PROCESSO DE SÍNTESE DE CIRCUITOS LÓGICOS ASSÍNCRONOS

Edson do Prado Granja

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Felipe Maia Galvão França, Ph. D.

Prof. Vladimir Castro Alves, Dr.

Prof. Antônio Carneiro de Mesquita Filho, Dr. d'État

Prof. Luigi Carro, D. Sc.

Prof. Altamiro Amadeu Suzim, Dr.

Prof. Federico Gálvez-Durand, D. Sc.

RIO DE JANEIRO, RJ – BRASIL

NOVEMBRO DE 2000

GRANJA, EDSON DO PRADO

Processo de Síntese de Circuitos Lógicos Assíncronos [Rio de Janeiro] 2000

XXII, 175 p. 29,7 cm (COPPE/UFRJ, D. Sc., Engenharia de Sistemas e Computação, 2000)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Circuitos Lógicos Assíncronos
2. Arquitetura de Computadores
3. Circuitos Integrados VLSI

I. COPPE/UFRJ II. Título (série)

Ao Luiz Fernando e ao Marcelo, meus filhos.

Ao Newton, meu irmão.

Aos meus pais.

E também à memória da Lourdes.

AGRADECIMENTOS

Inicialmente quero agradecer aos meus orientadores Edil Severiano Tavares Fernandes, Felipe Maia Galvão França e Vladimir Castro Alves, cuja atenção, conselhos e entusiasmo fizeram com que eu voltasse à COPPE para aprender os fundamentos da lógica assíncrona.

Agradeço aos Programas de Sistemas e de Engenharia Elétrica, que me receberam e me ofereceram todo o suporte técnico e administrativo para a realização desta tese.

A oportunidade de utilizar os laboratórios do LPC foi fundamental para combinar arquitetura de computadores com a síntese de circuitos integrados. A convivência proporcionada por um excelente ambiente de trabalho, onde a camaradagem e as brincadeiras podiam conviver com o trabalho sério, foi fundamental para a conclusão desta tese.

Quero agradecer a colaboração de Ricardo de Freire Cássia, cujas idéias levaram ao aperfeiçoamento dos controladores ASSERT, sendo também o autor das simulações do sistema BIST S2A que apresentamos no Capítulo 8.

Quero agradecer também a João Marcelo Silva de Alcântara e Sérgio Luís Cardoso Salomão, autores do SCOB original, o fornecimento de informações sobre criptografia e síntese VHDL desse sistema. A eles também agradeço o desenvolvimento do SCOB assíncrono, que serviu para validar a metodologia S2A e, indiretamente, a metodologia ASSERT.

Um agradecimento especial ao Prof. Ney Quadros, grande amigo, que fez a revisão de Português. E ao Prof. Amaranto Lopes Pereira, outro amigo de longa data, e a quem sou muito grato pelo incentivo recebido em todos esses anos de COPPE.

Também agradeço ao CNPq e à Capes pelo suporte financeiro que me permitiu voltar aos bancos escolares.

Quero ainda agradecer à minha família pela compreensão e pelo carinho demonstrados durante um período particularmente difícil na realização desta tese. Recebi todo o apoio possível de meus filhos, meu irmão e de minha mãe. A eles agradeço de coração.

Agradeço também aos que ficaram na lembrança. Meu pai, que sempre me incentivou em meus estudos, e também à Lourdes, pelo amor, pela dedicação e pelo entusiasmo demonstrado quando ingressei no Doutorado.

E finalmente, agradeço a todos que, de uma forma ou de outra, colaboraram para a realização deste trabalho.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D. Sc.)

PROCESSO DE SÍNTESE DE CIRCUITOS LÓGICOS ASSÍNCRONOS

Edson do Prado Granja

Novembro/2000

Orientadores: Felipe Maia Galvão França
Vladimir Castro Alves

Programa: Engenharia de Sistemas e Computação

Este trabalho desenvolve uma metodologia de síntese de circuitos lógicos assíncronos aplicada a circuitos integrados VLSI. Além disso desenvolve também um modelo para a conversão de projetos síncronos em assíncronos, concluindo por sua aplicabilidade. A possibilidade da utilização do ferramental preexistente voltado para síntese de circuitos síncronos convencionais é um dos principais aspectos analisados. Foram desenvolvidas aplicações nas áreas de sistemas auto-oscilantes e projetos visando a testabilidade na produção. Os resultados obtidos em simulações corroboram os resultados previstos analiticamente.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

A SYNTHESIS METHODOLOGY FOR
ASYNCHRONOUS LOGIC CIRCUITS

Edson do Prado Granja

November/2000

Advisors: Felipe Maia Galvão França
Vladimir Castro Alves

Department: Systems Engineering

This work presents a synthesis methodology for VLSI asynchronous logic circuits. A model for the conversion of synchronous design into an asynchronous version is also presented, showing the applicability of the proposal. The use of preexistent tools dedicated to the synthesis of conventional synchronous circuits is analyzed. Applications were developed for self-oscillatory systems and design for testability. Simulation results agree with those predicted analytically.

Índice

Índice de Figuras

Índice de Tabelas

Lista de Siglas

Sigla	Significado Original	Descrição
ADFT	*Asynchronous Design for Testability*	Projeto visando a testabilidade de sistemas assíncronos
ALU	*Arithmetic-Logic Unit*	Unidade Lógica e Aritmética
ASSERT	*Asynchronous Scheduling by Edge Reversal Timing Methodology*	Metodologia para escalonamento assíncrono, temporizado por reversão de arestas.
BDD	*Binary Decision Diagrams*	Diagramas de decisão binária
BILBO	*Built-in Logic Block Observation*	Tipo de registrador assíncronos que permitem a observação de blocos lógicos
BIST	*Built in Self Test*	Circuito de auto-teste incluído no integrado
CAD	*Computer Aided Design*	Projeto feito com a ajuda de computadores
CAE	*Computer Aided Engineering*	Projeto de engenharia feito com a ajuda de computadores
CCA	*Carry Completion Adder*	Um tipo de somador
CDA	*Carry Delayed Adder*	Um tipo de somador
CLA	*Carry Lookahead Adder*	Um tipo de somador
COPPE	*Coordenação dos Programas Pós-Graduados de Engenharia*	Atualmente Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia
CP	*Carry Propagation Block*	Bloco lógico utilizado para detecção do fim de operação da *ALU*
CPU	*Central Processing Unit*	Unidade Central de Processamento

Sigla	Significado Original	Descrição
CSA	*Conditional Sum Adder*	Um tipo de somador
CSvA	*Carry Save Adder*	Um tipo de somador
CUT	*Circuit Under Test*	Circuito sob teste
DCT	*Discrete Cosine Transform*	Transformada Discreta do Coseno
DCVSL	*Differential Cascode Voltage Swing Logic*	Lógica auto-temporizada por tensão diferencial em cascata
DES	*Data Encryption Standard*	Um algoritmo criptográfico
DFT	*Design for Testability*	Projeto visando a testabilidade
DI	*Delay Insensitive*	Classe de circuitos assíncronos que são insensíveis aos atrasos
EDIF	*Electronic Design Interchange Format*	Formato para Intercâmbio Eletrônico de Dados de Projeto
FIFO	*First In, First Out*	Um tipo de registrador no qual a fila de saída segue a ordem da fila de entrada
HSCLA	*Hybrid Statistical Carry Lookahead*	Um tipo de somador
IDEA	*International Data Encryption Algorithm*	Um algoritmo criptográfico
IEEE	*Institute of Electrical and Electronic Engineers*	Órgão dedicado à difusão do avanço tecnológico nas engenharias elétrica e eletrônica.
IPCMOS	*Interlocked Pipelined CMOS*	Uma família de blocos lógicos assíncronos
LFSR	*Linear Feedback Shift Register*	Registrador de deslocamento linearmente realimentado
LPC	*Laboratório de Projeto de Circuitos Integrados*	Laboratório da COPPE/UFRJ ligado ao Programa de Engenharia Elétrica
MIC	*Multiple Input Change*	Permissão para efetuar mudanças simultâneas em múltiplas entradas

Sigla	Significado Original	Descrição
MTBF	*Mean Time Between Failures*	Tempo médio entre falhas
PLL	*Phase Locked Loop*	Realimentação com controle de fase
QDI	*Quasi Delay Insensitive*	Classe de circuitos assíncronos que são quase insensíveis a atrasos
RCA	*Ripple Carry Adder*	Um tipo de somador
RISC	*Reduced Instruction Set Computer*	Computador com conjunto de instruções reduzido
RSA	*Rivest, Shamir and Adleman Encoding*	Um algoritmo de criptografia
S2A	*Synchronous to Asynchronous Methodology*	Metodologia para conversão de circuitos síncronos em assíncronos
SA	*Signature Analyzer*	Analisador de assinaturas
SCOB	*Soft-Core for the Blowfish cryptographic algorithm*	Uma descrição em software para a implementação do algoritmo Blowfish de criptografia
SEL	*Carry Select Adder*	Um tipo de somador
SER	*Scheduling by Edge Reversal*	Escalonamento por reversão de arestas
SI	*Speed Independent*	Uma classe de circuitos assíncronos cuja operação independe da velocidade das portas lógicas
SIC	*Single Input Change*	Circuitos onde apenas uma variável muda a cada vez.
SKP	*Carry Skip Adder*	Um tipo de somador
SMER	*Scheduling by Multiple Edge Reversal*	Escalonamento por Reversão Múltipla de Arestas
SPEC	*Standard Performance Evaluation Corp*	Empresa que produz programas padrão para avaliação de performance de computadores.
STG	*Signal Transition Graphs*	Uma técnica para síntese assíncrona

Sigla	Significado Original	Descrição
TPG	*Test Pattern Generator*	Gerador de vetores de teste
TRA	*Test Response Analyzer*	Analisador das respostas do circuito aos estímulos do teste.
Triple DES	*Triple Data Encryption Standard*	Codificação DES que utiliza três chaves aplicadas sucessivamente à mensagem
UFRJ	*Universidade Federal do Rio de Janeiro*	Local de desenvolvimento desta tese
UIC	*Unrestricted Input Change*	Circuito em que todas as entradas podem variar simultaneamente
ULSI	*Ultra Large Scale Integration*	Escala de Integração Ultra Alta
VBS	*Verilog Behavioral Simulator*	Simulador comportamental para a linguagem Verilog
VHDL	*VHSIC Hardware Description Language*	Linguagem para simulação e síntese de circuitos eletrônicos
VHSIC	*Very High Speed Integrated Circuits*	Circuitos Integrados de Muito Alta Velocidade
VLSI	*Very Large Scale Integration*	Escala de Integração Muito Alta

The first time it is a kludge. The second, a trick.
Later, it is a well-established technique! –
Mike Broido, Intermetrics.

Our vision is to speed up time, eventually eliminating it. –
Alex Schure.

Capítulo 1
Considerações Iniciais

1.1. Introdução

A síntese de circuitos computacionais utilizando lógica assíncrona faz parte da própria história dos computadores. Na década de 1950, quando os projetos de circuitos lógicos utilizavam principalmente um seqüenciamento do tipo eletromecânico, estudos teóricos já procuravam analisar os problemas inerentes à síntese de máquinas assíncronas [1].

A partir dos anos 60, o desenvolvimento de circuitos síncronos fez com que o uso de lógica assíncrona fosse praticamente esquecido. No entanto a tecnologia assíncrona sempre esteve presente, complementando técnicas síncronas. A implementação de interfaces seriais do tipo RS-232, ou mesmo paralelas, como a interface SCSI, são assíncronas por definição. Subsistemas para controle de interrupção, controle de barramento, arbitradores e registradores FIFO, além de contadores e temporizadores, são muitas vezes implementados de forma assíncrona mesmo em equipamentos síncronos.

A atenção dos projetistas voltou-se novamente para circuitos assíncronos a partir da década de 90, quando o desenvolvimento da integração em larga escala colocou em evidência algumas limitações da tecnologia síncrona. Essas limitações, que constituem motivação suficiente para o estudo e desenvolvimento de máquinas assíncronas, mostram que máquinas assíncronas possuem: [2]

- Um potencial para maior performance.
- Um potencial para menor dissipação de potência.
- Menor geração de ruídos e de emissões eletromagnéticas.

- Maior facilidade de integração entre sistemas com temporizações heterogêneas.

No início da década de 90, a pesquisa em lógica assíncrona ainda procurava desenvolver circuitos básicos e técnicas para a verificação formal da síntese. Com o desenvolvimento de técnicas básicas, maior ênfase pode ser dada ao desenvolvimento de sistemas *VLSI* inteiramente assíncronos. Microprocessadores como o *ARM* do Projeto *AMULET* [3] e o *80C51* Assíncrono da Philips [4] atestam o interesse e a viabilidade do desenvolvimento de sistemas assíncronos *VLSI*.

1.2. Objetivos Gerais

O estágio atual de desenvolvimento de circuitos assíncronos está voltado para a geração de técnicas de síntese de circuitos assíncronos que garantam um perfeito seqüenciamento dos estados da máquina, evitando-se quaisquer variações de estado que possam levar a um funcionamento não previsto. Essas técnicas no entanto, estão voltadas para o desenvolvimento de pequenos circuitos individuais ou subsistemas.

Circuitos mais complexos como microprocessadores ainda são projetados a partir de um processador síncrono preexistente. Porém, quando a performance for o objetivo principal, o microprocessador assíncrono precisa ser inteiramente concebido nessa metodologia. A conversão de um projeto pode apresentar benefícios, mas deve ser encarada como uma etapa intermediária para o desenvolvimento do sistema.

Nosso trabalho procura uma solução para a síntese assíncrona em nível de sistema. Sua aplicação está voltada para a interligação de diversos subsistemas que possam incluir quaisquer metodologias de síntese. O circuito básico a ser incluído na arquitetura deve apenas aceitar um comando de início e responder com um sinal de fim de operação. A granularidade de cada subsistema deve poder proporcionar desempenho adequado a sistemas de alta performance ou através de procedimentos de hierarquia, poder incluir circuitos tão complexos quanto um processador dedicado.

Problemas referentes à testabilidade dos circuitos projetados também devem ser considerados, uma vez que a dificuldade de se aplicar os métodos tradicionais de teste aos circuitos assíncronos é evidente. O teste de circuitos síncronos depende da aplicação de vetores de teste às entradas, e da análise das respostas do circuito após um pulso de relógio. Circuitos assíncronos não possuem relógio global.

A escassez de ferramentas apropriadas ao projeto de circuitos assíncronos, e de modelos apropriados para a implementação de circuitos integrados com

componentes auto-temporizados, aliada à pouca disponibilidade de recursos para a aquisição de novas ferramentas fez com que se procurasse ter também por objetivo o desenvolvimento de circuitos assíncronos valendo-se do ferramental existente para o desenvolvimento de máquinas síncronas.

1.3. Descrição do Trabalho

Uma avaliação do potencial de benefícios proporcionados pela utilização de máquinas assíncronas foi o ponto de partida deste trabalho. Uma revisão de trabalhos existentes sobre metodologia de síntese assíncrona proporcionou um cenário apropriado para a verificação da possibilidade de uma contribuição ao avanço do estado da arte.

Muito embora o desenvolvimento de circuitos assíncronos básicos tais como células C-Müller [5], somadores [6] e multiplicadores [7], arbitradores e sincronizadores [8] tenha recebido um grande impulso no início dos anos 90, e que esse impulso tenha sido estendido ao desenvolvimento de subsistemas fundamentais como o controle de pipelines [9], do ponto de vista da arquitetura de sistema poucas propostas foram feitas. A maior parte das propostas visava a otimização do escalonamento da seqüência de instruções [10]. Essas propostas eram voltadas para projeto de alto nível. Procuramos complementar os estudos existentes com uma proposta de metodologia de síntese voltada para a arquitetura de alto nível e capaz de incluir como subsistemas os desenvolvimentos já existentes.

Uma avaliação da metodologia de testes disponíveis para circuitos assíncronos [11] mostrou a dificuldade existente para a aplicação de testes do tipo *scan*. Essa dificuldade é devida tanto aos problemas para inicialização do circuito a ser testado como do controle do avanço para o próximo estado. Fizemos então os estudos necessários à proposição de dois subsistemas integrados de teste – *BIST* – sendo um deles inteiramente assíncrono e o outro como produto da conversão de um *BIST* síncrono para assíncrono.

Outro estudo de caso foi feito para a utilização de metodologia assíncrona na conversão de um subsistema de criptografia originalmente síncrono. Trata-se de uma aplicação razoavelmente complexa e que testou os limites das ferramentas disponíveis em nosso laboratório.

1.4. Contribuições

Dentre as contribuições desta tese temos o desenvolvimento de uma arquitetura voltada para a síntese de sistemas assíncronos de porte variado. Essa arquitetura, que inclui controladores de alto desempenho, é capaz de seqüenciar circuitos mono ou multifásicos e pode ser vista como uma unidade de controle distribuída. Não existem restrições de arquitetura quanto à elaboração do caminho de dados e as unidades de execução dependem apenas de um sinal de início de operação, devendo fornecer o correspondente sinal de fim da operação.

Também apresentamos uma metodologia adequada à avaliação do potencial de desempenho de circuitos assíncronos. Essa metodologia foi aplicada à avaliação de somadores auto-temporizados, e é também a base para a determinação da possibilidade de se obter ganhos na conversão de circuitos síncronos em assíncronos. Neste último caso, a aplicação de um somador auto-temporizado a um *soft-core* para criptografia proporcionou um ganho expressivo ao circuito, tendo sido rápida e facilmente implementada.

Na área de testes de sistemas assíncronos apresentamos nossa proposta para teste integrado de sistema – *BIST* – que representa um avanço na metodologia de testes de circuitos assíncronos. Esse avanço se deve ao uso da arquitetura proposta, que permite controlar a seqüência de testes na mesma velocidade utilizada para operação normal do circuito.

Ainda como contribuições desta tese apresentamos um conjunto de trabalhos apresentados internamente na COPPE/UFRJ como parte dos exames de qualificação. O seu desenvolvimento resultou em uma série de trabalhos apresentados em congressos nacionais e internacionais.

1.5. Terminologia

Tratando-se de assuntos como a integração em larga escala onde a maior parte dos termos específicos usuais têm sua origem na língua inglesa, é natural que a tradução desses termos nem sempre represente de maneira exata o significado original. É possível ainda que um excesso de preciosismo no emprego de termos na língua portuguesa leve a uma dificuldade na compreensão do trabalho pois terminologias diferentes são empregadas em centros técnicos diversos tanto no Brasil como em Portugal.

Assim, optamos por manter alguns desses termos, como *throughput* e *reset,* no idioma original, mas apresentando sempre que possível uma locução descritiva no corpo da tese. Quanto às abreviaturas e siglas, elas também aparecem no idioma original. Contudo, o significado e uma possível tradução constam da Lista de Abreviaturas. Isso acontece, por exemplo, nas referências a *ALU* e *VLSI.*

Outros termos foram traduzidos. É o caso, por exemplo, de *clock*, que foi traduzido por relógio. Nesta tese não tratamos de relógios contadores de tempo decorrido. O termo relógio refere-se apenas a um gerador de pulsos de sincronismo.

1.6. Organização do Trabalho

Neste primeiro capítulo apresentamos um resumo dos objetivos do trabalho e dos resultados alcançados. Devido à abrangência da tese, fazemos uma revisão de conceitos básicos no Capítulo 2.

Trabalhos recentes na área de síntese assíncrona e que foram relevantes para o direcionamento da tese são mencionados no Capítulo 3. A análise desses trabalhos bem como a motivação e a proposta de tese estão no Capítulo 4.

O Capítulo 5 apresenta o desenvolvimento teórico da proposta de tese, o que inclui a apresentação de metodologias de síntese para temporização de circuitos mono e multifásicos e também para a conversão de uma classe de circuitos síncronos em assíncronos. Os trabalhos desenvolvidos para a validação da proposta aparecem a seguir. O Capítulo 6 apresenta o desenvolvimento de uma ferramenta básica para síntese. Trata-se de um somador auto-temporizado que utiliza circuitos lógicos convencionais e a sua aplicação como parte de uma *ALU* auto-temporizada.

No Capítulo 7 a metodologia de síntese é aplicada à geração e simulação de um temporizador multifásico e de um sistema *BIST* cuja velocidade de operação é a mesma do circuito sob teste. O Capítulo 8 trata da conversão de circuitos síncronos em assíncronos com dois estudos de caso. O primeiro trata de testabilidade e propõe a conversão de um sistema *BIST* originalmente síncrono para operação assíncrona. O segundo caso trata da conversão de um *soft-core* para criptografia que utiliza o algoritmo Blowfish. Esse circuito reúne características das diversas metodologias propostas nesta tese.

Conclusões e sugestões para desenvolvimento futuro complementam o trabalho e podem ser vistos no Capítulo 9. No Anexo 1 são mencionados os trabalhos

derivados da pesquisa de tese e no Anexo 2 é apresentada a listagem em *VHDL* do circuito desenvolvido para a *ALU* auto-temporizada.

Capítulo 2
Conceitos Básicos

2.1. Introdução

Dada a abrangência dos assuntos tratados, procuramos fazer neste capítulo uma revisão de conceitos básicos utilizados no decorrer desta tese. A abordagem dos assuntos é feita de forma simplificada, visando estabelecer uma terminologia que será utilizada no desenvolvimento do trabalho. Maiores detalhes podem ser obtidos nas obras referidas no texto.

2.2. Histórico da Síntese de Sistemas VLSI

Apresentamos nesta seção alguns dos problemas inerentes à integração em larga escala quando projetados com o modelo síncrono tradicional. Esses problemas constituem motivação suficiente para o estudo de sistemas assíncronos como uma possível solução para a síntese de circuitos integrados em ultra larga escala.

Do ponto de vista histórico lembramos que os projetos de circuitos integrados dependiam inicialmente de procedimentos inteiramente manuais. Isso acontecia desde o projeto do circuito até a elaboração da máscara. Procedimentos relativos ao processo de fabricação já haviam sido automatizados durante o período em que se desenvolveram os circuitos discretos. Em pouco tempo isso tornou-se inteiramente inviável devido à complexidade dos circuitos, às dimensões físicas envolvidas e principalmente à falha humana.

Da necessidade de se criar ferramentas que permitissem a síntese eficiente e correta de sistemas complexos foram surgindo metodologias de projeto, que uma vez incorporadas às ferramentas, permitiram um melhor controle das diversas fases de projeto. Muitas dessas metodologias foram oriundas de procedimentos utilizados para a síntese e montagem de circuitos com componentes discretos, sendo adaptadas para a utilização em circuitos integrados.

Talvez a principal contribuição dessa fase heróica tenha sido a elaboração de um sistema hierárquico de projeto, onde sistemas complexos podem ser descritos como uma coleção de subsistemas cada vez mais simples, até chegarmos a um simples componente básico do sistema. Esse procedimento *top-down* foi combinado com procedimentos *bottom-up* que resultaram na criação de blocos lógicos previamente desenvolvidos e colocados à disposição do projetista como *standard-cells* [12].

O projeto de *standard-cells* e de outros blocos *full-custom* foi facilitado pelo método Lambda [13], onde as dimensões dos componentes da máscara é feita em função de um valor dimensional mínimo e seus múltiplos inteiros.

As ferramentas de CAD, originalmente voltadas para o desenho de máscaras, foram integradas como CAD/CAE, permitindo a captura de diagramas esquemáticos e a minimização automática de funções lógicas. Para cada passo do projeto foi elaborado um processo de verificação e simulação para garantir que uma descrição lógica correta pudesse gerar, no final do processo, uma máscara também correta.

Faltava então garantir que as falhas introduzidas durante o processo de fabricação seriam detectadas antes da comercialização do produto. Para isso foram desenvolvidas técnicas de teste e também técnicas de projeto visando a testabilidade (*design for testability*).

Os sistemas lógicos já podiam então serem tratados de forma eficiente através das linguagens de descrição de *hardware*. Essas linguagens, que inicialmente faziam apenas a listagem de componentes e suas interligações, permitiram um tratamento eficiente dos processos de captura de diagramas esquemáticos e também da simulação lógica e elétrica, pois elas podiam ser utilizadas como interface entre ferramentas dedicadas a diversas etapas de projeto.

A descrição da arquitetura do projeto vem sendo progressivamente substituída pela descrição do comportamento previsto para a lógica do sistema, aumentando o nível de abstração da síntese do projeto. Uma ferramenta especializada fica encarregada da síntese da arquitetura que produz o comportamento desejado.

A quantidade de ferramentas de projeto teve um tal desenvolvimento na última década que hoje o projetista tem uma garantia bastante forte de que o circuito irá funcionar já na primeira "fornada". Talvez o circuito não tenha o desempenho

esperado devido a deficiências das ferramentas de síntese e de simulação, mas a correção do projeto lógico é praticamente garantida.

Vimos neste histórico que todo o desenvolvimento das metodologias de síntese está baseado em um modelo síncrono de funcionamento dos circuitos. No entanto, o modelo síncrono apresenta limitações de desempenho que começam a se tornar importantes para tecnologias que utilizam definições muito abaixo do mícron *(deep submicron).*

Sistemas assíncronos existem desde a década de 1950. Após um breve período de interesse, a síntese assíncrona foi relegada a segundo plano, já que projetos controlados por um único sistema de sincronismo são mais facilmente implementados. No entanto, os fundamentos estabelecidos nessa época foram suficientes para despertar a curiosidade dos projetistas no momento em que a síntese síncrona começou a ter seus limites mais claramente definidos.

Cabe então apresentarmos uma revisão do conceito de máquinas síncronas e da metodologia básica de projeto dessas máquinas. A seguir veremos de que forma a utilização de máquinas assíncronas pode estender esses limites e quais os cuidados adicionais que devem ser tomados em um projeto voltado para funcionamento assíncrono.

2.3. Ferramentas Auxiliares

Para que a síntese de circuitos integrados possa ser feita de forma estruturada, algumas ferramentas matemáticas e de descrição estruturada dos circuitos devem ser adotadas. Algumas das principais ferramentas utilizadas neste trabalho estão descritas a seguir.

2.3.1. Grafos

Apresentamos a seguir uma revisão dos conceitos de Teoria dos Grafos de modo a estabelecer a terminologia utilizada neste trabalho. Os conceitos foram simplificados para evitar que um excesso de rigor matemático prejudicasse a compreensão das idéias tratadas. Maiores detalhes podem ser obtidos, por exemplo, na referência [14], na qual baseamos nosso trabalho, ou em outros trabalhos sobre Teoria de Grafos [15] [16].

Grafos não orientados. Um grafo G é definido por $G=(V(G),E(G),\psi_G)$, onde:

$V(G) = \{v_1, \ldots, v_n\}$, é um conjunto de vértices

$E(G) = \{e_1, \ldots, e_n\}$, é um conjunto de arestas, e

$\psi_{e_i} = (u_i, v_i) \; \forall \; e_i \in E$, é a função de incidência que associa a cada aresta um par de vértices u e v, distintos ou não.

Lembramos ainda que V não pode ser um conjunto vazio, e que V e E são conjuntos distintos. E que se V e E são finitos então G também será finito.

Para a construção de um grafo representamos os vértices por pontos e as arestas por linhas que unem os pontos de acordo com a função de incidência. As posições relativas dos pontos e linhas não afetam as propriedades do grafo.

Um vértice pode receber uma quantidade qualquer de arestas incidentes, mas cada aresta incide apenas sobre dois vértices, distintos ou não. Vértices ligados por uma aresta são ditos **adjacentes** ou **vizinhos**.

Se existe apenas uma aresta interligando dois vértices quaisquer, o grafo é dito **simples**. Quando os vértices são distintos a aresta forma um ***link***, caso contrário ela forma um laço ***(loop)***.

Um grafo G que não contém arestas é dito **vazio**. E um grafo onde cada par de vértices distintos está interligado por uma aresta é dito **completo**. Um subconjunto de G que forme um grafo completo é chamado de ***clique***.

Um grafo é dito **bipartido** quando ele pode ser particionado em dois conjuntos X e Y de tal forma que todas as arestas interliguem vértices em X com vértices em Y. Além disso ele será **bipartido completo** se todos os vértices de X estiverem ligados a todos os vértices de Y.

O **grau** de um vértice é dado pela quantidade de arestas incidentes. Um grafo será dito **regular** se todos os seus vértices forem do mesmo grau.

As definições a seguir, embora sejam feitas para grafos não orientados, referem-se aos pontos inicial e final de um percurso, o que implica em um sequenciamento das operações. Grafos orientados, que serão tratados posteriormente, impõem mais restrições ao percurso.

Um **passeio** (*walk*) W em G é qualquer seqüência não nula composta por arestas e vértices alternados que interligam o vértice de origem ao vértice terminal. Se

todas as arestas de W forem distintas teremos então uma **trilha** (*trail*). E se além disso todos os vértices forem distintos teremos um **caminho** (*path*).

Em um grafo **conexo** é sempre possível definir um caminho que interligue dois vértices quaisquer. A **distância** entre dois vértices é dada pela quantidade de arestas que formam o **caminho mínimo** entre esses vértices.

Um passeio que começa e termina no mesmo vértice é dito **fechado** (*closed*). Uma trilha fechada que contenha um ou mais vértices internos forma um **ciclo**. Um grafo conexo onde não seja possível criar uma trilha que forme um ciclo é dito **acíclico**, e constitui uma **árvore**. Um conjunto de árvores forma uma **floresta**.

Grafos orientados. São também chamados de grafos direcionados ou digrafos.

Até agora vimos grafos onde a função de incidência não especificava a ordem da conexão aos vértices. Para criarmos um digrafo basta redefinir a função de incidência, introduzindo o conceito de ordem ou sentido da conexão. Na função $\psi_e = (u, v)$ dizemos que a aresta e tem origem em u e aponta para v. Essa aresta passa a ser representada por uma seta.

Os conceitos vistos para grafos não orientados podem ser facilmente estendidos para digrafos. Por exemplo, para existir um caminho no digrafo é preciso que esse caminho exista no grafo não orientado subjacente. E é também necessário que, no digrafo, esse caminho possa ser percorrido seguindo a orientação das arestas.

Dizemos que um digrafo forma um **ciclo** se existir pelo menos uma trilha fechada que possa ser percorrida seguindo a orientação das arestas. Caso contrário o digrafo é acíclico.

Um vértice no qual todas as arestas apontam para outros vértices, ou seja, um vértice que é o ponto inicial de todas as suas arestas, é chamado de **fonte** (***source***). Um vértice que é o ponto terminal de todas as suas arestas incidentes é chamado de **sumidouro** (***sink***).

2.3.2. Linguagens de Descrição de Hardware

A ferramenta usual da engenharia eletrônica para descrição de um circuito é o diagrama esquemático (esquema). A representação pictórica de um circuito torna mais fácil a sua compreensão mas traz consigo alguns inconvenientes.

O procedimento histórico para criação do esquema envolvia a elaboração de um desenho básico feito pela engenharia, que era posteriormente "passado a limpo" por um desenhista. A produção do desenho do diagrama esquemático somente era feita após a depuração do circuito, quando havia uma probabilidade razoável de que o circuito não seria modificado. Eventuais modificações no circuito geralmente implicavam na criação de um novo desenho. Nessa época, a depuração do circuito era feita em bancada.

Com o advento dos processos de automação o trabalho foi simplificado, pois o desenho passou a ser feito no computador, muitas vezes pela própria equipe de engenharia. Começaram então a surgir programas capazes de simular o funcionamento de um circuito desde que esse circuito fosse descrito como uma rede de interligação de componentes. O próximo passo foi o surgimento de programas capazes de fazer a captura de um diagrama esquemático e gerar a lista de conexões (*netlist*).

Dentre os formatos possíveis para uma *netlist* cabe destacar o *EDIF* – Electronic Design Interchange Format, que foi adotado pela indústria como formato padrão para a geração de *netlists*. Uma das grandes vantagens proporcionadas pelo *EDIF* foi a padronização das entradas em formato adequado à leitura mecânica. Isto facilitou o processo de produção de circuitos e também a aplicação em simuladores, mudando o enfoque de depuração em bancada para depuração em computador.

Com o advento dos circuitos integrados a depuração em bancada passou a ter um custo proibitivo, além de um ciclo de depuração muito lento. A atenção dos projetistas voltou-se então para o desenvolvimento de simuladores poderosos que pudessem praticamente garantir que o circuito integrado produzido funcionaria a contento.

Devido à similaridade entre as linguagens de descrição de *hardware* e as linguagens de programação de computadores foram desenvolvidas algumas linguagens que dispensavam a geração do diagrama esquemático. O procedimento de síntese, que antes era inteiramente arquitetural, isto é, que procurava descrever cada componente e suas interligações, passou a ser comportamental. As novas linguagens já permitem que um circuito digital possa ser descrito pela função desejada, cabendo ao instrumento de síntese a geração de uma arquitetura adequada, sujeita a restrições

como tempo de propagação procurado, área permitida e ainda restrições próprias da tecnologia alvo.

Dentre as linguagens apropriadas para simulação lógica digital, duas merecem destaque, o Verilog e o *VHDL*.

2.3.2.1. Verilog

Verilog foi criada pela Gateway Design Automation e destinava-se inicialmente à simulação de circuitos. Essa empresa foi adquirida pela Cadence em 1989. Após um período de desenvolvimento, a Cadence colocou o Verilog em domínio público. Foi criado então o OVI – *Open Verilog International*, que fez as modificações necessárias para que ela fosse aceita como padrão pelo IEEE, o que veio a acontecer em 1995 [17]. O Verilog vem sendo ativamente desenvolvido também pela Synopsys.

A sintaxe do Verilog é muito próxima da sintaxe da linguagem C, o que facilita a sua difusão. Infelizmente, os fabricantes seguem seus próprios padrões de linguagem em detrimento do padrão IEEE, e isto dificulta a migração entre ferramentas de fabricantes diversos.

Referências sobre a linguagem podem ser obtidas com a Silicon Logic [18], ou no *newsgroup* `comp.lang.verilog`. Um compilador gratuito, porém bastante limitado, pode ser obtido com a Wellspring Solutions [19] ou na página do VBS[20].

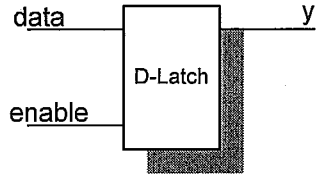


Figura 2.1: Um flip-flop do tipo D.

A Figura 2.1 representa um flip-flop do tipo D. Um exemplo de código Verilog para a descrição comportamental desse flip-flop pode ser visto na Figura 2.2 [21].

```
module d_latch (enable, data, y)
    input enable, data;
    output y;
    reg y;

always @ (enable or data)
    if (enable) y = data;
endmodule;
```

Figura 2.2: Código Verilog para um flip-flop do tipo D.

2.3.2.2. VHDL

O *VHDL* é uma linguagem que surgiu a partir de uma iniciativa do governo americano chamada *VHSIC – Very High Speed Integrated Circuits*, que teve início em 1980 [22].

VHDL é uma linguagem cujo padrão foi definido pelo IEEE em 1987 [23], ficando conhecida como *VHDL-87*. Esse padrão foi revisto em 1992 e essa revisão foi adotada em 1993, gerando o *VHDL-93* [24].

VHDL ou *VHSIC Hardware Description Language* é baseada na sintaxe da linguagem Ada, o que facilita a descrição de tarefas concorrentes. Referências sobre *VHDL* podem ser obtidas no *VHDL International User's Forum* e no *newsgroup* `comp.lang.vhdl`.

VHDL suporta o conceito de hierarquia, o que permite a descrição da arquitetura de um sistema através da definição dos módulos, submódulos e componentes, cada um deles interligando-se aos demais através de portas de Entrada e Saída.

Mais interessante que a descrição de arquitetura é a descrição do comportamento de um circuito. É possível, tanto em *VHDL* como em Verilog, especificar o comportamento funcional de um determinado circuito. É essa característica que permite que essas linguagens sejam utilizadas para a síntese de circuitos eletrônicos em alto nível. O processo todo é semelhante a uma compilação, onde a entrada corresponde à função desejada e a saída corresponde à arquitetura necessária para compor a função.

Um exemplo de código *VHDL* para a descrição comportamental do flip-flop D da Figura 2.1 pode ser vista na Figura 2.3.

```
entity d_latch is
port (enable, data: in std_logic;
                 y: out std_logic);
end d_latch;

architecture behave of d_latch is
begin process (enable, data)
begin
    if (enable = '1') then y <= data;
    endif;
end process;
end behave;
```

Figura 2.3: Código VHDL para um flip-flop do tipo D.

2.4. Máquinas Síncronas e Assíncronas

2.4.1. Definição

Máquinas síncronas podem ser descritas pelo arranjo da Figura 2.4, onde os sinais são binários e o tempo é discreto. Sinais binários permitem a utilização da lógica Booleana. Pelo fato do tempo ser discreto, estados transientes do sinais não precisam ser levados em consideração [2].

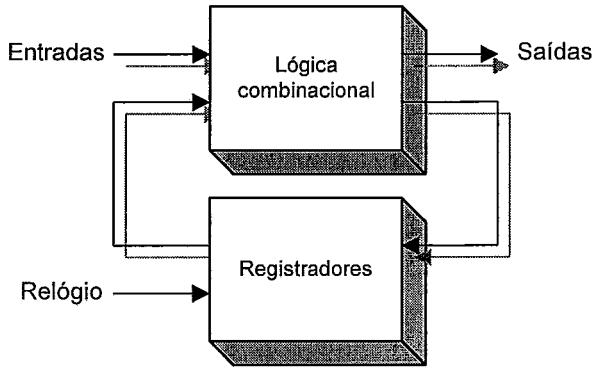


Figura 2.4: Máquina síncrona genérica

O estado atual da máquina fica armazenado em um conjunto de registradores. O valor do próximo estado é calculado por circuitos combinacionais que operam tanto sobre um vetor de entradas como sobre o vetor que define o estado atual. A mudança de estado ocorre após um pulso de relógio.

Nas **máquinas assíncronas** a mudança de estado se processa de forma independente em cada registrador, sendo sujeita apenas a restrições de ordem local, como por exemplo, a ocupação dos recursos vizinhos. Uma esquematização semelhante à da Figura 2.4 poderia ser feita retirando-se o sinal de relógio e lembrando que sinais de sincronismo são gerados juntamente com os dados.

2.4.2. Desempenho

Máquina síncrona. Para que se possa obter benefícios da definição de intervalos discretos de tempo, é necessário abstrair os efeitos de sinais transientes sobre as saídas. Assim, todo o conjunto lógico combinacional deve atingir um estado quiescente antes que o próximo pulso de relógio possa ser aplicado ao circuito.

Vemos então que o pior caso para os atrasos dos diversos circuitos combinacionais estabelece um piso para o período do relógio. Lembramos que ao calcular esse piso devemos levar em conta o pior caso das **condições de ambiente**, ou seja, tensão mínima de alimentação e valor máximo da temperatura, além dos parâmetros mínimos aceitáveis para o conjunto de entradas.

A esse piso ainda devemos acrescentar um percentual que cubra o espalhamento natural na fabricação dos circuitos mesmo quando os produtos são classificados por velocidade após a fabricação.

Máquina assíncrona. Ao contrário das máquinas síncronas, cujo desempenho fica limitado por um critério de pior caso, as máquinas assíncronas levam em consideração o tempo real de computação de cada estágio, podendo prover um desempenho equivalente ao tempo médio de computação.

Um exemplo típico é proporcionado por somadores do tipo RCA – *Ripple Carry Adder*. O desempenho desses somadores é proporcional à maior cadeia de *carry* da soma. Para entradas aleatórias, o desempenho médio é proporcional a $\log_2 n$, onde n corresponde à largura em bits do somador. No entanto, para utilização em circuitos síncronos, o pior caso, que é proporcional a n, é que deve ser considerado. É verdade que para utilização em circuitos síncronos o RCA não é o somador mais indicado. Mas esse somador, apesar de bastante simples, pode proporcionar desempenho satisfatório em circuitos assíncronos conforme n aumenta. Lembramos que os barramentos típicos dos processadores atuais (ano 2000) variam entre 32 e 128 bits.

O desempenho dos circuitos assíncronos acompanha naturalmente as variações ditadas pelas condições de ambiente e pelo espalhamento da fabricação, não sendo necessário implementar nenhuma providência especial de compensação.

Devemos mencionar no entanto que parte da vantagem proporcionada pelos circuitos assíncronos é perdida pela necessidade de se detectar o fim de cada operação.

2.4.3. Dissipação de Potência

Circuitos síncronos. Conforme mostramos anteriormente, a cada mudança de estado do relógio todos os registradores dissipam potência, independentemente de haver ou não mudança de estado. Caso existam circuitos combinacionais com lógica dinâmica, eles também irão dissipar potência a cada pulso de relógio.

Quando consideramos circuitos integrados do tipo CMOS, podemos dizer que a corrente quiescente dissipada pelo circuito é muito pequena, podendo normalmente ser desprezada na avaliação do consumo. Vemos então que a dissipação de potência cresce linearmente com a freqüência de operação. No momento em que processadores comerciais atingem a região de 1GHz, a dissipação de potência pode ser um fator limitante do desempenho. Os fabricantes tentam diminuir a potência consumida valendo-se de tensões de alimentação cada vez mais baixas. A tensão padrão para circuitos lógicos foi, durante muito tempo, de 5V. Hoje o núcleo dos integrados *VLSI* trabalha normalmente com 2.2V (para tecnologias de 0.25μm).

Circuitos assíncronos. Ao contrário do que ocorre com máquinas síncronas, nas assíncronas a dissipação de potência somente ocorre quando e onde houver atividade. Um co-processador aritmético assíncrono, quando em repouso, dissipa apenas potência quiescente.

Economia de potência pode ser obtida em máquinas síncronas pelo desligamento de módulos, como é feito em computadores portáteis, o que implica em perda temporária de funcionalidade. Circuitos especiais para reduzir a freqüência de relógio ou ainda para chaveamento seletivo de parte dos circuitos também podem ser empregados em máquinas síncronas. Com máquinas assíncronas todas essas vantagens são obtidas com menor granularidade e sem a necessidade de um procedimento especial ou de perda de funcionalidade.

Outra vantagem pode ser vista na utilização de contadores assíncronos. A dissipação de potência de uma cadeia contadora assíncrona está limitada a duas vezes a dissipação do primeiro elemento, sendo independente do tamanho da cadeia. Relógios digitais, por exemplo, valem-se de um contador assíncrono de 15 bits para gerar um sinal de 1Hz a partir de um cristal de 32 kHz. Para contadores síncronos a dissipação é proporcional ao tamanho da cadeia.

2.4.4. Ruído

Máquinas síncronas. Circuitos síncronos são naturalmente ruidosos, pois praticamente toda a energia consumida é dissipada no entorno da transição dos pulsos de relógio. No interior de um circuito integrado, onde as impedâncias são mais elevadas, existe uma significativa modulação nos barramentos de alimentação, causando interferência nas vias de sinais. A tendência atual de utilizar tensões de

alimentação mais baixas também aumenta a suscetibilidade dos circuitos aos diversos tipos de ruído.

Os pulsos de energia causados pelo sinal de relógio também causam uma significativa irradiação de energia eletromagnética na freqüência do relógio e seus harmônicos. Por essa razão, a blindagem de circuitos digitais deve sempre ser considerada quando existem circuitos analógicos de baixo nível nas proximidades.

Máquinas assíncronas. A geração de pulsos de ruído nas vias de alimentação e a emissão de radiação eletromagnética são consideravelmente menores em circuitos assíncronos devido a dois fatores:

1. Apenas os circuitos que estão em atividade dissipam potência, ao contrário do que acontece com circuitos síncronos. Contudo, essa vantagem pode ser diminuída pelo maior consumo de potência por função lógica das versões assíncronas, pois elas necessitam um circuito extra para prover o sincronismo devido à ausência de um relógio global.

2. A ausência de um relógio global espalha a interferência das operações de chaveamento. Uma comparação entre o espectro dos pulsos de corrente de um *80C51* assíncrono, da Philips, com a versão original, mostra uma redução acentuada no espectro mensurável da corrente de alimentação [4].

2.4.5. Distribuição do Sinal de Relógio

Circuitos síncronos VLSI dependem da correta distribuição de pulsos de relógio através de uma árvore de geração do relógio. Os pulsos assim distribuídos não são instantaneamente iguais, pois para isso seria necessário que todos os componentes ativos e passivos do circuito fossem exatamente iguais. A distribuição do relógio torna-se ainda mais crítica com o aumento da área de integração e com o aumento da freqüência de operação. Um fator adicional para a dificuldade de distribuição do relógio está ligado à escala da tecnologia de integração. Nas tecnologias com medidas abaixo de um mícron, o retardo dos sinais devido ao roteamento passa a ser tão ou mais importante que o retardo dos componentes ativos, o que provoca uma imprecisão extra na aplicação dos sinais de relógio em ramos diferentes da árvore geradora.

Podemos então definir uma região física equipotencial de tempo, onde é possível admitir que o circuito é insensível ao diferencial da aplicação do relógio. Essa região, também chamada de região **isocrônica** ou **equicrônica**, tem um diâmetro

que pode ser calculado a partir de dados disponíveis na literatura. Temos então uma região útil para operações síncronas que varia entre 1 e 1,5mm para uma freqüência de operação de 500MHz[25] [26]. Recentes avanços na tecnologia de interconexão de circuitos como, por exemplo, a deposição de cobre, tendem a estender esses limites. Fica evidente, contudo, que para se obter um processamento eficiente em circuitos *wafer-size* é necessário descentralizar a distribuição do relógio e prover algum meio para sincronizar circuitos cujos relógios são independentes. Atualmente, as técnicas mais utilizadas empregam sincronismo por *PLL – Phase Locked Loop* [27].

Circuitos assíncronos não possuem um relógio global, sendo portanto imunes a esses problemas. Essa solução, contudo, traz novos problemas de síntese, pois a temporização global é substituída por mecanismos de temporização independentes, o que implica no uso de protocolos de comunicação para a transferência de dados entre módulos com temporização heterogênea.

2.4.6. Temporização Heterogênea

A interconexão de **subsistemas síncronos** apresenta uma série de problemas, pois as interfaces síncronas contém apenas dados, sendo a temporização inteiramente dependente do relógio. Mesmo quando a interconexão é feita entre subsistemas com um mesmo relógio, problemas relativos aos tempos de *setup* e de *hold* podem ocorrer. Para evitá-los é preciso conhecer minúcias da temporização da interface. Quando os subsistemas tem relógios diferentes o problema fica ainda pior, pois é necessário o uso de um sincronizador. Sincronizadores aumentam a latência da interface como forma de garantir uma determinada probabilidade de estabilidade, ou seja como forma de garantir um certo *MTBF* – Tempo Médio Entre Falhas [28].

No caso de **subsistemas assíncronos**, cada módulo é inteiramente definido com relação à sua interface, pois os sinais englobam dados e sincronismo. A interconexão de subsistemas é facilitada pois não é necessário conhecer detalhes de cada subsistema. Basta interligá-los.

2.5. Técnicas de Síntese para Sistemas Assíncronos

Algumas das dificuldades inerentes à síntese de sistemas assíncronos são analisadas nesta seção.

2.5.1. Protocolos de Comunicação

A comunicação entre circuitos assíncronos é feita através de uma seqüência de *handshake*. Um circuito faz um *request* de comunicação e aguarda até receber um *acknowledge* do circuito destino. As formas mais utilizadas para isso são os protocolos conhecidos como ***4-phase RZ*** (Figura 2.5) e ***2-phase NRZ*** (Figura 2.6).

Embora o protocolo de 2 fases seja mais eficiente do ponto de vista do consumo de potência, já que todas as transições são utilizadas para transferência de dados, sua implementação exige uma maior quantidade de lógica auxiliar, o que de certa forma diminui a vantagem inicial.

É possível utilizar ambos os protocolos em um mesmo projeto, embora em interfaces distintas. A conversão entre esses protocolos também é possível [29].

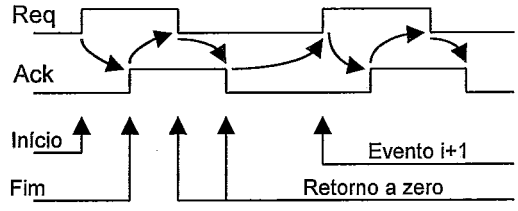


Figura 2.5: Protocolo de 4 fases

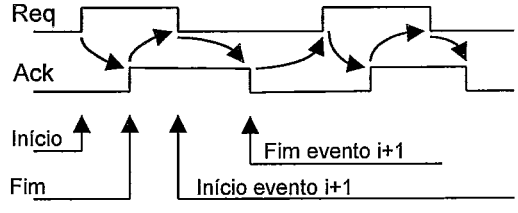


Figura 2.6: Protocolo de 2 fases

2.5.2. Transferência de Dados

Máquinas assíncronas necessitam de uma validação do sinal para promover uma transferência de dados. Os métodos usuais de temporização de dados incluem *bundled data* e *dual rail encoding*.

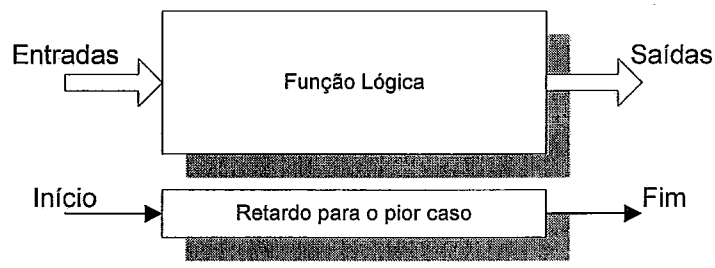


Figura 2.7: Single rail encoding

Bundled data encoding – Um único sinal é utilizado para sinalizar que os dados de saída são válidos. É o sistema mais simples, pois não apresenta redundâncias. É empregado principalmente para avaliar o retardo de uma mesma função lógica quando aplicada a um vetor de dados. Neste caso, faz-se uma avaliação do pior caso (por atraso limitado – *bounded delay*), mantendo-se uma acomodação para variações de ambiente. Um sistema típico pode ser visto na Figura 2.7.

