Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação
Departamento de Semicondutores, Instrumentos e Fotônica

"PROJETO DE UM CIRCUITO INTEGRADO DEDICADO À SIMULAÇÃO DE CIRCUITOS ULSI"

Autoria: Eliane França
Orientador: Prof.Dr. Furio Damiani

Banca Examinadora:
Furio Damiani – FEEC/UNICAMP
José Carlos Pereira – EESC/USP(São Carlos)
Norian Marranghello – DCC/UNESP(São José do Rio Preto)
José Raimundo Oliveira – FEEC/UNICAMP
Marconi Kohm Madrid - FEEC/UNICAMP

Tese apresentada à FEEC/UNICAMP, como
parte dos requisitos exigidos para obtenção
do título de Doutor em Engenharia Elétrica,
Área de Eletrônica e Comunicações

Campinas, Dezembro de 1999

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

F844p

França, Eliane
Projeto de um circuito integrado dedicado à simulação de circuitos ULSI / Eliane França.--Campinas, SP: [s.n.], 1999.

Orientador: Furio Damiani.
Tese (doutorado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Arquitetura de computador. 2. Microprogramação. 3. Microprocessadores. 4. Aritmética de vírgula flutuante. I. Damiani, Furio. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

O Senhor com sabedoria fundou a Terra; preparou os céus com inteligência.

Provérbios, 4-19

Agradecimentos

A Deus, por ter-me permitido perseverar no caminho do conhecimento.

Aos meus pais, Lia e Dorvil e à minha irmã Dalva, pelo apoio, incentivo e dedicação que permitiram concentrar minhas atenções neste trabalho.

Ao Prof. Dr. Furio Damiani pela sua amizade e orientação.

Ao Prof. Dr. José Raimundo Oliveira pela sua disponibilidade, atenção e valiosas informações técnicas.

Ao Prof. Dr. Peter Jürgen Tatsch pelo apoio e amizade.

À minha amiga Raquel, secretária do DSIF, pela presteza, bom humor e eficiência no atendimento às solicitações de alunos e professores do Departamento.

À Dirce, Charles, Francisca e Sueli, pela amizade, confiança e companheirismo que sempre me dedicaram.

Ao CNPq e à Coordenadoria de Pós-Graduação, pela concessão da bolsa de estudos em parte do período do desenvolvimento do projeto.

Resumo

O objetivo deste trabalho é o desenvolvimento de um microprocessador dedicado para a simulação de circuitos ULSI – *Ultra Large Scale Integration*, ou seja, circuitos integrados de larga escala de integração. Ele faz parte de um arranjo de processadores proposto para um sistema de simulação por *hardware*, denominado ABACUS, desenvolvido no DSIF/FEEC/UNICAMP.

Dentro do ABACUS este microprocessador, denominado MPH – *Model Processing Hardware* (processador de modelos) é a célula unitária de um arranjo de microprocessadores.

A arquitetura do MPH é formada pelos seguintes blocos: registros de entrada e saída, memória para armazenamento do programa de descrição do modelo – UMA; memória para dados e resultados da simulação – MEL; controle microprogramado e Unidade Aritmética e Lógica em ponto flutuante para 32 *bits*.

Por apresentar uma arquitetura microprogramada, encontra aplicabilidade em outros sistemas dedicados tais como: satélite para previsão do tempo, robótica, redes neurais, *hardware* evolutivo, etc.

O projeto foi descrito em linguagem VHDL – *VHSIC – (Very High Speed Integrated Circuits) Hardware Description Language* e simulado em ambiente *Mentor Graphics*.

Abstract

The aim of this work is the development of a custom microprocessor to simulate ULSI – Ultra Large Scale Integration circuits. It is part of an array of processors proposed as a system for circuit simulation by Hardware, named ABACUS.

Inside the ABACUS, the microprocessor, named MPH – Model Processing Hardware (model processor), is the basic cell of the microprocessor array.

The architecture of the MPH is composed by: input and output registers, memory to store the program of description model – UMA; a memory for the storage of simulation data and results – MEL; microprogramed control and Arithmethic and Logic Unit in 32 bits floating point.

As its architecture is microprogrammed, it can be employed in other custom systems like: time prevision satellite, robotics, neural networks, evolvable hardware and so on.

The design has been described in VHDL language – VHSIC Hardware Description Language and simulated in Mentor Graphics enviroment.

Índice

Capítulo III

Capítulo IV

Capítulo V

Introdução

O universo de ferramentas dedicadas utilizadas para auxiliar na concepção de projetos eletrônicos, CAD *(Computer Aided Design)*, se constitui, nos dias de hoje, num aliado indispensável dos projetistas de circuitos integrados nas diversas etapas de projeto desde o desenvolvimento da arquitetura propriamente dita até a implementação da mesma, dada a alta complexidade das estruturas atualmente propostas.

O objetivo principal, em se gerar novas ferramentas, é buscar redução no tempo e custo de concepção de circuitos, através da automação de várias etapas da elaboração dos mesmos e também se adaptar às necessidades do mercado atual onde os circuitos tem apresentado a cada dia, um crescimento no grau de complexidade. Atualmente projetos que apresentam arquiteturas da ordem de milhões de componentes, como é o caso dos circuitos ULSI *(Ultra Large Scale Integration)*, tornam inviável o uso de ferramentas utilizadas para desenvolvimento e simulações de estruturas de pequeno porte.

Em geral, as ferramentas de CAD usadas para simulação trabalham com a representação do comportamento do circuito integrado (CI). Com esta finalidade vários níveis de abstração podem ser considerados.

No nível mais elevado, com menor índice de detalhes, o modelamento se restringe à arquitetura do sistema, ou seja, a interação de grandes blocos ou subsistemas.

Em nível mais baixo, ou melhor, vislumbrando sua arquitetura mais intrínseca, o comportamento do circuito é tratado a partir do modelo de seus componentes elétricos: transistores, resistores, capacitores, etc. O particionamento do circuito se torna em uma estratégia importante na redução da complexidade para o tratamento de estruturas de grande porte.

Embora não exista uma classificação rígida para os simuladores de circuitos existentes, eles podem ser agrupados de acordo com os níveis de abstração adequados à modelagem dos circuitos analisados. Para os circuitos digitais temos a considerar cinco grupos [4]:

- simuladores a nível de arquitetura - permitem a descrição funcional em alto nível dos grandes blocos funcionais que compõem o circuito - as saídas são funções da entrada e dos estados internos.
- simuladores a nível de registradores - o tratamento dos sistemas se dá a partir de modelos de blocos discretos mais detalhados como, por exemplo, unidades lógicas funcionais, tais como, contadores e registradores de deslocamento.

- simuladores a nível de lógica - a descrição dos circuitos se dá a partir de portas lógicas elementares, tais como: portas NE *(NAND)*, NOU *(NOR)*, etc. Os níveis lógicos (0, 1, etc) são utilizados para análise das respostas do circuito em função do tempo, para sinais aplicados às entradas.
- simuladores a nível de temporização - utilizam modelos simplificados de transistores para a avaliação de sinais de um circuito e técnicas de simulação lógica para a propagação dos valores dos sinais.
- simuladores a nível de circuitos - que prevêem o desempenho do circuito a partir da utilização de dispositivos eletrônicos simulados, cujos parâmetros são obtidos por medidas em laboratório sobre dispositivos fabricados pelo mesmo processo a ser utilizado na integração do circuito.

Estas várias formas de tratamento da simulação devem portanto buscar a melhor adequação ao modelamento do circuito em análise.

É importante ressaltar que a discussão dada acima trata fundamentalmente de simulações de circuitos lógicos. Tradicionalmente faz-se uma divisão entre o procedimento para os circuitos lógicos e analógicos. Para os circuitos analógicos o procedimento de simulação se dá a partir de resolução de equações algébrico-diferenciais utilizando métodos implícitos de integração numérica, método de Newton para a solução de sistemas lineares esparsos. Por outro lado os circuitos digitais têm suas simulações fundamentadas no escalonamento de eventos (transições de estados) .

Um fator relevante na observação de diferenças entre as simulações lógicas e analógicas é que as primeiras trabalham com um nível de abstração bastante elevado onde os transistores podem até serem tratados como simples chaves enquanto que os analógicos exigem um maior grau de definição do comportamento físico dos componentes envolvidos. O saldo resultante destas diferenças se observa na falta de detalhes do comportamento do circuitos lógicos ao longo do tempo em oposição ao tempo de execução da simulação de um circuito analógico.

As presentes pesquisas realizadas na área, inclusive neste trabalho, buscam simplicidade e rapidez nas simulações para circuitos lógicos aliada à riqueza de detalhes dos simuladores analógicos.

O procedimento encontrado para se aliar eficiência com detalhes de representação está na utilização de técnicas adaptativas, aplicadas nos simuladores híbridos, onde partes relevantes do circuito são resolvidas através de técnicas de simulação analógica enquanto parte significativa pode permanecer dormente durante parte da simulação, como por exemplo ocorre nos circuitos digitais em tecnologia MOS. Para se conhecer o que é

realmente relevante são utilizadas técnicas de escalonamento de eventos utilizada em simuladores lógicos.

Os simuladores híbridos correspondem ao estado da arte no ramo da simulação de circuitos e necessitam de um *hardware* onde diversas estruturas interajam paralelamente.

O propósito do trabalho aqui exposto é o desenvolvimento de um microprocessador dedicado à simulação de circuitos para a implementação de parte do algorítmo *ABACUS (hArdware BAsed CircUit Simulator),* proposto em trabalhos de pesquisa realizados na Faculdade de Engenharia Elétrica [1-4], na tese de Doutorado de Norian Marranghello [5] e na dissertação de Mestrado de Wilfredo Machaca Luque [6], ambas sob a orientação do Prof. Dr. Furio Damiani.

O *ABACUS* apresenta uma nova metodologia voltada à simulação de circuitos, onde se pretende tirar o maior proveito do casamento das mais eficientes variações arquiteturais com modernas técnicas de simulação; buscando a viabilização da simulação elétrica de circuitos ULSI constituída por milhões de componentes.

O microprocessador que aqui será apresentado é dedicado para o uso nesta ideologia proposta, recebe o nome do MPH *(Model Processing Hardware)* e se configura como a célula-mãe do sistema *ABACUS* de simulação, dado que o mesmo se fundamenta basicamente no uso de um arranjo de processadores, mapeados e configurados convenientemente para o projeto a ser simulado. Cada componente do circuito é representado no sistema por um destes processadores.

Uma característica peculiar desta proposta é que a simulação de qualquer componente do circuito a ser simulado, se processe por *hardware,* buscando assim significativa redução no tempo de processamento de circuitos ULSI.

Embora o circuito aqui desenvolvido tenha sido inicialmente proposto para o sistema acima mencionado veremos, durante o transcorrer da descrição do trabalho, que a sua utilização não se restringe somente a esta arquitetura, mas apresenta larga versatilidade de aplicações.

A dissertação desta tese segue a seguinte disposição:

- **Capítulo I** - *Apresentação do projeto* – Descrição sintetizada dos fundamentos do sistema de processamento e a apresentação funcional de cada bloco do conjunto previsto para operar o algorítmo proposto pelo *ABACUS.*
- **Capítulo II** - *Fundamentos Teóricos do Projeto* - Abordagens teóricas das técnicas necessárias para o desenvolvimento dos vários blocos formadores do microprocessador MPH.

- **Capítulo III** – *Desenvolvimento do Projeto* - Desenvolvimento do projeto propriamente dito com a escolha de arquiteturas adequadas, tomadas a partir do conhecimento das técnicas conhecidas no capítulo anterior.
- **Capítulo IV** - *Descrições VHDL e simulações dos blocos pertencentes à arquitetura do MPH* – Utilização da linguagem VHDL *(VHSIC (Very High Speed Integrated Circuits) Hardware Description Language)* na descrição e apresentação dos resultados dos circuitos formadores do MPH, simulados em ambiente *Mentor Graphics*.
- **Capítulo V** - *Análise dos resultados das simulações, conclusões e sugestões para trabalhos posteriores*
- **Referências Bibliográficas**

Capítulo I

Apresentação do Projeto

I.1 - Introdução

Este projeto é parte integrante do desenvolvimento de uma nova metodologia para o processo de simulação de circuitos. Pretende-se criar uma estrutura de simulação fundamentada em processamento por *hardware* visando simplificar as simulações que geralmente se fundamentam na resolução de programas, onde se utilizam enormes sistemas de equações algébrico-diferenciais não lineares resultantes dos circuitos ULSI (Ultra Large Scale Integration).

A filosofia proposta é substituir as citadas equações por reduzidas expressões comportamentais de circuitos onde a simulação ocorra fazendo uso de um arranjo de microprocessadores sujeitos a um mapeamento pré definido.

Este trabalho versa portanto sobre o projeto deste microprocessador dedicado para o emprego no arranjo, parte integrante da arquitetura prevista para o simulador *ABACUS* (hArdware BAsed CircUit Simulator) [1]-[5], denominação dada ao sistema de simulação. O algoritmo foi desenvolvido por Norian Marranghello em sua tese de doutorado no DSIF/FEE/UNICAMP [5] e será abordado no transcorrer deste capítulo.

O sistema inicialmente previsto para simulações de circuitos busca a exploração do paralelismo dos arranjos computacionais de uma forma mais ampla, através da redução do tempo de simulação.

Este capítulo apresenta a proposta do sistema de simulação de uma forma ampla, com o objetivo de trazer ao leitor o conhecimento sobre a finalidade do microprocessador aqui desenvolvido que corresponde ao processo MPH *(Model Processing Hardware),* como será definido a seguir.

I.2 - Fundamentos e Objetivos

O simulador de circuitos *ABACUS* é baseado em *hardware* e projetado com a finalidade de executar simulações de circuitos VLSI e ULSI em alta velocidade, em um nível de detalhes comparável aos simuladores convencionais de bom desempenho. Seu diagrama de blocos pode ser visto na figura 1.1.

I.3 – Arquitetura do Sistema *ABACUS*

A arquitetura do sistema *ABACUS* prevê que o usuário mapeie circuitos ULSI, simule circuitos menores, visando uma redução na complexidade do circuito total. Estas simulações parciais, em um estágio final, devem ser utilizadas para o cálculo do resultado total.

O algorítmo de simulação pode ser representado através de cinco processos principais, quatro dos quais (HCP, HIP, HOP, AMP) rodariam no computador hospedeiro, ou seja, o detentor do mapeamento do circuito total e das diversas partições do circuito, e o quinto rodaria em cada MPH.

Vejamos a função de cada um destes processos [5]:

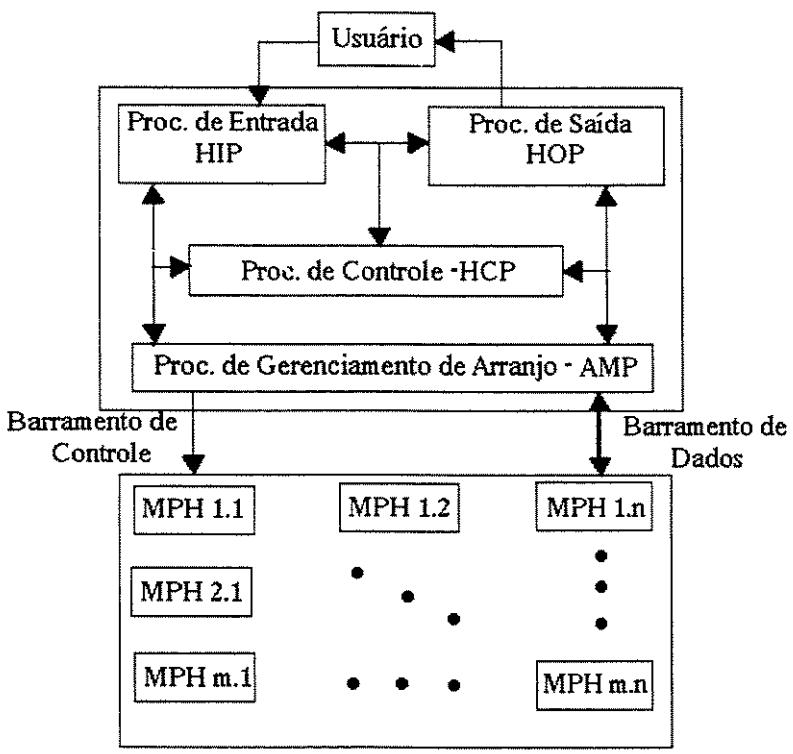


Fig. 1.1 - Arquitetura do "ABACUS"

Processo HIP *(Host Input Process)*

Processa os dados de entrada fornecidos pelo usuário, organizando-os para o tratamento pelos demais processos do simulador:

1 - Lê o arquivo de entrada

2 - Identifica os elementos do circuito

3 - Identifica as análises a serem feitas

4 - Monta a base de dados do circuito

5 - Monta o grafo de interconexões do circuito

6 - Verifica o número de MPHs disponíveis no arranjo

7 - Particiona o circuito em subcircuitos (se necessário), de acordo com o número de MPHs disponíveis e mantem os componentes fortemente conectados no mesmo sub-circuito sempre que possível

8 - Prepara cada (sub) circuito para simulação e transfere-o ao processo AMP (avisa o processo AMP a respeito disto)

9 - Finaliza o processo HIP e sinaliza o processo HCP

Processo HCP *(Host Control Process)*

Este processo encerra a operação do ambiente de simulação e ativa e desativa cada um dos demais processos do computador hospedeiro, sempre que necessário. Tem também a função de controlar a operação do sistema de simulação, não no sentido de sincronização dos demais processos mas como um gerente assíncrono dos mesmos.

As etapas de processamento desta unidade podem ser então resumidas como se segue:

1 - Inicia o processo de simulação

2 - Inicia o processo HIP

3 - Inicia o processo HOP

4 - Termina quando todos os processos estiverem concluídos

Processo HOP *(Host Output Process)*

Este processo realiza a função inversa do HIP, ou seja, a partir das soluções geradas pelo simulador e da base de dados organizada pelo processo HIP, fornece as respostas solicitadas pelo usuário. Suas funções podem ser resumidas como a seguir:

1 - Recebe a base de dados do HIP e formata-a para a saída

2 - Recebe os resultados da simulação do processo AMP e formata-os para a saída

3 - Quando os processos HIP e AMP sinalizarem o término de funcionamento, transfere os resultados finais formatados para a saída especificada pelo usuário

4 - Encerra o processo HOP e avisa o fato ao HCP

Processo AMP *(Array Manager Process)*

Tem como objetivo controlar o funcionamento do arranjo de processadores, ou seja, controla a operação do arranjo fundamentado nos dados que cada processador do arranjo fornece-lhe. Etapas deste processo:

1 - Verifica o número de MPHs disponível no arranjo, armazena-o e informa seu valor quando requisitado

2 - Recebe um circuito do processo HIP

3 - Mapeia o circuito nos processadores do arranjo

4 - Estabelece o estado inicial do arranjo

5 - Inicia o contador com o número de elementos do circuito mapeado

6 - Incrementa e decrementa o contador, conforme o estado de convergência dos processadores do arranjo, até zerar o contador

7 - Busca as soluções no arranjo

8 - Transmite as soluções ao processo HOP

9 - Repete os passos 2 a 8 até que todo o circuito seja simulado

10 - Encerra o processo AMP, avisando os processos HCP e HOP

Processo MPH *(Model Processing Hardware-element)*

Executa efetivamente a simulação do circuito, com base nos dados a ele fornecidos pelo processo AMP. Suas etapas podem ser assim descritas:

1 - Espera os dados do processo AMP

2 - Identifica o elemento a ser simulado

3 - Configura a ULA de acordo com o elemento

4 - Busca os dados de entrada da memória do hospedeiro

5 - Executa os passos:

5.1 - Busca novos dados nas filas de entrada

5.2 - Calcula o resultado correspondente ou decrementa o contador de AMP

5.3 - Salva o resultado parcial na memória local e transfere-os para as filas de saída ou transfere as soluções para o processo AMP

5.4 - Volta ao passo 5.1 ou passo 1

O processo de sincronização de tarefas dentro do sistema durante o processamento é obtido através de uma informação de uma "ficha" - *tag,* que o próprio dado carrega, semelhante aos computadores *data-flow.* Isto impede a ocorrência de superposição de eventos, o que conduziria a erros de simulação.

A arquitetura de arranjos de processadores é a forma de processamento escolhida para o projeto do arranjo de MPHs *(Model Processing Hardware-element).* Estes processadores dedicados à simulação de circuitos devem operar de modo assíncrono.

O fluxo de sinais computacionais nos arranjos MPHs é semelhante ao que ocorre em redes reais. A cada vez que um novo subcircuito é mapeado a configuração de

arranjos é colocada, por programação, na rede de chaveamento, uniformente distribuída por todo o arranjo, tal que o número máximo de interconexões possíveis seja conseguida com um número razoável de barramentos e circuitos de chaveamento.

Do exposto acima podemos concluir que cada MPH deve ter a estrutura de um microprocessador dedicado à realização de equacionamentos matemáticos que descrevam os modelos dos componentes eletrônicos presentes nos circuitos.

I.4 – Arquitetura do Microprocessador - MPH

A arquitetura inicialmente prevista para o microprocessador aqui desenvolvido, pode ser visualizada na figura 1.2, a seguir:

- Fila de Entrada/Saída (FIES)
- Unidade Lógica e Aritmética (ULA)
- Unidade Elementar de Controle (UCE)
- Unidade de Modelos Armazenados (UMA)
- Memória de Escrita e Leitura (MEL)

FIES - São *buffers* de alta velocidade, responsáveis pela comunicação necessária entre os MPHs, na etapa de entrada dos dados ou saída dos resultados.

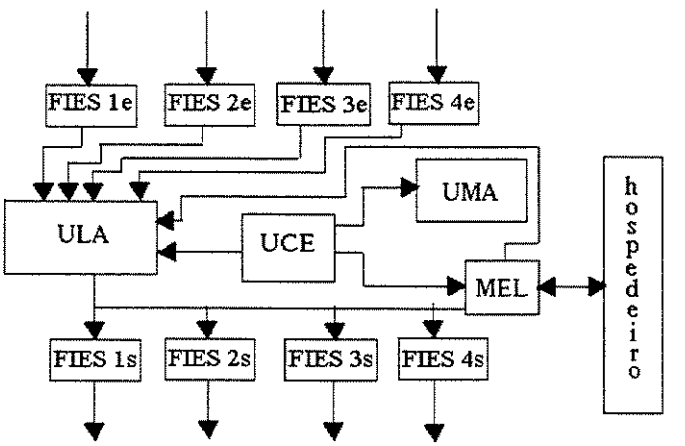


Fig. 1.2 - Arquitetura de um MPH

ULA - A Unidade Aritmética e Lógica é formada por somadores, subtratores, multiplicadores, divisores e unidades de deslocamento. 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 arranjo, ou seja, quando ocorre a mudança do circuito a ser simulado (programa presente na MEL, apresentada abaixo).

As operações matemáticas devem ocorrer com operandos representados em ponto flutuante, dado que os valores de componentes, suas correntes e tensões apresentam uma larga faixa numérica, sendo inviável suas representações por ponto fixo.

UCE - É uma unidade de controle cuja finalidade é controlar tanto o processamento dos MPHs como os seus interfaceamentos com o hospedeiro ou o resto do arranjo. A UCE controla três fluxos de informação: a sinalização para o gerente do estado de convergência do processador; a transferência das informações do gerente e dos resultados da simulação para o gerente e a comunicação dos resultados parciais da simulação entre os processadores. O controle interno de cada MPH é também realizado por esta unidade.

Neste trabalho, somente o controle interno individual de cada MPH será tratado, porque os demais controles, relacionados aos blocos externos ao MPH são realizados pelo controle do sistema *ABACUS* total.

No tipo de arquitetura prevista para este controle, por microprogramação, o procedimento para expansão do controle pode ser realizado de forma simples, como será esclarecido, no transcorer da dissertação.

UMA - É uma memória ROM – (*Read Only Memory* – memória somente de leitura) ou similar, de conteúdo endereçável, onde os diversos modelos conhecidos pelo *ABACUS* ficam alocados. Estes modelos podem ser definidos pelo usuário. Os modelos contidos na UMA são responsáveis pela configuração dos circuitos da ULA, durante a simulação de cada subcircuito, de acordo com o dispositivo a ser simulado.

Esta unidade deve ter armazenado um microprograma que aciona a ULA convenientemente, ou seja, a UMA deve armazenar o microprograma correspondente ao dispositivo simulado pelo MPH (resistor, capacitor, transistor, etc).

MEL - É uma memória de acesso aleatório, ou seja, uma RAM *(Random Access Memory)* utilizada para um 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 às 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 (FIES). Ao detetar 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 *protoboard.* Esta forma de análise vislumbra o paralelismo em sua expressão mais ampla, ou seja, 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 assincronamente tanto entre si como em relação ao hospedeiro.

I.5 - Ferramentas utilizadas para o desenvolvimento do projeto

Para o desenvolvimento do projeto utilizamos sofisticadas ferramentas de projeto. As descrições dos circuitos estão na linguagem VHDL [73] *((VHSIC - Very High Speed Integrated Circuits) Hardware Description Language)* que começou a ser desenvolvida em 1981 pelo *Program Office of Department of Defense of United States* com o objetivo de padronizar a linguagem de descrição de circuitos dentro da microeletrônica. Esta linguagem possibilita fazer descrições de circuitos desde altos níveis de abstração até a níveis de componentes, facilitando e documentando de forma significativa o trabalho do projetista de circuitos.

A disponibilidade do compilador desta linguagem *(Design Architect)* também como do simulador temporal - *Quicksim,* disponíveis no pacote de *software* da *Mentor Graphics,* na rede computacional do DSIF, da Faculdade de Engenharia Elétrica e de Computação, permitiram o desenvolvimento deste trabalho que envolve circuitos de alta complexidade.

I.6 - Comentários

Foram aqui expostos a arquitetura do sistema *ABACUS* de simulação, bem como os blocos previstos para o projeto do microprocessador MPH, objetivo deste trabalho. Estas informações nortearam a pesquisa de circuitos que devem ser utilizados em arquiteturas desta complexidade. Seguindo a disposição prevista na introdução, serão abordados no capítulo a seguir, os tratamentos técnicos referentes ao desenvolvimento destes circuitos.

Capítulo II

Fundamentos Teóricos do Projeto

II.1 - Introdução

Vamos aqui apresentar os fundamentos teóricos utilizados no projeto do circuito do microprocessador MPH, célula básica do arranjo de microprocessadores do *ABACUS* [5].

Inicialmente far-se-á um breve relato sobre formas de processamento paralelo, para situar o leitor dentro do contexto. É importante ressaltar que não se fará um prolongado estudo sobre a arquitetura prevista para o sistema *ABACUS*, porque a nossa atenção estará voltada para o projeto do microprocessador MPH, que já exige um amplo conhecimento técnico, na elaboração de seus blocos formadores.

O desenvolvimento do projeto do MPH, requer de forma abrangente, o conhecimento de arquiteturas computacionais muito elaboradas. Faz-se necessário portanto, um estudo detalhado dos blocos formadores da arquitetura:

- Unidade de controle projetada com técnicas de microprogramação,
- Unidade aritmética e lógica que deve operar em ponto flutuante, realizando as quatro operações por *hardware*,
- Memórias e demais blocos formadores do microprocessador.

É importante ressaltar que o enfoque principal deste trabalho é o desenvolvimento do circuito do microprocessador o qual tornará viável a realização do sistema *ABACUS* de simulação.

II.2 - Processamentos Computacionais

Os computadores tradicionais, realizam o processamento sequencial de tarefas. As instruções são executadas seguindo a ordem prevista pelo programador e é necessário que a execução de uma instrução seja concluída, para que outra venha a ser iniciada [7].

Estes sistemas ainda são muito utilizados; muitas formas de processamento paralelo têm sido desenvolvidas, nas quais os objetivos buscados são: reduzir o tempo de processamento e expandir a capacidade de processamentos.

Os computadores que operam com processamento paralelo podem ser divididos em três categorias [8]:

- Computadores *Pipeline* - Um computador *pipeline* realiza computações sobrepostas para explorar o paralelismo temporal.
- Arranjo de Processadores - Um arranjo de processadores usa diversas unidades aritméticas e lógicas para executar um paralelismo espacial.
- Sistemas de Multiprocessadores - Um sistema de multiprocessadores realiza paralelismo assíncrono através de um conjunto de processadores interativos com recursos compartilhados (memórias, base de dados, etc.).

Estas três formas de paralelismo não são mutuamente exclusivas. Há computadores formados por arranjo de processadores ou sistemas de multiprocessadores e que são também *pipeline*.

Devido ao aumento exagerado na complexidade da unidade de controle, não se utiliza o paralelismo de forma mais ampla. É tarefa do projetista buscar estruturas que permitam conciliar simplicidade e desempenho.

O microprocessador aqui apresentado é, como já citado, a célula básica de um arranjo de processadores dedicado à simulação de circuitos. O esquema apresentado na figura 1.1 elucida o fato de que o sistema em forma de arranjo, explora o paralelismo espacial em procedimentos para simulações de circuitos de alta complexidade.

Faremos, a seguir, um estudo teórico das diversas unidades que compõem o microprocessador, o que permitirá estabelecer sua arquitetura.

II.3 – Fundamentos sobre arquiteturas computacionais

Quando se projeta a arquitetura de um microprocessador [7],[9]-11] o primeiro passo é definir o caminho que os dados, endereços e resultados parciais de programas computacionais devem percorrer para que, através da execução das instruções, seja obtido como resultado final do processamento das tarefas previamente estabelecidas.

Para se compreender como os dados transitam pelos canais de informação e barramentos do microprocessador é necessário que se defina a sua arquitetura que é o meio físico que deve ser disponibilizado para que ocorra a troca de informações, de maneira organizada e previsível, entre os diversos blocos da máquina.

O meio físico ou circuitos, conhecido na literatura por *hardware*, deve ser projetado de forma a receber das vias de comunicação os sinais que tornam possível a realização de tarefas descritas em forma de programas, ou seja, o *software*.

A arquitetura de um microprocessador de uma forma geral compreende os seguintes blocos principais [7]:

- Entrada e Saída - responsáveis pelo interfaceamento do microprocessador com o seu meio externo;
- Unidade Central de Processamento (UCP) - formada pelos seguintes blocos:
  - Unidade de Controle - realiza o gerenciamento de execução de tarefas de forma ordenada;
  - Fluxo de Dados - constituído pelos seguintes sub-blocos:
    - Unidade Aritmética e Lógica - formada por circuitos que possibilitam a execução de operações matemáticas e lógicas;
    - Registradores – unidades de armazenamento temporário de dados e instruções utilizados no processamento.
- Memória Principal – utilizada para o armazenamento do programa a ser executado pelo microprocessador;

Apresentaremos, a seguir estes blocos, enfocando suas principais características.

II.4 –Entrada e Saída

As filas ou *buffers* de entrada/saída, são formadas por registradores para o armazenamento temporário dos dados necessários para o processamento. Devem ser projetados de forma a possibilitar um rápido acesso aos dados neles contidos.

II.5 - Unidade Central de Processamento

II.5.1 – Unidade de Controle

A finalidade principal da Unidade de Controle é gerenciar a execução das tarefas, que são impostas à máquina, na forma de instruções previamente armazenadas em memórias. É o órgão principal do processador que comanda todas as unidades previstas na arquitetura.

Um processamento convencional para a realização de uma instrução compreende três etapas:

- Busca da Instrução
- Decodificação
- Execução

A busca da instrução consiste na captação, na memória que contém o programa a ser executado, da palavra de instrução. Nesta é encontrada, sob a forma de código de operação e de maneira implícita, a forma como os operandos, endereços ou endereços de dados devem ser trabalhados.

A seguir, esta instrução passa pelo processo de decodificação ou interpretação, para definição dos controles que realizem microoperações que levem à execução da instrução.

A execução da instrução se concretizará após o estabelecimento de todas as etapas previstas pelo seu código.

Para se definir a seqüência de microoperações necessárias para a execução das instruções é necessário que se defina a forma pela qual a transferência dos dados ou endereços deve ser feita ou controlada, em intervalos de tempo convenientes.

A seguinte metodologia pode ser utilizada [7]:

- Analisam-se as exigências de transferência dos dados e endereços entre registradores
- Estabelecem-se os requisitos de fonte e destinação da Unidade Lógica e Aritmética.
- Simplificam-se os caminhos de dados, dentro das exigências de simultaneidade.
- Consideram-se o *clock* do fluxo de dados, as localizações, os atrasos e as possíveis necessidades dos registradores para reter os dados.
- Analisam-se os atrasos de *gate* do fluxo de dados e estima-se um tempo de ciclo mínimo do fluxo de dados que seja confiável.

Após definida a seqüência de execução das instruções, faz-se necessária a escolha da forma de controle a ser utilizada. Diversas são as técnicas conhecidas para realização desta etapa [9],[12]-[14], porém os controles podem ser classificados [13] em duas categorias:

- *hardwired*
- por microprogramação

Os controles *hardwired* são realizados a partir de:

- máquinas de estado de Moore e Mealy [9];
- circuitos combinacionais tais como: redes lógicas, multiplexadores e decodificadores.

