ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

ENGENHARIA ELÉTRICA

PCS 3422 – Organização e Arquitetura de Computadores 2

Unidade funcional compatível IEEE 754 (Grupo I)

Felipe Coelho de Abreu Pinna Nº USP 9837177
Gabriel Takeshi Medeiros Yamasaki Nº USP 9837538
Ian Alkmin Santos La Rosa Nº USP 9837271
Victor Passos de Pinho Nº USP 9426731

Bruno de Carvalho Albertini

Introdução

Este trabalho tem como objetivo produzir uma unidade funcional para cálculo com números em formato de ponto flutuante. O padrão mais utilizado para aritmética de ponto flutuante é o IEEE 754, o qual este trabalho será compatível, seguindo os padrões de representação numérica definida nesse padrão. Existem variações quanto a quais operações cada unidade funcional implementa. Como estamos criando uma unidade para um processador específico LEGv8, apenas será desenvolvido o hardware necessário para realizar as operações de ponto flutuante presente nesse conjunto de instruções.

A implementação dessa unidade funcional foi realizada por meio da linguagem de descrição de hardware VHDL, utilizando o programa Quartus da empresa Intel e Active-HDL da empresa Aldec para criação e simulação dos componentes desenvolvidos.

Representação Numérica

Um fator fundamental da computação é a representação de números reais em uma sequência finita de bits, para que o computador possa operá-lo. A representação de ponto flutuante é uma maneira de representar números reais que é amplamente utilizada atualmente, pois permite a representação de uma grande quantidade de valores numéricos. Como possuímos apenas um número finito de códigos representáveis por uma cadeia de bits, é de se esperar que não seja possível representar qualquer número real, mas sim apenas uma faixa de valores. Também, existe um erro de quantização na representação que está associado com a quantidade de bits utilizados e com o valor sendo representado (podendo ser nulo).

O padrão IEEE 754 define três tipos de precisão numéricas: *single precision* (*float*), *double precision* (*double*) e *double extended precision*. O processador que será desenvolvido apenas possui instruções para precisões simples e dupla, que utilizam 32 e 64 bits respectivamente, assim apenas esses formatos serão implementados.

Os números desse padrão são compostos por três campos, que são usados para representar o valor original da seguinte forma:

$$(-1)^s \times M \times 2^E$$

Onde s é o bit de sinal, que determina se o número é positivo ou negativo, M é a mantissa, um valor binário que representa a parte fracionária (cada bit da mantissa representa uma potência negativa de dois : 2^-1 , 2^-2, 2^-3 ....), e E é o expoente. Essa forma de representação é semelhante ao notação científica que usamos com números decimais, no sentido que deixamos apenas um valor inteiro antes da vírgula e multiplicamos por uma potência de dez, a diferença que aqui multiplicaremos por uma potência de 2.

A forma de guardar essas informações em uma palavra de 32 ou 64 bits é mostrado na Figura 1. Na representação de precisão simples o campo exp ocupa 8 bits e o campo e o campo fraq ocupa 23 bits, enquanto na representação de precisão dupla o campo exp ocupa 11 bits e o campo fraq ocupa 52 bits. Em ambos os casos o campo s apenas ocupa o bit mais significativo.

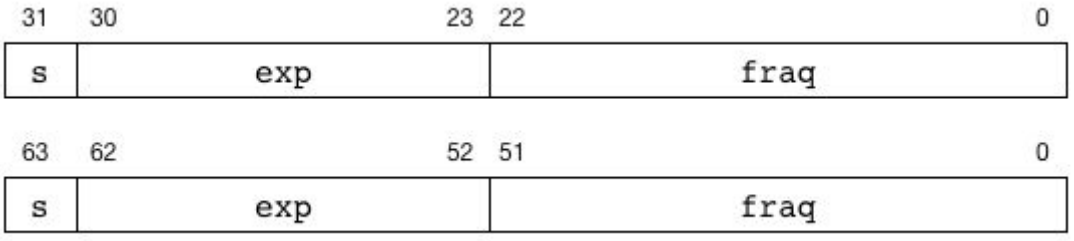

Figura 1: Codificação dos campos s, exp e fraq para representação *float* (acima) e *double* (abaixo).

A faixa de valores de valores válidos (normalizados) para a representação de precisão simples é de 2^{-126} a 2^{127}. Para precisão dupla a faixa de valores representáveis é de 2^{-1022} a 2^{1023}.

Será utiliza a representação normalizada, que é a mais comumente empregada. Nessa representação, o valor da mantissa está sempre no intervalo $[1.0,\ 2.0[$, assim ela sempre é da forma $M = 1 + fraq$. Dessa maneira, apenas precisamos armazenar a parte fracionária, a soma com "1" é sempre implícita.

O expoente E pode ser positivo ou negativo, mas não é representado em complemento de dois ou equivalente. Usa-se a representação por excesso da seguinte forma:

$$E = exp\ +\ bias$$

Onde $bias$ é o valor do excesso que corresponde a 127 para precisão simples e 1023 para precisão dupla.

A representação de números em ponto flutuante ainda possui especificidades quanto a representação de valores especiais como zero, infinito, menos infinito e NaN (*Not a Number*). A forma de representação desses valores é apresentada na Tabela 1.

