UNIVERSIDADE ESTADUAL PAULISTA
FACULDADE DE ENGENHARIA DE ILHA SOLTEIRA

VERÔNICA APARECIDA LOPES LIMA

DESENVOLVIMENTO DE UMA ARQUITETURA RECONFIGURÁVEL PARA O PROCESSAMENTO DE MODELOS NO AMBIENTE ABACUS.

Ilha Solteira
2007

VERÔNICA APARECIDA LOPES LIMA

DESENVOLVIMENTO DE UMA ARQUITETURA RECONFIGURÁVEL PARA O PROCESSAMENTO DE MODELOS NO AMBIENTE ABACUS.

Dissertação apresentada à Faculdade de Engenharia de Ilha Solteira da Universidade Estadual Paulista, Campus de Ilha Solteira, para obtenção do título de Mestre em Engenharia Elétrica.

Orientador: Prof. Dr. Norian Marranghello

Ilha Solteira
2007

Agradecimentos

A Deus, por me iluminar nos momentos difíceis me preservando no caminho do conhecimento.

Ao meu incentivador e orientador, Prof. Dr. Norian Marranghello, que pacientemente soube transmitir seus conhecimentos e respeitar o meu ritmo de trabalho.

Ao meu grande amigo Tiago Oliveira que disponibilizou sua atenção e dividiu seus conhecimentos técnicos comigo.

Ao meu marido Leandro Roberto que esteve sempre ao meu lado me incentivando.

Aos meus pais Pedro e Maria e avós Américo e Aparecida, pelo apoio, incentivo e dedicação que permitiram concentrar minhas atenções neste trabalho.

Aos novos amigos André Costa gerente técnico do Projeto BrazilIP da Unicamp e Edwin Cordeiro da PI componentes que dividiram seus conhecimentos técnicos comigo.

"Enriquecer nossas virtudes, adquirindo conhecimentos novos, é nosso simples dever."

Provérbio, 524-550

Resumo

O objetivo deste trabalho é o desenvolvimento de uma arquitetura reconfigurável estaticamente, de um elemento de processamento (MPH) para o ambiente de simulação de circuitos ABACUS. Este elemento de processamento consiste de um conjunto de unidades funcionais que podem ser relacionadas por meio de algumas palavras de controle armazenadas na ROM, e cuja interconexão pode ser alterada para que o *hardware* de processamento se adapte ao modelo do elemento de circuito a ser simulado. O projeto foi descrito em linguagem VHDL e simulado com o auxílio do *software* QUARTUS II.

Palavras-chave— sistemas digitais reconfiguráveis, simulação de circuitos, arquiteturas computacionais multiprocessadas.

Abstract

The aim of this work is the development of a statically reconfigurable architecture, of a processing element (MPH) for the ABACUS circuit simulation environment. This processing element consists of a set of functional units that can be related by means of some control words stored in the ROM, and whose interconnection can be modified so that the processing hardware be adapted to the model of the circuit element to be simulated. The project was described in VHDL, and simulated with the aid of Quartus II software.

Keywords — *reconfigurable digital systems, circuit simulation, multiprocessor computer architectures.*

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo I – Introdução

1.0 – A importância da simulação de circuitos em computadores.

Os projetos de circuitos integrados foram impulsionados com a simulação de circuitos em computadores, por serem ferramentas capazes de prever o comportamento de um circuito com detalhes suficientes para dar uma idéia muito próxima do seu comportamento real.

Através da simulação o projetista descobre qual a configuração disponível, que terá melhor desempenho para o seu projeto, entre várias possíveis. Com a simulação não há a necessidade de se construir vários protótipos diminuindo com isso o custo e o tempo gasto com essas etapas, que hoje em dia são desnecessárias. Por exemplo, quando fazemos a simulação de um circuito não corremos o risco desse circuito queimar, já quando fazemos à montagem de um protótipo em laboratório, corremos vários riscos, entre eles: usar componentes defeituosos, ou seja, com defeito de fábrica; do projeto estar com erros (isto também pode ocorrer na simulação); de fazer uma montagem errada e como conseqüência queimar os dispositivos.

No simulador de circuitos é possível realizar vários testes com a arquitetura tais como modificar valores dos componentes, alterar a configuração sem danificar o circuito.

Como diz MEHL [1]: "A disponibilidade dos modernos simuladores em microcomputadores tem tido notáveis implicações também no ensino da eletrônica. Desenvolvidos originalmente como ferramentas para pesquisas e projetos avançados de circuitos integrados, os simuladores de circuitos eletrônicos estão atualmente sendo largamente utilizados no ensino de Engenharia. Um grande número de Universidades tem adotado programas de simulação como auxiliares didáticos em disciplinas de análise de circuitos e de projeto de circuitos eletrônicos. O uso de simuladores para o ensino de análise de circuitos, longe de se considerar um modismo passageiro nos cursos de graduação de Engenharia Elétrica, é resultado da crescente tendência de se enfatizar o ensino de técnicas de projeto, nas quais os microcomputadores e *workstations* são ferramentas de inestimável valor".

1.1 – História dos simuladores de circuitos.

No começo o uso dos simuladores restringia-se a seus criadores ou aos militares, isso ocorria principalmente devido à complexidade desses sistemas e ao seu alto custo. Depois de um bom tempo, mais ou menos em 1972, que os simuladores ficaram disponíveis para a comunidade de projetistas em geral, como veremos na seqüência. O texto a seguir está baseado na apostila do Prof. Ewaldo Mehl [1].

Em 1959 a IBM desenvolveu um programa que permitia a análise de transitórios no chaveamento de transistores chamado PETAP, seu uso era restrito à IBM. Em 1964, a partir do PETAP a IBM desenvolveu o PREDICT, um programa que determinava o desempenho de sistemas eletrônicos frente a radiações nucleares, seu uso foi exclusivo dos militares.

Também em 1964 foi desenvolvido o NET1 no laboratório norte-americano de Los Alamos (onde foi desenvolvida a primeira bomba atômica, durante a Segunda Guerra Mundial), também foi desenvolvido para uso militar, verificando o efeito de radiações nucleares nos circuitos eletrônicos.

Em 1965 também produto da IBM, como evolução do programa PETAP, surgiu o ECAP1 porém para aplicações civis. Possuía defeitos comuns à programação de primeiras gerações: pouco amigável e difícil de ser utilizado. Tinha tendências de não convergência na solução das equações. Apesar desses problemas, foi colocado à disposição dos usuários de equipamentos IBM.

Em 1967 o SPECTRE, considerado como evolução do PREDICT, desenvolvido pela IBM para aplicações na área aeroespacial. Assim como o programa NET1 de Los Alamos, realizava a solução de equações diferenciais não-lineares por meio de sucessivas integrações numéricas em pequenos passos. Esse fato, aliado às baixas velocidades dos computadores de então, fazia com que o tempo de processamento se tornasse extremamente longo.

Em 1968 foi desenvolvido o TRAC na empresa aeroespacial norte americana Rockwell, através de uma equipe de engenheiros assessorada por matemáticos. Utilizado um método de integração que foi chamado de Método de Euler Reverso *(Backward_Euler Method).* Neste método as equações integro-diferenciais são transformadas a cada passo de integração em um conjunto de equações algébricas, ou seja, de mais fácil solução e conseqüentemente com menor tempo de processamento. O uso desse programa, no entanto, permaneceu restrito à área militar. O programa TRAC possuía uma série de subrotinas

escritas em linguagem “de máquina” (*assembly*), fato que o tornava não- transportável para outros computadores

Em 1969 foi desenvolvido na Universidade da Califórnia em Berkeley o programa TIME, com base no programa TRAC, porém com todas as subrotinas em linguagem FORTRAN, de modo a funcionar em computadores de diferentes fabricantes.

Também em 1969 surgiu o CIRPAC, desenvolvidos nos laboratórios Bell. Utilizava esquema de cálculo semelhante aos programas TRAC e TIME, porém pequenas melhorias nos métodos de integração faziam-no mais rápido que seus antecessores.

Em 1970 surgiu o BIAS e 1971 surgiu o BIAS3 que foram desenvolvidos na Universidade da Califórnia em Berkeley a partir de um problema prático, onde se desejava observar o efeito da variação de temperatura em circuitos transistorizados simples. Utilizavam o modelo de Ebers-Moll para modelamento dos transistores e obtinha a solução do sistema de equações pelo método de Newton-Raphson.

Também em 1971 foram desenvolvidos o CANCER e o SLIC, estes dois programas foram resultados de um trabalho desenvolvido na Universidade da Califórnia em Berkeley por Ronald A. Rohrer, coordenando uma equipe de alunos de pós-graduação, que estudaram técnicas de descrição de circuitos e de solução dos sistemas de equações obtidos. Estes programas utilizavam o método de Análise Nodal Modificada. Antes de se tornar professor em Berkeley, Rohrer havia trabalhado na empresa Fairchild, onde desenvolveu em 1968 um programa chamado FAIRCIRC para simulação dos primeiros circuitos integrados produzidos pela empresa.

Em 1973, a IBM desenvolveu o programa ASTAP. Foi apresentado na época como principal novidade no uso de técnicas inéditas para otimização da velocidade de processamento, tomando vantagem do fato das matrizes que descrevem os circuitos elétricos serem esparsas (possuem muitos elementos nulos). Este programa foi amplamente utilizado pelos projetistas da própria IBM e também esteve disponível para usuários externos à empresa.

Em 1972 surge o programa SPICE, desenvolvido na Universidade da Califórnia em Berkeley. Ele foi apresentado na Tese de Doutorado de Laurence W. Nagel, mas apresentava algumas limitações. Nagel posteriormente modificou o programa original, surgindo em 1975 à versão SPICE2. Estes programas foram colocados à disposição do público e ganharam reconhecimento internacional pela versatilidade dos modelos utilizados e velocidade de processamento. O programa SPICE2, codificado em FORTRAN, foi fornecido gratuitamente pela UC-Berkeley a diversas universidades e centros de pesquisas ao longo da década de 1980

e compilado para os mais variados computadores. Nos anos seguintes diversos alunos de Berkeley encarregaram-se de re-codificar o programa SPICE2 em linguagem "C", dando origem ao SPICE3, igualmente distribuído a usuários universitários.

Em 1975, a IBM desenvolveu o ICD, um programa da IBM muito parecido com o SPICE2 nos métodos de solução empregados. Foi descrito em linguagem APL com características interativas com o usuário.

Em 1980 Laurence W. Nagel (criador do SPICE2) desenvolveu o ADVICE, nos laboratórios Bell. Foi considerado uma versão mais amigável do programa SPICE2.

Surge então em 1985 o PSPICE, primeiro programa comercial desenvolvido exclusivamente para ser usado em microcomputadores. É basicamente o programa SPICE2, que foi adaptado pela empresa MicroSim para uso em microcomputadores IBM-PC e posteriormente para *workstations* com sistema operacional UNIX. A grande vantagem em relação ao programa SPICE2 de Berkeley foi à apresentação de um programa associado chamado PROBE, de modo a permitir a visualização dos resultados da simulação de modo mais interessante que as listagens originais do SPICE2. As versões mais recentes do programa SPICE incluem a possibilidade de se efetuar simulação lógica do circuito simultânea à simulação analógica. Estão também disponíveis diversas bibliotecas de dispositivos semicondutores comerciais pré-moldados. Há versões para microcomputadores (sistemas MS-DOS e Windows) e para diversos tipos de *workstations*;

Em 1989 surge o IG-SPICE, um programa comercial também baseado no SPICE2, para microcomputadores, produzido pela Empresa INTUSOFT. Tem como principal característica o interfaceamento com um programa gráfico que permite ao usuário desenhar seu circuito, extraindo-se automaticamente de tal desenho o arquivo de descrição do circuito necessário para simulação. Esta característica foi também incorporada pela MicroSim em versões posteriores do SPICE para o sistema Windows.

Até aqui mostramos apenas aqueles programas de simulação que tiveram maior importância histórica. De 1989 para cá surgiram vários outros programas de simulação, com diversas características, dentre eles destacamos o ABACUS [2], capaz de avançar significantemente a simulação de circuitos no nível de transistores, por meio de uma metodologia de simulação diferente da adotada nos simuladores de circuitos convencionais. Esta metodologia, contudo, depende do desenvolvimento de um *hardware* dedicado, no qual ela possa ser implementada. Este *hardware* foi especificado em trabalhos anteriores [3], [4], [5], [6].

A idéia da metodologia é ter um arranjo de processadores especializados, cada um dos

quais representa o comportamento de um elemento do circuito sob análise. Os processadores do arranjo são conectados por um sistema de barramentos programáveis. Deste modo, o arranjo pode ser programado para imitar a estrutura de interconexão do circuito que estiver sendo simulado. O computador hospedeiro, mestre do arranjo, é responsável por fornecer os dados de cada elemento de circuito ao elemento de processamento correspondente no arranjo, bem como por iniciar o processo de simulação. A partir de então os processadores do arranjo trabalham concorrentemente e assíncronamente. Esta operação inclui a leitura dos dados nos respectivos registradores de entrada, o processamento desses dados, e o armazenamento dos resultados nos registradores de saída. A chave para este funcionamento é viabilizar o fluxo de dados representativos do comportamento do circuito pelo arranjo de elementos de processamento, de forma análoga à propagação dos sinais elétricos no circuito real. Este ciclo de leitura-processamento-armazenagem de dados é repetido continuamente pelos processadores do arranjo até que eles atinjam um estado estável. Quando esta estabilização é alcançada os resultados são salvos, o tempo de simulação é incrementado e um novo processo iterativo é iniciado. Este processo é repetido até que se obtenham estimativas de comportamento para todo o intervalo de tempo desejado. Trabalhos anteriores [8] já mostraram que este ambiente é capaz de simular circuitos ULSI no nível de transistores com um ganho considerável de velocidade em relação a simuladores padrões, como o SPICE2.

O computador hospedeiro tem quatro processadores de uso geral totalmente conectados, cada um dos quais dedicados a uma de suas tarefas: o processamento de dados de entrada, incluindo a interface com o usuário e todo o pré-processamento de dados para encaminhamento ao arranjo; processamento de dados de saída, incluindo o pós-processamento das soluções e a preparação dos dados de saída, incluindo o pós-processamento das soluções e a preparação dos dados de saída a serem apresentados ao usuário; controle interno do hospedeiro e do andamento global da simulação; e controle e interface do hospedeiro com o arranjo de processadores. Estes processadores são conectados entre si por meio de canais de comunicação bidirecionais de alta velocidade, garantindo o máximo possível de paralelismo no hospedeiro. Para viabilizar o armazenamento e manipulação de dados dos circuitos ULSI, cada um destes processadores tem 1Gbyte de memória. O processador responsável pelo gerenciamento do arranjo tem canais especiais de altíssima velocidade para a transferência de dados entre o hospedeiro e os processadores do arranjo, bem como um canal de controle especial para coordenar estas transferências e para enviar informações para a configuração das chaves que compõem o barramento configurável do arranjo.

O arranjo de elementos de processamento deveria ser composto por tantos

processadores quanto fossem os elementos do circuito a ser analisado. Contudo, isto não é possível na prática. Sendo assim, é necessário escolher um número grande suficiente para comportar uma gama considerável de circuitos pequenos o suficiente para ser implementável.

Para viabilizar a relação de compromisso, estabeleceu-se que cada elemento de processamento será integrado pelos seguintes componentes: uma unidade de controle, para supervisionar o processamento local de dados; uma RAM local, para servir de memória de rascunho; uma ROM, para armazenar os modelos de elementos de circuitos e que servirão para configurar os elementos operativos do processador; uma ULA de ponto flutuante, que possa ser configurada para se moldar ao elemento de circuito a ser considerado; e um conjunto de canais bidirecionais com registradores de entrada e saída, para permitir a comunicação ágil entre os elementos de processamento do arranjo e entre cada um deles e o computador hospedeiro.

Esta dissertação segue a seguinte disposição:

- Capítulo I – *Introdução*: neste capítulo falamos da importância de simular circuitos em computadores, de vários simuladores importantes da história e o surgimento do simulador ABACUS.
- Capítulo II - *Fundamentação Teórica*: neste capítulo descrevemos a arquitetura do simulador de circuitos ABACUS, com seus processadores, conexões etc.
- Capítulo III – *Descrição e Desenvolvimento dos Elementos do Circuito em Blocos de Hardware*: neste capítulo mostramos a quantidade de blocos de *hardware* que foram precisos para descrever cada elemento do circuito, assim como o seu desenvolvimento, que compõem a parte operativa do processador de modelos ABACUS.
- Capítulo IV – *Desenvolvimento dos Blocos em Ponto Flutuante*: neste capítulo descrevemos como foram desenvolvidos os blocos do multiplicador, divisor, somador e subtrator em ponto flutuante.
- Capítulo V - Definição da Arquitetura Reconfigurável: neste capítulo definimos a arquitetura da unidade de processamento e os controles necessários para a escolha de cada elemento.
- Capítulo VI - *Resultados da Compilação, Simulação e de Teste Laboratorial da unidade elemento_circuito*: neste capítulo mostramos informações referentes à fase de compilação, simulação, implementação e teste do elemento_circuito no FPGA.

Abordando com isso, os procedimentos utilizados, a geração dos arquivos de simulação e dos resultados obtidos.

- Capítulo VII – *Considerações Finais*: neste capítulo, fazemos os comentários finais sobre o desenvolvimento da arquitetura proposta e seus resultados gerais. E sugestões para trabalhos futuros.
- Bibliografia
- Anexo I : *Composição do DVD que acompanha esta tese.*
- Anexo II: *Simulações realizadas.*
- Anexo III: *Exemplo do cálculo efetuado na simulação do resistor*.
- Anexo IV: *Pinagem do FPGA configurado.*

Capítulo II - Fundamentação Teórica

2.0 – Introdução

ABACUS (hArdware Based CircUit Simulator) é o nome dado a uma metodologia para a simulação de circuitos proposta por MARRANGHELLO [2]. O sistema baseia-se na exploração do paralelismo dos arranjos computacionais de uma forma ampla, para possibilitar a redução do tempo de processamento desses simuladores.

No sistema ABACUS, as simulações de circuitos ULSI (*Ultra Large Scale Integration*) procedem-se pelo uso de um arranjo especial de processadores, os quais incorporam modelos dos elementos de circuitos e sobre o qual é possível mapeá-los.

No decorrer deste trabalho vamos apresentar uma arquitetura reconfigurável estaticamente, de um elemento de processamento (MPH) para o simulador de circuitos ABACUS. Neste tipo de arquitetura o circuito apresenta várias configurações, e as reconfigurações acontecem apenas ao final de cada tarefa de processamento As arquiteturas desta classe são chamadas **SRA**- *Statically Reconfigurable Architecture* .

Neste trabalho este elemento de processamento consiste de um conjunto de unidades funcionais que podem ser relacionadas por meio de algumas palavras de controle armazenadas na ROM, e cuja interconexão pode ser alterada para que o *hardware* de processamento se adapte ao modelo do elemento de circuito a ser simulado.

Anteriormente já foi feito o projeto da arquitetura de um elemento de processamento (MPH) para o simulador de circuitos ABACUS, só que neste projeto [3], foi implementado com uma arquitetura convencional estática microprogramada.

A principal diferença do projeto de FRANÇA [3] para o descrito em nossa dissertação reside no fato de utilizarmos uma arquitetura reconfigurável dedicada.

2.1- Arquitetura do sistema ABACUS.

O sistema ABACUS é composto por cinco processos básicos, descritos a seguir, e sua arquitetura é mostrada na Fig.2.1.

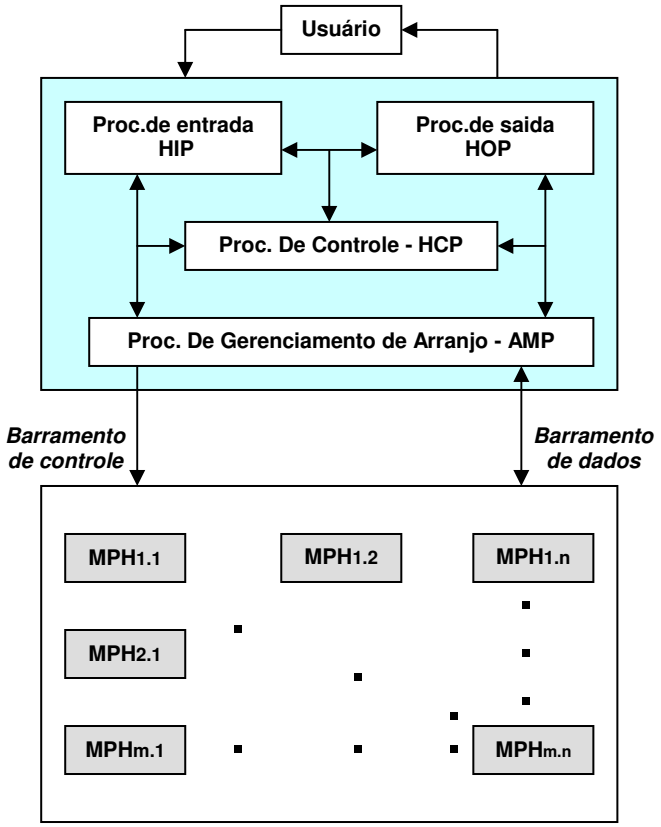


Figura 2.1 - Arquitetura do simulador de circuitos ABACUS.

