UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de Ilha Solteira

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

"Desenvolvimento de uma Arquitetura Multiprocessada e Reconfigurável para a Síntese de Redes de Petri em *Hardware*"

TIAGO DE OLIVEIRA

Orientador: Prof. Dr. Norian Marranghello

Tese apresentada à Faculdade de Engenharia - UNESP – Campus de Ilha Solteira, para obtenção do título de Doutor em Engenharia Elétrica.
Área de Conhecimento: Automação.

Ilha Solteira – SP
março/2008

FICHA CATALOGRÁFICA

Elaborada pela Seção Técnica de Aquisição e Tratamento da Informação
Serviço Técnico de Biblioteca e Documentação da UNESP - Ilha Solteira.

O48d

Oliveira, Tiago de
Desenvolvimento de uma arquitetura multiprocessada e reconfigurável para a síntese de redes de Petri em hardware / Tiago de Oliveira. -- Ilha Solteira : [s.n.], 2008
229 f. : il. (algumas color.), + 1 CD-ROM

Tese (doutorado) - Universidade Estadual Paulista. Faculdade de Engenharia de Ilha Solteira. Área de concentração: Automação, 2008

Orientador: Norian Marranghello
Bibliografia: p. 188-194

1. Arquitetura de computador. 2. Redes de Petri. 3. Síntese de sistemas. 4. Arquitetura reconfigurável.

Agradecimentos

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

À minha família, em especial, minha mãe Conceição e aos meus irmãos Classius e Gláucia, pelo apoio, incentivo e dedicação que permitiram concentrar minhas atenções neste trabalho.

Ao Prof. Dr. Norian Marranghello pela sua orientação, atenção, amizade e valiosas informações técnicas.

A todos os meus professores e amigos pela confiança e companheirismo que sempre me dedicaram.

Ao CNPq pela concessão da bolsa de estudos durante o período de desenvolvimento deste projeto.

OLIVEIRA, Tiago. *Desenvolvimento de uma arquitetura multiprocessada e reconfigurável para a síntese de redes de petri em hardware.* 2008. 229 f. Tese (Doutorado em Engenharia Elétrica) – Faculdade de Engenharia de Ilha Solteira, Universidade Estadual Paulista, Ilha Solteira, 2008.

Resumo

O objetivo desta tese é o desenvolvimento de uma arquitetura multiprocessada e reconfigurável que permita a implementação física de sistemas de controle descritos por meio de Redes de Petri coloridas de arcos constantes T-temporizadas e que possuam probabilidade de disparo nas transições. A arquitetura pode ser utilizada para implementar sistemas de controle (e não para a avaliação das propriedades da Rede de Petri), permitindo a implementação física por meio de mapeamento tecnológico diretamente no nível comportamental, sem a necessidade de se utilizar um processo de síntese de alto nível para descrever o sistema em equações booleanas e tabelas de transição de estados. A arquitetura é composta por um arranjo de blocos de configuração denominados BCERPs, por blocos reconfiguráveis denominados BCGNs e por um sistema de comunicação, implementado por um conjunto de roteadores. Os blocos BCERPs podem ser configurados para implementar as transições da Rede de Petri e seus respectivos lugares de entrada. Blocos BCGNs são utilizados pelos blocos BCERPs para a geração de números pseudo-aleatórios. Estes números podem definir a probabilidade de disparo das transições e também podem ser usados no processo de resolução de conflito, que ocorre quando uma transição possuir um ou mais lugares de entrada compartilhados com outras transições. O sistema de comunicação possui uma topologia de grelha, tendo como principal função o roteamento e armazenamento de pacotes entre os blocos de configuração. Os roteadores e blocos de configuração BCERPs e BCGNs foram descritos em VHDL e implementados em FPGAs.

Palavras-Chaves: Arquitetura Reconfigurável, FPGA, Síntese de Sistemas, Redes de Petri.

OLIVEIRA, Tiago. *Design of a reconfigurable multiprocessed architecture to synthesize petri nets in hardware.* 2008. 229 f. Tese (Doutorado em Engenharia Elétrica) – Faculdade de Engenharia de Ilha Solteira, Universidade Estadual Paulista, Ilha Solteira, 2008.

Abstract

The goal of this thesis is to develop a reconfigurable multiprocessed architecture that allows the physical implementation of systems described by T-timed colored Petri nets with constant arcs having transitions with firing probabilities. The architecture can be used to implement control systems (not to evaluation Petri net properties). With this architecture, physical implementation of systems can be achieved through technology mapping directly from behavioral level, without the need to go through an expensive high level synthesis process to describe the system into boolean equations and state transition tables. The architecture comprises an array of configuration blocks named BCERPs; reconfigurable blocks named BCGNs; and a communication system implemented using a set of routers. BCERP blocks can be configured to implement Petri net transitions as well as the corresponding input places. BCGN blocks are used by BCERPs for pseudo random number generation. These numbers can define transitions firing probabilities. They can also be used for conflict resolution, which happens when two or more transitions share one or more input places. The communication system presents a grid topology. Its main functions are packet storage and routing among configuration blocks. The routers, BCGNs and BCERPs configuration blocks were described in VHDL and implemented in FPGAs.

Keywords: Reconfigurable Architecture, FPGA, System Synthesis, Petri Nets.

Lista de Figuras

1.1 Processo de mapeamento tecnológico proposto . . . . . p. 25

2.1 Rede de Petri e a sua correspondente máquina de estados . . . . . p. 32

2.2 (a) Rede de Petri, (b) Controlador configurado por meio de uma Rede de Petri . . . . . p. 34

2.3 Rede de Petri a ser sintetizada . . . . . p. 36

2.4 Módulo da transição . . . . . p. 37

2.5 Seletor de transição do Achilles . . . . . p. 38

2.6 Simulação da Rede de Petri implementada . . . . . p. 39

2.7 Lógica do lugar . . . . . p. 40

2.8 Lógica da transição . . . . . p. 40

2.9 Diagrama de blocos do seletor . . . . . p. 40

2.10 Exemplo de Rede de Petri particionada em blocos de um lugar e uma transição . . . . . p. 41

2.11 Descrição dos blocos de implementação . . . . . p. 42

2.12 Esquema de conexão de uma Rede de Petri . . . . . p. 43

3.1 Estrutura de um pacote numa Rede-em-*Chip* . . . . . p. 47

3.2 Roteador com filas na entrada . . . . . p. 49

3.3 Roteamento (a) *store-and-forward* e (b) *cut-through* . . . . . p. 52

3.4 (a) Dependência cíclica – *deadlock*, (b) ciclos possíveis numa topologia de grelha e (c) roteamento X-Y (livre de *deadlock*) . . . . . p. 56

3.5 Comportamento da rede (a) sem o uso de canais virtuais e (b) com o uso de canais virtuais . . . . . p. 57

3.6 Rede-em-*chip* CLICHE com 16 primitivas, sendo R = **R**oteador e IR = **I**nterface de **R**ede . . . . . p. 58

5.1 Arquitetura proposta, onde R indica um **R**oteador, BCGN significa **B**loco de **C**onfiguração do **G**erador de **N**úmeros pseudo-aleatórios e BCERP indica um **B**loco básico de **C**onfiguração dos **E**lementos de uma **R**ede de **P**etri . . . . . p. 77

5.2 Composição de um pacote . . . . . p. 80

5.3 Situação de conflito numa Rede de Petri . . . . . p. 81

5.4 Transição habilitada para o disparo . . . . . p. 82

5.5 Exemplo de um modelo em uma Rede de Petri colorida de arcos constantes T-temporizada . . . . . p. 84

5.6 Mapeamento de uma Rede de Petri nos blocos BCERPs da arquitetura proposta . . . . . p. 85

5.7 Arquitetura numa topologia 3-D . . . . . p. 86

5.8 Interface de entrada/saída dos roteadores (a) com cinco canais e (b) com seis canais . . . . . p. 86

5.9 Pacote utilizado na comunicação entre camadas . . . . . p. 87

6.1 Interface de entrada/saída do roteador . . . . . p. 91

6.2 Diagrama de blocos da primeira arquitetura para o roteador . . . . . p. 93

6.3 Diagrama de blocos da segunda arquitetura para o roteador . . . . . p. 94

6.4 Diagrama de blocos da terceira arquitetura para o roteador . . . . . p. 95

6.5 As unidades de geração e tratamento de requisições . . . . . p. 98

6.6 Unidade de geração de requisição . . . . . p. 100

6.7 Unidade de tratamento de requisição . . . . . p. 100

6.8 Código VHDL do bloco de verificação de ocorrência de zero . . . . . p. 101

6.9 Código VHDL da lógica de roteamento . . . . . p. 102

6.10 Código VHDL do decrementador x . . . . . p. 103

6.11 Código VHDL para o armazenamento de pacote . . . . . p. 104

6.12 Código VHDL para o armazenamento do sinal de saída `pap` . . . . . p. 105

6.13 Código VHDL do registrador de prioridade . . . . . p. 106

6.14 Código VHDL da lógica de atualização . . . . . p. 106

6.15 Código VHDL do seletor dinâmico de requisição . . . . . p. 107

6.16 Código VHDL do seletor de pacote . . . . . p. 109

7.1 Composição do pacote que deve ser enviado à unidade de geração pelo bloco BCERP . . . . . p. 117

7.2 Composição do pacote que deve ser enviado pela unidade de geração para o bloco BCERP que solicitou um número pseudo-aleatório . . . . . p. 118

7.3 Bloco responsável pela geração do número pseudo-aleatório . . . . . p. 121

7.4 Arquitetura da unidade de geração de números pseudo-aleatórios . . . . p. 122

7.5 Fluxograma da unidade de controle de pacotes recebidos . . . . . p. 123

7.6 Ordem dos pacotes de configuração que devem ser direcionados à unidade de geração de números pseudo-aleatórios . . . . . p. 124

7.7 Fluxograma da unidade de controle de pacotes a enviar . . . . . p. 125

7.8 Código VHDL do sub-bloco de armazenamento do penúltimo número gerado . . . . . p. 127

7.9 Código VHDL do sub-bloco de armazenamento do parâmetro de multiplicação . . . . . p. 128

7.10 Código VHDL do sub-bloco de armazenamento do controle do sinal de transporte . . . . . p. 129

7.11 Código VHDL do sub-bloco de armazenamento do sinal de Transporte . p. 130

7.12 Código VHDL do sub-bloco de multiplicação de dois operandos . . . . p. 131

7.13 Código VHDL do sub-bloco de soma de dois operandos . . . . . p. 131

7.14 Código VHDL do bloco que memoriza os endereços enviados pelos blocos BCERPs da arquitetura proposta . . . . . p. 132

7.15 Código VHDL do bloco de controle da memória de endereços . . . . . p. 133

7.16 Código VHDL do bloco Contador de Endereços . . . . . p. 134

7.17 Código VHDL de armazenamento de estados do controle de pacotes recebidos . . . . . p. 135

7.18 Código VHDL do bloco de controle de pacotes recebidos . . . . . p. 136

7.19 Código VHDL do bloco de controle de pacotes a enviar . . . . . p. 138

7.20 Código VHDL do bloco de armazenamento do sinal PAP . . . . . p. 139

7.21 Tempo de geração no FPGA, onde E.L. significa **E**lementos **L**ógicos e MULT. significa **Mult**iplicadores . . . . . p. 141

7.22 Tempo de geração no computador . . . . . p. 142

8.1 Mapeamento em blocos BCERPs de uma transição que possui conflito estrutural . . . . . p. 147

8.2 Diagrama simplificado da arquitetura do bloco BCERP . . . . . p. 149

8.3 Composição da carga útil no processo de execução da Rede de Petri implementada . . . . . p. 150

8.4 Instruções de processamento dos pacotes de dados . . . . . p. 150

8.5 Composição da carga útil no processo de configuração da arquitetura proposta . . . . . p. 151

8.6 Formatação dos pacotes de saída, onde B.M. indica um dado proveniente do **B**anco de **M**emória . . . . . p. 161

8.7 Blocos BCERPs auxiliares que implementam transições de convergência para possibilitar o mapeamento de um número maior de tipos diferentes de marcas . . . . . p. 162

8.8 Blocos BCERPs escravos para possibilitar o envio de um número maior de pacotes de dados . . . . . p. 163

8.9 Bloco BCERP auxiliar para as subtrações das marcas no processo de resolução de conflito . . . . . p. 163

8.10 Sistema utilizado para o teste do bloco BCERP no FPGA . . . . . p. 164

9.1 Definição de uma arquitetura 3x3 utilizada nos processos de simulação e de teste . . . . . p. 168

9.2 Pacotes inseridos no sistema de comunicação da arquitetura 3x3 . . . . p. 169

9.3 Descrição de uma Rede de Petri utilizada nos processos de simulação e de teste da arquitetura proposta . . . . . p. 170

9.4 Mapeamento da Rede de Petri na arquitetura proposta . . . . . p. 170

9.5 Sistema utilizado para o teste da arquitetura 3x3 no FPGA . . . . . p. 182

A.1 Sistema digital com duas possibilidades diferentes de implementação . . p. 196

A.2 Cálculo da quantidade de níveis de lógica para portas com (a) dois operandos, (b) três operandos, (c) quatro operandos e (d) cinco operandos p. 199

A.3 Cálculo da quantidade de níveis de lógica para portas com n operandos p. 200

A.4 Uso das funções de máximo e soma no cálculo da quantidade de níveis de lógica, onde NL é o número de níveis de lógica . . . . . p. 201

A.5 Cálculo da quantidade de portas lógicas *and* com (a) dois operandos, (b) três operandos, (c) quatro operandos e (d) cinco operandos . . . . . p. 202

A.6 Cálculo da quantidade de portas lógicas com n operandos . . . . . p. 202

A.7 Somador com transporte em cascata . . . . . p. 203

A.8 Circuito somador completo utilizado na composição do somador com transporte em cascata . . . . . p. 203

A.9 Circuito decrementador com transporte em cascata para o projeto do roteador . . . . . p. 204

A.10 Caminho crítico para o decrementador com transporte em cascata . . . p. 205

A.11 Somador com transporte antecipado . . . . . p. 206

A.12 Caminho crítico para o decrementador com transporte antecipado . . . p. 208

A.13 Disposição dos operandos lógicos *and* no decrementador com transporte antecipado . . . . . p. 214

A.14 Disposição dos operandos lógicos *and* no decrementador com transporte antecipado modificado . . . . . p. 215

A.15 Somador com transporte selecionado . . . . . p. 216

A.16 Processo de multiplexação do somador com transporte selecionado para C_M . . . . . p. 216

A.17 Decrementador com transporte selecionado para o projeto do roteador . p. 217

A.18 Caminho crítico para o decrementador com transporte selecionado . . . p. 218

A.19 Panorama 1 – Comparação de desempenho entre os decrementadores . p. 222

A.20 Panorama 2 – Desempenho do roteador com os decrementadores inseridos na arquitetura . . . . . p. 223

A.21 Comparação dos decrementadores inseridos na arquitetura do roteador para cada peso especificado . . . . . p. 226

Lista de Tabelas

6.1 Especificações de alguns roteadores . . . . . p. 110

8.1 Decodificador de pacotes: tabela de transição de estados na fase de configuração . . . . . p. 152

8.2 Decodificador de pacotes: tabela de transição de estados na fase de execução da Rede de Petri . . . . . p. 153

8.3 Análise de Disparo: tabela de transição de estados referente à ativação de componentes . . . . . p. 154

8.4 Análise de Disparo: tabela de transição de estados referente a busca de um novo número pseudo-aleatório . . . . . p. 154

8.5 Análise de Disparo: tabela de transição de estados para a comunicação com a lógica de envio de pacotes . . . . . p. 155

8.6 Análise de Disparo: tabela de transição de estados para a comunicação com os componentes de resolução de conflito e de processo de disparo . p. 156

8.7 Resolução de Conflito: tabela de transição de estados . . . . . p. 157

8.8 Processo de Disparo: tabela de transição de estados . . . . . p. 158

8.9 Lógica de Envio de Pacotes: tabela de transição de estados para a funções solicitadas . . . . . p. 159

8.10 Lógica de Envio de Pacotes: tabela de transição de estados para a formatação dos pacotes . . . . . p. 160

8.11 Pacotes de configuração armazenados na memória RAM . . . . . p. 165

8.12 Pacotes de dados armazenados na memória RAM . . . . . p. 165

9.1 Pacotes de configuração do BCERP(1,1) . . . . . p. 171

9.2 Pacotes de configuração do BCERP(1,2) . . . . . p. 173

9.3 Pacotes de configuração do BCERP(1,3) . . . . . p. 174

9.4 Pacotes de configuração do BCERP(2,1) . . . . . p. 176

9.5 Pacotes de configuração do BCERP(2,3) . . . . . p. 177

9.6 Pacotes de configuração do BCERP(3,1) . . . . . p. 179

9.7 Pacotes de configuração do bloco BCGN . . . . . p. 180

9.8 Pacotes de dados armazenados na memória ROM e utilizados no processo de execução da Rede de Petri . . . . . p. 182

9.9 Marcação da Rede de Petri . . . . . p. 183

A.1 Acréscimo de portas lógicas de um termo para outro . . . . . p. 210

Abreviaturas

AHDL	***A**ltera **H**ardware **D**escription **L**anguage*
BCERP	**B**loco básico de **C**onfiguração dos **E**lementos de uma **R**ede de **P**etri
BCGN	**B**loco de **C**onfiguração do **G**erador de **N**úmeros pseudo-aleatórios
CAD	***C**omputer-**A**ided **D**esign*
CLB	***C**onfigurable **L**ogic **B**lock*
CLICHE	***C**hip-**L**evel **I**ntegration of **C**ommunicating **H**eterogeneous **E**lements*
CPLD	***C**omplex **P**rogrammable **L**ogic **D**evice*
FPGA	***F**ield-**P**rogrammable **G**ate **A**rray*
IBM	***I**nternational **B**usiness **M**achines*
IOB	***I**nput/**O**utput **B**uffer*
LUT	***L**ook**U**p **T**ables*
NoC	***N**etwork-**o**n-**C**hip*
OS4RS	***O**perating **S**ystem **for** **R**econfigurable **S**ystems*
OSI	***O**pen **S**ystems **I**nterconnection*
PAL	***P**rogrammable **A**rray **L**ogic*
ParIS	***Par**ameterizable **I**nterconnect **S**witch for networks-on-chip*
PLA	***P**rogrammable **L**ogic **A**rray*
PLD	***P**rogrammable **L**ogic **D**evice*
RAM	***R**andom **A**ccess **M**emory*
ROM	***R**ead **O**nly **M**emory*
RTL	***R**egister **T**ransfer **L**anguage*
SoCIN	***S**ystem-**o**n-**C**hip **I**nterconnection **N**etwork*
SoC	***S**ystem-**o**n-**C**hip*
SPIN	***S**calable **P**rogrammable **I**ntegrated **N**etwork*
VHDL	***V**HSIC **H**ardware **D**escription **L**anguage*
VHSIC	***V**ery-**H**igh-**S**peed **I**ntegrated **C**ircuits*

Sumário

1 Introdução p. 21

Resumo . . . . . p. 21

1.1 Contextualização . . . . . p. 21

1.2 Objetivo e Justificativas do Projeto . . . . . p. 23

1.3 Ferramentas Utilizadas no Desenvolvimento do Projeto . . . . . p. 27

1.4 Organização do Texto . . . . . p. 28

2 Implementações Físicas Existentes de Sistemas Modelados em Redes de Petri p. 30

Resumo . . . . . p. 30

2.1 Introdução . . . . . p. 30

2.2 Formas de Implementação Física de Redes de Petri . . . . . p. 31

2.3 Sistema Multiprocessado Controlado por Modelos em Redes de Petri . p. 33

2.4 Arquitetura Achilles para Implementar Modelos Descritos em Redes de Petri . . . . . p. 35

2.5 Exemplo Prático: Síntese de uma Rede de Petri num FPGA . . . . . p. 36

2.6 O Seletor de Transição . . . . . p. 39

2.7 Utilização de Recursos num FPGA . . . . . p. 41

2.8 Outras Formas de Implementação . . . . . p. 43

2.9 Comentários . . . . . p. 44

3 Redes-em-*Chip* p. 45

Resumo . . . . . p. 45

3.1 Introdução . . . . . p. 45
3.2 Alguns Conceitos sobre Redes-em-*Chip* . . . . . p. 46
3.2.1 Topologia . . . . . p. 48
3.2.2 Memorização . . . . . p. 48
3.2.3 Arbitragem . . . . . p. 50
3.2.4 Chaveamento . . . . . p. 51
3.2.5 Controle de Fluxo . . . . . p. 51
3.2.6 Roteamento . . . . . p. 52
3.3 *Starvation*, *Livelock* e *Deadlock* . . . . . p. 55
3.4 A Rede-em-*Chip* CLICHE . . . . . p. 57
3.5 Outras Redes-em-*Chip* . . . . . p. 59
3.6 Comunicação numa Rede-em-*Chip* . . . . . p. 60
3.7 Comentários . . . . . p. 61

4 Geração de Números Pseudo–Aleatórios p. 62
Resumo . . . . . p. 62
4.1 Introdução . . . . . p. 62
4.2 Gerador Linear Congruente . . . . . p. 63
4.3 Gerador Linear Múltiplo . . . . . p. 64
4.4 Gerador Linear com Transporte . . . . . p. 65
4.5 Gerador Linear do Tipo *Feedback Shift Register* . . . . . p. 67
4.6 Gerador Linear do Tipo *Generalized Feedback Shift Register* . . . . . p. 68
4.7 Gerador do Tipo *Lagged-Fibonacci* . . . . . p. 68
4.8 Gerador *Mersenne Twister Generalized Feedback Shift Register* . . . . . p. 69
4.9 Geradores Combinados . . . . . p. 70
4.10 Geradores Disponíveis . . . . . p. 72
4.11 Comentários . . . . . p. 73

5 Arquitetura Proposta para Implementar Fisicamente Sistemas Modelados em Redes de Petri p. 74

Resumo . . . . . p. 74

5.1 Introdução . . . . . p. 74

5.2 A Arquitetura Proposta . . . . . p. 76

5.2.1 Topologia . . . . . p. 78

5.2.2 Roteador . . . . . p. 79

5.2.3 Bloco de Configuração BCGN . . . . . p. 80

5.2.4 Bloco de Configuração BCERP . . . . . p. 81

5.2.5 Semântica da Rede de Petri Implementada . . . . . p. 82

5.2.6 Estendendo a Arquitetura para uma Estrutura 3-D . . . . . p. 85

5.3 Comentários . . . . . p. 89

6 Roteador: Responsável pelo Sistema de Comunicação da Arquitetura Proposta p. 90

Resumo . . . . . p. 90

6.1 Introdução . . . . . p. 90

6.2 Funcionamento do Roteador . . . . . p. 91

6.3 Arquitetura do Roteador . . . . . p. 92

6.3.1 Primeira Arquitetura . . . . . p. 93

6.3.2 Segunda Arquitetura . . . . . p. 94

6.3.3 Terceira Arquitetura . . . . . p. 95

6.4 Comparações das Arquiteturas . . . . . p. 96

6.5 Aspectos de Implementação . . . . . p. 97

6.5.1 Verificação de Ocorrência de Zero . . . . . p. 100

6.5.2 Lógica de Roteamento . . . . . p. 101

6.5.3 Decrementador x . . . . . p. 103

6.5.4 Registrador - Pacote . . . . . p. 103
6.5.5 Registrador - Estado . . . . . p. 104
6.5.6 Registrador - Prioridade . . . . . p. 104
6.5.7 Lógica de Atualização . . . . . p. 106
6.5.8 Seletor Dinâmico . . . . . p. 107
6.5.9 Seletor de Pacote . . . . . p. 108
6.6 Síntese e Simulação do Roteador . . . . . p. 108
6.7 Resultados . . . . . p. 110
6.8 Roteadores na Topologia 3-D . . . . . p. 111
6.9 Comentários . . . . . p. 113

7 BCGN: Bloco de Configuração do Gerador de Números Pseudo-Aleatórios p. 115

Resumo . . . . . p. 115
7.1 Introdução . . . . . p. 115
7.2 Funcionamento da Unidade de Geração . . . . . p. 116
7.3 Configuração da Unidade de Geração . . . . . p. 118
7.4 Arquitetura da Unidade de Geração . . . . . p. 120
7.5 Aspectos de Implementação . . . . . p. 126
7.5.1 Armazenamento dos Últimos Números Gerados . . . . . p. 127
7.5.2 Armazenamento dos Multiplicadores . . . . . p. 128
7.5.3 Armazenamento do Controle do Sinal de Transporte . . . . . p. 129
7.5.4 Armazenamento do Sinal de Transporte . . . . . p. 129
7.5.5 Multiplicação de Operandos . . . . . p. 131
7.5.6 Soma de Operandos . . . . . p. 131
7.5.7 Memória de Endereços . . . . . p. 132
7.5.8 Controlador da Memória de Endereços . . . . . p. 133

7.5.9 Contadores . . . . . p. 134
7.5.10 Armazenamento de Estados . . . . . p. 135
7.5.11 Controle de Pacotes Recebidos . . . . . p. 136
7.5.12 Controle de Pacotes a Enviar . . . . . p. 137
7.5.13 Armazenamento do Sinal PAP . . . . . p. 139
7.6 Testes e Resultados . . . . . p. 140
7.7 Comentários . . . . . p. 142

8 BCERP: Bloco Básico de Configuração dos Elementos de uma Rede de Petri p. 144

Resumo . . . . . p. 144
8.1 Introdução . . . . . p. 144
8.2 Funcionamento do Bloco BCERP . . . . . p. 145
8.3 Arquitetura do Bloco BCERP . . . . . p. 148
8.3.1 Decodificador de Pacotes . . . . . p. 150
8.3.2 Análise de Disparo . . . . . p. 153
8.3.3 Resolução de Conflito . . . . . p. 155
8.3.4 Processo de Disparo . . . . . p. 158
8.3.5 Lógica de Envio de Pacotes . . . . . p. 159
8.4 Utilização de Blocos BCERPs Auxiliares . . . . . p. 161
8.5 Síntese, Simulação e Teste do Bloco BCERP . . . . . p. 164
8.6 Comentários . . . . . p. 166

9 Simulação e Teste da Arquitetura Proposta p. 167

Resumo . . . . . p. 167
9.1 Introdução . . . . . p. 167
9.2 Simulação e Teste do Sistema de Roteamento . . . . . p. 168
9.3 Exemplo de uma Rede de Petri Mapeada na Arquitetura Proposta . . . p. 169

9.3.1 Configuração do Bloco BCERP(1,1) . . . . . p. 171
9.3.2 Configuração do Bloco BCERP(1,2) . . . . . p. 172
9.3.3 Configuração do Bloco BCERP(1,3) . . . . . p. 174
9.3.4 Configuração do Bloco BCERP(2,1) . . . . . p. 175
9.3.5 Configuração do Bloco BCERP(2,3) . . . . . p. 176
9.3.6 Configuração do Bloco BCERP(3,1) . . . . . p. 178
9.3.7 Configuração do Bloco BCGN . . . . . p. 180
9.4 Simulação e Teste da Arquitetura 3x3 . . . . . p. 180
9.5 Comentários . . . . . p. 183

10 Considerações Finais p. 185
10.1 Tópicos Desenvolvidos . . . . . p. 185
10.2 Sugestões para Trabalhos Futuros . . . . . p. 186
10.3 Conclusão . . . . . p. 187

Referências p. 188

Anexo A – Uma Abordagem para Análise de Desempenho no Projeto de um Roteador p. 195
Resumo . . . . . p. 195
A.1 Introdução . . . . . p. 195
A.2 Desenvolvimento das Fórmulas de Comparação . . . . . p. 198
A.2.1 Conceitos Fundamentais . . . . . p. 198
A.2.2 Decrementador com Transporte em Cascata . . . . . p. 203
A.2.3 Decrementador com Transporte Antecipado . . . . . p. 205
A.2.4 Modificação do Decrementador com Transporte Antecipado . . . p. 211
A.2.5 Decrementador com Transporte Selecionado . . . . . p. 215
A.2.6 Quantidades de Portas e de Níveis de Lógica do Roteador . . . . p. 218

A.2.7 Fórmula de Desempenho . . . . . p. 219
A.3 Resultados . . . . . p. 221
A.4 Discussão . . . . . p. 224
A.5 Comentários . . . . . p. 226

Anexo B – Composição do CD-ROM Anexado a esta Tese p. 228

1 Introdução

Resumo

O objetivo proposto neste trabalho é o desenvolvimento de uma arquitetura multiprocessada e reconfigurável que permita a implementação física de sistemas descritos em Redes de Petri, por meio de um mapeamento tecnológico diretamente no nível comportamental, sem a necessidade de se utilizar um processo de síntese de alto nível para descrever o sistema em equações booleanas e tabelas de transição de estados, como acontece atualmente.

1.1 Contextualização

A constante evolução tecnológica tem possibilitado a implementação de funções progressivamente mais complexas e a complexidade dos sistemas digitais atuais exige que eles sejam analisados e verificados detalhadamente, no intuito de minimizar os erros de implementação. Os erros de projeto de um produto no mercado atual podem ser extremamente custosos não apenas em termos financeiros, mas também afeta a imagem do fabricante. Outrossim, é importante automatizar o processo de síntese desses sistemas para viabilizar a inserção dos produtos resultantes no mercado o mais rapidamente possível, pois o tempo de desenvolvimento é crítico para o sucesso do produto.

A evolução dos processos de síntese nas pesquisas científicas tem se realizado no sentido ascendente, ou seja, dos níveis menos abstratos em direção aos mais abstratos. Inicialmente, desenvolveu-se a síntese lógica, seguida da síntese RTL e, mais recentemente, a síntese de alto nível.

Processos de síntese lógica (STRUM; CHAU; NETO, 1999) permitem ao projetista definir o comportamento de um circuito por meio da descrição de funções booleanas, para circuitos combinatórios, ou tabelas de transição de estados, para circuitos seqüenciais.

A partir daí o processo de síntese lógica realiza tarefas de minimização e codificação de estados, fatoração de variáveis booleanas, minimização de produtos entre outras, seguidas pela tarefa de mapeamento tecnológico. No final da síntese o processo gera, baseado numa biblioteca de alocação, a descrição de uma estrutura física capaz de executar o comportamento especificado pelo projetista.

Com o transcorrer do tempo, o comportamento de um circuito passou a ser representado não apenas por funções booleanas e transições de estados, mas também por unidades funcionais simbolizadas por operações aritméticas, lógicas, relacionais e de deslocamentos. Desta forma, as bibliotecas de alocação passaram a incorporar um aglomerado de células, denominadas de macro-células, capazes de executar o conjunto das operações especificadas. As macro-células são, portanto, os módulos de *hardware* que compõem o esquema lógico de um circuito. Processos capazes de mapear as unidades funcionais de uma arquitetura no nível da lógica de transferência entre registradores (arquitetura RTL) sobre macro-células, de forma a resultar num circuito que obedeça o comportamento e as demais especificações de desempenho, caracterizam a metodologia de projeto denominada síntese da lógica de transferência entre registradores, ou simplesmente, síntese RTL (STRUM; CHAU; NETO, 1999).

O avanço das linguagens de descrição de *hardware*, como por exemplo VHDL (MAZOR; LANGSTRAAT, 1993) ou Verilog (MANO; KIME, 1999), permitiu aos projetistas a especificação de um circuito no nível de sistema ou comportamental e intensificou as atividades de pesquisa científica para a elaboração de processos automáticos de síntese neste nível de abstração, que atualmente é denominado síntese de alto nível ou síntese comportamental.

Portanto, o processo de síntese de alto nível (STRUM; CHAU; NETO, 1999) refere-se à transformação da descrição do comportamento de um circuito numa arquitetura dedicada à execução de um algoritmo, o qual foi estruturado por meio de uma linguagem de descrição de *hardware*. Durante a síntese todas as operações da descrição são mapeadas sobre algum recurso de *hardware*.

Dependendo da natureza do problema e da complexidade do projeto, podem-se adotar diferentes processos de síntese de alto nível, tais como a síntese de alto nível hierárquica e a recursiva.

Síntese de alto nível hierárquica (STRUM; CHAU; NETO, 1999) é uma metodologia de projeto que resolve um problema através da estratégia de dividir para conquistar. A descrição do problema algorítmico é feita de forma hierárquica, existindo um procedimento topo e um conjunto de sub-rotinas denominadas módulos comportamentais. O processo

hierárquico é então usado para sintetizar uma arquitetura para cada módulo comportamental, e, em seguida, uma para o procedimento topo. Cada módulo comportamental é obtido através de sessões de síntese de alto nível.

Por sua vez, a síntese de alto nível recursiva (STRUM; CHAU; NETO, 1999) refere-se ao processo de otimização de uma arquitetura modular hierárquica, por meio de transformações arquiteturais aplicadas às unidades funcionais presentes nesta arquitetura.

Além do mais, a partir de 1990, viu-se uma forte popularização de dispositivos lógicos programáveis (MANO; KIME, 1999), como CPLDs e FPGAs, permitindo a realização de sistemas com os mais variados níveis de complexidades (de algumas centenas, a dezenas de milhares de portas lógicas) em um espaço de tempo bastante curto. Particularmente, um arranjo de portas programáveis em campo (FPGA) constitui-se, basicamente, de uma estrutura lógica combinacional, *flip-flops* pré-implementados e elementos de interconexão programáveis, como chaves eletrônicas. Para o mapeamento tecnológico num FPGA, é imprescindível que o comportamento do circuito procurado seja descrito por meio de funções booleanas e equações de estado, utilizando-se algum processo de síntese durante esta etapa de refinamento.

1.2 Objetivo e Justificativas do Projeto

O objetivo proposto neste projeto de pesquisa é a definição de uma arquitetura reconfigurável e multiprocessada que possa ser utilizada para a implementação física de sistemas de controle modelados por meio de Redes de Petri. Uma Rede de Petri (WANG, 1998) (MURATA, 1989) pode ser definida como sendo uma linguagem descritiva visual utilizada para facilitar o processo de especificação formal de sistemas e por ser uma ferramenta matemática, possui uma grande quantidade de propriedades e métodos de análise que permitem ao projetista analisar e interpretar o comportamento e a estrutura do modelo especificado.

Com a implementação da arquitetura proposta sobre um FPGA, por exemplo, e o desenvolvimento futuro de uma plataforma de trabalho, o projetista poderá, utilizar a plataforma para descrever graficamente o sistema em uma Rede de Petri e após a sua descrição, a plataforma poderá realizar alguns cálculos para a verificação de erros da rede e executar alguns processos intermediários para permitir o mapeamento tecnológico na arquitetura proposta. A Rede de Petri é utilizada apenas como uma linguagem visual para a descrição do sistema de controle a ser fisicamente implementado, portanto, a arquitetura

implementa o sistema de controle, não sendo um acelerador em *hardware* para avaliar as propriedades da Rede de Petri.

Os componentes constituintes da arquitetura podem ser programados ou configurados para permitir o mapeamento de diversos sistemas modelados por Redes de Petri com diferentes estruturas e comportamentos. Vale a pena ressaltar que o FPGA é configurado apenas uma vez para implementar a arquitetura proposta. Como a arquitetura proposta também é reconfigurável, o sistema descrito em uma Rede de Petri a ser fisicamente implementado é diretamente mapeado nos componentes dessa arquitetura. Portanto, o sistema a ser implementado é configurado sobre a arquitetura proposta e não sobre o FPGA.

Atualmente, já são utilizados FPGAs para implementar fisicamente uma Rede de Petri. Porém, como um FPGA contém basicamente pequenos blocos lógicos (CLBs), tabelas de programação (LUTs) e barramentos programáveis, a Rede de Petri a ser implementada em FPGA tem que ser transformada em uma linguagem de descrição de *hardware* e na seqüência deve-se realizar a síntese de alto nível para que se possa programar as funções booleanas e tabelas de transição de estados gerados nos recursos do FPGA.

A arquitetura proposta permite um mapeamento tecnológico diretamente no nível comportamental ou sistêmico, como mostrado na figura 1.1. Desta forma, não é necessário transformar o modelo de Rede de Petri em descrições com níveis de abstração intermediários e nem realizar um custoso processo de síntese de alto nível para mapear a Rede de Petri num FPGA.

A arquitetura proposta é composta de blocos lógicos reconfiguráveis exclusivamente desenvolvidos para a implementação dos estados (lugares) e das ações (transições) de uma Rede de Petri. Assim, os lugares e as transições da Rede de Petri que descrevem o sistema a ser fisicamente implementado podem ser diretamente mapeados na arquitetura, sem haver a necessidade de se utilizar um processo de síntese de alto nível para descrever o sistema por meio de equações booleanas e tabelas de transição de estados.

Num processo de síntese de alto nível convencional, a transformação de uma especificação comportamental numa descrição funcional de uma arquitetura composta por unidades de processamento de dados e de controle é um processo longo, tendo sido decomposto numa sucessão de tarefas.

Inicialmente o projeto deve ser definido numa linguagem de alto nível, sendo em seguida transformado num formato intermediário sobre o qual atuarão as demais tarefas.

