IURI CASTRO FIGUEIRÓ

ARQUITETURA DE UMA UNIDADE ARITMÉTICA EM PONTO FLUTUANTE PADRÃO IEEE754 32 BITS

Trabalho de conclusão de curso apresentado como parte das atividades para obtenção do título de Engenheiro Eletricista, do curso de Engenharia Elétrica da Universidade Federal do Pampa – Campus Alegrete

Orientador: Alessandro G. Girardi

ALEGRETE
2011

ERRATA

Folha	Linha	Onde se lê	Leia-se

Autoria: Iuri Castro Figueiró

Título: Arquitetura de uma Unidade Aritmética em Ponto Flutuante Padrão IEEE754 32 bits

Trabalho de conclusão de curso apresentado como parte das atividades para obtenção do título de Engenheiro Eletricista, do curso de Engenharia Elétrica da Universidade Federal do Pampa – Campus Alegrete.

Os componentes da banca de avaliação, abaixo listados, consideram este trabalho aprovado.

	Nome	Titulação	Assinatura	Instituição
1	ALESSANDRO GIRARDI	DOUTOR		UNIPAMPA
2	SIDINEI GHISSONI	MESTRE		UNIPAMPA
3	EWERSON CARVALHO	DOUTOR		UNIPAMPA

Data da aprovação: 22 de JUNHO de 2011

Dedico este trabalho à minha família
pelo apoio durante os momentos mais difíceis
desta caminhada.

AGRADECIMENTOS

Agradeço em primeiro lugar a DEUS, por ter me dado força e coragem para enfrentar os mais diversos obstáculos encontrados e à minha família, pelas palavras de carinho e apoio nas horas mais difíceis. Aos integrantes do Grupo de Arquitetura de Computadores e Microeletrônica – GAMA, pela amizade e grande contribuição para o desenvolvimento deste trabalho.

Ao orientador Alessandro Girardi, pela confiança, direção e amizade. Ao professor Sidinei Ghissoni e demais amigos e colegas que tiveram papéis importantes para a realização deste trabalho.

Se alguém o criticar e você estiver realmente errado, alegre-se, agradeça e reconheça as críticas. Isto lhe proporcionará a oportunidade de corrigir as falhas.

Joseph Murphy

RESUMO

Aritmética em ponto flutuante é utilizada por diversas aplicações que exigem um alto processamento de dados aliados a uma excelente precisão, como processamento de voz e imagens. O hardware responsável pela aritmética em ponto flutuante é denominado de FPU (Floating Point Unit). A FPU pode ser utilizada tanto como co-processador em um sistema, como agregado a um processador mais complexo para uma aplicação específica. Normalmente, estas unidades são projetadas seguindo um padrão para a aritmética em ponto flutuante. O padrão IEEE754, o que estipula normas a serem seguidas para a representação de números em ponto flutuante, é adotado de forma a uniformizar e compatibilizar hardwares de diferentes fabricantes.

Este trabalho apresenta a proposta de uma arquitetura para operações aritméticas em ponto flutuante no Padrão IEEE754 no nível transferência entre registradores para implementação em FPGA. A FPU projetada utiliza a precisão simples e conta com as operações de soma, subtração, multiplicação e divisão. Através da metodologia adotada pelo Programa Brazil-IP, várias etapas de projetos são realizadas visando o desenvolvimento de um ip-core de custo e qualidade desejável.

A FPU visa aplicações onde o requisito de área utilizada é limitado, como sistemas embarcados e sistemas integrados em um único chip (SoC).

Os resultados apresentados são validados através de comparações com topologias *open source* existentes de FPU's.

Palavras-chave: Aritmética em Ponto Flutuante, Padrão IEEE754, Brazil-IP, FPU, SystemVerilog, FPGA.

ABSTRACT

Floating point arithmetic is used for numerous applications requiring a high data together with a great precision, such as voice and image processing. The hardware responsible for floating point arithmetic is called FPU (Floating Point Unit). The FPU can be used both as a co-processor in a system or embedded on a more complex processor for a specific application. Typically, these units are designed to follow a standard for floating point arithmetic. The IEEE754 standard, which provides guidelines for representing floating point numbers, is adopted in order to standardize and match hardware from different manufacturers.

This paper presents a proposed architecture for floating point arithmetic in the IEEE 754 Standard in register transfer level (RTL) for implementation in FPGA. The proposed FPU uses single precision and is able to perform operations such as addition, subtraction, multiplication and division. By means of the methodology adopted by the Brazil-IP program, several steps are realized aimed the development an IP-core with cost and quality desired.

The FPU is intended for applications where the requirement of area is limited, as embedded systems and integrated systems on a chip (SoC).

The results are validated by comparing topologies of existing open source FPUs.

Keywords: Floating point arithmetic, IEEE754 Standard, Brazil-IP, FPU, SystemVerilog, FPGA.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A maioria das aplicações científicas e de engenharia exige o uso de aritmética em ponto flutuante. Historicamente, o inconveniente relatado a esse tipo de formato é a alta complexidade quando se comparado com a aritmética inteira (NURMI, 2007). As operações em ponto flutuante foram implementadas primeiramente através de emulações em software, onde o ponto flutuante era simulado como um valor inteiro. Tais rotinas de implementação traduziam-se em alto tempo de execução e, por essa razão, passam a ser implementadas em hardware. Este hardware, comumente conhecido como Unidade em Ponto Flutuante – FPU, é capaz de desempenhar com maior velocidade as operações em ponto flutuante.

A alta complexidade destas unidades e por conseqüência a área utilizada por esse tipo de projeto, era um fator desencorajador aos projetistas (NURMI, 2007). Hoje, com o avanço da tecnologia VLSI (*Very Large Scale Integrated*), as FPU's estão aparecendo integradas a processadores com requisitos limitados de área e sendo utilizadas como ip-*cores* para compor sistemas maiores e mais complexos que necessitam da aritmética em ponto flutuante.

Antes do predomínio de um padrão para a aritmética em ponto flutuante, utilizava-se diferentes formatos para a representação de um número em ponto flutuante. Três formatos importantes foram utilizados pelo IBM 370, o DEC VAX e o Cray (PATTERSON et al., 2003). Estes formatos variavam o intervalo entre expoentes, bits alocados à fração, base numérica e bits de precisão. O processo de padronização começou em 1977, porém só foi aprovado em 1985 quando o IEEE (*Institute of Electrical and Eletronics Engineers*) publicou uma norma para a representação de números em ponto flutuante. Assim, o Padrão IEEE754 passou a ser adotado para o tratamento da aritmética binária em ponto flutuante relativo ao formato, precisão, exceções e operações aritméticas básicas (IEEE, 1985).

Cabe aqui destacar um ponto histórico da aritmética em ponto flutuante. Este ponto foi o *bug* clássico encontrado na divisão em processadores Pentium da Intel. Esse *bug* foi encontra-

do por um professor de matemática ao efetuar sucessivas divisões entre recíprocos de números primos (PATTERSON et al., 2003). Este erro custou 300 milhões de dólares à Intel, que teve que fazer o recall dos seus processadores comercializados.

1.1.Motivação

Assim como no desenvolvimento de software, a complexidade do hardware impõe a reutilização de componentes tanto para lidar com esta complexidade como também para diminuir o tempo de projeto dos sistemas digitais (LEITE, 2006). Dessa maneira, tem-se como motivação deste trabalho o desenvolvimento de um *ip-core* dedicado a operações em ponto flutuante, o qual poderá ser inserido em sistemas onde o requisito de área é fundamental, como sistemas embarcados ou sistemas integrados em um único circuito integrado - *System on a Chip* (SoC).

Também, por se tratar de um *ip-core,* ele torna-se reutilizável, já que é pré-projetado e pré-verificado para certa aplicação. Dependendo da forma que o *ip-core* é disponibilizado para o projetista, ele pode ter acesso ao *layout* do núcleo, ao *netlist*, como também ao código em HDL.

1.2. Objetivos

O objetivo deste trabalho é a obtenção de um módulo *ip-core* de uma nova alternativa de arquitetura de FPU para operações aritméticas em ponto flutuante em nível de FPGA (*Field Programable Gate Array*). O *ip-core* deve ser descrito em linguagem de descrição de hardware SystemVerilog utilizando a metodologia de projeto adotada pelo Programa Brazil-IP.

O Programa Brazil-IP tem por objetivo o desenvolvimento de recursos humanos na área de microeletrônica e desenvolvimento de *ip-cores* através de centros de projetos distribuídos em universidades brasileiras. A Universidade Federal do Pampa - UNIPAMPA, através do Grupo de Arquitetura de Computadores e Microeletrônica – GAMA está inserido neste programa há dois anos, onde o projeto da FPU se encontra em desenvolvimento pelos integrantes do grupo. O programa Brazil-IP fornece o suporte para as universidades, como treinamentos específicos para cada etapa do projeto do *ip-core,* licenças de softwares, utilizando a metodologia de projeto ipPROCESS (LIMA, 2005).

A metodologia ipPROCESS, prevê várias fases na concepção de um projeto em diferentes níveis de abstração, desde o nível de sistema até o nível de portas lógicas. Esta metodologia consiste de um processo de desenvolvimentos de *ip-cores*, de modo a transformar os re-

quisitos do ip-core em um produto de custo e qualidade desejável. Ela divide o projeto de um *ip-core* em 4 fases: concepção, arquitetura, projeto RTL e prototipação. Na fase de concepção, são elicitados os requisitos funcionais e não funcionais, a fim de definir o escopo do projeto, critérios de aceitação, escopo negativo e o que deve ser entregue com o produto. Na fase de arquitetura, é elaborada uma arquitetura estável, servindo de base para a implementação e verificação. No projeto RTL, é desenvolvido um protótipo do *ip-core* baseado na arquitetura previamente definida. Nesta fase, inclui-se a implementação e a verificação funcional. E por fim, na fase de prototipação é criado um protótipo físico.

A arquitetura proposta deve estar no padrão IEEE754, que prevê um tratamento uniforme para aritmética binária em ponto flutuante, de maneira a facilitar a portabilidade no sistema em que será inserido.

A FPU suportará as operações de soma, subtração, multiplicação e divisão de maneira que ocorra um compartilhamento de unidades funcionais e por consequência, uma menor utilização de área em FPGA.

1.3.Estrutura do trabalho

Este trabalho está estruturado da seguinte forma:

No Capítulo 2 serão vistos conceitos importantes da aritmética em ponto flutuante, como representações, padrões, fluxos das operações, etc. No Capitulo 3 será descrito o princípio de funcionamento da FPU através de uma visão funcional da arquitetura, assim como um maior detalhamento da parte operativa e de controle da arquitetura proposta. Por fim, o Capítulo 4 mostra os resultados obtidos de síntese lógica e métricas de desempenho, propondo comparações deste trabalho com outras arquiteturas *open-source* de FPU’s.

2 FUNDAMENTAÇÃO TEÓRICA

2.1. Representação de Números em Ponto Flutuante

Muitas aplicações exigem números que não são inteiros, ou seja, há aplicações que exigem um alto processamento de dados aliados a uma necessidade de alta precisão, como por exemplo, o processamento de voz e imagens. Alguns dos problemas mais complexos, como computadores utilizados para previsão do tempo, exigem uma aritmética de não inteiros de alto desempenho (MURDOCA et al., 2001).

Ao se tratar de uma representação de números inteiros, suponha-se, por exemplo, que um computador possa representar números tão grandes como um bilhão, mantendo pelo menos 20 bits à esquerda do ponto, uma vez que $2^{20} \approx 10^6$. Também, poderia representar um bilionésimo com 20 bits à esquerda do ponto, totalizando 40 bits por número.

Na prática, a manipulação desses números exigiria um hardware muito sofisticado, sendo que o processamento de todos esses dados levaria muito mais tempo do que se comparado com um pequeno número de bits (MURDOCA et al., 2001). Então, essa representação possui algumas limitações. Ela não possibilita representar números muito grandes e nem frações muito pequenas (STALLINGS, 2005).

Por exemplo, o número 3.155.760.000 em decimal pode ser representado como 3,15576 x 10^9. Isso só é possível com a utilização da notação científica. A vírgula é deslizada dinamicamente para uma posição conveniente e é usado o expoente adequado para representar o mesmo valor original. Isso possibilita expressar números muito grandes e muito pequenos com poucos dígitos (STALLINGS, 2005).

Assim, sendo possível a representação de um número decimal em notação científica, o número binário também pode ser representado dessa maneira, sendo que a vírgula passa a se chamar ponto binário e sua dinâmica de deslizamento de uma posição para outra é que dá o

nome de ponto flutuante. Assim, um número binário em ponto flutuante pode ser representado na seguinte forma:

$$\pm \mathrm{M} \times \mathrm{B}^{\pm \mathrm{E}} \quad (1)$$

onde (±) representa o sinal, *M* a mantissa, *E* o expoente e *B* a base.

Analisando a Equação 1 e inserindo nesta notação representações diferentes para um mesmo número binário, temos o seguinte exemplo:

$+0{,}111 \times 2^{0}$

$+111{,}0 \times 2^{-3}$

$+1{,}11 \times 2^{-1}$

Como a base *B* é a mesma para todos os números, ela torna-se implícita e não carece de ser armazenada. Também, nota-se que podemos ter diversas representações para um mesmo número, o que faz que comparações e operações aritméticas tornem-se complexas (MURDOCCA, 2001).

A fim de evitar representações múltiplas para o mesmo número, normalmente é requerido que esse número esteja normalizado, ou seja, o ponto é deslocado para a esquerda ou para a direita (e o expoente ajustado) até que o ponto esteja à direita do dígito diferente de zero mais à esquerda (MURDOCCA, 2001). Assim, um número normalizado em ponto flutuante tem a forma da Equação 2.

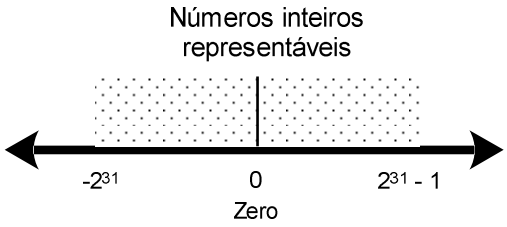
$$\pm 1, \mathrm{b_n b_{n-1}} \ldots \ \mathrm{b_0} \times 2^{\pm \mathrm{E}} \quad (2)$$

onde *b* é um dígito binário (0 ou 1).

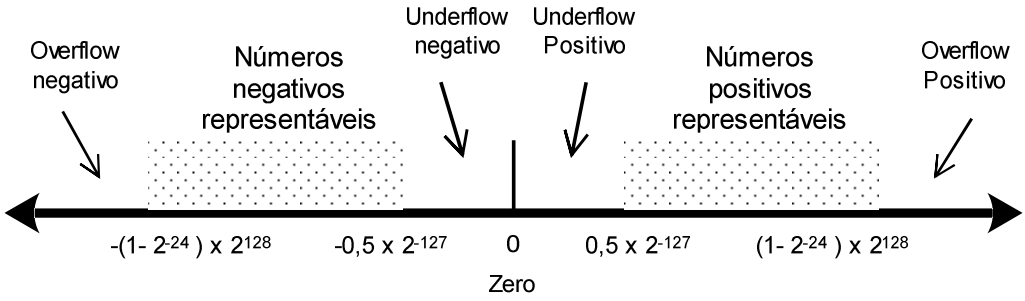
Se a mantissa for representada em binário, como é o caso, e se a condição de normalização exigir que o número mais à esquerda do ponto seja 1, então não é necessário armazenar este bit. A maioria dos formatos de ponto flutuante não o armazena (MURDOCCA, 2001). Em vez disso, ele é eliminado antes do armazenamento no formato de uma palavra em ponto flutuante e recolocado ao voltar no formato expoente e mantissa. Esse bit é chamado de bit escondido ou implícito.

2.1.1. Faixa de Representação de Números em Ponto Flutuante

Para melhor entender a vantagem da utilização de números em ponto flutuante, cabe-se destacar a FIGURA 1, contendo a faixa de números representáveis em uma palavra de 32 bits.



(a) Números inteiros em complemento de dois.



(b) Números de ponto flutuante.

FIGURA 1 - Números representáveis em formato 32 bits.

Na faixa de números inteiros representáveis, considerando a notação em complemento de dois, está o intervalo de -2^{31} a $(2^{31} - 1)$. A quantidade de números representáveis é na ordem de 2^{32} números distintos. Na faixa de números em ponto flutuante, notam-se regiões para valores representáveis positivos e negativos e regiões de *overflow* e *underflow*.

Um *overflow* ocorre quando o expoente do resultado é maior do que pode ser expresso pelo expoente, no caso maior que 128 (por exemplo, $2^{121} + 2^{100} = 2^{221}$). Um *underflow* ocorre quando o valor do expoente é muito pequeno, podendo o resultado geralmente ser arredondado para zero, o que o torna mais fácil de manipulação.

É importante ressaltar que a notação em ponto flutuante não permite uma maior representação de valores distintos. O número máximo de valores que podem ser representados para 32 bits, continua sendo 2^{32}. Porém, há uma maior densidade de números próximos a zero do que na outra notação e os valores não estão igualmente distribuídos na reta de números, como é o caso de números inteiros (STALLINGS, 2005).

2.2.Padrão IEEE754

Com as diferentes representações para os números em ponto flutuante, não havia uma característica uniforme a todos os padrões. O comportamento das operações em ponto flutuante variava desde o intervalo de representação até a precisão e a quantidade de números represen-

táveis. Assim, de maneira a padronizar a utilização desses números e facilitar a portabilidade entre softwares de processadores, o IEEE (*Institute of Electrical and Electronics Engineers*) publicou uma norma para representação de números em ponto flutuante. Assim, em 1985, o IEEE criou um padrão conhecido como IEEE 754 (IEEE, 1985).

Este padrão contém normas a serem seguidas pelos criadores de software e fabricantes de hardware, que prevê um tratamento da aritmética binária em ponto flutuante relativo ao formato, precisão, armazenamento, exceções, arredondamento, e as operações aritméticas básicas (VIANA, 1999).

Há também o padrão IEEE 854 (IEEE, 1987), que é um padrão tanto para binário quanto para decimal, porém ele não especifica como os números de ponto flutuantes são codificados em bits e não define a sua precisão (GOLDBERG, 1991).

2.2.1. Formatos do Padrão IEEE754

O padrão IEEE 754 provê dois formatos para a representação de números em ponto flutuante: simples e duplo.

O formato simples, ou precisão simples, é composto por 32 bits, sendo 1 bit para o sinal, 8 bits para o expoente e 23 bits destinados à fração, como mostrado na FIGURA 2.

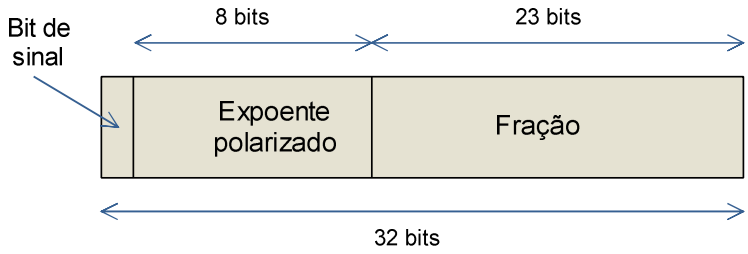


FIGURA 2 - Formato precisão simples no padrão IEEE754.

A base, como já discutida, por ser comum a todos os números binários, encontra-se implícita no formato.

Um número em ponto flutuante em precisão simples pode ser representado de forma analítica pela seguinte equação:

$$(-1)^s \, 2^e \times 1.f \qquad (3)$$

onde $f = (b_{23}^{-1} + b_{22}^{-2} + ... + b_0^{-23})$ é a fração, s é o bit de sinal (0 para positivo, 1 para negativo), *e* é o expoente polarizado (e= E + 127 (*bias*), E é o expoente não-polarizado). A mantissa M está representada pela Equação 4, sendo composto pela fração f e um bit implícito.

$$M = 1.f \quad (4)$$

O expoente polarizado se refere a uma maneira não comum de representar números não positivos através da soma de um número de desvio (PATTERSON, 2003). Neste caso, o desvio ou *bias* é 127 de forma que a variação do expoente esteja entre 1 e 254. A vantagem de uma representação polarizada é que números de ponto flutuante não negativos podem ser tratados como inteiros em uma comparação (GOLDBERG,1991).

O formato precisão dupla ocupa 64 bits, sendo que o sinal continua com o mesmo tamanho, 11 bits para o expoente com polarização de 1023 e 52 bits para a fração (ou 53 bits considerando o bit implícito). Na FIGURA 3 é mostrada a precisão dupla no formato ponto flutuante IEEE754.

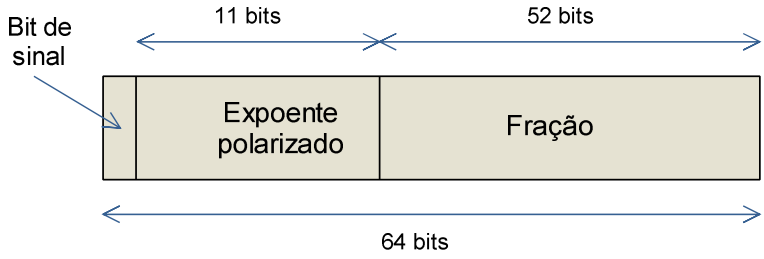


FIGURA 3 – Formato precisão dupla padrão IEEE754.

Além dos formatos de precisão simples e dupla existem os formatos estendidos simples e estendido duplo. Eles são usados para cálculos intermediários a fim de reduzir erros de arredondamento e fornecer uma maior precisão. Nestes formatos, bits adicionais são inseridos ao expoente e mantissa, de forma que o compromisso entre alcance e precisão seja atendido.

Para melhor entender os parâmetros desses quatro formatos no padrão IEEE754, estão mostrados o número total de bits (N) correspondente a cada tipo de precisão. Considere que *n*s represente o número de bits do sinal, *n*e o número de bits do expoente e *n*f o número de bits da fração. Sendo que N = (*n*s + *n*e + *n*f) bits.

a) **Precisão simples:** *n*s = 1, *n*e = 8, *n*f = 23, N = 32 bits

   bias: 127

b) **Precisão simples Estendida:** *n*s = 1, *n*e ≥ 11, *n*f ≥ 32, N ≥ 43 bits

   bias: 127

c) **Precisão Dupla:** *n*s = 1, *n*e = 11, *n*f = 52, N = 64 bits

   bias: 1023

d) **Precisão Dupla Estendida:** ns = 1, ne ≥ 15, nf ≥ 64, N ≥ 79 bits

bias: 1023

Grande parte dos processadores atuais possui co-processadores responsáveis pelos cálculos em ponto flutuante e utilizam pelo menos três dos formatos de precisão (TANENBAUM, 2007).

2.3. Interpretação de Números em Ponto Flutuante

No padrão IEEE754 os bits podem ser interpretados de diversas maneiras, desde valores usuais a valores especiais, dependendo dos vários padrões de bits. Vamos pegar como exemplo, a TABELA 1 para números em precisão simples, que será o foco e pré-requisito deste trabalho.

TABELA 1 - Interpretação de números em ponto flutuante.

Precisão Simples				
	Sinal	Expoente Polarizado	Fração	Valor
Zero Positivo	0	0	= 0	0
Zero Negativo	1	0	= 0	-0
Infinito Positivo	0	255 (todos 1s)	= 0	∞
Infinito Negativo	1	255 (todos 1s)	= 0	-∞
NaN Silenc.	0 ou 1	255 (todos 1s)	≠ 0	NaN
NaN Sinal.	0 ou 1	255 (todos 1s)	≠ 0	NaN
Normalizado Positivo	0	1 < e < 254	f	$2^{e-127}(1,f)$
Normalizado Negativo	1	1 < e < 254	f	$2^{e-127}(1,f)$
Não - Normalizado Positivo	0	0	≠ 0	$2^{e-126}(0,f)$
Não - Normalizado Negativo	1	0	≠ 0	$2^{e-126}(0,f)$

Como se pode observar, um expoente igual a zero junto com a fração igual a zero representa zero, logo é útil ter uma representação para o valor exato de zero (STALLINGS, 2005). O bit de sinal pode ser 0 ou 1, e então existem representações para zero positivo e para o zero negativo.

O padrão de bits para um número infinito difere da representação anterior somente no expoente, onde o mesmo é composto somente por uns (11111111). Também o infinito possui uma representação positiva e negativa.

Quando ambos os operandos são infinitos e a operação for uma divisão, por exemplo, temos um caso especial *Not a Number* (NaN).

Em um NaN, temos um expoente somente com uns e uma fração diferente de zero, podendo ser positivo ou negativo. O NaN pode ser interpretado no sistema aritmético de duas maneiras: NaN silencioso ou NaN sinalizador.

Um NaN silencioso ocorre de forma que sua propagação em uma operação aritmética não seja detectada como uma exceção. Normalmente ela advém de operações envolvendo casos especiais, como por exemplo, a soma entre dois números infinitos. Já o NaN sinalizador pode ser facilmente identificado já que gera uma operação inválida sempre que é usado como operando.

Temos também uma representação para números normalizados diferentes de zero, em que o intervalo de valores do expoente polarizado varia entre 1 e 254 e a fração recebe um bit implícito, o que dá uma mantissa. Esta representação é descrita pela Equação 3 vista na seção 2.2.1.

Por fim, observa-se um padrão de bits para números não-normalizados. Neste padrão o expoente possui um valor zero e corresponde a um valor real de -126. A fração possui um padrão real para a magnitude do número, sendo que o bit mais à esquerda da fração é zero, e não 1 (um), como utilizado para números normalizados. Números subnormais, como também são denominados, podem ser distinguidos dos normalizados porque não é permitido que ambos possuam um expoente igual a zero (TANENBAUM, 2007).

Os números subnormais foram adicionados ao padrão de forma a preencher o buraco existente entre zero e o primeiro menor número representável normalizado. Este vazio, que é notado na FIGURA 4, representa um intervalo bem maior do que se comparado com o intervalo entre o menor número não nulo representável e o número imediatamente maior.

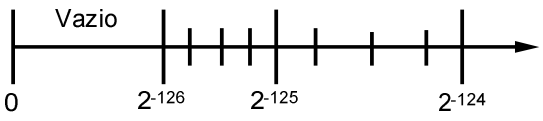


FIGURA 4 - Intervalo de uma representação normalizada 32 bits.

O uso de números subnormais é conhecido como underflow gradual (STALLINGS, 2005). O underflow gradual preenche esse vazio reduzindo o impacto de um underflow normal no expoente e introduz uniformemente 2^{23} valores entre 0 e 2^{-126}.

2.4.Aritmética de Números em Ponto Flutuante

O padrão IEEE754 prevê quatro operações aritméticas básicas: soma, subtração, multiplicação e divisão. Cada operação possui uma peculiaridade, tanto na manipulação de seus operandos quanto na complexidade do seu hardware.

2.4.1. Soma e Subtração

Dentre as operações em ponto flutuante suportadas, a soma e subtração levam destaque devido à necessidade de alinhamento dos expoentes (STALLINGS, 2005). De acordo com o diagrama de blocos apresentado na FIGURA 5, esse alinhamento somente será necessário se houver diferença entre os expoentes.

Havendo diferença, deve-se incrementar o menor expoente até igualar-se com o maior expoente, enquanto que a mantissa deverá ser deslocada para direita, podendo resultar em perdas de bits importantes ao decorrer do fluxo da operação. Em seguida, as mantissas são somadas ou subtraídas através de um somador de números inteiros. Caso as mantissas tenham o mesmo sinal, a operação poderá resultar em *overflow*, que seria um estouro da capacidade suportada de bits. Isto pode ser facilmente resolvido com o tratamento adequado da mantissa, ou seja, ela poderá ser deslocada para a direita, de forma que o bit menos significativo é perdido. Em consequência podemos ter overflow no expoente, sendo que sua ocorrência é reportada como um erro de operação.

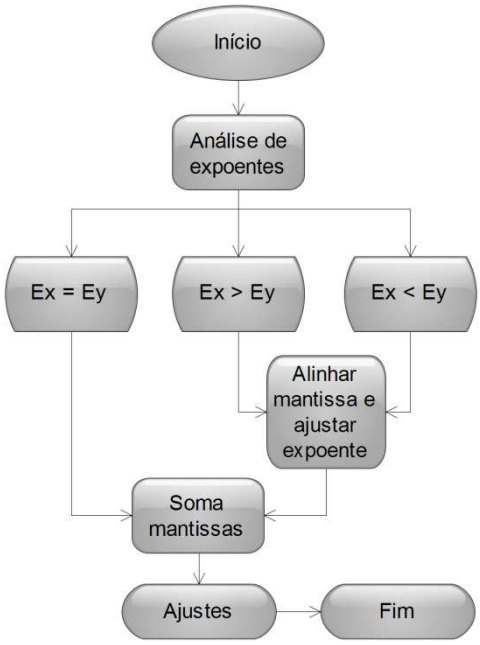


FIGURA 5 – Diagrama de blocos da soma/subtração em ponto flutuante.