Tabela 1: Representação de ponto flutuante de valores especiais.

zero	S = 1/0	exp = 0	M = 0
$+\infty$	S = 0	exp = 111...111	M = 0
$-\infty$	S = 1	exp = 111...111	M = 0
NaN	S = 0	exp = 111...111	M ≠ 0

Um fator que deverá ser implementado são as propriedades aritméticas com esses valores especiais. Em sua maioria as propriedades são as usuais da matemática conhecida com números reais. A lista de operações aritméticas com um valor a de ponto flutuante é a seguir:

- $a + \infty = \infty$
- $a - \infty = -\infty$
- $\infty - \infty = NaN$
- $NaN + a = NaN$
- $a / \pm\infty = 0$
- $\pm\infty \times \pm\infty = \pm\infty$
- $a / 0 = \pm\infty$, se $a \neq 0$
- $\pm 0 / \pm 0 = NaN$
- $\pm\infty / \pm\infty = NaN$

- $\pm \infty \times 0 = NaN$

Instruções

A unidade funcional de ponto flutuante projetada deve de ser capaz de realizar todas as operações de ponto flutuante no conjunto de instruções do processador LEGv8. As operações que serão implementadas estão apresentadas na Tabela 2.

Tabela 2: Operações de ponto flutuante do LEGv8.

Nome	Mnemônico	Formato	Opcode/Shamt	Operação
Floating-point ADD Single	FADDS	R	0F1/0A	S[Rd] = S[Rn] + S[Rm]
Floating-point ADD Double	FADDD	R	0F3/0A	D[Rd] = D[Rn] + D[Rm]
Floating-point CoMPare Single	FCMPS	R	0F1/08	FLAGS = (S[Rn] vs S[Rm])
Floating-point CoMPare Double	FCMPD	R	0F3/08	FLAGS = (D[Rn] vs D[Rm])
Floating-point DIVide Single	FDIVS	R	0F1/06	S[Rd] = S[Rn] / S[Rm]
Floating-point DIVide Double	FDIVD	R	0F3/06	D[Rd] = D[Rn] / D[Rm]
Floating-point MULtiply Single	FMULS	R	0F1/02	S[Rd] = S[Rn] * S[Rm]
Floating-point MULtiply Double	FMULD	R	0F3/02	D[Rd] = D[Rn] * D[Rm]
Floating-point SUBtract Single	FSUBS	R	0F1/0E	S[Rd] = S[Rn] - S[Rm]
Floating-point SUBtract Double	FSUBD	R	0F3/0E	D[Rd] = D[Rn] - D[Rm]

A unidade de ponto flutuante desenvolvida está dividida em componentes que realizam as operações de soma/subtração/comparação, multiplicação e divisão. Há duas versões de cada componente para lidar com operações de precisão simples e dupla. Cada uma dessas unidades possui o seu fluxo de dados e unidade de controle.

Operações com valores especiais

Na unidade projetada há um componente, posicionado perto de sua entrada, que realiza checagem pelas operações envolvendo os valores especiais mencionadas anteriormente. Este componente recebe os dois operandos, a operação, um sinal *go* e um *reset*. Esse componente é implementado por uma máquina de estados.

Durante uma operação que não envolva valores especiais, esse componente atua de forma transparente, repassando os operandos da entrada para saída e o resto da unidade de ponto flutuante pode realizar suas operações normalmente.

Caso detecte algum dos casos excepcionais na borda de subida do sinal *go*, quando este componente estiver em seu estado inicial, ocorre uma mudança estado, de forma que tenha em sua saída a resposta adequada para a operação exigida. Existe um estado específico para cada tipo de saída possível envolvendo valores especiais. Também, um alerta a unidade de ponto flutuante é enviado informando que saída deve ser a produzida por este componente, o sinal que possui essa funcionalidade de avisar o resto da unidade é nomeado *special*. Por exemplo, se observarmos na entrada a soma de infinito com menos infinito, realizamos uma transição de estado para um estado que coloque NaN na saída.

Operações

Quando uma operação não se encaixe em nenhum dos casos especiais, segue-se para que sejam realizadas as operações normalmente. Para isso, é necessário que ambos operandos estejam adequados para a realização da operação, lembrando que a representação dos números no padrão IEEE 754 é desta forma:

$$(-1)^s \times M \times 2^E$$

Para todas as operações, primeiramente são separadas as três secções dos operandos (sinal, expoente, mantissa), em seguida expande-se a mantissa com seu ‘1’ inteiro pressuposto e acomodam-se em vetores de tamanho suficiente para eventuais overflows serem captados. Esses passos iniciais estão implícitos em todas as operações que são comentadas a seguir.

Adição *(add_sub_64.vhd, add_sub.vhd)*

Para adequar os operandos à operação de adição é necessário que eles tenham o mesmo valor de expoente, para isto toma-se o maior expoente como base e então desloca-se a mantissa do operando de menor expoente para a direita até que os expoentes sejam iguais.

Estando os operandos adequados para a realização da operação somam-se os valores da mantissa, normalizando o resultado se necessário (desloca-se a mantissa até ter posicionado seu '1' pressuposto, expoente é corrigido).

Durante a execução se checa casos de overflow e a operação aborta.

Subtração/Compare *(add_sub_64.vhd, add_sub.vhd)*

A subtração não se diferencia da adição, dada que a adição pode operar com números de sinais quaisquer. O componente de operações especiais apenas inverte o sinal do segundo operando quando repassa operandos para uma operação normal de subtração.