Dual rail encoding – Emprega uma máquina seqüencial para codificar em dois bits o valor e o estado (válido ou não) de cada bit de dados. Por exemplo, o valor 00 pode indicar que a operação está em andamento; os valores 10 e 01 podem representar valores 0 e 1 estáveis, e 11 seria um valor impossível.

2.5.3. Fim de Operação

Circuitos assíncronos necessitam de uma forma de detecção do fim de operação. É esse sinal que vai alimentar o protocolo de comunicação.

Uma forma possível consiste no estudo do pior caso do tempo de propagação do sinal. Uma cadeia de inversores cujo retardo seja suficiente para garantir a validade do sinal de saída é então intercalada no circuito de controle. Embora esta forma seja similar à utilizada em circuitos síncronos, o âmbito é local e existe adaptação às variações do ambiente.

Circuitos aritméticos como somadores e multiplicadores podem detectar o fim de operação através análise da propagação do sinal de *carry* [30].

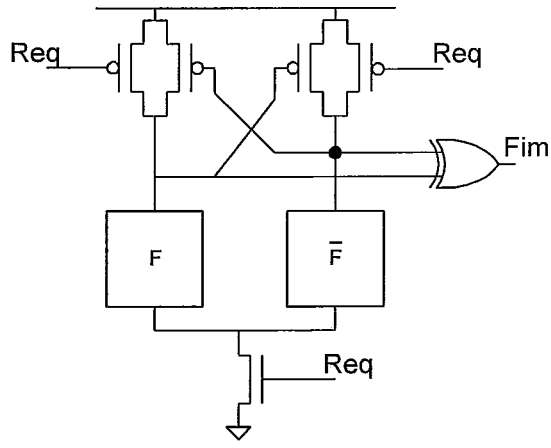


Figura 2.8: Lógica DCVSL

Valores complementares de funções lógicas podem ser utilizados juntamente com um ou-exclusivo para detectar o fim de operação, como por exemplo, na implementação de circuitos DCVSL – *Differential Cascode Voltage Swing Logic*, mostrada na Figura 2.8. Neste caso, o sinal Req em FALSO faz a pré-carga das saídas da função. Quando Req fica VERDADEIRO a função é calculada e seu valor é mantido pela configuração com realimentação [31].

Outro método, baseado na análise do consumo de corrente, também tem se mostrado promissor, conforme mencionado em [32].

2.5.4. Classes de Circuitos Assíncronos

Circuitos assíncronos podem ser classificados de acordo com o modelo utilizado para caracterizar a propagação do sinal. Um operador Δ, por exemplo, define um **atraso puro**, onde a onda propagada não sofre alteração. Um **atraso inercial** pode ser caracterizado por um operador δ, que define a largura mínima para a propagação de um pulso. Pode-se ainda considerar que a propagação de um sinal sofre um **atraso fixo** (*fixed delay*), um **atraso limitado** (*bounded delay*) ou ainda um **atraso ilimitado** (*unbounded delay*).

Circuitos síncronos são usualmente desenvolvidos através do conceito de atraso limitado.

Circuitos cujas portas e conexões são caracterizadas por atrasos ilimitados são chamados de circuitos insensíveis a atrasos ***(DI – delay-insensitive)***. Essa classe engloba apenas operadores lógicos simples, mas pode ser estendida para operadores

compostos por várias portas. Se a temporização interna for coerente, a operação externa será insensível a atrasos [33].

Uma outra extensão pode ser criada para circuitos *DI* quando consideramos que as derivações (*forks*) fisicamente próximas são isocrônicas. Temos então circuitos quase insensíveis a atrasos ***(QDI – quasi-delay-insensitive)***. Neste caso, as interfaces podem ser consideradas como insensíveis a atrasos [34].

Quando se considera que o tempo de propagação nas interconexões do circuito é negligível, mas que o tempo de propagação nas portas lógicas tem atraso ilimitado, temos circuitos ***SI – speed-independent***, isto é, circuitos que não são afetados pela velocidade dos componentes [35].

Um circuito é dito auto-temporizado ***(self-timed)*** quando ele é formado por elementos capazes de determinar seu próprio tempo de operação. Um circuito auto-temporizado deve estar inteiramente contido em uma única região equipotencial de tempo. A comunicação entre regiões, no entanto, é do tipo *DI* [26].

2.5.5. Hazards

Em circuitos síncronos as saídas são amostradas apenas durante o pulso do relógio. Eventuais valores transientes contendo *glitches* não são importantes. Em circuitos assíncronos não é possível distinguir um *glitch* de uma saída válida, e por isso o *glitch* deve ser evitado. Um ***hazard*** define o potencial de um circuito gerar *glitches* quando em operação.

Um *hazard* combinacional pode ser atenuado utilizando-se o modelo de *inertial-delay*. Quando se considera um modelo de atraso limitado, o *hazard* pode ser eliminado pela introdução de atrasos nos caminhos críticos. Existem métodos de síntese capazes de criar circuitos sem *hazards* [36].

Quando se considera circuitos combinacionais onde apenas uma variável muda a cada vez (***SIC***– *Single Input Change*), pode-se evitar *hazards* em circuitos AND-OR, através da inclusão de mintermos redundantes que cubram todas as transições 1→1 do mapa de Karnaugh [37].

Para mudanças simultâneas de múltiplas entradas (***MIC***– *Multiple Input Change*), pode acontecer que a própria definição do valor da função de saída não seja monotônica, isto é, pode ocorrer mais de uma transição no mapa de Karnaugh até que ela estabilize em um valor final. Diz-se que a função possui um ***function hazard***, e

este não pode ser eliminado [38]. Para funções que sejam ***function hazard free***, procura-se sintetizar um circuito sem *glitches*, ou seja, um circuito ***logic hazard free***.

Uma transição é dita estática quando o valor da saída permanece constante. Uma transição em circuitos AND–OR é dita dinâmica quando existe uma e somente uma transição 1→0 em todos os caminhos que levam do mintermo de partida ao mintermo destino. Um *hazard* lógico estático, que pode existir por exemplo em transições 1→1, é tratado de forma similar aos sistemas SIC, utilizando-se uma cobertura adequada dos mintermos. *Hazards* lógicos dinâmicos podem acontecer em circuitos AND–OR, em transições 1→0 ou 0→1. Um *hazard* lógico dinâmico específico pode ser eliminado se os produtos que interceptam essa transição incluírem o valor inicial das entradas ou o valor final [39]. Cabe notar que a eliminação simultânea de todos os *hazards* lógicos dinâmicos pode não ser possível. Em circuitos AND–OR, transições 0→0 não geram *hazards* [36].

Existe ainda a classe sem restrições nas entradas (***UIC***– *unrestricted input change*). No entanto, a dificuldade de síntese de sistemas desta classe faz com que ela seja pouco utilizada [39].

2.5.6. Máquinas de Estado

O método tradicional para síntese de máquinas de estado assíncronas foi proposto por Huffman, e baseia-se nos métodos de síntese para máquinas síncronas. Para um perfeito funcionamento, essas máquinas devem operar no **modo fundamental**, isto é, é preciso garantir que qualquer mudança de estado seja provocada pela variação de apenas uma única entrada, o que implica que entradas sucessivas somente podem ocorrer após a estabilização do circuito. Estas máquinas podem ser descritas como na Figura 2.9.

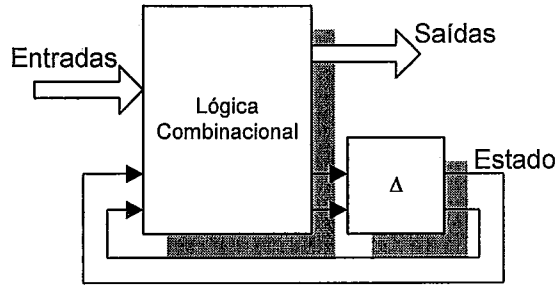


Figura 2.9: Máquina de Huffman assíncrona

Máquinas de Huffman podem ser sintetizadas para os modelos SIC, MIC ou UIC. Os *hazards* podem ser eliminados por síntese ou filtrados por *inertial delay*. Muitas vezes os *glitches* das saídas são ignorados, e apenas os que afetam a lógica de mudança de estado são tratados. Essa máquinas são chamadas de ***S-proper***. Devido à dificuldade de se sintetizar máquinas de Huffman corretas, outros modelos foram propostos.

Máquinas *self-synchronized* são similares às máquinas de Huffman, mas utilizam registradores para armazenar os estados. O relógio dos registradores é função das variações das entradas. Um diagrama de blocos pode ser visto na Figura 2.10.

Máquinas auto-sincronizadas funcionam em rajada *(burst-mode)* , o que permite a mudança simultânea de múltiplas entradas (*MIC*). Essas máquinas esperam a chegada de uma coleção de entradas e só então respondem com uma coleção de saídas (*burst*) [40]. Essas máquinas são *hazard-free*, e possibilitam alta performance.

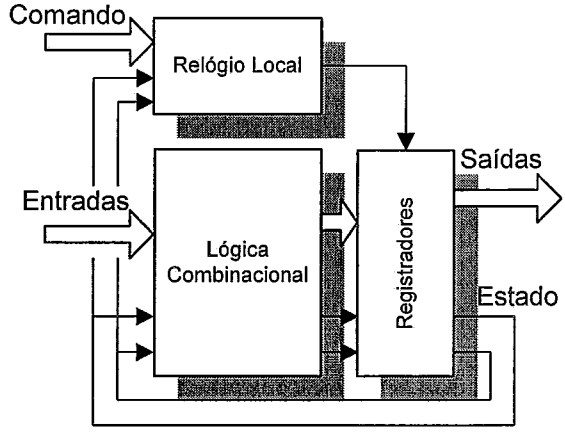


Figura 2.10: Máquina do tipo auto-sincronizada

2.5.7. Síntese por Rede de Petri

A síntese por Rede de Petri é utilizada como alternativa à idéia de uma máquina de estados tradicional. Ela utiliza o conceito de seqüência de eventos parcialmente ordenados. Um exemplo de rede de Petri pode ser visto na Figura 2.11.

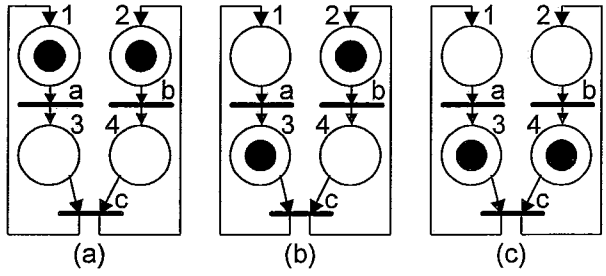


Figura 2.11: Representação por Rede de Petri

A rede é composta por lugares ***(places)*** e transições ***(transitions)***. Um lugar marcado por uma ficha ***(token)*** indica que uma determinada condição foi atingida. A transição que depende dessa ficha pode então ser iniciada. Vemos na figura que as transições a e b dependem da presença de fichas em 1 e 2 respectivamente. A transição c depende da presença de fichas tanto em 3 como em 4. Uma vez feita a transição, a ficha é enviada ao lugar seguinte.

A Rede de Petri pode ser mapeada diretamente em *hardware* como em [41], ou ser usada apenas como especificação comportamental. Uma variante restrita chamada STG – *Signal Transition Graphs* ou ainda Gráficos de Transição de Sinais, tem sido bastante utilizada por permitir a síntese de sistemas do tipo que independe da velocidade *(speed-independent)* [42]. Existem algoritmos otimizados para particionamento [43], minimização e *assignment* de estados [44].

2.6. Elementos de Síntese

Alguns dos principais elementos de composição de circuitos assíncronos serão vistos a seguir.

2.6.1. Sincronizadores e Arbitradores

O principal elemento sincronizador utilizado em circuitos assíncronos é o *flip-flop* C de Müller. Seu funcionamento para entradas a e b, e saída c, pode ser descrito como:

```
if (a==b) then c=a;
else c=c;
```

Ou seja, havendo coincidência nos valores das entradas, a saída acompanha esse valor, caso contrário ela mantém o estado anterior. Uma implementação possível para o *flip-flop* C pode ser vista na Figura 2.12 [31].

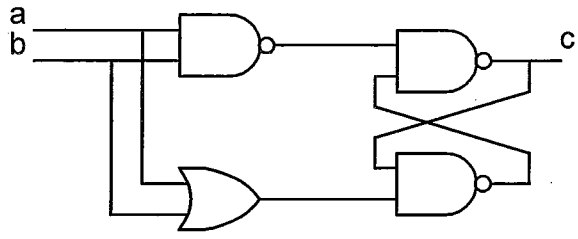


Figura 2.12: Estrutura de um flip-flop C de Müller.

Um arbitrador é um circuito que provê duas saídas, e é utilizado para determinar a ordem de chegada dos sinais de entrada. Em um determinado instante apenas uma saída pode estar ativa, e um pedido pendente só pode ser atendido após o término da atividade em andamento. Um exemplo de arbitrador pode ser visto na Figura 2.13.

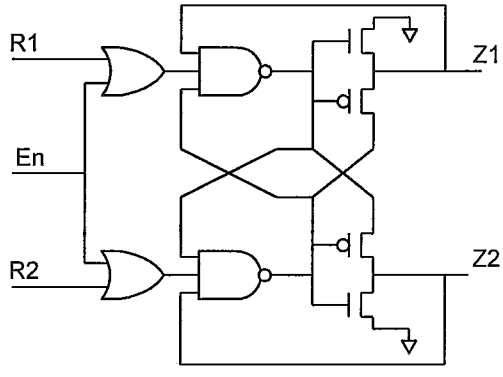


Figura 2.13: Exemplo de um arbitrador com enable.

2.6.2. Pipelines

A *pipeline* é o elemento básico para a elaboração de circuitos complexos. Os principais tipos de interesse para circuitos assíncronos estão descritos a seguir.

2.6.2.1. Pipeline Síncrona

A *pipeline* síncrona clássica é sempre vista como um elemento de referência para comparar o desempenho de pipelines assíncronas.

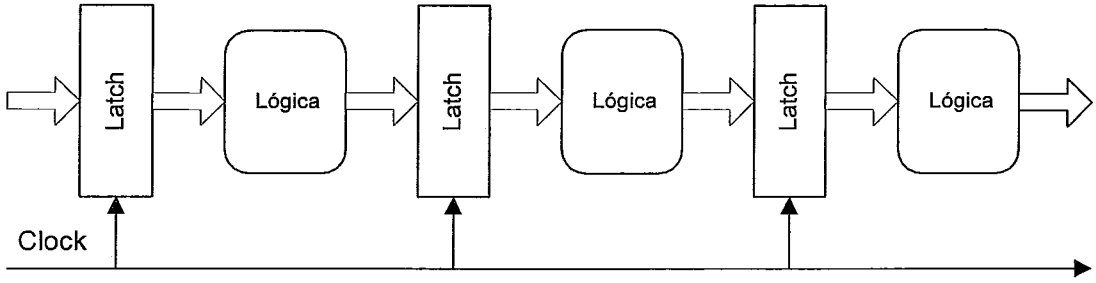


Figura 2.14: Pipeline síncrona

Este tipo de *pipeline* segue o modelo de atraso limitado. Todos os atrasos introduzidos pela lógica combinacional, bem como os tempos de *setup* e *hold* dos elementos de memorização devem ser calculados. O período do relógio fica sujeito ao pior caso de todos os estágios, sendo considerada uma margem que garanta o espalhamento de parâmetros na fabricação e o pior caso da variação de ambiente [45].

2.6.2.2. Pipeline Assíncrona Auto-temporizada

Pipelines assíncronas são elásticas, isto é, para o seu funcionamento normal não é necessário que todos os estágios contenham dados. O funcionamento dessas *pipelines* é semelhante ao de um *FIFO*. Dados recebidos na entrada progridem para a saída sempre que um estágio esteja disponível. Caso a operação prevista não necessite utilizar todos os estágios do *pipe*, ela pode ser descartada no interior do *pipe* tão logo as operações necessárias sejam completadas [45].

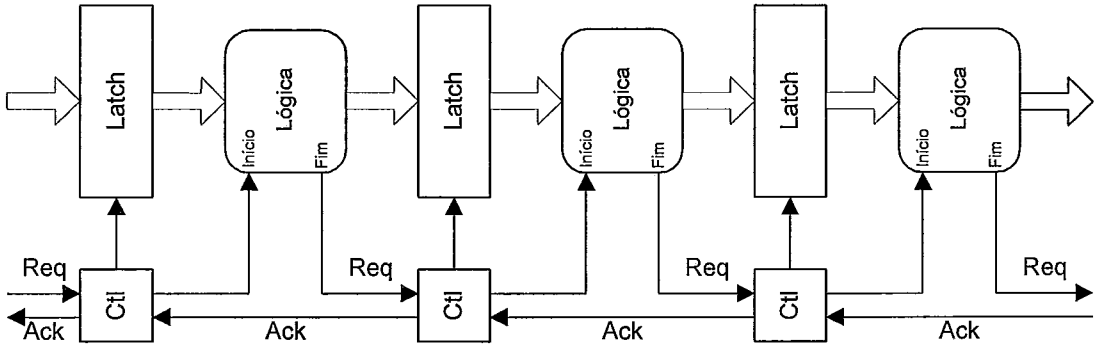


Figura 2.15: Pipeline assíncrona auto-temporizada

A *pipeline* assíncrona aqui mostrada utiliza protocolo DI para habilitar a progressão de dados entre estágios. A duração de cada operação é determinada localmente.

2.6.2.3. Micropipelines

Micropipelines são uma criação de Ivan Sutherland, e foram apresentadas na palestra de entrega do Turing Award de 1988. A maior parte dos projetos atuais de pipelines deriva deste trabalho pioneiro, cujo diagrama de blocos encontra-se na Figura 2.16 [46].

Micropipelines combinam uma interface entre estágios do tipo *2-phase* DI com uma avaliação da lógica combinacional feita por *bundled data*, o que implica em projeto do tipo atraso limitado [47]. O projeto lógico fica simplificado pela utilização das ferramentas disponíveis para circuitos síncronos e o atraso do estágio, embora calculado para o pior caso, tem determinação local e adaptação às condições de ambiente [48].

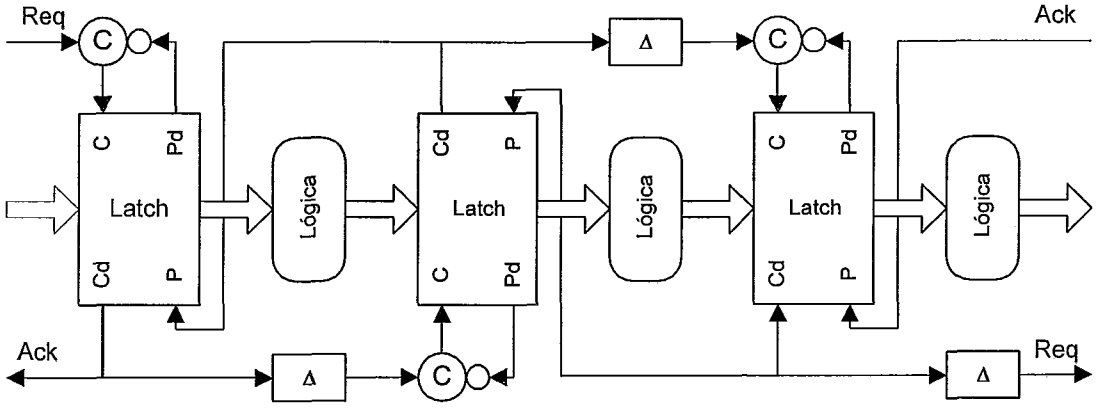


Figura 2.16: Micropipeline

A recepção de dados é feita após a habilitação de um *request (Req)*, o que provoca a captura do vetor de entrada. O *acknowledge (Ack)* é fornecido através do sinal *capture done (Cd)*. Ele é encaminhado ao estágio anterior, onde habilita o modo transparente do *latch* (*pass mode P*) e posteriormente vai habilitar uma nova captura. Após um retardo conveniente esse mesmo sinal fará um *request* para o próximo estágio.

2.6.3. Somadores

As alternativas para o projeto de somadores são oriundas de trabalhos originalmente orientados para aplicações em sistemas síncronos. Por esta razão, nos estudos de desempenho existe uma grande preocupação com a latência do pior caso, já que esse é o fator fundamental que limita o desempenho em sistemas síncronos.

Aliás, é interessante mencionar que é usual chamarmos de **somador assíncrono** a qualquer somador projetado para ser inserido em uma máquina assíncrona. Por extensão, poderíamos também utilizar o termo **operador assíncrono** mesmo para circuitos combinacionais. **Operador síncrono** seria então qualquer operador projetado para funcionar em uma máquina síncrona [7] [92] [97] [99].

Com o desenvolvimento de sistemas assíncronos, o tempo médio de execução passou a ser mais relevante que o tempo de pior caso. Mesmo assim, ainda existem poucos estudos a respeito do desempenho de somadores assíncronos, da complexidade de integração dos circuitos e das técnicas para detecção de fim de operação [49].

2.6.3.1. Propriedades Gerais de Somadores

Os somadores podem ser classificados quanto à sua estrutura em:

- Somadores de estrutura serial, como o *Ripple Carry Adder*
- Somadores com estrutura em árvore, como o *Conditional Sum Adder.*
- Somadores de estrutura híbrida, como o *Carry Select Adder.*

É fácil entender que um somador serial tem complexidade máxima de tempo de $O(n)$, onde n corresponde à largura dos operandos de entrada. No entanto, não é tão evidente que a complexidade média em tempo do somador serial é de $O(\log n)$. Isso acontece porque, para operandos randômicos, podemos esperar que o comprimento da maior cadeia de *carry* seja $\log n$. Um somador serial tem estrutura simples, repetida para cada bit, o que faz com que a complexidade de integração seja de $O(n)$ [49].

Os somadores rápidos utilizados em sistemas síncronos têm estrutura baseada em árvore. Dessa forma é possível conseguir uma complexidade de tempo de $O(\log n)$, o que corresponde à altura da árvore. Isso no entanto, implica no uso de

$(n\log n)-1$ elementos somadores, o que corresponde a uma complexidade de integração de $O(n\log n)$.

Outra possibilidade corresponde ao uso de uma estrutura híbrida, onde o somador é dividido em blocos. Cada bloco calcula em paralelo o valor do *carry* que vai ser enviado ao bloco seguinte. Esse tipo de somador tem complexidade máxima de tempo de $O(\sqrt{n})$ e complexidade de integração de $O(n)$. Deve-se procurar minimizar o tempo de cálculo do somador em função tanto da quantidade de blocos quanto da complexidade de cada um dos blocos. O desempenho médio fica um pouco prejudicado já que $O(\sqrt{n})$ tende a ser superior a $O(\log n)$. A redução no tempo de pior caso corresponde a uma constante que depende da implementação. Isto porém não afeta a determinação da complexidade máxima de tempo, que continua sendo $O(n)$ [50].

Um resumo dessa avaliação pode ser visto na Tabela 2.1.

Tabela 2.1: Comparação de estruturas de somadores

Somadores:	Complexidade de integração	Complexidade média de tempo	Complexidade máxima de tempo
Seriais	$O(n)$	$O(\log n)$	$O(n)$
Árvore	$O(n\log n)$	$O(\log n)$	$O(\log n)$
Híbridos	$O(n)$	$O(\sqrt{n})$	$O(n)$

Vemos então que a ocupação de área necessária para a implementação de somadores pode ser considerada pequena para somadores seriais, média para somadores híbridos e alta para somadores em árvore [49].

2.6.3.2. Principais Tipos de Somadores

Ripple Carry Adder (RCA) e Carry Completion Adder (CCA) – O processo da adição de n bits pode ser feito através do mapeamento de um algoritmo recursivo/iterativo em *hardware*, como por exemplo, nos somadores seriais do tipo *RCA*. A geração de *carry* é função dos operandos de entrada, aqui chamados de A e B, e do valor do *carry* de entrada.

$a_i = b_i = 0 \quad \rightarrow carry = 0$ (*carry cease*).

$a_i = b_i = 1 \quad \rightarrow carry = 1$ (*carry generate*).

$a_i \neq b_i \quad \rightarrow carry =$ *carry* anterior (*carry propagate*).

Para diminuir a latência do somador *RCA* é possível modificá-lo ligeiramente, acrescentando circuitos para separar a cadeia de soma em blocos independentes, de menor comprimento, de acordo com o tipo de *carry* previsto para cada bit. Neste caso temos então um somador *Carry Completion Adder* (*CCA*), que é mais adequado para sistemas assíncronos, já que apresenta tempo de operação variável.

A complexidade de tempo de pior caso do somador *RCA* é $O(n)$, o que torna este tipo de somador inadequado para sistemas síncronos de alta performance. Porém, utilizando-se as técnicas especificadas acima para somadores *CCA*, a cadeia média de *carry* para operandos aleatórios, e conseqüentemente, o tempo médio de execução fica reduzido a $\log n$ [51].

Somadores *CCA* são usualmente chamados de *RCA* quando tratamos de circuitos assíncronos.

Conditional Sum Adder (CSA) – Este é um somador de estrutura em árvore onde a soma e o *carry* são calculados simultaneamente para ambos os valores possíveis para o *carry* de entrada. O valor correto da soma e do *carry* é selecionado pelo carry de entrada e os resultados são encaminhados ao nível seguinte. Novamente o carry é utilizado para selecionar o valor correto de pares de bits, e assim por diante. O resultado é obtido exatamente em $O(\log n)$, o que torna este somador interessante para sistemas síncronos. Em contrapartida a complexidade de integração é alta. Para n = 32 bits, o custo de integração corresponde a 5 vezes o de um somador *CCA/RCA*. Como os blocos do somador são bastante complexos e existe um custo adicional para a interligação dos blocos, o custo final pode ficar entre 7 e 10 vezes o do somador *CCA/RCA* [49] [52].

Carry Lookahead Adder (CLA) – Este é um somador de estrutura híbrida, no qual o carry é calculado por uma estrutura em árvore e a soma por somadores seriais. Os sinais *Propagate* e *Generate* são calculados em paralelo para um determinado bloco. A soma dentro de cada bloco é feita por somador *RCA*. Para que um somador *CLA*

seja eficiente, o tempo de cálculo da árvore de *carry* deve ser menor que o tempo de propagação do sinal de carry no interior do bloco. Na prática, o tamanho mínimo de um bloco deve ser 4, e para somadores com mais de 16 bits são necessários pelo menos dois níveis de *CLA*. Cada nível adicional de *CLA* dobra o alcance efetivo da previsão de *carry*. No entanto, o tempo necessário para o cálculo de um nível adicional faz com que os ganhos produzidos pela adição de mais um nível sejam progressivamente menores [53].

Carry Skip Adder (SKP) – Como a função *Propagate* é mais simples do que a *Generate*, este somador calcula *Propagate* como função booleana das entradas e *Generate* por *carry rippling*, assumindo um valor de zero para o *carry* de entrada em cada bloco. A complexidade de integração fica entre a do *CCA* e do *CSA* [54].

Carry Select Adder (SEL) – Outro somador híbrido. Os sinais *Propagate* e *Generate* são calculados por *carry rippling*. *Propagate* assume *carry in* = 1 e *Generate* assume *carry in* = 0. Os resultados parciais são encaminhados a um somador do tipo *CSA*. A complexidade de integração é semelhante à do *CSA* [55].

O desempenho médio de todos os somadores híbridos considerados é semelhante, e chega a ser 19% melhor do que o de um somador *CCA*. Este cálculo, feito em [49], considera apenas somadores *CCA* que implementam a detecção da propagação do sinal *carry in* = 1. Um somador *CCA* que também empregue a detecção de *carry in* = 0 terá forçosamente um desempenho melhor.

A maior vantagem dos somadores híbridos consiste na obtenção de um desvio padrão dos tempos das somas menor que o do *CCA*, o que torna o *throughput* desses somadores mais homogêneo. Usualmente o *SEL* é considerado uma variante do somador *CSA*, sendo também denominado na literatura por *CSA*.

Carry Save Adder (CSvA) e Carry Delayed Adder (CDA) – É interessante mencionar a existência de somadores com complexidade no tempo de $O(1)$. O somador *CSA* opera um conjunto de 3 vetores de entrada A, B e C, e produz como resultado os vetores S e C' tais que A+B+C=S+C'. O somador *CDA* usa como primeiro nível um somador *CSA* e calcula os vetores T e D tais que A+B+C=T+D.

Obviamente esses somadores não produzem um resultado em complemento a 2, e por isso não podem ser utilizados quando se necessita do valor final da soma. Mas

o uso de vetores com o valor parcial de uma soma tem aplicações importantes em operações encadeadas como a multiplicação [56].

2.7. Teste de Circuitos Lógicos Digitais

O custo do reparo em circuitos eletrônicos cresce com o nível de integração do produto. A simples eliminação de circuitos integrados defeituosos antes mesmo de serem encapsulados é muito mais econômica do que o eventual reparo do produto final. É por esse motivo que os fabricantes de circuitos integrados utilizam procedimentos de teste capazes de identificar defeitos nas pastilhas ainda presas ao *wafer*.

No entanto, a miniaturização e a complexidade dos integrados atuais tornam essa tarefa bastante difícil. Para facilitar o procedimento de teste, circuitos modernos costumam incluir estruturas especiais como veremos a seguir [57] [58].

2.7.1. Terminologia de Teste

A terminologia e a descrição dos conceitos adotados para esta seção seguem, em grande parte, as definições apresentadas em [59].

As entradas e saídas do circuito a ser testado (***CUT*** – *Circuit Under Test*) são chamadas de **entradas** e **saídas primárias**.

Controlabilidade de um circuito é a propriedade que indica a facilidade de se definir o valor de qualquer nó do circuito através de valores colocados nas entradas primárias. **Observabilidade** refere-se à facilidade de se propagar o valor de qualquer nó do circuito até as saídas primárias. **Testabilidade** denota a facilidade com que se realizam testes em um circuito. Um circuito de alta testabilidade normalmente possui alta controlabilidade e alta observabilidade.

Considera-se que um circuito apresenta **erro** (*error*) quando a sua operação produz, nas saídas primárias, valores diferentes daqueles especificados. Uma **falha** (*fault*) de fabricação pode produzir, ou não, um circuito com erro. A **detecção de falhas** (*fault detection*) é feita pela aplicação ao circuito de uma seqüência de valores de entrada, os **vetores de teste** (*test vectors*), e pela observação das saídas primárias. Qualquer diferença entre os resultados obtidos e os esperados implica na existência de uma falha.

Um procedimento de **teste** (*test*) detecta uma certa quantidade de falhas, determinada pela **cobertura prevista de falhas** (*fault coverage*). O **tamanho do teste** refere-se à quantidade de vetores de teste, e a aplicação do procedimento implica em um **tempo de teste**.

Um procedimento de teste é desenvolvido para um determinado **modelo de falhas** (*fault model*). Os modelos usuais consideram uma única falha, ao nível das portas lógicas, onde uma entrada ou saída fica permanentemente presa a um determinado nível lógico. Este modelo é chamado de ***stuck-at fault model***. Uma simplificação pode ser feita ao se considerar que apenas as saídas ficam presas. Temos então um ***output stuck-at fault model***.

É sempre importante enfatizar que existe uma diferença entre diagnóstico e teste. O resultado de um teste provém de uma decisão binária. Se o circuito atende às especificações previstas no procedimento de teste, dizemos que ele **passou no teste**. O procedimento que visa identificar os motivos que produziram um funcionamento inadequado do circuito chama-se **diagnóstico**. O escopo desta tese não abrange o diagnóstico de circuitos.

2.7.2. DFT – Projeto Visando a Testabilidade

Os conceitos e o modo de implementação de testes em CIs foram inicialmente desenvolvidos para circuitos síncronos e aos poucos vem sendo estendidos para circuitos assíncronos. Nesta seção fazemos uma revisão do conceito de **DFT,** sigla em inglês para projeto visando a testabilidade, sob o ponto de vista da sua utilização em circuitos síncronos. Nas seções seguintes apresentamos as principais diferenças existentes na implementação de técnicas de teste em circuitos assíncronos.

Vimos que o procedimento de teste consiste em se aplicar uma seqüência de vetores de teste a um circuito, e em seguida pela avaliação dos resultados. Para aumentar a cobertura de falhas pode ser necessário aumentar a controlabilidade e a observabilidade de um circuito. Para isso introduzimos pontos de teste no interior desse circuito. Novas entradas e saídas primárias ficam então disponíveis ao projetista. Caso a quantidade de pontos de teste seja pequena, esta técnica é bastante eficiente. Para os casos em que se necessita de uma grande quantidade de pontos de teste é necessário organizá-los de alguma forma para que o *overhead* não inviabilize o projeto [60]. Existem duas técnicas principais para aumentar a testabilidade de CIs, a saber o *scan test* e o BIST – *Built in Self Test*.

Para implementação do ***scan test*** os registradores do circuito são construídos de tal forma que além da operação normal, eles também possam operar no modo teste como uma série de registradores de deslocamento interligados por um caminho, o *scan-path* [61].

A operação em modo teste consiste na introdução de um vetor de teste através de uma entrada de dados em série (*scan-in*) de modo a colocar o circuito no estado desejado. A seguir o circuito opera normalmente por um único pulso de relógio. As saídas amostradas são então encaminhadas a uma saída primária (*scan-out*). Um analisador verifica se a resposta ao vetor de teste foi adequada. Um esquema simplificado da estrutura de scan pode ser visto na Figura 2.17.

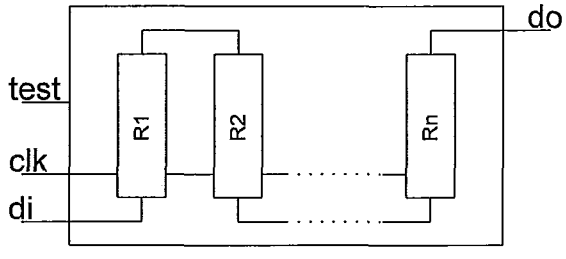


Figura 2.17: Estrutura para scan-test

Como toda entrada e saída de dados é feita de forma serial, o procedimento de teste pode se tornar bastante lento. Várias técnicas existem para minimizar o tempo de teste, seja pelo recobrimento parcial dos dados de entrada e saída ou ainda pela utilização de estruturas internas série-paralelas.

Uma padronização da técnica de *scan* foi adotada pela indústria, gerando a norma IEEE 1149.1 [62] que define uma estrutura chamada *boundary scan*. O *boundary scan* foi concebido visando o teste de placas eletrônicas. A interface é composta por um sinal de seleção do modo de teste, um sinal de relógio e dois sinais de dados, *test data-in* e *test data-out*. Opcionalmente pode existir ainda um sinal de *reset*.

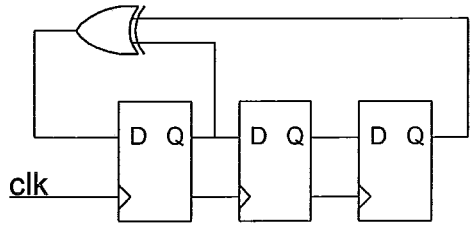


Figura 2.18: Estrutura de um TPG com LFSR.

Para agilizar o tempo de teste é possível utilizar a técnica ***BIST***, onde a geração de vetores de teste e a análise dos resultados é feita no próprio CI [61].

A geração dos vetores de teste é feita por um ***TPG*** – *Test Pattern Generator*, como por exemplo, um registrador de deslocamento linearmente realimentado ***(LFSR*** – *Linear Feedback Shift Register)*. Um exemplo de *TPG* com *LFSR* pode ser visto na Figura 2.18. O *LFSR* é uma estrutura capaz de gerar seqüências pseudo-aleatórias, sendo possível, através da escolha adequada do polinômio de realimentação, obter até mesmo uma seqüência de 2^n-1 vetores, onde *n* é a quantidade de *flip-flops* da estrutura [63].

Cada saída do *LFSR* é aplicada a uma entrada do circuito sob teste, o ***CUT*** – *Circuit Under Test.* A saída do *CUT* é encaminhada a um analisador de respostas, o **TRA** – *Test Response Analyzer* , que pode ser implementado por outro *LFSR*. Neste caso, este se comporta como um analisador de assinaturas, o ***SA*** – *Signature Analyzer.* O *SA* é projetado para fazer uma compressão de dados, sendo que o valor final obtido é a assinatura do circuito. Se o valor da assinatura coincidir com o valor previsto considera-se que o circuito passou no teste. Uma possível implementação para o *SA* pode ser vista na Figura 2.19.

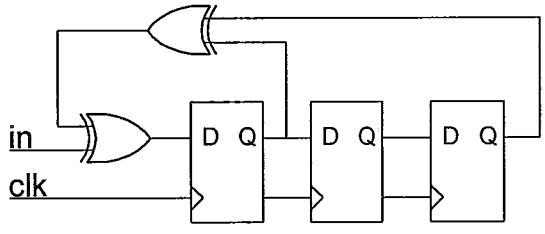


Figura 2.19: Estrutura de um SA com LFSR.

2.7.3. Teste de Circuitos Assíncronos

Aqui apresentamos os principais problemas relativos ao teste de circuitos assíncronos e enfatizamos as diferenças existentes para o teste de circuitos síncronos convencionais. O trabalho desta seção segue principalmente as diretrizes propostas em [64].

O teste de circuitos assíncronos difere dos testes para circuitos síncronos por vários motivos. A ausência de um relógio global nos circuitos assíncronos tem um impacto negativo sobre a controlabilidade do sistema, já que a técnica de verificação passo a passo não pode ser empregada. Circuitos assíncronos normalmente utilizam mais elementos de memorização que os circuitos síncronos correspondentes, o que implica em maior quantidade de vetores de teste e maior impacto na área utilizada para implementar o circuito de teste, caso este seja necessário. Além disso, as falhas podem introduzir *hazards* ou corridas *(races),* que são notoriamente difíceis de detectar.

No entanto, como os circuitos assíncronos fazem sincronismo através de *handshake*, qualquer falha na estrutura de controle faz com que o circuito pare, o que é observável.

2.7.3.1. Circuitos Auto-verificadores

Como não existe um relógio global, o acoplamento entre etapas deve ser feito ou por handshake ou pelo estudo cuidadoso dos atrasos da propagação do sinal. Esta última forma, utilizada na síntese clássica de máquinas assíncronas torna-se impraticável para circuitos complexos.

A utilização de circuitos redundantes para a eliminação de *hazards* torna impossível obter uma cobertura de falhas de 100%, e ainda pode introduzir corridas *(races)* além de outros problemas.

Circuitos que utilizam *handshake* são simples e extremamente robustos. Eles produzem excelente observabilidade, já que qualquer falha faz com que todo o circuito pare. Para estes circuitos o teste consiste em se aplicar um *request* e aguardar o *acknowledge* durante um tempo τ, que é função da especificação do circuito e da tecnologia de fabricação. Cuidados especiais devem ser tomados em presença de circuitos arbitradores, que podem permanecer em situação meta-estável por um intervalo de tempo indeterminado [65].

Circuitos que, em presença de uma falha, param após um intervalo de tempo determinado são chamados de auto verificadores *(self-checking)* [25]. Essa definição difere da utilizada em circuitos síncronos, onde um circuito verificador específico como o verificador de paridade, por exemplo, determina a existência de um código não válido e sinaliza o erro.

Circuitos do tipo *DI* operam corretamente para quaisquer valores de atrasos das portas e das conexões. Para isso todas as transições de uma saída devem ser referendadas por todas as portas de entrada antes que se possa alterar o valor dessa saída (*acknowledgement property*). Neste caso, qualquer falha *stuck-at* em qualquer entrada impede o prosseguimento das operações e o circuito é auto-verificador.

Circuitos *SI* consideram que o tempo de propagação nas interconexões do circuito é negligível, e que o tempo de propagação nas portas lógicas pode ter qualquer valor. Neste caso basta que o sinal de saída seja referendado por uma única porta de entrada. O circuito ainda é auto-verificador para o modelo *output stuck-at fault*. É interessante notar que grande parte dos circuitos *SI* são também auto-verificadores mesmo para o modelo *input stuck-at fault* [66].

Circuitos *QDI* consideram que apenas as derivações (*forks*) fisicamente próximas são isocrônicas. As interfaces podem ser consideradas como insensíveis a atrasos. Foi desenvolvido um modelo de falhas especial que considera *input stuck-at faults* para derivações não isocrônicas e *output stuck-at faults* para derivações isocrônicas. Para o modelo considerado, esses circuitos são auto-verificadores.

2.7.3.2. DFT em Circuitos Assíncronos

Assim como nos circuitos síncronos, pode-se, por exemplo, conectar todos os elementos de memorização em um ou mais registradores de deslocamento (*shift-registers*). A controlabilidade pode ser feita pela introdução de um vetor de teste (*scan-in*) e a observabilidade é feita pela leitura do vetor de saída (*scan-out*). A implementação do *scan-path* pode ser feita tanto em modo síncrono como em modo assíncrono. Os compromissos necessários para minimizar o tempo de teste também são semelhantes aos dos circuitos síncronos, sendo no entanto necessário utilizar um relógio de teste para acionar os registradores de scan e circuitos *dual-rail encoded* [67].

Uma técnica para a implementação de sistemas BIST assíncronos será apresentada no desenvolvimento desta tese.

2.7.3.3. Falhas Devidas a Atrasos

Um modelo de falhas chamado *path delay fault model* pode testar a existência de falhas em um caminho π. Para isso é necessário conhecer todos os atrasos envolvidos na propagação do sinal. Aplica-se então dois vetores (V1, t0) e (V2, t1). V2 é escolhido de modo a provocar transições em todos os nós de π. O intervalo entre t1 e t0 deve ser suficiente para garantir que o circuito atinja o estado quiescente. A saída é lida em t2, sendo a diferença entre t2 e t1 o atraso máximo esperado para o caminho. A leitura de um valor diferente do esperado implica em existência de falha.

Um **teste** é dito **robusto** quando ele independe dos atrasos em portas que não pertencem ao caminho considerado. Existem técnicas para a geração de vetores de teste robustos [68].

Para que um circuito seja completamente testável com relação a atrasos basta utilizar um *scan-path*. Os registradores de scan devem armazenar ambos os vetores V1 e V2. O *scan-path* precisa ser síncrono e o *clock skew* deve ser cuidadosamente considerado.

Os problemas na aplicação do teste de atraso incluem a possível inexistência de testes robustos, a geração de *hazards* e corridas pela introdução do circuito extra necessário à implementação do teste, além do alto custo, em termos de área, da geração de novos pontos de teste. Isso tudo sem considerar o próprio tempo de teste.

2.8. Criptografia

Embora esta tese não seja sobre criptografia, no procedimento de validação da metodologia de conversão da síntese síncrona para a assíncrona utilizamos, para estudo de caso, um *soft-core* para criptografia desenvolvido no LPC – Laboratório de Projeto de Circuitos Integrados da COPPE/UFRJ. Os dados a seguir foram obtidos principalmente das teses citadas nas referências [69] e [70].

2.8.1. Introdução

O procedimento de se manter o segredo de uma mensagem através do bloqueio do acesso ao meio físico é frágil por natureza. A criptografia surgiu da necessidade de se manter o segredo de mensagens que eventualmente poderiam ser interceptadas. Para isso, a mensagem original deveria sofrer uma transformação que a tornasse ininteligível para outros que não o destinatário.

A mensagem convenientemente cifrada não necessita de um canal seguro para transmissão. Ela pode até mesmo ser publicamente disseminada, pois apenas o destinatário deverá ter conhecimento suficiente para decifrá-la em tempo hábil. Isso porque as informações também envelhecem e podem deixar de constituir segredo após algum tempo.

2.8.2. Métodos Utilizados para Cifragem

A cifragem baseia-se na aplicação de um algoritmo ao corpo da mensagem. A decifragem utiliza um outro algoritmo para a reconstituição.

A cifragem exclusivamente algorítmica é considerada insegura, já que o criador do algoritmo pode decodificar mensagens de terceiros. Métodos que incluem uma chave de codificação são mais seguros. Para decodificar a mensagem é preciso conhecer o algoritmo e a chave.

Quando a mesma chave é utilizada na cifragem e na reconstituição da mensagem o algoritmo de cifragem é dito simétrico ou de chave privada. Neste caso deve haver uma senha para cada par de endereços e a segurança do sistema depende da distribuição da chave através de um canal seguro [71].

Para aumentar a segurança pode-se utilizar um sistema assimétrico. A chave de codificação pode então ser tornada pública. A chave de decodificação, no entanto, é de conhecimento apenas do destinatário da mensagem. Uma característica interessante dos algoritmos assimétricos é a possibilidade de autenticação das mensagens. Isto pode ser feito utilizando-se a chave privada para cifrar e a pública para decifrar.

2.8.3. Definições

Algumas definições necessárias à descrição dos algoritmos criptográficos mencionados nesta tese são apresentadas a seguir. Um glossário de termos de criptografia pode ser obtido em [72] e uma introdução à criptografia pode ser obtida em [73].

Os algoritmos aqui apresentados destinam-se a trabalhar com blocos de dados de tamanho fixo. Métodos adequados são utilizados para o tratamento de dados cujo tamanho não seja múltiplo do tamanho do bloco. Existem outros algoritmos que são adequados à cifragem de um fluxo de dados, mas não são considerados nesta tese.

O tratamento usual consiste em se dividir o bloco de dados em duas ou mais partes e aplicar um conjunto de operações básicas a cada uma das partes do bloco. Essas operações são tipicamente a adição, a multiplicação e o ou-exclusivo. Os resultados costumam ser recombinados através da permuta de um ou mais bits do bloco.

Uma seqüência de operações que produz uma transformação única no algoritmo é chamada de **camada** *(layer)*. Quando aplicada de forma repetida, cada aplicação é chamada de ***round***. Tipicamente um algoritmo vale-se de *rounds* para a cifragem e camadas para a difusão e confusão de dados redundantes.

Muitos algoritmos de criptografia utilizam a **Rede de Feistel** [71] em uma seqüência de vários rounds para a geração das cifras. Uma rede de Feistel pode ser criada pela divisão do bloco de entrada em duas partes, A e B. Utiliza-se então uma função f, definida de acordo com o algoritmo criptográfico desejado. A cada round aplica-se também uma chave k.

O primeiro round consiste em fazer $B' = f(A,k) \oplus B$. No segundo round faz-se $A'' = f(B',k') \oplus A$. Os rounds continuam até que se esgotem as chaves. Como cada round da rede pode ser desfeito por repetição devido ao operador $\oplus$, f não precisa ser uma função inversível. Dessa forma, o procedimento para decifrar o código é o mesmo utilizado na cifragem, bastando inverter a ordem das chaves.

2.8.4. Algoritmos Criptográficos

Os algoritmos de chave simétrica mais conhecidos são o *DES – Data Encryption Standard* e o *IDEA – International Data Encryption Algorithm.*

O *DES* [74], que foi desenvolvido pela IBM e adotado como padrão nos Estados Unidos em 1977, utiliza uma rede de Feistel em 16 rounds para cifrar blocos de 64 bits, usando para isso uma chave de 56 bits mais 8 bits de paridade.

O *IDEA* [69] cifra blocos de 64 bits utilizando uma chave de 128 bits. As operações são feitas em blocos de 16 bits, e necessitam de 8 rounds.

Recentemente, o *Blowfish* [75] vem sendo utilizado em sistemas que exigem alto desempenho. Este algoritmo codifica blocos de 64 bits com chaves que podem ter até 448 bits. A codificação é usualmente efetuada por uma Rede de Feistel, usualmente em 16 rounds. A performance do algoritmo torna possível a sua aplicação em Redes Locais.

O algoritmo de chave assimétrica mais utilizado atualmente é o RSA [76], iniciais de Rivest, Shamir e Adleman, seus autores. O tamanho das chaves pode ser escolhido de forma a garantir a segurança do sistema. A determinação da chave através de métodos matemáticos depende da fatoração de números primos muito grandes, e não se conhecem algoritmos rápidos para isso. Existem trabalhos publicados onde o tamanho da chave pode chegar a 2048 bits. A segurança do sistema é garantida pelo Teorema dos Números Primos, de Euclides. Pelo teorema, a quantidade p de números primos menores que um inteiro n tende assintoticamente para $n/\ln n$. Então para uma chave de 512 bits, $n=2^{512}-1$, e $p \cong 10^{150}$ [73].

A maior desvantagem dos algoritmos assimétricos consiste no baixo desempenho desses algoritmos quando comparados aos simétricos. No entanto, uma combinação das duas metodologias pode proporcionar alta segurança e alto desempenho. Para isto podemos empregar um algoritmo assimétrico para divulgação da chave e um outro, simétrico, para operação normal.

2.8.5. Criptoanálise

A criptoanálise procura decifrar uma mensagem sem prévio conhecimento dos métodos de codificação ou das chaves. A tentativa de criptoanálise denomina-se **ataque**, e a segurança do sistema depende da capacidade demonstrada pelo método para resistir a ataques feitos por especialistas, os criptoanalistas.

Os ataques de base matemática podem ser feitos a partir de um único texto cifrado, de um conjunto de mensagens originais e cifradas, ou ainda de um conjunto de mensagens originais fornecidas pelo criptoanalista e cifradas pelo sistema.

Outro tipo de ataque, chamado de ataque por força bruta, procura exaurir todo o conjunto possível de chaves. Neste caso, quanto maior for a quantidade de chaves do algoritmo, tanto menor será a probabilidade de sucesso do ataque em um tempo predeterminado.

Os ataques podem ainda se servir de quaisquer informações públicas existentes, aplicando métodos heurísticos baseados na combinação dessas informações para a reconstituição da chave original [77].