- HIP *(Host Input Process)*: O processo de entrada HIP, tem a função de receber os dados correspondentes à descrição do circuito a ser simulado, e organizá-los para o tratamento pelos demais processos do simulador;
- HCP *(Host Control Process)*: O processo de controle HCP, tem a função de controlar e organizar o tráfego dos dados, que foram enviados pelo processador de entrada e as soluções que serão enviados ao usuário através do processador de saída;
- AMP *(Array Manager Process)*: O processo de gerenciamento do arranjo AMP, tem como objetivo gerenciar o funcionamento do arranjo de processadores, ou seja, controlar a operação dos processadores do arranjo fundamentalmente com base nos dados que cada processador do arranjo fornece ao processo AMP;
- HOP *(Host Output Process)*: O processo de saída HOP, é o inverso do processador HIP, fornece as respostas solicitadas pelo usuário;
- MPHs *(Model Processing Hardware-element)*: Os processadores MPHs, executam efetivamente a simulação dos componentes do circuito, com bases nos dados a ele

fornecido pelo processador AMP, ou seja dedica-se à tarefa exclusiva de simular os modelos dos componentes de circuito.

2.2- O funcionamento do processador Hospedeiro.

O processador hospedeiro é composto por quatro processos, *Host Control Pro*cess, *Host Input Process, Host Output Process* e *Array Management Process*. É um processador mais complexo que os MPHs que serve de gerente do arranjo. Além das tarefas de interfaceamento com o usuário ele é responsável pelo gerenciamento do arranjo.O gerente lê a *netlist* de entrada e as demais informações referentes à simulação. Identifica os elementos do circuito, suas interconexões e as análises a serem feitas. Se o circuito é muito grande e não é possível mapeá-lo no arranjo, ele particiona o circuito num número adequado de subcircuitos e então mapeia no arranjo cada subcircuito, fazendo as conexões necessárias entre cada subcircuito. A configuração do arranjo é estabelecida sempre que um novo circuito é nele mapeado. Após mapear o circuito no arranjo, o gerente transfere os parâmetros relativos a cada elemento e as informações sobre as análises a serem feitas para os processadores correspondentes. Enquanto espera pelos resultados da simulação, o gerente, além de controlar a convergência global do arranjo, pode fazer o tratamento dos dados disponíveis para a saída. Quando o arranjo chega a uma solução, o hospedeiro para momentaneamente seu trabalho, recupera os resultados, disponíveis, reinicializa o arranjo e retorna seu processamento.Após todas as análises serem resolvidas pelo arranjo, o gerente apresenta os resultados na forma solicitada pelo usuário, terminando a simulação.

2.3 - A arquitetura do processador MPH.

A arquitetura do processador é composta por filas de entrada e saída (FES); uma unidade lógica e aritmética (ULA); uma unidade de controle elementar (UCE); uma unidade de modelos armazenados (UMA); e uma memória de escrita e leitura (MEL). A Fig. 2.2 mostra essa arquitetura e seus blocos são descritos a seguir.

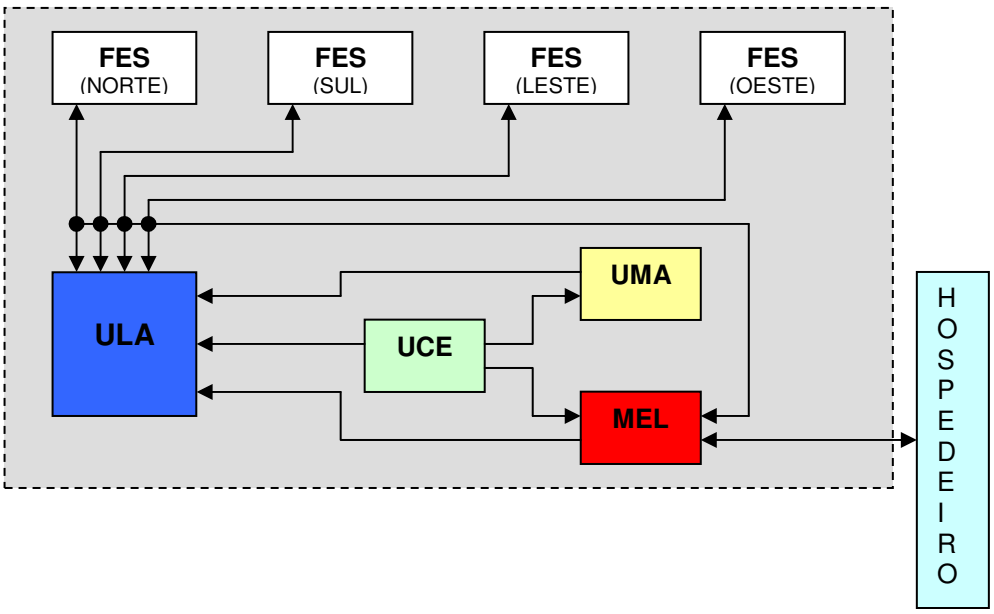


Figura 2.2 - Arquitetura do processador MPH

- FES: São *buffers* bidirecionais de alta velocidade, responsáveis pela comunicação necessária entre os MPHs, na etapa de entrada dos dados ou saída dos resultados.
- ULA: A Unidade Aritmética é formada por blocos que realizam as operações aritméticas e lógicas que são: somadores, multiplicadores, subtratores, divisores e comparadores que serão utilizados neste trabalho.

Durante a fase do arranjo de configuração estas unidades podem ser convenientemente configuradas de acordo com os modelos dos dispositivos disponíveis na UMA. Esta configuração só é alterada quando há o processo de troca do circuito, ou seja, quando ocorre a mudança do circuito a ser simulado pelo arranjo.

- UCE: É uma unidade que tem finalidade de controlar tanto o processamento dos MPH, como os seus interfaceamentos com o hospedeiro ou o resto do arranjo. O controle interno de cada MPH é também realizado por esta unidade. A UCE controla os três fluxos de informação abaixo.

1º- sinalização para o gerente do estado de convergência do processador;

2º- a transferência das informações do gerente e dos resultados da simulação para o gerente;

3º- a comunicação dos resultados parciais da simulação entre os processadores.

- UMA: É uma memória ROM (*Ready Only Memory*- memória somente de leitura) na qual os diversos modelos conhecidos pelo ABACUS ficam alocados. Estes modelos podem ser definidos pelo usuário. Eles são responsáveis pela configuração dos circuitos da ULA, durante a simulação de cada sub-circuito, de acordo com o dispositivo a ser simulado. Esta unidade armazena microprogramas que acionam a ULA convenientemente, ou seja, a UMA armazena os microprogramas correspondentes aos dispositivos conhecidos pelo MPH.
- MEL: É uma RAM (*Randon Access Memory* – memória de acesso aleatório) utilizada para o armazenamento temporário, de curta duração, dos dados de entrada e dos resultados obtidos na simulação.

Durante o mapeamento do circuito o gerente armazena os dados de cada dispositivo simulado e aqueles relativos as análises, na MEL do MPH correspondente. Durante cada fase da simulação, as MELs ficam dedicadas ao respectivo MPH. A comunicação entre os MPHs se processa pelas filas de entrada e saída disponíveis (FES). Ao detectar o fim da simulação o gerente lê os resultados na MEL de cada MPH.

Em resumo, esta arquitetura prevê a simulação do circuito por um tratamento real em condições de fluxo de sinais, ou seja, é como se os componentes estivessem alocados sobre uma matriz de contato. Nesta forma de análise somente os instantes inicial e final do funcionamento dos processadores do arranjo são determinados pelo computador hospedeiro, porém durante a execução da simulação, os MPHs operam concorrente e assíncronamente tanto entre si como em relação ao hospedeiro.

2.4 - O arranjo de processadores.

Os elementos de processamento de modelos (MPHs) descritos acima, são os componentes básicos para a formação do arranjo de processadores. A sua configuração é realizada efetivamente pelo processador hospedeiro (gerente), na execução do processo AMP, como vimos anteriormente.

Antes do início da simulação todos os MPHs estão num estado desconhecido. Quando o arranjo é iniciado pelo gerente, cada MPH acessa suas portas de entrada na busca por dados. Na primeira tentativa apenas aqueles MPHs que estiverem nas entradas primárias do circuito encontrarão dados de entrada reais.

Após computarem suas respostas, estas serão colocadas nas suas portas de saída de modo que os processadores vizinhos possam acessar e calcular suas respostas.

Enquanto este grupo de processadores, ligados diretamente aos processadores primários, computa suas análises, aqueles primeiros iniciam um segundo ciclo do cálculo, buscando novos dados nas entradas, processando e transferindo os resultados para as saídas. Quando todos os processadores convergirem para uma única solução, os dados são transferidos para o gerente. A Fig. 2.3 representa o diagrama esquemático das ligações de um MPH, com seus vizinhos (Norte, Sul, Leste e Oeste).

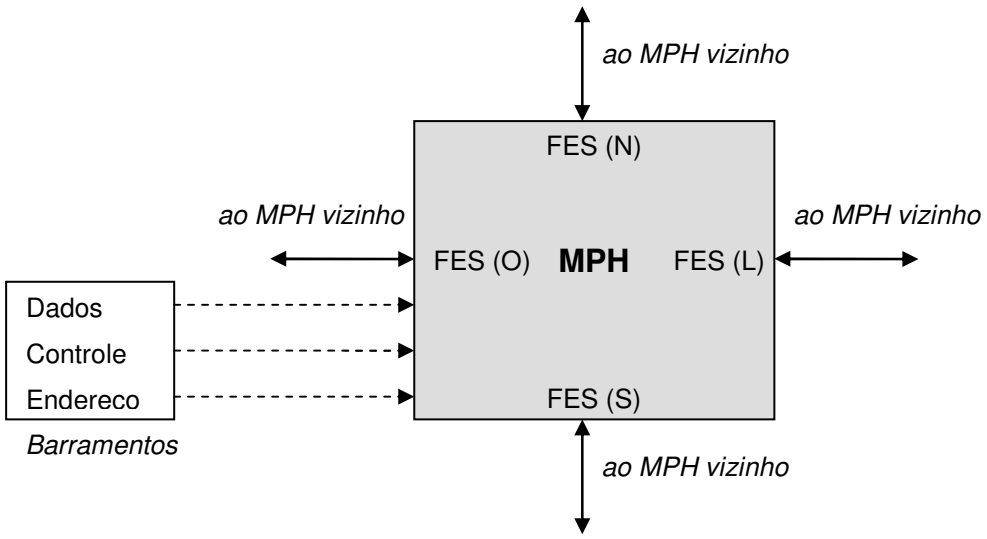


Figura 2.3 – Bloco MPH

2.5- Algoritmo de processamento dos elementos de circuito.

O algoritmo empregado na solução do sistema é baseado no método numérico da bisseção, e consiste fundamentalmente no seguinte: cada processador do arranjo recebe os valores de tensão e corrente calculados pelos processadores a ele conectados e calcula a média aritmética das tensões (va' e vb') em cada um dos terminais (a e b) do elemento de circuito a ele confiado e a respectiva diferença de potencial (Vab'), bem como o somatório dos valores algébricos das correntes que a ele confluem (ia' e ib') e a média da corrente (Iab') Estes cálculos iniciais das correntes e tensões que estão entrando no MPH chamamos na Fig. 2.4 de BCPI (bloco de circuito do processo inicial, é responsável por calcular va', vb', Vab' e Iab') que será explicado com mais detalhes no Capítulo III. Obtido o valor deste somatório das correntes, calcula-se uma segunda diferença de potencial (Vab'') sobre o elemento. Estabelece como nova tensão (Vab) sobre o elemento, o valor médio das duas diferenças de

potencial recém mencionadas (Vab' e Vab''). Calcula a corrente (Iab) correspondente à tensão estabelecida (Vab), de acordo com o modelo do elemento em questão. Tomando a voltagem num dos terminais como referência (va), atribui ao outro (vb) a soma da voltagem deste (va) com a tensão estabelecida (Vab) para o elemento. Finalmente, informa os novos valores de corrente e voltagem aos processadores vizinhos. Ao final de cada iteração, verifica se os novos valores obtidos estão em conformidade com a expectativa de erro, declarando seu estado de convergência ao processador gerente. Este diagrama de fluxo do algoritmo pode ser visto na Fig.2.4.

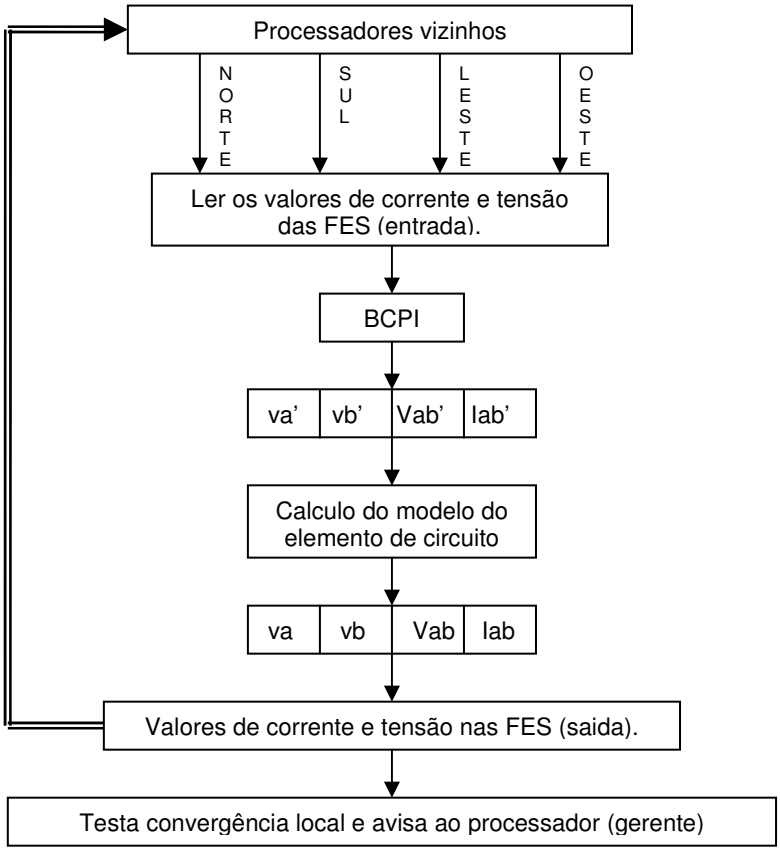


Figura 2.4 - Diagrama do algoritmo de processamento dos modelos.

Na Fig. 2.5 mostramos a estrutura de conexão entre os processadores. O MPH acessa suas portas de entradas na busca por dados, o processador central está representando um componente. Ele busca nas portas de saídas dos MPHs vizinhos (Norte, Sul, Leste e Oeste) os valores de tensão e corrente para fazer a simulação do componente de circuito. Para facilitarmos os cálculos iniciais das correntes e tensões que vão entrar no processador, dividimos o MPH em dois pólos "a" e "b", como vemos na Fig. 2.5 no pólo "a" temos o MPH

Norte (com a tensão v0 e corrente i0) e o MPH Leste (com a tensão v1 e a corrente i1), no pólo “b” temos o MPH Sul (com a tensão v2 e a corrente i2) e o MPH Oeste (com a tensão v3 e a corrente i3).

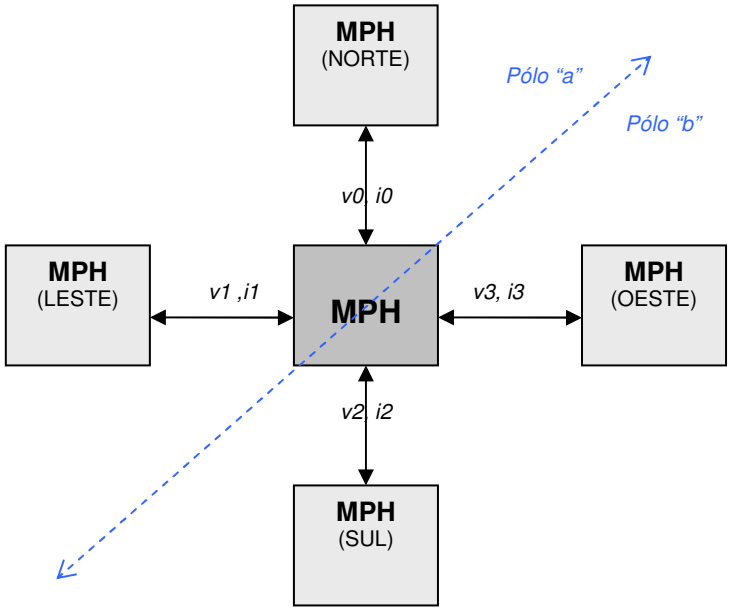


Figura 2.5 – Estrutura de conexão entre os processadores.

Capítulo III – Descrição e Desenvolvimento dos Elementos do Circuito em Blocos de *Hardware*.

3.0 – Introdução

Neste capítulo descrevemos os modelos dos elementos de circuitos que podem ser simulados pelo ABACUS em blocos de *hardware*, os modelos foram tirados da tese de MARRANGHELLO [2]. Inicialmente montamos o BCPI (Bloco de Circuito do Processo Inicial) que calcula os valores de corrente e tensão iniciais que serão utilizadas pelos modelos dos elementos de circuito, para o cálculo do elemento escolhido. E em seguida montamos separadamente a arquitetura de cada elemento de circuito, onde no decorrer deste trabalho faremos à união destes modelos em uma única arquitetura reconfigurável.

3.1 – Bloco de circuito do processo inicial

O Bloco de Circuito do Processo Inicial (BCPI) realiza os cálculos iniciais de corrente e tensão que estão entrando no MPH através de seus processadores vizinhos (Norte, Sul Leste e Oeste). Estes cálculos iniciais serão utilizados pelos modelos de elementos de circuito, para o cálculo do elemento que foi escolhido para o MPH.

O BCPI, que é calculado da seguinte maneira, no pólo "a" calculamos ia'que é a soma de i0 e i1, e va' a soma de v0 e v1 dividido por dois, no pólo "b" é a mesma coisa, ib' é a soma de i2 e i3 e vb' a soma de v2 e v3 dividido por dois, temos a média Iab' que é a soma de ia' e ib' dividido por dois e Vab' que é a diferença entre va' e vb'.

Na Fig. 3.1 apresenta-se o BCPI. Ele é composto por seis blocos de somador/subtrator e três deslocadores. Esta configuração nos possibilita executar os cálculos necessários, que são:

ia'=i0+i1 va'=(v0+v1)/2
ib'=i2+i3 vb'=(v2+v3)/2
Iab'=(Ia'+Ib')/2 Vab'=va'-vb'

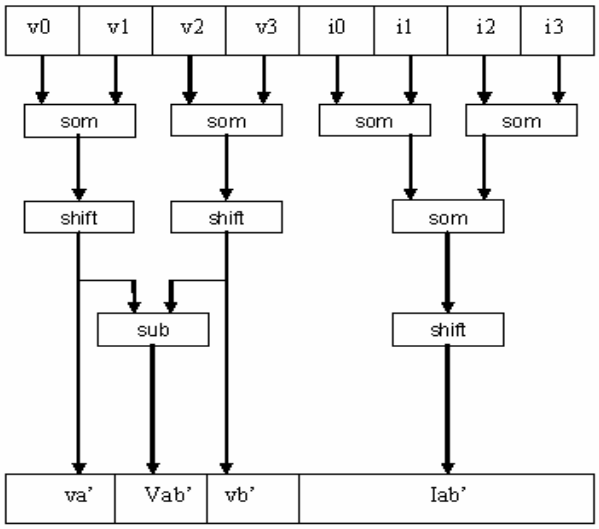


Figura 3.1 - Bloco de Circuito do Processo Inicial.

3.2 – Bloco de circuito do resistor.

Na Fig 3.2 apresenta-se o diagrama do resistor descrito em blocos de *hardware* pronto para ser implementado em VHDL. A sua estrutura é composta pelo BCPI, um bloco divisor, dois blocos de multiplicadores, três blocos de somadores, dois blocos de deslocadores, dois testes de zero, uma porta AND e dois multiplexadores. Os cálculos realizados por este circuito são:

Vab''= R*Iab'
Iab''= Vab'/R
Vab = (Vab'+ Vab'')/2
Iab = (Iab' + Iab'')/2
se va' = 0 então vb = -Vab
senão se vb' = 0 então va = Vab
senão se va'≠0 e vb'≠0
então va = vb + Vab e vb = vb'

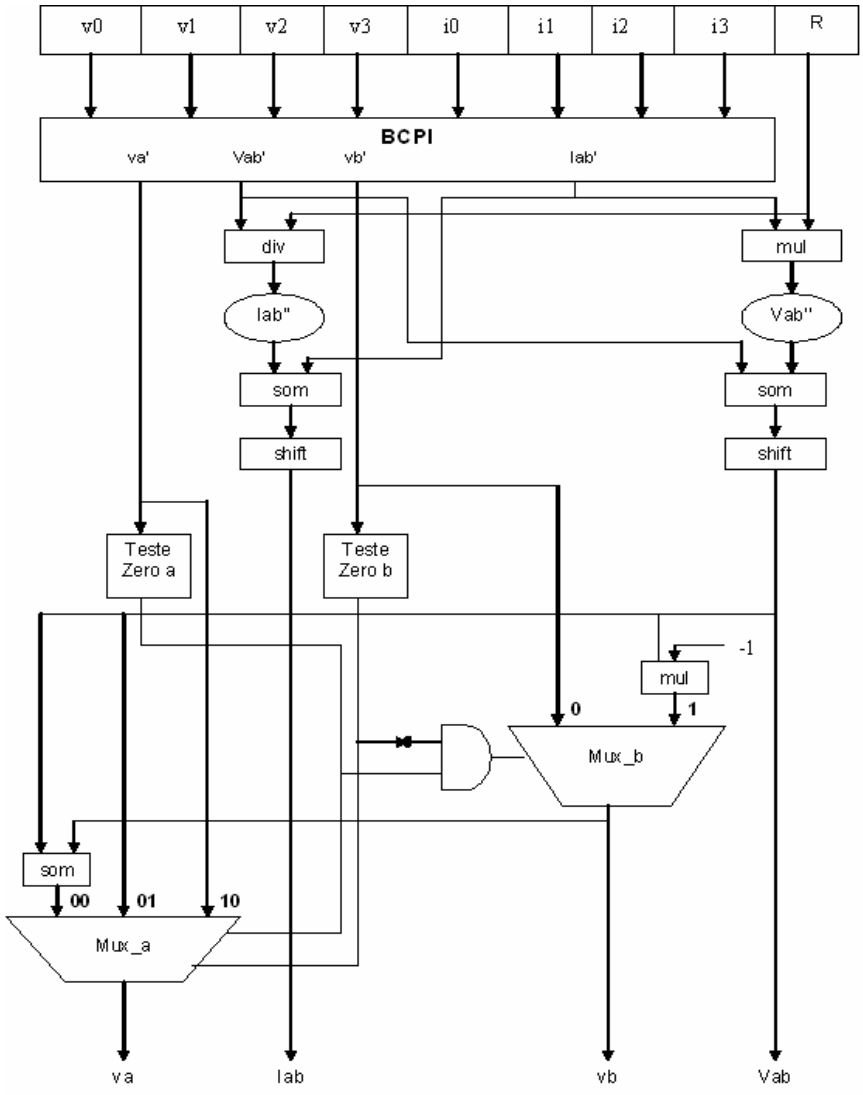


Figura 3.2- Resistor.