A operação de compare equivale a de subtração, dado que nessa UF todas as operações atualizam os condition codes da CPU.

Multiplicação *(mult.vhd, mult_64.vhd)*

Na multiplicação não se é necessário se fazer algo para adequar os operandos, pois já estão adequados para esta operação.

Cada elemento que compõe o número é calculado de uma diferente forma.

O valor do sinal é calculado através de um XOR dos valores dos sinais dos fatores.

O expoente do produto é obtido somando os expoentes dos fatores e subtraindo o valor de excesso (bias).

Para o cálculo da mantissa multiplicam-se as mantissas dos fatores, sendo o resultado arredondado para se adequar ao número de bits permitidos na mantissa.

Com todos os elementos calculados é possível que seja necessário normalizar o resultado.

Durante a operação analisa-se o expoente para checar se houve overflow.

Divisão *(div_64.vhd, div.vhd, manDiv.vhd)*

Cada elemento que compõe o número é calculado de uma diferente forma.

O valor do sinal é calculado através de um XOR dos valores dos sinais do dividendo e divisor.

Para o cálculo da mantissa divide-se a mantissa do dividendo pela do divisor, para isso utiliza-se de um componente específico (manDiv.vhd), este componente realiza sucessivos deslocamentos do dividendo e subtrações do divisor no dividendo, aos poucos forma-se o quociente e também se calcula um valor de correção ao expoente final que deverá ser aplicado.

O expoente do quociente é obtido subtraindo o expoente do divisor do expoente do dividendo e somando o valor de excesso (bias), e após a divisão da mantissa é corrigido.

Após o cálculo de todos os elementos é possível que seja necessário normalizar o resultado.

Durante a operação analisa-se o expoente para checar se houve overflow.

Unidade Funcional - Diagrama de Blocos

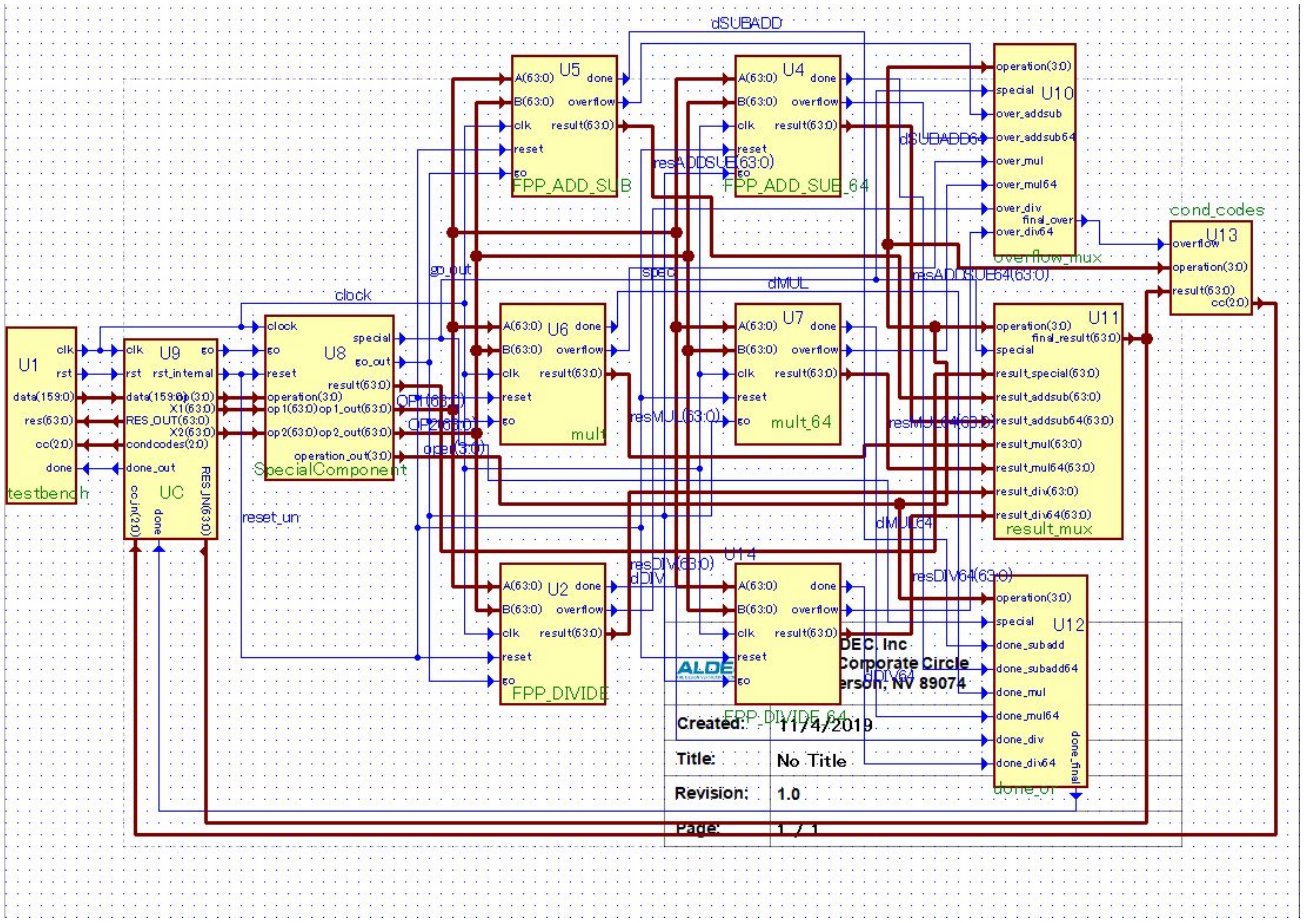

Figura 2: Visão geral da unidade funcional.