Dentre os algoritmos mencionados, sabe-se que o *DES* tornou-se suscetível a ataques de criptoanálise [78] face à evolução dos sistemas computacionais e dos métodos de análise. Brevemente ele deverá ser substituído como padrão nos Estados

Unidos. Até que isso aconteça, o procedimento recomendado consiste na codificação do documento com três chaves distintas (*Triple DES*). Recentemente foi demonstrado que o RSA, que é o algoritmo recomendado para os protocolos S/MIME, SSL e S/WAN, utilizados por exemplo, para operações seguras na Internet, também é vulnerável a ataques quando utilizado com chave de 512 bits [73].

É importante lembrar que a segurança de um sistema não pode ser garantida apenas pela qualidade da criptografia utilizada. Um operador descuidado pode ser induzido a fornecer sua chave para um sistema aparentemente seguro. Se o operador utilizar a mesma chave para o acesso a vários sistemas, todos eles ficarão com a segurança comprometida. Também outros métodos, como chantagem, suborno e tortura, podem ser mais eficientes que a criptoanálise.

2.9. Resumo de Conceitos Básicos

Neste capítulo fizemos uma revisão de diversas técnicas utilizadas no decorrer da elaboração desta tese. As referências bibliográficas utilizadas são bastante abrangentes, e permitem que o leitor interessado no conhecimento detalhado dos assuntos tratados possa buscar as informações necessárias.

Os aspectos tratados variam desde conceitos a respeito da síntese de circuitos integrados e as vantagens e desvantagens da utilização de técnicas síncronas e assíncronas até uma introdução à criptografia.

Este capítulo também pode ser visto como um glossário que permite referenciar a terminologia utilizada nesta tese.

Capítulo 3
Trabalhos Correlatos

3.1. Introdução

Neste capítulo apresentamos um panorama do estado da arte na síntese de circuitos assíncronos. Várias tendências na especificação do processo de síntese, bem como a performance possível a partir dessas especificações formam um balizamento para o desenvolvimento de novas propostas.

Alguns dos artigos apresentados são bastante recentes. Evidentemente, esses artigos não foram utilizados na elaboração desta tese, mas são aqui mencionados para que possamos ter uma idéia das novidades existentes na síntese de lógica assíncrona.

3.2. Problemas de Síntese de Processos Paralelos

O desenvolvimento do processo de síntese de circuitos assíncronos depende em grande parte de um conhecimento teórico suficiente para a implementação correta desses circuitos.

Ferramentas que fazem a análise de processos paralelos a partir de grafos são extensivamente utilizadas, em particular aquelas derivadas das Redes de Petri. Circuitos que utilizam as metodologias *DI* e *SI* constituem a base de síntese para a lógica assíncrona. Alguns aspectos da sua aplicabilidade são vistos nesta seção.

3.2.1. Análise de Fluxo em Síntese de Alto Nível

Grafos são extensivamente utilizados para a síntese de circuitos assíncronos. No artigo [79], o comportamento de uma coleção de módulos é analisado de forma a que se possam concluir com segurança quais execuções são concorrentes e quais podem ser serializadas.

Ferramentas apropriadas foram desenvolvidas e estão sendo utilizadas para a síntese de alto nível no processo SHILPA.

3.2.2. Especificação de Interfaces Assíncronas

Muitas vezes, os problemas que ocorrem na síntese de sistemas assíncronos decorrem de uma especificação incompleta das interfaces entre estágios assíncronos. O artigo [80] faz uma análise das especificações para que se possa sintetizar um sistema com funcionamento correto.

A metodologia utiliza uma descrição da interface através de Redes de Petri e apresenta como exemplo de síntese um controlador para barramento VME.

3.2.3. O Custo de Implementações com Insensibilidade a Atrasos

Este artigo mostra que embora o custo em área e em perda de desempenho de circuitos com *DI* seja maior que o de o de circuitos *SI*, é possível obter uma solução intermediária [81].

O circuito deve ser decomposto em módulos *SI* com interfaces *DI*, o que permite a composição de um sistema globalmente *DI* e localmente *SI*. Assim o acréscimo em área é razoável, ficando em cerca de 40% e a perda de desempenho mantém-se inferior a 20%.

O estudo de caso é feito a partir de circuitos descritos por *STG*, sendo utilizada uma transformação do grafo que permite, em alguns casos, o relaxamento de relações causais entre entradas assíncronas em um sistema MIC.

3.2.4. Dependências de Velocidade em Portas Lógicas

Os modelos de síntese usuais pressupõem que a propagação de um sinal no interior de uma porta lógica tem um tempo de atraso similar para todos os valores das entradas. Nos casos onde isso se verifica é, quase sempre possível adotar um modelo *SI*, desde que as derivações de saída estejam dentro de uma mesma região isocrônica.

No trabalho apresentado em [82] faz-se a análise das restrições necessárias a uma implementação SI utilizando-se um processo de síntese por BDD – *Binary Decision Diagrams*. O sistema independe de métrica, e procura localizar apenas as restrições essenciais.

3.2.5. Temporização Min-max

Na referência [82], a análise de temporização buscava identificar problemas durante a fase de projeto do sistema. Ao se fazer a implementação do projeto, isto é, ao se adotar uma determinada tecnologia para síntese, novas restrições podem surgir.

Uma abordagem que leva em consideração os atrasos possíveis de serem gerados na fabricação dos circuitos está descrita em [83]. Nessa técnica os atrasos são tratados como atrasos desconhecidos porém limitados – *bounded* – conforme a tecnologia empregada.

Através do relaxamento dos critérios de síntese é possível obter circuitos com melhor performance do que os tradicionalmente gerados pelas metodologias *DI* ou *SI*.

3.2.6. Síntese de Circuitos no Modo Extended Burst

Máquinas de estado que operam em rajada *(burst-mode)* são uma variante da Máquina de Mealy. A chegada de um conjunto predeterminado de transições de entrada implica na geração concorrente de um conjunto de saídas. Contudo, não é permitida a concorrência entre entradas e saídas.

Uma extensão desse modelo prevendo a existência de entradas *don't care* e também de entradas condicionais, o que permite uma concorrência parcial entre entradas e saídas, foi proposto em [84]. As máquinas geradas são do tipo *s-proper*.

A síntese é feita por *STG*. Esse grafo, que é derivado da Rede de Petri, originalmente [85] propunha que mudanças de estado fossem feitas por transição do sinal de entrada. Recentemente, a possibilidade de selecionar entradas por nível foi estabelecida em [86], o que permitiu o desenvolvimento dos *Extended Burst-mode Circuits*.

3.3. Síntese de Circuitos Integrados VLSI

Nesta seção tratamos do problema da síntese sob um aspecto mais relacionado com a implementação. Algumas arquiteturas são apresentadas e também mencionamos uma importante ferramenta de síntese.

3.3.1. Controle de Pipelines Assíncronas com Duas Fases

A implementação da comunicação entre estágios através de protocolos de duas ou quatro fases possibilita discussões bastante criativas. Os autores mostram que se o modelo *SI* for substituído pelo modelo por atraso limitado *(bounded-model)*, ganhos de até 50% podem ser obtidos na performance do acoplamento [87]. Nesse artigo, os autores apresentam resultados preliminares relativos a um protótipo de microprocessador com acoplamentos em duas fases.

O uso do modelo por atraso limitado simplifica o formalismo necessário à implementação do circuito, mas torna necessário um maior conhecimento dos parâmetros impostos pelo processo de fabricação. Circuitos síncronos são desenvolvidos através do modelo por atraso limitado.

3.3.2. Petrify: Uma Ferramenta para Síntese Assíncrona

A maior parte dos sistemas assíncronos atuais utiliza um paradigma baseado em Redes de Petri. Um exemplo bastante completo de ferramenta de síntese é o programa Petrify [88].

Petrify analisa circuitos descritos por Rede de Petri ou *STG – Signal Transition Graphs*. A saída é uma lista que forma um circuito *SI*, sendo também eliminadas as redundâncias da descrição inicial.

Apesar de não utilizarmos síntese por Rede de Petri nesta tese, é importante mencionar que esta é uma das poucas ferramentas já desenvolvidas para síntese assíncrona.

3.3.3. Arquitetura Globalmente Assíncrona e Localmente Síncrona

O artigo [89] apresenta uma avaliação dos benefícios da utilização de uma interligação assíncrona entre diversos processos síncronos em circuitos *VLSI*. A idéia é simplificar a distribuição dos pulsos de relógio mantendo cada processo dentro de uma região equipotencial de tempo.

Foi apresentado um estudo de caso feito para um processador com um milhão de portas para o qual foram determinadas 24 partições. Dessa forma o overhead gerado pela implementação assíncrona é suficientemente pequeno e uma economia de cerca de 30% pode ser obtida no consumo de potência.

No entanto, o uso de um protocolo global com freqüência fixa e fase variável pode degradar a performance e causar *deadlocks*. Esses problemas ainda estão sendo estudados pela equipe.

Recentemente a IBM anunciou o desenvolvimento de circuitos experimentais que utilizam uma variante dessa técnica, chamada IPCMOS – *Interlocked Pipelined CMOS*. Essa técnica pode atingir 4,5GHz [90].

3.3.4. Relógios com Pausas

Uma proposta interessante para o acoplamento de dois sistemas não necessariamente assíncronos é apresentada em [91]. Nessa proposta os sistemas são interligados através de *FIFO*s. O relógio de cada sistema pode funcionar com um ciclo normal ou com um ciclo estendido, controlado pela interface. Dessa forma, os tempos de *setup* e *hold* são mantidos e a queda de performance acontece apenas durante a comunicação de dados.

Foi feita uma simulação na tecnologia MOSIS de 1.2μm. Utilizando-se circuitos síncronos, a freqüência máxima de relógio chegou a 220MHz, limitada pela capacidade de variação do oscilador de relógio utilizado.

3.4. Síntese de Blocos Funcionais

Nesta seção relacionamos alguns artigos que tratam da síntese de blocos funcionais específicos. Operadores aritméticos, em especial os somadores, são apresentados para que se tenha uma idéia dos progressos obtidos. Não incluímos artigos voltados para o desenvolvimento de multiplicadores, pois o desenvolvimento da tese não inclui este tipo de operador.

Somadores constituem a base de todas as operações aritméticas. Existe uma quantidade razoável de pesquisa a respeito de somadores síncronos. No entanto, apenas recentemente os somadores assíncronos passaram a receber a mesma atenção. É interessante notar que a maior preocupação dos projetistas de circuitos síncronos é relativa ao tempo de pior caso, ao passo que para circuitos assíncronos o tempo médio de execução e o consumo de potência são as funções que devem ser minimizadas.

Complementando a relação mencionamos o desenvolvimento de um núcleo *(core)* para *VLSI* como exemplo da síntese de um bloco funcional de grande complexidade.

3.4.1. Somadores Auto-temporizados com Estruturas DCVSL

As técnicas usuais para aceleração do desempenho de somadores síncronos são aplicadas a somadores auto-temporizados em *DCVSL* [92].

Os resultados favorecem somadores *CSA (Conditional Sum Adder)* e *CSkA (Carry Skip Adder)*. Somadores *CLA (Carry Lookahead)* são considerados pouco interessantes para circuitos assíncronos devido ao pequeno espalhamento dos tempos

de saída. O espalhamento é devido principalmente à diferença de propagação do sinal no circuito, não apresentando dependência significativa dos operandos.

3.4.2. Consumo de Potência em Somadores CMOS

Uma avaliação de somadores de 32 bits utilizando tecnologia CMOS de 2 μm foi apresentada em [93]. A avaliação foi feita em termos de desempenho e consumo de potência. Os somadores comparados foram um *Ripple Carry Adder (RCA)* estático, um *RCA* auto-temporizado dinâmico (*ADA*), um somador *RCA* dinâmico (*SDA*), e um somador estático do tipo Manchester Atlas (ATLAS).

O somador *RCA* estático é um circuito convencional com 7 portas. Este é o somador de referência. A potência dissipada é uma função de todas as transições, inclusive as transições espúrias. Essas transições são estatisticamente calculadas para um conjunto de dados aleatórios e corrigidas para um conjunto de dados de uma aplicação real.

O somador *RCA* auto-temporizado utiliza *dual rail encoding* na linha de *carry* para detectar o fim da operação de soma. Para isso ele utiliza um AND/OR dinâmico de 2x4 entradas com pré-carga seguido por um AND também dinâmico com 8 entradas. A existência de circuitos com funções redundantes e as cargas dinâmicas aumentam a energia consumida.

A versão síncrona do circuito anterior não necessita de *dual rail encoding*, e o complemento do sinal de *carry* é feito por um inversor. O consumo fica bastante reduzido, porém ainda é maior que o da versão *RCA* estática devido à pré-carga.

Tabela 3.1: Desempenho de alguns somadores CMOS

Somador	Atraso típico (ns)	Ciclo (ns)	Energia x tempo (pJ x ns)
RCA	23.08	30.00	2314
ADA	8.08 (médio)	9.83	2038
SDA	15.10	21.38	3445
ATLAS	16.97	22.06	1468

O somador Manchester Atlas utiliza um transistor de passagem para a propagação do *carry*. Essa é uma solução mais simples que a do *RCA* convencional, e apresenta maior velocidade e menor consumo, bem como maior regularidade topológica.

O resultado da simulação pode ser visto na Tabela 3.1. Essa tabela apresenta os resultados para um conjunto de 1000 operações não randômicas. Considera-se um ciclo de operação com tempo de pré-carga de 1.75 μs para somadores dinâmicos. Para somadores estáticos obtém-se o tempo do ciclo acrescentando 30% ao tempo da soma.

Pela tabela vemos que o somador auto-temporizado é nitidamente mais rápido. Normalmente os somadores representam apenas uma pequena parcela do consumo total, de modo que um consumo maior pode ser justificado quando se procura alto desempenho. No entanto, quando se considera consumo, área ocupada e regularidade do *lay-out*, o somador ATLAS é o que apresenta o melhor compromisso.

3.4.3. Uma Avaliação de Somadores Auto-temporizados

Neste caso, o autor procura mostrar que a implementação de somadores auto-temporizados não traz os resultados esperados [94]. Problemas advindos do protocolo necessário para tratar interfaces *DI* e a lógica adicional para detecção do fim de operação afetam de maneira negativa o desempenho desses circuitos, fazendo com que o desempenho final seja apenas ligeiramente melhor que o dos circuitos convencionais.

Outra observação importante diz respeito à variância dos tempos de saída do somador. Como em um circuito assíncrono a transferência de dados se faz em regime de exclusão mútua entre vizinhos, o desempenho de uma *pipeline* fica prejudicado toda vez que o somador necessita de um tempo maior que o tempo médio para efetuar a soma.

3.4.4. Somador do Tipo Manchester Auto-temporizado

Os autores apresentam um somador *DCVSL* [95] onde a propagação de *carry* é feita por cadeia Manchester em *dual-rail*. A inclusão de *buffers* na cadeia torna o circuito cerca de 50% mais rápido que a implementação sem *buffers*. O *overhead* da detecção de fim de operação é, aproximadamente, de 30%.

3.4.5. Somadores de Alta Performance com Finalização Especulativa

Somadores *RCA* apresentam complexidade de desempenho *O(n)* para o pior caso. Várias técnicas existem para melhorar o desempenho de somadores em circuitos síncronos, onde o pior caso é o fator limitante do desempenho. No entanto, quando essas técnicas são aplicadas a somadores auto-temporizados ocorre uma perda de desempenho quando a cadeia de *carry* é pequena, pois inevitavelmente uma lógica adicional vai se interpor na propagação do sinal. Esse fenômeno não é importante para somadores síncronos, onde a maior preocupação é com o caminho crítico para o pior caso, mas é fundamental para somadores auto-temporizados.

O artigo em referência [96] faz uma avaliação do desempenho de somadores Brent-Kung de 32 e 64 bits. Esses são somadores dinâmicos do tipo *carry lookahead.* Os autores propõem diversas técnicas para melhorar o desempenho. A análise dos sinais da árvore de *carry* permite obter os resultados em 2 (32 bits) ou 3 (64 bits) tempos fixos de retardo. Simulações mostram um ganho de 14 a 29% com relação às versões síncronas.

3.4.6. Projeto e Análise de Somadores Auto-temporizados

Este artigo apresenta uma comparação entre somadores estáticos *RCA - Ripple Carry Adder* e *CLA – Carry Lookahead*, utilizando lógica *DCVSL* de 1.2μm [97].

Os autores mostram que para um *CLA* de 2 bits é possível obter um somador com melhor desempenho e menor consumo que um somador *RCA* equivalente. A área necessária para a implementação é semelhante à do somador *RCA*.

3.4.7. Um Somador Auto-temporizado de 56 bits

Este somador utiliza uma combinação das técnicas *CSA* e *SCLA* (*Statistical Carry Lookahead*), chamada *HSCLA (Hybrid SCLA)* para obter um somador de alto desempenho. Somadores *CLA* apresentam um alto *throughput* mesmo para operandos não randômicos [98].

Somadores *SCLA* e *HSCLA* de 0,5μm são avaliados, mostrando que o HSCLA apresenta desempenho nitidamente superior a custo apenas 10% maior. Para esse somador, desenvolvido em tecnologia MOSIS, foi obtido um tempo médio de 1.28ns e uma dissipação de 52mW.

3.4.8. Somadores Auto-temporizados Sub-logarítmicos

Os autores apresentam uma arquitetura para somadores auto-temporizados baseada em uma variante do *CSA*. A complexidade da soma é de $O\left(\sqrt{\log n}\right)$ [99].

A detecção de fim de operação é feita por blocos, o que permite grande economia na implementação do circuito, mas sem afetar de maneira significativa o desempenho. Mesmo assim, a área ocupada é pelo menos o dobro da área de um somador *RCA* equivalente.

3.4.9. Micropipeline para Discrete Cosine Transform

Um núcleo para *VLSI*, contendo um processador para transformada discreta do coseno, *DCT* (*Discrete Cosine Transform*) foi apresentado em [100]. Simulações efetuadas mostram que o desempenho obtido é semelhante ao de um circuito síncrono com a mesma arquitetura.

O desempenho do circuito sofre com o tempo necessário para processar o protocolo de comunicação assíncrono entre estágios, mas a parte aritmética do circuito tem um bom desempenho e restaura o equilíbrio entre as metodologias de projeto.

Esses resultados estão de acordo com os obtidos para processadores de uso geral em Manchester[101] e na Caltech[102], muito embora para um processador especializado, com grande dependência de operações aritméticas, fosse esperado um desempenho superior. Estudos continuam sendo feitos no sentido de procurar uma arquitetura que diminua o impacto das técnicas assíncronas para que se possa melhorar o desempenho.

3.5. Testabilidade

O desenvolvimento de técnicas voltadas para ***ADFT*** (*Asynchronous Design for Testability*) ainda é bastante incipiente. A adaptação de técnicas básicas como *o scan-path* e o ***BIST*** é problemática, e as soluções encontradas estão voltadas para aplicações específicas. Relacionamos a seguir algumas das principais técnicas disponíveis na literatura.

3.5.1. BIST para Micropipelines

Este artigo detalha o procedimento de teste utilizado no projeto do processador AMULET 2e [103]. O procedimento de teste foi implementado com registradores do tipo ***BILBO*** – *Built-in Logic Block Observation*. Esses registradores são compostos por um par de *latches* e funcionam de modo similar a um *flip-flop master/slave*, podendo funcionar em modo normal ou ainda ser reconfigurado como registrador de deslocamento, *LFSR* ou ainda *SA*.

A principal vantagem do uso desta metodologia deve-se ao fato de que registradores assíncronos do tipo *BILBO* são similares às versões síncronas desses registradores, tendo portanto as mesmas propriedades. Isso permite que o *BIST* seja implementado com facilidade.

Existe contudo um *overhead* em área e uma diminuição de performance já que, em operação normal, o sinal precisa atravessar portas lógicas adicionais.

3.5.2. Scan Path para Micropipelines

Micropipelines são freqüentemente utilizadas como o modelo padrão para a implementação de um *pipe* assíncrono. É natural, portanto, que boa parte dos desenvolvimentos relacionados com testabilidade estejam voltados para essa área.

O artigo em questão [104] faz considerações sobre o uso de um *scan-path* para o teste de *bundled-data* em micropipelines. Registradores especiais possibilitam o *scan* assíncrono. Esses registradores contudo apresentam área superior à dos registradores convencionais e um tempo de atraso adicional durante o modo normal de operação.

3.5.3. Inicialização de Circuitos Assíncronos

Um aspecto freqüentemente esquecido pelos projetistas diz respeito à inicialização dos circuitos, especialmente os assíncronos.

As ferramentas de síntese podem, às vezes, assumir um valor inicial para uma variável sem que haja uma correspondência direta com o circuito elétrico gerado. O artigo em tela [105] procura explorar técnicas que permitam identificar essas inconsistências da síntese.

3.6. Resumo de Trabalhos Recentes

Apresentamos um resumo do estado da arte em técnicas assíncronas. Projetos de somadores, arquiteturas gerais e problemas para a implementação de testabilidade em circuitos assíncronos foram vistos neste capítulo.

Muitas dessas técnicas dependem da existência de uma biblioteca de circuitos especialmente desenvolvidas para síntese assíncrona e de que não dispomos no momento. Nosso projeto, no entanto, depende apenas de circuitos convencionais, conforme será visto nos próximos capítulos. Mesmo assim, muitos dos conceitos apresentados nesses trabalhos foram utilizados durante o desenvolvimento da tese.

Capítulo 4
Proposta de Tese

4.1. Motivação

Apresentamos no Capítulo 2 os conceitos básicos relacionados com o desenvolvimento desta tese. A síntese de Circuitos Integrados *VLSI* utilizando metodologia assíncrona é na realidade um procedimento multidisciplinar e bastante complexo. Dentre as razões que nos levam a procurar a síntese de circuitos lógicos assíncronos como tema de tese, queremos destacar a evidência que a síntese de circuitos assíncronos vem ganhando nesta última década. Isso acontece, dentre outros motivos, devido à crescente utilização de equipamentos que necessitam de grande capacidade computacional e baixo consumo de potência. Essas características aliadas a uma limitação nos níveis de interferência eletromagnética podem ser encontradas nos novos equipamentos portáteis, como os telefones celulares de 3ª geração.

Outra característica dos sistemas digitais assíncronos refere-se à independência da tecnologia de implementação. Cada bloco funcional é inteiramente independente dos demais, exceto com relação à interface. É possível interligar, ou ainda substituir, blocos funcionais cuja organização interna e método de temporização sejam diversos. A síntese da interface apresenta apenas restrições de ordem elétrica (níveis de tensão/corrente), que quando respeitadas, permitem a interligação de módulos desenvolvidos para diferentes escalas de integração. É por isso que consideramos que circuitos digitais assíncronos são independentes da tecnologia de implementação.

Um dos problemas críticos para a síntese de circuitos integrados síncronos de alta performance é a geração de um relógio coerente. Quanto maior a freqüência e mais longos os ramos da árvore, mais difícil é a distribuição do relógio. Soluções para esse problema tem sido do tipo evolucionário. No entanto, a cada progresso obtido com relação à freqüência do relógio, a dissipação de potência aumenta, o que traz novas restrições. A própria existência de um relógio central implica na produção de

pulsos de ruído de alta energia que podem provocar interferência em circuitos analógicos próximos. Sabemos que o aumento do consumo de corrente pode inviabilizar um projeto por torná-lo inadequado para alimentação por baterias. Em alguns casos, até mesmo o limite de dissipação térmica do invólucro pode ser ultrapassado. A compensação para esse problema vem sendo feita com a utilização de tensões de operação cada vez menores, o que provoca novos problemas de suscetibilidade ao ruído.

É claro que existem progressos tanto no desenvolvimento de baterias como na utilização parcimoniosa de periféricos e do próprio pulso de relógio. A utilização de tensões diferentes para operação interna e para as interfaces dos circuitos integrados vem sendo utilizada para amenizar o problema do ruído. Mas apenas a utilização de lógica assíncrona permite que o consumo do circuito seja intrinsecamente uma função do trabalho executado e que o ruído gerado seja distribuído ao longo do tempo.

Sendo o problema da síntese assíncrona bastante vasto, fizemos um apanhado dos principais desenvolvimentos existentes. Após esse estudo, foram feitas várias considerações que permitiram reduzir o problema da síntese a dimensões compatíveis com o trabalho de uma tese. Essas considerações serão mostradas no decorrer deste capítulo.

4.2. Dificuldades na Síntese de Circuitos Assíncronos

Descrevemos a seguir algumas das dificuldades existentes na síntese de lógica assíncrona. Essas dificuldades levaram a uma sistematização dos objetivos da tese, que serão vistos na seção 4.3.

4.2.1. Ferramentas

Um dos principais problemas na síntese assíncrona é a falta de ferramentas adequadas. No entanto, ferramentas voltadas para a síntese síncrona podem ser encontradas em muitos laboratórios. Seria muito interessante implementar um sistema que permitisse facilitar a transição de projetos essencialmente síncronos para projetos assíncronos. Dessa forma seria possível aproveitar o conhecimento já existente para o desenvolvimento de projetos síncronos para gerar um sistema assíncrono simples, possivelmente com melhor desempenho.

Ainda com relação às ferramentas, é importante mencionar que não dispomos no momento de bibliotecas adequadas à implementação de circuitos assíncronos auto-

temporizados. A implementação de uma biblioteca especializada está fora do escopo desta tese, sendo indicada para um desenvolvimento futuro. Assim, nossos projetos precisam de um tipo de desenvolvimento onde ferramentas síncronas possam ser utilizadas. Os sistemas de síntese disponíveis no *LPC* estão voltados para a simulação lógica e lógica-temporal de sistemas descritos em *VHDL*. Eventualmente compensações de roteamento podem ser utilizadas (*back annotation*). Porém, para fazermos síntese assíncrona a esse nível seria necessário desenvolver no mínimo um sistema de *standard cells* específicas, complementada por outro sistema para a síntese automática de funções lógicas complexas. Consideramos que esse desenvolvimento deveria ser feito em outros trabalhos dedicados especificamente à síntese de circuitos integrados.

4.2.2. Classes de Circuitos Assíncronos

Vimos anteriormente que apenas a classe de circuitos *DI* apresenta características que permitem um tratamento assíncrono total. Nessa classe, todos os atrasos são considerados, o que a torna essencialmente segura. No entanto, essa característica tem um preço. Ela exige uma quantidade maior de circuitos, havendo um *handshake* para cada sinal recebido. Além disso os circuitos de *handshake* são inerentemente lentos, pois devem obedecer a um protocolo.

Nas demais classes, desde a *SI* até o circuito síncrono, existe algum tipo de compromisso que pressupõe um certo conhecimento da tecnologia de implementação do circuito para que o modelo seja válido. Nessas classes é possível relaxar alguns critérios de síntese, tornando o circuito mais simples. Um ganho de performance é possível desde que exista uma garantia de que o circuito gerado irá operar dentro de limites definidos.

Assim, de um lado temos os circuitos auto-temporizados do tipo *DI,* que podem operar na maior velocidade permitida pela sua lógica intrínseca. Essa velocidade inclui todo o tipo de propagação interna de sinais, sendo também uma função dos operandos. Além disso, a performance do circuito acompanha automaticamente o ambiente de operação e o espalhamento da fabricação. Evidentemente, o custo dessa implementação é elevado.

No outro extremo, circuitos síncronos seguem padrões de projeto mais simples, pois sinais transientes não constituem problema desde que eles se estabilizem antes do pulso do relógio. Porém, a necessidade de se considerar o pior caso tanto

para o conjunto das operações quanto para o espalhamento de fabricação e para as condições de ambiente faz com que a performance máxima de um sistema síncrono em produção seja inferior à metade daquela que poderia ser obtida para um sistema *DI* auto-temporizado em condições ótimas.

Dessa forma, a síntese de sistemas *VLSI* e *ULSI* fica bastante interessante se pudermos seguir as recomendações de [89], ou seja, a utilização de módulos sintetizados segundo um dos modelos mais simples, apropriados para funcionamento em uma mesma região isocrônica, e interligados por uma interface *DI*.

4.2.3. Portas Lógicas

A temporização de interfaces assíncronas é feita normalmente através de um sistema que incorpora relógio e dados em um mesmo conjunto de sinais, utilizando para isso um protocolo. Esse é o caso, por exemplo, de interfaces que utilizam *dual-rail encoding*.

A avaliação de uma função combinacional, no entanto, costuma ser feita em canais distintos, sendo um para dados e outro para temporização. Até o momento, o tipo de porta lógica assíncrona mais adequado para o desenvolvimento de circuitos auto-temporizados parece ser o *DCVSL*. Essa lógica utiliza circuitos dinâmicos na entrada mas produz um sinal de saída que é mantido pela própria lógica, evitando o uso de um *latch* entre etapas. A temporização é feita por um sinal de *request*, que inicia a operação, e o término acontece quando a lógica produz o sinal *done*.

Em alguns casos, mesmo quando não se utiliza lógica auto-temporizada, ainda é possível obter a informação de término da operação. Um operador aritmético, por exemplo, pode ter o seu tempo de operação avaliado através da observação do sinal de *carry*, e pode ser sintetizado utilizando apenas lógica convencional.

4.2.4. Metodologia de Síntese

Existe uma quantidade de trabalhos bastante razoável onde a metodologia baseada em Redes de Petri é empregada. Essa teoria é importante porque ela permite estabelecer uma correspondência simples entre circuitos descritos e o modelo. O reconhecimento de processos concorrentes também fica bastante facilitado. No entanto, circuitos sintetizados a partir de Redes de Petri não são necessariamente corretos ou de alta performance. Extensões tem sido propostas, como os *STG*, que apresentam melhores

condições de análise dos circuitos. A inexistência de *deadlocks* deve ser garantida por construção.

4.2.5. Testabilidade

Também não se deve esquecer que a produção de circuitos integrados depende fortemente de um sistema de teste adequado. As metodologias de *DFT*, ou seja, de projeto visando a testabilidade, já estão bastante desenvolvidas no que diz respeito a circuitos síncronos. No entanto, no que diz respeito a circuitos assíncronos, elas ainda são incipientes. Métodos para diagnóstico são praticamente inexistentes.

4.2.6. Desempenho de Operadores Aritméticos

Para a síntese síncrona, a preocupação principal do projetista diz respeito à determinação do pior caso, pois é ele que irá determinar a performance do sistema. Os operadores aritméticos, em especial os somadores, vem sendo projetados de forma a minimizar o tempo de pior caso.

Para a síntese assíncrona, o que deve ser procurado é a minimização do tempo médio da operação. A distribuição dos tempos de operação também é importante, pois um desvio padrão muito grande pode levar a retenções em outras partes do circuito.

Vimos na seção 3.4. que o desenvolvimento de operadores aritméticos, principalmente somadores, vem recebendo especial atenção dos pesquisadores. Os artigos apresentados mostram que é muito difícil inovar no que diz respeito ao algoritmo da soma. Quase todos os artigos partem de algum tipo de somador já utilizado na síntese síncrona e, através de modificações, procuram uma adequação à síntese assíncrona. O desempenho desses somadores depende fortemente da implementação escolhida.

4.3. Objetivo do Trabalho

Integram esta tese projetos para aplicar os conceitos teóricos desenvolvidos a circuitos de ordem prática. No entanto, é importante ressaltar que o principal objetivo deste trabalho não é o projeto e a implementação física de sistemas assíncronos. O que procuramos nesta tese é fornecer ao projetista um ferramental adequado ao desenvolvimento de projetos assíncronos. Deixamos ao projetista a liberdade de escolher a metodologia de síntese dos blocos funcionais para que ele possa, se assim desejar, integrar conceitos desenvolvidos para o ambiente síncrono a uma

metodologia voltada para a síntese assíncrona. Acreditamos que dessa forma será possível aplicar esses conceitos de uma forma eficiente, aumentando a produtividade da etapa de projeto.

Neste trabalho iremos apresentar um conjunto de metodologias voltadas para a síntese do controle de temporização de sistemas assíncronos auto-temporizados, sejam eles compostos por blocos funcionais *DI* (insensíveis a atrasos),ou temporizados por *matched delay*. Essas metodologias, que foram integradas a partir de uma arquitetura básica ou seja, de um *framework*, podem ser úteis tanto para o desenvolvimento de sistemas inteiramente assíncronos, como para facilitar a migração de projetos originalmente síncronos e que foram desenvolvidos com a utilização de ferramentas convencionais.

A possibilidade de se ter um projeto síncrono inteiramente depurado com as ferramentas de uso corrente do projetista, o qual, através de pequenas modificações que podem ser feitas gradualmente no projeto, pode ser transformado em sistema assíncrono com a performance necessária para propiciar um salto de qualidade no produto final, será mostrada como um dos pontos fortes da arquitetura que nos propusemos desenvolver.

Vimos que existe atualmente uma grande ênfase na análise do desempenho de somadores. Vimos também que existem poucos estudos sobre integração de somadores em sistemas mais complexos. É nossa proposta verificar o desempenho possível de ser obtido por um somador assíncrono quando integrado a uma *ALU*. A avaliação do desempenho do sistema leva em consideração as operações aritméticas e as operações lógicas. Uma simulação da execução de *benchmarks* é utilizada como medida de desempenho.

A nossa metodologia de síntese tem características hierárquicas, sendo capaz de atender a uma topologia arbitrária, não ficando limitada ao fluxo de dados em forma de *pipeline*. Uma grande preocupação na síntese de sistemas assíncronos diz respeito à possibilidade da geração de *deadlocks*. Nossa proposta foi desenvolvida de forma a evitar a formação de *deadlocks* nos circuitos onde não existam defeitos de fabricação.

Nossa proposta também contempla aspectos de projeto visando a testabilidade, *DFT*. As dificuldades inerentes à implementação de um *scan-path* e a ausência de desenvolvimentos dedicados à técnica *BIST* fizeram com que propuséssemos uma

estrutura capaz de testar automaticamente os blocos funcionais de um sistema assíncrono que utilize a nossa metodologia de síntese e que possa talvez ser utilizado com outras metodologias.

Com relação às aplicações previstas para verificação da tese, é interessante estabelecer um limite adequado a uma realização feita por uma equipe pequena e sem acesso a ferramentas adequadas para síntese assíncrona. Uma proposta nesse sentido decorre da observação da complexidade de um trabalho como o da referência [100]. A síntese de um subsistema assíncrono que implementa um algoritmo criptográfico mostra o desempenho da nossa metodologia de síntese quando aplicado a um projeto complexo.

4.4. Resumo da Proposta de Tese

Faremos neste trabalho o desenvolvimento de uma arquitetura de sistemas voltada para a temporização de sistemas assíncronos do tipo *DI*. Essa arquitetura deverá prever a fácil migração de sistemas síncronos para um ambiente assíncrono com um mínimo de reprojeto. As considerações de ordem teórica necessárias para o desenvolvimento do *framework* serão apresentadas no Capítulo 5.

Uma série de aplicações, desenvolvidas como ferramenta de trabalho e também como prova de conceito das metodologias apresentadas serão apresentadas nos capítulos subsequentes. As simulações necessárias, sejam elas lógicas, elétricas ou produto da síntese de alto nível dos circuitos serão vistas conforme necessário.

Capítulo 5
Desenvolvimento Teórico

5.1. Introdução

A concorrência assume importância cada vez maior na especificação e na implementação de sistemas digitais. A especificação de sistemas complexos é feita normalmente através da sua decomposição em sistemas concorrentes mais simples, que através das suas interações irão produzir o comportamento desejado para o sistema. Também é desejável que na implementação do sistema seja maximizado o paralelismo das operações, procurando sempre a minimização do tempo de resposta do sistema.

Sistemas concorrentes são notoriamente difíceis de se projetar ou mesmo de se analisar. A quantidade de estados de um sistema e a interação entre eles costumam ser de tal forma complexos que somente através de uma especificação formal rigorosa é possível chegar a uma metodologia precisa para a análise do comportamento do sistema.

Neste trabalho utilizamos o paradigma dos filósofos de Dijkstra [106] [107] para definirmos o problema geral da concorrência e, através de uma solução para esse problema, buscarmos uma metodologia que permita propor uma nova arquitetura, ou *framework*, para o desenvolvimento de sistemas assíncronos.

5.2. Filósofos e Processos Concorrentes

Vários conjuntos de filósofos com comportamento adequado a diferentes problemas de concorrência tem sido descritos na literatura [108] [109] a partir do trabalho original de Dijkstra.

Um filósofo pode ser definido como um ser que, enquanto acordado, passa a maior parte do tempo envolto em seus pensamentos. Mas, ocasionalmente, ele precisa comer. A atividade de um dia na vida desse filósofo pode ser modelada por um

processo seqüencial conforme a representação da Figura 5.1. Utilizamos nesta seção a notação usual para Redes de Petri apenas para facilidade de entendimento.

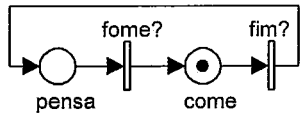


Figura 5.1: O dia de um filósofo

Vemos que inicialmente o filósofo está pensando. Quando fica com fome, ele se alimenta e depois volta a pensar. Neste modelo não existe concorrência, já que cada atividade ocorre a seu tempo.

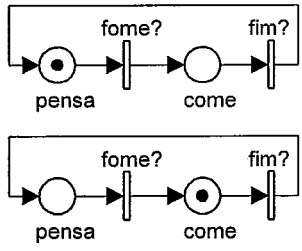


Figura 5.2: O dia de 2 filósofos independentes

Para que haja concorrência é preciso que várias ações possam ocorrer ao mesmo tempo. Seja então um sistema composto por dois filósofos, como representado na Figura 5.2. Esse sistema é inteiramente concorrente, pois não existe nenhuma relação causal entre as atividades dos filósofos. Ambos podem pensar ou comer simultânea ou alternadamente conforme lhes seja mais conveniente.

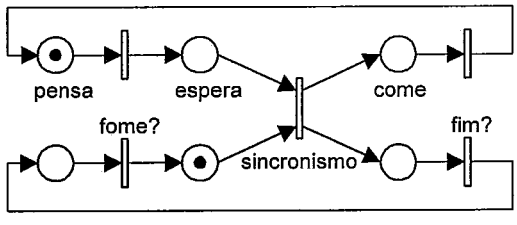


Figura 5.3: O dia de 2 filósofos sociais

O processo torna-se mais interessante se considerarmos que os dois filósofos são seres sociais, que desejam fazer suas refeições em conjunto. Essa situação pode ser vista no diagrama da Figura 5.3. Neste caso, a barreira que sincroniza o início da refeição somente pode ser ultrapassada quando ambos os filósofos demonstrarem que

estão aguardando comida. Vemos ainda que cada filósofo pode deixar a mesa antes que seu colega termine a refeição. Não existe sincronismo nesse ponto.

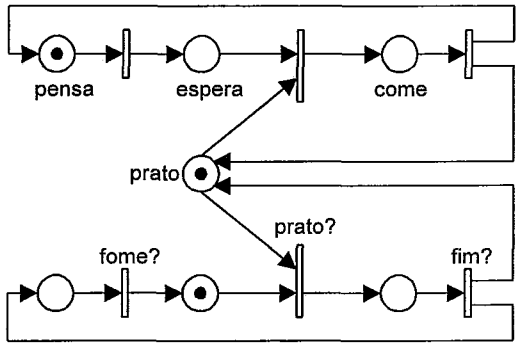


Figura 5.4: Um único prato para 2 filósofos

Vamos agora imaginar que os filósofos só disponham de um prato. Assim criamos uma situação oposta à anterior, pois se um dos filósofos estiver comendo o outro precisa esperar até que o prato esteja disponível. Esta é uma situação de exclusão mútua, que pode ser vista na Figura 5.4. Neste sistema é possível que o prato esteja disponível e que nenhum dos filósofos esteja com fome. Dizemos então que esse sistema opera sob carga normal. Pode acontecer também que os filósofos tenham passado por um jejum prolongado e por isso estejam com muita fome. Nessa situação, a cada vez que um prato se torna disponível um dos filósofos começa imediatamente a comer. Essa situação caracteriza a operação em alta carga (*heavy load*).

No problema original dos *dining philosophers*, cinco filósofos sentam-se à mesa para comer espaguete. Acontece que o espaguete é tão escorregadio que a refeição é feita com o auxílio de dois garfos. Os pratos são servidos, os filósofos estão com fome, mas apenas um garfo é colocado entre pratos vizinhos.

Se os filósofos estão com muita fome, e cada um deles agarra um garfo cria-se uma situação de *deadlock*. Ou seja, ninguém come até que todos morram de fome (filósofo morto não devolve o garfo) [110].

Outra situação pode ocorrer caso um dos filósofos não consiga receber o segundo garfo, ficando impedido de comer. Essa é uma situação de inanição *(starvation)*. Ela pode ocorrer, por exemplo, caso os demais filósofos sincronizem seus acessos à comida. O quinto filósofo terá, alternadamente um garfo à sua direita ou à sua esquerda, mas nunca receberá os dois garfos.

Nota-se então a necessidade da existência de um protocolo para acesso à comida. Já que esse é um problema que envolve exclusão mútua e operação em alta carga, na sua solução deve-se procurar um protocolo que evite a formação de *deadlocks*. O problema da inanição deve ser tratado de maneira a que todos os filósofos tenham acesso à comida, observando-se caso seja possível, o conceito de *fairness*. Ou seja, todos os filósofos deverão dispor de acesso razoável à comida. Outro parâmetro a ser observado é o da concorrência. Como existem apenas cinco garfos e cada filósofo necessita utilizar dois deles, só podem acontecer, no máximo, duas refeições simultâneas. Dizemos então que a concorrência desse sistema fica limitada a 2/5 no máximo.

Uma variação desse problema foi proposta em [111], para tratar graus variados de necessidade de comida. Filósofos Zen dividem a mesa com filósofos normais. A característica principal dos filósofos Zen é que eles comem pouco, isto é, para cada duas refeições dos filósofos normais eles fazem apenas uma. Na representação do problema os garfos são substituídos por palitos (*ohashi*). Filósofos Zen dividem dois pares de palitos, mas entre os pratos dos filósofos normais fica apenas um palito. A refeição do filósofo Zen é feita com dois pares de palitos, que ele devolve ao final. Filósofos normais, embora eventualmente utilizem três palitos, só devolvem um palito para cada vizinho. Dessa forma eles garantem que farão duas refeições enquanto os filósofos Zen fazem apenas uma.

5.3. Escalonamento por Reversão de Arestas

A metodologia de Escalonamento por Reversão de Arestas *(SER – Scheduling by Edge Reversal)* [112] foi desenvolvida para facilitar a implementação de processos concorrentes em uma rede de computadores. Nesse tipo de rede, um computador processa parte de um programa e eventualmente aguarda até que uma troca de mensagens entre seus vizinhos indique que ele pode prosseguir sua operação, possivelmente com um novo conjunto de operandos. Nesse tipo de operação considera-se que a capacidade de processamento local é suficiente, o que implica em um pequeno custo de processamento. Por outro lado, o custo da comunicação, normalmente feita através de linhas seriais de velocidade relativamente baixa, é sempre alto. Dessa forma, os algoritmos para processamento distribuído procuram minimizar tanto a quantidade de mensagens quanto o volume de informações presentes em cada mensagem.

Em [111] ficou demonstrado que, utilizando-se o algoritmo *SER,* seria possível construir redes auto-oscilantes controladas pelo tempo de execução dos processos e de forma independente da topologia da rede.

Neste trabalho utilizamos esses conceitos para implementar em hardware um sistema de sincronismo para processos independentes, mas restritos pelo acesso a recursos comuns. No nosso caso consideramos que o custo do processamento, definido tanto pela área dedicada à implementação do processo quanto pelo próprio tempo de processamento, é sempre alto, e que o custo da comunicação é relativamente baixo.

A forma utilizada para implementar circuitos com essas características será vista mais adiante. Nesta seção faremos apenas a descrição do funcionamento do mecanismo de reversão de arestas e apresentaremos algumas das propriedades que o tornam interessante para a implementação de uma arquitetura (*framework*) para circuitos lógicos assíncronos.

5.3.1. Funcionamento e Propriedades do SER

Seja um sistema S composto por um conjunto de P de processos e um conjunto R de recursos. Ou seja, $S=(P,R)$. Como R deve ser compartilhado por P, sujeito a considerações que evitem situações de *deadlock* ou inanição *(starvation)*, procura-se um mecanismo de escalonamento que tenha essas características.

Consideramos um sistema que opera em alta carga *(heavy load).* Assim, um processo pode estar em duas situações: ou ele está operando, ou ele está aguardando a disponibilidade de recursos para operar novamente. Consideramos também, para garantir o conceito de *fairness*, que um mesmo processo P_i não pode receber permissão para operar uma segunda vez antes que todos os processos que dividam recursos com P_i já tenham tido a oportunidade de operar. Sistemas desse tipo são tipicamente sistemas assíncronos, onde cada processo tem sua própria base de tempo.

Podemos construir para o sistema S um digrafo G, composto por N nós, que representam os processos, e por E arestas, que representam os recursos compartilhados. Na construção do digrafo $G=(N,E)$, é gerada uma aresta, e apenas uma, para cada conjunto de recursos compartilhado entre dois processos. Com isto forma-se um *clique* para cada nó existente no grafo. Lembramos que processos que compartilham um recurso não podem operar simultaneamente, ou seja, definem um problema de exclusão mútua.

Para efeito de terminologia nesta tese, passaremos a empregar os termos processo e nó como sinônimos.

A orientação das arestas, por exigência do algoritmo, deve ser acíclica. Com isto é possível garantir que para o conjunto Ω de orientações acíclicas de G, teremos em cada orientação ω individual pelo menos um nó fonte *(source)* e também pelo menos um nó sumidouro *(sink)*. Nós do tipo sumidouro, isto é, nós que dispõem de todos os recursos necessários para operar, podem operar pelo tempo que for necessário. Contudo, após a operação, eles devem devolver a seus vizinhos os recursos utilizados. Isto é representado pela transformação das orientações ω de G em um novo conjunto de orientações ω', através da reversão das arestas incidentes no sumidouro que terminou sua operação. Lembramos que se a orientação ω era inicialmente acíclica, então a orientação ω' também será necessariamente acíclica. A repetição dessa transformação fará com que o sistema evolua através de uma decomposição de *sinks*. Caso os processos P_i tenham tempo de execução constante a decomposição será cíclica, voltando eventualmente a uma orientação ω_i, sendo possível definir um período p para o ciclo de execução do grafo, bem como a quantidade de operações m_i para cada nó, em cada ciclo. Mesmo que os tempos não sejam constantes, ainda assim é possível definir uma ordem parcial das operações dos nós. Ou ainda, visto de outra forma, pode-se garantir que os processos não serão executados fora da ordem inicialmente estabelecida.

Seja a decomposição S dos *sinks* de G, composta pelos conjuntos S_0 até $S_{\lambda-1}$. Dizemos que um nó pertence a S_k, onde k varia de 0 a $\lambda-1$, sendo λ o tamanho da decomposição gerada, se e somente se o caminho mínimo desse nó até um nó sumidouro de G tiver tamanho igual a k. Dois componentes S_k, e S_l, são considerados sucessivos se $|l-k|=1$. Uma propriedade interessante da decomposição em *sinks* diz que para qualquer nó sumidouro de S_k, existe pelo menos um nó vizinho que também é sumidouro em S_{k-1}. Ou seja, a ordem parcial do processamento fica estabelecida pela topologia de G [113].

Para efeito de demonstração faremos considerações sobre um caso particular, onde todos os processos tem o mesmo tempo de execução e uma base de tempo comum. Isto não implica em perda de generalidade, conforme foi demonstrado em [112]. Seja então o exemplo da Figura 5.5. Nessa topologia existem inicialmente dois

sumidouros *(sinks)*. Aplicando-se as transformações definidas pela reversão de arestas, o grafo eventualmente voltará a uma configuração ω_i, não necessariamente ω_0, executando ciclos de comprimento k.

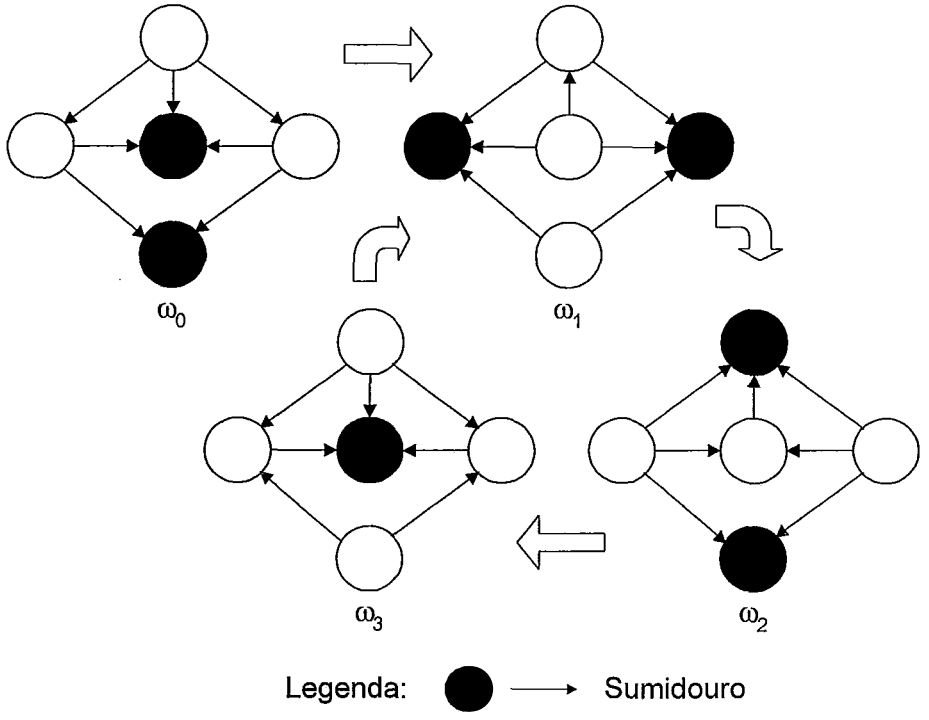


Figura 5.5: Exemplo de sistema SER com 5 nós.

