Modelagem e Implementação em VHDL de Soma e Multiplicação em Ponto Flutuante de 32 Bits Segundo o Padrão IEEE-754

Cainã C. Trevisan[1], Clara D. H. Daru[1], Jean C. K. Diogo[1], João M. P. Filho[1], Roberto A. Hexsel[1]

[1]Departamento de Informática – Universidade Federal do Paraná – UFPR
Caixa Postal 19.081 – 81.531-980 – Curitiba – PR – Brazil

{cct11,cdhd12,jckd12,jmpf13,roberto}@inf.ufpr.br

Abstract. *This work describes a floating point arithmetic unit (FPU). The FPU is modeled in VHDL and synthesized for an FPGA development board. The floating point numbers are represented according to the 32 bits floating point IEEE-754 standard. The unit works as a peripheral that is acessed through the processor's data bus. The addition and multiplication operations implement the algorithms described in [Patterson and Hennessy 2009] and were split into pipeline stages to yield one operation result per clock cycle since floating point operations are expensive.*

Resumo. *Este trabalho descreve uma Unidade de Ponto Flutuante (UPF) modelada em VHDL e sintetizada para um módulo de desenvolvimento com FPGA. A representação de números reais no processador é feita segundo o padrão IEEE-754 de ponto flutuante de 32 bits. A unidade é acessada como um periférico no barramento de dados do processador, e as operações de soma e multiplicação estão implementadas de acordo com os algoritmos descritos em [Patterson and Hennessy 2009] e divididas em estágios de pipeline para possibilitar a conclusão de uma operação a cada ciclo de relógio, uma vez que operações de ponto flutuante são custosas.*

1. Introdução

Em 2012, no Departamento de Informática da Universidade Federal do Paraná, foi iniciado o desenvolvimento do cMIPS [Hexsel and Carmo 2013], um processador baseado no *design* clássico de cinco estágios do processador MIPS de 32 bits descrito em [Patterson and Hennessy 2009]. O modelo do processador caracteriza-se como uma ferramenta de ensino e pesquisa de arquitetura de computadores. As unidades básicas do processador de 5 estágios foram implementadas e sintetizadas para um módulo de desenvolvimento com um FPGA Altera. O cMIPS implementa somente instruções com números inteiros. Para um projeto que se propõe a compor um computador completo, é imprescindível suportar operações com números reais e é importante que a execução e a representação se adequem ao padrão utilizado na indústria. Este artigo apresenta a implementação das operações de soma e multiplicação de ponto flutuante de 32 bits.

O artigo apresenta brevemente o padrão IEEE-754 de representação de ponto flutuante de 32 bits. São descritos os algoritmos das operações de multiplicação e

adição, nas Seções 3 e 4. A Seção 5 descreve a interface da unidade com a CPU. Na Seção 6 é detalhada a implementação das operações com adaptações para o cMIPS, em especial a divisão de estágios de *pipeline*. As Seções 7 e 8 descrevem os procedimentos de testes e as considerações finais, respectivamente.

2. Representação de Ponto Flutuante

Representar números reais com um número limitado de bits é uma tarefa complexa. É impossível, por exemplo, representar dízimas periódicas, números irracionais, e sempre haverá um limite de precisão pois apenas um subconjunto pequeno dos reais pode ser representado por aproximações em 32 bits.

Na representação em ponto flutuante de 32 bits, um número é representado por três campos:

1 bit	8 bits	23 bits
sinal	expoente	mantissa

O valor real do expoente é o valor decimal do vetor de 8 bits subtraído de um deslocamento (*bias*) de 127, que permite que sejam representados valores negativos. O número representado pode ser computado pela Equação 1.

$$(-1)^s * (1 + m/2^{23}) * 2^{exp-127} \tag{1}$$

Dois expoentes são reservados:

- "1111111": para infinito positivo e negativo, e números não definidos (i.e. *Not a Number* ou NaN)
- "00000000": zero e números denormalizados.