A unidade funcional divide-se em 3 grandes partes: primeiro o controle e tratamento de entrada (UC e SpecialComponent), em seguida os componentes de cálculo (FPP_ADD_SUB/64 , mult/_64, FPP_DIVIDE/_64) , por fim a controle de fim e tratamento de resultados (*overflow_mux, result_mux, done_or, cond_codes*). Também há um componente de teste (*testbench*) na extrema esquerda para simular a presença de uma CPU.

Controle e tratamento de entrada

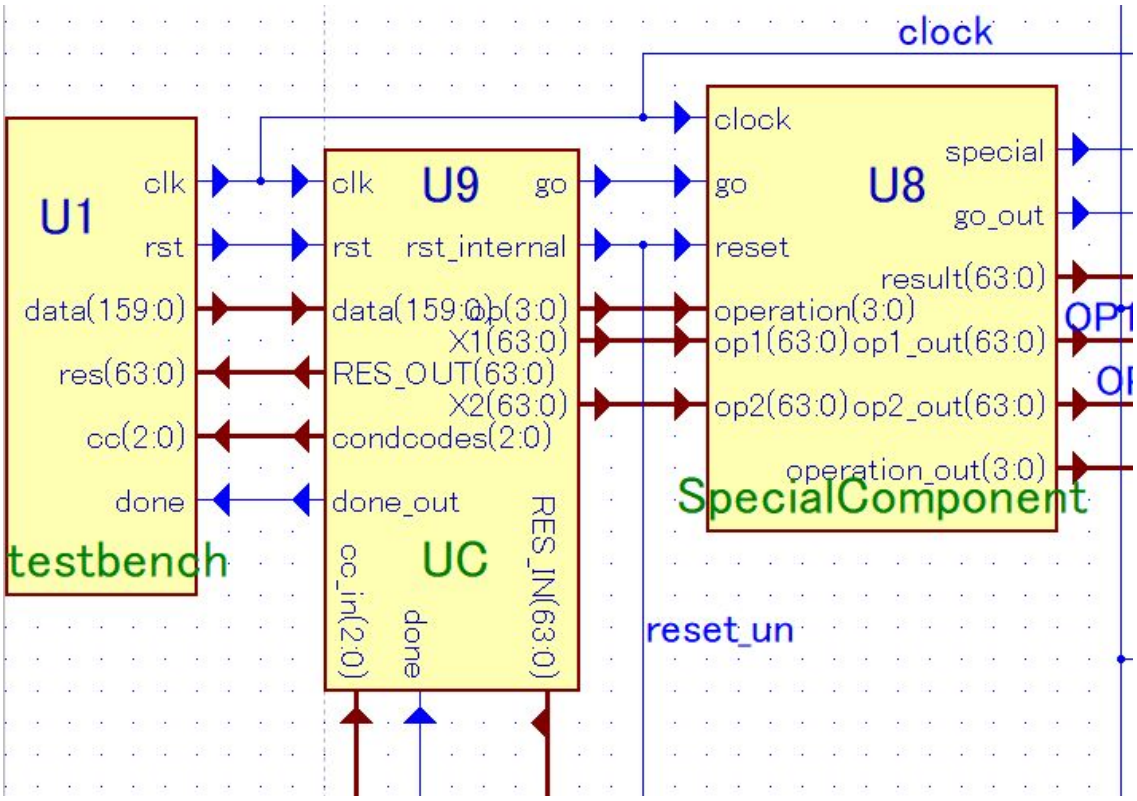

Figura 3: Componentes de entrada e controle.

A unidade de controle *UC* recebe um extenso bus de entrada que contém 32 bits de instrução junto com os valores de 64 bits de cada registrador envolvido com a conta atual, além de sinais de clock e reset. A *UC* serve de controle geral da unidade funcional, enviando os dados relevantes para o resto dos componentes e mantendo o estado geral da UF. O *SpecialComponent* recebe os operandos e a operação e conclui se a operação é especial, caso verdadeiro irá gerar um resultado próprio e subir o sinal de 'special', ignorando a passagem do sinal de 'go' para os componentes de cálculo. Caso falso irá repassar normalmente suas entradas.