A microprogramação [12],[13] utiliza-se de palavras de controle para o acionamento dos blocos formadores da arquitetura e tem se destacado em controles de alta complexidade. Foi a forma escolhida para o gerenciamento das instruções do nosso microprocessador e merecerá uma especial atenção nesta tese. Vejamos, a seguir, seus fundamentos e características.

II.5.1.1 - Microprogramação

A microprogramação foi definida por Maurice V. Wilkes da Universidade de Cambridge em 1950, como uma técnica utilizada para fazer a unidade de controle mais sistemática e flexivel [14].

Circuitos de controle projetados por microprogramação recebem instruções, interpreta-as e através de toda uma arquitetura particular, as transforma em sinais necessários ao controle da máquina.

O controle de informação é armazenado na forma de microprograma - sequências de microinstruções armazenadas em uma memória de controle.

Há geralmente um microprograma, ou seja, palavras de controle definidas e distintas para cada tipo de instrução, onde microinstruções são usadas em funções de *fetching* (procura) de instrução, de endereço ou operando e processos de interrupção. É uma filosofia diferente das unidades de controle que usam PLAs- *Programmable Logic Array* [9],[18] ou máquinas de estado, onde através de estruturas lógicas são construídos circuitos capazes de executar sequências necessárias para a realização de tarefas.

Na microprogramação o controle se dá por programa, viabilizando controles com maior número de opções ou complexidade, daí o seu uso inclusive em sistemas *pipeline* [8], onde problemas de acessos simultâneos ocasionados pela utilização de uma mesma unidade de processamento por tarefas concorrentes ocorrem com muita facilidade.

A principal exigência para controle microprogramável é que ele proporcione os vários sinais de controle necessários para cada estágio na sequência apropriada.

Um microprograma é executado procurando suas microinstruções uma a uma armazenadas em uma memória de controle. Cada microinstrução é carregada dentro do registro de microinstrução, de onde os correspondentes sinais de controle são obtidos, ocasionalmente depois de uma etapa de decodificação.

A figura 2.1, apresenta um diagrama de blocos [13] de um controle microprogramado clássico, que servirá de referência para a explicação de um circuito com este propósito. Nesta estrutura o registro de instrução IR é utilizado para armazenamento temporário da instrução que terá o seu código de operação decodificado pelos circuitos de decodificação. Estes fornecerão ao multiplexador (mux) de endereço (end)., o endereço da palavra de controle da instrução presente.

O endereço selecionado é temporariamente armazenado no registro de endereço da microinstrução para ser buscado na memória de controle.

A palavra de controle apresenta campos de próximo endereço e para controles gerais, denominados por F_1 a F_n. O decodificador de microinstrução acionará convenientemente as unidades correspondentes para a realização da instrução como, por exemplo, habilitando registradores de uso comum R1 ou R2 como fonte e R3 ou R4 como destino de dados.

Nesta forma de representação da palavra de controle, o registro de próximo endereço do microprograma é parte integrante da palavra e faz referência à posição na

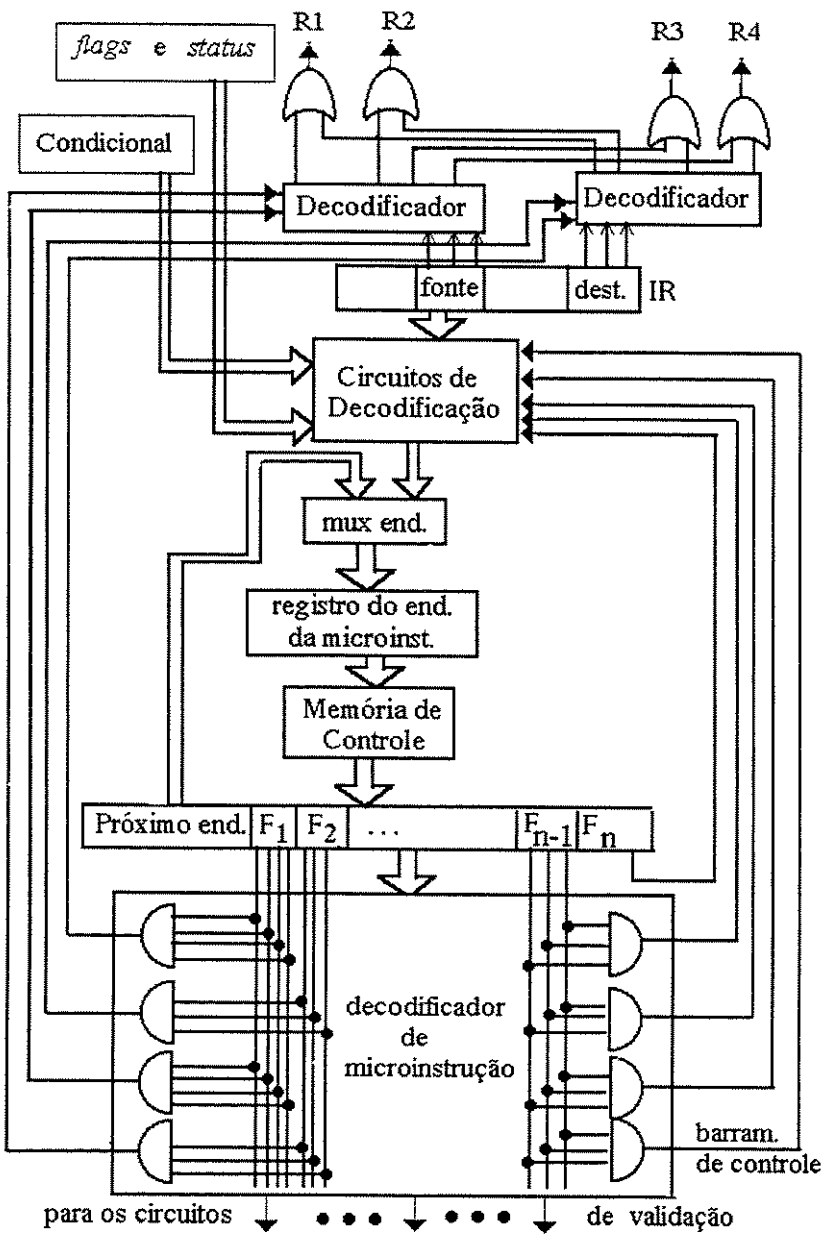


Fig. 2.1- Diagrama de blocos de um controle microprogramado clássico

memória de controle, da próxima microinstrução a ser usada.

Análises condicionais e ocorrência de *flags* ou *status* podem produzir desvios na seqüência de execução das instruções.

Em casos onde instruções necessitam de diversas etapas para completa execução, a seqüência de palavras de controle é dada através dos endereços previamente definidos pelo programador, no campo de próximo endereço utiliza-se o incremento de um para a obtenção das diversas etapas de execução e em caso de desvios condicionais, a unidade que realizou o teste da condição é que deve provocar a mudança na seqüência de leitura das palavras de controle.

É uma prática usual colocar a etapa de busca da instrução no endereço zero da memória de controle permitindo que, a cada final de execução de uma instrução o desvio ocorra para aquela posição [13].

É importante que se perceba a distinção entre instrução para máquina e para microprograma. Consideremos por exemplo, uma instrução típica de máquina para um microprocessador de 16 *bits* onde 6 *bits* são utilizados para código de operação possibilitando a definição de 64 (2^6) instruções. Estas instruções serão decodificadas e uma máquina de estado será acionada para a realização das mesmas.

A instrução típica de microprogramação, por outro lado, pode conter um número maior de bits da ordem de 30 a 128 *bits* ou até mais *bits* em máquinas mais velozes [14]. Os diversos campos da palavra de controle microprogramada são usadas diretamente para controlar os vários dispositivos dentro da máquina.

Esta forma de controle é mais direta que o uso de lógica combinacional e conduz a projetos mais organizados[14].

Uma microinstrução geralmente tem duas partes a serem observadas:

- a definição e controle de todas as microoperações elementares a serem executadas.
- a definição e controle do endereço da próxima instrução a ser executada.

As ações controladas por microinstruções tomam a forma de microoperações que podem ser realizadas por *hardware*.

Estas microoperações incluem carregamento de registros, roteamento de dados via multiplexadores, decodificadores, barramentos, etc. Cada microinstrução define um conjunto de uma ou mais microoperações que podem ser realizadas simultaneamente em um ciclo de microinstrução.

Os projetos de microprogramas são geralmente implementados em uma linguagem simbólica chamada de *microassembly language*. O código fonte escrito nesta linguagem é

posteriormente traduzido por um programa *microassembler* e armazenado em uma memória de controle.

Como os microprogramas constituem uma forma de *software* mais próxima do *hardware*, sendo controlada por programas de instruções, são geralmente conhecidas como *firmware*.

Podemos ainda citar algumas características importantes na microprogramação [13]:

1. Os controles das funções são implementados por *software (firmware)* ao invés de *hardware*.
2. O processo do projeto é ordenado e sistemático.
3. Mudanças para acomodar novas especificações do sistema ou para corrigir erros de projeto podem ser implementadas rápida e economicamente.
4. Funções complexas tais como aritmética com ponto flutuante podem ser implementadas mais eficientemente por *firmware* do que por rotinas de instruções.
5. Microprogramação pode resultar em alguma redução na velocidade de processamento da instrução.

Embora a quinta característica apresente uma desvantagem desta técnica, este fator é compensado pela forma versátil e prática com que se pode realizar controles principalmente de máquinas complexas.

Escolhemos esta técnica devido a complexidade do circuito de controle para nosso microprocessador, como também em razão da necessidade de se conseguir um acionamento direto e imediato do controle a partir dos modelos de descrição dos componentes eletrônicos, buscando assim uma redução no tempo de processamento da simulação dos mesmos.

A primeira hipótese levantada para a utilização da microprogramação surgiu devido a constatação de que, por serem fixos os modelos dos componentes, este tipo de controle tomaria uma forma organizada, acelerando o processo de conclusão das simulações.

Posteriormente, modificamos a utilização da memória de modelos armazenados UMA, tornando-a retentora não mais dos modelos fixos mas, de controles responsáveis pela execução das instruções. Esta mudança contudo, não inviabiliza a forma de controle por microprograma mas, vem ampliar o espectro de utilização do projeto para os mais diversos tipos de aplicações.

II.5.2 – Fluxo de Dados

II.5.2.1- Unidade Lógica e Aritmética - ULA

Este bloco apresenta significativa importância na formação de uma arquitetura em computadores tanto sequenciais como por processamento paralelo, onde os primeiros constituem a base para o desenvolvimento de todos os sistemas computacionais, podendo estes conceitos serem ampliados para projetos de estruturas mais sofisticadas como é o caso dos que operam por processamento paralelo.

As operações geralmente realizadas por ULAs em microprocessadores, de finalidades matemáticas, são dentre outras as mais comuns: lógicas (*NAND, NOR* e *XOR*, etc.), deslocamentos e operações aritméticas, tais como, soma, subtração, multiplicação, divisão, etc. As operações matemáticas podem ser realizadas em ponto fixo ou ponto flutuante dependendo das aplicações previstas para o processador desenvolvido.

Em nosso projeto propomos uma ULA que realize as operações matemáticas necessárias à resolução de expressões que descrevam o comportamento matemático dos elementos de circuitos elétricos. Optamos pela resolução das quatro principais operações matemáticas por *hardware*.

Os valores de componentes, correntes e tensões a serem utilizados neste microprocessador podem utilizar uma ampla faixa de ordens de grandeza, portanto, as operações matemáticas devem ser processadas em ponto flutuante já que esta forma de representação, como veremos em breve, apresenta capacitação adequada para estes casos.

Como veremos, em etapas posteriores desta tese, o procedimento para o cálculo de resultados, quando as operações transcorrem em ponto flutuante, envolve importantes particularidades e mereceram uma descrição pormenorizada.

As operações aritméticas propriamente dita são semelhantes tanto quando trabalhamos com ponto fixo como ponto flutuante porém, neste segundo caso necessita-se de todo um tratamento numérico para a obtenção de resultados intermediários o que exige um alto grau de sofisticação do conjunto de circuitos dedicados para esta finalidade.

Veremos inicialmente, de forma sucinta, alguns conceitos importantes para a compreensão dos passos que, virão a constituir o projeto de uma unidade aritmética e lógica nos moldes aqui citados.

II.5.2.1-1 - Representações Numéricas Binárias de Números Inteiros Positivos e Negativos

A nossa matemática convencional utiliza-se do sistema decimal (que não foi escolhida por acaso - dez é quantidade de dedos da mão) de representação numérica devido a grande facilidade mental de manipulação dos operandos quando se deseja realizar qualquer operação matemática. Esta forma de representação contudo, não se mostrou adequada quando se pretendeu projetar circuitos eletrônicos dedicados à resoluções matemáticas. Buscou-se na representação binária de sinais a facilidade de implementação utilizando o manuseio de circuitos aliado à alta versatilidade na aquisição de estruturas matemáticas. A álgebra booleana ou binária veio permitir um significativo avanço em projetos nas mais variadas áreas de aplicações.

Ao longo da história da engenharia eletrônica, surgiram diversas codificações binárias que sempre buscaram facilitar a representações de valores passíveis de modelamentos lógicos.

Vamos aqui apresentar as formatações mais utilizadas de representações numéricas que foram sendo padronizadas ao longo dos anos, como uma maneira de tornar homogênea a descrição de valores viabilizando os tratamentos algébricos por meio de máquinas.

Dado que diversas são as formas de representação numérica, é de fundamental importância que se tome conhecimento de qual formato está sendo utilizado durante todo o procedimento de desenvolvimento de projetos aritméticos para que se possa referenciar de maneira correta todo o processo utilizado para cálculos matemáticos.

Vamos abordar as seguintes representações numéricas binárias [7],[9]:

- Sinal e magnitude
- Complemento de um
- Complemento de dois

A representação por sinal e magnitude considera o *bit* mais significativo como referente ao sinal do operando, sendo 0 para positivo e 1 para negativo. Os demais *bits* indicam o valor numérico propriamente dito.

Os valores estão compreendidos no intervalo de $(-2^{(n-1)} - 1)$ a $(2^{(n-1)} - 1)$, onde n é o número de bits utilizado para a representação binária.

Embora esta forma se caracterize por sua grande simplicidade, apresenta o inconveniente de possuir duas representações para o zero, ou seja, teríamos o zero positivo e o negativo. Este fato provoca uma dificuldade acentuada principalmente na realização de operações de adição e subtração.

O Complemento de Um consiste na complementação, ou melhor, na negação de todos os bits formadores do valor positivo, para representá-lo na forma negativa. Surge aqui o mesmo problema ocorrido no caso anterior, ou seja uma representação dupla para o valor zero. O intervalo de valores representáveis é o mesmo do anterior.

O Complemento de Dois consiste em somar mais 1 no *bit* menos significativo do complemento de Um do número. Esta forma veio solucionar o problema surgido nas representações anteriores, com uma única representação para o zero e apresenta valores no intervalo de $(-2^{(n-1)}$ a $(+2^{(n-1)}-1)$.

Os formatos sinal e magnitude e complemento de dois serão utilizados intensivamente durante todo o procedimento de tratamento dos operandos dentro das unidades operadoras de processamentos matemáticos.

II.5.2.1-2 - Operações Aritméticas em Ponto Fixo

II.5.2.1-2.1 - Soma e Subtração em Ponto Fixo

Estas duas operações podem ser tratadas simultaneamente porque a partir de um controle, que indica o sinal de um dos operandos se positivo ou negativo, pode-se identificar se tratar de uma operação de soma ou subtração respectivamente. Isto facilita a implementação destes dois operadores de uma única vez.

Para se projetar um somador ou qualquer outro operador matemático partimos inicialmente da tabela verdade, construída sob os princípios da álgebra booleana em conformidade com as expressões matemáticas que satisfaçam a realização da operação desejada. O circuito utilizando portas lógicas, deve obedecer a estas expressões, gerando assim a operação matemática propriamente dita.

Inicialmente vejamos a forma mais simples de soma de dois valores, ou seja, um meio somador ou *half adder,* como é denominado na literatura de circuitos digitais [15],[16].

A Tabela 2.1 mostra os possíveis estados do meio somador e a Fig.2.2 apresenta um circuito lógico que implementa esta função.

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Tabela 2.1 – Tabela Verdade do meio somador

Na tabela 2.1 A e B correspondem aos operandos de entrada, S é o resultado da soma e C é o *carry*, ou seja, o "vai um" da operação entre os dois operandos.

Por análise, dos termos da expressão lógica que define o resultado da soma S, podemos verificar que a soma consiste da operação lógica *EXCLUSIVE-OR* dos dois operandos e o *carry* é formado pela operação *AND* entre as duas entradas do somador, que simbolicamente representamos por:

S = A *xor* B

C = A *and* B

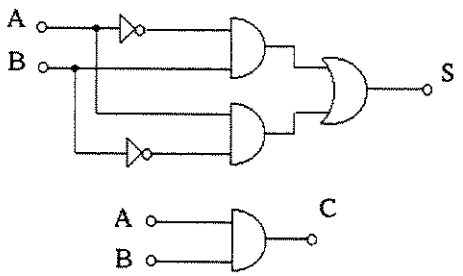


Fig. 2.2 - Meio Somador - *Half Adder*

A ilustração da figura acima, recebe a denominação de meio somador, porque não prevê a propagação do termo "vai um", ou seja, o *carry*.

Uma configuração mais completa, que segue o comportamento apresentado pela tabela 2.2, prevê a propagação das componentes "vai um" e constitui um somador *Full Adder*, ou seja, somador completo.

Estes somadores (Fig. 2.3) podem ser facilmente obtidos a partir de circuitos lógicos combinacionais, o que lhes atribui a denominação de somadores combinacionais *(combinational adder)*.

A	B	Cin	Soma	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tabela 2.2 – Tabela Verdade do somador completo

A expressão para a soma, pode ser então obtida:

Soma = A *xor* B *xor* Cin

ou seja, a soma é o resultado da operação lógica *XOR* ou *EXCLUSIVE-OR* dos dois termos A e B com o Cin (*carry* de entrada) resultado da soma de parcelas anteriores.

Esta expressão pode ser reescrita a partir de seu equivalente lógico utilizando portas *AND* e *OR*, resultando:

$$\text{Soma} = \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + ABC$$

Obs: Os grifados correspondem aos sinais complementares,

O *Cout* (*carry* de saída), ou seja, o "vai um" gerado pela soma é dado por:

Cout = AB + ACin + BCin

Estas são as células básicas utilizadas para a soma de dois *bits* de operandos distintos. Esta estrutura pode ser repetida na intenção de se obter somadores com um maior número de *bits*. Através da repetição destas células pode-se também implementar multiplicadores, como veremos oportunamente.

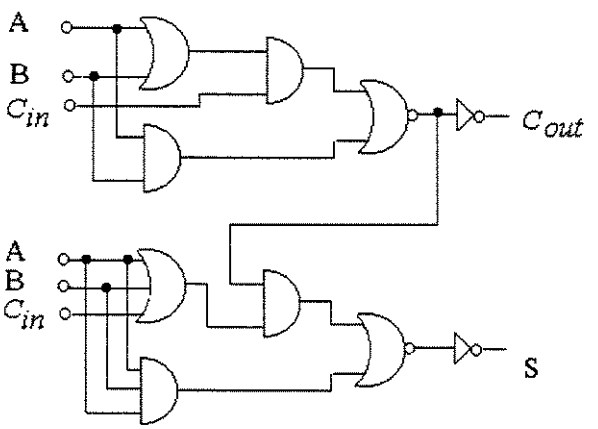


Fig. 2.3 - Somador Completo - *Full Adder*

A soma de vários *bits* pode ser realizada pelo simples cascateamento destes somadores onde o sinal de *carry* se propaga pelos estágios, ou seja, a entrada do estágio posterior recebe sinal de *carry* anterior. Isto significa que o sinal de *carry* **modula** o sinal de entrada. Por este motivo este tipo de circuito é também conhecido por *ripple adder* (somador modulado).

Embora práticos estes somadores apresentam um forte atraso na propagação do sinal de *carry* pela rede de somadores, o que os tornam inadequados para aplicações onde um maior número de *bits* e velocidade de execução da operação de soma são necessários. A figura 2.4, a seguir, apresenta este tipo de arranjo para n *bits*.

Nesta estrutura, um grande atenuador na velocidade de execução de operações de soma é o elemento de *carry* - Cn que deve se propagar de forma rápida por toda a rede somadora, atingindo com isso uma otimização na velocidade de obtenção do resultado.

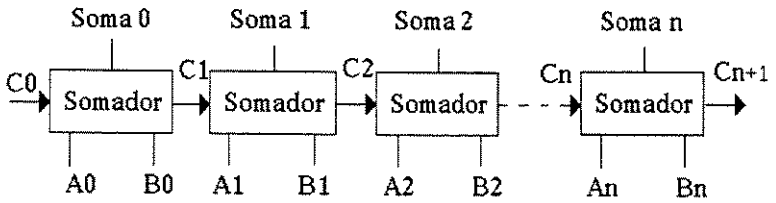


Fig. 2.4 - Somador *ripple carry* de n *bits*

Estes problemas podem ser solucionados por várias técnicas [17] : *carry-lookahead*, *carry-skip*, *carry-predict*, *carry-select*, também com combinações entre essas técnicas que objetivam a redução no tempo de propagação do *carry*, onde estes fatores são tratados de forma diferenciada à prevista por *ripple carry*. O estudo destas técnicas é muito extenso e foge ao objetivo principal deste trabalho, podendo ser minuciosamente conhecido na referência dada acima.

Vamos nos restringir à analise da técnica de *carry-lookahead*, para o tratamento da propagação do *carry*, por ser uma técnica clássica, simples e de eficiência comprovada.

As expressões para soma S e *carry* de saída - Cout de cada parcela, são válidas, independentemente da técnica adotada, contudo, o cálculo antecipado do *carry* de agrupamentos de parcelas, agilizam em muito a obtenção do resultado do *carry* da soma final.

Para tanto iniciemos o processo de análise do somador a partir da observação das propriedades do sinal de *carry*.

Pela tabela verdade anteriormente exposta notamos o seguinte: em situações nas quais as entradas A e B são nulas o *carry* de saída também é nulo. Quando tanto A quanto B são iguais a 1, o *carry* de saída também é igual a 1 e para A e B diferentes entre si o *carry* de saída é igual ao da entrada. Estas propriedades levaram os projetistas a caracterizarem dois tipos de *carries*: *Propagate* (P) e *Generate* (G), sendo:

P= A *xor* B e G = A *and* B.

Desta forma uma nova tabela (Tabela 2.3) pôde ser elaborada, onde se observam as diversas expressões que caracterizam um somador completo. Além das expressões de soma e *carry* de saída, já anteriormente constatadas, dispomos agora dos parâmetros P e G para uma melhor avaliação de como se comportam os *carries* envolvidos em somadores com qualquer número de *bits*.

Visto que atrasos relacionados à propagação do *carry* podem se tornar significativos quando há um maior número de *bits*, podemos dividir os *bits* formadores dos operandos em estágios e resolver as parcelas destes estágios simultaneamente, agilizando o processo de obtenção do resultado final.

A	B	Cin	P	G	S	Cout
0	0	0	0	0	0	0
0	0	1	0	0	1	0
0	1	0	1	0	1	0
0	1	1	1	0	0	1
1	0	0	1	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	1
1	1	1	0	1	1	1

Tabela 2.3 – Tabela verdade para o somador completo com avaliação de *carry generate* (G) e *propagate* (P)

A expressão que define o *carry*, como analisado anteriormente, é dada por:

$Cout = AB + (A+B)Cin$

O *carry* devido ao iésimo estágio pode ser expresso por:

$Ci = Gi + Pi \,.\, Ci\text{-}1$ (2.1)

onde

$Gi = Ai \,.\, Bi$ (*carry* gerado)

$Pi = Ai \oplus Bi$ (carry propagado)

$Ci\text{-}1 = Gi\text{-}1 + Pi\text{-}1 \,.\, Ci\text{-}2$ (2.2)

e assim por diante, por indução, chegamos a uma expressão geral que define a expansão dos *carries* dada por:

$Ci = Gi + PiGi\text{-}1 + Pi\ Pi\text{-}1\ Gi\text{-}2 + \ldots + Pi \ldots P1\ C_0$ (2.3)

e para a soma,

$Si = Pi \oplus Ci{+}1$ (2.4)

Estas expressões condensam o comportamento de somadores *carry-lookahead* e serão amplamente utilizadas na descrição de circuitos que envolvam somas binárias neste projeto.

II.5.2.1-2.2 - Multiplicação em Ponto Fixo

A multiplicação em sua forma mais simples é realizada envolvendo dois operandos. Seguindo o conhecimento básico de execução desta operação, sabemos que ela é realizada a partir de operações de somas e deslocamentos. O processo se constitui portanto, de duas etapas:

- execução de produtos parciais
- alocação de produtos parciais deslocados

A execução dos produtos parciais consiste da operação *AND* de cada bit do multiplicando por todos do multiplicador seguindo-se deslocamentos para alocação dos produtos parciais de acordo com a posição ocupada pelo *bit* do multiplicador envolvido.

Após a realização dos produtos parciais deve-se proceder os deslocamentos convenientes, para que se processe a soma dos subprodutos de forma correta.

Há várias técnicas para se realizar multiplicações e são inúmeras as arquiteturas elaboradas para esta finalidade, sendo extensa a literatura disponível[16],[18]-[24].

Características tais como, velocidade, precisão numérica, compactação, e multiplicidade de execução de operações entre outras é que nos conduzem a escolha de determinada arquitetura.

O desempenho deve pesar na escolha da complexidade do circuito.

Podemos optar por diversas formas de execução de operações de multiplicação, como exposto a seguir [18]:

- serial
- serial/paralela
- paralela

Serial – A forma mais simples de multiplicador serial, ilustrada na figura 2.5, opera

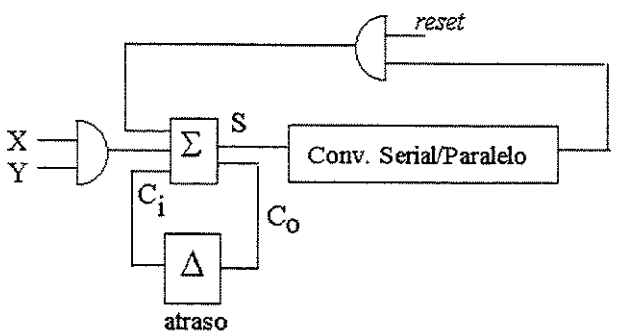


Fig.2.5 - Um multiplicador serial básico

através de somas sucessivas realizadas por somadores completos– *full adder*. Embora seja facilmente concebível, por se tratar de uma estrutura serial, apresenta significativo comprometimento na velocidade de operação.

Os dados X,Y,Ci são injetados na unidade somadora Σ de forma sequencial, em intervalos de tempos diferenciados para multiplicando e multiplicador. Ocorre um atraso Δ, no rearmazenamento do *carry* resultante em cada etapa do resultado parcial.

Os resultados parciais são armazenados temporariamente no conversor serial/paralelo até que todos os *bits* envolvidos na multiplicação sejam convenientemente processados. O sinal de *reset* é acionado a cada início de processamento, para inicialização da geração do resultado.

Serial/Paralela – Fazendo uso de várias unidades da forma anteriormente citada, pode-se obter um produto a partir da soma de sucessivas colunas da matriz de produtos

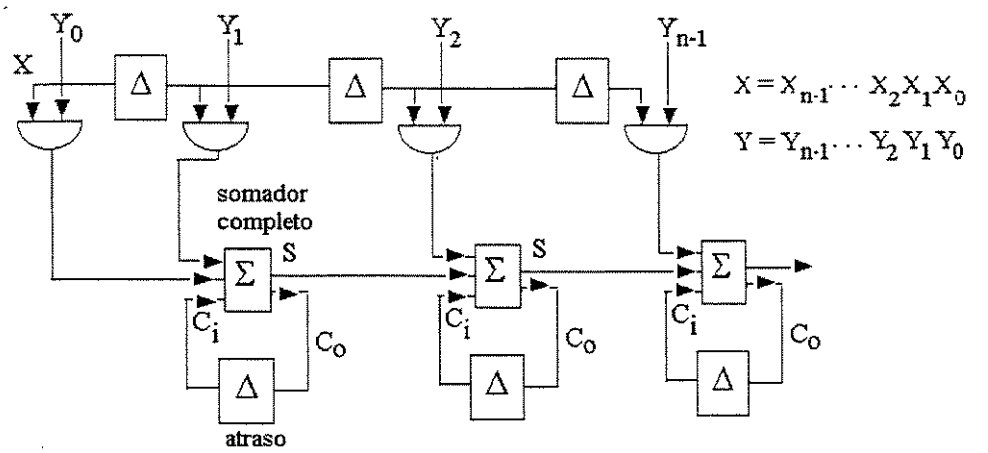


Fig. 2.6 - Estrutura básica para um multiplicador serial/paralelo

parciais deslocados, fig.2.6. Aqui a principal limitação é a freqüência máxima de operação que é limitada pelo tempo de propagação, pelo arranjo de somadores.

Paralela – Esta forma de obtenção de produto se baseia na possibilidade de que produtos parciais são independentemente calculados e podem ser computados paralelamente. A figura 2.7 mostra uma célula simplificada que pode ser utilizada para construir um multiplicador paralelo. O termo Xi de um dos operandos se propaga verticalmente, enquanto que o termo Yi correspondente ao outro operando e se propaga horizontalmente. C_i corresponde ao *carry* de entrada, P_{i+1} ao produto parcial e C_{i+1} ao *carry* resultante obtido em cada célula.

Vejamos, por exemplo, a multiplicação de dois operandos X e Y, que podem ser representados por uma somatória de dígitos binários, de acordo com a representação usual.

$$X = \sum_{i=0}^{m-1} X_i 2^i \quad , \quad Y = \sum_{j=0}^{n-1} Y_j 2^j$$

O produto de X por Y fica determinado por P_t:

$$P_t = X.Y = \sum_{i=0}^{m-1} X_i 2^i . \sum_{j=0}^{n-1} Y_j 2^j = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (X_i Y_j) 2^{i+j} = \sum_{k=0}^{m+n-1} P_k 2^k$$

Onde P_k são os produtos parciais. Cada um dos termos dos subprodutos é produzidos por portas *AND*. Há m x n destes subprodutos produzidos de forma paralela. O problema básico em projetos para alta velocidade é reduzir o tempo de somar os 1s em cada coluna da matriz de subprodutos resultante. Um multiplicador de n x n algarismos [18] necessita de n(n-2) somadores completos, n meio somadores e n^2 portas *AND*. O atraso de pior caso associado com tal multiplicador é $(2n + 1).\tau_g$, onde τ_g é o atraso de cada somador nestas condições.

As estruturas da fig.2.7 podem ser agrupadas, como ilustrado na fig.2.8, de forma a

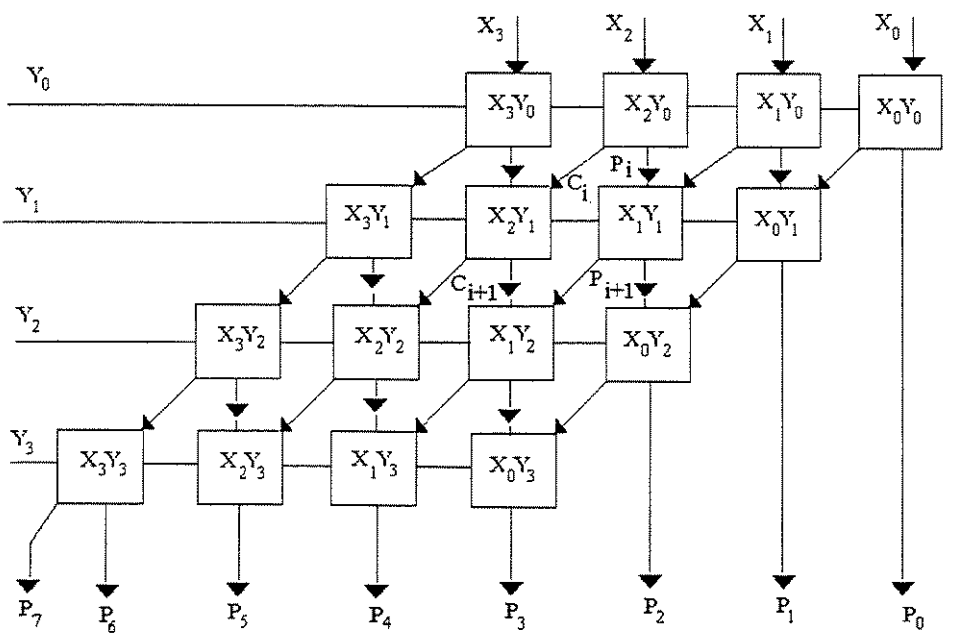


Fig. 2.8 - Arranjo multiplicador paralelo

se obter o produto de vários *bits*, de forma simplificada.

Um fator de atenuação na velocidade de execução da operação nestes multiplicadores é o que se refere ao processo de soma dos diversos fatores de uma mesma coluna obtida dos produtos parciais, devido a presença dos *carries* propagados.

A solução para este problema consiste em se utilizar um arranjo de *carry lookahead* para reduzir o tempo de execução da operação.

II.5.2.1-2.3 - Divisão em Ponto Fixo

Várias são as técnicas utilizadas para a obtenção da divisão [16],[19],[21],[22]. Geralmente os processos de divisão consistem de três partes: inicialização dos operandos, geração do quociente e determinação do resto.