Como requisito do padrão IEEE754, a resposta deve estar em um formato normalizado, ou seja, no fluxo da operação deve constar uma etapa de normalização. Deste modo, a mantissa deverá ser ajustada de maneira que obedeça ao padrão.

Para simplificar o entendimento sobre como é realizada a operação, suponha-se que tenhamos um sistema aritmético que trabalhe na base decimal, 4 dígitos na mantissa, e expoente *e* no intervalo *e*[5,-5].

Considerando X e Y como dois operandos, sendo $X = 0{,}9370 \times 10^4$ e $Y = 0{,}1272 \times 10^2$ e que os dois números serão somados resultando em Z = X +Y.

Note, que para a soma é necessário alinhar os dois expoentes, de maneira que a mantissa seja deslocada. Assim, $Y = 0{,}1272 \times 10^2 = 0{,}001272 \times 10^4$. A mantissa é alinhada com a adição de dois zeros após a vírgula, o que extrapola os 4 dígitos da mantissa requerida. De modo a não perder precisão, adota-se uma mantissa intermediária para os dois operandos.

Teremos então:

$Z = X + Y = 0{,}93700000 \times 10^4 + 0{,}001272 \times 10^4 = 0{,}93827200 \times 10^4$

Este é o resultado exato da operação, sendo que o mesmo passará por um processo de arredondamento e concatenação para ficar nos 4 dígitos exigidos.

Através deste exemplo, nota-se a importância de dígitos extras em operações intermediárias, ou bits de guarda, como também são chamados em um sistema binário, que propiciam uma melhor precisão, e consequentemente, em menores erros de arredondamento.

2.4.2. Multiplicação

O fluxo da operação de multiplicação, o qual está destacado na FIGURA 6, se dá de forma mais simples do que o da soma. Primeiramente são somados os expoentes e subtraídos o *bias*, já que na soma, o valor do *bias* tem o seu valor duplicado. A multiplicação das mantissas se dá através de números inteiros, sendo que o resultado terá o dobro do comprimento dos operandos, necessitando de uma precisão estendida para evitar erros de arredondamento.

Por fim, o resultado é normalizado e arredondado, tal como é feita na adição e na subtração.

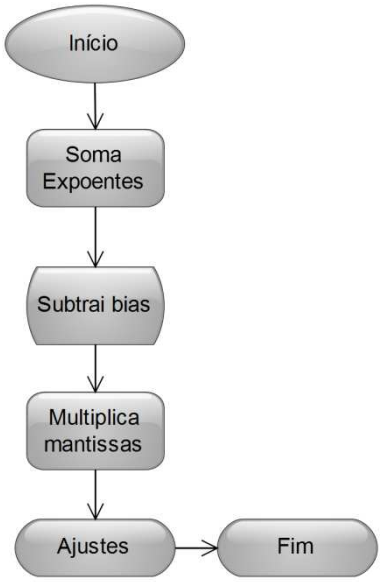


FIGURA 6 – Diagrama de blocos da multiplicação em ponto flutuante.

Considerando novamente $X = 0{,}9370 \times 10^4$ e $Y = 0{,}1272 \times 10^2$ para a exemplificação dessa operação, e que as mesmas condições de alcance e precisão são adotadas, teremos:

$$Z = X * Y = (0{,}9370 * 0{,}1272) \times (10^4 + 10^2) = 0{,}1191864 \times 10^6$$

Aqui as mantissas são multiplicadas e os expoentes somados. Note que o limite do expoente foi ultrapassado, resultando em um overflow. Isto poderá ser resolvido na etapa de normalização, deslocando-se a vírgula para a esquerda e decrementando o expoente.

2.4.3. Divisão

Ao contrário da multiplicação, na operação de divisão em ponto flutuante, cujo diagrama de blocos está mostrado na FIGURA 7, as mantissas são divididas e os expoentes subtraídos. Ao subtrair os expoentes, a polarização também é removida, bastando então adicioná-la ao resultado.

Como se pode observar, em todas as operações descritas até o momento a mantissa era tratada como uma operação de números inteiros e o resultado esperado entre inteiros só poderia ser um número inteiro.

O mesmo não acontece para a divisão. A divisão entre dois inteiros poderá resultar em um número real. Então, para a divisão das mantissas será necessário um divisor de números reais.

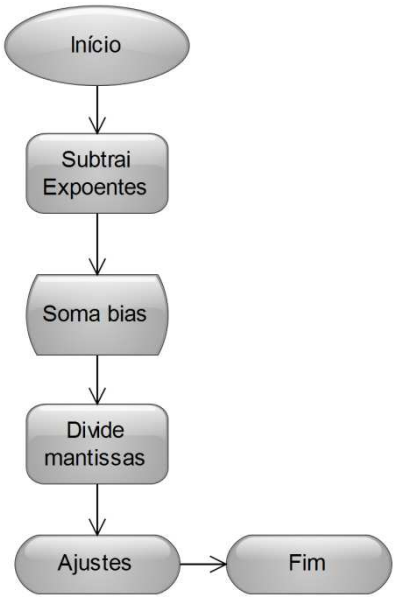


FIGURA 7 – Diagrama de blocos da divisão em ponto flutuante

2.5.Precisão e bits de guarda

A precisão de um resultado depende das operações intermediárias a que o número foi submetido. Muitas vezes, perdem-se bits importantes em processos de normalização como, por exemplo, no deslocamento da mantissa para o alinhamento dos expoentes. Também, há perdas em operações aritméticas que exigem seguidos deslocamentos em seus algoritmos.

Uma das formas de se reduzir erros desnecessários é com a utilização de bits de guarda, ou seja, dígitos extras ao número. Este dígito faz com que a precisão seja maior e consequentemente, um resultado de melhor exatidão.

Para entender a razão do uso desses bits, suponha que tenhamos dois números em um sistema decimal, com uma mantissa de 4 dígitos. Considerando X e Y como dois operandos, sendo $X = 0{,}9370 \times 10^4$ e $Y = 0{,}1272 \times 10^2$ e que os dois números serão somados resultando em Z = X +Y. Lembrando-se do exemplo da seção 2.4.1, para a soma de números em ponto flutuante é necessário o alinhamento dos expoentes. Assim:

$Y = 0{,}1272 \times 10^2 = 0{,}001272 \times 10^4$

Como a mantissa é de 4 dígitos, nesse processo Y perde dois dígitos, resultando em:

$Y = 0{,}0012 \times 10^4$

Então, a soma é:

$Z = 0{,}9382 \times 10^4$

A mesma operação é repetida, agora adicionando dois dígitos de guarda aos dois operandos.

$X = 0{,}937000 \times 10^4$ e $Y = 0{,}127200 \times 10^2$

Alinhando os expoentes:

$Y = 0{,}001272 \times 10^4$

Assim, quando o expoente de Y é alinhado, os dois dígitos importantes para a operação não são perdidos.

$$Z = X + Y = 0{,}937000 \times 10^4 + 0{,}001272 \times 10^4 = 0{,}938272 \times 10^4$$

Dado que no sistema adotado o número de dígitos na mantissa é igual a 4, os dois dígitos de guarda servirão para o arredondamento do resultado. Se for usado um arredondamento, teremos $Z = 0{,}9383 \times 10^4$ e se for usado truncamento, teremos, $Z = 0{,}9382 \times 10^4$.

Assim, com a adição de bits de guarda teremos uma diminuição de erros em etapas intermediárias e menor propagação de erros a etapas posteriores, como por exemplo, na etapa de arredondamento descrita logo a seguir.

2.5.1. Arredondamentos

Devido o fato de haver uma representação finita para números em ponto flutuante, como por exemplo, uma precisão simples, um resultado que apresente uma precisão maior necessita de uma política de arredondamento adequada (STALLINGS, 2003).

Este arredondamento, também é visto como a necessidade de resolver problemas da própria representação em ponto flutuante, ou seja, muitos dos valores não podem ser representados exatamente (CARTER, 2002), do mesmo modo que muitos dos números reais não podem ser representados por um número decimal. Assim, quando um cálculo cria um valor que não pode ser representado no formato em ponto flutuante, tem-se a necessidade de arredondar o resultado para um valor mais próximo em sua faixa.

No padrão IEEE754, estão previstos 4 modos de arredondamento:

- **Arredondamento para o mais próximo:** é o modo padrão de arredondamento, sendo que o número é arredondado para o número representável mais próximo com precisão infinita. Neste caso, é verificado se os bits excedentes ultrapassam mais que a metade do ultimo bit representável. Se for o caso, é somado 1 ao último bit representável ou subtraído 1, dependendo do sinal do número a ser arre-

dondado. Caso contrário, o resultado é truncado. Há casos, em que o valor dos bits excedentes está exatamente entre dois números representáveis, sendo que o adotado pelo padrão, neste caso, é o arredondamento para o próximo par.

- **Arredondamento em direção a mais infinito (+∞):** o resultado deve ser o valor, dentro do formato, mais próximo e não menor que o resultado infinitamente preciso (SILVA, 2007).
- **Arredondamento em direção a menos infinito (-∞):** o resultado deve ser o valor, dentro do formato, mais próximo e não maior que o resultado infinitamente preciso (SILVA, 2007).
- **Arredondamento em direção a zero:** Neste caso, o valor é truncado e simplesmente os bits excedentes são descartados.

De forma a exemplificar os formatos de arredondamento, está mostrado na FIGURA 8 o diagrama de blocos para os quatro modos de arredondamento em um sistema decimal. Considere neste caso, o dígito excedente de uma precisão como dígito de guarda.

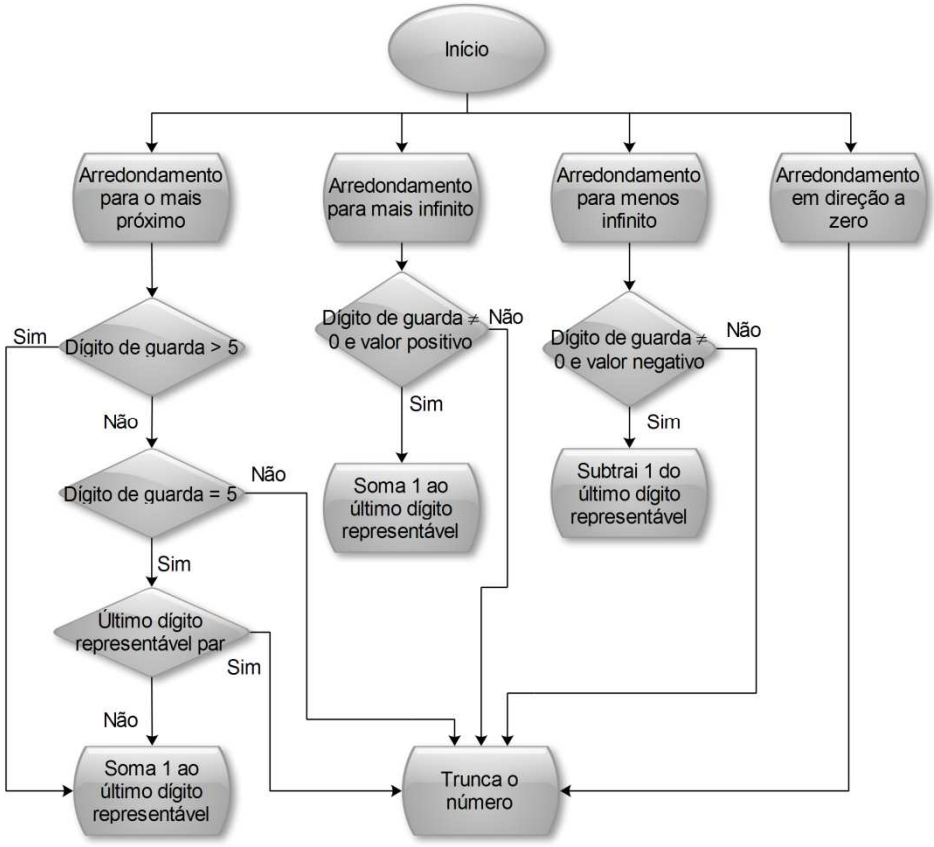
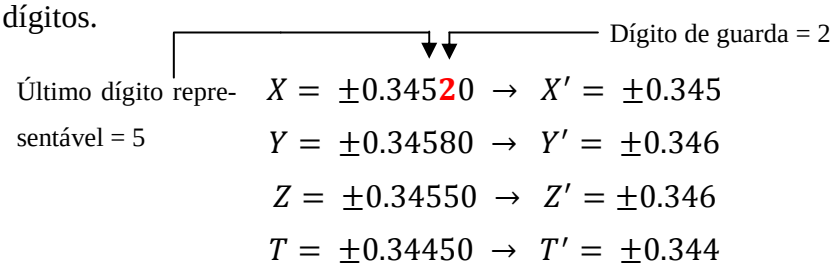


FIGURA 8 – Diagrama de blocos para os modos de arredondamento.

No arredondamento para o mais próximo, primeiro é verificado se o dígito de guarda é maior do que 5. Caso for, o valor é arredondado somando “1” ao último dígito representável. Caso contrário, é verificado se o dígito de guarda é igual a 5. Dependendo do valor, será necessário analisar o último dígito representável para efetuar o arredondamento ou o truncamento. Abaixo estão mostrados alguns exemplos para este modo de arredondamento, onde nota-se o dígito de guarda grifado em vermelho para um valor de 5 dígitos após a vírgula e uma precisão de 3 dígitos.



$$X = \pm 0.34520 \rightarrow X' = \pm 0.345$$
$$Y = \pm 0.34580 \rightarrow Y' = \pm 0.346$$
$$Z = \pm 0.34550 \rightarrow Z' = \pm 0.346$$
$$T = \pm 0.34450 \rightarrow T' = \pm 0.344$$

Para o arredondamento em direção ao mais infinito é verificado se os dígitos de guarda são diferentes de zero e se o valor é positivo. Caso esta condição seja aceita, é somado “1” ao último dígito representável. Caso contrário, o número é truncado. Abaixo é mostrado alguns exemplos para este modo de arredondamento.

$$X = +0.34521 \rightarrow X' = +0.346$$
$$Y = -0.34521 \rightarrow Y' = -0.345$$

No arredondamento em direção ao menos infinito, também é verificado se os dígitos de guarda são diferentes de zero e se o valor é negativo. Caso esta condição seja aceita, é subtraído “1” do último dígito representável. Caso contrário, o número é truncado. Abaixo é mostrado alguns exemplos para este caso.

$$X = +0.34521 \rightarrow X' = +0.345$$
$$Y = -0.34521 \rightarrow Y' = -0.346$$

Por fim, no arredondamento em direção a zero o número é truncado e o dígito de guarda é descartado, como mostrado no exemplo a seguir.

$$X = +0.34521 \rightarrow X' = +0.345$$
$$Y = -0.34521 \rightarrow Y' = -0.345$$

3 CONCEPÇÃO DA ARQUITETURA

Visando a implementação de uma FPU padrão IEEE754 com uso reduzido de área em FPGA, nesta seção serão vistos os elementos de hardware necessários para o suporte das operações aritméticas de soma, subtração, multiplicação e divisão. A reusabilidade de elementos funcionais na arquitetura permitirá que ambas as operações utilizem o mesmo caminho de dados, resultando em uma maior economia de hardware.

Utilizando como base a metodologia de projeto IpPROCESS (LIMA, 2005), a proposta de arquitetura será descrita em diferentes níveis de abstração, desde uma visão sistêmica de sua funcionalidade até o detalhamento de cada bloco e a descrição de seu comportamento no sistema. Em se tratando de descrição de um projeto digital, utiliza-se linguagens de descrição de hardware, comumente conhecidas como HDL's, não só como métodos para a descrição do comportamento do sistema, mas também como uma descrição estrutural dos componentes que o compõem.

A HDL SystemVerilog (SUTHERLAND, 2004), utilizada neste projeto, é uma nova linguagem que evoluiu da HDL Verilog e é utilizada em projetos orientados a hardware (LEITE, 2006). Assim, todos os elementos funcionais da FPU foram descritas com esta linguagem, sendo que alguns códigos serão mostrados para a exemplificação do comportamento do bloco.

3.1. Visão e Casos de Uso

Antes de apresentar a arquitetura proposta, é necessário ter uma visão sistêmica da funcionalidade do projeto. Assim, na FIGURA 9 apresenta o diagrama de casos de uso da FPU, que tem por finalidade ter uma visão de alto nível do sistema projetado e sua relação com sistemas externos.

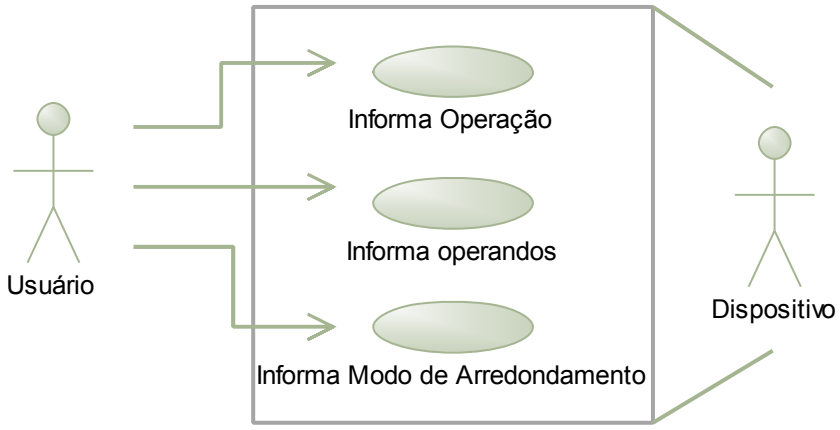


FIGURA 9 - Casos de uso da FPU.

Neste diagrama temos o usuário, dispositivo e os casos de usos que os relacionam. O usuário pode ser qualquer elemento externo ao projeto, como por exemplo, um processador. O dispositivo é o sistema projetado, ou seja, a própria FPU. Os casos de uso, que estão descritas abaixo das elipses, são serviços que o dispositivo fornecerá ao usuário para que o processo seja inicializado e finalizado de maneira ideal.

Assim, o usuário sempre inicia o uso ou recebe um valor de caso de uso. Neste caso, ele inicia com as entradas fornecidas ao dispositivo, como a operação escolhida, modo de arredondamento e os operandos a serem manipulados. O dispositivo utiliza estes casos de uso e fornece saídas ao usuário, como o resultado da operação ou algumas *flags* indicadoras de exceção.

3.2. Visão do Fluxo de dados

Considerando os casos de uso, e tendo uma visão menos abstrata do que mostrada anteriormente, na FIGURA 10 está mostrada o diagrama de blocos simplificado para o fluxo das operações de soma, subtração, multiplicação e divisão em ponto flutuante. Este diagrama de blocos foi obtido com base nos algoritmos das operações, nos casos de exceções previstos pelo padrão IEEE754 e ajustes como a normalização e arredondamento.

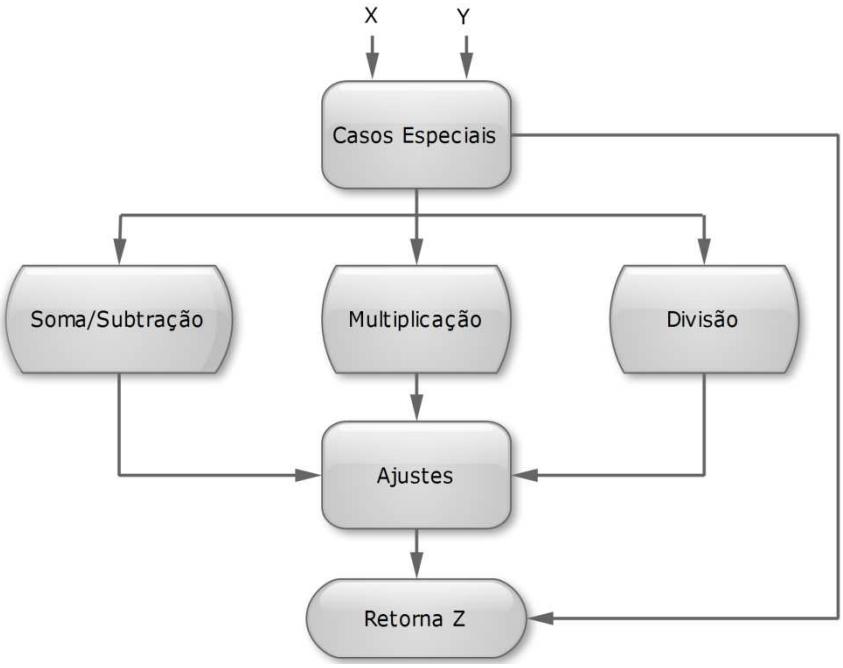


FIGURA 10 - Fluxo das operações em ponto flutuante.

Suponha-se então, que o usuário já tenha fornecido os dados de entrada ao dispositivo, como os operandos, operação e modo de arredondamento. O dispositivo, que agora está ilustrado pelo diagrama de blocos, analisa os operandos X e Y na etapa Casos Especiais. Nesta etapa é verificado se os operandos podem ser tratados como exceção, evitando processamentos desnecessários. Após está etapa, os operandos são manipulados de acordo com a operação definida pelo usuário, sendo que a resposta é enviada para a etapa de Ajuste. Nesta etapa, o resultado passa por processos de normalização e arredondamento, antes de ser fornecido ao usuário como resposta final Z.

Este fluxo de dados simplificado possui etapas que são comuns às quatro operações, como a etapa de casos especiais e ajustes. Ele é básico, porém representa fielmente o que a arquitetura proposta irá desempenhar.

3.3.Especificações da Arquitetura

Observa-se que até agora, a FPU não foi especificada quanto às entradas e saídas, nem o número de bits de cada sinal. Dessa maneira, tem-se a FIGURA 11 a qual representa uma caixa preta da FPU contendo somente os sinais de entrada e saída.

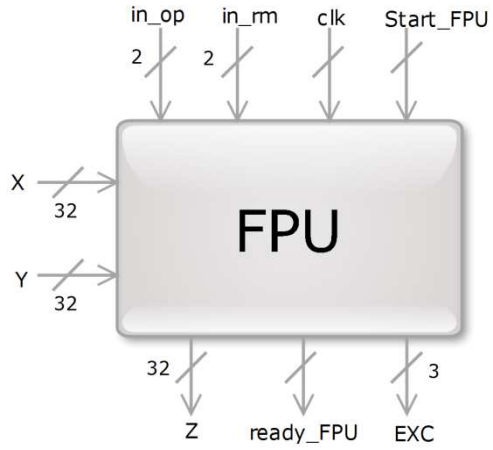


FIGURA 11 – Especificações de E/S da FPU.

Como requisito inicial deste trabalho, a FPU projetada estará no padrão IEEE754 precisão simples, portanto, os operandos X e Y terão 32 bits cada. O sinal de operação terá uma palavra de 2 bits denominado *in_op*. Na TABELA 2 está mostrado a sequência em binário para cada tipo de operação.

TABELA 2 – Sinais para as operações.

in_op	Significado
00	Adição
01	Subtração
10	Multiplicação
11	Divisão

O sinal *in_rm* terá 2 bits representando os 4 modos de arredondamento disponíveis na FPU. Na TABELA 3 está mostrado a sequência em binário para cada tipo de arredondamento.

TABELA 3 – Sinais para os modos de arredondamento.

in_rm	Significado
00	Em direção a zero
01	Em direção a +∞
10	Em direção a -∞
11	Em direção ao mais próximo

A FPU terá um sinal de inicialização e finalização, ambos de 1 bit e denominados *start_FPU* e *ready_FPU*, respectivamente. O sinal Z será de 32 bits, representando a resposta

da operação. Há ainda, uma *flag* sinalizadora de exceções de 3 bits denominada EXC e um sinal de sincronismo clk. Na TABELA 4 está mostrado a sequência em binário para cada tipo de exceção.

TABELA 4 – Sinais de exceções.

EXC	Significado
000	Resultado inexato
001	Overflow
011	Underflow
100	Divisão por zero
101	Infinito
110	QnaN
111	SnaN

3.4.Arquitetura Proposta

Após definido o nível comportamental da FPU, onde preocupou-se somente com suas funcionalidades e não com os componentes que a compõem, parte-se então para um nível mais detalhado da arquitetura. Na FIGURA 12, encontra-se a arquitetura proposta que será responsável pelas operações de soma, subtração, multiplicação e divisão no padrão IEEE754 precisão simples (NEVES et al., 2010). Nesta figura, observam-se registradores inseridos para o armazenamento de dados de entrada de maneira que os operandos sejam armazenados no formato sinal, expoente e mantissa. Há também registradores e blocos intermediários, que além de serem utilizados para a multiplicação e divisão, são utilizados para operações intermediárias. Há um bloco somador, que será a parte principal da arquitetura, sendo que qualquer operação utiliza o somador em seu fluxo, o que o torna importante para o desempenho global do sistema. Por fim, há controles individuais para a multiplicação e divisão devido a cada operação possuir um algoritmo diferenciado, e um controle central, responsável pela codificação e funcionamento de toda arquitetura.

De maneira a compreender o funcionamento da arquitetura, optou-se por mostrá-la em partes, cabendo à parte de descrição da organização da FPU detalhar cada elemento, funcionamento e papel dentro da arquitetura, e a parte de descrição do fluxo de dados, onde estará sendo realizada uma análise detalhada do caminho de dados de cada operação.

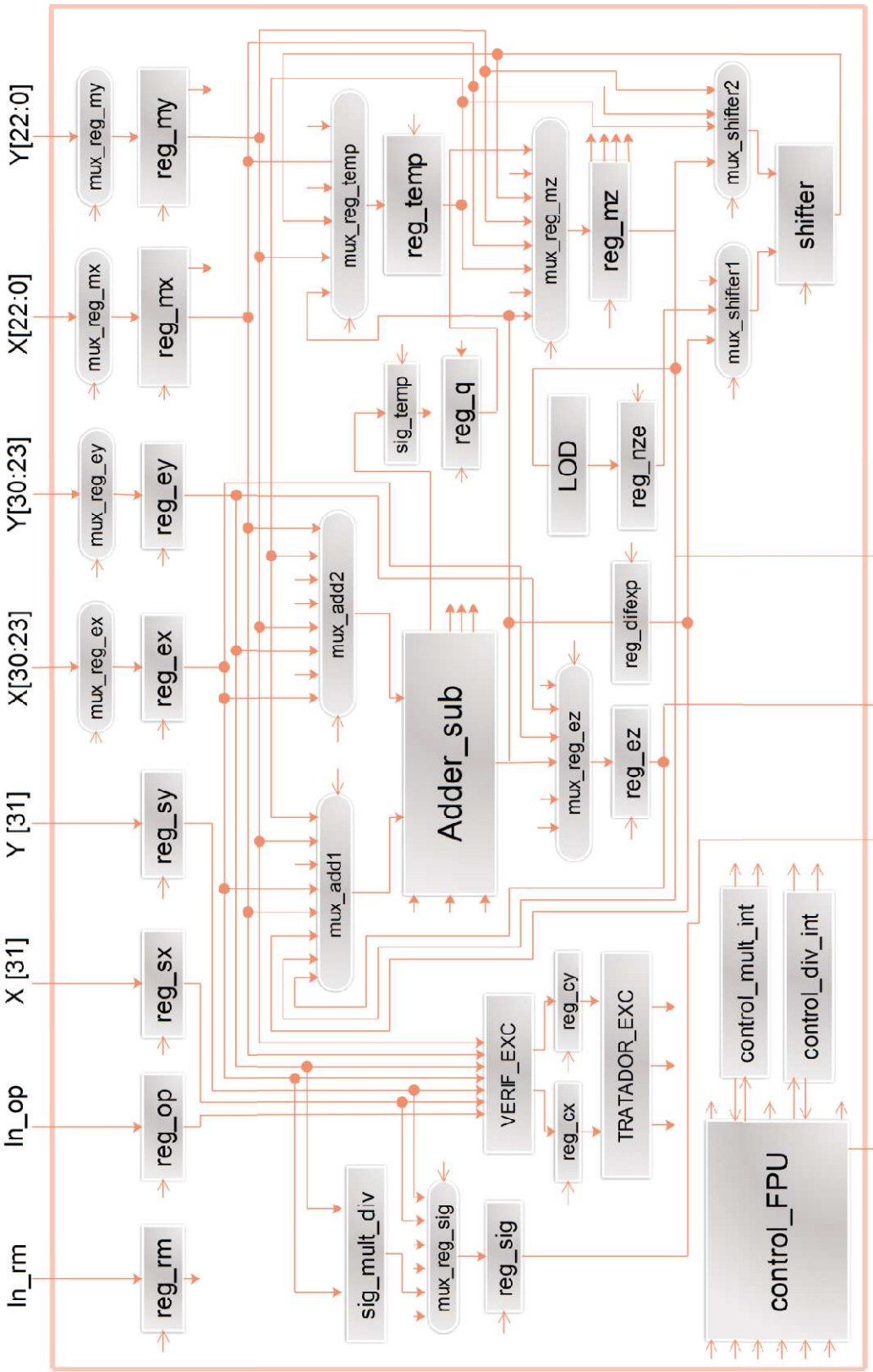


FIGURA 12 – Arquitetura FPU IEEE754 32 bits

3.4.1. Descrição organizacional da FPU

Nesta seção será visto detalhadamente cada elemento ou conjunto de elementos, de forma a especificar não só os sinais de entrada e saída, como também o seu comportamento em linguagem de descrição de hardware – SystemVerilog. Os elementos estarão divididos em Registradores de Entrada, Bloco de Casos Especiais, Blocos e Registradores intermediários, e Bloco Somador.