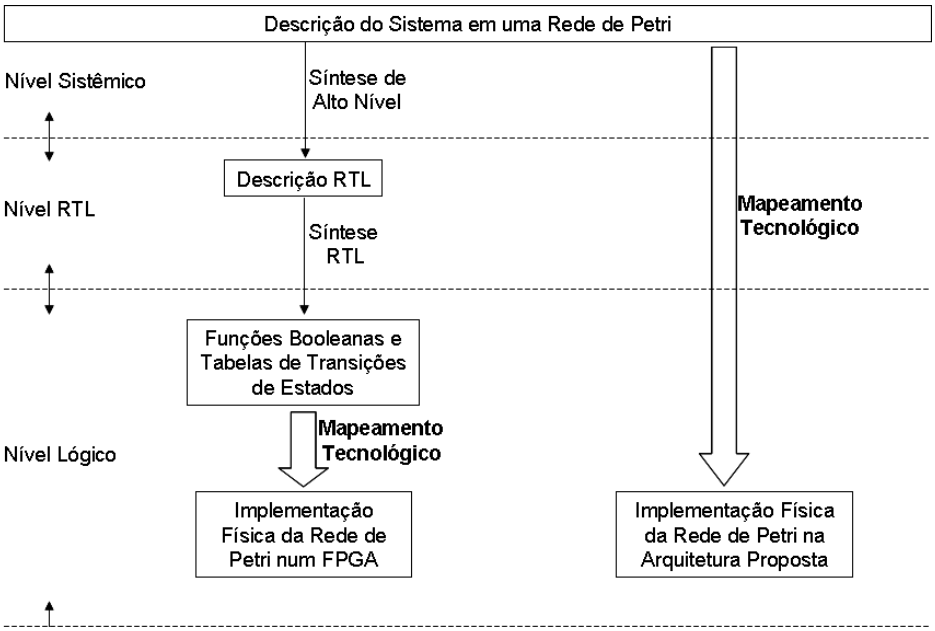


Figura 1.1: Processo de mapeamento tecnológico proposto

Em seguida vem a etapa de geração da arquitetura através das tarefas de ordenação e alocação de recursos de *hardware*. Ao longo de todo este processo podem ocorrer várias tarefas de otimização que permitem encontrar um resultado final de melhor desempenho (STRUM; CHAU; NETO, 1999).

Na descrição inicial, transforma-se a especificação do comportamento algorítmico do projeto num modelo de formato intermediário baseado em grafos. A obtenção deste formato é feita em três etapas, quais sejam: descrição da especificação do projeto numa linguagem formal, geração do grafo e otimização do grafo. Apresenta-se, a seguir, uma breve descrição destas três tarefas.

A descrição algorítmica de entrada é inicialmente submetida a uma análise léxica. Em seguida, é realizada uma análise sintática gerando, com isso, uma representação em árvore. Esta representação é então transformada em grafos que indicam as dependências entre as operações da descrição algorítmica. Dois grafos podem ser gerados para descrever o fluxo de dados e o fluxo de controle. Alguns processos de síntese combinam os dois numa única representação chamada grafo de fluxo de dados e controle (STRUM; CHAU; NETO, 1999).

A otimização da representação alcançada é realizada através de transformações de alto nível que preservam a semântica original. Tais propriedades têm por objetivo modificar o grafo inicial de forma que o grafo resultante facilite a busca de uma arquitetura otimizada, servindo de base para as tarefas de síntese de alto nível que se seguem.

Na seqüência, o processo de geração da unidade de processamento da arquitetura é acionado, podendo ser realizado através de duas tarefas: ordenação e alocação de *hardware* (STRUM; CHAU; NETO, 1999).

A ordenação consiste em determinar a seqüência de execução das operações provenientes do comportamento do projeto, explorando seu paralelismo. Por sua vez, a alocação de *hardware* consiste na seleção dos tipos e quantidades de componentes RTL (unidades funcionais, registradores e interconexão), explorando seu compartilhamento por diferentes operações. O problema de alocação pode ser dividido em três tarefas, quais sejam: seleção do tipo, que indica qual dos elementos armazenados na biblioteca será usado para executar cada operação, definição da quantidade, que determina quantos elementos de cada tipo serão necessários, e a associação entre operações e os recursos de *hardware* selecionados, que define quais dos recursos alocados executarão cada operação RTL (STRUM; CHAU; NETO, 1999).

No processo de alocação de *hardware*, técnicas baseadas em cobertura de um grafo (MCHUGH, 1990) podem ser usadas para determinar as possibilidades de compartilhamento entre operações e para associar dados a serem armazenados aos registradores. Outra técnica utilizada para resolver estes problemas é a associação bipartite com peso (MCHUGH, 1990), na qual são montadas duas listas, uma contendo as operações e a outra contendo as unidades funcionais, no caso de alocação de unidades funcionais, ou então, no processo de alocação de elementos de armazenamento, uma contendo os dados e a outra os registradores.

As tarefas de ordenação e alocação de *hardware* não são independentes, mas são executadas separadamente devido à complexidade computacional para realizá-las simultaneamente.

Após gerar a unidade de processamento de dados, o processo de síntese de alto nível faz a descrição funcional da unidade de controle representada por uma máquina de estados finitos.

Antes do encerramento do processo de síntese de alto nível, algumas otimizações podem ser realizadas para melhorar a qualidade da arquitetura gerada. Exemplos de otimizações são encadeamento de operações, operações multiciclo, operações segmentadas e exploração das propriedades comutativa e distributiva.

A utilização de processos de síntese mais complexos que façam uso de hierarquia e recursividade, implica na realização de novas tarefas, além das mencionadas até o momento.

No processo de síntese de alto nível hierárquica, por exemplo, são agregadas a tarefa de partição da descrição inicial, que consiste na decomposição da descrição comportamental do circuito topo em blocos de operações, e a tarefa de encapsulamento dos módulos arquiteturais resultantes para sua reutilização mais adiante. Devido à complexidade envolvida nestes tipos de processos, estruturas, como por exemplo árvores, podem ser utilizadas juntamente com grafos para facilitarem a execução de determinadas tarefas.

Após terem sido geradas as especificações para as unidades funcionais, elementos de armazenamento, rede de comunicações e unidade de controle, pode-se então iniciar o mapeamento tecnológico responsável pela geração da arquitetura.

Comparativamente com o processo de síntese de alto nível explicado, o processo de mapeamento que está sendo proposto pode reduzir significativamente a quantidade de tarefas a serem executadas na obtenção de uma arquitetura configurada para executar o comportamento algorítmico de uma Rede de Petri. Além do mais, a redução no número de tarefas a serem executadas para a realização do mapeamento tecnológico, evita a ocorrência de alguns problemas algorítmicos enfrentados durante um processo de síntese de alto nível, como a dificuldade de paralelizar operações executadas em diferentes módulos arquiteturais durante a síntese hierárquica do circuito e a possibilidade das unidades de controle e memória se tornarem desnecessariamente grandes, o que pode fazer com que a execução do processo de síntese de alto nível leve um tempo excessivamente longo, ou mesmo torne inviável a obtenção de uma descrição RTL a partir do comportamento algorítmico do sistema (STRUM; CHAU; NETO, 1999).

1.3 Ferramentas Utilizadas no Desenvolvimento do Projeto

A descrição de todos os componentes da arquitetura proposta foi realizada em VHDL. As etapas de simulação da descrição VHDL e de implementação em FPGAs foram realizadas por meio do *software* Quartus II da Altera (ALTERA CORPORATION, 2004). A placa educacional UP2 distribuída pela Altera foi utilizada para a implementação em FPGA dos componentes da arquitetura. Outras ferramentas foram utilizadas no trabalho realizado, como o *software* MATLAB (HANSELMAN; LITTLEFIELD, 1997), a linguagem de programação C (ARAUJO, 2004) (KERNIGHAN; RITCHIE; RITCHIE, 1988) e o programa Dev-C++, para a compilação do código C gerado.

1.4 Organização do Texto

O texto desta tese de doutorado possui a seguinte disposição:

- **Capítulo 2** – Implementações Físicas Existentes de Sistemas Modelados em Redes de Petri: Descrição do estado-da-arte sobre o processo de implementação em *hardware* de sistemas descritos em Redes de Petri.
- **Capítulo 3** – Redes-em-Chip: Levantamento bibliográfico sobre uma nova forma de comunicação de sistemas-em-*chip* (SoCs).
- **Capítulo 4** – Geração de Números Pseudo–Aleatórios: Descrição das características e das fórmulas matemáticas de diversos tipos de geradores lineares propostos na literatura.
- **Capítulo 5** – Arquitetura Proposta para Implementar Fisicamente Sistemas Modelados em Redes de Petri: Definição dos elementos constituintes da arquitetura proposta.
- **Capítulo 6** – Roteador: Responsável pelo Sistema de Comunicação da Arquitetura Proposta: Especificação do sistema de comunicação projetado para a arquitetura proposta.
- **Capítulo 7** – BCGN: Bloco de Configuração do Gerador de Números Pseudo-Aleatórios: Desenvolvimento do bloco de configuração do gerador de números pseudo-aleatórios da arquitetura proposta.
- **Capítulo 8** – BCERP: Bloco Básico de Configuração dos Elementos de uma Rede de Petri: Desenvolvimento do bloco responsável pela implementação dos elementos de uma Rede de Petri.
- **Capítulo 9** – Simulação e Teste da Arquitetura Proposta: Descrição dos processos de simulação e de teste realizados na arquitetura proposta.
- **Capítulo 10** – Considerações Finais: Comentários finais sobre a arquitetura proposta e sugestões para trabalhos futuros.
- **Referências Bibliográficas**
- **Anexo A** – Uma Abordagem para Análise de Desempenho no Projeto de um Roteador: Comparação do desempenho do roteador especificado com

as principais e mais conceituadas técnicas de obtenção do sinal de transporte de circuitos decrementadores.

- **Anexo B** – Composição do CD-ROM Anexado a esta Tese: Descrição do conteúdo do CD-ROM anexado e comentários sobre o *software* Quartus II utilizado no projeto da arquitetura proposta.

2 *Implementações Físicas Existentes de Sistemas Modelados em Redes de Petri*

Resumo

Atualmente, existem arquiteturas que podem ser utilizadas para implementar sistemas descritos em Redes de Petri, como é o caso da arquitetura Achilles (MORRIS; BUNDELL; THAM, 2000), que realiza a conversão da Rede de Petri para uma descrição em linguagem VHDL utilizando uma biblioteca de códigos que descrevem os lugares e as transições da rede. Em (ANZAI et al., 1993) descreve-se um sistema multiprocessado que pode ser programado por meio de um modelo em Rede de Petri. A Rede de Petri é armazenada num controlador em forma de tabelas que descrevem a estrutura e a dinâmica da rede. Como exemplo prático de implementação de uma Rede de Petri, utilizou-se, neste trabalho, o *software* QuartusII para descrever em VHDL os lugares e as transições do modelo de rede sintetizado, o qual foi mapeado em um FPGA. Um algoritmo de seleção foi utilizado para associar uma prioridade de disparo a cada transição da rede.

2.1 Introdução

Uma metodologia de projeto deve prover ferramentas para a modelagem de sistemas num nível elevado de abstração, além de possibilitar a verificação, a validação e a implementação de sistemas cada vez mais complexos.

A Rede de Petri tem se mostrado uma linguagem formal poderosa para especificar e modelar o comportamento algorítmico de sistemas paralelos síncronos e assíncronos num nível de abstração bem elevado. Além disso, a Rede de Petri possui diversos métodos de análise que permitem verificar a ocorrência de erros antes de se iniciar a fase de implementação.

Comentam-se, a seguir, as formas de implementação física de sistemas modelados em Redes de Petri, posteriormente, expõem-se algumas arquiteturas, extraídas da literatura científica, que implementam modelos de Redes de Petri em *hardware*, como um sistema multiprocessado controlado por Redes de Petri e a arquitetura Achilles utilizada para implementar modelos de Redes de Petri lugar/transição. Um exemplo prático de síntese de uma Rede de Petri em FPGA é apresentado. Por fim, comentam-se sobre a implementação de um seletor pseudo-aleatório de transição e sobre a utilização de recursos num FPGA.

2.2 Formas de Implementação Física de Redes de Petri

Os métodos de implementação de Redes de Petri podem ser classificados em duas categorias: *software* e *hardware*. Uma implementação em *software* é a simulação de Redes de Petri usando-se sistemas computacionais que, de modo geral, consomem grande tempo de processamento (CHANG; KWON; PARK, 1996).

A implementação em *hardware* pode ser subdividida em (GOMES, 1999) realizações indiretas e diretas.

As realizações indiretas têm por base a tradução da Rede de Petri numa representação intermediária, como por exemplo um grafo de estados, o qual será posteriormente sintetizado em equações lógicas (GOMES, 1999).

Ao caracterizar uma Rede de Petri limitada com uma determinada marcação inicial através de um grafo de estados finito, encontra-se o que pode ser chamado de uma máquina de estados, em que os estados são associados às várias marcações da Rede de Petri e as condições de transição entre estados estão associadas aos disparos das transições na rede (GOMES, 1999).

No exemplo de Rede de Petri da figura 2.1(a), o grafo de estados finito apresentado na figura 2.1(b) pode ser caracterizado como uma máquina de estados (figura 2.1(c)), o que permite a aplicação de métodos de tradução em equações lógicas para a implementação física da rede (GOMES, 1999).

Uma forma de implementação física do grafo de estados pode ser realizada utilizando-se um conjunto de elementos de memória, organizados num vetor, para o armazenamento do estado atual, e uma lógica booleana para determinar a evolução do sistema. O vetor de elementos de memória representa os diferentes nós do grafo. O elemento de

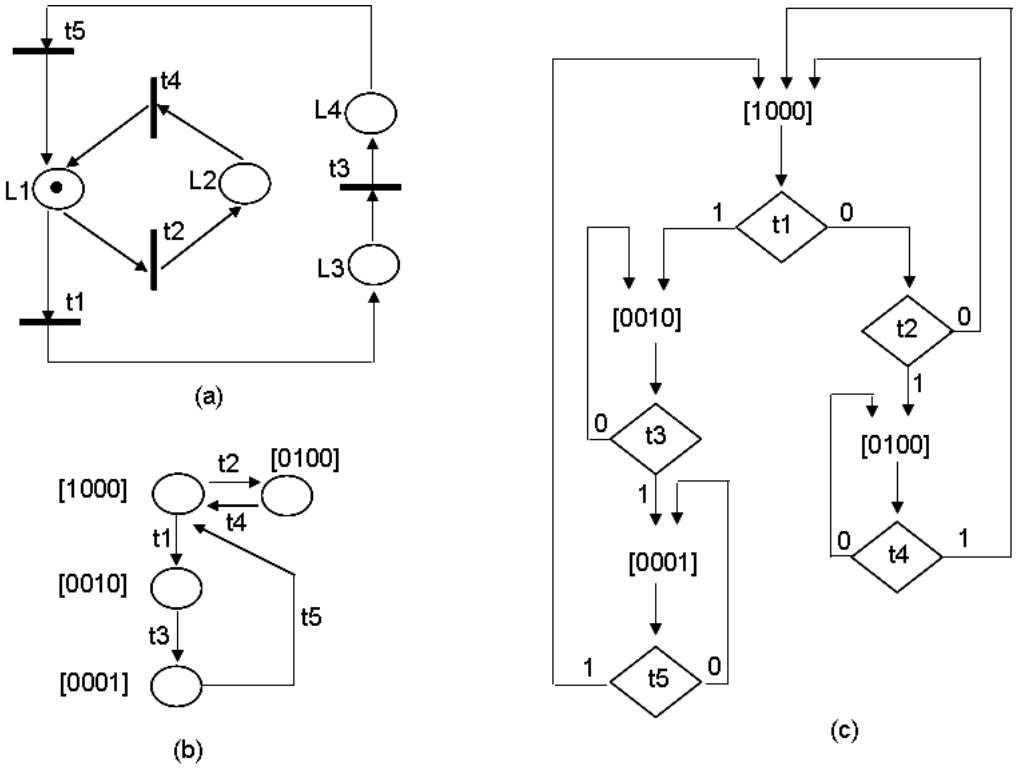


Figura 2.1: Rede de Petri e a sua correspondente máquina de estados
Adaptada de (GOMES, 1999)

memória associado ao estado ativo armazenará o estado lógico "1", enquanto todos os outros elementos conterão "0"(GOMES, 1999). Assim, uma mudança de estado se traduz na desativação do elemento de memória representando o estado anterior e na ativação do elemento de memória associado ao estado seguinte, de acordo com a avaliação das condições de transição (GOMES, 1999).

Considerando que o elemento de memória dispõe de entradas de ativação e desativação (*set* e *reset*, respectivamente), tudo se resume a construir as expressões lógicas associadas (GOMES, 1999).

Uma desvantagem desse tipo de realização indireta é a grande quantidade de memória necessária para a implementação, visto que o grafo de estados associado a uma Rede de Petri pode se tornar bastante extenso.

Por sua vez, as realizações diretas de Redes de Petri em *hardware* baseiam-se numa tradução, nos quais os elementos da rede (lugares e transições) são implementados por meio de componentes digitais pré-estabelecidos.

Pode-se utilizar um programa que realiza a conversão de uma descrição do sistema em Redes de Petri para uma descrição que viabilize uma implementação em *hardware*. As

linguagens mais comuns para realizar este refinamento a um nível RTL e de portas lógicas são Verilog e VHDL. Deste modo, o código VHDL ou Verilog gerado é utilizado como entrada para um processo de síntese, o qual transformará a especificação do sistema num conjunto de portas lógicas devidamente interconectados (ROKYTA; FENGLER; HUMMEL, 2000).

Para o processo de mapeamento tecnológico, tem-se utilizado FPGAs. A arquitetura de um FPGA possui milhares de células lógicas programáveis disponíveis e a baixo custo. Estas células proveêm circuitos lógicos suficientes para a implementação de sistemas digitais complexos em uma única pastilha. Além disso, podem-se utilizar ferramentas de programação de FPGAs que fazem uso de linguagens de alto nível para a descrição do *hardware* a ser implementado.

Como conseqüência, os FPGAs são excelentes dispositivos para a implementação de um sistema computacional. Tal sistema, quando modelado através de uma linguagem de descrição de *hardware*, possui um rápido ciclo de desenvolvimento, pois é automaticamente sintetizado através do processo de programação de FPGAs.

Anteriormente aos FPGAs usavam-se os dispositivos conhecidos como PLDs, PALs e PLAs que possuem somente duas matrizes, uma com portas AND e a outra com portas OR (SOTO; PEREIRA, 2001). Os FPGAs são diferentes porque são compostos de blocos lógicos configuráveis (CLB) que trabalham de forma semelhante a um sistema seqüencial. Os CLBs são compostos de uma memória RAM e de uma sistema combinatorial. A memória RAM é programada pelo sistema combinatorial que define o comportamento do sistema seqüencial. Com isso, uma grande vantagem da tecnologia FPGA é o alto grau de flexibilidade no processo de mapeamento de uma Rede de Petri, permitindo uma boa alocação dos elementos da rede e otimização das interconexões existentes (SOTO; PEREIRA, 2001).

2.3 Sistema Multiprocessado Controlado por Modelos em Redes de Petri

Neste sistema (ANZAI et al., 1993) (KAMAKURA et al., 1997) implementa-se um controlador programado por meio de uma descrição em Rede de Petri. O controlador é utilizado pelo sistema para a ativação e desativação de processos paralelos.

Na figura 2.2 apresenta-se a arquitetura do sistema multiprocessado e a implementação de uma Rede de Petri (ANZAI et al., 1993) no controlador. Na figura 2.2(a) apresenta-se

um exemplo de uma Rede de Petri que pode ser usada para programar o controlador. A cada lugar da Rede de Petri atribui-se um determinado trabalho, o qual deve ser realizado por meio de um programa executado por processadores elementares.

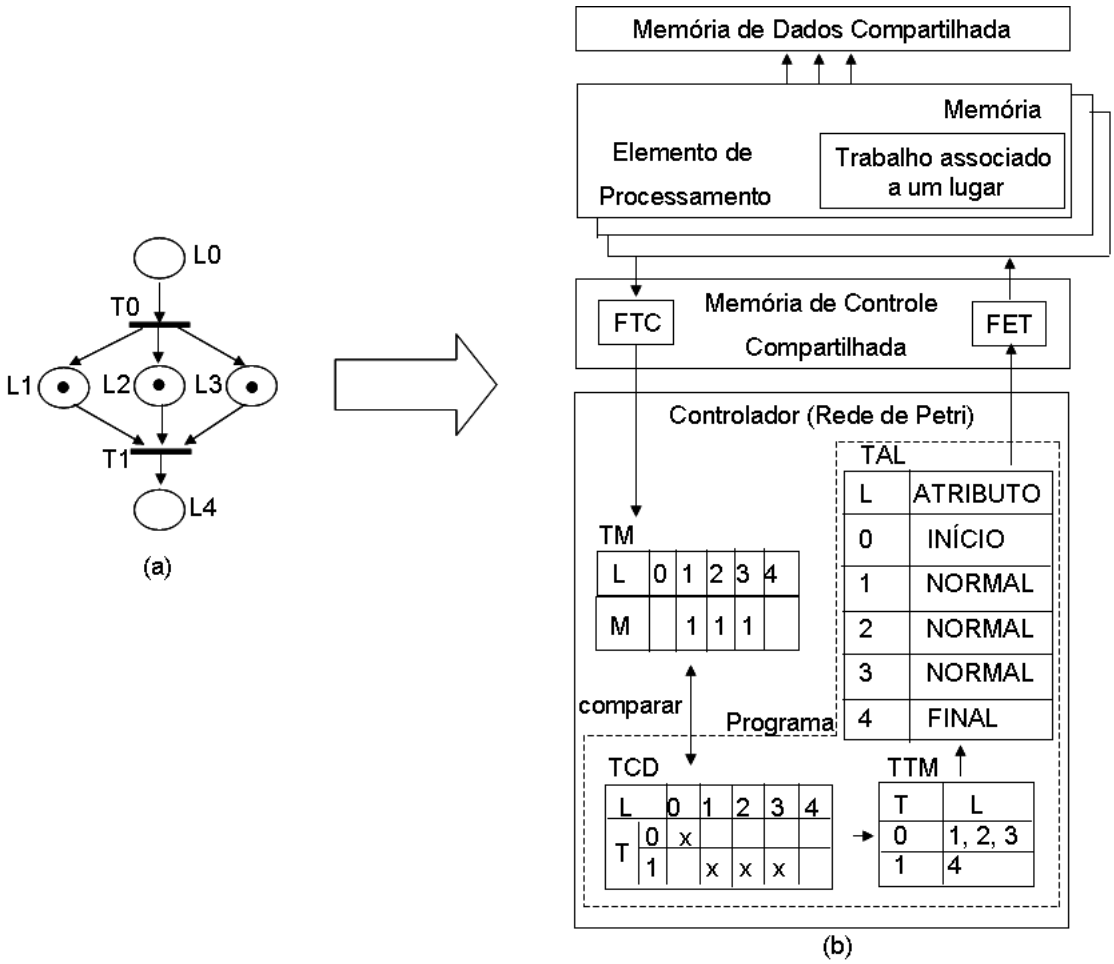


Figura 2.2: (a) Rede de Petri, (b) Controlador configurado por meio de uma Rede de Petri

Adaptada de (ANZAI et al., 1993)

Na figura 2.2(b) apresenta-se a configuração do sistema multiprocessado definido pela Rede de Petri da figura 2.2(a). Basicamente, essa arquitetura consiste de um controlador, um conjunto de processadores elementares, uma memória de controle compartilhada para permitir a comunicação entre o controlador e os processadores elementares, e uma memória de dados compartilhada para a troca de informações entre os processadores (ANZAI et al., 1993).

A descrição em Rede de Petri é armazenada no controlador por meio das **t**abelas de **c**ondição de **d**isparo (`TCD`), de **t**ransferência de **m**arca (`TTM`) e de **a**tributos de **l**ugares (`TAL`). A marcação da Rede de Petri é armazenada na tabela de marcação (`TM`). Os programas executados são armazenados nos processadores elementares. A tabela `TCD` indica

as transições de saída de cada lugar. A tabela `TTM` indica os lugares de saída de cada transição. A tabela `TAL` indica os atributos de cada lugar e a tabela `TM` indica a distribuição de marcas nos lugares da rede.

O controlador compara a tabela `TM` com a tabela `TCD` na tentativa de encontrar uma transição habilitada. Se encontrar uma, o controlador procura os lugares de saída daquela transição na tabela `TTM`. Se o atributo do lugar indexado for `normal`, o controlador escreve um número (identificador) correspondente a esse lugar na **f**ila de **e**xecução de **t**arefas `FET`. Se o lugar possuir o atributo `final`, o controlador encerra a execução dos processos.

Por sua vez, cada processador retira um número da fila `FET` e executa um programa correspondente àquela numeração (lugar). Ao término da execução da tarefa, o processador elementar escreve na **f**ila de **t**arefas **c**oncluídas `FTC`, o número cujo programa associado foi executado.

Assim, o controlador lê a numeração em `FTC`, reconhece a conclusão de uma tarefa e então cria uma marca no lugar correspondente na tabela `TM`. O ciclo é repetido novamente, ou seja, o controlador compara `TM` com `TCD` e dá seqüência ao procedimento explicado anteriormente.

2.4 Arquitetura Achilles para Implementar Modelos Descritos em Redes de Petri

Na universidade *Western Australia*, Nedlands, foi desenvolvido um procedimento para gerar códigos VHDL de modelos descritos em Redes de Petri lugar/transição. Há códigos VHDL pré-definidos que implementam os lugares e transições de uma Rede de Petri e um código que implementa o seletor de transição, utilizado para definir as transições que devem ser disparadas. Uma vez gerado o código VHDL do modelo de Rede de Petri, este é mapeado em um ou mais FPGA's, que passam a executar a Rede de Petri especificada. Este sistema foi denominado Achilles (MORRIS; BUNDELL; THAM, 2000) e pode ser mapeado em vários FPGAs interligados por meio de placas eletrônicas projetadas para realizar a interconexão entre os FPGAs. A arquitetura desenvolvida é particularmente interessante para a implementação de centenas de lugares e transições devido às suas flexibilidade e escalabilidade.

O Achilles possui uma arquitetura síncrona, onde um único *clock* distribui a sincronização para cada placa via um barramento de *clock* dedicado.

Os FPGAs do Achilles são conectados a um computador usando um adaptador de interface, o qual controla as transações no processo de troca de informação entre os FPGAs e o computador, além de executar algum pré-processamento e pós-processamento úteis na conversão de informação entre os FPGAs e o computador.

A função peso e a marcação inicial da Rede de Petri são transportadas por meio de um barramento serial. O mesmo barramento também pode ser utilizado para ler a marcação atual da rede para o computador.

Com a implementação em *hardware* de uma Rede de Petri, o Achilles pode realizar muitas operações em paralelo. Em particular, a avaliação de todas as transições para determinar quais poderão disparar é realizada simultaneamente para toda a rede.

No mesmo momento em que se determina a possibilidade de disparo de todas as transições, novas marcas nos lugares de saída são determinadas e ficam prontamente disponíveis para o armazenamento quando ocorrer a escolha da transição a ser disparada.

2.5 Exemplo Prático: Síntese de uma Rede de Petri num FPGA

Baseando-se nos conceitos introduzidos na proposta da arquitetura Achilles (MORRIS; BUNDELL; THAM, 2000), foram desenvolvidos, neste trabalho, alguns blocos lógicos no *software* QuartusII (ALTERA CORPORATION, 2004) com intuito de sintetizar uma Rede de Petri num FPGA. Para este exemplo, modelou-se a Rede de Petri lugar/transição mostrada na figura 2.3.

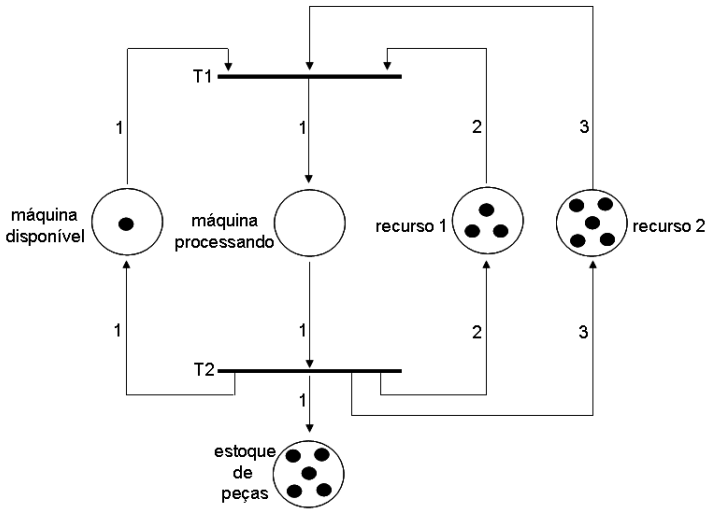


Figura 2.3: Rede de Petri a ser sintetizada

Nesta arquitetura, definiram-se os blocos lógicos correspondentes aos elementos da Rede de Petri (lugares e transições) e o fluxo das marcas permitido pela estrutura da rede. Na figura 2.4, apresenta-se a descrição em *hardware* de uma transição. O bloco lógico que define o funcionamento da transição possui somadores, subtratores e comparadores que indicam a possibilidade de disparo da transição e calcula os valores das marcas nos lugares de entrada e saída se, por determinação, esta transição vier a disparar.

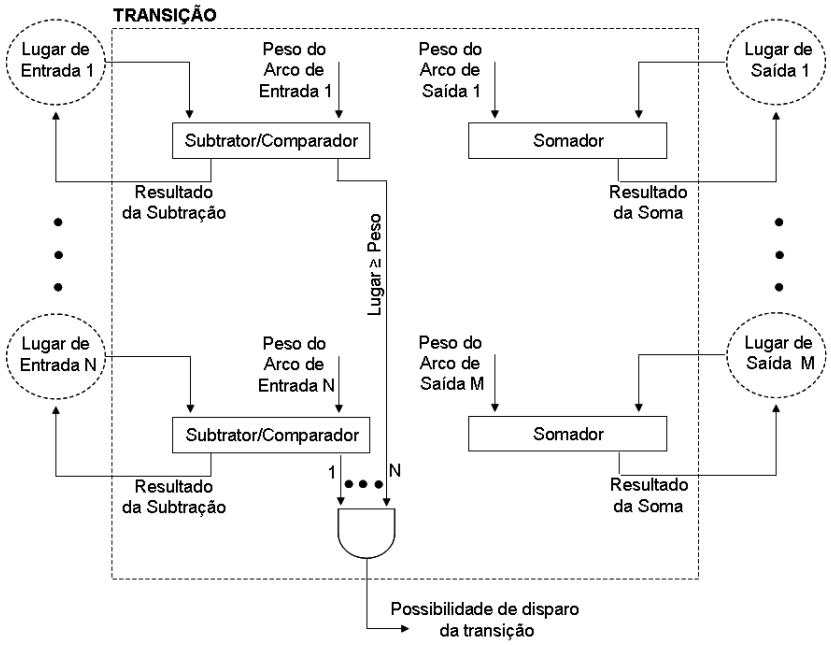


Figura 2.4: Módulo da transição
Adaptada de (MORRIS; BUNDELL; THAM, 2000)

O bloco lógico correspondente ao comportamento do lugar é composto por um registrador para armazenar a quantidade de marcas disponíveis e um multiplexador para conectar todas as transições relacionadas com este lugar.

Os componentes básicos de processamento (somadores, subtratores, comparadores, registradores e os multiplexadores) que integram os blocos lógicos do lugar e da transição foram descritos em VHDL.

Os sinais de seleção do multiplexador e o sinal `habilitação` do registrador são disponibilizados por uma unidade de controle, a qual realiza a dinâmica da rede. Esta unidade recebe como entrada os sinais de possibilidade de disparo de todas as transições da rede e, dentre as transições passíveis de disparo, escolhe uma para disparar através dos sinais de controle. Após a ocorrência do `clock`, a transição escolhida é efetivamente disparada. Esta unidade foi desenvolvida utilizando a linguagem VHDL.

A arquitetura projetada neste exemplo possui um alto grau de paralelismo, ou seja, a

avaliação das transições para um possível disparo e o cálculo das marcas de entrada e saída da transição são realizados simultaneamente. O algoritmo de seleção de uma transição, neste exemplo, associa uma prioridade de disparo para cada transição, assim, a transição t_1 terá maior prioridade de disparo do que a transição t_2.

Na arquitetura Achilles, o processo de seleção da transição a ser disparada é realizado por um registrador de deslocamento cíclico, como mostrado na figura 2.5, onde, uma transição habilitada pode levar até p ciclos de relógio para ser disparada.

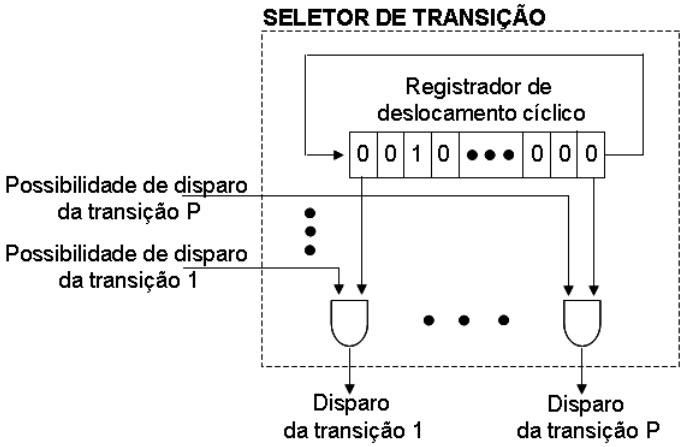


Figura 2.5: Seletor de transição do Achilles
Adaptada de (MORRIS; BUNDELL; THAM, 2000)

Utilizando o *software* QuartusII descreveu-se, neste trabalho, a estrutura da Rede de Petri da figura 2.3 juntamente com o bloco de controle, responsável pela dinâmica da rede. A estrutura da rede e o bloco de controle foram compilados e mapeados no FPGA EPF10K20. Uma vez compilado e mapeado, o sistema pôde ser testado com o auxílio do editor de ondas do QuartusII.

Na figura 2.6, apresentam-se os resultados obtidos no processo de simulação da rede implementada. Na primeira subida do `clock`, quando o sinal `inicio` estava no nível lógico "1", o sistema armazenou a marcação inicial nos registradores correspondentes aos lugares da rede. Na segunda subida do `clock`, a transição t_1 disparou e a marcação da rede passou a ser [0 1 5 1 2], onde a primeira posição do vetor indica o lugar *máquina disponível*, a segunda posição *máquina processando*, a terceira *estoque de peças*, a quarta *recurso 1* e a quinta e última *recurso 2*. Na terceira subida do `clock`, a transição t_2 disparou e a marcação passou a ser [1 0 6 3 5]. Após os disparos de t_1 e t_2 a marcação passou a ser [1 0 7 3 5].

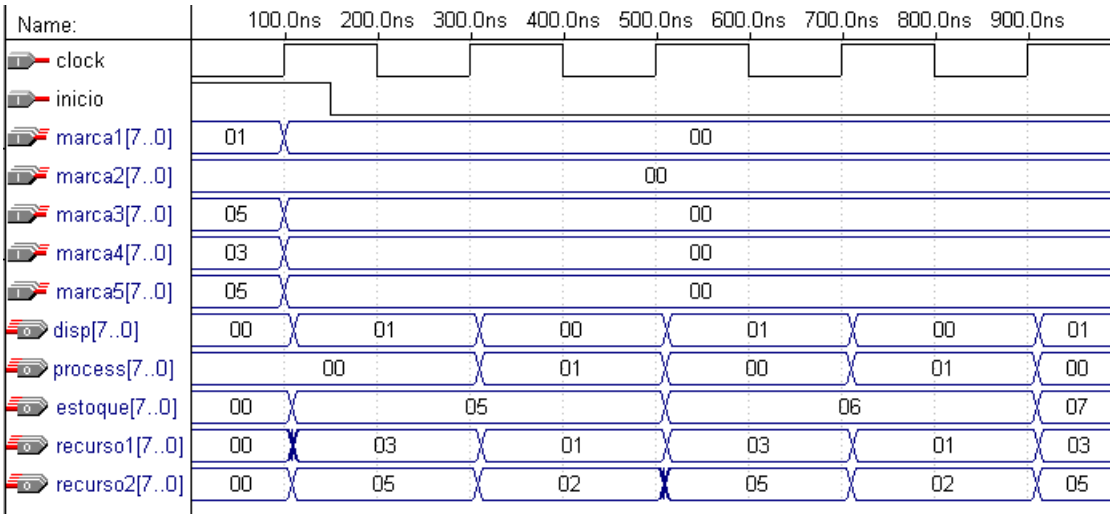


Figura 2.6: Simulação da Rede de Petri implementada

2.6 O Seletor de Transição

Uma desvantagem de se utilizar o seletor de transição mostrado na figura 2.5 é a quantidade de ciclos de relógio que pode ser necessário para encontrar uma transição habilitada. Imagine-se que, na figura 2.5, a única transição habilitada seja a transição p, mas o registrador de deslocamento cíclico tenha o valor lógico "1"apenas no primeiro bit. Assim, para disparar a transição p, o seletor de transição deverá utilizar p ciclos de relógio até que o nível lógico alto "1"seja levado a última posição do registrador de deslocamento, o que só então, permitiria o disparo da transição.

Em (KUBÁTOVÁ, 2003b) (KUBÁTOVÁ, 2003a) foi proposta uma outra implementação em *hardware* de um seletor. Nesta proposta o seletor é capaz de escolher aleatoriamente uma transição a ser disparada.