Em um sistema assíncrono cada sumidouro *(sink)* irá operar de forma independente. Em sistemas onde existam mais de um sumidouro, e um controle global dos tempos que possibilite a identificação contínua dos estados, é bastante provável que se observe a reversão de arestas de apenas um nó a cada vez. Sistemas síncronos onde todos os nós sumidouro revertem suas arestas simultaneamente formam um sistema guloso (*greedy*), cuja principal propriedade é a minimização do tamanho λ da decomposição de *sinks*. Ou, visto de outra forma, sistemas gulosos maximizam a operação dos processos. É um sistema desse tipo que está representado na Figura 5.5. Sistemas gulosos definem os possíveis estados que aparecem em uma fotografia (*snapshot*) de um sistema assíncrono genérico [114].

Lema: *Sejam as orientações acíclicas* ω *e* ω' *de G, onde* $\omega'=g(\omega)$. *Um sumidouro (sink) de* ω' *tem pelo menos um vizinho que é sumidouro de* ω [112].

Por definição do algoritmo, um nó não pode ser sumidouro em duas operações seguidas. Como a operação de transformação g ocorre apenas entre elementos vizinhos a um sumidouro, então, para qualquer sumidouro em ω' deve existir pelo menos um vizinho que é fonte em ω' e que não era fonte em ω. Para que um nó seja fonte em ω' é preciso que ele tenha sido sumidouro em ω. Daí o lema.❐

Podemos definir $m_i(\sigma,q)$ como a quantidade de operações efetuadas pelo nó i, $i \in N$ nas primeiras q orientações, $q \geq 1$ do escalonamento σ. Então $m_i(\sigma,1) = 0$ para todo nó $i \in N$ e qualquer $\sigma \in \Sigma$. Isso porque qualquer operação em um nó produz uma nova orientação, e aí teríamos $q>1$. Definimos Σ como a união dos σ_i.

Teorema: *Sejam os nós i e j de N. Seja r o tamanho do caminho mínimo entre i e j. Dizemos que* $|m_i(\sigma,q) - m_j(\sigma,q)| \leq r$ *para qualquer* $\sigma \in \Sigma$ *quando* $q \geq 1$ [112].

Este teorema garante que o algoritmo *SER* não permite inanição, pois todos os nós irão eventualmente operar. Existe a garantia que os nós mais adiantados somente poderão prosseguir em sua operação até atingir um distância máxima r do nó mais atrasado, distância essa limitada ao tamanho do caminho mínimo entre os nós. Chamamos de distância à diferença entre as quantidades de operações para as q orientações especificadas. Podemos facilmente verificar que quando $r=1$, os nós são vizinhos, e operam alternadamente. Então $|m_i - m_j| \leq 1$. Por extensão, para $r=2$ podemos dizer que a diferença é menor ou igual a 2. E, por indução, pode-se provar que o teorema é válido. ❐

Teorema: *Seja* σ *um escalonamento qualquer e* σ_g *um escalonamento guloso, ambos com orientação inicial* ω. *Então* $m_i(\sigma,q) \leq m_i(\sigma_g,q)$ *para todo* $i \in N$ *e qualquer* $q \geq 1$ [112].

Mencionamos anteriormente que a maior eficiência das operações é obtida através de um escalonamento guloso, pois este minimiza a decomposição de *sinks*, o que implica na maximização das operações para uma determinada quantidade q de orientações. Este teorema mostra que um escalonamento qualquer que não seja um escalonamento guloso irá necessitar de pelo menos mais uma orientação para executar a mesma quantidade de operações de um escalonamento guloso. ❐

Outra propriedade importante dos escalonamentos gulosos é que a seqüência de orientações acíclicas irá eventualmente se repetir, formando um período cujo tamanho é denominado por p, sendo $p \geq 2$. Isto leva a um corolário importante.

Corolário: *Para um escalonamento guloso, todos os nós operam um mesmo número de vezes, m, a cada período* [112].

Este corolário garante a propriedade de *fairness*. O teorema da inanição *(starvation)* estabelece um limite máximo para a diferença das quantidades de operações dos nós. Para um escalonamento guloso e periódico, essa é a única forma possível para garantir a ausência de inanição *(starvation freedom)*. □

É conveniente lembrar que quando o escalonamento não é guloso, a propriedade de *fairness* também é mantida, embora sujeita à restrição de inanição, já que a ordenação parcial dos processos fica determinada pela topologia de G.

5.3.2. Paralelismo e Concorrência em Sistemas SER

Uma medida de concorrência pode ser definida por $\gamma = m/p$. Os valores m e p correspondem àqueles definidos para um escalonamento guloso, a saber:

$m \rightarrow$ quantidade de operações de um nó durante um ciclo.

$p \rightarrow$ comprimento do ciclo.

Em qualquer sistema que opere sujeito ao critério da exclusão mútua, o maior valor possível para a concorrência é dado por $\gamma = ½$. Sistemas de máxima concorrência podem ser representados por grafos bipartidos onde cada uma dessas partições é executada alternadamente. E ainda, para que o valor máximo possa ser atingido, é necessário que o tempo de execução de todos os processos de uma mesma partição seja o mesmo.

Para um sistema como o da Figura 5.5 temos uma concorrência máxima $\gamma_{max} = 1/3$.

É interessante notar que a concorrência é uma função da conectividade de G. Mostramos que grafos esparsos bipartidos podem atingir o valor máximo de concorrência. Por outro lado, em um grafo completamente conectado apenas um nó

poderá operar a cada vez. Então os limites da concorrência em função da conectividade são dados por $1/n \leq \gamma \leq 1/2$.

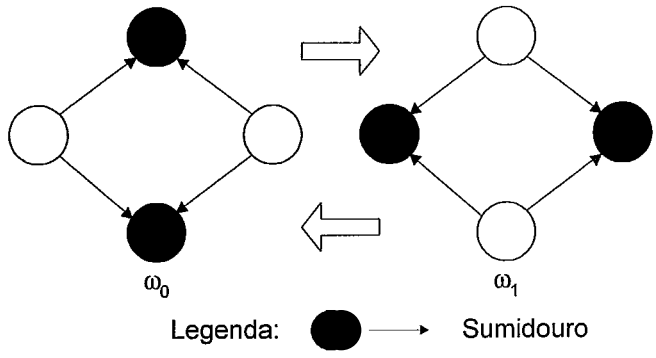


Figura 5.6: Anel com concorrência máxima, m=1, p=2.

Além da conectividade, a concorrência também é função da orientação inicial das arestas de *G*. Mostramos um exemplo de grafo com topologia em anel, que, dependendo da orientação inicial das arestas pode atingir a concorrência máxima como na Figura 5.6 ou ficar limitado à concorrência mínima como na Figura 5.7.

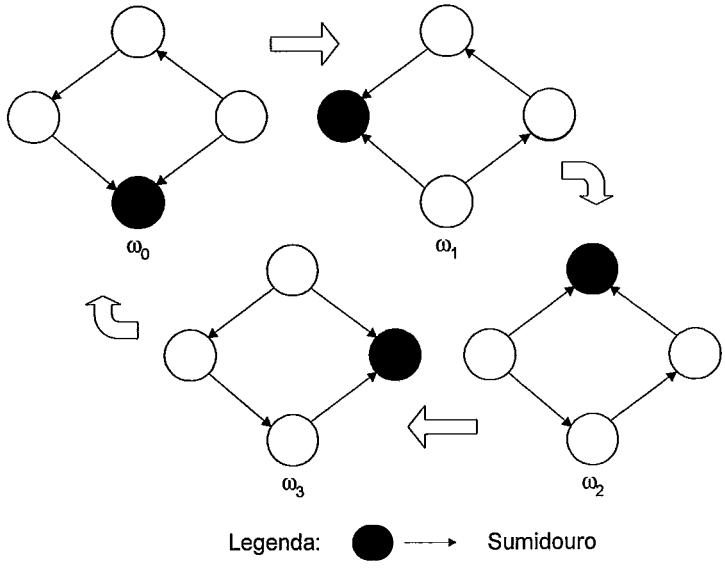


Figura 5.7: Anel com concorrência mínima, m=1, p=4.

A avaliação prévia da concorrência pode ser feita para topologias simples. Grafos em anel com uma quantidade par de nós, e grafos em árvore são naturalmente bipartidos. As concorrências máxima e mínima, em anéis de tamanho par, podem ser obtidas conforme as orientações mostradas nas figuras desta seção. Em anéis de tamanho ímpar a concorrência máxima é dada por $\gamma = (n-1)/2n$. No problema dos *dining philosophers*, a concorrência máxima é $\gamma = 2/5$. Em árvores, qualquer

orientação é acíclica, e irá produzir em regime um período $p = 2$. Deve-se mencionar que muitas das estruturas em eletrônica digital, como por exemplo as pipelines, podem ser representadas por árvores ou anéis. Também os algoritmos, em sua forma geral, podem ser representados por uma árvore (inicialização) seguida por um laço (*loop*) e um procedimento de saída do laço.

Contudo, encontrar a orientação que maximize a concorrência de um grafo genérico G é um problema NP-completo [112]. Existem heurísticas para estabelecer uma orientação acíclica em um grafo genérico [111] [115], mas a determinação da concorrência produzida por uma determinada orientação nesse grafo é um problema ainda em aberto.

Vimos que os conceitos desenvolvidos para ambiente síncrono e escalonamento guloso são também válidos para ambiente assíncrono, pois o algoritmo *SER* depende, para funcionar, apenas de informações que são do conhecimento de cada nó, não dependendo de informações globais. Também pode-se mostrar facilmente que o conceito original de tempo de operação muito pequeno e tempo de comunicação longo pode ser transformado em um modelo onde parte do tempo de comunicação fica atribuído ao próprio tempo de operação do nó.

5.3.3. Topologias Especiais

Vimos aplicações do algoritmo *SER* para topologias em anel. Sabemos contudo, que esse algoritmo pode ser aplicado a qualquer topologia, pois suas propriedades estão relacionadas apenas com a vizinhança de um nó, não havendo restrições de ordem global.

Um caso especial é representado por grafos bipartidos, que apresentam a maior concorrência possível para sistemas com recursos compartilhados, ou seja $\gamma(\omega) = ½$. Dentre os grafos bipartidos, um caso especial é o grafo em árvore. Uma árvore é por definição acíclica, e qualquer orientação inicial das arestas irá produzir um sistema de máxima concorrência. Um exemplo pode ser visto na Figura 5.8.

Dentre as árvores existe o caso particular onde a árvore tem um único ramo (*strip graph*). Nesse tipo de grafo a seqüência de eventos ocorre de forma linear, de acordo com a disponibilidade de recursos, podendo a concorrência variar entre os valores mínimo e máximo conforme a orientação inicial das arestas. Essa é uma forma possível para representar pipelines.

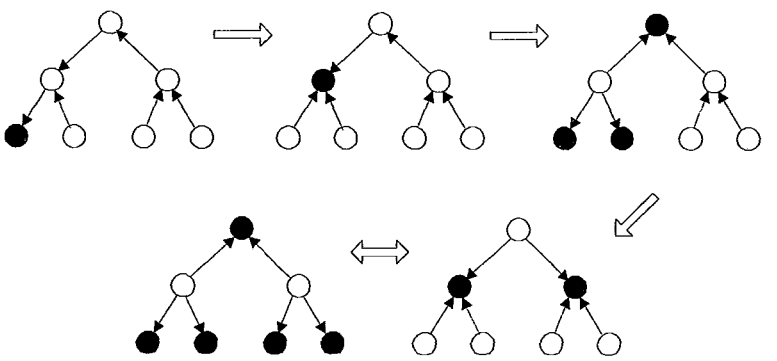


Figura 5.8: Topologia em árvore

5.3.4. Reversão Múltipla de Arestas

Para o desenvolvimento do algoritmo *SER* verificamos que o conceito de *fairness* é fundamental. Uma extensão desse algoritmo foi feita com o intuito de se estabelecer prioridades no acesso aos recursos compartilhados, isto é, o acesso deixa de ser *fair.*

A motivação para um acesso com prioridades advém do problema dos filósofos Zen, descrito anteriormente. Com o algoritmo *SMER – Scheduling by Multiple Edge Reversal,* é possível atribuir freqüências de acesso aos recursos compartilhados que são diferentes para cada processo. Além disso, o novo algoritmo apresenta características que impedem a criação de *deadlocks* ou mesmo de inanição.

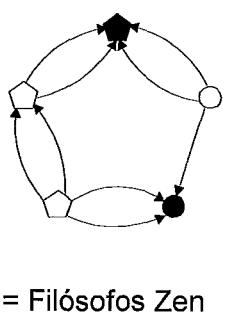


Figura 5.9: Filósofos Zen e outros – Apresentação

Para criarmos o multigrafo *M* que representa o problema dos filósofos Zen fazemos um mapeamento onde os nós representam filósofos, e as arestas representam os talheres. Teremos então, como uma das situações possíveis, a orientação mostrada na Figura 5.9, onde 3 filósofos Zen e 2 filósofos normais sentam-se à mesma mesa.

O algoritmo *SMER* é uma generalização do *SER* e funciona de maneira semelhante. Na realidade, o algoritmo *SER* corresponde a uma situação onde cada conjunto de dois nós do multigrafo é interligado pela mesma quantidade de arestas. Neste caso o multigrafo pode ser reduzido a um grafo simples.

Também é necessário dizer que um multigrafo é dito acíclico caso a sua decomposição em grafos simples sempre produza grafos acíclicos. Dessa forma vemos que ciclos existentes entre nós vizinhos não afetam a execução do algoritmo.

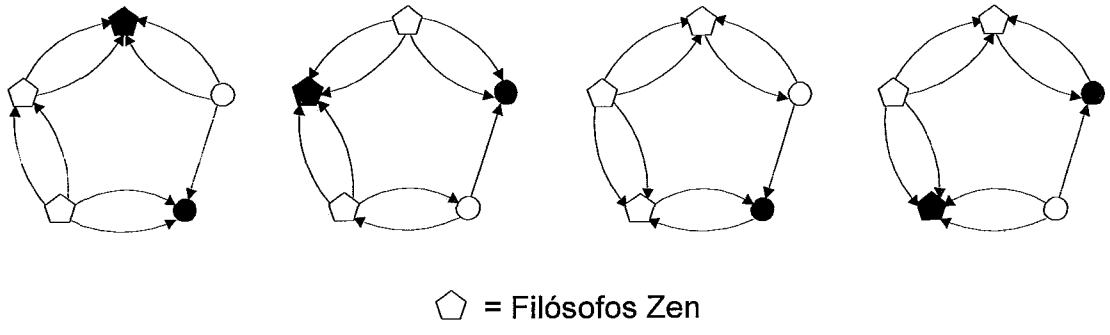


Figura 5.10: Filósofos Zen e outros – Solução

Uma solução para o problema dos filósofos Zen pode ser vista na Figura 5.10. Após a operação cada nó reverte apenas as arestas correspondentes aos recursos utilizados. Lembramos que o fim da refeição de um filósofo Zen provoca a reversão de dois pares de arestas. O fim da refeição dos filósofos normais, no entanto, provoca a reversão de apenas um par de arestas. Verificamos também que, para esse exemplo, existe uma fase onde a concorrência fica limitada à operação de um único nó. Assim podemos dizer que a concorrência de um ciclo eqüivale a $\gamma = 7/20$.

É interessante notar que dependendo da quantidade de recursos necessários para operar e da quantidade de recursos existentes, um determinado nó pode operar duas vezes seguidas. Embora isso contrarie o princípio de alta carga, existem aplicações onde essa operação pode ser desejável. Por exemplo, em um sistema de tempo partilhado, um determinado processo poderia utilizar dois *time-slices* em seqüência. Nas aplicações desta tese esse tipo de funcionamento não é desejável, e a garantia de que ele não vai acontecer é dada pelo lema a seguir.

Lema: *Um nó i de M não poderá operar duas vezes seguidas caso exista pelo menos um vizinho j para o qual a relação* $|e_{ij}| < 2r_i$ *seja válida.*

Chamamos de r_i o grau de reversibilidade de um nó *i*. Para que um nó opere em seqüência é necessário que a quantidade de recursos ofertados por cada um de seus vizinhos seja pelo menos o dobro da quantidade de recursos compartilhados em cada operação. A prova é imediata. □

Lema: *Sejam i e j dois nós vizinhos em M. A relação* $e_{ij}=r_i+r_j-1$ *é válida para qualquer orientação inicial das arestas entre i e j que não produza deadlock* [111].

Seja um grafo *M* composto por dois nós *i* e *j*. Para que não haja *deadlock* é necessário que $e_{ij} > (r_i-1) + (r_j-1)$. Isso para evitar uma situação onde *i* receberia apenas r_i-1 arestas enquanto *j* receberia r_j-1 arestas, ficando ambos impossibilitados de operar. Para que *i* e *j* não operem simultaneamente é necessário que $e_{ij} < r_i + r_j$. Segue-se o lema. □

Esse resultado é interessante porque ele implica na existência de um período *p* para as operações de reversão de arestas entre dois nós do multigrafo *M*, pois *r* é sempre uma quantidade finita. A existência de um período para operações entre dois nós implica na existência de um período para as operações em *M*.

Para o algoritmo *SMER*, o problema de se encontrar uma orientação inicial que produza concorrência máxima também pertence ao conjunto dos problemas NP-completos, e sua solução ainda é uma questão em aberto.

5.4. Metodologia ASSERT

A metodologia ***ASSERT*** – *Asynchronous Scheduling by Edge Reversal Timing* descreve um ambiente de trabalho onde processos independentes interagem através do compartilhamento de recursos comuns. Não existe restrição, por parte da arquitetura, com relação à quantidade de processos do sistema ou à forma de comunicação entre eles. Todos os processos agem de forma independente, sujeitos apenas às restrições de exclusão mútua com seus vizinhos e à sua própria temporização [116].

O problema da geração de um sistema com essa metodologia pode ser dividido em três etapas, a saber.

1. Representação do sistema alvo.

2. Construção da rede de temporização.

3. Inicialização do sistema.

5.4.1. Representação do Sistema Alvo

Seja o sistema alvo representado na Figura 5.11, que é uma representação planar dos processos (blocos funcionais) e sua vizinhança. A vizinhança dos processos é definida a partir da existência ou não da troca de mensagens entre quaisquer pares de blocos funcionais. Consideramos, neste caso, que a simples propagação de um conjunto formado por um ou mais sinais elétricos, independentemente da definição de um protocolo específico, constitui uma mensagem. Fazemos a identificação dos blocos funcionais de acordo com a ordem de execução dos processos.

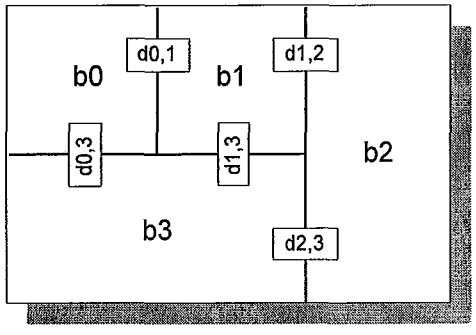


Figura 5.11: Processos ou blocos funcionais do sistema alvo.

Podemos então criar o grafo $C = (B,D)$. Este é um grafo conexo representando o sistema alvo, onde B corresponde ao conjunto de seus processos, normalmente identificados como blocos funcionais. Nesse mesmo grafo, D designa o conjunto de arestas definido pela vizinhança de B, isto é, para cada par de vizinhos b_u e $b_v \in B$ existe uma aresta $d_{u,v}$. Tomamos então o cuidado de manter a mesma ordem da identificação inicial do sistema alvo. Lembramos que neste exemplo os processos são executados em seqüência. Veremos mais adiante como modelar processos concorrentes. O grafo C correspondente ao sistema alvo da Figura 5.11 pode ser visto na Figura 5.12.

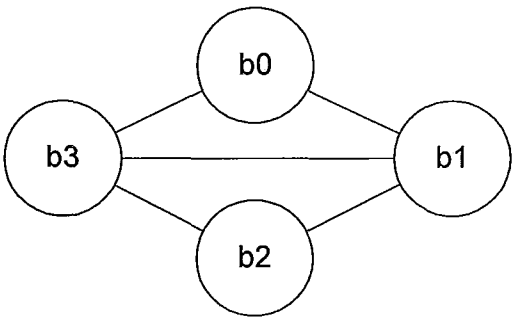


Figura 5.12: O grafo C=(B,D) gerado para o sistema alvo.

5.4.2. Construção da Rede de Temporização *G*

Seja $G = (N,E)$ um grafo representando a rede auto-osciladora alvo. A construção de G é feita a partir de C. Cada bloco funcional de C corresponde a um nó n_i em G. E a cada aresta d_j de G corresponde uma outra aresta e_j em G.

Nesta etapa, o grafo G ainda não está completo, pois ainda falta determinar a orientação das arestas.

5.4.3. Inicialização da Rede *G*

Sabemos que o grafo C é do tipo não orientado, e sabemos também que existe uma correspondência unívoca entre os b_i e os n_i. É necessário então criar uma orientação acíclica para G, o que pode ser feito de maneira bastante simples. Se, durante a construção de G, mantivermos a numeração dos nós de acordo com a ordem de execução, uma orientação acíclica apropriada pode ser obtida através do seguinte critério:

Para cada aresta que interliga os nós vizinhos n_i e $n_j \in N$, deve-se orientar a aresta no sentido de i para j sempre que i for maior que j.

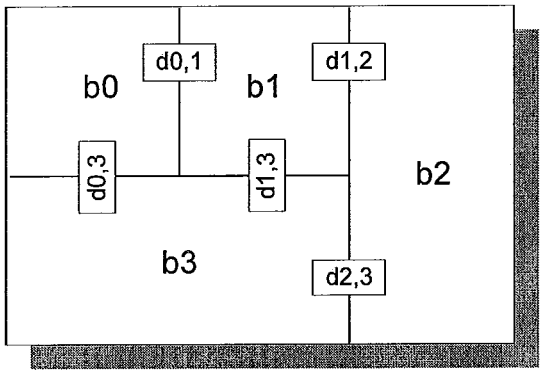
5.5. Metodologia ASSERT Multifásica

Vimos na seção anterior um procedimento para a correta geração de sistemas assíncronos com execução seqüencial. Circuitos desse tipo, mesmo quando utilizam técnicas assíncronas, têm seu desempenho comprometido pela execução serial dos processos. Muito embora o sistema apresentado permita a execução concorrente dos processos, a geração automática proposta para o grafo G implica em uma execução serializada. Vimos que a determinação da concorrência máxima é, no caso geral, um problema NP-completo. No entanto, para determinadas topologias é possível obter

esse resultado de maneira simples. Uma extensão da metodologia, que permite tratar automaticamente processos concorrentes será vista nesta seção [117].

5.5.1. Representação do Sistema Alvo

Para facilidade de entendimento da nossa proposição vamos considerar um sistema definido por um conjunto de blocos funcionais e suas interligações com a vizinhança. Cada bloco funcional compreende várias atividades serializadas, que chamaremos de fases. Esses blocos operam de maneira concorrente, respeitando restrições de compartilhamento de recursos existentes entre duas fases quaisquer de dois blocos funcionais distintos.



Edges	From	To
d0,1	b0ϕ2	b1ϕ0
d0,3	b0ϕ1	b3ϕ0
d1,2	b2ϕ2	b1ϕ1
d1,3	b1ϕ2	b3ϕ1
d2,3	b3ϕ2	b2ϕ1

Figura 5.13: Blocos funcionais de um sistema multifásico.

Seja por exemplo, o sistema alvo definido na Figura 5.13. Este sistema, além da definição dos blocos funcionais e da interligação entre eles apresenta também uma tabela que especifica as interligações entre as fases dos diversos blocos.

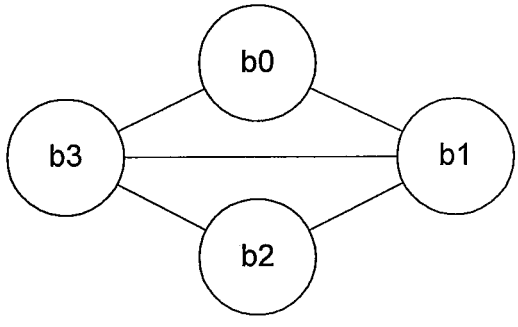


Figura 5.14: O grafo C=(B,D) gerado para o sistema multifásico.

Da mesma forma que no caso anterior, criamos um grafo $C = (B,D)$. A definição dos componentes do grafo é feita da mesma forma e o resultado pode ser visto na Figura 5.14.

5.5.2. Construção da Rede de Temporização *G*

Seja $G = (N,E)$ o grafo que representa o sistema auto-oscilante a ser construído. Cada bloco pode conter f fases, $f \geq 2$, definidas de modo a fazer com que o circuito opere corretamente.

A construção de G inicia com a definição de k anéis, R_0 a $R_{k-1} \in N$, correspondendo a cada um dos k nós de B, isto é, $k = |B|$. A quantidade de fases de cada nó é arbitrária, mas é necessariamente a mesma para todos os nós. Seja $r_{u,i}$ o i-ésimo nó do anel $R_u \in N$. Todos os nós r devem obedecer às relações $r_{u,\,i\,prev} = r_{u,(i-1) \bmod f}$ e $r_{u,i\,post} = r_{u,(i+1) \bmod f}$. Dessa forma qualquer um dos nós $r_{u,i} \in R_u$, $R_u \in N$ fica conectado tanto a $r_{u,\,i\,prev}$ como a $r_{u,\,i\,post}$, o que define a ordem de execução dentro de um anel.

Finalmente, cada nó $r_{u,i} \in R_u$, $R_u \in N$ fica conectado a $r_{v,\,i\,post}$ se e somente se os elementos dos blocos funcionais b_u e b_v são vizinhos.

A versão final de G, já contendo a inicialização das arestas será mostrada na próxima seção.

5.5.3. Inicialização da rede *G*

Para que se obtenha uma orientação acíclica adequada ao funcionamento de G, e adotando-se a restrição que não existe comunicação entre processos em uma mesma fase, é necessário que:

1. Arestas entre dois nós de um mesmo anel $r_{u,i}$ e $r_{u,j} \in N$ sejam orientadas no sentido de i para j quando $i > j$.

2. Arestas entre dois nós de anéis diferentes $r_{u,i}$ e $r_{v,j} \in N$ sejam orientadas no sentido de i para j quando $i > j$.

Um detalhe do grafo G correspondente ao anel R_0, gerado de acordo com a metodologia proposta, e que demonstra a construção das arestas orientadas dentro do anel, bem como sua interconexão aos demais elementos do sistema pode ser visto na Figura 5.15.

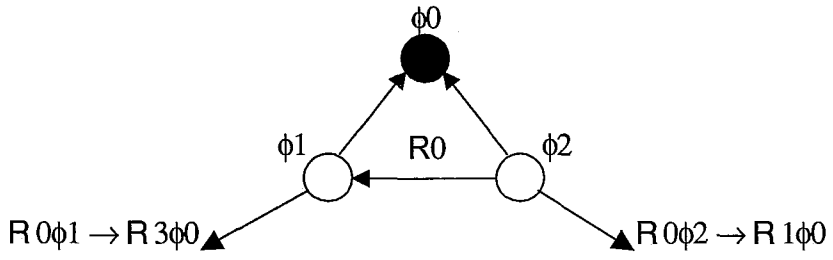


Figura 5.15: Detalhe correspondente ao anel R0

O grafo *G* completo, gerado para o sistema de blocos funcionais definido na Figura 5.13, e criado a partir do grafo *C*, definido na Figura 5.14, será visto então no exemplo da Figura 5.16.

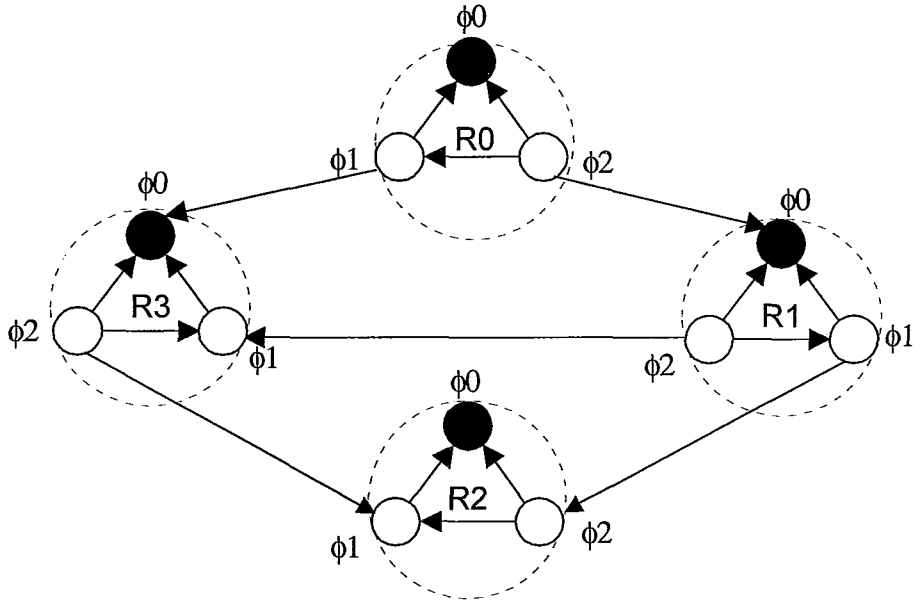


Figura 5.16: Grafo G gerado para um sistema multifásico.

5.6. Metodologia S2A

A maioria dos projetos desenvolvidos atualmente (ano 2000) utilizam metodologia síncrona. Isto é devido à cultura tradicional para o desenvolvimento de projetos, que induz o projetista a achar que apenas projetos síncronos são possíveis. A existência de uma grande quantidade de ferramentas apropriadas para esse tipo de projeto parece confirmar essa suposição.

Compreendendo que as dificuldades inerentes ao desenvolvimento de projetos inteiramente assíncronos aliada à escassez de ferramentas apropriadas para esse tipo de projeto possam fazer com que os projetistas sintam-se mais confortáveis utilizando

apenas a metodologia tradicional, procuramos desenvolver uma metodologia que combinasse vantagens desses dois métodos de projeto.

A metodologia ***S2A*** – *Synchronous to Asynchronous Methodology*, foi criada a partir da metodologia *ASSERT* e tem por finalidade facilitar a migração de projetos síncronos para projetos assíncronos. Um projeto que utiliza a metodologia *S2A* deve ser desenvolvido e simulado como um projeto inteiramente síncrono. Na etapa seguinte busca-se identificar os blocos lógicos que devem ser convertidos para lógica assíncrona. Os blocos são então reunidos por um controlador *ASSERT* que varia a duração do pulso de relógio em função da operação corrente e das características do ambiente de operação.

Dentre as vantagens do uso dessa metodologia temos uma grande facilidade para a conversão de projetos. O potencial de ganho em desempenho depende evidentemente das características das operações do sistema, podendo ser significativamente alto. O ruído interno e a emissão eletromagnética serão sempre inferiores ao do sistema síncrono original devido ao espalhamento da freqüência de relógio, que será sempre muito maior do que a de um sistema controlado por cristal.

A maior desvantagem da metodologia *S2A* consiste em não aproveitar todo o potencial proporcionado por um projeto inteiramente assíncrono. Essa limitação é compensada pelo menor tempo de projeto, uma vez que boa parte do circuito é mantida na forma síncrona original. Módulos assíncronos devem ser desenvolvidos e integrados ao projeto, mas podem ser reaproveitados em projetos subsequentes.

5.6.1. Representação do Sistema Alvo

A representação do sistema alvo é feita de modo semelhante àquela já descrita nas seções anteriores. Descrevemos inicialmente o sistema através de seus blocos funcionais B, correspondentes ao conjunto dos processos. A partir deste conjunto definimos o conjunto D, que designa a vizinhança de cada um dos B_i. Teremos então um sistema como o da Figura 5.17.

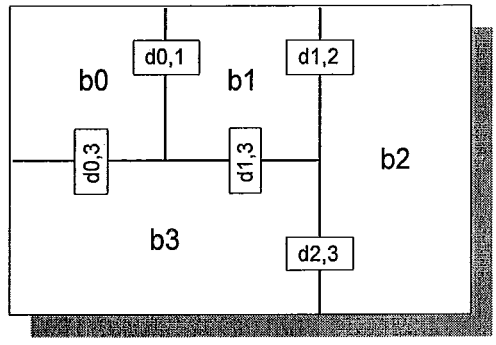


Figura 5.17: Processos ou blocos funcionais do sistema alvo.

O sistema assim definido pode ser simulado e otimizado utilizando-se ferramentas convencionais de modo a garantir o correto funcionamento do circuito quando operando em modo síncrono. Caminhos críticos para cada um dos blocos podem ser determinados, o que vai possibilitar o conhecimento da freqüência máxima de operação para o sistema nesse modo.

Dentre os blocos B_i, alguns não irão apresentar interação entre os elementos do vetor de saída. Esses blocos definem um conjunto onde a complexidade algorítmica corresponde a $O(1)$. Para esse conjunto, que inclui decodificadores, multiplexadores e lógica randômica, o desempenho depende fundamentalmente da implementação.

Os demais estágios estão sujeitos a uma complexidade algorítmica maior, podendo em alguns casos ter uma complexidade média inferior à complexidade de pior caso. Essa é uma situação típica de blocos que implementam operações aritméticas. Um somador do tipo *RCA* (*Ripple Carry Adder*), por exemplo, apresenta uma complexidade de pior caso de $O(n)$, embora sua complexidade média seja de $O(\log_2 n)$. Circuitos aritméticos, quando operando em modo síncrono, são aqueles que normalmente limitam a freqüência máxima do relógio, sendo por isso mesmo candidatos naturais para uma implementação assíncrona. Outros circuitos, como por exemplo o circuito de acesso à memória, também podem limitar o desempenho do sistema.

Podemos agora criar o nosso grafo $C=(B,D)$, onde B representa o conjunto de processos síncronos e D representa a vizinhança dos B_i. Em um circuito síncrono, a definição da vizinhança possível para um nó, corresponde a um grafo inteiramente conectado. Podemos então modificar ligeiramente o grafo criando um nó S de sincronismo, para o qual todos os processos convergem independentemente da sua vizinhança. O nosso grafo $C=(B,D)$ fica então conforme mostrado na Figura 5.18.

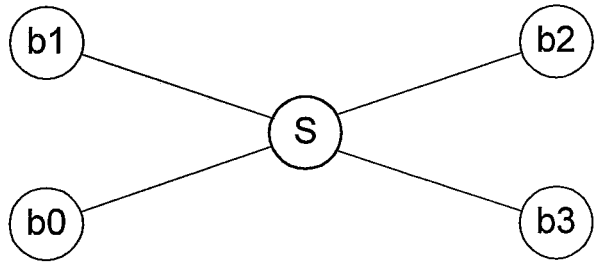


Figura 5.18: Grafo C para a metodologia S2A.

5.6.2. Construção da Rede de Temporização

A construção da rede *G* é imediata a partir do grafo *C*, conforme pode ser visto na Figura 5.19. Os nós de *G* são de dois tipos. O primeiro corresponde a apenas um nó, aqui chamado de *N0*, e que corresponde ao nó *S* de *C*. Esse nó é responsável pela geração do nível baixo do pulso de relógio. Os demais nós, designados por $N1_i$ controlam a duração do nível alto do relógio.

Para cada estágio do circuito existe uma aresta ligando $N1_i$ a *N0*. O início de uma operação é dado pela transição onde *N0* passa de sumidouro a fonte. O final do nível alto do pulso de relógio é determinado pelo último nó a terminar sua operação. Cada final de operação em um dos nós provoca a reversão da aresta desse nó, que fica apontada para *N0*. Quando *N0* volta a ser sumidouro, o nível do relógio sofre uma transição para o nível baixo. Como não existe processamento em *N0*, a duração do nível baixo do relógio depende apenas das características internas do circuito.

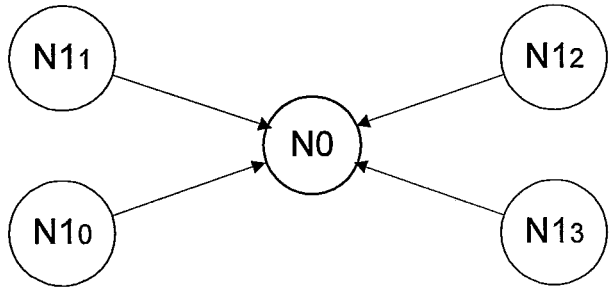


Figura 5.19: Grafo G para a metodologia S2A.

5.6.3. Redução da Rede Assíncrona

A partir da definição do item anterior deveríamos converter todos os circuitos síncronos em circuitos assíncronos auto-temporizados. Isso poderia ser feito sem necessidade de reprojetar os circuitos. Bastaria incluir circuitos com *matched delay*, o

que pode ser feito a um custo relativamente baixo. No entanto isso não é necessário. Estágios com complexidade *O(1)* têm tempo de atraso praticamente constante, isto é, o tempo de propagação é uma função das variáveis de ambiente, mas é praticamente independente dos valores das entradas.

Neste caso podemos utilizar o nó *i*, para o qual $t_{d\ min\ i} > t_{d\ max\ j}$, $i \neq j$, para fazer a temporização de todos os nós desse tipo. Caso essa relação não seja possível, escolhemos arbitrariamente um nó e aumentamos seu tempo de atraso de modo a garantir a validade da relação. Os demais nós devem ser implementados na forma assíncrona auto-temporizada.

Vemos que o desempenho do sistema fica limitado pela temporização do conjunto de nós com complexidade *O(1)*. Vemos também que não é possível iniciar uma operação em um dos nós de forma independente dos demais. Essas limitações no entanto ainda são bastante razoáveis e podem produzir um desempenho significativamente melhor a um custo relativamente baixo.

5.6.4. Inicialização da Rede

A inicialização pode ser facilmente determinada uma vez que existem apenas dois tipos de nós, o nó de sincronismo e os nós de processamento. Assim definimos o nó *N0* como sendo o sumidouro inicial.

5.7. Resumo do Desenvolvimento Teórico

O desenvolvimento teórico relacionado com o uso da reversão de arestas e sua aplicação à construção de uma rede de temporização assíncrona foram vistos neste capítulo.

Também apresentamos as propostas de metodologias para temporização assíncrona em circuitos mono e multifásicos. Outra metodologia, adequada à conversão de circuitos síncronos em assíncronos com um mínimo de esforço também foi apresentada. A validação da proposta será feita a partir de simulações de circuitos que utilizam essas metodologias. Isso será visto nos próximos capítulos.

Capítulo 6
Ferramental Assíncrono

6.1. Introdução

Os problemas enfrentados para a obtenção de ferramentas assíncronas que permitissem a síntese de sistemas fizeram com que procurássemos desenvolver nossas próprias ferramentas. Para isso procuramos um método de síntese que combinasse o desenvolvimento síncrono tradicional com aspectos assíncronos. O projeto de uma *ALU* auto-temporizada buscou criar uma ferramenta que permitisse avaliar o desempenho de circuitos assíncronos utilizando apenas circuitos lógicos convencionais. A comparação entre as versões síncrona e auto-temporizada de uma *ALU* foi feita através de simulação em *VHDL*. Cálculos mais complexos foram efetuados a partir de conjuntos de operações lógicas e aritméticas necessários para a execução da série programas de teste *SPEC*. O trabalho original foi apresentado em [118], e é aqui reproduzido de forma reduzida para facilitar o acesso a referências feitas no corpo da tese.

Neste estudo descrevemos uma ***ALU*** auto-temporizada com funcionalidade semelhante à ***ALU*** especificada na definição da arquitetura *Sun* ***SPARC***. A organização de arquitetura e as instruções executadas pela ***IU*** – Unidade de Inteiros – dos processadores ***SPARC***, são inicialmente apresentadas. As instruções executadas pela ***ALU*** são analisadas, sendo então escolhido um subconjunto para efeito de teste.

O desempenho de somadores assíncronos além de depender da implementação do circuito, também é uma função estatística dos dados de entrada. Por isso foi feita uma avaliação extra para o somador escolhido, comparando-se o potencial de ganho do tempo de execução com operandos aleatórios *(speedup)* e o obtido com aplicações reais do *benchmark SPEC*.

Uma avaliação geral do desempenho bem como dos problemas da síntese de uma ***ALU*** completa, e não apenas do circuito somador, conclui o trabalho.

6.2. A arquitetura SPARC

A arquitetura ***SPARC*** utiliza duas unidades funcionais: A unidade de inteiros *IU* e a de ponto flutuante *FPU* [119].

6.2.1. A Unidade de Inteiros do SPARC

É a unidade processadora básica, que efetua todas as instruções de controle da máquina, juntamente com as operações lógicas e as operações aritméticas relativas a números inteiros. A arquitetura da ***IU*** pode ser vista na Figura 6.1. O bloco **Y, PSR, WIM, TBR** identifica um conjunto de registradores de controle. As operações da ***ALU*** podem alterar o valor do campo *icc* do ***PSR*** conforme veremos mais adiante.

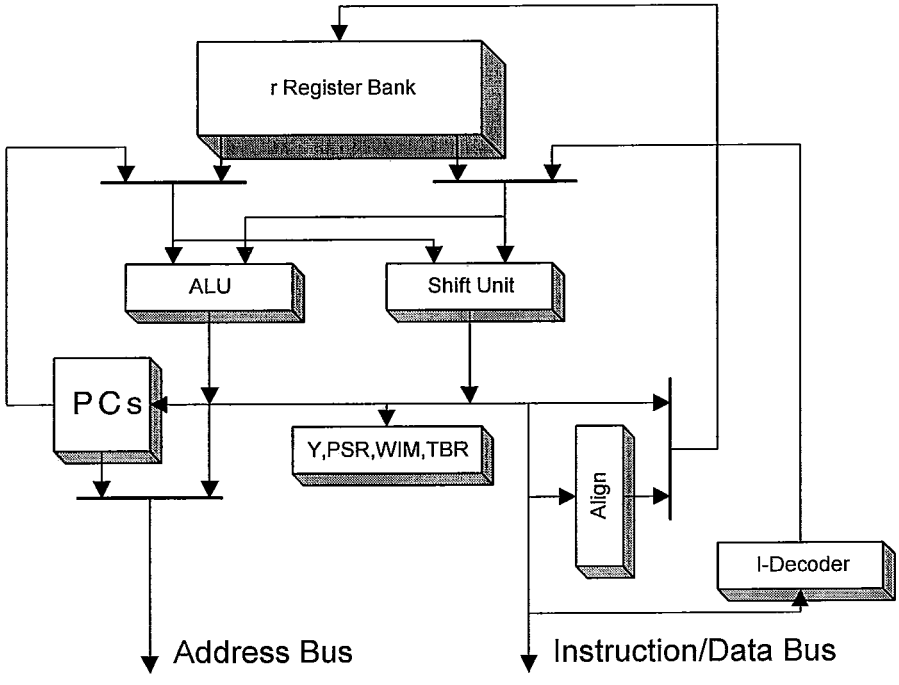


Figura 6.1: Arquitetura da Integer Unit do processador SPARC

6.2.2. FPU – Unidade de Ponto Flutuante

Executa suas instruções através de um conjunto de unidades aritméticas de ponto flutuante. A quantidade destas unidades aritméticas pode variar conforme o modelo implementado, sendo que a execução das operações é sempre feita em modo concorrente com a ***IU***.

6.2.3. Registradores da Máquina e Tamanho dos Operandos

Os registradores da ***IU*** são de 32 bits. O conjunto de registradores inteiros é estruturado sob a forma de janelas de registradores. Cada janela pode acessar 32 registradores. Existem 8 registradores globais e 24 registradores locais.

As operações de ponto flutuante podem ser de precisão simples (32 bits), dupla (64 bits) ou estendida (128 bits). Existem 32 registradores de ponto flutuante para precisão simples. Para precisão dupla ou estendida, dois ou quatro registradores, respectivamente, podem ser concatenados e acessados como um único registrador.

6.2.4. Instruções da IU – Unidade de Inteiros

1. **Instruções de Load/Store** – Fazem a movimentação de dados entre os registradores e a memória. Para o cálculo do endereço efetivo são utilizados dois registradores da ***IU*** ou um registrador e um valor imediato. A fonte ou destino dos dados pode ser qualquer registrador da ***CPU*** visível ao programador.

2. **Instruções Lógicas, Aritméticas e de Deslocamento** – As instruções deste tipo utilizam dois operandos fonte, contidos em registradores inteiros, sendo o operando destino também um registrador. Existe ainda uma instrução especial que é utilizada para montar constantes de 32 bits em uma seqüência de duas instruções.

3. **Instruções de Transferência** – Inclui as instruções usuais de *jump*, *branch*, *call* e *trap*.

4. **Instruções que Manipulam os Registradores de Controle** – Essas instruções são privilegiadas e só podem ser executadas em modo supervisor. Elas não são objeto deste trabalho.

6.2.5. Registradores de Controle e de Status da IU

- **Integer Program Counters** (*PC e nPC*) – São os contadores de programa.
- **Window Invalid Mask** (*WIM*) – Atualizado por *Save*, *Restore* e *Rett*.
- **Trap Base Register** (*TBR*) – É a base de uma tabela de vetores de interrupção.

- **Y Register** – Usado pela instrução de passo de multiplicação para gerar produtos de 64 bits.

- **Processor State Register** (*PSR*) – Composto de vários campos, dentre eles o ***icc*** que é atualizado pela ***ALU***.

O registrador ***icc*** pode ser alterado pelo resultado das operações da ***ALU*** caso a instrução executada assim o especifique. Esse registrador é formado por quatro posições de um bit, representando:

$n \leftarrow 1$ se o resultado for negativo.
$z \leftarrow 1$ se o resultado for zero.
$v \leftarrow 1$ caso haja *overflow* em operação da ALU.
$c \leftarrow 1$ caso ocorra *carry* em operação da ALU.

6.2.6. Formato das Instruções do Processador SPARC

As instruções do processador ***SPARC*** são todas de 32 bits, podendo ser agrupadas em 3 formatos. O formato 1 identifica a instrução CALL. O formato 2 é utilizado pelas instruções de desvio e também por SETHI (set immediate higher 22 bits). O formato 3 engloba as demais instruções.

As instruções executadas pela *ALU* são do formato 3, podendo serem codificadas em duas variantes. Existe uma variante específica para operações entre registradores, e uma outra para instruções onde o segundo operando é um valor imediato. Dados imediatos são de 13 bits, que são estendidos em sinal para formar um operando de 32 bits. Dessa forma a ***ALU*** sempre opera com dados de 32 bits.

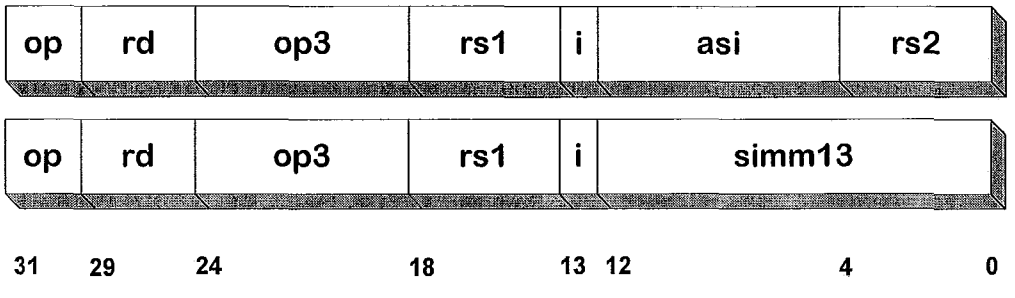


Figura 6.2: Formato das instruções da ALU

Os campos das instruções do formato 3, vistas na Figura 6.2, são:

- *op* formato da instrução
- *op3* códigos de operação das instruções do formato 3.
- *rs1* registrador fonte primário
- *rs2* registrador fonte secundário
- *rd* registrador de destino
- *i* código que identifica operações imediatas
- *simm13* valor imediato
- *asi* **Address Space Identifier**. Não é utilizado pela **ALU**.

6.2.7. Descrição das Instruções da ALU

Dentre as instruções executadas pela ***IU*** apenas as instruções lógicas e aritméticas são executadas pela ***ALU***. As instruções de deslocamento são executadas por deslocador específico, externo à ***ALU***.

As instruções básicas são: **Add**, **And**, **Or**, **Xor** e **Muls** (não considerada). Às instruções básicas podem ser acrescidas variantes para negação do segundo operando, uso do *carry* de entrada e atualização do valor do *icc*. Existem ainda instruções que não são utilizadas pela maioria dos compiladores atuais. As instruções de Tagged Add/Subtract (**Tadd/Tsub**) geram um sinal de *overflow* nos casos usuais e ainda quando os bits 0 e 1 de qualquer operando sejam diferentes de zero. Estas instruções não foram implementadas neste trabalho.

6.2.8. Instruções Consideradas

As operações da **ALU** do processador **SPARC** implementadas neste projeto estão definidas na Tabela 6.1.

6.3. Somador escolhido para o projeto

Para este projeto estamos interessados em procurar um somador que mantenha uma boa relação entre desempenho e complexidade de implementação. Nosso estudo visa conhecer os motivos que determinam o desempenho de um somador não apenas quando visto isoladamente, mas quando integrado a uma ***ALU*** e operando com dados

reais. Dessa forma, eventuais ganhos passíveis de serem obtidos utilizando-se somadores de alta complexidade de integração ficam diminuídos ao se considerar o projeto completo de uma ***ALU***, pois a latência final é composta pela latência do somador acrescida da latência específica das funções da ***ALU***.