Na inicialização dos operandos procede-se a padronização do dividendo e divisor, ou seja, a definição dos operandos em algum sistema de representação numérica e a avaliação do possível *overflow* do quociente.

A geração do quociente consiste na seleção dos dígitos do quociente sequencialmente do mais significativo ao menos significativo.

A obtenção do resto se dá, em geral, automaticamente ao final do processo de geração do quociente.

Os processos de divisão podem ser classificados, dependendo dos valores permitidos de cada dígito do quociente gerado, em duas principais categorias [16]: *restoring* ou *nonrestoring*, ou seja, por rearmazenamento ou não, respectivamente.

Em uma divisão por rearmazenamento - *restoring* com base ou raiz - *radix r* , cada dígito do quociente é selecionado do conjunto convencional de dígitos

{ 0,1,2, . . . , r-1}. O valor de cada dígito de quociente q_{j+1} para j = 0,1, . . . n-1 é selecionada para satisfazer a equação:

$0 \leqslant R_{j+1} < D$, onde R é o resto

O critério de seleção de quociente pode ser implementado com sucessivas subtrações do divisor D do dividendo parcial corrente r x R_j, até que a diferença se torne negativa. O número de subtrações realizadas antes que o resto se torne negativo determina o valor do dígito do quociente a ser selecionado.

Vejamos um exemplo:

Dados:	dividendo $(0.1257)_{10}$	
	divisor D = 0.39 para r = 10	
resulta em:	1.257 - 0.39 = 0.867	(> 0) - primeira subtração
	0.867 - 0.39 = 0.477	(> 0) - segunda subtração
	0.477 - 0.39 = 0.087	(> 0) - terceira subtração
	0.087 - 0.39 = - 0.69	(< 0) - quarta subtração

resultando $q_1 = 3$ (número de vezes, com resultado positivo, que o divisor foi subtraído do dividendo)

restoring ou rearmazenamento para adição

0.087 -> 0.87

0.87 - 0.39 = 0.48 (> 0) - primeira subtração

0.48 - 0.39 = 0.09 (> 0) - segunda subtração

0.09 - 0.39 = - 0.30 (< 0) - terceira subtração

resultando $q_2 = 2$ (idem ao anterior)

Portanto o quociente será : 0.32

O resto será 0.09 x 10^{-2}

Na divisão *nonrestoring* os dígitos do quociente são selecionados do conjunto de dígitos sinalizados, com zero excluído. A operação a ser realizada por cada resultado parcial do arranjo *nonrestoring* é adição ou subtração dependendo se o sinal do resultado anterior é o mesmo sinal do dividendo.

$\{-(r-1), -(r-2), \ldots, -1, +1, \ldots, r-2, r-1\}$

Exemplo:

Dividendo N = $(0.101001)_2 = (41/16)_{10}$

Divisor D = $(0.111)_2 = (7/8)_{10}$

Subtração: 0.101001 - 1.001 - divisor em complemento de 2

= 1.110001 < 0 $q_0 = 0$ - resto parcial negativo

Resto deslocado: 1.10001

1.10001 + 0.111 = 0.01101 > 0 $q_1 = 1$

Resto deslocado: 0.1101

0.1101 - 1.001 = 1.1111 < 0 $q_2 = 0$

Resto deslocado: 1.111

1.111 + 0.111 = 0.110 > 0 $q_3 = 1$

Quociente Q = $q_0q_1q_2q_3 = (0.101)_2 = (5/8)_{10}$

Resto = $(0.00r_3r_4r_5r_6)_2 = (0.000110)_2 = (6/64)_{10}$

Por análise da exposição sobre as operações de multiplicação e divisão, concluímos que as duas operações podem ser solucionadas utilizando circuitos de soma ou subtração contínuas, respectivamente.

Das opções de arquitetura de circuitos apreciadas nas diversas estruturas, que permitem a realização destas operações, escolhemos a referência [19], por esta apresentar uma forma circuital muito transparente de visualização e com excelente nível de precisão de resultados.

Esta escolha, embora muito interessante pela precisão apresentada, mostrou-se de difícil obtenção em termos de descrição de circuito na linguagem VHDL, tendo exigido particular atenção na definição de índices e expressões lógicas, vindo a culminar na criação de uma variação na arquitetura exposta na referência. No capítulo III será apresentado o diagrama da referência citada e discutidas as modificações realizadas.

As formas de obtenção de operadores vistas até aqui constituem os casos clássicos de operadores com ponto fixo, ou seja, aqui os operandos são tratados como números inteiros e as operações podem ser realizadas normalmente, com alocação dos operandos nas redes de cálculo simplesmente. Estes circuitos podem ser utilizados tanto para processamentos seqüenciais como paralelos.

Tendo adquirido o conhecimento das formas de implementação dos operadores matemáticos em ponto fixo, utilizaremos oportunamente estas estruturas em operadores para ponto flutuante.

As operações em ponto flutuante exigem, no entanto, que outros conhecimentos sejam a estes incorporados em aspectos referentes ao seu formato, onde a representação numérica não se restringe a definir o número de forma extensa em sinal e magnitude mas prevê um campo para sinal, expoente e mantissa e com isso a expansão de valores representáveis.

Serão feitas, a seguir, a apresentação e as considerações referentes às particularidades que envolvem tal tipo de representação.

II.5.2.1-3 - Padrão IEEE e Aritmética em Ponto Flutuante

As resoluções de operações matemáticas, em ponto fixo, vistas acima podem ser realizadas, a princípio, com qualquer número de *bits* que desejarmos.

Devido a ampla escolha possível para o tamanho dos operandos e conseqüente obtenção de resultados muito extensos aliada a adequação da representação binária de operandos e por razões comerciais, foram padronizadas normas que permitiram uma maior homogenidade entre os diversos fabricantes de circuitos integrados.

Assim surgiu a IEEE/ANSI (Institute of Electrical and Electronic Engineers/American National Standards Institute) Standard 754 que normatiza projetos de CIs.

Para implementar operadores matemáticos em ponto flutuante segundo o padrão ANSI/IEEE 754, deve-se estabelecer inicialmente um dos formatos padronizados de representação dos valores numéricos[24]: formatos básicos ou estendido com precisão

simples ou dupla. Os mais utilizados são os básicos sendo os estendidos usados quando se tem em vista reduzir erros de arredondamento.

II.5.2.1-3.1 - Formatos Básicos

Os formatos básicos (Fig. 2.9) compreendem:

- Um *bit* de sinal (s)
- Expoente polarizado e = E + polarização
- Fração ou mantissa (f)

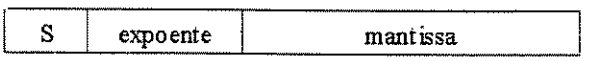


Fig. 2.9 – Formato para números em ponto flutuante

O formato básico de precisão simples adota a representação dos operandos com 32 *bits*, distribuidos em três campos: 1 *bit* para o sinal (s), 8 *bits* para a parte exponencial (e) e 23 *bits* para a mantissa ou parte fracionária (f).

O formato básico de precisão dupla apresenta a seguinte distribuição: 1 *bit* de sinal, 11 *bits* para o expoente e 52 *bits* para a mantissa, totalizando 64 *bits*.

Vamos apresentar mais detalhadamente as características do formato básico de precisão simples, porque este foi o adotado neste trabalho.

É importante esclarecer que o desenvolvimento do projeto em si, não está atrelado somente a esta representação podendo ser utilizado, tomadas as devidas proporções, formatos maiores para os operadores.

O campo de sinal s estabelece que se o *bit* é zero o número é positivo, caso contrário, negativo. Este é considerado o *bit* de maior importância desta representação, daí sua posição ser a de *bit* mais significativo.

No formato de precisão simples o campo de representação binária de expoente é de 8 *bits*, o que possibilita a representação de $2^8 - 1 = 255$ números.

A precisão simples, pode ser descrita por uma representação, onde os 32 *bits*, podem ser assim visualizados:

$(-1)^s (1.f_1f_2 \ldots f_{23}) \, 2^{(e-127)}$, onde **f** simboliza a mantissa ou parte fracionária, como também pode ser denominada, e **e** representa o expoente.

Os valores decimais máximos e mínimos correspondentes são:

máximo negativo: $-1{,}17 \times 10^{-38}$
mínimo negativo: $-3{,}37 \times 10^{38}$
mínimo positivo: $1{,}17 \times 10^{-38}$
máximo positivo: $3{,}37 \times 10^{38}$

II.5.2.1-3.2 - Propriedades da representação numérica segundo o Padrão ANSI/IEEE 754

II.5.2.1-3.3 - Formatos Numéricos

- **Números Normalizados** - Para se obter maior precisão, ou seja, maximizar o número de *bits* significativos na representação, a mantissa de um número em ponto flutuante deve ser normalizada, ou seja, o *bit* mais significativo deve ser igual a "1". Este *bit* por não pertencer à representação numérica do valor, deve ser colocado em uma posição (*bit* mais significativo da mantissa) que permita a sua desconsideração na hora da avaliação dos resultados finais, sendo por isso denominado *bit* escondido.

A normalização padroniza a execução de todo o processo matemático e caracteriza o intervalo de valores para possíveis operandos, por isso o procedimento de normalização deve ser mantido por todo o processo de execução das operações matemáticas, sendo adotadas renormalizações intermediárias quando se realizam operações em ponto flutuante, como ficará melhor esclarecido posteriormente.

- **Números Denormalizados** - Números fora do padrão estabelecido para números normalizados, são números não nulos, em ponto flutuante cujo expoente tem um valor definido, geralmente o valor mínimo do formato, e cujo *bit* escondido, explícito ou implícito da mantissa, é zero. A denormalização, indica que o operando tem magnitude menor que o menor número normalizado representável no formato. Números denormalizados possuem o campo de expoente zero e a parte fracionária, ou seja, a mantissa diferente de zero.

- **Bit Escondido** – Nas operações matemáticas com operandos em ponto flutuante, deve-se concatenar um *bit* à mantissa, ou seja, associar um *bit* indicador da condição de normalização do número. Este *bit* é conhecido como "*bit* escondido", de valor 1 para indicar que a mantissa está normalizada e 0 quando denormalizada, e deve ser preservado como unitário na mantissa por todas as etapas, nos resultados parciais e final dos cálculos matemáticos.

- **Expoente Polarizado** - A representação dos expoentes positivos e negativos não é sinalizada com o uso de um *bit* de sinal, como ocorre em representações de sinal e magnitude, mas através da técnica de polarização do expoente.

A polarização consiste na soma de uma constante ao expoente de forma a deslocar a referência numérica e obter somente valores positivos na representação.

Vejamos como se dá este procedimento:

Na representação básica simples temos 8 *bits* para alocação do expoente ou seja, 256 valores possíveis.

Utilizando-se todos os *bits* para descrever valores, estes poderiam representar números de zero a 255. Podemos, por outro lado, dividir este intervalo para a representação de valores de –127 a +127. Nestas circunstâncias o menor expoente representável assume o valor -127 e o maior +127. A polarização do expoente se dá através da soma da constante inteira 127, a todos os valores definidos com expoentes em inteiro. Este acréscimo, conhecido pela denominação de excesso de 127, deve acompanhar todo o processo de obtenção de resultados parciais dos expoentes, sendo subtraído no final da avaliação.

II.5.2.1-3.4 - Arredondamentos numéricos

Durante o processo de execução de operações matemáticas em ponto flutuante ocorrem situações onde, resultados parciais devem ser arredondados devido ao excesso de *bits* gerados durante o processo de execução da operação. Casos típicos ocorrem quando procedemos o alinhamento das mantissas de operandos na soma, utilizando deslocamentos para a obtenção de um expoente comum entre os dois operandos. Este procedimento ocasiona a geração de *bits* excedentes que precisam ser convenientemente analisados para que sejam incorporados ou não ao resultado obtido.

Um outro exemplo desta situação ocorre na multiplicação, onde se faz necessária a redução no número de *bits* gerados durante a execução da operação de 24 *bits* de um dos operandos com 24 do outro, resultando 48 *bits* que devem ser condensados novamente em 24 para satisfazer a padronização dentro do formato.

Os modos de arredondamento afetam todas as operações aritméticas com exceção da comparação e o resto da divisão.

A ocorrência destas situações exige que se utilize a técnica do arredondamento que consiste em tomar um número como infinitamente preciso, ou seja, com todos os *bits* resultantes da operação considerados e, se necessário modifica-o para colocá-lo no formato padronizado.

Para toda operação matemática dentro do padrão IEEE deve-se obter primeiramente um resultado intermediário correto em precisão infinita, sem limite, na obtenção do resultado e posteriormente realizar o arredondamento.

Os *bits* imediatos ao tamanho do padrão devem ser tratados de forma distinta para cada operação e reduzidos a três *bits* significativos que recebem as seguintes denominações específicas: *guard bit*, *round bit* e *sticky bit*, respectivamente do *bit* mais para o menos significativo da parte a ser arredondada. Estes *bits* devem ser analisados por um circuito de

arredondamento para que produzam ou não alterações dos *bits* menos significativos de resultados numéricos obtidos em etapas intermediárias das operações.

II.5.2.1-3.5 - Exceções

Em operações com ponto flutuante podem ocorrer condições de erro ou exceções, como são denominados os casos onde operandos ou resultados se apresentam fora do intervalo de valores representáveis e portanto devem ser excluídos do processo de execução de operações, e sinalizados, para que o usuário possa apreciar a situação de exceção.

Estas exceções são identificadas conforme a seguinte classificação [25]:

- **Resultado Inexato** - Ocorre quando o resultado de uma operação não pode ser representado precisamente no formato de destino desejado. Um exemplo é a divisão de 1,0 por 3,0, que não pode ser representada precisamente na forma binária.

- **Underflow numérico** - Esta situação é observada quando um resultado, diferente de zero é muito pequeno, em valor absoluto, para ser representado. Um resultado diferente de zero se localizar no intervalo compreendido entre $+2^{Emin}$ e -2^{Emin}. Por exemplo dividindo-se 1,0 pelo maior número em valor absoluto.

- **Overflow numérico** - Ocorre quando o resultado excede o valor máximo representável. Por exemplo a soma do valor máximo consigo próprio.

- **NaN - "Not a number"** - Operações numéricas devem sempre retornar resultados numéricos, exceção feita às comparações. Se uma exceção de operação inválida ocorre dentro de uma operação então, o resultado de retorno é NaN. Há muitos NaN diferentes. Se qualquer um dos operandos for NaN, a exceção é de operação inválida.

Existem dois tipos de NaNs: NaNs sinalizados (*signaling* NaNs) que sinalizam a operação inválida sempre que aparecem como operandos e NaN quietos (*quiet* NaN) que propagam-se através de quase todas as operações aritméticas sem sinalizar exceções.

- **Operação Inválida** - Engloba todas as operações matematicamente impossíveis e deve ser sinalizada por um NaN quieto. Os casos de operações inválidas podem ser assim sintetizados:

  - Qualquer operação sobre NaN sinalizados
  - Adição ou Subtração de valores infinitos, ou seja, quando algum ou ambos dos operandos for infinito.
  - Multiplicação de zero por infinito
  - Divisão de qualquer valor por zero ou de infinito por infinito
  - Resto da divisão quando o numerador é zero, operação reconhecidamente impossível ou divisão com o denominador infinito.

II.5.2.1-3.6 - Resumo do Formato

Resumindo todas as definições dadas acima podemos formalizar, que um número v em ponto flutuante deve ser representado no formato simples com as seguintes características:

Se $0 < e < 255$ então $v = (-1)^s . 2^{e-127} (1.f)$, que corresponde a forma normalizada.

Se $e = 0$ e $f \neq 0$ então $v = (-1)^s . 2^{e-126} (0.m)$, isto é, valores denormalizados.

Se $e = 255$ e $f \neq 0$, então v é NaN, qualquer que seja s.

Se o valor de $f = 0$ qualquer que seja o valor de e, $v = 0$.

Se $e = 255$ e $f = 0$, então $v = (-1)^s$ Infinito.

Onde **f** corresponde à mantissa, **e** ao expoente e **s** ao sinal.

II.5.2.1-3.7 - Considerações numéricas do projeto

Vamos neste projeto utilizar os NaNs sinalizados que representarão a ocorrência de situações de exceção, uma vez que os resultados de cada etapa das simulações deverão gerar resultados para simulações posteriores e estes valores devem ser numericamente bem definidos.

Devido às condições que operandos e resultados devem obedecer inicialmente e durante o procedimento das operações matemáticas para que resultados parciais e finais não se tornem exceções, várias etapas de análise e deslocamentos de operandos devem ser cumpridas, ocasionando, como já dissemos anteriormente, um acréscimo significativo nas etapas de conclusão das operações matemáticas em ponto flutuante diferentemente do que ocorre nas execuções em ponto fixo.

II.5.2.1-4 - Operações Aritméticas em ponto flutuante

A aritmética em ponto flutuante difere notadamente de operações realizadas em ponto fixo. Far-se-á, a seguir, uma breve exposição das particularidades envolvidas nestas operações, relacionando-as aos tópicos citados até o presente momento, abordando o assunto com um nível de detalhes que pretende trazer ao leitor uma introdução teórica necessária ao entendimento das arquiteturas escolhidas para o projeto, sem contudo esgotar o assunto, que é muito extenso e elaborado.

Inicialmente veremos as operação de adição e consequentemente subtração e posteriormente multiplicação e divisão com ponto flutuante.

II.5.2.1-4.1 - Soma e Subtração em Ponto Flutuante

II.5.2.1-4.1-1 - Operação Efetiva

Antes de nos direcionarmos para a análise das diversas etapas necessárias para a realização da soma ou subtração em ponto flutuante, vejamos o conceito de operação efetiva que se refere à operação que efetivamente ocorre quando sinais estão atrelados a valores numéricos nestas operações.

Temos a considerar dois casos: a adição e a subtração efetiva.

Considerando A e B dois operadores e os sinais a eles vinculados temos que a soma efetiva ocorre nos seguintes casos:

$(+A) + (+B) = + (|A| + |B|)$

$(-A) + (-B) = - (|A| + |B|)$

$(-A) - (+B) = - (|A| + |B|)$

$(+A) - (-B) = + (|A| + |B|)$,

Como vemos, nestes casos, embora tenhamos até situações onde a operação inicialmente prevista seja diferença, devido a presença dos sinais dos operandos, a operação se torna efetivamente uma soma.

Ocorre a subtração efetiva nos seguintes casos:

$(+A) - (+B) = + (|A| - |B|)$

$(+A) + (-B) = + (|A| - |B|)$

$(-A) - (-B) = - (|A| - |B|)$

$(-A) + (+B) = - (|A| - |B|)$.

A expressão abaixo pode descrever em termos booleanos a operação efetiva:

operação efetiva = sinal de A *XOR* sinal de B *XOR* operação

Vale recordar aqui que o sinal positivo é representado por zero, o sinal negativo por 1 e a operação efetiva obtida a partir da expressão vista acima poderá ser logicamente apresentada por 0 ou 1 para soma ou subtração respectivamente.

II.5.2.1-4.1-2 - Arquitetura do Somador/Subtrator em Ponto Flutuante

Para se proceder as operações de soma e subtração em ponto flutuante é necessário que as entradas sejam números normalizados em ponto flutuante, que possam ser representados na forma:

$A = a \times 2^p$

$B = b \times 2^q$

aqui os números estão descritos na forma binária, a e b são as mantissas e p e q são seus respectivos expoentes. A soma então obtida pode ser representada como a seguir:

$S = A + B$

Onde o resultado S, pode ser representado por $s \times 2^r$ (na forma não normalizada) sendo, $r = \max(p,q)$, o resultado inicial, não definitivo para o expoente.

Uma renormalização é então realizada, resultando:

$S = d \times 2^v$ (forma normalizada)

onde $0.5 < d < 1$, ou seja, a mantissa normalizada deve estar compreendida entre 2^{-1} e 2^0 e o expoente será reavaliado assumindo o valor v.

Todas as etapas intermediárias na obtenção do resultado para mantissa e expoente devem ser realizadas com bastante critério para que não ocorram aproximações incorretas e, conseqüentemente, geração de resultados imprecisos.

São os seguintes blocos necessários para a obtenção do *hardware* dedicado para a realização da soma e subtração em ponto flutuante.

- Análise de Mantissas
- Análise de Expoentes
- Comparação de Mantissas
- Comparação de Expoentes
- Lógica para Controle de Exceção
- Multiplexação de Expoentes
- Concatenação de *Leading Bit*
- Multiplexação de Mantissas
- Subtração de Expoentes
- Avaliação de Expoentes
- Deslocamento de Mantissa
- Lógica de Cálculo dos Bits de Arredondamento
- Soma de Mantissas
- Renormalização de Mantissa por causa de *Overflow* ou *Underflow*
- Lógica de Arredondamento da Mantissa
- Renormalização de expoente (ajuste)
- Deslocamento de 1 bit para a direita
- Verificação de *Overflow/Underflow*
- Lógica para Cálculo de Exceção

A figura 2.10 apresenta o diagrama que permite a visualização da interligação entre estes blocos.

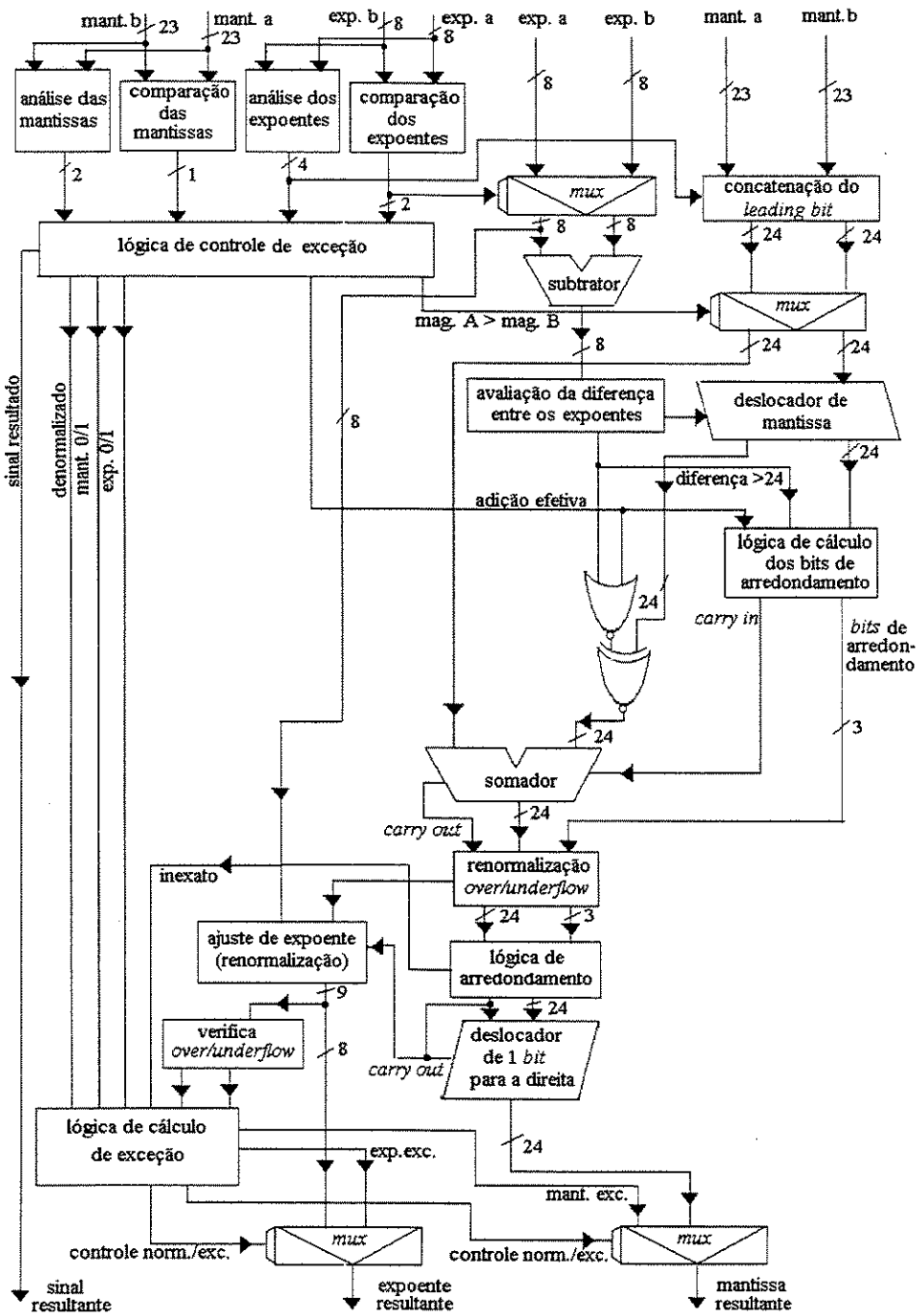


Fig. 2.10 - Somador/subtrator em ponto flutuante

A seguir, será feita uma breve descrição da função operacional dos mesmos. Em algumas situações as descrições dos blocos se dará de forma concorrente para a mantissa e expoente, devido a semelhança funcional.

II.5.2.1-4.1-2.1 - Análise de Mantissas e Expoentes

Para que se realize o processamento de soma ou subtração de dois números em ponto flutuante devemos inicialmente verificar se os valores utilizados são convenientes para este tipo de operação, ou seja, verifica-se se os dados de entrada do somador não constituem exceções, como as já comentadas anteriormente.

Para isso o primeiro passo para a execução da soma consta da verificação da validade da mantissa e do expoente como operandos. O bloco de análise dos expoentes verifica se algum dos expoentes é zero ou máximo (255). Somente esta análise já é suficiente para dizer se o número é normalizado porque todos os números não normalizados apresentam expoentes com o valor mínimo.

O bloco de análise das mantissas verifica se a mantissa é diferente ou igual a zero. Esta análise complementa a anterior na caracterização de valores reservados.

Se não ocorrerem exceções, a operação seguirá para as etapas seguintes, caso contrário, o circuito de controle de exceções conduzirá a um resultado sinalizado de exceção.

II.5.2.1-4.1-2.2 - Comparação de Mantissas e Expoentes

O comparador de expoentes fornece os dados para um multiplexador que seleciona os expoentes de forma que o resultado da subtração dos expoentes seja sempre positivo. O maior valor será tomado como referência para determinação do expoente do resultado.

O comparador das mantissas conjuntamente com o comparador de expoentes define o operando de maior magnitude para que seja selecionada a mantissa correspondente ao maior operando como um todo.

II.5.2.1-4.1-2.3 - Lógica para Controle de Exceção

Os blocos acima citados injetam seus resultados nesta unidade para que se processe a verificação da ocorrência de algum dado indesejável, ou seja, uma exceção. A ocorrência de exceção deve ser sinalizada produzindo o desvio do processamento para o bloco de Lógica de Cálculo de Exceção.

Este bloco também é responsável pela determinação, através da análise dos sinais dos operandos, da operação que efetivamente irá ocorrer, conforme já comentado anteriormente.

dos *bits* de arredondamento, para que se consiga uma melhor aproximação possível para o resultado da soma que será gerado futuramente.

O número deve portanto ser reconduzido ao formato normalizado, com os deslocamentos compensados em forma de arredondamento. Para isso uma lógica de cálculo dos *bits* de arredondamento deve ser realizada para que possa influir na renormalização, ocorrida após a soma das mantissas propriamente dita.

Neste circuito também é definido o *carry* de entrada do somador de mantissas, através da análise da operação que efetivamente está ocorrendo. Em caso de subtração efetiva este *carry* assume o valor binário 1, visando o complemento de dois do subtraendo.

II.5.2.1-4.1-2.8 - Soma de Mantissas

Este bloco consiste de um somador clássico onde a técnica de *carry lookahead* pode ser utilizada para acelerar o processo de obtenção do resultado.

A mantissa do operando de maior magnitude selecionada pelo multiplexador é somada à mantissa do operando de menor magnitude. Quando a operação é soma efetiva o menor operando não se altera e a soma se realiza de forma convencional porém, quando a operação resultante é subtração efetiva o menor operando sofre um processo de complementação de um. Com o acréscimo de um *carry* forçado igual a 1 obtido do bloco de lógica de cálculo dos *bits* de arredondamento, obtém-se a complementação de dois, necessária neste caso.

II.5.2.1-4.1-2.9 - Renormalização de Mantissa por causa de *Overflow* ou *Underflow*

Realizada a soma ou subtração o resultado deve ser reavaliado tanto com relação a valores excedentes para mais (*overflow*) no caso de soma efetiva ou para menos (*underflow*) quando subtração efetiva. O resultado deve ser reavaliado para que se verifique se o mesmo se encontra dentro do intervalo de valores permitidos para o formato.

Como sabemos o número normalizado se caracteriza por apresentar o *bit* mais significativo ou *bit* escondido igual a 1. O *bit* de *overflow*, localizado em uma posição mais significativa que o *bit* escondido estará com valor 1 quando se realiza soma efetiva, caso o resultado exceda o limite de valores representáveis. Assim sendo para se renormalizar um número com *overflow* procede-se um deslocamento para a direita. Este deslocamento deve produzir a devida alteração no expoente, para que o resultado permaneça correto. Este

acréscimo no valor do expoente pode também ocasionar uma situação de *overflow* no expoente, sendo que neste caso o resultado passa a ser considerado como uma exceção por ter ultrapassado o limite de valores representáveis.

A situação de *underflow*, que ocorre quando a operação realizada é subtração efetiva, é caracterizada pela presença de zero na posição do *leading bit*. Desta forma o deslocamento deve se processar para a esquerda até que a posição do *leading bit* seja preenchida com 1. Estes deslocamentos devem ser acompanhados de decrementos no expoente, o que pode gerar uma situação de exceção, caso ocorra uma situação de *underflow* de expoente.

II.5.2.1-4.1-2.10 - Lógica de Arredondamento da Mantissa

Do circuito de lógica de arredondamento surgem dois resultados, quais sejam, a mantissa renormalizada e os *bits* de arredondamento resultantes dos deslocamentos executados na renormalização. Os *bits* de renormalização devem agora ser incorporados à mantissa. Para isso é realizada uma nova etapa de arredondamento que pode gerar um resultado inexato, que pode ser sinalizado, ou um *carry* que deve ser considerado para ajuste do expoente após a renormalização.

A mantissa resultante, caso ocorra este *carry*, deverá ser deslocada de um *bit* para a direita, conforme explicado no ítem a seguir. Este deslocamento deve ser também sinalizado para o bloco que realiza o ajuste do expoente.

II.5.2.1-4.1-2.11 - Deslocamento de 1 bit para a direita

Após a etapa de arredondamento onde a mantissa incorpora os *bits* de arredondamento, pode ocorrer uma situação de *overflow*. Para que esta situação seja corrigida, procede-se o deslocamento da mantissa de 1 *bit* para a direita. Neste caso o expoente deve ser compensado por ocasião da renormalização do expoente.

II.5.2.1-4.1-2.12 - Renormalização de expoente (ajuste)

Após a realização do deslocamento de 1 *bit* na mantissa para situações previstas no ítem anterior, o expoente deve ser ajustado. É necessário um ajuste também em casos de ocorrência de *overflow* de mantissa durante a etapa de renormalização da mesma.

II.5.2.1-4.1-2.13 - Verificação de *Overflow/Underflow* do Expoente

Após o arredondamento e renormalização da mantissa, o expoente resultado é analisado quanto ao *overflow* ou *underflow*. Se ocorrer alguma dessas situações é sinalizada ocorrência de exceção.

Processadas as operações para o ajuste de expoente, este deve ser avaliado quanto à sua ordem de grandeza, ou seja, deve-se proceder a verificação de *under* ou *overflow* (excesso para menos ou mais respectivamente dos valores aceitos para expoentes).

No caso de ocorrência de exceções, a lógica para cálculo de exceção fará a avaliação dos resultados. Em condições normais os resultados serão liberados via multiplexadores de saída.

II.5.2.1-4.1-2.14 - Lógica para Cálculo de Exceção

Encontrada alguma situação de exceção, a mesma deve ser sinalizada através dos *bits* de exceção que deverão produzir um resultado com indicação de erro.

A saída do resultado pode se dar por dois multiplexadores que liberem separadamente o resultado da operação ou de ocorrência de exceção, ou somente um multiplexador que faça as devidas sinalizações aos resultados obtidos.

II.5.2.1-4.2 - Multiplicação e Divisão em Ponto Flutuante

O multiplicador/divisor em ponto flutuante da mesma forma que o somador/subtrator exige todo um tratamento e análise dos operandos a cada passo da resolução.

A complexidade para obtenção de produtos e quocientes é menor do que a envolvida na obtenção de resultados de soma e subtração porque aqui os expoentes realizam uma operação simples de soma para o caso de multiplicação e subtração para o caso de divisão. Não é necessária a detecção das alterações de valores de expoente ao longo do processo da resolução matemática como ocorre nas operações vistas anteriormente.

II.5.2.1-4.2-1 - Arquitetura do Multiplicador/Divisor em Ponto Flutuante

Os padrões estabelecidos pelo IEEE/ANSI Standard 754, semelhante ao que ocorre com o somador/subtrator, prevêem várias etapas para a resolução das operações de multiplicação/divisão.