3.3- Bloco de circuito do capacitor.

Na Fig.3.3 apresenta-se o capacitor, também descrito em blocos de *hardware*. Faz parte de sua estrutura o capacitor, sete blocos de multiplicação, dois blocos de divisão, dois blocos de soma, um bloco de subtração, dois testes de zero, uma porta AND e dois multiplexadores. Os cálculos realizados por este circuito são:

$I_{ab} = I_0 \, e^{-(t-\Delta t)/\tau}$
$V_{ab} = V_{ab}'' + V_{ab}'''$
$V_{ab}'' = V_f \, (1-e^{-(t-\Delta t)/\tau})$
$V_{ab}''' = V_{carga} \, e^{-(t-\Delta t)/\tau}$
$\tau = V_f \, C / I_0$
se va' = 0 então vb = -Vab
senão se vb' = 0 então va = Vab
senão se va'≠0 e vb'≠0
então va = vb + Vab e vb = vb'

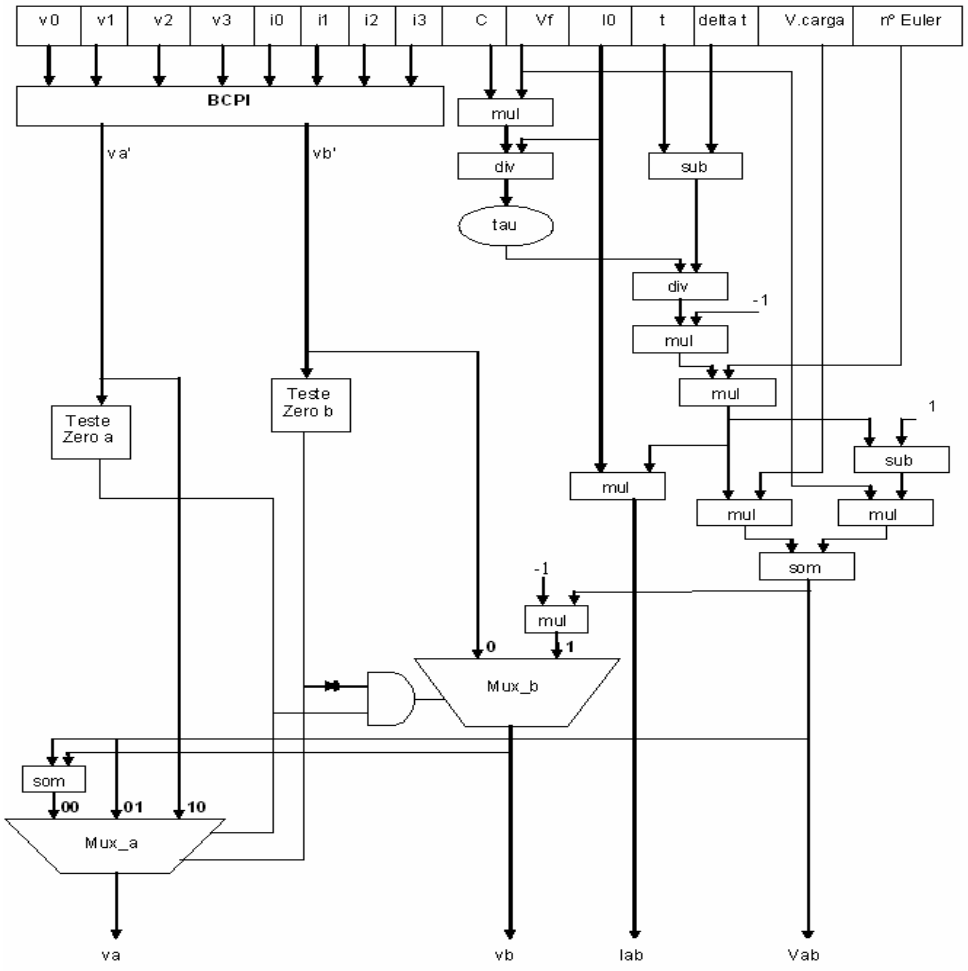


Figura 3.3- Capacitor.

3.4- Bloco de circuito do diodo.

Na Fig. 3.4 apresenta-se a construção em blocos de *hardware d*o modelo do diodo, no qual além dos blocos de *hardware* do BCPI, utilizamos dois divisores, seis multiplicadores, cinco somador/subtrator, dois testes de zero, uma tabela de logaritmo e uma tabela de exponencial. Os cálculos realizados para este circuito são:

Iab'> 0 e Vab'> 0 então

Vab = vt . log (Iab'/Imax.Isat +1)
Iab = Iab'
Iab'< 0 e Vab'< 0 então
Vab = -vt . log (fabs(Iab'/Imax.Isat +1))
Iab = -Imax . Isat + (Vab . e-12)
Iab'= 0 e Vab'= 0,65 então
Iab = Imax.Isat.(exp(Vab'/vt)-1) + (Vab'. e-12)

se va' = 0 então vb = -Vab
senão se vb' = 0 então va = Vab
senão se va'≠0 e vb'≠0
então va = vb + Vab e vb = vb'

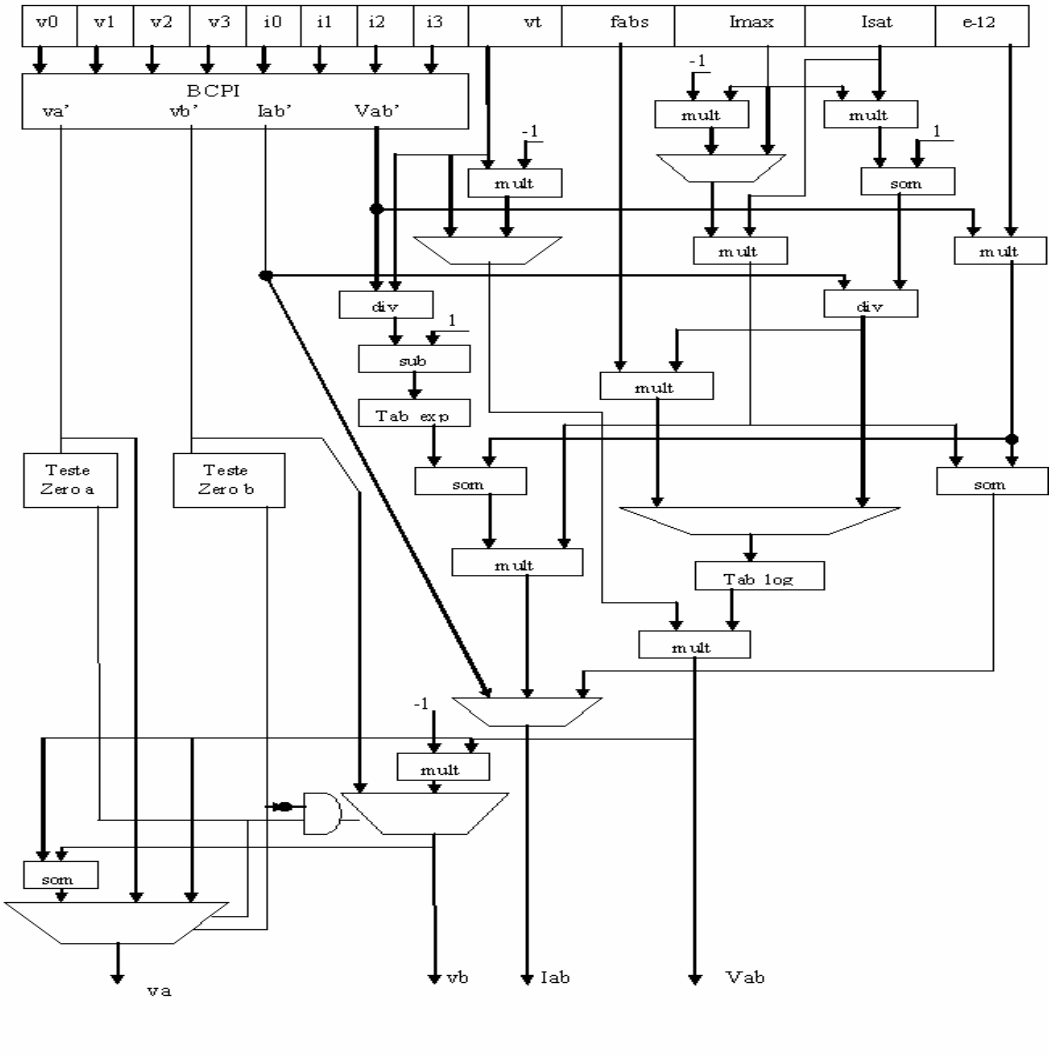


Figura 3.4 - Diodo

3.5 - Bloco de circuito da fonte de tensão.

Na Fig.3.5 apresenta-se a construção em blocos de *hardware d*o modelo da fonte de tensão, no qual, além dos blocos de *hardware* do BCPI, utilizamos dois multiplicadores, dois somadores, dois testes de zero, uma tabela de co-seno e dois multiplexadores. Os cálculos realizados para este circuito são:

$$Vab = V\max.\cos(2\pi.ft)$$

va'=0 então vb=Vab

vb'=0 então va=Vab

va' e vb' forem $\neq$ de 0 então va = vb+Vab

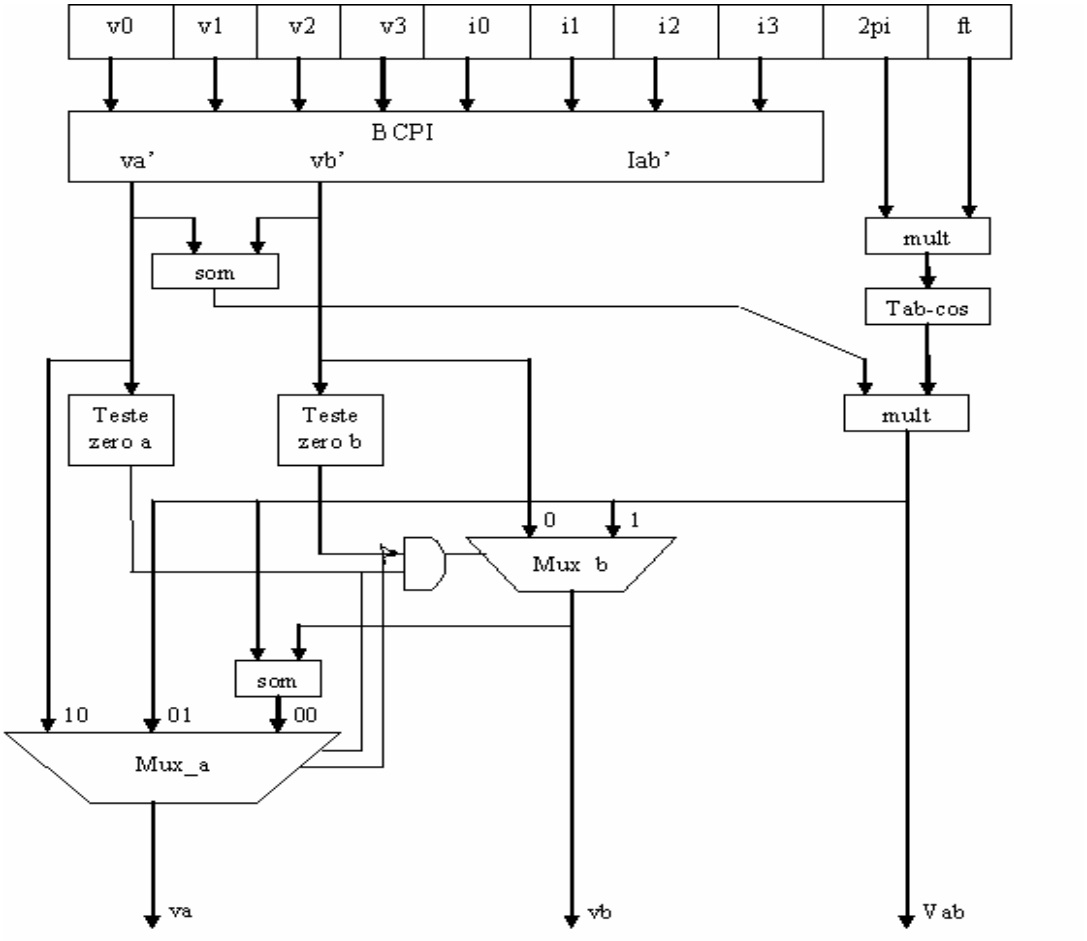


Figura 3.5 - Fonte de Tensão.

3.6 - Bloco de circuito da fonte de corrente.

Na Fig.3.6 apresenta-se à construção em blocos de *hardware d*o modelo da fonte de corrente, na qual, além dos blocos de *hardware* do BCPI, utilizamos dois multiplicadores, um somador e uma tabela de co-seno Os cálculos realizados para este circuito são:

$$Iab = \mathrm{Im}\, ax.\cos(2\pi.ft)$$

va=va'

vb=vb'

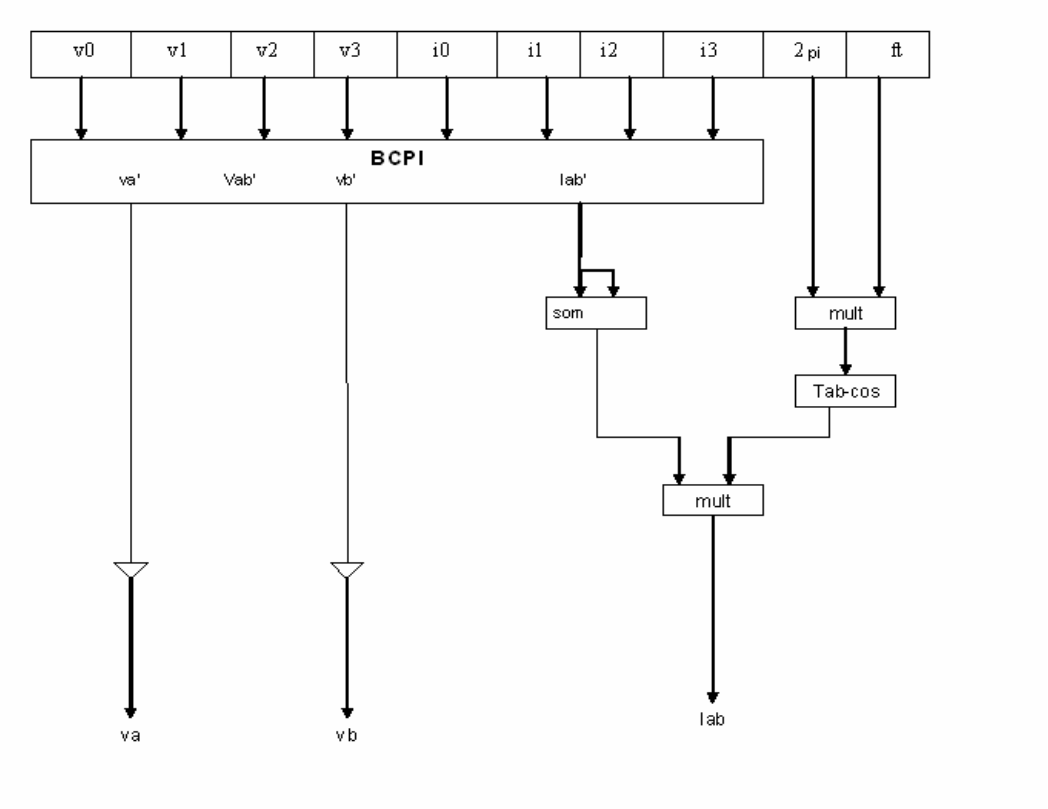


Figura 3.6 - Fonte de Corrente

Capítulo IV – Desenvolvimento dos Blocos em Ponto Flutuante.

4.0 – Introdução

Devido à ampla escolha possível para o tamanho dos operandos de ponto flutuante e conseqüente obtenção de resultados muito extensos aliada à adequação da representação binária de operandos e por razões comerciais, foram padronizadas normas que permitiram uma maior homogeneidade entre os diversos fabricantes de circuitos integrados. Surgindo assim a IEEE/ANSI (*Institute of Electrical and Electronic Engineers/ American National Standards Insitute*) *Standard* 754 que normaliza projetos de CIs .

Para implementar operandos matemáticos em ponto flutuante segundo o padrão ANSI/IEEE754 [7], deve-se escolher inicialmente um dos formatos padronizados de representação dos valores numéricos: formatos básico ou estendido de precisão simples ou dupla. Neste trabalho foi adotado o formato básico de precisão simples. Este formato adota a representação dos operandos com 32 *bits*, distribuídos em três campos: 1 *bit* para sinal (s), 8 *bits* para a parte exponencial (e) e 23 *bits* para a mantissa ou parte fracionária (f), como mostra abaixo a fig.4.0.

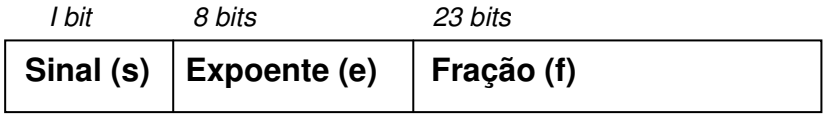


Figura 4.0 – Formato para números em ponto flutuante.

Nos blocos de ponto flutuante desenvolvidos foi preciso renomear todas as entidades internas, por estarem dando conflitos na hora da compilação do elemento escolhido. Para garantir o funcionamento correto de cada bloco dentro da estrutura foram necessários alguns acréscimos como mostrado no restante deste capítulo.

4.1- Somador e subtrator de ponto flutuante.

O somador/subtrator que foi utilizado neste trabalho foi desenvolvido por OLIVEIRA [4]; é um circuito digital seqüencial capaz de somar e subtrair números de ponto flutuante, compostos por 32 bits e representados no formato básico de precisão simples segundo o padrão do IEEE754. Sua arquitetura foi descrita de forma modular, possuindo uma unidade de controle por *hardware*, dois somadores/subtratores de ponto fixo, uma unidade de deslocamento, alguns registradores para o armazenamento dos operandos e de resultados intermediários, entre outros componentes digitais clássicos, como por exemplo, multiplexadores, decodificadores e comparadores. Neste trabalho também foi implementada a técnica de arredondamento para a incorporação dos bits excedentes gerados durante o processo de execução de uma determinada operação.

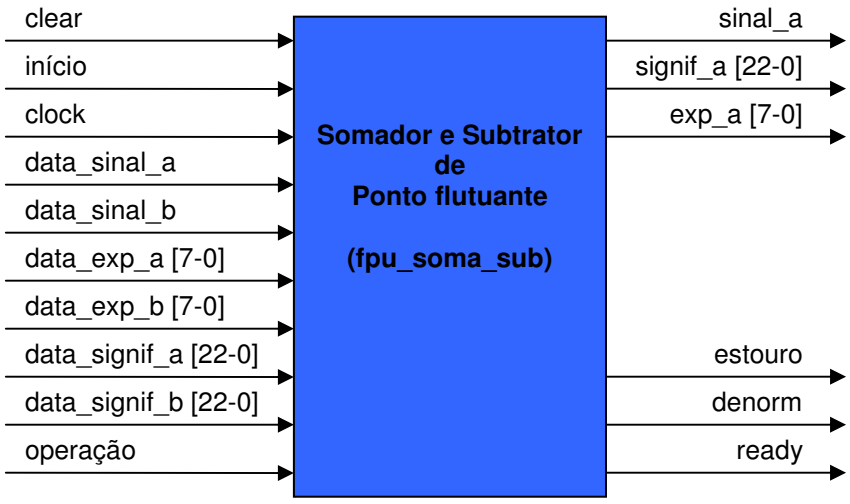


Figura 4.1 – Somador /subtrator de ponto flutuante.

Na Fig. 4.1 vemos o somador/subtrator de ponto flutuante com seus sinais de entrada: `clear`, `início`, `clock`, `data_sinal_a`, `data_signif_a [22 - 0]`, `data_exp_a [7 - 0]`, `operação`, `data_sinal_b`, `data_signif_b [22 - 0]`, `data_exp_b [7 - 0]` e seus sinais de saída: `sinal_a`, `signif_a [22 -0]`, `exp_a[7 -0]`, `estouro` (este sinal sinaliza através do valor lógico 1, a ocorrência de estouro, ou seja, o resultado da operação ultrapassou o número de ponto flutuante máximo responsável pelo sistema), `denorm`(que sinaliza para o usuário, através do nível lógico alto, que o resultado é um número de ponto flutuante denormalizado) e `ready`.(quando passa para

o nível lógico 1 indica que a operação de soma ou subtração requisitada já foi realizada, sendo que o resultado se encontra disponível nos sinais de saída.

Para este somador/subtrator, funcionar corretamente como um módulo de nosso circuito foi preciso acrescentar alguns registradores, como mostramos na Fig. 4.2, que funcionam da seguinte maneira: na primeira subida do relógio, o `reg_clear` deverá estar com o sinal `clear` no nível lógico 1, garantindo com isso a inicialização do circuito. Após a subida do relógio e durante toda a execução da operação, o nível lógico de `clear` deverá permanecer em 0.