Os blocos básicos definidos em (KUBÁTOVÁ, 2001) e (KUBÁTOVÁ, 2003a) são:

- **lugar:** este bloco, figura 2.7, é responsável pela implementação de um lugar da Rede de Petri. O barramento `Saída` indica os arcos que ligam os lugares às transições. O barramento `Incrementa` indica os arcos que ligam as transições aos lugares, e o barramento `Decrementa` sinaliza que uma marca deve ser retirada de um determinado lugar. Na figura 2.7, o número de marcas é armazenado no contador. O sinal `Saída` terá valor lógico "1"se o conteúdo do contador for maior ou igual a 1, ou seja, se houver marcas no respectivo lugar.

- **transição:** este bloco, por meio de operações booleanas AND e OR, como mostrado no exemplo da figura 2.8, realiza a remoção e a criação de marcas. As conexões entre

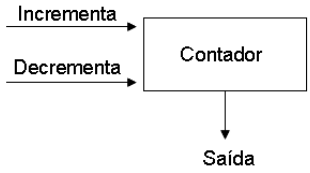


Figura 2.7: Lógica do lugar
Adaptada de (KUBÁTOVÁ, 2003a)

os lugares e as transições dependem da estrutura da Rede de Petri e são gerados de acordo com a matriz de incidência.

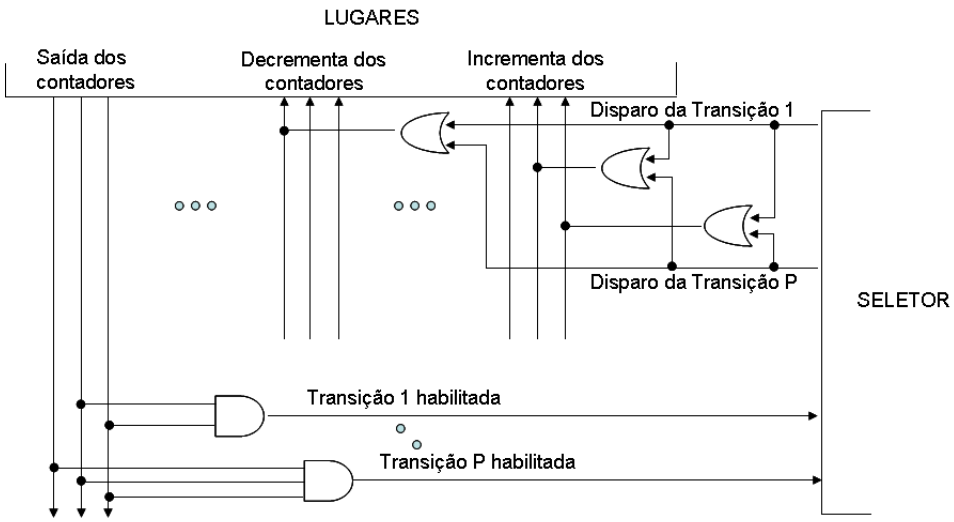


Figura 2.8: Lógica da transição
Adaptada de (KUBÁTOVÁ, 2003a)

- **seletor pseudo-aleatório:** este bloco escolhe uma transição para ser disparada entre as transições habilitadas. O bloco possui um gerador linear de números pseudo-aleatórios do tipo *linear feedback shift register*, como esquematizado na figura 2.9. Como entrada, o seletor recebe um vetor contendo todas as transições habilitadas.

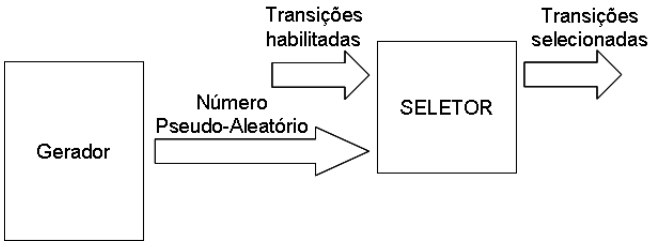


Figura 2.9: Diagrama de blocos do seletor
Adaptada de(KUBÁTOVÁ, 2003a)

2.7 Utilização de Recursos num FPGA

É necessário levar em consideração os seguintes aspectos na implementação de um sistema seqüencial em um FPGA (SOTO; PEREIRA, 2001):

- O sistema deve ser dividido em subsistemas menores para a alocação em cada CLB de um FPGA;
- Usualmente, CLBs possuem somente quatro ou cinco entradas e uma ou duas saídas, forçando uma segmentação grande do sistema; e
- CLBs são interconectados por barramentos. Esses barramentos são configurados com lógicas baseadas em matrizes, as quais possuem uma capacidade limitada. Assim, o posicionamento dos subsistemas no FPGA pode interferir na quantidade de lógica utilizada para a configuração dos barramentos, ou seja, subsistemas com forte dependência devem ser alocados próximos uns dos outros.

Estes aspectos devem ser levados em consideração quando se pretende implementar Redes de Petri complexas, devido à grande quantidade de células lógicas necessárias para o processo de síntese em FPGAs. O objetivo é encontrar um método capaz de integrar o maior número possível dos elementos de uma Rede de Petri em FPGA. Em (SOTO; PEREIRA, 2005) e (SOTO; PEREIRA, 2001) foi proposta uma solução, que consiste em implementar o sistema usando blocos especiais compostos de um lugar e uma transição, como mostrado no exemplo da figura 2.10.

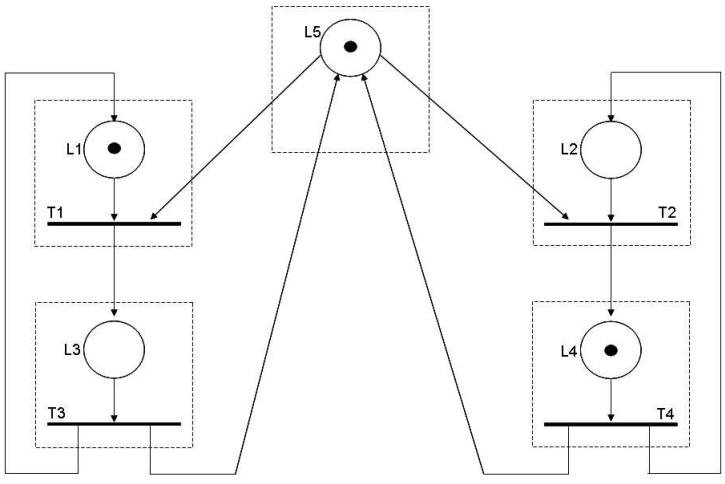
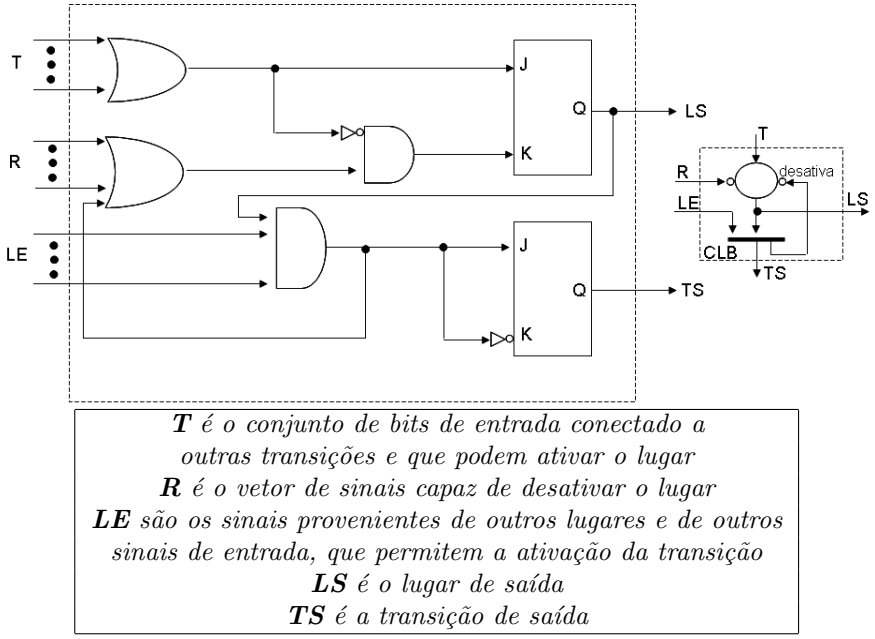


Figura 2.10: Exemplo de Rede de Petri particionada em blocos de um lugar e uma transição

Adaptada de (SOTO; PEREIRA, 2005)

Nesta implementação, todo CLB é composto de um lugar conectado a uma transição. O lugar pode ser ativado, isto é, permitir o armazenamento de uma marca, por meio de sinais de entrada provenientes de outras transições, e desativado por meio de entradas de *reset* ou da própria transição integrante do CLB. A transição será ativada, ou seja, disparada, quando o lugar precedente estiver ativo e as entradas da transição possuirem valores apropriados. Todos os blocos possuem apenas duas saídas, um bit de estado correspondente ao lugar e um bit para a transição. Na figura 2.11 apresenta-se o esquema lógico dessa forma de implementação de Redes de Petri elementares em *hardware*.



***T** é o conjunto de bits de entrada conectado a outras transições e que podem ativar o lugar*
***R** é o vetor de sinais capaz de desativar o lugar*
***LE** são os sinais provenientes de outros lugares e de outros sinais de entrada, que permitem a ativação da transição*
***LS** é o lugar de saída*
***TS** é a transição de saída*

Figura 2.11: Descrição dos blocos de implementação
Adaptada de (SOTO; PEREIRA, 2005)

São utilizados dois *flip-flops*, um para armazenar a presença ou ausência de uma marca no lugar e outro utilizado para indicar o disparo da transição.O lugar e a transição são interconectados por dois sinais. Tais sinais nem sempre precisam ser conectados a blocos exteriores. Com isso, pode-se reduzir a quantidade de lógica de conexão em um FPGA.

Com a utilização dessa metodologia, podem-se utilizar os blocos lógicos descritos para a implementação de lugares e de transições da Rede de Petri, como um conjunto de objetos parametrizados em VHDL. Assim, a implementação da Rede de Petri em FPGA refere-se à interconexão dos modelos obtidos em VHDL.

Como exemplo de interconexão (SOTO; PEREIRA, 2001), apresenta-se, na figura 2.12, a implementação da Rede de Petri da figura 2.10, onde cada bloco lógico pode ser mapeado em um CLB de um FPGA.

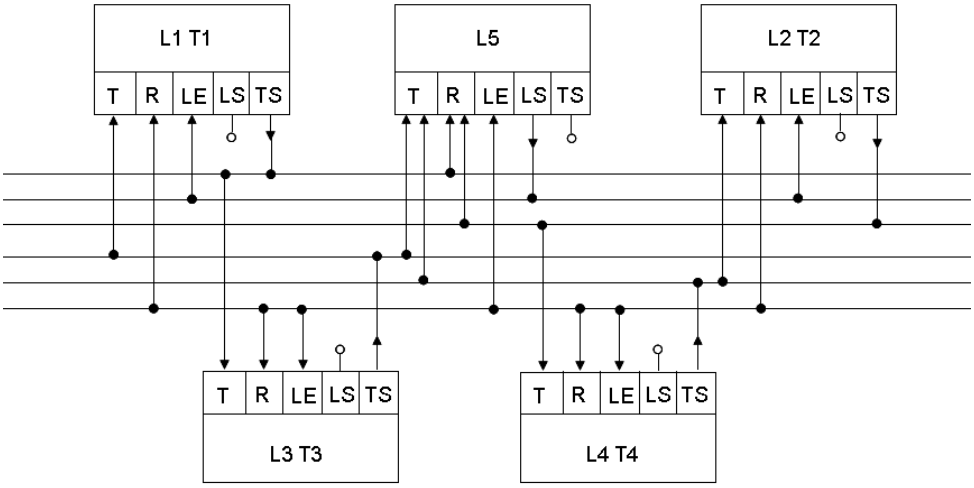


Figura 2.12: Esquema de conexão de uma Rede de Petri
Adaptada de (SOTO; PEREIRA, 2005)

2.8 Outras Formas de Implementação

Para obter um projeto compacto, não se pode associar para cada lugar da Rede de Petri um bit (*flip-flop*), visto que nem sempre todas as possíveis combinações de bits serão utilizadas no sistema (SOTO; PEREIRA, 2005). Em muitos casos, se um lugar é ativado implica que um conjunto de lugares será desativado. A codificação dos lugares utilizando um número reduzido de bits pode ser uma boa solução para a redução do número de CLBs (SOTO; PEREIRA, 2005). Por exemplo, se uma Rede de Petri com seis lugares não tiver mais do que um lugar ativo, então serão necessários apenas três bits para codificar os estados possíveis da rede.

Existem esforços no sentido de se implementar a estrutura da Rede de Petri, como descrito nas seções anteriores, para a realização de análises da rede. Assim, a implementação pode ser considerada como um acelerador de *hardware* para se conseguir, em menor tempo, uma análise sobre a Rede de Petri especificada. Em (CSERTÁN et al., 1997) descreve-se uma implementação de Redes de Petri em *hardware* para a geração de grafos de estados. Em (BUNDELL, 1997) propôs-se uma arquitetura na qual um co-processador integrado com um ambiente de trabalho é utilizado para acelerar a execução de simulações de Redes de Petri por meio de um mapeamento da Rede de Petri em blocos lógicos pré-definidos, como explicado nas seções anteriores.

2.9 Comentários

Este capítulo apresentou algumas técnicas de implementação física de um sistema descrito em Redes de Petri, como o controlador definido por meio de tabelas que armazenam a estrutura da Rede de Petri bem como a arquitetura Achilles, capaz de implementar modelos de Redes de Petri com grandes quantidades de elementos (transições e lugares) através de uma estrutura que permite adicionar vários FPGAs em pilhas. Para se conseguir incluir um maior número de lugares e transições de uma Rede de Petri num FPGA, blocos lógicos contendo cada um deles apenas um lugar e uma transição podem ser usados. Como exemplo prático de síntese de uma Rede de Petri em um FPGA foi utilizado, neste trabalho, o *software* Quartus II para sintetizar os blocos lógicos de lugares, transições e da dinâmica da rede. O processo de simulação mostrou que o circuito implementado se comportou de acordo com a Rede de Petri inicialmente especificada.

No próximo capítulo apresenta-se uma introdução às redes-em-*chip* visando fornecer alguns conceitos utilizados no desenvolvimento do projeto que está sendo proposto.

3 *Redes-em-Chip*

Resumo

Rede-em-*chip* é uma forma de comunicação emergente que possibilita o projeto de sistemas escaláveis e reutilizáveis, sendo uma alternativa às arquiteturas que vêm sendo utilizadas nos *Systems-on-Chips* atuais, como barramentos e canais dedicados. A rede-em-*chip* é baseada nos fundamentos já consolidados das redes de computadores e pode ser definida como um conjunto de roteadores interligados por canais ponto-a-ponto cujo modelo de comunicação é o de troca de mensagens, sendo que a comunicação entre unidades de processamento, as quais são acopladas aos roteadores, é feita pelo envio e recebimento de mensagens de requisição e de resposta.

3.1 Introdução

Devido aos avanços no processo de fabricação de circuitos integrados, os projetistas passaram a incorporar dezenas de primitivas num único *chip*. Primitivas estas, que deixaram de ser apenas um conjunto limitado de portas lógicas e passaram a ser unidades complexas como por exemplo, processadores específicos ou de propósito geral. *Chips* baseados nessa metodologia passaram a ser chamados de SoC (*System-on-Chip*) (ZEFERINO, 2003) (BENINI; MICHELI, 2002).

O crescente avanço em relação ao tempo de resposta e a diminuição das dimensões dos dispositivos no processo de fabricação, permitirá que SoCs passem a integrar de dezenas a centenas de primitivas e realizem comunicação com banda de Gbits/s (GUERRIER; GREINER, 2000) (ZEFERINO et al., 2002) (LIU; ZHENG; TENHUNEN, 2004).

A comunicação utilizada nos SoCs atuais é feita de duas formas diferentes: canais dedicados e barramentos compartilhados(ZEFERINO et al., 2002).

Os canais dedicados possuem o melhor desempenho, pois a comunicação entre pri-

mitivas (unidades funcionais, processadores, memórias entre outros) ocorre por meio de canais exclusivos, contudo, não são muito escaláveis.

Em contrapartida, um barramento é escalável, mas não permite paralelismo entre as primitivas, assim, sua largura de banda é compartilhada por todas as primitivas do sistema, e sua freqüência de operação diminui com o crescimento do sistema. Contudo, algumas dessas restrições podem ser otimizadas por meio de arquiteturas hierárquicas (ZEFERINO et al., 2002) ou barramentos mais largos.

Em arquiteturas hierárquicas ou barramentos segmentados, dois ou mais barramentos são conectados por meio de pontes e dependendo da forma de implementação, cada barramento tem autonomia suficiente para enviar informação a primitivas que estejam nele acopladas, conseguindo com isso, um certo grau de paralelismo entre os diversos barramentos. O paralelismo é reduzido quando uma primitiva acoplada a um barramento precisa se comunicar com uma outra primitiva que esteja acoplada a um segundo barramento, sendo necessário a utilização das pontes para garantir o controle de todos os barramentos que em conjunto, permitem o acesso à primitiva desejada.

Agora, imagine-se um único *chip* com diversas primitivas produzindo os mais variados tráfegos na rede e exigindo do sistema de comunicação uma vazão bastante elevada. Estas exigências continuarão crescendo conforme se incorporem mais primitivas ao *chip* (sistema escalável), e conforme o desempenho de tais primitivas melhore (GUERRIER; GREINER, 2000).

Alguns trabalhos recentes (DALLY; TOWLES, 2001), (BENINI; MICHELI, 2002) têm proposto uma nova forma de interconexão entre as primitivas, cujo protocolo de comunicação permite simultaneamente a realização de sistemas escaláveis e de bom desempenho.

Nas próximas seções especifica-se essa forma emergente de interconexão de SoCs.

3.2 Alguns Conceitos sobre Redes-em-*Chip*

As Redes-em-*chip* ou *networks-on-chip* (NoCs) são baseadas nos fundamentos já consolidados das redes de computadores (TANENBAUM, 1996) e provêem uma infraestrutura de comunicação para as primitivas do sistema. Com isso, é possível o desenvolvimento independente de primitivas sem levar em consideração o projeto dos demais elementos do sistema. Uma rede-em-*chip* é estabelecida pelo acoplamento de primitivas na malha de comunicação, a qual é capaz de executar um protocolo de comunicação que efetivamente

realiza o transporte de informação de uma primitiva a outra, de forma semelhante ao que ocorre quando dois computadores em rede se comunicam (KUMAR et al., 2002).

Redes-em-*chip* com a característica de configurabilidade, podem constituir uma plataforma de projeto flexível, sendo capaz de se adaptar às necessidades das mais variadas cargas no tráfego de informação da rede (KUMAR et al., 2002).

Assim, uma rede-em-*chip* pode ser definida como um conjunto de processadores de rotas ou simplesmente roteadores interligados por canais ponto-a-ponto. Tipicamente, o modelo de comunicação utilizado é o de troca de mensagens, sendo que a comunicação entre primitivas, as quais são acopladas aos roteadores, é feita pelo envio e recebimento de mensagens de requisição e de resposta (ZEFERINO, 2003) (DALLY; TOWLES, 2001) (LIU; ZHENG; TENHUNEN, 2004).

Uma mensagem é transferida utilizando-se uma seqüência de pacotes, os quais podem se constituir em três partes, como mostrado na figura 3.1 (CULLER; GUPTA; SINGH, 1997) (ZEFERINO, 2003):

- CABEÇALHO: sinaliza o início do pacote e inclui informações de controle e roteamento que permitem à rede determinar o procedimento a ser adotado para a transferência do pacote;
- CARGA ÚTIL: é o conteúdo do pacote que está sendo transmitido;
- TERMINADOR: finaliza o pacote e pode conter informações para a verificação de erros.

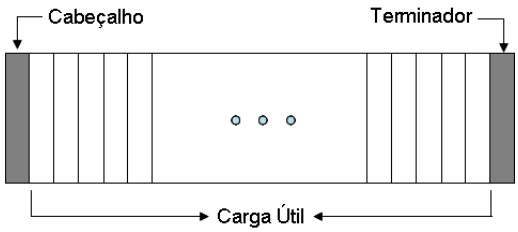


Figura 3.1: Estrutura de um pacote numa Rede-em-*Chip*
Adaptada de (ZEFERINO, 2003)

Assim como em redes de computadores, uma rede em *chip* é também caracterizada pela sua topologia e pelos mecanismos de comunicação utilizados. Os principais mecanismos de comunicação são (ZEFERINO et al., 2002) (CULLER; GUPTA; SINGH, 1997):

- MEMORIZAÇÃO: determina o esquema de filas utilizado para armazenar uma mensagem;
- ARBITRAGEM: resolve conflitos internos na rede, quando duas ou mais mensagens competem por um mesmo recurso (fila ou canal de saída);
- CHAVEAMENTO: define como uma mensagem é transferida da entrada de um roteador para um de seus canais de saída;
- CONTROLE DE FLUXO: lida com a alocação dos recursos (filas e canais) necessários para uma mensagem avançar pela rede, regulando o tráfego nos canais; e
- ROTEAMENTO: define o caminho a ser utilizado por uma mensagem para atingir o seu destino.

3.2.1 Topologia

A topologia consiste na organização da rede sob a forma de um grafo, no qual os roteadores são os vértices do grafo e os canais de comunicação são os arcos.

Na literatura de arquitetura paralela de computadores, diversas topologias são mencionadas, como malhas bidimensionais, *torus*, cubos, árvores, borboletas e redes Bene (CULLER; GUPTA; SINGH, 1997). Apesar das redes-em-*chip* serem baseadas nas redes de computadores, elas não possuem exatamente o mesmo escopo de projeto do que as redes de computadores. Isso se deve ao fato de que existem restrições e requisitos diferentes entre os dois níveis de implementação. Por exemplo, algumas topologias citadas anteriormente são estruturas 3-D. Já as redes-em-*chip* são construídas utilizando-se estruturas 2-D, como a malha e a árvore, as quais são mais adequadas às tecnologias atuais de fabricação (ZEFERINO et al., 2002) (KUMAR et al., 2002).

Outra diferença diz respeito à largura dos canais de dados. Nos roteadores de redes de computadores, essa largura é limitada pelo número de pinos do encapsulamento, enquanto nas redes-em-*chip* não existe tal limitação, pois os canais de comunicação são internos (CULLER; GUPTA; SINGH, 1997) (ZEFERINO et al., 2002).

3.2.2 Memorização

A organização das filas nos roteadores pode ter um impacto significativo no desempenho e custo do sistema de comunicação. Existem quatro possibilidades: roteadores sem

fila (apenas ocorre o chaveamento dos sinais da entrada para a porta de saída), filas na entrada, filas na saída, ou uma fila centralizada (CULLER; GUPTA; SINGH, 1997).

Com a opção de fila na entrada, como mostrado na figura 3.2, o roteador deve ser capaz de armazenar um conjunto de dados (pacotes) para cada porta de entrada.

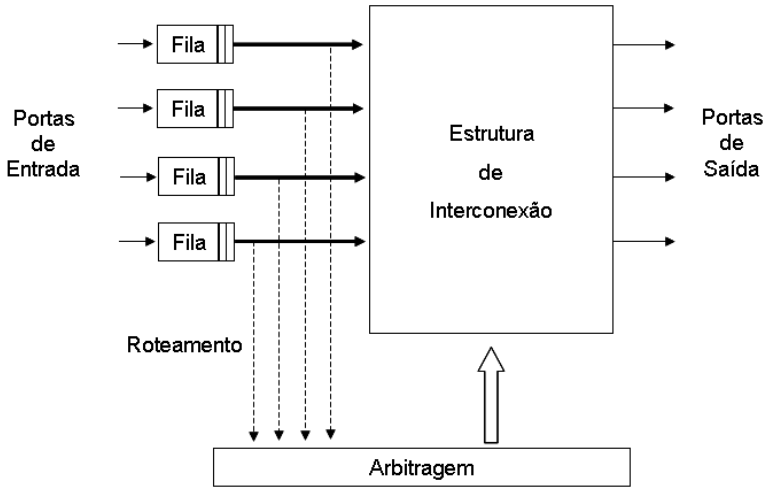


Figura 3.2: Roteador com filas na entrada
Adaptada de (CULLER; GUPTA; SINGH, 1997)

A operação do roteador é relativamente simples, monitora-se o topo da fila de entrada, determina-se a porta de saída desejada, e faz-se o correto chaveamento para efetivar a transferência de informação da fila de entrada para a porta de saída. Assim, normalmente associa-se, a cada porta de entrada, a lógica de roteamento para determinar a saída desejada (CULLER; GUPTA; SINGH, 1997).

Um problema associado com o uso de filas na entrada é a possibilidade de ocorrência de *head-of-line* (CULLER; GUPTA; SINGH, 1997). Suponha-se que duas portas de entrada possuam pacotes destinados à mesma porta de saída. Uma delas será chaveada para a saída desejada enquanto a outra porta de entrada será bloqueada. Outro pacote que, por ventura, estivesse logo atrás do pacote bloqueado e fosse destinado a uma porta de saída não utilizada estaria também bloqueado.

O problema de *head-of-line* pode ser resolvido utilizando-se filas nas portas de saída (CULLER; GUPTA; SINGH, 1997). Assim, os pacotes que chegam nas portas de entrada são primeiramente roteados e depois armazenados nas filas de saída. Roteadores projetados com filas na saída podem precisar de uma quantidade maior de memória e de interconexões.

Numa fila centralizada, cada porta de entrada deposita dados em uma memória cen-

tralizada. Espera-se que, devido à centralização da informação, se possam utilizar melhor os recursos disponíveis. Em contrapartida, as portas de entrada/saída podem precisar de acesso simultâneo à memória centralizada, o que dificultaria o seu projeto. Outro problema ocorre quando uma única porta de entrada utiliza quase toda a memória centralizada para armazenar dados: isto poderia impedir outros pacotes de seguir adiante na malha de comunicação (CULLER; GUPTA; SINGH, 1997).

3.2.3 Arbitragem

O problema de arbitragem, em determinados casos, pode ser visto como vários módulos de arbitragem indepententes, um para cada porta de saída (CULLER; GUPTA; SINGH, 1997). Assim, a lógica de roteamento determina a porta de saída à qual deve ser entregue a informação (pacote), estabelecem-se os sinais de requisição para a porta de saída selecionada e a lógica de arbitragem daquela saída selecionada escolhe um sinal de requisição entre os vários pedidos realizados. Então a interconexão é estabelecida no caminho de dados para a realização da transferência de informação, enviando-se, à porta de entrada escolhida pela lógica de arbitragem, um sinal de pedido de requisição atendido.

Diversos algoritmos podem ser utilizados para a realização da arbitragem, como por exemplo prioridade estática, aleatória ou *round-robin*.

A implementação da prioridade estática é bem simples, em contrapartida pode ocasionar o bloqueio, por tempo indefinido, de um pacote numa porta de entrada/saída cuja prioridade seja baixa. Isto ocorrerá quando houver um congestionamento em portas com prioridades mais altas. O algoritmo *Round-robin* pode ser utilizado para eliminar este problema, mas necessita de lógica extra para armazenar e modificar as ordens de prioridade. Por sua vez, uma prioridade determinada aleatoriamente pode se tornar bastante complexa, o que exigiria uma quantidade razoável de lógica para a sua implementação (CULLER; GUPTA; SINGH, 1997).

O problema de arbitragem se torna mais complexo quando a lógica de roteamento puder indicar várias portas de saída como candidatas a receber um determinado pacote (CULLER; GUPTA; SINGH, 1997). Assim, duas ou mais portas de saída receberão pedidos de requisições para a transferência do mesmo pacote. O problema ocorre quando duas ou mais portas de saída selecionarem o mesmo pacote. Para impedir esse problema, as lógicas de arbitragem não podem ser mais independentes e devem interagir umas com as outras para garantir que apenas uma porta de saída receba o pacote.

O problema pode ser solucionado com a formação de um grafo bipartido e por meio de algoritmos de casamento ou correspondência (*matching*) (CULLER; GUPTA; SINGH, 1997). Este caso envolve questões de sofisticação *versus* velocidade, pois, talvez possa ser mais interessante desenvolver um algoritmo mais simples, permitindo um processamento mais rápido do que utilizar um algoritmo de arbitragem complexo que aumenta o ciclo de relógio.

3.2.4 Chaveamento

A estratégia de chaveamento pode ser subdividida em dois grandes grupos, quais sejam: circuito ou pacote (CULLER; GUPTA; SINGH, 1997). No chaveamento por circuito, estabelece-se um caminho entre a primitiva de origem e a primitiva de destino, reservando os recursos necessários para então realizar a transmissão da mensagem sobre o circuito formado. Esta estratégia é interessante quando uma quantidade muito grande de informação tiver que ser transferida, em contrapartida, pode ocasionar muitos bloqueios visto que o circuito de comunicação fica reservado somente para uso das primitivas que estabeleceram a comunicação, só sendo liberado no processo de encerramento.

O chaveamento por pacotes reparte a mensagem em uma sequência de pacotes. Um pacote contém informações relacionadas ao roteamento e seqüenciamento, além da carga útil. Assim, pacotes são individualmente roteados da origem ao destino e, por meio das informações de seqüenciamento, a mensagem é remontada no destino. Tipicamente, o chaveamento por pacotes permite uma melhor utilização da rede de comunicação, porque canais e filas são ocupados momentaneamente na travessia dos pacotes pelos roteadores (CULLER; GUPTA; SINGH, 1997).

3.2.5 Controle de Fluxo

Um controle de fluxo se faz necessário porque a capacidade de memorização de um roteador de destino pode não ser suficiente para armazenar as informações provenientes do roteador de origem. Utilizando, por exemplo, um protocolo *handshake* pode-se impedir a perda de informação quando os recursos de memorização se esgotarem; o remetente envia um pedido de requisição ao destinatário, que responde com um sinal de aceitação (CULLER; GUPTA; SINGH, 1997). Desta forma, se os recursos de memória do destinatário estiverem disponíveis, a transferência se efetiva, os recursos de memória do destinatário se tornam indisponíveis e os do remetente livres.

Outro tipo de controle de fluxo é por meio de um protocolo baseado em créditos (CULLER; GUPTA; SINGH, 1997). Quando o destinatário liberar uma certa quantidade de informação enviada por um roteador, a lógica de destino pode enviar créditos para o roteador de origem. Assim, o envio de créditos para um roteador de origem sinaliza a liberação de recursos, que permite a este roteador enviar novas informações; contudo, o envio destas diminue os créditos disponíveis no roteador de destino, limitando, assim, a quantidade de informação que poderá ser transmitida.

3.2.6 Roteamento

O processo de roteamento em um chaveamento pode utilizar basicamente duas técnicas: *store-and-forward* e *cut-through* (CULLER; GUPTA; SINGH, 1997), mostradas na figura 3.3, onde um pacote composto por quatro quadros atravessa três roteadores da origem até o destino. A técnica *store-and-forward*, como mostrada na figura 3.3(a), armazena todo o pacote num roteador para depois roteá-lo, determinando o caminho que deve ser seguido.

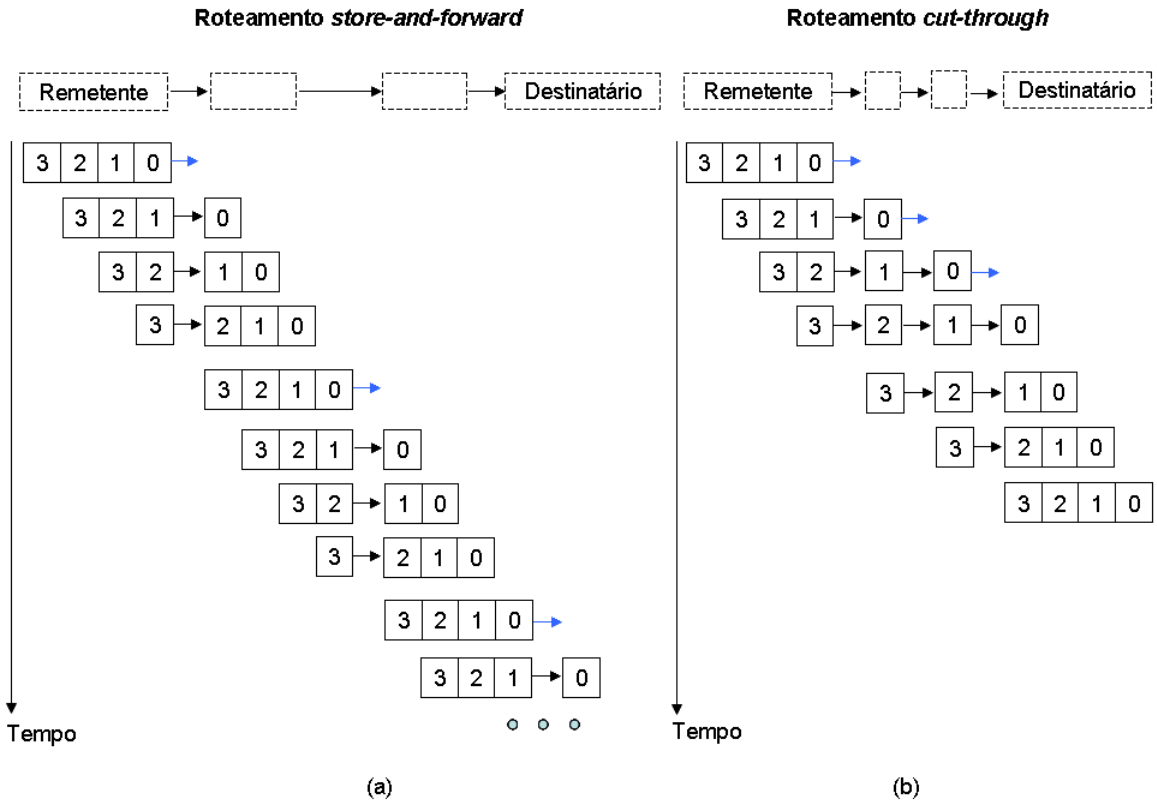


Figura 3.3: Roteamento (a) *store-and-forward* e (b) *cut-through*
Adaptada de (CULLER; GUPTA; SINGH, 1997)

Por sua vez, a técnica de roteamento *cut-through* com apenas um único quadro (cabeçalho) armazenado no roteador é capaz de determinar a próxima rota do pacote.

Assim, de acordo com a figura 3.3(b), os quadros subseqüentes são transferidos automaticamente para a porta de saída estabelecida anteriormente pelo primeiro quadro (cabeçalho), fazendo com que a transmissão do pacote seja realizada como em um *pipeline*. Normalmente, a técnica de roteamento *cut-through* costuma ter uma latência menor do que a técnica *store-and-forward* devido à formação do *pipeline* na transmissão dos pacotes (CULLER; GUPTA; SINGH, 1997).

Uma questão importante no processo de roteamento é o procedimento adotado para realizar a contenção da rede. Num chaveamento por circuito, a contenção pode ocorrer na tentativa de se estabelecer uma conexão entre a primitiva de origem e a de destino: a lógica do roteador identifica o próximo roteador que fará parte da conexão e este dará sequência ao algoritmo de roteamento até que se alcance o roteador de destino, efetivando-se a conexão entre a primitiva de origem e a de destino. A contenção ocorrerá quando um dos roteadores que pertence a rota de conexão estiver ocupado e não puder ser reservado para realizá-la, obrigando os roteadores já reservados a se tornar disponíveis. A tentativa de estabelecer a conexão poderá ser retomada pela origem após um tempo determinado aleatoriamente, por exemplo (CULLER; GUPTA; SINGH, 1997).

Por sua vez, utilizando um chaveamento por pacotes e o roteamento *store-and-forward*, a contenção ocorrerá quando vários pacotes armazenados num roteador precisarem ser roteados para a mesma porta de saída. Com isso, apenas um pacote será selecionado enquanto os outros ficarão bloqueados nas filas até serem selecionados (CULLER; GUPTA; SINGH, 1997).

No roteamento *cut-through*, duas opções de bloqueio estão disponíveis, quais sejam: *virtual cut-through* e *wormhole* (CULLER; GUPTA; SINGH, 1997).

A opção *virtual cut-through* armazena todos os quadros do pacote num único roteador, levando a rede de comunicação a se comportar, em casos de contenção, de forma semelhante ao roteamento *store-and-forward*.

A opção *wormhole*, em casos de contenção, permite o armazenamento de apenas alguns quadros do pacote em um roteador. Assim, o pacote passa a ser armazenado nos vários roteadores que pertencem ao trajeto já estabelecido pela lógica de roteamento.

O processo de roteamento pode utilizar três mecanismos básicos para determinar o trajeto de um pacote: armazenamento de tabelas de rotas, processamento na origem ou operações aritméticas (CULLER; GUPTA; SINGH, 1997).

No roteamento baseado em tabelas, cada roteador possui uma tabela de rotas utilizada

para determinar a porta de saída de um pacote. O cabeçalho de um pacote apresenta um campo de roteamento que funciona como índice na entrada da tabela de rotas. O roteador, com esse índice e com a tabela de rotas, determina a porta de saída e o novo valor (índice) do campo de roteamento do pacote. Um problema desta abordagem é a quantidade variável de memória necessária para a implementação da tabela. Além disso, pode ser necessária lógica extra para estabelecer o conteúdo das tabelas (CULLER; GUPTA; SINGH, 1997).