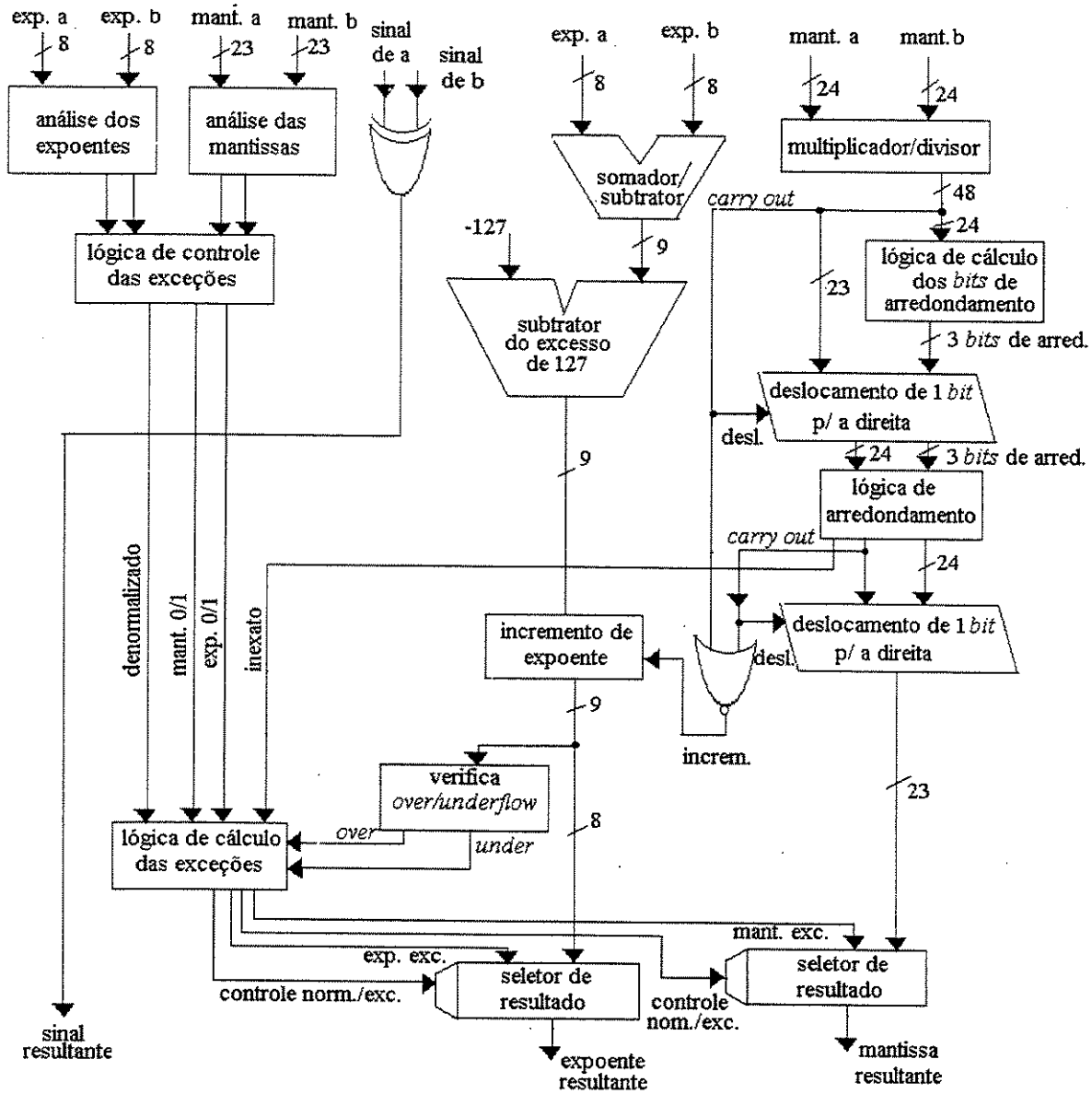


Fig. 2.11 - Multiplicador /Divisor em ponto flutuante

Acompanhando pelo diagrama exposto na Fig. 2.11, vejamos os blocos e a função de cada um em particular, que formam o conjunto de circuitos necessários para a obtenção de resultados de multiplicações e divisões em ponto flutuante:

- Análise de expoentes
- Análise de mantissas
- Lógica de controle de exceção
- Lógica para obtenção do sinal resultado
- Soma/ Subtração de expoentes

- Subtração do excesso de 127
- Multiplicação/ Divisão de mantissas
- Lógica de cálculo dos bits de arredondamento da mantissa
- Lógica de arredondamento da mantissa
- Deslocamentos de 1 bit para a direita
- Incremento de expoente
- Verificação de *underflow* e *overflow*
- Lógica de cálculo de exceções

II.5.2.1-4.2-1.1 - Análise de Mantissas e Expoentes

O primeiro procedimento antes de iniciar qualquer operação em ponto flutuante é verificar se os operandos são normalizados ou se apresentam valores reservados, tais como NaN, denormalizado, zero ou infinito, já descritos no item II.5.2.1-3.2. A análise dos expoentes pode ser simplesmente a verificação se os mesmos não são valores mínimo (zero) ou máximo (255) que caracterizam os casos exceção e número denormalizado.

A análise das mantissas consiste apenas em verificar se não se trata de valor zero e associada à informação da análise do expoente, já identifica operandos convenientes para as operações.

Caso os operandos apresentem valores reservados, ou seja, que possam ser classificados como exceções, o resultado é obtido a partir da lógica de tratamento das exceções, semelhante à realizada para a soma e subtração.

II.5.2.1-4.2-1.2 - Lógica de controle de exceção

Este bloco recebe as informações referentes à análise das mantissas e expoentes e, caso seja encontrada alguma exceção, o resultado será determinado pelo bloco de cálculo de exceção, caso contrário a operação segue o seu curso normal.

II.5.2.1-4.2-1.3 - Lógica para obtenção do sinal resultado

Esta etapa realiza a comparação entre o sinal dos dois operandos e define como positivo o resultado para sinais iguais e negativo para sinais diferentes utilizando para isso uma simples lógica *XOR* entre os *bits* mais significativos do formato.

II.5.2.1-4.2-1.4 – Soma/Subtração de expoentes

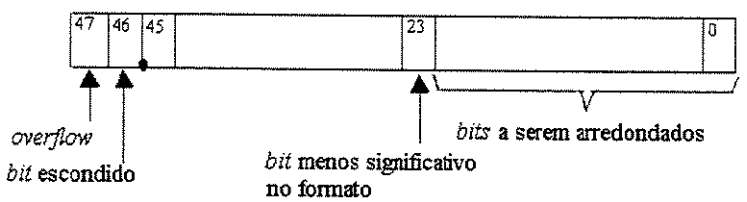
A determinação do expoente é realizada de forma bem simplificada a partir de uma soma para a multiplicação ou subtração para a divisão. O bloco somador é facilmente transformado em subtrator, de acordo com o sinal dos operandos envolvidos, como já tivemos oportunidade de comentar.

II.5.2.1-4.2-1.5 - Subtração do excesso de 127

Como anteriormente exposto, os expoentes originais dos operandos devem ser somados a 127 para que não haja preocupação com os sinais destes expoentes ao longo do processamento de quaisquer operações. Dos procedimentos aritméticos sabe-se que a multiplicação envolve a soma de dois expoentes, ambos neste caso, convertidos para a representação de excesso de 127, portanto, é necessário que se faça uma subtração de um dos excessos para que o expoente prossiga pelas etapas normais de processamento e venha a ser obtido de forma correta, raciocínio análogo se aplica à divisão.

II.5.2.1-4.2-1.6 - Multiplicação / Divisão de mantissas

O multiplicador deve apresentar uma organização que permita a multiplicação de 24 por 24 *bits* resultando em 48 *bits*. Estes *bits* estão assim distribuidos.



Os 25 *bits* mais significativos (inclusive o *overflow* e o *bit* escondido) constituem o resultado nos limites do formato. Os 23 *bits* da seqüência devem seguir para a lógica de renormalização e arredondamento.

No caso de ocorrência de *overflow* na multiplicação, há necessidade de um deslocamento para a direita e uma conseqüente compensação no expoente.

Caso o expoente apresente um *overflow* após esta compensação, o resultado será considerado exceção.

II.5.2.1-4.2-1.7 - Lógica de cálculo dos bits de arredondamento da mantissa

Para as operações de multiplicação e divisão, o cálculo dos *bits* de arredondamento se processa de forma bem mais simples do que para a operação de soma . Aqui os 23 *bits* da parte a ser arredondada serão reduzidos a dois *bits* de arredondamento, que serão assim determinados: o mais significativo corresponde ao mais significativo dos *bits* arredondados e o menos significativo deve ser o resultado do *OR* lógico dos demais 22 *bits*.

II.5.2.1-4.2-1.8 - Lógica de Arredondamento da mantissa

Os *bits* gerados pelo bloco anterior devem ser incorporados à mantissa para que a mesma continue no seu formato padrão. Um incremento deve ser condicionado aos *bits* de arredondamento e o *bit* menos significativo da mantissa.

II.5.2.1-4.2-1.9 - Deslocamentos de 1 bit para a direita

Os circuitos de deslocamento ou de renormalização são acionados a cada ocorrência de *overflow* no resultado da mantissa, após a multiplicação de mantissas ou após arredondamento. Trata-se de um simples deslocador acionado por um sinal de controle, que é o próprio *bit* de *overflow*, proporcionando ou não um deslocamento para a direita.

Estes deslocamentos, caso ocorram devem ser compensados pelo incremento no expoente.

II.5.2.1-4.2-1.10 - Incremento de expoente

O expoente deve ser alterado para compensar os deslocamentos ocorridos na mantissa devido a *overflow* do multiplicador de mantissas tambem como resultado do arredondamento.

II.5.2.1-4.2-1.11 - Verificação de *underflow* e *overflow*

Após o ajuste do expoente, há necessidade de uma análise para verificar se o mesmo apresenta um *bit* de *underflow* ou *overflow*. Se isto ocorrer o número será reconhecido como uma exceção e a mesma deve ser sinalizada. No caso de *overflow* realizam-se deslocamentos para a direita e compensações de um único incremento no expoente. Em casos de *underflow* deve-se detectar a posição do primeiro *bit* significativo do resultado e

proceder os deslocamentos necessários à normalização, decrementando convenientemente o expoente.

Temos aqui concluída a etapa de análise dos algorítmos que serão utilizados no projeto da Unidade Aritmética e Lógica. Este conhecimento se mostrou de grande importância no desenvolvimento deste trabalho, por corresponder a um assunto muito complexo, com acentuado nível de detalhes técnicos.

Na seqüência serão apresentadas considerações teóricas sobre registradores, tipos de endereçamento e memórias. As colocações vistas a seguir pretendem somente fazer uma abordagem sucinta dos assuntos, vislumbrando uma introdução suficiente para o prosseguimento de apresentação da arquitetura do microprocessador aqui proposta.

II.5.2.2 - Registradores

Um registrador é um grupo de elementos de memória que armazena temporariamente as palavras que devam estar em trânsito durante as diversas etapas do processamento.

Os registradores são utilizados nas mais variadas situações durante o processamento, mantendo endereços de busca na memória principal, alocando palavras retiradas ou carregadas em memórias, mantendo a instrução durante o processo de decodificação de seu conteúdo, retendo valores intermediários para posterior processamento.

Por serem elementos de isolamento temporário de palavras, podem também realizar operações lógicas especiais tais como, deslocamentos para direita ou esquerda e contagem de programas.

Durante o processamento os registradores desempenham um papel de relevante importância na execução de programas, uma vez que colocam dados para processamento, em localizações de maior disponibilidade de uso. É o que ocorre em situações onde dados existentes em memórias são temporariamente alocados em registradores, evitando assim buscas prolongadas nas memórias de dados, que na maior parte das vezes estão localizadas fora da unidade de processamento.

Por ser um artifício, que utilizado, produz importantes resultados na velocidade de processamento, máquinas para finalidade de alto desempenho apresentam um maior número de registradores que as providas de poucos recursos.

II.5.2.2-1 – Tipos de Endereçamento

Para que se concretize o processamento de dados dentro de uma máquina computadorizada, uma programação deve ser realizada. Se torna portanto necessário o armazenamento do *software* ou programa que execute as tarefas de maneira conveniente.

O programa nada mais é que uma seqüência de códigos definida pelo programador que vai ser lida, interpretada ou decodificada pelo computador e, ao término da execução da seqüência prevista pelo programador, resultados serão obtidos deste processamento.

É necessário que se defina o conteúdo das instruções que constam do programa a ser executado pela máquina, para que os circuitos do microprocessador sejam acionados de maneira a se atingir o sucesso no processamento.

As instruções devem conter informações que após interpretadas proporcionem que o fluxo dos dados ou *data-path* dentro da máquina, execute as tarefas previamente definidas.

Para que a interpretação da instrução se processe corretamente tem-se que se conhecer os tipos de endereçamento adotados pelo projetista do *assembler* , ou linguagem simbólica da máquina, durante o projeto do microprocessador e que delinearam os formatos das instruções e permitem o acionamento das unidades necessárias para execução de quaisquer das instruções previstas para a máquina.

Vários são os modos de endereçamento existentes, sendo alguns de maior complexidade e portanto utilizam maior tempo na execução e outros que por permitirem fácil acesso a operandos ou dados proporcionam uma aceleração no processamento.

Cabe ao projetista da máquina prever a necessidade do uso de instruções mais elaboradas, a custo de uma redução na taxa de execução de instruções em um determinado intervalo de tempo, ou optar por instruções, que mesmo sendo mais simples, atendam satisfatóriamente as exigências operacionais do microprocessador.

Vejamos somente alguns dos modos de endereçamento mais amplamente utilizados nos microprocessadores comercialmente disponíveis, dentre os quais faremos a escolha dos modos mais convenientes para a arquitetura aqui desenvolvida.

- **por Registrador**: Os dados a serem utilizados na instrução se encontram armazenados nos registradores internos, explícitos no formato da instrução. Por exemplo, ADD Rx,Ry indica que os operandos a serem somados estão presentes nos

código de operação	Rx	Ry

registradores Rx e Ry. Para instruções de transferência Rx e Ry podem representar fonte e destino respectivamente, ou em ordem contrária, dependendo da escolha definida pelo projetista do microprocessador.

- **Direto**: O endereço onde está o dado na memória é parte explícita da instrução. Por exemplo, ADDD end. – Representa que o dado presente no endereço dado por end. deve ser somado ao conteúdo de um registrador específico da arquitetura.

código de operação
endereço do dado

- **Imediato** : O dado é parte integrante da instrução. Por exemplo, ADI 25 significa que um dos dados a serem somados tem valor 25. O outro já deverá estar armazenado em um registrador específico da arquitetura.

código de operação
dado

- **Indireto**: A instrução contém o endereço onde está o endereço do dado armazenado na memória.

código de operação
endereço do endereço do dado

Os formatos acima expostos podem apresentar variações de acordo com as necessidades na definição das instruções pelo projetista da máquina. Os campos indicados por código de operação podem conter informações sobre o modo de operação e índices necessários para a definição completa da forma como os dados presentes na instrução serão utilizados. O tamanho dos campos para dados ou endereços também pode ser alterado de acordo com a dimensão da palavra prevista para os mesmos.

II.6 - Memórias

É impossível se concretizar o projeto de um computador sem a utilização de memórias porque dentro da arquitetura de computadores elas ocupam um lugar de relevante

importância uma vez que se constituem nos componentes capazes de armazenar dados, instruções e resultados, ou mais amplamente, programas, que vão dar funcionalidade aos demais blocos formadores das máquinas programáveis.

O fundamento primeiro destes componentes é a utilização de portas lógicas que devidamente interligadas têm a capacidade de reter informações por intervalos de tempo curto, longo ou até mesmo definitivo.

A fig. 2.12, apresenta a estrutura da forma mais simples[12] de se guardar informações temporariamente em portas lógicas. Este circuito é chamado de SR *latch*, e apresenta duas entradas S (*set* ou acionada para obtenção de nível lógico 1) e R(*reset* ou acionada para 0) e duas saídas com sinais complementares Q e Qbar. As figuras 2.12(a) e (b) ilustram as duas possibilidades de estado gerados na saída dependendo do sinal previsto

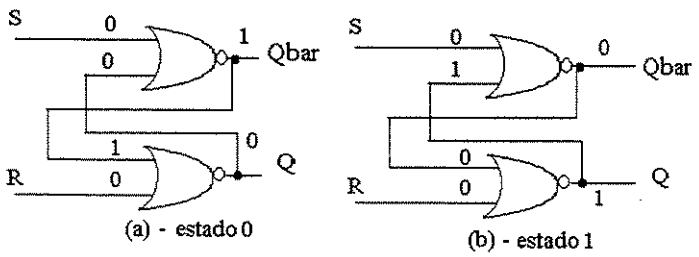


Fig. 2.12 - (a) - *latch* NOR no estado 0. (b) - *latch* NOR no estado 1.

inicialmente para Q.

O SR *latch* foi sendo progressivamente otimizado e grandes quantidades destes elementos foram sendo compactados dando origem às memórias comercialmente disponíveis. Graças ao estado de sofisticação atualmente atingido nos processos de microeletrônica, em áreas muito reduzidas é possível o armazenamento de centenas de milhões de *bits*.

A estrutura apresentada fig. 2.12 não conta com recursos de mudança de estado de forma sincronizada, muito importante nos projetos lógicos de uma forma geral, que trabalham referenciados a uma determinada base de tempo.

Sendo associado a esta estrutura um sinal pulsante, obtém-se mudanças de estado somente em intervalos de tempos pré definidos como ilustrado na figura 2.13. Estes se constituem nos *latches* SR sincronizáveis.

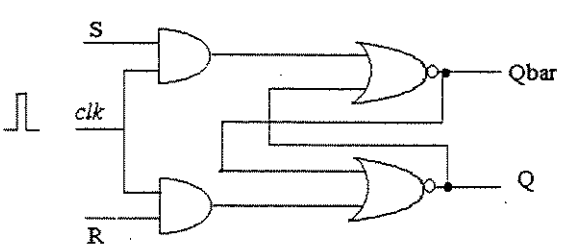


Fig. 2.13 - *Latch* SR sincronizável

Para analisarmos os sinais resultantes do chaveamento das portas lógicas consideramos o fato das duas entradas serem complementares. Pode ocorrer, no entanto o caso em que ambas as entradas são nulas e através da análise lógica verifica-se que as saídas podem apresentar estados indefinidos.

Para se solucionar este problema as entradas S e R devem ser interligadas através de um inversor, conforme visto na figura 2.14, para que se mantenha a situação de sinais

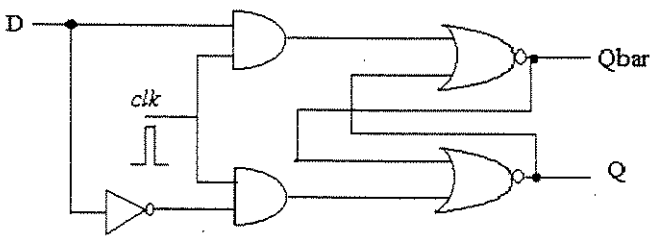


Fig. 2.14 - *Latch* D sincronizável

complementares, dando assim origem a um circuito de estados lógicos bem definidos, o *latch* D, sincronizável.

Estes dois *latches* analisados apresentam o sincronismo dos estados lógicos através do uso de um pulso de *clock* ou relógio.

Nos procedimentos de armazenamento de dados muitas vezes se torna necessário se realizar o armazenamento durante um instante de tempo muito pequeno e pulsos muito estreitos de clock são de difícil obtenção. Neste caso ao invés de se tomar o pulso de *clock* para armazenamento do dado, faz-se uso de seu instante de transição, dando origem aos chamados flip-flops. No caso dos *latches* dizemos que operam por gatilhamento por nível ou *level triggered*, enquanto que os *flip-flops* se caracterizam pelo gatilhamento por borda ou *edge triggered*.

Conhecidas as células de memória, podemos agrupá-las de forma matricial para o armazenamento de uma grande quantidade de *bits*, como é o caso das memórias universalmente utilizadas em projetos lógicos.

As memórias podem ser caracterizadas pelo seu tamanho, ou seja, sua capacidade de armazenar palavras lógicas. Por exemplo, uma memória 32K x 8 apresenta 32K (32 vezes 1024) palavras de 8 *bits*.

Uma outra característica a ser considerada quando se realiza a leitura dos dados armazenados em memórias é o tempo de acesso aos mesmos, que está intimamente ligado à tecnologia utilizada na implementação. As memórias utilizadas comercialmente apresentam tempo de acesso da ordem de poucas dezenas de nanosegundos. O tempo de armazenamento, como já comentado, pode ser bastante variável.

As formas de *latches* vistas acima são utilizadas para a fabricação de memórias RAM – *Random Access Memory* e apresentam capacidade para escrita e têm também seus dados disponíveis para leitura. As RAMs podem ser classificadas em estáticas e dinâmicas.

Estáticas são constituídas por *latches* D e armazenam os dados enquanto estiverem ligadas a uma fonte de alimentação.

Dinâmicas são formadas por arranjos de capacitores integrados que devem ser periodicamente reatualizados em curtos intervalos de tempo, da ordem de milisegundos– *refreshed*, para que seus conteúdos sejam conservados convenientemente.

Uma outra característica das memórias, que estão diretamente relacionadas à técnica de construção das memórias é a volatilidade, ou seja, a capacidade da memória manter as informações nela armazenadas.

O armazenamento permanente dos computadores pode ser obtido em memórias ROM – *Read Only Memory,* onde as palavras, uma vez armazenadas permanecem inalteradas por tempo indeterminado.

De uma forma geral as ROMs consistem de uma matriz de diodos ou transistores integrados que podem ser programados ou fundidos quando percorridos por uma corrente de intensidade suficiente para provocar a interrupção de fluxo de corrente em posições previstas na programação.

Diversas são as formas de manutenção dos dados armazenados nas ROMs, que ocasionaram o surgimento de suas derivações, na busca não só de formas de reutilização, como também flexibilidade para a programação e barateamento na gravação.

PROM – *Programmable ROM*, ou memórias de leitura programáveis, que se comportam como uma ROM, exceto que ela pode ser programada pelo usuário através de um gravador, não necessitando um processamento por difusão como o necessário às ROMs. Uma vez realizada a programação o seu conteúdo se torna permanente.

EPROM – *Erasable PROM* ou memórias de leitura programáveis e apagáveis que não só podem ser gravadas, como também apagadas. Uma exposição a uma forte luz ultravioleta por aproximadamente 30 minutos permite o apagamento das informações contidas na memória, tornando-as disponíveis para uma nova programação.

EEPROM – *Electrically Erasable PROM*, também chamada de EAROM – *Electrically Alterable ROM*, ou eletricamente apagável ou ainda eletricamente alterável, podem ser apagadas pela aplicação de pulsos elétricos ao invés de exposição ao ultravioleta.

II.7 - Comentários

Vimos neste capítulo o tratamento teórico necessário para a elaboração do nosso projeto. Este conhecimento se faz necessário para a formação de uma base técnica para a aplicação de procedimentos que permitam o desenvolvimento de um projeto que atenda às necessidades não só acadêmicas como também do mercado emergente de desenvolvimento de circuitos integrados.

Nos capítulos seguintes veremos, de forma mais direcionada, os vários blocos formadores do projeto e posteriormente os resultados obtidos por simulações com ferramentas da *Mentor Graphics*.

Capítulo III

Desenvolvimento do projeto

III.1 - Introdução

Este capítulo aborda o desenvolvimento do *hardware* do projeto, com base nas estruturas propostas e dentro dos fundamentos teóricos apresentados no capítulo anterior.

Foram projetadas estruturas que tornam possível a resolução pelo MPH de programas de descrição de componentes eletrônicos comercialmente disponíveis, tais como resistores, capacitores, transistores, etc.

O desenvolvimento foi elaborado a partir da proposta do projeto do MPH apresentada no diagrama de blocos da figura 1.2 do capítulo I através da análise da arquitetura de cada um dos blocos formadores do microprocessador.

Inicialmente foi construído um diagrama de blocos mais elaborado que após progressivos estudos deu origem a uma versão completa do microprocessador MPH. As unidades projetadas foram descritas em linguagem VHDL, própria para descrição de circuitos e simuladas em ferramentas *Mentor Graphics*.

Faremos aqui considerações referentes às diversas etapas do desenvolvimento do projeto, também como a apresentação do repertório das instruções, que uma vez descrito em termos das microinstruções e armazenadas na UMA, têm a função de controlar a execução de tarefas pelo microprocessador.

III.2 – Considerações gerais sobre o microprocessador proposto

Em todo projeto de um microprocessador é necessário inicialmente que se defina a sua arquitetura, semelhante ao que ocorre no trabalho cotidiano de um arquiteto de construção civil quando se propõe à idealização de projetos habitacionais: o projeto deve vir a satisfazer os anseios de seus usuários.

Da mesma forma, na arquitetura de projetos eletrônicos, os circuitos e programas formadores da estrutura física do componente devem ser direcionadas às aplicações a que ele se destina.

Em primeira instância se sabia que a arquitetura do microprocessador deveria ser capaz de simular componentes eletrônicos a partir da resolução dos modelos matemáticos dos mesmos. As descrições dos modelos de alguns componentes foram estudadas [5],[6] em teses de doutorado e mestrado, desenvolvidas no DSIF, anteriores a este trabalho.

Da observação das expressões definidas para a representação dos modelos, constatou-se que, as operações aritméticas no MPH deveriam ocorrer em ponto flutuante para com isso possibilitar um maior espectro de valores passíveis de cálculo.

Fundamentado o desenvolvimento do projeto de simulação por *hardware*, ficou definido que a Unidade Aritmética e Lógica realizaria as quatro operações fundamentais por *hardware*; as expressões trigonométricas básicas tais como seno e cosseno seriam obtidas pelo uso de tabelas, ou *lookup tables*.

O processamento numérico adotado para os dados, foi o formato básico em ponto flutuante em 32 *bits*, obedecendo ao padrão ANSI/IEEE 754 "Single Precision". Este formato possibilita uma maior capacidade numérica quando comparado aos processamentos em ponto fixo.

É também nosso objetivo, em realizando as operações matemáticas fundamentais com ponto flutuante, por *hardware*, atingir uma maior velocidade de processamento com satisfatória precisão numérica na obtenção dos resultados parciais e finais das simulações.

Com relação à arquitetura para a unidade de controle, optou-se pela microprogramação, por viabilizar de forma ordenada e sistemática controles de alta complexidade.

III.3 – Arquitetura do MPH

A figura 3.1 mostra o diagrama de blocos da arquitetura aqui desenvolvida durante o projeto do microprocessador, e esta figura será utilizada para visualização do processamento dos dados, ou seja, o caminho que os mesmos percorrem para a realização de instruções.

Será feita a análise de cada um dos blocos formadores desta arquitetura, para que o leitor possa acompanhar as etapas envolvidas num processamento desta natureza.

Foi adotado o procedimento comum, em projetos computacionais, onde se procede o desmembramento de blocos funcionais muito complexos em vários sub-blocos para que se atinja um entendimento mais sistemático das diversas etapas envolvidas. Vamos nos valer deste recurso para facilitar o entendimento do circuito aqui desenvolvido.

III.3.1 -Filas de Entrada/Saída

As filas ou *buffers* de entrada/saída (fies in/out), são formadas por registradores que realizam a interface do MPH com o *HOST*, constituindo assim o circuito de entrada e saída de dados deste microprocessador.

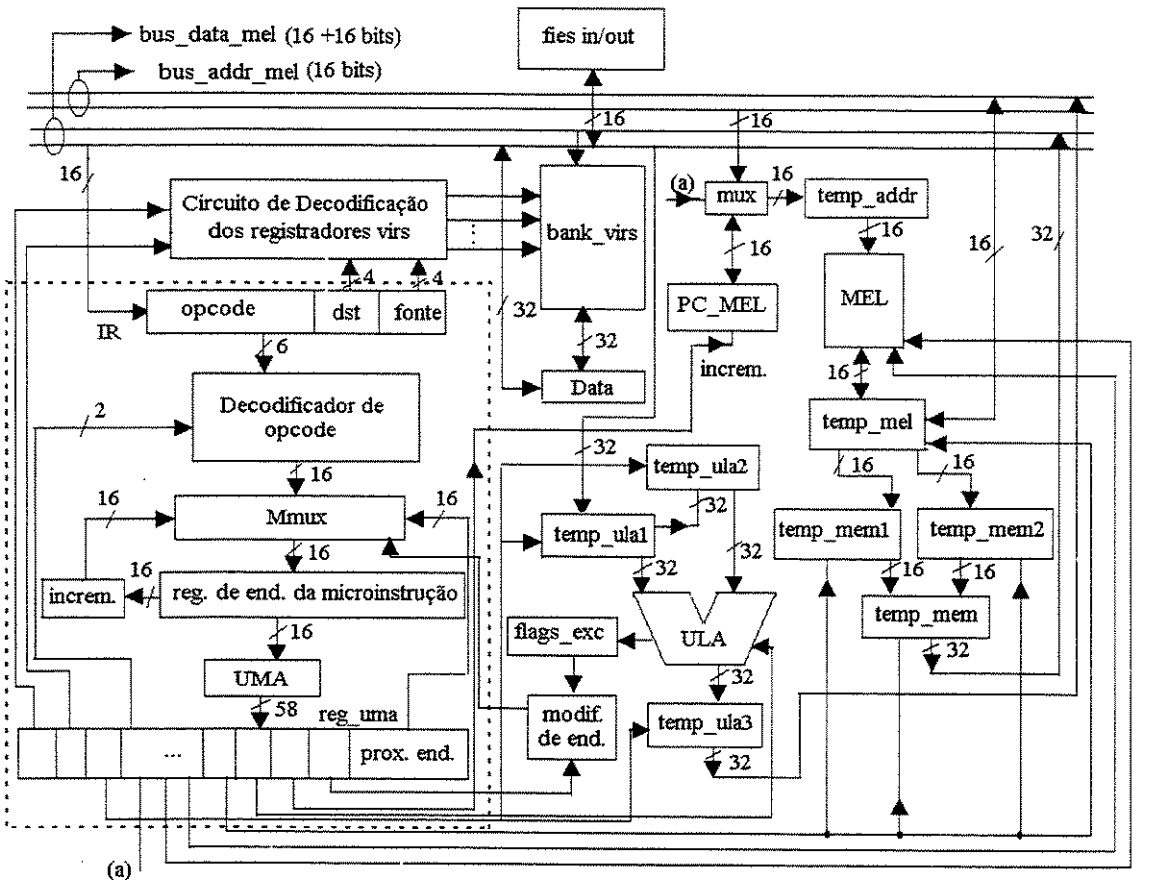


Fig. 3.1 - Diagrama de blocos do microprocessador MPH

O *HOST*, não pertence à arquitetura aqui desenvolvida, sendo responsável pelo particionamento e mapeamento do circuito total a ser simulado e tendo como função a distribuição dos programas de execução dos modelos dos componentes para cada um dos microprocessadores MPH do arranjo de microprocessadores no início de cada iteração da simulação.

Associadas às informações numéricas para os componentes a serem simulados devem estar presentes informações a respeito da direção de recebimento dos dados e da transferência de resultados, ou seja, como se dá o acoplamento do componente ora em simulação com os demais a ele acoplados, das direções de recepção e envio dos sinais

presentes no componente simulado. Estas informações devem ser armazenadas a cada inicio de simulação do componente e devem permanecer em um registrador particularmente destinado a esta função, para que se defina a direção e o sentido da corrente do resultado a cada iteração do processamento.

III.3.2 –Unidade de Controle

Esta unidade é aqui denominada de Unidade de Controle Elementar - UCE, tendo a conotação "elementar" associada à sua função de controlar cada **elemento** (MPH) do circuito simulado. Ela esta em destaque na área tracejada e é constituída pelos seguintes blocos da fig. 3.1:

- Registro de Instrução – IR
- Decodificador de código de operação (opcode)
- Mmux – multiplexador de endereços para a memória de controle
- Registro de endereço da microinstrução – acoplado a um incrementador
- Unidade de Modelos Armazenados – UMA – memória de controle
- Registro da Palavra de Controle

Para um melhor entendimento da função de cada uma destas unidades vejamos inicialmente como se realiza o processamento de dados dentro desta arquitetura.

O programa de simulação do componente, vindo do *HOST* é inicialmente armazenado na memória MEL. A execução deste programa será controlada por microprogramação a partir de sinais de acionamento formadores das palavras armazenadas na memória UMA.

Na etapa de *fetch* ou procura, a instrução é lida da memória MEL e alocada no registro de instrução IR, para que se proceda a decodificação da mesma.

A palavra de instrução contém, entre outras informações, um código de operação ou *opcode* que é enviado a um decodificador para que o mesmo possa ser analisado.

O Mmux fornece o endereço da UMA que contém a palavra de controle para execução da instrução analisada. Definido esse endereço, a palavra de controle é alocada em um registro temporário reg_uma para a devida distribuição aos vários blocos controlados pelos sinais de acionamento nela contidos.

A seqüência de realização das instruções é definida pelo campo de endereço presente em reg_uma, na palavra de microinstrução selecionada pelo Mmux, seja através da definição dada a partir da decodificação do código de operação, seja pela seqüência de etapas necessárias à instrução em curso. Situações de desvio podem ocorrer pela geração de

situações de operandos ou resultados indevidos, como é o caso de operandos considerados exceções estudados no capítulo anterior. Neste caso, o Mmux é informado pela ULA, produzindo o desvio para endereço previamente definido.