Antes da segunda subida do relógio, deve-se disponibilizar os operandos e a operação que são os sinais de entrada descritos na Fig.4.1. Na segunda subida do relógio, o `reg_inicio` envia para o FPU o sinal de início com o nível lógico 1, indicando ao sistema que o processamento da operação especificada pode ser iniciada. Após a subida do segundo relógio, o sinal do `reg_inicio` deverá retornar para o nível lógico 0. Depois de algumas subidas do relógio que variam de acordo com os sinais de entrada, o nível lógico do sinal "`ready`" passará de 0 para 1, sinalizando para o `acionamento_fpu`, que ele deve mandar para o `reg_fim` o sinal de finalizado e que os resultados da operação desejada se encontram disponíveis nos sinais de saída especificados na Fig.4.1. Foram feitas várias simulações garantindo o correto funcionamento do módulo, as quais podem ser vistas no anexo I.

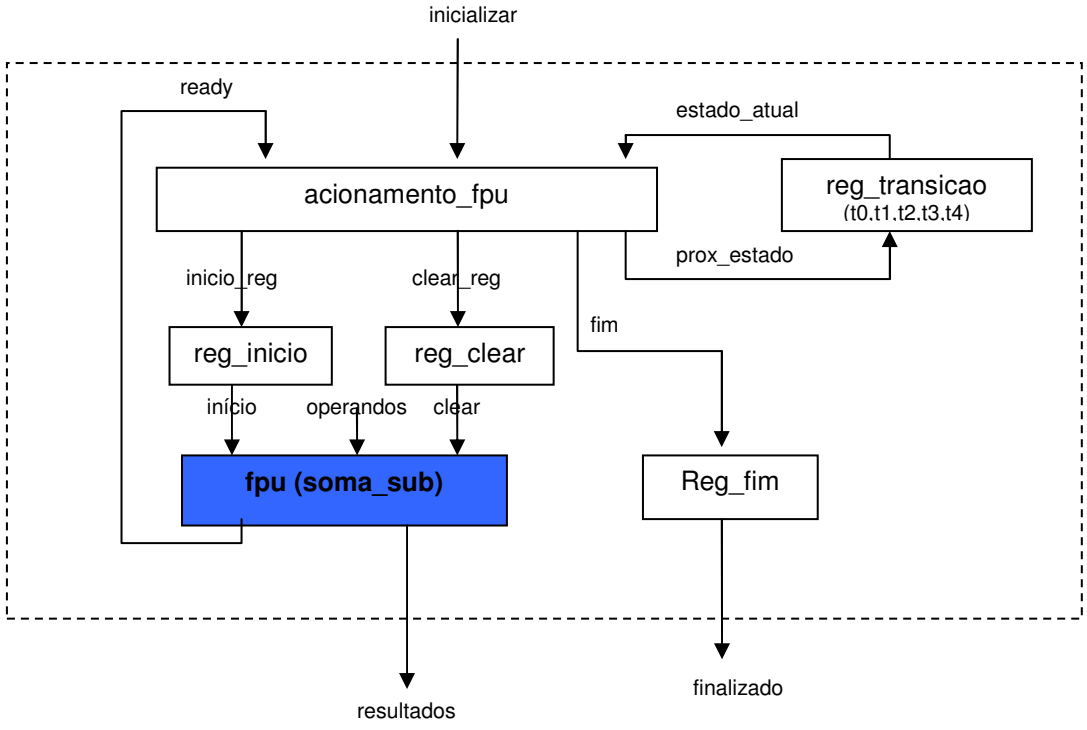


Figura 4.2 – Funcionamento do somador/subtrador.

4.2- Multiplicador de ponto flutuante

O multiplicador que foi utilizado neste trabalho foi desenvolvido por PAIVA [5]; é um circuito digital seqüencial capaz de multiplicar números de ponto flutuante compostos por 32 bits e representados no formato básico de precisão simples segundo o padrão do IEEE754. Sua arquitetura foi descrita em módulos, possuindo uma unidade de controle por *hardware*, um multiplicador de ponto fixo, uma unidade de pós_normalização, um somador de ponto fixo e alguns registradores para o armazenamento dos operandos e dos resultados intermediários, entre outros componentes digitais comuns. Neste projeto foi utilizado o algoritmo básico de multiplicação de ponto flutuante (admitindo-se que os operandos são números normalizados representados no formato básico de precisão simples segundo o padrão IEEE754), que consiste em multiplicar os significandos, somar os expoentes e verificar a ocorrência de *overflow* no expoente resultante; caso o produto dos significandos seja maior ou igual a 2, deslocá-lo uma posição para a direita e incrementar o expoente por 1. Por último, deve-se subtrair da soma dos expoentes o valor 127 referente ao *bias*, utilizado para normalizar o expoente. Na Fig 4.3 temos o multiplicador de ponto flutuante com seus sinais de entrada: `reset, clock, sinal_a, mantissa_a [22 -0], expoente_a [7 -0], sinal_b, mantissa_b [22 -0], expoente_b [7 -0]` e seus sinais de saída: `sinal_r, mantissa_r [22 - 0], expoente_r [7 - 0], overflow_expoente` (sinal que indica se houve *overflow* no cálculo do expoente resultante) `e pronto` (quando igual a 1, indica que o processo de multiplicação de ponto flutuante foi finalizada).

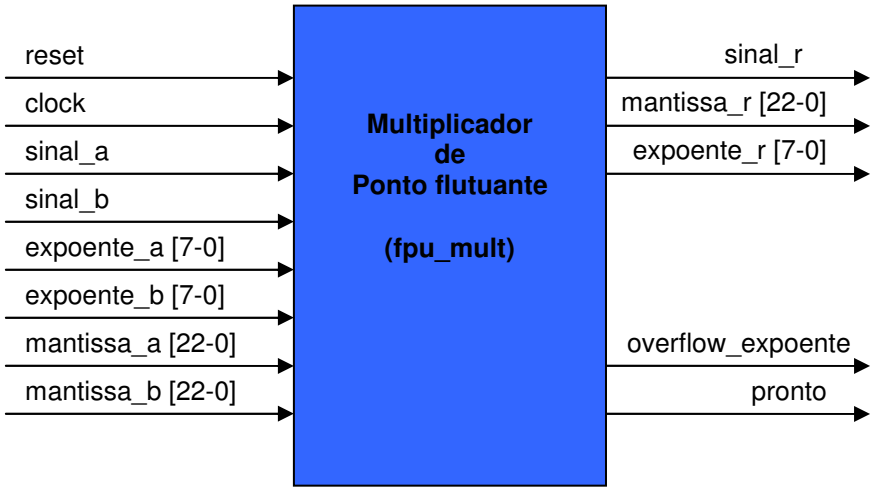


Figura 4.3 – Multiplicador de ponto flutuante.

Para este multiplicador, funcionar corretamente como um módulo do nosso circuito foi preciso acrescentar alguns registradores, como mostramos na Fig. 4.4, que funcionam da seguinte maneira: na primeira subida do relógio, o `reg_reset1` deverá estar com o sinal `reset` no nível lógico 1, garantindo com isso a inicialização do circuito. Após a subida do relógio e durante toda a execução da operação, o nível lógico de `reset` deverá permanecer em 0.

Antes da segunda subida do relógio, deve-se disponibilizar os operandos que são os sinais de entrada descritos na figura acima. Depois de algumas subidas do relógio que variam de acordo com os sinais de entrada, o nível lógico do sinal "`pronto`", passará de 0 para 1, sinalizando para o `acionamento_fpu_mult`, que ele deve mandar para o `reg_fim_mult` o sinal de `finalizado` e que os resultados da operação desejada se encontram disponíveis nos sinais de saída especificados na Fig. 4.3; foram feitas várias simulações garantindo o correto funcionamento do módulo, estas simulações podem ser vistas no anexo I.

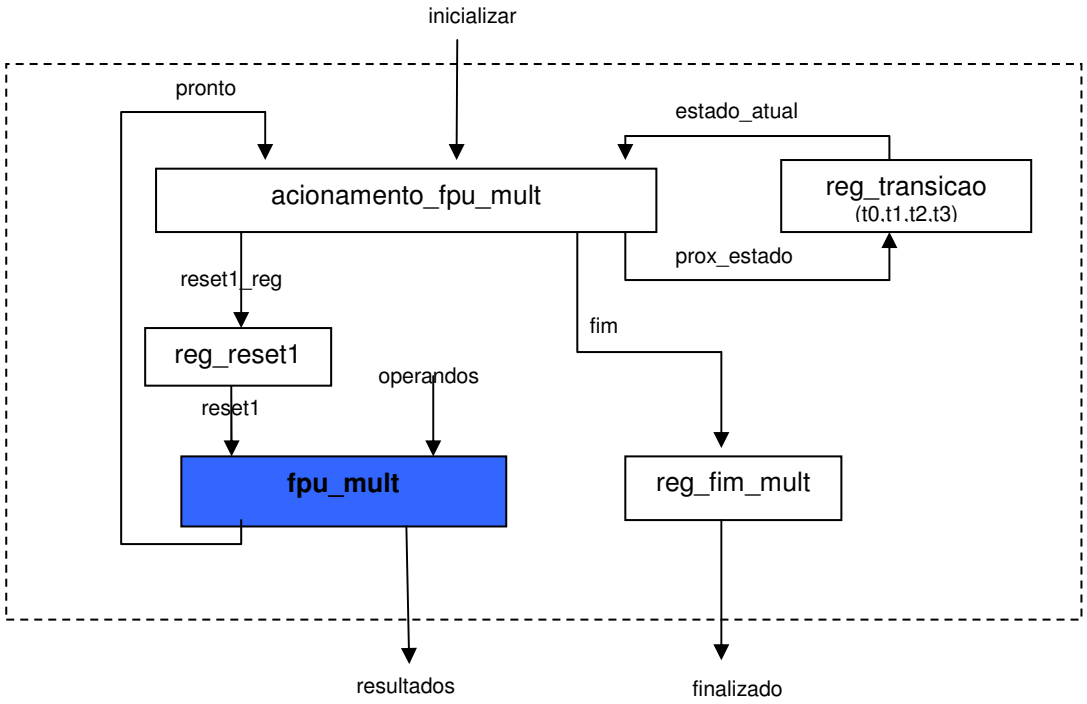


Figura 4.4 – Funcionamento da multiplicação

4.3- Divisor de ponto flutuante.

O divisor que foi utilizado neste trabalho foi desenvolvido por ASSONI [6], é um circuito digital seqüencial capaz de dividir números de ponto flutuante compostos por 32 bits e representados no formato básico de precisão simples segundo o padrão do IEEE754. A

arquitetura foi descrita em módulos, possuindo uma U.C. (unidade de controle) por *hardware*, dois somadores/subtratores de ponto flutuante, uma unidade de deslocamento, uma unidade de pós-normalização e alguns registradores para o armazenamento dos operandos e de resultados intermediários, entre outros componentes digitais. Foi utilizado no divisor o algoritmo de decisão básico, onde primeiro os bits do dividendo são examinados da esquerda para a direita, até o conjunto de *bits* examinados representar um número maior ou igual ao divisor; ou seja referenciado como o divisor sendo capaz de dividir o número. Até este evento ocorrer, zeros são colocados no quociente da esquerda para a direita. Quando um evento ocorre, um 1 é colocado no quociente e o divisor é subtraído do dividendo parcial. O resultado é referenciado como resíduo parcial. A partir deste ponto, a divisão segue um padrão cíclico. A cada ciclo, bits adicionais do dividendo são juntados ao resíduo parcial até o resultado ser maior ou igual ao divisor. Como antes, o divisor é subtraído deste número para produzir um novo resíduo parcial. Este processo continua até todos os *bits* do dividendo forem exauridos. Na Fig. 4.5 temos o divisor de ponto flutuante com suas entradas: `clear`, `início`, `clock`, `data_sinal_a`, `data_signif_a [22 - 0]`, `data_exp_a [7 - 0]`, `data_sinal_b`, `data_signif_b [22 - 0]`, `data_exp_b [7 - 0]` e suas saídas: `ready`, `sinal_quociente`, `expoente_formatado [7 - 0]`, `saída_formatada [46 - 0]`.

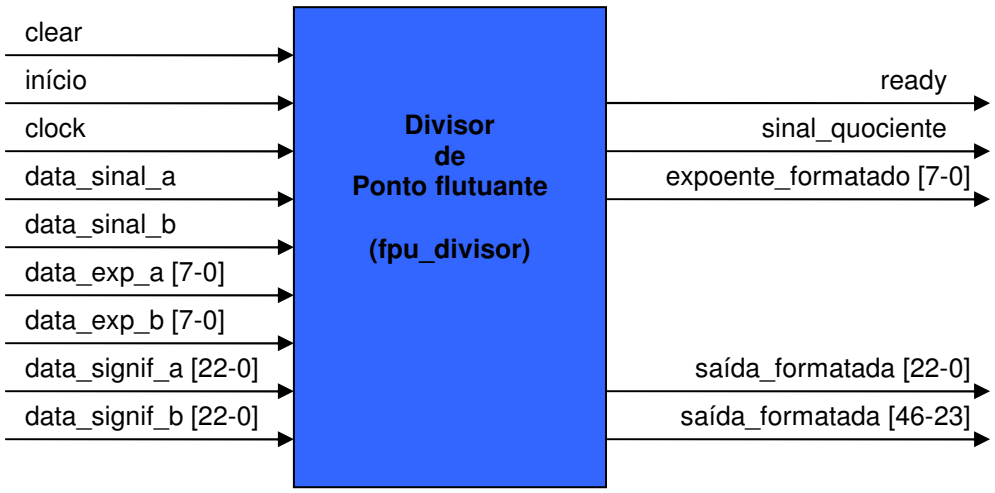


Figura 4.5 – Divisor de ponto flutuante.

Para este divisor, funcionar corretamente como um módulo de nosso circuito foi preciso acrescentar alguns registradores, como mostramos na Fig. 4.6, que funcionam da seguinte maneira: na primeira subida do relógio, o `reg_clear` deverá estar com o sinal

clear no nível lógico 1, garantindo com isso a inicialização do circuito. Após a subida do relógio e durante toda a execução da operação, o nível lógico de clear deverá permanecer em 0.

Antes da segunda subida do relógio, deve-se disponibilizar os operandos que são os sinais de entrada descritos na figura acima. Na segunda subida do relógio, o reg_inicio envia para o FPU_divisor o sinal de início com o nível lógico 1, indicando ao sistema que o processamento da divisão pode ser iniciada. Após a subida do segundo relógio, o sinal do reg_inicio deverá retornar para o nível lógico 0. Depois de algumas subidas do relógio que podem variam de acordo com os sinais de entrada, o nível lógico do sinal de saída "ready" passará de 0 para 1, sinalizando para o acionamento_fpu_divisor, que os resultados da divisão se encontram disponíveis nos sinais de saída especificados na Fig. 4.5; foram feitas várias simulações garantindo o correto funcionamento do módulo, estas simulações podem ser vistas no anexo I.

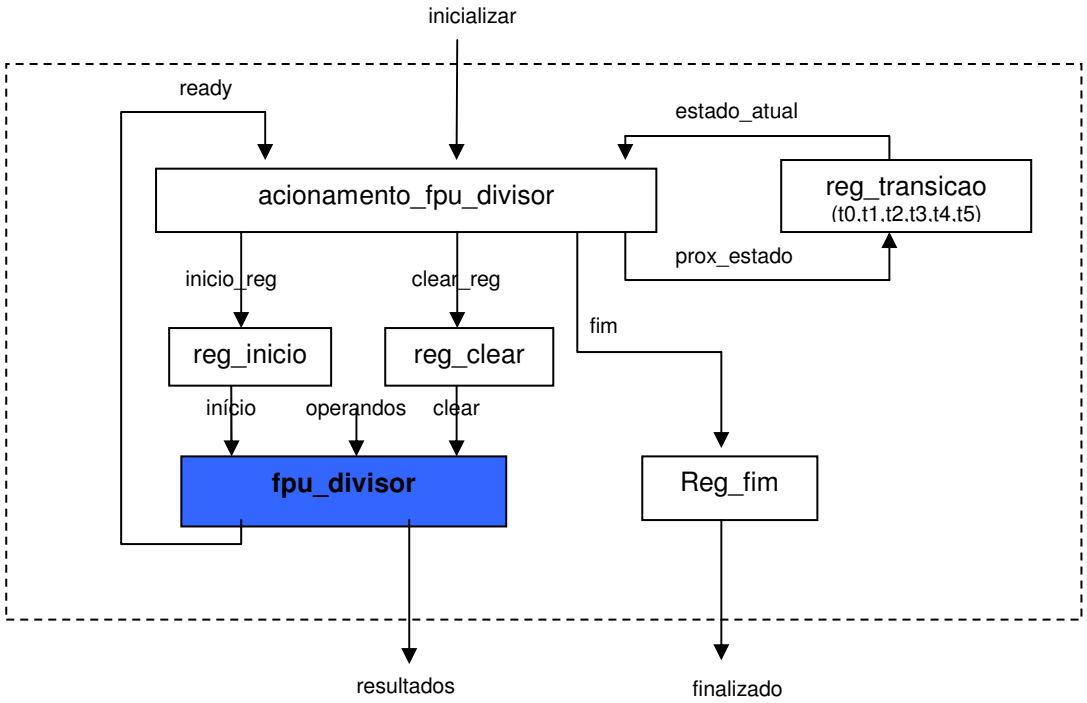


Figura 4.6 – Funcionamento da divisão.

Capítulo V – Definição da Arquitetura Reconfigurável

5.0- Introdução

Pesquisas recentes na área de arquiteturas reconfiguráveis mostram que elas oferecem um desempenho melhor que os processadores de propósito geral (GPPs – *General Purpose Processors*), aliado a uma maior flexibilidade que os ASICs (*Aplication Specific Integrated Circuits*). Uma mesma arquitetura reconfigurável pode ser adaptada para implementar aplicações diferentes, permitindo a especialização do *hardware* de acordo com a demanda computacional de aplicação.

Neste trabalho utilizamos a arquitetura reconfigurável estaticamente, como vimos no Capítulo II.

5.1 – Definição da Arquitetura.

A arquitetura da unidade de processamento é chamada de elemento_circuito, possui os seguintes componentes (todos no formato padrão da IEEE754 como descrito na introdução do Capítulo IV): o Bloco de Circuito do Processo Inicial como vimos no Capítulo III, que possui seis blocos de somador e subtrator de ponto flutuante e três deslocadores, dois blocos de divisão de ponto flutuante, que são chamados de `div_1` e `div_2`, seis blocos de multiplicação de ponto flutuante, que são chamados `de mult_1, mult_2, mult_3, mult_4,mult_5` e `mult_6`, três blocos de soma e subtração de ponto flutuante que chamamos de `som_sub_1, som_sub_2` e `som_sub_3`, dois deslocadores que trabalham com a parte exponencial do número que chamamos de `shift_1` e `shift_2` (utilizamos deslocadores ao invés de blocos de divisão porque a divisão a ser feita é por dois, sendo assim o resultado da divisão é o mesmo do deslocamento e o processo é mais rápido). Além disso, encontram-se três tabelas que chamamos de `tabela_log, tabela_exp` e `tabela_cos`, um bloco que soma 1 (incrementador), para o significando do número chamado de `soma_1`, um bloco que subtrai 1 (decrementador) para o significando do número chamado de `subtrai_1`, três blocos que trabalham com o sinal do número, onde se ele é positivo passa a ser negativo e vice versa que chamamos de `mult_menos1_1, mult_menos1_2` e

mult_menos1_3 e dois teste de zero que chamo de teste_zero_a e teste_zero_b. Quanto aos multiplexadores de duas entradas temos: mux1_diodo, mux2_diodo, mux3_diodo, mux2_a, mux2_b, mux2_c, mux2_d, mux2_e, mux2_f , mux2_g, mux2_h, mux2_i, mux2_j, mux2_l, mux_b; de três entradas temos: mux3_a, mux3_b, mux3_c, mux3_d, mux3_e, mux _a e mux4_diodo; de quatro entradas temos: mux4_a, mux4_b, mux4_c e mux4_d; e de cinco entradas temos: mux5_a, mux5_b, mux5_c, mux5_d, mux5_e, mux5_f, mux5_g, mux5_Iab e mux5_Vab.

Para estabelecer os controles nos diversos multiplexadores foi preciso a criação de dois blocos de decodificadores, o controle_sel_diodo que seleciona o tipo do diodo a ser simulado e o controle_sel que seleciona o elemento de circuito que vai ser simulado no momento no multiplexador. Juntos formam o elemento_circuito, cuja arquitetura reconfigurável pode simular tanto um resistor como um capacitor ou um diodo ou uma fonte de tensão ou uma fonte de corrente. Todos estes componentes e o arquivo principal; elemento_circuito, estão listado no anexo I.

Nas figuras que serão mostradas abaixo: Fig. 5.0 (resistor), Fig. 5.1 (capacitor), Fig. 5.2 (diodo), Fig. 5.3 (fonte de tensão) e Fig. 5.4 (fonte de corrente), mostramos apenas a parte da arquitetura que foi preciso para simular cada elemento separadamente, só na Fig. 5.5 que mostramos toda a arquitetura e como exemplo configuramos a arquitetura para simular um diodo, mas mostramos todas as outras configurações possíveis representadas por cores como vemos na legenda da Fig. 5.5.