Componentes de Cálculo

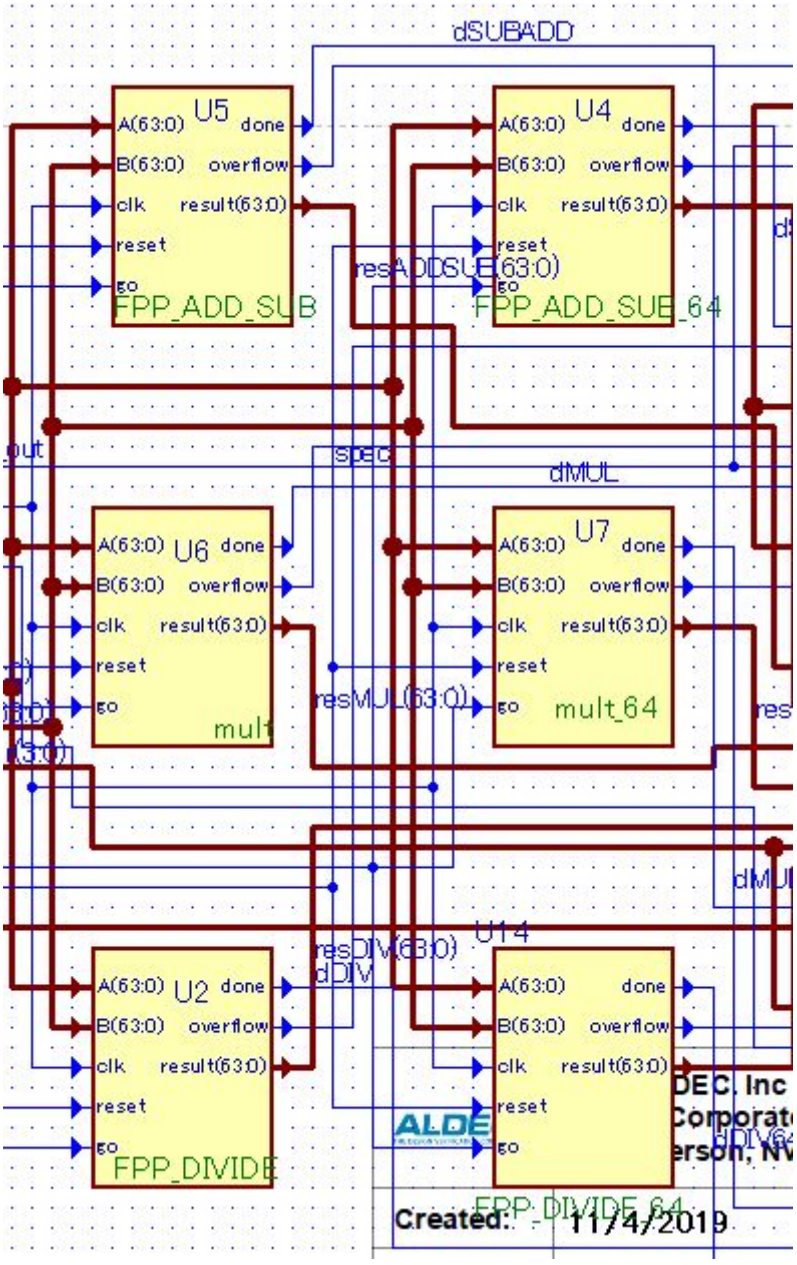

Figura 4: Seis componentes de operação aritmética.

Ao receber o sinal ‘go’ em alto, todos os 6 componentes de cálculo começam suas operações, sem exceções. Quando acaba sua execução mandam sinais de ‘done’ e ‘overflow’ (caso necessário), junto com seu resultado.

Controle de fim e tratamento de resultados

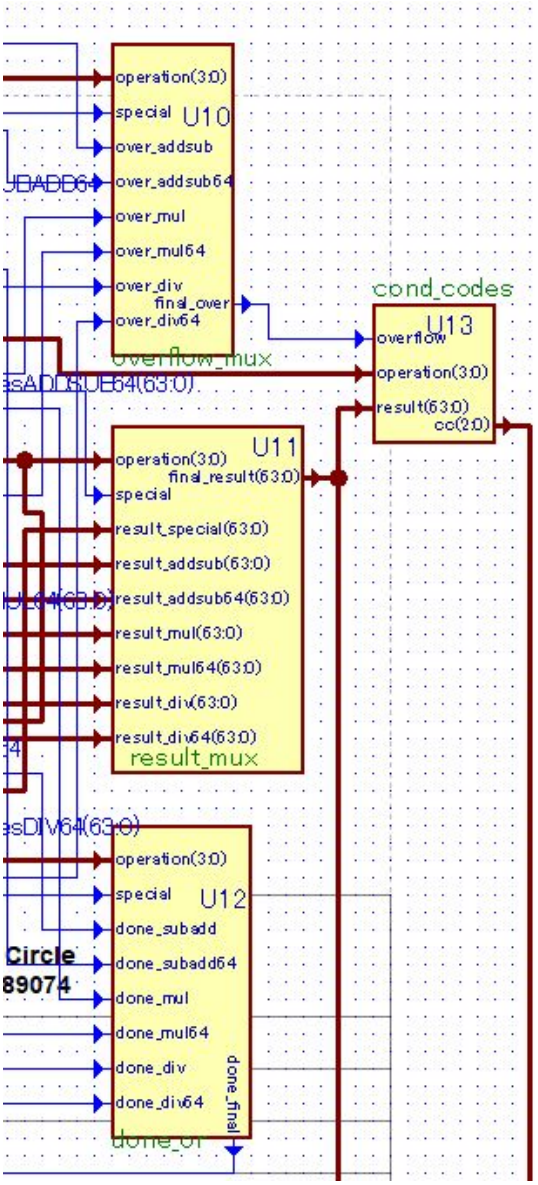

Figura 5: Geração de condition codes e controle de fim da operação.