No roteamento baseado na origem, a primitiva responsável pela criação da mensagem deve construir um cabeçalho indicando a porta de saída para cada roteador do trajeto. Assim, cada roteador extrai a porta de saída correspondente e efetiva a transferência. Esta abordagem permite a confecção de roteadores bastante simples visto que a tarefa mais complexa de roteamento é realizada na primitiva. Um problema desta abordagem é o tamanho variável do cabeçalho, que pode ficar muito extenso (CULLER; GUPTA; SINGH, 1997).

A realização de operações aritméticas simples pode ser suficiente para selecionar a porta de saída em cada roteador e com isso, definir o trajeto do pacote na rede de comunicação. Por exemplo, numa topologia de grelha (sistema cartesiano x e y) e utilizando a técnica de roteamento X-Y ou *dimension order* (CULLER; GUPTA; SINGH, 1997), o cabeçalho de um pacote pode ser composto por duas variáveis, Δx e Δy, indicando as variações nos eixos x e y, respectivamente, além de mais dois bits, um indicando se o sentido que o pacote deve percorrer é leste-oeste ou oeste-leste, enquanto o outro bit indica o sentido norte-sul ou sul-norte. Assim, quando um roteador receber um pacote numa porta de entrada, deverá ser verificado o valor da variável Δx. Caso Δx seja maior que zero, o pacote deverá continuar o trajeto na direção x e no sentido indicado no pacote que está sendo analisado (leste-oeste ou oeste-leste). Para tanto, o roteador deverá decrementar uma posição de Δx e enviar o pacote cujo campo Δx foi atualizado para a porta de saída correspondente. Agora, caso o roteador identifique que $\Delta x = 0$, a variável Δy deverá então ser analisada. Caso Δy seja maior que zero, o roteador deverá enviar o pacote na direção y e no sentido indicado pelo bit de sentido (norte ou sul), decrementando a variável Δy em uma posição. O pacote alcançará o seu destino quando tanto a variação Δx quanto a variação Δy forem iguais a zero, o que fará com que o roteador entregue o pacote à primitiva nele conectada.

A técnica de roteamento X-Y faz com que o pacote percorra primeiro os roteadores na direção x do trajeto, para só depois percorrer os roteadores na direção y.

Um algoritmo de roteamento pode ser classificado como determinístico ou adaptativo (CULLER; GUPTA; SINGH, 1997): é considerado determinístico (não adaptativo) se levar em consideração apenas a origem e o destino do pacote, sem se preocupar com o tráfego da rede e adaptativo se levar em consideração o tráfego da rede, podendo rotear o pacote por caminhos que estão sendo pouco utilizados.

Roteadores baseados em algoritmos determinísticos podem ser mais simples e mais rápidos; em contrapartida, os adaptativos podem ser tolerantes a falhas e utilizar melhor a rede de comunicação.

Se o processo de roteamento escolher sempre os caminhos mais curtos, da origem até o destino, o seu algoritmo de rotas é dito mínimo, caso contrário, é não-mínimo (CULLER; GUPTA; SINGH, 1997).

O algoritmo de roteamento X-Y é determinístico e mínimo. Um exemplo de algoritmo de roteamento adaptativo e não mínimo é o *hot-potato* (batata quente) (NILSSON, 2002) (NILSSON et al., 2003), onde o roteador trabalha com uma topologia de grelha e realiza o algoritmo no qual todo pacote que chega em suas portas de entrada, exceto o que provém da primitiva, é direcionado para uma porta de saída levando-se em consideração o valor de *stress* de cada roteador vizinho e a prioridade referente a cada pacote de entrada. Com isso, pacotes com baixa prioridade podem ser obrigados a seguir caminhos não mínimos. A prioridade aumenta gradualmente conforme o pacote percorre uma maior quantidade de roteadores. Com esse critério de prioridades elimina-se a possibilidade de ocorrência de *livelock* no qual o mecanismo de roteamento sempre seleciona um canal de saída que afasta a mensagem do seu destinatário, impedindo que a mesma chegue a seu destino.

3.3 *Starvation*, *Livelock* e *Deadlock*

Existem três possibilidades que podem impedir que uma mensagem chegue ao seu destinatário, quais sejam: *starvation*, *livelock* e *deadlock* (ZEFERINO, 2003) (CULLER; GUPTA; SINGH, 1997).

Quando o cabeçalho de uma mensagem chega a um roteador, ele é processado pelo mecanismo de roteamento que determina o canal de saída a ser utilizado e emite uma requisição ao árbitro responsável pela alocação do canal. Quando existirem múltiplas requisições para um mesmo canal de saída, o árbitro aplica um critério de prioridade para selecionar uma delas. Dependendo de como esse critério é atualizado e em uma situação de alto tráfego neste canal de saída, uma mensagem pode ser preterida indefinidamente e

nunca ser selecionada para utilizar o canal requisitado, vindo a sofrer o que se chama de *starvation*.

Se o mecanismo de roteamento selecionar um canal de saída que sempre afaste a mensagem do seu destinatário tem-se a ocorrência de *livelock*.

O terceiro problema, denominado *deadlock*, é o mais grave de todos, pois, além de impedir que uma mensagem chegue ao seu destinatário, ele pode levar à paralisação da rede. O *deadlock* ocorre quando há uma dependência cíclica na rede como ilustra a figura 3.4(a), na qual cada mensagem garantiu a alocação de um canal e requer o uso de um outro canal que não está disponível. Em uma rede como essa (topologia de grelha), podem ocorrer dois tipos de ciclos, ilustrados na figura 3.4(b), sendo que a solução de menor custo utilizada para evitar a ocorrência de *deadlock* consiste em proibir a realização de um subconjunto das curvas (ou voltas) que uma mensagem poderia realizar, evitando o surgimento de ciclos. Uma solução é a proibição das mudanças do eixo Y para o X como mostrado na figura 3.4(c). Com isso, qualquer mensagem deve primeiro trafegar pelo eixo X para depois percorrer o eixo Y, como no roteamento X-Y ou *dimension order* (CULLER; GUPTA; SINGH, 1997).

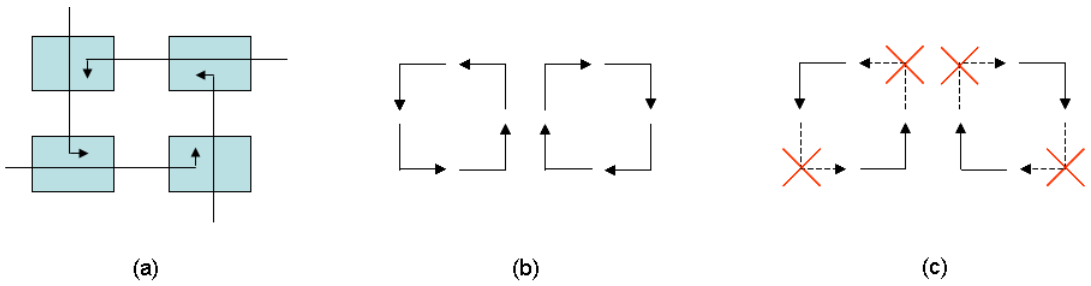


Figura 3.4: (a) Dependência cíclica – *deadlock*, (b) ciclos possíveis numa topologia de grelha e (c) roteamento X-Y (livre de *deadlock*)
Adaptada de (ZEFERINO, 2003)

Outra forma de se evitar ciclos é utilizando a técnica de canais virtuais (DALLY, 1992) (CULLER; GUPTA; SINGH, 1997). Um canal virtual consiste de uma fila capaz de armazenar uma certa quantidade de dados e associar algum tipo de informação de estado sobre estes dados. Por exemplo, na figura 3.5(a), o pacote B permanecerá bloqueado enquanto o pacote A também estiver bloqueado. Com o uso de canais virtuais, como mostrado na figura 3.5(b), o pacote B pode percorrer o seu trajeto sem ser bloqueado pelo pacote A devido ao uso de filas adicionais (canais virtuais). O uso de canais virtuais pode aumentar a quantidade de lógica na implementação de roteadores, além de aumentar o seu tempo de resposta (WU, 2002).

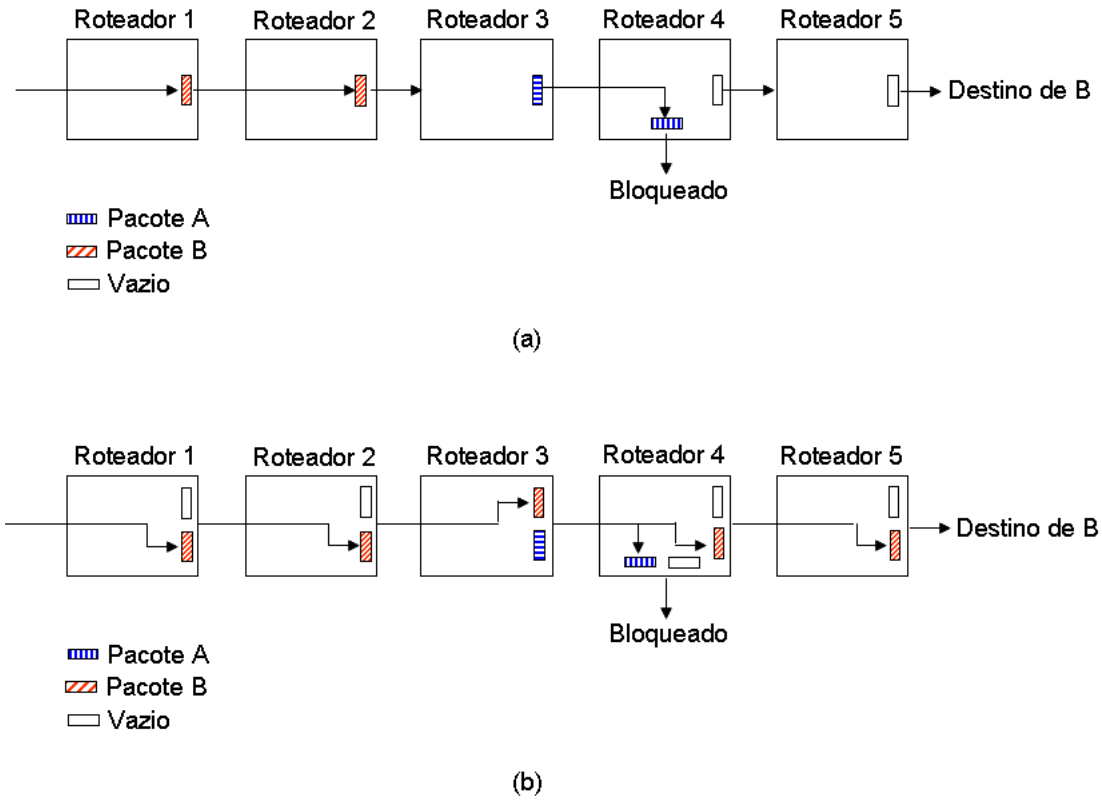


Figura 3.5: Comportamento da rede (a) sem o uso de canais virtuais e (b) com o uso de canais virtuais

Adaptada de (DALLY, 1992)

Em geral, a garantia da ausência de *starvation* é dada pelo mecanismo de arbitragem utilizado, enquanto que a ausência de *livelock* e *deadlock* depende exclusivamente do algoritmo de roteamento adotado (ZEFERINO, 2003).

3.4 A Rede-em-*Chip* CLICHE

Na figura 3.6 apresenta-se a rede-em-*chip* denominada CLICHE (*Chip-Level Integration of Communicating Heterogeneous Elements*) (KUMAR et al., 2002).

Essa arquitetura é constituída de primitivas e roteadores conectados por uma topologia de malha bidimensional, a qual permite a comunicação entre primitivas e roteadores via troca de mensagem. Tipicamente, as topologias de malhas bidimensionais estão sendo muito utilizadas em redes-em-*chip* devido ao fato das interconexões locais entre as primitivas e roteadores serem independentes do tamanho da rede. Além disso, o procedimento de roteamento em duas dimensões numa topologia de malha é de mais fácil desenvolvimento, o que resulta em roteadores com baixo custo de implementação, boa capacidade

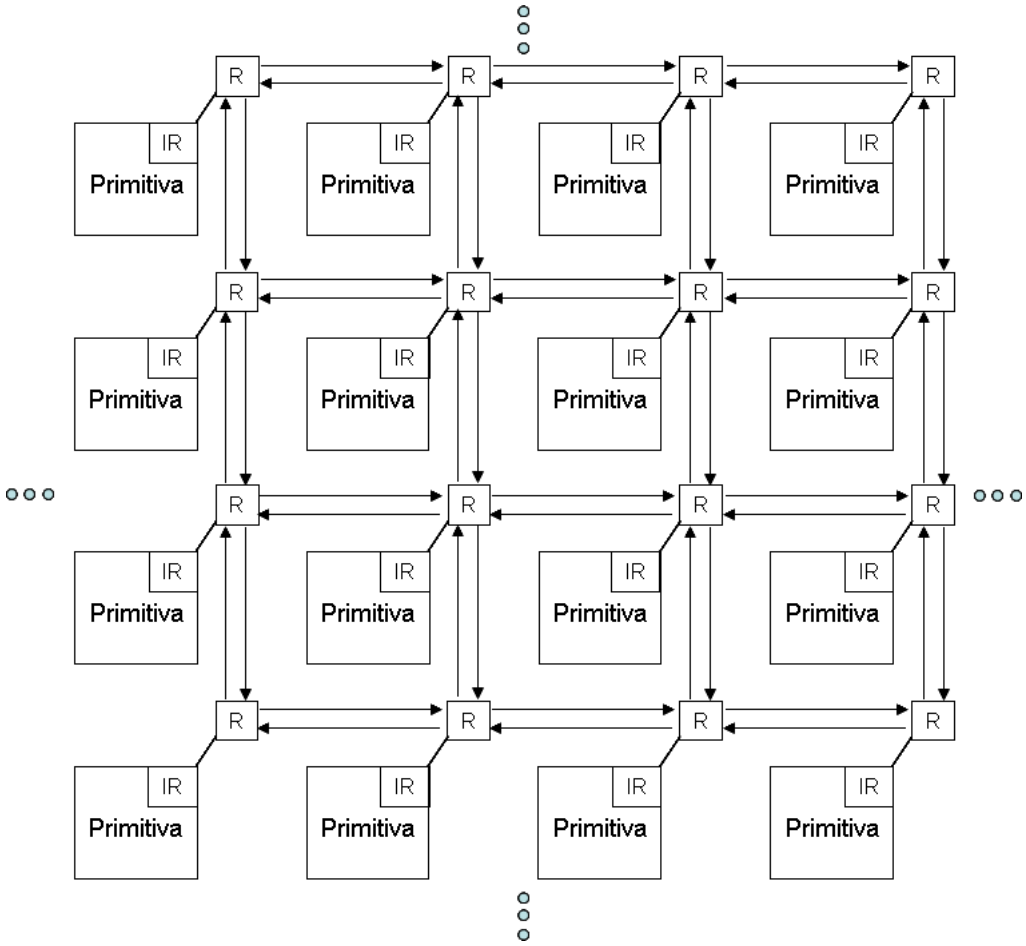


Figura 3.6: Rede-em-*chip* CLICHE com 16 primitivas, sendo R = **R**oteador e IR = **I**nterface de **R**ede

Adaptada de (KUMAR et al., 2002)

computacional, ciclos de relógio curtos e escaláveis (KUMAR et al., 2002).

Na figura 3.6, uma primitiva pode indicar uma unidade computacional, uma unidade de armazenamento ou uma combinação de ambas. Por sua vez, o roteador tem como principal função o roteamento e armazenamento de pacotes entre as primitivas. Cada roteador é conectado com outros quatro roteadores vizinhos por meio dos canais de entrada e saída e a uma primitiva com a qual se torna responsável pelo envio e recebimento de informações provenientes ou direcionadas a essa primitiva (KUMAR et al., 2002).

Um canal de comunicação consiste de dois barramentos unidirecionais ponto-a-ponto entre dois roteadores, ou entre um roteador e uma primitiva. O roteador é constituído por um conjunto de filas de entrada para armazenar pacotes e controlar um possível congestionamento da rede, multiplexadores para efetivar as interconexões necessárias para a transferência de um pacote para outro roteador ou para a primitiva nele acoplada, além de possuir controladores que implementam os mecanismos de comunicação.

A rede CLICHÉ permite o acoplamento dos mais variados tipos de primitivas. Exemplos típicos de primitivas que podem ser colocadas na arquitetura são: processadores de propósito geral e específicos, memórias cache, memórias RAMs e *hardware* reconfigurável. As primitivas também podem ser uma combinação das estruturas digitais citadas acima.

O modelo computacional desta rede-em-*chip* é, portanto, uma rede heterogênea de primitivas realizando um processamento local. A comunicação entre as diversas primitivas ocorre pelo envio de mensagens sobre a malha bidimensional. Cada primitiva opera assincronamente em relação às outras primitivas. Sincronismo entre primitivas deve ser realizado via primitivas de sincronismo, que são implementadas por meio da passagem de mensagem.

A comunicação de uma rede CLICHÉ com o ambiente externo é realizado por meio de primitivas de interface dedicadas. Tais primitivas podem ser utilizadas, por exemplo, para o interfaceamento com memórias externas ou para agrupar outras redes-em-*chip*. Primitivas de interface devem controlar o processo de armazenamento de dados, ordenação e formato dos pacotes para compatibilizar o sistema externo com a rede-em-*chip* em que estão acopladas (KUMAR et al., 2002).

Toda primitiva possui um endereço único, sendo conectada à rede por um roteador. Ela se comunica com o roteador por meio de uma interface de rede, mostrada na figura 3.6. Assim, qualquer primitiva equipada com a interface de rede pode ser acoplada à rede em um dos espaços disponíveis. A interface de rede é responsável pela conversão das mensagens num formato apropriado que possa ser interpretado ora por um roteador ora por uma primitiva.

3.5 Outras Redes-em-*Chip*

Existem outras redes já descritas na literatura, como a SPIN, OCTAGON e a SoCIN. A SPIN (GUERRIER; GREINER, 2000) possui uma topologia de árvore-gorda, roteamento adaptativo, *wormhole*, arbitragem distribuída, memorização na entrada e centralizada, e um controle de fluxo baseado em créditos. A rede-em-*chip* OCTAGON (KARIM; NGUYEN; DEY, 2002) possui uma topologia de anel cordal, roteamento determininístico, chaveamento por circuito ou pacotes e arbitragem centralizada. A rede-em-*chip* SoCIN (ZEFERINO; SUSIN, 2003) (ZEFERINO, 2003) possui uma topologia de grelha 2-D, roteamento determinístico, *wormhole*, arbitragem distribuída, memorização na entrada, e um controle de fluxo *handshake*.

3.6 Comunicação numa Rede-em-*Chip*

Numa arquitetura convencional de uma rede-em-*chip* definem-se quatro camadas de um protocolo de comunicação (BENINI; MICHELI, 2002) baseado no padrão OSI para redes de computadores (TANENBAUM, 1996), quais sejam: camadas física, enlace de dados, rede e transporte:

- CAMADA FÍSICA: determina o número e comprimento dos canais digitais que conectam primitivas e roteadores. Tais canais podem não ser muito confiáveis devido a possibilidade de ruídos, transmitindo-se, com isso, dados alterados;
- CAMADA DE ENLACE DE DADOS: a principal tarefa desta camada é tornar o canal físico aparentemente livre de erros para a próxima camada. Para tanto, a camada de enlace de dados deve agrupar os bits de uma mensagem em quadros e transmití-los seqüencialmente. Informações de início/fim do quadro e controle de fluxo, para garantir o sincronismo entre remententes mais rápidos e receptores mais lentos, podem ser necessários. Também podem ser utilizadas aqui algumas técnicas para a detecção e correção de erros, transmitindo-se informação adicional e redundante;
- CAMADA DE REDE: Nos SoCs atuais, onde as primitivas são interligadas por meio de um barramento, a camada de rede se torna, na maioria das vezes, desnecessária. Mas, no caso de uma rede-em-*chip*, onde existe uma coleção de canais que interligam as primitivas, se torna necessário estabelecer um procedimento para a transmissão de informação entre os canais sucessivos, além da necessidade de informações relacionadas ao roteamento de uma primitiva remetente para uma receptora;
- CAMADA DE TRANSPORTE: é uma camada fim-a-fim, isto é, são protocolos instalados apenas nas primitivas remetentes e receptoras e que de forma transparente, trabalham com informações provenientes das primitivas, sem levar em consideração todo o gerenciamento necessário para a transmissão de informações pela rede. Portanto, esta camada é independente da tecnologia implementada no sistema de comunicação e tem como principal objetivo transformar uma mensagem em pacotes que devem ser entregues à camada de rede para que se possa realizar, efetivamente, o envio dessa mensagem. Esta camada pode, por exemplo, estabelecer mais de uma conexão de rede e dividir a transmissão e recebimento de dados entre essas conexões para garantir maior vazão. Mas, se o custo de manter mais de uma conexão de rede for alto, a camada de transporte pode multiplexar várias mensagens numa mesma conexão de rede, reduzindo o custo do estabelecimento desta conexão.

Com relação a arquitetura CLICHE da figura 3.6, a interface de rede juntamente com a malha de comunicação, implementam as quatro camadas do protocolo de comunicação. Já a estrutura roteador-roteador implementa apenas as três primeiras camadas do protocolo (KUMAR et al., 2002).

3.7 Comentários

Este capítulo apresentou uma introdução às redes-em-*chip* visando fornecer alguns conceitos a respeito do tema, quais sejam: vantagens e desvantagens de uma rede-em-*chip* em relação às arquiteturas atualmente utilizadas, composição de um pacote, topologias e mecanismos de comunicação, exemplificando-se ao longo do texto uma rede-em-*chip* rotulada, na literatura científica, de CLICHÉ.

No próximo capítulo apresentam-se alguns conceitos matemáticos sobre a geração de números pseudo-aleatórios visando fornecer alguns conceitos utilizados no desenvolvimento do projeto que está sendo proposto.

4 Geração de Números Pseudo–Aleatórios

Resumo

Números pseudo-aleatórios podem ser gerados por meio de equações matemáticas recursivas. Estas equações geram uma seqüência periódica de números que depende dos valores escolhidos para os parâmetros do gerador, como os multiplicadores, os incrementos e os módulos. A escolha da semente do gerador determina o ponto no qual a seqüência é iniciada e os valores adotados para os parâmetros devem ser escolhidos para garantir um grande período e boas propriedades de uniformidade e aleatoriedade.

4.1 Introdução

A disponibilidade de números pseudo-aleatórios se faz necessária em diversas aplicações, como simulações de fenômenos físicos, na tomada de decisões e até mesmo em sistemas de entretenimento. Particularmente, para este projeto de pesquisa, números pseudo-aleatórios são utilizados para determinar as transições que poderão disparar, quando estas se encontrarem em situação de conflito. e também para permitir a implementação em *hardware* de modelos descritos em Redes de Petri com associação de probabilidade de disparo entre as transições.

Algumas fontes de números possivelmente aleatórios são o lançamento de dados, a retirada de bolas numeradas de uma urna (com reposição) ou o uso de uma roleta. Entretanto na maioria das vezes usa-se o que foi convecionado chamar de números pseudo-aleatórios, pois números aleatórios causam um impecílio no desenvolvimento de um sistema: não se tem como repetir uma dada seqüência de números com a intenção de verificar a simulação ou tentar corrigir uma falha de projeto (ROSA; JUNIOR, 2002).

Neste capítulo comentam-se sobre as características e as fórmulas matemáticas de

diversos tipos de geradores lineares propostos na literatura, quais sejam: geradores lineares congruentes, múltiplos, com trasporte, *Feedback Shift Register*, *Generalized Feedback Shift Register*, *Lagged-Fibonacci*, *Mersenne Twister Generalized Feedback Shift Register* e combinações entre geradores. Por fim, comenta-se sobre a disponibilidade em bibliotecas de programação dos geradores mais utilizados na simulação e implementação de sistemas.

4.2 Gerador Linear Congruente

A grande maioria dos geradores de números pseudo-aleatórios são implementações do método linear congruente. Este método utiliza a seguinte fórmula recursiva (HULL; DOBELL, 1964) (THESEN; SUN; WANG, 1984):

$$x_{i+1} = (a * x_i + c) \mod M$$

Onde a é o multiplicador, c é o incremento e M é o módulo. O módulo é um inteiro positivo. O multiplicador, o incremento e os x_i são todos inteiros positivos na faixa de 0 a $M-1$. A equação é realizada tomando o produto de a e x_i somando c, dividindo por M, e fazendo o próximo número pseudo-aleatório gerado, x_{i+1}, ser o resto desta divisão. O número resultante está na faixa de 0 a $M-1$. Esta equação gera uma seqüência periódica de números com período máximo sendo de M números dependendo dos valores escolhidos para os parâmetros a, c e M. A escolha de x_0 é chamada semente do gerador de números aleatórios, e determina o ponto no qual a seqüência é iniciada. Os valores para a, c e M devem ser escolhidos para obter um período largo e boas propriedades de uniformidade e aleatoriedade.

Quando $c \neq 0$ o gerador de números pseudo-aleatórios é dito *mixed*, caso contrário, é dito multiplicativo (HUTCHINSON, 1966) (MACLAREN; MARSAGLIA, 1965). Num gerador linear congruente multiplicativo não é permitido que x_i seja igual a 0, pois todos os números que serão gerados após x_i serão também iguais a zero. Por esse motivo, o período deste gerador é de no máximo $M-1$ números (SCHOLLMEYER; TRANTER, 1991).

A operação que mais consome tempo em um programa de geração de números aleatórios é a divisão e a multiplicação. Este custo computacional pode ser relevante quando se deseja gerar grandes quantidades de números. Por isso, muitos geradores de números pseudo-aleatórios utilizam valores específicos para os multiplicadores e módulos de forma a reduzir este custo computacional. Módulos da forma $M = 2^e$ com $e > 1$, denominados módulos em potência de dois, ou módulos da forma $M = 2^e + h$, com h

inteiro e suficientemente pequeno, e multiplicadores da forma $\pm 2^q \pm 2^r$, onde q e r são inteiros, possibilitam a geração de números pseudo-aleatórios utilizando apenas operações de deslocamentos, somas e/ou subtrações (WU, 1997) (L'ECUYER, 1998) (CARTA, 1990).

Apesar dessas técnicas diminuirem o tempo de processamento na geração de números pseudo-aleatórios, elas podem reduzir significativamente o período máximo da seqüência e gerar grandes degradações na qualidade dos números gerados (L'ECUYER; SIMARD, 1999) (L'ECUYER, 1997) (SCHOLLMEYER; TRANTER, 1991).

4.3 Gerador Linear Múltiplo

Os geradores lineares congruentes são simples, fáceis de implementar e razoavelmente rápidos computacionalmente. Contudo, o período desses geradores é limitado pelo tamanho do módulo M, o qual, por sua vez, é limitado pela arquitetura interna do sistema (L'ECUYER; BLOUIN; COUTURE, 1993).

Geradores mais recentes têm sido propostos como extensões naturais dos geradores lineares congruentes, nos quais o próximo número gerado é computado iterativamente a partir dos k números anteriores (DENG; XU, 2003) (L'ECUYER, 1998):

$$x_i = (a_1 * x_{i-1} + \ldots + a_k * x_{i-k}) \mod M, \quad i \geq k$$

O módulo M e a ordem k são inteiros positivos e os coeficientes a_i pertencem a $\mathbb{Z}_M = \{0, 1 \ldots, M-1\}$. Os primeiros números representados por $(x_0, x_1, \ldots, x_{k-1})$ são as sementes do gerador. Para $k = 1$ tem-se o gerador linear congruente clássico.

Com determinados valores primos para o módulo M, a seqüência de números gerada pelo método linear múltiplo pode atingir um período de até $M^k - 1$ (L'ECUYER; TOUZIN, 2000). A obtenção de uma seqüência de números com boas propriedades de aleatoriedade e uniformidade depende dos parâmetros escolhidos. Assim, os autores em (L'ECUYER; BLOUIN; COUTURE, 1993) identificam parâmetros adequados para a obtenção de alguns geradores lineares múltiplos.

Os geradores lineares múltiplos podem alcançar períodos maiores do que os geradores lineares congruentes, visto que dependem também da ordem K utilizada na equação. Desta forma, se for necessário um período maior para a simulação de um determinado processo, por exemplo, basta aumentar a ordem de recursão da equação. Com os geradores lineares congruentes isso não seria possível, pois o período é limitado pelo módulo M, o

qual é limitado pelas restrições da arquitetura onde o gerador está sendo implementado.

Nos geradores lineares múltiplos é possível a repetição de um mesmo número dentro da seqüência gerada, visto que o gerador só reiniciará a seqüência quando os últimos k números forem iguais aos valores atribuídos às sementes do gerador. O mesmo não acontece com os geradores lineares congruentes, pois possuem somente uma semente e a repetição de um número, na verdade, indica o reinício da seqüência.

Contudo, os geradores lineares múltiplos podem ser menos eficientes do que os geradores congruentes, porque várias operações de multiplicação são necessárias. Para aumentar a velocidade na geração dos números pseudo-aleatórios, o autor em (L'ECUYER, 1998) considerou a utilização da equação recursiva com apenas dois coeficientes a_i, $1 \leq i < k$ e a_k sendo diferentes de zero. Os autores em (DENG; XU, 2003) atribuiram o mesmo valor para diversos coeficientes, desta forma pôde-se agrupar os x_i correspondentes e realizar apenas operações de soma ou subtração.

4.4 Gerador Linear com Transporte

A utilização de um número em potência de dois para o módulo de um gerador linear múltiplo torna a implementação mais fácil, permitindo uma execução mais rápida, porém reduz significativamente tanto o período da seqüência como a qualidade dos números gerados se comparado a um módulo primo (L'ECUYER, 1998).

Uma forma de utilizar uma potência de dois para o módulo, mantendo um longo período e o potencial para a obtenção de seqüências de boa qualidade é por meio de um gerador linear com transporte do tipo multiplicação-com-transporte ou *multiply-with-carry* (*MWC*). Um gerador linear do tipo multiplicação-com-transporte pode ser assim esquematizado (L'ECUYER, 1997) (COUTURE; L'ECUYER, 1995):

$$\begin{aligned} x_i &= (a_1 * x_{i-1} + \ldots + a_k * x_{i-k} + c_{i-1}) \mod B, \quad i \geq k \\ c_i &= (a_1 * x_{i-1} + \ldots + a_k * x_{i-k} + c_{i-1}) \text{ div } B \end{aligned}$$

onde a função *div* realiza uma divisão retornando um valor inteiro, B pode ser uma potência de dois e a variável inteira c_i representa o transporte na iteração i.

A seqüência de números pseudo-aleatórios gerada pode passar por dois estágios diferentes: um transiente e um cíclico. A partir de um estado inicial, o gerador pode produzir uma seqüência transiente de números e depois entrar num estágio periódico ou cíclico, no

qual os números pseudo-aleatórios gerados começarão a se repetir a partir de um determinado momento. Uma vez no estágio periódico, os valores computados para a variável de transporte permanecem em um intervalo finito. Assim, a duração do estágio transiente depende de quão longe a variável de transporte está deste intervalo (GORESKY; KLAPPER, 2003).

O gerador linear do tipo multiplicação-com-transporte pode ser aproximadamente considerado um gerador linear congruente de módulo $M = \sum_{i=0}^{k} a_i * b^i$, com $a_0 = -1$ (L'ECUYER, 1997).

Outros dois tipos de geradores lineares com transporte apresentados na literatura, quais sejam: gerador linear do tipo soma-com-transporte, *add-with-carry* (*AWC*), e o gerador do tipo subtração-com-empréstimo, *subtract-with-borrow* (*SWB*), podem ser considerados casos especiais do gerador do tipo multiplicação-com-transporte, onde somente dois coeficientes a_i são diferentes de zero e ambos iguais a ± 1 (L'ECUYER, 1997).

O gerador linear do tipo soma-com-transporte é baseado na seguinte recursão (TEZUKA; L'ECUYER; COUTURE, 1993):

$$\begin{aligned} x_i &= (x_{i-s} + x_{i-k} + c_{i-1}) \mod B \\ c_i &= I(x_{i-s} + x_{i-k} + c_{i-1} \geq B) \end{aligned}$$

onde k e s são inteiros positivos, sendo $k > s$, e I é a função de indicação de transporte, cujo valor é 1 se o argumento for verdadeiro, e 0 caso contrário. O período máximo da seqüência com este tipo de gerador é de $B^k + B^s - 2$ números (TEZUKA; L'ECUYER; COUTURE, 1993).

O gerador linear do tipo subtração-com-empréstimo é baseado na seguinte recursão (TEZUKA; L'ECUYER, 1992):

$$\begin{aligned} x_i &= (x_{i-s} - x_{i-k} - c_{i-1}) \mod B \\ c_i &= I(x_{i-s} - x_{i-k} - c_{i-1} < B) \end{aligned}$$

onde c_i representa o empréstimo na iteração i. O período máximo da seqüência com este tipo de gerador é de $B^k - B^s$ números (TEZUKA; L'ECUYER, 1992).

4.5 Gerador Linear do Tipo *Feedback Shift Register*

Esta abordagem trabalha diretamente sobre variáveis booleanas para formar um conjunto binário pseudo-aleatório cuja seqüência pode ter boas propriedades estatísticas, além de possuir um período arbitrariamente longo e independente do tamanho do módulo, o qual é limitado pela arquitetura interna do sistema onde o gerador for implementado (THESEN; SUN; WANG, 1984).

Geradores lineares do tipo *feedback shift register*, também conhecidos como geradores Tausworthe, baseiam-se na seguinte equação de recorrência (L'ECUYER; PANNETON, 2000) (BAREL, 1983):

$$b_i = (a_1 * b_{i-1} + \ldots + a_k * b_{i-k}) \mod 2$$

onde b_i representa uma variável booleana (0 ou 1), $k > 1$ é a ordem de recorrência, o coeficiente $a_k = 1$ e os coeficientes $a_j \in \{0, 1\}$ para $j = 1, \ldots, k-1$.

Para que toda a seqüência de valores booleanos gerada comece a se repetir novamente é necessário que o conjunto formado pelas últimas k variáveis booleanas geradas se repita e por isso o período máximo é de $2^k - 1$, visto que o número de valores diferentes possíveis para o conjunto é de $2^k - 1$ excluindo-se o vetor de valores $(0, 0, \ldots, 0)$ (BAREL, 1983) (CHIU, 1988).

Utilizando apenas dois coeficientes diferentes de zero, a_k e a_r, pode-se construir geradores do tipo *feedback shift register* cuja seqüência seja máxima (BAREL, 1983). Assim, a equação de recorrência anterior passa a ser:

$$b_i = (b_{i-r} + b_{i-k}) \mod 2$$

onde k e r são inteiros fixos com $0 < r < k$.

A operação de soma módulo 2 da equação anterior é equivalente à operação binária ou-exclusivo, assim (BAREL, 1983):

$$b_i = b_{i-r} \oplus b_{i-k}$$

Para a obtenção de uma seqüência de inteiros pseudo-aleatórios a partir da equação binária anterior, pode-se agrupar consecutivamente as variáveis booleanas geradas para a formação das representações binárias dos inteiros (CHIU, 1988).

4.6 Gerador Linear do Tipo *Generalized Feedback Shift Register*

Como comentado anteriormente, o gerador do tipo *feedback shift register* trabalha diretamente sobre variáveis booleanas e para a obtenção de uma seqüência de inteiros pseudo-aleatórios pode-se agrupar consecutivamente as variáveis geradas para a formação das representações binárias dos inteiros. Contudo esta prática, muitas vezes, pode levar a uma seqüência pseudo-aleatória ruim (CHIU, 1988).

O gerador do tipo *generalized feedback shift register* é capaz de trabalhar diretamente com números inteiros e se baseia na seguinte equação de recorrência (L'ECUYER, 1998) (LEWIS; PAYNE, 1973):

$$x_i = x_{i-r} \oplus x_{i-k}$$

onde k e r são inteiros fixos com $0 < r < k$, $\oplus$ é a operação binária ou-exclusivo e x_i são variáveis inteiras. O gerador deve ser iniciado com k números, ou sementes, devidamente escolhidos para a obtenção de uma boa seqüência pseudo-aleatória (COLLINGS; HEMBREE, 1986). Este gerador possui um período máximo de $2^k - 1$ números (LEWIS; PAYNE, 1973) (MARSAGLIA, 1985).

4.7 Gerador do Tipo *Lagged-Fibonacci*

Este gerador, na realidade, é uma generalização da equação de recorrência que representa o gerador linear do tipo *Generalized Feedback Shift Register*, no qual a operação lógica ou-exclusivo pode ser trocada por uma função lógica ou aritmética arbitrária, como soma, subtração ou multiplicação de módulo M (L'ECUYER, 1998). Assim, a equação de recorrência pode ser estabelecida da seguinte forma (MARSAGLIA, 1985) (BRENT, 1992):

$$x_i = x_{i-r} \odot x_{i-k}$$

onde k e r são inteiros fixos com $0 < r < k$, x_i são variáveis inteiras e $\odot$ é uma operação lógica ou aritmética arbitrária, como soma (mod M), subtração (mod M), multiplicação (mod M) ou $\oplus$.

Dependendo dos valores atribuídos aos parâmetros r, k e M, com $M = 2^w$, o período máximo gerado pela equação de recorrência acima pode alcançar $2^k - 1$ números se $\odot = \oplus$, $(2^k - 1) * 2^{w-1}$ se $\odot = \pm \mod M$ e $(2^k - 1) * 2^{w-3}$ se $\odot = * \mod M$, onde $\pm$ indica

soma ou subtração e $*$ indica multiplicação (MARSAGLIA, 1985) (BRENT, 1992).