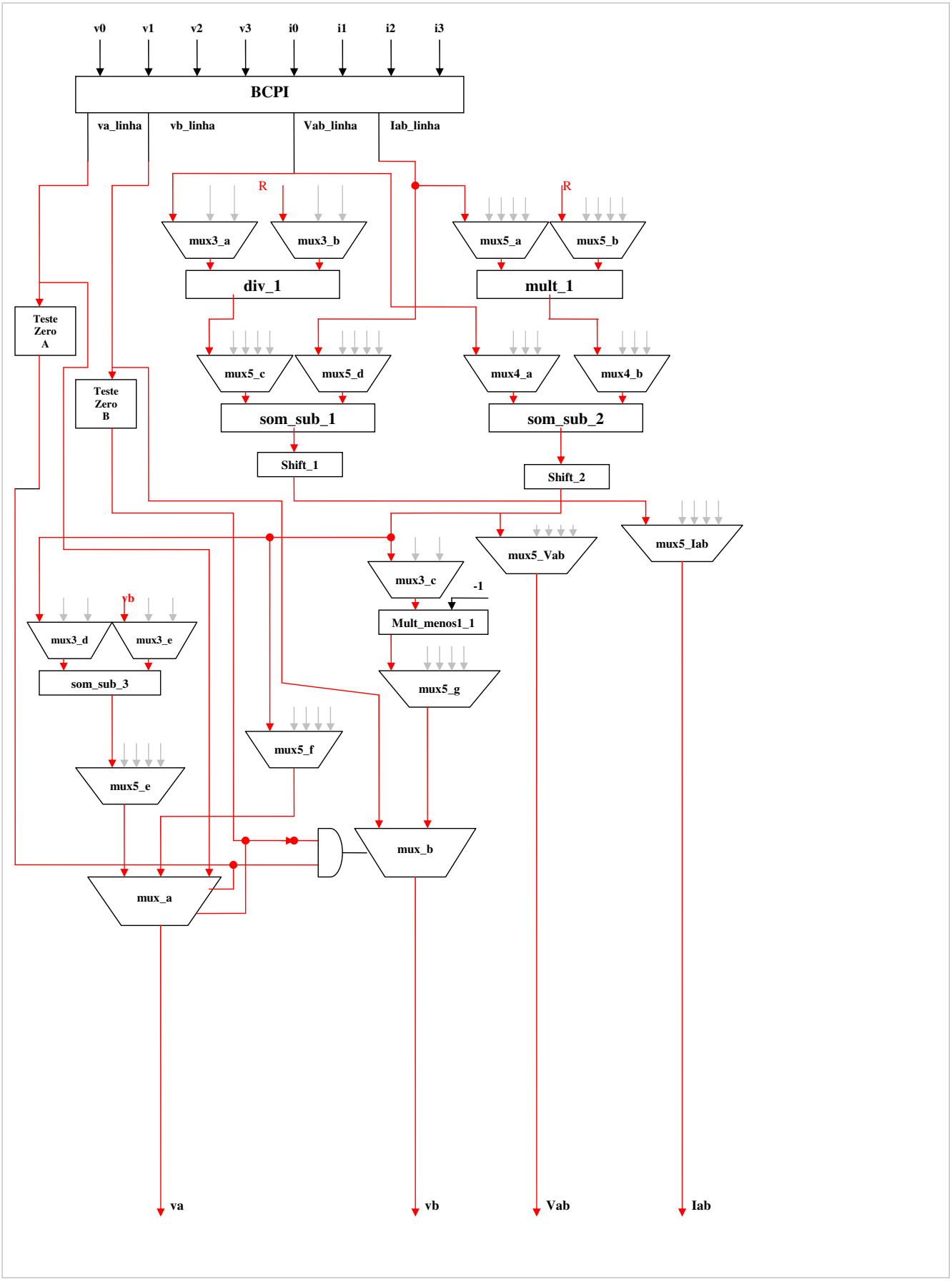


Figura 5.0 – a arquitetura `elemento_circuito` simulando um resistor.

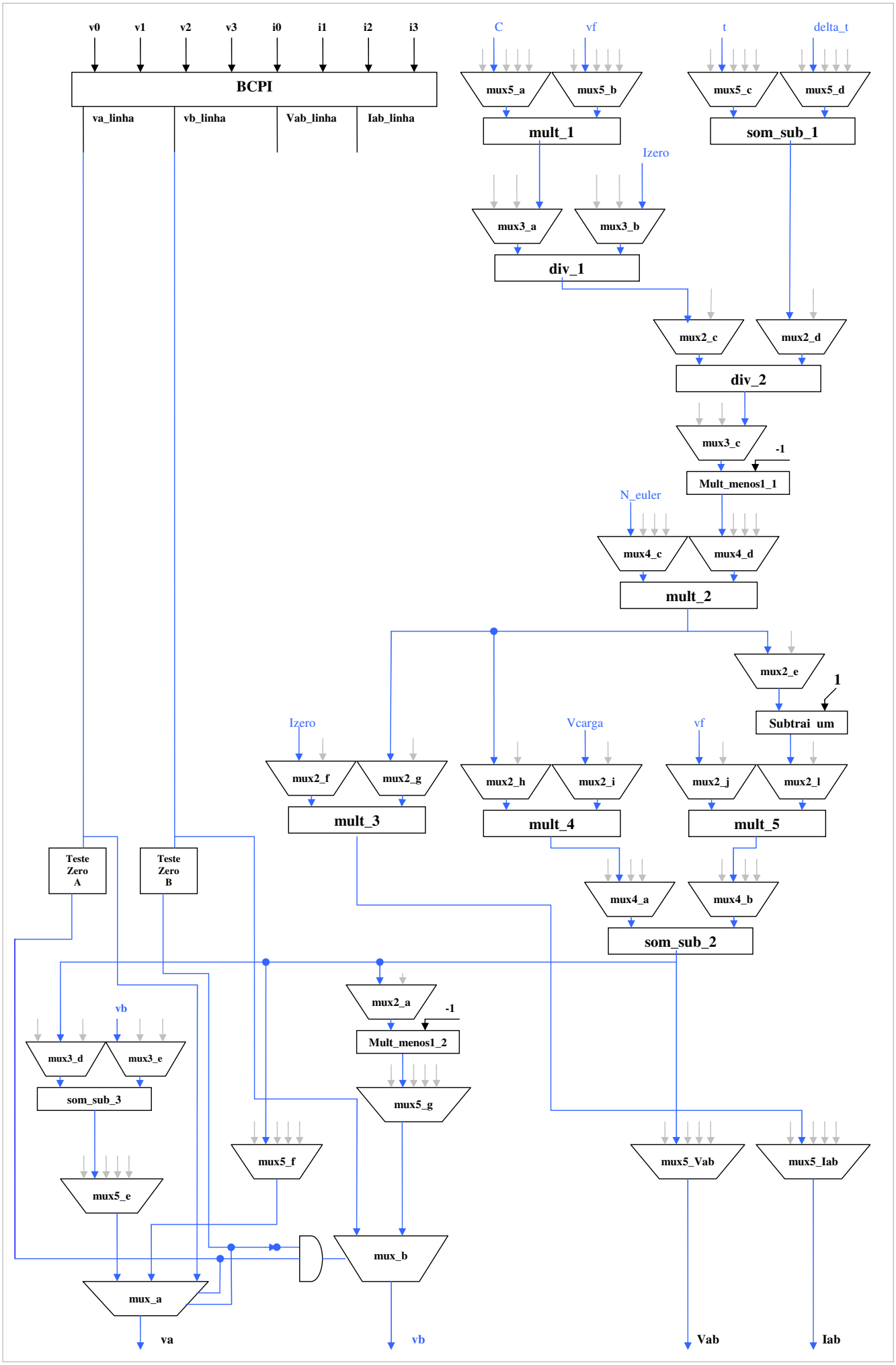


Figura 5.1 – a arquitetura `elemento_circuito` simulando um capacitor.

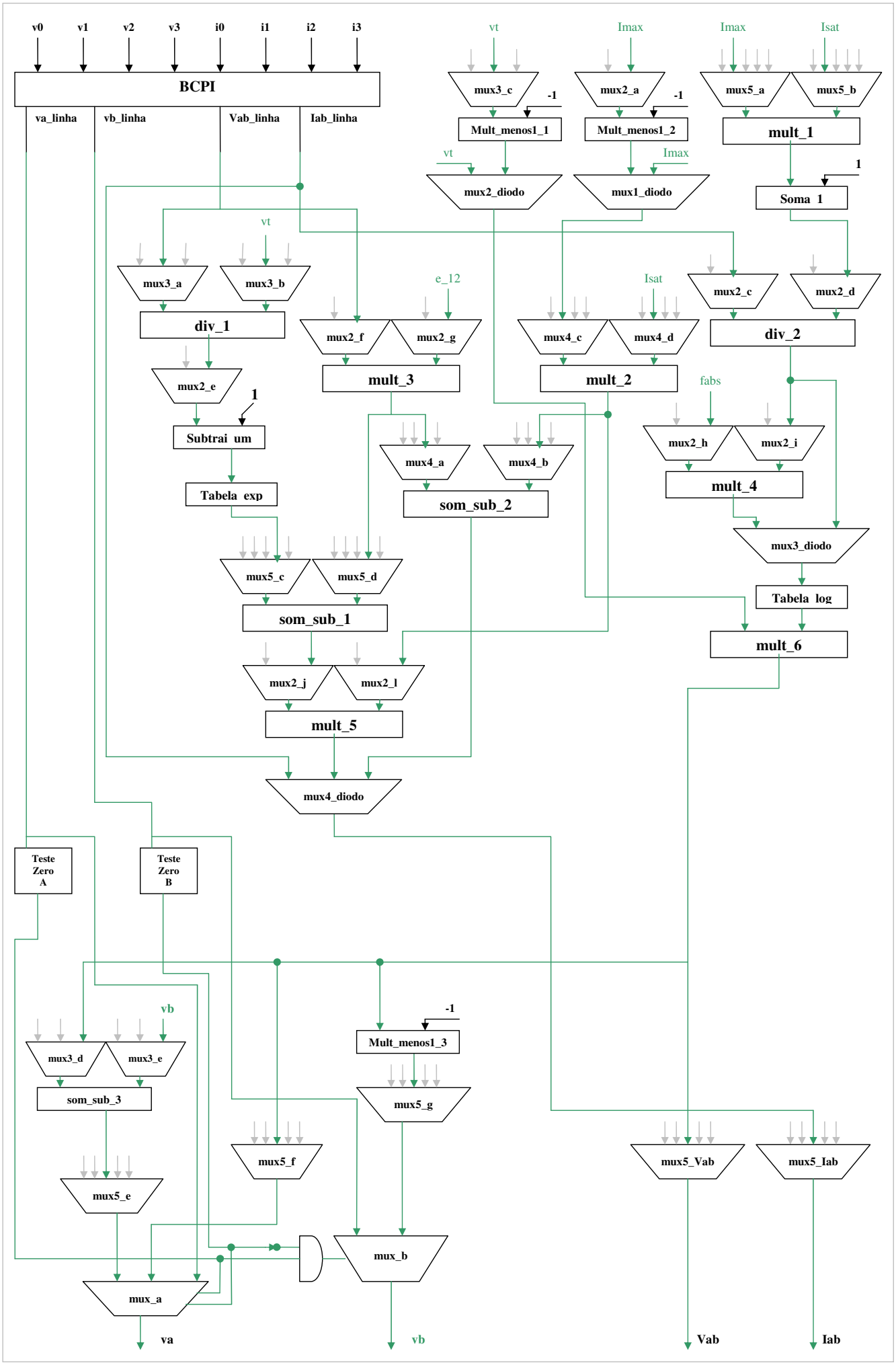


Figura 5.2 – a arquitetura `elemento_circuito` simulando um diodo.

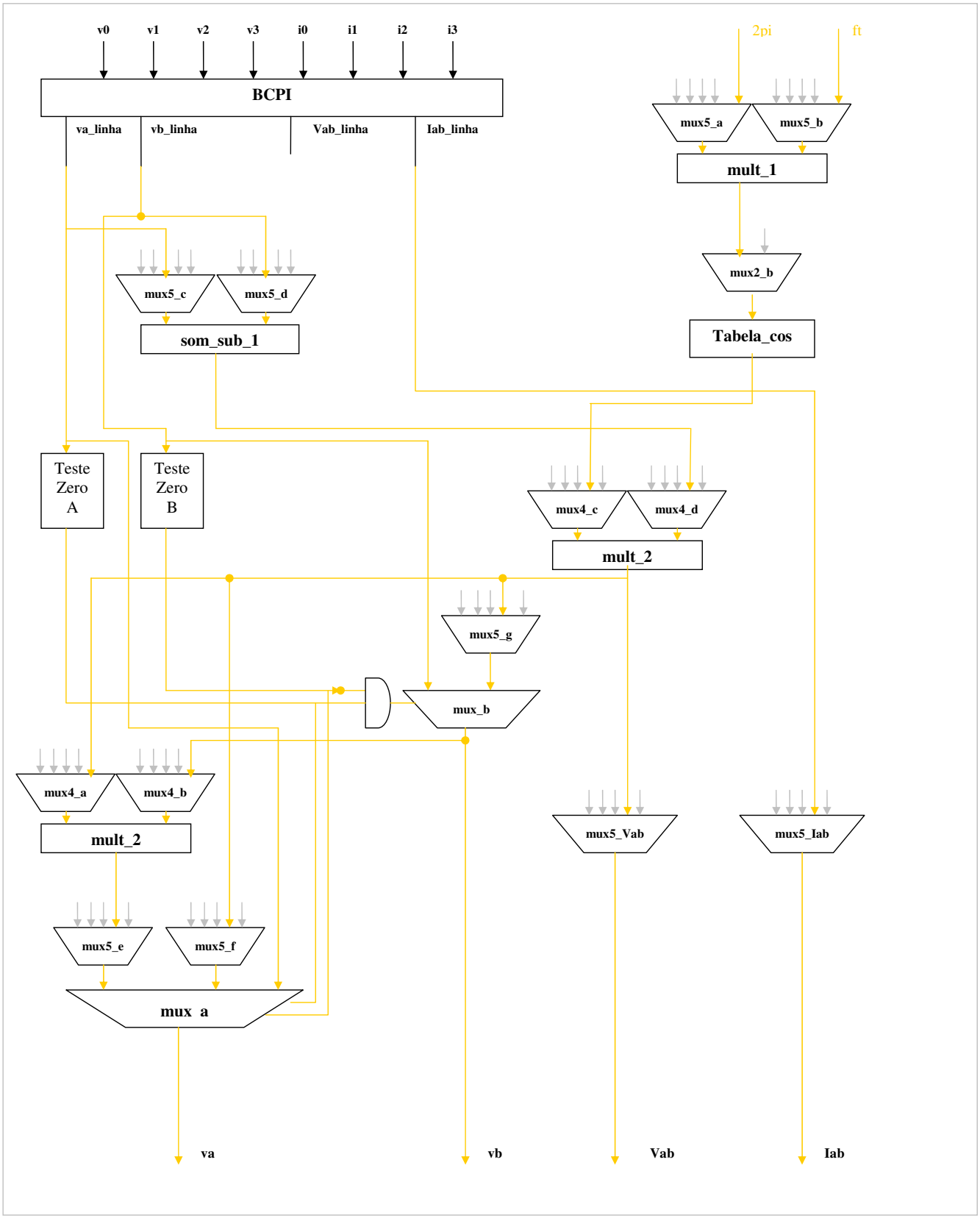


Figura 5.3 – a arquitetura `elemento_circuito` simulando uma fonte de tensão.

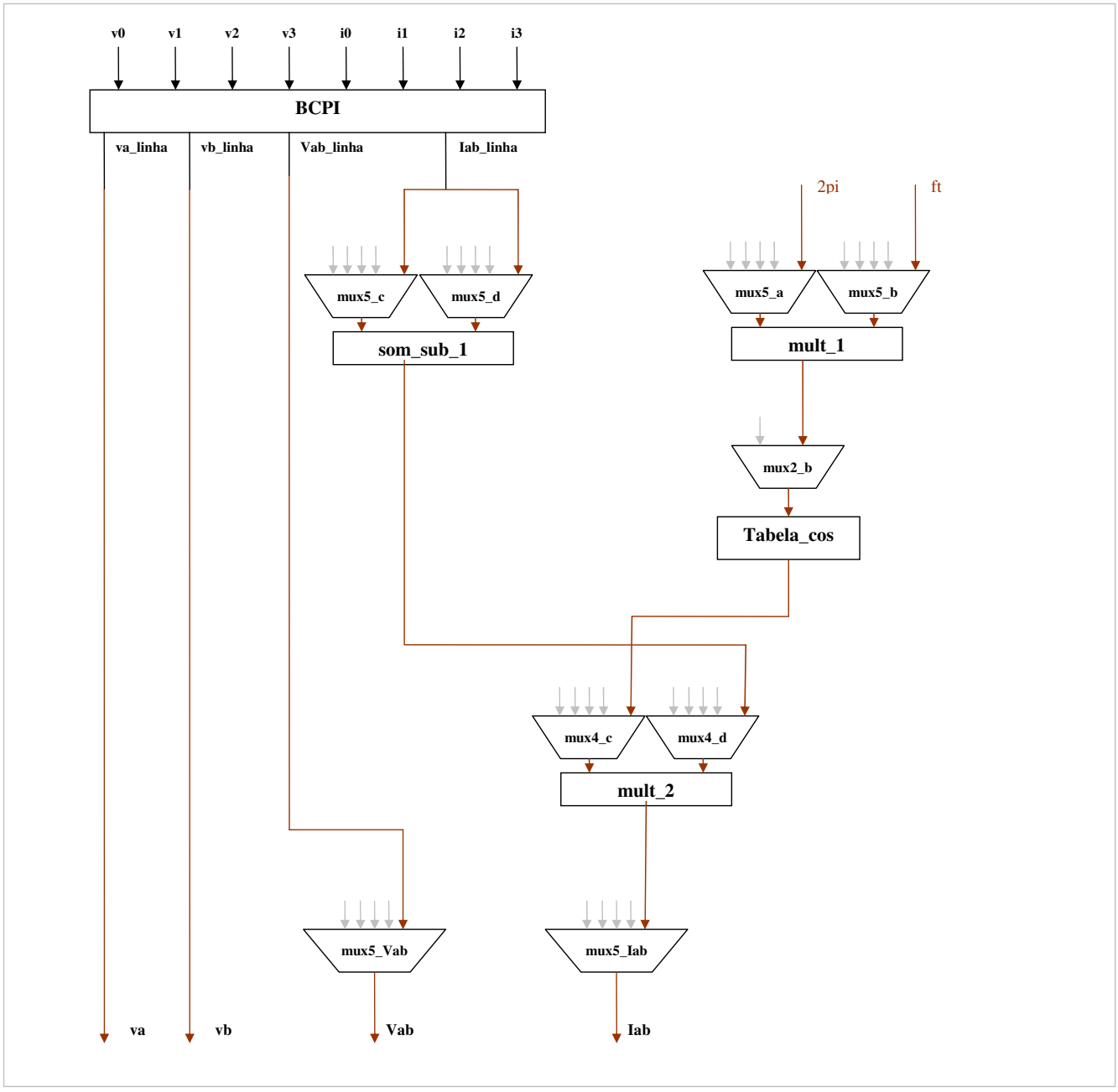
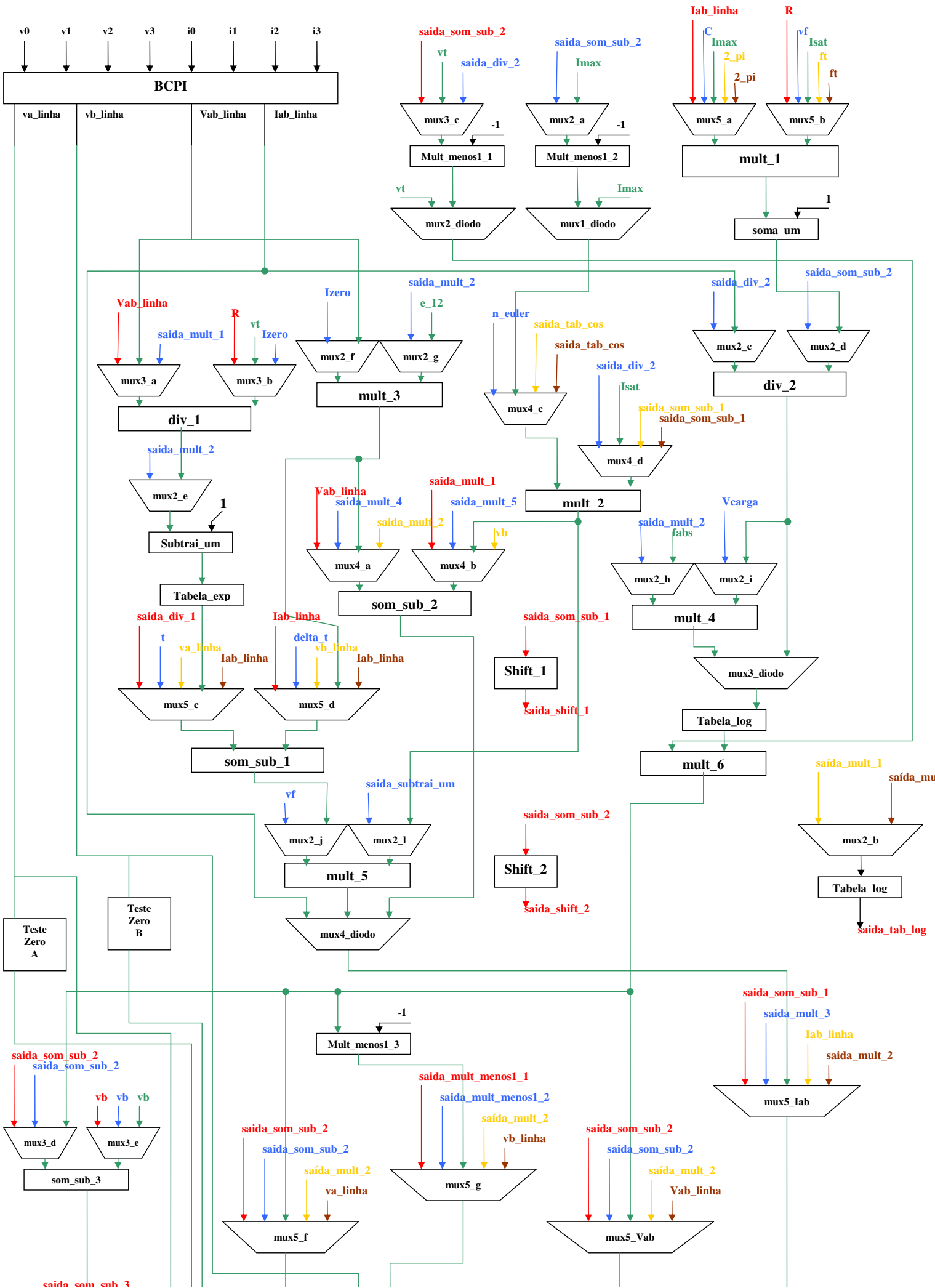