Os três multiplexadores de fim recebem os sinais de ‘done’, ‘overflow’ e ‘result’ , utilizando do sinal de ‘operation’ para selecionar sua entrada escutada. O componente *cond_codes* recebe o resultado e o overflow e calcula as flags de condição para mandar para a unidade de controle. Os códigos usados por essa UF são N Z V , (negative - ‘1’ caso sinal do resultado seja negativo, zero - ‘1’ caso resultado seja zero, e overflow - ‘1’ caso sinal de overflow esteja em ‘1’), e são modificados e passados por todas as operações. Assim que o sinal de ‘done’ da operação relevante é ativo ele é repassado para a *UC*, junto com o resultado e os códigos de condição, em seguida a *UC* os repassa para a CPU (testbench), sinalizando o fim da operação e a volta para o estado inicial.

Simulação e Resultados

Para a garantia de funcionamento correto da unidade funcional se fez uso do componente de *testbench* , de simples construção consistindo de uma máquina de estados que realiza duas operações em seguida, esperando o sinal de 'done' para prosseguir.

```
when A =>
    state <= B;
when B =>
    X1 <= x"0000000000000000";
    X2 <= x"0000000000000000";
    opcode <= "00000000000";
    shamt <= "000000";
    state <= C;
when C =>
    X1 <= x"2DD5280000000000";
    X2 <= x"6DD1890248E70104";
    opcode <= "00011110011";
    shamt <= "000010";
    if (done = '1') then
        state <= D;
    end if;
when D =>
    X1 <= x"0000000000000000";
    X2 <= x"0000000000000000";
    opcode <= "00000000000";
    shamt <= "000000";
    state <= E;
when E =>
    X1 <= x"7FF0000000000000";
    X2 <= x"7FF0000000000000";
    opcode <= "00011110011";
    shamt <= "000110";
```

Figura 6: Simulação de entrada de dados.

Como todas as operações ocorrem simultaneamente, é possível escolher a de maior demora (divisão) e é possível observar o resultado de todas as outras, facilitando a checagem do funcionamento da UF (ignorando resultados dos componentes de 32 bits quando operando com 64, e vice-versa). Exemplos:

X1 : C04114BC6A7EF9DB : -34.161999999999999903366187936 63

X2 : 404B8FBE76C8B439 : +55.122999999999997557329889853

MUL64 : C09D6C729CBAB649 : -1883.111925999999812120222486 56

DIV64: BFE3D4EBB80D7C94 : -0.619741305807013365125612835982

ADD_SUB_64 : 4034F604189374BC : +20.960999999999998522071109619

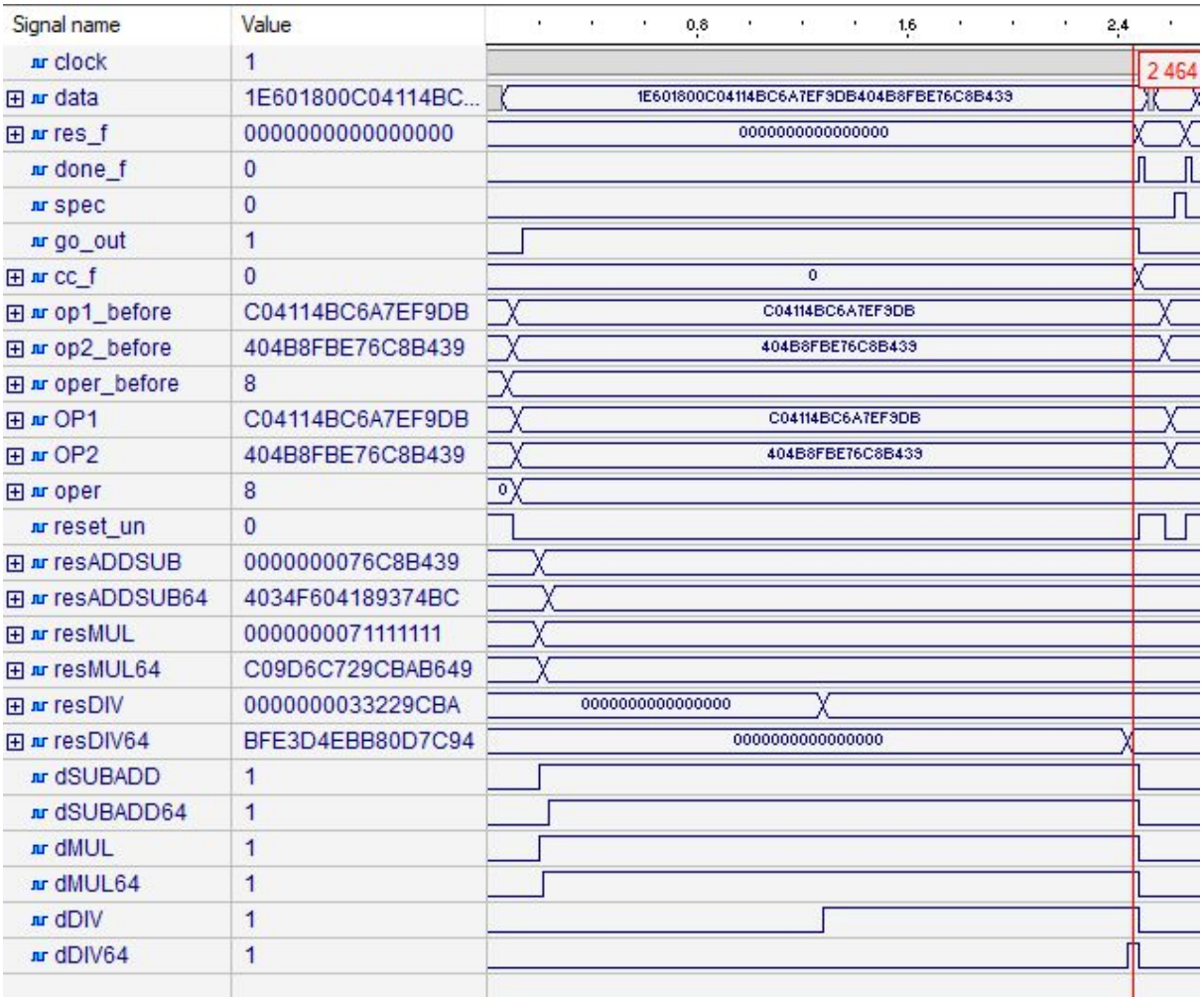

Figura 7: Waveform dos sinais relevantes da UF, exemplo 1. No final da lista sinais de done de cada operação, com as divisões demorando consideravelmente mais ciclos.

X1 : FFA3D4EBB8000000 : -6.96315431664669187880208216059E306
X2 : 82A064E29883A3AA : -5.013504271234874474797751598038E-296
MUL64 : 425451F755C97079 : +3.490980390777573852539062 5E11
DIV64: Espúrio, ocorreu overflow
ADD_SUB_64 : FFA3D4EBB8000000 : == X1 , dado que X2 é de valor insignificante comparado a X1, muito longe do nível de precisão mínimo de números ponto flutuante de precisão dupla.

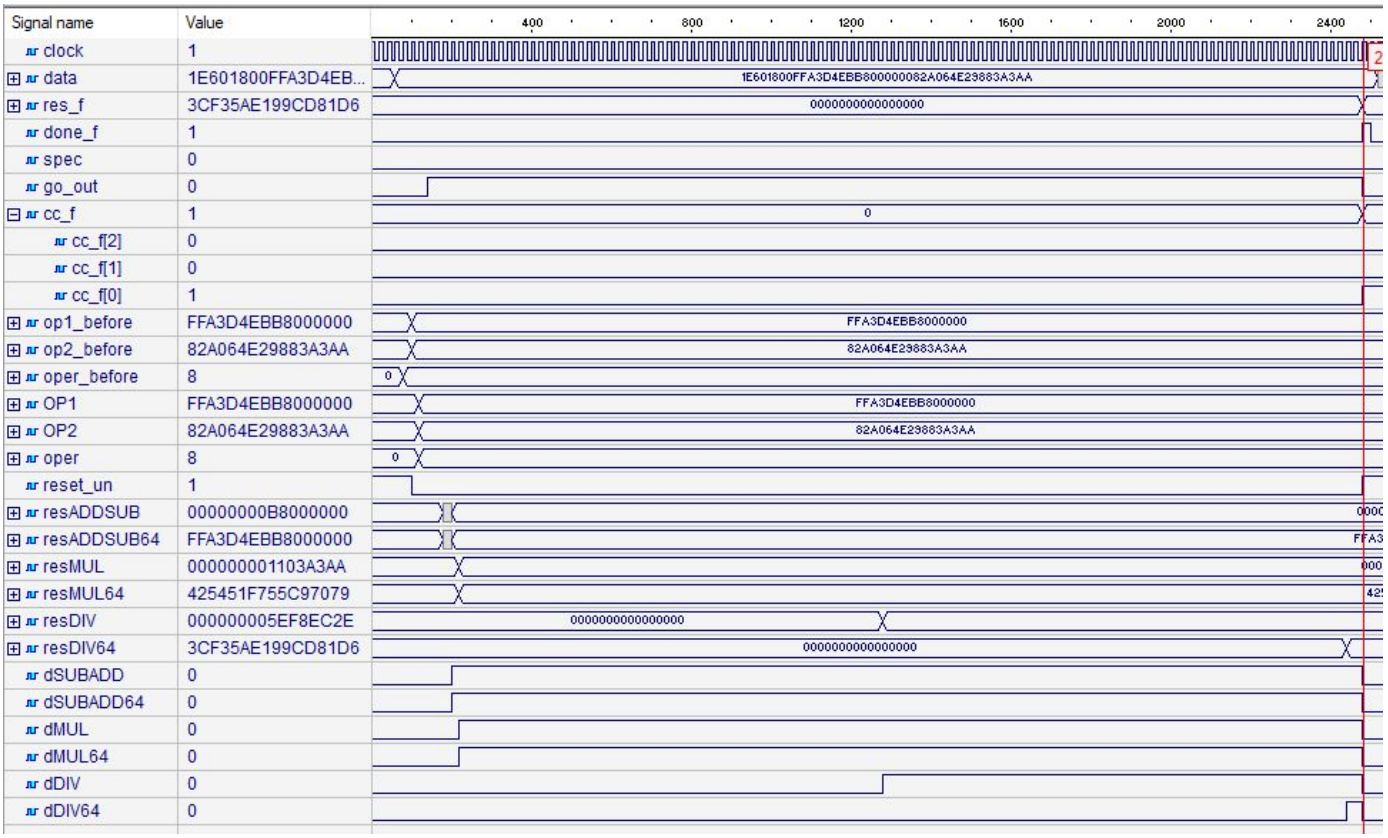

Figura 8: Waveform dos sinais relevantes da UF, exemplo 2. Bit (0) do vetor de Condition Codes (*cc_f*) sinaliza a ocorrência de overflow na divisão (expoente do resultado seria por volta de 10^+603).