O valor de um número denormalizado é menor que o menor número representável e é computado pela Equação 2.

$$(-1)^s * (0 + m/2^{23}) * 2^{exp-127} \tag{2}$$

No caso do ponto flutuante de 32 bits, podem ser representados $127 * 2^{25}$ números normalizados e 2^{24} números denormalizados. As minúcias da representação podem ser encontradas em [Kannan 2009].

3. Multiplicação de Ponto Flutuante

A implementação da multiplicação em ponto flutuante é baseada no algoritmo descrito em [Patterson and Hennessy 2009].

A mantissa tem um bit implícito, totalizando 24 bits, e seu valor depende do expoente. Se o expoente for diferente de 00000000 o bit implícito é 1, caso contrario (denormalizado ou zero), é 0.

O algoritmo para a multiplicação consiste dos seguintes passos:

- verificar o tipo das entradas (NaN, Inf,0 e denormal) e acionar *flags* caso o resultado seja um padrão ($\pm\infty * \pm 0 = NaN$, ...);
- calcular o expoente parcial somando os expoentes e subtraindo o deslocamento do resultado;

- calcular o sinal resultante com um ou-exclusivo entre os sinais;
- prever se a operação gera resultado denormalizado e calcular previamente o deslocamento necessário - quando o expoente parcial for menor do que EXPmin-1 (00000000), seu valor absoluto é o número de casas que o significando do resultado final deve ser deslocado para a direita;
- multiplicar as mantissas dos operandos;
- arredondar e normalizar o resultado.

Caso haja *underflow*, que ocorre quando o resultado parcial fica entre o menor número positivo ou o maior número negativo representável, o resultado final é forçado para zero, e é representado por:

sinal	expoente	mantissa
?	00000000	0.00000000000000000000000

Sendo o sinal o resultante da operação que causou o *underflow*.

Caso haja *overflow*, que ocorre quando o resultado parcial fica entre o menor número negativo ou o maior número positivo representável, o resultado é forçado a infinito, positivo ou negativo, dependendo do sinal resultante, e é representado por:

sinal	expoente	mantissa
?	11111111	1.00000000000000000000000

Sendo o sinal o resultante da operação que causou o *overflow*.

NaN (*Not a Number*) ocorre quando algum operando é NaN ou se ocorre a multiplicação de ± 0 por $\pm\infty$, e é representado por:

sinal	expoente	mantissa
0	11111111	1.00000000000000000000000

Casos especiais são tratados conforme o definido na Tabela 1.

Tabela 1. Casos Especiais da Multiplicação

$A \setminus B$	NaN	$+\infty$	$-\infty$	$+0$	-0	+Num	−Num
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
$+\infty$	NaN	$+\infty$	$-\infty$	NaN	NaN	$+\infty$	$-\infty$
$-\infty$	NaN	$-\infty$	$+\infty$	NaN	NaN	$-\infty$	$+\infty$
$+0$	NaN	NaN	NaN	$+0$	-0	$+0$	-0
-0	NaN	NaN	NaN	-0	$+0$	-0	$+0$
+Num	NaN	$+\infty$	$-\infty$	$+0$	-0	+Num ou $+\infty$	−Num ou $-\infty$
−Num	NaN	$+\infty$	$+\infty$	-0	$+0$	−Num ou $-\infty$	+Num ou $+\infty$

4. Adição

A adição em ponto flutuante é efetuada sobre dois operandos de 32 bits A e B, e é baseada no algoritmo descrito em [Patterson and Hennessy 2009].

Na adição devem ser utilizados os valores dos significandos, que são obtidos a partir das mantissas m_A e m_B com:

$$A = 1 + m_A/2^{23},\ \ldots\ B = 1 + m_B/2^{23}$$

exceto para números denormalizados, que não têm o "1" implícito. O número de bits do significando totaliza 24. A implementação exige um bit a mais para comportar a representação negativa em complemento de 2 de todos os valores possíveis do significando.