Tratada até aqui de forma abrangente, a Unidade de Controle merece especial atenção por ser responsável pelo comando de toda a arquitetura computacional, por isso vamos, a seguir, complementar as informações acima expostas, analisando a função de cada bloco desta unidade individualmente.

III.3.2.1 - Registro de Instrução –IR – *Instruction Register*

O registro de instrução com capacidade para palavras de 16 *bits*, armazena temporiamente a instrução a ser executada pelo microprocessador. O acesso da instrução ao registrador IR se dá através do barramento de dados, interligado à memória MEL. A transferência da palavra de instrução para IR, se dá a partir da informação dada por um sinal de controle presente na memória UMA, para que se proceda a busca de instrução.

III.3.2.2 - Decodificador do Código de Operação (opcode)

A informação presente em IR, ou seja, a instrução, deve ser decodificada. Para todos os tipos, o código de operação é definido por 6 *bits* que possibilitam a definição de 64 instruções distintas. A instrução apresenta representações diferentes para os campos binários de acordo com o tipo de endereçamento estabelecido, conforme já exposto no ítem II.8.

Para o endereçamento por registradores, por exemplo, em instruções de transferência entre registradores teremos 4 *bits* indicativos da origem do dado e 4 *bits* de definição do seu destino, já para instruções por endereçamento indireto os campos de indicação do caso anterior de origem e destino devem agora fazer referência à localização na memória MEL dos operandos a serem processados.

III.3.2.3 - Mmux – Multiplexador de Endereços para a Memória de Controle

Este circuito tem a função de selecionar dentre as diversas escolhas de endereços da UMA, que lhe são apresentados, o endereço conveniente para a etapa do processamento em curso.

Vejamos como se processa a escolha entre as opções disponíveis neste bloco, a partir das informações presentes na palavra de controle:

- Quando o processamento se encontra na etapa de busca de instrução este multiplexador busca o endereço dado pelo decodificador de código de operação.
- Na fase de conclusão da instrução este circuito é carregado com o endereço da UMA que produz a busca de uma nova instrução, ou seja, endereço zero.
- Durante o processamento de instruções formadas de várias etapas, onde mais que uma palavra de controle é necessária, o próximo endereço é dado pelo incrementador de endereço.
- Ocorrendo a existência de alguma exceção durante o processamento matemático, o circuito é informado por uma análise condicional e sinaliza, através de um *bit* enviado ao Mmux, que o endereço escolhido deve ser o contido na palavra de microinstrução.

III.3.2.4 - Registro de Endereço da Microinstrução – associado ao incrementador

Uma vez definido um dos endereços da UMA para a realização de uma etapa do processamento, este registro deve armazenar temporariamente o endereço da palavra de controle a ser buscada na memória de controle –UMA. Em casos onde a instrução é constituída de mais de uma palavra de controle, ocorre o incremento no endereço corrente, para que se processe a seqüência da execução.

III.3.2.5 - Unidade de Modelos Armazenados – UMA – memória de controle

Cabem aqui algumas considerações relativas às modificações ocorridas durante a etapa de definição da arquitetura do microprocessador, no transcorrer do desenvolvimento do projeto.

Em uma versão inicial previa-se que os modelos deveriam estar armazenados na ROM interna, ou seja na UMA - Unidade de Modelos Armazenados interna à Unidade de Microprograma, denominada no capítulo I de UCE - Unidade de Controle Elementar que consiste na Unidade de Controle deste microprocessador.

Com o transcorrer do projeto, concluímos que a Unidade de Modelos Armazenados deveria conter o programa de controle das instruções previstas para o microprocessador e

não, programas de modelos de componentes em particular. Isto nos direcionou à geração de uma arquitetura mais versátil, que poderá vir a ser empregada em tarefas mais abrangentes.

Na Unidade de Modelos Armazenados prevista inicialmente para o projeto, seria mantida estática a programação de cada microprocessador do arranjo previsto no simulador *ABACUS*, o que dificultaria, uma vez definida a programação da ROM de controle, fazer alterações nos modelos de simulação ou expandir sua quantidade.

A programação de definição dos modelos dos componentes está agora armazenada na MEL, viabilizando desta forma, a realização de mudanças nos modelos para simulação.

A Unidade de Modelos Armazenados, está agora com a função de armazenar os microprogramas referentes à resolução das instruções, ou seja, controle da máquina, e a partir dos sinais presentes em cada *bit* ou grupo de *bits*, enviados aos circuitos de validação, é viabilizada a execução dos programas que compõem os modelos.

A UMA apresenta uma estrutura que permite o armazenamento de 256 palavras de 58 *bits*. Os sinais de controle foram adotados a partir do projeto de cada unidade, isto é, das necessidades de controle para cada bloco. A seguir estes controles foram agrupados nas palavras armazenadas na UMA. O formato da palavra de microinstrução está super dimensionado, apresentando mais *bits* que o estritamente necessário, por razões a serem comentadas posteriormente, quando da definição dos campos da palavra de controle.

É comum em estruturas microprogramadas optar-se por sistemáticas *top-down*[7], ou seja, primeiro é projetado um formato de microinstrução, sendo o fluxo de dados definido posteriormente, proporcionando ao projetista grande versatilidade para modificações que venham a ser futuramente necessárias.

III.3.2.6 - Registro da Palavra de Controle

Este é o registro temporário no qual é armazenada a palavra de controle vinda da UMA. A partir da decodificação dos *bits* ou campos de *bits* presentes na palavra, serão alocados os sinais de controles nos diversos blocos formadores do microprocessador.

Este registrador é dividido em dois campos:

- Um campo referente ao controle própriamente dito, necessário para acionar convenientemente todos os blocos da máquina. Deve conter quantos *bits* forem necessários para controlar todas as funções previstas para a máquina.
- Um campo que define o endereço da memória UMA, onde se encontra a palavra de controle para a realização da próxima microinstrução.

III.4 - Campos da Palavra de Controle

Bits	função
0 e 1	controle do decodificador Mmux para a definição de end. da UMA
2	habilitação do mux de endereços da MEL
3 e 4	habilitação de entrada do endereço no PC da MEL
5	habilitação de incremento do PC da MEL
6	desativado
7	habilitação de entrada do end. no registro temporário de endereço da MEL
8 a 10	controle de entrada de dados nos registros temporários da MEL
11 a 13	controle de saída de dados nos registros temporários da MEL
14 e 15	controle de entrada/saída do registrador temp_mem
16	habilitação para leitura da memória MEL
17	habilitação para escrita na memória MEL
18	habilitação à entrada da instrução em IR
19 e 20	habilitação de decodificação do código de instrução
21	habilitação para entrada de dados no registrador data
22	habilitação para a saída de dados do registrador data
23	habilitação para transferência de dados entre registradores VIRs, de uso geral
24 e 25	habilitação para transferência de dados nos registradores temporários da Unidade Lógica e Aritmética - ULA
26 a 29	definição da operação a ser executada pela ULA
30	habilitação da ULA
31 a 41	ociosos
42 a 57	próximo endereço da palavra de controle

É importante ressaltar que os *bits* ociosos foram mantidos prevendo-se otimizações posteriores da máquina, como por exemplo, em processamentos aritméticos onde a Unidade Aritmética e Lógica poderá vir a operar de forma *pipeline*, para redução do tempo de obtenção de resultados. Neste caso as palavras de controle presentes na UMA poderão ser redefinidas para acréscimo de *bits* responsáveis pelo gerenciamento dos *latches* de retenção temporária de resultados matemáticos intermediários.

III.5 - Decodificação da Microinstrução

Os controles presentes na palavra de microinstrução podem ser decodificados para serem enviados aos circuitos de validação das instruções, mas também podem ser decodificados nas próprias unidades a que se destinam. Optamos pela segunda sistemática por se mostrar mais simples e imediata.

Na organização computacional microprogramada é importante que se faça inicialmente um estudo da arquitetura do microprocessador e a seqüência na qual as tarefas venham a ser cumpridas, para com isso definir a temporização de execução das instruções, que deve ocorrer de forma previsível.

A partir das microinstruções, ou seja, dos comandos acionados em cada fase da execução da instrução, deve ser observado o número de ciclos necessários para a realização da instrução.

Estas definições se constituíram, dentro do projeto, em significativas dificuldades na elaboração da UCE.

Foram definidos quatro pulsos de relógio - *clock* para o gerenciamento temporal, permitindo assim a obtenção de uma melhor distribuição das tarefas controladas pela unidade microprogramada.

Definidas as instruções, que serão vistas no ítem III.8, foram definidas as palavras de controle convenientes.

A nossa arquitetura apresenta dois pontos que devem ser compatibilizados para que se execute uma instrução:

- Inicialmente o programa sobre o componente deve estar alocado na MEL.
- O decodificador de *opcode* deve estar descrito de forma a associar diretamente o código de operação com o endereço da palavra de controle, uma vez que a partir da decodificação deste código é que se define o endereço da palavra de controle presente na UMA, que está relacionada àquela instrução. Há portanto uma correlação direta entre a decodificação do código da instrução presente no IR e o endereço que contém a palavra de controle.

III.6 – Registradores de Uso Comum

De grande importância para a realização do fluxo de dados, são os registradores. Indispensáveis no armazenamento de dados temporários, são formados basicamente de *flip-flops*, como já estudado no capítulo anterior. Quando internos à Unidade de Controle,

viabilizam o processo de execução das instruções. Podem ser agrupados em conjunto ou em banco de registradores e permitem o armazenamento temporário de informações durante o trânsito dos dados pelas diversas unidades do processador.

Este projeto define um banco com quinze registradores – *bank_virs* de uso geral, aqui denominados VIRs – *Voltage/Current Registers,* que armazenam temporariamente dados, operandos e resultados das operações matemáticas. Este conjunto pode armazenar também as condições iniciais do componente, presentes na memória MEL para que sejam buscados durante a resolução do equacionamento do modelo. Há também o registrador Data responsável pelo interfaceamento entre os registradores VIRs e demais registradores.

Durante o processamento do modelo os resultados podem ser armazenados temporariamente nos registradores de uso comum, para posteriormente serem alocados na memória MEL.

Dentre os registradores de dados sobre o componente temos prevista a presença de um registrador CR - *component register* que deve estar alocado na fila de entrada/saída (fies) do MPH e define o componente que está sendo processado, REG_DIR - define a direção do fluxo de corrente no componente, POLO_A e POLO_B, DIR_IN e DIR_OUT, contém os polos de ligação do componente. Estes registradores não aparecem explicitados na figura 3.1.

Para a realização das simulações foi suposto que os programas dos componentes já se encontram armazenados na memória MEL. O HOST é que será o responsável pela etapa de armazenamento do programa sobre o componente, ou seja, a escrita nas memórias MEL do arranjo.

III.7 - Memória de Escrita e Leitura – MEL e Registradores a ela associados

Esta memória é carregada durante a fase de alocação do programa a ser executado, ou seja, a definição da seqüência de execução de tarefas previstas pelo programador. A denominação de MEL se deu em alusão à sua capacidade de escrever e ler dados.

O gerenciamento de execução das instruções e utilização dos dados presentes na MEL se dá via micropramação presente na UMA.

O programa presente na MEL deve ser enviado pelo *HOST* tendo em vista a simulação do componente mapeado. Durante e após o processamento sua função é guardar temporariamente os resultados obtidos durante a simulação, para uma posterior utilização pelos demais microprocessadores mapeados e gerenciados pelo *HOST*, no acoplamento entre os MPHs.

Sua alocação interna ao microprocessador, contribui de forma significativa para a dimunuir o tempo de processamento das instruções, devido a redução no trajeto imposto às palavras do programa durante sua utilização.

A atualização da MEL deve ser realizada a cada término de iteração, enviando os resultados da simulação ao *HOST* e recebendo novas informações para continuidade da simulação do circuito mapeado total.

Os registradores associados à memória de escrita e leitura (MEL) são:

- PC_MEL - contador de programas da MEL. Este registrador apresenta também com um circuito incrementador que permite a atualização do ponteiro indicador da palavra da memória a ser utilizada durante a execução do programa. Associado ao PC_MEL existe um multiplexador de endereços da MEL que opera a escolha entre o endereço enviado pelo contador de programas e pelo barramento de endereços. Este último caso é observado em situações onde há desvio da sequência de execução do programa da MEL, como por exemplo, instruções do tipo armazenamento de resultados em determinado endereço da memória.
- TEMP_ADDR - É responsável pelo armazenamento do endereço do dado a ser buscado na MEL. Ele armazena temporariamente o endereço definido pelo multiplexador a ele imediatamente acoplado. Definido este endereço, a memória pode ser acionada para escrita ou leitura dependendo da instrução em curso.
- TEMP_MEL - O dado a ser lido ou escrito na memória é temporariamente armazenado neste registrador para que possa seguir o seu destino por dois caminhos diferentes dentro do processamento. No primeiro caso, onde a informação consiste de uma instrução, a palavra lida da memória deve ser enviada para decodificação no registro de instrução IR e em caso de dados numéricos, há de se proceder duas etapas de leitura da MEL, já que as palavras presentes na memória contem 16 *bits* e os dados envolvidos no cômputo matemático necessitam de 32 *bits*. Os dois registradores citados a seguir, realizam o armazenamento temporário para a recuperação dos dados armazenados em endereços consecutivos da memória .
- TEMP_MEM1, TEMP_MEM2 – São armazenadores de dados de etapas consecutivas de leitura da MEL e em uma etapa posterior seus conteúdos são agrupados no registrador TEMP_MEM de 32 *bits*, permitindo assim que transcorra normalmente a execução de instruções que envolvam valores numéricos para a ULA ou dados para os registradores de uso comum.

III.8 – Repertório de Instruções previstas para o microprocessador

É o repertório de instruções que permite ao programador, lançar mão dos recursos da máquina e através da expressão destas instruções, na forma de programas, obter a solução de problemas.

Dentro do repertório de instruções para a resolução dos modelos, um número reduzido de instruções já é suficiente porque a máquina tendo sido projetada para realização de algorítmos matemáticos por *hardware*, necessita em termos de instrução basicamente a possibilidade de transferência de dados entre os vários blocos da arquitetura, para realizar as tarefas relacionadas às resoluções matemáticas, facilmente acionável por comandos presentes na micropalavra de controle.

Os modos de endereçamento previstos para esta máquina foram inicialmente os seguintes:

- direto
- por registros
- imediato
- imediato32

Vimos no capítulo anterior, detalhadamente, a forma de apresentação dos três primeiros tipos de endereçamento acima citados. O quarto tipo corresponde, da mesma forma, ao endereçamento imediato, contudo a designação 32, ocorre como indicação do uso de palavras de 32 *bits* para os dados expressos na seqüência do código de operação. A denominação somente pela palavra imediato indica o uso de 16 *bits* para o campo de dados.

A arquitetura permite, através de seus comandos, uma ampliação dos modos de endereçamento também como do número de instruções. Para isso deve-se acrescentar as palavras de controle e os códigos de operação nos seguintes programas da arquitetura do microprocessador, descrita em linguagem VHDL e que podem ser encontrados no relatório interno do DSIF [27].

- opcode_analyze – onde estão definidas as correspondências entre os códigos de operação e os endereços das palavras de controle da UMA, aqui armazenadas no *package* pack_uma_words_res;
- package pack_uma_words_res – *package* onde se encontram as palavras de controle propriamente ditas, vinculadas às instruções definidas no opcode_analyze.

O conjunto de instruções previstas para a máquina é:

Instruções de carga/ armazenamento e tranferência de dados

- carga/armazenamento

LDI32 dado	carrega, o dado especificado na instrução, no registrador Data – end. imediato32
LDI_ULA32 dado	carrega o dado de 32 bits expresso na instrução para o registrador temp_ula2 – end. imediato32
LDD32 temp_addr	carrega o conteúdo de memória dado por temp_addr e por temp_addr+1 no registrador temp_mem – endereçamento direto
STD address	armazena os 32 bits presentes em temp_mem no endereço dado por temp_addr - end. direto

- transferência de dados

MOVIR dst_vir, src_vir	transfere o dado presente no registrador src_vir para o dst_vir – end. registro
MOV dst_vir, Data	transfere o dado presente em Data para o registrador definido em dst – end. registro
MOV temp_ula1, temp_mem	MOV temp_ula1, temp_mem
MOV temp_ula2, temp_mem	idem para temp_ula2
MVI32temp_ula1, Data	transfere o dado presente em Data para temp_ula1

Instruções aritméticas (em ponto flutuante)

- modo registrador

ADDFR temp_ula1, temp_ula2	realiza soma em ponto flutuante, entre os operandos presentes em temp_ula1 e temp_ula2, o resultado é colocado em temp_ula3
SUBFR temp_ula1, temp_ula2	idem para a subtração
MULFR temp_ula1, temp_ula2	idem para a multiplicação
DIVFR temp_ula1, temp_ula2	idem para a divisão
ABSFR temp_ula1	expressa o valor absoluto do operando presente em temp_ula1, o resultado é colocado em temp_ula3

SINFR temp_ula1	calcula o seno do dado presente em temp_ula1, o resultado é colocado em temp_ula3
COSFR temp_ula1	calcula o cosseno do dado presente em temp_ula1, o resultado é colocado em temp_ula3

- modo imediato para 32 bits

ADIFI dado	soma o operando presente nas duas palavras imediatas após a instrução com o conteúdo de temp_ula1
SUBFI dado	idem para a subtração
MULFI dado	idem para a multiplicação
DIVFI dado	idem para a divisão
ABSFI dado	expressa o valor absoluto do dado presente na instrução, o resultado é colocado em temp_ula3
SINFI dado	calcula o seno do dado presente em temp_ula1
COSFI dado	idem para o cosseno

Obs: A tangente e as demais funções trigonométricas podem ser calculadas a partir das instruções trigonométricas acima citadas, utilizando expressões compatíveis.

Instruções lógicas não se mostraram necessárias já que todo o processamento matemático dos modelos se processa por *hardware*, porém, a arquitetura permite que estas instruções venham a ser incorporadas a este *menu*, através da descrição de sinais de controle que possam acionar individualmente unidades já pertencentes à arquitetura. Alguns exemplos destes operadores são: rotatores para a esquerda, comparadores de 8, 16 ou 24 *bits* e funções lógicas simples tais como *AND*, *OR* e *XOR*.

Instruções de Desvio do Fluxo de Dados

JPI address	Salto incondicional para o endereço dado em temp_addr – end. imediato

Os desvios condicionais ocorridos devido a resultados de exceções não precisam figurar como instruções porque já estão vinculados ao comportamento da ULA nestas situações.

Instruções de Desvio de Controle

Interrupções devem ocorrer em caso de encontro de exceções nos operandos. Esta situação já é sinalizada pelo *hardware* dos operadores matemáticos.

III.8.1 - Base de Tempo para Execução das Microinstruções

Esta se mostrou uma etapa de difícil elaboração na conciliação do tempo de execução de cada etapa referente a uma microinstrução, com a sequência necessária para a realização da instrução como um todo.

São diversos passos, dependendo de cada instrução, que devem ocorrer desde a sua busca na memória até a sua completa execução. Estes passos devem estar devidamente sincronizados para que as diversas unidades da arquitetura venham a operar de forma conveniente.

Foram definidos como base de tempo, os quatro ciclos de relógio apresentados na figura 3.2, com atraso de 50ns entre si e com a duração de 200ns sendo 50ns em nível alto e 150ns em nível baixo.

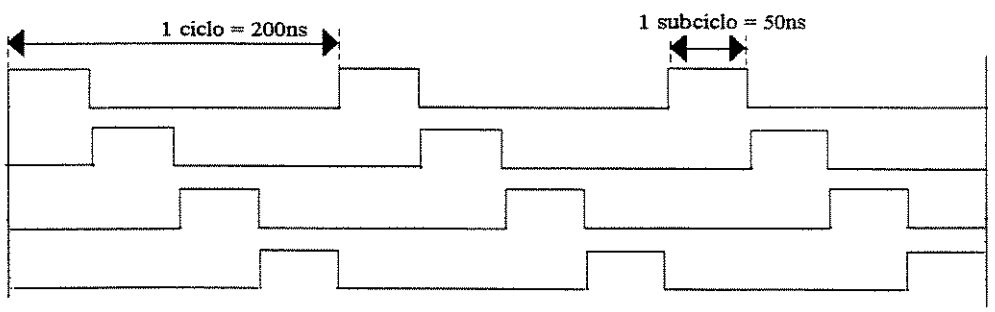


Fig.3.2 - Diagrama da base de tempo - *clock*

Estes relógios foram distribuidos na arquitetura a partir de uma previsão inicial do comportamento do fluxo de dados. Algumas instruções foram monitoradas na entrada da palavra de controle, e com isso foi possível realizar ajustes que permitiram um desempenho satisfatório no tempo e seqüência de realização das microinstruções.

III.8.2 – Algumas Instruções Descritas em Termos de Microinstruções

A execução de uma instrução se dá a partir da definição de microinstruções, ou seja, palavras onde os controles para as diversas unidades da arquitetura devem figurar.

As unidades devem ser rigorosamente temporizadas para que os endereços e dados envolvidos sejam devidamente computados.

Vejamos a seguir, como devem ocorrer as transferências entre registradores para que se realize uma instrução de carga imediata de um dado de 32 *bits* no registrador Data.

Este exemplo busca esclarecer a maneira utilizada para definição das microinstruções referentes à execução das instruções.

LDI32 dado – carrega o dado, especificado na instrução, no registrador Data

- tipo de endereçamento: imediato32

Para esta instrução foram utilizadas três palavras de microinstrução formadas por microcontroles, sendo possível a execução de duas microinstruções a cada palavra de controle. Os dados presentes na MEL se apresentam em palavras de 16 *bits*, mas os registradores de uso comum e aritméticos utilizam 32 *bits* portanto, dois passos de transferência são necessários para a recuperação dos dados da memória.A transferência de dados foi programada da seguinte forma:

- temp_mel <= dado_1 – transferência dos *bits* mais significativos do dado da memória para o registrador temporário temp, de saída de dados da MEL;
- temp_mem1 <= temp_mel – armazenamento temporário do dado_1 em temp_mem1;
- temp_mel <= dado_2 – transferência dos bits menos significativos do dado;
- temp_mem2 <= temp_mel – armazenamento temporário de dado_2 em temp_mem2;
- temp_mem <= temp_mem1 & temp_mem2 – concatenação dos dados contidos nos registradores temp_mem1 e temp_mem2, para recuperação do dado completo;
- Data <= temp_mem – transferência de temp_mem para o registrador Data.

A instrução transcorreu no intervalo de 1,2 μs em 6 ciclos de *clock* de 200 ns.

Instruções de resolução matemática foram realizadas num intervalo de tempo máximo de 2,4μs, para multiplicação ou divisão e 1,0 μs para soma ou subtração. As operações matemáticas simplesmente exigem um acionamento de comando de execução da instrução e um controle no decodificador da ULA, na definição da operação a ser realizada.

Para a instrução de transferência de um dado do registrador Data para algum dos registradores do banco de registros VIR utilizamos a seguinte microoperação:

dst_vir <= Data, onde *dst* se refere ao código do registrador escolhido para destino, explícito na instrução e decodificado dentro de *bank_virs*. Foram necessários 400ns, para esta execução.

III.9 - Modelos

Os modelos a serem adotados nas simulações, foram anteriormente [5],[6] definidos a partir do comportamento descrito em termos de expressões matemáticas dos seguintes componentes: resistor, capacitor, diodo, fonte de tensão contínua, fonte de tensão quadrada

e extensor (interface entre componentes com mais de dois terminais, por exemplo quadripolos).

Os equacionamentos previam não só componentes de comportamento linear, também como os não lineares. Neste último caso, as descrições que, por exemplo, envolvem funções logarítmicas ou exponenciais, podem ser obtidas fazendo uso de programas, tabelas, séries de funções ou outros algorítmos que utilizem os operadores matemáticos aqui desenvolvidos.

Dentre os modelos formalizados, utilizamos o resistor como componente para teste do *hardware* do MPH e obtivemos sucesso na resolução do programa deste modelo.

O resistor foi modelado pelo seguinte conjunto de equações:

$I = (I + V/R)/2.0$

$V = I * R$

sendo o I no primeiro membro (iteração atual) dependente do I do segundo membro (iteração anterior)

R é o valor do resistor

Estas expressões são validas somente para resistores sujeitos a tensões contínuas porque no caso onde as correntes e tensões são alternadas, o formato do sinal periódico, deverá estar explícito nas expressões.

III.9.1 - Programa para o Cálculo da Corrente no Resistor – programa armazenado no package pack_mel _instr

Conhecidas as expressões que caracterizam os componentes, passemos à descrição das suas expressões matemáticas em linguagem *Assembly* da máquina projetada.

A informação sobre as características da ligação a que o componente está submetido, tais como, os polos de ligação do componente no circuito e o sentido de direção da corrente devem ser armazenados nas *fies* de entrada e saída para orientar o sentido do fluxo do sinal da próxima iteração do mapeamento do circuito e portanto não fazem parte do *menu* de instruções para resolução dos modelos propriamente ditos.

$I = (I + V/R)/2.0$

- LDI32 dado - carrega o registrador Data com os dados (32 *bits*) presentes na instrução na seqüência do código de operação – valor da tensão
- MOV temp_ula1, Data – move o conteúdo presente em Data para temp_ula1 – alocação do dividendo na ULA

- LDI32 dado - carrega o registrador Data com os dados (32 bits) presentes na instrução na seqüência do código de operação – valor do resistor
- MOV temp_ula2, Data – move o conteúdo presente em data para temp_ula1 – alocação do divisor na ula
- DIVFR temp_ula1,temp_ula2 – realiza a divisão entre os conteúdos de temp_ula1 e temp_ula2
- MOV temp_ula1,temp_ula3 – transfere o resultado da divisão para o registro temp_ula1
- LDI32 Data - carrega o registrador Data com os dados (32 *bits*) presentes na instrução na seqüência do código de operação – valor da corrente
- ADDFR temp_ula1,temp_ula2 – realiza a soma entre os conteúdos de temp_ula1 e temp_ula2
- MOV temp_ula1,temp_ula3 – transfere o resultado da soma para o registro temp_ula1
- DIVFR temp_ula1,temp_ula2 – realiza a divisão entre os conteúdos de temp_ula1 e temp_ula2
- MOV Data, temp_ula3 – transfere o resultado de temp_ula3 para o registrador Data.
- STD 300 – armazena o resultado presente em Data no endereço 300_H da memória MEL

III.9.2 - Programa para o Cálculo da Tensão no Resistor

V = I * R

- MOV temp_ula1, temp_ula3 – transfere o resultado da soma para o registro temp_ula1
- MULFR temp_ula1, temp_ula2 – realiza a multiplicação entre os conteúdos dos registros temp_ula1 e temp_ula2. O resultado está presente em temp_ula3.
- MOV Data, temp_ula3 – transfere o resultado de temp_ula3 para o registrador Data.
- STD 301 – armazena o resultado presente em Data no endereço 301_H da memória MEL

Pela programação dada acima, podemos notar que a obtenção de tensão e corrente em um componente se dá de forma bem compacta. É certo que modelos mais elaborados

com um número maior de expressões vão incorrer em maior tempo de processamento, porém, os modelos são executados de forma imediata, a partir de simples acionamentos de sinais de controle.

É interessante ressaltar também que a programação para a resolução do modelo envolveu apenas algumas poucas instruções, tornando o processamento mais rápido. Esta característica se deve ao fato do processamento se realizar fundamentalmente por *hardware*.

III.10 - ULA - Unidade Lógica e Aritmética em Ponto Flutuante

Como já citado, esta unidade foi projetada para operar em ponto flutuante e, como já exposto no capítulo II desta tese, se mostra de alta complexidade técnica.

Grande parte do tempo dispendido na elaboração deste trabalho, foi destinado ao desenvolvimento desta unidade . Na proposta deste projeto foi enfáticamente definida que o processamento matemático fundamental, deveria ocorrer por *hardware*, não tendo sido possível desta forma fazer uso, de pacotes definidos por funções pré-existentes no pacote de simulações *Mentor Graphics*. Particularidades nas descrições dos circuitos, na linguagem VHDL, produziram um importante fator de complexidade a ser vencida.

Os procedimentos para a definição da arquitetura deste bloco foram baseados na teoria vista detalhadamente no capítulo anterior, portanto, nos limitaremos a comentar algumas particularidades observadas durante a fase de definição da arquitetura como um todo.

Teceremos considerações a respeito das particularidades, dificuldades e soluções encontradas durante o projeto.

A primeira consideração a ser feita é quanto ao formato dos dados numéricos. Fizemos uso da palavra de 32 *bits* para a determinação dos operandos, sendo distribuidos em 1 *bit* de sinal, 8 *bits* para o expoente e 23 para a mantissa sendo que mais um *bit* foi acrescentado à mantissa, como sinalizador de normalização, para que durante todo o desenvolvimento matemático a mesma fosse mantida.

Dos blocos necessários para o somador em ponto flutuante, vistos no capítulo anterior, enfatizaremos o projeto do somador de mantissas, que realiza a soma de operandos com vinte e quatro *bits* (considerando-se o *bit* escondido). Antes de nos dirigirmos para as considerações sobre o somador propriamente dito, veremos a arquitetura dos registradores imediatos à Unidade Aritmética e Lógica, que permitem o acesso dos dados e obtenção dos resultados neste bloco.

III.10.1 - Registradores internos à Unidade Aritmética e Lógica

A Unidade Aritmética e Lógica possui dois registradores para armazenamento temporário dos dados de entrada (TEMP_ULA1 e TEMP_ULA2) e um registrador para a saída dos resultados TEMP_ULA3. Conta ainda com um registrador para *flags* de exceção, que podem determinar a mudança ou interrupção no fluxo de execução da operação.

III.10.2 - Somador/Subtrator em Ponto Flutuante

As operações de soma e a subtração, podem ser realizadas por um mesmo circuito uma vez que a subtração é equivalente à soma, a menos da troca do sinal do operando a ser subtraído. O sinal dos operandos é que vai definir a operação efetiva, ou seja, aquela que realmente ocorre entre os operandos, portanto pudemos tratar as duas operações de forma geral, como um somador.

O somador, apresenta a obtenção do *carry* de saída como um fator limitante em sua velocidade e como já exposto diversas são as técnicas utilizadas para se resolver este problema. A referência [17], faz uma ampla abordagem das diversas técnicas existentes para a agilizar a obtenção do *carry* resultante e apresenta a técnica de *carry lookahead* como uma das melhores.

A técnica de *carry lookahead* [32],[33],[35] permite uma maior rapidez de obtenção de resultados quando comparada à tradicional *ripple carry*, sem contudo exigir controles muito sofisticados, como é o caso da técnica por *carry-skip*.

Uma característica importante desta estrutura é a de que simples funções lógicas AND e XOR aplicadas aos operandos, permitem o conhecimento do valor de *carry* no final da operação, antes mesmo da conclusão da mesma.

Estão ilustrados na figura 3.3(a) os fatores que podem ser calculados precocemente quando do início da operação e permitem a agilidade na obtenção do resultado da soma. Se denominam *carry generate* (G) e *propagate* (P), fazendo alusão ao *carry* que ocorrerá, respectivamente, local e futuramente devido a cada par de *bits* de operandos nas entradas.

Os sinais G e P, entram a seguir na rede lógica, apresentada na figura 3.3(b), para conformar os sinais previstos pelas expressões 2.3 e 2.4, vistas no capítulo anterior, para obtenção de *carry* e soma resultantes.

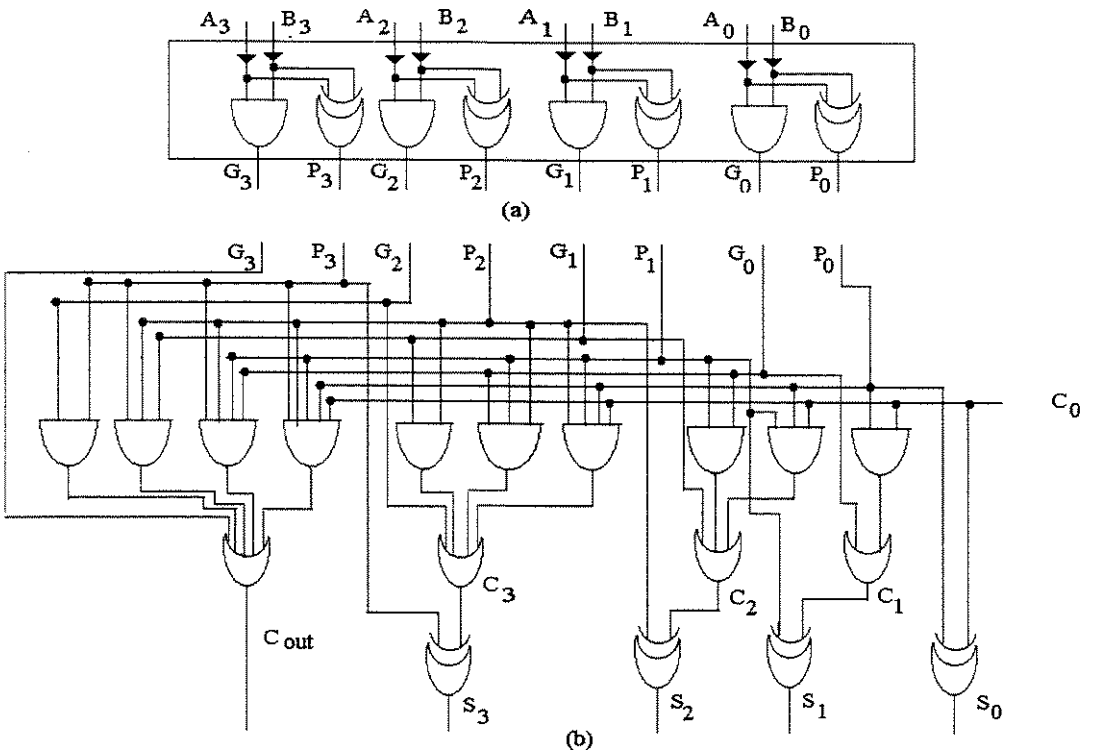


Fig. 3.3 (a) - Circuito gerador de *carrys generate* e *propagate*
(b) - Uma unidade de *carry lookahead* para quatro *bits*

Esta estrutura se mostrou conveniente para as aplicações necessárias ao projeto, por apresentar uma configuração simples, e é um tipo de descrição clássica para somadores em descrições por linguagem VHDL. Para esta descrição, conformamos os 24 *bits* referentes à mantissa, em 6 grupos de 4 *bits*, onde cada grupo apresenta a estrutura mostrada na figura 3.3.