Figura 5.4 – a arquitetura `elemento_circuito` simulando uma fonte de corrente.


v0
v1
v2
v3
i0
i1
i2
i3
BCPI
va_linha
vb_linha
Vab_linha
Iab_linha
saida_som_sub_2
vt
saida_div_2
mux3_c
-1
Mult_menos1_1
saida_som_sub_2
Imax
mux2_a
-1
Mult_menos1_2
Iab_linha
C
Imax
2_pi
2_pi
R
vf
Isat
ft
ft
mux5_a
mux5_b
mult_1
vt
Imax
mux2_diodo
mux1_diodo
1
soma_um
Vab_linha
mux3_a
saida_mult_1
R
vt
Izero
mux3_b
Izero
saida_mult_2
e_12
mux2_f
mux2_g
mult_3
div_1
n_euler
saida_tab_cos
saida_tab_cos
mux4_c
saida_div_2
Isat
saida_som_sub_1
saida_som_sub_1
mux4_d
saida_div_2
saida_som_sub_2
mux2_c
mux2_d
div_2
saida_mult_2
mux2_e
1
Subtrai_um
Tabela_exp
Vab_linha
saida_mult_4
saida_mult_2
saida_mult_1
saida_mult_5
vb
mux4_a
mux4_b
som_sub_2
mult_2
saida_mult_2
fabs
Vcarga
mux2_h
mux2_i
mult_4
saida_div_1
t
va_linha
Iab_linha
Iab_linha
delta_t
vb_linha
Iab_linha
mux5_c
mux5_d
saida_som_sub_1
Shift_1
saida_shift_1
mux3_diodo
Tabela_log
som_sub_1
mult_6
saida_mult_1
saída_mult
vf
saida_subtrai_um
mux2_j
mux2_l
mult_5
saida_som_sub_2
Shift_2
saida_shift_2
mux2_b
Tabela_log
saida_tab_log
mux4_diodo
Teste Zero A
Teste Zero B
-1
Mult_menos1_3
saida_som_sub_1
saida_mult_3
Iab_linha
saida_mult_2
mux5_Iab
saida_som_sub_2
saida_som_sub_2
vb
vb
vb
mux3_d
mux3_e
som_sub_3
saida_mult_menos1_1
saida_mult_menos1_2
saída_mult_2
vb_linha
mux5_g
saida_som_sub_2
saida_som_sub_2
saída_mult_2
va_linha
mux5_f
saida_som_sub_2
saida_som_sub_2
saída_mult_2
Vab_linha
mux5_Vab
saida_som_sub_3

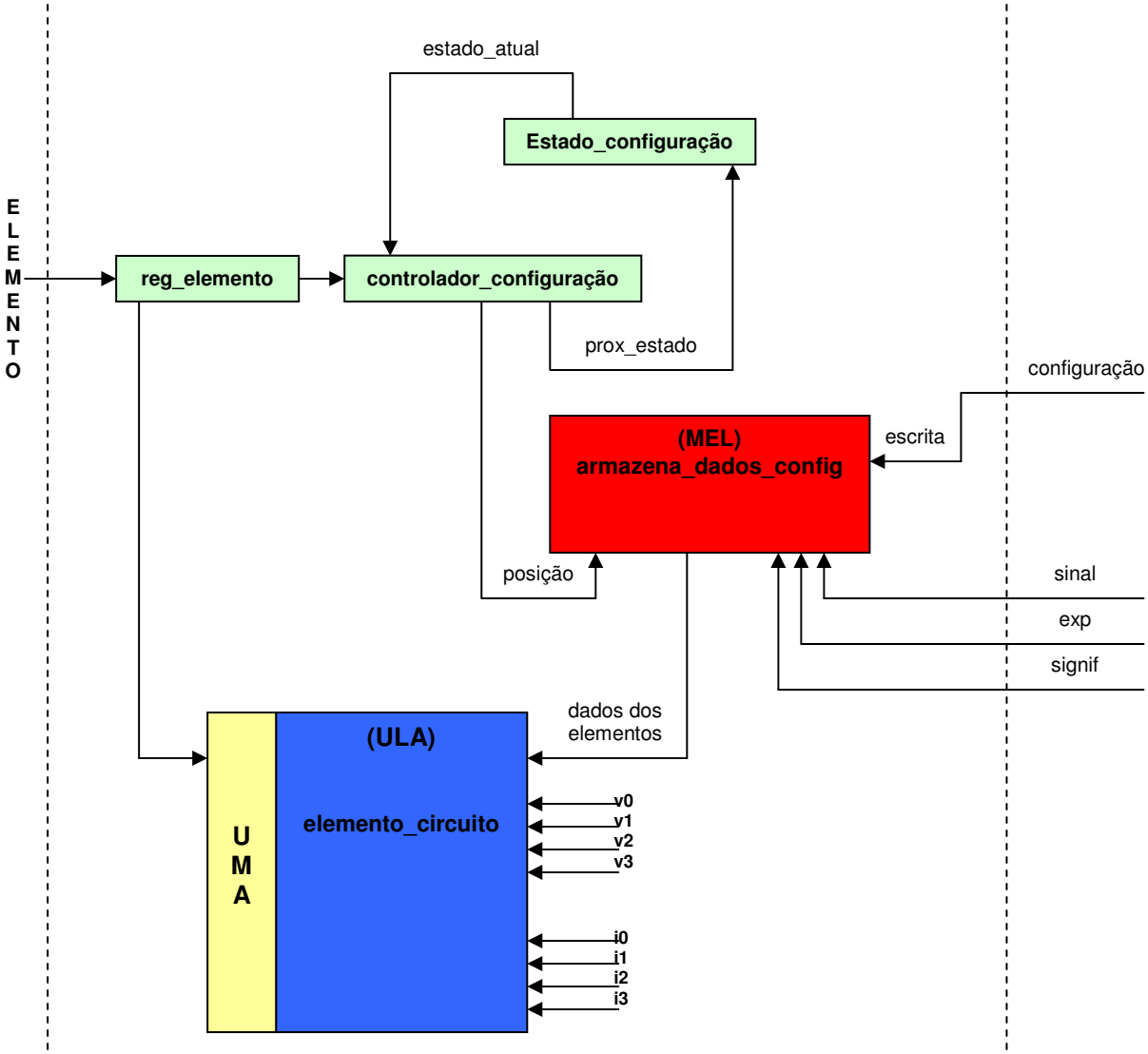


Figura 5.6 – Configuração da unidade `elemento_circuito`.

Temos também como vimos na fig. 5.6 um bloco que se chama `armazena_dados_config` (chamada na arquitetura do MPH de **MEL**) que é uma memória para armazenagem dos dados de configuração dos elementos de circuito que podem ser simulados pelo ABACUS. É esta memória que envia os dados dos elementos para a unidade `elemento_circuito`, de acordo com o elemento a ser simulado no momento ("000" – resistor, "001" – capacitor, "010" – diodo, "011" – fonte de tensão e "100" – fonte de corrente). O `elemento_circuito` é a **ULA**, onde estão armazenados os módulos que serão configurados de acordo com o elemento a ser simulado, também no `elemento_circuito` embutimos a **UMA**, que são os decodificadores responsáveis pela configuração dos módulos para a simulação do elemento escolhido.

Os registradores `reg_elemento`, `controlador_configuração` e o `estado_configuração` fazem o papel da **UCE** (unidade de controle elementar), que controla o processamento do MPH. Esta configuração, e os arquivos.vhd estão listados no anexo I.

Capítulo VI – Resultados da Compilação, Simulação e de Teste Laboratorial da unidade `elemento_circuito`.

6.0 - Introdução

Neste capítulo mostramos os resultados da compilação (que é a verificação de possíveis erros na linguagem de descrição) da simulação (visando comprovar a funcionalidade da arquitetura), e em seguida da sua implementação num FPGA e dos testes realizados para mostrar o funcionamento da arquitetura. Todos estes passos foram realizados com a ajuda do *software* Quartus II.

6.1 - Descrições da compilação do elemento_circuito.

Após a etapa de construção e feita a compilação dos módulos que compõem o `elemento_circuito`, o arquivo principal possui 1620 linhas de código e foi preciso 40 minutos para a sua compilação, o *software* Quartus II gera alguns arquivos indicando em qual família e, mais especificamente, com qual dispositivo a unidade `elemento_circuito` foi configurada. Tais arquivos possuem uma série de outras informações como, por exemplo, a porcentagem de pinos de I/O utilizados, a porcentagem de registradores lógicos dedicados, a família que utilizamos e o dispositivo que escolhemos, etc.

6.2 – Simulação do elemento_circuito

Terminada a fase de compilação executamos uma bateria de simulações visando comprovar a funcionalidade da unidade `elemento_circuito`. O *software* Quartus II permite que simulações funcionais sejam realizadas utilizando-se de um editor de ondas. Neste editor indicamos os valores lógicos dos sinais de entrada enquanto que o *software*, ao reproduzir os sinais de entrada, gera neste mesmo arquivo os valores lógicos dos sinais de saída. Várias simulações foram feitas para garantir o correto funcionamento.

De acordo com o elemento a ser simulado, é incrementado em v0, antes de seu valor real, os valores das variáveis que o elemento escolhido necessita para seu cálculo, estas variáveis estão armazenadas em posições da memória, e quem controla a sua ordem de entrada na simulação é o `controlador_configuração`.

Os arquivos de simulação devem executar um determinado procedimento para o correto funcionamento da arquitetura definida para o `elemento_circuito` que consiste de:

- gerar uma onda quadrada no sinal do relógio;
- na primeira subida do relógio, o sinal `reset` deverá estar no nível lógico 1, garantindo com isso a inicialização do circuito. Antes da segunda subida do relógio e durante toda a execução da operação, o nível lógico de `reset` deve permanecer em 0;
- antes da segunda subida do relógio, o sinal de `configuração` passa para o nível lógico 1, e só volta para o nível lógico 0 quando todos os dados de configuração forem inseridos em v0; dependendo do elemento a ser simulado utilizamos mais ou menos pulsos do relógio;
- o sinal `elemento` informa qual elemento vai ser simulado no momento, se é um resistor ("000"), um capacitor ("001"), um diodo ("010"), uma fonte de tensão ("011") ou uma fonte de corrente ("100");
- o sinal de `inicializar` vai subir quando o sinal de `configuração` passar para o nível lógico 0, é quando também são informados os sinais de entrada(`data_sinal_v0, data_sinal_v1, data_sinal_v2, data_sinal_v3, data_sinal_i0, data_sinal_i1, data_sinal_i2, data_sinal_i3, data_exp_v0, data_exp_v1, data_exp_v2, data_exp_v3, data_exp_i0, data_exp_i1, data_exp_i2, data_exp_i3, data_signif_v0, data_signif_v1, data_signif_v2, data_signif_v3, data_signif_i0, data_signif_i1, data_signif_i2, data_signif_i3`).

.

As simulações foram realizadas em ponto flutuante, colocamos apenas uma simulação de cada elemento de circuito, o restante das tabelas de simulações de cada elemento estão no anexo II.

Tabela 1 – A arquitetura do `elemento_circuito` simulando um resistor.

Entradas do simulador			
elemento	"000"		
	sinal	expoente	significando
(v0)R	0	10000001	01000000000000000000000
v0	0	10000011	01000000000000000000000
v1	0	10000011	10010000000000000000000
v2	0	10000000	00000000000000000000000
v3	0	10000000	10000000000000000000000
i0	0	10000001	00000000000000000000000
i1	0	10000000	00000000000000000000000
i2	0	10000000	00000000000000000000000
i3	0	00000000	00000000000000000000000
Saídas do simulador			
va	0	10000011	01101000000000000000000
vb	0	10000000	01000000000000000000000
Vab	0	10000011	01000000000000000000000
Iab	0	10000001	00000000000000000000000

Tabela 2 – A arquitetura do `elemento_circuito` simulando um capacitor.

Entradas do simulador			
elemento	"001"		
	sinal	expoente	significando
(v0)C	0	00000000	00000000000000000000000
(v0)vf	0	00000000	00000000000000000000000
(v0)Izero	0	00000000	00000000000000000000000
(v0)t	0	00000000	00000000000000000000000
(v0)delta_t	0	00000000	00000000000000000000000
(v0)Vcarga	0	00000000	00000000000000000000000
(v0)n_euler	0	10000000	01011011111100001010001
v0	0	10000000	00000000000000000000000
v1	0	10000001	00000000000000000000000
v2	0	10000000	00000000000000000000000
v3	0	10000000	00000000000000000000000
i0	0	01111111	00000000000000000000000
i1	0	10000000	00000000000000000000000
i2	0	01111111	00000000000000000000000
i3	0	00000000	00000000000000000000000
Saídas do simulador			
va	0	10000000	00000000000000000000000
vb	0	10000000	00000000000000000000000
Vab	0	00000000	00000000000000000000000
Iab	0	00000000	00000000000000000000000

Tabela 3 – A arquitetura do `elemento_circuito` simulando um diodo.

Entradas do simulador			
elemento	"010"		
	sinal	expoente	significando
(v0)vt	0	01111110	00001010001111010111000
(v0)fabs	0	01111100	00110011001100110011010
(v0)Imax	0	10000001	00000000000000000000000
(v0)Isat	0	01111011	11101011100001010001111
(v0)e_12	0	10000000	01011011111100001010001
v0	0	10000000	00000000000000000000000
v1	0	10000001	00000000000000000000000
v2	0	10000000	00000000000000000000000
v3	0	10000000	00000000000000000000000
i0	0	01111111	00000000000000000000000
i1	0	10000000	00000000000000000000000
i2	0	01111111	00000000000000000000000
i3	0	00000000	00000000000000000000000
Saídas do simulador			
va	0	10000000	00101110100001001101000
vb	0	10000000	00000000000000000000000
Vab	0	01111101	01110100001001100111100
Iab	0	10000000	00000000000000000000000

Tabela 4 – A arquitetura do `elemento_circuito` simulando uma fonte de tensão.

Entradas do simulador			
elemento	"011"		
	sinal	expoente	significando
(v0)2_pi	0	10000001	10010010000111111011011
(v0)ft	0	01111110	00000000000000000000000
v0	0	10000000	00000000000000000000000
v1	0	10000001	00000000000000000000000
v2	0	10000000	00000000000000000000000
v3	0	10000000	00000000000000000000000
i0	0	01111111	00000000000000000000000
i1	0	10000000	00000000000000000000000
i2	0	01111111	00000000000000000000000
i3	0	00000000	00000000000000000000000
Saídas do simulador			
va	1	10000000	10000000000000000000000
vb	0	10000000	00000000000000000000000
Vab	1	10000001	01000000000000000000000
Iab	0	10000000	00000000000000000000000

Tabela 5 – A arquitetura de `elemento_circuito` simulando uma fonte de corrente.

Entradas do simulador			
elemento	"100"		
	sinal	expoente	significando
(v0)2_pi	0	10000001	10010010000111111011011
(v0)ft	0	01111110	00000000000000000000000
v0	0	10000000	00000000000000000000000
v1	0	10000001	00000000000000000000000
v2	0	10000000	00000000000000000000000
v3	0	10000000	00000000000000000000000
i0	0	01111111	00000000000000000000000
i1	0	10000000	00000000000000000000000
i2	0	01111111	00000000000000000000000
i3	0	00000000	00000000000000000000000
Saídas do simulador			
va	0	10000000	10000000000000000000000
vb	0	10000000	00000000000000000000000
Vab	0	01111111	00000000000000000000000
Iab	1	10000001	00000000000000000000000

Podemos notar através das tabelas acima que os resultados foram satisfatórios, é só pegarmos as fórmulas de cada elemento de circuito que estão no Capítulo- II e substituir as variáveis das fórmulas pelos valores de entrada das tabelas, fazer os cálculos e veremos que as saídas são as mesmas fornecidas pelo editor de ondas. Um exemplo foi montado e está no anexo III. As simulações indicam um correto funcionamento da arquitetura reconfigurável aqui proposta. Todos os arquivos de simulação podem ser encontrados em DVD, anexados a esta dissertação.

6.3 – Implementação e testes do `elemento_circuito` no FPGA.

A implementação e os testes do `elemento_circuito` foram realizados no laboratório LACE, no Departamento de Ciências da Computação e Estatística da UNESP de São José do Rio Preto. Os equipamentos utilizados foram:

- um dispositivo Stratix II EP2S60F672C5ES ;
- um cabo *USB BLASTER*;
- um display LCD de 7 segmentos;
- uma fonte de alimentação de 16v, para alimentação da placa;

- computador Pentium4 de 3GHz e 512MB de RAM, com o *software* QUARTUS II versão 7.1 full instaldo.

Após a compilação da unidade elemeneto_circuito, o *software* Quartus II, gera um arquivo denominado elemento_circuito.sof, o qual é utilizado no processo de configuração da placa.

O dispositivo EP2S60F672C5ES possui 24.176 módulos de lógica adaptável (ALM) e 2.544.192 bits de memória *on chip*. Ao configurar o EP2S60F672C5ES, o *software* Quartus II carrega o arquivo elemento_circuito.sof para o dispositivo através do cabo USB Blaster.

Na fig. 6.1 mostramos o FPGA com o cabo *USB Blaster* ligado no conector JTAG J24 e fonte de alimentação ligado no J26, todas estas informações estão no manual da placa que estão listados no anexo I.

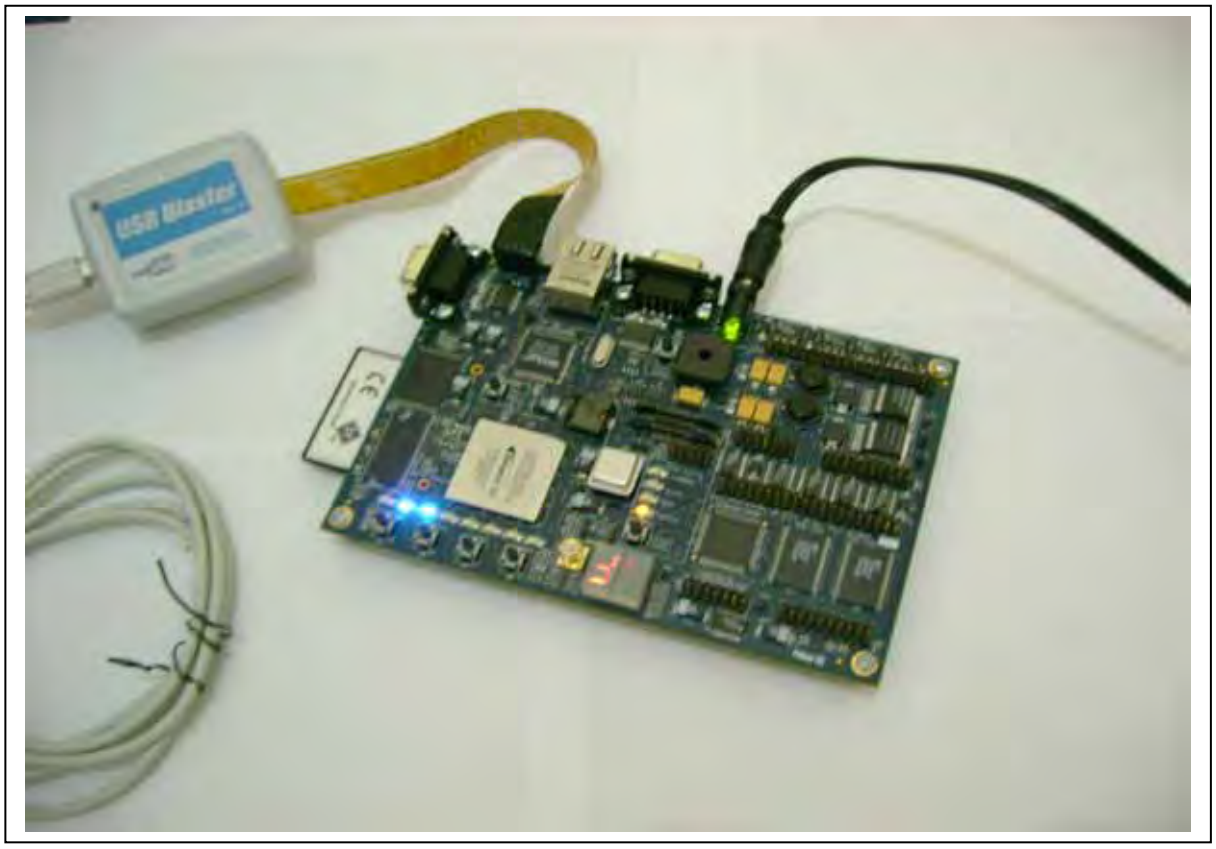


Figura 6.1 – Stratix II EP2S60F672C5ES

O dispositivo `elemento_circuito` é composto por 263 pinos de entrada e 128 pinos de saída, dando um total de 391 pinos utilizados, ele ocupou 79% da pinagem do FPGA, a utilização lógica se deu em torno de 24% como mostramos no Relatório com resumo dos resultados da compilação que mostramos na fig. 6.2. Informações referentes à distribuição dos

sinais especificados nos pinos de entrada/saída que compõem o dispositivo configurado podem ser visualizadas no anexo IV.

