$
{ normalização }
$K = M'_V$
$M' = M' \; shift \; right \; K$
$E' = E' + K$
{ arredondamento }
$M'_{V..L} = M'_{V..L} + M'_G$
$K = M'_V$
$M' = M' \; shift \; right \; K$
$E' = E' + K$

{ teste de *underflow* }
se $(E'_2)_3 = 1$, então
    $E' = 0, M' = 0$
{ teste de *overflow* }
se $E'_2 > 0$, então
    $E' = 255, A = 0$
{ armazenamento final }
se $(Ex \neq 0)$ *and* $(Ey \neq 0)$ *and* $(Ex \neq 255)$ *and* $(Ey \neq 255)$, então
    $Ez = E'$
    $Fz = F'_{N-1..L}$

Para a implementação deste algoritmo na arquitetura EP4BR, foram utilizadas algumas variáveis auxiliares mapeadas na memória externa. São elas: o expoente auxiliar (E'), a palavra mais significativa da mantissa do operando *x* (*D*) e duas palavras para o armazenamento dos resultados de testes envolvendo os dois operandos (*STATUS1* e *STATUS2*). Estas duas últimas têm a mesma função das variáveis de mesmo nome utilizadas no algoritmo de soma, ou seja, evitar a repetição dos mesmos testes diversas vezes ao longo da operação. Na multiplicação este problema é ainda mais grave, visto que alguns resultados de testes são utilizados quatro vezes durante o cálculo. Da mesma forma que na soma, cada *bit* destas variáveis guarda o resultado de um teste, porém a definição de cada *bit* é diferente. Procurou-se otimizar o acesso aos *bits* mais usados, colocando-os nos extremos das palavras. A Figura IV.6 apresenta a descrição das duas variáveis, onde X representa um valor indefinido:

STATUS1	*Fy==0*	X	X	*Fx==0*
STATUS2	*Ey==255*	*Ey==0*	*Ex==0*	*Ex==255*

Figura IV.6 - Descrição das variáveis STATUS1 e STATUS2 na multiplicação

Para diminuir o tempo gasto no cálculo destas duas variáveis, sempre que possível procurou-se efetuar os testes simultaneamente a outras operações. A variável auxiliar M', que armazena resultados parciais da mantissa, foi mapeada no banco de registradores A e no registrador de *carry* C, conforme apresentado na seção IV.2.3.

Assim como na soma, o armazenamento deste resultado parcial não segue o formato *single* do padrão IEEE. Como, após esta primeira etapa do cálculo, o expoente só pode assumir dois valores: todos os *bits* iguais a zero (0) ou todos os *bits* iguais a um (255), é desnecessário o seu armazenamento em duas palavras de 4 *bits*. Sendo assim, o expoente é armazenado em apenas uma palavra (P_6) de quatro *bits*, deixando a palavra mais significativa (P_7) para o armazenamento do sinal. O sinal é representado no *bit* mais significativo desta palavra. As seis palavras reservadas para a mantissa guardam não só a sua parte fracionária como também um *bit* a mais (o mais significativo), que pode conter qualquer valor. Este

formato facilita o final do algoritmo de multiplicação, quando o resultado será escolhido entre o valor armazenado na variável *P* e o novo valor calculado.

O algoritmo desenvolvido (Apêndice A) processa a multiplicação em 273 ciclos. Destes, mais de 60% (169 ciclos) são gastos na multiplicação das mantissas. O cálculo das variáveis auxiliares *D*, *E′*, *STATUS1* e *STATUS2* demanda 36 ciclos. O cálculo dos resultados parciais leva 30 ciclos e o alinhamento e arredondamento são efetuados em 15 ciclos. Os testes de *underflow* e *overflow* e a formatação e armazenamento final do resultado gastam 23 ciclos.

IV.5 - Análise de Desempenho

Sem dúvida, tanto as modificações na arquitetura quanto as restrições ao padrão contribuíram sensivelmente no desempenho dos algoritmos desenvolvidos. Ainda assim, as regras do padrão IEEE são bastante complexas, envolvendo muitas operações condicionais, prevendo uma série de exceções, dificultando a criação de algoritmos mais rápidos.

A soma é o maior exemplo dos problemas encontrados no desenvolvimento de operações em ponto-flutuante. A soma das mantissas, aí incluindo o cálculo do menor operando e seu alinhamento, que deveria ser a essência do algoritmo de soma, consome apenas 59 dos 219 ciclos demandados por esta operação, ou seja, menos de 30% do tempo de execução.

Uma forma de aumentar o desempenho da arquitetura nas operações de ponto-flutuante seria a criação de registradores e unidades lógicas e aritméticas específicas para estas operações, gerando uma unidade própria para o cálculo dos expoentes e outra para o tratamento de exceções. Considerando a simplicidade da arquitetura de cada EP, esta possibilidade foi descartada, pois o seu custo seria muito elevado relativamente ao custo total do EP.

A Tabela IV.7 apresenta uma comparação entre os resultados obtidos por este trabalho e os resultados de arquiteturas comerciais. Mais uma vez o EP4BR perde apenas para o MasPar. O desempenho do EP4BR é apenas cerca de duas vezes pior do que o do MasPar, apesar deste possuir elementos na sua arquitetura voltados especialmente para as operações de ponto-flutuante, além de uma extensa memória interna.

Tabela IV.7 - Ciclos necessários à execução de operações em ponto flutuante com 32 *bits*

Operação	DAP510	MPP	RPA	MasPar	EP4BR
Soma/Subtração	1025	384	556	96	219
Multiplicação	1443	769	1718	216	273