O microprocessador conta também com o acionamento para a subtração, somente por motivos de transparência disponível ao usuário, uma vez que a operação efetiva estudada no capítulo anterior vai ser basicamente determinada pelos sinais dos operandos envolvidos na operação, podendo-se resumir às soma e subtração em um único caso de operação.

III.10.3 – Mutiplicador/Divisor em Ponto Flutuante

O circuito para a obtenção da multiplicação e divisão em ponto flutuante corresponde ao diagrama de blocos exposto na figura 2.16 do capítulo anterior.

Como o circuito citado já foi apresentado com grande nível de detalhes quanto à sua arquitetura vamos aqui dar especial atenção à definição do arranjo do multiplicador/divisor propriamente dito, porque se mostrou uma estrutura de grande elaboração técnica devido à complexidade relacionada com o seu tamanho e suas interconexões.

Visualizemos na fig. 3.4 a rede, na ilustração dada em menores proporções à necessária para o projeto do MPH.

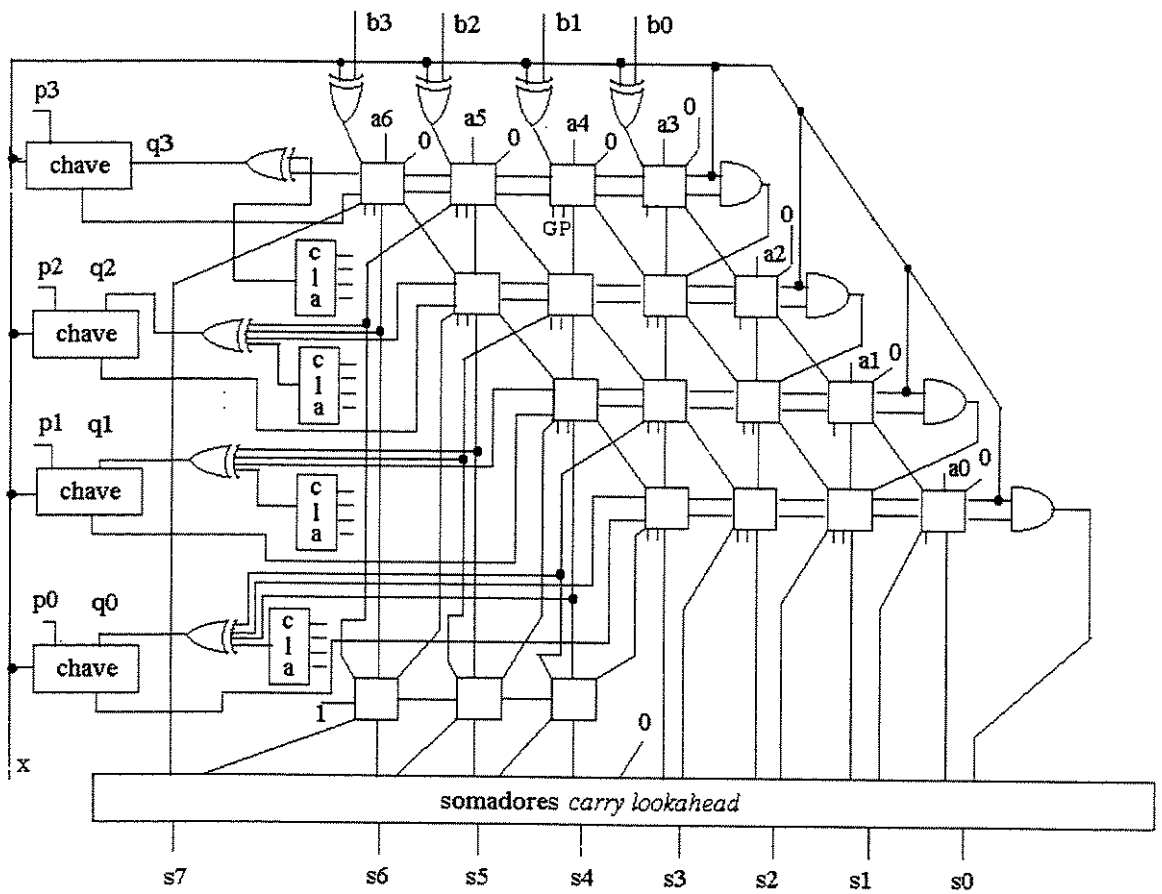


Fig.3.4 - Arranjo multiplicador/divisor de 6 por 4 *bits* [19]

Em nosso caso as componentes horizontais são formadas por 24 *bits* cada e essas linhas foram interligadas em número de 24. Devido ao grande número de *bits* envolvidos, para a descrição VHDL, utilizou-se a estratégia de repartir a rede em linhas, que foram interligadas posteriormente em colunas, respeitando-se os deslocamentos sucessivos.

Como ponto de partida para a descrição deste bloco, foi utilizada a referência [19]. Basicamente este algorítmo utiliza somadores completos que tratam individualmente cada par de mesmo índice de bits dos operandos.

O seu funcionamento ocorre da seguinte forma: Quando a operação é multiplicação, os operandos válidos são b e p, alocados respectivamente na diagonal e horizontal da rede.

A soma parcial devida a cada linha é formada pelo produto de b por p mais as componentes devidas aos *carries* e aos resultados da soma da linha anterior. Estas parcelas devem ser somadas, gerando *carries* e resultados que serão computados na próxima linha da seqüência. O produto está representado na figura 3.4 pelo sinal s, composto de 8 *bits* resultante do produto de 4 por 4 *bits* presente nas entradas b e p respectivamente.

No caso da divisão valem os operandos localizados nas posições a (dividendo) e b (divisor). Aqui os operandos são subtraídos *bit* a *bit* em cada linha. Para a determinação do *carry* de saída de cada uma das parcelas da horizontal utilizamos a técnica de *carry lookahead.*

Esta forma de obtenção de quocientes, conhecida por *restoring,* foi estudada no capítulo anterior e é uma prática comum quando se pretende conciliar em um mesmo circuito a divisão e a multiplicação.

O sinal x é utilizado para a seleção da operação desejada e atua na entrada do operando b complementando-o de 2, para o caso da operação de divisão e mantendo-o inalterado para o caso de multiplicação. A chave controlada por x permite o fluxo de entrada do multiplicando p ou de saída do resultado q correspondente a cada linha, na divisão.

Foram necessárias algumas modificações desta arquitetura para adequá-la às descrições VHDL desenvolvidas. As formas de descrição e alocação dos somadores de *bit* a *bit* se tornou muito complexa quanto à definição de índices, porque as linhas devem ser sucessivamente deslocadas de uma posição.

III.10.4 - Funções Trigonométricas

As funções trigonométricas foram resolvidas a partir de tabelas ou *lookup tables.* Estas tabelas são memórias fixas onde se armazenam valores pré-definidos e o acesso é imediato. Esta forma se mostrou propícia porque para o caso de senos e cossenos o intervalo de variação é de –1 a +1, ou seja, é um intervalo pequeno e bem definido de valores.

Os ângulos foram inicialmente analisados e reduzidos por um programa ao primeiro quadrante para que a definição da equivalência de valores trigonométricos, para restringir ainda mais o intervalo de valores angulares a serem armazenados.

Foi adotada uma memória de 512 palavras tendo sido utilizadas 256 para o armazenamento do seno e 256 para o cosseno. Os ângulos foram uniformemente distribuídos nas tabelas de 0 a $\pi/2$, em intervalos de 0,006 radianos, com valor máximo de 1,570 radianos correspondente a $\pi/2$.

III.11 - Comprovação de resultados

Para comprovação dos resultados obtidos nas simulações dos circuitos através do simulador *quicksim* da *Mentor Graphics*, foi utilizada a calculadora disponível *on line* nas estações de trabalho SUN. A representação utilizada para a visualização foi a hexadecimal.

A representação numérica utilizada para estas análises foi a hexadecimal devido a sua disponibilidade de uso e compactabilidade quando valores com grande número de bits na forma binária, são testados. Estas representações, embora práticas, devem ser cuidadosamente analisadas em casos onde o número de *bits* não é múltiplo de quatro (divisão de *bits* para a representação em hexadecimal), porque podem mascarar erros inexistentes.

III.12 - Pinagem do Microprocessador

Após a definição de toda a arquitetura mostrada acima vamos tecer alguns comentários a respeito da pinagem prevista para o microprocessador, que não deve ser vista como uma proposta definitiva porque este projeto deverá ser posteriormente testado na sua função de simulador para que se faça então, os acertos convenientes nesta primeira tentativa.

É previsto o uso de 32 pinos para a entrada e saída de dados.

O processador deverá permitir a sua conexão a outros quatro microprocessadores, ou seja, um em cada direção (norte,sul,leste,oeste) de sua vizinhança, como também poderá ser utilizado individualmente em outras tarefas afins. Desta forma prevemos que estes 32 *bits* possam ser divididos em 4 partes, sendo 8 *bits* para cada direção. Sendo assim possívelmente será necessário que este particionamento seja realizado pelo *HOST*, com os dados sendo inseridos em grupos de 8 *bits*.

Oito *bits* devem ser utilizados para o endereçamento da memória MEL. Um pino em cada direção deve habilitar a entrada e saída dos dados.

Quatro pinos, distribuídos um em cada uma das direções Norte, Sul, Leste e Oeste devem ser reservados à informação das direções dos fluxos de corrente e das ligações a que o componente está sujeito. Estes sinais podem ser sequenciais e bidirecionais. Outros pinos a serem alocados são: alimentação, *start*, que dá habilitação para início da execução da simulação, *reset* que inicializa todos os registradores presentes no microprocessador, *hold*, que é habilitado se ocorrer alguma irregularidade de dados de entrada, ou de resultados

parciais e demais pinos que possam vir a ser necessários para o acoplamento ao *HOST* e demais microprocessadores formadores do arranjo simulador.

III.13 - Comentários

Foram aqui comentados de forma individualizada os blocos formadores do microprocessador. Cada um desses blocos aqui estudado foi descrito em linguagem VHDL, compilado e simulado em ambiente *Mentor Graphics*. Foi grande o número de programas necessários para que se atingisse a arquitetura aqui apresentada não tornando-se viável anexá-los a esta tese, porém os mesmos se acham disponíveis em um relatório interno [27] do Departamento – DSIF.

No capítulo seguinte, comentaremos os principais mapeamentos dos circuitos – *port maps* e no capítulo V far-se-á a apresentação de saídas gráficas dos resultados das simulações para os blocos mais relevantes da arquitetura.

Capítulo IV

Descrições VHDL e simulações dos blocos pertencentes à arquitetura do MPH

IV.1 - Introdução

Apresentaremos as descrições dos blocos formadores do MPH, em linguagem de descrição de circuitos VHDL. Esta linguagem tem sido amplamente usada [28]-[31] para descrever circuitos nos mais diversos graus de abstração, viabilizando projetos de grande nível de complexidade.

As ferramentas *Mentor Graphics* utilizam estas descrições para a simulação do comportamento dos circuitos digitais possibilitando a previsão de resultados. Valemo-nos destes recursos para a simulação dos circuitos dos diversos blocos da arquitetura, descritos em linguagem VHDL. Inicialmente as simulações foram feitas de forma individual; os blocos foram sendo progressivamente acoplados resultando na arquitetura total do MPH.

Vamos apresentar apenas os blocos mais significativos, porque o projeto exigiu um grande número de programas, sendo a sua apresentação de forma integral inviável.

As descrições apresentadas pretendem nortear o leitor ao entendimento do projeto; um relatório interno do DSIF [27], apresenta todas as descrições que possam ser necessárias para um maior aprofundamento.

Devido a grande complexidade do projeto o particionamos em quatro blocos:

- somador/subtrator em ponto flutuante;
- multiplicador/divisor em ponto flutuante;
- unidade de controle microprogramada;
- tabelas trigonométricas.

Estes quatro grandes blocos foram descritos na forma de *port map*, onde inicialmente foram tratados os blocos pertencentes ao mapeamento e a seguir estes blocos foram interligados em programas mais abrangentes.

Veremos a descrição dos quatro grupos citados acima, esclarecendo a funcionalidade de cada um em particular. O entendimento destas descrições pode ser melhor observado acompanhando-se as arquiteturas descritas com as figuras correspondentes, porisso vamos reapresentá-las neste capítulo.

A seqüência será dada nesta ordem: nome do bloco dentro do *port_map*, nome do programa utilizado e nome correspondente no diagrama de blocos da figura citada e finalmente uma breve descrição da função exercida.

IV.2 – Somador/Subtrator em Ponto Flutuante

A figura 4.1, correspondente à figura 2.10 do capítulo II. e ilustra o circuito

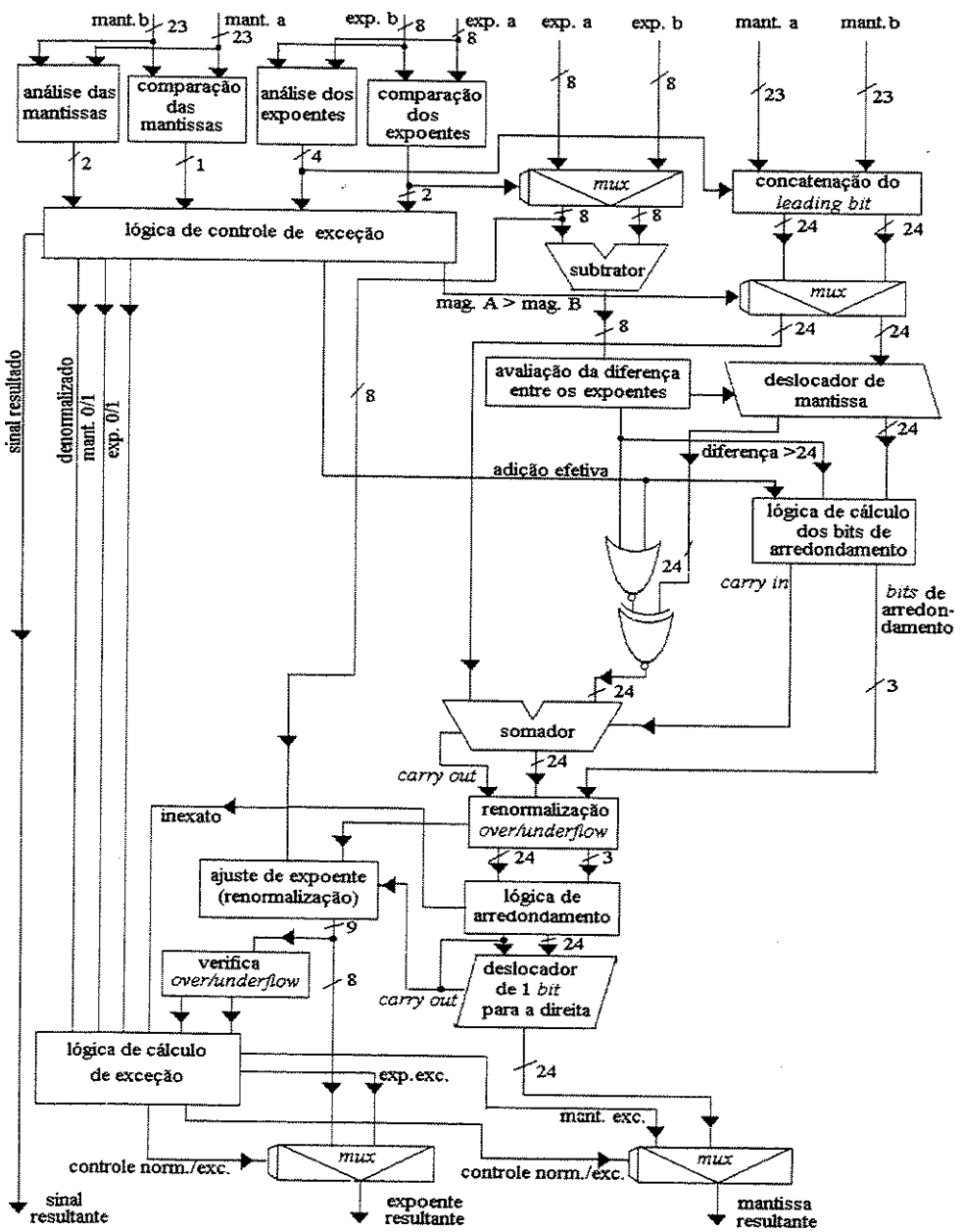


Fig. 4.1 - Somador/subtrator em ponto flutuante

somador/subtrator em ponto flutuante.

Nome do programa de mapeamento geral: map_adderfp_parts1.

Programas formadores:

- bloco 1: adder_exp_127 – adder_127_excess – somador de excesso de 127 (não representado na figura) - utilizado para deslocar a referência de expoentes, dispensando a indicação explícita do sinal.
- bloco 2: expoente_analyze – exp_analyze – análise dos expoentes – analisa os expoentes para detetar se não são casos de exceção.
- bloco 3: compare_expoente – compare_exp – comparação dos expoentes – os expoentes são comparados para que se defina o maior. Um controle é enviado ao bloco 4 para que este aloque o expoente maior em exp. a e o menor em exp. b, ou seja, prepara os expoentes de forma a não gerar valores negativos, quando forem subtraídos.
- bloco 4: mux_exp – multiplex_exp – multiplexador de expoente – a partir do sinal do bloco 3, define a ordem dos expoentes para realização da subtração necessária para a avaliação do número de bits a que estará sujeita a mantissa do menor operando.
- bloco 5: mantissa_analyze – mant_analyze – análise das mantissas - analisa as mantissas para ver se estão dentro dos valores permitidos para cálculo.
- bloco 6: lead_bit_man – concat_lead_bit – concatenação do leading bit – associa mais um bit às mantissas a e b da entrada, inicialmente formadas de 23 bits. Na saída tem-se 24 bits. Este bit acompanha todo o processamento da mantissa indicando sua normalização.
- bloco 7: compare_mantissa – compare_man1 – comparação das mantissas – complementa a comparação dos expoentes para a definição do operando de maior magnitude.
- bloco 8: mux_man – multiplex_man – mux – O sinal de controle dado pelo bloco 7 coloca a mantissa maior na entrada a e a menor na entrada b.
- bloco 9: sut_expoent – subt_exp – subtrator – realiza a subtração entre os expoentes para definição da diferença entre as mesmas para que se proceda o deslocamento do operando com menor magnitude de quantos *bits* forem necessários para igualar o seu expoente ao maior.
- bloco 10: shift_man – right_shift – deslocamento de mantissa – após a avaliação da diferença entre os expoente, este bloco desloca para a direita a mantissa do operando de menor magnitude de tantos *bits* quantos forem necessários para que o valor do operando não se altere (compensação para que o expoente maior prevaleça).
- bloco 11: exc_control – ctrl_exc – lógica de controle de exceção – responsável pelo destino dado às exceções.

- bloco 12: bits_arred_calc – calc_bits_arred – lógica de cálculo dos bits de arredondamento – a mantissa deslocada, correspondente ao operando de menor magnitude deve ser avaliada para que o efeito do deslocamento seja incorporado ao resultado final.
- bloco 13: adder_man – adder_subt_pf - somador de mantissas – soma ou subtrai as mantissas conforme a operação efetiva definida.
- bloco 14: renormalize_man – renorm_man – realiza a renormalização do resultado da soma entre as mantissas, caso seja detetado algum overflow ou underflow. Estes casos devem ser sinalizados ao bloco 17, para que se realize posteriormente um ajuste no expoente.
- bloco 15: arredon_mantissa – lógica de arredondamento – Permite a incorporação dos bits resultantes da renormalização à mantissa resultado.
- bloco 16: right_shifter_man – right_one_bit – deslocador de 1 bit p/ a direita – deslocamento da mantissa para mante-la no formato normalizado, caso ocorra um overflow devido ao arredondamento da mantissa.
- bloco 17: renorm_exp_adjust – exp_adjust – ajuste de expoente (renorm) – alteração do expoente (incremento de 1) no expoente inicialmente tomado como resultado caso tenha ocorrido algum overflow durante a etapa de renormalização ou arredondamento da mantissa.
- bloco 18: over_under_exp – over_under_exp - verifica under ou overflow – analisa o resultado do bloco 17 e sinaliza a ocorrência caso haja uma ultrapassagem do valor permitido pelo formato.
- bloco 19: calc_exc – calc_exc_logic – lógica de cálculo de exceção – recebe *bits* de resultado de análise de overflow e underflow e caso tenha ocorrido resultado fora do valor do formato, a exceção será sinalizada à saída na forma de *flag*.
- bloco 20: mux_exp_end – multiplex_exp_end – coloca na saída o resultado do expoente calculado.
- bloco 21: mux_man_end – multiplex_man_end – coloca na saída o resultado da mantissa calculada.
- bloco 22: subt_excess_127_exp – subt_127_excess_8 – não representado na figura – subtrai o excesso de 127 do valor do expoente obtido.

IV.3 – Multiplicador/Divisor em Ponto Flutuante

A figura 4.2 correspondente à figura 2.11 do capítulo II. e ilustra o multiplicador/divisor em ponto flutuante.

Nome do programa de mapeamento geral: map_multidivi_fp.

Programas formadores:

- bloco 1: muldi_adder_exp_127 – adder_127_excess – não representado na figura – coloca os expoentes no formato padrão.
- bloco 2: muldi_operators_definition – operators_definition – não representado na figura – reconhece como os operandos devem ser alocados na rede. A alocação se processa diferentemente para a multiplicação e divisão.

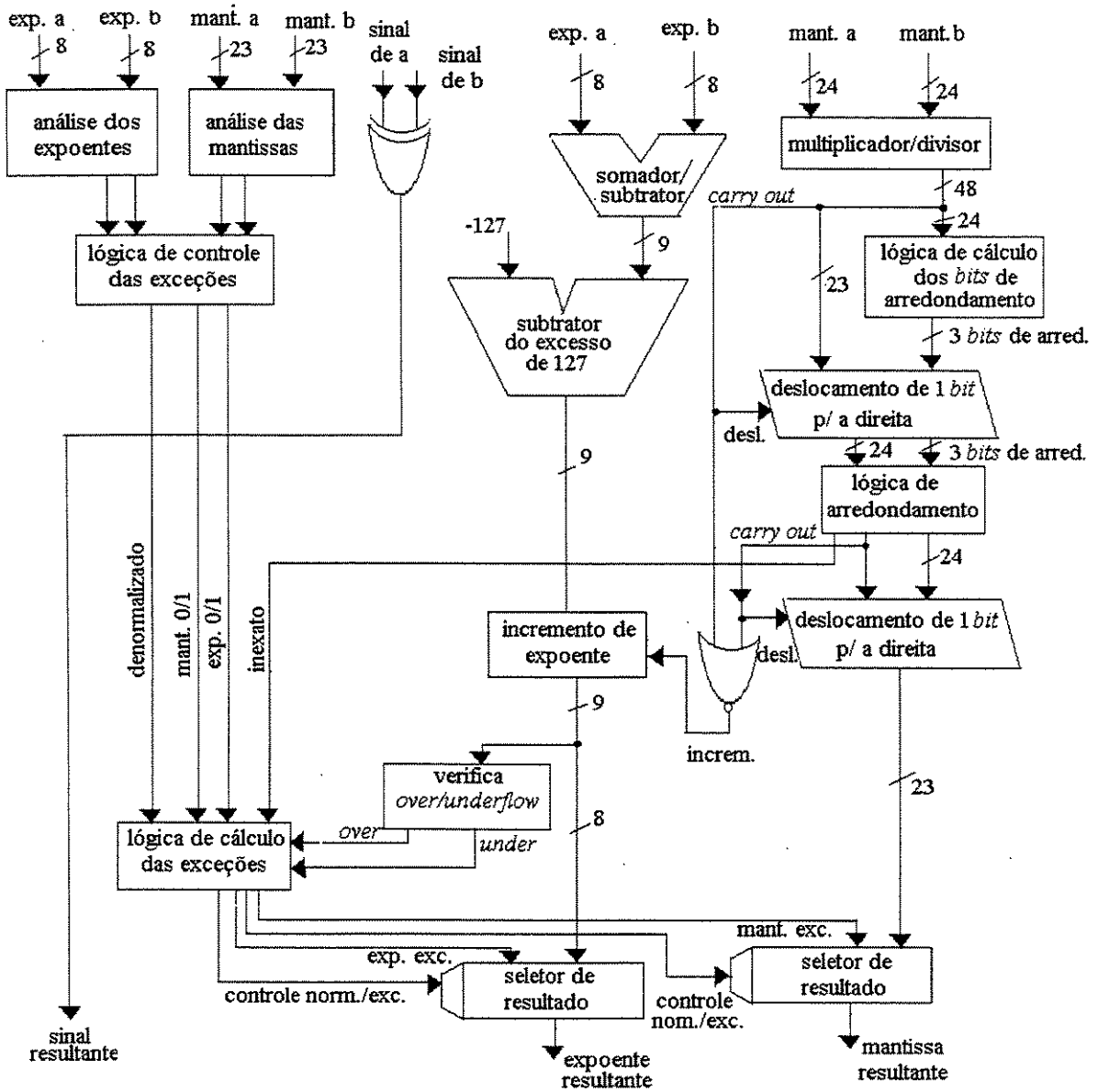


Fig. 4.2 - Multiplicador/Divisor em ponto flutuante

- bloco 3: muldi_exp_analyze – exp_analyze – análise dos expoente – verifica se os expoentes estão dentro do formato e se não constituem nenhum caso de exceção.
- bloco 4: muldi_mant_analyze – mant_analyze - análise das mantissas – semelhante ao bloco 3, aplicado à mantissa.
- bloco 5: muldi_exc_control – ctrl_exc_muldi – lógica de controle das exceções – caso tenha ocorrido algum operando na forma de exceção o controle de exceção pode interromper o processamento.
- bloco 6: muldi_add_expoents – add_sub_exp – somador/subtrator – realiza a soma dos expoentes caso a operação seja multiplicação ou subtração quando se tratar de divisão.
- bloco 7: subt_excess_127 – subt_127_excess – subtrator do excesso de 127 – retira um dos excessos presentes no expoente resultado, já que ambos expoentes foram sujeitos a este acréscimo, na entrada dos dados.
- bloco 8: lead_bit_man – concat_lead_bit_muldi – não representado na figura – acrescenta um bit escondido que vai permitir, durante todo o procedimento matemático com a mantissa, a verificação de que a normalização está sendo mantida.
- bloco 9: interface_operands – interf_concat_lead_bit_muldi - não representado na figura - Dos bits previstos para entrada do multiplicador e divisor, faz a seleção dos operandos para um dos operadores, dependendo da operação a ele indicada.
- bloco 10: mult_div – map_muldi_test4 – multiplicador/divisor – Seleciona e realiza a operação definida pelo código de operação.
- bloco 11: muldi_bits_arred_calc – calc_bits_arred_muldi – lógica de cálculo dos *bits* de arredondamento – analisa os *bits* a serem arredondados para que seja recuperado o formato, já que o bloco 10 envia 48 *bits* como resultado da multiplicação.
- bloco 12: right_shifter1_man – muldi_right_one_bit – deslocador de 1 bit para a direita – Os bits arredondados devem ser incorporados ao primeiro resultado da multiplicação.
- bloco 13: muldi_arredon_mantissa – arredon_man_muldi – lógica de arredondamento – Após o deslocamento de 1 bit para a direita pode ocorrer a necessidade de se fazer um novo arredondamento, para que sejam incorporados os *bits* arredondados.
- bloco 14: right_shifter2_man – muldi_right_one_bit1 – deslocamento de 1 bit para a direita – Após o arredondamento da mantissa pode ocorrer *overflow*. Neste caso a mantissa deve ser deslocada para a direita acompanhada de um sinal para incremento no bloco 16.

- bloco 15: logic_inc_expoent – porta_or – controla o incremento no expoente, caso tenha ocorrido deslocamentos devido aos arredondamentos acima citados ou *overflow* na etapa de multiplicação das mantissas.
- bloco 16: incr_exp – exp_incr_muldi – incremento de expoente - realiza o incremento do expoente dependendo do sinal vindo do bloco 15.
- bloco 17: over_under_exp – over_under_exp – verifica over/underflow - Após o incremento do expoente verifica se não ocorreu um valor maior que o máximo permitido para o formato, ou se ocorreu um underflow (para a divisão), ou seja, um valor abaixo do mínimo do formato.
- bloco 18: muldi_calc_exc – calc_exc_muldi – lógica de cálculo das exceções – Os casos de under e overflow são reconhecidos e sinalizados para a entrada do seletor de resultado.
- bloco 19: mux_exp_end_muldi – multiplex_exp_end – seletor de resultado do expoente – seleciona o resultado a ser enviado à saída. Para ocorrência de resultados dentro do formato o resultado é buscado do bloco que realiza incremento de expoente, para a exceção, o resultado é buscado da lógica de cálculo das exceções.
- bloco 20: mux_man_end_muldi – multiplex_man_end – seletor de resultado da mantissa – na situação de resultados dentro do formato, a mantissa selecionada é a que vem do bloco 14, caso contrário vem do bloco 18.
- bloco 21: xor_signals – xor_two – porta xor – lógica necessária para definição do sinal do resultado.
- bloco 22: subt_excess_127_exp – subt_127_excess_8 – subtrator do excesso de127 do expoente resultante final.

IV.4 – Unidade de Controle Microprogramada

A figura 4.3 correspondente à figura 3.1 do capítulo III e ilustra, na área tracejada, a unidade de controle microprogramada.

Nome do programa de mapeamento geral do MPH: three_blocks4_portmap. Programas formadores:

- bloco 1: clock_gen – clock_generator – não explícito na figura– definição da base de tempo para o processamento
- bloco 2: mux_address_uma – mux_mpc – Mmux – define, de acordo com os controles presente na micropalavra, qual o endereço da próxima microinstrução. As opções são: busca de instrução (vindo do decodificador de opcode), endereço presente

no campo indicado na micropalavra por prox.endereço, condição imposta por resultados indesejáveis na Unidade Aritmética ou endereço dado pelo incrementador –increm.

- bloco 3: register_addr_micrinstr – reg_micrpc – reg_de end. de microinstrução – é um registrador temporário que exerce o interfaceamento entre o multiplexador Mmux e a UMA.