3.4.1.1.Registradores de Entrada

Um registrador é um conjunto de células de memória sensíveis à transição do sinal de clock, agrupadas de forma a armazenar uma dada palavra binária (UYEMURA, 2002). Neste caso, os registradores são utilizados para armazenar os dados de entrada da FPU de forma paralela, como o modo de arredondamento, operação e os operandos. Como cada dado possui uma largura de palavra diferente, foi implementado em SytemVerilog um registrador do tipo *reg_n_bits* parametrizável para referenciá-lo para qualquer tamanho de palavra. O código em SystemVerilog está mostrado em ALGORITMO 1.

```
1 module reg_n_bits
2  #(parameter integer Data_Width =2)
3 (clk, rst,write, reg_input, reg_output);
4  input logic clk;
5   input logic rst;
6   input logic write;
7   input logic  [Data_Width -1:0] reg_input;
8   output logic [Data_Width -1:0] reg_output;
9  always_ff @ (posedge clk)
10       if (rst) reg_output <=0;
11       else if (write)
12            reg_output <=reg_input;
13 endmodule
```

ALGORITMO 1 - Código parametrizável para um registrador de n bits.

Para referenciá-lo basta o seguinte comando:

reg_n_bits #(.Data_Width(n_bits)) Nome do Bloco (Entrada1, Entrada N,...., Saída1, Saída N).

A seguir serão descritos os registradores utilizados para armazenar dos dados de entrada da FPU.

- **Reg_rm**

Registrador do tipo *reg_n_bits* utilizado para armazenar o modo de arredondamento. Este tipo de registrador é parametrizável, ou seja, o número de bits é escolhido pelo projetista no momento de referenciá-lo no projeto global. Como são utilizados 4 modos de arredondamento, o tamanho deste registrador é de 2 bits. Neste bloco, mostrado na FIGURA 13, temos o sinal de clock *clk* utilizado como referência para a gravação dos dados, os sinais de controle de reset e write, ambos denominados *rst_reg_rm* e *w_reg_rm*.

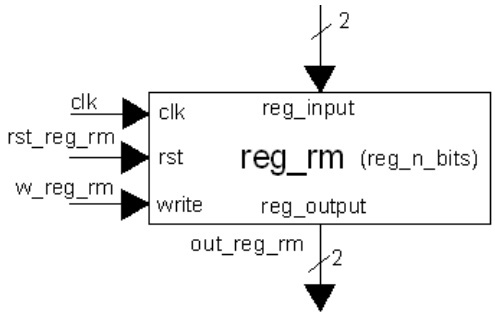


FIGURA 13 - Registrador para o modo de arredondamento.

- **Reg_op**

Registrador do tipo *reg_n_bits* utilizado para armazenar a operação escolhida. Como podem haver quatro operações, *reg_op* é de 2 bits. Neste bloco, que está mostrado na FIGURA 14 temos os sinais *clk*, *rst_reg_op* e *w_reg_op*, representando os sinais de clock, escrita e reset, respectivamente.

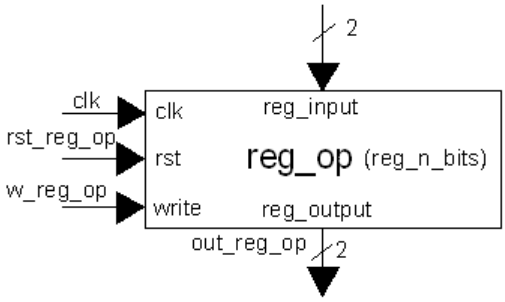


FIGURA 14 - Registrador para a operação.

- **Reg_sx e reg_sy**

Registradores do tipo *reg_n_bits* utilizados para armazenarem o bit mais significativo do operando X e Y, que representam os bits de sinais dos mesmos. Os sinais de controles destes registradores são *clk*, *w_reg_sx*, *w_reg_sy*, *rst_reg_sx e rst_reg_sy* e podem ser visualizadas na FIGURA 15.

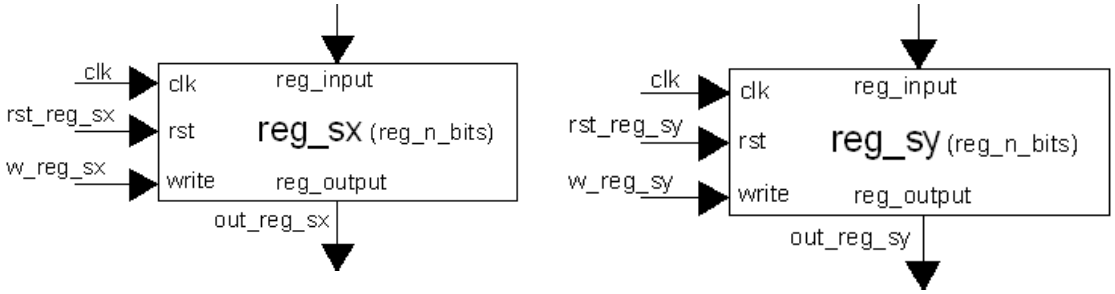


FIGURA 15 - Registradores para sinais de X e Y.

- **Reg_ex e Reg_ey**

Registradores do tipo *reg_n_bits* utilizados para armazenar os expoentes de X e Y. Estes registradores possuem um tamanho de 8 bits, sendo que os bits são concatenados das posições 23 a 30 de cada operando. Além do sinal de clock, possuem sinais de escrita e reset. Estes registradores, os quais podem ser notados na FIGURA 16, possuem em sua entrada multiplexadores de maneira a selecionar o expoente correto em casos de números subnormais. Quando se trata de um número subnormal, o expoente tem seu valor real de -126. Ao ser inserido na FPU, o expoente é tratado como zero.

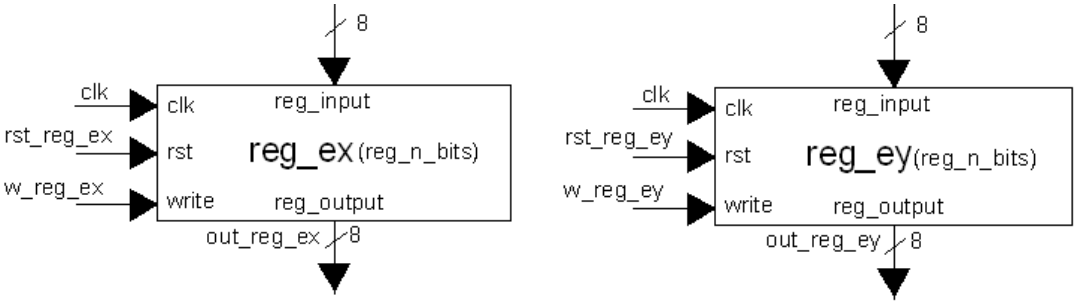


FIGURA 16 - Registradores para os expoentes.

- **Reg_mx e Reg_my**

Registradores do tipo *reg_n_bits* utilizados para armazenar as mantissas de X e Y. Estes registradores, os quais estão mostrados na FIGURA 17, diferem do tipo *reg_n_bit* convencional somente pelas linhas de comando mostradas no ALGORITMO 2.

Neste algoritmo, é identificado se o operando armazenado não é zero. Caso for, é reportado um sinal interno *zero* com nível lógico alto que correspondem aos sinais externos *zero_mx* e *zero_my*. Na FIGURA 17 notam-se também sinais de controle, como para escrita, reset e clock.

```
1 always_comb
2       begin
3           if (reg_output == 0)
4           zero<= 1'b0;
5         else
6           zero<=1'b1;
7     end
```

ALGORITMO 2 - Complemento ao código de um registrador de n bits para identificação de zero.

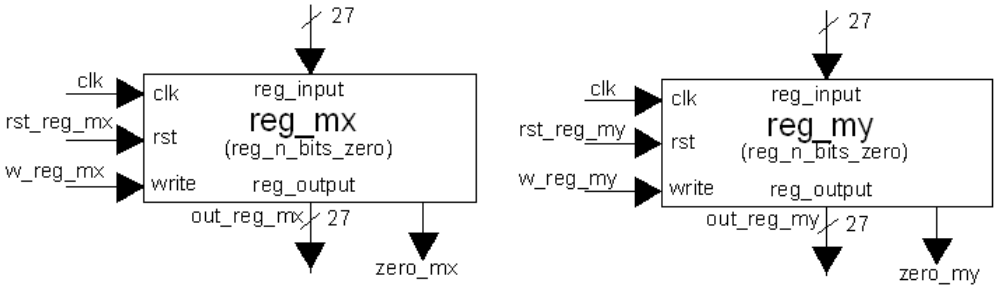
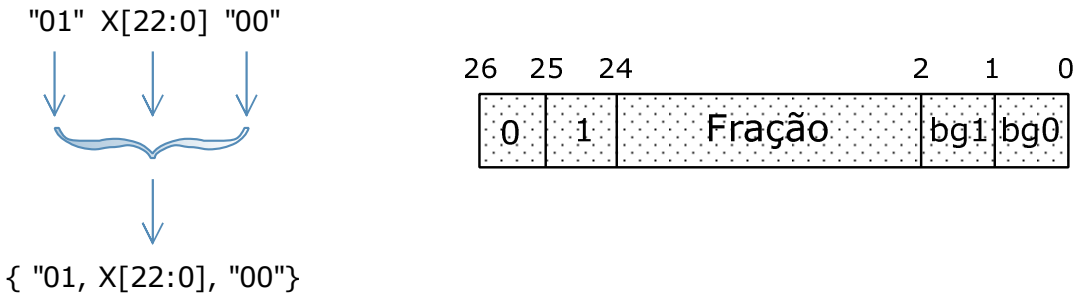


FIGURA 17 - Registradores para as mantissas de X e Y.

Devido ao padrão IEEE754, o bit implícito na mantissa não é armazenado no formato padrão precisão simples, porém, ao ser inserido na FPU, o bit implícito deve ser recolocado novamente para os cálculos posteriores. Logo, a fração de 23 bits passa para 24 bits da mantissa. O padrão interno para armazenar a mantissa normalizada está apresentado através da FIGURA 18.



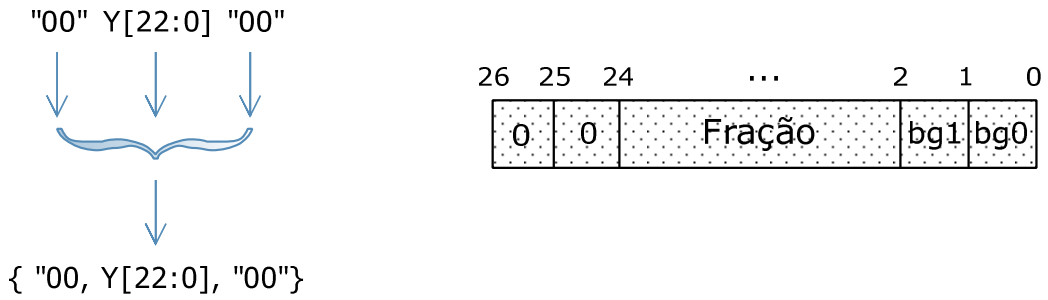
(a) Padrão de concatenação de bits **(b) Mantissa de 27 bits**

FIGURA 18 - Padrão de concatenação de bits para uma mantissa normalizada.

Observa-se na FIGURA 18a, que devido ao bit implícito não estar armazenado pelo padrão IEEE754, é necessário reinserí-lo novamente na representação. Logo a notação {"01, X[22:0], "00"}, refere-se a uma concatenação de bits realizada internamente na FPU para que

os cálculos sejam realizados. O bit “1” representa o bit implícito do padrão, X[22:0] a fração do operando X, e "00" os bits de guarda. Na FIGURA 18b, observa-se a mantissa do operando X com uma palavra final de 27 bits, sendo que o bit “0” da posição 26 foi inserido para casos intermediários de overflow na mantissa.

Esta representação é utilizada para números normalizados, sendo que para números subnormais a representação da mantissa pode ser observada na FIGURA 19. Onde “01” é agora substituído por “00” para representar um número subnormal.



(a) Padrão de concatenação de bits **(b) Mantissa de 27 bits**

FIGURA 19 - Padrão de concatenação de bits para uma mantissa subnormalizada.

Os mesmos procedimentos de representação são considerados para o operando Y, e a seleção do formato adequado é realizada por meio de multiplexadores.

3.4.1.2.Bloco de Casos Especiais

Este bloco é responsável pela detecção de casos especiais, como operações que envolvam números zeros, infinitos ou números inválidos (NaN), e os respectivos tratamentos, tendo em vista que, esses casos podem ser previstos sem que os dados percorram todo o fluxo da operação. Este bloco é constituído por um verificador de exceções (VERIF_EXC), registradores (*reg_cx* e *reg_cy*), e um tratador de exceções (TRATADOR_EXC).

- **VERIF_EXC**

O bloco verificador de exceções (VERIF_EXC), o qual está apresentado na FIGURA 20, é utilizado para verificar os operandos de entrada da FPU e identificá-los de acordo com a TABELA 1 mostrada na Seção 2.3, onde são apresentados valores usuais e especiais dependendo do padrão de bits do sinal, expoente e mantissa.

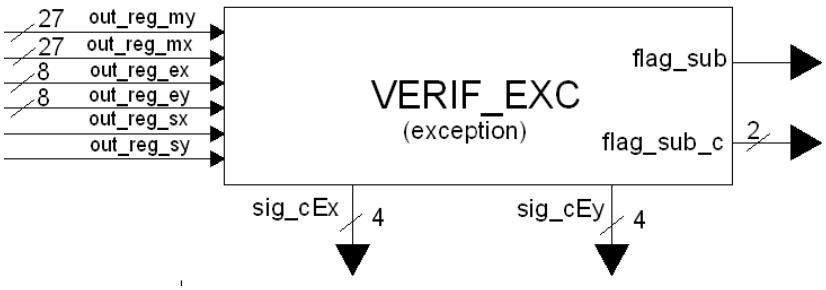


FIGURA 20 - Verificador de exceções.

Este bloco recebe os valores de entrada (sinal, expoente e mantissa) e gera um código que irá identificar cada tipo de representação para cada operando, que será tratado adequadamente no bloco Tratador de Exceção (TRATADOR_EXC). A codificação gerada para cada tipo de representação está mostrada na TABELA 5, onde os sinais *sig_cEx* e *sig_cEy* representam estes sinais para o operando X e o operando Y, respectivamente.

TABELA 5 - Código gerado pelo verificador de exceção.

Representação	***sig_cEx* e *sig_cEy***
Zero positivo	000
Zero negativo	100
NaN (positivo ou negativo)	011
Infinito positivo	010
Infinito Negativo	110
Normal (positivo ou Negativo)	001

De maneira a compreender o funcionamento do bloco VERIF_EXC, no ALGORITMO 3 é mostrado algumas partes importantes do código em SystemVerilog.

```
1   module exception
2   input out_reg_sx,
3   input out_reg_sy,
4   input logic [7:0]out_reg_ex,
5   input logic [7:0]out_reg_ey,
6   input logic [22:0]out_reg_mx,
7   input logic [22:0]out_reg_my,
8   output logic [1:0]flag_sub_c,
9   output logic       flag_sub,
10  output logic [2:0]   sig_cEx,
11  output logic [2:0] sig_cEy );
12   always_comb
```

```
13        Begin
14        fx1<=((out_reg_mx[0] | out_reg_mx[1] | … | out_reg_mx[22]));
15        fy1<=((out_reg_my[0] | out_reg_my[1] | … | out_reg_my[22]));
16        ex<=((out_reg_ex[0] & out_reg_ex[1] & … & out_reg_ex[7]));
17        ey<=((out_reg_ey[0] & out_reg_ey[1] & … & out_reg_ey[7]));
18        ex1<=((out_reg_ex[0] & out_reg_ex[1] & … & out_reg_ex[7]));
19        ey1<=((out_reg_ey[0] & out_reg_ey[1] & … & out_reg_ey[7]));
20        ex2<=((out_reg_ex[0] | out_reg_ex[1] | … | out_reg_ex[7]));
21        ey2<=((out_reg_ey[0] | out_reg_ey[1] | … | out_reg_ey[7]));
22        fx<=fx1|(!fx1& ((!ex1&ex2)|(ex1&!ex2)));
23        fy<=fy1|(!fy1& ((!ey1&ey2)|(ey1&!ey2)));
24       end
25     always_comb
26       begin
27         x<={out_reg_sx,ex,fx};
28         y<={out_reg_sy,ey,fy};
29       end
30    always_comb
31          case (x)
32              3'b000:  sig_cEx <=3'b000; // +0
33              3'b100:  sig_cEx <=3'b100; // -0
34              3'b010:  sig_cEx <=3'b010; // +inf
35              3'b110:  sig_cEx <=3'b110; // -inf
36              3'b011:  sig_cEx <=3'b011; // NAN
37              3'b111:  sig_cEx <=3'b011; // NAN
38              default: sig_cEx <=3'b001; // Normal
39         endcase
40   always_comb
41        case (y)
42              3'b000:  sig_cEy <=3'b000; // +0
43              3'b100:  sig_cEy <=3'b100; // -0
44              3'b010:  sig_cEy <=3'b010; // +inf
45              3'b110:  sig_cEy <=3'b110; // -inf
46              3'b011:  sig_cEy <=3'b011; // NAN
47              3'b111:  sig_cEy <=3'b011; // NAN
48             default: sig_cEy <=3'b001; // Normal
49      endcase
                    …
```

ALGORITMO 3 - Código utilizado para a verificação de exceções.

O bloco também identifica se os operandos são subnormais através de *flags* que serão enviadas ao controle geral da FPU. Estas *flags* estão grifadas em negrito, sendo que código apresentado no ALGORITMO 4 é uma continuação do ALGORITMO 3.

```
      ...
50 always_comb
51    begin
52     if  ((fx1 == 1'b1)&(ex2==1'b0))
53           flag_sub_x<=1'b1;
54     else
55          flag_sub_x<=1'b0;
56    end
57 always_comb
58    begin
59     if ((fy1 == 1'b1)&(ey2==1'b0))
60        flag_sub_y<=1'b1;
61    else
62        flag_sub_y<=1'b0
63    end
64  always_comb
65   begin
66    flag_sub_c   <={flag_sub_y,flag_sub_x};
67    flag_sub   <= (flag_sub_y | flag_sub_x);
68    end
69 end module
```

ALGORITMO 4 - Identificação de números subnormais.

No ALGORITMO 4, o sinal *flag_sub* indica se pelo menos um dos operandos é subnormal. Enquanto que o sinal *flag_sub_c* fornece um padrão de bits que serve para identificar se o operando X ou Y é subnormal ou se ambos são. Na TABELA 6 é mostrado o código utilizado pelo sinal *flag_sub_c* para designar cada operando.

TABELA 6 - Código gerado para identificação de operando subnormais.

flag_sub_c	Significado
00	X e Y normais
01	Y normal e X subnormal
10	Y subnormal e X normal
11	X e Y subnormais

- **Reg_cx e Reg_cy**

Nota-se que no ALGORITMO 3, os valores são atribuídos aos sinais *sig_cEx* e *sig_Cy*. Estes valores são armazenados nos registradores *reg_cx* e *reg_cy*, ambos do tipo *reg_n_bits*. Estes registradores, os quais estão mostrados na FIGURA 21, possuem uma palavra de 3 bits, e sinais de controle de clock, escrita e reset.

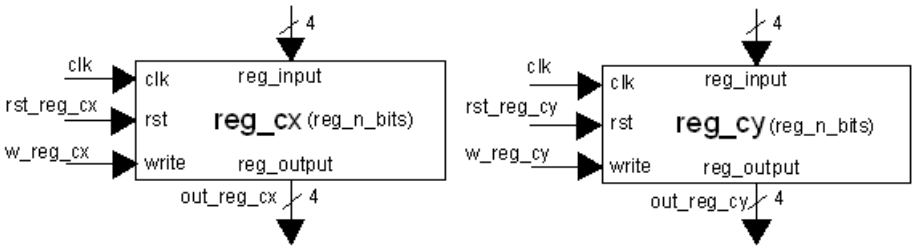


FIGURA 21 - Registradores para armazenar codificação gerada por VERIF_EXC.

- **TRATADOR_EXC**

Sabendo da característica do operando X ou Y através do código gerado pelo verificador de exceções, há também a necessidade de verificar a operação escolhida pelo usuário. Essa verificação é necessária, pois ambas as operações possuem comportamentos diferentes para os mesmos valores de entrada. O bloco tratador de exceções realiza essa verificação com base em tabelas que possuem todos os valores possíveis de exceções, inclusive operações entre números especiais e números normais, de maneira a ter uma representação para cada caso.

Na TABELA 7 estão mostrados os casos que podem ocorrer quando realizada a soma entre números especiais e números normais de acordo com o padrão IEEE754. Considere que N é um número normal diferente de zero.

TABELA 7 - Operações de exceção da soma.

X / Y	**NaN**	**0**	**+ ∞**	**- ∞**	**N**
NaN	NaN	NaN	NaN	NaN	NaN
0	NaN	0	+ ∞	- ∞	N
+ ∞	NaN	+ ∞	+ ∞	NaN	+ ∞
- ∞	NaN	- ∞	NaN	- ∞	- ∞
N	NaN	N	+ ∞	- ∞	**Fluxo Somador**

Analisando a TABELA 7, pode-se observar que para a subtração, bastaria inverter os sinais do operando Y e tratá-lo como uma soma. A única diferença ocorre nas operações entre números infinitos, como mostrado na TABELA 8.

TABELA 8 - Operações de exceção da subtração.

X / - Y	**NaN**	**0**	**-(+ ∞)**	**-(- ∞)**	**N**
NaN	NaN	NaN	NaN	NaN	NaN
0	NaN	0	+ ∞	- ∞	n
(+ ∞)	NaN	+ ∞	NaN	+ ∞	- ∞
(- ∞)	NaN	- ∞	- ∞	NaN	+ ∞
N	NaN	N	- ∞	+ ∞	**Fluxo Somador**

Para a multiplicação entre dois operandos, a TABELA 9 mostra todas as operações possíveis entre números especiais ou normais.

TABELA 9 - Operações de exceção da multiplicação.

X / Y	**NaN**	**0**	**+ ∞**	**- ∞**	**N**
NaN	NaN	NaN	NaN	NaN	NaN
0	NaN	0	NaN	NaN	0
+ ∞	NaN	NaN	+ ∞	- ∞	+ ∞
- ∞	NaN	NaN	- ∞	+ ∞	- ∞
N	NaN	0	+ ∞	- ∞	**Fluxo Multiplicador**

Na operação de divisão, as possíveis operações entre números especiais ou não encontram-se destacadas na TABELA 10.

TABELA 10 - Operações de exceção da divisão.

X / Y	**NaN**	**0**	**+ ∞**	**- ∞**	**N**
NaN	NaN	NaN	NaN	NaN	NaN
0	NaN	NaN	0	0	0
+ ∞	NaN	+ ∞	NaN	NaN	+ ∞
- ∞	NaN	- ∞	NaN	NaN	- ∞
N	NaN	∞	0	0	**Fluxo Divisor**

Assim, havendo um caso especial dentre todas as possibilidades destacadas, o tratador de exceção, o qual está mostrado na FIGURA 22, sinalizará uma flag indicando a ocorrência da exceção com a respectiva representação. A *flag* indicando a exceção é denominda INVALID_flag, enquanto que a representação (sinal, expoente e mantissa) são definidos pelos sinais t_sz, t_ez e t_mz, respectivamente.

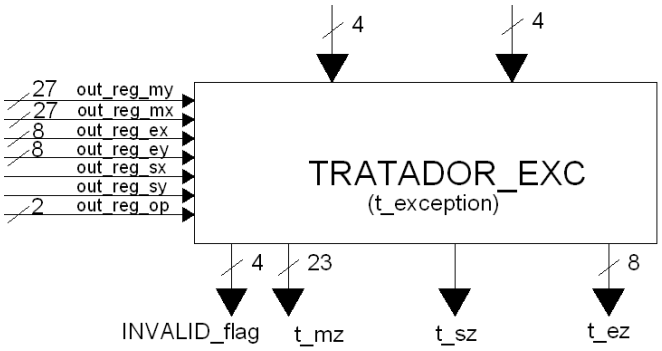


FIGURA 22 - Tratador de exceção.

A lógica de descrição do bloco TRATADOR_EXC em SystemVerilog é baseada nas tabelas apresentadas anteriormente, onde cada possibilidade é testada dentro da operação escolhida, sendo gerado um sinal interno para que a representação seja enviada para a saída. Para a exemplificação do código, no ALGORITMO 5 está destacado apenas um caso da operação de soma: X + 0.

```
1 always_comb
2     case(out_reg_op)
3       //soma
4        2'b00:
5           begin
6              if  (( NAN ));
7                 sel_cod_add<=3'b011;
8                  else if (( zero ));
9                    sel_cod_add<=3'b000;
10                 else if ((+ infinito))
11                  sel_cod_add<=3'b010;
12                 else if ((-infinito))
13                  sel_cod_add<=3'b110;
14                 else if   ((out_reg_cx[2:0] == 3'b001)&&(out_reg_cy[2:0] == 3'b000))
15                  sel_cod_add<=3'b101; \\X + 0
16                else  if ((Y + 0 ))
17                   sel_cod_add<=3'b111;//Y + 0
```

```
18              else ((Número Normal ))
19               sel_cod_add<=3'b001
20      end
            …
21 always_comb
22   case(out_reg_op)
23          2'b00:
24          begin
25              case (sel_cod_add)
26                 3’b001:
27                    ...
28                 3’b000:
29                    ...
30                3’b010:
31                    ...
32                3’b110:
33                   ...
34                3’b101:
35                  begin
36                    t_Sz<=out_reg_sx;
37                    t_Mz<=out_reg_mx;
38                    t_Ez<=out_reg_ex;
39                    INVALID_FLAG<=1'b1;
40                  end
                      …
```

ALGORITMO 5 - Código utilizado para o tratamento da operação (X + 0).

3.4.1.3. Blocos e registradores intermediários

Nesta seção serão vistos blocos e registradores que são utilizados para operações intermediárias na FPU, como definição de sinais, deslocamentos e armazenamentos em geral.

- **Sig_mult_div e reg_sig**

Estes blocos são utilizados para obter e armazenar o sinal da operação a ser realizada. O bloco *sig_nult_div* é dedicado às operações de multiplicação e divisão onde o sinal é determinado por uma porta lógica XOR (ou exclusivo). Lembrando que no fluxo em ponto flutuante destas operações a mantissa é manipulada desconsiderando o sinal, sendo o mesmo reinserido na resposta final. Já na operação de soma ou subtração, o sinal do resultado pode depender dos sinais dos operandos e da operação escolhida. Então, tem-se um registrador de 1 bit de-

nominado *reg_sig* do tipo *reg_n_bits* que será responsável por armazenar o sinal desta operação e o sinal vindo do bloco s*ig_mul_div*. Assim, tem-se a necessidade de um multiplexador de modo a selecionar qual sinal deverá ser gravado no registrador dependendo da operação e sinais de controle. Os blocos *sig_mult_div* e *reg_sig* estão mostrados na FIGURA 23.

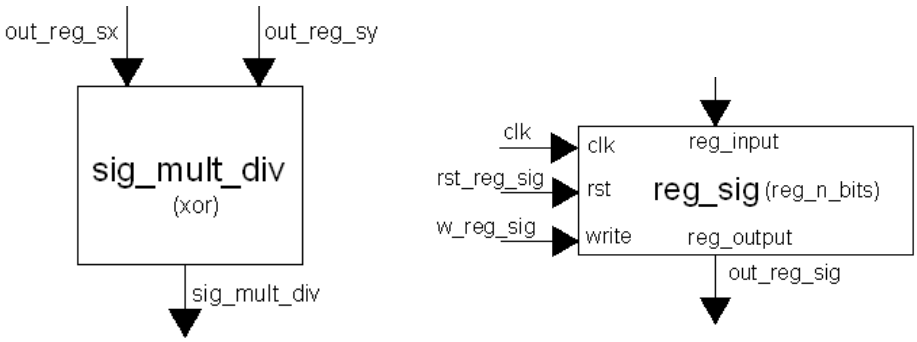


FIGURA 23 - Blocos para cálculo de sinais e armazenamento.

- **Bloco Shifter**

O *shifter* é um deslocador de 54 bits que permite deslocamentos de N bits para direita ou para a esquerda em apenas um ciclo de clock. O *shifter* é utilizado tanto para a soma ou subtração, onde se exige deslocamentos da mantissa para alinhamento, quanto para a multiplicação e divisão, onde são necessário deslocamentos em seus hardwares para cálculos com a mantissa. Também na etapa de normalização é necessário deslocamentos à direita e à esquerda.