Operação -> MUL64

X1 : 40B53A1CBFB15B57 : 5.434112299999999777355697005 99E3

X2 : FFF0000000000000 : -infinito

Res : FFF0000000000000 : -infinito

Operação -> DIV64

X1 : 40B53A1CBFB15B57 : 5.434112299999999777355697005 99E3

X2 : FFF0000000000000 : -infinito

Res : 8000000000000000 : -0

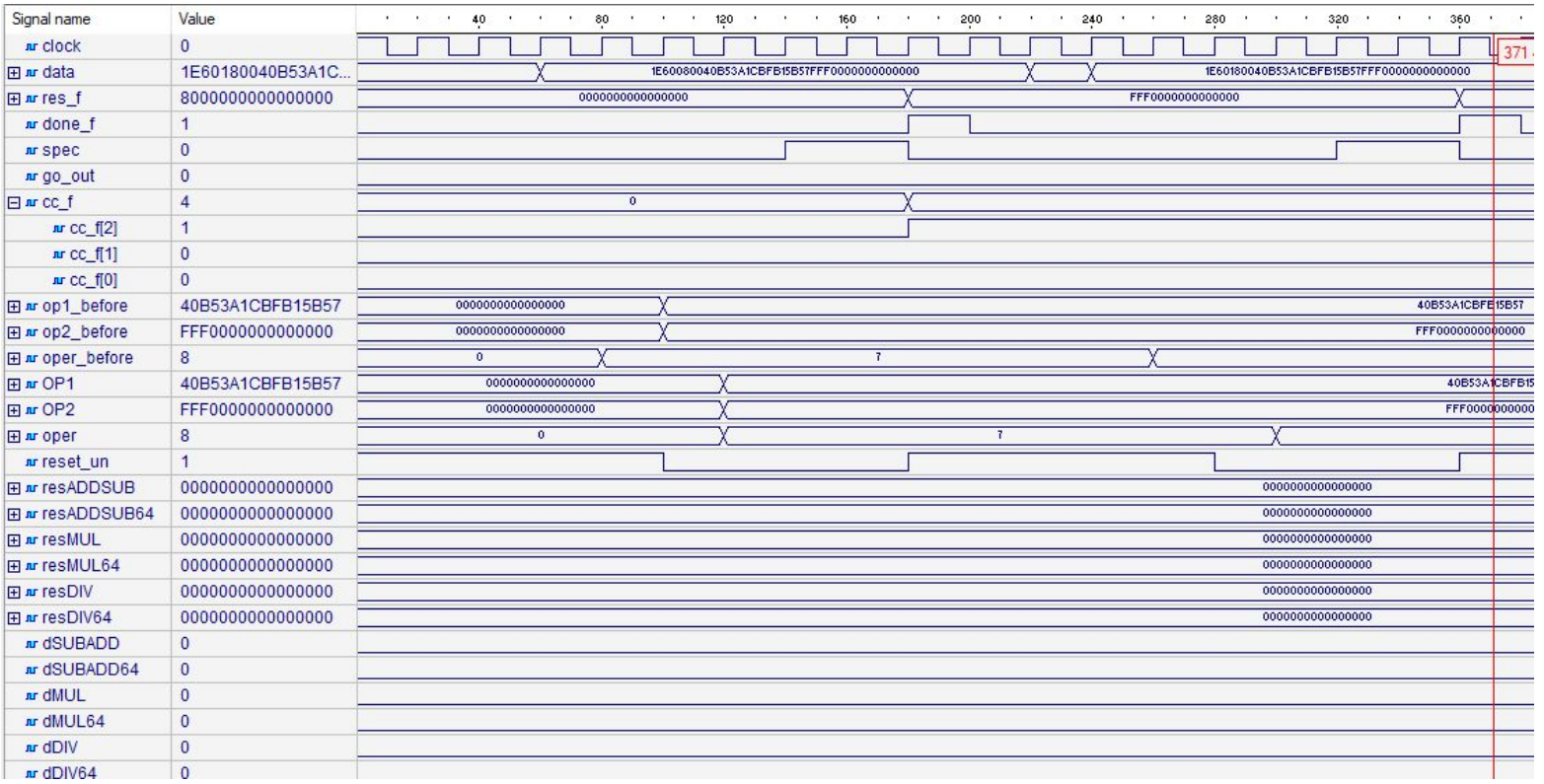

Figura 9: Exemplo 3, duas operações ocorrendo em sequência, uma de multiplicação outra de divisão, as duas sendo especiais e ativando o sinal ‘special’. Possível observar que nesse caso nenhuma unidade de cálculo começa sua operação.

Dependências com outros grupos

A implementação da unidade de ponto flutuante tem pouca dependência de outros grupos, pois a unidade funcional a ser desenvolvida não depende das demais para seu funcionamento, apenas é preciso receber os dados do processador principal e fornecer o resultado quando estiver pronto. Assim, existe uma dependência do processador principal sobre o módulo de operações de ponto flutuante para executar operações aritméticas específicas do conjunto de instruções.

Referências

IEEE Computer Society (1985) IEEE Standard for Binary Floating-Point Aritmetic, IEEE Std 754-1985

Representação em Ponto Flutuante. http://www-di.inf.puc-rio.br/~endler/courses/inf1612/aula-9.pdf