Tabela 6.1: Códigos de máquina do processador SPARC

Mnemônico	Código op3	Descrição
Add	000000	Soma
Addcc	010000	Soma e altera *icc*
Addx	001000	Soma com *carry*
Addxcc	011000	Soma com *carry*, altera *icc*
Sub	000100	Subtração
Subcc	010100	Subtrai e altera *icc*
Subx	001100	Subtrai com *carry*
Subxcc	011100	Subtrai com *carry*, altera *icc*
And	000001	And
Andcc	010001	And, altera *icc*
Andn	000101	And not
Andncc	010001	And not, altera *icc*
Or	000010	Inclusive or
Orcc	010010	Inclusive or, altera *icc*
Orn	000110	Inclusive or not
Orncc	010110	Inclusive or not, altera *icc*
Xor	000011	Exclusive or
Xorcc	010011	Exclusive or, altera *icc*
Xnor	000111	Exclusive nor
Xnorcc	010111	Exclusive nor, altera *icc*

A estrutura típica de uma ***ALU*** auto-temporizada pode ser vista na Figura 6.3. A diferença desta ***ALU*** com relação a uma ***ALU*** síncrona consiste, a nível de estrutura, apenas em um sinal de início e uma lógica de detecção do fim de operação. Assim, podemos dizer que alguns dos fatores determinantes do desempenho de uma ***ALU*** de *n* bits são:

- Determinação do valor do *carry* de entrada, que é função da operação desejada. Isso somente é possível após o sinal de início da operação. A propagação do valor do *carry* para o primeiro estágio é feita através de um multiplexador, o que prejudica o tempo necessário para obtenção da soma.

- Negação do segundo operando. Obriga o segundo operando a passar por uma porta **XOR**, aumentando a latência da ***ALU***.
- Seleção para escolha da saída desejada. Feita por multiplexador a partir do código de operação, o que prejudica as operações mais rápidas. De qualquer forma, o sinal de saída precisa ser propagado através do multiplexador.
- Cálculo do valor dos indicadores de condição, que só pode ser feito após completada a operação.
- Latência do circuito de detecção do fim de operação.

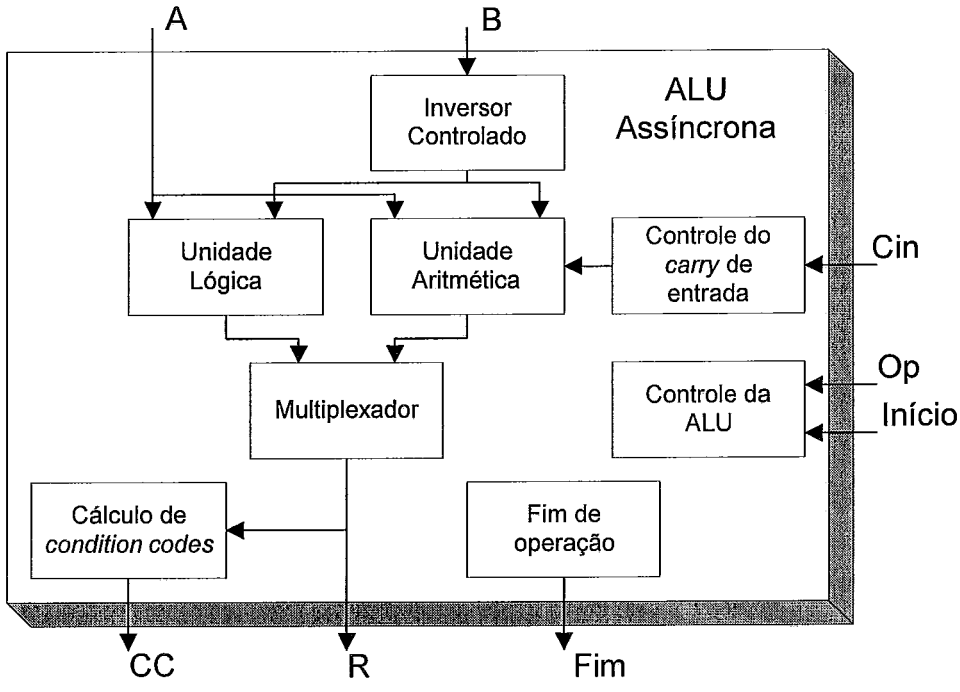


Figura 6.3 : Estrutura proposta para a ALU auto-temporizada

Escolhemos para implementação no projeto um somador do tipo ***CCA*** modificado. Neste somador os sinais de *carry* e *não carry (cease)* são utilizados em um sistema de *dual-rail encoding*, para que se possa detectar o fim de operação. Esse somador tem desempenho médio e desvio padrão melhor que o descrito em [49].

Para este somador, a lógica de detecção de fim de operação é inteiramente independente da tecnologia escolhida para integração. O somador ***CCA*** pode, por exemplo, ser inteiramente implementado com portas lógicas convencionais em

tecnologia **CMOS**. Outra vantagem, proveniente da nossa implementação da ***ALU*** utilizando somadores ***CCA***, advém do uso do mesmo circuito de detecção de fim de operação tanto para as operações aritméticas como para as lógicas. Dessa forma é possível reduzir o tempo das operações lógicas ao tempo necessário para a soma de um bit.

6.4. Potencial de Ganho de Desempenho da ALU

O tempo necessário para efetuar uma operação aritmética é função da largura da palavra. Isso não acontece para operações lógicas, pois estas são executadas inteiramente em paralelo. No entanto, para facilidade de implementação, nos sistemas síncronos em geral, as operações lógicas e as operações aritméticas simples são implementadas com o mesmo tempo de execução.

A implementação de uma ***ALU*** auto-temporizada permite que, em função dos operandos, cada operação tenha seu próprio tempo de execução. E que todas as operações lógicas possam ser executadas no tempo correspondente à soma de um bit.

O ganho em tempo de execução, ou ganho de desempenho *(speedup),* é adimensional por definição. O ganho de desempenho de uma operação assíncrona, com relação à mesma operação no modo síncrono é por definição: [50].

$$Speedup = \frac{\text{Tempo de execução}_{\text{Op sinc}}}{\text{Tempo de execução}_{\text{Op asinc}}}$$

Nos somadores ***CCA***, o fator limitante do desempenho é a maior cadeia de *carry* gerada pelos operandos. Se os operandos de uma soma fossem inteiramente aleatórios poderíamos esperar um tempo médio de execução inferior a $\lceil \log_2 n \rceil$, ou o equivalente a uma soma de 5 bits no caso de operandos de 32 bits. Temos então que o ganho de desempenho esperado para um somador auto-temporizado ***CCA*** de 32 bits, com relação a um somador ***RCA*** é, para operandos randômicos, de 6,4.

Na implementação da ***ALU*** existe ainda um ganho adicional referente às operações lógicas, que podem ser efetuadas de forma muito rápida. No entanto, uma avaliação mais rigorosa do ganho de desempenho deve ser feita com base em dados reais, considerando-se a aleatoriedade, ou a falta dela, nos operandos aritméticos, além do peso relativo das operações lógicas e do ganho extra que isto pode proporcionar. Essa análise é feita no próximo item.

6.4.1. Avaliação do Ganho de Desempenho em Aplicações Padrão

Para melhor avaliar o potencial de ganho de desempenho foi feito um levantamento estatístico das operações realizadas por um processador ***SPARC*** durante a execução de programas padrão de teste. Foram utilizados programas do Open Systems Group, da Standard Performance Evaluation Corp. (**SPEC**). Foram efetuados testes com programas da série **Cint** dos *benchmarks* **SPEC92** e **SPEC95**.

No conjunto **SPEC Cint92,** [120], foi utilizado o programa:

008.espresso	Geração e otimização de *Programmable Logic Arrays*

No *benchmark* **SPEC Cint95,** [121], foram analisados:

099.go	Implementa o jogo GO.
124.m88ksim	Simulador do microprocessador Motorola 88100
126.gcc	Compilador C baseado no GNU C versão 2.5.3
129.compress	Compressor de arquivos padrão do Unix.
132.ijpeg	Compressão e expansão de imagens em memória

Foi utilizado o Simulador de Processador ***SPARC*** [122], desenvolvido no Programa de Engenharia de Sistemas da COPPE/UFRJ. Esse é um simulador funcional que permite executar o próprio código objeto gerado em máquinas ***SPARC*** sem qualquer modificação no seu conteúdo. Nesse simulador foram incluídas instruções para gerar um arquivo contendo a relação das instruções executadas pela ***ALU***, e dos seus operandos.

O arquivo assim gerado foi posteriormente analisado por um outro programa que calcula o tempo médio de execução das operações aritméticas da ***ALU*** baseado na maior cadeia de *carry* gerada pelos operandos de cada instrução. Chamamos então de u ao tempo necessário para efetuar a soma de 1 bit. A soma de dois operandos cuja maior cadeia de *carry* seja de r bits deverá levar ru unidades de tempo.

Os testes efetuados geraram arquivos intermediários da ordem de 200MB. Este valor foi calculado em função da disponibilidade de espaço em disco nos arquivos de sistema. A maior parte dos programas ***SPEC*** precisaria de muito mais espaço em disco para completar sua execução. Isto significa que dependendo da estrutura interna de cada programa e do que eles realizam em cada fase de execução, os resultados finais podem apresentar alguma diferença nos pesos relativos das operações e conseqüentemente, das cadeias de *carry*, o que traz implicações também para o

desempenho médio. No entanto, acreditamos que a quantidade de operações amostradas já sejam significativas, pois foram pesquisadas cerca de 3.000.000 operações da ALU dentre as aproximadamente 20.000.000 executadas pelo processador SPARC em cada um dos programas.

Tabela 6.2: Quantidade de instruções da ALU nos programas SPEC

ALU	espresso	go	m88ksim	gcc	compress	ijpeg
Add	1716168	514872	344641	1824389	170998	2194647
And	629793	373	177297	385915	557	1195263
Or	2540230	1069458	532312	2536234	321345	4191421
Xor	8243	531	430	15624	146	7
Sub	12708	81510	19335	194223	32	163
Andn	20	0	10	2903	3	6
Orn	0	0	0	0	0	0
Xnor	101140	2	448	27278	100	6
Addx	0	3	177	5253	2	0
Subx	7951	5	233	6861	49	1
Addcc	6322	13885	109708	47738	7	1376
Andcc	0	0	0	0	0	0
Orcc	1903	2051	79188	86941	5	1
Xorcc	0	0	0	0	0	0
Subcc	1900364	251292	552713	2919353	48986	2591756
Andncc	3638	12973	64	12506	1	1376
Orncc	0	0	0	0	0	0
Xnorcc	0	0	0	0	0	0
Addxcc	0	0	3860	0	0	0
Subxcc	0	0	0	0	0	0
Taddcc	0	0	0	0	0	0
Tsubcc	0	0	0	0	0	0
Taddcctv	0	0	0	0	0	0
Tsubcctv	0	0	0	0	0	0
Mulscc	0	0	0	0	0	0

Para melhor avaliação desses resultados incluímos a seguir a Tabela 6.2, que mostra a quantidade de operações efetuadas pela *ALU* para cada um dos programas de teste. As instruções *Tagged* e *Muls* não foram utilizadas pelo compilador.

Vemos na Figura 6.4 uma normalização dos tamanho das cadeias de *carry* geradas pelas instruções aritméticas executadas pela ***ALU*** nos programas da série **SPEC**. O tempo de execução dessas operações varia tanto em função dos operandos quanto em função da operação desejada.

Todas as operações necessitam de um tempo mínimo para detecção de fim, que é equivalente à validação de um bit de *carry*. Por isso, operações aritméticas que geram cadeias máximas de *carry* de 0 ou 1 bits aparecem juntas na tabela. As operações lógicas estão listadas em separado.

Na tabela em pauta podemos observar que a maior parte das instruções aritméticas pode ser executada em tempo inferior a $10u$. No entanto, existe uma concentração de instruções na região de $20u$ a $32u$, que afetam fortemente o desempenho da *ALU*.

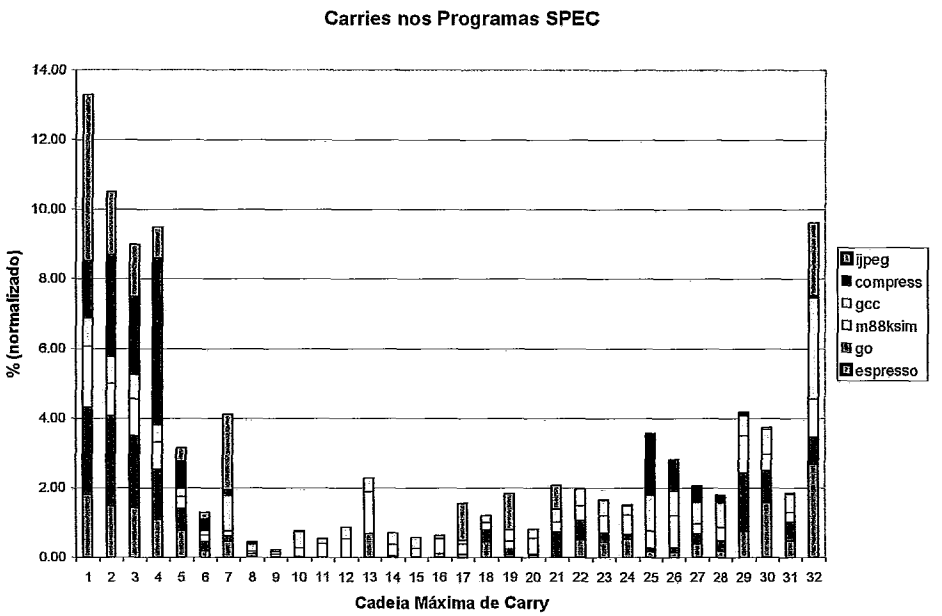


Figura 6.4: Peso relativo da execução de instruções aritméticas nos programas SPEC

A Tabela 6.3 mostra os tempos médios de operação para operações aritméticas e os benefícios que podem ser obtidos pela inclusão nessa média, das operações lógicas. Os resultados mostram que embora esperássemos um tempo médio de execução de operações aritméticas no entorno de $\log n$, ou seja, de $5u$, obtivemos, na execução dos programas de teste, um tempo médio que variou entre $8{,}27u$ e $19{,}33u$. Analisando os dados da tabela vemos que o percentual de operações com cadeias de *carry* com tamanhos variando entre 0 e 19 apresenta valores bastante próximos aos esperados. No entanto existe uma concentração de operações com cadeias de *carry* com tamanho variando entre 20 e 30 bits, que pesam de maneira significativa na apuração da média de execução.

Tabela 6.3: Operações e tempos médios de execução nos programas SPEC

ALU	espresso	go	m88ksim	gcc	compress	ijpeg
Op. Arit.	3643513	861567	1030667	4997817	220074	4787943
Op. log.	3284967	1085388	789749	3067401	322157	5388080
t_{med} op arit.	16.39u	12.73u	15.89u	19.33u	8.27u	9.26u
t_{med} total	8.62u	5.63u	9.00u	11.98u	3.36u	4.35u
$Speedup_{arit}$	1.95	2.51	2.01	1.66	3.87	3.46
$Speedup_{total}$	3.71	5.68	3.56	2.67	9.52	7.36

Vimos anteriormente que a quantidade de operações de soma com cadeias de *carry* muito longas deveria ser muito pequena, de maneira a quase não influir no resultado final. Mas as operações aritméticas também incluem a subtração, e neste caso a análise precisa ser refeita. As operações de subtração mais comuns são feitas com minuendos de pequeno valor, sendo muito comum uma operação do tipo $r \leftarrow r\text{-}1$. Essa operação é típica de controles de laço *(loop)*. Uma operação desse tipo gera uma cadeia de *carry* muito longa. Seja por exemplo, a subtração (2-1). A ***ALU*** deverá executar:

Soma aritmética	Cadeia de *carry*
00000002	00000002
FFFFFFFE +	FFFFFFFE ⊕
00000001	FFFFFFFC

Vemos que neste caso a cadeia de *carry* gerada possui 30 bits de largura.

Instruções que requerem um operando de pequeno valor são muitas vezes codificadas como instruções imediatas. As instruções que envolvem uma soma imediata com valor positivo não prejudicam o desempenho da ***ALU***. Mas quaisquer instruções de subtração ou soma com valor negativo na forma imediata irão penalizar de maneira significativa o desempenho da ***ALU***. O efeito da cadeia de *carry* gerada pelo uso de instruções com operandos imediatos na arquitetura ***SPARC*** pode ser calculado como:

Tamanho da palavra:	32 bits
Largura de operandos imediatos:	13 bits
Cadeia de *carry* mínima em subtrações:	19 bits

Fizemos uma reavaliação de desempenho do programa *espresso* eliminando as instruções aritméticas de decremento unitário. Os valores utilizados no gráfico estão disponíveis em [118]. Conforme fica evidente na Figura 6.5, essas instruções têm um peso significativo no desempenho da ***ALU***.

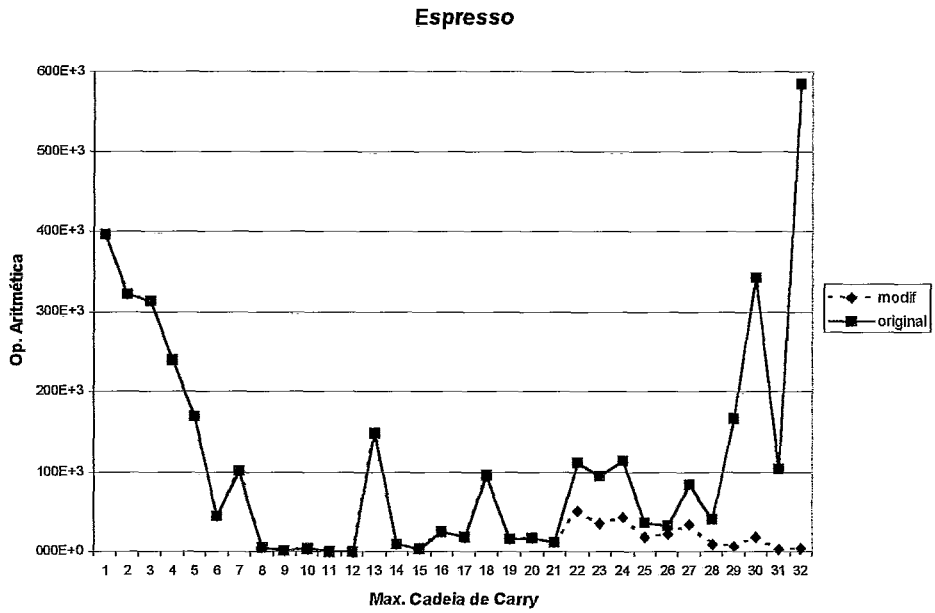


Figura 6.5: Operações da ALU no programa espresso

Os resultados mostrados na Tabela 6.4 comprovam que o desempenho da *ALU* pode ser significativamente melhorado caso a arquitetura do computador seja projetada de forma a diminuir a incidência de operações aritméticas com longas cadeias de *carry*. Propostas nesse sentido serão vistas mais adiante nesta seção.

ALU	espresso (original)	espresso (s/ r← r-1)
Op. arit.	3643513	2290043
Op. log.	3284967	3284967
t_{med} op arit.	16.39u	9.52u
t_{med} total	8.62u	4.50u

Tabela 6.4: Desempenho da ALU para o programa espresso

A existência de instruções com operandos imediatos nos códigos para a arquitetura ***SPARC***, como visto nos programas de teste, ajuda a entender o porque da quantidade razoavelmente alta de operações com grandes cadeias de *carry*. Mesmo assim, um ganho de desempenho que varia entre 1,66 e 3,86, conforme mostrado na Tabela 6.3, é ainda bastante significativo. Se, no entanto, passarmos a considerar não

apenas as operações aritméticas, mas também as operações lógicas, que geram cadeias de *carry* de tamanho um, o ganho de desempenho das execuções passa a variar entre 2,67 e 9,52. Dessa forma, o projeto de uma ***ALU*** auto-temporizada torna-se ainda mais interessante.

Quando consideramos as instruções aritméticas, vimos que se não foi possível atingir o resultado esperado inicialmente, isso foi devido à falta de aleatoriedade nos dados encontrados num programa real, aliado às características da arquitetura da máquina e da geração de código do compilador. Na *ALU* auto-temporizada, instruções de incremento são mais eficientes que instruções de decremento. Assim a geração de código para execução de um laço deve ser feita, sempre que possível, conforme o exemplo a seguir.

Máquina síncrona	Máquina assíncrona
Inicialização	Inicialização
n←N	n←0
:loop	:loop
Operação	Operação
n ← n - 1	n ← n + 1
Se n ≥ 0 loop	Se n ≠ N loop
Continua	Continua

A modificação proposta tem evidentemente impacto negativo na utilização do banco de registradores, porém isso é amplamente justificado pelo menor tempo de execução do laço. Para maior segurança, o compilador também deve garantir que a condição n = N realmente ocorra. Em linguagem de alto nível não deve ser possível alterar os valores de N e do incremento do laço *(loop)*.

6.5. Projeto Básico de Arquitetura da ALU

O projeto de ***ALU*** auto-temporizada que estamos apresentando não tem por objetivo uma eventual utilização na arquitetura ***SPARC*** atual. Contudo, sua aplicação em processadores de uso geral, do tipo ***RISC***, é perfeitamente viável se o projeto do processador for baseado em lógica assíncrona. O conjunto de instruções da ***ALU*** do processador ***SPARC*** é ao mesmo tempo simples e suficientemente poderoso para equipar processadores de alto desempenho.

Para facilitar a verificação do projeto utilizamos o mesmo conjunto dos sinais de comando da ***ALU*** encontrado nos códigos de instrução do processador ***SPARC***.

Contudo, esses sinais podem ser facilmente modificados de forma a atender outros códigos de operação, caso isso seja proposto em um nível mais elevado de projeto.

Reiteramos que as especificações da arquitetura ***SPARC*** serviram de base apenas para a especificação da ***ALU*** auto-temporizada. A implementação dessa ***ALU*** propositadamente não segue as convenções de *hardware* utilizadas internamente em um processador ***SPARC***. Convencionamos que os sinais transmitidos pelos barramentos são de lógica positiva. Da mesma forma, os sinais gerados e encaminhados ao barramento são gerados em lógica positiva. Isso nos permite trabalhar com lógica positiva no interior da ***ALU***.

Todos os sinais externos à ***ALU***, tais como os provenientes dos registradores de uso geral e os enviados para os flip-flops de condição, são considerados apenas como entradas e saídas da ***ALU***. Consideramos também, para efeito de simulação, que ao receber o sinal de habilitação (*start*), os sinais dos operandos já se encontram estabilizados na entrada da ***ALU***. O sinal de fim de operação refere-se à estabilização dos dados na saída da ***ALU*** acrescido do atraso da lógica de detecção de fim. Não foi considerado o eventual tempo de propagação do sinal em um barramento.

6.5.1. Arquitetura da ALU

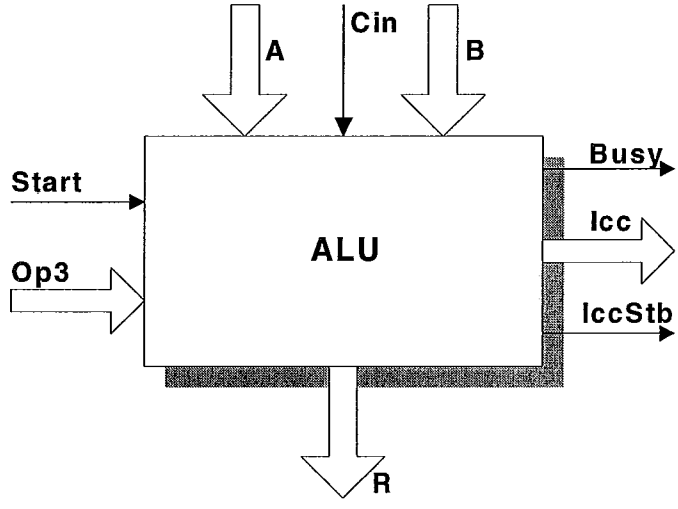


Figura 6.6: Diagrama da interface da ALU

A ***ALU*** auto-temporizada recebe como sinais de entrada o código de operação, dois operandos de 32 bits e um bit de *carry*. Os sinais de saída são compostos pelo resultado em 32 bits, um *carry* de saída, e os sinais necessários para acionamento dos

flip-flops de condição. Toda operação depende de um sinal de *start*, e gera um sinal de *busy* durante o processamento dos dados. O sinal de *start* deve ser gerado a partir de uma requisição de transferência de dados do estágio anterior que garanta a estabilização do vetor de entrada.

6.5.2. Unidade Somadora Auto-temporizada

Os somadores da Figura 6.7 calculam a função $S_i = (A_i \oplus B_i) \oplus C_{i-1}$.

O bloco de propagação de *carry* utiliza lógica com *dual rail encoding* para detectar o fim da operação. Ele implementa as equações a seguir.

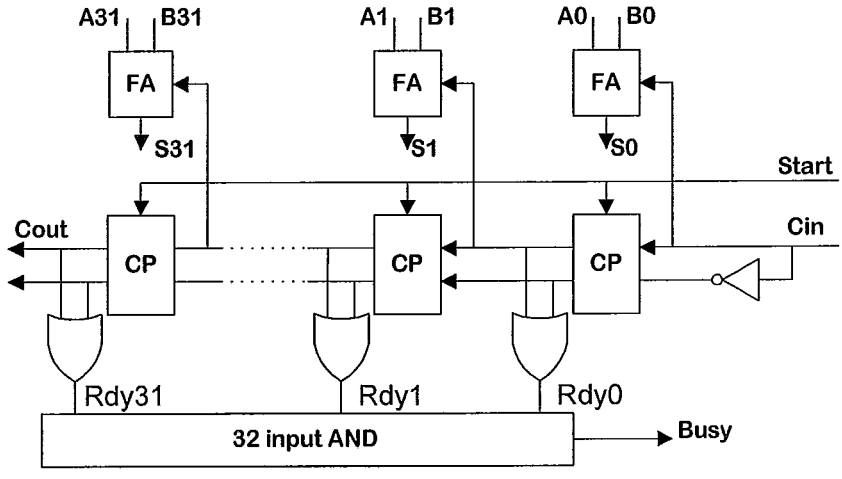


Figura 6.7: Diagrama de blocos do somador auto-temporizado

$$C_n^1 = G_n^1 + C_{n-1}^1 P_n \qquad (6.1)$$

$$C_n^0 = G_n^0 + C_{n-1}^0 P_n \qquad (6.2)$$

onde:

P_n	*Propagate* – Propaga o *carry* que vem do estágio anterior
G_n^0 e G_n^1	*Generate* – Gera os sinais *carry* e *não carry* (cease).
C_n^0 e C_n^1	*Carry out* – *Carry* de saída do estágio n.
C_{n-1}^0 e C_{n-1}^1	*Carry in* – *Carry* de entrada do estágio n.

Essas equações foram modificadas para permitir que o bloco *CP* recebesse um sinal de *start* e que ele implemente a lógica que inicializa os valores de todos os *carries* com o valor FALSO (0). A existência de um sinal VERDADEIRO (1) nas linhas de *carry* ou na de *não carry* indica que existe um *carry* válido para esse estágio. A Figura 6.8 detalha a implementação do bloco *CP*.

A detecção de fim numa operação aritmética é feita pela observação da propagação do sinal de *carry*. Quando todos os somadores sinalizarem o término de suas somas individuais a operação do somador estará completa.

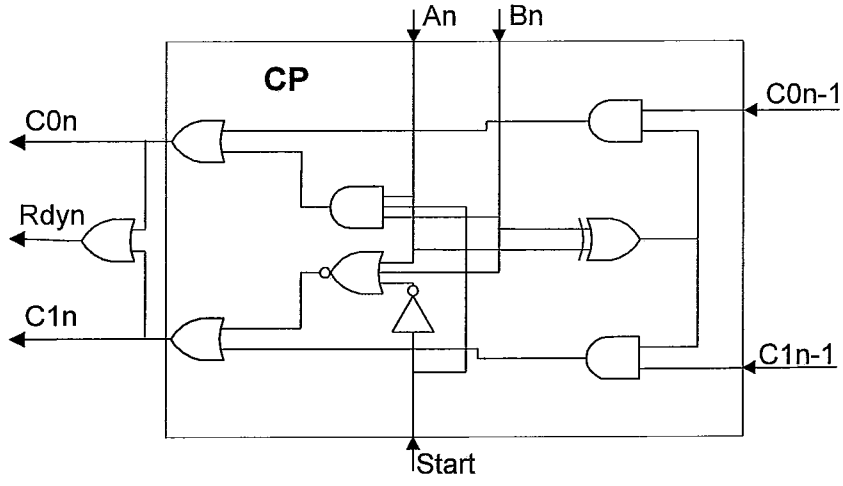


Figura 6.8: Circuito do bloco CP – Propagação de carry

6.5.3. Unidade Lógica

Ao receber o sinal de *start*, tanto a unidade aritmética quanto a lógica iniciam as operações selecionadas pelo controle. O controle da função afeta:

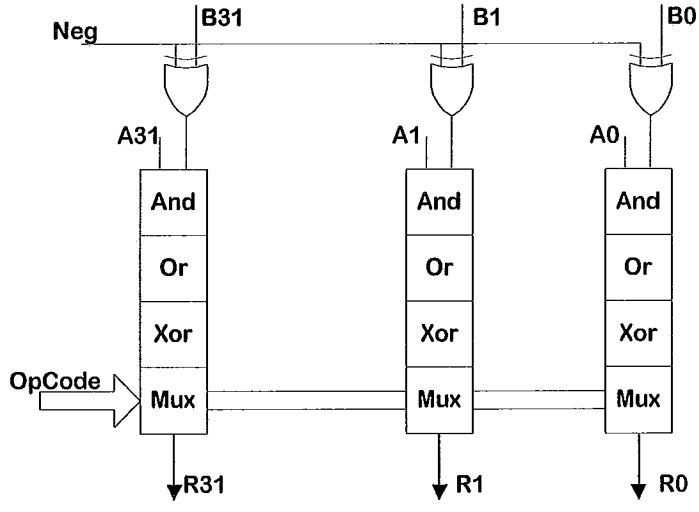


Figura 6.9: Diagrama de blocos da Unidade Lógica

- o segundo operando, que pode ter seu valor invertido para facilitar a implementação de operações lógicas complexas,

- a geração ou não de sinal para registrar novos valores nos flip-flops de condição, conforme codificado na instrução, e
- a seleção do resultado solicitado na instrução dentre aqueles produzidos pelos operadores da unidade lógica.

6.5.4. Célula Básica da ALU

As unidades lógica e aritmética da ***ALU*** são integradas em um mesmo conjunto para que se possa aproveitar parte da lógica que é comum a ambas as partes. Vimos anteriormente que o somador é formado pela equação $S = (A \oplus B) \oplus C$. O primeiro termo é aproveitado na síntese da unidade lógica. Um mesmo multiplexador de saída é utilizado para os sinais provenientes das unidades aritmética lógica.

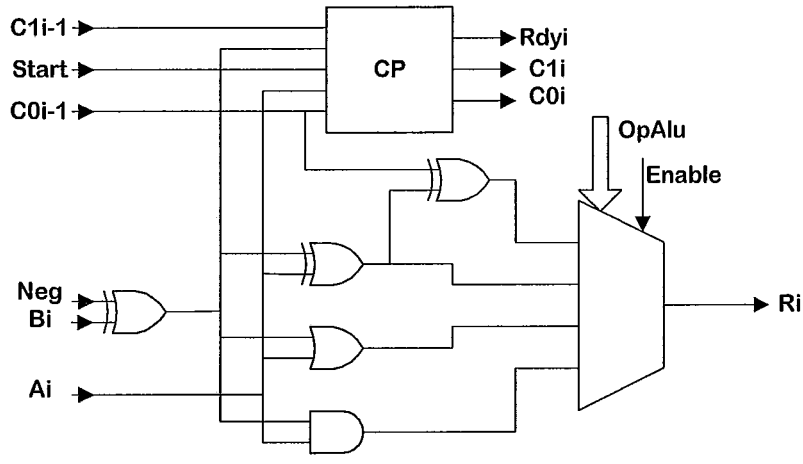


Figura 6.10: Célula básica da ALU

6.5.5. Síntese de uma ALU completa

A partir da célula básica da Figura 6.10 é possível compor uma ***ALU*** com qualquer tamanho de palavra desejado, mantendo sempre a mesma funcionalidade da ***ALU*** da arquitetura ***SPARC*** de 32 bits.

Como as cargas de entrada (*fan in*) de alguns sinais das células básicas, como **Neg, OpAlu** e **Enable** estão ligadas em paralelo, a geração desses sinais na unidade de controle da ***ALU*** tomou por base um *fan out* apropriado para 32 bits. Caso seja necessário compor uma **ALU** com capacidade para operandos de mais de 32 bits, os diversos *buffers* de saída da unidade de controle devem ser revistos. Da mesma forma,

as árvores de detecção do fim de operação e do resultado zero devem ser redimensionadas em função do tamanho da palavra.

Uma *ALU* genérica, com *n* bits pode então ser composta como na Figura 6.11.

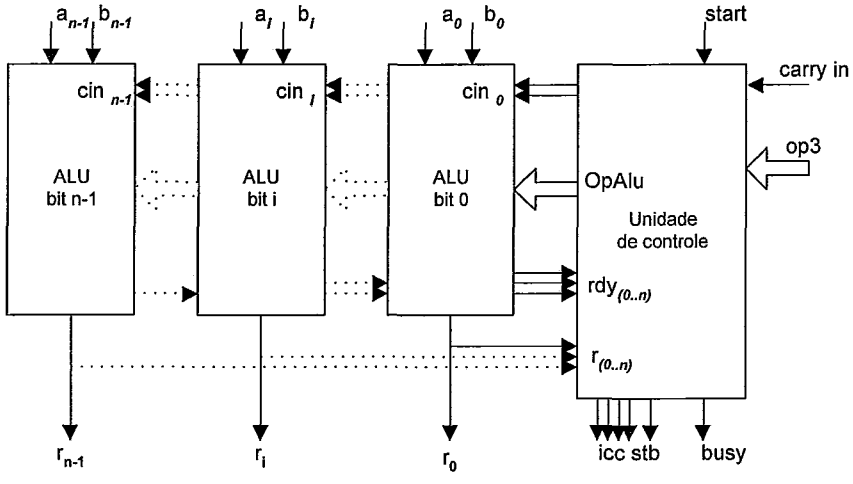


Figura 6.11: Diagrama de blocos da ALU

6.6. Implementação e Avaliação do Desempenho

A linguagem ***VHDL*** foi criada com o objetivo de permitir simulações de sistemas, tendo sido padronizada pelo ***IEEE***. O padrão mais antigo corresponde ao ***VHDL '87***, que foi revisto para gerar o ***VHDL '92*** [123].

Neste projeto, a síntese do circuito foi feita de maneira convencional, e o maior objetivo do uso de ***VHDL*** é possibilitar uma análise da correção de funcionamento e da performance da ***ALU***. Dessa forma, o uso de ferramenta baseada em ***VHDL '87*** não penaliza os resultados obtidos.

6.6.1. Descrição em Alto Nível

O projeto da ***ALU*** foi simulado por compilador da V-System, versão 3.3, cujo padrão de linguagem é o ***VHDL '87***. Uma descrição dessa linguagem pode ser encontrada em [124]. Como esse compilador implementa apenas as portas lógicas And, Or e Xor, foi necessário criar as demais. As portas lógicas utilizadas nesta simulação estão na Tabela 6.5. A descrição completa dessas portas encontra-se no arquivo `newgates.vhd`, listado no Anexo 2.

Para as portas básicas foi considerado um atraso de propagação de 1 ns. Blocos lógicos mais complexos, como multiplexadores tiveram seus tempos de atraso

avaliados a partir do uso de portas Nand e Nor, cujo atraso individual é aproximadamente a metade do das portas And e Or. Portas de 16 entradas foram implementadas através de uma estrutura em árvore baseada na utilização de portas de 4 entradas. Todo o projeto foi feito utilizando-se de portas lógicas com saída restaurada. Em todas as portas, as saídas estão sujeitas a um limite de carga de no máximo 4 entradas, conforme estabelecido em [125].

Tabela 6.5: Portas utilizadas no projeto da ALU

Porta	Descrição
andg	Porta And de 2 entradas
and3g	Porta And de 3 entradas
and4g	Porta And de 4 entradas
and16g	Porta And de 16 entradas
invg	Inversor
mux4e	Multiplexador de 4 entradas com enable
mux4e3	Multiplexador 3-state de 4 entradas com enable
norg	Porta Nor de 2 entradas
nor3g	Porta Nor de 3 entradas
nor4g	Porta Nor de 4 entradas
nor16g	Porta Nor de 16 entradas
org	Porta Or de 2 entradas
or3g	Porta Or de 3 entradas
or4g	Porta Or de 4 entradas
xorg	Porta Xor de 2 entradas

Um melhor desempenho certamente pode ser obtido mapeando-se o circuito de modo a maximizar o emprego de portas lógicas complexas ou mesmo de lógica não restaurada ou ainda empregando-se um modelo de previsão de tempos de propagação para outras tecnologias.

Nosso interesse, neste caso, consiste em avaliar o ganho relativo que pode ser obtido pelo emprego de somadores ***CCA*** em substituição aos ***RCA***. Neste caso, o ganho obtido independe da tecnologia empregada, já que quaisquer ganhos obtidos na implementação irão se refletir também no somador ***RCA***.

O projeto da ***ALU*** foi feito em nível estrutural. Um somador de 1 bit foi incluído na célula básica da ***ALU***, conforme descrito na Figura 6.10. A célula básica, por sua vez foi replicada 32 vezes. A seguir foram adicionados circuitos para detecção de fim de operação, cálculo dos valores dos flip-flops de condição e controle

operacional. A descrição completa dos circuitos implementados encontra-se no arquivo `aalu32.vhd`, listado no Anexo 2.

6.6.2. Verificação do Funcionamento

Para efeito de validação do projeto todas as operações básicas da *ALU*, inclusive com suas variantes, foram testadas com operandos escolhidos de modo a explorar o funcionamento dos diversos componentes da *ALU*. A seguir apresentamos, sob a forma de gráficos, os resultados correspondentes às operações mais significativas. Lembramos que os códigos de operação estão descritos na Tabela 6.1. Todos os gráficos foram obtidos diretamente da simulação do circuito.

Addxcc – Foi escolhida uma operação de soma, com *carry* de entrada, fazendo também a atualização dos flip-flops de condição.

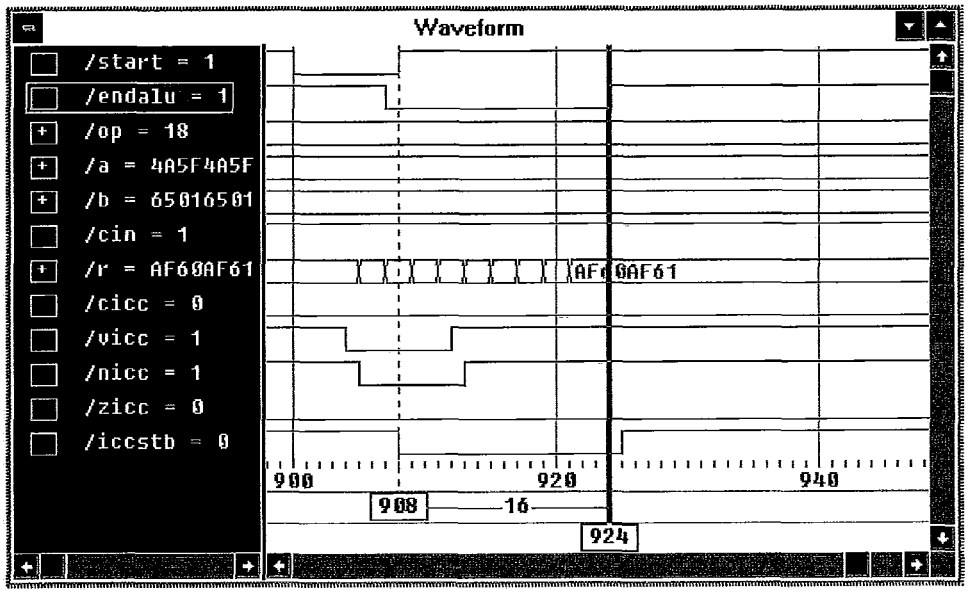


Figura 6.12: Soma com carry e atualização de icc.

Conforme pode ser visto na Figura 6.12, os operandos da soma foram escolhidos de forma a se obter um *overflow,* o que também se manifesta como resultado negativo. A borda do pulso de atualização dos registradores de condição *icc (iccstb)* é gerada uma unidade de tempo após o sinal de fim de operação. Isto não deve causar problemas, pois ela ficaria no início do tempo de sincronização para a próxima operação. Esta operação levou 16 unidades de tempo.

Sub – Mostra-se agora o teste de uma operação de subtração sem *carry* de entrada e sem a atualização dos flip-flops de condição.

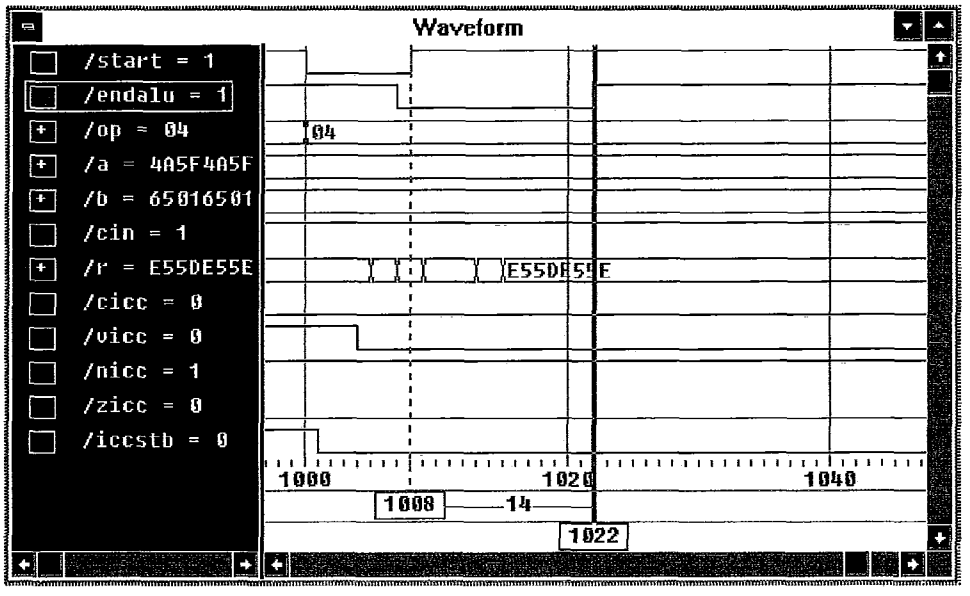


Figura 6.13: Subtração sem carry e sem atualização de icc.

Neste caso não há *overflow*, mas o resultado é negativo. Não é gerada a borda do pulso que atualiza *icc*. Esta operação levou 14 unidades de tempo.

And – Operação lógica sem atualizar os flip-flops de condição.

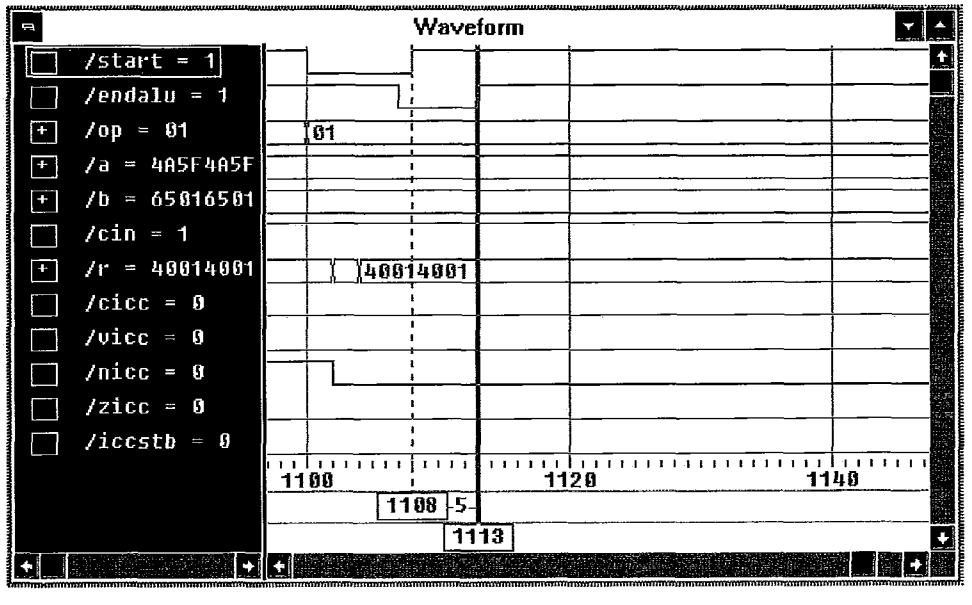


Figura 6.14: And sem atualização de icc.

Nas operações lógicas, o cálculo do valor dos flip-flops de condição é feito de maneira análoga ao das operações lógicas. Neste exemplo não há atualização de *icc*. Esta operação levou 5 unidades de tempo.

Orncc – Neste caso existe a negação do segundo operando e a atualização dos flip-flops de condição.

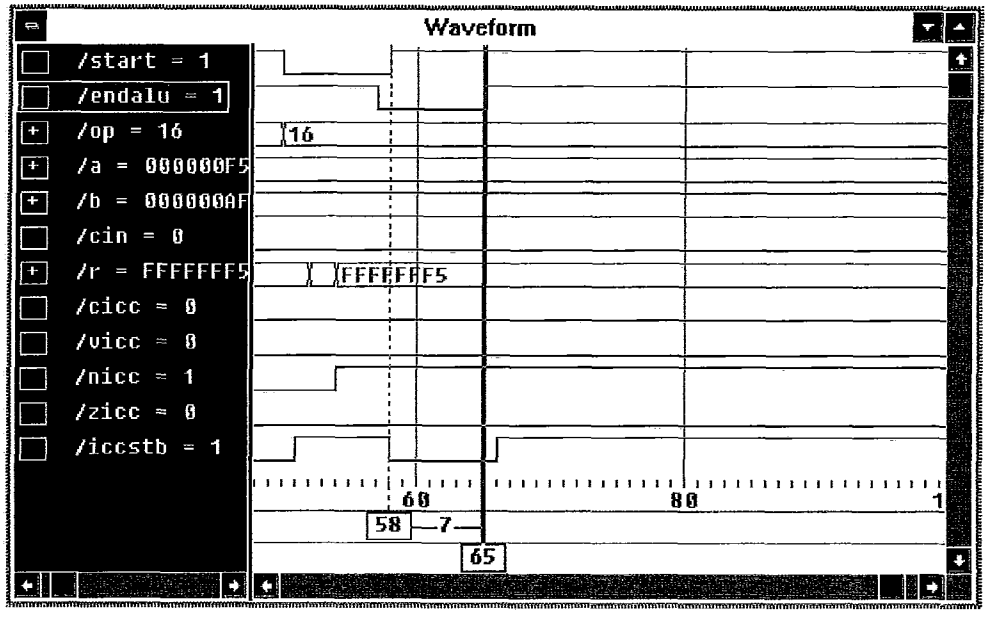


Figura 6.15: Orn com atualização de icc.

Esta operação tem um resultado negativo. A primeira borda de subida do sinal de atualização de *icc* foi provocada pela operação anterior. Esta operação levou 7 unidades de tempo.

Xorcc – A operação demonstra uma detecção de zero, fazendo também a atualização dos flip-flops de condição.

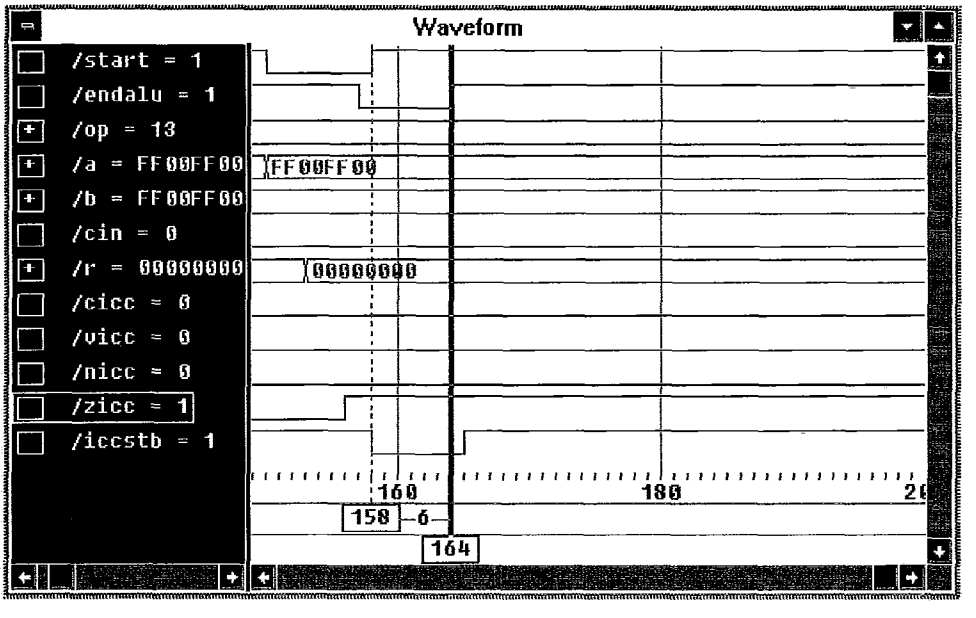


Figura 6.16: Xor com atualização de icc.

A operação mostra uma detecção do resultado zero. Esse resultado é armazenado em *icc*. Esta operação levou 6 unidades de tempo.

6.7. Desempenho

O desempenho da *ALU* pode ser expresso pela equação:

$t_{op} = t_{cy} \cdot r + t_{prop} - t_{comum}$, onde:

t_{op}	Tempo de operação da *ALU*
t_{cy}	Tempo de propagação do circuito de *carry* em cada somador.
r	Tamanho da maior cadeia de *carry*.
t_{prop}	Tempo de propagação do sinal no interior da *ALU*.
t_{comum}	Tempo comum aos circuitos de *carry* e da operação considerada.

O valor desses parâmetros pode ser determinado a partir da análise de operações típicas.