Alguns testes realizados sobre estes geradores mostraram que a utilização da operação de multiplicação na equação de recorrência produzem melhores resultados de aleatoriedade (MARSAGLIA, 1985) (CODDINGTON, 1997) (BRENT, 1992). Na seqüência vem as operaçoões de soma ou subtração e, por último, a operação ou-exclusivo.

Contudo, o custo computacional também cresce com a utilização das operações que produzem os melhores resultados, ou seja, a multiplicação é a operação computacionalmente mais lenta, seguida da operação de soma ou subtração e, por último, a operação ou-exclusivo, a mais rápida.

As propriedades de aleatoriedade da seqüência gerada podem ser melhoradas com a utilização de um maior número de variáveis na fórmula de recorrência, como, por exemplo, o gerador de três variáveis: $x_i = x_{i-r} \odot x_{i-t} \odot x_{i-k}$, com $0 < r < t < k$ (BRENT, 1992) (CODDINGTON, 1997).

4.8 Gerador *Mersenne Twister Generalized Feedback Shift Register*

Este gerador, na realidade, pode ser considerado uma modificação do gerador do tipo *generalized feedback shift register*, sendo capaz de aumentar o período máximo da seqüência gerada, além de possibilitar a obtenção de boas seqüências aleatórias (L'ECUYER, 1998) (L'ECUYER; PANNETON, 2000).

O algoritmo é baseado na seguinte relação de recorrência (MATSUMOTO; NISHIMURA, 1998):

$$x_i = x_{i-r} \oplus (x^u_{i-k} | x^l_{i-k+1}) * A, \quad i = k, k+1, \ldots$$

A variável x representa um inteiro por meio de um vetor contendo w valores booleanos. As constantes r e k são inteiras, k é chamado grau de recorrência, sendo que $0 \leq r \leq k-1$. A matriz A possui $w \times w$ constantes booleanas.

As variáveis iniciais $x_0, x_1, \ldots, x_{k-1}$ são as sementes do gerador. O cálculo do número pseudo-aleatório x_k ocorre por meio da fórmula de recorrência acima para $i = k$. Fazendo $i = k+1, k+2, \ldots$, o gerador encontra os números subsequentes $(x_{k+1}, x_{k+2}, \ldots)$.

A simbologia $(x^u_{i-k} | x^l_{i-k+1})$ indica uma concatenação de valores booleanos presentes nas variáveis x_{i-k} e x_{i-k+1}. Assim, x^u_{i-k} representa um vetor booleano contendo os $(w-c)$

valores mais significativos de x_{i-k}, sendo c uma constante inteira, $0 \leq c \leq w-1$. A variável x^l_{i-k+1} representa um vetor booleano contendo os c valores menos significativos de x_{i-k+1}. Exemplificando, se $x = (x_{w-1}, x_{w-2}, \ldots, x_0)$ então $x^u = (x_{w-1}, x_{w-2}, \ldots, x_c)$ e $x^l = (x_{c-1}, \ldots, x_0)$.

A expressão $(x^u_{i-k}|x^l_{i-k+1})$ é simplesmente a concatenação entre os $(w-c)$ valores mais significativos de x_{i-k} e os c valores menos significativos de x_{i-k+1}. Este vetor concatenado é então multiplicado pela matriz A e o resultado é utilizado para a realização da operação ou-exclusivo com a variável x_{i-r}. O resultado de todo este procedimento gera o número pseudo-aleatório x_i.

Dependendo da escolha de cada elemento da matriz A, pode-se simplificar a operação de multiplicação entre a matriz e o vetor concatenado, utilizando-se apenas operações de deslocamento e ou-exclusivo (MATSUMOTO; NISHIMURA, 1998). Se os parâmetros constantes do gerador forem escolhidos corretamente, o período máximo pode chegar a $2^{k*w-c}-1$ números (MATSUMOTO; NISHIMURA, 1998).

Se $c = 0$, a equação de recorrência representará o gerador do tipo *Twisted Generalized Feedback Shift Register* (MATSUMOTO; KURITA, 1994) e se $c = 0$ e $A = I$(matriz identidade), a equação indicará um gerador do tipo *Generalized Feedback Shift Register*.

Algumas transformações lineares, como operações lógicas e de deslocamento, podem ser utilizadas sobre cada número gerado pela equação de recorrência para melhorar as propriedades de equidistribuição e aleatoriedade da seqüência (MATSUMOTO; NISHIMURA, 1998) (L'ECUYER; PANNETON, 2000).

4.9 Geradores Combinados

A combinação de dois ou mais geradores pode aumentar o tamanho do período e melhorar as características de aleatoriedade da seqüência gerada. Contudo, os geradores combinados costumam ser computacionalmente bem mais lentos do que os geradores sem combinação (SCHOLLMEYER; TRANTER, 1991) (CODDINGTON, 1997).

Como exemplo de combinação de geradores, pode-se citar o gerador denominado COMBO, o qual realiza a seguinte equação (MARSAGLIA, 1985):

$$W_i = (x_i - y_i) \mod 2^{32}$$

sendo $x_i = (x_{i-1} * x_{i-2}) \mod 2^{32}$ e $y_i = (y_{i-3} - x_{i-1}) \mod (2^{30} - 35)$.

O método proposto por *Wichmann–Hill* considera a combinação de J geradores lineares congruentes multiplicativos com diferentes módulos $M^{(1)}$, $M^{(2)}$, ..., $M^{(J)}$ e, sendo $x_i^{(j)}$ o número pseudo-aleatório gerado na iteração i do gerador j, o método combina esses geradores da seguinte forma (SAKAMOTO; MORITO, 1995):

$$W_i = (\sum_{j=1}^{J} \delta^{(j)} * x_i^{(j)}/M^{(j)}) \mod 1$$

sendo δ um inteiro diferente de zero e o maior divisor comum entre ($M^{(j)}$ e $|\delta^{(j)}|$) sendo igual a 1. A seqüência W_i gerada se encontra no intervalo (0, 1).

L'Ecuyer propôs a seguinte combinação de geradores lineares congruentes multiplicativos (L'ECUYER, 1997) (L'ECUYER, 1988):

$$W_i = (\sum_{j=1}^{J} \delta^{(j)} * x_i^{(j)}) \mod M_{max}$$

sendo δ um inteiro diferente de zero e o maior divisor comum entre ($M^{(j)}$ e $|\delta^{(j)}|$) sendo igual a 1. O módulo M_{max} indica o maior módulo dentre todos os geradores utilizados na combinação. A seqüência W_i gerada se encontra no intervalo (0, $M_{max} - 1$). Para reduzir o custo computacional no cálculo de cada gerador, foi proposta a utilização de geradores com multiplicadores da forma $a = \pm 2^q \pm 2^r$, com isso, a multiplicação pode ser implementada por meio de operações de deslocamentos e somas/subtrações (L'ECUYER; TOUZIN, 2000).

A combinação de geradores do tipo *Generalized Feedback Shift Register* pode ser estabelecida de maneira semelhante utilizando, contudo, a operação ou-exclusivo entre as saídas de cada gerador (L'ECUYER, 1997) (MARSAGLIA, 1985).

Outro método de combinação utiliza dois geradores e uma tabela contendo uma determinada quantidade de números pseudo-aleatórios (THESEN; SUN; WANG, 1984) (MACLAREN; MARSAGLIA, 1965). Quando for necessária a obtenção de um número pseudo-aleatório, deve-se gerar um índice pseudo-aleatório por meio de um dos geradores e o número pseudo-aleatório armazenado na posição correspondente da tabela é utilizado como resultado. Depois disso, o número armazenado nesta posição é substituito por um novo número pseudo-aleatório, gerado por meio do segundo gerador. Uma variação deste método utiliza a tabela de números e apenas um gerador para a obtenção de reais pseudo-aleatórios, o índice da tabela a ser utilizado é determinado pela seguinte fórmula: $\lceil W * n \rceil$, na qual n representa o tamanho da tabela e W indica o último número gerado (BAYS;

DURHAM, 1976).

4.10 Geradores Disponíveis

A seguir apresenta-se uma lista contendo alguns geradores amplamente utilizados na simulação e implementação de diversos tipos de sistemas. Nesta lista, o gerador linear congruente (GLG), o qual possui a relação de recorrência $x_{i+1} = (a * x_i + c) \mod M$, é expresso, simbolicamente, por GLG(M, a, c, x_0), com x_0 sendo a semente do gerador (ENTACHER, 1997) (ENTACHER, 1998):

- Função rand – ANSI C: GLG(2^{31}, 1103515245, 12345, 12345)
- Computadores IBM: GLG(2^{32}, 69069, 0, 1) ou GLG(2^{32}, 69069, 1, 0)
- *Software* Matemático DERIVE: GLG(2^{32}, 3141592653, 1, 0)
- Computadores APPLE: GLG(2^{35}, 5^{13}, 0, 1)
- MAPLE: GLG($10^{12} - 11$, 427419669081, 0, 1)
- Função drand48 – ANSI C: GLG(2^{48}, 25214903917, 11, 0)
- Computadores CRAY e biblioteca no PASCAL: GLG(2^{48}, 44485709377909, 0, 1)
- RANDU – Pacote de subrotinas científicas da IBM: GLG(2^{31}, 65539, 0, 1)
- NAG – Biblioteca FORTRAN e C: GLG(2^{59}, 13^{13}, 0, $123456789 * (2^{32} + 1)$)

A obtenção de seqüências de números pseudo-aleatórios com boas propriedades de aleatoriedade, uniformidade e períodos longos dependem não só do tipo de gerador selecionado como também dos parâmetros escolhidos. Muitos geradores implementados em bibliotecas e amplamente utilizados, como RANDU, RAND e RANDOM não apresentam boas propriedades de aleatoriedade e uniformidade em diversas aplicações e, portanto, não devem ser utilizados nestes casos (PARK; MILLER, 1988) (ENTACHER, 1998). Por isso, deve-se tomar muito cuidado na utilização de geradores que estão disponíveis em bibliotecas e, dependendo do caso, deve-se construir um gerador pseudo-aleatório específico que atenda as necessidades da aplicação a ser simulada ou implementada.

4.11 Comentários

Este capítulo apresentou as características e as fórmulas matemáticas de diversos tipos de geradores lineares extraídos da literatura científica para a produção de números pseudo-aleatórios.

No próximo capítulo apresentam-se a arquitetura proposta e os seus elementos constituintes que permitem a implementação física de sistemas modelados em Redes de Petri.

5 *Arquitetura Proposta para Implementar Fisicamente Sistemas Modelados em Redes de Petri*

Resumo

A arquitetura é composta por um arranjo de blocos de configuração denominados BCERPs, por blocos reconfiguráveis denominados BCGNs e por um sistema de comunicação, implementado por um conjunto de roteadores. Os blocos BCERPs podem ser configurados para implementar as transições da Rede de Petri e seus respectivos lugares de entrada. Os blocos BCERPs se comunicam entre si para a realização de sincronismo de tempo e de conflito e para a realização do disparo das transições. Blocos BCGNs são utilizados pelos blocos BCERPs para a geração de números pseudo-aleatórios. Estes números podem ser usados no processo de resolução de conflito, que ocorre quando uma transição tiver lugares de entrada compartilhados com outras transições. O sistema de comunicação possui uma topologia de grelha, tendo como principal função o roteamento e armazenamento de pacotes entre os blocos de configuração.

5.1 Introdução

A implementação de sistemas modelados em Redes de Petri pode ser subdividida em realizações indiretas e diretas (GOMES, 1999). As realizações indiretas têm por base a tradução da Rede de Petri numa representação intermediária, como por exemplo um grafo de estados, o qual será posteriormente sintetizado em equações lógicas (GOMES, 1999). As realizações diretas baseiam-se numa tradução, onde os elementos da rede, isto é, estados e ações, são implementados por meio de componentes digitais pré-definidos. As linguagens mais comuns que descrevem esses componentes digitais e que permitem um refinamento

no nível RTL e de portas lógicas são VERILOG e VHDL. Para a implementação física do sistema modelado em Redes de Petri tem-se utilizado em larga escala FPGAs.

A implementação de uma Rede de Petri em *hardware* reduz significativamente o tempo de processamento se comparado ao de uma implementação em processadores seqüênciais convencionais devido ao alto grau de paralelismo existente na avaliação da habilitação de todas as ações da rede, na seleção de uma ação para disparar, e na determinação do próximo estado da rede quando ocorrer o disparo da ação selecionada. Como comentado no capítulo 2, a arquitetura Achilles (MORRIS; BUNDELL; THAM, 2000), permite a implementação de modelos em Redes de Petri lugar/transição utilizando uma placa eletrônica capaz de integrar vários FPGAs em pilhas, possibilitando a implementação de um número elevado de elementos da rede.

Outra forma de implementação de Redes de Petri em *hardware*, comentada no capítulo 2, refere-se a um sistema multiprocessado que pode ser programado por meio de uma descrição em Redes de Petri (ANZAI et al., 1993) (KAMAKURA et al., 1997). O controlador utilizado pelo sistema multiprocessado realiza a ativação e desativação de processos paralelos. Neste caso, a Rede de Petri é armazenada no controlador em forma de tabelas, que indicam a estrutura e a dinâmica da rede. A cada lugar da Rede de Petri atribui-se um determinado trabalho, o qual deve ser realizado pelos processos paralelos quando houver a criação de uma nova marca.

Como comentado no capítulo 2, tem ocorrido esforços no sentido de se implementar a estrutura da Rede de Petri em *hardware* para acelerar o processo de análise e para a geração do grafo de alcançabilidade da rede implementada (CSERTÁN et al., 1997).

Dentro deste contexto, a arquitetura que está sendo proposta apresenta algumas inovações em relação às arquiteturas mencionadas anteriormente. Uma das inovações é a característica já comentada no capítulo 1 sobre a possibilidade de um mapeamento tecnológico no nível sistêmico. A arquitetura possui blocos lógicos exclusivamente desenvolvidos para a implementação dos estados (lugares) e das ações (transições), o que permite que a Rede de Petri seja diretamente mapeada na arquitetura, sem haver a necessidade de se utilizar um processo de síntese de alto nível para descrever o sistema por meio de equações booleanas e tabelas de transição de estados, o que ocorre atualmente para a implementação de Redes de Petri em FPGAs.

Além disso, as arquiteturas atualmente disponíveis são capazes de implementar Redes de Petri de baixo nível (Redes de Petri ordinárias e generalizadas), como as redes lugar/transição (REISIG, 1992) e elementares (THIAGARAJAN, 1987). A arquitetura pro-

posta permite o mapeamento direto de Redes de Petri coloridas de arcos constantes (REISIG, 1992) (JENSEN, 1997). A vantagem dessas redes em relação às redes de baixo nível refere-se a uma maior facilidade gráfica para a modelagem do sistema.

Outra inovação da arquitetura proposta refere-se à possibilidade de mapeamento direto de algumas extensões de Redes de Petri, permitindo ao projetista a implementação de Redes de Petri com um poder de modelagem maior do que o atualmente disponível. A arquitetura permite o mapeamento de Redes de Petri T-temporizadas (WANG, 1998), autônomas, sincronizadas (DAVID; ALLA, 1992) e com probabilidade de disparo entre as transições. Além disso, a arquitetura poderá futuramente ser adaptada para implementar Redes de Petri temporais (WANG, 1998) e até mesmo redes estocásticas (MARSAN, 1990) (WANG, 1998), aumentado em muito o poder de modelagem, visto que tais extensões não podem ser convertidas em redes de baixo nível atemporais.

As arquiteturas atualmente disponíveis têm, normalmente, o propósito de implementar as Redes de Petri para a realização de análises comportamentais e estruturais das redes. Assim, as transições implementadas são disparadas uma de cada vez ou apenas um conjunto pré-estabelecido é disparado. Em contrapartida, a arquitetura proposta neste trabalho permite a implementação da Rede de Petri em sua forma mais original, ou seja, todas as transições habilitadas e sem conflito podem ser disparadas simultaneamente e com relação às transições em conflito, a arquitetura projetada é capaz de escolher aleatoriamente as transições que serão disparadas, por meio de um algoritmo distribuído.

Na seqüência, especifica-se esta arquitetura, detalhando os seus elementos constituintes: os roteadores e os blocos de configuração BCGNs e BCERPs. Comenta-se também sobre a possibilidade de extensão da topologia 2-D da arquitetura para uma estrutura 3-D.

5.2 A Arquitetura Proposta

A arquitetura é composta por um arranjo de blocos de configuração denominados BCERPs, cada um dos quais representando, na Rede de Petri a ser implementada, o comportamento de uma transição e os seus lugares de entrada, por unidades reconfiguráveis de geração de números pseudo-aleatórios representadas pelos blocos de configuração BCGNs e por um sistema de comunicação, implementado por um conjunto de roteadores. Na figura 5.1 apresenta-se um diagrama de blocos da arquitetura proposta, onde R indica um **R**oteador, BCERP significa **B**loco básico de **C**onfiguração dos **E**lementos de uma **R**ede

de **P**etri e BCGN indica um **B**loco de **C**onfiguração do **G**erador de **N**úmeros pseudo-aleatórios.

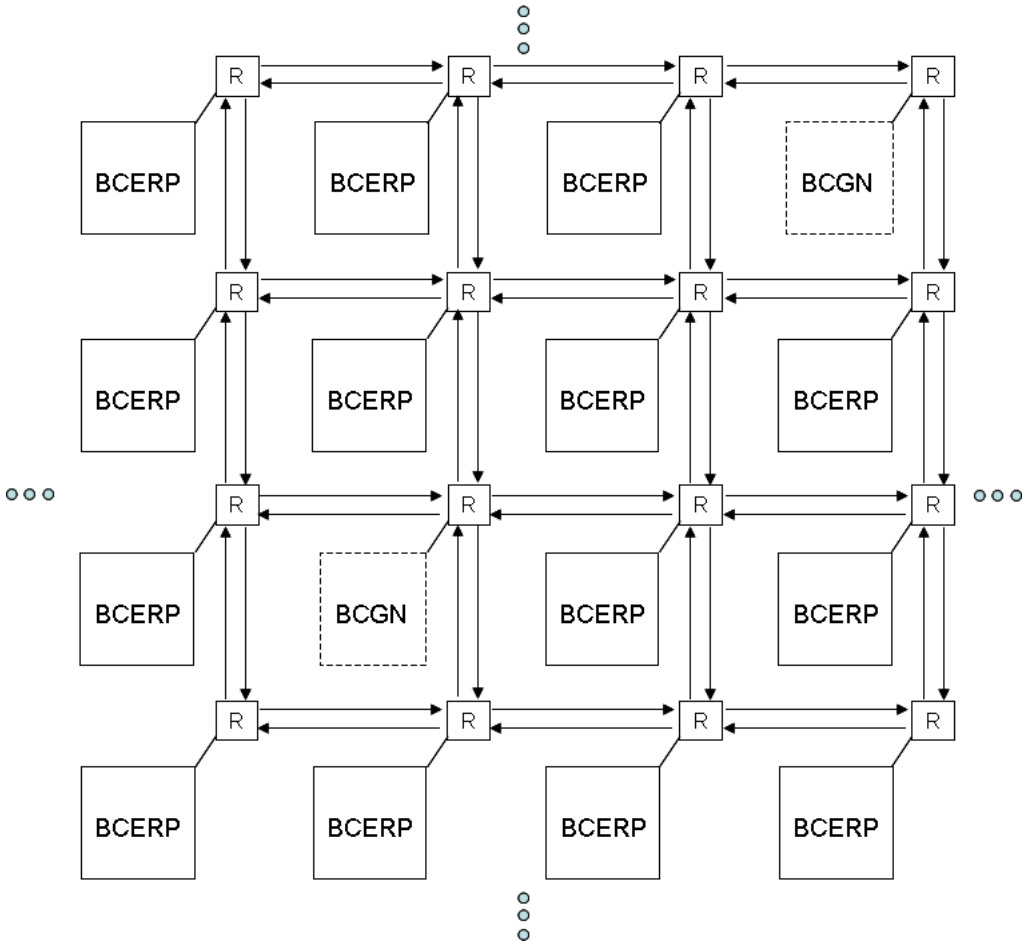


Figura 5.1: Arquitetura proposta, onde R indica um **R**oteador, BCGN significa **B**loco de **C**onfiguração do **G**erador de **N**úmeros pseudo-aleatórios e BCERP indica um **B**loco básico de **C**onfiguração dos **E**lementos de uma **R**ede de **P**etri

Cada transição da Rede de Petri e seus respectivos lugares de entrada podem ser mapeados em um ou mais BCERPs e o sistema de comunicação é usado para conectar os blocos de configuração BCERPs e BCGNs e permitir a troca de informação entre eles. As unidades de geração de números pseudo-aleatórios são utilizadas pelos blocos de configuração BCERPs para resolverem problemas de conflito, ou seja, os números pseudo-aleatórios gerados são utilizados para determinar de forma pseudo-aleatória as transições em conflito comportamental que podem ser disparas simultaneamente. Além disso, os números pseudo-aleatórios também são utilizados para permitir a implementação em *hardware* de modelos descritos em Redes de Petri com associação de probabilidade de disparo entre as transições.

O sistema de comunicação é composto por um grupo de roteadores distribuídos em

uma topologia de malha bidimensional. Desta forma, o sistema pode ser configurado para imitar a estrutura da Rede de Petri. Uma vez configurado, os BCERPS trabalham concorrentemente e geram marcas e informações de controle e sincronismo que devem ser transportados pelo sistema de comunicação. Cada BCERP possui um grupo de registradores responsável pelo armazenamento das marcas que são geradas quando determinadas transições da Rede de Petri disparar. Um comparador de marcas é usado para ler simultaneamente o conteúdo de todos os registradores e indicar a presença (ou ausência) de marcas suficientes para o disparo da transição correspondente.

A utilização de um sistema de comunicação composto por roteadores ao invés de um barramento com estruturas reconfiguráveis (MANGIONE-SMITH, 1997) (BHATIA, 1997) se deve a três motivos, discutidos nos parágrafos a seguir.

Primeiro, pode-se ganhar tempo no processo de mapeamento da Rede de Petri na arquitetura, visto que não há a necessidade de se encontrar caminhos disponíveis entre os blocos de configuração para realizar a programação do barramento.

Segundo, os algoritmos utilizados para encontrar os caminhos de comunicação nos barramentos programáveis podem não convergir e tornar inviável o processo de mapeamento dos elementos. No caso proposto, o sistema permite a comunicação entre um bloco de configuração e qualquer outro, o que é muito interessante para o mapeamento de Redes de Petri que possuem muitos arcos de entrada e saída.

Terceiro, em barramentos reconfiguráveis é necessário a utilização de sinais de controle para realizar a configuração do barramento. No sistema composto por roteadores não há a necessidade da implementação desses sinais, visto que os próprios roteadores possuem capacidade de processamento suficiente para realizar o transporte de informação de um bloco de configuração ao outro.

Na seqüência, comentam-se sobre a topologia adotada e os elementos constituintes da arquitetura: roteadores, BCGNs e BCERPs. Por fim, mostra-se como estender a arquitetura proposta para a formação de uma estrutura 3-D.

5.2.1 Topologia

O conjunto de roteadores possui uma topologia de grelha (malha bidimensional). Existem outras topologias, como as citadas no capítulo 3, utilizadas em redes-em-*chip* e redes de computadores que são destinadas ao processamento paralelo. A escolha de uma malha bidimensional se deve ao fato de que as estruturas 2-D, como a grelha, são

mais adequadas às tecnologias atuais de fabricação (ZEFERINO et al., 2002) (KUMAR et al., 2002).

Além disso, o procedimento de roteamento em duas dimensões numa topologia de malha é de desenvolvimento mais fácil, o que resulta em roteadores com baixo custo de implementação, boa capacidade computacional, ciclos de relógio curtos e totalmente escaláveis (KUMAR et al., 2002).

5.2.2 Roteador

Na arquitetura proposta, o roteador tem como principal função o roteamento e armazenamento de pacotes entre os blocos de configuração. Cada roteador é conectado com outros quatro roteadores vizinhos, através dos canais de comunicação de entrada e saída e, a um bloco de configuração BCERP ou BCGN.

Um canal de comunicação consiste de dois barramentos unidirecionais ponto-a-ponto entre dois roteadores, ou entre um roteador e um bloco de configuração. O roteador é constituído por um registrador em cada canal de saída para armazenar pacotes, multiplexadores para efetivar as interconexões necessárias para a transferência de um pacote para outro roteador ou para um bloco de configuração nele acoplado, além de possuir controladores que implementam os mecanismos de comunicação.

O armazenamento de pacotes nos canais de saída, ao invés da utilização de filas na entrada, elimina o problema de *head-of-line*, comentado no capítulo 3. Assim, os pacotes que chegam nas portas de entrada são primeiramente roteados e depois armazenados nas filas de saída.

O modelo de comunicação utilizado é o de troca de mensagens, sendo que a comunicação entre os blocos de configuração é feita pelo envio e recebimento de mensagens de requisição e de resposta, sendo que uma mensagem deve ser transferida por meio de uma seqüência de pacotes.

O pacote enviado pelos roteadores da arquitetura proposta é subdividido em apenas dois campos, quais sejam: cabeçalho e carga útil. O cabeçalho inclui as variações nos eixos x e y (topologia de grelha), e as direções leste-oeste ou oeste-leste, e norte-sul ou sul-norte. A carga útil é efetivamente a informação que deve ser entregue a um bloco de configuração. As variações nos eixos x e y constituem o endereço de destino do pacote e as direções indicam a trajetória.

Na figura 5.2 apresenta-se a composição de um pacote. Cada pacote contém 32 bits, dos quais 20 são destinados à carga útil e 12 ao endereçamento do pacote: 5 para a variação no eixo x, 5 para o eixo y e os outros dois bits restantes indicam o sentido que um pacote deverá caminhar (norte-sul ou sul-norte e oeste-leste ou leste-oeste). Os 5 bits atribuídos para a variação em cada um dos eixos define um endereçamento de até $2^5 = 32$ roteadores por eixo, totalizando $32 * 32 = 1024$ roteadores em toda a arquitetura.

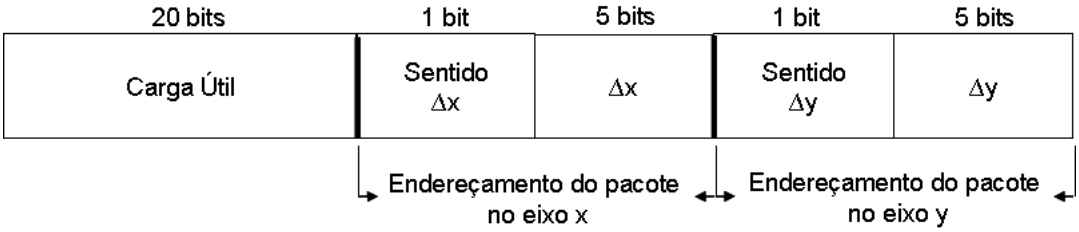


Figura 5.2: Composição de um pacote

No capítulo 6 comenta-se com maiores detalhes o projeto do sistema de comunicação da arquitetura proposta.

5.2.3 Bloco de Configuração BCGN

Muitas aplicações exigem a construção de um gerador pseudo-aleatório específico que atenda certas necessidades de uniformidade e aleatoriedade. Um gerador padrão não reconfigurável pode se comportar muito bem para determinadas aplicações, mas muito mal em outras. Por isso, o bloco BCGN desenvolvido pode ser configurado para executar um gerador linear congruente multiplicativo, um gerador linear congruente *mixed*, um gerador linear múltiplo convencional com ordem de recorrência igual a dois, um gerador linear múltiplo *mixed* ou um gerador linear com transporte do tipo multiplicação-com-transporte, explicados no capítulo 4. Desta forma, dependendo do sistema que está sendo implementado na arquitetura proposta pode-se configurar um gerador de números que atenda especificamente as necessidades deste sistema.

Basicamente, no processo de execução, o bloco BCGN recebe um pacote de requisição proveniente de um bloco BCERP. Ao receber este pacote de requisição, o bloco BCGN produz um número pseudo-aleatório de acordo com a sua configuração. O número gerado é enviado para o bloco BCERP que solicitou a geração do número, por meio do sistema de roteamento implementado.

No processo de configuração, sete pacotes de configuração devem ser enviados para a programação do bloco BCGN. Ao receber um pacote de configuração, o bloco BCGN

armazena determinados dados provenientes desses pacotes em registradores, os quais são utilizados para realizar o algoritmo de geração de números pseudo-aleatórios adotado.

No capítulo 7 comenta-se com maiores detalhes o projeto do bloco de configuração BCGN da arquitetura proposta.

5.2.4 Bloco de Configuração BCERP

Na sua forma padrão, cada bloco BCERP pode implementar uma transição da Rede de Petri e os seus lugares de entrada. Basicamente, o bloco BCERP possui um banco de memória contendo os pacotes de dados que devem ser enviados se a transição implementanda for disparada. Nesses pacotes de dados encontram-se a identificação da marca e a quantidade que deve ser adicionada nos respectivos lugares de saída da transição. Dessa forma, no processo de disparo da transição, os pacotes de dados da memória são enviados para blocos BCERPs que armazenam os lugares de saída dessa transição. Cada bloco permite que a transição seja disparada após um determinado intervalo de tempo, assim, no disparo da transição, enviam-se também pacotes de sincronismo de tempo para blocos BCERPs que podem modificar os lugares de entrada da transição disparada.

As unidades de geração de números pseudo-aleatórios são utilizadas pelos blocos de configuração BCERPs para implementar transições com probabilidades de disparos e para a resolução de problemas de conflito, ou seja, os números gerados são utilizados para determinar de forma pseudo-aleatória as transições em conflito comportamental que podem ser disparadas simultaneamente.

Um conflito comportamental (GOMES, 1999) ocorre quando, havendo transições habilitadas, somente uma pode ser disparada, uma vez que o disparo de uma remove a marca no lugar de entrada compartilhado, desabilitando as outras transições. Na figura 5.3 apresenta-se a ocorrência de conflito entre as transições t_1 e t_2, pois possuem um lugar de entrada em comum. O disparo de qualquer uma retira a marca do lugar de entrada p, desabilitando a outra transição.

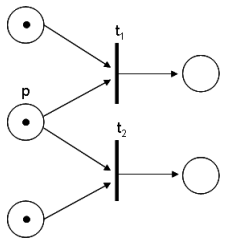


Figura 5.3: Situação de conflito numa Rede de Petri

A arquitetura projetada é capaz de evitar que duas ou mais transições residentes em blocos BCERPs diferentes e que estejam em situação de conflito comportamental, sejam disparadas simultaneamente, o que causaria uma incoerência na execução da Rede de Petri implementada.

5.2.5 Semântica da Rede de Petri Implementada

Em conjunto, os blocos BCERPs são capazes de implementar uma Rede de Petri T-temporizada (WANG, 1998) com diferenciação entre as marcas (rede colorida). Uma Rede de Petri Colorida de arcos constantes é constituída por (REISIG, 1992) (JENSEN, 1997):

- lugares, transições e arcos, semelhante a uma Rede de Petri lugar/transição;
- tipos ou cores que individualizam as marcas da rede;
- uma marcação inicial que identifica, para cada lugar, as quantidades e os tipos de marcas disponíveis;
- um rótulo em cada arco indicando as quantidades e os tipos de marcas que devem ser adicionados ou removidos em cada lugar no disparo das transições.

Uma transição da Rede de Petri Colorida de arcos constantes está habilitada quando, para cada lugar de entrada da transição, as quantidades de marcas de cada tipo forem maiores ou iguais às quantidades definidas nos arcos de entrada, como mostrado no exemplo da figura 5.4. Neste exemplo, a transição T1 está habilitada para disparo visto que os lugares L1 e L2 possuem marcas suficientes para o disparo, o lugar L1 possui duas marcas do tipo e e o lugar L2 possui uma marca do tipo f e duas marcas do tipo e.

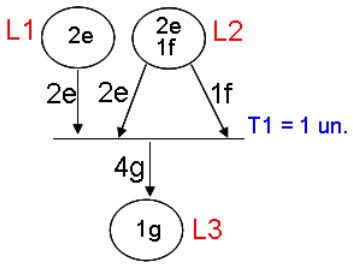


Figura 5.4: Transição habilitada para o disparo

No disparo da transição, o seguinte procedimento deve ocorrer: em cada lugar de entrada da transição deve-se subtrair, de acordo com o rótulo definido nos arcos de entrada, as quantidades de marcas de cada tipo. Em cada lugar de saída, deve-se adicionar

as quantidades de marcas de cada tipo, como definido no rótulo dos arcos de saída da transição. No exemplo da figura 5.4, o disparo da transição T1 fará com que duas marcas do tipo e do lugar L1, uma marca do tipo *f* do lugar L2 e duas marcas do tipo *e* do lugar L2 sejam removidas. No lugar L3 devem ser produzidas quatro marcas do tipo *g*, como especificado no rótulo do arco de saída da transição (L3 = 4 *g* + 1 *g* = 5 *g*).

A temporização é realizada na transição, desta forma, quando uma transição disparar, ela imediatamente retira, dos seus respectivos lugares de entrada, as marcas nas quantidades e qualidades definidas nos arcos de entrada. Após o tempo lógico definido ter se esgotado, a transição irá produzir, nos lugares de saída, as marcas nas quantidades e qualidades definidas nos arcos de saída. A Rede de Petri implementada não realiza múltiplos disparos de uma mesma transição num único tempo lógico. Além disso, cada lugar da Rede de Petri não pode ter mais do que 2^{15} marcas, devido ao limite de armazenamento dos registradores internos da arquitetura. No exemplo da figura 5.4, a inscrição T1= 1un. indica que a transição T1 só poderá ser disparada após transcorrida uma unidade de tempo.

Cada transição da Rede de Petri pode ter uma determinada probabilidade de disparo. Esta probabilidade indica, ao longo do tempo, a quantidade de vezes que uma transição poderá ser disparada. Desta forma, nem toda vez que uma transição estiver habilitada, ela será disparada.

Na modelagem da rede é permitido a construção de modelos com concorrência, conflito e confusão entre as transições. Contudo, quando duas ou mais transições estiverem em situação de conflito comportamental, uma escolha pseudo-aleatória identificará as transições que poderão ser disparadas. Entenda-se por conflito comportamental o conflito que ocorre entre duas ou mais transições durante a execução da Rede de Petri.

Como a arquitetura foi projetada para a implementação física de modelos para controle (e não para avaliação das propriedades da rede), todas as transições que estiverem habilitadas poderão disparar simultaneamente, salvo quando ocorrer um conflito comportamental.

Na figura 5.5 mostra-se um exemplo de uma Rede de Petri colorida de arcos constantes com temporização na transição (T-temporizada). Este exemplo de Rede de Petri pode ser mapeado ou implementado nos blocos BCERPs da arquitetura proposta como mostrado na figura 5.6. A Rede de Petri possui oito lugares e seis transições, sendo mapeada em seis BCERPs, um para cada transição da rede.

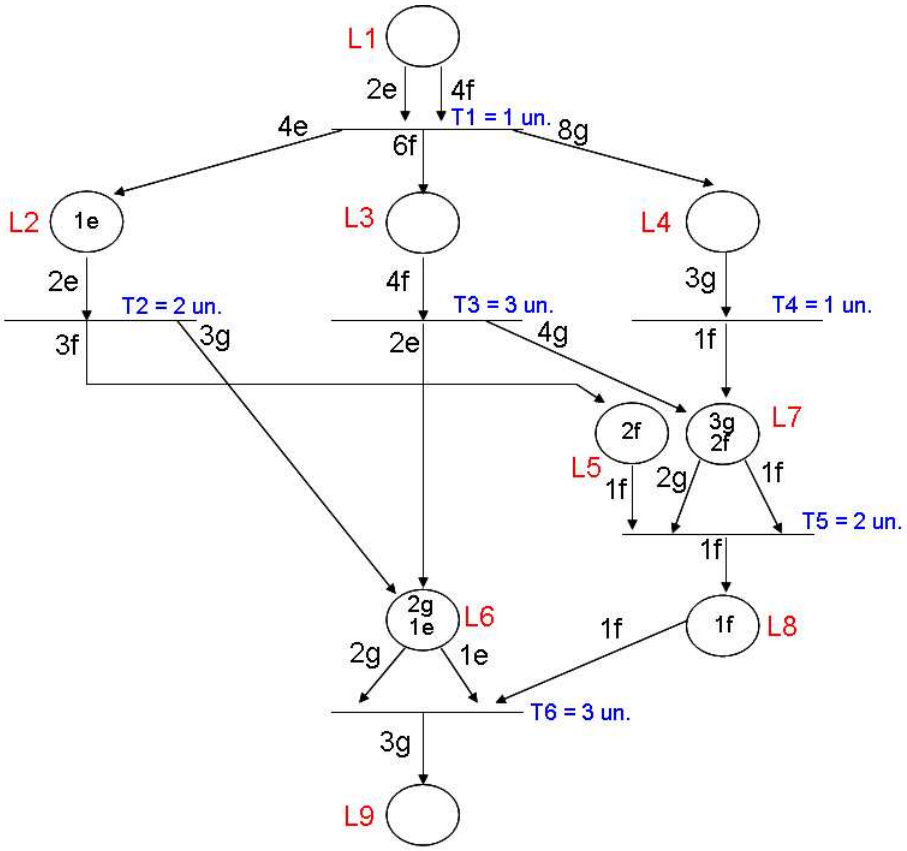


Figura 5.5: Exemplo de um modelo em uma Rede de Petri colorida de arcos constantes T-temporizada

O sistema de roteamento da arquitetura proposta trabalha tanto com pacotes de dados, enviados pelos blocos BCERPs e BCGNs no processo de execução da Rede de Petri implementada, como com pacotes de configuração, utilizados para a programação dos blocos de configuração. Ao receber um pacote de configuração, o bloco BCERP armazena determinados dados provenientes desses pacotes em registradores, os quais são utilizados para definir o comportamento do bloco durante o processo de execução da Rede de Petri. Como normalmente há vários pontos de entrada de dados na arquitetura, vários pacotes de configuração podem ser enviados ao mesmo tempo, distribuindo-se paralelamente o processo de programação dos blocos. Além disso, com a utilização dos canais de entrada de dados, elimina-se a necessidade de se utilizar sinais específicos para a realização desse processo de configuração.