Flow Summary	
Flow Status	Successful - Thu Jun 21 11:07:15 2007
Quartus II Version	7.1 Build 156 04/30/2007 SJ Full Version
Revision Name	elemento_circuito
Top-level Entity Name	elemento_circuito
Family	Stratix II
Device	EP2S60F672C5ES
Timing Models	Final
Met timing requirements	No
Logic utilization	24 %
Combinational ALUTs	10,942 / 48,352 (23 %)
Dedicated logic registers	2,217 / 48,352 (5 %)
Total registers	2217
Total pins	391 / 493 (79 %)
Total virtual pins	0
Total block memory bits	0 / 2,544,192 (0 %)
DSP block 9-bit elements	0 / 288 (0 %)
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 2 (0 %)

Figura 6.2 – Relatório com resumo dos resultados da compilação.

Terminada a fase de compilação do dispositivo, fizemos testes para mostrar que a arquitetura funciona. Para a realização dos testes foram desenvolvidos três novos blocos em VHDL, quais sejam: uma memória denominada `rom_entrada_dados`, uma memória denominada `rom_verifica_resultados` e um registrador `estado_rom_entrada`. Estes três novos blocos interagem com o bloco principal `elemento_circuito`, como mostrado na fig.6.3.

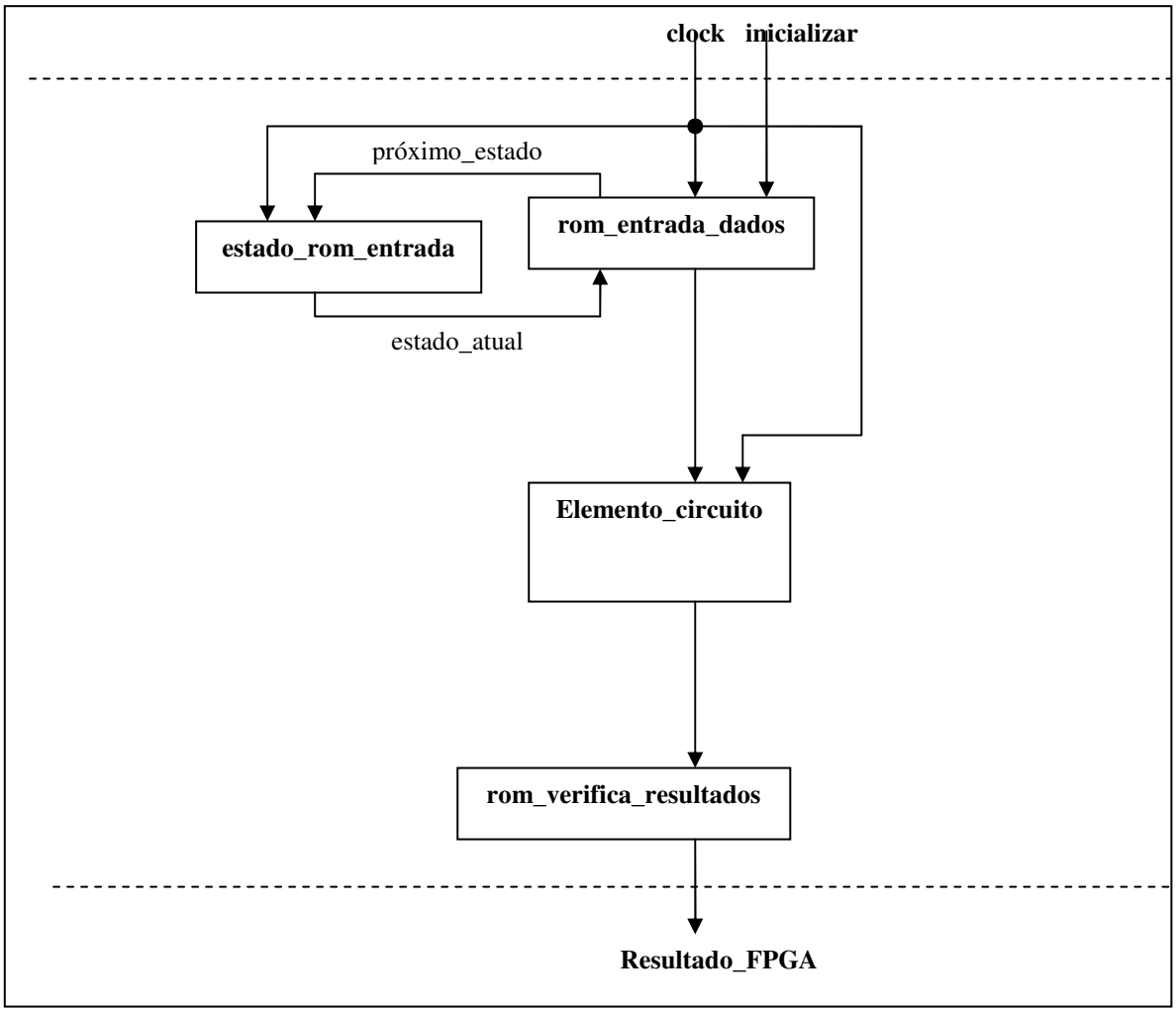


Figura 6.3 – Teste_FPGA

A memória `rom_entrada_dados` é usada como vetor teste no circuito; é nela que estão inseridas as entradas para o cálculo do elemento escolhido, ou seja, se o `estado_atual` estiver na posição `'00'` então armazenamos as seguintes palavras de memória nesta posição:

```
reset_dados <= '1';
inicializar_dados <= '0';
configuração <= '0';
proximo_estado <= "01";
```

Quando `reset_dados` for igual a `'1'` ele zera os registradores e passa para o `proximo_estado "01"` que será armazenado no registrador `estado_rom_entrada`.

Se o `estado_atual` estiver na posição `"01"` então armazenaremos as seguintes palavras de memória nesta posição:

```
reset_dados <= '0';
inicializar_dados <= '0';
configuração <= '1';
elemento <= "000";
data_sinal_v0 <= '0';
data_exp_v0 <= "10000001";
data_signif_v0 <= "00000000000000000000000";
proximo_estado <= "10";
```

Quando a `configuração` estiver no nível lógico `'1'` é informado ao `elemento_circuito` os valores das variáveis necessários para emular o elemento escolhido e, no caso deste exemplo, a variável que esta sendo atribuída a `v0` é `R` (como foi explicado no Capítulo 5). O elemento a ser executado pela arquitetura que escolhemos para este exemplo é um resistor `"000"`, e assim passamos para o `próximo_estado "10"`.

Se o `estado_atual` estiver na `posição "10"` então armazenamos as seguintes palavras de memória nesta posição:

```
reset_dados <= '0';
inicializar_dados <= '1';
configuração <= '0';
data_sinal_v0 <= '0';
data_exp_v0 <= "10000000";
data_signif_v0 <= "00000000000000000000000";
data_sinal_v1 <= '0';
data_exp_v1 <= "10000001";
data_signif_v1 <= "00000000000000000000000";
data_sinal_v2 <= '0';
data_exp_v2 <= "10000000";
data_signif_v2 <= "00000000000000000000000";
data_sinal_v3 <= '0';
data_exp_v3 <= "10000000";
data_signif_v3 <= "00000000000000000000000";
data_sinal_i0 <= '0';
data_exp_i0 <= "01111111";
data_signif_i0 <= "00000000000000000000000";
data_sinal_i1 <= '0';
data_exp_i1 <= "10000000";
```

```
data_signif_i1 <= "00000000000000000000000";
data_sinal_i2 <= '0';
data_exp_i2 <= "01111111";
data_signif_i2 <= "00000000000000000000000";
data_sinal_i3 <= '0';
data_exp_i3 <= "00000000";
data_signif_i3 <= "00000000000000000000000";
```

Quando `inicializar_dados` passar para o nível lógico `'1'` os valores de corrente e tensão (dos MPHs vizinhos N, S, L e O) serão inseridos na memória. Depois de inseridos, o `inicializar_dados` passa para `'0'`, finalizando-se o armazenamento das palavras de memória nas posições requeridas.

Depois de carregar estas variáveis, no `elemento_circuito`, a arquitetura projetada irá realizar a configuração e a execução do elemento escolhido, fornecendo os seguintes resultados nas variáveis de saída:

```
data_sinal_va <= '0';
data_exp_va <= "10000001";
data_signif_va <= "10100000000000000000000";
data_sinal_vb <= '0';
data_exp_vb <= "10000000";
data_signif_vb <= "00000000000000000000000";
data_sinal_Vab <= '0';
data_exp_Vab <= "10000001";
data_signif_Vab <= "00100000000000000000000";
data_sinal_Iab <= '0';
data_exp_Iab <= "01111111";
data_signif_Iab <= "00100000000000000000000";
```

Estes resultados serão comparados com os resultados esperados nos vetores de comparação armazenados na memória `rom_verifica_resultados`. Os valores armazenados nesta memória foram:

```
data_sinal_va <= '0' and data_exp_va <= "10000001" and
data_signif_va <= "10100000000000000000000" and data_sinal_vb <= '0'
and data_exp_vb <= "10000000" and data_signif_vb <=
"00000000000000000000000" and data_sinal_Vab <= '0' and data_exp_Vab
<= "10000001" and data_signif_Vab <= "00100000000000000000000" and
```

```
data_sinal_Iab <= '0' and data_exp_Iab <= "01111111" and
data_signif_Iab <= "00100000000000000000000";
```

Se os valores armazenados na `rom_verifica_resultados`, em `posição_rom` `"11"`, forem iguais aos que foram fornecidos pelas variáveis de saída no cálculo do resistor do `elemento_circuito`, então o sinal de saída `resultado vai para o` estado lógico baixo, acendendo com isso um *led* do FPGA comprovando que o circuito emulou um resistor e respondeu de acordo com o esperado. Se o resultado não for o esperado então `resultado <= '1'.`

Os componentes de toda a arquitetura desenvolvida em VHDL estão em um DVD, anexado a este trabalho.

Capítulo VII – Considerações Finais.

7.0 - Conclusões

O primeiro passo para o desenvolvimento deste projeto foi o domínio da linguagem de descrição de *hardware* VHDL paralelamente com o estudo de componentes digitais. Depois foi preciso estudar o ABACUS, e em seguida demos início ao desenvolvimento de nossa arquitetura, descrevendo os elementos de circuito em blocos de *hardware*, descrevemos bloco por bloco em VHDL e testamos separadamente cada elemento, para depois juntarmos todos os elementos e definirmos a arquitetura reconfigurável da unidade de processamento e estabelecer os controles para que assim possa simular o elemento que for pedido. Fizemos a compilação da unidade, o reconhecimento da FPGA que utilizamos, simulamos, implementamos e testamos a arquitetura no FPGA.

A bateria de simulações e testes mostrou que, a arquitetura da unidade `elemento_circuito` é viável, baseada na arquitetura proposta para o projeto.

Das 48.352 ALUTs, 10.942 foram utilizadas pela unidade `elemento_circuito`, totalizando 23% de ocupação da FPGA e dos registradores lógicos dedicados foram utilizados 2.217 totalizando 5% de ocupação da FPGA, a utilização lógica da FPGA ficou em torno de 24%, e a pinagem em 79%. Neste trabalho desenvolvemos uma arquitetura reconfigurável para um processador de modelos (MPH) no ABACUS, enquanto FRANÇA [3] implementou seu projeto de um elemento de processamento com uma unidade de controle microprogramada, ou seja, trata da ativação de linhas de controle através de microinstruções, armazenadas em uma memória endereçável especial chamada de memória de controle.

7.1 – Sugestões para trabalhos futuros.

Para trabalhos futuros sugerimos a implementação de um mini circuito, por exemplo, com pelo menos três MPHs em um FPGA. Isto serviria para termos uma idéia real do funcionamento do sistema em *hardware* e nos daria indicações melhores para a implementação de um sistema com muitos MPHs, capaz de simular um circuito para executar um conjunto maior de funções ou uma função mais complexa.

Bibliografia

[1] MEHL, E. L. M. *Apostila de simulação de circuitos eletrônicos em computadores.* Paraná, UFPR, 1991. 17p. Disponível em : <http:// www.eletrica.ufpr.br/**mehl**/simulacao.pdf >. Acessado em: 10 de janeiro de 2007.

[2] MARRANGHELLO, N. *Uma metodologia para a simulação de circuitos ULSI.* 1992. 105f.. Tese (Doutorado)-Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, Campinas, 1992.

[3] FRANÇA, E. *Projeto de um circuito integrado dedicado à simulação de circuitos ULSI.* 1999.112f. Tese (Doutorado)- Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, Campinas, 1999.

[4] OLIVEIRA, T. *Projeto de um somador/subtrator de ponto flutuante*. São José do Rio Preto:UNESP, 2002. 123p.

[5] PAIVA, L.C. *Projeto de um multiplicador de ponto flutuante.* São José do Rio Preto: UNESP, 2005. 59p.

[6] ASSONI, V.T.K. *Projeto de um divisor de ponto flutuante.* São José do Rio Preto: UNESP, 2003. 55p.

[7] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS-IEEE. Revista do IEEE América latina. Disponível em: <http://ieee.org.br>. Acessado em: 15 Abr. 2006.

[8] ALTERA CORPORATION. Download do software Quartus II.California: [s.n.], 2007. Disponível em:<www.altera.com>. Acessado em: 8 Set. 2005.

[9] MANO, M. M. *Digital design.* 2.ed. United States: Pretince Hall, 1991. 516p.

[10] MANO, M. M. *Logic and computer design fundamentals.* 2.ed. United States: Pretince Hall, 2000. 652 p.

[11] MAZOR, S.; LANGSTRAAT, P. *A Guide to VHDL.* 2.ed. Boston: Kluwer Academic Publishers, 1993. 180p.

[12] COELHO, D. *The VHDL handbook.* Norwerll: Kluwer Academic Publishers, 1989. 86p.

[13] PELLERIN, D. *Introduction to VHDL* – for synthesis and simulation. United States, Accolade Design Automation, 1997. 230p. Disponível em: <www.acc_eda.com/h_intro.htm>. Acessado em: 14 Abril 2005.

[14] ZHU, Z.; SHI, R.; CHENG, C. K.; KUH, E.S. *An unconditional stable general operator splitting method for transistor level transient analysis* . New York: ACM Press, 2006. 113p.

[15] TANENBAUN, A.S. *Organização estruturada de computadores,* 3.ed. Rio de Janeiro: Pretince Hall do Brasil, 1992. 464p.

[16] CHANG, S.M. Sze, *ULSI technology.* New York, McGHraw, 1996. 180p.

[17] HODGES, D., JACKSON, H. *Analysis and design of digital integrated circuits* . New York: McGraw-Hill, 1983. 529p.

[18] SEDRA, A. SMITH, K. *Microelectronic circuits.* Oxford: Oxford University Press, 2004. 223p.

Anexo I: Composição do DVD

Esta dissertação vem acompanhada de um DVD com as seguintes pastas: CONVERSOR, MANUAL_PLACA, ELEMENTO_CIRCUITO, QUARTUS_II e SIMULAÇÕES.

- Na pasta CONVERSOR temos o arquivo decimal.htm de 35kb que é conversor de números decimais para ponto flutuante segundo a norma da IEEE-754;
- Na pasta MANUAL_PLACA está o arquivo manual_EP2S60F672C5ES.pdf de 1,41KB, que é o manual da placa utilizada neste trabalho e uma outra past CONFIG_PLACA onde estão os arquivos: 71_nios2eds_windows.exe de 618MB e 71_ip_windows.exe de 98,1MB, que são utilizados para restaurar a programação de fábrica do dispositivo;
- Na pasta QUARTUS_II está o arquivo 71_quartus_windows.exe de 1,37GB é a versão integral deste *software*, este arquivo não vem acompanhado de licença por ela ser específica para cada computador;
- Na pasta ELEMENTO_CIRCUITO estão os arquivos da arquitetura `elemento_circuito` e todos seus componentes estão na pasta COMPONENTES_ELEMENTO_CIRCUITO;
- Na pasta SIMULAÇÕES estão todos os arquivos de simulação do `elemento_circuito`.
- Na pasta TESTE está à arquitetura utilizada para a realização dos testes na FPGA.

Anexo II: Tabelas de simulações do elemento_circuito.

Tabela 6 – O `elemento_circuito` simulando o resistor.

Entradas do simulador			
elemento	"000"		
	sinal	expoente	significando
(v0)R	0	10000001	01000000000000000000000
v0	0	10000010	01000000000000000000000
v1	0	10000011	11100000000000000000000
v2	0	00000000	00000000000000000000000
v3	0	00000000	00000000000000000000000
i0	0	10000001	00000000000000000000000
i1	0	01111111	00000000000000000000000
i2	0	10000000	00000000000000000000000
i3	0	01111111	00000000000000000000000
Saídas do simulador			
va	0	10000011	01000000000000000000000
vb	1	00000000	00000000000000000000000
Vab	0	10000011	01000000000000000000000
Iab	0	10000001	00000000000000000000000
Entradas do simulador			
elemento	"000"		
	sinal	expoente	significando
(v0)R	0	10000000	01000000000000000000000
v0	0	10000011	01000000000000000000000
v1	0	10000010	10000000000000000000000
v2	0	10000010	01000000000000000000000
v3	0	10000000	00000000000000000000000
i0	0	10000000	00000000000000000000000
i1	0	10000000	10000000000000000000000
i2	0	01111111	00000000000000000000000
i3	0	10000000	00000000000000000000000
Saídas do simulador			
va	0	10000011	00000000000000000000000
vb	0	10000001	10000000000000000000000
Vab	0	10000010	01000000000000000000000
Iab	0	10000001	00000000000000000000000

Tabela 7 – O `elemento_circuito` simulando o capacitor.

Entradas do simulador			
elemento	"001"		
	sinal	expoente	significando
(v0)C	0	01111111	10000000000000000000000
(v0)vf	0	01111110	00000000000000000000000
(v0)Izero	0	01111111	00000000000000000000000
(v0)t	0	01111101	10011001100110011001101
(v0)delta_t	0	01111101	11001100110011001100110
(v0)Vcarga	0	01111101	00000000000000000000000
(v0)n_euler	0	10000000	01011011111100001010001
v0	0	00000000	00000000000000000000000
v1	0	00000000	00000000000000000000000
v2	0	10000000	00000000000000000000000
v3	0	10000000	10000000000000000000000
i0	0	01111111	10000000000000000000000
i1	0	10000000	00000000000000000000000
i2	0	10000000	11000000000000000000000
i3	0	10000001	00000000000000000000000
Saídas do simulador			
va	0	00000000	00000000000000000000000
vb	1	01111101	10010110010101000110101
Vab	0	01111101	10010110010101000110101
Iab	0	01111101	10100001100001110010011
Entradas do simulador			
elemento	"001"		
	sinal	expoente	significando
(v0)C	0	01111111	11011001100110011001101
(v0)vf	0	01111011	11101011100001010001111
(v0)Izero	0	01111101	10101110000101000111101
(v0)t	0	01111101	00000000000000000000000
(v0)delta_t	0	01111101	00110011001100110011010
(v0)Vcarga	0	01111110	10110011001100110011010
(v0)n_euler	0	10000000	01011011111100001010001
v0	0	10000000	00000000000000000000000
v1	0	10000000	10000000000000000000000
v2	0	00000000	00000000000000000000000
v3	0	00000000	00000000000000000000000
i0	0	01111110	00000000000000000000000
i1	0	01111110	00000000000000000000000
i2	0	01111111	00000000000000000000000
i3	0	00000000	00000000000000000000000
Saídas do simulador			
va	0	01111111	01001010111101001111000
vb	0	00000000	00000000000000000000000
Vab	0	01111111	01001010111101001111000
Iab	0	01111110	00111001011001010010110

Tabela8 – O `elemento_circuito` simulando o diodo.

Entradas do simulador			
elemento	"010"		
	sinal	expoente	significando
(v0)vt	0	01111101	01100110011001100110011
(v0)fabs	0	00000000	00000000000000000000000
(v0)Imax	0	00000000	00000000000000000000000
(v0)Isat	0	01111100	00001010001111010111000
(v0)e_12	0	10000000	01011011111100001010001
v0	0	01111111	10000000000000000000000
v1	0	01111110	00000000000000000000000
v2	0	01111101	11001100110011001100110
v3	0	01111101	00000000000000000000000
i0	0	00000000	00000000000000000000000
i1	0	00000000	00000000000000000000000
i2	0	00000000	00000000000000000000000
i3	0	00000000	00000000000000000000000
Saídas do simulador			
va	0	01111111	00000000000000000000000
vb	0	01111101	01100110011001100110011
Vab	0	01111110	01001100110011001100110
Iab	0	01111111	11000100010011010000001
Entradas do simulador			
elemento	"010"		
	sinal	expoente	significando
(v0)vt	0	01111110	10000000000000000000000
(v0)fabs	0	01111111	00101110000101000111101
(v0)Imax	1	10000000	00100000000000000000000
(v0)Isat	0	01111110	10111101011100001010010
(v0)e_12	0	10000000	01011011111100001010001
v0	0	01111110	00000000000000000000000
v1	0	01111101	00110011001100110011010
v2	0	01111111	00110011001100110011010
v3	0	10000000	00100110011001100110011
i0	0	00000000	00000000000000000000000
i1	1	01111111	10000000000000000000000
i2	1	01111110	10000000000000000000000
i3	0	00000000	00000000000000000000000
Saídas do simulador			
va	0	01111111	10000110001101010100000
vb	0	01111111	11000000000000000000000
Vab	1	01111100	11001110010101100000010
Iab	0	01111111	01011000000001100110110

Tabela 9 – O `elemento_circuito` simulando a fonte de tensão.