Melhor Caso – As operações mais rápidas são as operações lógicas, que são realizadas em $5u$, conforme mostrado na Figura 6.14.

Cabe aqui um esclarecimento com relação ao tempo de execução mostrado na Figura 6.15 e também na Figura 6.16, que correspondem a $7u$ e $6u$, respectivamente. Essa diferença deve-se à falta de simetria dos sinais *carry* e *start*, que obrigam o uso de inversores em um dos ramos do circuito. Assim, a detecção do fim de operação, dependendo do ramo em que ela é gerada *(cease, nstart)*, pode acrescentar até $2u$ ao tempo de execução.

Melhor operação aritmética – A melhor operação aritmética ocorre quando a maior cadeia de *carry* é, no máximo, de tamanho um. Mesmo assim, o sinal de fim de operação deve avaliar os sinais de fim provenientes de todos os 32 somadores. O tempo de propagação do *carry* para o primeiro somador aparece nesta avaliação e influi na latência da ***ALU***. Como as operações lógicas não usam *carry* de entrada e têm um procedimento de detecção de fim mais simples do que o das operações aritméticas, o melhor tempo de uma soma aritmética é necessariamente maior do que o das operações lógicas.

Neste exemplo não houve propagação de *carry*, por isso a detecção de fim se dá no ramo de ***não carry***, o que aumenta o tempo de execução em $1u$.

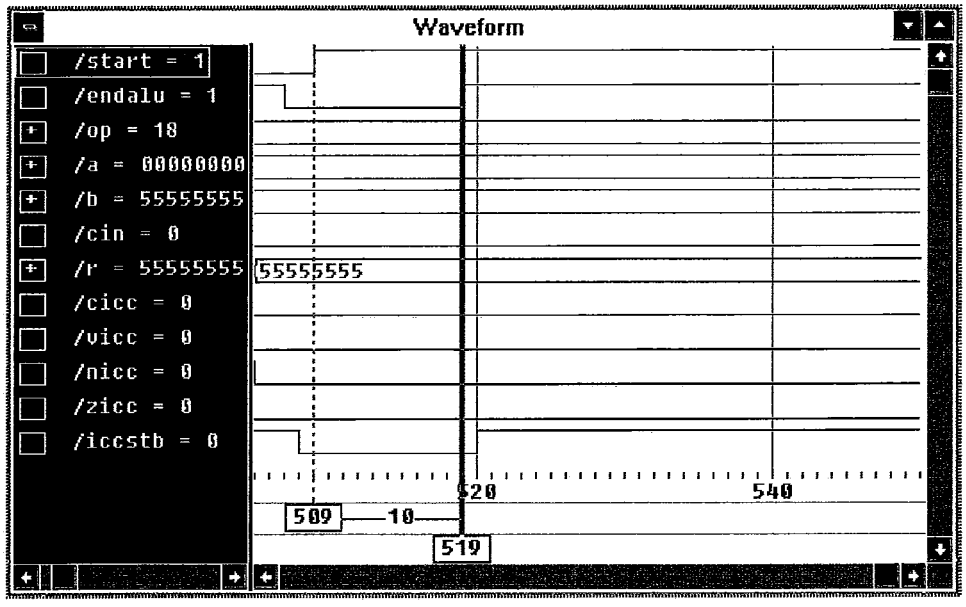


Figura 6.17: Melhor operação aritmética.

Pior caso – Conforme esperado, o pior caso corresponde a uma operação aritmética cuja cadeia de *carry*, de tamanho 32, leva ao tempo de 64*u*. Esse tempo é acrescido do tempo necessário para propagação dos sinais no interior da *ALU*.

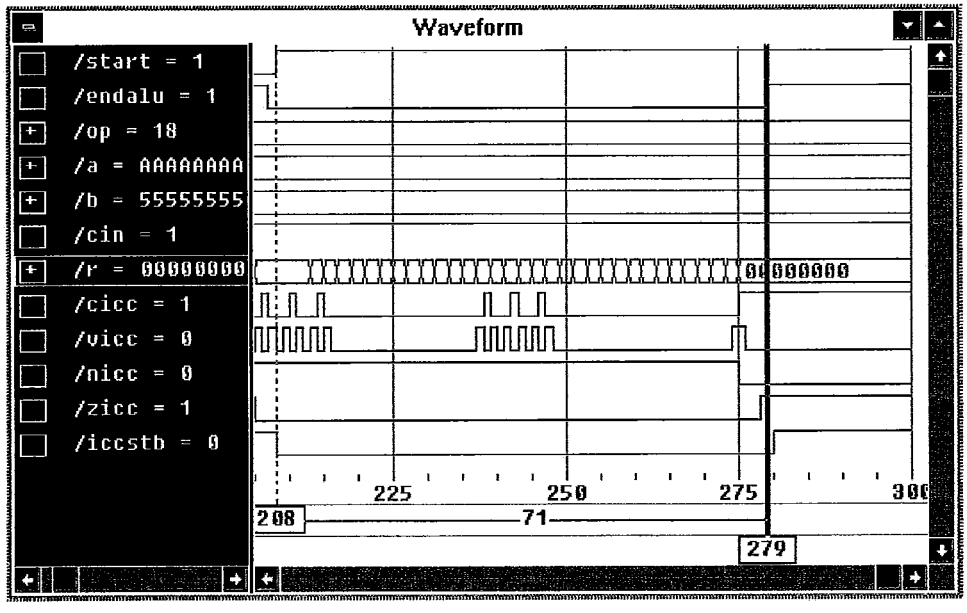


Figura 6.18: Pior caso.

Desempenho típico – O desempenho médio, de acordo com avaliação estatística feita no item 6.4.1. , corresponde a um conjunto de operações que geram cadeias de *carry* cujo valor médio varia entre 2,67 e 9,52. Tomamos então por base, para avaliação do desempenho típico, as operações que geram cadeias de *carry* de tamanho 6.

Assim, a equação que determina o tempo de execução de cada operação pode ser determinada, sendo seu valor final $t_{op} = 2r + 6$. Nesta equação, para operações lógicas temos $r = 0$, e para as aritméticas temos $1 \leq r \leq 32$.

As estatísticas do item 6.4.1. são reavaliadas em função destes resultados, obtendo-se o desempenho da Tabela 6.6, o que nos leva a um ganho de desempenho típico em relação ao somador *RCA*, de aproximadamente 4 vezes.

Tabela 6.6 : Desempenho da ALU auto-temporizada com programas SPEC

ALU	espresso	go	m88ksim	gcc	compress	ijpeg
Op. arit.	3643513	861567	1030667	4997817	220074	4787943
Op. log.	3284967	1085388	789749	3067401	322157	5388080
t_{med} op arit.	38.78µs	31.46µs	37.78µs	44.66µs	22.54µs	24.52µs
t_{med} total	23.24µs	17.26µs	24.00µs	29.96µs	12.72µs	14.7µs
Speedup	3.01	4.06	2.92	2.34	5.50	4.76

6.7.1. Análise do Desempenho

Todos os tempos medidos até aqui consideram o início do sinal de *start* como *t=0*. No entanto, para que o sistema funcione em ciclos contínuos, é necessário que o sinal de *start* permaneça no nível FALSO durante 8µs, para que ocorra a estabilização dos sinais internos à *ALU* de maneira a se evitar problemas de meta-estabilidade.

Caso o tempo de nível FALSO do sinal *start* fosse reduzido para 1µs, ainda assim o circuito forneceria resultados corretos após o tempo de propagação previsto. No entanto, o sinal de *busy* não teria tempo para atingir o nível lógico VERDADEIRO, e isto poderia ser interpretado como um resultado imediatamente válido, o que não é o que desejamos.

Apresentamos a seguir a avaliação de desempenho para o ciclo da *ALU* Auto-temporizada levando em consideração um tempo mínimo de permanência no nível FALSO para o sinal de *start* de 8µs. Apresentamos também uma avaliação da influência que o tempo de estabilização do circuito antes do início da operação propriamente dita poderia ter sobre o desempenho final da *ALU*. Para isso consideramos que o nível FALSO do sinal de *start* seria reduzido para 1µs, ou seja, o equivalente ao tempo de propagação de uma única porta lógica na tecnologia considerada. O ganho potencial para o ciclo de execução pode ser visto na Tabela 6.7.

Tabela 6.7: Ganho potencial da ALU para um tempo mínimo de estabilização

ALU	espresso	go	m88ksim	gcc	compress	ijpeg
T_{med}(8μs)	31.24μs	25.26μs	32.00μs	37.96μs	20.72μs	22.7μs
T_{med} (1μs)	24.24μs	18.26μs	25.00μs	30.96μs	13.72μs	15.7μs
Ganho Potencial	1.29	1.38	1.28	1.23	1.51	1.45

6.8. Conclusão

A partir de um circuito extremamente simples, formado basicamente por um somador *ripple carry adder,* foi possível desenvolver uma *ALU* com desempenho potencialmente superior àquela utilizada nas máquinas com arquitetura ***SPARC***.

Os problemas causados pelas operações de subtração de pequenos valores, em especial os das operações imediatas, foi devidamente analisado através do simulação extensiva de programas **SPEC** através de software especialmente desenvolvido para a arquitetura **SPARC**. Vimos que embora exista uma elevada quantidade de operações com longas cadeias de *carry* causadas por essas operações, que prejudicam o desempenho médio da ***ALU*** auto-temporizada, elas são em parte compensadas pela velocidade extremamente rápida das operações lógicas.

Também foi proposta uma modificação na geração de código de laço *(loop)* em compiladores, de modo a se obter menores tempos de execução. Essa modificação implica no uso de incrementadores para o passo de laço *(loop)*.

Para melhor desempenho, a implementação deverá procurar obter a maior velocidade possível para a função *carry*, já que o cálculo dessa função é que faz com que a complexidade máxima, em tempo, do circuito da ***ALU*** seja *O(n)*. O desempenho da função soma de cada bit do somador não é tão importante, pois como o resultado dessa função não influencia o cálculo dos demais estágios, sua complexidade é *O(1)*.

O correto funcionamento do circuito foi verificado através de simulação que testou um subconjunto de operações e operandos escolhidos de forma a procurar cobrir todos os casos críticos previstos. O desempenho final do circuito, levando em consideração os tempos médios de operação dos somadores, a influência das operações lógicas e dos circuitos auxiliares também foi analisado. O escopo desta

análise fica restrito ao funcionamento da *ALU*. Sistemas mais abrangentes, como a *CPU* ou o processador completo não foram considerados.

6.9. Resumo do Projeto de uma ALU Auto-temporizada

Vimos neste capítulo o projeto de uma *ALU* auto-temporizada inteiramente desenvolvida com lógica convencional. Apesar da simplicidade do projeto, o desempenho dessa *ALU* é bastante interessante para operandos que sigam uma distribuição randômica.

Verificamos também que mesmo em presença de operandos cuja distribuição se afasta significativamente de uma distribuição randômica, o desempenho do circuito ainda é suficiente interessante para ser levado em consideração mesmo nesses projetos.

Para o caso da *ALU* compatível com máquinas *SPARC* apresentamos uma análise dos fatores que causam a falta de aleatoriedade e apresentamos soluções para compensar a tendenciosidade dos operandos.

Todo o sistema foi desenvolvido em *VHDL*, apresentando uma alta portabilidade. Em especial, o somador da *ALU* pode ser facilmente incorporado a diversos projetos assíncronos.

Capítulo 7
Síntese com Metodologia ASSERT

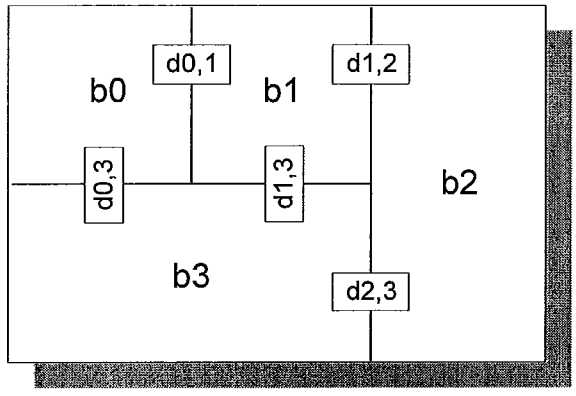
7.1. Introdução

Para um melhor entendimento das características de um sistema *ASSERT* apresentamos dois exemplos de síntese. No primeiro temos a simulação de um sistema bastante complexo. Esse sistema faz a temporização de uma rede multifásica composta por quatro nós de processamento interligados por um padrão escolhido de forma a exercitar fortemente diversas interdependências entre os nós. Os resultados desse trabalho foram apresentados em forma preliminar em [126]. Os resultados finais foram apresentados em [117]. Esse trabalho também foi utilizado para a obtenção de patente industrial referente ao processo [127].

O segundo exemplo foi desenvolvido visando aplicações em testabilidade. Trata-se da proposição de um sistema *BIST* assíncrono, que também é operado através da metodologia *ASSERT*. Os resultados obtidos a partir do sistema proposto constam de trabalho apresentado em [128].

7.2. Sistema ASSERT Multifásico

Seja o sistema descrito pela Figura 7.1. Vemos que nesse sistema existem quatro blocos funcionais, designados pelo intervalo *b0* a *b3*. Nosso procedimento não limita a quantidade de fases e nem mesmo exige que todos os blocos tenham a mesma quantidade de fases. Porém, neste exemplo, todos os blocos operam em 3 fases distintas, chamadas $\phi 0$, $\phi 1$ e $\phi 2$. A comunicação entre os blocos também fica definida pela tabela que acompanha a Figura 7.1. Nosso problema consiste em propor uma rede de temporização que, respeitando as condições estabelecidas, faça a temporização assíncrona dessa rede.



Arestas	De	Para
d0,1	b0ϕ2	b1ϕ0
d0,3	b0ϕ1	b3ϕ0
d1,2	b2ϕ2	b1ϕ1
d1,3	b1ϕ2	b3ϕ1
d2,3	b3ϕ2	b2ϕ1

Figura 7.1: Blocos funcionais do Sistema Alvo

Aplicando-se a metodologia *ASSERT*, geramos inicialmente o grafo *C=(B,D)*, conforme consta da Figura 7.2. Esse grafo mostra o relacionamento entre os diversos blocos funcionais como se eles operassem em rede monofásica.

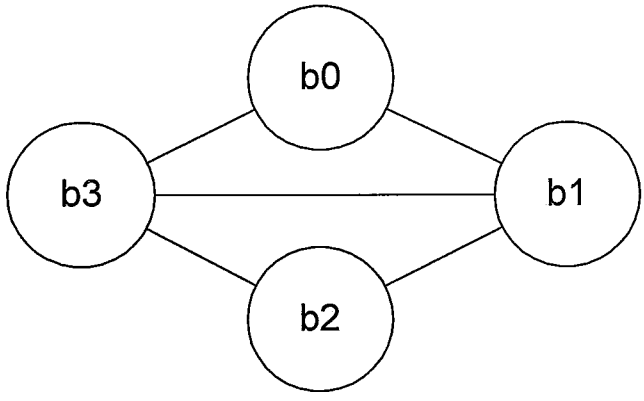


Figura 7.2: Grafo C=(B,D) para o Sistema Multifásico

O detalhamento do problema, onde aparecem as relações entre cada fase de um bloco funcional e as demais fases vizinhas, aparece na Figura 7.1. A fase inicial $\phi 0$ e a ordenação das arestas foram estabelecidas conforme disposto na metodologia.

Para cada bloco funcional as fases devem se suceder de forma ordenada, porém sujeitas às restrições de tempo proveniente dos recursos locais e também dos recursos dos blocos vizinhos. Lembramos que na metodologia *ASSERT*, cada fase de um bloco funcional corresponde a um nó de processamento. E que a topologia utilizada para seqüenciar fases corresponde a um anel.

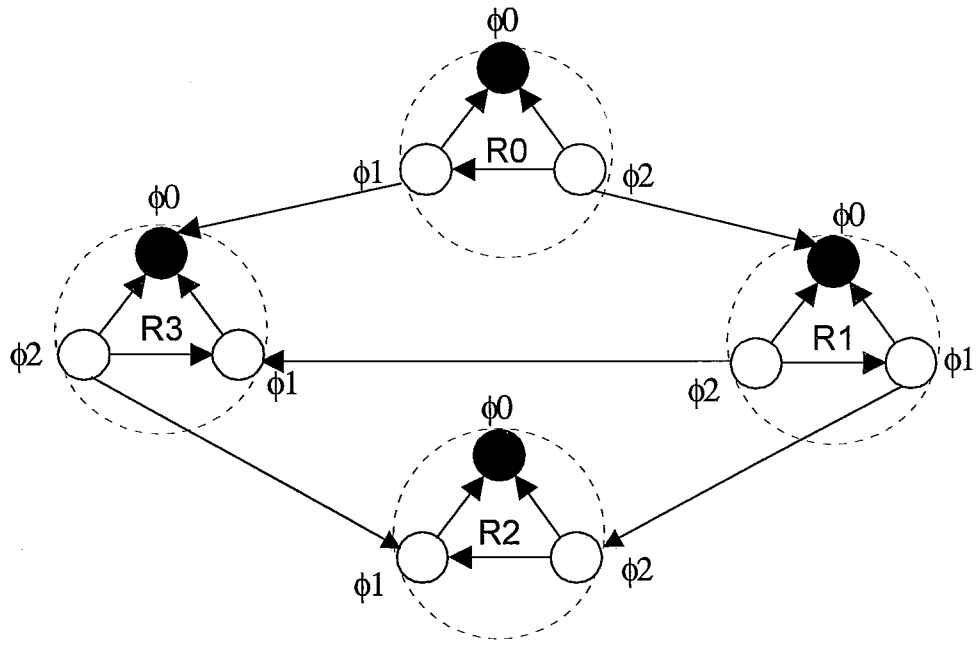


Figura 7.3: Configuração inicial para o grafo G do Sistema Multifásico

7.2.1. Controladores

A rede de temporização pode ser feita de maneira semelhante para qualquer um dos anéis descritos no problema. Assim, sem perda de generalidade, consideramos a seguir o anel *R0* da Figura 7.3, para o qual iremos mostrar a implementação da lógica de *hardware*.

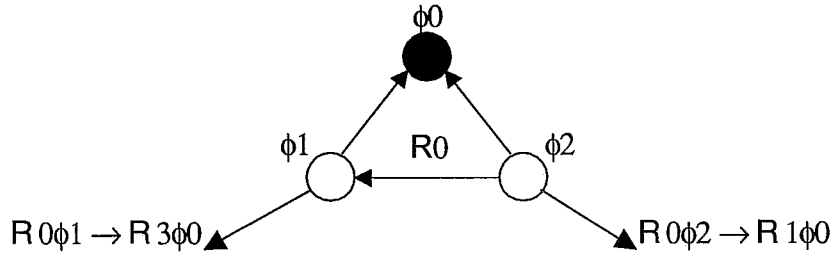


Figura 7.4: O anel R0 do Sistema Multifásico

A partir do anel *R0* da Figura 7.4, estabelecemos um mecanismo de controle das atividades do anel em função dos eventos que podem ocorrer em um nó. Esses eventos são:

1. **Uma reversão de arestas** proveniente de um vizinho que termina sua operação.

2. **Início de uma operação**, como resultado da detecção de que o nó passou a ser sumidouro.

3. **Fim de operação**, que gera um sinal que provoca a reversão de todas as arestas que apontam para o nó sumidouro.

Os eventos mencionados podem ser controlados de diversas maneiras. Escolhemos um sistema composto por dois tipos básicos de controladores, de forma tanto a procurar uma padronização para o projeto, como para procurar o hardware mínimo necessário para a operação dos controladores.

Nosso sistema compõe-se de controladores de nós e controladores de arestas. O controlador do nó atua exclusivamente sobre a lógica do nó. Ele coordena o início de uma operação. O controlador de aresta interliga um par de nós. Para definir o sentido de uma aresta associamos a ponta da aresta a uma ficha, correspondente ao nível lógico '**1**'.

A função de um **controlador de nó** consiste em avaliar a presença de fichas em um determinado nó e detectar o momento em que o nó passa a ser sumidouro, ou seja, detectar o momento em que se deve iniciar a operação do nó.

Como a interligação de dois nós é feita por uma aresta, fazemos com que cada aresta seja controlada por um **controlador de aresta,** que através da geração de uma ficha define a extremidade para onde a aresta aponta.

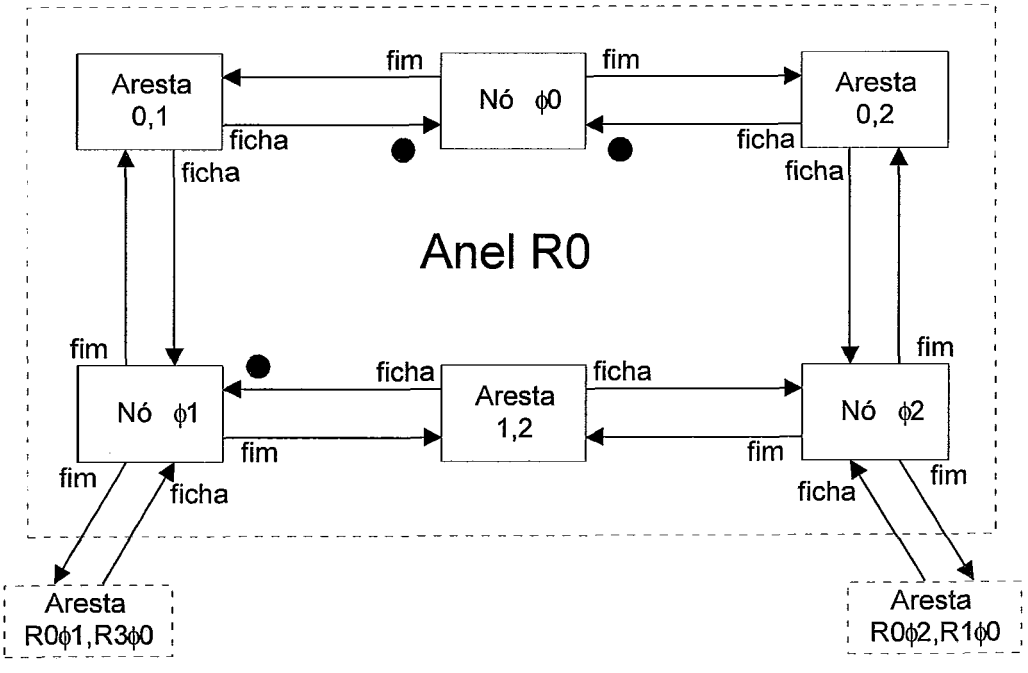


Figura 7.5: Controle de temporização para o anel R0

Terminada a operação, a lógica do processamento deve prover um sinal de fim de operação. Esse sinal é enviado a todos os controladores de arestas vizinhos ao nó, provocando uma reversão de arestas. A reversão de arestas é feita através do envio da ficha para a ponta da aresta que estava sem ficha. A representação final do sistema de temporização para o anel *R0* pode ser vista na Figura 7.5.

7.2.1.1. Controlador de Nó

Nossa implementação para o controlador de nó pode ser vista na Figura 7.6. O início de operação fica condicionado ao resultado de uma função AND de todas as fichas que chegam a um determinado nó. Quando esse resultado torna-se VERDADEIRO, ele é armazenado em um flip-flop do tipo RS. Decorrido o tempo τ_{op} correspondente à duração da operação assíncrona, o sinal de fim de operação faz com que o flip-flop desabilite o processamento do nó até que ele volte a ser sumidouro.

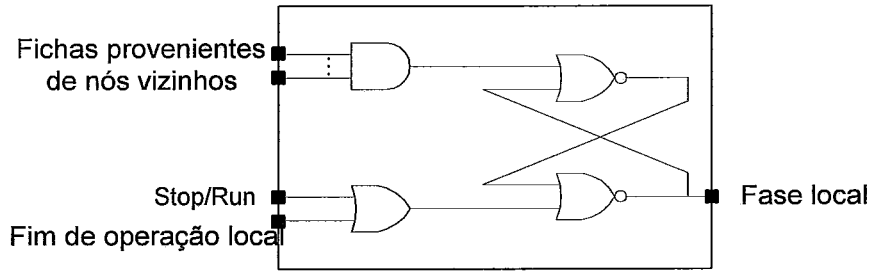


Figura 7.6: Diagrama lógico para o Controlador de Nó

O sinal de *stop/run* é utilizado durante a fase de inicialização do sistema para que possamos garantir que todas as arestas foram inicializadas corretamente.

7.2.1.2. Controlador de Aresta

Vemos na Figura 7.7 a nossa proposta para o controlador de aresta. A base do controlador é constituída por um flip-flop do tipo *toggle*, dispondo também de *clear*. A entrada de *clear* é utilizada apenas na fase de inicialização, de modo a se obter a inicialização correta das fichas. Uma porta XOR, acionada pelos sinais de fim de operação dos nós interligados, é suficiente para reverter a posição das fichas durante a operação normal.

Implementações alternativas, feitas a partir da solução clássica que emprega um flip-flop do tipo C-Müller poderiam ter sido utilizadas. No entanto, como dois nós vizinhos não podem operar ao mesmo tempo, isso não é estritamente necessário.

Talvez a principal vantagem da nossa solução seja a de utilizar células comuns, que podem ser encontradas em quaisquer bibliotecas do tipo *standard-cell*.

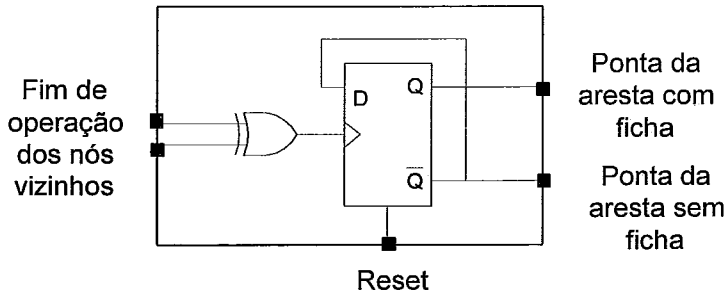


Figura 7.7: Diagrama lógico para o Controlador de Aresta

7.2.2. Simulação

O circuito completo foi logicamente simulado através do programa Opus executado em máquina Sun. Em nossas simulações adotamos um tempo fixo de operação que é o mesmo para cada uma das fases de um anel, mas que é diferente para cada anel. Dessa simulação não constam variações de ambiente.

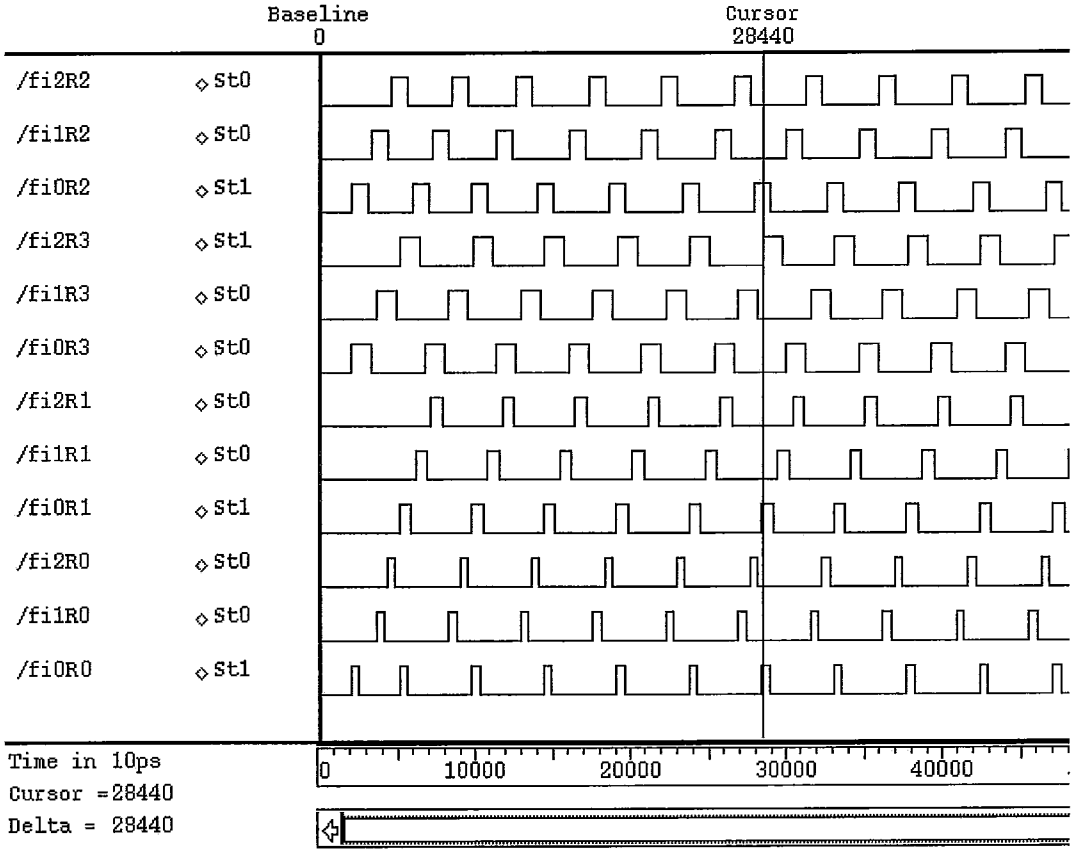


Figura 7.8: Ativação das fases do Sistema Multifásico

Os resultados dessa simulação podem ser vistos na Figura 7.8. Alguns dos resultados obtidos são mostrados a seguir.

1. Muito embora tenhamos definido a fase $\phi 0$ como a fase inicial, após o primeiro ciclo de operação as diversas fases $\phi 0$ do sistema serão executadas de acordo com a disponibilidade dos recursos, podendo ou não ocorrer simultaneamente em todos os blocos.
2. No sistema simulado, mesmo a primeira ocorrência da fase $\phi 0$ do anel *R1* precisa esperar o término da fase $\phi 2$ *de R0,* isto por especificação de projeto.
3. As fases da cada anel são executadas em seqüência. Porém, o intervalo entre elas é ditado pela disponibilidade dos recursos necessários para a operação.

Mostramos que o sistema de temporização aqui apresentado atende às especificações de projeto. Sua simplicidade permite uma implementação de baixo custo, o que torna factível desenvolver sistemas assíncronos complexos. Também mostramos que a formação de hierarquias é viável, pois os componentes de cada anel podem ser considerados como um nível hierárquico inferior que define um super nó correspondente ao bloco funcional do anel.

Evidentemente essa implementação tem um custo em *hardware* e em performance. A aplicabilidade do processo depende em grande parte da granularidade do projeto a ser desenvolvido. O limite da granularidade é dado pela condição de funcionamento da máquina de Huffman em modo fundamental, ou seja, as entradas não podem variar até que as saídas estejam estabilizadas. Nos casos em que essa condição é mantida, isto é, nos casos em que o funcionamento do sistema é garantido, é sempre interessante verificar a alocação de tempo necessária ao controle do processo em relação à alocação de tempo para o processo controlado.

Deve-se, no entanto, mencionar que a metodologia apresenta fácil escalabilidade e independência de topologia. Vimos também que a metodologia garante uma temporização correta e a ausência de *deadlock.*

7.3. Sistema BIST Assíncrono

Quando um circuito funciona em modo síncrono, a existência de um relógio global permite a implementação de técnicas como o *scan-path* e o *BIST* a um custo relativamente baixo. Essas técnicas são possíveis devido à existência de estados que são definidos para toda a máquina. Em outras palavras, um pulso de relógio faz com que toda a máquina avance exatamente um estado.

Porém, quando o sistema é assíncrono, a inexistência desse relógio global faz com que tenhamos apenas uma ordenação parcial dos estados, tornando impossível a aplicação direta das técnicas conhecidas. Conforme vimos no Capítulo 3, a pesquisa nessa área ainda é incipiente, e os resultados obtidos estão voltados para aplicações específicas. Nossa proposição visa encontrar uma solução para o funcionamento de sistemas *BIST* assíncronos.

Analisando o funcionamento de sistemas *BIST* síncronos vemos que um circuito de controle exercita o Circuito Sob Teste *(CUT)* utilizando dados fornecidos por um Gerador de Vetores de Teste *(TPG)*. A saída do *CUT* é encaminhada a um Analisador da Resposta ao Teste *(TRA)*. Essas operações devem ocorrer no intervalo de um pulso de relógio.

No caso de um circuito assíncrono auto-temporizado, o *CUT* depende de informações explícitas para o seu funcionamento.

Para que ele possa operar de forma eficiente, é necessário que:

- o sistema de controle forneça o sinal de início da operação; e que
- o *CUT* forneça um sinal de fim de operação.

Para uma determinada tecnologia, o tempo necessário à operação irá depender das condições de ambiente e possivelmente dos operandos de cada ciclo. É por isso que não podemos estabelecer a priori um tempo de execução para o circuito. No caso de circuitos assíncronos, toda métrica é estabelecida por estatística.

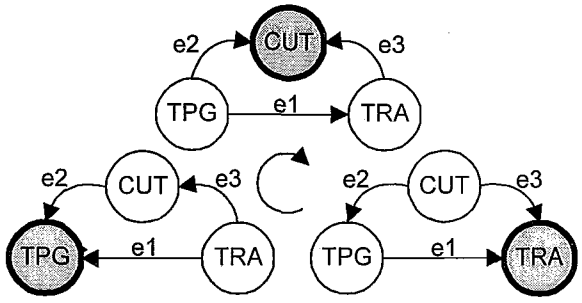


Figura 7.9: Configuração ASSERT para um sistema BIST

Utilizando-se a metodologia *ASSERT*, o seqüenciamento das operações de um *BIST* pode ser descrito através da Figura 7.9. Vemos na figura que o *TPG* é inicializado como sumidouro. A seguir o *CUT* opera, e seus resultados são encaminhados ao *TRA*. O sistema então retorna ao padrão inicial, o que permite nova operação do *TPG*.

O sistema proposto executa as operações básicas de um sistema *BIST* através de um processo auto-oscilatório que deverá ser interrompido por um comando externo quando a seqüência de padrões de teste for completada. O resultado final, correspondente à assinatura do circuito, fica disponível no *TRA*. Vemos também que cada bloco do sistema *BIST* determina seu próprio tempo de operação. Isto equivale a dizer que o teste acontece à velocidade nominal prevista para a operação do circuito, sujeita contudo às variações de ambiente e do espalhamento de fabricação. Essas variações são automaticamente compensadas.

A temporização prevista para o sistema *BIST* pode ser vista na Figura 7.10.

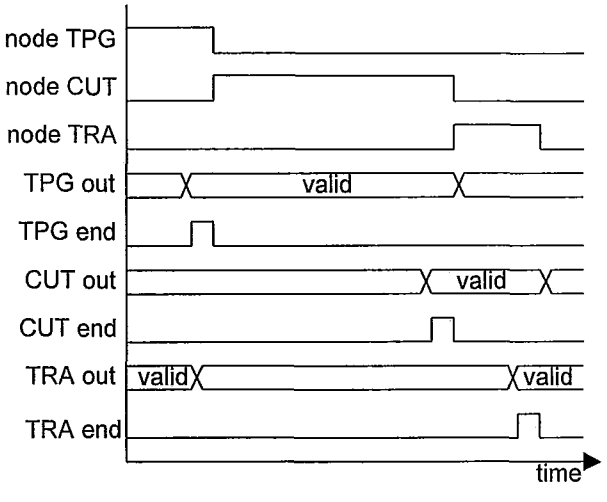


Figura 7.10: Temporização do BIST com metodologia ASSERT

7.3.1. Implementação dos Controladores

Da mesma forma que na seção anterior, implementamos o sistema de controle de temporização através de controladores de nós e de controladores de arestas. As funções desses controladores são as mesmas descritas anteriormente, isto é,

1. detectar que um nó atingiu a condição de sumidouro, comandando o início de operação e;
2. ao receber um sinal de fim de operação, reverter as arestas incidentes ao nó.

7.3.1.1. Controlador de nó

Dadas as características da implementação da metodologia *ASSERT*, podemos garantir que a detecção da condição de sumidouro irá ocorrer sem falhas desde que:

1. A porta lógica de detecção não apresente oscilações transientes na saída;
2. Os sinais de entrada não apresentem oscilações transientes em seus valores.

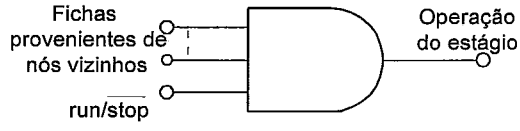


Figura 7.11: Controlador de Nó para o sistema BIST

A segunda condição é garantida pela metodologia *ASSERT*, pois um mesmo nó não pode operar duas vezes seguidas. Isto é, quando um nó recebe uma ficha, essa ficha fica retida até que o nó opere. A garantia da primeira condição é conseqüência da segunda.

7.3.1.2. Controlador de aresta

O controlador de aresta é o mesmo da seção anterior. Ele é reproduzido a seguir apenas para referência.

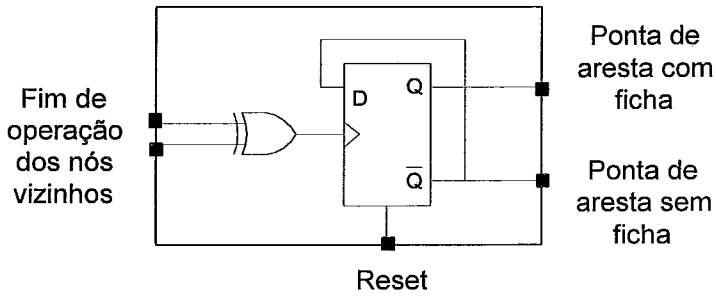


Figura 7.12; Controlador de Aresta para o sistema BIST

7.3.2. Exemplo de um CUT

Partimos da premissa que o *CUT* é um circuito auto-temporizado. Circuitos desse tipo fornecem um sinal de fim de operação que garante valores finais estáveis nas suas saídas. O início de operação fica sujeito a um gatilho disparado pelo controlador de nó. O diagrama esquemático para um *CUT* genérico pode ser visto na Figura 7.13. Para efeito de simulação utilizamos um somador *RCA* de 4 bits semelhante ao utilizado no desenvolvimento da *ALU* auto-temporizada.

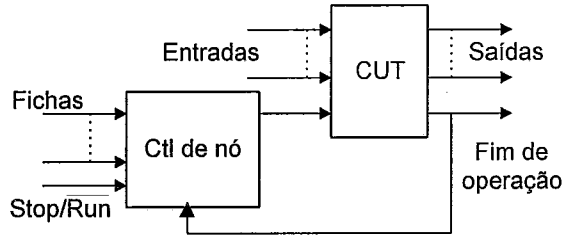


Figura 7.13: CUT genérico a ser simulado

7.3.3. Implementação do TPG e do TRA

Circuitos *TPG* e *TRA* podem ser implementados de várias formas. Nossa implementação foi feita a partir de um registrador de deslocamento linearmente realimentado *(LFSR)* síncrono. O modelo síncrono foi escolhido para que se pudesse utilizar lógica convencional. Esse tipo de circuito é bastante rápido e normalmente não é o fator limitante do desempenho do *BIST*. Em nossa implementação, a temporização do *LFSR* é feita por *matched delay,* a partir de uma simulação elétrica. O circuito resultante pode ser visto na Figura 7.14.

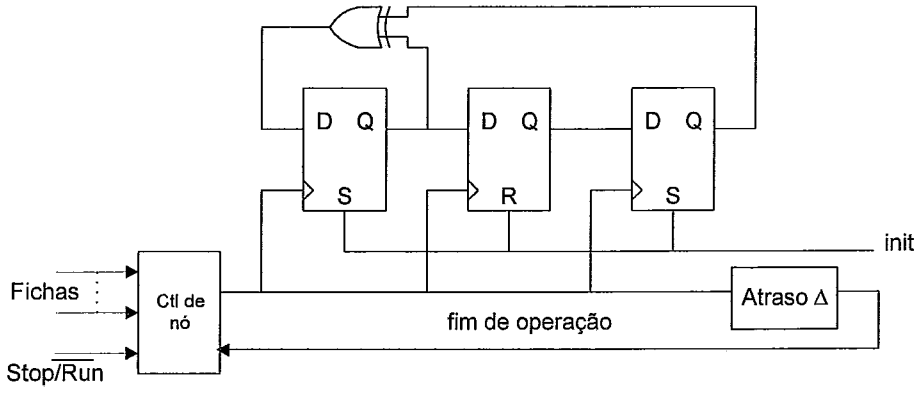


Figura 7.14: Circuito LFSR para o TPG

O circuito do *TRA* é implementado de forma semelhante ao do *TPG*. Seu diagrama esquemático pode ser visto na Figura 7.15.

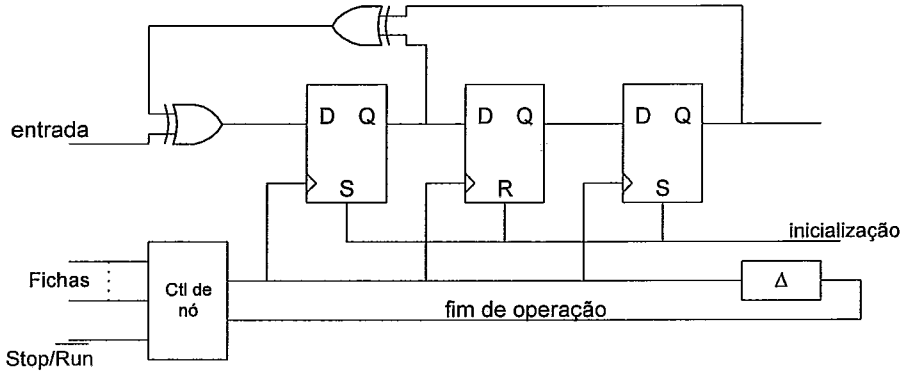


Figura 7.15: Circuito LFSR para o TRA

7.3.4. Simulação do BIST

É importante mencionar que todo o circuito foi projetado a partir de células de uma biblioteca *standard*. Assim, para completar o sistema *BIST* basta fazermos a interligação dos blocos apresentados anteriormente. Fazemos a interligação do *data path* normalmente. Em seguida as arestas são adicionadas, resultando no circuito da Figura 7.16. O funcionamento do LFSR é feito passo a passo, a cada operação.

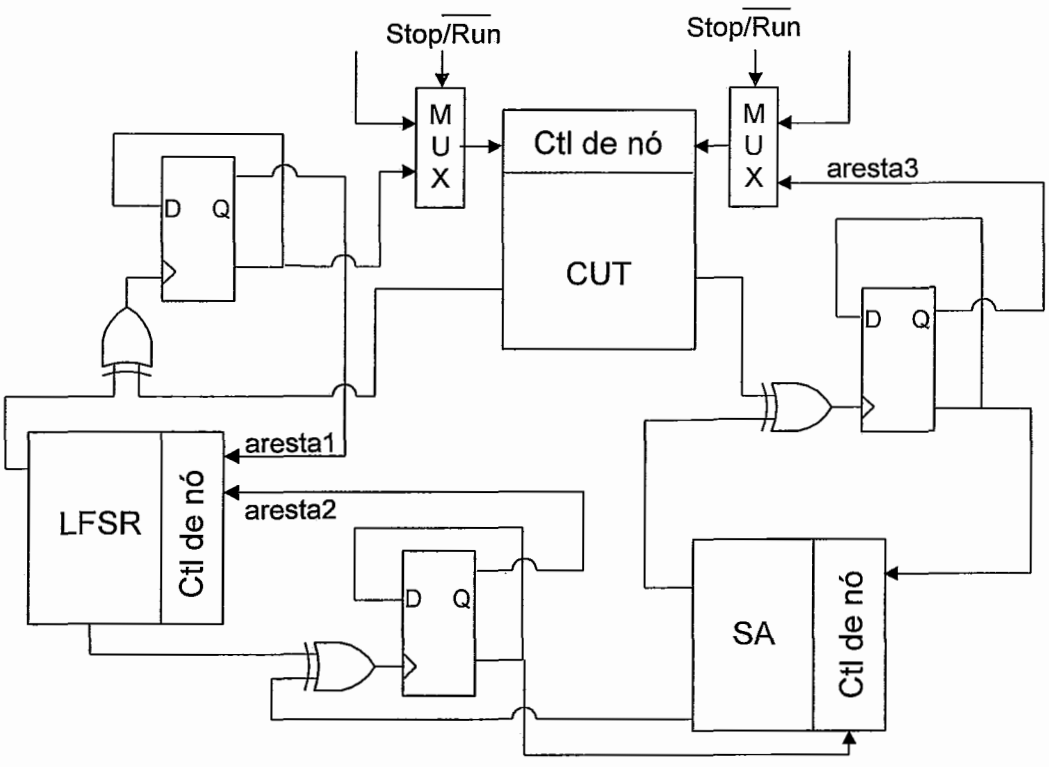


Figura 7.16: Sistema BIST completo

A operação de um *BIST* só acontece quando ele é requisitado a fazer um teste de funcionamento. Para uma implementação completa devemos incluir multiplexadores que controlem as entradas, e possivelmente, as saídas do *CUT*. No nosso projeto mostramos que o sinal *Stop/Run* poderia ser utilizado como controle de configuração dos multiplexadores e também para automaticamente dar início à operação *BIST*.

O resultado da simulação do circuito completo pode ser visto na Figura 7.17.

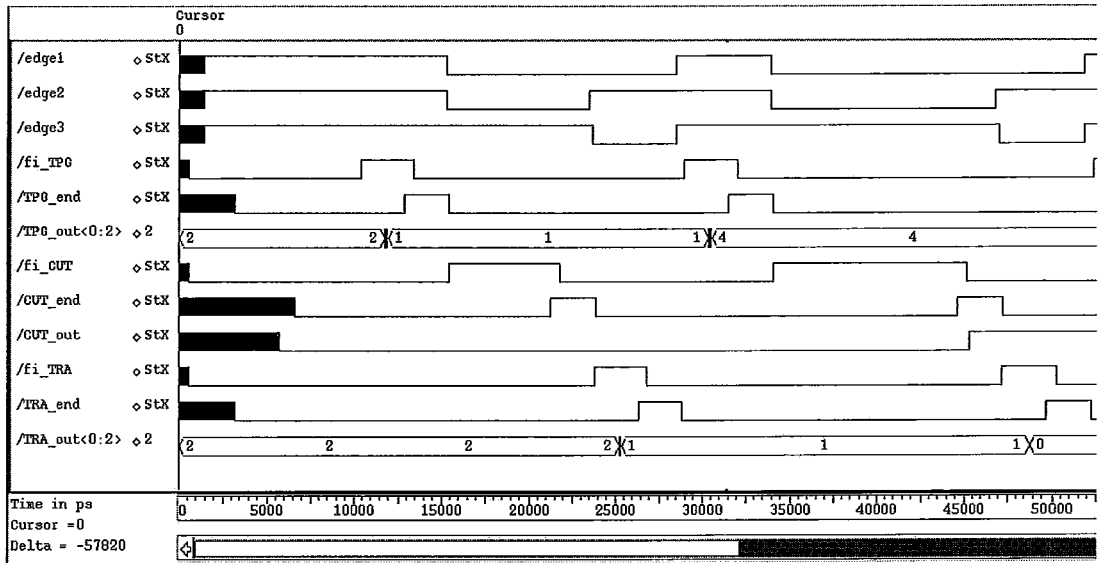


Figura 7.17: Simulação do sistema BIST com metodologia ASSERT

Após o término da seqüência de vetores de teste, o sinal *Stop/Run* deve ser acionado para reconfigurar o sistema no modo normal de operação. Uma forma possível para implementar o fim da operação do sistema *BIST* está descrito a seguir.

Qualquer que seja a implementação do *TPG*, é necessário detectar o fim da seqüência de teste. A detecção de fim pode ser utilizada como uma aresta extra no nó do *TPG*. Inicialmente essa aresta aponta para o *TPG* e permite a operação normal do sistema. Ao final do teste essa aresta é revertida. Ela só voltará a apontar para o *TPG* quando houver um novo comando de operação do *BIST*.

Uma propriedade dos circuitos projetados com metodologia *ASSERT* diz que ao se congelar qualquer um dos nós, o circuito irá parar seu funcionamento em *O(n)*, isto é, em um tempo finito. O reconhecimento de fim de teste pode ser feito no próprio nó *TPG*, quando ele voltar a receber as fichas dos seus vizinhos. Nesse momento saberemos que o *TRA* contém o valor da assinatura. O valor da assinatura pode ser comparado com um valor conhecido ou mesmo encaminhado a um supervisor de teste. Neste último caso é preciso reconfigurar o *LFSR* do *TRA* como parte de uma cadeia de *scan* assíncrona. Um exemplo de implementação de um *scan path* assíncrono pode ser visto na Figura 7.18. [129].

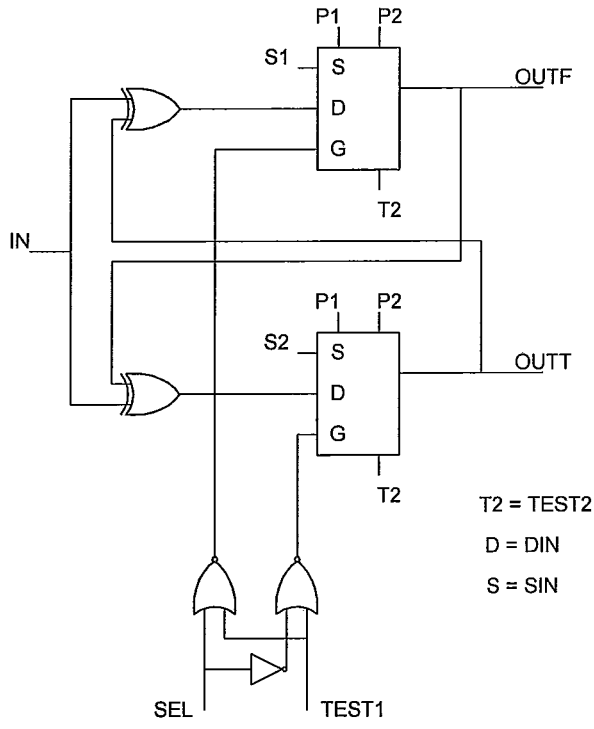


Figura 7.18: Exemplo de um scan-path assíncrono

O sinal *Stop/Run* pode ser utilizado para reconfigurar os multiplexadores, permitindo o funcionamento normal do *CUT*.