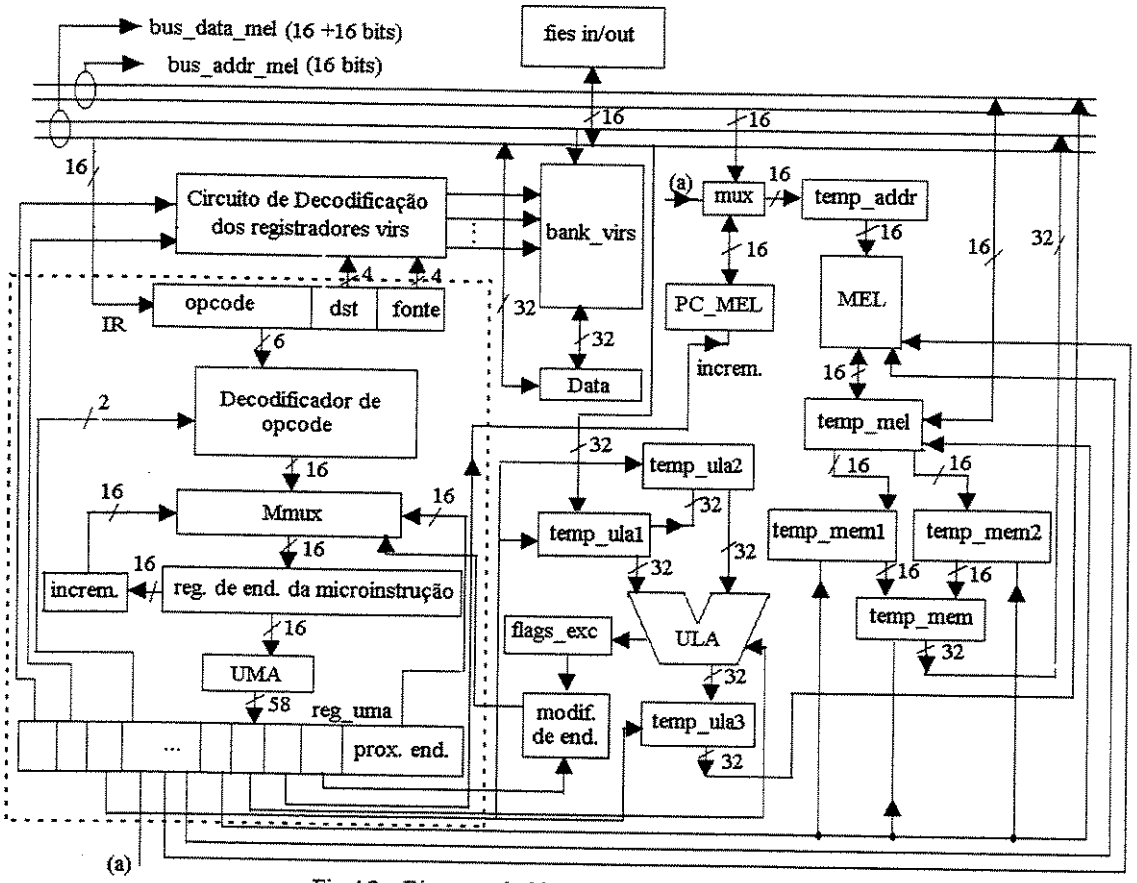


Fig. 4.3 - Diagrama de blocos do microprocessador MPH

- bloco 4: uma – uma_model – UMA – memória de alocação das palavras de controle de todas as instruções previstas para o microprocessador.
- bloco 5: microinstr_reg – reg_latch_mir – registro temporário da palavra de microinstrução em curso utilizada nos circuitos de validação.
- bloco 6: buffer_inout – buffer_input_output – buffer in/out – realiza o interfaceamento do MPH com o HOST. É o bloco de entrada e saída dos dados no MPH.
- bloco 7: mux_mel – mux_mel_address – mux – realiza a seleção entre o endereço dado pelo barramento bus_addr_mel e o PC_MEL, ou seja, o contador de programa da MEL. Seu controle é realizado pela palavra de microinstrução.

- bloco 8: pc_reg – pc_mel_reg – PC_MEL – registro contador do programa presente na MEL. Um controle previsto pela palavra de microinstrução habilita o incremento quando a seqüência do programa assim o exigir.
- bloco 9: temp_addr_register – reg_temp_addr – temp_addr – registro temporário do endereço definido pelo mux.
- bloco 10: conv_bv_to_integer – conv_bv_to_int – não consta da figura – utilizado para facilitar o acesso a endereços da MEL
- bloco 11: temp_registers1_mel – temporary_data_mel –temp_mel – realiza o armazenamento temporário dos dados lidos da memória.
- bloco 12: temp_registers2_mel – register_temp_mem – compreende os registradores temp_mem1, temp_mem2 e temp_mem, que são registradores temporários utilizados à saída da memória MEL. Permitem a transformação de dados da memória de 16 bits para 32 por concatenação do conteúdo de dois registradores de 16 bits intermediários.
- bloco 13: mel – mel_memory – MEL – memória de escrita e leitura, onde os programas de execução dos modelos e resultados obtidos devem ser armazenados.
- bloco 14: ir_register – reg_IR – IR – realiza o armazenamento temporário da instrução, sujeita à decodificação no Decodificador de opcode.
- bloco 15: analysis_of_opcode – opcode_analyze – Decodificador de opcode – realiza a decodificação do campo relativo ao código de instrução e informa ao Mmux o endereço selecionado. É controlado pela palavra de microinstrução que define seu acionamento na etapa de procura da instrução.
- bloco 16: dec_virs – regs_transfer –circuito de decodificação dos registradores vir, bank_virs e data – realiza a decodificação da região onde estão codificados os registros de destino e fonte do dado. Data é um registrador temporário que recebe o dado retirado ou a ser armazenado no banco de registradores – bank_virs.
- bloco 17: temps_ula – registers_temp_ula – temp_ula1, temp_ula2, temp_ula3 – registradores de uso para inserção ou captação de resultados da Unidade Aritmética.
- bloco 18: dec_micrinstr – decode_micr_instr1 – decodificador de microinstrução – não explícito na figura por se tratar de um bloco utilizado para a organização da palavra de microinstrução.
- bloco 19: ula – ULA – realiza as operações matemáticas em ponto flutuante.

IV.5 – Tabelas Trigonométricas

Nome do programa na forma de *entity*: calc_sen_cos

Programas formadores:

- *pack_tables_addr* – produz a correspondência dos ângulos aos endereços nos *packages* abaixo relacionados
- *pack_lokup_table_sen* – tabela referente aos valores de seno
- *pack_lokup_table_cos* – idem para os cossenos

Para ilustrar o formato em que devem ser construídas estas tabelas ou *lookup tables* apresentamos, a seguir, a tabela referente à determinação do seno, conforme procedimento exposto no ítem III.10.4.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
Package pack_lookup_table_sen IS
Subtype mem_trigon_real IS real range 0.000 to 1.574; -- valor correspondente a  /2
type t_data_sen IS ARRAY(0 TO 255) of mem_trigon_real;
Constant sen_value: t_data_sen :=
(0.003, 0.009, 0.015, 0.021, 0.027, 0.033, 0.039, 0.045,
0.051, 0.057, 0.063, 0.069, 0.075, 0.081, 0.087, 0.093,
0.099, 0.105, 0.111, 0.116, 0.122, 0.128, 0.134, 0.140,
0.147, 0.153, 0.159, 0.165, 0.171, 0.177, 0.183, 0.189,
0.195, 0.199, 0.205, 0.211, 0.217, 0.223, 0.229, 0.235,
0.240, 0.246, 0.252, 0.257, 0.263, 0.269, 0.274, 0.280,
0.286, 0.291, 0.297, 0.303, 0.309, 0.313, 0.319, 0.326,
0.330, 0.336, 0.342, 0.347, 0.353, 0.358, 0.364, 0.369,
0.375, 0.381, 0.386, 0.392, 0.397, 0.403, 0.408, 0.413,
0.419, 0.425, 0.430, 0.435, 0.440, 0.446, 0.452, 0.457,
0.463, 0.467, 0.473, 0.478, 0.483, 0.489, 0.495, 0.500,
0.505, 0.510, 0.515, 0.520, 0.525, 0.530, 0.535, 0.540,
0.545, 0.550, 0.555, 0.560, 0.565, 0.570, 0.575, 0.578,
0.582, 0.587, 0.592, 0.596, 0.600, 0.605, 0.610, 0.615,
0.623, 0.630, 0.634, 0.639, 0.643, 0.647, 0.652, 0.657,
0.661, 0.665, 0.670, 0.673, 0.679, 0.684, 0.688, 0.692,
0.697, 0.701, 0.705, 0.709, 0.713, 0.718, 0.722, 0.726,
0.730, 0.734, 0.738, 0.742, 0.746, 0.750, 0.754, 0.758,
0.762, 0.766, 0.770, 0.774, 0.778, 0.781, 0.785, 0.788,
0.792, 0.796, 0.800, 0.803, 0.806, 0.810, 0.814, 0.818,
0.821, 0.825, 0.827, 0.830, 0.834, 0.838, 0.840, 0.843,
0.847, 0.851, 0.853, 0.855, 0.858, 0.862, 0.865, 0.868,
0.871, 0.875, 0.877, 0.880, 0.882, 0.885, 0.888, 0.891,
0.894, 0.897, 0.899, 0.901, 0.904, 0.907, 0.909, 0.912,
0.915, 0.917, 0.919, 0.921, 0.924, 0.926, 0.929, 0.931,
0.933, 0.935, 0.937, 0.939, 0.941, 0.943, 0.945, 0.947,
```

```
0.949, 0.951, 0.952, 0.954, 0.956, 0.958, 0.960, 0.961,
0.963, 0.965, 0.966, 0.968, 0.969, 0.970, 0.972, 0.974,
0.975, 0.976, 0.978, 0.979, 0.980, 0.981, 0.982, 0.983,
0.984, 0.985, 0.986, 0.987, 0.988, 0.989, 0.990, 0.991,
0.992, 0.993, 0.994, 0.995, 0.996, 0.996, 0.997, 0.997,
0.998, 0.998, 0.998, 0.999, 0.999, 0.999, 0.999, 1.000);

Function bring_mem_value (addr_dado: In Integer) RETURN mem_trigon_real;
End  pack_lookup_table_sen;