O algoritmo da adição consiste em:

- deslocar os bits de um dos significandos para a direita de acordo com a diferença entre os expoentes dos operandos, de modo a que ambos estejam no mesmo grau, que é o grau do maior expoente;
- efetuar a soma dos significandos e normalizar o resultado, levando em consideração a ocorrência de *overflow* e garantindo que o bit mais significativo da mantissa do resultado seja 1, exceto quando o resultado for 0, $\pm\infty$ ou potências de 2;
- corrigir o expoente do resultado de acordo com os deslocamentos realizados na normalização.

Na descrição do algoritmo do padrão IEEE-754, os operandos são "reposicionados" de modo que o A é sempre o de menor expoente. Isso resulta em que, caso haja diferença de expoentes, serão sempre os bits do significando A os bits deslocados para a direita.

Caso o sinal dos operandos seja diferente, é utilizado o complemento de dois do significando de A, independentemente de qual dos operandos é negativo, uma vez que o valor absoluto da diferença é sempre o mesmo. O sinal do resultado é o sinal do operando de maior expoente.

Caso os expoentes sejam iguais, os sinais sejam diferentes e o significando A seja maior do que o significando B, o resultado da adição dos significandos será negativo. Assim, será tomado o complemento de 2 deste resultado, para garantir que seja positivo e, neste caso, o sinal do resultado final deve ser o do operando A.

Os casos de *underflow* e *overflow* são processados como na multiplicação e os demais casos especiais são tratados como definido na Tabela 2.

Tabela 2. Casos Especiais da Adição

$A \setminus B$	NaN	$+\infty$	$-\infty$	$+0$	-0	+Num	−Num
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
$+\infty$	NaN	$+\infty$	NaN	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$-\infty$	NaN	NaN	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$+0$	NaN	$+\infty$	$-\infty$	$+0$	$+0$	+Num	−Num
-0	NaN	$+\infty$	$-\infty$	$+0$	-0	+Num	−Num
+Num	NaN	$+\infty$	$-\infty$	+Num	+Num	+Num ou $+\infty$	±Num ou $+0$
−Num	NaN	$+\infty$	$-\infty$	−Num	−Num	±Num ou $+0$	−Num ou $-\infty$

5. Interface com o processador

As duas unidades, de soma e de multiplicação, são utilizadas pelo processador como periféricos, sendo acessadas por meio de endereços de 32 bits a partir do endereço base da unidade de ponto flutuante (UPF). A CPU realiza dois *stores* nos endereços

dos operandos, os resultados são guardados em *buffers* de saída com quatro registradores de 32 bits e são enviados ao barramento mediante requisição da CPU – um *load* no endereço do primeiro operando, como mostram os trechos de programa abaixo.

```
Multiplication          Addition
sw r1, 0(UPF)           sw r1, 8(UPF)
sw r2, 4(UPF)           sw r2, 12(UPF)
lw r3, 0(UPF)           lw r3, 8(UPF)
```

O *buffer* de saída é usado para aumentar a concorrência entre a UPF e o processador. Cada registrador é acompanhado um bit de validade que indica se o resultado já foi enviado à CPU e portanto está obsoleto. Os dois bits menos significativos do endereço são utilizados para identificar uma instrução – eles acompanham os operandos pelo pipeline, indicam em que posição do *buffer* de saída de cada unidade um resultado deve ser guardado e qual resultado deve ser retornado para uma requisição da CPU. São usados estes dois bits para evitar acréscimo de bits de controle no barramento, enquanto oferece alguma flexibilidade ao programador para reordenar as requisições de resultados.