No capítulo 8 comenta-se com maiores detalhes o projeto do bloco de configuração BCERP da arquitetura proposta.

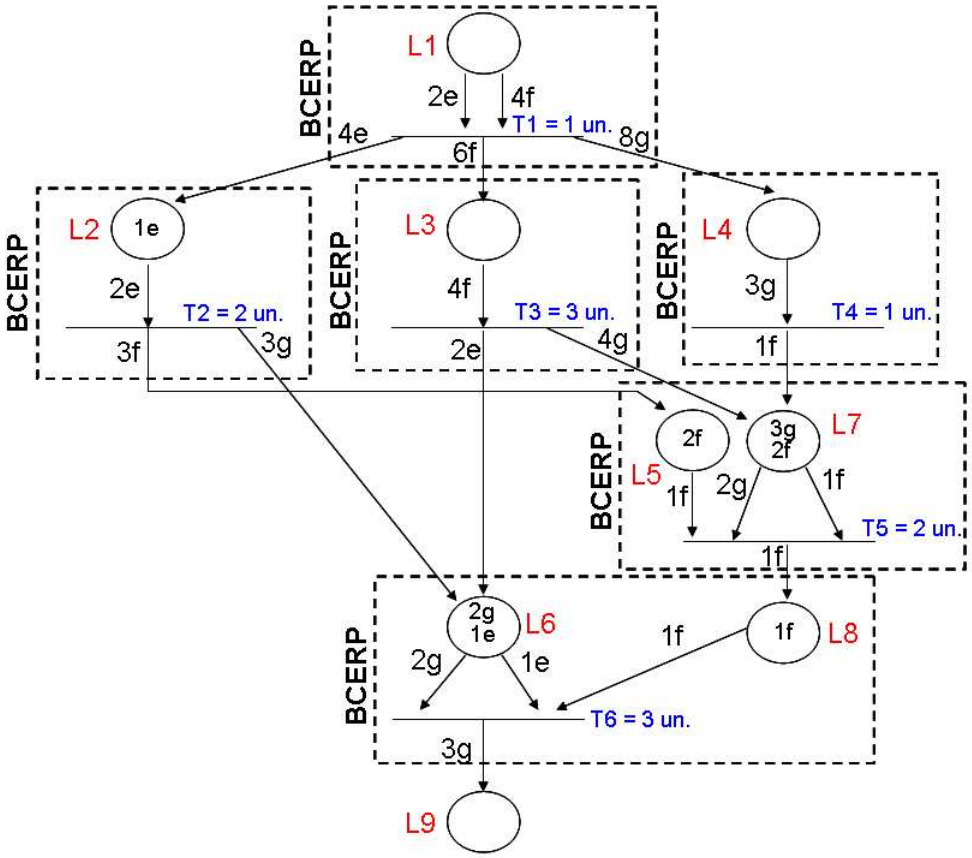


Figura 5.6: Mapeamento de uma Rede de Petri nos blocos BCERPs da arquitetura proposta

5.2.6 Estendendo a Arquitetura para uma Estrutura 3-D

A arquitetura de topologia 2-D mostrada na figura 5.1 pode ser estendida para formar uma estrutura 3-D, como pode ser observado na figura 5.7, onde várias camadas de estruturas 2-D são utilizadas em conjunto para a realização de uma topologia 3-D.

Na figura 5.7, três pontos de comunicação são utilizados entre as camadas, mas pode-se definir, durante a fase de projeto da arquitetura, uma quantidade maior de pontos de ligação para permitir um melhor desempenho quando muitos blocos de configuração necessitarem se comunicar com outros blocos situados em camadas diferentes, evitando assim, sobrecarregar os roteadores de camadas.

Uma modificação que ocorre em cada camada, se comparada à arquitetura mostrada na figura 5.1, refere-se ao sistema de comunicação que passa a ter dois roteadores diferentes, um com cinco canais de comunicação, figura 5.8(a), e outro com seis, figura 5.8(b).

O roteador com cinco canais de comunicação é semelhante ao roteador da figura 5.1,

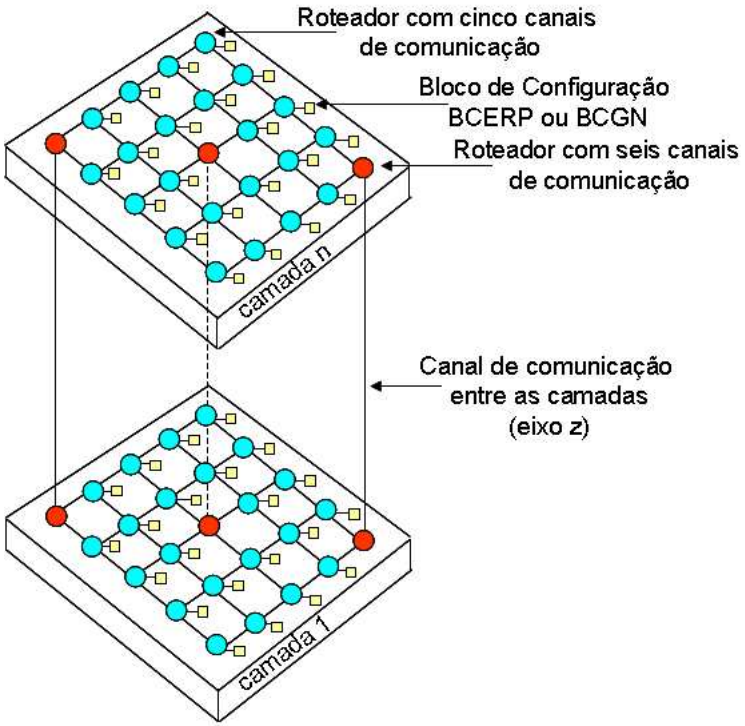


Figura 5.7: Arquitetura numa topologia 3-D

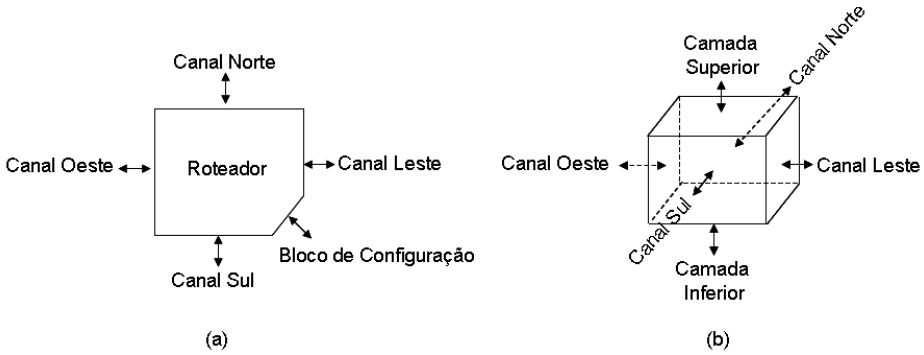


Figura 5.8: Interface de entrada/saída dos roteadores (a) com cinco canais e (b) com seis canais

sendo responsável apenas pelo roteamento de pacotes dentro de sua própria camada. Quatro canais são responsáveis pela comunicação com os roteadores vizinhos (norte, sul, leste e oeste) e, um canal é responsável pela comunicação com o bloco de configuração que se encontra acoplado ao roteador.

A diferença ocorre na existência de um sinal de controle, o qual deve ser enviado juntamente com o primeiro pacote, indicando que um segundo pacote será também enviado. Este sinal de controle indicará que o canal responsável pelo recebimento do primeiro pacote deverá ter a maior prioridade. Com isso, após o primeiro pacote ser encaminhado, o segundo pacote será processado imediatamente.

Isso se faz necessário para evitar que o roteador com seis canais receba, como segundo pacote, um pacote que não seja do bloco de configuração remetente.

O roteador com seis canais de comunicação é responsável pelo roteamento de pacotes entre as camadas. Quatro canais, como ocorre nos roteadores de cinco canais, referem-se à comunicação com os roteadores vizinhos do norte, sul, leste e oeste. Os outros dois canais referem-se à comunicação com a camada imediatamente acima e imediatamente abaixo. Neste caso, não há blocos de configuração acoplados aos roteadores.

O princípio de funcionamento para enviar um pacote de uma camada para uma outra camada qualquer é o seguinte: o bloco de configuração remetente deve enviar um primeiro pacote composto apenas do cabeçalho, ou seja, deverá informar o endereço de destino, o qual se compõe de 10 campos, como mostrado na figura 5.9.

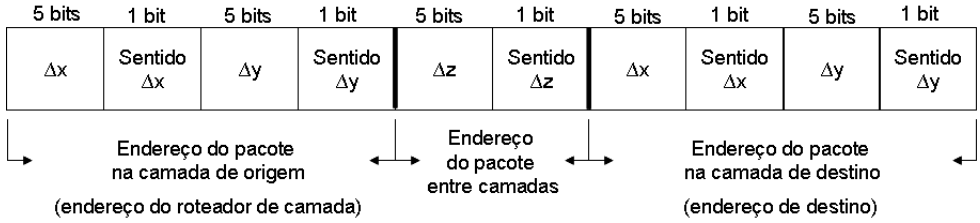


Figura 5.9: Pacote utilizado na comunicação entre camadas

O primeiro campo, `Δx_origem`, indica a variação no eixo x da camada de origem. Entende-se como camada de origem a camada em que foi injetado o pacote pelo bloco de configuração remetente, ou seja, a camada na qual o bloco de configuração remetente reside. Desta forma, este campo indica a quantidade de roteadores que o pacote deve atravessar no eixo x da camada de origem até iniciar o percurso no eixo y, caso necessário.

O campo `sentido_Δx_origem` refere-se ao sentido, leste-oeste ou oeste-leste, que o pacote deverá percorrer no eixo x da camada de origem.

O campo `Δy_origem` indica a variação no eixo x da camada de origem, ou seja, indica a quantidade de roteadores que o pacote deve atravessar no eixo y da camada de origem.

O campo `sentido_Δy_origem` refere-se ao sentido, norte-sul ou sul-norte, que o pacote deverá percorrer no eixo y da camada de origem.

O próximo campo, `Δz`, indica a quantidade de roteadores de seis canais que o pacote deverá atravessar até chegar à camada de destino. Entende-se como camada de destino a camada na qual o bloco de configuração destinatário reside.

O campo `sentido_Δz` refere-se ao sentido, camadas superiores ou inferiores, que o

pacote deverá percorrer no eixo z até atingir a camada de destino.

O campo `Δx_destino`, indica a variação no eixo x da camada de destino. Desta forma, este campo indica a quantidade de roteadores que o pacote deve atravessar no eixo x da camada de destino até iniciar o percurso no eixo y, caso necessário.

O campo `sentido_Δx_destino` refere-se ao sentido, leste-oeste ou oeste-leste, que o pacote deverá percorrer no eixo x da camada de destino.

O campo `Δy_destino` indica a variação no eixo x da camada de destino, ou seja, indica a quantidade de roteadores que o pacote deve atravessar no eixo y da camada de destino.

O campo `sentido_Δy_destino` refere-se ao sentido, norte-sul ou sul-norte, que o pacote deverá percorrer no eixo y da camada de destino.

Desta forma, os campos `Δx_origem`, `sentido_Δx_origem`, `Δy_origem` e `sentido_Δy_origem` são utilizados no processo de rotear o pacote injetado na rede de comunicação até um ponto de ligação com as camadas superiores/inferiores, ou seja, até alcançar um roteador de seis canais.

Os campos `Δz` e `sentido_Δz` são utilizados pelos roteadores de seis canais para encontrar a camada de destino do pacote.

Os campos `Δx_destino`, `sentido_Δx_destino`, `Δy_destino` e `sentido_Δy_destino` são utilizados para rotear um pacote de um roteador de seis canais para um roteador de cinco canais no qual o bloco de configuração de destino esteja acoplado.

Cada pacote utilizado na comunicação entre camadas deve ser distribuído da seguinte forma:

- 12 bits são destinados ao endereçamento na camada de origem, sendo 5 para a variação no eixo x, 5 para a variação no eixo y, 1 para o sentido no eixo x e 1 para o sentido no eixo y;
- 6 bits são referentes ao endereçamento da camada, onde 5 bits indica a variação no eixo z e 1 bit define o sentido da trajetória; e
- 12 bits são destinados ao endereçamento na camada de destino: 5 para a variação no eixo x, 5 para a variação no eixo y, 1 para o sentido no eixo x e 1 para o sentido no eixo y.

Ao todo 30 bits são utilizados no cabeçalho. Como a carga útil foi definida com 20 bits, é necessário que o bloco de configuração remetente envie um segundo pacote

contendo a carga útil. Este pacote deve também ser composto pelos campos `Δx_origem`, `sentido_Δx_origem`, `Δy_origem` e `sentido_Δy_origem` para que a carga útil possa ser transportada até o roteador de seis canais, o qual juntamente com o primeiro pacote (cabeçalho) iniciará o processo de roteamento para identificar a camada de destino.

5.3 Comentários

Neste capítulo comentou-se sobre uma arquitetura que possibilita a implementação física de sistemas modelados em Redes de Petri. A arquitetura é composta por um arranjo de blocos de configuração denominados BCERPs, cada um dos quais representando, na Rede de Petri a ser implementada, o comportamento de uma transição e dos seus lugares de entrada, por blocos de configuração para a geração de números pseudo-aleatórios (BCGNs) e por um sistema de comunicação, implementado por um conjunto de roteadores.

Nos próximos três capítulos comentam-se, com maiores detalhes, os projetos dos elementos da arquitetura proposta: roteadores, BCGNs e BCERPs.

6 *Roteador: Responsável pelo Sistema de Comunicação da Arquitetura Proposta*

Resumo

O sistema de comunicação da arquitetura proposta recebe pacotes contendo, em seu cabeçalho, alguns campos referentes ao endereço de destino do pacote (variações nos eixos x e y). Ao receber um pacote, o roteador verifica os valores armazenados nos campos de variações. Se forem zeros, o pacote chegou a seu destino e o roteador deve entregá-lo ao bloco de configuração nele conectado. Caso contrário, um campo de variação é decrementado em uma unidade e o pacote é enviado a um roteador vizinho, definido de acordo com o estabelecido no cabeçalho. O roteador descrito em linguagem VHDL e implementado em FPGA apresenta uma quantidade significativamente pequena de portas lógicas. Isso ocorre porque os modelos de Redes de Petri que podem ser implementados na arquitetura proposta realizam uma comunicação de granularidade fina, sendo que as informações necessárias para um determinado processamento são enviadas em um único pacote. Assim, a combinação do sistema de comunicação de granularidade fina com os projetos específicos dos blocos BCERPs e BCGNs permitiu a redução no controle de fluxo e na estrutura de memorização do roteador.

6.1 Introdução

O projeto do roteador para ser integrado ao sistema multiprocessado e reconfigurável de topologia 2-D especificado no capítulo 5 é apresentado neste capítulo. Três diferentes propostas de arquitetura para o roteador foram comparadas quanto à quantidade de portas lógicas necessárias à implementação e à profundidade do caminho crítico; a arquitetura do roteador que obteve o melhor resultado foi descrita em VHDL e submetida a uma série

de testes de funcionalidade.

Na seqüência, comenta-se sobre o funcionamente deste roteador, especificam-se as três arquiteturas propostas, realizando uma comparação entre elas. Depois, expõem-se alguns aspectos da implementação do roteador em VHDL, síntese, simulação e teste do código desenvolvido e finaliza-se com a extensão do roteador para permitir a realização de uma topologia 3-D.

6.2 Funcionamento do Roteador

O roteador possui cinco portas de comunicação nomeadas de Norte (N), Sul (S), Leste (L), Oeste (O) e uma para o bloco de configuração BCERP ou BCGN (BC), como mostrado na figura 6.1. Cada porta de comunicação possui três diferentes sinais, sendo: `PAP` (pedido de armazenamento de pacote), `PACA` (indica o armazenamento de pacote) e `PACOTE` (sinaliza os bits representantes do pacote que está sendo enviado).

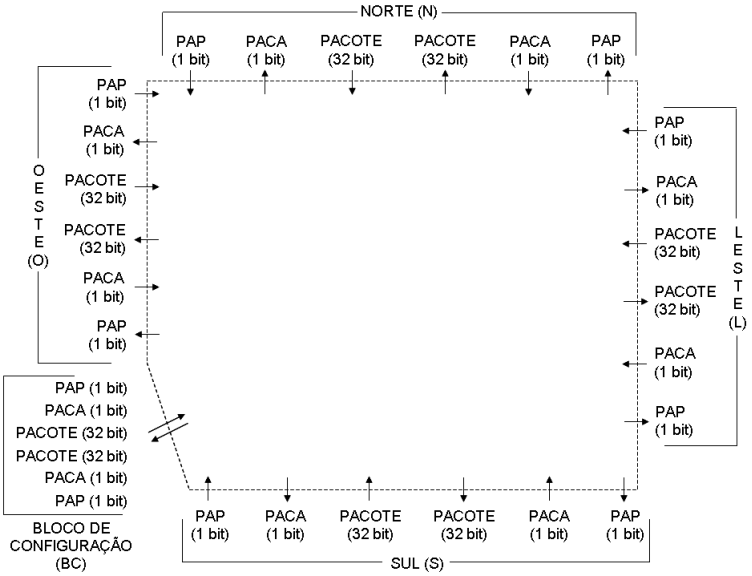


Figura 6.1: Interface de entrada/saída do roteador

A porta BC é responsável pela transferência de pacotes entre o BCERP ou o BCGN e a rede de comunicação, representada pelo conjunto de roteadores. Os blocos BCERPs e BCGNs, comentados no capítulo 5, são responsáveis, além de todo o processamento de dados do sistema, por injetar pacotes na rede e definir o seu endereço de destino de tal forma que a rede de comunicação possa se encarregar de enviar os pacotes ao destino definido.

As quatro portas restantes são responsáveis pela comunicação entre os roteadores

vizinhos, assim, a porta Norte (N) é responsável pela comunicação com o roteador situado ao Norte; por sua vez, a porta Sul (S) é responsável pela comunicação com o roteador situado ao Sul, e assim por diante. Imagine-se que um roteador situado ao Norte (N) de um outro roteador, precise enviar um pacote a este roteador. Para tanto, o roteador do Norte deve sinalizar através da porta Sul, um pedido de armazenamento de pacote, o que implicaria em elevar o nível lógico para "1"no sinal de saída `PAP` da porta Sul. Uma vez efetuado o pedido, o roteador receptor, se disponível, enviará ao roteador do Norte um sinal de pacote armazenado, ou seja, sinalizará com um nível lógico alto no sinal de saída `PACA` através da sua porta Norte. Quando o roteador do Norte receber a informação de que o pacote foi armazenado pelo roteador Sul, então o roteador do Norte tornará o nível lógico do sinal `PAP` baixo.

O pacote é composto pela carga útil, variação no eixo x e no eixo y (topologia de grelha), e pelas direções leste-oeste ou oeste-leste, e norte-sul ou sul-norte. A carga útil é efetivamente a informação que deve ser entregue ao bloco de configuração. A variação nos eixos x e y constitui o endereço de destino do pacote e as direções indicam a trajetória.

Através da técnica de roteamento X-Y, decrementa-se uma posição de Δx até que $\Delta x = 0$. Depois, decrementa-se uma posição de Δy até que $\Delta y = 0$. Quando $\Delta x = 0$ e $\Delta y = 0$ o pacote chegou a seu destino e o roteador deve entregá-lo ao bloco de configuração nele conectado retirando-se, com isso, o pacote da rede de comunicação. O roteador ao receber um pedido de armazenamento de pacote `PAP`, deverá identificar a porta em que o pacote será enviado, decrementar a variação x ou y selecionada e então sinalizar um pedido de armazenamento de pacote `PAP` na porta selecionada.

6.3 Arquitetura do Roteador

Modelos básicos de roteadores (GUERRIER; GREINER, 2000) (ZEFERINO et al., 2002) (PANDE et al., 2003) citados na literatura de Redes-em-*Chip* subdividem o pacote em um conjunto de quadros, e estes são transmitidos pela rede. Como a idéia em Redes-em-*Chip* é realizar um sistema de comunicação capaz de se adaptar a diferentes primitivas num sistema heterogêneo, tais primitivas podem enviar pacotes de tamanhos variados e assim, a subdivisão de pacotes em quadros se torna necessária. Os modelos de Redes de Petri que podem ser implementados na arquitetura proposta realizam uma comunicação de granularidade fina, ou seja, as informações necessárias para um determinado processamento são enviadas em um único pacote. Assim, combinando o sistema de comunicação

de granularidade fina com os projetos específicos dos blocos BCERPs e BCGNs, foram elaboradas, neste trabalho, três arquiteturas possíveis para o roteador.

6.3.1 Primeira Arquitetura

Na primeira arquitetura, mostrada num diagrama de blocos da figura 6.2, o roteador é capaz de escolher um dentre vários possíveis pedidos de armazenamento de pacote `PAP`, armazenar o pacote no único registrador disponível e definir o próximo trajeto do pacote. Enquanto estiver ocupado tentando enviar um pacote armazenado, o roteador não poderá atender nenhum outro pedido de armazenamento de pacote.

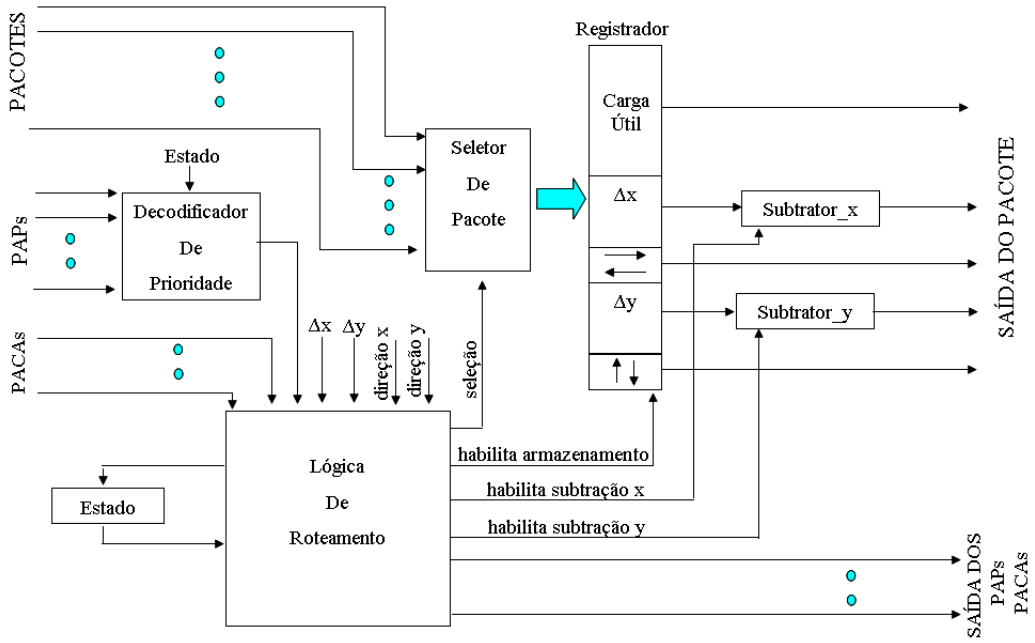


Figura 6.2: Diagrama de blocos da primeira arquitetura para o roteador

Internamente, o roteador possui um decodificador de prioridades de cinco entradas, dois decrementadores (subtratores), um seletor de pacote, um *flip-flop*, um registrador, além da lógica de roteamento.

O decodificador de prioridades é utilizado para a resolução de conflitos quando ocorrer vários pedidos de armazenamento de pacote enviados pelos roteadores vizinhos. A ordem de prioridade é variável, e neste caso, foi implementado o algoritmo *round-robin*.

O seletor de pacotes identifica qual pacote de entrada terá acesso ao registrador, que por sua vez, é utilizado para o armazenamento do pacote. O estado do roteador é armazenado num *flip-flop*. O estado "0"indica que o roteador está ocioso, possibilitando o armazenamento do pacote e o estado "1"indica que o roteador está ocupado tentando

enviar um pacote já armazenado internamente no registrador.

Os dois decrementadores são capazes de realizar as variações nos eixos x e y, indicando para o próximo roteador a receber o pacote, que o destino deste pacote está se aproximando. As variações x e y, em conjunto, constituem o endereço de destino do pacote a ser entregue.

6.3.2 Segunda Arquitetura

Na segunda arquitetura, mostrada num diagrama de blocos da figura 6.3, o roteador é capaz de realizar as mesmas funcionalidades do roteador implementado com a primeira arquitetura. Porém, ao invés de esperar o envio de um pacote armazenado para o atendimento de outros pedidos, no próximo ciclo de relógio o roteador com esta segunda arquitetura poderá atender um novo pedido de armazenamento de pacote desde que haja um canal de comunicação disponível.

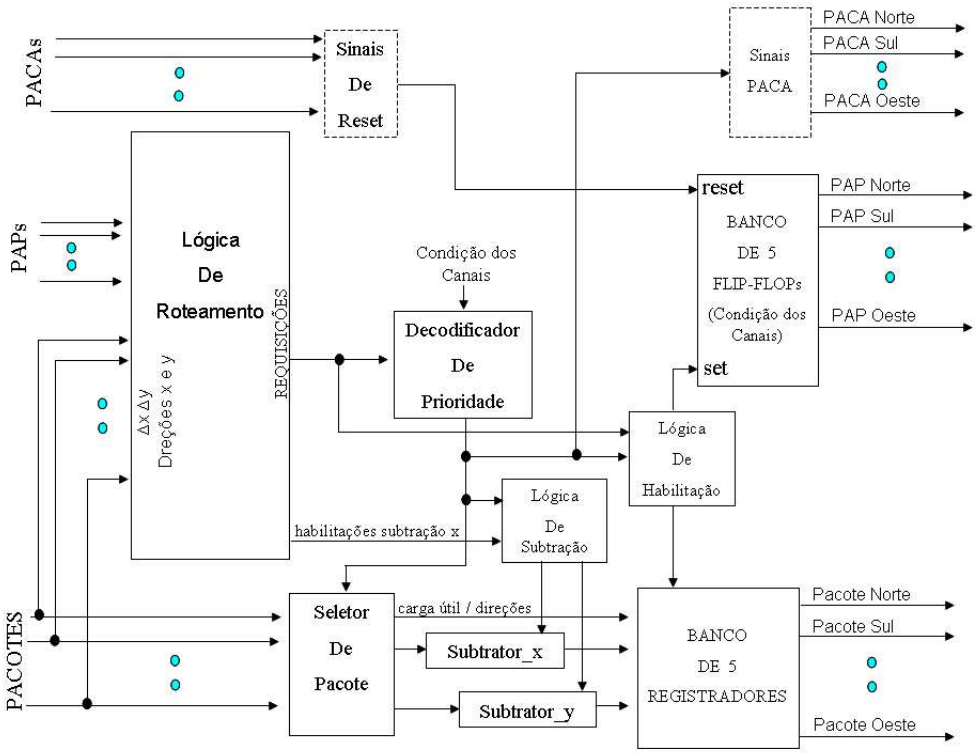


Figura 6.3: Diagrama de blocos da segunda arquitetura para o roteador

Internamente, o roteador possui um decodificador de prioridades de cinco entradas, dois decrementadores, um seletor de pacote, cinco *flip-flops*, cinco registradores, além da lógica de roteamento.

Nesta arquitetura foi implementado um decodificador dinâmico com uma ordem de

prioridade cíclica. Os cinco registradores são utilizados para o armazenamento de pacotes, enquanto os cinco *flip-flops* armazenam os sinais de saída `PAPs` para as cinco portas de comunicação. Quando o conteúdo de um *flip-flop* for "1", para uma determinada porta, o roteador estará enviando um pedido de armazenamento de pacote `PAP`, caso for "0"não há um pedido `PAP` nesta porta de comunicação. Os dois decrementadores são capazes de realizar as variaçõs nos eixos x e y.

6.3.3 Terceira Arquitetura

Na terceira arquitetura, mostrada num diagrama de blocos da figura 6.4, o roteador é capaz de realizar vários pedidos simultaneamente. Um pedido de armazenamento de pacote só não será atendido num único ciclo de relógio em dois casos: no primeiro caso, o registrador pertencente ao canal de comunicação aonde se deve armazenar o pacote está ocupado com outro pacote proveniente de um pedido atendido anteriormente; e no segundo caso, há, no mesmo ciclo de relógio, outro pedido com prioridade superior para o armazenamento de pacote num mesmo registrador. Neste segundo caso, o pedido cuja prioridade momentânea é superior será atendido primeiro, postergando o atendimento dos outros pedidos para os próximos ciclos de relógio.

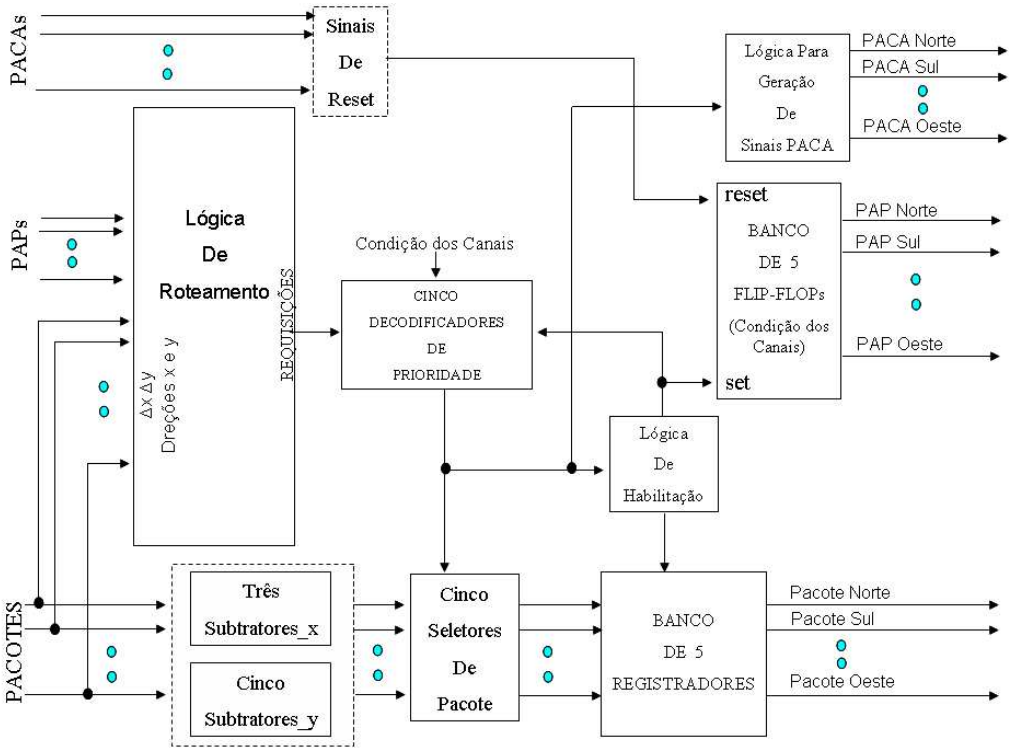


Figura 6.4: Diagrama de blocos da terceira arquitetura para o roteador

Internamente, o roteador possui cinco decodificadores de prioridade, três decrementadores para o eixo x e cinco para o eixo y, cinco seletores de pacote, cinco *flip-flops*, cinco registradores, além da lógica de roteamento. A ordem de prioridade é variável, e também neste caso, foi implementado o *round-robin*.

6.4 Comparações das Arquiteturas

Considerando apenas portas lógicas de no máximo duas entradas e tomando como variáveis o tamanho do pacote (TAM), a quantidade de bits na representação da variação x (Δx) e a quantidade de bits na representação da variação y (Δy), pode-se inferir a quantidade de portas lógicas, QP, necessárias a implementação de cada uma das três arquiteturas:

$$QP_1 = 15 * TAM + 8 * \Delta x + 8 * \Delta y + 165, \quad \text{com} \quad \Delta x \geq 2 \quad \text{e} \quad \Delta y \geq 2 \tag{6.1}$$

$$QP_2 = 39 * TAM + 4 * \Delta x + 5 * \Delta y + 251, \quad \text{com} \quad \Delta x \geq 2 \quad \text{e} \quad \Delta y \geq 2 \tag{6.2}$$

$$QP_3 = 57 * TAM + 2 * \Delta x + 11 * \Delta y + 314, \quad \text{com} \quad \Delta x \geq 2 \quad \text{e} \quad \Delta y \geq 2 \tag{6.3}$$

As expressões 6.1, 6.2 e 6.3 referem-se, respectivamente, às quantidades de portas lógicas da primeira, segunda e terceira arquiteturas.

Da mesma forma, pode-se inferir a quantidade de níveis de lógica, NL, necessários para cada uma das três arquiteturas:

$$\begin{aligned} NL_1 &= \max\{\lceil log_2 \Delta x \rceil + 7, \quad 2 * \Delta max + 4, \quad \lceil log_2 \Delta max \rceil + 18\}, \quad \text{sendo:} \\ \Delta max &= \max\{\Delta x, \Delta y\}, \quad \text{com} \quad \Delta max \geq 3 \end{aligned} \tag{6.4}$$

$$\begin{aligned} NL_2 &= \lceil log_2 \Delta max \rceil + 2 * \Delta max + 17, \quad \text{sendo:} \\ \Delta max &= \max\{\Delta x, \Delta y\}, \quad \text{com} \quad \Delta max \geq 3 \end{aligned} \tag{6.5}$$

$$\begin{aligned} NL_3 &= \max\{\lceil log_2 \Delta max \rceil + 13, \quad 2 * \Delta y + 1, \quad 2 * \Delta x\}, \quad \text{sendo:} \\ \Delta max &= \max\{\Delta x, \Delta y\}, \quad \text{com} \quad \Delta x \geq 3 \quad \text{e} \quad \Delta y \geq 3 \end{aligned} \tag{6.6}$$

As expressões 6.4, 6.5 e 6.6 referem-se, respectivamente, às quantidades de níveis de lógica da primeira, segunda e terceira arquiteturas. O símbolo $\lceil x \rceil$ indica o menor inteiro maior ou igual a x.

Os conceitos fundamentais e as deduções dessas equações matemáticas foram colocadas no anexo A desta tese.

Tomando TAM = 32, $\Delta x = 5$ e $\Delta y = 5$ como valores razoáveis para o projeto do roteador a ser utilizado na constituição do sistema multiprocessado e reconfigurável proposto, tem-se que $QP_1 = 725$, $QP_2 = 1544$, $QP_3 = 2203$, $NL_1 = 21$, $NL_2 = 30$ e $NL_3 = 15$.

A primeira arquitetura possui uma quantidade de portas lógicas significativamente menor se comparada às outras duas arquiteturas, utiliza aproximadamente 46,95% de portas lógicas em relação à segunda arquitetura e aproximadamente 32,90% em relação à terceira. Contudo, dependendo da configuração do tráfego em um dado momento no sistema de comunicação, podem ocorrer casos de *deadlock*. Este fato ocorre, mesmo utilizando a técnica de roteamento X-Y, porque com esta primeira arquitetura o roteador não é capaz de receber novos pacotes quando este se encontrar impossibilitado de enviar um pacote internamente armazenado. O problema de *deadlock* deve então ser eliminado em tempo de compilação no processo de alocação e mapeamento das transições e dos lugares da Rede de Petri na arquitetura proposta. No entanto, como uma das características do sistema proposto é a eliminação de uma série de tarefas que são realizadas para permitir o mapeamento tecnológico, esta primeira arquitetura deixa de ser atraente.

Dentre as arquiteturas restantes, tem-se que apesar da terceira arquitetura utilizar aproximadamente 42,68% a mais de portas lógicas do que a segunda, a quantidade de níveis de lógica é reduzida em 50% (de 30 níveis de lógica para apenas 15). Além disso, a terceira arquitetura é capaz de processar simultaneamente mais do que um único pedido de armazenamento de pacote.

6.5 Aspectos de Implementação

A terceira arquitetura para o roteador foi implementada utilizando a linguagem de descrição de *hardware* VHDL e o *software* Quartus II Web Edition, versão 5.0 da corporação Altera.

Para este projeto, dividiu-se as tarefas de processamento do roteador em duas grandes unidades, quais sejam:

- Lógica de Geração de Requisição: é responsável pela geração de pedidos de armazenamento de pacotes em registradores. Esta unidade é composta de qua-

tro blocos principais: Verificação de Ocorrência de Zero, Lógica de Roteamento, Decrementador x e, Decrementador y.

- Lógica de Tratamento de Requisição: é responsável pelo gerenciamento dos pedidos de armazenamento de pacotes provenientes da unidade lógica de geração de requisição. Esta unidade é composta por seis blocos principais: Registrador - Seletor, Registrador - Pacote, Registrador - Estado, Seletor Dinâmico, Lógica de Atualização e, Seletor de Pacote.