7.3.5. Aumentando o Paralelismo

Vimos na Figura 7.9, que o funcionamento do sistema *BIST* depende da ativação serializada dos nós *TPG*, *CUT* e *TRA*. Para aumentar o paralelismo, podemos fazer com que a geração do vetor de teste e a análise da assinatura ocorram simultaneamente. Por essa proposta, os nós *TPG* e *TRA* poderiam iniciar suas operações ao mesmo tempo, ao término da operação do *CUT*.

No início da operação apenas o *TPG* seria sumidouro. Após atingir o padrão oscilatório, o funcionamento do *BIST* seria como na Figura 7.19.

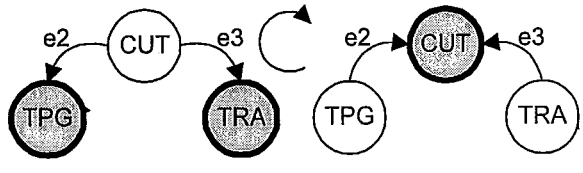


Figura 7.19: Aumentando o paralelismo do sistema BIST

A temporização esperada para um circuito deste tipo pode ser vista na Figura 7.20.

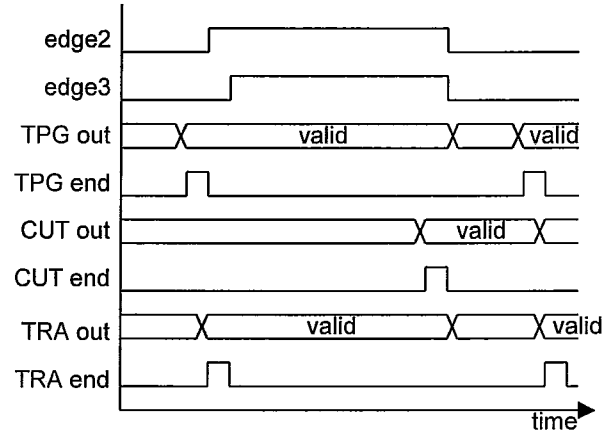


Figura 7.20: Temporização para o BIST aperfeiçoado

Conforme esperado, o diagrama de tempos se mantém quando fazemos a simulação do circuito, conforme mostra a Figura 7.21.

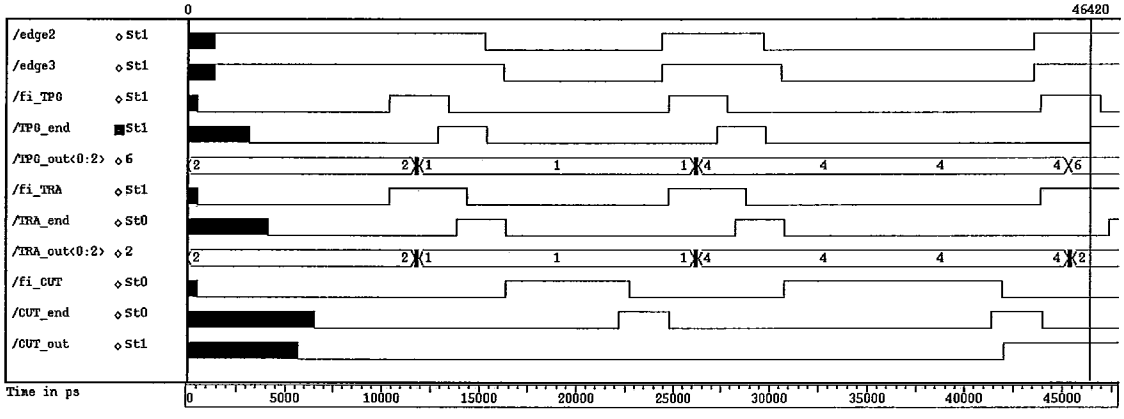


Figura 7.21: Simulação do circuto BIST aperfeiçoado

A vantagem desta nova configuração pode ser vista pela avaliação do tempo de teste. Para o sistema da Figura 7.9, o tempo de teste é dado por:

$$\text{Tempo de teste} = \sum_{i=1}^{n} \left(T_{TPGi} + T_{CUTi} + T_{TRAi} \right)$$

Para o novo sistema, o tempo de teste é dado pela expressão:

$$\text{Novo tempo de teste} = \sum_{i=1}^{n} \left(\max\left(T_{TPGi} + T_{TRAi} \right) + T_{CUTi} \right)$$

É evidente que no segundo caso o tempo de teste é menor.

7.3.6. Detecção de Falhas Topológicas

Uma falha topológica pode ser caracterizada pela maneira como ela afeta o mecanismo *ASSERT*. Falhas que ocorrem no *data-path* ficam normalmente restritas ao próprio *data-path* e não influem no funcionamento assíncrono do circuito. No entanto, em certos casos, essa falha pode impedir a geração do sinal de fim de operação.

O bloqueio do sinal de fim de operação causa o congelamento de um nó, da mesma forma que uma falha topológica nos controladores de nó e de aresta. Neste caso sabemos que o funcionamento do circuito cessa após um período de *O(n)*. Aliás, essa mesma propriedade já foi explorada para implementar o fim do conjunto de operações do sistema *BIST*. A simulação de um circuito com falha do tipo *stuck at 0* está na Figura 7.22.

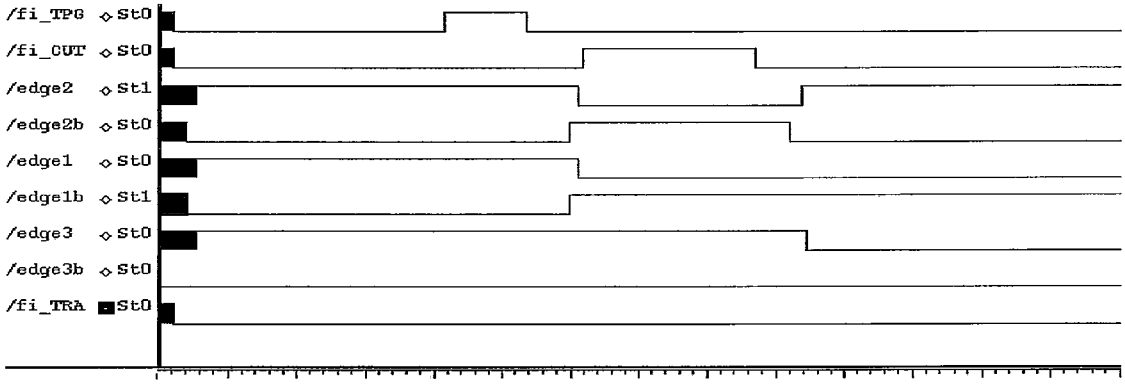


Figura 7.22: Simulação de um nó com falha do tipo stuck at 0.

O método de teste tradicional nesse caso consiste em se implementar um sistema de vigia *(watch-dog)*, que irá descartar o circuito por decurso de prazo *(time-out)*.

7.4. Resumo dos Sistemas com ASSERT

Neste capítulo mostramos duas aplicações para a metodologia *ASSERT*. A primeira delas foi baseada na simulação lógica de um sistema multifásico assíncrono. Foram desenvolvidos controladores especiais que implementam o funcionamento dos nós e arestas do sistema de temporização.

Os resultados da simulação mostram que o sistema implementado tem funcionamento correto, sendo que os custos do *hardware* adicional composto pelos controladores de nó e de aresta anteriormente descritos, são também bastante baixos. A perda de performance é função da granularidade do sistema a ser sintetizado e deve

ser vista caso a caso. Nos casos onde o uso da metodologia está voltado para aplicações que procuram uma solução para a comunicação entre áreas eletricamente distantes, a granularidade é alta e a perda de performance é pequena.

A segunda implementação também utiliza metodologia *ASSERT*. Trata-se de um sistema *BIST* assíncrono. O circuito necessário para a implementação e a simulação desse circuito, bem como suas características, são analisados. Apresentamos também uma proposta para melhorar a performance desse circuito utilizando uma topologia com maior potencial de paralelismo.

Capítulo 8
Síntese com Metodologia S2A

8.1. Introdução

Neste capítulo apresentamos dois exemplos de sistemas sintetizados com a metodologia *S2A*. O primeiro deles trata da conversão de um sistema *BIST* originalmente síncrono, utilizando essa técnica de conversão. Em nosso exemplo apresentamos um sistema de teste desenvolvido para um pequeno somador, e as simulações do circuito.

A seguir, mostramos a síntese de um subsistema assíncrono para criptografia. A partir do subsistema *SCOB*, desenvolvido no *LPC* [130], aplicamos a metodologia de conversão *S2A* proposta nesta tese. Os resultados obtidos constam, em forma resumida, do trabalho apresentado na referência [131].

8.2. Conversão de um BIST Síncrono em Assíncrono

Esta aplicação foi desenvolvida como prova de conceito para a metodologia *S2A*. Partimos do mesmo sistema *BIST* do capítulo anterior, porém considerando que os blocos componentes, isto é, *TPG*, *CUT* e *TRA*, operam seqüencialmente, e em modo síncrono. O funcionamento geral do circuito é definido na Figura 8.1.

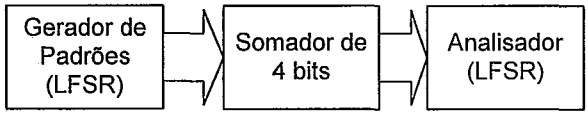


Figura 8.1: O circuito BIST síncrono

Para conversão deste circuito do modo síncrono para o assíncrono de acordo com a metodologia *S2A*, fizemos a representação do sistema através dos grafos *B*, *C* e *G*. O grafo *B* pode ser derivado diretamente da Figura 8.1. A geração dos grafos *C* e *G* também é imediata, pois conforme a metodologia, devemos criar um nó *ASSERT* auto-temporizado para cada bloco.

Aos nós operacionais acrescentamos mais um nó, que chamamos de *N0*, para fazer o sincronismo dos sinais de relógio.

Figura 8.2: Sistema BIST S2A com super nó em N1

O super nó N1 da Figura 8.2 é uma representação que engloba os nós N1-1*(CUT)*, N1-2*(TPG)* e N1-3*(TRA)*. Para simplicidade de notação, passaremos a denominar esses nós apenas por N1, N2 e N3.

8.2.1. Controladores de Nó e de Aresta

Os controladores de nó e de aresta são os mesmos que apresentamos anteriormente na seção 7.3.1. .

8.2.2. CUT: Somador RCA de 4 bits

Para simulação de um *CUT* utilizamos um somador *RCA* auto-temporizado de quatro bits. O circuito do somador foi desenvolvido a partir do somador da *ALU* auto-temporizada (V. Capítulo 6). O circuito somador utilizado tem por finalidade fornecer tempo de operação que é variável em função dos operandos. A limitação da largura permite exercitar o circuito e acompanhar seu funcionamento passo a passo.

8.2.3. TPG e TRA

Os circuitos *TPG* e *TRA* são os mesmos desenvolvidos para a seção 7.3.3. .

8.2.4. Simulação

A implementação do circuito foi feita conforme o esquema da Figura 8.3. Conforme visto anteriormente, os nós *N1* a *N3* operam de forma independente. Ao final de cada operação eles sinalizam o nó *N0*, que faz o sincronismo e provê um novo ciclo de operação.

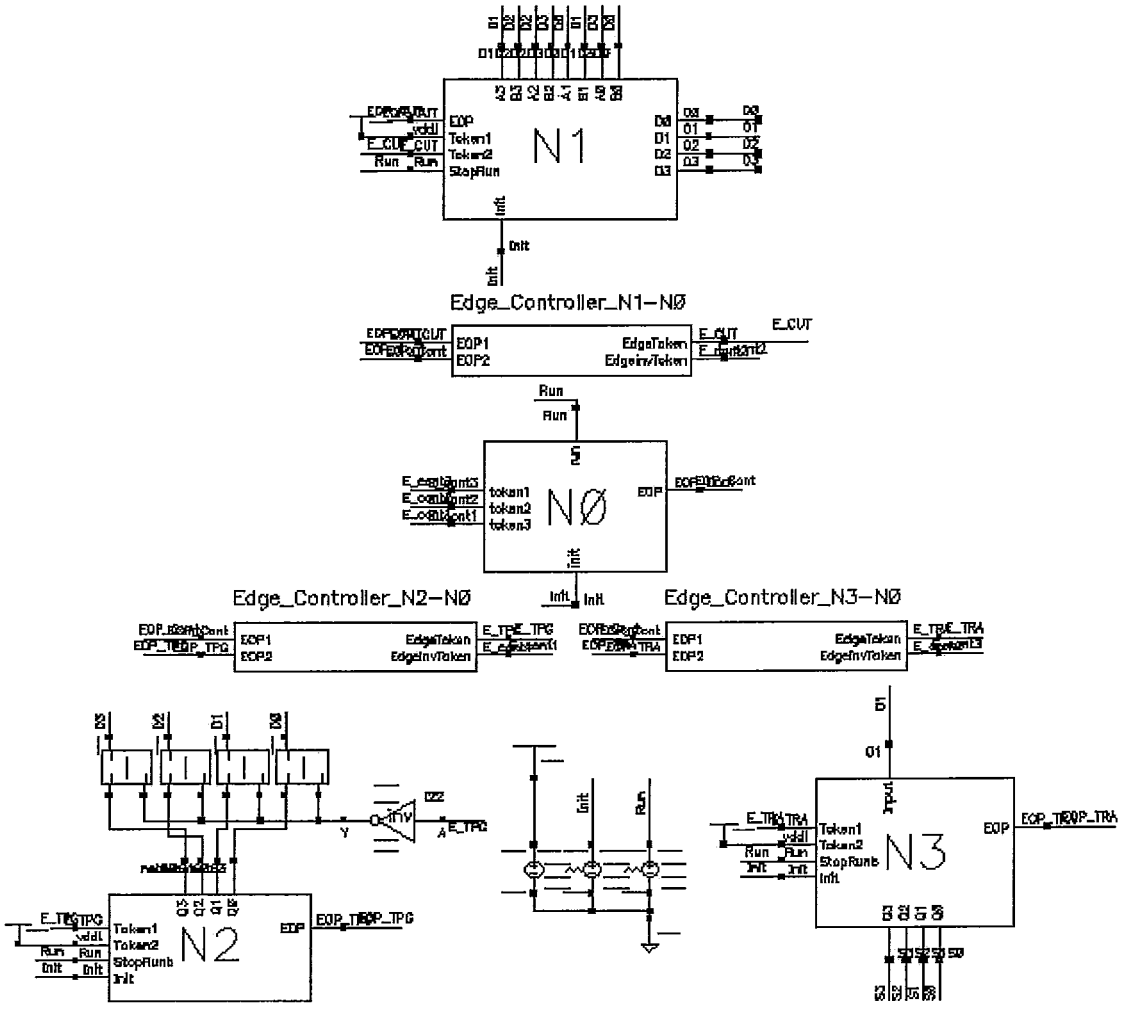


Figura 8.3: Diagrama esquemático do circuito BIST com metodologia S2A

Os problemas de inicialização, fim de operação do *BIST* e a implementação dos multiplexadores para operação nos modos normal e teste foram analisados anteriormente, na implementação do *BIST* assíncrono. Na implementação do *BIST S2A* eles não foram considerados. Assim foi possível desenvolver um sistema cuja simulação elétrica pôde ser feita nos equipamentos do *LPC/COPPE/UFRJ*. Simulações elétricas exigem grande quantidade de memória principal e também de memória secundária. O tempo de computação também é elevado, e face à pequena quantidade de equipamentos, ele é forçosamente limitado.

Observando-se a Figura 8.4 podemos observar a habilitação da operação dos nós através da reversão de arestas nos nós *N1* a *N3*. A duração de cada ciclo fica limitada pela duração das operações de todos os nós simultaneamente. Caso o sistema houvesse sido desenvolvido para operação em modo síncrono, todos os passos do sistema BIST ficariam limitados pelo pior caso do somador.

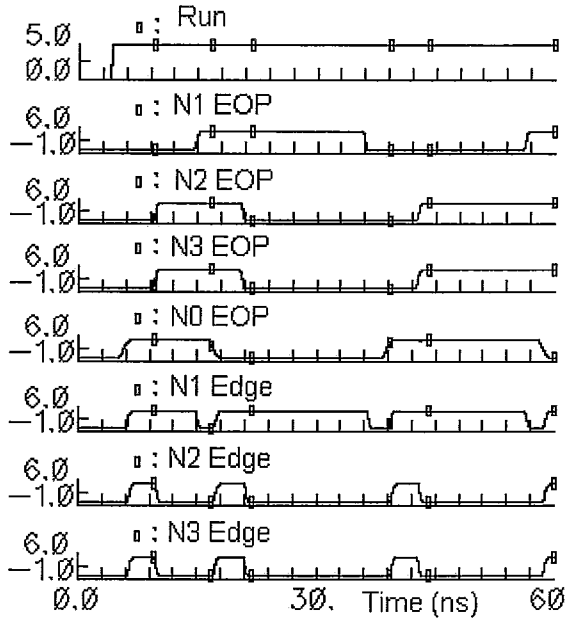


Figura 8.4: Simulação do sistema BIST S2A

8.3. Soft-Core para o Algoritmo Blowfish

O segundo exemplo deste capítulo trata da conversão de um *soft-core* para o algoritmo Blowfish.

O Blowfish foi proposto em 1994 [75] e revisado em 1995 [132]. Esse algoritmo é um candidato a substituir o *DES*, que já se encontra no final da vida útil. O *DES* tornou-se vulnerável a ataques por força bruta [133], e criptoanálise linear [134] e diferencial [135]. Embora existam estudos para o Blowfish mostrando a existência de chaves fracas, isto é, chaves onde $E_k(E_k(m)) = m$, os estudos foram feitos para variantes com menos de 8 rounds de cifragem, não sendo extensivos para implementações mais seguras, isto é, que tenham mais rounds de cifragem [132]. Dessa forma pode-se considerar que o Blowfish apresenta alta imunidade a ataques.

8.3.1. O Algoritmo Blowfish

O Blowfish é um algoritmo simétrico que foi colocado no domínio público. Ele cifra blocos de 64 bits com uma chave de tamanho variável. Para manter uma complexidade algorítmica relativamente baixa o Blowfish utiliza a Rede de Feistel, podendo ser implementado em máquinas simples como *smart cards* ou ainda sintetizado em *VLSI*. Todas as operações são feitas em módulo 32, e consistem apenas de soma, ou-exclusivo e troca mútua *(swap)* de bytes, além de pesquisa em tabela.

Inicialmente o algoritmo expande a chave de codificação, que pode ter até 448 bits, em uma matriz de subchaves que pode chegar a 4168 bytes. Esta fase de inicialização tem um custo computacional relativamente alto, tornando o Blowfish mais adequado para aplicações onde a chave não se modifica com freqüência.

A seguir faz-se a cifragem dos dados por Rede de Feistel, que no nosso caso tem 16 rounds.

Nossa implementação parte do princípio que as subchaves necessárias serão pré-computadas e armazenadas em registradores. Dessa forma, o hardware fará apenas a cifragem dos dados. Deve-se então inicializar um vetor de subchaves P, de 18 x 32 bits, e um conjunto de 4 tabelas, as S-boxes, de 256 x 32 bits cada. O método exato de geração dos valores iniciais de P e S está descrito em [75].

O algoritmo de cifragem para um bloco X de 64 bits, onde $X = X[63..0]$ pode ser descrito resumidamente como:

$X_L = X[63..32]$; $X_R = X[31..0]$
For $i = 1$ to 16
    $X_L = X_L \oplus P_i$
    $X_R = X_R \oplus F(X_L)$
    $X = (X_R, X_L)$
    end For
$X_R = X_R \oplus P_{17}$
$X_L = X_L \oplus P_{18}$
$X = (X_R, X_L)$

O procedimento geral de cifragem pode também ser descrito pelo diagrama da Figura 8.5, que mostra uma divisão em blocos para uma possível implementação em *hardware*.

Falta então definir o procedimento de cálculo da função F, o que é feito a seguir.

$a = X_L[31..24]$
$b = X_L[23..16]$
$c = X_L[15..8]$
$d = X_L[7..0]$
$F(X_L) = \left|\left(\left|S_a^1 + S_b^2\right|_{2^{32}} \oplus S_c^3\right) + S_d^4\right|_{2^{32}}$

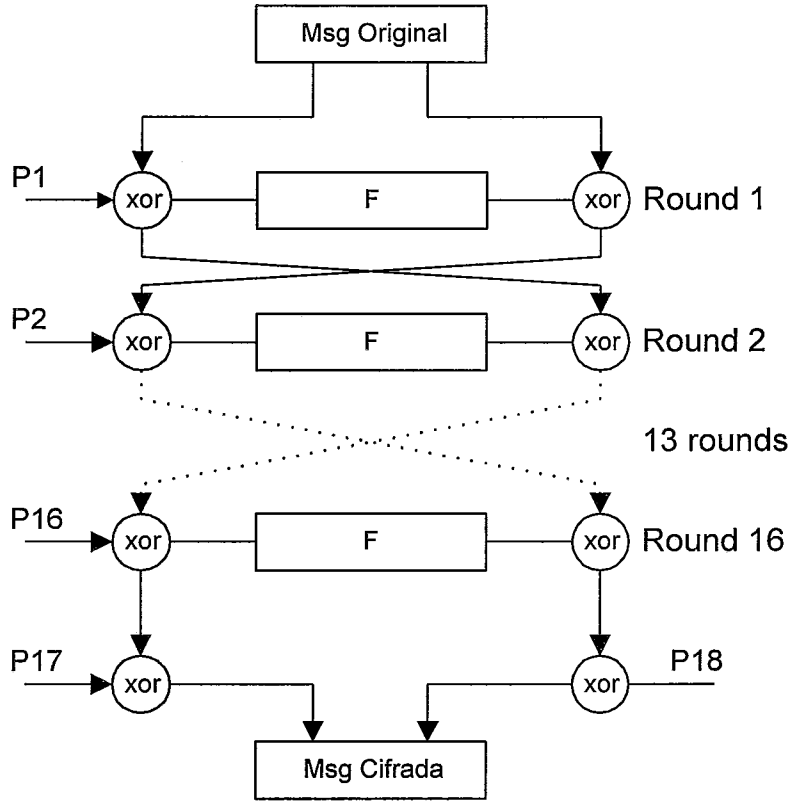


Figura 8.5: Estrutura do algoritmo Blowfish

O diagrama de blocos que mostra uma possível implementação para a função *F* pode ser visto na Figura 8.6.

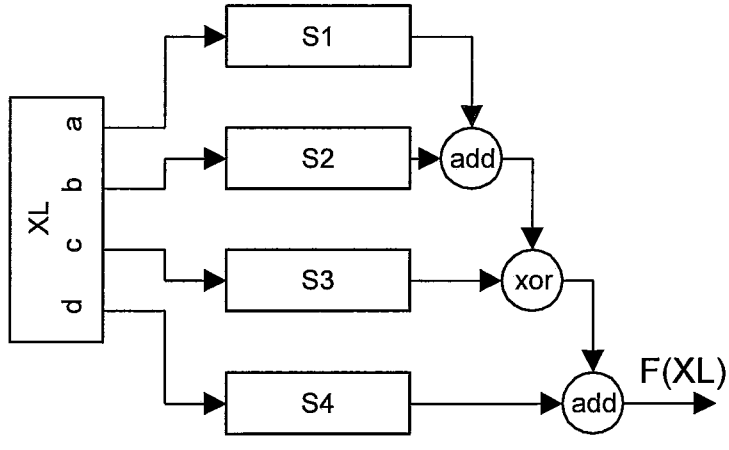


Figura 8.6: Cálculo da função $F(X_L)$

Como a cifragem é feita por Rede de Feistel, podemos facilmente decifrar o texto utilizando o mesmo hardware. Para isso carregamos as subchaves *P* na ordem inversa e reprocessamos o texto criptografado.

8.3.2. Implementação Síncrona

O diagrama de blocos feito para a implementação do algoritmo Blowfish demonstra claramente as oportunidades para explorar o paralelismo inerente ao algoritmo. Tratando-se de diagrama onde existem várias iterações, vemos de imediato duas possibilidades de otimização.

Pode-se explorar o paralelismo através da implementação de um estágio de *pipe* para cada iteração (*loop unroll*). Esse tipo de paralelismo tem custo de implementação elevado e é específico para uma determinada quantidade de *rounds*.

Pode-se também explorar a independência dos dados existentes em um único *round* de cifragem. Esse tipo de implementação facilita a geração de cifras utilizando-se de uma quantidade de *rounds* que pode ser facilmente modificada. A otimização de um passo do laço *(loop)* implica em um ganho de desempenho proporcional à quantidade de passos executados.

A otimização de um passo do laço *(loop)* foi escolhida para implementação. O diagrama de blocos dessa implementação pode ser visto na Figura 8.7.

Vimos que o algoritmo Blowfish tem uma fase inicial destinada ao cálculo das subchaves. Essa fase é seguida pela cifragem propriamente dita. A fase inicial é razoavelmente complexa, mas não ocorre com freqüência. Ela pode ser executada por software, pois não participando do laço, ela terá pouca influência no desempenho geral do sistema de criptografia.

O sistema da Figura 8.7 teve sua arquitetura descrita em *VHDL*. O *soft-core* ***SCOB*** assim obtido permitiu a implementação em hardware do algoritmo Blowfish.

A validação do sistema foi feita em diversas etapas. Inicialmente foi feita a comparação entre as cifras geradas por programa e pela descrição estrutural, sendo também incluídos os testes para validação da função inversa (*decryption*).

A seguir, o código *VHDL* foi sintetizado como *ASIC*, utilizando-se a biblioteca ES2. Essa é uma biblioteca CMOS para tecnologia de 0.7μm, com dois níveis de metalização. A implementação também foi feita em FPGA da família Altera Flex10K.

Detalhes dessas implementações podem ser vistos em [130].

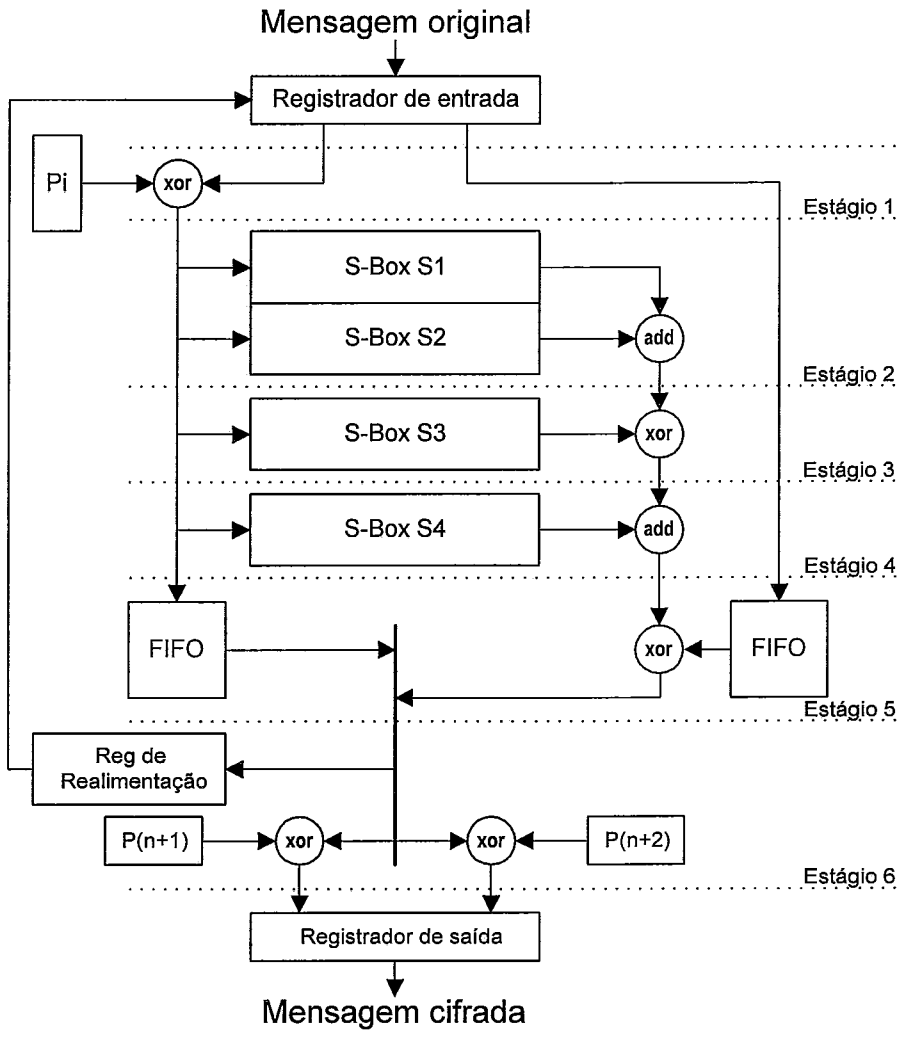


Figura 8.7: Pipeline desenvolvida para o algoritmo Blowfish

8.3.3. Implementação Assíncrona

A implementação do ***SCOB*** assíncrono foi feita utilizando-se a metodologia *S2A*. A idéia neste caso era a de validar a metodologia proposta observando-se tanto os problemas decorrentes da conversão quanto a performance obtida.

A conversão partiu de um projeto já existente e para o qual nenhum procedimento especial para operação em modo assíncrono havia sido previsto. A arquitetura original do ***SCOB*** foi mantida, e apenas os somadores foram substituídos por somadores do tipo *RCA*, desenvolvidos para a *ALU* auto-temporizada.

A equipe que fez a conversão foi a mesma que desenvolveu o ***SCOB***. Trata-se de equipe experiente em criptografia e em síntese de circuitos digitais, inclusive a

partir dos diversos tipos de descrições em *VHDL*. No entanto, esse foi o primeiro trabalho desenvolvido com lógica assíncrona.

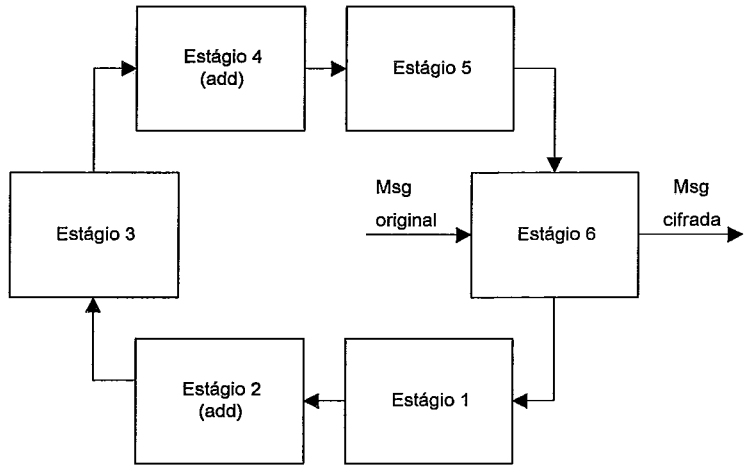


Figura 8.8: Grafo C para o soft-core do Blowfish

Utilizando-se a metodologia *S2A*, fizemos inicialmente a representação do circuito da Figura 8.7 através dos grafos *B* e *C*. Devido à semelhança entre os dois, apresentamos aqui, na Figura 8.8, apenas o grafo *C*.

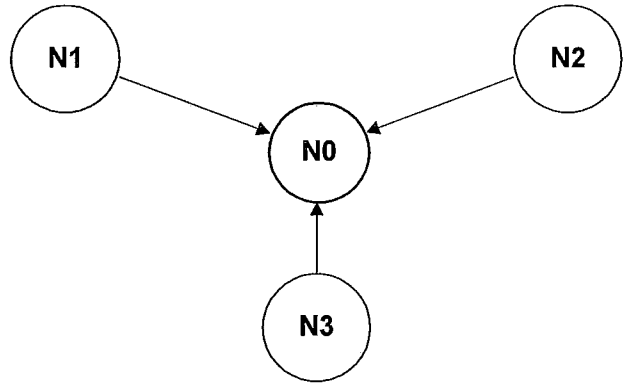


Figura 8.9: O grafo G reduzido para o soft-core Blowfish

Conforme descrito na metodologia, foi gerado um grafo *G*, a partir do grafo *C*. Também conforme a metodologia, o grafo *G* inicial, contendo 6 nós operacionais, foi reduzido para 3 nós de operação mais um nó *N0* para sincronismo, o que resultou no grafo da Figura 8.9. Os nós *N1* e *N2* correspondem à parte da *pipeline* que contém os somadores. A redução dos demais 4 nós operacionais em um único nó temporizado foi feita com a geração do nó *N3*. Este nó engloba todos os estágios do circuito que

são independentes dos somadores, ou seja, a parte do circuito cuja complexidade implica em um tempo de operação aproximadamente constante.

O tempo de operação do nó *N3* foi calculado a partir de uma simulação de pior caso para os circuitos nele incluídos. Esse tipo de simulação é bastante comum e está disponível em várias ferramentas de *CAD* como o Synopsys.

O sinal de relógio para o novo circuito passou a ser gerado pelo nó *N0*, que é habilitado quando todos os demais nós terminam de executar suas tarefas. Assim, o tempo de operação para um ciclo do *SCOB* assíncrono varia de um mínimo determinado pelo nó *N3* até o maior dos tempos necessários para operação dos somadores em um determinado ciclo. Para completar o ciclo deve-se ainda adicionar o tempo necessário para a reversão das arestas do nó *N0*.

8.3.4. Controladores de Nó e de Aresta

Os controladores de nó e de aresta são aqueles utilizados para síntese dos sistemas BIST. Um exemplo desses controladores pode ser visto na seção 7.3.1.1.

8.3.5. Somadores Auto-temporizados

Utilizamos o mesmo somador desenvolvido para o projeto da *ALU* auto-temporizada, com largura de 32 bits. A descrição em *VHDL* desse somador encontra-se no Anexo 2.

8.3.6. Resultados Obtidos

O tempo total necessário para conversão do *SCOB* foi de aproximadamente uma semana. A conversão do código e a depuração dos erros de codificação, envolvendo cerca de 5000 linhas de código *VHDL*, levou aproximadamente dois dias. O restante do tempo foi empregado na depuração de problemas de ordem dinâmica e testes de performance.

O principal problema de síntese encontrado pelos projetistas está descrito a seguir. Observando-se a estrutura da *pipeline*, pode-se notar que nos estágios que contêm somadores, pelo menos um dos operandos é dependente de consulta a uma tabela. Essa consulta é feita no início do ciclo, e o tempo necessário para se obter um valor estável nas saídas precisa ser compensado antes de se dar início à operação de soma. Ao contrário do que acontece em sistemas síncronos, um somador auto-temporizado só pode ser habilitado quando os operandos estiverem estáveis.

Identificado o problema, a solução foi incluir a compensação da pesquisa em tabela, o que foi feito por *matched delay* a partir da simulação do sistema. A equipe ainda levou mais um dia fazendo testes de performance.

Queremos ressaltar que nenhuma mudança foi feita na arquitetura inicial, e os resultados obtidos representam a aplicação da metodologia S2A a um circuito previamente existente. Para efeito de teste de performance foram aplicados ao sistema mensagens em modo texto e em modo binário (arquivos comprimidos). Uma vez que o sistema de criptografia tem por função embaralhar os dados apresentados em sua entrada, quaisquer dados de entrada, após poucos rounds de cifragem, pode ser considerado aleatório. A performance do sistema obtido está descrita na Tabela 8.1.

Tabela 8.1: Desempenho do SCOB assíncrono

	SCOB síncrono	**SCOB assíncrono**
Transistores	43280	48771
Ciclo mínimo	-	13 ns
Ciclo médio	20 ns	16 ns
Ciclo máximo	-	25 ns
Throughput médio	200 Mbps	250 Mbps

8.4. Resumo dos Sistemas com S2A

As implementações que utilizam metodologia *S2A* foram desenvolvidas visando o aproveitamento do ferramental existente no LPC para síntese de circuitos síncronos. A conversão para modo assíncrono não apresentou maiores dificuldades.

Ambos os circuitos foram simulados e apresentaram o desempenho esperado. A performance do *BIST S2A*, embora inferior à do sistema *ASSERT* original, mostrou-se bastante atraente para a implementação de circuitos complexos.

O *soft-core SCOB* foi convertido para operação em modo assíncrono através da metodologia *S2A*, tendo apresentado um desempenho médio 25% melhor que a versão síncrona.

A metodologia de conversão mostrou ser simples e confiável. Os resultados obtidos justificam plenamente a sua utilização.

Capítulo 9
Considerações Finais

9.1. Elaboração da Tese

A elaboração desta tese foi feita com base em estudos teóricos e aplicações práticas desse estudo. A seguir apresentamos uma lista dos resultados relevantes obtidos tanto em decorrência dos estudos teóricos quanto pela análise das aplicações.

Uma série de publicações foi apresentada em congressos internacionais, todas elas voltadas para a divulgação da metodologia de reversão de arestas, e freqüentemente, associadas a uma aplicação de interesse específico do congresso. Essas publicações estão listadas no Anexo 1.

9.1.1. Resultados e Contribuições

Os principais resultados obtidos e as contribuições desta tese são:

- Estudo teórico da técnica de reversão de arestas. O mecanismo desenvolvido para aplicação em grafos sugeriu a sua extensão para aplicações em hardware.

- Metodologia para síntese de sistemas assíncronos por reversão de arestas. Uma aplicação teórica dessa metodologia permitiu a criação de metodologia específica para a conversão de classes de sistemas síncronos em assíncronos.

- Desenvolvimento de hardware capaz de implementar a temporização assíncrona através do uso de controladores de nó e de aresta. O uso desses controladores apresenta baixo custo tanto em área quanto em performance.

- Avaliação da dificuldade da implementação de projetos *DFT*, isto é, de projetos visando a testabilidade, em sistemas assíncronos, além da proposição de um sistema *BIST* assíncrono.

- Simulações dos sistemas resultantes. Dependendo do caso foram feitas simulações lógicas, elétricas e da síntese de circuitos sintetizados a partir de uma descrição *VHDL*.

- Estudo de somadores e de sua aplicabilidade a sistemas assíncronos.

- Desenvolvimento de somador assíncrono que utiliza apenas portas lógicas convencionais.

- Desenvolvimento de *ALU* auto-temporizada, compatível com *ALU Sun SPARC*. Avaliação do desempenho quando comparada a um modelo síncrono.

- Avaliação do desempenho da *ALU* auto-temporizada quando submetida a uma massa de dados gerada por programas da série *SPEC*. Análise da queda de desempenho provocada por características dos programas, do compilador utilizado e também em decorrência das limitações da *ALU* original.

- Emprego de técnicas assíncronas, com metodologia *S2A*, para incrementar a performance de um *soft-core* para criptografia de alto desempenho.

9.2. Desenvolvimento Futuro

Em todo trabalho é necessário delimitar o objetivo a ser atingido dentro do prazo proposto. Assim, vários aspectos para desenvolvimento foram vistos no decorrer deste trabalho. Esses aspectos são apresentados a seguir, e constituem motivação para um desenvolvimento futuro.

9.2.1. ALU Auto-temporizada

Vimos que o desempenho do circuito fica em parte prejudicado pela falta de simetria dos sinais *start* e *carry*. Isto nos obriga a incluir circuitos inversores que alteram o tempo de propagação em um dos ramos do circuito. Caso seja possível obter valores simétricos para esses sinais, o uso de inversores seria desnecessário e o desempenho do circuito seria independente do ramo onde se dá a detecção. O custo dessa implementação seria então o da inclusão de mais dois sinais na interface da ***ALU***.

Os problemas de meta-estabilidade mencionados no item 6.7.1. podem ser resolvidos através da inclusão de um sincronizador baseado em flip-flop tipo C, de Müller. Neste caso, o retorno do sinal de *busy* para *not busy* somente poderia ocorrer ao final do sinal de *start*. Isto também serviria para diminuir o tempo total de ciclo da *ALU*.

Neste trabalho consideramos que todos os sinais de entrada somente estão estabilizados em um tempo imediatamente anterior àquele onde ocorre o comando de início de operação. Caso o projeto do subsistema seja feito de forma a garantir a pré-carga dos comandos da *ALU*, a propagação dos sinais de seleção da operação e eventualmente do *carry,* podem ocorrer antes do comando de início, diminuindo a latência do circuito.

As especificações de *hardware* utilizadas neste trabalho visam facilitar o entendimento dos circuitos. Foi utilizada apenas lógica positiva e todas as portas funcionam com níveis lógicos restaurados. É possível obter um melhor desempenho utilizando-se lógica negativa tanto em barramentos como em circuitos internos. Isto permitiria economizar inversores, melhorando o desempenho. Da mesma forma, o uso de lógica não restaurada pode trazer benefícios ao desempenho da *ALU*.

A representação binária dos operandos negativos em palavras de dados merece algumas considerações. Pequenos valores negativos são na realidade valores com extensão de sinal para completar a largura da palavra. Assim, seria interessante detectar as operações que envolvem esses valores para gerar um sinal de propagação de *carry* antecipado a cada byte estendido. Ou ainda, para encerrar rapidamente a soma utilizando-se apenas os bytes estritamente necessários, calculando-se o valor dos demais por extensão de sinal. O custo da lógica adicional é baixo e o desempenho obtido ficaria significativamente melhor. O tempo de execução de uma operação de decremento unitário, no caso de operações com valores que possam ser representados em um byte, seria reduzido ao tempo necessário para a propagação de 8 bits.

Deixamos estas sugestões para um desenvolvimento futuro, uma vez que o objetivo a que nos propusemos era simplesmente estudar o desempenho e conseqüentemente a viabilidade do desenvolvimento de uma *ALU* auto-temporizada baseada em somadores simples, objetivo esse que acreditamos havermos atingido.

9.2.2. Sistema ASSERT com Topologia Variável

Nesta tese estudamos sistemas assíncronos cuja temporização é feita pela metodologia *ASSERT*. Vimos que quaisquer sistemas auto-oscilantes podem se beneficiar dessa metodologia de síntese.

Uma outra classe de problemas que merece atenção é a relativa ao mapeamento de algoritmos específicos diretamente em *hardware*. Um algoritmo é composto de uma fase de inicialização, que é seguida por um processamento que se repete até que uma condição seja alcançada. O corpo do algoritmo pode conter hierarquias que executam outros algoritmos.

O teste de término de execução pode ter seu conceito estendido de modo a propiciar testes lógicos e/ou aritméticos que selecionem um caminho de execução dentre os n caminhos possíveis. Pode-se ainda pensar na execução concorrente de alguns ou de todos os caminhos disponíveis. Para que o processamento possa prosseguir é necessário um elemento extra, que faça a junção desses caminhos após o processamento paralelo ou alternativo.

Uma solução que busque a possibilidade de se determinar caminhos alternativos de execução a partir de circuitos sintetizados com a metodologia *ASSERT* irá necessitar de grafos com topologia dinamicamente variável.

Esse aperfeiçoamento dependeria da determinação teórica das condições de contorno que seriam necessárias para manter a validade dos sistemas *ASSERT* em sistemas com topologia variável. Dependeria também de controladores especialmente desenvolvidos para implementar os pares de funções *fork/merge*. Temos aqui mais um campo cujo desenvolvimento futuro poderá render soluções interessantes.

9.2.3. Considerações Sobre Desempenho

Vimos anteriormente que sistemas assíncronos em geral podem apresentar um desempenho superior ao dos sistemas síncronos convencionais, pois a medida do desempenho passa a ser uma função estatística dos desempenhos médios esperados para cada célula assíncrona.

Deve-se observar, contudo, que os preceitos de projeto estabelecidos para máquinas síncronas não se tornaram obsoletos, apenas tiveram seu enunciado ligeiramente modificado para que pudessem ser aplicados aos sistemas assíncronos.

Da mesma forma que em um sistema síncrono, a execução serial de um caminho assíncrono fica limitada em seu desempenho pela célula mais lenta. Dessa forma, a performance de uma *pipeline* assíncrona fica limitada pelo desempenho médio esperado para a sua célula mais lenta. Como os tempos de execução variam dinamicamente, existe a possibilidade de que o tempo de execução mais lento possa ocorrer em células diversas ao longo do tempo.

Sabemos, a partir do projeto de sistemas síncronos, que o projeto de uma *pipeline* procura fazer com que todos os estágios tenham aproximadamente o mesmo tempo de execução. Eventualmente pode haver a transferência de funções de um estágio para outro com o objetivo de se otimizar o desempenho final do *pipe*. Em sistemas assíncronos o mesmo conceito deve ser empregado, tendo-se o cuidado de substituir o tempo de execução determinístico dos sistemas síncronos por um tempo médio esperado para a execução de um certo conjunto de operandos no sistema assíncrono.

Também sabemos que o acoplamento entre sistemas de produção e consumo que operam com velocidades diferentes deve ser feito através de uma fila cujo atendimento seja suficiente para fazer com que o tamanho máximo previsto dificilmente seja atingido. Deve existir também uma política de tratamento para os casos em que isso ocorre. Em termos eletrônicos, isso significa fazer o acoplamento entre estágios através de módulos *FIFO*, com largura e profundidade adequadas aos operandos. Nos casos em que o tamanho da fila pode vir a exceder a capacidade do *FIFO* basta fazer com que os módulos anteriores aguardem a liberação da fila. E se isso ocorrer com freqüência tal que prejudique o desempenho final do circuito, a criação de um novo balcão, isto é, a distribuição de tarefas por processadores concorrentes, deve ser implementada.

Essas considerações foram feitas para mostrar que em sistemas síncronos o acoplamento entre estágios depende normalmente de um banco de registradores com profundidade 1. Já em circuitos assíncronos, cada acoplamento deve ser estudado para que se possa determinar a profundidade dos registradores *FIFO* que serão necessários.

É por isso que queremos ressaltar a importância do desenvolvimento de uma ferramenta que faça automaticamente a análise dos tempos médios de execução e calcule o tamanho dos *FIFO*s, além de verificar a necessidade de processamento

paralelo para aumentar o desempenho. Esta é mais uma sugestão para desenvolvimento futuro.

9.3. Conclusão

Fizemos nesta tese o desenvolvimento de uma metodologia voltada para a temporização de sistemas assíncronos. Essa metodologia tem um embasamento teórico bastante forte e mostrou ser capaz de representar uma solução simples e elegante para a síntese de várias classes de sistemas assíncronos.

A interconexão dos sistemas assíncronos é feita através de circuitos do tipo *DI* - insensíveis a atrasos, como mandam as mais recentes descobertas no campo da integração em ULSI. Dentro de cada nó de processamento, circuitos auto-temporizados podem ser empregados, ou a característica de auto-temporização pode ser simulada, de modo a evitar um comprometimento do desempenho de circuitos que apenas abrangem uma mesma região equipotencial de tempo.

O custo dessa implementação é bastante baixo. O tempo necessário para atuação dos circuitos de controle é pequeno, permitindo até mesmo uma granulação bastante fina dos circuitos controlados sem comprometer o desempenho.

Propostas para implementação de controladores que utilizam essa técnica foram apresentadas no decorrer da tese e as diversas simulações efetuadas demonstraram a correção das implementações a partir da metodologia proposta.O desenvolvimento dessa tecnologia propiciou a geração de patente no INPI/Brasil, referente à metodologia de síntese.

A principal ferramenta desenvolvida para aplicações assíncronas constitui-se na descrição em *VHDL* de um somador auto-temporizado simples e eficiente, inteiramente desenvolvido com portas lógicas convencionais. As aplicações desse somador nesta tese, vão desde o desenvolvimento de uma *ALU* auto-temporizada até a utilização em um núcleo de criptografia para circuitos integrados. Esse somador também vem sendo utilizado em cursos da UFRJ como parte de uma introdução à lógica assíncrona, já tendo sido sintetizado em diversas tecnologias.

Uma série de trabalhos foi apresentada em congressos, mostrando a aplicabilidade da metodologia a diversas situações, notadamente quanto à testabilidade de circuitos assíncronos e o desenvolvimento de subsistemas complexos como o *soft-core* de criptografia.

Anexo 1
Trabalhos Gerados

Os trabalhos desenvolvidos para os exames de qualificação formam a base desta tese. Esses trabalhos foram:

Síntese de Alto Nível
Examinador: Antônio Carneiro de Mesquita Filho
Programa de Engenharia Elétrica, COPPE/UFRJ, 1995.

Asynchronous Digital Circuits as Neighborhood-Constrained Systems
Examinador: Felipe França
Publicação ES-357/95 do Programa de Engenharia de Sistemas, COPPE/UFRJ , 1995.

Projeto de uma Unidade Lógica e Aritmética Assíncrona
Examinador: Eliseu Chaves Filho
Publicação ES-414/96 do Programa de Engenharia de Sistemas, COPPE/UFRJ, 1996.

O modelo utilizado para síntese de sistemas assíncronos resultou em patente, que foi registrada como:

Patente BR PI 9703819-9 – Processo de Síntese e Aparelho para Temporização Assíncrona de Circuitos e Sistemas Digitais Multifásicos
Autores: Felipe França, Vladimir Castro Alves e Edson do Prado Granja.
INPI, Brasil, 1997.

Os trabalhos apresentados em congressos sofrem restrições para a divulgação, pois o *copyright* pertence ao IEEE e à Klüwer. No entanto, informações a respeito

desses trabalhos podem ser obtidas diretamente com os autores nos seguintes endereços eletrônicos:

- Edson P. Granja egranja@lpc.ufrj.br
- Felipe M. G. França felipe@cos.ufrj.br
- Vladimir Castro Alves castro@lpc.ufrj.br

Esta é a relação dos trabalhos apresentados em congressos.