O bloco *shifter*, o qual está mostrado na FIGURA 24, possui uma entrada *shamt* de 6 bits que determina o número de vezes que a entrada *in_shifter* de 54 bits deverá ser deslocada, dependendo do sinal de controle *e_d* para definir a direção do deslocamento.

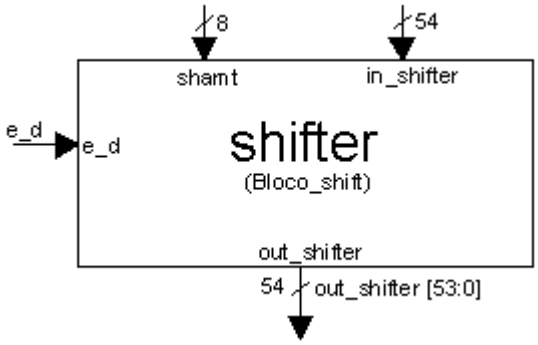


FIGURA 24 - Bloco deslocador de n bits.

O código principal em SystemVerilog do Bloco *Shifter* está mostrado no ALGORITMO 6.

```
1 module Bloco_shift
2 (
3
4        input [53:00]in_shifter,
5        input e_d,
6        input [5:0]shamt,
7        output logic [53:00]out_shifter);
8        logic [53:00]b;
9        logic [53:00]c;
10
11       right          x1  ( in_shifter[53:00] , shamt[5:0] , b[53:00]      );
12       left           x2  ( in_shifter[53:00] , shamt[5:0] , c[53:00]      );
13       Mux_2p_1    # (.n_bits(54))   x3  ( c[53:00] , b[53:00]   ,e_d , out_shifter[53:00]  );
14 endmodule
```

ALGORITMO 6 - Código top-level do shifter com instâncias para o controle e deslocamento de bits.

Neste código, notam-se instâncias declaradas que estão grifadas em negrito. Elas são responsáveis pelos deslocamentos dos bits em ambas as direções. Por exemplo, para o deslocamento para direita, o bloco *right* é utilizado, que por sequência utiliza outra instância interna denominada *Mux54bits* para a lógica de seleção de bits. O código do bloco *right* é mostrado no ALGORITMO 7, e do bloco *Mux54bits* no ALGORITMO 8.

```
1 module right
2   (
3      input logic [53:0]a,
4      input logic [5:0]shamt,
5      output logic [53:0]y);
6
7     Mux54bits  x0 (a[53:0],   shamt[5:0], y[0]);
8     Mux54bits  x1 ({1'b0,a[53:1]   },   shamt[5:0], y[1]);
9     Mux54bits  x2 ({2'b0,a[53:2]   },   shamt[5:0], y[2]);
10    Mux54bits  x3 ({3'b0,a[53:3]   },   shamt[5:0], y[3]);
11                                …
12    Mux54bits  x51 ({51'b0,a[53:51]},   shamt[5:0], y[51]);
13    Mux54bits  x52 ({52'b0,a[53:52]},   shamt[5:0], y[52]);
14    Mux54bits  x53 ({53'b0,a[53]},   shamt[5:0], y[53]);
15
16 Endmodule
```

ALGORITMO 7 - Código para deslocamento dos bits para a direita.

```
1 module Mux54bits
2
3 (
4 input [53:0]a,
5 input [5:0]shamt,
6 output logic y);
7
8 always_comb
9   begin
10
11     if (shamt[5:0]==6'b000000)
12        y = a[0];
13      else if (shamt[5:0]==6'b000001)
14         y = a[1];
15          else if (shamt[5:0]==6'b000010)
16             y = a[2];
17                 …
18           else if (shamt[5:0]==6'b110100)
19             y = a[52];
20          else if (shamt[5:0]==6'b110101)
21             y = a[53];
22          else
23            y = a[53];
24  end
25
26 endmodule
```

ALGORITMO 8 - Código para a seleção dos bits deslocados a direita.

- **Reg_temp**

Registrador temporário de 54 bits do tipo *reg_n_bits* utilizado para armazenar cálculos intermediários da soma, subtração, multiplicação e divisão. Grande parte dos resultados é armazenada nele antes de serem enviados às etapas de normalização e arredondamento. O *Reg_temp*, o qual está mostrado na FIGURA 25, possui sinais de clock, reset e escrita.

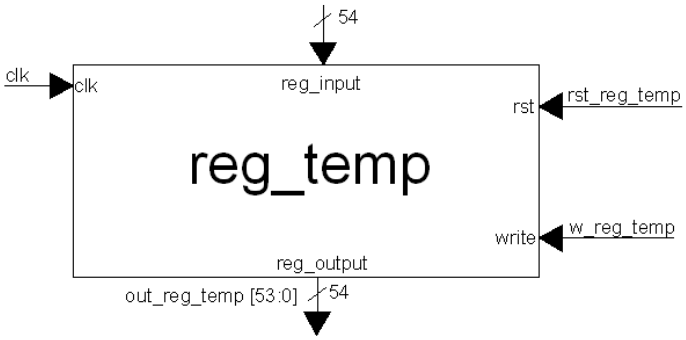


FIGURA 25 - Registrador temporário para operações intermediárias.

- **Reg_q**

Diferentemente dos outros registradores onde os dados eram carregados de forma paralela, este registrador é utilizado de forma serial, ou seja, cada bit entra serialmente no registrador de 27 bits, precisando de 27 ciclos de clock para que o dado seja armazenado no registrador. Este registrador é dedicado à operação de divisão, onde o resultado é obtido de forma sequencial bit a bit. Este registrador, o qual está mostrado na FIGURA 26, possui a entrada de 1 bit, uma saída de 27 bits e sinais para escrita, clock e reset.

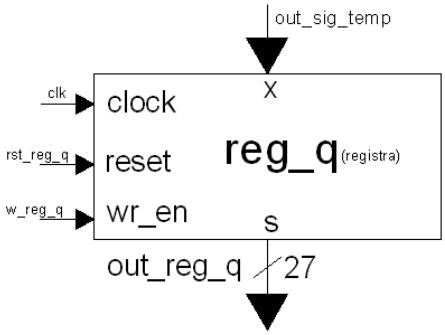


FIGURA 26 - Registrador série para armazenar o resultado da divisão.

O código em SystemVerilog do bloco *reg_q* está mostrado no ALGORITMO 9.

```
1 module registra
2 (
3
4   input       x,
5   input       reset, wr_en, clock,
6   output logic      [26:0] s);
7
8    flip_flop f26 (x, reset, wr_en, clock, s[0]);
9    flip_flop f25 (s[0], reset, wr_en, clock, s[1]);
10   flip_flop f24 (s[1], reset, wr_en, clock, s[2]);
```

```
11    flip_flop f23 (s[2], reset, wr_en, clock, s[3]);
12              …
13    flip_flop f2  (s[23], reset, wr_en, clock, s[24]);
14    flip_flop f1  (s[24], reset, wr_en, clock, s[25]);
15    flip_flop f0  (s[25], reset, wr_en, clock, s[26]);
16
17 endmodule
18
19 module flip_flop
20
21  (
22   input d, rst, w_e, clk,
23   output logic q);
24
25   always_ff @ (posedge rst or posedge clk)
26     begin
27       if (rst) q<=0;
28       else if (w_e) q<=d;
29     end
30 endmodule
```

ALGORITMO 9 - Código do registrador utilizado para armazenar o quociente da divisão.

- **Reg_mz**

Registrador utilizado para armazenar a mantissa intermediária ou definitiva da operação. Este registrador de 27 bits também é do tipo *reg_n_bits*, porém possui sinais para a identificação de bits que serão utilizados para processos de normalização e arredondamento. Os sinais *norm* e *msb* são utilizados para identificar se o número está normalizado, enquanto que *bg0* e *bg1* representam os bits de guarda e são utilizados nas etapas de arredondamento. *Reg_mz*, o qual está ilustrado na FIGURA 27, ainda possui sinais de escrita, reset e clock.

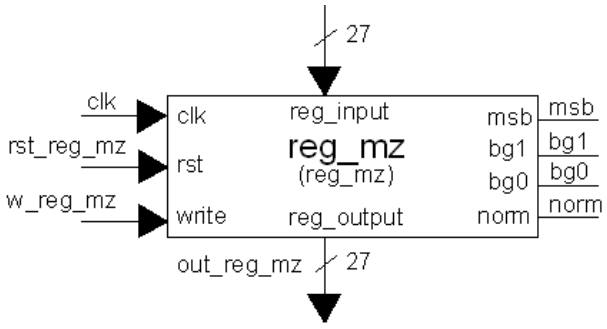


FIGURA 27 - Registrador para armazenar o resultado intermediário e final.

O código em System Verilog de *reg_mz* está mostrado no ALGORITMO 10.

```
1 module reg_mz
2  #(parameter integer Data_Width =27)
3  (clk, rst,write, reg_input, reg_output, msb, norm, bg0, bg1,bg2);
4   input logic clk;
5   input logic rst;
6   input logic write;
7
8   input logic  [Data_Width-1:0] reg_input;
9   output logic msb;
10  output logic norm;
11  output logic bg0;
12  output logic bg1;
13 output logic [Data_Width-1:0] reg_output;
14
15  always_ff @ (posedge clk)
16      if (rst) reg_output <=0;
17      else if (write)
18          reg_output <=reg_input;
19   always_comb
20          begin
21            msb <=  reg_output [Data_Width-1];
22            bg0 <= reg_output [0];
23            bg1 <= reg_output [1];
24            norm <= ((~reg_output[Data_Width-1]) & reg_output[Data_Width-2]);
25          end
26  endmodule
```

ALGORITMO 10 - Registrador de n bits com identificação de bits de normalização e de guarda.

- **Reg_ez**

Registrador do tipo *reg_n_bits* utilizado para armazenar o expoente final e intermediário de uma operação. Este registrador possui o tamanho de 9 bits e não 8 bits como era de se esperar para o expoente. Isto se deve ao fluxo da multiplicação em ponto flutuante. No fluxo, os expoentes são somados resultando em uma dupla polarização, o que poderia acarretar no término da operação se o resultado fosse traduzido como *overflow*. Como o *bias* é subtraído, o resultado volta a ter 8 bits. *Reg_ez* ainda possui sinais de escrita, reset e clock, que podem ser vistos na FIGURA 28.

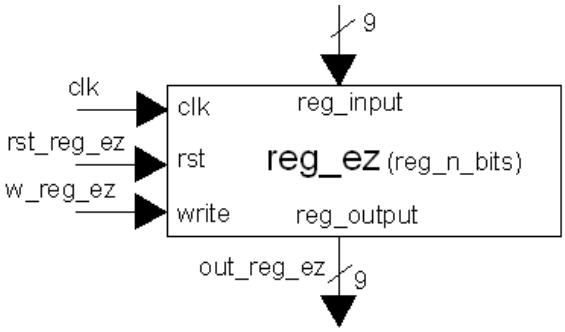


FIGURA 28 - Registrador para armazenar expoentes intermediários e finais.

- **Reg_difexp**

Registrador de 8 bits do tipo *reg_n_bits* utilizado para armazenar a diferença entre os expoentes de X e Y. Essa diferença é necessária para etapas de alinhamento dos expoentes, onde a mantissa é deslocada para direita ou para esquerda dependendo do valor do registrador. Este registrador, o qual está mostrado na FIGURA 29, possui sinais de clock, escrita e reset.

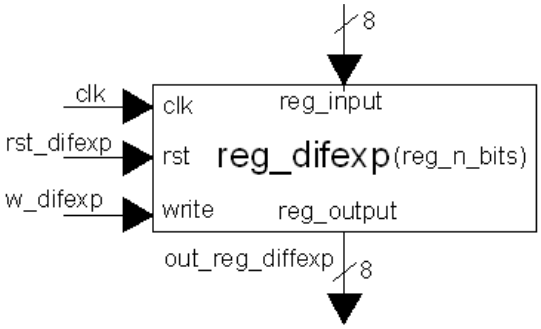


FIGURA 29 - Registrador para armazenar a diferença entre os expoentes X e Y.

- **Sig_temp**

Registrador do tipo *reg_n_bits* utilizado para armazenar o sinal do resultado das operações de soma ou subtração, e operações intermediárias. Este registrador é de um bit, e possui sinais de clock, escrita e reset. Na FIGURA 30 é mostrado o registrador sig_temp.

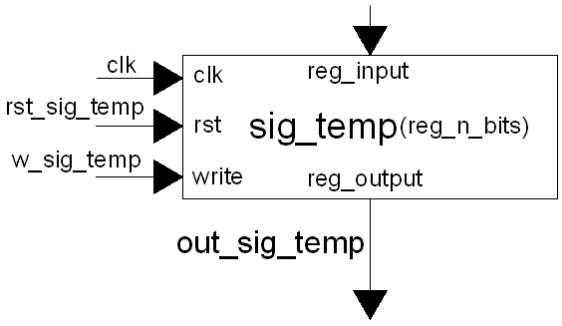


FIGURA 30 - Registrador para o sinal de operações como soma e subtração.

- **LOD e reg_nze**

O LOD (*Leading One Detector*) consiste em detectar, na palavra inserida em sua entrada, a posição do primeiro bit com o valor “1”. Ou seja, o LOD percorre a palavra em busca deste bit e quando o encontra fornece em sua saída o número da posição. Assim, esta posição é armazenada no registrador *reg_nze* de 5 bits do tipo *reg_n_bits*. Os dois blocos estão mostrados na FIGURA 31.

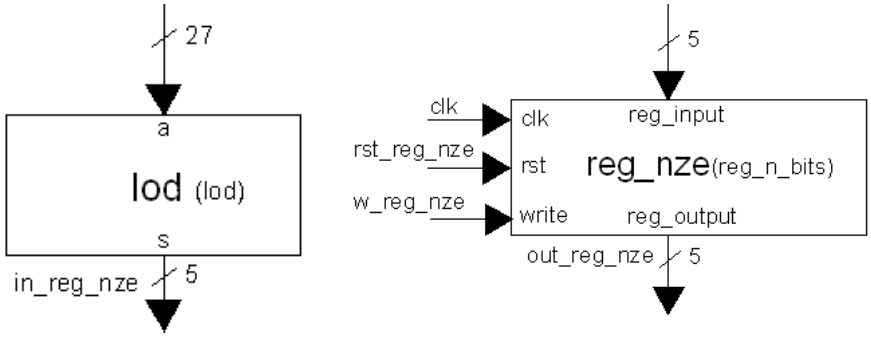


FIGURA 31 - LOD e registrador responsável por armazenar o conteúdo do LOD.

O código em System Verilog do bloco LOD está mostrado no ALGORITMO 11.

```
1 module lod
2 (
3 input [26:00]a,
4 output logic [4:0]s);
5
6 always_comb
7  begin
8    if (a[26]==1)
9      s <= 5'b00000;
10  else if (a[25]==1)
11     s <= 5'b00001;
12  else if (a[24]==1)
13     s <= 5'b00001;
14  else if (a[23]==1)
15     s <= 5'b00010;
16              ...
17  else if (a[2]==1)
18     s <= 5'b10111;
19  else if (a[1]==1)
20     s <= 5'b11000;
```

```
21  else if (a[0]==1)
22     s <= 5'b11001;
23  else
24     s <= 5'b11010;
25 end
26 endmodule
```

ALGORITMO 11 - Código para detecção da posição do primeiro "1" em uma palavra Binária.

Suponha-se que a palavra inserida na entrada do LOD seja o valor 001101_2 de 6 bits. Logo, o LOD analisa bit por bit a palavra, do bit mais significativo para o menos significativo, fornecendo a posição 2 em sua saída.

3.4.1.4. Bloco Somador

Este bloco é a parte principal de todas as operações realizadas na FPU e merece um melhor detalhamento, pois é utilizada tanto na adição, subtração, multiplicação e divisão das mantissas, quanto para processos intermediários na FPU.

3.4.1.4.1. Soma e Subtração em binário

A adição de dois números binários é possivelmente a mais comum das operações que é realizada com dados em um sistema digital (VAHID, 2008). A adição é exatamente o que se esperaria nos computadores. Dígitos são somados bit a bit, da direita para a esquerda, com *carries* (“vai-uns”) sendo passados para o próximo dígito à esquerda, como fosse feito manualmente (PATTERSON et al., 2005). A subtração utiliza a adição, ou seja, o operando é simplesmente negado antes de ser somado. Esta negação é uma representação denominada de *complemento de dois* que é universalmente utilizada por operações aritméticas. O complemento de dois envolve o complemento de cada bit e a soma de “1” ao valor complementado. Por exemplo, para negar 0001_2, complemente-o para obter 1110_2 e depois some “1” para obter 1111_2. Deste modo, para implementar uma operação $a - b$, basta complementar o operando b. Isto resulta em uma economia de hardware já que torna-se desnecessário a subtração (MURDOCA, 2000).

3.4.1.4.2. Hardware para soma e subtração

Neste contexto, para a realização da soma ou subtração é necessário um hardware que implemente tais operações. O elemento central deste hardware é um somador binário, ou seja,

um circuito lógico que manipule dois operandos de maneira adequada de acordo com a operação escolhida.

Por haver diversas topologias que implementam a soma em hardware, e por a FPU necessitar de um somador de 27 bits, o *carry-look-ahead* (CLA) foi o mais indicado devido ao desempenho apresentado em frequência e área ocupada (SCHLOSSER et al., 2009). Assim, foram utilizados 7 somadores *CLA's* de 4 bits. Na FIGURA 32 é apresentada a arquitetura do somador CLA de 4 bits.

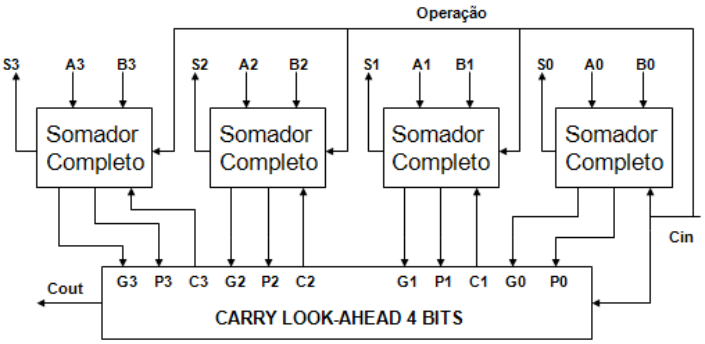


FIGURA 32 - Arquitetura de somador CLA de 4 bits.

Esta arquitetura é formada por somadores completos e por um bloco CLA que tem como objetivo acelerar o cálculo dos *carries* através de uma reestruturação lógica de somadores *ripple carry*. A lógica considera os bits de entrada do somador e fornece os *carries* antecipadamente em processos de geração e propagação (SCHLOSSER et al., 2009).

Os elementos de hardware propostos em Schlosser (SCHLOSSER et al., 2009), fundamentais para a criação de um bloco somador/subtrator que atenda as especificações da FPU são mostrados na FIGURA 33.

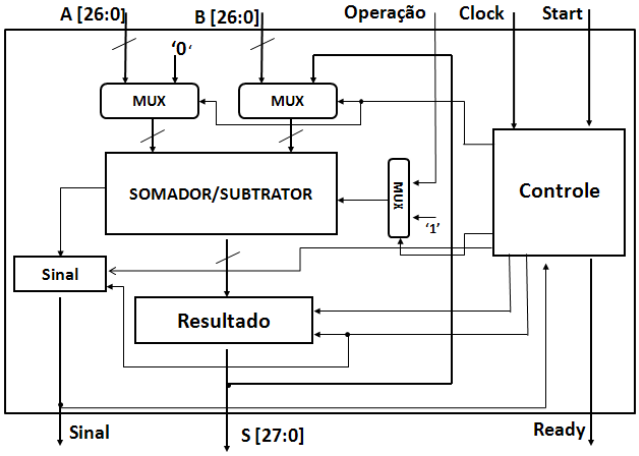


FIGURA 33 - Arquitetura do somador/subtrator de 27 bits em sinal-magnitude.

Nesta arquitetura temos multiplexadores, registradores para armazenar o sinal e o resultado, e um somador binário. Esta arquitetura realiza operações em sinal - magnitude, porém há um controle interno em casos que o operando B ou o resultado forem negativos. Dessa maneira, o operando B ou o resultado são representados em *complemento de dois*. Observa-se ainda que os operandos possuem uma palavra de 27 bits, sendo que são utilizados 7 CLAs de 4 bits resultando em 28 bits. Então os operandos são inseridos adicionando-se zero a esquerda do número.

O funcionamento deste bloco está descrito a seguir:

- **Para operação de soma:**

O bloco realiza a soma dos dois operandos e armazena o resultado nos registradores de saída (*Resultado*). Este valor será escrito nos registradores a partir de um sinal de escrita vindo do controle interno. O controle libera também um sinal de pronto (*Ready*) para identificar o término da operação.

- **Para a operação de Subtração:**

Primeiramente é realizado o complemento de dois do operando B para depois ser somado ao operando A. O resultado desta operação é armazenado no registrador de saída (*Resultado*). Este valor será escrito nos registradores a partir de um sinal de escrita vindo do controle interno. O valor do sinal desta operação (responsável por indicar se o resultado é positivo ou negativo) também deverá ser armazenado em um registrador.

Caso o resultado da operação seja negativo, o valor de 28 bits deve retornar ao multiplexador de entrada do bloco somador para que seja realizado o complemento de dois. Após inverter e somar 1 ao resultado, este valor deverá ser escrito novamente nos registradores de saída. O controle libera também um sinal de pronto (*Ready*) para identificar o término da operação.

3.4.1.4.3. Adder_sub

A arquitetura proposta é então inserida na arquitetura da FPU como *adder_sub*. Na FIGURA 34 estão mostrados todos os sinais de controle, com a adição dos sinais *zero* e *over_exp*. Estes sinais servirão para identificar se uma operação resultou em zero ou overflow.

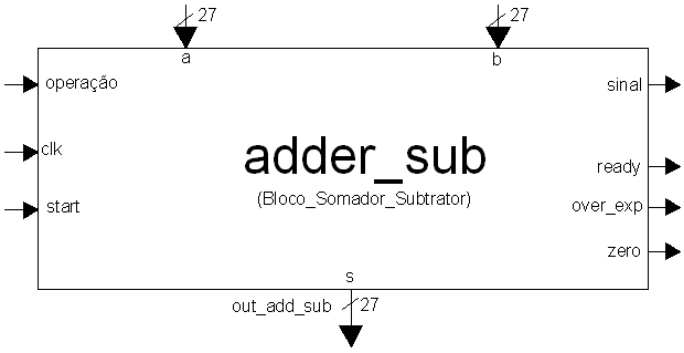


FIGURA 34 - Bloco responsável pelas operações de soma e subtração.

No ALGORITMO 12 é destacado o código do somador CLA de 4 bits utilizado para compor o CLA de 28 bits do bloco *adder_sub*.

```
1 module Somador_Subtrator4bits
2 (
3   input  [3:0]a,
4   input [3:0]c,
5   input sinal,
6   input cin,
7   output logic [3:0]s,
8   output logic cout);
9   logic c0,c1,c2,c3,p0,p1,p2,p3,g0,g1,g2,g3,gen,prop;
10   logic [3:0]b;
11   always_comb
12    begin
13
14      b[0]<= (c[0] ^ sinal);
15      b[1]<= (c[1] ^ sinal);
16      b[2]<= (c[2] ^ sinal);
17      b[3]<= (c[3] ^ sinal);
18
19      s[0] <= ((a[0] ^ b[0])^ cin);
20      s[1] <= ((a[1] ^ b[1])^ c0);
21      s[2] <= ((a[2] ^ b[2])^ c1);
22      s[3] <= ((a[3] ^ b[3])^ c2);
23
24      g0 <= (a[0] & b[0]);
25      g1 <= (a[1] & b[1]);
26      g2 <=(a[2] & b[2]);
27      g3 <=(a[3] & b[3]);
```

```
28      p0 <= (a[0] ^ b[0]);
29      p1 <= (a[1] ^ b[1]);
30      p2 <=(a[2] ^ b[2]);
31      p3 <=(a[3] ^ b[3]);
32
33      c0 <=( g0 | (p0 & cin));
34      c1 <= ( g1 | (p1 & c0));
35      c2 <= ( g2 | (p2 & c1));
36      gen <=( g3 | (p3 & c2));
37      prop <=( p0 & p1 & p2 & p3);
38      cout<= (gen |(prop & cin));
39
40      end
41 endmodule
```

ALGORITMO 12 - Código de um somador CLA de 4 bits.

3.4.2. Descrição do fluxo de dados da FPU

Para o controle de todos os elementos funcionais destacados na seção de descrição organizacional da FPU, foram utilizadas Máquinas de Estados Finitos (UYEMURA, 2002), onde em cada estado sinais de controle são atribuídos à saída de acordo com a entrada.

Para o controle do fluxo de dados da FPU, foram utilizados três blocos de controle, sendo o controle principal (*control_fpu*) responsável por todo o gerenciamento dos estados da soma, normalização, arredondamento e a própria inicialização da FPU. Também, é responsável pela inicialização do controle da multiplicação e da divisão, denominadas *control_mult_int* e *control_div_int* respectivamente. O controle de multiplicação é responsável pelo controle da multiplicação das mantissas, sendo que após realizar esta etapa o mesmo é desativado. O controle da divisão também tem o mesmo princípio, controla somente a divisão das mantissas.

Para facilitar o entendimento e visualização da máquina de estados do controle principal (*control_fpu*), optou-se por dividi-la em estágios, e explicá-las quanto ao funcionamento e caminho de dados na arquitetura da FPU.

3.4.2.1. Estágio I - Inicialização

A inicialização da FPU, gravação de dados de entrada e escolha da operação é apresentada a FIGURA 35, a qual contém o diagrama de estados da parte inicial.

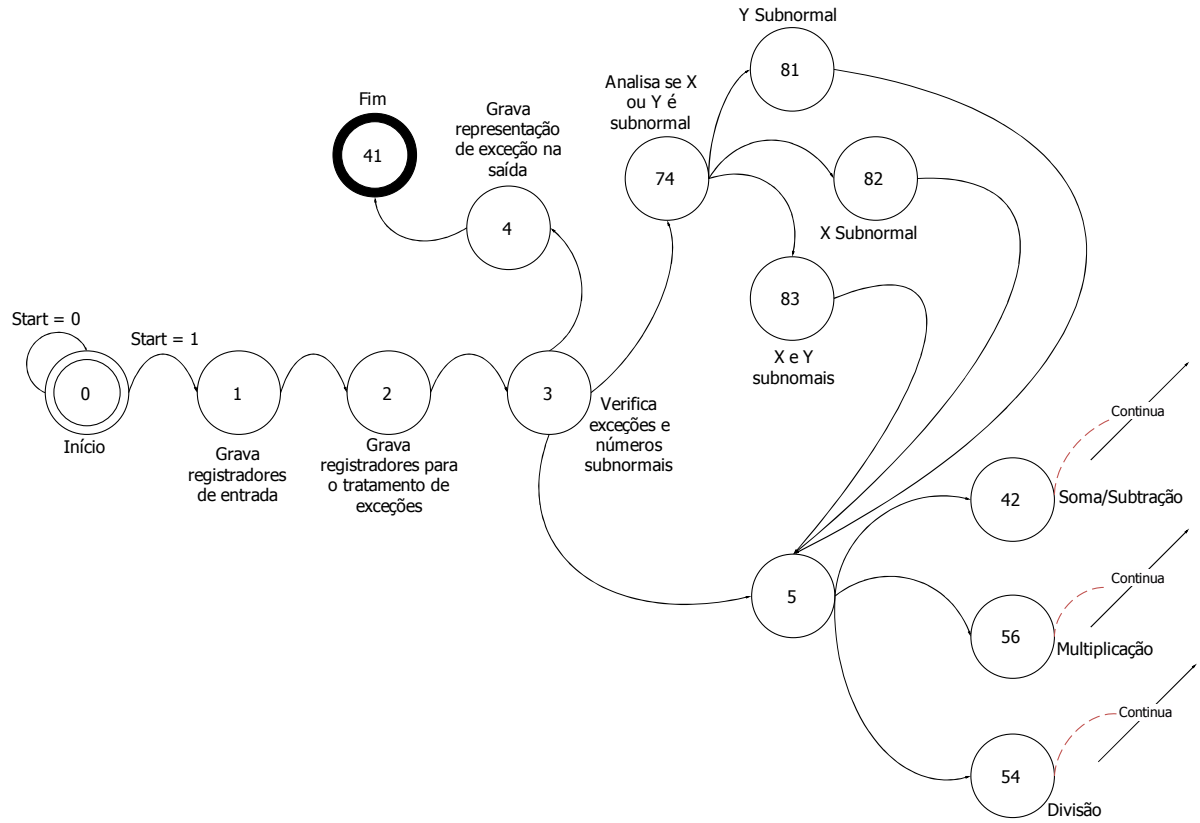


FIGURA 35 - Diagrama de estados na parte de inicialização.