Entradas do simulador			
elemento	"011"		
	sinal	expoente	significando
(v0)2_pi	0	10000001	10010010000111111011011
(v0)ft	0	00000000	00000000000000000000000
v0	0	00000000	00000000000000000000000
v1	0	00000000	00000000000000000000000
v2	0	10000000	00000000000000000000000
v3	0	10000000	10000000000000000000000
i0	0	01111111	10000000000000000000000
i1	0	10000000	00000000000000000000000
i2	0	10000000	11000000000000000000000
i3	0	10000001	00000000000000000000000
Saídas do simulador			
va	0	00000000	00000000000000000000000
vb	0	10000000	01000000000000000000000
Vab	0	10000000	01000000000000000000000
Iab	0	10000001	01100000000000000000000
Entradas do simulador			
elemento	"011"		
	sinal	expoente	significando
(v0)2_pi	0	10000001	10010010000111111011011
(v0)ft	1	01111110	00000000000000000000000
v0	0	10000000	00000000000000000000000
v1	0	10000000	10000000000000000000000
v2	0	00000000	00000000000000000000000
v3	0	00000000	00000000000000000000000
i0	0	01111110	00000000000000000000000
i1	0	01111110	00000000000000000000000
i2	0	01111111	00000000000000000000000
i3	0	00000000	00000000000000000000000
Saídas do simulador			
va	1	10000000	01000000000000000000000
vb	0	00000000	00000000000000000000000
Vab	1	10000000	01000000000000000000000
Iab	0	01111111	00000000000000000000000

Tabela 10 – O `elemento_circuito` simulando a fonte de corrente.

Entradas do simulador			
elemento	"100"		
	sinal	expoente	significando
(v0)2_pi	0	10000001	10010010000111111011011
(v0)ft	0	00000000	00000000000000000000000
v0	0	00000000	00000000000000000000000
v1	0	00000000	00000000000000000000000
v2	0	10000000	00000000000000000000000
v3	0	10000000	10000000000000000000000
i0	0	01111111	10000000000000000000000
i1	0	10000000	00000000000000000000000
i2	0	10000000	11000000000000000000000
i3	0	10000001	00000000000000000000000
Saídas do simulador			
va	0	00000000	00000000000000000000000
vb	0	10000000	01000000000000000000000
Vab	1	10000000	01000000000000000000000
Iab	0	10000010	01100000000000000000000
Entradas do simulador			
elemento	"100"		
	sinal	expoente	significando
(v0)2_pi	0	10000001	10010010000111111011011
(v0)ft	1	01111110	00000000000000000000000
v0	0	10000000	00000000000000000000000
v1	0	10000000	10000000000000000000000
v2	0	00000000	00000000000000000000000
v3	0	00000000	00000000000000000000000
i0	0	01111110	00000000000000000000000
i1	0	01111110	00000000000000000000000
i2	0	01111111	00000000000000000000000
i3	0	00000000	00000000000000000000000
Saídas do simulador			
va	0	10000000	01000000000000000000000
vb	0	00000000	00000000000000000000000
Vab	0	10000000	01000000000000000000000
Iab	1	10000000	00000000000000000000000

Anexo III: Exemplo do cálculo efetuado na simulação do resistor.

BCPI

ia' = i0+i1	va' = (v0+v1)/2
ib' = i2+i3	vb' = (v2+v3)/2
Iab' = (ia' + ib')/2	Vab' = va' – vb'

Onde: i0 = 4, i1 = 2, i2 = 2, i3 = 0, v0 =20, v1 = 25, v2 = 2, v3 = 3

ia' = 4 + 2 = 6	va' = (20 + 25)/2 = 22,5
ib' = 2 + 0 = 2	vb' = (2 + 3)/2 = 2,5
Iab' = (6 + 2)/2 = 4	Vab' = 22,5 – 2,5 = 20

Elemento – Resistor

Vab'' = R * Iab'	se (va'= 0) então (vb = -Vab)
Iab'' = Vab'/ R	senão se (vb'= 0) então (va = Vab)
Vab = (Vab' + Vab'')/2	senão se (va'≠ 0) e (vb' ≠ 0)
Iab = (Iab' + Iab'')/2	então (va = vb + Vab) e (vb = vb')

Onde R= 5

Vab'' = 5 * 4 = 20

Iab'' = 20 / 5 = 4

Vab = (20 + 20)/2 = 20

Iab = (4 + 4)/2 = 4

vb = 2,5

va = 2,5 + 20 = 22,5

Anexo IV: Pinagem da FPGA configurada.

Abaixo expõem-se a identificação do dispositivo onde a unidade elemento_circuito foi configurada e a sua pinagem, a qual relaciona os sinais de entrada/saída definidos no projeto com a numeração dos pinos de entrada/saída referentes ao dispositivo configurado. Essas informações foram extraídas do arquivo elemento_circuito.pin, o qual foi gerado no processo de compilação da arquitetura pelo Quartus II.

Quartus II Version 7.1 Build 156 04/30/2007 SJ Full Version
CHIP "elemento_circuito" ASSIGNED TO AN: EP2S60F672C5ES

Pin Name/Usage	Location	Dir	Pin Name/Usage	Location	Dir
signif_Vab[8]	:A3	:output	data_signif_v3[18]	: AA24	: input
exp_Vab[1]	: A5	:output	signif_Iab[4]	: AA25	: output
signif_Vab[17]	: A7	:output	data_signif_i0[10]	: AB1	: input
signif_Iab[9]	: A8	: output	data_signif_i1[15]	: AB2	: input
data_signif_i2[14]	: A9	:input	data_signif_i0[19]	: AB3	: input
signif_vb[16]	: A10	:output	data_signif_i1[16]	: AB4	: input
data_exp_i2[6]	: A12	: input	exp_Vab[0]	: AB8	: output
signif_vb[9]	: A15	: output	exp_Iab[7]	: AB9	: output
signif_va[8]	: A17	: output	exp_Vab[3]	: AB10	: output
signif_vb[10]	: A18	: output	data_signif_i1[0]	: AB11	: input
signif_va[14]	: A19	: output	data_exp_i1[7]	: AB12	: input
sinal_va	: A20	: output	data_exp_v3[0]	: AB13	: input
signif_Iab[18]	: A21	: output	data_exp_v2[5]	: AB14	: input
data_signif_v1[0]	: A22	: input	data_exp_v2[7]	: AB15	: input
data_signif_i1[22]	: AA1	: input	data_sinal_v2	: AB16	: input
data_signif_i0[7]	: AA2	: input	data_exp_v3[6]	: AB17	: input
data_signif_i0[17]	: AA3	: input	signif_Iab[22]	: AB18	: output
data_signif_i1[9]	: AA4	: input	data_signif_v3[3]	: AB23	: input
data_signif_i0[20]	: AA5	: input	data_signif_v2[0]	: AB24	: input
data_signif_i0[16]	: AA6	: input	data_signif_v3[22]	: AB25	: input
data_signif_i1[8]	: AA10	: input	data_signif_v2[17]	: AB26	: input
data_sinal_i1	: AA11	: input	data_signif_i1[6]	: AC2	: input
data_exp_v2[3]	: AA14	: input	data_signif_i0[11]	: AC3	: input
data_exp_v3[1]	: AA15	: input	data_signif_i0[6]	: AC7	: input
data_signif_v3[12]	: AA16	: input	exp_Iab[5]	: AC8	: output
data_signif_v2[13]	: AA17	: input	data_exp_v3[7]	: AC13	: input
signif_Iab[8]	: AA19	: output	data_exp_v2[2]	: AC14	: input
data_signif_v3[1]	: AA21	: input	data_exp_v2[1]	: AC15	: input
data_signif_v2[2]	: AA22	: input	exp_Iab[2]	: AC18	: output
data_signif_v3[6]	: AA23	: input	exp_Iab[3]	: AC19	: output

Pin Name/Usage	Location	Dir
data_signif_v2[7]	: AC24	: input
data_exp_v3[5]	: AC25	: input
data_signif_i1[10]	: AD1	: input
signif_Vab[6]	: AD2	: output
exp_Vab[5]	: AD3	: output
signif_Vab[12]	: AD4	: output
signif_Iab[14]	: AD5	: output
data_signif_i0[14]	: AD7	: input
exp_Iab[1]	: AD12	: output
data_sinal_i2	: AD13	: input
data_exp_i2[2]	: AD14	: input
data_exp_v3[2]	: AD15	: input
data_signif_v3[19]	: AD16	: input
data_signif_v3[2]	: AD17	: input
data_signif_v3[17]	: AD18	: input
data_signif_v2[10]	: AD19	: input
signif_Vab[18]	: AD23	: output
data_signif_v2[5]	: AD25	: input
data_signif_v3[13]	: AD26	: input
signif_Iab[0]	: AE4	: output
data_signif_i1[12]	: AE6	: input
data_signif_i0[13]	: AE7	: input
data_signif_i1[14]	: AE8	: input
signif_Vab[9]	: AE12	: output
signif_Iab[10]	: AE13	: output
data_sinal_v3	: AE14	: input
data_exp_v2[6]	: AE15	: input
data_signif_v3[20]	: AE16	: input
data_signif_v2[20]	: AE17	: input
data_signif_v3[4]	: AE18	: input
data_signif_v2[19]	: AE19	: input
exp_Vab[6]	: AE20	: output
signif_Vab[13]	: AE22	: output
sinal_Iab	: AF3	: output
data_signif_i0[5]	: AF6	: input
data_signif_i1[13]	: AF7	: input
data_exp_i0[3]	: AF8	: input
signif_Iab[3]	: AF12	: output
data_exp_v2[0]	: AF15	: input
data_signif_v3[15]	: AF17	: input
data_signif_v2[18]	: AF18	: input
data_signif_v3[14]	: AF19	: input
data_signif_v2[15]	: AF20	: input
sinal_Vab	: AF24	: output
signif_Iab[2]	: B4	: output
signif_Vab[22]	: B5	: output
signif_Iab[21]	: B6	: output
signif_Iab[16]	: B8	: output

Pin Name/Usage	Location	Dir
data_signif_i3[19]	: B9	: input
data_exp_i3[1]	: B10	: input
data_exp_i3[7]	: B11	: input
signif_vb[18]	: B12	: output
signif_vb[7]	: B13	: output
data_exp_i2[3]	: B14	: input
signif_vb[17]	: B15	: output
signif_vb[1]	: B16	: output
signif_va[0]	: B17	: output
signif_va[7]	: B18	: output
signif_va[4]	: B19	: output
signif_Iab[15]	: B20	: output
data_signif_v1[21]	: B21	: input
signif_Vab[3]	: B23	: output
exp_Vab[2]	: C3	: output
signif_Iab[1]	: C7	: output
exp_Iab[0]	: C8	: output
data_exp_i3[3]	: C9	: input
data_exp_i3[0]	: C10	: input
data_exp_i3[6]	: C11	: input
signif_vb[13]	: C12	: output
signif_vb[5]	: C13	: output
signif_vb[19]	: C14	: output
signif_vb[0]	: C15	: output
signif_va[9]	: C16	: output
signif_vb[12]	: C17	: output
signif_va[6]	: C18	: output
signif_va[20]	: C19	: output
exp_Iab[4]	: C20	: output
signif_Vab[5]	: C22	: output
exp_Iab[6]	: D6	: output
signif_Vab[0]	: D7	: output
data_exp_i3[5]	: D8	: input
data_signif_i3[13]	: D9	: input
signif_vb[15]	: D10	: output
signif_vb[6]	: D12	: output
data_exp_i2[5]	: D13	: input
signif_vb[21]	: D14	: output
exp_vb[3]	: D15	: output
exp_vb[4]	: D17	: output
signif_Iab[20]	: D18	: output
signif_Iab[17]	: D19	: output
signif_Iab[6]	: D24	: output
data_exp_v1[6]	: D25	: input
signif_Vab[20]	: E4	: output
signif_Iab[12]	: E9	: output
exp_Vab[7]	: E11	: output
data_exp_i3[4]	: E12	: input

Pin Name/Usage	Location	Dir
signif_vb[2]	: E13	: output
exp_vb[1]	: E14	: output
signif_va[12]	: E15	: output
signif_Iab[11]	: E17	: output
data_signif_v1[17]	: E18	: input
data_exp_v1[7]	: E23	: input
exp_vb[7]	: E24	: output
data_signif_v1[5]	: E25	: input
data_exp_v1[0]	: E26	: input
exp_Vab[4]	: F1	: output
signif_Vab[11]	: F2	: output
data_exp_i2[1]	: F12	: input
signif_vb[14]	: F13	: output
signif_va[5]	: F14	: output
signif_va[21]	: F15	: output
exp_vb[5]	: F16	: output
exp_va[6]	: F17	: output
signif_Vab[2]	: F19	: output
data_signif_v1[18]	: F21	: input
data_signif_v1[19]	: F22	: input
reset	: F23	: input
data_signif_v1[12]	: F24	: input
data_exp_v1[4]	: F25	: input
data_signif_v1[1]	: F26	: input
data_signif_i2[4]	: G1	: input
data_signif_i3[8]	: G2	: input
signif_Vab[10]	: G6	: output
signif_Vab[4]	: G10	: output
data_signif_i2[0]	: G11	: input
signif_va[18]	: G14	: output
exp_vb[6]	: G15	: output
signif_vb[11]	: G16	: output
signif_va[15]	: G17	: output
signif_Vab[15]	: G20	: output
data_exp_v1[1]	: G21	: input
data_signif_v1[10]	: G23	: input
data_signif_v1[15]	: G24	: input
data_signif_v1[6]	: G25	: input
data_signif_v0[8]	: G26	: input
data_signif_i3[22]	: H1	: input
data_exp_i3[2]	: H2	: input
data_exp_i2[7]	: H3	: input
data_exp_i2[0]	: H4	: input
data_signif_i2[9]	: H11	: input
exp_vb[2]	: H15	: output
signif_va[17]	: H16	: output
exp_va[7]	: H17	: output
data_exp_v1[5]	: H18	: input

Pin Name/Usage	Location	Dir
data_exp_v1[2]	: H21	: input
data_signif_v1[16]	: H22	: input
data_signif_v1[8]	: H23	: input
data_signif_v0[5]	: H24	: input
signif_va[3]	: H25	: output
signif_va[2]	: H26	: output
data_signif_i2[17]	: J1	: input
data_signif_i3[11]	: J2	: input
data_signif_i2[11]	: J3	: input
data_signif_i2[8]	: J4	: input
signif_Iab[7]	: J5	: output
signif_Vab[19]	: J7	: output
signif_Vab[14]	: J8	: output
exp_va[1]	: J14	: output
signif_vb[22]	: J15	: output
signif_vb[20]	: J17	: output
data_exp_v1[3]	: J19	: input
data_signif_v1[20]	: J20	: input
data_signif_v1[7]	: J21	: input
data_signif_v1[22]	: J22	: input
signif_vb[3]	: J23	: output
data_signif_v1[9]	: J24	: input
signif_va[22]	: J25	: output
exp_va[3]	: J26	: output
data_signif_i3[3]	: K1	: input
data_signif_i2[3]	: K2	: input
data_signif_i3[21]	: K3	: input
data_signif_i3[7]	: K4	: input
signif_Iab[19]	: K7	: output
signif_Iab[5]	: K8	: output
sinal_vb	: K16	: output
signif_Vab[7]	: K18	: output
data_signif_v1[13]	: K19	: input
data_signif_v0[13]	: K20	: input
data_signif_v1[3]	: K21	: input
data_sinal_v0	: K22	: input
signif_vb[4]	: K23	: output
data_signif_v1[2]	: K24	: input
data_exp_v0[3]	: K25	: input
exp_vb[0]	: K26	: output
data_signif_i3[0]	: L2	: input
data_signif_i2[21]	: L3	: input
signif_vb[8]	: L4	: output
data_signif_i2[16]	: L6	: input
data_signif_i3[17]	: L7	: input
data_exp_i2[4]	: L8	: input
data_signif_i3[2]	: L9	: input
exp_va[0]	: L18	: output

Pin Name/Usage	Location	Dir
signif_va[16]	: L19	: output
data_signif_v1[14]	: L20	: input
data_sinal_v1	: L21	: input
data_signif_v1[11]	: L22	: input
data_signif_v1[4]	: L23	: input
signif_va[19]	: L24	: output
signif_va[1]	: L25	: output
data_signif_i2[2]	: M1	: input
data_signif_i3[4]	: M2	: input
data_signif_i2[12]	: M3	: input
data_signif_i3[10]	: M4	: input
data_signif_i3[9]	: M5	: input
data_signif_i3[5]	: M6	: input
data_signif_i2[1]	: M7	: input
data_signif_i2[22]	: M8	: input
data_signif_v0[6]	: M19	: input
data_exp_v0[4]	: M20	: input
signif_va[10]	: M21	: output
signif_va[11]	: M22	: output
exp_va[4]	: M23	: output
exp_va[5]	: M24	: output
data_signif_v0[10]	: M25	: input
data_signif_v0[21]	: M26	: input
data_signif_i2[20]	: N2	: input
data_signif_i2[5]	: N3	: input
data_signif_i3[16]	: N4	: input
data_signif_i3[12]	: N5	: input
data_signif_i3[20]	: N6	: input
data_signif_i2[15]	: N7	: input
data_signif_v0[12]	: N19	: input
data_signif_v0[22]	: N20	: input
exp_va[2]	: N21	: output
signif_va[13]	: N22	: output
data_signif_v0[11]	: N24	: input
data_signif_v0[16]	: N25	: input
data_signif_i2[10]	: P2	: input
data_signif_i2[19]	: P3	: input
data_signif_i3[6]	: P4	: input
data_signif_i2[6]	: P5	: input
data_exp_v0[2]	: P22	: input
clock	: P23	: input
data_exp_v0[0]	: P24	: input
data_exp_v0[5]	: P25	: input
data_signif_i2[7]	: R1	: input
data_signif_i2[18]	: R2	: input
data_signif_i3[18]	: R3	: input
data_signif_i3[1]	: R4	: input
elemento[0]	: R23	: input
data_signif_v0[17]	: R24	: input
data_signif_v0[3]	: R25	: input
data_exp_v0[1]	: R26	: input
data_exp_i0[4]	: T2	: input
data_signif_i2[13]	: T3	: input
data_exp_i0[1]	: T4	: input
data_sinal_i3	: T5	: input
data_signif_i0[22]	: T6	: input
data_signif_i0[21]	: T7	: input
data_exp_i1[5]	: T8	: input
data_signif_i1[19]	: T9	: input
data_signif_v2[11]	: T19	: input
data_signif_v2[21]	: T20	: input
data_signif_v3[11]	: T21	: input
data_signif_v3[21]	: T22	: input
elemento[1]	: T24	: input
elemento[2]	: T25	: input
data_exp_i1[1]	: U1	: input
data_exp_i1[0]	: U2	: input
data_exp_i0[0]	: U3	: input
data_exp_i1[6]	: U4	: input
data_sinal_i0	: U5	: input
data_exp_i1[2]	: U6	: input
data_signif_i1[21]	: U7	: input
data_signif_i0[12]	: U8	: input
data_signif_v2[16]	: U19	: input
data_signif_v2[6]	: U20	: input
data_signif_v3[0]	: U22	: input
data_signif_v0[20]	: U23	: input
data_signif_v0[19]	: U24	: input
data_exp_v0[6]	: U25	: input
data_exp_v0[7]	: U26	: input
data_exp_i1[4]	: V1	: input
data_exp_i1[3]	: V2	: input
data_exp_i0[2]	: V3	: input
data_exp_i0[5]	: V4	: input
data_signif_i0[18]	: V5	: input
data_signif_i0[8]	: V6	: input
data_signif_i1[20]	: V7	: input
data_signif_i0[9]	: V8	: input
signif_Vab[16]	: V10	: output
data_signif_i1[2]	: V12	: input
data_exp_v3[4]	: V14	: input
data_signif_v2[14]	: V16	: input
data_signif_v2[9]	: V17	: input
data_signif_v2[4]	: V19	: input
data_signif_v3[8]	: V20	: input
data_signif_v0[7]	: V21	: input

Pin Name/Usage	Location	Dir
data_signif_v0[15]	: V22	: input
data_signif_v0[9]	: V23	: input
data_signif_v0[1]	: V24	: input
data_signif_v0[0]	: V25	: input
data_signif_v0[2]	: V26	: input
data_exp_i0[6]	: W1	: input
data_exp_i0[7]	: W2	: input
data_signif_i0[15]	: W3	: input
data_signif_i0[3]	: W4	: input
data_signif_i0[4]	: W5	: input
data_signif_i1[4]	: W6	: input
data_signif_i0[2]	: W7	: input
data_signif_i1[18]	: W8	: input
signif_Vab[1]	: W10	: output
data_signif_i1[1]	: W11	: input
data_signif_i3[14]	: W12	: input
data_signif_v3[7]	: W15	: input
data_signif_v2[1]	: W16	: input
data_signif_v2[8]	: W17	: input
signif_Iab[13]	: W20	: output
data_signif_v0[4]	: W21	: input
configuracao	: W22	: input
data_signif_v2[3]	: W23	: input
data_signif_v2[12]	: W24	: input
data_signif_v0[14]	: W25	: input
data_signif_v0[18]	: W26	: input
data_signif_i1[3]	: Y1	: input
data_signif_i1[7]	: Y2	: input
data_signif_i0[0]	: Y3	: input
data_signif_i0[1]	: Y4	: input
data_signif_i1[17]	: Y7	: input
data_signif_i1[11]	: Y9	: input
data_signif_i1[5]	: Y10	: input
data_signif_i3[15]	: Y11	: input
data_exp_v2[4]	: Y15	: input
data_signif_v3[16]	: Y16	: input
data_signif_v3[5]	: Y17	: input
data_exp_v3[3]	: Y20	: input
data_signif_v2[22]	: Y21	: input
data_signif_v3[9]	: Y23	: input
data_signif_v3[10]	: Y24	: input
inicializar	: Y25	: input
signif_Vab[21]	: Y26	: output