FRANÇA, F; ALVES, V.; GRANJA, E – A Multi-phase Asynchronous Timing Scheme – Proceedings of the X Brazilian Symposium on Integrated Circuit Design, SBCCI – Gramado, RS , 1997.

FRANÇA F., ALVES V., GRANJA E. – Edge Reversal-based Asynchronous Timing Synthesis. – Proceedings of the IEEE International Symposium on Circuits and Systems – Monterey, CA, USA, June 1998.

FELIPE M. G. FRANÇA VLADIMIR C. ALVES AND EDSON P. GRANJA. – A BIST Scheme for Asynchronous Logic – Proceedings of the Asian Test Symposium, Singapore, 1998.

ALCÂNTARA J.; SALOMÃO S.; GRANJA, E.; ALVES V.; FRANÇA F. – Synchronous to Asynchronous Conversion – A Case Study: The Blowfish Algorithm Implementation – VLSI '99 – 10th International Conference on VLSI , 1999 , Lisbon, Portugal. Also published in VLSI: Systems on a Chip – Klüwer Academic Publishers, 1999 , p. 173 –180.

Cabe finalmente mencionar que alguns dos trabalhos apresentados estão incluídos na página de referências bibliográficas sobre circuitos assíncronos elaborada por Ad Peeters, da Philips Research.

Essa página pode ser consultada utilizando-se o endereço http://liinwww.ira.uka.de/bibliography/Misc/async.circuits.html.

Anexo 2 Descrição em VHDL da ALU Auto-temporizada

```
-- aalu.vhd
-- AALU Asynchronous ALU
-- 32 bit operands
-- Basic operatios: sum, and, or, xor
-- Variants: 2nd operand negation: sub, andn, orn, xnor
--           Ops using Carry in
--           Ops affecting condition FF

-------------------------------------------------------------------
-- Single-bit ALU
-------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity alu is
    port (start: in std_logic;
          op   : in std_logic_vector (5 downto 0);
          nop5 : in std_logic;
          a    : in std_logic;
          b    : in std_logic;
          c0in : in std_logic;
          c1in : in std_logic;
          c0out: out std_logic;
          c1out: out std_logic;
          res  : out std_logic;
          ready : out std_logic);
end alu;

-- description of adder using concurrent signal assignments
architecture rtl of alu is

signal bin: std_logic;
signal sum: std_logic;
signal gen0, gen1: std_logic;
signal c0, c1: std_logic;
signal prop: std_logic;
signal andab, orab, xorab: std_logic;
signal opsum, opand, opor, opxor: std_logic;

begin
    bin <= b xor op(2);
    sum <= (a xor bin) xor c0in;
    gen0 <= (a and bin and start);
    gen1 <= not (a or bin or not start);
    prop  <= (a xor bin);
    c0 <= gen0 or (c0in and prop);
    c1 <= gen1 or (c1in and prop);
    ready  <= c0 or c1;
    c0out <= c0;
    c1out <= c1;
    andab <= a and bin;
    orab  <= a or bin;
    xorab <= a xor bin;
    opsum <= (sum and not op(0) and not op(1));
    opand <= (andab and op(0) and not op(1));
    opor  <= (orab and not op(0) and op(1));
    opxor <= (xorab and op(0) and op(1));
    res   <= (opsum or opand or opor or opxor) and nop5;
end rtl;

-- description of alu using component instantiation statements
use work.gates.all;
architecture structural of alu is
    signal nstart, bin: std_logic;
    signal prop: std_logic;
    signal g0, g1: std_logic;
    signal c0, c1: std_logic;
    signal c0p, c1p: std_logic;
    signal sum, aandb, aorb, axorb: std_logic;
begin
-- full adder
    xor1: xorg port map(
        in1  => a,
        in2  => bin,
        out1 => axorb);
    xor2: xorg port map(
                in1 => axorb,
                in2 => c0in,
                out1 => sum);
-- Negate
    xor3: xorg port map(
        in1 => b,
        in2 => op(2),
        out1 => bin);

-- Carry propagation
-- Propagate Control
    xor4: xorg port map(
```

```
        in1 => a,
        in2 => bin,
        out1 => prop);
-- Propagate0
    and1: andg port map(
        in1 => c0in,
        in2 => prop,
        out1   => c0p);
-- Generate0
    and2: and3g port map(
        in1 => start,
        in2 => a,
        in3 => bin,
        out1   => g0);
-- Carry 0 = Generate0 + Propagate0
    or1: org port map(
        in1 => g0,
        in2 => c0p,
        out1   => c0);
-- Propagate1
    and3: andg port map(
        in1 => c1in,
        in2 => prop,
        out1   => c1p);
-- Generate1
    inv1: invg port map(
        in1 => start,
        out1   => nstart);
    nor1: nor3g port map(
        in1 => a,
        in2 => bin,
        in3 => nstart,
        out1   => g1);
-- Carry 1 = Generate1 + Propagate1
    or2: org port map(
        in1 => g1,
        in2 => c1p,
        out1   => c1);
-- End of operation: This bit
    or3: org port map(
        in1 => c0,
        in2 => c1,
        out1   => ready);
        c0out <= c0;
        c1out <= c1;

-- Logic operations
-- AND
   and4: andg port map(
        in1 => a,
        in2 => bin,
        out1=> aandb);
-- OR
    or4: org port map(
        in1 => a,
        in2 => bin,
        out1 => aorb);
-- XOR
-- axorb from full adder

-- Results mux
    mux1: mux4e3s port map(
        in0 => sum,
        in1 => aandb,
        in2 => aorb,
        in3 => axorb,
        sel0 => op(0),
        sel1 => op(1),
        enable => nop5,
        out1   => res);

end structural;

--------------------------------------------------------------------------
-- N-bit alu
-- The width of the alu is determined by generic N
--------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
entity aluN is
    generic(N : integer := 32);
    port (start: in std_logic;
          endalu : out std_logic;
          op    : in std_logic_vector (5 downto 0);
          a     : in std_logic_vector ((N-1) downto 0);
          b     : in std_logic_vector ((N-1) downto 0);
          cin   : in std_logic;
          r     : out std_logic_vector ((N-1) downto 0);
          cicc, vicc, nicc, zicc : out std_logic;
          iccstb : out std_logic);
end aluN;

-- structural implementation of the N-bit alu
use work.gates.all;
architecture structural of aluN is
    component alu
        port (start: in std_logic;
              op    : in std_logic_vector (5 downto 0);
              nop5 : in std_logic;
              a     : in std_logic;
              b     : in std_logic;
              c0in : in std_logic;
              c1in : in std_logic;
              c0out: out std_logic;
```

```
            c1out: out std_logic;
            res  : out std_logic;
            ready : out std_logic);
    end component;

    signal cp0, cp1  : std_logic_vector (N downto 0);
    signal res, ready: std_logic_vector ((N-1) downto 0);
    signal nop0, nop1, nop4, nop5, ncin: std_logic;
    signal logic0, logic1 : std_logic;
    signal vsum, endop, nendop: std_logic;
    signal logicop, endlogic: std_logic;
    signal endarith, halfend0, halfend1: std_logic;
    signal z0, halfz0, halfz1: std_logic;

begin
-- Initialization
    r <= res;
    endalu <= endop;

-- We need those signals
    inv2: invg port map(
        in1 => op(0),
        out1   => nop0);
    inv3: invg port map(
        in1 => op(1),
        out1   => nop1);
    inv4: invg port map(
        in1 => op(4),
        out1   => nop4);
    inv5: invg port map(
        in1 => op(5),
        out1   => nop5);
    inv6: invg port map(
        in1 => cin,
        out1   => ncin);
    or5: org port map(
        in1 => op(0),
        in2 => op(1),
        out1 => logicop);

-- Overflow detection
    xor5: xorg port map(
        in1 => cp0(N),
        in2 => cp0(N-1),
        out1 => vsum);

-- Instantiate a single-bit alu N times
   gen: for I in 0 to (N-1) generate
    aalu: alu port map(
        op => op,
        nop5 => nop5,
        a => a(I),
        b => b(I),
        c0in => cp0(I),
        c1in => cp1(I),
        c0out=> cp0(I+1),
        c1out=> cp1(I+1),
        res  => res(I),
        ready => ready(I),
        start => start );
   end generate;

-- Carry-in control
        logic0 <= '0';
        logic1 <= '1';
    mux2: mux4e port map(
        in0 => logic0,
        in1 => logic1,
        in2 => cin,
        in3 => ncin,
        sel0 => op(2), -- Negate
        sel1 => op(3), -- Use Carry in
        enable => start,
        out1   => cp0(0)); -- Direct output

    mux3: mux4e port map(
        in0 => logic1,
        in1 => logic0,
        in2 => ncin,
        in3 => cin,
        sel0 => op(2), -- Negate
        sel1 => op(3), -- Use Carry in
        enable => start,
        out1   => cp1(0)); -- Inverted output

-- End of arithmetic operation
   and5: and16g port map(
        in0 => ready(0),
        in1 => ready(1),
        in2 => ready(2),
        in3 => ready(3),
        in4 => ready(4),
        in5 => ready(5),
        in6 => ready(6),
        in7 => ready(7),
        in8 => ready(8),
        in9 => ready(9),
        inA => ready(10),
        inB => ready(11),
        inC => ready(12),
        inD => ready(13),
        inE => ready(14),
        inF => ready(15),
        out1 => halfend0);
```

```
   and6: and16g port map(
        in0 => ready(16),
        in1 => ready(17),
        in2 => ready(18),
        in3 => ready(19),
        in4 => ready(20),
        in5 => ready(21),
        in6 => ready(22),
        in7 => ready(23),
        in8 => ready(24),
        in9 => ready(25),
        inA => ready(26),
        inB => ready(27),
        inC => ready(28),
        inD => ready(29),
        inE => ready(30),
        inF => ready(31),
        out1 => halfend1);

   and7: andg port map(
        in1 => halfend0,
        in2 => halfend1,
        out1 => endarith);

-- End of logical operation
   and8: andg port map(
        in1 => ready(0),
        in2 => logicop,
        out1 => endlogic);

-- End of operation
    or6: org port map(
        in1 => endarith,
        in2 => endlogic,
        out1 => endop);

    inv7: invg port map(
        in1 => endop,
        out1   => nendop);

-- Zero detection
   nor2: nor16g port map(
        in0 => res(0),
        in1 => res(1),
        in2 => res(2),
        in3 => res(3),
        in4 => res(4),
        in5 => res(5),
        in6 => res(6),
        in7 => res(7),
        in8 => res(8),
        in9 => res(9),
        inA => res(10),
        inB => res(11),
        inC => res(12),
        inD => res(13),
        inE => res(14),
        inF => res(15),
        out1 => halfz0);

   nor3: nor16g port map(
        in0 => res(16),
        in1 => res(17),
        in2 => res(18),
        in3 => res(19),
        in4 => res(20),
        in5 => res(21),
        in6 => res(22),
        in7 => res(23),
        in8 => res(24),
        in9 => res(25),
        inA => res(26),
        inB => res(27),
        inC => res(28),
        inD => res(29),
        inE => res(30),
        inF => res(31),
        out1 => halfz1);

    and9: andg port map(
        in1 => halfz0,
        in2 => halfz1,
        out1 => z0);

-- Signals sent to icc
-- Copy command
    and10: andg port map(
        in1 => endop,
        in2 => op(4), -- Affects icc
        out1 => iccstb);
-- Sign
    nicc <= res(N-1);
-- Carry
    and11: and3g port map(
        in1 => cp0(N),
        in2 => nop0,
        in3 => nop1,
        out1 => cicc);
-- Overflow
    and12: and3g port map(
        in1 => vsum,
        in2 => nop0,
        in3 => nop1,
```

```
      out1 => vicc);
-- Zero
    zicc <= z0;
-- end if;

end structural;
```

```
-- newgates.vhd
--------------------------------------------------------------------------
-- package with component declarations
--------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
package gates is
    component andg
        generic (tpd_hl : time := 1 ns;
                 tpd_lh : time := 1 ns);
        port (in1, in2 : std_ulogic;
              out1 : out std_ulogic);
    end component;

    component and3g
        generic (tpd_hl : time := 1 ns;
                 tpd_lh : time := 1 ns);
        port (in1, in2, in3 : std_ulogic;
              out1 : out std_ulogic);
    end component;

    component and4g
        generic (tpd_hl : time := 1 ns;
                 tpd_lh : time := 1 ns);
        port (in1, in2, in3, in4 : std_ulogic;
              out1 : out std_ulogic);
    end component;

    component and16g
        generic (tpd_hl : time := 2 ns;
                 tpd_lh : time := 2 ns);
        port (in0, in1, in2, in3, in4, in5, in6, in7,
              in8, in9, inA, inB, inC, inD, inE, inF : std_ulogic;
```

```
              out1 : out std_ulogic);
end component;

component invg
    generic (tpd_hl : time := 1 ns;
             tpd_lh : time := 1 ns);
    port (in1 : std_ulogic;
          out1 : out std_ulogic);
end component;

component mux4e
    generic (tpd_hl : time := 2 ns;
             tpd_lh : time := 2 ns);
    port (in0, in1, in2, in3 : std_logic;
          sel0, sel1 : std_logic;
          enable : std_logic;
          out1 : out std_logic);
end component;

component mux4e3s
    generic (tpd_hl : time := 2 ns;
             tpd_lh : time := 2 ns);
    port (in0, in1, in2, in3 : std_logic;
          sel0, sel1 : std_logic;
          enable : std_logic;
          out1 : out std_logic);
end component;

component norg
    generic (tpd_hl : time := 1 ns;
        tpd_lh : time := 1 ns);
    port (in1, in2 : std_ulogic;
        out1 : out std_ulogic);
end component;

component nor3g
    generic (tpd_hl : time := 1 ns;
             tpd_lh : time := 1 ns);
    port (in1, in2, in3 : std_logic;
          out1 : out std_logic);
end component;

component nor4g
    generic (tpd_hl : time := 1 ns;
             tpd_lh : time := 1 ns);
    port (in1, in2, in3, in4 : std_logic;
          out1 : out std_logic);
end component;

component nor16g
    generic (tpd_hl : time := 2 ns;
             tpd_lh : time := 2 ns);
```

```
        port (in0, in1, in2, in3, in4, in5, in6, in7,
              in8, in9, inA, inB, inC, inD, inE, inF : std_ulogic;
              out1 : out std_ulogic);
    end component;

    component org
        generic (tpd_hl : time := 1 ns;
                 tpd_lh : time := 1 ns);
        port (in1, in2 : std_logic;
              out1 : out std_logic);
    end component;

    component or3g
        generic (tpd_hl : time := 1 ns;
                 tpd_lh : time := 1 ns);
        port (in1, in2, in3 : std_logic;
              out1 : out std_logic);
    end component;

    component or4g
        generic (tpd_hl : time := 1 ns;
                 tpd_lh : time := 1 ns);
        port (in1, in2, in3, in4 : std_logic;
              out1 : out std_logic);
    end component;

    component xorg
        generic (tpd_hl : time := 1 ns;
                 tpd_lh : time := 1 ns);
        port (in1, in2 : std_logic;
              out1 : out std_logic);
    end component;
end gates;

--------------------------------------------------------------------
-- 2-input and gate
--------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
entity andg is
    generic (tpd_hl : time := 1 ns;
             tpd_lh : time := 1 ns);
    port (in1, in2 : std_ulogic;
          out1 : out std_ulogic);
end andg;

architecture only of andg is
begin
    p1: process(in1, in2)
        variable val : std_logic;
    begin
        val := in1 and in2;
        case val is
            when '0' =>
                out1 <= '0' after tpd_hl;
            when '1' =>
                out1 <= '1' after tpd_lh;
            when others =>
                out1 <= val;
        end case;
    end process;
end only;

--------------------------------------------------------------------
-- 3-input and gate
--------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
entity and3g is
    generic (tpd_hl : time := 1 ns;
             tpd_lh : time := 1 ns);
    port (in1, in2, in3 : std_ulogic;
          out1 : out std_ulogic);
end and3g;

architecture only of and3g is
begin
    p1: process(in1, in2, in3)
        variable val : std_logic;
    begin
    val := in1 and in2 and in3;
        case val is
            when '0' =>
                out1 <= '0' after tpd_hl;
            when '1' =>
                out1 <= '1' after tpd_lh;
            when others =>
                out1 <= val;
        end case;
    end process;
end only;

--------------------------------------------------------------------
-- 4-input and gate
--------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
entity and4g is
    generic (tpd_hl : time := 1 ns;
             tpd_lh : time := 1 ns);
    port (in1, in2, in3, in4 : std_ulogic;
          out1 : out std_ulogic);
end and4g;
```

```
architecture only of and4g is
begin
    p1: process(in1, in2, in3, in4)
        variable val : std_logic;
    begin
    val := in1 and in2 and in3 and in4;
        case val is
            when '0' =>
                out1 <= '0' after tpd_hl;
            when '1' =>
                out1 <= '1' after tpd_lh;
            when others =>
        out1 <= val;
        end case;
    end process;
end only;

-------------------------------------------------------------------
-- 16-input and gate
-- 16-inverted input or gate
-- uses 4 4-input NAND gates cascaded to a 4-input NOR gate
-------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
entity and16g is
    generic (tpd_hl : time := 2 ns;
             tpd_lh : time := 2 ns);
    port (in0, in1, in2, in3, in4, in5, in6, in7,
          in8, in9, inA, inB, inC, inD, inE, inF : std_ulogic;
          out1 : out std_ulogic);
end and16g;

architecture only of and16g is
begin
    p1: process(in0, in1, in2, in3, in4, in5, in6, in7,
                in8, in9, inA, inB, inC, inD, inE, inF)
        variable val, val1, val2, val3, val4 : std_logic;
    begin
    val1 := not (in0 and in1 and in2 and in3);
    val2 := not (in4 and in5 and in6 and in7);
    val3 := not (in8 and in9 and inA and inB);
    val4 := not (inC and inD and inE and inF);
    val  := not (val1 or val2 or val3 or val4);
    case val is
            when '0' =>
                out1 <= '0' after tpd_hl;
            when '1' =>
                out1 <= '1' after tpd_lh;
            when others =>
        out1 <= val;
        end case;
    end process;
end only;

-------------------------------------------------------------------
-- inv gate
-------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
entity invg is
    generic (tpd_hl : time := 1 ns;
         tpd_lh : time := 1 ns);
    port (in1 : std_ulogic;
      out1 : out std_ulogic);
end invg;

architecture only of invg is
begin
    p1: process(in1)
        variable val : std_logic;
begin
    val := not in1;
    case val is
        when '0' =>
        out1 <= '0' after tpd_hl;
        when '1' =>
        out1 <= '1' after tpd_lh;
        when others =>
        out1 <= val;
    end case;
    end process;
end only;

-------------------------------------------------------------------
-- 4-input mux with enable
-------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4e is
    generic (tpd_hl : time := 2 ns;
         tpd_lh : time := 2 ns);
    port (in0, in1, in2, in3 : std_logic;
          sel0, sel1 : std_logic;
          enable : std_logic;
      out1 : out std_logic);
end mux4e;
architecture only of mux4e is
begin
    p1: process(in0, in1, in2, in3, sel0, sel1, enable)
        variable line0, line1, line2, line3 : std_logic;
        variable val : std_logic;
    begin
line0 := in0 and not sel1 and not sel0;
```

```
    line1 := in1 and not sel1 and     sel0;
    line2 := in2 and     sel1 and not sel0;
    line3 := in3 and     sel1 and     sel0;
    val := line0 or line1 or line2 or line3;
    if (enable = 'U') then
         out1 <= 'U' ;
    elsif (enable /= '1') then
         out1 <= '0' after 2 ns;
    else
        case val is
            when '0' =>
            out1 <= '0' after tpd_hl;
            when '1' =>
            out1 <= '1' after tpd_lh;
            when others =>
            out1 <= val;
        end case;
    end if;
end process;
end only;

-------------------------------------------------------------------
-- 4-input 3-state  mux with enable
-------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4e3s is
    generic (tpd_hl : time := 2 ns;
         tpd_lh : time := 2 ns);
    port (in0, in1, in2, in3 : std_logic;
          sel0, sel1 : std_logic;
          enable : std_logic;
      out1 : out std_logic);
end mux4e3s;
architecture only of mux4e3s is
begin
    p1: process(in0, in1, in2, in3, sel0, sel1, enable)
        variable line0, line1, line2, line3 : std_logic;
        variable val : std_logic;
    begin
    line0 := in0 and not sel1 and not sel0;
    line1 := in1 and not sel1 and     sel0;
    line2 := in2 and     sel1 and not sel0;
    line3 := in3 and     sel1 and     sel0;
    val := line0 or line1 or line2 or line3;
    if (enable = 'U') then
         out1 <= 'U' ;
    elsif (enable /= '1') then
                out1 <= 'Z' after 2 ns;
    else
        case val is
            when '0' =>
                out1 <= '0' after tpd_hl;
                when '1' =>
                out1 <= '1' after tpd_lh;
                when others =>
                out1 <= val;
            end case;
    end if;
    end process;
end only;

-------------------------------------------------------------------
-- 2-input nor gate
-------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity norg is
generic (tpd_hl : time := 1 ns;
          tpd_lh : time := 1 ns);
    port (in1, in2 : std_logic;
      out1 : out std_logic);
end norg;
architecture only of norg is
begin
    p1: process(in1, in2)
        variable val : std_logic;
    begin
    val := in1 nor in2;
    case val is
        when '0' =>
        out1 <= '0' after tpd_hl;
        when '1' =>
        out1 <= '1' after tpd_lh;
        when others =>
        out1 <= val;
    end case;
    end process;
end only;

-------------------------------------------------------------------
-- 3-input nor gate
-------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity nor3g is
    generic (tpd_hl : time := 1 ns;
         tpd_lh : time := 1 ns);
    port (in1, in2, in3 : std_logic;
      out1 : out std_logic);
end nor3g;
```

```
architecture only of nor3g is
begin
    p1: process(in1, in2, in3)
        variable val : std_logic;
    begin
    val :=  not (in1 or in2 or in3);
    case val is
        when '0' =>
        out1 <= '0' after tpd_hl;
        when '1' =>
        out1 <= '1' after tpd_lh;
        when others =>
        out1 <= val;
    end case;
    end process;
end only;

--------------------------------------------------------------------
-- 4-input nor gate
--------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity nor4g is
    generic (tpd_hl : time := 1 ns;
         tpd_lh : time := 1 ns);
    port (in1, in2, in3, in4 : std_logic;
      out1 : out std_logic);
end nor4g;
architecture only of nor4g is
begin
    p1: process(in1, in2, in3, in4)
        variable val : std_logic;
    begin
    val :=  not (in1 or in2 or in3 or in4);
    case val is
        when '0' =>
        out1 <= '0' after tpd_hl;
        when '1' =>
        out1 <= '1' after tpd_lh;
        when others =>
        out1 <= val;
    end case;
    end process;
end only;

--------------------------------------------------------------------
-- 16-input nor gate
-- 16-inverted input and gate
-- uses 4 4-input NOR gates cascaded to a 4-input NAND gate followed
--      by an inverter
--------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
entity nor16g is
    generic (tpd_hl : time := 2 ns;
             tpd_lh : time := 2 ns);
    port (in0, in1, in2, in3, in4, in5, in6, in7,
          in8, in9, inA, inB, inC, inD, inE, inF : std_ulogic;
          out1 : out std_ulogic);
end nor16g;

architecture only of nor16g is
begin
    p1: process(in0, in1, in2, in3, in4, in5, in6, in7,
                in8, in9, inA, inB, inC, inD, inE, inF)
        variable val, val1, val2, val3, val4 : std_logic;
    begin
    val1 := not (in0 or in1 or in2 or in3);
    val2 := not (in4 or in5 or in6 or in7);
    val3 := not (in8 or in9 or inA or inB);
    val4 := not (inC or inD or inE or inF);
    val  := val1 and val2 and val3 and val4;
    case val is
            when '0' =>
                out1 <= '0' after tpd_hl;
            when '1' =>
                out1 <= '1' after tpd_lh;
            when others =>
                out1 <= val;
        end case;
    end process;
end only;

--------------------------------------------------------------------
-- 2-input or gate
--------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity org is
    generic (tpd_hl : time := 1 ns;
              tpd_lh : time := 1 ns);
    port (in1, in2 : std_logic;
          out1 : out std_logic);
end org;
architecture only of org is
begin
    p1: process(in1, in2)
        variable val : std_logic;
    begin
        val := in1 or in2;
        case val is
            when '0' =>
```

```
out1 <= '0' after tpd_hl;
            when '1' =>
                out1 <= '1' after tpd_lh;
            when others =>
                out1 <= val;
        end case;
    end process;
end only;

-------------------------------------------------------------------
-- 3-input or gate
-------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity or3g is
    generic (tpd_hl : time := 1 ns;
             tpd_lh : time := 1 ns);
    port (in1, in2, in3 : std_logic;
          out1 : out std_logic);
end or3g;
architecture only of or3g is
begin
    p1: process(in1, in2, in3)
        variable val : std_logic;
    begin
    val := in1 or in2 or in3;
        case val is
            when '0' =>
                out1 <= '0' after tpd_hl;
            when '1' =>
                out1 <= '1' after tpd_lh;
            when others =>
                out1 <= val;
        end case;
    end process;
end only;

-------------------------------------------------------------------
-- 4-input or gate
-------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity or4g is
    generic (tpd_hl : time := 1 ns;
             tpd_lh : time := 1 ns);
    port (in1, in2, in3, in4 : std_logic;
          out1 : out std_logic);
end or4g;
architecture only of or4g is
begin
p1: process(in1, in2, in3, in4)
        variable val : std_logic;
    begin
    val := in1 or in2 or in3 or in4;
        case val is
            when '0' =>
                out1 <= '0' after tpd_hl;
            when '1' =>
                out1 <= '1' after tpd_lh;
            when others =>
                out1 <= val;
        end case;
    end process;
end only;

-------------------------------------------------------------------
-- 2-input xor gate
-------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity xorg is
    generic (tpd_hl : time := 1 ns;
             tpd_lh : time := 1 ns);
    port (in1, in2 : std_logic;
          out1 : out std_logic);
end xorg;
architecture only of xorg is
begin
    p1: process(in1, in2)
        variable val : std_logic;
    begin
        val := in1 xor in2;
        case val is
            when '0' =>
                out1 <= '0' after tpd_hl;
            when '1' =>
                out1 <= '1' after tpd_lh;
            when others =>
                out1 <= val;
        end case;
    end process;
end only;
```

Referências Bibliográficas

[1] KEISTER W., RITCHIE A., WASHBURN S. – *The Design of Switching Circuits* – Princeton, NJ, USA – Van Nostrand, 1951

[2] VAN BERKEL C., JOSEPHS M., NOWICK S. – *Scanning the Technology: Applications of Asynchronous Circuits* – Proceedings of the IEEE Vol. 87, Number 02 – February, 1999

[3] FURBER S., GARSIDE J., GILBERT D. – *AMULET3: A High-Performance Self-Timed ARM Microprocessor* – Proceedings of the International Conference on. Computer Design (ICCD) – October, 1998

[4] GAGELDONK H., BAUMANN D., VAN BERKEL K., GLOOR D., PEETERS A., STEGMANN G. – *An Asynchronous Low-Power 80C51 Microcontroller* – Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems – 1998

[5] WUU T., VRUDHULA S. – *A Design of a Fast and Area Efficient Multi-Input Muller C-element* – IEEE Transactions on VLSI Systems – June, 1993

[6] ESCRIBA J., CARRASCO J. – *Self-timed Manchester chain carry propagate adder* – Electronics Letters Vol. 32 Number 8 – 1996

[7] ANGEL, A., SWARTZLANDER JR., E. – *A new asynchronous multiplier using enable/disable CMOS differential logic* – Proceedings of the International Conf. Computer Design (ICCD), IEEE Computer Society Press – October 1994

[8] MENG T. – *Synchronization Design for Digital Systems* – Klüwer Academic Publishers, 1991

[9] AUMANN O., PFLEIDERER H. – *Design of Self-Timed Pipelined Architectures Using Petri Nets* – Power and Timing Modeling, Optimization and Simulation (PATMOS), October 1995

[10] ARVIND D., REBELLO V. – *Optimisation of Instruction Schedules for Micronet-based Asynchronous Processors* – Proceedings of the International

Symposium on Advanced Research in Asynchronous Circuits and Systems – IEEE Computer Society Press, March 1996

[11] HULGAARD H., BURNS S., BORRIELLO G. – *Testing Asynchronous Circuits: A Survey* – Department of Computer Science and Engineering, University of Washington, Seattle, TR 94-03-06 – 1994

[12] PUCKNELL D., ESHRAGHIAN K. – *Basic VLSI Design* – Sydney, Australia – Prentice-Hall, 1988

[13] MEAD C., CONWAY L. – *Introduction to VLSI Systems* – USA – Addison-Wesley, 1980.

[14] BONDY J., MURTY U. – *Graph Theory With Applications* – Elsevier North Holland – New York, USA, 1980.

[15] HARARY F. – *Graph Theory* – ISBN 0201410338 – Perseus Press – Jan 1995.

[16] WILSON R. – *Introduction to Graph Theory* – ISBN 0582249937 – Addison-Wesley – Feb 1997.

[17] IEEE – *Verilog Language Reference Manual* – IEEE Standard 1364-1995 – New York, NY, USA – 1995.

[18] SILICON LOGIC – *Verilog FAQ* – http://www.silicon.logic.com/Verilog – May 2000.

[19] WELLSPRING SOLUTIONS – *A free Verilog compiler/simulator* – http://www.wellspring.com – May 2000.

[20] CING J. – *VBS: Verilog Behavioral Simulator* – http://www.flex.com/~jching – May 2000.

[21] ACTEL CORP. – *Actel HDL Coding Style Guide* – Actel Corporation – Sunnyvale, CA, USA – 1997.

[22] GRANJA, E., MESQUITA, A. – *Síntese de Alto Nível* – Exame de Qualificação – Programa de Engenharia Elétrica, COPPE/UFRJ, 1995.

[23] IEEE – *VHDL Language Reference Manual* – IEEE Standard 1076-1987 – New York, NY, USA – 1987.

[24] ASHENDEN P. – *The Designer's Guide to VHDL* – Morgan Kaufmann Publishers, 1996.

[25] KISHINEVSKY M., KONDRATYEV A., TAUBIN A. et al. – *Concurrent Hardware* – Chichester, England – John Wiley & Sons, 1994.

[26] SEITZ C. – *System Timing* – in Mead and Conway's Introduction to VLSI, chapter 7 – Addison-Wesley, 1980.

[27] WESTE N., ESHRAGHIAN K. – *Principles of CMOS VLSI Design – A Systems Perspective* – 2nd Edition – Addison-Wesley, 1993.

[28] CHANEY T., ORNSTEIN S., AND LITTLEFIELD W. – *Beware the Synchronizer* – IEEE 6th International Computer Conference, 1972.

[29] DAVIS A., COATES B., STEVENS K. – *Automatic Synthesis of Fast Compact Self-Timed Control Circuits* – 1993 IFIP Working Conference on Asynchronous Design Methodologies – Manchester, England, 1993.

[30] HWANG K. – Computer Arithmetic: Principles, Architecture and Design – John Wiley and Sons, 1979.

[31] RENAUDIN M. – *Architectures VLSI Asynchrones* – TELECOM Bretagne – Grenoble, FR, 1996.

[32] DEAN M., DILL D., AND HOROWITZ M. – *Self-Timed Logic Using Current-Sensing Completion Detection (CSCD)* – Proceedings of the IEEE International Conference on Computer Design – IEEE Computer Society Press, October 1991.

[33] MARTIN A. – *The Limitation of Delay-Insensitivity in Asynchronous Circuits* – Advanced Research In VLSI: Proceedings Of The Sixth MIT Conference – MIT Press, 1990.

[34] BURNS S. – *Performance Analysis and Optimization of Asynchronous Circuits* – Technical Report CS-TR-91-01, Ph.D. Thesis – California Institute of Technology, 1991.

[35] KONDRATIEV A., KISHINEVSKY M., LIN B et al. – *Basic Gate Implementation of Speed-Independent Circuits* – Proceedings of the 31st ACM/IEEE Design Automation Conference – ACM, June 1994.

[36] UNGER S. – *Asynchronous Sequential Switching Circuits* – Wiley-Interscience – New York, NY, 1969.

[37] DAVIS A., NOWICK S. – *An Introduction to Asynchronous Circuit Design* – Technical Report UUCS-97-013 – U. Of Utah, USA – September 1997.

[38] KONDRATYEV A., KISHINEVSKY M., YAKOVLEV A. – *Hazard-Free Implementation of Speed-Independent Circuits* – IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 17, No. 9 – September 1998.

[39] BEISTER J. – *A Unified Approach to Combinational Hazards* – IEEE Transactions on Computers C-23(6) – June 1974.

[40] NOWICK S. – *Automatic Synthesis of burst-mode asynchronous controllers* – Ph.D. Thesis – Stanford University Technical Report CSL-TR-95-686, December 1995.

[41] PATIL S. – *An Asynchronous Logic Array* – Technical Report Technical Memorandum 62 – Massachusetts Institute of Technology, Project MAC, 1975.

[42] CHU T. – *Automatic Synthesis and Verification of Hazard-free Control Circuits from Asynchronous FSM Specifications* – Proceedings of the IEEE International Conference on Computer Design – IEEE Computer Society Press, 1992.

[43] PURI R., GU J. – *A Modular Partitioning Approach for Asynchronous Circuit Synthesis* – Proceedings of the 31st ACM/IEEE design Automation Conference – ACM, June 1994.

[44] LAVAGNO L., MOON C., BRAYTON R. SANGIOVANNI-VINCENTELLI A. – *Solving the state assignment problem for Signal Transition Graphs* – Proceedings of the 29th IEEE/ACM Design Automation Conference – IEEE Computer Society Press, June 1992.

[45] ENDECOTT P. – *Parallel Structures for Asynchronous Microprocessors* – Computer Architecture Newsletter – IEEE Computer Society Technical Committee on Computer Architecture, October 1995.

[46] SUTHERLAND I. – *Micropipelines* – Communications of the ACM Vol. 32, No. 6 – June 1989.

[47] HULGAARD H. – *Timing Analysis and Verification of Timed Asynchronous Circuits* – Ph.D. Thesis – University of Washington, 1995.

[48] FURBER S. – Asynchronous Design for Low Power – 1997

[49] FRANKLIN M., PAN T. – *Performance Comparison of Asynchronous Adders* – Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems – – IEEE Computer Society Technical Committee on VLSI, Salt Lake City, Nov 94.

[50] PATTERSON, D., HENESSY, J. – *Computer Architecture, a Quantitative Approach* – Morgan Kaufmann, CA, USA, 1990

[51] BURKS A., GOLDSTINE H., VON NEUMANN J. – *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*, in Papers of John von Neumann on Computing and Computer Theory – MIT Press, USA, 1987

[52] SKLANSKY J. – *Conditional Sum Addition Logic* – IRE Transactions on Electronic Computers, N. 9, Jun 1960

[53] FLORES I. – *The Logic of Computer Arithmetic* – Prentice-Hall 1963.

[54] LEHMAN M., BURLA N.– *Skip Techniques for High Speed Carry Propagation in Binary Arithmetic Units* – IRE Transactions on Electronic Computers, N. 10, Dec 1961

[55] BEDRIJ J. – *Carry Select Adder* – IRE Transactions on Electronic Computers, N. 11, Jun 1962

[56] KOÇ Ç. – *RSA Hardware Implementation* – RSA Laboratories, CA, USA, 1995. Available in http://www.rsasecurity.com, Jun 24, 1999.

[57] FUJIWARA H. – *Logic Testing and Design for Testability* – The MIT Press, 1985.

[58] STRUNZ B., FLANAGAN C., HALL T. – *Design for Testability* – http://www.ul.ie/~strunz – University of Limerick, Ireland – COMETT 1994.

[59] ABRAMOVICI M., BREUER M., FRIEDMAN A. – *Digital Systems Testing and Testable Design* – IEEE Press, New York, USA – 1990.

[60] LALA P. – *Digital Circuit Testing and Testability* – ISBN 0124343309 – Academic Press – February 1997.

[61] SMITH M. – *Application Specific Integrated Circuits* –Addison-Wesley, June 1997.

[62] IEEE – *IEEE Std 1149.1-1990 with supplements 1149.1a-1993 and 1149.1b-1994: Access Port and Boundary-scan Architecture* – Ref. 1-55937-350-4 and 1-55937-497-7 – 1990-1994.

[63] LEE M. – *High-Level Test Synthesis of Digital VLSI Circuits* – ISBN 0890069077 – Artech House – February 1997.

[64] HULGAARD H., BURNS S. AND BORRIELLO G. – *Testing Asynchronous Circuits: A Survey* – Technical Report 94-03-06, Dept. Of Computer Science and Eng., U of Washington, 1994.

[65] KEUTZE K., LAVAGNO L., SANGIOVANNI-VINCENTELLI A. – *Synthesis for Testability Techniques for Asynchronous Circuits* – Proceedings of the IEEE International Conference on Computer-Aided Design – IEEE Computer Society Press, November 1991.

[66] STAUNSTRUP J., GREENSTREET M. – *Synchronized Transitions*, in Formal Methods for VLSI Design – North Holland, Elsevier 1990.

[67] HAZEWINDUS P. – *Testing Delay Insensitive Circuits* – Ph. D. Thesis – California Institute of Technology, 1992.

[68] LIN C., REDDY S. – *On delay fault testing in Logic Circuits* – IEEE Transactions on Computer Aided Design, Sep 1987.

[69] SALOMÃO S. – *Uma Proposta em Hardware para o Algoritmo Criptográfico IDEA* – Tese M.Sc. – COPPE/UFRJ, Rio de Janeiro, Brasil, 1997.

[70] VIEIRA A. – Estudo do algoritmo criptográfico RSA visando sua implementação em eletrônica digital – Tese de M.Sc. – COPPE/UFRJ – Rio de Janeiro, Brasil, 1999.

[71] SCHEIER B. – *Applied Cryptography* – 2nd Edition – John Wiley & Sons, 1996.

[72] RITTER T. – *Ritter's Crypto Glossary and Dictionary of Technical Cryptography* – http://www.io.com/~ritter, Jun 24, 1999.

[73] RSA Data Security Inc. – *RSA Laboratories FAQ about Today's Cryptography* – http://www.rsasecurity.com, Jun 24, 1999.

[74] ANSI X3.106 – *American National Standard for Data Encryption Algorithm* - American National Standard Institute, 1981

[75] SCHNEIER B. – *Description of a New Variable-length Key, 64-bit Block Cipher (Blowfish)* – Cambridge Security Workshop Proceedings, pp. 191-204 – Springer-Verlag, December 1993.

[76] RIVEST R., SHAMIR A., ADLEMAN L. – *A Method for Obtaining Digital Signatures and Public-key Cryptosystems* – Communications of the ACM, vol. 21, pp. 120-126, 1978.

[77] BRICKELL E., ODLYZKO A. – *Cryptanalysis: A survey of Recent Results* – Proceedings of the IEEE – Vol. 76, no. 5, pp. 578-593, Oct 1993.

[78] WEINER M. – *Efficient DES Key Search* – Advances in Cryptology – CRYPTO '93 Proceedings – Springer-Verlag

[79] AKELLA V., GOPALAKRISHNAM G. – *Flow analysis techniques in High-Level Asynchronous Circuit Synthesis* – Technical Report, U. Of California, Davis – 1999.

[80] KONDRATYEV A., KISHINEVSKY M., CORTADELLA J., LAVAGNO L., YAKOVLEV A – *Asynchronous Interface Specification, Analysis and Synthesis* – Technical Report, Universitat Politècnica de Catalunya – March 1998.

[81] SAITO H., KONDRATYEV A., CORTADELLA J., et al. – *What is the Cost of Delay Insensitivity?* – IEEE/ACM International Conference on Computer-Aided Design, ICCAD '99 – IEEE Press, 1999.

[82] NEGULESCU R. – *A Technique for Finding and Verifying Speed-dependencies in Gate Circuits* – Research Report CS-97-28 – Computer Science Dept, U. Of Waterloo, Canada – August 1997.

[83] CHAKRABORTY S., DILL D., YUN K. – *Min-max Timing Analysis and an Application to Asynchronous Circuits* – Proceedings of the IEEE, 87(2):332-346 – February 1999.

[84] YUN K., DILL D. – *Automatic Synthesis of Extended Burst-mode Circuits, Parts I and II* – IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, No. 2 – February 1999.

[85] CHU T. – *Synthesis of Self-timed VLSI Circuits from Graph-theoretic specifications* – Ph.D. Thesis, MIT Laboratory for Computer Science – Cambridge, MA, USA – June 1987.

[86] VANBEKBERGEN P. – *Synthesis of Asynchronous Control Circuits from Graph-Theoretic Specifications* – Ph.D. Thesis, Catholic University of Leuven, Belgium – September 1993.

[87] APPLETON S., MORTON S., LIEBELT M. – *Two-Phase Asynchronous Pipeline Control* – Proceedings of the 3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems, 1997.

[88] CORTADELLA J., KISHINEVSKY M., KONDRATYEV A., LAVAGNO L., YAKOVLEV A – *Petrify: A Tool For Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers* – IEICE Transactions on Information and Systems – March 1997.

[89] MEINCKE T., HEMANI A., KUMAR S., et al. – *Globally Asynchronous Locally Synchronous Architecture for Large High Performance Asics* – Proceedings of the International Symposium on Circuits and Systems (ISCAS) '99.

[90] IBM RESEARCH INC. – *Interlocked Pipelined CMOS* – http://forum.iee.org.uk/news/items/00013701.htm – February 2000.

[91] YUN K., DONOHUE R. – *Pausible Clocking: A First Step Toward Heterogeneous Systems* – Proceedings of the International Conference on Computer Design (ICCD) – October, 1996.

[92] RENAUDIN M., EL HASSAN B. – The Design of Fast Asynchronous Adder Structures and Their Implementation Using DCVS Logic – International Symposium on Circuits and Systems, 1994.

[93] KINNIMENT D., GARSIDE J., GAO B. – *A comparison of power consumption in some CMOS adder circuits* – Power and Timing Modeling, Optimization and Simulation (PATMOS), Oldenburg, Germany– IEEE Press, October, 1995.

[94] KINNIMENT D. – *An Evaluation of Asynchronous Addition* – IEEE Transactions on VLSI Systems Vol. 4, No. 1– IEEE Press, March 1996.

[95] ESCRIBÁ J., CARRASCO J. – *Self-timed Manchester Carry Chain Propagate Adder* – Electronics Letters, vol.32, No. 8, April 1996.

[96] NOWICK S., YUN K., BEEREL P., and DOOPLY A. – *Speculative Completion for the Design of High-Performance Adders* – Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems – IEEE Computer Society Press, April 1997.

[97] JOHNSON D., AKELLA V. – *Design and Analysis of Asynchronous Adders* – IEE Proceedings, Computer Digital Techniques, Vol. 145, No. 1 – January 1998.

[98] CORSONELLO P., PERRI S., COCORULLO G. – *A 56-bit Self-timed Adder for High Speed Asynchronous Datapath* – Proceedings of the 6th International Conference on Electronics, Circuits and Systems – IEEE Press, 1999.

[99] MULLER J., TISSERAND A. – *Asynchronous Sub-logarithmic Adders* – 1997 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, Vol.2 – IEEE Press 1997.

[100] JOHNSON D., AKELLA V. STOTT B. – *Micropipelined Asynchronous Discrete Cosine Transform (DCT/IDCT) Processor* – IEEE Transactions on Very large Scale Integration (VLSI) Systems, Vol. 6 No. 4 – December 1998.

[101] WOODS J., DAY P., FURBER S. et al. – *AMULET1: An Asynchronous ARM Microprocessor* – IEEE Transactions on Computers, Vol. 48 – April 1997.

[102] MARTIN A., BURNS S., LEE T. et al. – *Design of an Asynchronous Microprocessor* – Proceedings of the 1989 Conference on Advanced Research in VLSI – MIT Press 1989.

[103] PETLIN O., FURBER S. – *Built-in Self-Testing of Micropipelines* – Proceedings of the 3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems – IEEE Press, 1997.

[104] SCHÖBER V., KIEL T. – *An asynchronous scan-path concept for micropipelines using the bundled data convention* – Proceedings of the International Test Conference – Oct, 1996.

[105] BANERJEE S., ROY R., CHAKRADHAR S. – *Initialization Issues in Asynchronous Circuit Synthesis* – Journal of Electronic Testing: Theory and Applications – Dec 1996.

[106] DIJKSTRA, E. – *Two Starvation Free Solutions to a General Exclusion Problem* – EWD – 625, Plataanstraat 5, 5671 AL Nuenen, The Netherlands – 1965.

[107] DIJKSTRA E. – *Cooperating Sequential Processes* – In Programming Languages, F. Genuys Editor – Academic Press, New York, USA, 1968.

[108] CHANDY K., MISRA J. – *The Drinking Philosophers Problem* – ACM Transactions of Programming Languages and Systems v.6 no.4 – October 1984.

[109] KRAMER J., MAGEE J. – *The Evolving Philosophers Problem: Dynamic Change Management* – IEEE Transactions on Software Engineering, v. 16, no. 11 – November 1990.

[110] HULGAARD H. – *Timing Analysis and Verification of Timed Asynchronous Circuits* – Ph.D. Thesis – University of Washington, WA, USA –1995.

[111] FRANÇA F. – *Neural Networks as Neighbourhood Constrained Systems* – Ph.D. Thesis – Imperial College, London, UK – May 1994.

[112] BARBOSA V., GAFNI E. – *Concurrency in Heavily Loaded Neighborhood-Constrained Systems* – ACM Transactions on Programming Languages and Systems, Vol. 11, No. 4, October 1989.

[113] BERGÉ C. – *Graphs and Hypergraphs* – North Holland – Amsterdam, The Netherlands, 1976.

[114] LAMPORT L. – *Time, Clocks and the Ordering of Events in a Distributed System* – Communications of the ACM, Vol. 21, No. 7, July 1978.

[115] BARBOSA V., LIMA P. – *On the Distributed Parallel Simulation of Hopfield's Neural Networks* – Software – Practice and Experience Vol. 20, No. 10 – 1990.

[116] GRANJA E; FRANÇA F.; ALVES V. – *Asynchronous Digital Circuits as Neighborhood-Constrained Systems* – Exame de Qualificação – Relatório Técnico ES-357/95, Programa de Engenharia de Sistemas, COPPE/UFRJ , 1995.

[117] FRANÇA F., ALVES V., GRANJA E. – *Edge Reversal-based Asynchronous Timing Synthesis* – Proceedings of the International Symposium on Circuits and Systems – Monterey, CA, USA , June 1998.

[118] GRANJA E., CHAVES FILHO E. – *Projeto de uma ALU Assíncrona* – Exame de Qualificação – Relatório Técnico ES-414/96, COPPE/UFRJ 1996.

[119] SUN MICROSYSTEMS – *The SPARC Architecture Manual* – Sun Microsystems 1987.

[120] STANDARD PERFORMANCE EVALUATION CORP. Open Systems Group – *SPEC CPU92 Benchmarks* [Online] – Available: http://open.specbench.org/ [1996, Aug 14]

[121] STANDARD PERFORMANCE EVALUATION CORP. Open Systems Group – *SPEC CPU95 Benchmarks* [Online] – Available: http://open.specbench.org/ [1996, Aug 14]

[122] CHAVES FILHO, E. – *Arquiteturas Super Escalares: Efeito de Alguns Parâmetros sobre o Desempenho* – Tese D. Sc., Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, 1994.

[123] BERGÉ J., FONKUA A., MAGINOT S., ROUILLARD J. — *VHDL '92* — Kluwer Academic Publishers, Boston, USA, 1993

[124] ASHENDEN, P. — *The VHDL Cookbook* — U. of Adelaide, Australia: 1990

[125] PUCKNELL, D., ESHRAGHIAN, K. — *Basic VLSI Design* — Prentice-Hall, Australia, 1988

[126] FRANÇA F; ALVES V.; GRANJA E – *A Multi-phase Asynchronous Timing Scheme* – Proceedings of the X Brazilian Symposium on Integrated Circuit Design – Gramado, RS , 1997.

[127] FRANÇA F; ALVES V.; GRANJA E – *Processo de Síntese e Aparelho para Temporização Assíncrona de Circuitos e Sistemas Digitais Multifásicos* – Patente BR PI 9703819-9, Brasil.

[128] FRANÇA F., ALVES V., GRANJA E. – *A BIST Scheme for Asynchronous Logic* – Proceedings of the Asian Test Symposium, Singapore, 1998.

[129] KHOCHE A., BRUVAND E. – *A Partial Scan Methodology for Testing Self-Timed Circuits* – 2nd Working Conference on Asynchronous Design Methodologies – ACiD-WG, IEEE Computer Society – London, UK – May 1995.

[130] SALOMÃO S., ALCÂNTARA J., ALVES V. et al. – *SCOB, a Soft-core for the Blowfish Cryptographic Algorithm* – Proceedings of the XII SBCCI, Natal, RN 1999.

[131] ALCÂNTARA J., SALOMÃO S., GRANJA E. et al. – *Synchronous to Asynchronous Conversion, A Case Study: The Blowfish Algorithm*

Implementation – X IFIP International Conference on VLSI, Lisbon 1999 – in VLSI: Systems on a Chip, Kluwer Publishing 1999.

[132] SCHNEIER B. – *The Blowfish Encryption Algorithm - One Year Later* – Dr. Dobb's Journal, Sep. 1995.

[133] WEINER M. – *Efficient DES Key Search* – Advances in Cryptology – CRYPTO '93 Proceedings, Springer-Verlag 1994.

[134] MATSUI M. – *Linear Cryptanalysis Method for DES Cipher* – Advances in Cryptology – CRYPTO '93 Proceedings, Springer-Verlag 1994.

[135] BIHAM E., SHAMIR A. – Differential Cryptanalysis of the Data Encryption Standard – Springer-Verlag 1993.