6. Pipelining

Com o relógio de 50 MHz do módulo de desenvolvimento seria impossível executar uma operação de ponto flutuante em um ciclo. As unidades de adição e de multiplicação foram portanto segmentadas em cinco estágios. Os estágios do circuito somador são:

- no primeiro, é recebido o primeiro operando, pelo endereço base + 8;
- no segundo estágio, é recebido o segundo operando, pelo endereço base + 12. É feita a subtração dos expoentes e a "inversão" dos operandos se necessário, liga-se os bits de status para entrada denormal, verifica-se se os sinais são diferentes e complementa-se o significando A se necessário;
- no terceiro, realiza-se a adição dos significandos, o complemento de 2 do resultado se necessário, quaisquer deslocamentos de bits necessários à normalização, e o arredondamento do resultado;
- no quarto estágio, realiza-se a adição de 1 ao resultado se o arredondamento assim determinar, desloca-se o resultado desta adição para a direita caso haja *overflow*, e corrige-se o expoente de acordo com os deslocamentos realizados;
- no quinto e último estágio, o resultado é guardado em um *buffer* de saída a esperar pela requisição da CPU.

Os estágios do multiplicador são:

- no primeiro, é recebido o primeiro operando, pelo endereço base;
- no segundo estágio, é recebido o segundo operando, pelo endereço base + 4. É feito o cálculo do expoente parcial, são ativadas as *flags* de resultados triviais (eg.: $\pm 0 \times x = \pm 0,\ x \in \mathcal{R}$), liga-se os bits de status para entrada denormal (usados no estágio seguinte), e é calculado o número de deslocamentos necessários caso se verifique que resultado será denormal;

- no terceiro, realiza a multiplicação das mantissas, utilizando um multiplicador de inteiros combinacional, e utiliza os bits de status de entrada denormal do segundo estágio para adicionar ou não o '1' implícito a mantissa;
- no quarto estágio são feitos o arredondamento e normalização da mantissa. O arredondamento utilizado foi o "Simple Round to Nearest/up Algorithm" [Santoro et al. 1989]. São então feitas verificações finais, tais como se o número é pequeno demais para representar (nem como denormal, sendo *underflow* ou zero), ou se o número é grande demais para representar;
- no quinto estágio, o resultado é guardado em um *buffer* de saída a esperar pela requisição da CPU.

7. Testes

Diversos casos de testes foram elaborados manualmente com base no fluxo de execução dos algoritmos. Efetuamos pelo menos um teste de cada fluxo de execução possível, e cada componente das unidades comportou-se como esperado. Complementarmente, foi escrito um programa de testes que gera dois números em ponto flutuante aleatórios e executa a multiplicação ou adição desses. As saídas foram comparadas com a ferramenta disponível em [Schmidt 2015], as mesmas entradas foram inseridas na unidade de ponto flutuante, e os resultados foram comparados. Desse modo temos alguma certeza da corretude dos resultados.

8. Considerações Finais

As unidades foram analisadas e simuladas com o compilador GHDL [Gingold 2012], e os resultados dos testes foram satisfatórios. Como trabalhos futuros serão executados testes da UPF em FPGA – independentes do processador – para adequar a distribuição de componentes nos estágios dos pipelines ao período de clock do processador. Em seguida, a UPF será acoplada ao cMIPS e embarcada em FPGA juntamente com os outros periféricos. Futuramente será adicionada a operação de divisão à UPF.

Referências

Gingold, T. (2012). Ghdl – g hardware design language. `http://ghdl.free.fr`. Acessado em 10-08-2015.

Hexsel, R. A. and Carmo, R. (2013). cMIPS – uma ferramenta pedagógica para o estudo de arquitetura. In *WEAC'13: Workshop sobre Educação em Arquitetura de Computadores*, pages 1–4.

Kannan, B. (2009). The design of an IC half precision floating point arithmetic logic unit. Msc dissertation, Clemson University, Dept of Electrical and Computer Engineering.

Patterson, D. A. and Hennessy, J. L. (2009). *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 4th edition. ISBN 9780123744937.

Santoro, M. R., Bewick, G., and Horowitz, M. R. (1989). Rounding algorithms for IEEE multipliers. In *Proc of 9th Symp on Computer Arithmetic*, pages 176–183.

Schmidt, H. (2015). Ieee 754 converter. `http://http://www.h-schmidt.net/FloatConverter/IEEE754.html`. Acessado em 10-08-2015.