Para dar início às operações, a FPU necessita de um sinal de inicialização, que é verificado no **Estado 0** (zero). A condição de passagem para o estado seguinte é o sinal de *start* em "1". No **Estado 1**, primeiramente são habilitados todos os registradores de entradas através de sinais de escritas. Dessa maneira, cada operando é concatenado em registradores de maneira a separar o sinal, expoente e mantissa. Da mesma forma, os bits do modo de arredondamento e a operação escolhida também são armazenados em registradores específicos. Lembrando que na precisão simples os operandos possuem uma palavra de 32 bits e o modo de arredondamento e operação possuem palavras de 2 bits cada. Assim, totalizamos uma palavra de entrada de 68 bits.

No **Estado 2** os registradores *reg_cx* e *reg_cy* são habilitados de maneira a armazenar o código gerado pelo bloco verificador de exceções. Esse código gerado identifica se um dos operandos pode ser tratado como um número especial ou subnormal pelo Tratador de Exceções (TRATADOR_EXC). O tratador de exceções emite *flags* ao controle principal para que sejam analisadas no **Estado 3**. A *flag* que reproduz uma exceção é denominada de *INVALID_flag*. Caso este sinal esteja ativado, a representação de bits de uma exceção é gravada nos registradores de saída no **Estado 4**. Lembrando que cada tipo de exceção tem sua representa-

ção particular. Logo, a operação será finalizada com a exceção reportada na saída e haverá um bit sinalizando o término da operação (*ready_FPU).*

No mesmo **Estado 3**, é verificado se pelo menos um dos operandos é um número subnormal através do sinal *flag_sub*. Vale lembrar, que quando um número é subnormal ele assume um expoente real de -126 e é interpretado internamente na FPU como zero. Na mantissa a modificação é realizada nos bits implícitos, passando de “01” para “00”. Dessa maneira, no **Estado 74**, é identificado pelo sinal *flag_sub_c* qual operando é subnormal ou se ambos são, cabendo aos **Estados 81**, **82**, ou **83** definir o novo padrão de entrada a ser gravado nos registradores *reg_ex*, *reg_ey*, *reg_mx* e *reg_my*.

Caso não ocorram as situações citadas, é verificado no **Estado 5** o conteúdo do registrador *reg_op* que contém os bits da operação escolhida. Assim, dependendo do código armazenado neste registrador, teremos as operações de soma, subtração, multiplicação e divisão representadas pelos **Estados 42, 56** e **54,** respectivamente.

3.4.2.2. Estágio II - Alinhamento dos Expoentes

Nesta etapa, cujo diagrama de estados é apresentado na FIGURA 36, veremos a operação de soma ou subtração, mais especificamente a parte de alinhamento dos expoentes que são necessários para realização destas operações.

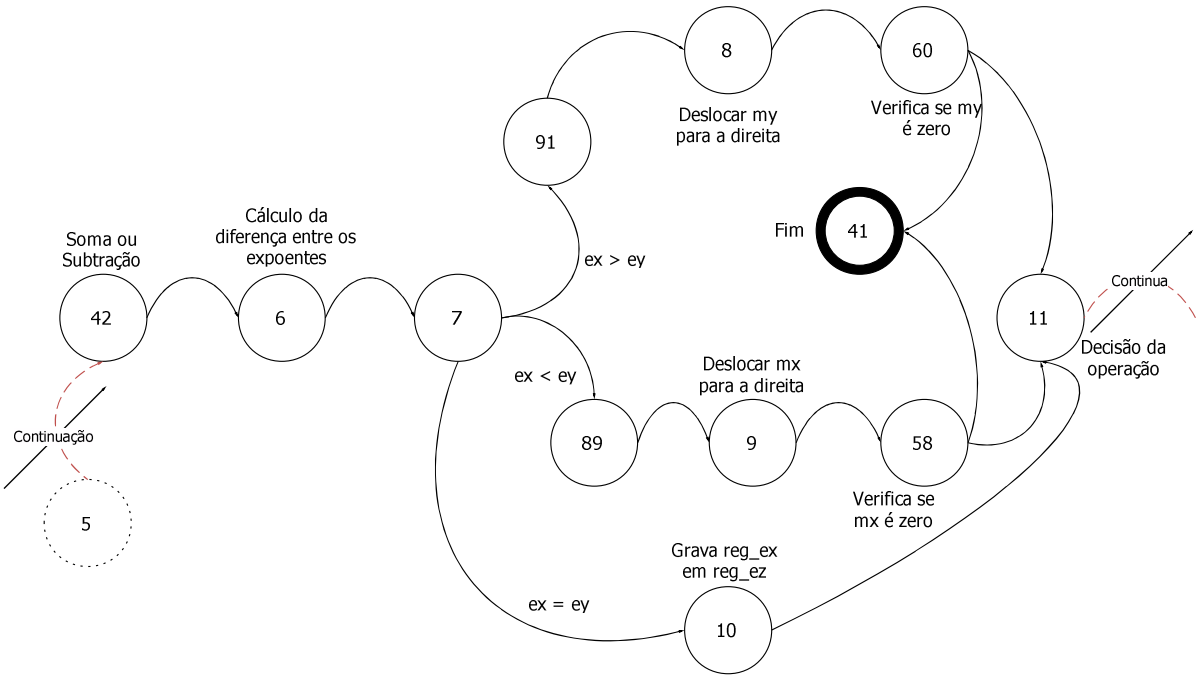


FIGURA 36 - Diagrama de estados referente ao alinhamento dos expoentes.

Suponha-se que no **Estado 5**, a operação escolhida seja a soma ou a subtração. Assim, é enviado um sinal de *start* para o bloco somador (*adder_sub*) para que ele inicie a operação no **Estado 6**. Logo são subtraídos os expoentes e o resultado é armazenado em *reg_difexp* enquanto que o sinal desta operação é armazenado no registrador *sig_temp*. Este sinal armazenado é verificado no **Estado 7**, servindo de base para definir qual mantissa será deslocada. Lembrando que no fluxo da soma em ponto flutuante primeiro é necessário o alinhamento dos expoentes e o deslocamento de uma das mantissas, para depois efetuar a operação escolhida.

Se o conteúdo do registrador *sig_temp* apresentar o valor 0 (zero), entende-se que o expoente X é maior que o expoente Y. Assim, no **Estado 8** deve-se deslocar a mantissa de Y para direita do valor armazenado em *reg_difexp*. Caso *sig_temp* apresente o valor 1 (um), entende-se que o expoente Y é maior do que o expoente X. Sendo que agora, no **Estado 9** a mantissa de X deverá ser deslocada para a direita com o valor de *reg_difexp*. Os deslocamentos mencionados são realizados através do bloco *shifter*, possuindo então uma entrada para o valor que deverá ser deslocado e outra entrada para a quantidade de bits a serem deslocados para a direita. Nestes dois casos temos os **Estados** intermediários **91** e **89** que servem para armazenar em um registrador específico os bits que seriam perdidos nos deslocamentos. Dependendo de quanto for a diferença entre os dois expoentes, tem-se necessidade de verificar se a mantissa não foi inteiramente deslocada para fora do registrador, cabendo aos **Estados 60** e **58** verificar esta situação. Ressalta-se que quando a mantissa é deslocada, ela é armazenada novamente no seu registrador de origem.

Nos dois casos citados, também é verificado se o sinal *zero* do bloco *adder_sub* está ativado. Se estiver ativado, entende-se que o resultado a ser armazenado no *reg_difexp* é diferente de zero. Caso contrário, cairemos no **Estado 10**: expoentes iguais. Deste modo, simplesmente o valor gravado em *reg_ex* é gravado em *reg_ez*.

Ao final do alinhamento dos expoentes, dá-se sequência ao fluxo da soma em ponto flutuante que será mais bem detalhado no Estágio III.

3.4.2.3. Estágio III – Soma ou subtração das mantissas

De acordo com o fluxo da soma em ponto flutuante, após os expoentes estarem alinhados é necessário somar ou subtrair as mantissas. Dessa maneira foi criada a TABELA 11 com todas as possibilidades para a operação.

TABELA 11 - Maneiras de determinar a operação a ser realizada e o sinal da resposta.

Situação	Out_reg_op	Sx	Sy	Add_sub_opc	Operação	Sinal
1	+	+	+	+	**Mx + My**	**Out_sig_temp**
2	+	+	-	-	**Mx – My**	**Out_sig_temp**
3	+	-	+	-	**My – Mx**	**Out_sig_temp**
4	+	-	-	+	**Mx + My**	**Sinal_control**
2	-	+	+	-	**Mx - My**	**Out_sig_temp**
1	-	+	-	+	**Mx + My**	**Out_sig_temp**
4	-	-	+	+	**Mx +My**	**Sinal_control**
3	-	-	-	-	**My - Mx**	**Out_sig_temp**

Nesta tabela considera-se a operação escolhida e os sinais de cada operando para determinar a melhor forma de manipulá-los. O sinal *out_reg_op* representa o conteúdo do registrador *reg_op* e Sx e Sy os conteúdos dos registradores *reg_sx* e *reg_sy*. Para facilitar o entendimento, optou-se por ilustrar os conteúdos dos registradores através de caracteres matemáticos.

Assim, através de uma álgebra booleana é implementada uma lógica combinacional que satisfaça as condições estabelecidas na TABELA 11, sendo que para melhorar a visualização cada situação foi definida com uma cor e um número. Por exemplo, suponha-se que o fluxo caia na Situação 1 cor azul, sendo que na tabela observam-se duas situações em comum e que terão o mesmo tratamento, ou seja, Mx e My serão somados nas duas situações (Mx + My), mesmo que, as operações sejam diferentes e os sinais dos operandos também.

Com a simplificação das possibilidades tem-se uma lógica mais complexa, porém com um código mais simples e reduzido.

O diagrama de estados que exemplifica o estágio de soma ou subtração das mantissas está mostrado na FIGURA 37.

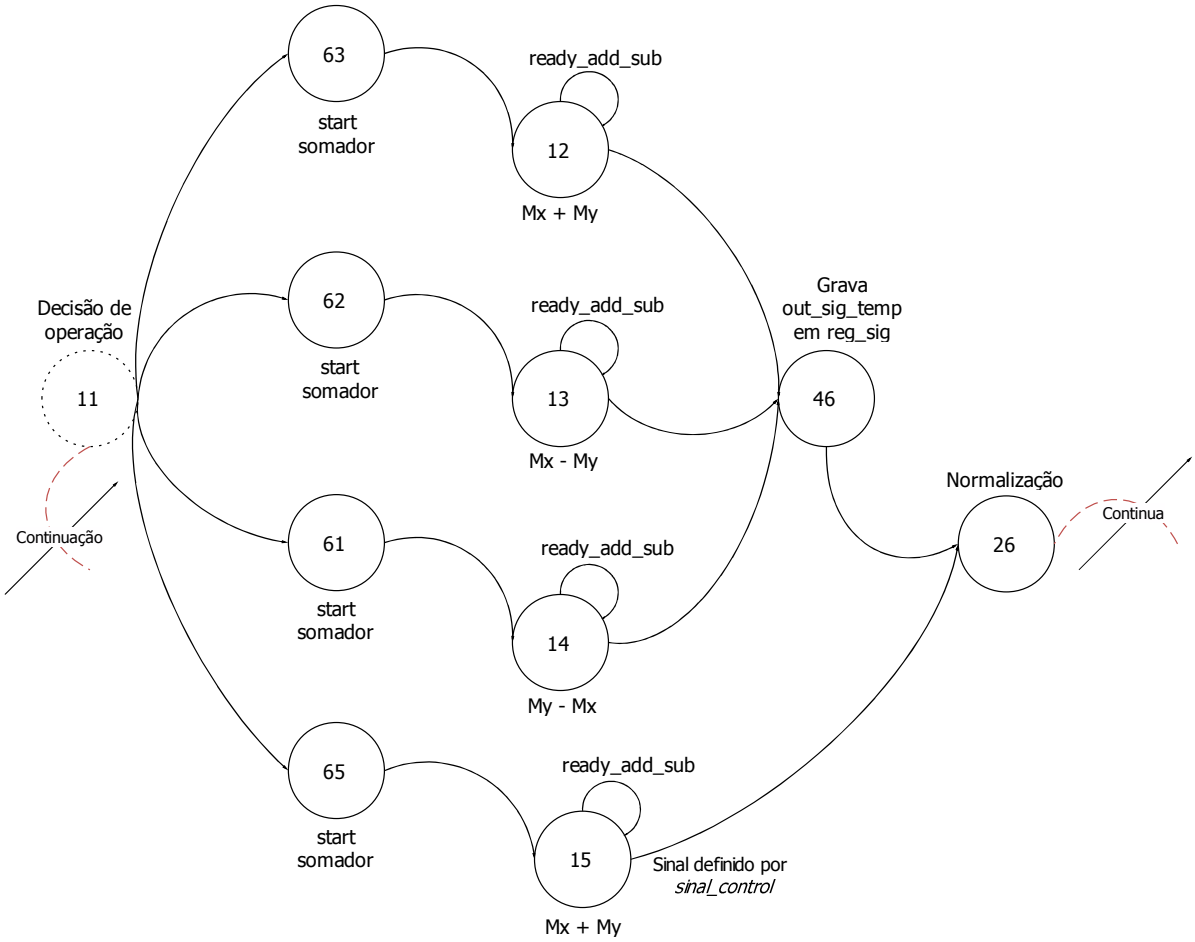


FIGURA 37 - Diagrama de estados para a definição da operação a ser realizada.

No **Estado 11** é verificado em qual das situações os operandos se encaixam para que a operação seja realizada. No **Estado 12, 13, 14** e **15**, temos as Situações 1, 2, 3 e 4 respectivamente, onde é realizada a operação através do Bloco somador *adder_sub* e o resultado é armazenado no registrador *reg_mz*. Anteriormente a estes estados, temos estados intermediários para o *start* do somador. Como observado na coluna Sinal da TABELA 11, nas situações 1, 2 e 3 o sinal da operação é definido pela saída *out_sig_temp* do registrador *sig_temp* e gravado no registrador *reg_sig* no **Estado 46**. Porém, na situação 4, o sinal da operação é definido pelo sinal interno do controle denominado *sinal_control*.

Ao terminar este estágio, temos os 32 bits que correspondem à resposta intermediária, porém ainda separados, sendo que o bit de sinal está em *reg_sig*, o expoente em *reg_ez* e a mantissa em *reg_mz*. Após isto, este resultado deverá ser enviado para as etapas de normalização e arredondamento.

3.4.2.4. Estágio IV- Multiplicação em ponto flutuante

Retornando ao diagrama de estados do Estágio I, mais especificamente no **Estado 5** onde é realizada a escolha da operação, temos a opção da multiplicação em ponto flutuante. O diagrama de estados que está mostrado na FIGURA 38 obedece ao fluxograma descrito na seção 2.4.2 que prevê a soma dos expoentes polarizados, a subtração do *bias* e, por fim, a multiplicação das mantissas.

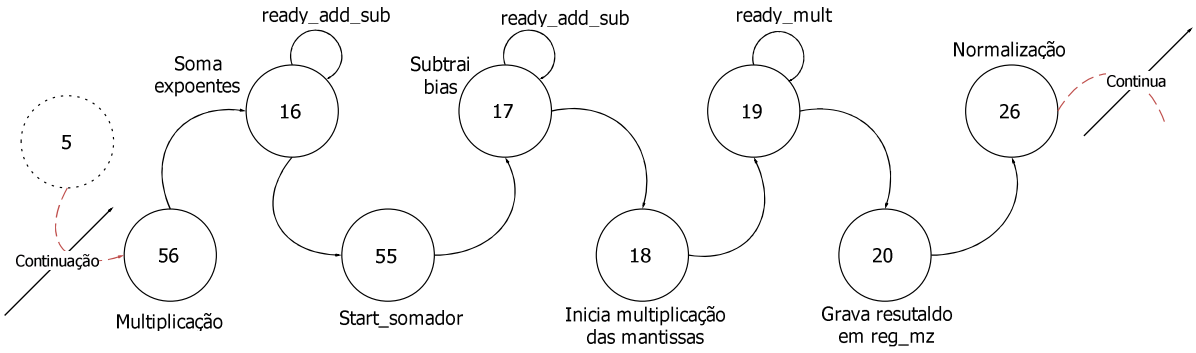


FIGURA 38 - Diagrama de Estados da multiplicação em ponto flutuante.

A multiplicação dá-se início no **Estado 56**, onde é dado o sinal de start para o bloco *adder_sub* realizar a operação no **Estado 16**. Neste estado os expoentes são somados e gravados no registrador *reg_ez*, ao mesmo tempo em que o sinal dessa operação é calculado através do bloco *sig_mult_div* e gravado no registrador *reg_sig*. No **Estado 17**, é subtraído *bias*, já que, quando os expoentes são somados tem-se uma polarização duplicada. Deste modo, o expoente que está armazenado em *reg_ez* é subtraído do valor 127 através do bloco *adder_sub*, sendo que o resultado novamente é gravado em *reg_ez*.

Parte-se agora para a etapa principal da multiplicação em ponto flutuante: a multiplicação das mantissas. A multiplicação é realizada através de um hardware para números inteiros, já que as mantissas dos operandos são números inteiros. Nesta etapa da multiplicação, o bloco *control_mult_int*, que é o responsável pelo controle da multiplicação das mantissas, é acionado pelo bloco *control_FPU* no **Estado 18**. No **Estado 19** o bloco *control_FPU* fica à espera por um sinal de finalização da operação com as mantissas para dar sequência aos estados.

3.4.2.4.1. Multiplicação de inteiros binários e algoritmo sequencial

A multiplicação de números inteiros binários assemelha-se em muito com o método usual utilizado para a multiplicação em decimal (PATTERSON et al., 2002). Na multiplicação em decimal, o algoritmo *"pencil and paper"* consiste em pegar os dígitos um a um do multiplicador, da direita para a esquerda e multiplicar pelo único dígito do multiplicando. Assim, é des-

locado o produto parcial um dígito para esquerda dos produtos parciais anteriores. Na FIGURA 39 está ilustrada a identificação de cada operando.

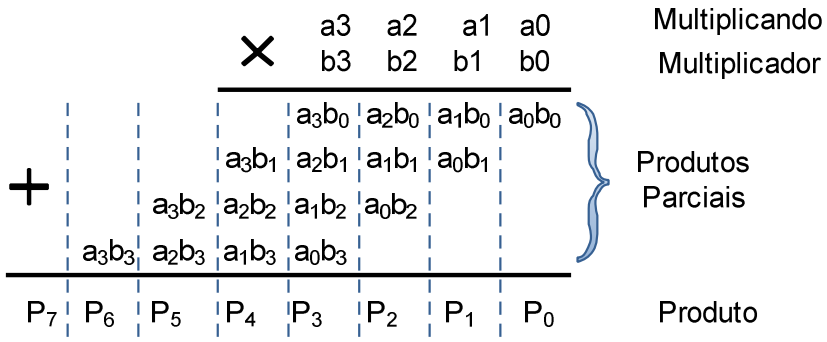


FIGURA 39 - Convenção dos operandos em uma multiplicação de n dígitos.

Observa-se que a multiplicação envolve a geração de produtos parciais, um para cada dígito do multiplicador. Assim, é necessário deslocar o produto parcial gerado um bit para a esquerda e somá-los para obter o produto final. Como a multiplicação é entre números binários, só temos duas opções de dígitos: 0 e 1. Quando o dígito do multiplicador é 0 o produto parcial é zero. Quando é 1, o produto parcial é o próprio multiplicando.

A multiplicação de dois números inteiros binários (*n* x *m*) resulta em um produto com tamanho de (*n* + *m*), e aí se vê a necessidade de um hardware que armazene todos esses bits, sendo que seriam necessários vários registradores para armazenar os produtos parciais para depois somá-los.

De acordo com Stallings (STALLINGS, 2005), podem-se acumular imediatamente cada produto parcial obtido e consequentemente, utilizar um número menor de registradores. Isto pode ser notado no hardware proposto por Patterson (PATTERSON et al., 2005) que está mostrado na FIGURA 40, que consiste em armazenar o produto parcial em um único registrador que também é compartilhado pelo valor do multiplicador. Neste hardware o multiplicando possui um registrador dedicado e o produto parcial é gravado na parte mais significativa do registrador *Produto,* enquanto que o multiplicador é gravado na parte menos significativa do mesmo registrador. Para a soma dos produtos parciais, o hardware possui um somador de números inteiros.

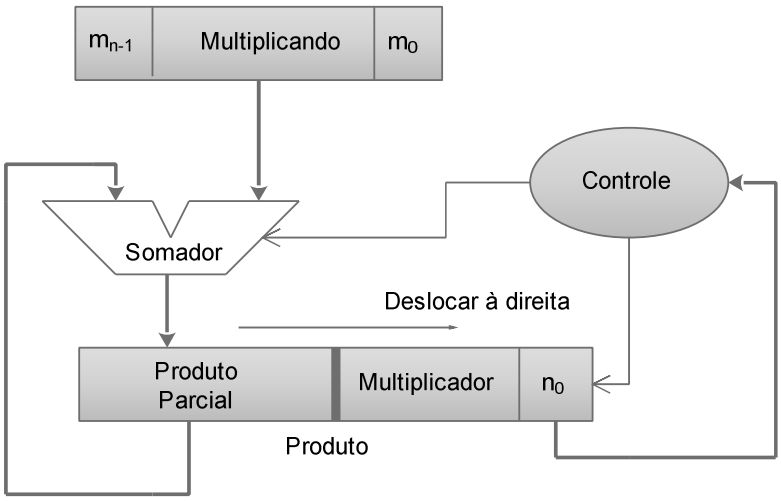


FIGURA 40 - Hardware para a multiplicação de dois números inteiros binários.

O hardware proposto é baseado no algoritmo sequencial para a multiplicação, que possui os seguintes passos:

a) Verificar o bit n_0 do *multiplicador*.

b) Se n_0 for zero, deslocar para direita o registrador *Produto*.

c) Se n_0 for 1, soma-se o registrador *Multiplicando* com o *Produto Parcial* e então desloca-se para direita o registrar *Produto*.

d) Após n deslocamentos, o resultado da operação estará armazenado nos bits menos significativos do registrador *Produto*.

Este hardware sequencial será adotado pelo projeto por apresentar melhores resultados em área do que outros multiplicadores para 27 bits (NEVES et al., 2009). Vale ressaltar que há inúmeros algoritmos e hardwares que implementam esta operação, porém cada um possui uma característica e são adequados para certas aplicações.

Para relacionar esta arquitetura com a arquitetura da FPU, basta designar o registrador *Produto* como *reg_temp* e o somador como *adder_sub*. Os deslocamentos que são mencionados são realizados pelo *shifter*. O multiplicando está no registrador *reg_mx*, enquanto que o multiplicador está gravado nos 27 bits menos significativos de *reg_temp*. Lembrando que o bloco *control_mult_int* é responsável por analisar cada bit de *reg_temp* e efetuar as somas e deslocamentos.

Quando a operação for finalizada, um sinal de pronto (*ready_mult*) é enviado ao Bloco *control_FPU* para que o mesmo dê sequência aos estados. Então, no **Estado 20** o resultado que está armazenado nos 27 bits menos significativos de *reg_temp* é armazenado no registrador *reg_mz*. O sinal está gravado em *reg_sig* e o expoente em *reg_ez*.

A operação é dada como concluída neste estado, passando imediatamente para as etapas de Normalização e Arredondamento.

3.4.2.5. Estágio V – Divisão em Ponto Flutuante

Retornando ao **Estado 5** temos a operação de divisão. Como mostrado na seção 2.4.3, na divisão em ponto flutuante deve-se subtrair os expoente, somar o bias e realizar a divisão das mantissas. O diagrama de estados da divisão em ponto flutuante está mostrado na FIGURA 41.

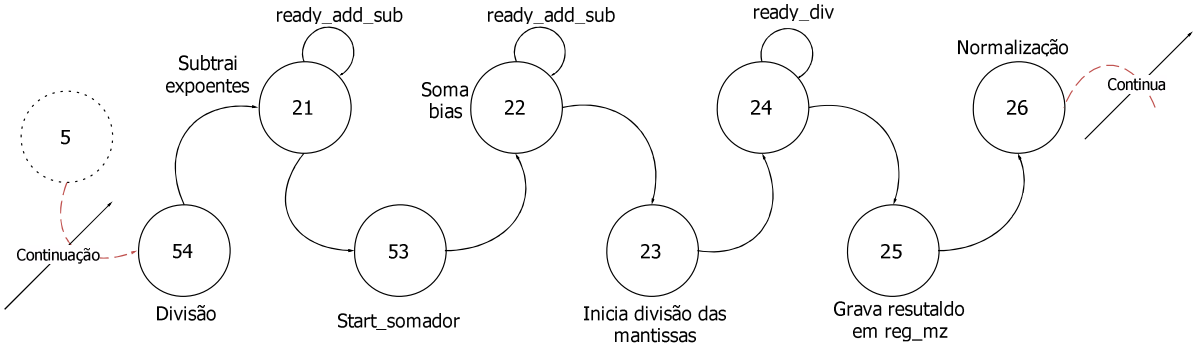


FIGURA 41 - Diagrama de estados para a divisão em ponto flutuante.

A operação é inicializada no **Estado 54**, no qual é enviado um sinal de *start* ao Bloco *adder_sub*. No **Estado 21**, os expoentes são subtraídos e gravados no registrador *reg_ez*, ao mesmo tempo em que o sinal dessa operação é calculado através do bloco *sig_mult_div* e gravado no registrador *reg_sig*. No **Estado 22**, o *bias* é somado, já que na subtração dos expoentes a polarização é retirada. Deste modo, o expoente que está armazenado em *reg_ez* é somado ao valor 127 através do bloco *adder_sub*, sendo que o resultado novamente é gravado em *reg_ez*.

No **Estado 23**, é enviado um sinal de start para o Bloco *control_div_int*, que é o responsável pela divisão das mantissas, sendo que o **Estado 24** fica aguardando a finalização da operação para dar sequência aos estados.

Os operandos a serem divididos são dois números inteiros que estão armazenados nos registradores *reg_mx* e *reg_my*. Como na divisão de dois números inteiros pode resultar em um número real, então é necessário um algoritmo que se aplique a esta situação.

3.4.2.5.1. Divisão de números binários e Algoritmo de Restauração

Muitos algoritmos são utilizados para implementação da divisão de dois números em hardware. Estes algoritmos diferem em muitos aspectos, incluindo convergência de quociente, primitivas fundamentais de hardware e formulações matemáticas (OBERMAN et al., 1997). Na literatura encontram-se algoritmos em que o hardware implementado baseia-se em sucessivas somas e subtrações, multiplicações, tabelas look-up, etc.

O algoritmo de restauração, que descreve o método clássico da divisão, pode ser definido como um método de sucessivas somas e subtrações, tornando-se, neste caso, apropriado para a divisão de números reais e adequado para ser implementado na arquitetura da FPU (CASTRO et al., 2009). Para compreensão do algoritmo de restauração, veremos *a priori* algumas convenções utilizadas para designação dos operandos, assim como o fluxo da operação. Na FIGURA 41, está mostrado o método clássico da divisão "*pencil-and-paper*".

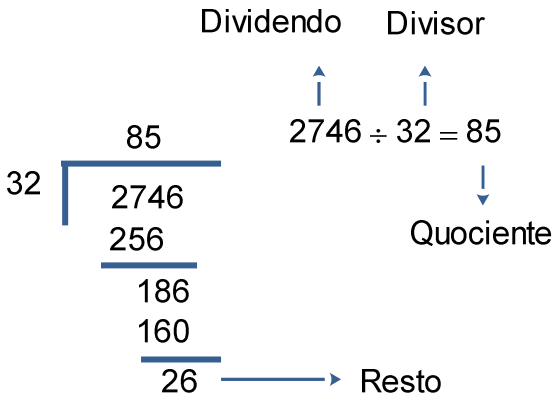


FIGURA 42 - Designação dos operandos para uma divisão.

Aqui, 2746 é referido como dividendo, 32 é o divisor, 85 o quociente e 26 o resto. Em sistemas computacionais, a maioria das operações de divisão está descrita pelo procedimento recursivo representado pela Equação 5:

$$R^{(j+1)} = r \times R^{(j)} - q_{j+1} \times D \qquad (5)$$

onde j = 0,1, ... , n-1, é o índice de recursão e $R^{(j)}$ é o resto parcial da j-ésima iteração. O resto parcial inicial $R^{(0)}$ equivale ao dividendo e $R^{(n)}$ é o resto final. D é o divisor e r é a base. Neste exemplo, r = 10. O quociente q_{j+1} é determinado dígito a digito no procedimento recursivo.

Utilizando a base 2, a Equação 5 pode ser reescrita da seguinte forma:

$$R^{(j+1)} = 2R^{(j)} - q_{j+1} \times D \qquad (6)$$

E com base na Equação 6, tem-se o fluxograma mostrado na FIGURA 43.