Internamente, cada canal de comunicação do roteador possui uma unidade lógica de geração de requisição e uma unidade lógica de tratamento de requisição, como mostrado na figura 6.5.

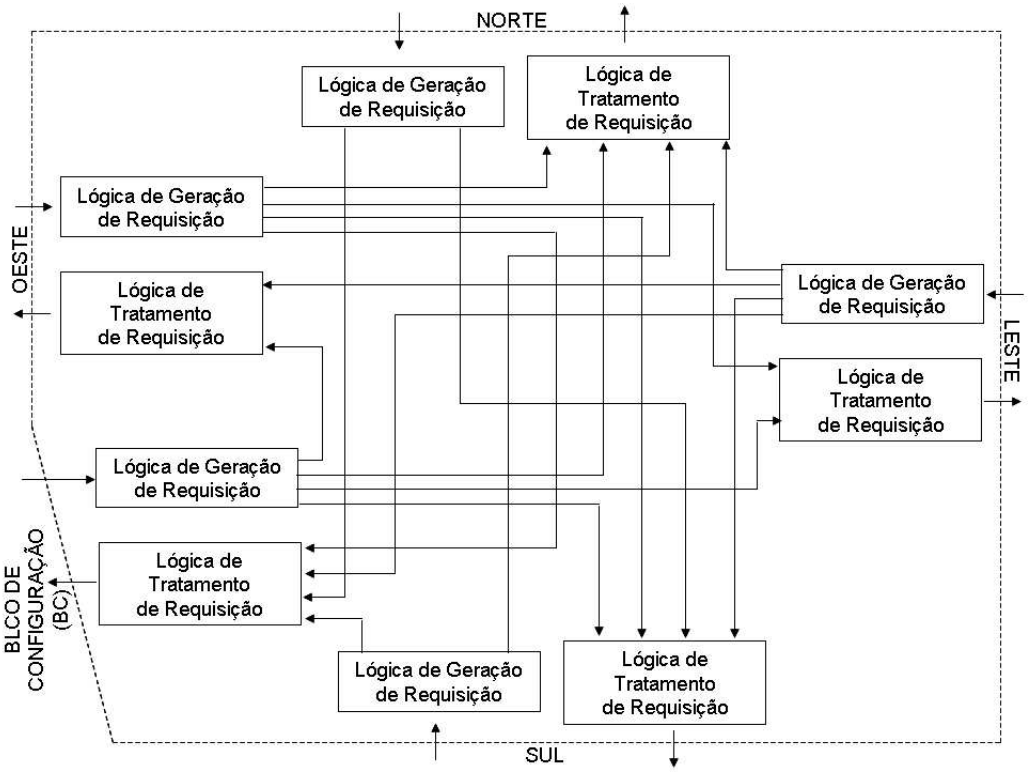


Figura 6.5: As unidades de geração e tratamento de requisições

Note-se que as unidades de geração de requisição dos canais Oeste, Leste e BC possuem quatro sinais de saída, enquanto as unidades do Norte e Sul possuem apenas duas. Esses sinais são responsáveis pela geração de um pedido de armazenamento de pacote para um registrador que se encontra em um dos cinco canais de comunicação. Por exemplo, a unidade de geração de requisição do Oeste pode realizar um pedido de armazenamento de pacote para uma das quatro unidades de tratamento de requisição, quais sejam: Norte, Leste, Sul e BC.

O código VHDL do roteador foi projetado para permitir, por meio do comando `GENERIC`, a sintetização de uma quantidade variável de bits que compõem o pacote, ou seja, basta atribuir valores à algumas variáveis para definir o tamanho do pacote, da variação no eixo x, da variação no eixo y e conseqüentemente, o tamanho da carga útil.

Assim, para o comando `GENERIC`, três variáveis foram estabelecidas, quais sejam: `INDICE_PACOTE`, `INDICE_VARIACAO_X` e `INDICE_VARIACAO_Y`.

A variável `INDICE_PACOTE` determina o tamanho do pacote, `PACOTE = [INDICE_PACOTE downto 0]`. Por sua vez, a variável `INDICE_VARIACAO_X` indica o tamanho da variação no eixo x, ou seja, `VARIAÇÃO NO EIXO X = [INDICE_VARIACAO_X downto 0]`. A quantidade de bits utilizada para representar a variação no eixo x está diretamente relacionada com a quantidade de roteadores que um pacote poderá percorrer neste eixo. Por último, a variável `INDICE_VARIACAO_Y` representa o vetor de variação no eixo y, e o seu valor está relacionado com a quantidade de roteadores que um pacote poderá atravessar no eixo y.

Por padrão, definiu-se `INDICE_PACOTE` = 31, `INDICE_VARIACAO_X` = 4 e `INDICE_VARIACAO_Y` = 4. Isto implica num pacote contendo 32 bits, dos quais 20 são destinados à carga útil e 10 ao endereçamento do pacote (5 para a variação no eixo x e 5 para o eixo y). Os outros dois bits restantes indicam o sentido que um pacote deverá caminhar, ou seja: norte-sul ou sul-norte e oeste-leste ou leste-oeste. Os 5 bits atribuidos para a variação em cada um dos eixos define um endereçamente de até $2^5 = 32$ roteadores por eixo, totalizando $32 * 32 = 1024$ roteadores em toda a arquitetura.

Contudo, a unidade de geração de requisição do Norte, por exemplo, só pode realizar um pedido para as unidades de tratamento do Sul e BC. Essa mesma característica ocorre para a unidade de geração do canal Sul, que só pode se comunicar com as unidades de tratamento do Norte e BC. Esta diferença acontece devido ao uso do algoritmo de roteamento X-Y, que obriga o pacote a primeiro caminhar na direção x e, só depois na direção y. Assim, um pacote que esteja trafegando nas direções norte ou sul não pode mais ser encaminhado para as direções leste ou oeste.

A utilização da técnica de roteamento X-Y implicou numa redução de lógica tanto para a unidade de geração de requisição quanto para a unidade de tratamento.

Como mencionado anteriormente, as unidades de geração e tratamento são basicamente subdivididas, respectivamente, em quatro e seis blocos. Na figura 6.6 apresenta-se com mais detalhes a unidade de geração de requisição do canal de comunicação Oeste e, na figura 6.7, apresenta-se a unidade lógica de tratamento de requisição do canal Norte.

Na sequência comenta-se sobre a implementação em VHDL de cada um dos dez blocos.

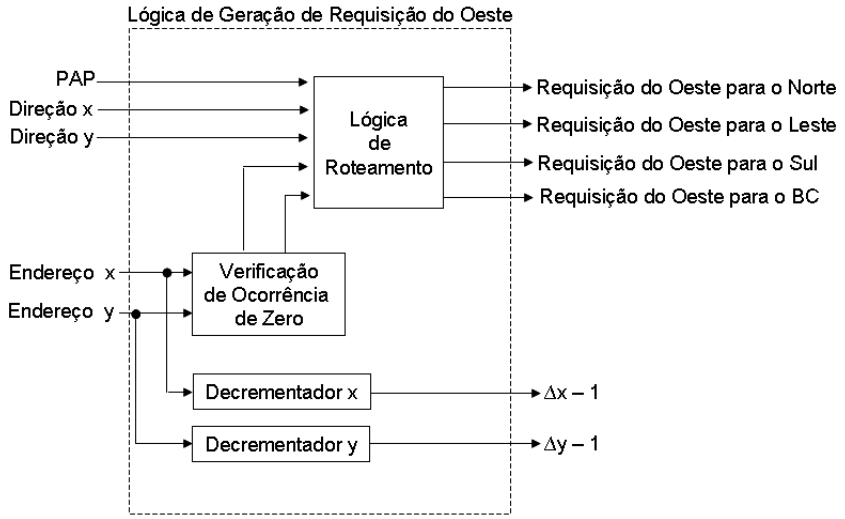


Figura 6.6: Unidade de geração de requisição

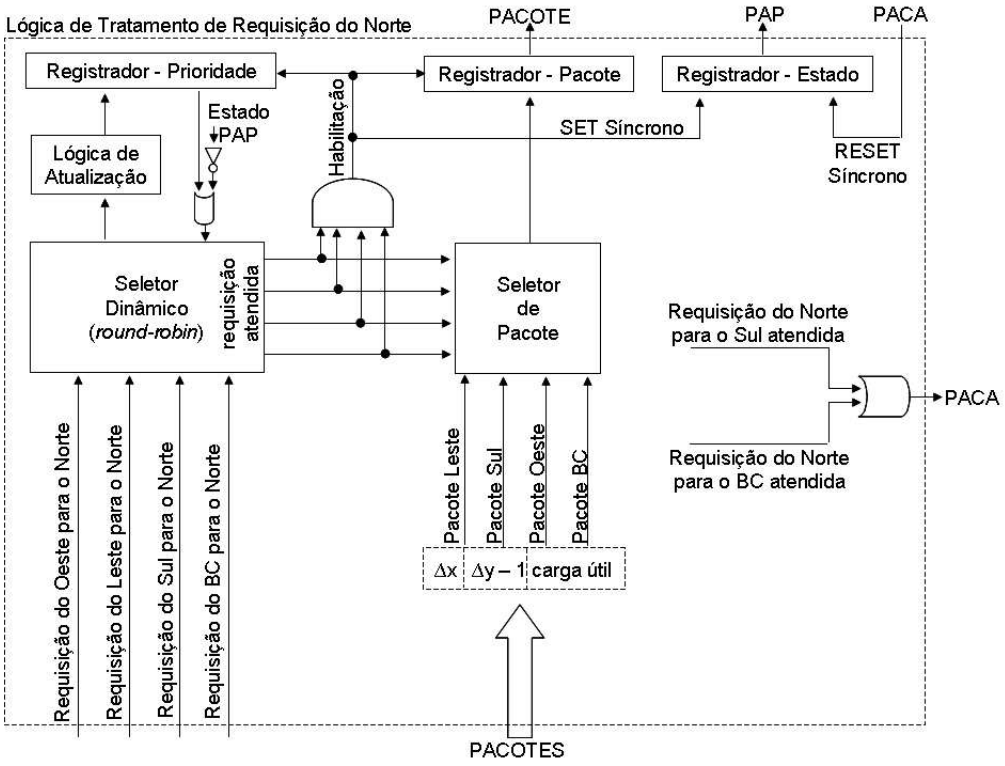


Figura 6.7: Unidade de tratamento de requisição

6.5.1 Verificação de Ocorrência de Zero

Este bloco, pertencente à unidade de geração de requisição, é responsável por identificar quando um pacote já percorreu todos os roteadores necessários na direção x ou y.

Quando as variações nos eixos x ou y forem iguais a zero, este bloco retornará um valor lógico baixo, sinalizando a ocorrência de zero para o bloco lógico de roteamento.

Na figura 6.8 apresenta-se um trecho do código VHDL deste bloco. Este bloco possui como entrada, o sinal `variacao_x`, e como saída, o sinal `zero_variacao_x`. Este bloco apenas realiza uma operação OR sobre os bits do sinal `variação_x` atribuindo o resultado ao bit de saída `zero_variacao_x`.

```
PROCESS(variacao_x)
VARIABLE x_zero : STD_LOGIC;
BEGIN

x_zero := variacao_x(INDICE_VARIACAO_X);

FOR I IN (INDICE_VARIACAO_X - 1) DOWNTO 0 LOOP
x_zero:=variacao_x(I) or x_zero;
END LOOP;

zero_variacao_x <= x_zero;

END PROCESS;
```

Figura 6.8: Código VHDL do bloco de verificação de ocorrência de zero

Um código VHDL semelhante também foi construído para realizar a verificação de ocorrência de zero no eixo y.

6.5.2 Lógica de Roteamento

Este bloco, pertencente à unidade de geração de requisição, realiza o roteamento de pacotes identificando a unidade lógica de tratamento de requisição que deve receber o pedido de armazenamento de pacote.

Na figura 6.9 apresenta-se um trecho do código VHDL deste bloco. O bloco possui como entrada, os sinais `pap_O`, `zero_variacao_x_O`, `zero_variacao_y_O` e `direcao_y_O` e, como saída, os sinais `requisicao_O_N`, `requisicao_O_L`, `requisicao_O_S` e `requisicao_O_PD`.

O sinal `pap_O` (um bit) indica o pedido de armazenamento de pacote realizado por um roteador situado ao leste. Um valor lógico alto indica um pedido de armazenamento.

Os sinais `zero_variacao_x_O` e `zero_variacao_y_O` são provenientes do bloco anterior (verificação de ocorrência de zero). O bit de sinal `direcao_y_O`, identifica se o pacote está

```
PROCESS(pap_0, zero_variacao_x_0, zero_variacao_y_0, direcao_y_0)
BEGIN

CASE pap_0 IS

    WHEN '0' =>
        requisicao_0_N  <= '0';
        requisicao_0_L  <= '0';
        requisicao_0_S  <= '0';
        requisicao_0_PD <= '0';

    WHEN '1' =>
        IF(zero_variacao_x_0 = '1')
        THEN
                requisicao_0_N  <= '0';
                requisicao_0_L  <= '1';
                requisicao_0_S  <= '0';
                requisicao_0_PD <= '0';

        ELSIF(zero_variacao_y_0 = '1')
        THEN

                CASE direcao_y_0 IS

                    WHEN '1' =>
                        requisicao_0_N  <= '1';
                        requisicao_0_L  <= '0';
                        requisicao_0_S  <= '0';
                        requisicao_0_PD <= '0';

                    WHEN '0' =>
                        requisicao_0_N  <= '0';
                        requisicao_0_L  <= '0';
                        requisicao_0_S  <= '1';
                        requisicao_0_PD <= '0';

                    WHEN OTHERS =>
                END CASE;
        ELSE
                requisicao_0_N  <= '0';
                requisicao_0_L  <= '0';
                requisicao_0_S  <= '0';
                requisicao_0_PD <= '1';

        END IF;

    WHEN OTHERS =>

END CASE;
END PROCESS;
```

Figura 6.9: Código VHDL da lógica de roteamento

indo na direção norte-sul ou sul-norte.

Assim, se o roteador situado ao leste realizar um pedido de armazenamento, este bloco lógico de roteamento será capaz de definir, baseado nos sinais de entrada `zero_variacao_x_O`, `zero_variacao_y_O` e `direcao_y_O`, para qual unidade lógica de tratamento o pacote deverá ser enviado. Para tanto, a lógica de roteamento gera os sinais de saída `requisicao_O_N` (1 bit), `requisicao_O_L` (1 bit), `requisicao_O_S` (1 bit) e `requisicao_O_PD` (1 bit). Com a geração de um sinal lógico alto em um desses sinais de saída, o bloco de roteamento poderá selecionar a unidade de tratamento de requisição do Norte, Leste, Sul ou do Processador de Dados (Bloco de Configuração).

6.5.3 Decrementador x

Este bloco, pertencente à unidade de geração de requisição, é capaz de subtrair, em uma unidade, a variação x. Toda vez que um pacote é enviado de um roteador para o outro na direção x, o campo do pacote referente a variação x deve ser decrementada, pois o pacote só chegará ao seu destino quando tanto a variação no eixo x quanto a variação no eixo y forem iguais a zero.

Na figura 6.10 apresenta-se um trecho do código VHDL deste bloco. Este bloco possui como entrada, o sinal `variacao_x`, e como saída, o sinal `resultado`.

Um código VHDL semelhante também foi construído para realizar a subtração no eixo y (Decrementador y).

```
PROCESS(variacao_x)
BEGIN

resultado <= variacao_x - '1';

END PROCESS;
```

Figura 6.10: Código VHDL do decrementador x

6.5.4 Registrador - Pacote

Este bloco, pertencente à unidade de tratamento de requisição, realiza o armazenamento de um determinado pacote. Na figura 6.11 apresenta-se um trecho do código VHDL deste bloco. Este bloco possui como entrada, os sinais `clock`, `habilitacao` e `pacote_in`,

e como saída, o sinal `pacote_out`. Se o sinal `clock` estiver transitando de baixo para alto e o sinal `habilitacao` for igual "1", então o registrador armazenará o conteúdo do sinal de entrada `pacote_in`.

```
PROCESS(clock, habilitacao, pacote_in)
VARIABLE pacote:STD_LOGIC_VECTOR(INDICE_PACOTE DOWNTO 0);
BEGIN

IF((clock'EVENT) and (clock='1') and (habilitacao = '1'))
THEN

    pacote := pacote_in;

END IF;

pacote_out <= pacote;

END PROCESS;
```

Figura 6.11: Código VHDL para o armazenamento de pacote

6.5.5 Registrador - Estado

Este bloco armazena o estado em que se encontra a unidade de tratamento de requisição. Dois estados são possíveis, disponível ou ocupado. O estado disponível indica que a unidade de tratamento pode armazenar pacotes no registrador de pacote. O estado ocupado significa que o registrador já possui um pacote armazenado que ainda não foi entregue a um roteador vizinho ou ao bloco de configuração. O conteúdo do registrador de estado gera o sinal PAP (Pedido de Armazenamento de Pacote).

Na figura 6.12 apresenta-se um trecho do código VHDL deste bloco. Este bloco possui como entrada, os sinais `clock`, `inicializar`, `set` e `reset`, e como saída, o sinal `pap_out`. Se o sinal `inicializar` for igual a "1", o conteúdo do registrador de estado se torna zero, indicando que não há nenhum pedido de armazenamento de pacote, `PAP` $= 0$. Caso contrário, a lógica de atualização do registrador é determinada pelos sinais `set` e `reset` síncronos.

6.5.6 Registrador - Prioridade

Este bloco juntamente com os blocos de lógica de atualização e seletor dinâmico, todos pertencentes à unidade de tratamento de requisição, implementam um árbitro *round-*

```
PROCESS(clock, inicializar, set, reset)
VARIABLE pap: STD_LOGIC;
BEGIN

IF(inicializar = '1')
THEN
    pap := '0';

ELSIF((clock'EVENT) and (clock='1'))
THEN
    IF(set = '1')
    THEN
        pap := '1';

    END IF;

    IF(reset = '1')
    THEN
        pap := '0';

    END IF;

END IF;

pap_out <= pap;

END PROCESS;
```

Figura 6.12: Código VHDL para o armazenamento do sinal de saída `pap`

robin, o qual será capaz de escolher um pedido de requisição para ser atendido, ou seja, determinará o pacote que será armazenado no registrador.

O registrador de prioridade armazena a ordem de prioridade na qual o seletor dinâmico realizará o cálculo da requisição que será atendida. Na figura 6.13 apresenta-se um trecho do código VHDL deste bloco. Este bloco possui como entrada, os sinais `clock`, `habilitacao`, `inicializar` e `prioridade_in`, e como saída, o sinal `prioridade_out`.

Se o sinal de inicialização estiver no nível lógico alto então o registrador armazenará o vetor "1000", fixando-se assim, uma determinada ordem de prioridade. Este procedimento deverá ocorrer somente na inicialização do sistema.

Caso o sinal de relógio transitar de baixo para alto e o sinal de habilitação for igual a "1", então o registrador armazenará o sinal de entrada `prioridade_in`, proveniente do bloco lógico de atualização de prioridade.

```
PROCESS(clock, habilitacao, inicializar, prioridade_in)
VARIABLE prioridade: STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN

IF(inicializar = '1')
THEN
    prioridade := "1000";

ELSIF((clock'EVENT) and (clock='1') and (habilitacao = '1'))
THEN
    prioridade := prioridade_in;

END IF;

prioridade_out <= prioridade;

END PROCESS;
```

Figura 6.13: Código VHDL do registrador de prioridade

6.5.7 Lógica de Atualização

A lógica de atualização é utilizada para modificar a ordem de prioridade com a qual o seletor dinâmico realiza a arbitragem. Este bloco fornece a entrada para o registrador de prioridades, comentado anteriormente. Na figura 6.14 apresenta-se um trecho do código VHDL deste bloco. Este bloco possui como entrada, os sinais `requisicao_1`, `requisicao_2`, `requisicao_3` e `requisicao_4`, e como saída, o sinal `prioridade`.

```
PROCESS(requisicao_1, requisicao_2, requisicao_3, requisicao_4)
BEGIN

prioridade(3) <= requisicao_4;
prioridade(2) <= requisicao_1;
prioridade(1) <= requisicao_2;
prioridade(0) <= requisicao_3;

END PROCESS;
```

Figura 6.14: Código VHDL da lógica de atualização

A lógica de atualização apenas realiza operações de deslocamento sobre os sinais de entrada. Esse deslocamento realiza o algoritmo *round-robin*, em que quando uma requisição for atendida, a sua ordem de prioridade, no próximo ciclo de relógio, será a menor possível.

6.5.8 Seletor Dinâmico

Na figura 6.15 apresenta-se um trecho do código VHDL do seletor dinâmico com duas requisições de entrada, denominadas `req_1_in` e `req_2_in`.

```
PROCESS(req_1_in, req_2_in, prioridade)
VARIABLE req_1_sel_1, req_2_sel_1: STD_LOGIC;
VARIABLE req_1_sel_2, req_2_sel_2: STD_LOGIC;

BEGIN

--Seletor Estatico 1 => ativado pela prioridade(1)
IF(req_1_in = '1')
THEN
    req_1_sel_1 := '1';
    req_2_sel_1 := '0';
ELSIF(req_2_in = '1')
THEN
    req_1_sel_1 := '0';
    req_2_sel_1 := '1';
ELSE
    req_1_sel_1 := '0';
    req_2_sel_1 := '0';
END IF;

--Seletor Estatico 2 => ativado pela prioridade(0)
IF(req_2_in = '1')
THEN
    req_1_sel_2 := '0';
    req_2_sel_2 := '1';
ELSIF(req_1_in = '1')
THEN
    req_1_sel_2 := '1';
    req_2_sel_2 := '0';
ELSE
    req_1_sel_2 := '0';
    req_2_sel_2 := '0';
END IF;

req_1_out <= (req_1_sel_1 and prioridade(1)) or (req_1_sel_2 and
prioridade(0));

req_2_out <= (req_2_sel_1 and prioridade(1)) or (req_2_sel_2 and
prioridade(0));

END PROCESS;
```

Figura 6.15: Código VHDL do seletor dinâmico de requisição

O seletor dinâmico de requisição, por meio da ordem de prioridade armazenada no registrador de prioridade, escolhe uma requisição dentre os pedidos provenientes da unidade de geração de requisição. O seletor só selecionará uma requisição caso o conteúdo do

registrador de estado, comentado anteriormente, for "0", o que indica a disponibilidade da unidade de tratamento de requisição. O sinal de entrada `prioridade` é proveniente do registrador de prioridade e indica a ordem de prioridade do seletor.

Na realidade, este seletor dinâmico pode ser decomposto em dois seletores estáticos. O primeiro seletor realiza uma arbitragem na qual a entrada `req_1_in` tem prioridade sobre a entrada `req_2_in`. O segundo seletor realiza uma arbitragem na qual a entrada `req_2_in` tem prioridade sobre a entrada `req_1_in`.

Assim, o sinal de entrada `prioridade` é utilizado para identificar o seletor estático que será responsável pelo processo de arbitragem. Desta forma, se o bit `prioridade(1)` for igual a "1", então o seletor estático na qual a entrada `req_1_in` tem a maior prioridade será ativado. Se o bit `prioridade(0)` for igual a "1"então o seletor estático na qual a entrada `req_2_in` tem a maior prioridade será, por sua vez, ativado.

O seletor dinâmico possui os sinais de saída `req_1_out` e `req_2_out`, os quais identificam a requisição escolhida.

6.5.9 Seletor de Pacote

Este bloco, pertencente à unidade de tratamento, realiza a multiplexação de pacotes por meio dos sinais de requisição atendida, os quais são gerados pelo bloco de seleção dinâmica, comentado anteriormente. Na figura 6.16 apresenta-se um trecho do código VHDL deste bloco. Este bloco possui os sinais de entrada `requisicao_1`, `requisicao_2`, `requisicao_3` e `requisicao_4` que indicam o pacote que deverá ser selecionado e os sinais `pacote_1`, `pacote_2`, `pacote_3` e `pacote_4`, provenientes dos quatro canais de comunicação. Como saída, produz o sinal `pacote_selecionado`.

6.6 Síntese e Simulação do Roteador

Com o auxílio do *software* Quartus II Web Edition versão 5.0, o roteador foi mapeado no FPGA EP2A15B724C7, da família APEX II. De 16640 elementos lógicos disponíveis, usou-se apenas 483 (2%), e de 492 pinos, utilizou-se 330 (67%). Com relação ao processo de simulação, foi realizado o seguinte procedimento: gerar um sinal de relógio; enviar um pacote para as saídas do roteador; gerar um nível lógico alto no sinal `PAP` de alguma porta (N, S, L, O ou BC); esperar o recebimento de confirmação de armazenamento de pacote sinalizado por um nível lógico alto no sinal `PACA` correspondente à porta em questão;

```
PROCESS(requisicao_1, requisicao_2, requisicao_3, requisicao_4,
        pacote_1, pacote_2, pacote_3, pacote_4)

VARIABLE sel_pac_1 : STD_LOGIC_VECTOR(INDICE_PACOTE DOWNTO 0);
VARIABLE sel_pac_2 : STD_LOGIC_VECTOR(INDICE_PACOTE DOWNTO 0);
VARIABLE sel_pac_3 : STD_LOGIC_VECTOR(INDICE_PACOTE DOWNTO 0);
VARIABLE sel_pac_4 : STD_LOGIC_VECTOR(INDICE_PACOTE DOWNTO 0);

BEGIN

pac_1: FOR I IN INDICE_PACOTE DOWNTO 0 LOOP

    sel_pac_1(I) := pacote_1(I) and requisicao_1;

END LOOP pac_1;


pac_2: FOR I IN INDICE_PACOTE DOWNTO 0 LOOP

    sel_pac_2(I) := pacote_2(I) and requisicao_2;

END LOOP pac_2;


pac_3: FOR I IN INDICE_PACOTE DOWNTO 0 LOOP

    sel_pac_3(I) := pacote_3(I) and requisicao_3;

END LOOP pac_3;


pac_4: FOR I IN INDICE_PACOTE DOWNTO 0 LOOP

    sel_pac_4(I) := pacote_4(I) and requisicao_4;

END LOOP pac_4;


pacote_selecionado <= sel_pac_1 or sel_pac_2 or sel_pac_3 or
sel_pac_4;

END PROCESS;
```

Figura 6.16: Código VHDL do seletor de pacote

tornar o nível lógico do sinal `PAP` baixo; esperar o envio de um nível lógico alto no sinal `PAP` em uma determinada porta, o que indicaria a tentativa de transferência do pacote para um outro roteador ou para um bloco de configuração; enviar um nível lógico alto no sinal `PACA` pertencente à porta selecionada no item anterior e verificar se o nível lógico do sinal `PAP` transita de alto para baixo.

6.7 Resultados

Na literatura científica existem alguns projetos de roteadores, como o ParIS (ZEFERINO; SANTO; SUSIN, 2004), OS4RS (BARTIC et al., 2003) e o *Hot-Potato* (NILSSON et al., 2003) mostrados na tabela 6.1. Os roteadores podem ser especificados levando em consideração os seguintes fatores: algoritmo de arbitragem, tamanho do canal, endereçamento, controle de fluxo, memorização e arbitragem.

Tabela 6.1: Especificações de alguns roteadores

	ParIS	**OS4RS**	**Roteador *Hot-Potato***	**Roteador Proposto**
Algoritmo de Roteamento	X–Y	Tabelas de roteamento	*Hot–potato*	X–Y
Tamanho do canal	32 bits	16 bits de dados	127 bits	32 bits
Endereçamento	8 bits	–	8 bits	8 bits
Controle de Fluxo	*Handshake*	*Handshake*	–	*Handshake*
Arbitragem	*Round-Robin*	*Round-Robin*	*Hop-counter/ stress value*	*Round-Robin*
Memorização	Entrada	Saída	*Buffer-less*	Saída
Síntese	FPGA-Altera APEX II	FPGA-Xilinx Virtex2Pro	Synopsys lsi10K	FPGA-Altera APEX II
Portas lógicas equivalentes	–	17500 (sem RAM)	13964	2203
LUTs / FFs	698 / 113 (sem RAM)	–	–	483 / 169

O roteador ParIS possui o algoritmo de roteamento X-Y, um canal de comunicação de 32 bits, um endereçamento de 8 bits, um controle de fluxo baseado na técnica *Handshake*, uma memorização de pacotes na entrada e realiza uma arbitragem *Round-Robin*. A implementação deste roteador possui 698 LUTs e 113 *flip-flops*, além de uma memória RAM utilizada para o armazenamento de pacotes no processo de contenção da rede.

O roteador OS4RS possui um algoritmo de roteamento baseado em tabelas, um canal de comunicação de dados de 16 bits, um controle de fluxo baseado na técnica *Handshake*, uma memorização de pacotes na saída e realiza uma arbitragem *Round-Robin*. A implementação deste roteador possui 17500 portas lógicas, além de uma memória RAM utilizada para o armazenamento de pacotes.

O roteador *Hot-Potato* possui um algoritmo de roteamento baseado na técnica *hot-potato*, um canal de comunicação de 127 bits, um endereçamento de 8 bits, uma memorização *Buffer-less* e realiza uma arbitragem utilizando contadores e valores de *stress* (*Hop-counter / Stress value*). A implementação deste roteador possui 13964 portas lógicas.

O roteador que está sendo proposto neste trabalho possui o algoritmo de roteamento X-Y, um canal de comunicação de 32 bits, um endereçamento de 8 bits, um controle de fluxo baseado na técnica *Handshake*, uma memorização de pacotes na saída e realiza uma arbitragem *Round-Robin*. A implementação do roteador proposto possui 483 LUTs e 169 *flip-flops*, totalizando 2203 portas lógicas, os pacotes são armazenados em registradores internos não necessitando de uma estrutura maior de memorização como uma RAM, por exemplo.

Deve-se observar que cada um desses roteadores tem um propósito diferente, sendo destinados a aplicações específicas, o que implica em implementações e soluções de projeto distintas. Além disso, foram utilizados diferentes processos para a realização da síntese de cada roteador. Contudo, pode-se notar que a arquitetura proposta apresenta uma quantidade significativamente pequena de portas lógicas. Isso ocorre porque os modelos de Redes de Petri que podem ser implementados na arquitetura proposta realizam uma comunicação de granularidade fina, ou seja, as informações necessárias para um determinado processamento são enviadas em um único pacote. Assim, a combinação do sistema de comunicação de granularidade fina com os projetos específicos dos blocos BCERPs e BCGNs permitiu a redução no controle de fluxo e na estrutura de memorização do roteador.

O projeto do roteador foi mapeado em outros FPGAs para a verificação da quantidade de lógica gasta para a sua implementação e da quantidade de roteadores que podem ser incluídos em um FPGA. No FPGA EP1S10F780C5, da família STRATIX, é possível o mapeamento de até 18 roteadores. Em FPGAs maiores, como o EP2C50F672C6, da família CYCLONE II, um roteador utiliza apenas 464 elementos lógicos de 50528 disponíveis (1%). Neste FPGA é possível o mapeamento de até 100 roteadores.

6.8 Roteadores na Topologia 3-D

Como comentado no capítulo 5, os roteadores com cinco e seis canais de comunicação são utilizados para estender a topologia 2-D original da arquitetura proposta para uma estrutura 3-D.

O roteador com cinco canais de comunicação é semelhante ao roteador apresentado nas seções anteriores, sendo responsável apenas pelo roteamento de pacotes dentro de sua própria camada.

Como comentado no capítulo 5, a diferença ocorre na existência de um sinal de contro-

le, o qual deve ser enviado juntamente com o primeiro pacote, indicando que um segundo pacote será entregue.

Em termos de implementação, a única diferença ocorre no bloco responsável pela lógica de atualização, apresentado na seção 6.5.7. A lógica de atualização é utilizada para modificar a ordem de prioridade com a qual o seletor dinâmico realiza a arbitragem. Desta forma, deve-se utilizar os sinais de controle para definir a ordem de prioridade desejada.

O roteador com seis canais é responsável pelo roteamento de pacotes entre as camadas. Quatro canais, como ocorre nos roteadores de cinco canais, referem-se à comunicação com os roteadores vizinhos do norte, sul, leste e oeste. Os outros dois canais referem-se à comunicação com a camada imediatamente acima e imediatamente abaixo. Neste caso, não há blocos de configuração acoplados aos roteadores.

Como comentado no capítulo 5, o bloco de configuração que precisar se comunicar com outros blocos de configuração residentes em outras camadas, deverá enviar dois pacotes.

O primeiro pacote conterá o endereço na camada de origem, um endereço na camada de destino e a variação no eixo z. O endereço na camada de origem é utilizado para enviar o pacote até um roteador de seis canais. O endereço na camada de destino é utilizado para enviar o pacote de um roteador de seis canais para o bloco de configuração destinatário e a variação no eixo z informa a quantidade de camadas que o pacote deverá percorrer.

O segundo pacote conterá a carga útil e o endereço na camada de origem, utilizado para rotear o pacote (carga útil) até o roteador de seis canais.

Este roteador também utiliza a técnica de roteamento X-Y, decrementando-se uma posição de Δx até que $\Delta x = 0$ e depois, decrementa-se uma posição de Δy até que $\Delta y = 0$.

O processo de roteamento, quando $\Delta x > 0$ ou $\Delta y > 0$, segue o mesmo procedimento do roteador apresentado para a arquitetura 2-D.

A diferença ocorre quando $\Delta x = 0$ e $\Delta y = 0$. No roteador convencional, o pacote seria entregue ao bloco de configuração nele conectado. Já no roteador com seis canais de comunicação, o pacote será enviado ou para uma camada imediatamente acima ou para uma camada imediatamente abaixo. Para tanto, o roteador deve esperar que o bloco de configuração remetente envie o segundo pacote para a composição de um novo pacote contendo apenas o endereço na camada de destino, a variação no eixo z e a carga útil.

Como foram definidos 20 bits para a carga útil, 12 bits para o endereço na camada de

origem, 5 bits para a variação no eixo z e 1 bit para identificar o sentido (camada acima ou abaixo), os dois canais referentes às camadas possuem 38 bits.

O processo de roteamento entre as camadas é semelhante ao processo de roteamento no eixo x ou y, ou seja, o pacote de 38 bits é enviado de uma camada para outra decrementando-se uma posição da variável Δz. Quando $\Delta z = 0$, indicando que o pacote chegou na camada de destino, o roteador eliminará os 6 bits referentes à variação e direção no eixo z, e o pacote, agora contendo 32 bits, será roteado para o bloco de configuração de destino.

Em termos de implementação, algumas modificações sobre os blocos apresentados na seção 6.5 são necessários. Como o roteador possui seis canais de comunicação, duas lógicas extras de geração de requisição e de tratamento de requisição são necessárias. Os dois canais referentes às camadas imediatamente superior e inferior devem implementar um registrador de 38 bits, cada um. Os canais do norte, sul, leste e oeste implementariam um registrador de 32 bits, cada um.

A lógica de roteamento continuaria semelhante, visto que todos os canais realizam o roteamento X-Y. Contudo, a lógica deveria ser estendida para o processamento no eixo z. Uma vez identificado que $\Delta x = 0$ e $\Delta y = 0$, deve-se decrementar uma posição da variável Δz, e de acordo com o bit referente ao sentido (camada superior ou inferior) a lógica de roteamento enviaria um sinal de requisição para a lógica de tratamento do canal apropriado. Se a lógica de tratamento estiver disponível, um sinal de armazenamento de pacote (`PACA`) será enviado. Neste instante, a lógica de roteamento passará para um próximo estado, aguardando que o segundo pacote seja entregue. Assim, no próximo ciclo de relógio, a lógica de roteamento enviaria a carga útil para o registrador da lógica de requisição do canal selecionado, iniciando o processo de roteamento entre camadas.

6.9 Comentários

Apresentou-se, neste capítulo, o projeto de um roteador utilizado na implementação do sistema de comunicação da arquitetura proposta. O roteador implementa o algoritmo de roteamento X-Y, possui um canal de comunicação de 32 bits, um endereçamento de 8 bits, um controle de fluxo baseado na técnica *Handshake*, uma memorização de pacotes na saída e utiliza a arbitragem denominada *Round-Robin*. O roteador proposto apresenta uma quantidade significativamente pequena de portas lógicas. Isso ocorre porque os modelos de Redes de Petri que podem ser implementados na arquitetura proposta re-

alizam uma comunicação de granularidade fina, ou seja, as informações necessárias para um determinado processamento são enviadas em um único pacote. Assim, combinando o sistema de comunicação de granularidade fina com os projetos específicos dos blocos BCERPs e BCGNs pode-se reduzir o controle de fluxo e a estrutura de memorização do roteador.

No próximo capítulo apresenta-se a arquitetura reconfigurável desenvolvida para a geração de números pseudo-aleatórios.

7 *BCGN: Bloco de Configuração do Gerador de Números Pseudo-Aleatórios*

Resumo

O bloco BCERP, quando precisar de um número pseudo-aleatório, enviará um pacote de requisição para um bloco BCGN. A carga útil deste pacote de requisição deverá conter um endereço de destino. Ao receber este pacote, o bloco BCGN armazenará o endereço de destino numa memória RAM. Cada entrada da memória RAM é percorrida à procura de solicitações não cumpridas. Se houver uma entrada cuja solicitação ainda não tiver sido executada, o bloco BCGN produzirá um número pseudo-aleatório, o qual será enviado ao endereço de destino armazenado naquela entrada da memória RAM. A arquitetura do gerador descrita em VHDL e mapeada em um FPGA é capaz de gerar milhões de números em milésimos de segundo, mesmo para números compostos por uma grande quantidade de bits.