PACKAGE BODY pack_lookup_table_sen IS
Function bring_mem_value (addr_dado: In Integer) RETURN mem_trigon_real  IS
Variable var_out: mem_trigon_real;
BEGIN
var_out := sen_value(addr_dado);
RETURN var_out;
End bring_mem_value;
End pack_lookup_table_sen;
```

IV.6 – Comentários

Foram apresentados os mapeamentos e simulações resultantes dos principais blocos formadores da arquitetura do MPH. Os diagramas de tempo dos resultados das principais unidades da arquitetura serão apresentados no capítulo V e visam comprovar a obtenção de resultados compatíveis com os valores esperados.

Capítulo V

Análise dos resultados das simulações, conclusões e sugestões para trabalhos posteriores

V.1 – Introdução

Ambientes de projeto disponibilizados pela Mentor Graphics, para a FEEC da Unicamp, que utilizam a linguagem VHDL para descrever arquiteturas de circuitos em diferentes graus de abstração, possibilitaram que este trabalho pudesse ser realizado fazendo uso de técnicas avançadas de projeto de circuitos integrados.

Os diagramas de tempo que figuram neste capítulo são resultados obtidos com o programa *quicksim* aplicado a compilações dos programas descritos na linguagem VHDL.

Os dados e resultados aqui apresentados se encontram representados em hexadecimal, devido à facilidade desta representação para valores com um grande número de *bits*. Vamos analisar os resultados mais relevantes para o funcionamento da arquitetura e fazer a comparação de desempenho com produtos já disponíveis no mercado.

V.2 – Resultados da Unidade Aritmética e Lógica em ponto flutuante

Este bloco dispendeu grande parte do trabalho de desenvolvimento do projeto devido à sua complexidade. É grande o número de circuitos envolvidos na elaboração desta unidade por isso, vamos enfocar os diagramas resultantes da simulação dos principais blocos, ou seja, dos operadores propriamente ditos ou, como podem ser considerados, operadores em ponto fixo e em seguida dos circuitos resultantes para ponto flutuante.

V.2.1 – Somador/ subtrator em ponto fixo

O somador em ponto fixo, referente ao bloco 13 descrito no ítem IV.2, foi projetado pela técnica de *carry lookahead*. Este se constitui em um dos principais blocos para o projeto da Unidade Aritmética e Lógica porque é o fundamento para as operações básicas abordadas neste trabalho.

Para a análise desta simulação isoladamente, foi adotada uma base de tempo *clk_ula4*, com período de 50ns e *duty cycle*, ou seja, simetria do ciclo de 50%, caracterizando uma freqüência de operação de 20 MHz.

Os demais sinais utilizados nesta simulação foram:

- *enable_add_sub* - entrada da habilitação do bloco;
- *operation* – entrada de definição da operação a ser realizada, sendo 0 para a soma e 1 para a subtração;
- *input_man_a* e *input_man_b* - entradas das mantissas dos operandos representadas por 24 *bits* (*bit* escondido incluso);
- *output_result_man* – saída do resultado da operação;
- *flag_over* – saída de *overflow* de resultados;
- *flag_under* – idem para *underflow*.
- *renorm_acknow* – saída de controle responsável pela habilitação do bloco imediatamente posterior. Atua em conformidade com o *enable_add_sub*, para garantir a continuidade de tarefas subseqüentes quando o bloco estiver operando em ponto flutuante.

O tempo de execução da operação de soma que pode ser observado através da análise do gráfico, deve ser interpretado pelo número de ciclos de relógio necessários para a conclusão da operação porque em termos de valores numéricos de tempo de processamento, está vinculado ao valor adotado para o ciclo do relógio. Esta constatação foi confirmada por simulações realizadas com tempos de *clock* de 1ns (unitário).

Para a análise dos tempos de processamento faremos portanto sempre a referência ao número de ciclos.

Cabe a ressalva que o tempo para obtenção de respostas aritméticas é dependente não só do algorítmo adotado mas também é definido pela tecnologia de fabricação utilizada na implementação do circuito.

Para esta operação estão retratados na figura 5.1 alguns valores utilizados arbitrariamente para os operandos, de forma a se observar o comportamento geral do circuito.

Dos resultados obtidos podemos verificar que a soma exige cerca de meio ciclo enquanto que a subtração se dá no intervalo de tempo correspondente a um ciclo e meio.

V.2.2 – Somador/ subtrator em ponto flutuante

Este bloco foi gerado a partir dos sub-blocos descritos no ítem IV.2 necessários para a formação da arquitetura para este operador, conforme o mapeamento apresentado.

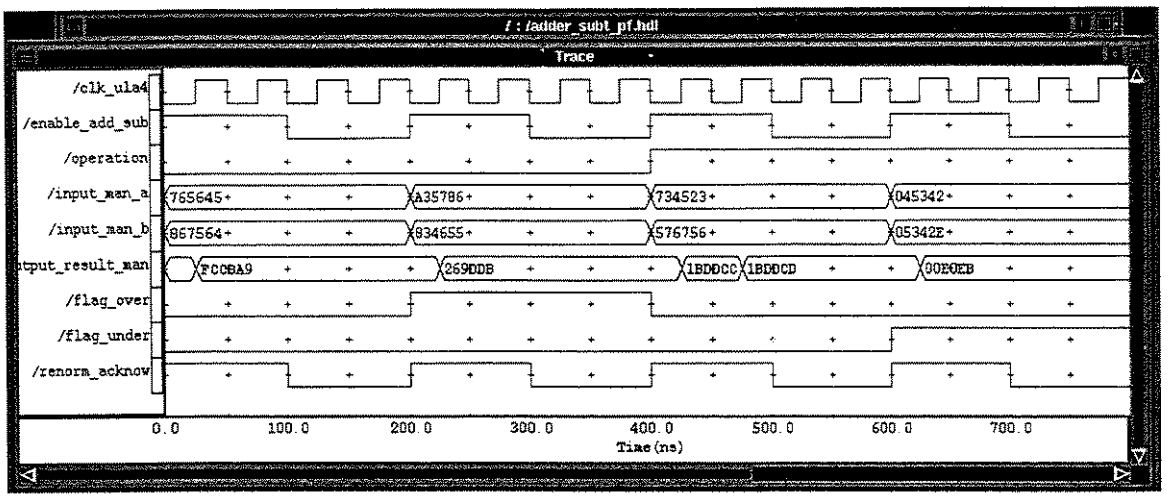


Fig. 5.1 – Diagrama de simulação da soma/subtração em ponto fixo

A descrição do circuito em linguagem VHDL, nesta forma é denominada *port map* e permite o desenvolvimento de circuitos de grande porte a partir de estruturas menores com maior facilidade de caracterização. Utilizando este recurso sub-blocos podem ser inicialmente testados separadamente para depois serem acoplados aos demais necessários para a formação de estruturas maiores.

Os sinais adotados na simulação, ilustrados na figura 5.2, apresentam a seguinte nomenclatura correspondente:

- *clk_ula_in* – entrada do sinal de relógio para sincronismo, aqui com duração de ciclos de 50ns e *duty cicle* de 50%
- *start_add_sub* – entrada de habilitação de acionamento do bloco
- *operation* – entrada de seleção para a operação a ser executada, 1 para soma e 2 para subtração
- *sig_a* e *sig_b* – sinais de entrada referentes ao *bit* de sinal dos dois operandos envolvidos na operação
- *input_mant_a* e *input_mant_b* – entrada das mantissas normalizadas
- *input_expoent_a* e *input_expoent_b* – entrada dos expoentes
- *sig_result* – saída com o sinal resultante
- *exp_result* – ídem para o expoente
- *mantissa_result* – ídem para a mantissa
- *flags_byte_out* – saída da palavra resultante, detetora de exceções, *over* e *underflow*
- *interrupt_bit* – sinalização de interrupção do processamento
- *enable_reg_exp_out* e *enable_reg_man_out* – habilitação do multiplexador de saída de resultados.

Para a apresentação da simulação do comportamento deste circuito foram utilizados operandos com valores que permitiram a verificar resultados em situações

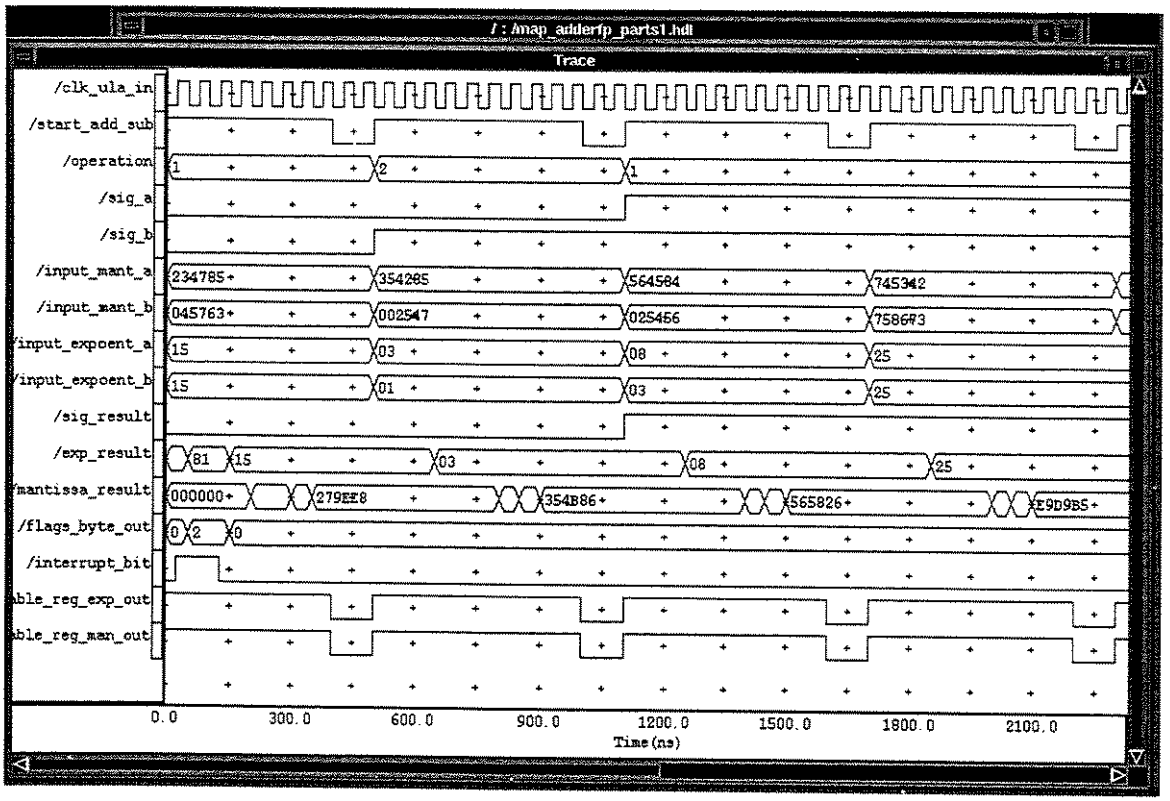


Fig. 5.2 – Diagrama de simulação da soma/subtração em ponto flutuante

variadas tais como: expoentes iguais e mantissas diferentes, onde a operação entre mantissas fosse evidenciada; expoentes diferentes e sujeitos a deslocamentos para o estabelecimento de um valor comum entre eles; o efeito dos sinais dos operandos, na determinação da operação efetiva.

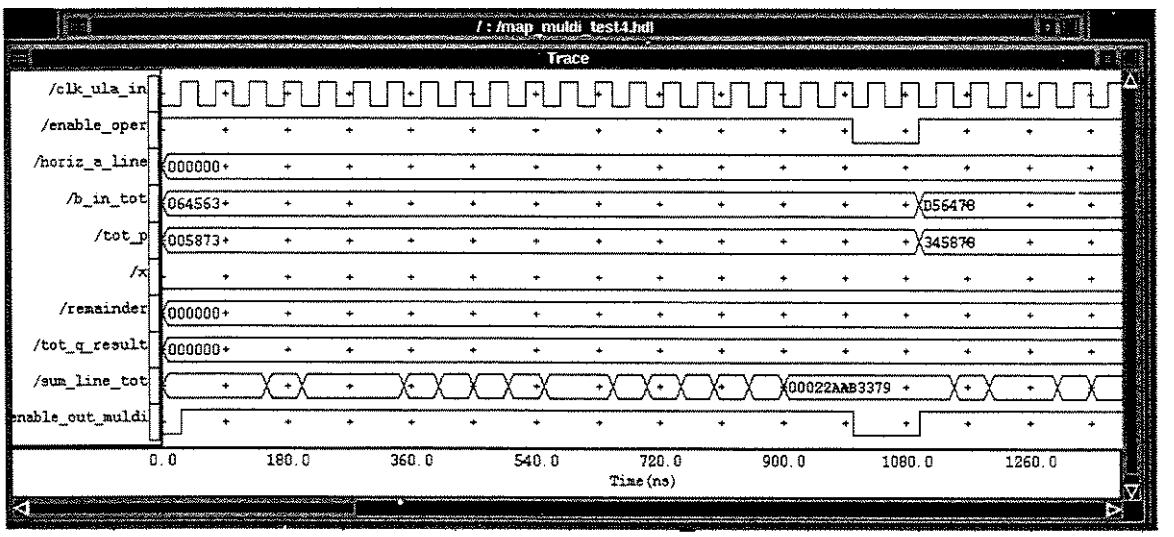
O diagrama de simulação mostra que os resultados foram obtidos após 8 ciclos de *clock* para cada situação.

V.2.3 – Multiplicador/divisor em ponto fixo

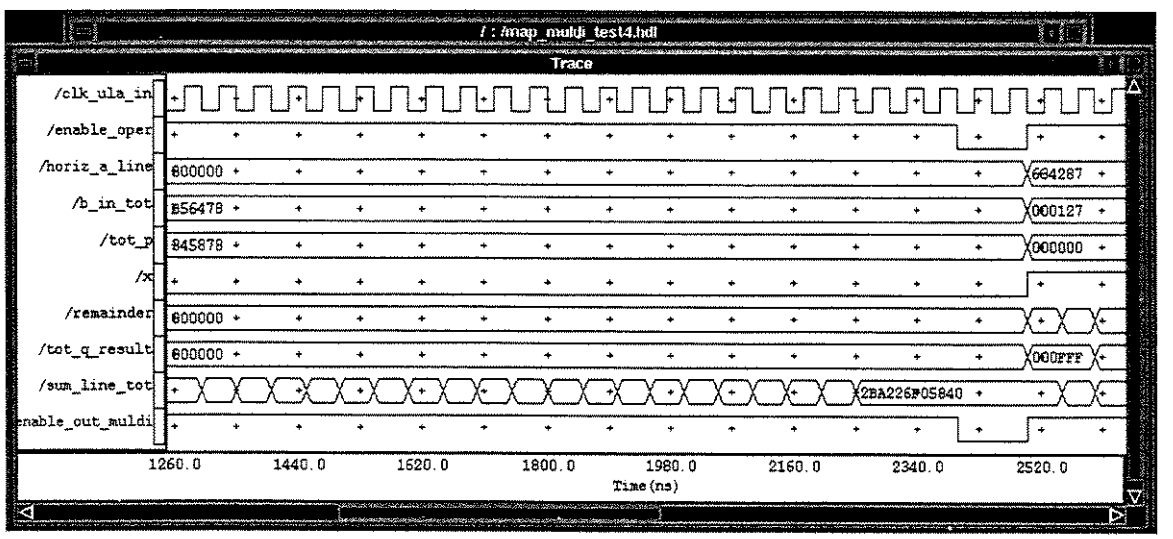
Este bloco apresentou grande dificuldade de projeto, dada a extensão da cadeia formada de 24 linhas e 24 colunas dispostas diagonalmente, exigindo uma especial atenção na alocação de índices nas associações das linhas, para o mapeamento interno.

As figuras 5.3(a) e (b) mostram os resultados obtidos nas simulações em dois diagramas visando uma melhor apreciação dos resultados, uma vez que a condensação da figura para dimensões menores não possibilitaria a visualização numérica. Estão

mostradas duas operações de multiplicação com valores envolvendo um grande número de *bits*.



(a)



(b)

Fig. 5.3 (a),(b) – Resultados obtidos para a multiplicação de b por p quando x=0

Os sinais utilizados para a descrição do bloco multiplicador/divisor foram:

- *clk_ula_in* – entrada do sinal para base de tempo
- *enable_oper* – entrada da habilitação do bloco
- *horiz_a_line* – entrada do operando a – só utilizada para o divisor
- *b_in_tot* – entrada do operando b – multiplicando ou divisor

- *tot_p* – entrada do multiplicador – só utilizado na multiplicação
- *x* – entrada para a definição da operação: 0 para a multiplicação e 1 para a divisão
- *remainder* – saída do resto da divisão
- *tot_q_result* – saída do resultado da divisão
- *sum_line_tot* – saída do resultado da multiplicação em 48 *bits*
- *enable_out_muldi* – saída de habilitação do próximo bloco imediatamente conectado

O diagrama dado na fig. 5.4, apresenta o resultado da divisão, onde se pode notar

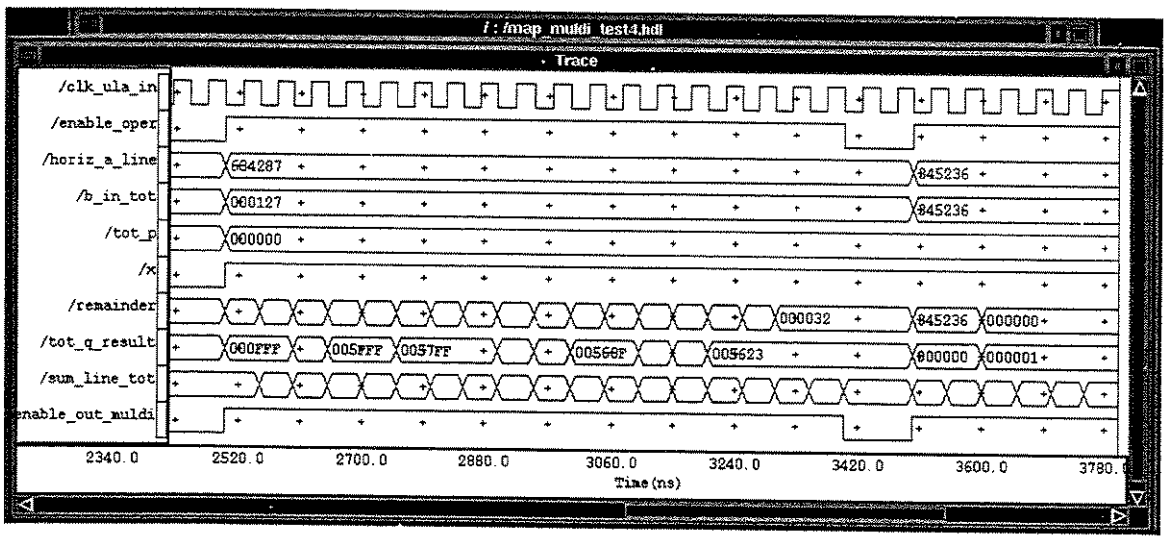


Fig.5.4 – Resultado da divisão de a por b em ponto fixo quando (x= 1)

a dependência que o número de ciclos necessários para o processamento tem com o tamanho dos operandos.

Para o circuito multiplicador tivemos no primeiro exemplo o transcurso de 18 ciclos e 25 no segundo. Para o divisor os resultados obtidos são de aproximadamente 18 ciclos e 3,5 ciclos respectivamente para o primeiro e segundo casos.

V.2.4 – Multiplicador/divisor em ponto flutuante

O circuito que realiza multiplicação/divisão em ponto flutuante tem uma significativa importância no desempenho de processamento da Unidade Lógica e Aritmética. A multiplicação e divisão por se tratarem de operações com maior duração dentro dos processamentos previstos para a máquina, constituem-se em "gargalos" no desempenho da unidade, ou seja, o fator limitante de velocidade do processamento.

Estes operadores estão sujeitos a arredondamentos ou aproximações de resultados parciais que ocorrem no decorrer da operação e podem vir a se constituir em fontes de erro. A figura 5.5 esboça o comportamento destes operadores com os sinais denominados segundo o glossário apresentado a seguir.

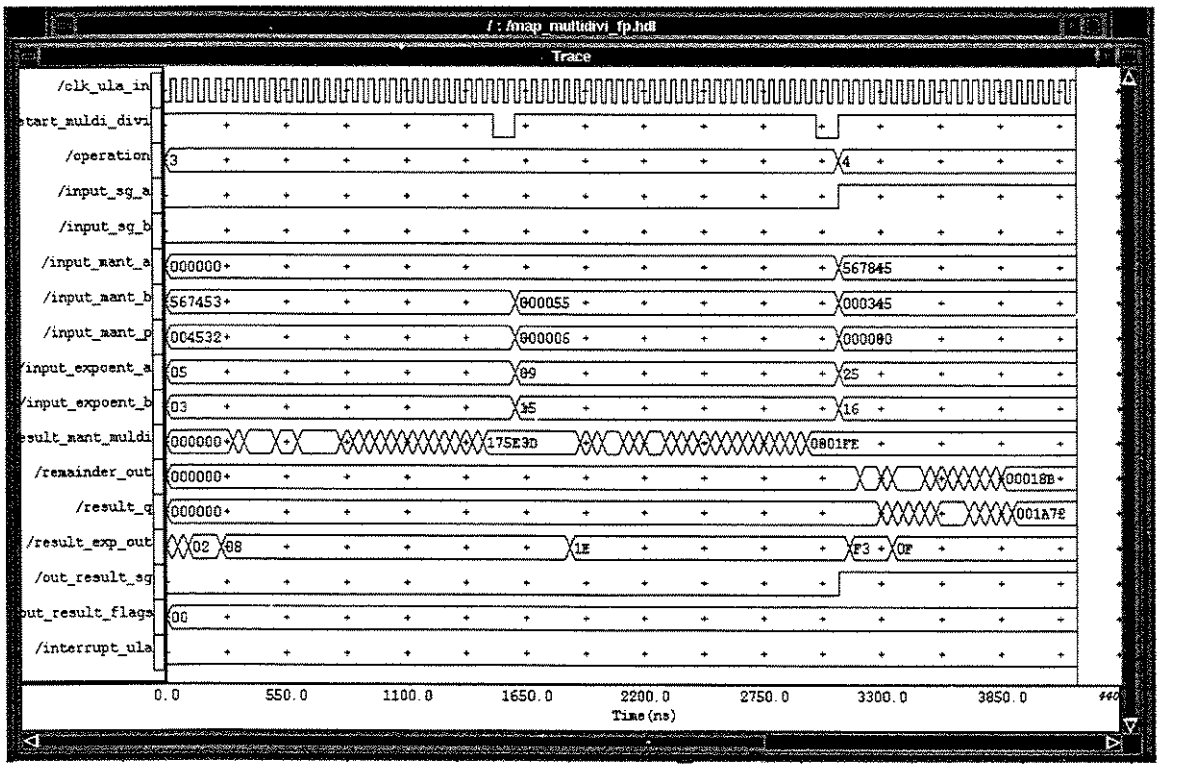


Fig.5. 5 – Resultados da multiplicação de b por p, para *operation*= 3 ou da divisão de a por b para *operation* = 4

- *clk_ula_in*: entrada da base de tempo
- *start_muldi_divi* – entrada para habilitação de início da operação
- *operation* – entrada para definição da operação: 3 para a multiplicação e 4 para a divisão
- *input_sig_a* e *input_sig_b* – entrada do sinal dos operandos a e b
- *input_mant_a* – entrada do sinal da mantissa a – dividendo
- *input_mant_b* – entrada do sinal da mantissa b – multiplicador ou dividendo
- *input_mant_p* – entrada do sinal da mantissa do multplicando
- *input_expoent_a* e *input_expoent_b* – entrada dos expoentes a e b
- *out_result_mant_muldi* – saída do resultado da multiplicação
- *remainder_out* – saída corespondente ao resto da divisão
- *result_q* – saída do resultado da multiplicação

- *result_exp_out* – saída do expoente resultante
- *out_result_flags* – *flags* resultantes de resultados indesejados
- *out_result_sg* – saída do sinal da operação resultante
- *interrupt_ula* – saída para condição de interrupção

Segundo os resultados obtidos 30 ciclos de *clock* são necessários para multiplicação em ponto flutuante no nosso projeto. Como podemos constatar a multiplicação em ponto flutuante apresenta um significativo acréscimo no seu tempo de execução quando comparada à em ponto fixo. Podemos atribuir este acréscimo às várias etapas do procedimento para ponto flutuante também como para a redução dos bits de resultado. A divisão apresentou, em média, 18 ciclos para conclusão da operação, semelhante ao obtido para ponto fixo.

V.3 – Resultados das funções trigonométricas

Conforme comentário realizado no capítulo III, funções trigonométricas são imediatamente calculadas a partir de memórias previamente armazenadas. Esta técnica reduz o tempo de obtenção de resultados, mas pode incorrer em imprecisão porque, dependendo do tamanho da memória o número de posições da memória restringe o número de resultados previstos dentro do intervalo adotado. Um outro aspecto a ser considerado é a não linearidade de valores do seno e cosseno, devendo ser levado em consideração na hora de definição das tabelas. Há ainda que se considerar truncamentos e aproximações ocorridos nos cálculos dos valores a serem adotados para as funções. O estudo do nível da precisão perdida nestas situações é muito elaborado e foge ao escopo deste trabalho.

As *lookup tables* se mostram atraentes pelo seu curto tempo de processamento, sendo dependentes apenas do acesso e leitura a dados de uma memória. Portanto, é considerada uma forma prática e pode ser aplicada com segurança onde sejam definidos critérios para minimizar os efeitos negativos acima expostos. O diagrama da figura 5.6 apresenta a obtenção de seno e cosseno obtidos por simulação utilizando as tabelas já descritas. Os sinais utilizados na programação são os seguintes:

- *start_trig*: entrada para habilitação do bloco de cálculo trigonométrico
- *clk_in*: entrada para base de tempo
- *angle_data* : entrada do ângulo definido a ser convertido ao primeiro quadrante para a análise de seus valores pelas *lookup tables*

- *sig_sen_out* e *sig_cos_out*: saída do sinal do resultado
- *result_cos* e *result_sen*: resultado numérico do seno e cosseno

O diagrama ilustra que apenas 1 ciclo de relógio é necessário para obtenção dos resultados destas funções comprovando as considerações anteriormente abordadas.

V.4 – Resultados da Unidade de Controle

A Unidade de Controle é o núcleo de coordenação de todo o processamento e

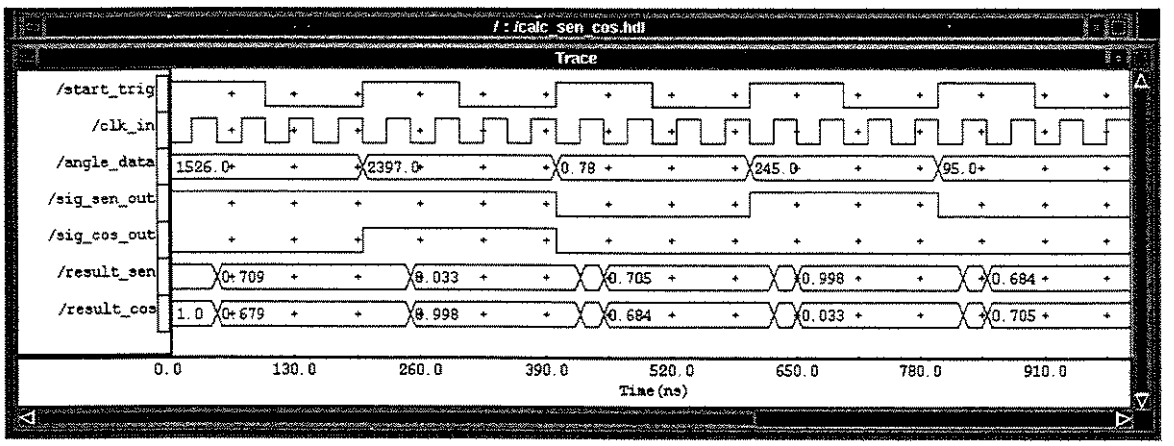


Fig.5. 6 – Resultados das funções seno e cosseno

tem o seu desempenho diretamente relacionado à base de tempo dentro da qual a seqüência das microinstruções são realizadas. Estes sinais são importantes para a definição do sincronismo na realização de tarefas pela máquina porque através do acionamento dos sub-blocos opera a funcionalidade da arquitetura. Dada a importância desta informação, apresentamos na figura 5.7 o diagrama dos sinais utilizados para a

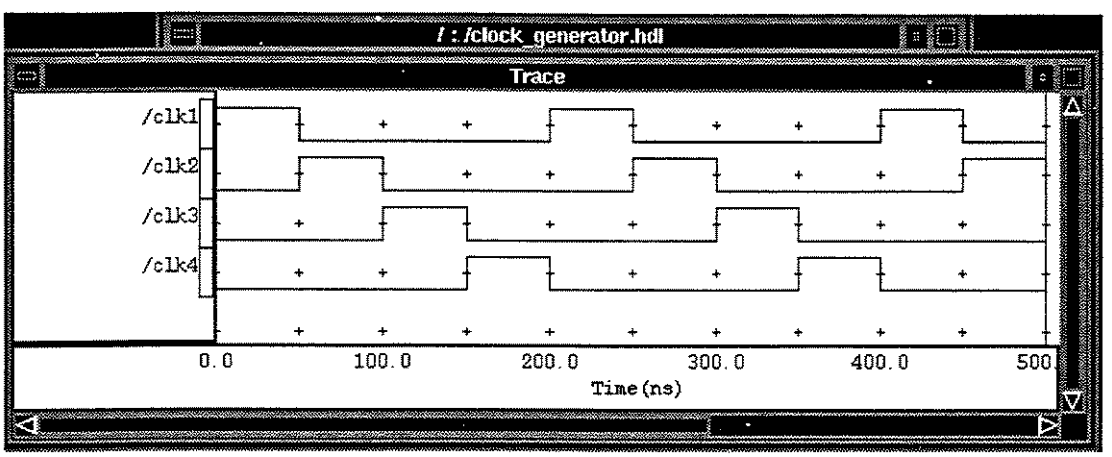


Fig.5.7 – Quatro saídas do sinal de *clock* adotado para a simulação da Unidade de Controle

temporização na simulação das instruções da máquina.

Nas descrições foi adotado um *clock* com quatro saídas seguindo informações obtidas da literatura [12] para arquiteturas semelhantes à nossa.

A tarefa de obtenção de temporização para os blocos pertencentes à arquitetura não é de fácil aquisição devido à coordenação de seqüência das microoperações das várias etapas que resultam na realização da instrução dentro do prazo previsto. As respostas individuais dos sub-blocos devem ser sincronizadas de forma a não gerar situações de inconsistência temporal na seqüência de execução. A coordenação destes sinais dentro da arquitetura, através da definição da seqüência de microinstruções correlacionadas aos pulsos de *clock*, ocupou boa parte do tempo de desenvolvimento do projeto.

V.4.1 – Processamento de Instruções

A seqüência de microinstruções necessárias para a resolução de modelos será aqui tratada utilizando algumas das principais instruções para a resolução de qualquer modelo.

Escolhemos a instrução LDI32 dado, representada na figura 5.8 por ser de ampla utilização na descrição dos programas para quaisquer componentes. A etapa de busca da instrução ou *fetch* ocorre no intervalo definido pelos sinais onde address3_uma assume o valor 1 até o instante onde ir_inter recebe o valor 1400 que é o código da citada instrução. Este intervalo de tempo apresenta 2 ciclos de *clock*. A execução da instrução tem a duração de 6 ciclos.

Os sinais envolvidos neste processamento estão descritos a seguir, dentro da seguinte ordem: o bloco a que pertencem e seu efeito dentro da arquitetura. Os blocos correspondem ao ítem IV.4 do capítulo IV. A seqüência para execução desta instrução está descrita no ítem III.8.2.

Nome do programa de mapeamento geral: three_blocks4_portmap – Unidade de Controle

- *start* – blk_7 (*mux_mel*) – define o início do processamento
- *en_mux_uma* – blk_2 (*mux_address_uma*) - habilita a definição de endereço pela UMA
- *clk1 a clk4* – gerados no blk_1 e presentes por toda a arquitetura
- *barr_mel_address* - blk_6 (*buffer_inout*) – barramento de endereços da MEL
- *mel_mux_out* – blk_7(*mux_mel*) - saída do mux ligado ao PC_MEL

- *pc_in* – blk_8 (*pc_reg*) – entrada do registrador PC_MEL
- *pc_out* – blk_8 (*pc_reg*) – saída do registrador PC_MEL
- *temp_mel_sec_out* – blk_11(*temp_registers1_mel*) – saída para o barramento de ligação entre a MEL e *temp_mel*
- *out_temp_mem1* - blk_11(*temp_registers1_mel*) – saída do registrador *temp_mem1* para *temp_mel*

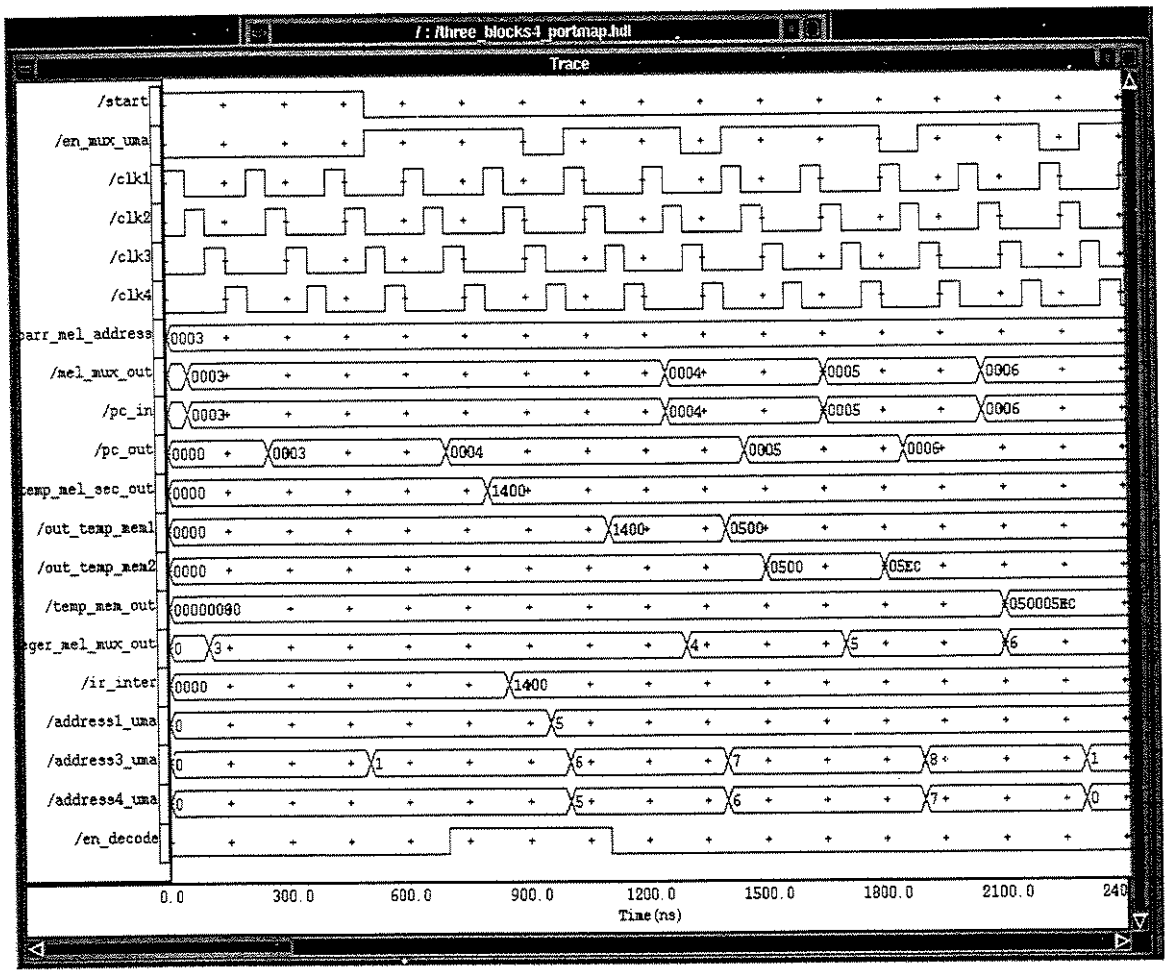


Fig.5.8 – Sinais referentes à execução da instrução LDI32 dado

- *out_temp_mem2* - blk_11(*temp_registers1_mel*) – saída do registrador *temp_mem2* para *temp_mel*
- *temp_mem_out* - blk_12 (*temp_registers2_mel*) – saída para o barramento *bus_data_mel*
- *integer_mel_mux_out* - blk_10 (*conv_bv_to_integer*) – saída do conversor para inteiro de endereços da MEL

- *ir_inter* - blk_14 (*ir_register*) – entrada do registrador IR.
- *address1_uma*- blk_2 (*mux_address_uma*) – entrada do *Mmux*
- *address3_uma* - blk_3 (*register_addr_micrinstr*) – entrada do registro de endereço de microinstrução
- *address4_uma* - blk_3 (*register_addr_micrinstr*) – saída do registro de endereço de microinstrução
- *en_decode* - blk_15 (*analysis_of_opcode*) – habilita a decodificação do código de instrução presente em IR

A visualização da execução de uma operação de divisão em ponto flutuante é dada na figura 5.9, para a operação de divisão em ponto flutuante no modo registrador. Os sinais dos operandos presentes nos registradores da ULA se encontram nos registradores *data_ula1* - dividendo e *data_ula2* – divisor. O resultado é encontrado em *data_ula3*. Um sinal relevante é o *en_mux_uma*, que define a entrada da palavra de controle presente em *ctrl_word* e foi simulado de forma a permitir o ajuste de tempo na execução da instrução.

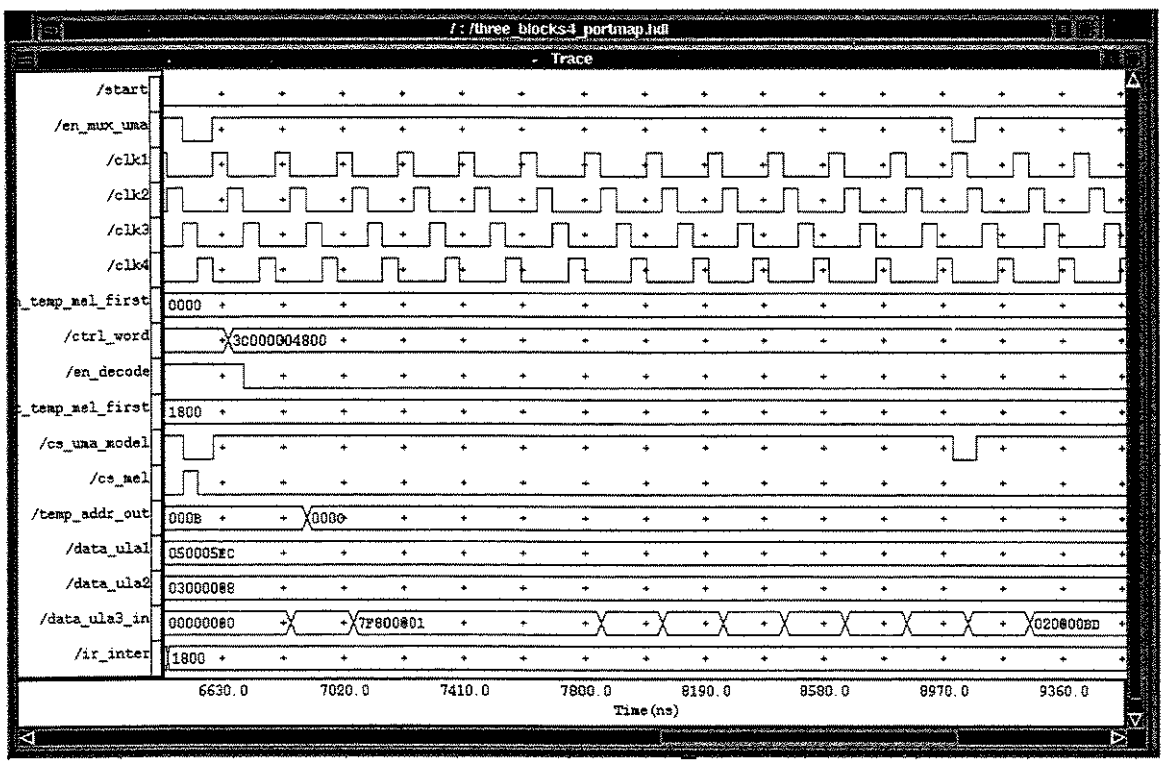


Fig.5.9 – Operação de Divisão de *data_ula1* por *data_ula2*

V.5 – Processadores comercialmente disponíveis

Ao se concluir um projeto é tarefa do projetista avaliar o desempenho da máquina fazendo comparações de seus resultados com os produtos disponíveis no comércio. Para isso buscamos[24] algumas informações referentes a microprocessadores existentes para consumo. Faremos uma breve avaliação do desempenho, a partir da avaliação do comportamento do nosso processador comparando-o a similares existentes.

A tabela 5.1 dá informações sobre dois processadores genéricos e três dedicados, apresentados nesta ordem.

processador aritmético	**fabricante**	**operações básicas disponíveis**
80387	Motorola	adição, subtração, multiplicação, divisão
68882	Motorola	adição, subtração, multiplicação,divisão
WTL 1233	Weitek	adição, subtração, multiplicação
ADSP 3201	Analog Devices	adição, subtração, multiplicação
WTL 2265	Weitek	adição, subtração, multiplicação
ADSP 3222	Analog Devices	adição, subtração, multiplicação, divisão

Tabela 5.1 – Processadores comercialmente disponíveis, fabricantes e operações básicas

Os processadores genéricos, também conhecidos como coprocessadores, se destinam a operar acoplados a microprocessadores principais, sendo responsáveis pela execução da parte matemática mais elaborada. Os coprocessadores aritméticos por serem voltados para propósitos gerais, apresentam unidades operativas genéricas com bibliotecas de operações e funções aritméticas implementadas por *hardware* para a resolução de algorítmos fundamentados em somas e deslocamentos. Este princípio disponibiliza resoluções de operações básicas, trigonométricas e exponenciais de forma simplificada.

Os dedicados apresentam uma arquitetura com circuitos específicos para operação por processamento *pipeline*.

Os tempos de processamentos destes microprocessadores estão apresentados nas tres tabelas (5.2, 5.3 e 5.4) se referem às operações de adição/subtração, multiplicação e divisão respectivamente.

processador aritmético	número de ciclos de latência	freqüência de operação	tempo de latência
80387	24 a 32	20 MHz	1,2 a 1,6 μs
68882	69	25 MHz	1,5 a 1,8 μs
WTL 1233	9	20 MHz	450ns
ADSP 3201	3	25 MHz	120ns
WTL 2265	4	20 MHz	200 ns
ADSP 3222	3	20 MHz	150 ns

Tabela 5.2 – adição e subtração em ponto flutuante

processador aritmético	número de ciclos de latência	freqüência de operação	tempo de latência
80387	27 a 35	20 MHz	1,2 a 1,6 μs
68882	89	25 MHz	3,6 μs
WTL 1232	9	20 MHz	450ns
ADSP 3202	3	20 MHz	120ns
WTL 2264	4	20 MHz	200ns
ADSP 3221	3	20 MHz	150ns

Tabela 5.3 – multiplicação em ponto flutuante

processador aritmético	número de ciclos de latência	freqüência de operação	tempo de latência
80387	89	20 MHz	4,45 μs
68882	121	25 MHz	4,84 μs
ADSP3222	18	20 MHz	850ns
ADSP 3202	16	25 MHz	720ns
WTL 2264	13	20 MHz	650ns
ADSP 3221	8	20 MHz	400ns

Tabela 5.4 – divisão em ponto flutuante

Os parâmetros abordados são: o tempo de latência que é o tempo necessário para realizar a operação, o número de ciclos envolvidos na realização desta tarefa e a freqüência de operação, ou seja, a base de tempo que serve de referência para o processamento da máquina.

V.6 - Análise dos resultados aritméticos obtidos para as quatro operações fundamentais

No nosso projeto, conforme apresentado no ítem V.2.2., as operações de soma e subtração em ponto flutuante foram obtidas em 8 ciclos de *clock*. As simulações foram realizadas com base de tempo de 50ns, o que corresponde a uma freqüência de 20 MHz.

Observando a tabela 1, vemos que processadores seqüênciais tais como o 80387 e o 68882 realizam estas operações em tempos que variam entre 1,2μs 1,8 μs.

Para a multiplicação apresentada no ítem V.2.4, obtivemos 30 ciclos de *clock* o que corresponde a 1,5μs, muito compatível com os valores apresentados na tabela 2 que variam de 1,5 a 3,6 μs, para circuitos, sem o recurso de *pipelines*. Quanto à divisão os nossos resultados são obtidos em 18 ciclos, correspondentes a 3,5 μs enquanto que os comerciais são da ordem de 4,5 μs.

V.7 – Comentários sobre o desempenho do MPH

A análise do desempenho de um microprocessador é realizada a partir do conhecimento do número de ciclos envolvidos na execução de cada uma das instruções, a porcentagem de instruções com um mesmo número de ciclos e o *clock* definido para a arquitetura. Este último fator está intimamente relacionado com o processo utilizado na implementação do componente. Desta forma uma visão realística do comportamento do MPH só poderá ocorrer, se a tecnologia de sua fabricação estiver definida. Achamos portanto, que o trabalho dispendido para avaliação do desempenho da arquitetura total deva ser realizado a posteriori para que não se chegue a conclusões distorcidas da realidade.

Faremos contudo, algumas observações com relação as algumas instruções que nos permitam prever o desempenho do MPH. A execução da instrução LDI32 dado representa a classe de instruções com maior número de passos porque ocupa três palavras de microinstrução. A instrução foi concluída em um intervalo de tempo de 6 ciclos. A literatura técnica [33] apresenta valores na ordem de 10 ciclos para computadores sequenciais e de 5 ciclos para processamentos paralelos. Podemos com isto concluir que o desempenho do nosso projeto é muito satisfatório. A parte relativa às

instruções matemáticas, foram extensamente tratadas e mostraram também resultados muito animadores levando-nos a concluir que o projeto desenvolvido apresenta um potencial técnico que permite o seu empregado no ABACUS e em outras aplicações similares.

V.8 - Sugestões para Trabalhos Posteriores

Diante dos resultados obtidos neste trabalho foi constatada a viabilidade da proposta e considerando-se o contexto de aplicações possíveis para o microprocessador desenvolvido, é importante que se dê prosseguimento às pesquisas aqui realizadas. Para dar continuidade ao projeto do sistema ABACUS, sugerimos que em uma próxima etapa seja desenvolvida a estrutura prevista para o *HOST* para que se analise a operacionalidade do sistema como um todo.

Com relação às pesquisas sobre a arquitetura do MPH alguns refinamentos devem ser buscados na sua estrutura, visando uma otimização de seu desempenho. Um exemplo das melhoras que podem ser pesquisadas, diz respeito à alteração da forma de processamento da Unidade Aritmética e Lógica que pode vir a operar por processamento *pipeline* [35],[36], buscando com isso, significativo aumento na velocidade de processamento.

A técnica *pipeline* poderá ser futuramente aplicada na execução das instruções de resolução dos modelos para que algumas instruções possam ser realizadas concorrentemente obtendo-se assim importantes acréscimos no desempenho do microprocessador, uma vez que os recursos disponíveis em processamentos microprogramados permitem ainda associar etapas de execução entre instruções desvinculadas de forma a se realizar o processamento em intervalos de tempo reduzidos.

V.9 – Conclusão

Quando a proposta do desenvolvimento deste projeto foi lançada pretendia-se realizar um trabalho abrangente, não só direcionado a proposta do sistema *ABACUS,* mas que também tivesse um largo espectro de aplicações práticas.

O desenvolvimento de técnicas de projeto de circuitos dentro da microeletrônica sempre deve buscar versatilidade para que se possibilite, na medida do possível, a obtenção de resultados para um amplo espectro de aplicações.

Pretendia-se que a pesquisa viesse ao encontro as tendências de projeto para o mercado de microprocessadores e circuitos controlados por microprogramas que tem

sido empregados em inúmeras pesquisas de sistemas dedicados tais como: satélite para previsão do tempo, robótica, redes neurais, processamento digital de imagens[37],[38], *hardware* evolutivo [39]-[41], etc. A sistemática envolvida no projeto também se mostrou atraente porque apresenta larga aplicação no campo dos microprocessadores reconfiguráveis que estão em evidência nas pesquisas em microeletrônica. Estudos nesta área buscam disponibilizar o desenvolvimento de diversos projetos a partir de módulos básicos previamente consolidados.

A proposta deste projeto estava de acordo com todos estas pesquisas emergentes, prevendo o uso de controles microprogramados associado a pesquisa de operadores matemáticos realizáveis por *hardware* e em ponto flutuante. Um fator que estimulou o desenvolvimento do MPH foi a visão do seu emprego em situações onde os modelos ora dedicados a simulação de simples componentes eletrônicos poderiam vir descrever circuitos completos de grandes sistemas além das aplicações em arranjos de microprocessadores dedicados como o inicialmente proposto.

Embora o projeto tenha apresentado acentuado grau de dificuldade nas suas elaborações, as tendencias tecnológicas foram o parâmetro decisivo e encorajador desde o início.

Os primeiros passos para o desenvolvimento do projeto constou do estudo de arquiteturas de operadores matemáticos. Paralelamente, foi buscado o domínio da linguagem VHDL na descrição destes circuitos.

Os primeiros blocos desenvolvidos foram os referentes à Unidade Aritmética e Lógica e mostraram grandes dificuldades técnicas não só na escolha dos circuitos adequados, como nas descrições em linguagem VHDL, uma vez que o projeto previa um processamento estritamente por *hardware*.

Devido a complexidade do projeto, foi adotada a estratégia de se buscar a resposta de cada bloco individualmente prevendo posteriores interligações para a formação de blocos mais complexos.

A Unidade de Controle microprogramada foi elaborada nas etapas finais da execução do projeto porque as microinstruções previstas para o microprocessador deveriam acionar esta unidade de forma imediata e para isso todas as demais unidades do processador deveriam estar concluídas.

O desenvolvimento deste microprocessador proporcionou um aprendizado de técnicas de circuitos e descrições em linguagem VHDL mas exigiu um árduo trabalho.

Os resultados obtidos foram muito satisfatórios nesta primeira versão do projeto que, atingiu seu objetivo mostrando a viabilidade da arquitetura proposta, para o microprocessador MPH.

Esperamos que o desenvolvimento do sistema tenha continuidade e que venha contribuir com as pesquisas acadêmicas e industriais na área de projeto de circuitos integrados.

Referências Bibliográficas

[1] N. Marranghello e A.J. Monticelli, "Estudo sobre a Viabilidade de Implementação de um Simulador Híbrido Baseado no Programa de Simulação em Nível de Circuitos SPICE", Rel. Tec. nº 010/88, 41 pp.,1988.

[2] N. Marranghello e F. Damiani, "Um Estudo sobre a Simulação de Circuitos", Anais do 10º Seminário ADUNESP - Guaratinguetá, pp. 167-179, 1990.

[3] N. Marranghello and F. Damiani, "ABACUS: A Hardware-Based Circuit Simulator", SID Digest of Tech. Papers, pp. 177-179,1990.

[4] N. Marranghello and F. Damiani, "A Methodology for Circuit Simulation", Proc. of The 22nd SCSC, pp.111-115,1990.

[5] N. Marranghello and F. Damiani, Tese de Doutorado: "Uma Metodologia para a Simulação de Circuitos ULSI " - DSIF/FEE/UNICAMP,1992.

[6] W.M.Luque, Tese de Mestrado: "Modelamento em VHDL de um Sistema de Simulação de Circuitos" – DSIF/FEE/UNICAMP, 1994.

[7] G.G. Langdon Jr., "Projeto de Computadores", Cartgraf Editora Ltda., 1985.

[8] K. Hwang and F.A. Briggs, "Computer Architecture and Parallel Processing", McGraw-Hill Book Company, 1985.

[9] T.C. Bartee, "Digital Computer Fundamentals", - Sixth Edition - McGraw-Hill Book Company, 1985.

[10] L.H. Pollard, "Computer Design and Architecture", Prentice-Hall Inc., 1990.

[11] L.H. Pollard, "The Designn Book - Techniques and Solutions for Digital Computer Systems", Prentice-Hall Inc., 1990.

[12] A.S.Tanenbaum, "Structured Computer Organization" – Third Edition – Prentice-Hall International, Inc. – 1990.

[13] V.C.Hamacher, Z.G.Vranesic, S.G.Zaki, "Computer Organization" – Second Edition- McGraw-Hill Book Company – 1978.

[14] J. Mick and J. Brick , "Bit-Slice Microprocessor Design", McGraw-Hill Book Company, 1980.

[15] J.P. Hayes, "Digital System Design and Microprocessors", McGraw-Hill Book Company, 1984.

[16] K. Hwang, "Computer Arithmetic"- Principles, Architecture and Design - John Wiley & Sons, Inc. - 1979.

[17] R. Silveira, Tese de Mestrado - Título: "Unidades Lógicas e Aritméticas de Alto Desempenho: Uma Experiência Utilizando Técnicas Alternativas- USP - 1993.

[18] N.H.E. Weste and K. Eshraghian, "Principles of CMOS VLSI Design - A Systems Perspective", Addison - Wesley Publishing Company, 1985.

[19] D.P. Agrawal, "High-Speed Arithmetic Arrays"- IEEE Transactions on Computers, Vol. C-28, No. 3, pp 215-224, March 1979.

[20] T.G.Noll, D. Schmitt-Landsiedel, H. Klar, G. Enders, "A Pipelined 330-MHz Multiplier" - IEEE Journal of Solid State Circuits, vol. SC-21, No. 3, June 1986.

[21] J.H.P. Zurawski and J.B. Gosling, "Design of a High-Speed Square Root, Multiply and Divide Unit"- IEEE Transactions on Computers, Vol.C-36, No. 1, pp 13-23, January 1987.

[22] S.E.McQuillan and J.V.McCanny, "VLSI Module for High-Performance Multiply, Square Root and Divide "- IEEE Proceedings-E, Vol. 136, No. 6, pp 505-510, November 1992.

[23] G. Goto, T. Sato, M. Nakajima and T. Sukemura, "A 54 x 54-b Regularly Structured Tree Multiplier" - IEEE Journal of Solid State Circuits, vol. 27, No. 9, pp 1229-1235, September 1992.

[24] C.L. David, Tese de Mestrado - Título: "Projeto de Operadores Aritméticos de Ponto Flutuante em Tecnologia CMOS"- CPGCC da UFRGS - Porto Alegre - RS - 1990.

[25] S.P. Morse and D.J. Albert, "The 80286 Architecture", John Wiley & Sons, Inc, 1986.

[26] S.P. Morse, E.J. Isaacson and D.J. Albert, "The 80386/387 Architecture", John Wiley & Sons, Inc, 1987.

[27] França, E, "Programas para o Desenvolvimento do Microprocessador MPH", relatório interno do DSIF/FEEC/UNICAMP, 1999.

[28] R. Lipsett, C. Schaefer and C. Ussery, "VHDL: Hardware Description and Design", Kluwer Academic Publishers, 1989.

[29] D.L. Perry, "VHDL" McGraw-Hill, Inc. , 1991.

[30] "V8.0 - An Introduction to Modeling in VHDL Student Workbook", Mentor Graphics Corporation, 1990.

[31] J.R.Armstrong, "Chip-Level Modeling with VHDL", Prentice Hall International, Inc., 1989.

[32] H.Ling, "High-Speed Binary Adder" – IBM J. Res. Develop., vol.25, No. 3, pp 156- 166, May 1981.

[33] S. Dasgupta – "Computer Architecture, A modern synthesis" – Vol.2, Advanced Topics, John Wiley & Sons, 1989.

[34] M. Mittal and C.A.T. Salama, "DPTL 4-b Carry Lookahead Adder" - IEEE Journal of Solid State Circuits, vol. 27, No. 11, pp 1644-1647, November 1992.

[35] N.Ide, H.Fakuhisa, Y.Kondo, T.Yoshida, M.Nagamatsu, J.Mori, I.Yamazaki and K.Ueno, "A 320-MFlops CMOS Floating-Point Processing Unit for Superscalar Processors", vol.28, No.3, pp 352-360, March 1993.

[36] J.Clouser, M.Matson, R.Badeau, R. Dupcak, S.Samudrala, R.Allmon and N.Fairbanks, " A 600-MHz Superscalar Floating-Point Processor" - IEEE Journal of Solid State Circuits, vol. 34, No. 7, pp 1026-1029, July 1999.

[37] H.Fujii,C.Hori,T.Takada,N.Hatanaka,T.Demura and G.Ootomo, " A Floating-Point Cell Library and a 100-MFlops Image Signal Processor" – IEEE Journal of Solid State Circuits, vol. 27, No. 7, pp 1080-1087, July 1992.

[38] M.Maruyama, H. Nakahira, T.Araki, S.Sakiyama,Y.Kitao, K.Aono and H.Yamada, "An Image Signal Multiprocessor on a Single Chip" - – IEEE Journal of Solid State Circuits, vol. 25, No. 6, pp 1476-1483, December 1990.

[39] M. Murakawa, S.Yoshizawa, I. Kajitani, T.Furuya, M. Iwata, T. Higuchi, "Hardware Evolution at Function Level" - Lecture Notes in Computer Science, vol.1062, pp 62-71 1996.

[40] B.Manderick, T.Higuchi., "Evolvable Hardware: An Outlook" – Lecture Notes in Computer Science, vol.1259, pp 305-311, 1997 .

[41] R.S.Zebulum, M.A.Pacheco, M.Vellasco, "Evolvable Systems in Hardware Design: Taxonomy, Survey and Applications" - Lecture Notes in Computer Science, vol 1259, pp 344-358, 1997 .