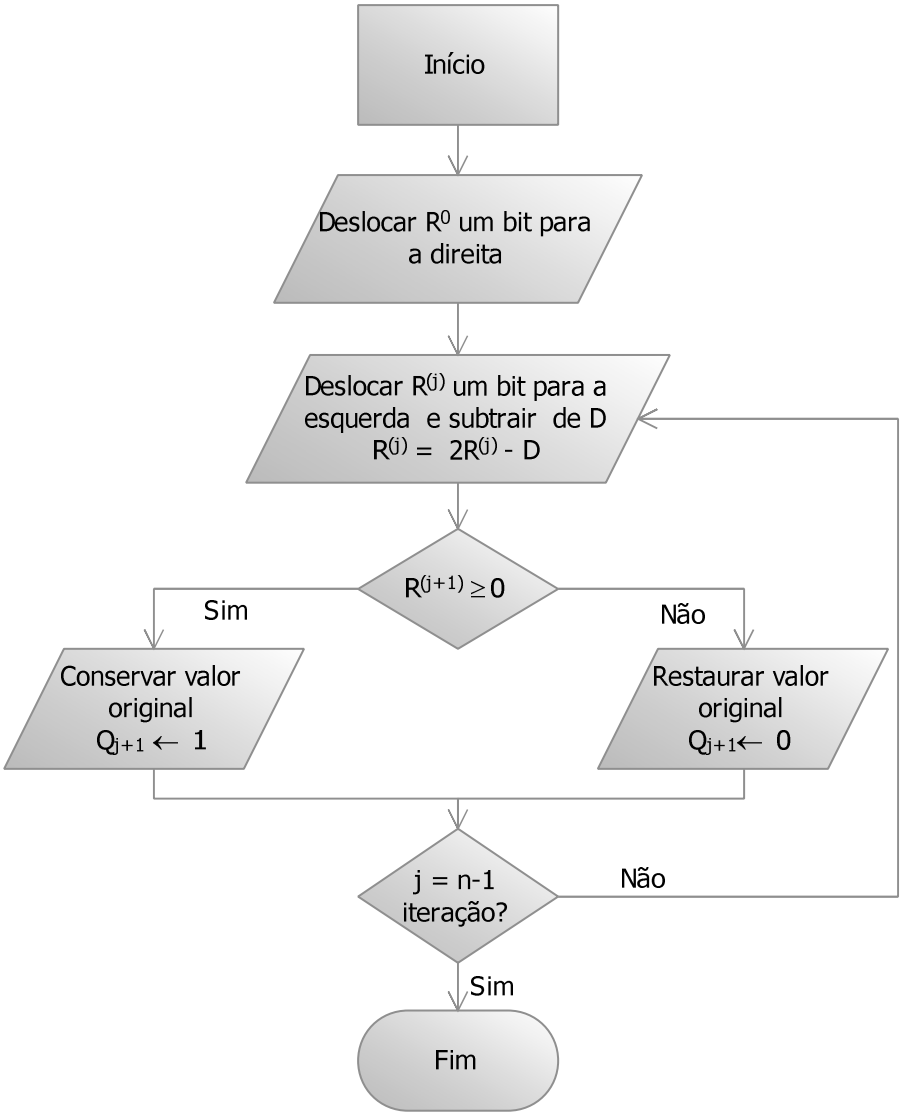


FIGURA 43 - Fluxograma do algoritmo da divisão com restauração.

No primeiro passo deste algoritmo, desloca-se $R^{(0)}$ um bit para direita, sendo este deslocamento necessário para garantir que o dividendo sempre será menor que o divisor. Logo, desloca-se $R^{(j)}$ um bit para esquerda e subtrai-se de D. Deste modo, é verificado o sinal de $R^{(j+1)}$, ou seja, a possibilidade de restauração. Na Equação 6 temos o fator q_{j+1} multiplicando D. Este fator é uma previsão do sinal de $R^{(j+1)}$, ou seja, se $R^{(j+1)} \geq 0$, $q_{j+1} = 1$. Logo, $R^{(j+1)} = 2R^{(j)} - 1 \times D$. Por outro lado, se $R^{(j+1)} < 0$, $q_{j+1} = 0$, logo $R^{(j+1)} = 2R^{(j)} - 0 \times D$, o que seria o mesmo que uma restauração. Como a máquina não faz uma previsão do sinal da operação, ela precisa efetuar inúmeras subtrações e restaurá-las, se necessário.

A arquitetura proposta por Castro (CASTRO et al., 2009), a qual pode ser visualizada na FIGURA 44, expõe a ideia de que os mesmos elementos de hardware utilizados pelo multiplicador podem ser utilizados pelo divisor, traduzindo-se em reusabilidade de blocos funcionais

e economia de hardware. Esta arquitetura foi implementada visando os requisitos da FPU e, portanto, possui os mesmo elementos de hardware que são utilizados na FPU.

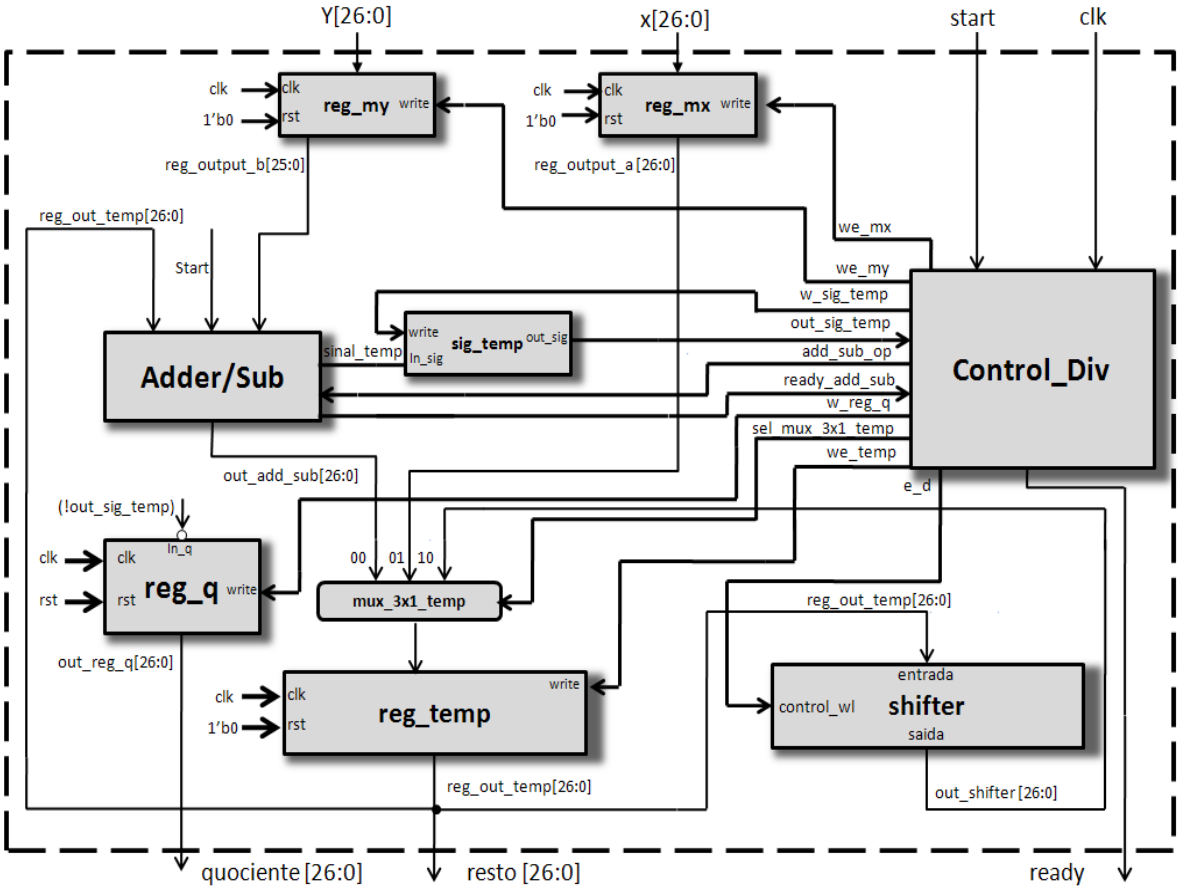


FIGURA 44 - Arquitetura para divisão de mantissas no Padrão IEEE754.

Nesta arquitetura, X é o dividendo e Y o divisor, e ambos são gravados nos registradores *reg_mx* e *reg_my*, respectivamente. O dividendo é gravado em *reg_temp* e deslocado no *shifter* um bit para direita, sendo novamente gravado em *reg_temp*. Este deslocamento só é necessário na primeira iteração. Então, desloca-se o dividendo um bit para a esquerda e subtrai-se do divisor utilizando o Bloco *adder_sub*, tendo o sinal da operação gravado em *sig_temp* e o resultado em *reg_temp*.

O controle analisa o conteúdo de *sig_temp* e determina qual bit deverá entrar em *reg_q* dependendo de haver ou não a restauração. No final da operação, o resultado encontra-se armazenado em *reg_q*. Assim, é enviado ao bloco *control_FPU* um sinal de que a operação está finalizada.

No **Estado 25**, o conteúdo de *reg_q* é armazenado em *reg_mz*, passando agora para os estágios de normalização e arredondamento.

3.4.2.6. Estágio VI – Normalização

Com todos os fluxos das operações definidas, independendo da operação, temos o resultado armazenado nos registradores *reg_sig*, *reg_ez* e *reg_mz*. Sabemos que o resultado a ser apresentado ao usuário deverá estar na sua forma normalizada pelo padrão IEEE754, e que a mantissa do resultado deve ter a forma apresentada na Equação 2. O diagrama de estados da primeira parte da normalização encontra-se ilustrada na FIGURA 45.

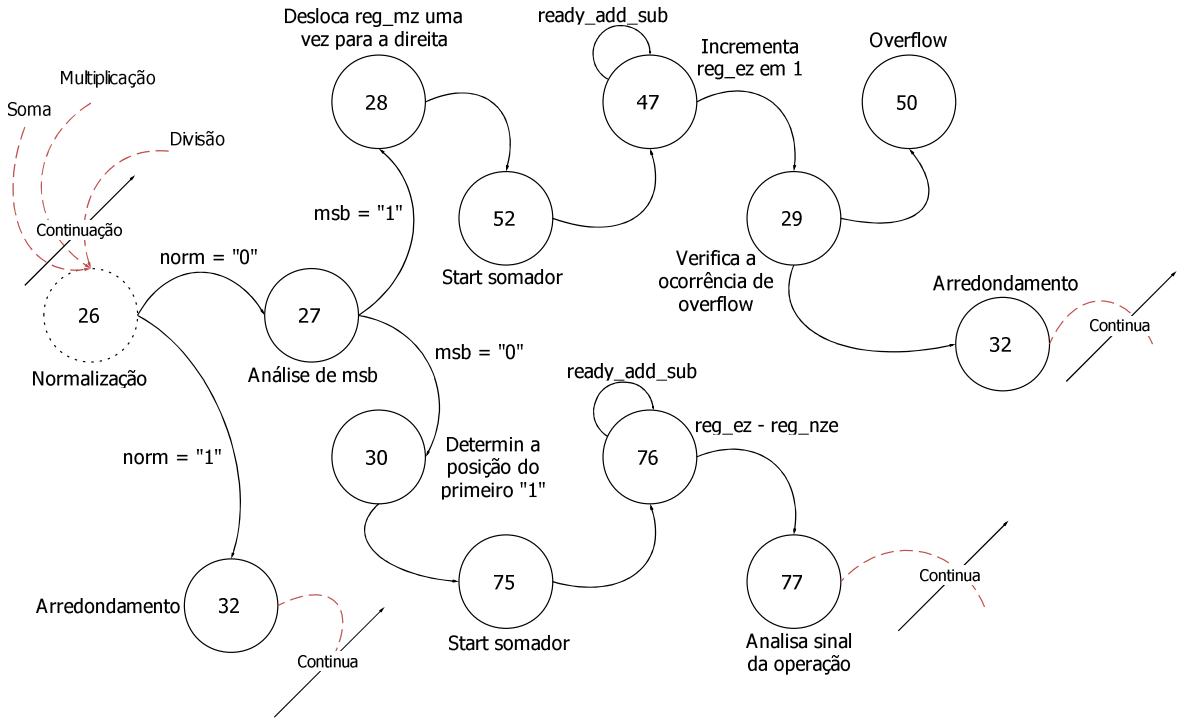


FIGURA 45 - Diagrama de estados para a primeira parte da normalização.

Então, no **Estado 26,** é verificado se o número encontra-se na sua forma normalizada. Primeiramente o controle analisa o sinal *norm* vindo do registrador *reg_mz*. Este sinal é uma verificação dos dois bits mais significativos do registrador *reg_mz*. Se este sinal estiver ativo, entende-se que a mantissa está na sua forma normalizada, bastando efetuar o arredondamento no **Estado 32**. Caso contrário, passamos para o **Estado 27** para a normalização da mantissa.

Como o número está desnormalizado, verifica-se o bit mais significativo de *reg_mz* que é representado pelo sinal *msb* do próprio registrador. Se este sinal estiver ativo, entende-se que o número está a uma posição de sua normalização. Então no **Estado 28**, a mantissa é deslocada no bloco *shifter* uma posição para a direita e gravada novamente em *reg_mz*. Ao se delocar a mantissa, deve-se incrementar *reg_ez* em “1” no **Estado 47** e verificar no **Estado 29** se não ocorreu overflow no expoente. Caso ocorra o overflow, o resultado deve ser tratado no **Estado 50**. Caso contrário, a mantissa é enviada para a etapa de arredondamento.

Retornando ao **Estado 27**, caso o sinal *msb* estiver inativo, compreende-se que o primeiro "1" necessário para deixar o número na forma normalizada pode estar em qualquer posição à direita deste bit. Deste modo, no **Estado 30**, os 27 bits de *reg_mz* são enviados para o bloco *LOD* de maneira que a posição do primeiro "1" seja determinada e gravada no registrador *reg_nze*. Em seguida, o conteúdo de *reg_ez* é subtraído do conteúdo de *reg_nze* de maneira que somente o sinal desta operação é analisado pelo controle. Esta subtração e a análise do sinal devem-se ao pressuposto que o expoente não pode tornar-se negativo, já que o mesmo é decrementado para que a mantissa seja deslocada para a direita. Assim, no **Estado 77**, é analisado o sinal da operação. Este caso pode ser visualizado na segunda parte do diagrama de estados da normalização, o qual está mostrado na FIGURA 46.

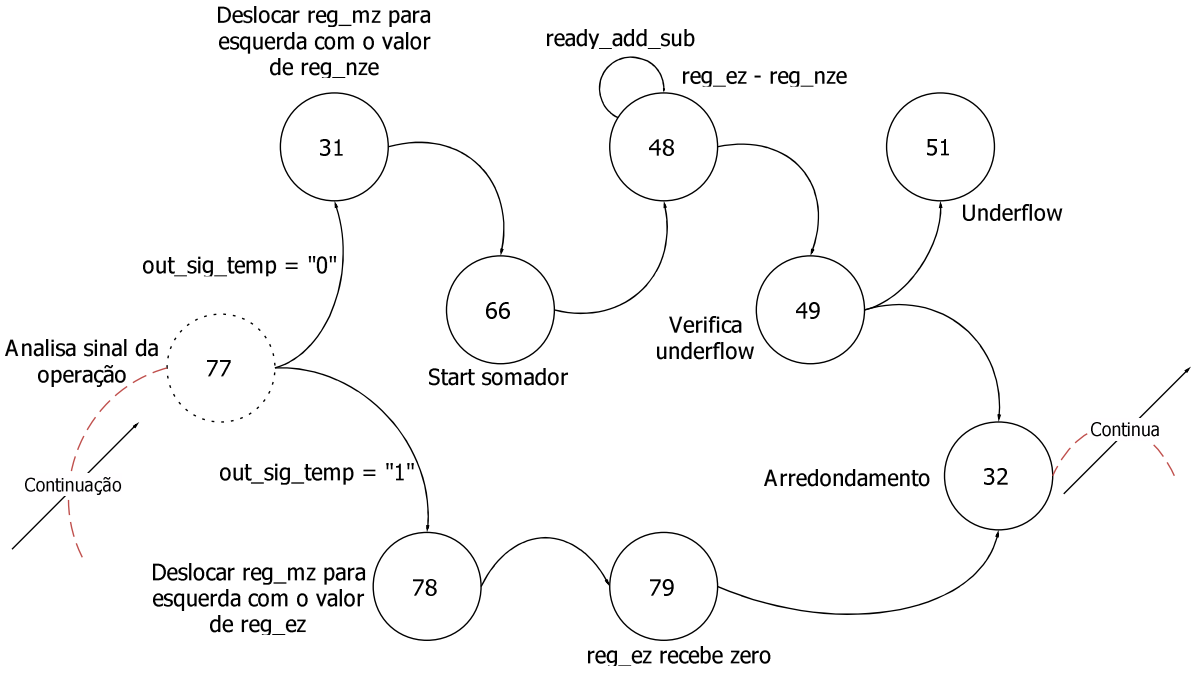


FIGURA 46 - Segunda parte do diagrama de estados para a normalização.

Se o sinal for positivo, no **Estado 31**, a mantissa será deslocada para a esquerda com o valor de *reg_nze* e o expoente decrementado no **Estado 48** com o mesmo valor de *reg_nze*. Antes de passar para a etapa de arredondamento, no **Estado 49**, é verificado a ocorrência de *underflow* no expoente.

Se o sinal da subtração entre *reg_ez* e *reg_nze* for negativo, no **Estado 78**, a mantissa é deslocada para a esquerda com o valor do expoente e o mesmo recebe o valor zero no **Estado 79**.

Após a normalização, passamos para a última etapa: o arredondamento.

3.4.2.7. Estágio VII – Arredondamento

A última etapa do fluxo em ponto flutuante de qualquer operação, que inicia no **Estado 32**, consiste em verificar o método de arredondamento a ser aplicado à mantissa. Lembrando que no padrão IEEE754 temos 4 modos de arredondamento: em direção a zero, em direção ao infinito positivo e negativo, e para o mais próximo. O diagrama de estados do arredondamento está ilustrado na FIGURA 47.

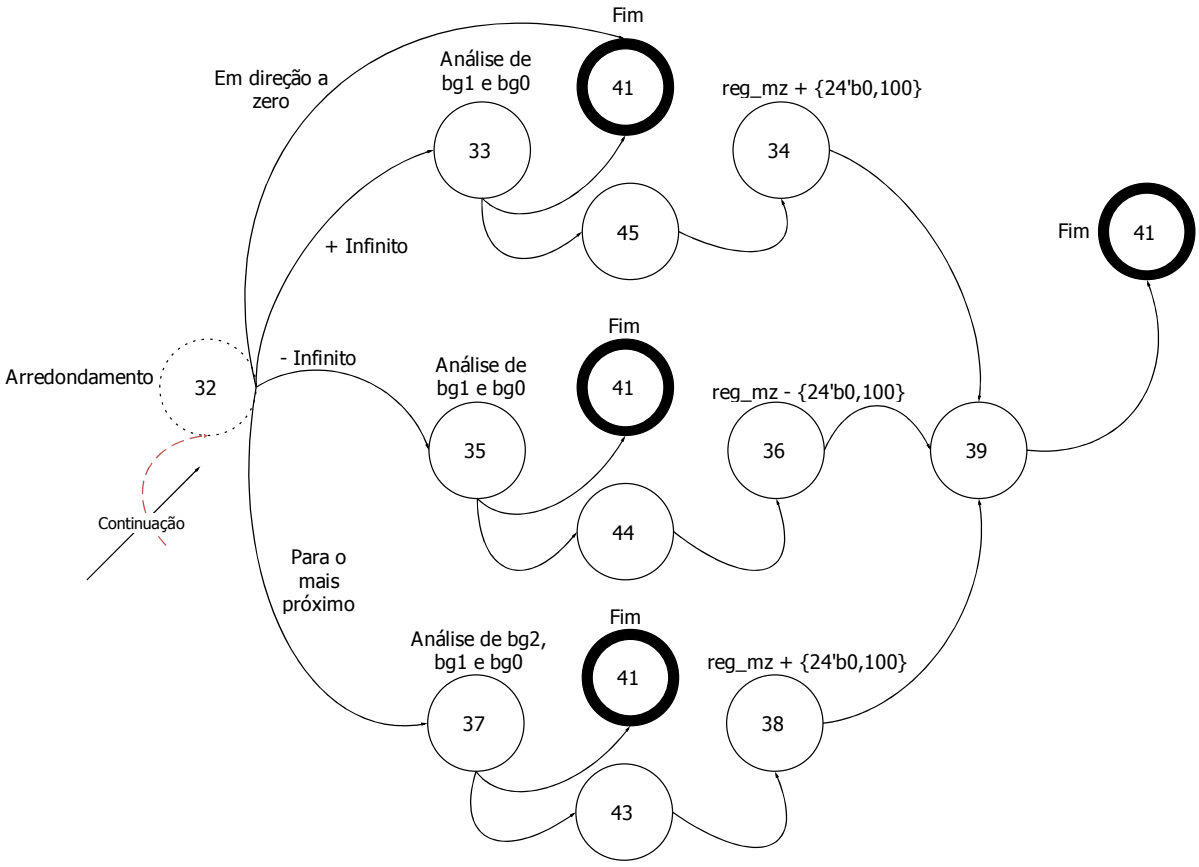


FIGURA 47 - Diagrama de estados para o arredondamento.

O método em direção a zero consiste em simplesmente truncar o número, ou seja, os bits de guarda são descartados e a operação é finalizada no **Estado 41**. Já nos casos de arredondamento para infinito positivo e negativo, é necessário analisar os bits de guarda da mantissa, representados pelos sinais *bg0* e *bg1* vindos do registrador *reg_mz*. Essa verificação é realizada nos **Estados 33** e **35**. Se pelo menos um dos bits de guarda forem igual a "1", é somado "1" ao último bit representável ou subtraído "1" no caso do arredondamento para menos infinito. A soma e a subtração são representadas pelos **Estados 34** e **36**. Caso os bits de guarda não satisfaçam a primeira condição, a mantissa deverá ser truncada dando por realizada a operação no **Estado 41**.

No último caso temos o arredondamento *default* do padrão IEEE754: para o mais próximo. Neste arredondamento, que está representado pelo **Estado 37**, além da análise dos bits de guarda de *reg_mz*, também é necessário analisar o último bit representável que neste caso chamaremos de bg2. Este tipo de arredondamento analisa bg2, bg1 e bg0 simultaneamente para efetuar o arredondamento ou o truncamento. Se bg1 e bg0 ultrapassarem mais que a metade de bg2, é somado “1” ao bg2 no **Estado 38**, caso contrário a mantissa é truncada e finalizada no **Estado 41**.

Ambos os modos de arredondamento passam pelo **Estado 39**, que é dedicado a verificar se o número não foi desnormalizado com o arredondamento.

4 RESULTADOS

Após a proposta da arquitetura e sua descrição, torna-se necessário obter parâmetros para validá-la, como área utilizada, consumo e frequência de operação. Normalmente estes parâmetros são obtidos via ferramentas de síntese, que transcrevem o código em HDL para um nível mais baixo de abstração, como o nível de portas lógicas. Também há a necessidade de testar em um meio físico o circuito sintetizado, sendo que uma possibilidade viável é a utilização de circuitos programáveis. A forma mais popular de tecnologia programável é conhecida como FPGA (*Field - Progamable Gate Array*) (VAHID, 2009). FPGA é um circuito integrado que contém um grande número de blocos lógicos programáveis denominados de CLBs (*Configurable Logic Block*), redes de interconexão e blocos de entrada e saída (DIAS, 2006). Uma arquitetura básica de um FPGA está mostrada na FIGURA 48.

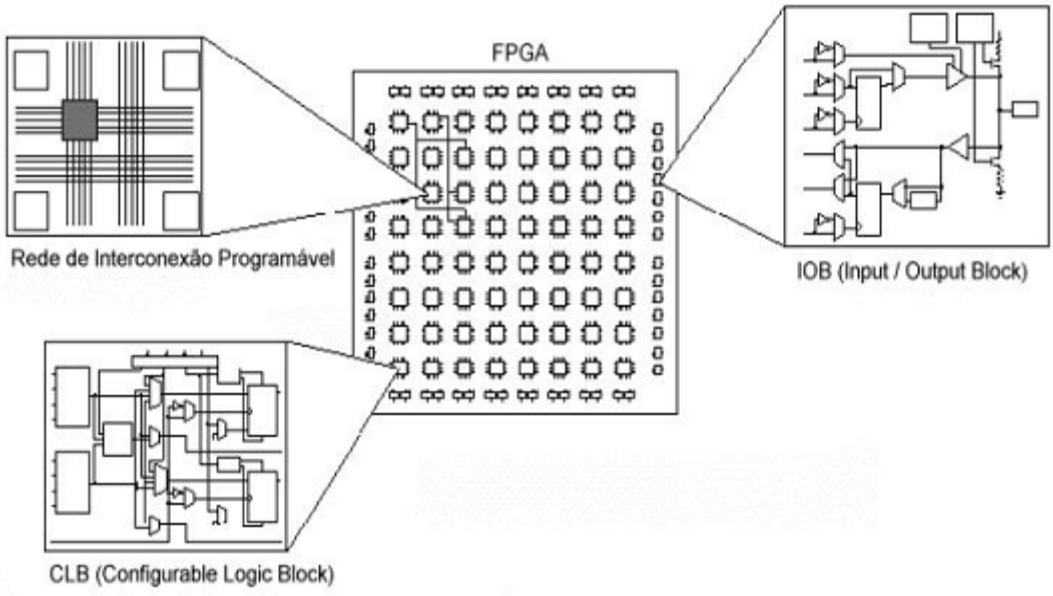


FIGURA 48 - Arquitetura interna de um FPGA.
Fonte: DIAS, 2006.

Quando a ferramenta sintetiza um circuito para um determinado FPGA, ele já especifica a função de cada CLB utilizada e as interconexões necessárias. Cada família de FPGA possui uma designação para CLBs, sendo que em FPGAs da Altera, CLBs são denominados de elementos lógicos (LE's). LE's basicamente são formados por funções combinacionais, multiplexadores e registradores.

4.1.Síntese Lógica

De acordo com (LEITE, 2006), síntese é um processo automatizado de tradução e otimização onde uma especificação mais abstrata é transformada numa especificação menos abstrata. Por exemplo, a síntese lógica consiste na tradução de um código HDL em RTL em sua equivalente *netlist* de células digitais no nível de portas lógicas. Assim, o sistema é descrito como uma rede de portas e *flip-flops* e seu comportamento é especificado por equações lógicas.

Os aplicativos de desenvolvimento de projetos que realizam esta tarefa de conversão de forma automática são conhecidos como ferramentas de síntese lógica. O software utilizado para a síntese lógica da FPU foi o Quartus II 9.1 Web Edition da Altera Corporation. A síntese foi realizada para o dispositivo FPGA Cyclone II EP2C35F672C6.

Dessa maneira, o software Quartus II sintetiza a descrição do circuito, que foi realizada em SystemVerilog, para uma *netlist* que define os elementos lógicos (LE's) necessários e as conexões entre os elementos.

Para a síntese da FPU, a ferramenta realiza a compilação do projeto, que abrange não só o processo de síntese, como também a análise de erros relacionados ao projeto como curtos circuitos e sinais duplicados, além também de otimizações do próprio projeto (TUTORIAL, 2010).

No fluxo da compilação, cuja interface está mostrada na FIGURA 49, estão as etapas de *Analisys* e *Synthesys*, *Fitter*, *Assembler* e *Classic Timing Analisys*. Estas etapas são responsáveis pela síntese do projeto, posição e roteamento dos LE's para o respectivo FPGA, geração de arquivos para programação do dispositivo e análises de temporização.

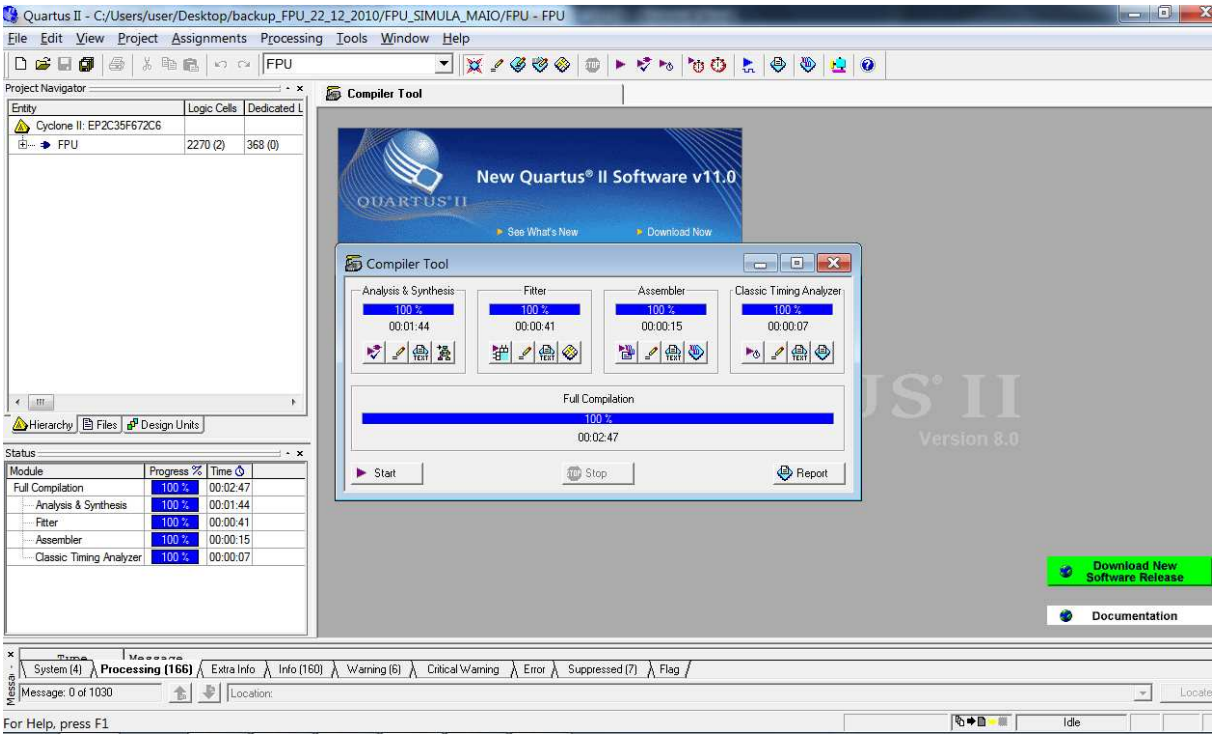


FIGURA 49 - Interface para o processo de compilação no Quartus II.

Após a conclusão do processo de compilação, os resultados de síntese lógica são mostrados na própria interface para serem analisados. Na FIGURA 50 está o sumário com os resultados de síntese lógica para a entidade *Top-Level* FPU no dispositivo Cyclone II EP2C35F672C6.

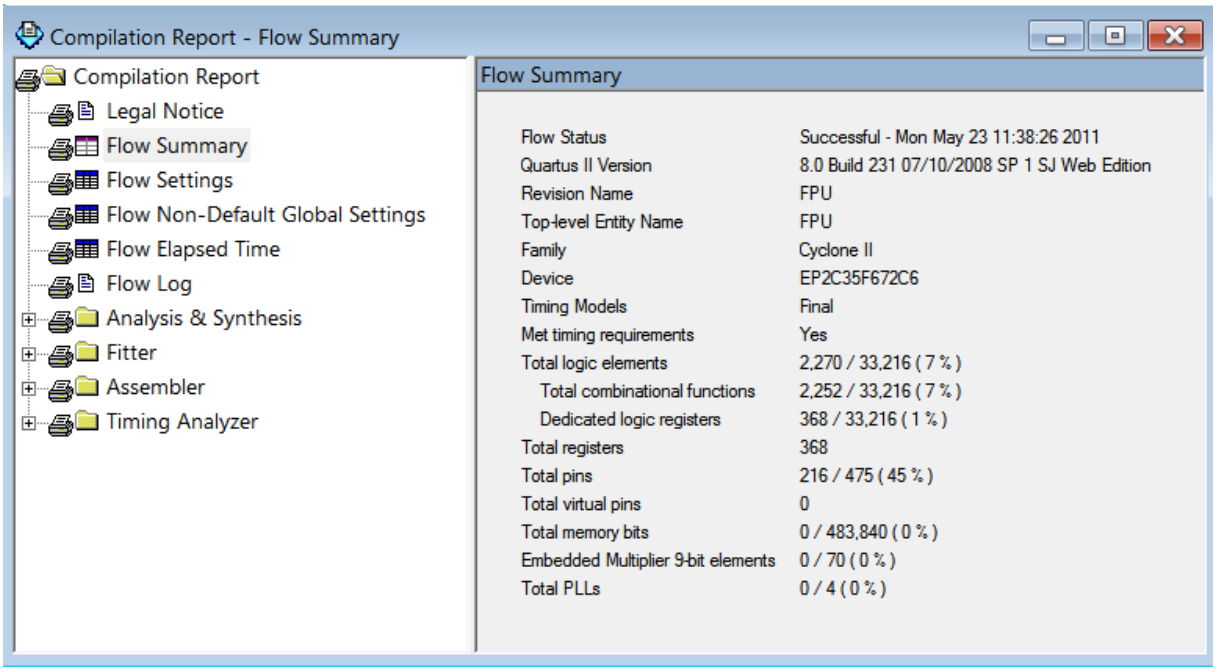


FIGURA 50 - Sumários de resultados de síntese lógica do Quartus II.

Observa-se na FIGURA 50 que a FPU ocupou 2270 elementos lógicos, não ultrapassando 7% da capacidade total do FPGA. De forma a verificar a importância deste resultado, será visto na seção 4.3 uma comparação entre outros projetos de FPU's.

Além da síntese lógica, o Quartus II também dispõe de um visualizador gráfico do *netlist* sintetizado, onde se dá destaque ao *State Machine Viewer* e *RTL Viewer*. O primeiro representa os estados e as transições da máquina de estado e o segundo uma visão a nível RTL (*Register Transfer Level*) que representa todas as ligações em um nível mais baixo de abstração. Para a exemplificação destes dois visualizadores, está mostrada na FIGURA 51 a máquina de estados gerada do controle interno do bloco *adder_sub* e uma visão do RTL do mesmo bloco.

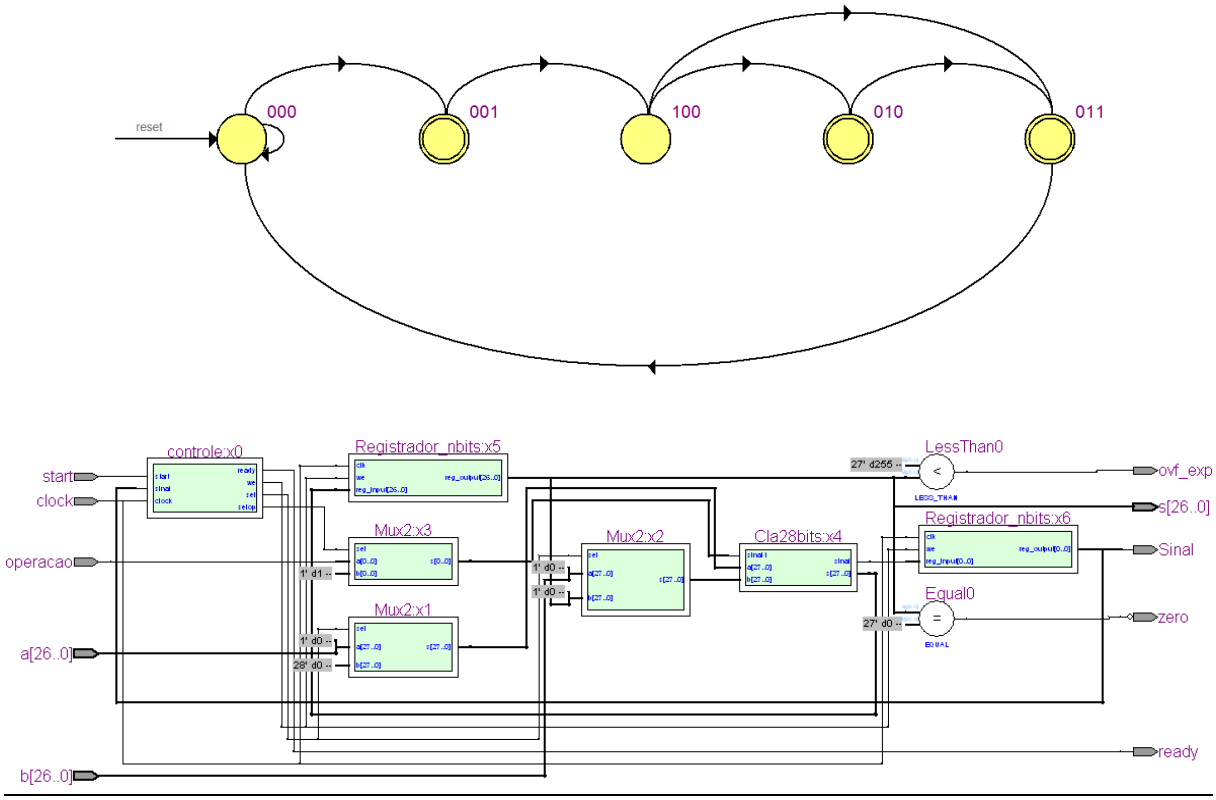


FIGURA 51 - Visualização gráfica da máquina de estados e do RTL do *adder_sub*.

4.2. Consumo de energia e temporização

Além da síntese, o Quartus II possui ferramentas de estimativa de energia consumida e frequência de operação, denominadas *PowerPlay Power Analyzer Tool* e *Classic Timing Analyzer tool*, respectivamente.

4.2.1. Classic Timing Analyzer tool

No analisador de temporização *Timing Analyser tool*, cuja interface está mostrada na FIGURA 52, é estimado os tempos de percurso entre os sinais de entrada e saída do circuito, de forma a obter a freqüência máxima de operação considerando as restrições temporais dos sinais.

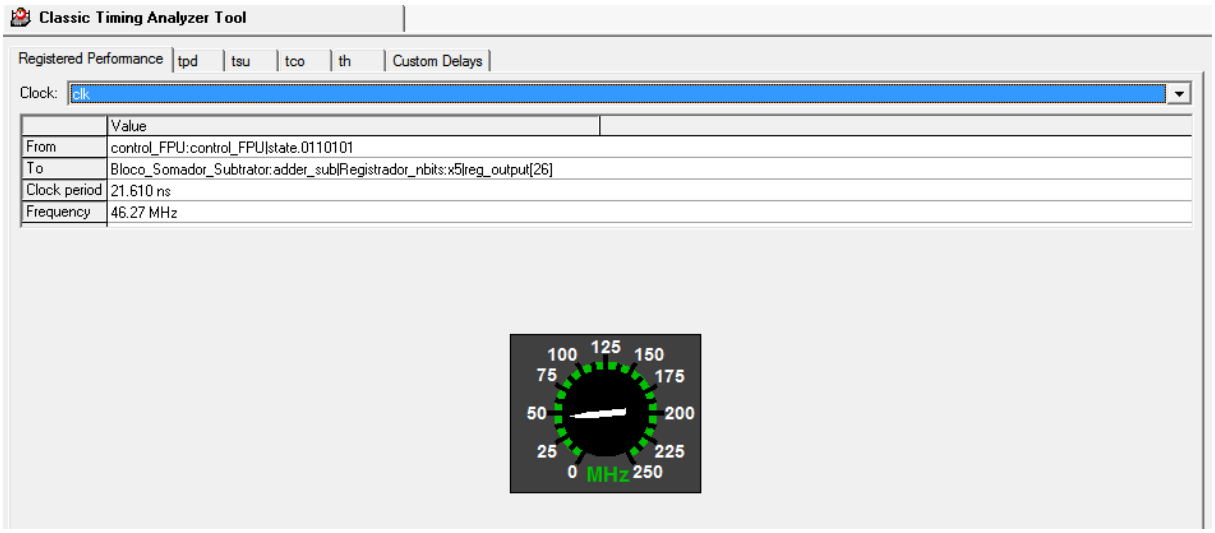


FIGURA 52 - Interface do analisador de temporização.

Dessa forma, observa-se na FIGURA 52 um período de *clock* de 21.610 *n*s e uma frequência de operação de 46.27 M*Hz* para o circuito sintetizado. Esta frequência é determinada pelo caminho crítico da FPU, que ocorre desde o carregamento dos dados nos registradores de entrada até a operação realizada no somador e a gravação do resultado em um registrador intermediário. Como o somador é amplamente utilizado nesta arquitetura, o desempenho global estará associado a este bloco.

4.2.2. PowerPlay Power Analyzer

Na ferramenta de estimativa de potência *Power Analyzer* é possível obter uma estimativa bastante precisa do consumo de energia do circuito sintetizado.

Dentre os fatores que interferem na dissipação de potência em FPGA's citados em (HOLANDA, 2007), o número de elementos lógicos e arquiteturais tem uma forte influência sobre a dissipação de potência. Logo, quanto maior o número de elementos, maior será a tendência de dissipar energia.

Ainda de acordo com (HOLANDA, 2007), a dissipação de potência é constituída por três componentes: potência estática, potência dinâmica e potência de E/S.

A potência estática é obtida quando o dispositivo não está em chaveamento, ou seja, ele se encontra em *standby*. Logo, a energia consumida é resultado de correntes dispersivas. A potência dinâmica é resultado da constante mudança de nível lógico de um sinal, ou seja, quando um sinal passa de “1” lógico para “0” lógico ou vice-versa, tem-se uma potência dissipada nesta mudança. Por último, tem-se a potência de E/S, que está relacionado com os pinos de entrada e saída e qualquer rede externa de terminação.

Os resultados de estimativa de potência estão mostrados na FIGURA 53.

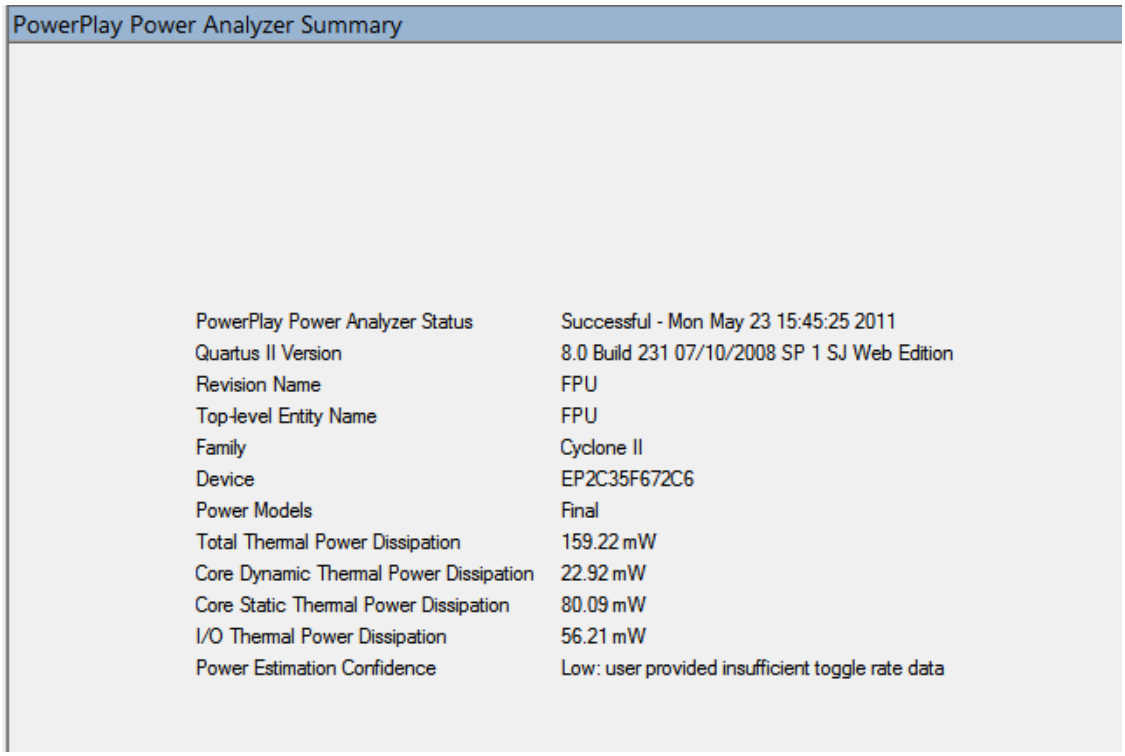


FIGURA 53 – Sumário de resultados da ferramenta *Power Play Power Analyzer*.

Observa-se nesta figura, que o consumo total foi de 159.22 *m*W, sendo 22.92 *m*W para potência dinâmica e 80.09 *m*W para a potência estática. A potência dinâmica varia com a operação a ser realizada, sendo que neste caso apenas uma soma entre dois operandos foi realizada para a obtenção deste parâmetro. Já para o consumo de E/S foi obtido um valor de 56.21 mW, que está relacionado com o número de E/S ativados no FPGA, que também tem relação com a operação escolhida.

4.3. Comparações entre FPU’s

De forma a validar os resultados obtidos da síntese lógica da FPU quanto aos requisitos de área ocupada em FPGA e frequência de operação, foram realizadas comparações com projetos de FPU’s realizados por (ULSSEMANN, 2000) e (AL-ERYANE, 2006), ambas dispo-

níveis como *open-source* no repositório *opencores* e descritas no padrão IEEE754 precisão simples.

Os resultados de síntese lógica e temporização estão mostrados na TABELA 12, sendo que ambas as arquiteturas foram recompiladas para o dispositivo Cyclone II EP2C35F672C. Na FPU de (Al- Eryane, 2006), a operação de raiz quadrada foi removida para a justa comparação.

TABELA 12 - Comparação de desempenho entre FPUs.

FPU	Número de LE's	Registradores dedicados	Total de funções combinacionais	Funções embarcadas	Frequência de operação (MHz)
FPU (Este trabalho)	2272	368	2252	0	46.27
(Al-Eryane, 2006)	3213	1212	3010	4	96,09
(Usselmann, 2000)	6673	610	6611	7	4.56

Pela TABELA 12, nota-se que a FPU projetada neste trabalho apresentou melhores resultados em área ocupada em FPGA, tendo uma redução de 30% em relação à FPU de (AL-Eryane, 2006), e 66% de redução em relação à FPU de (Usselmann, 2000). Porém, em frequência de operação, a FPU de (Al-Eryane, 2006) mostrou-se melhor devido à utilização de *pipeline* para acelerar o hardware, ao custo de maior área ocupada. Por fim, a FPU de (Usselmann, 2000) apresentou a maior área ocupada e menor freqüência operação, enquanto que, utilizou mais funções embarcadas, ou seja, funções prontas e otimizadas para determinada aplicação em FPGA.

De maneira a ter uma métrica para avaliar o desempenho de cada topologia, nas Tabelas 13, 14 e 15 é mostrado o número de ciclos de clock, freqüência de operação e a avaliação de cada operação para cada projeto. Define-se como ciclo de clock um intervalo básico de tempo nos quais são executadas as operações elementares, como por exemplo, troca de dados entre registradores e operações aritméticas (GÜNTZEL, 2011). A avaliação de desempenho, também denominada de *throughput*, é obtida através da multiplicação do número de ciclos pelo período de clock e tem como objetivo determinar o tempo gasto para uma operação.

TABELA 13 - Medidas de desempenho para a soma ou subtração.

	Ciclos de clock	Frequência de clock (M*Hz*)	Ciclos x (1/freq) (*us*)
FPU (Este trabalho)	22	46,27	0,47
FPU (Ulssemann, 2000)	3	4,56	0,65
FPU (Al-Eryane, 2006)	7	96,09	0,07

TABELA 14 - Medidas de desempenho para a multiplicação.

	Ciclos de clock	Frequência de clock (M*Hz*)	Ciclos x (1/freq) (*us*)
FPU (Este trabalho)	191	46,27	4,12
FPU (Ulssemann, 2000)	3	4,56	0,65
FPU (Al-Eryane, 2006)	12	96,09	0,12

TABELA 15 - Medidas de desempenho para a divisão.

	Ciclos de clock	Frequência de clock (M*Hz*)	Ciclos x (1/freq) (*us*)
FPU (Este trabalho)	309	46,27	6,67
FPU (Ulssemann, 2000)	3	4,56	0,65
FPU (Al-Eryane, 2006)	34	96,09	0,35

O número de ciclos de clock das FPU's utilizadas na comparação foi disponibilizado pelos autores em *opencores*, enquanto que a frequência de operação foi obtida através da síntese realizada neste trabalho.

Nota-se que em todas as operações, a FPU proposta por (Al-Eryane, 2006) apresentou o melhor desempenho, o que justificou a utilização de *pipeline* pelo autor. A FPU de (Ulsselman, 2000) apesar de utilizar um menor número de ciclos na operação de soma, apresentou um *throughput* inferior a da FPU proposta neste trabalho, mesmo utilizando 22 ciclos de clock.

Vale destacar que, apesar da FPU de (Al-Eryane, 2006) apresentar um melhor *throughput*, ela ocupa 30% a mais de área em FPGA do que a FPU proposta neste trabalho, ou seja, o conceito de *tradeoff* sugere que, ao melhorarmos um critério em projeto, outros critérios poderão ser afetados (VAHID, 2008).

4.4. Simulação Temporal

De maneira a testar a funcionalidade da FPU, serão apresentados resultados de simulação temporal para as quatro operações. Esta simulação considera não só a funcionalidade, mas também atrasos de temporização. Para a simulação, dois números foram escolhidos aleatoriamente para serem somados, subtraídos, multiplicados ou divididos. O modo de arredondamento utilizado é para o mais próximo, sendo o modo *default* do padrão IEEE754.

Na TABELA 16 estão representados os operandos X e Y utilizados para este teste, no formato decimal e no formato binário para ponto flutuante padrão IEEE754 32 bits.

TABELA 16 - Formatos dos números utilizados para a simulação temporal.

	X	**Y**
Decimal	0,3401	2,4498
Ponto Flutuante	00111110101011100010000110010110	01000000000111001100100110000110

A primeira simulação, mostrada na FIGURA 54, representa a operação de soma para os operandos X e Y.

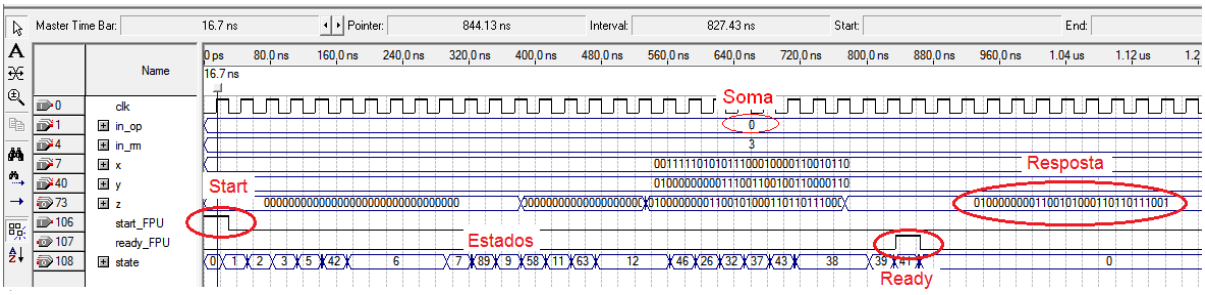


FIGURA 54 - Resultados de simulação para a operação de soma.

Nesta simulação a variável de entrada *in_op* está em zero, indicando a operação de soma e o modo de arredondamento *in_rm* em 3, indicando o arredondamento para o mais próximo. O sinal de *start* representa o início da simulação e a resposta é obtida após um sinal de *ready* fornecida pela variável ready_FPU. Também, podem-se observar os estados do controle da FPU representados pela variável *state*, que indica a sequência de estados para a operação de soma.

A TABELA 17 apresenta a comparação da resposta obtida via simulação com a resposta esperada.

TABELA 17 - Comparação entre simulação e valor esperado para a soma.

	Decimal	Binário
Resposta da simulação	2,7899	01000000001100101000110110111001
Resposta esperada	2,7899	01000000001100101000110110111001

Para a operação de subtração, a simulação está apresentada na FIGURA 55 onde nota-se a variável in_op em “1” indicando a operação escolhida.

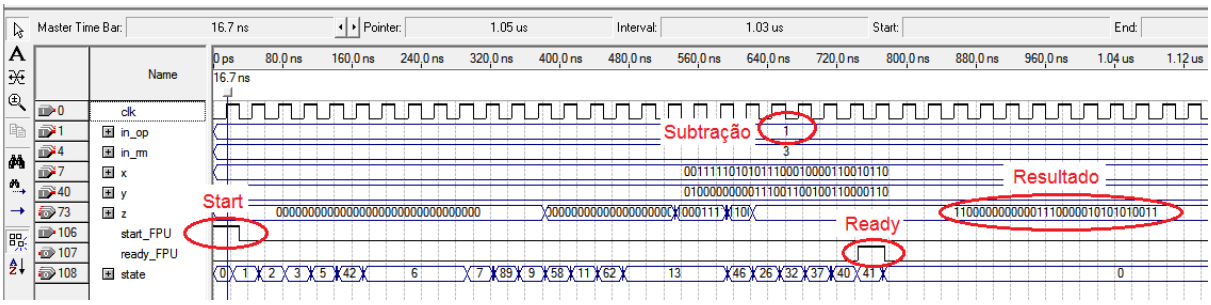


FIGURA 55 - Resultados de simulação para a operação de subtração.

A TABELA 18 apresenta a comparação da resposta obtida via simulação com a resposta esperada.

TABELA 18 - Comparação entre simulação e valor esperado para a subtração.

	Decimal	Binário
Resposta da simulação	-2,1096	11000000000001110000010101010011
Resposta esperada	-2,1096	11000000000001110000010101010011

Para a operação de divisão, a imagem da simulação foi dividida em duas partes de forma a visualizar a passagem do controle da FPU para o controle do divisor, previsto no Estágio V da seção 3.4.2.6 entre os estados 23, 24 e 25. A FIGURA 56 apresenta a primeira parte da simulação da divisão.

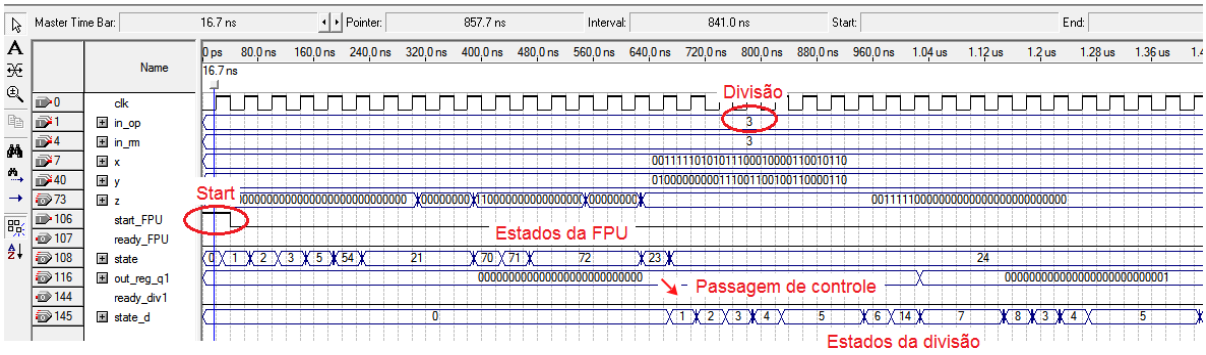


FIGURA 56 – Primeira parte da simulação da divisão.

Observa-se que o controle da divisão das mantissas é ativado, passando do **Estado 0** para o **Estado 1** na variável *state_d*. O controle da FPU fica em espera no **Estado 24** até o término da operação sobre as mantissas.

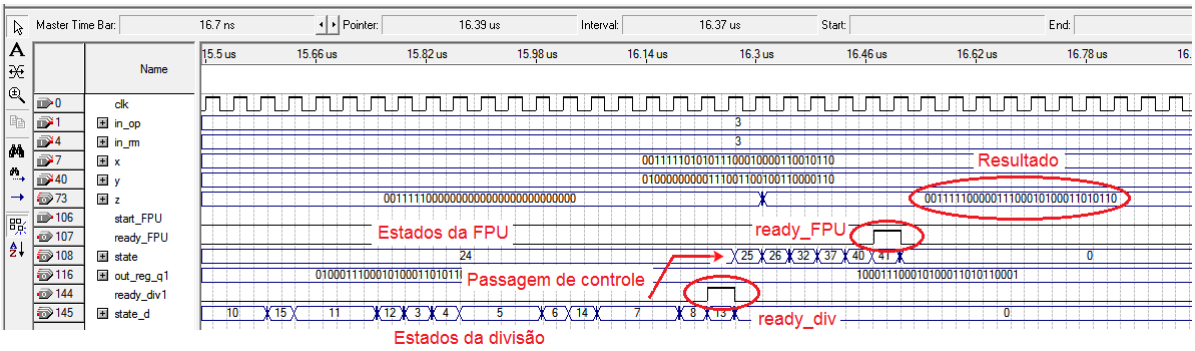


FIGURA 57 - Segunda parte da simulação da divisão.

Já na FIGURA 57, nota-se que o controle da FPU dá sequência aos estados após receber um sinal de pronto (*ready_div*) do controle da divisão. Na TABELA 19 é apresentada a comparação da resposta obtida via simulação com a resposta esperada.

TABELA 19 - Comparação entre simulação e valor esperado na divisão.

	Decimal	**Binário**
Resposta da simulação	0,138827666640282	00111110000011100010100011010110
Resposta Esperada	0,13882765173912	00111110000011100010100011010111

Nota-se na TABELA 19 que houve uma diferença no último bit da resposta da simulação, ocasionando um erro nos últimos dígitos da resposta em decimal. Este erro pode ter ocorrido em algum processo de deslocamento na operação de divisão, o que afeta na precisão do resultado e no arredondamento. Esta falha ainda precisa ser corrigida.

A simulação da multiplicação está mostrada na FIGURA 58, onde também se dá destaque à passagem do controle da multiplicação para o controle da FPU na variável *stade_m*.

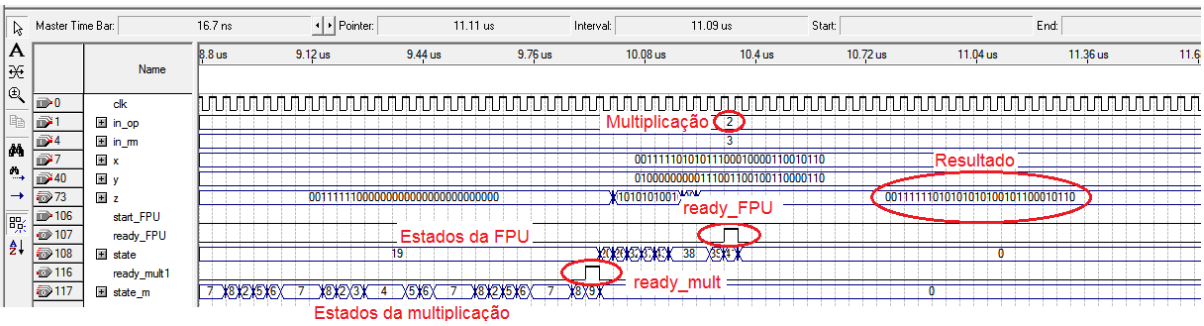


FIGURA 58 - Simulação da multiplicação com destaque na passagem de controle.

Na TABELA 20 é apresentada a comparação da resposta obtida via simulação com a resposta esperada.

TABELA 20 - Comparação entre simulação e valor esperado na multiplicação.

	Decimal	Binário
Resposta da simulação	0,833176970481873	00111111010101010100101100010110
Resposta Esperada	0,833176970481873	00111111010101010100101100010110

4.5. Testes em FPGA

De forma a automatizar a parte de testes da FPU, foi necessária a utilização de uma interface de comunicação para o envio e recebimento de dados automático com o FPGA. O FPGA Cyclone II está inserido no kit de desenvolvimento da Altera DE2 o qual possui diversos periféricos, tais como LEDs, chaves, botões, pinos de E/S, etc. Um diagrama da placa DE2 encontra-se mostrada na FIGURA 59.

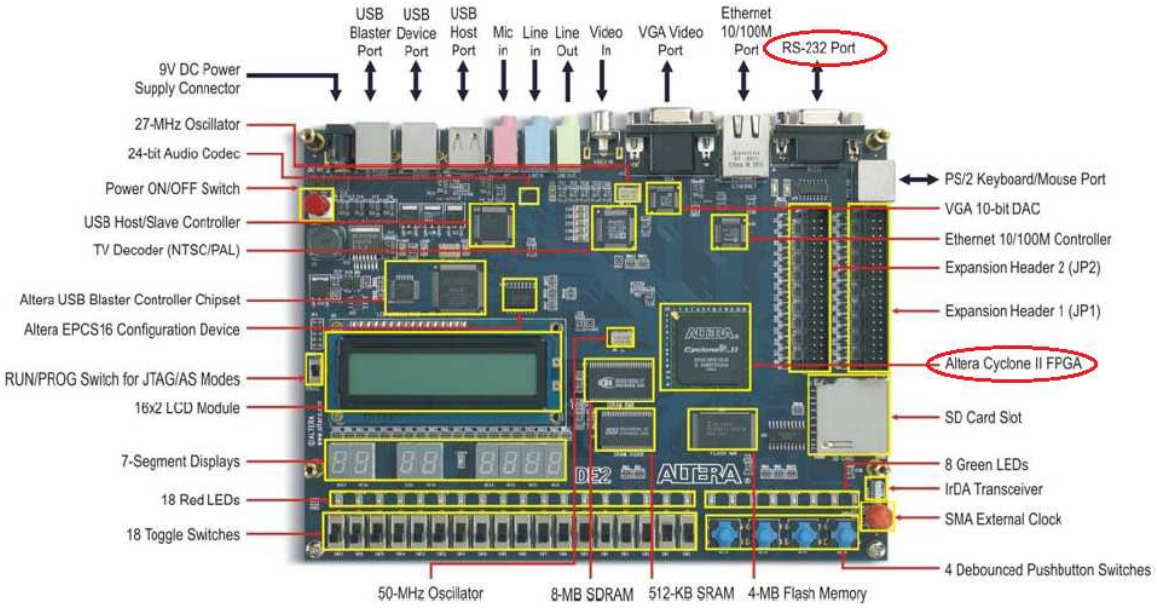


FIGURA 59 - Kit de desenvolvimento DE2 Altera.

Apesar do kit DE2 possuir uma pinagem suficiente para o envio e recebimento de dados, não seria viável ter um pino de FPGA para cada bit de entrada ou saída da FPU. Como o kit possui uma porta RS-232, os dados podem ser enviados e recebidos de forma serial utilizando uma interface com o protocolo de comunicação RS-232. Este padrão foi desenvolvido nos anos 60 pela *Eletronic Industries Association* (EIA), que especifica as tensões, temporizações

e funções dos sinais, de forma a prover a comunicação entre dois periféricos, que em nosso caso, será entre o computador e o FPGA.

A transmissão de dados é geralmente realizada em bytes. Para cada byte, normalmente é utilizado um bit para dar início ao envio de dados e um bit para sinalizar o termino de envio. Também podem ser utilizados bits de paridade para a verificação de erros na transmissão do dado.

A interface de comunicação proposta por Marques (MARQUES et al., 2009), foi implementada visando os requisitos de bits entrada e saída da FPU. Esta interface, cujo esquemático simplificado está na FIGURA 60, será responsável pela recepção e transmissão de dados utilizando o protocolo de comunicação RS-232.

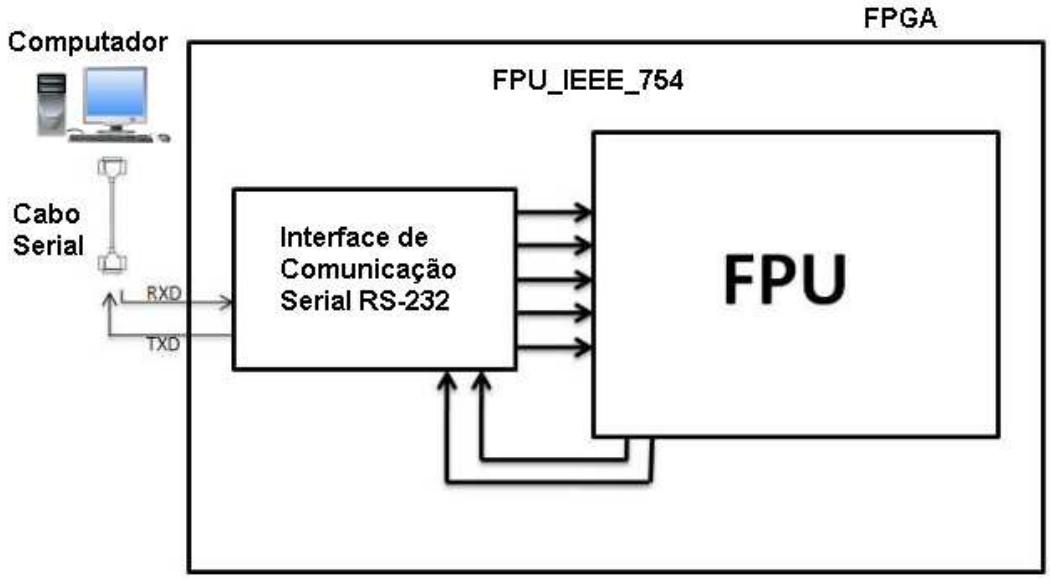


FIGURA 60 - Esquemático da interface de comunicação RS-232.

Nota-se que o projeto da FPU e da interface (comunicação serial RS-232) estão instanciadas no mesmo FPGA, sendo que agora a FPU é capaz de comunicar-se com qualquer dispositivo externo que utiliza o protocolo de comunicação. Como o foco desta seção é apenas o teste da FPU, vê-se a necessidade de um meio físico para envio de dados e uma interface gráfica para a visualização dos valores de entrada e saída. O meio físico utilizado é um cabo serial denominado DB9, que possui pinos de envio (TXD) e recebimento de dados (RXD). Este cabo estará conectado na saída serial do dispositivo, que em nosso caso é o computador, e na entrada serial do FPGA. Lembrando que a interface de comunicação é que será responsável pelo gerenciamento do protocolo dos dados no cabo serial, cabendo a ele somente a transferência de dados.

A interface apresentada no esquemático da FIGURA 60 tem como função básica o recebimento de dados enviados pelo computador, sendo que primeiramente estes dados são armazenados em uma memória interna na interface. Como os dados estão sendo enviados serialmente, a interface trata de organizá-los de forma paralela para serem enviados corretamente para as entradas da FPU. Após o término do processamento dos dados pela FPU, outra memória interna armazena as respostas que serão enviadas novamente ao computador de forma serial. Todos os bits de entrada totalizam 68 bits e de saída 32 bits.

Para o envio de dados pelo computador foi desenvolvido um software que possibilita a visualização dos dados a serem enviados e recebidos. Ele também faz uma comparação dos resultados recebidos pela FPU com o esperado para a operação. Na FIGURA 61 está ilustrada a interface gráfica para a comunicação entre computador e FPGA.

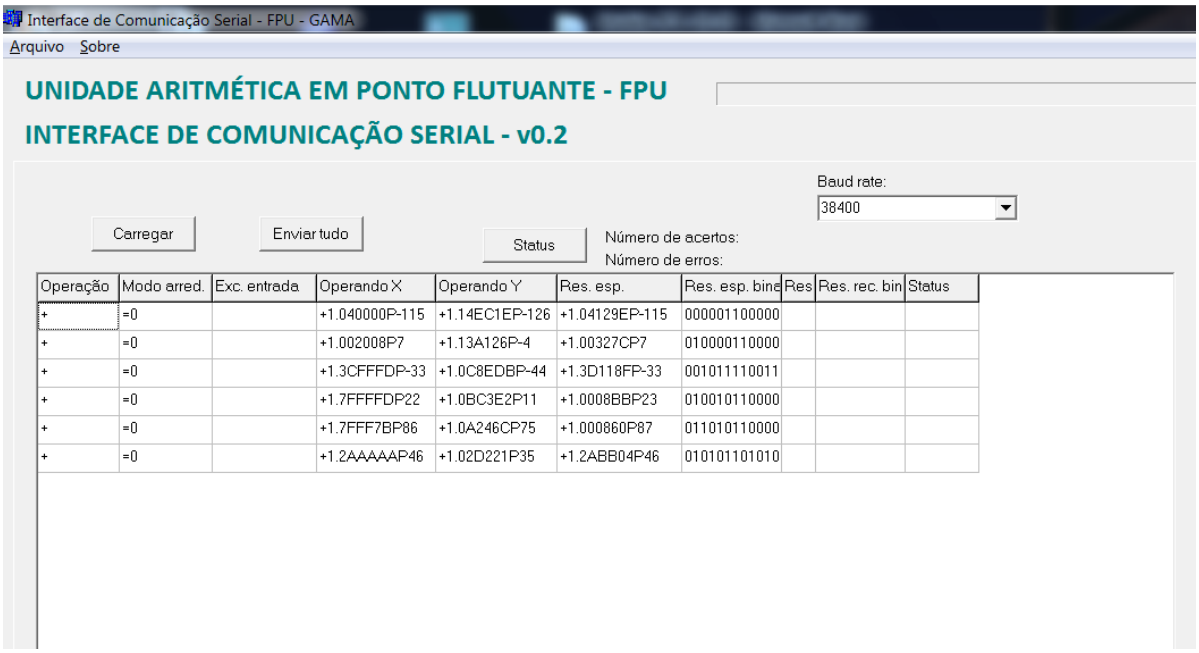


FIGURA 61 - Interface gráfica do software para a comunicação serial.

Este software também foi criado com o propósito da utilização de conjuntos de testes disponibilizados pela IBM, denominado de FPGen (AHARONI, 2003). FPgen é conjunto de testes (*test-suit*) para verificação de unidades em ponto flutuante no padrão IEE754, baseada na experiência acumulada da verificação de vários processadores da IBM. FPgen possui vários modelos de testes, cada um para uma parte específica da FPU.

Na FIGURA 62 apresenta um exemplo de um conjunto de testes enviados e recebidos pela interface de comunicação.

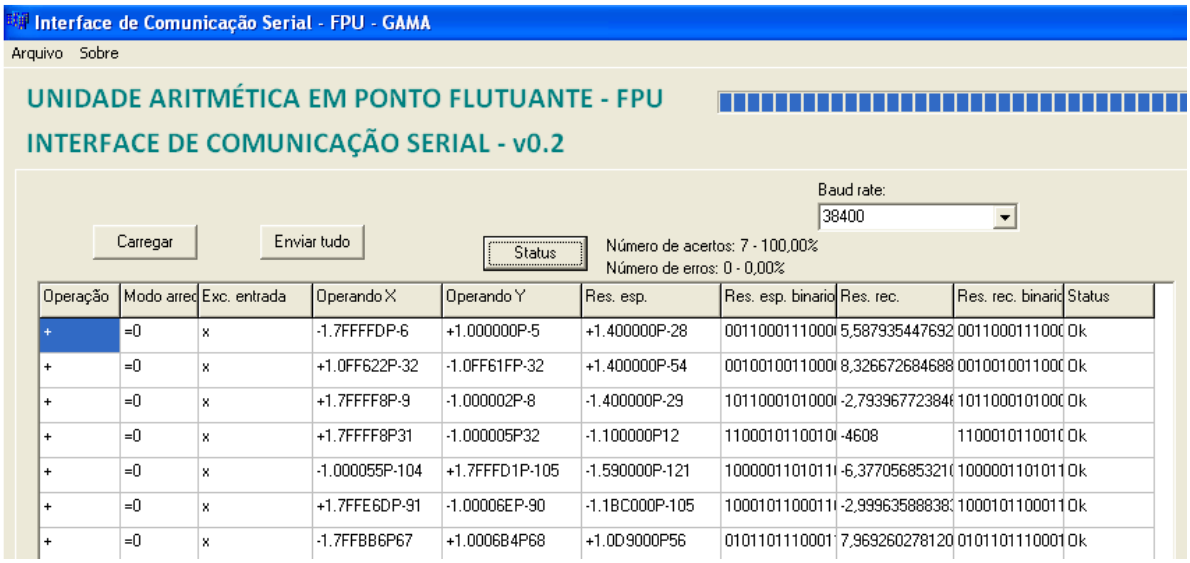


FIGURA 62 - Vetores de testes enviados e recebidos pelo software.

Este caso de teste, denominado de *Add-Cancellation*, considera a soma de duas magnitudes muito próximas, de maneira que o expoente do resultado seja muito menor do que os expoentes de entrada. Neste caso foram testados 52 valores, dos quais 7 estão mostrados na FIGURA 62. Destes 50 valores, apenas 2 erros foram reportados devido a um erro no último bit. Estes erros foram testados separadamente, sendo que na FIGURA 63 é destacado um dos erros encontrados.

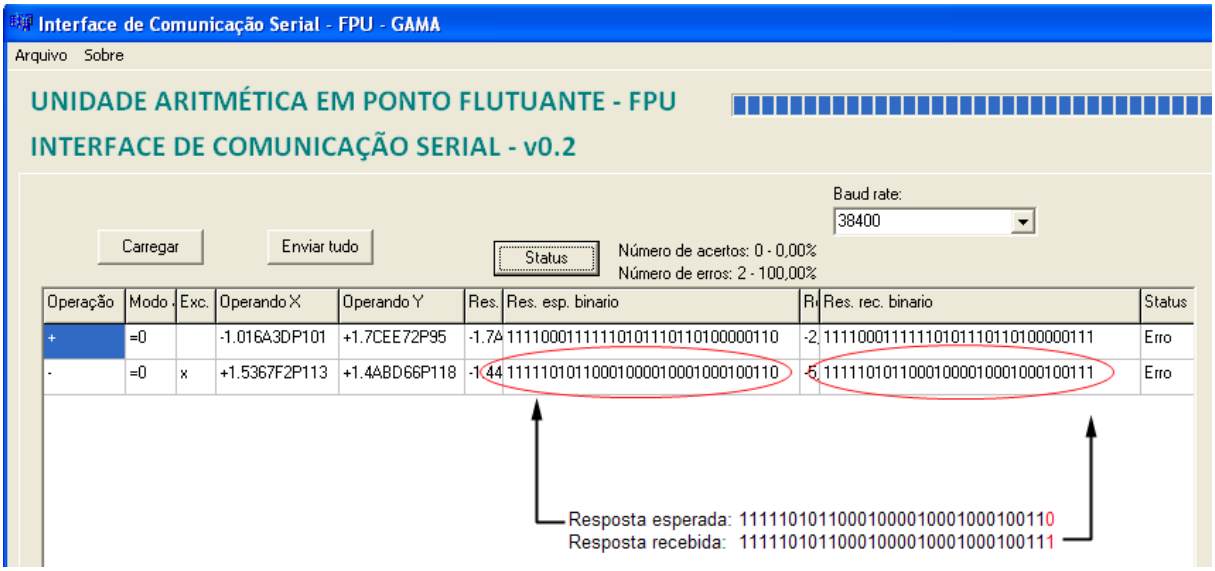


FIGURA 63 - Erros encontrados para o modelo de teste.

Dentre os modelos de testes disponibilizados pelo FPgen, abaixo são listados alguns casos que foram utilizados para a validação da operação de soma na FPU.

- ***Add-Cancellation and Subnormal-Result***: Este caso de teste considera operações de soma entre números normais, em que o resultado torna-se subnormal e operações entre subnormais, em que o resultado também é subnormal. Nestes testes, 1192 casos foram utilizados sendo que nenhum erro foi encontrado.
- ***Add-Shift***: Este caso de teste considera uma gama variada de expoentes de entrada onde é necessário um deslocamento da mantissa para alinhamento do expoente final. Dos 114 valores enviados, 12 erros foram encontrados.
- ***Add-Shift and Special Significands***: Este caso é o mesmo citado anteriormente, porém com a adição de valores especiais, como por exemplo, sequência de zeros e de uns. Neste modelo, foram validados mais de 10 mil casos e nenhum erro foi encontrado.

Cabe-se destacar o modelo de teste *Add-shift* onde foram encontrados 12 erros. Estes erros ocorreram para casos em que há uma grande diferença entre os expoentes de entrada, sendo necessário um grande deslocamento da mantissa para o alinhamento dos expoentes. Neste deslocamento, bits importantes são perdidos e que geram erros semelhantes ao da FIGURA 63.

CONSIDERAÇÕES FINAIS

Neste trabalho foi proposto uma nova alternativa de arquitetura em ponto flutuante no padrão IEEE754, precisão simples. A arquitetura foi descrita em SystemVerilog e sintetizada através do software Quartus II 9.1 Web Edition. Este software foi o responsável pela síntese, ou seja, pela transcrição da linguagem HDL em um nível RTL para ser testado em FPGA.

Utilizando-se a metodologia adotada pelo programa Brazil-IP, esta arquitetura passou pelas fases de concepção, de forma a ter uma visão sistêmica de sua funcionalidade, como também pela fase de definição da arquitetura, onde cada elemento utilizado foi detalhado quanto aos sinais de entrada e saída e interação no sistema.

Para o controle das unidades funcionais da arquitetura, utilizaram-se máquinas de estados, onde, em cada estado, sinais eram atribuídos de acordo com a entrada. O controle ficou dividido em três partes, sendo que foi necessário um controle para o gerenciamento de toda a arquitetura e controles para a divisão e multiplicação das mantissas. Nestes dois últimos, algoritmos sequenciais foram utilizados visando uma economia de hardware, já que ambas as operações utilizaram o mesmo hardware, diferindo apenas no controle. Por estas operações utilizarem somas e subtrações para a obtenção da resposta, o somador fica como o elemento central da arquitetura. Por ser elemento central, o desempenho global do sistema estará associado ao somador. Por este motivo, foi utilizado um CLA de 28 bits, que apresenta uma aceleração do cálculo da propagação dos *carries* e mostra-se mais eficiente em requisitos de área entre somadores de 28 bits comparados (SCHLOSSER, 2009).

No controle principal da FPU, foram adotados estágios para representar cada parte de um fluxo em ponto flutuante para as operações aritméticas. Alguns estágios são fundamentais para determinar a precisão da resposta final, como o estágio de alinhamento dos expoentes. Aqui, bits importantes podem ser perdidos em processos de deslocamento da mantissa, que afeta etapas posteriores como o arredondamento. Uma das maneiras de se reduzir os erros de arredondamento é com a utilização de bits de guarda, que tem por finalidade garantir a precisão inserindo bits extras à mantissa. Porém, com a utilização de dois bits de guarda, não foi o suficiente para de garantir a precisão, por exemplo, da operação de divisão em sua simulação temporal.

Para a validação da FPU quanto aos requisitos de área em FPGA e métricas de desempenho, duas topologias de arquiteturas foram colocadas para a comparação através resultados de

síntese lógica. A FPU proposta neste trabalho apresentou uma área em FPGA 30% menor do que a FPU de (AL-Eryane, 2006), que apresentou o melhor *throughput.* Logo, cada arquitetura comparada pode ser utilizada dependendo da aplicação desejada, como menor área, ou maior desempenho.

Com a utilização de *test-suits* da IBM, a arquitetura foi validada em partes no FPGA, sendo que ainda restam ajustes a serem realizados relacionados a precisão e arredondamento, ficando como trabalho futuro a verificação e teste completo da arquitetura da FPU.

REFERÊNCIAS BIBLIOGRÁFICAS

AHARONI M. ; ASAF S. ; FOURNIER L. ; KOIFMAN A.; NAGEL R. **"FPgen - A Test Generation Framework for Datapath Floating-Point Verification"**. In Proc. IEEE International High Level Design Validation and Test Workshop 2003 (HLDVT03), 2003. Disponível em: https://www.research.ibm.com/haifa/projects/verification/fpgen/doc.html. Acesso em: 4 jun. 2011.

AL-ERYNE, J. , **FPU**. Disponível em: <http://opencores.org/project,fpu100>. Acesso em: 27 mai. 2011.

CARTER, Nicholas, **Arquitetura de computadores**, Bookman, 2002.

CASTRO, I. ; NEVES, R. C.; MARQUES J.; GIRARDI. A., **Arquitetura de um divisor seqüencial de 27 bits para divisão de mantissas em uma unidade aritmética em ponto flutuante**. In: XXIII Congresso Regional de Iniciação Cientifica e Tecnologia em Engenharia (XXII CRICTE), UDESC, Joinville-SC, 2009.

GOLDBERG, David. **What Every Computer Scientist Should Know About Floating-Point Arithmetic**, ACM Computing Surveys. - 1991. - 1 : Vol. 23. Disponível em: <http://perso.ens-lyon.fr/jean-michel.muller/goldberg.pdf>. Acesso em: 27 mai. 2011.

GÜNTZEL, **Arquitetura e Organização de Computadores – Avaliação de desempenho.** Disponível em: <www.inf.ufsc.br/~guntzel/ine641400/AOC2_aula7.pdf>. Acesso em: 24 mai. 2011.

HOLANDA, José Arnoldo Mascagni de, **Projeto de um estimador de potência para o processador NIOS II da Altera**, São Carlos/SP, 2007.

IEEE Computer Society (1985), **IEEE Standard for Binary Floating-Point Arithmetic**, IEEE Std 754,1985.

IEEE Computer Society (1987), **IEEE Standard for Radix-Independent Floating-Point Arithmetic**, Std 854-1985, New York, 1987.

LEITE, Risério Dourado, **Implementação de um Módulo Controlador de Vídeo na forma "Intellectual Property"**, Belo Horizonte, 2006.

LIMA, M. S. M. ; SANTOS, Francielle Silva dos ; SILVA, J. F. B. ; BARROS, E. N. S. . **ip-PROCESS: A Development Process for Soft IP-core with Prototyping in FPGA**. In: Forum on Specification and Design Languages (FDL) 2005,. p. 487-498.

MALAVAZI, Mazílio Coronel, **Aritmética intervalar: histórico, topologia e algoritmos**, Cáceres MT, 2004. Disponível em: <www2.unemat.br/rhycardo/projects/Mazilio.doc>. Acesso em: 26 mai. 2011.

MARQUES, J. P. ; NEVES, R. C. ; FIGUEIRO, I. C. ; SEVERO, L. C. ; GIRARDI, A. G., **Implementação de uma Arquitetura de Comunicação Serial RS232 com Aplicação em uma Unidade Aritmética em Ponto Flutuante de 32 bits**, In: XXIII Congresso Regional de Iniciação Cientifica e Tecnologia em Engenharia (XXII CRICTE), UDESC, Joinville-SC, 2009.

MURDOCA, Miles; VINCENT, Heuring. **Introdução a Arquitetura de Computadores**, 4. Ed. Rio de Janeiro: Campus, 2001.

NEVES, R. C. ; FIGUEIRO, I. C. ; GHISSONI, S. ; GIRARDI A., **Comparação de Multiplicadores séries e Paralelos de 27 bits com aplicação em FPGA**. In: XXIII Congresso Re-

gional de Iniciação Cientifica e Tecnologia em Engenharia (XXII CRICTE), UDESC, Joinville-SC, 2009.

NEVES, R. C. ; FIGUEIRO, I. C. ; MARQUES, J. P. ; SCHLOSSER, E. R. ; PREDIGER, D. L. ; GHISSONI, S. ; GIRARDI A., **Implementação de uma Unidade em Ponto Flutuante para operações Aritméticas em FPGA**. In: Iberchip 2010, Foz do Iguaçu, 2010. Anais do Iberchip 2010.

NURMI, J. Processor design: system-on-chip computing for ASICs and FPGAs, 1. Ed. Springer, 2007.

PATTERSON, DAVID A.; HENNESY, Jonh L. **Computer Architecture: A Quantitative Approach**, 3th Ed., 2003.

SCHLOSSER, E. R. ; PREDIGER, D. L. ; GHISSONI, S. ; GIRARDI, A. G., **Comparação de Desempenho de Somadores Binários Inteiros de 28 Bits com Aplicação em uma Unidade Aritmética em Ponto Flutuante.** In**:** XXIII Congresso Regional de Iniciação Cientifica e Tecnologia em Engenharia (XXII CRICTE), UDESC, Joinville-SC, 2009.

SILVA, José Frank Viana da, **JFLOAT: Uma biblioteca de ponto flutuante para a linguagem JAVA com suporte a arredondamento direcionado**, Dissertação, Natal/RN, 2007. Disponível em: < http://www.dimap.ufrn.br/~bedregal/Tese-alunos/Frank.pdf>. Acesso em: 25 mai. 2011.

SILVA, Danniela Dias Cavalcante da. **Desenvolvimento de um ip core de pré processamento digital de sinais de voz para aplicação em sistemas embutidos** – Dissertação Mestrado – Campina Grande/PB, 2006. Disponível em: <http://docs.computacao.ufcg.edu.br/posgraduacao/dissertacoes/2006>. Acesso em: 06 jun. 2011.

STALLINGS, William, **Arquitetura e Organização de Computadores**, 5. ed, Pearson Education, 2005.

Stuart F. Oberman, Michael J. Flynn, Fellow, **"Division Algorithms and Implementations"**, IEEE TRANSACTIONS ON COMPUTERS, VOL. 46, NO. 8, AUGUST 1997.

SUTHERLAND S., DAVIDMAN S, FLAKE P, **SystemVerilog For Design – A Guide to Using SystemVerilog for Hardware Design and Modeling**. Kluwe Academic Plubishers, 2004.

TANENBAUM, A. S. **Organização Estruturada de Computadores**. 5. ed. São Paulo: Pearson, 2007.

TUTORIAL, **Altera Quartus II e kit de Desenvolvimento DE1**. Disponível em: <tp://ftp.dca.fee.unicamp.br/pub/docs/ea773/Tutorial_QuartusII_v2.0.pdf>. Acesso em: 27 mai. 2011.

ULSSEMANN, R., **Floating Point Unit**. Disponível em: <http://opencores.org/project,fpu>. Acesso em: 27 mai. 2011.

UYEMURA, Jonh P., **Sistemas digitais – Uma abordagem integrada,** Editora Thomson 2000.

VAHID, Frank – **Sistemas digitais – Projeto, Otimização e HDLs** – Bookman, 2008 – Porto Alegre.

VIANA, G. V. R. **Padrão IEEE-754 para Aritmética Binária de Ponto Flutuante**. Ciências e Tecnologia (UECE), Fortaleza-CE, v. 1, n. 1, p. 29-33, 1999. Disponível em <www.lia.ufc.br/~valdisio/download/ieee.pdf>. Acesso em: 12 mai. 2011.