7.1 Introdução

As unidades de geração de números pseudo-aleatórios são utilizadas pelos blocos de configuração BCERPs para resolverem problemas de conflito, ou seja, os números gerados são utilizados para determinar de forma pseudo-aleatória as transições que podem ser disparadas simultaneamente quando ocorrer uma situação de conflito entre as transições. Além disso, os números pseudo-aleatórios também são utilizados para permitir a implementação em *hardware* de modelos descritos em Redes de Petri com associação de probabilidade de disparo entre as transições, como comentado no capítulo 5.

Este capítulo aborda o desenvolvimento do bloco de configuração do gerador de números pseudo-aleatórios ou simplesmente BCGN. Muitas aplicações exigem a cons-

trução de um gerador pseudo-aleatório específico que atenda certas necessidades de uniformidade e aleatoriedade. Um gerador padrão não reconfigurável pode se comportar muito bem para determinadas aplicações, mas muito mal em outras. Por isso, a unidade de geração de números pseudo-aleatórios desenvolvida pode ser configurada para executar um gerador linear congruente multiplicativo, um gerador linear congruente *mixed*, um gerador linear múltiplo convencional com ordem de recorrência igual a dois, um gerador linear múltiplo *mixed* ou um gerador linear com transporte do tipo multiplicação-com-transporte. Desta forma, dependendo do sistema que está sendo implementado na arquitetura proposta pode-se configurar um gerador de números que atenda especificamente as necessidades deste sistema.

Nas próximas seções, explicam-se o funcionamento da unidade de geração projetada, a forma de configuração desta unidade e a arquitetura desenvolvida, comentando-se sobre alguns aspectos da sua implementação. Por fim, são expostos os testes realizados e os resultados obtidos.

7.2 Funcionamento da Unidade de Geração

A unidade de geração de números pseudo-aleatórios possui quatro sinais diferentes, sendo eles: `PAP` (pedido de armazenamento de pacote), `PACA` (indica o armazenamento de pacote), `PACOTE` (sinaliza os bits representantes do pacote que está sendo enviado) e `CARGA_ÚTIL` (indica somente os bits da carga útil de um pacote). Os sinais `PAP` e `PACA` são utilizados para controlar a comunicação entre a unidade de geração e o roteador que está acoplado nesta unidade. Esta comunicação permite a extração da carga útil do pacote, representado pelo sinal de entrada `CARGA_ÚTIL`. O envio do número pseudo-aleatório gerado para o bloco BCERP que solicitou o serviço é realizado por meio do sinal de saída `PACOTE`.

No processo de execução do sistema mapeado na arquitetura, uma transição poderá entrar em conflito com uma ou mais transições da rede. Neste momento, cada bloco BCERP que estiver mapeando uma transição em conflito deverá buscar na unidade de geração correspondente um número pseudo-aleatório para que se possa determinar, por meio de um algoritmo distribuído, a transição que deverá ser disparada.

O processo de obtenção de um número pseudo-aleatório por um bloco BCERP ocorre por meio de três etapas, quais sejam:

- Solicitação de serviço: nesta etapa o bloco BCERP deve enviar um pacote para a unidade de geração pseudo-aleatória, indicando a necessidade da geração de um número. Este pacote deve conter informações específicas que permitem localizar tanto o bloco BCERP como a unidade de geração na arquitetura proposta;

- Execução do serviço: A unidade de geração recebe o pacote proveniente do bloco BCERP que solicitou o serviço. As informações do pacote referentes a localidade do bloco BCERP são então armazenadas numa mémoria RAM. A unidade percorre cada entrada da memória RAM a procura de solicitações de serviço não cumpridas. Se houver uma entrada cuja solicitação ainda não estiver sido executada, a unidade ativará o processo de obtenção de um número pseudo-aleatório e assim, o número gerado e as informações provenientes naquela entrada da memória RAM sobre o endereço de destino do número gerado são agrupados em um pacote e este é enviado para o bloco BCERP que solicitou o serviço.

- Recebimento do número: O pacote enviado pela unidade de geração percorre a malha de roteadores até chegar ao bloco BCERP de destino. O bloco BCERP de destino recebe a carga útil do pacote enviado o qual, armazena o número pseudo-aleatório gerado.

Como a comunicação entre os blocos BCERPs e BCGNs se dá através do envio e recebimento de pacotes sustentado pela malha de roteadores, o bloco BCERP deverá enviar um pacote contendo informações específicas para a unidade de geração correspondente. Na figura 7.1 apresenta-se a composição de um pacote que deve ser enviado à unidade de geração para a produção de um número pseudo-aleatório.

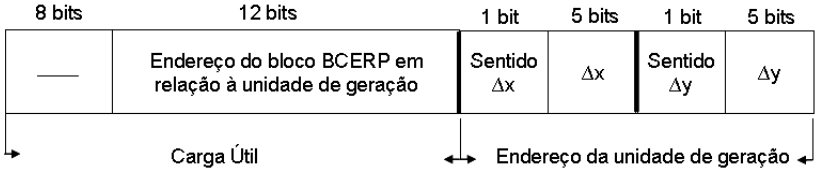


Figura 7.1: Composição do pacote que deve ser enviado à unidade de geração pelo bloco BCERP

No campo de endereço deve-se indicar a unidade de geração de números pseudo-aleatórios como destino do pacote. No campo referente à carga útil, o bloco BCERP deve informar o endereço para o qual a unidade de geração deva enviar o número gerado. Assim, a carga útil conterá o endereço do bloco BCERP em relação à unidade de geração. Cada pacote é dividido basicamente em duas partes, quais sejam: uma contendo o endereço

de destino do pacote (12 bits) e outra contendo a carga útil do pacote (20 bits). Nos 12 bits referentes ao endereço de destino do pacote deve-se indicar a unidade de geração de números pseudo-aleatórios para onde se quer enviar uma solicitação de serviço. Dos 20 bits referentes à carga útil, apenas 12 são usados para informar à unidade de geração o endereço de destino do número pseudo-aleatório a ser gerado.

Após o recebimento e execução de uma solicitação de serviço, a unidade de geração enviará um pacote contendo o número gerado para o bloco BCERP que solicitou o serviço. Na figura 7.2 apresenta-se a configuração do pacote que será enviado pela unidade de geração para o bloco BCERP.

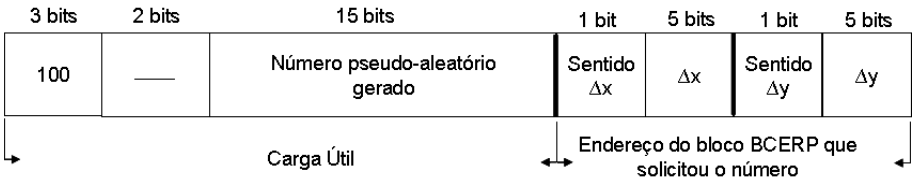


Figura 7.2: Composição do pacote que deve ser enviado pela unidade de geração para o bloco BCERP que solicitou um número pseudo-aleatório

No campo de endereço deve-se indicar como destino do pacote o bloco BCERP que solicitou o serviço. Cada número pseudo-aleatório gerado possui 15 bits, sendo integralmente armazenado na carga útil de um único pacote. Os três primeiros bits da carga útil deve ser "100", este campo é utilizado pelo bloco BCERP para indicar que o número pseudo-aleatório enviado é proveniente de um bloco BCGN e não de um bloco BCERP. Isso se faz necessário porque números pseudo-aleatórios também podem ser enviados por blocos BCERPs com o intuito de determinar as transições que poderão disparar, quando estas se encontrarem em situação de conflito.

7.3 Configuração da Unidade de Geração

Como comentado anteriormente, muitas aplicações exigem a construção de um gerador pseudo-aleatório específico que atenda as necessidades de uniformidade e aleatoriedade da aplicação, visto que um gerador padrão pode se comportar muito bem para determinadas aplicações, mas muito mal em outras.

Assim, para manter uma certa flexibilidade na obtenção de uma seqüência pseudo-aleatória de boa qualidade, a unidade pode ser configurada para executar um gerador que melhor atenda as necessidades da aplicação a ser mapeada na arquitetura.

A unidade de geração pode ser configurada para implementar um dentre cinco diferentes geradores, quais sejam:

- Gerador linear congruente multiplicativo
- Gerador linear congruente *mixed*
- Gerador linear múltiplo convencional de ordem dois
- Gerador linear múltiplo *mixed* de ordem dois
- Gerador linear com transporte do tipo multiplicação-com-transporte

Os geradores lineares congruentes são os mais comuns e bastante utilizados, são implementados em bibliotecas padrões como as funções RAND na linguagem C e RANDU nos computadores IBM. Portanto, é importante que a unidade configure esse tipo de gerador. Atualmente, os geradores lineares múltiplos estão sendo muito utilizados e podem apresentar seqüências pseudo-aleatórias com propriedades melhores e períodos mais longos. Portanto, estes geradores também podem ser configurados na unidade de geração. Geradores lineares com transporte não são muito utilizados, porém, dependendo dos parâmetros escolhidos, podem apresentar ótimas seqüências pseudo-aleatórias além de períodos longos, mesmo utilizando módulos em potência de dois (L'ECUYER, 1998). A utilização de um módulo em potência de dois pode reduzir significativamente o tempo de obtenção dos números pseudo-aleatórios, visto que não há necessidade de utilizar a operação binária de divisão, ao invés disso, realizam-se apenas algumas operações de deslocamento. Como o gerador linear com transporte é uma grande promessa na geração de seqüências pseudo-aleatórias, a unidade de geração também pode ser configurada para implementar esse tipo de gerador.

A obtenção de seqüências de números pseudo-aleatórios com boas propriedades de aleatoriedade, uniformidade e períodos longos dependem não só do tipo de gerador selecionado como também dos parâmetros escolhidos. A unidade de geração permite não só a configuração do tipo de equação de recorrência utilizada como também permite a configuração de alguns de seus respectivos parâmetros, ou seja, permite a escolha das sementes, dos multiplicadores, incremento ou transporte inicial, no caso de se configurar um gerador com transporte, conseguindo-se com isso, um maior grau de flexibilidade na construção do gerador. O módulo da equação foi fixado como sendo uma potência de dois, no caso, 2^{15}, o que permite uma rápida geração de números pseudo-aleatórios devido a

utilização de deslocamentos ao invés da operação de divisão. Além disso, o deslocamento dos 15 bits é realizado simultaneamente, o que acelera ainda mais o processo de obtenção dos números. Resumidamente, apresenta-se a seguir a equação que generaliza todas as possibilidades de configuração na unidade de geração:

$$\begin{aligned} x_i &= (a_1 * x_{i-1} + a_2 * x_{i-2} + c_{i-1}) \mod 2^{15}, \quad i \geq k \\ c_i &= (a_1 * x_{i-1} + a_2 * x_{i-2} + c_{i-1}) \text{ div } 2^{15}, \quad \text{ou} \\ c_i &= c_{i-1} \quad \text{ou} \quad \text{constante} \end{aligned}$$

De acordo com essa equação, os parâmetros que podem ser configurados são: a_1, a_2, as sementes do gerador x_1 e x_2 e o tipo de equação responsável pela computação de c_i. No processo de configuração da unidade, deve-se enviar um pacote indicando se a equação responsável pela geração de c_i será uma equação de recorrência ou se o valor de c_i não se modificará durante as iterações da equação, o que, na realidade, tornaria esta variável apenas um número de incremento cujo valor é definido pelo envio de outro pacote à unidade de geração, estabelecendo o valor desta constante. Desta forma, se a constante enviada à unidade de geração for zero, têm-se um gerador linear multiplicativo, caso contrário, têm-se um gerador linear *mixed*. Além disso, se for enviado um pacote indicando que o parâmetro a_2 é igual a zero, têm-se um gerador congruente, caso contrário, têm-se um gerador linear com ordem de recorrência igual a 2. Agora, se for escolhida a equação de recorrência para computar os valores de c_i, então têm-se um gerador linear com transporte do tipo multiplicação-com-transporte. No caso do gerador com transporte, deve-se enviar um pacote indicando o valor inicial de c_1.

7.4 Arquitetura da Unidade de Geração

Como comentado anteriormente a unidade de geração de números pseudo-aleatórios possui três sinais de entrada, sendo: `PAP_IN`, `PACA_IN`, `CARGA_ÚTIL`; e três sinais de saída, sendo: `PAP_OUT`, `PACA_OUT` e `PACOTE`. O sinal `PACOTE` foi definido com 32 bits, o sinal `CARGA_ÚTIL` foi definido com 20 bits e os sinais restantes, denominados sinais de controle da comunicação, possuem apenas 1 bit cada um.

A arquitetura da unidade de geração foi subdivida em dois grandes blocos digitais. No primeiro bloco, realiza-se, efetivamente, a geração de um número pseudo-aleatório. Neste bloco, deve-se configurar tanto a equação de recorrência como os seus respectivos parâmetros. A configuração desse bloco é realizado pelo segundo bloco, o qual é res-

ponsável pela realização da maioria das tarefas. Este bloco, além de ser responsável pela configuração do primeiro bloco, realiza todo o processamento de controle necessário para a comunicação da unidade de geração com o ambiente externo, representado neste caso, pelo roteador que está acoplado à unidade de geração.

O primeiro bloco, denominado `Geração de Números Pseudo-Aleatórios` pode ser observado na figura 7.3. Este bloco possui quatro registradores de 15 bits para o armazenando dos parâmetros a_1, a_2, x_{i-1} e x_{i-2}, um registrador de 30 bits para o armazenamento do parâmetro c_{i-1} e um registrador de apenas um bit para controlar a geração da variável de transporte c_i. Se o conteúdo deste registrador for igual a um, têm-se que $c_i = c_{i-1} =$ constante, caso contrário, $c_i = (a_1 * x_{i-1} + a_2 * x_{i-2} + c_{i-1}) \quad \text{div} \quad 2^{15}$. Este bloco possui dois multiplicadores e dois somadores utilizados para computar a equação $x_i = (a_1 * x_{i-1} + a_2 * x_{i-2} + c_{i-1})$.

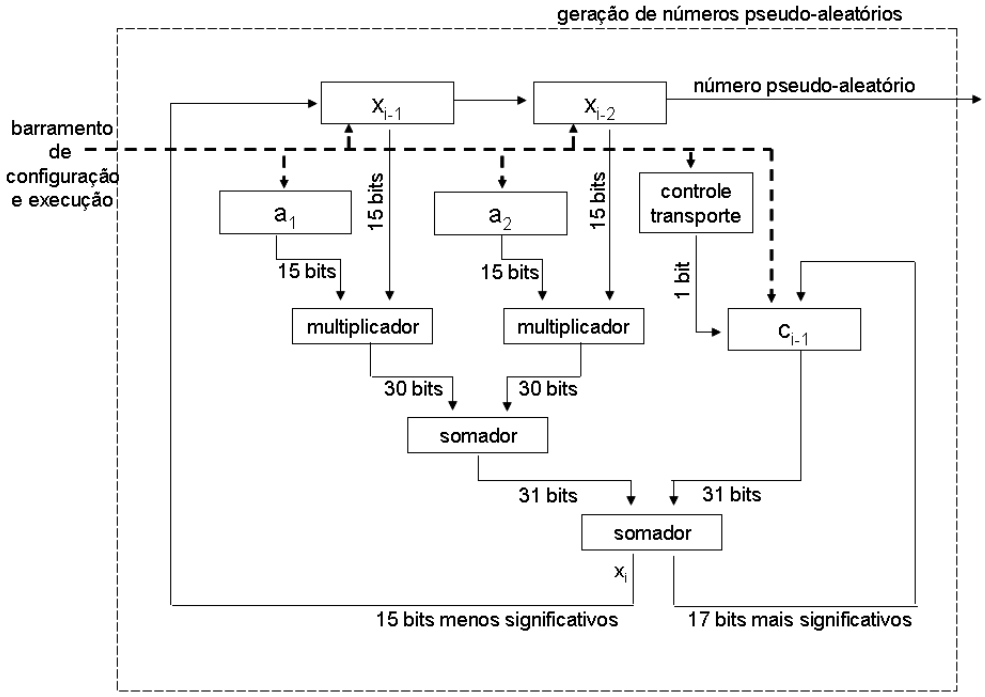


Figura 7.3: Bloco responsável pela geração do número pseudo-aleatório

Os 32 bits resultantes da última soma, como mostrado na figura 7.3, são distribuídos da seguinte forma, os 15 bits menos significativos serão armazenados no registrador x_{i-1} e os 17 bits mais significativos serão armazenados no registrador c_{i-1}, isso se o controle de transporte for igual a zero, pois caso contrário, o valor armazenado no registrador c_{i-1} não se alterará. Além disso, o conteúdo do registrador x_{i-1} será transferido para o registrador x_{i-2} e o atual conteúdo do registrador x_{i-2} será entregue ao segundo bloco como sendo o número pseudo-aleatório gerado.

O barramento de configuração e execução é utilizado de duas formas diferentes. Num primeiro momento, durante o processo de configuração da unidade, este barramento é responsável pelo armazenamento dos valores provenientes dos pacotes de configuração em cada um dos seis registradores, configurando-se assim o gerador. Num segundo momento, durante a execução da unidade de geração, este barramento é utilizado pelo segundo bloco para ordenar a transferência de conteúdo do registrador x_{i-1} para o registrador x_{i-2}, o armazenamento dos 15 bits menos significativos do resultado do último somador no registrador x_{i-1} e o armazenamento dos 17 bits mais significativos no registrador c_{i-1}, caso o controle de transporte seja igual a zero. Este procedimento gerará um número pseudo-aleatório o qual será enviado, posteriormente, para o bloco BCERP que solicitou a sua geração.

A configuração desse bloco é realizada pelo segundo bloco, o qual é responsável pela realização da maioria das tarefas. O segundo bloco, além de ser responsável pela configuração do primeiro bloco, realiza todo o processamento de controle necessário para a comunicação da unidade de geração com o ambiente externo, representado neste caso, pelo roteador que está acoplado à unidade de geração.

Na figura 7.4 apresenta-se toda a arquitetura da unidade de geração, composta pelo primeiro e pelo segundo blocos.

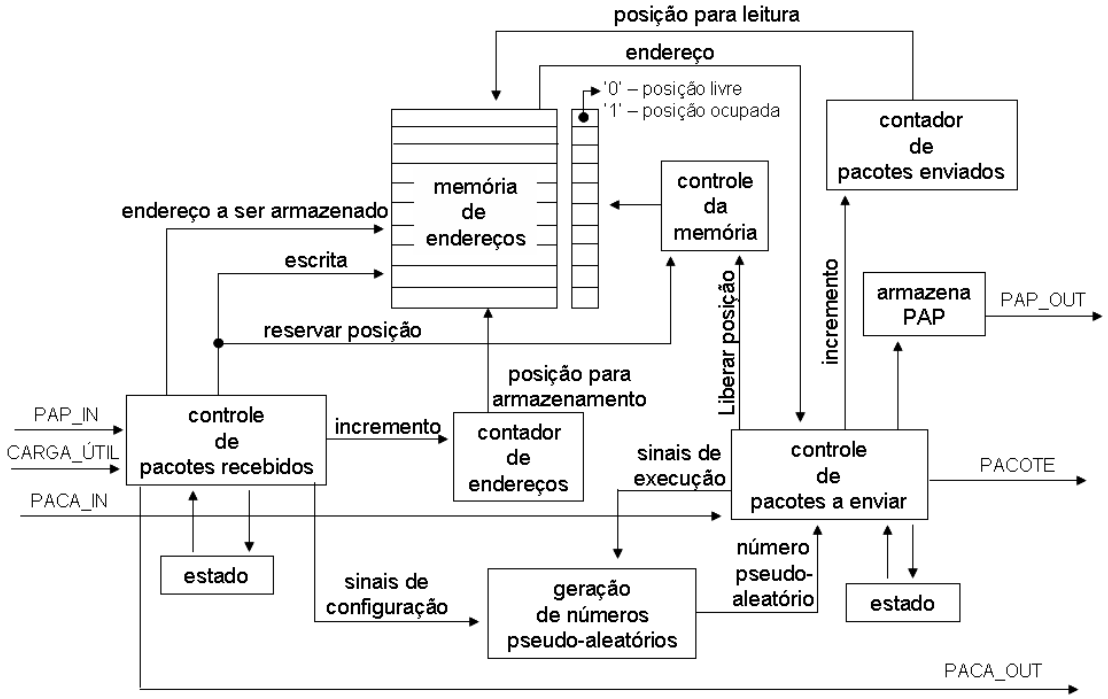


Figura 7.4: Arquitetura da unidade de geração de números pseudo-aleatórios

O segundo bloco realiza basicamente duas grandes tarefas. Na primeira tarefa, ocorre a organização dos pacotes recebidos, identificando-se as requisições de serviço e extraindo-se as informações necessárias para o posterior envio do número pseudo-aleatório gerado ao bloco BCERP remetente. Na segunda tarefa, acontece o envio do número gerado no primeiro bloco para o bloco BCERP que solicitou o serviço, realizando-se aqui todo o processo de comunicação com o meio externo.

Para tanto, o segundo bloco é composto de vários componentes que auxiliam na execução dessas duas tarefas, quais sejam: dois contadores, um de endereço e um de pacotes enviados, uma memória de leitura e escrita, um controlador de memória, uma unidade de controle de pacotes recebidos, uma unidade de controle de pacotes a enviar, um registrador para armazenar o estado da unidade de controle de pacotes recebidos, um registrador para armazenar o estado da unidade de controle de pacotes a enviar e um registrador para armazenar o sinal `PAP_OUT`. A utilização da memória de leitura e escrita elimina a ocorrência de *deadlocks* pois permite que a unidade de geração, mesmo impossibilitada de enviar informações ao sistema de comunicação devido a um alto tráfego, receba e armazene na memória novas requisições de geração de números.

Na figura 7.5 apresenta-se o fluxograma implementado na unidade de controle de pacotes recebidos.

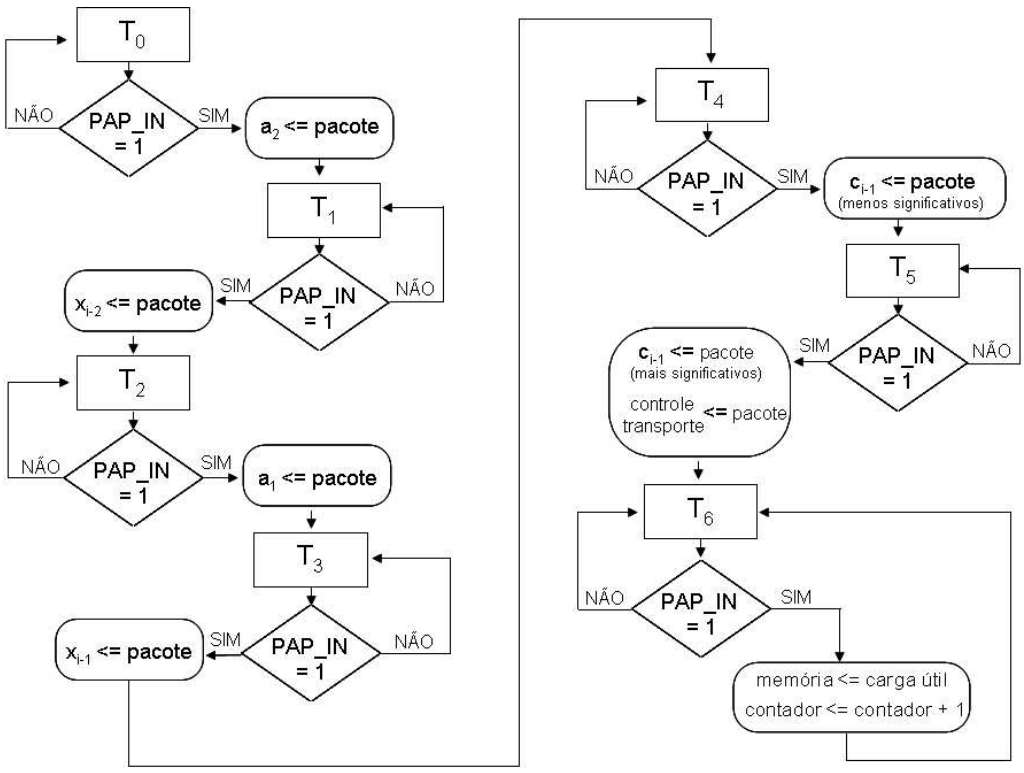


Figura 7.5: Fluxograma da unidade de controle de pacotes recebidos

Ao receber um pacote de solicitação de serviço (`PAP_IN` = '1'), esta unidade, extrai dele a informação sobre o endereço de destino do número pseudo-aleatório a ser gerado e a envia para a memória de leitura e escrita aonde a informação será armazenada. A posição de memória que será utilizada para armazenar o endereço de destino se encontra no contador de endereços. Uma vez armazenado, o contador de endereços é então incrementado para permitir que novas solicitações de serviços possam ser armazenadas em posições subseqüentes da memória.

Porém, no processo de inicialização da unidade de geração, os pacotes recebidos pela unidade são, na realidade, pacotes de configuração e devem ser utilizados para configurar o bloco `Geração de Números Pseudo-Aleatórios`. Na inicialização da unidade de geração, o conteúdo do registrador utilizado para armazenar o estado da unidade de controle de pacotes recebidos é zerado. Portanto, quando o estado da unidade de controle de pacotes recebidos for igual a zero, os pacotes de configuração serão recebidos e enviados um de cada vez para os registradores pertencentes ao bloco `Geração de Números Pseudo-Aleatórios`. A ordem de envio dos pacotes de configuração para os registradores é pré-definida e deve seguir a disposição mostrada na figura 7.6.

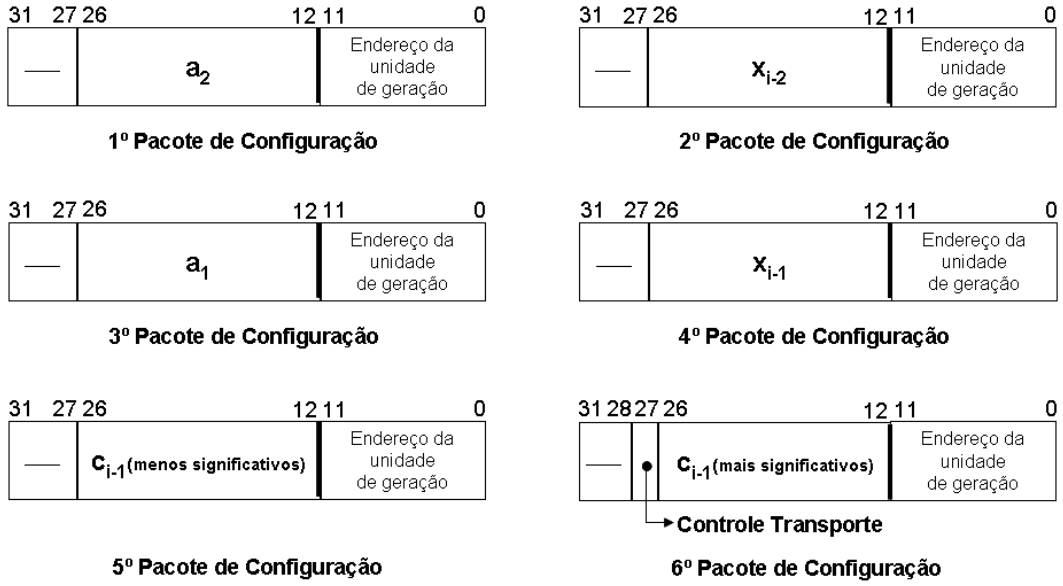


Figura 7.6: Ordem dos pacotes de configuração que devem ser direcionados à unidade de geração de números pseudo-aleatórios

Por sua vez, a unidade de controle de pacotes a enviar realiza toda a comunicação necessária para o envio de um pacote contendo o número pseudo-aleatório para o bloco BCERP que solicitou a geração do número. Na figura 7.7 apresenta-se o fluxograma implementado na unidade de controle de pacotes a enviar. Esta unidade percorre a memória de endereços na busca por solicitações que precisam ser executadas. Assim, ao encontrar

uma posição de memória ocupada (valor lógico igual a um) esta unidade executará a solicitação armazenada nesta posição. Para tanto, a unidade enviará um pacote com o número pseudo-aleatório gerado para o bloco BCERP remetente, cujo endereço se encontra armazenado na posição de memória. Ao enviar o pacote (sinal `PAP_OUT` no nível lógico alto), a unidade espera a confirmação de recebimento proveniente do roteador. Isso só ocorrerá quando o sinal `PACA_IN` for igual a um.

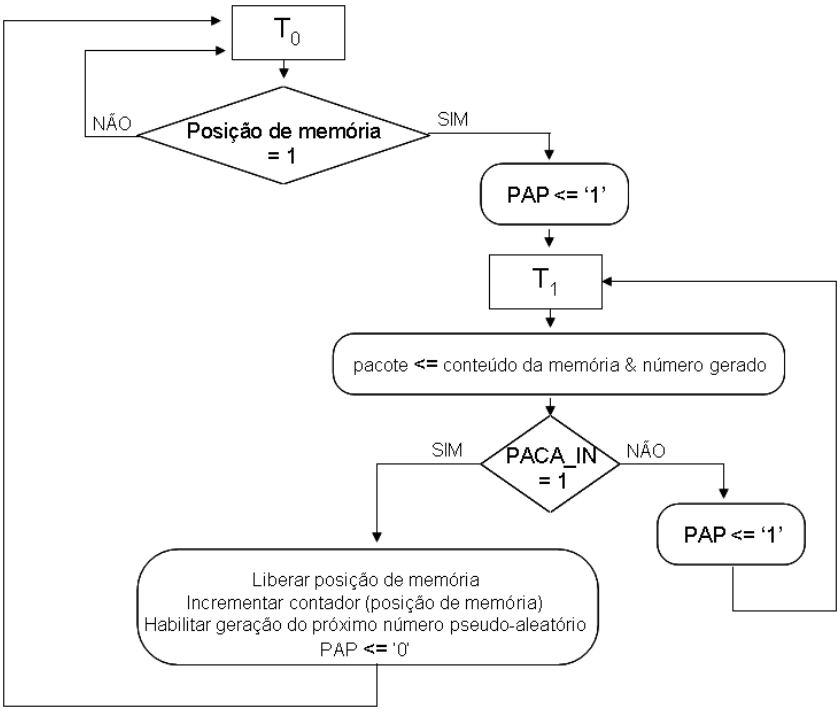


Figura 7.7: Fluxograma da unidade de controle de pacotes a enviar

Ao receber a confirmação de recebimento de pacote do roteador, a unidade liberará a posição de memória, para permitir que uma nova solicitação possa ser armazenada nesta posição. Além disso, o contador de pacotes a enviar (contador de posições) será incrementado para que se possa continuar realizando a busca por solicitações não atendidas em outras posições de memória e, para finalizar, um sinal será enviado até a unidade de geração de números pseudo-aleatórios para que ocorra, neste bloco, a transferência de conteúdo do registrador x_{i-1} para o registrador x_{i-2}, o armazenamento dos 15 bits menos significativos do resultado do último somador no registrador x_{i-1} e o armazenamento dos 17 bits mais significativos no registrador c_{i-1}. Este procedimento gerará um número pseudo-aleatório o qual poderá ser utilizado posteriormente pela unidade de controle de pacotes a enviar, para atender a solicitação de um novo pedido.

7.5 Aspectos de Implementação

A arquitetura da unidade de geração de números pseudo-aleatórios apresentada na seção 7.4 foi implementada utilizando a linguagem de descrição de *hardware* VHDL e o *software* Quartus II.

O código VHDL da unidade de números pseudo-aleatórios foi projetado para permitir, por meio do comando `GENERIC`, a síntese de uma quantidade variável de bits que compõem carga útil, os números gerados e o tamanho da memória, ou seja, basta atribuir valores a algumas variáveis para definir a quantidades de bits de endereçamento do sistema de roteamento, o tamanho da carga útil dos pacotes, a quantidade de bits que representa o número pseudo-aleatório gerado e a quantidade de posições da memória.

Assim, para o comando `GENERIC`, quatro variáveis foram previamente estabelecidas, quais sejam: `INDICE_ENDERECO`, `INDICE_CARGA_UTIL`, `INDICE_NUM` e `INDICE_MEM`.

A variável `INDICE_ENDERECO` determina a quantidade de bits utilizada pelo sistema de roteamento para definir o seu endereçamento, `ENDERECO = [INDICE_ENDERECO downto 0]`. A quantidade de bits utilizada para representar o endereçamento está diretamente relacionada com a quantidade máxima de roteadores que um pacote poderá percorrer. Por sua vez, a variável `INDICE_CARGA_UTIL` indica a quantidade de bits atribuída à carga útil do pacote de comunicação, ou seja, `CARGA UTIL = [INDICE_CARGA_UTIL downto 0]`. A variável `INDICE_NUM` representa a quantidade de bits utilizada para definir um número pseudo-aleatório gerado, ou seja, `NUMERO GERADO = [INDICE_NUM downto 0]`. Por último, a variável `INDICE_MEM` representa a quantidade de posições (palavras) que podem ser armazenadas na memória, `MEMORIA = [INDICE_MEM downto 0]`.

Por padrão, definiu-se `INDICE_ENDERECO` = 11, `INDICE_CARGA_UTIL` = 19, `INDICE_NUM` = 14 e `INDICE_MEM` = 19. Isto implica num pacote contendo 32 bits, dos quais 20 são destinados à carga útil e 12 ao endereçamento do pacote, sendo 5 para a variação no eixo X e 5 para o eixo Y e os outros dois bits restantes do endereçamento indicam o sentido que um pacote deverá caminhar, ou seja: norte-sul ou sul-norte e oeste-leste ou leste-oeste. Os 5 bits atribuidos para a variação em cada um dos eixos definem um endereçamento de até $2^5 = 32$ roteadores por eixo, totalizando $32 * 32 = 1024$ roteadores em toda a arquitetura.

Como mostrado na figura 7.4, a unidade de geração de números pseudo-aleatórios possui 10 blocos lógicos, sendo eles: Controle de Pacotes Recebidos, Estado do Controle de Pacotes Recebidos, Contador de Endereços, Geração de Números Pseudo-Aleatórios, Memória de Endereços, Controle da Memória, Controle de Pacotes a Enviar, Estado do

Controle de Pacotes a Enviar, Contador de Pacotes Enviados e Armazena `PAP`.

O bloco Geração de Números Pseudo-Aleatórios, como mostrado na figura 7.3, é constituido de outros 10 sub-blocos, sendo: Número x_{i-1}, Número x_{i-2}, Multiplicador a_1, Multiplicador a_2, Controle do sinal de Transporte, Transporte c_{i-1}, Multiplicador de dois operandos (dois blocos), Somador de dois operandos (dois blocos).

Descreve-se, a seguir, a implementação em VHDL dos sub-blocos que compõem o bloco Geração de Números Pseudo-Aleatórios e depois, a implementação dos blocos restantes da unidade de geração.

7.5.1 Armazenamento dos Últimos Números Gerados

Durante a etapa de execução da unidade, tanto o sub-bloco Número x_{i-1} como o sub-bloco Número x_{i-2}, ambos pertencentes ao bloco Geração de Números Pseudo-Aleatórios, são responsáveis, respectivamente, pelo armazenamento do último e do penúltimo número pseudo-aleatório gerado. Durante o processo de configuração, estes registradores armazenam os números iniciais, ou sementes do gerador que está sendo implementado. Na figura 7.8, apresenta-se um trecho do código VHDL do sub-bloco Número x_{i-2} que possui como entrada os sinais `clock`, `ha_config`, `ha_exec`, `numero_config` e `numero_exec`, e como saída, o sinal `numero_out`.

```
PROCESS(clock, ha_config, ha_exec, numero_config, numero_exec)
VARIABLE numero: STD_LOGIC_VECTOR(INDICE_NUM DOWNTO 0);
BEGIN

IF((clock'EVENT) and (clock = '1'))
THEN
    IF(ha_config = '1')
    THEN
        numero := numero_config;
    ELSIF(ha_exec = '1')
    THEN
        numero := numero_exec;
    END IF;
END IF;
numero_out <= numero;

END PROCESS;
```

Figura 7.8: Código VHDL do sub-bloco de armazenamento do penúltimo número gerado

Se houver uma transição lógica de baixo para alto no sinal `clock` e o sinal de entrada

`ha_config` estiver no nível lógico alto, então a etapa de configuração foi acionada e deve-se armazenar o conteúdo que está disponível no barramento de configuração, `numero_config`. Caso houver transição no sinal `clock` e o sinal `ha_exec` estiver no nível lógico alto então deve-se armazenar o sinal `numero_exec`, pois a unidade se encontra na fase de execução. O sinal `numero_exec` é produzido no sub-bloco Somador – 31 bits de entrada e corresponde ao número pseudo-aleatório recentemente gerado pela unidade. O sub-bloco mostra no sinal de saída `numero_out` o conteúdo armazenado no registrador.

7.5.2 Armazenamento dos Multiplicadores

Durante a etapa de configuração da unidade de geração, os sub-blocos Multiplicador a_1 e Multiplicador a_2, pertencentes ao bloco Geração de Números Pseudo-Aleatórios, são responsáveis pelo armazenamento dos dois multiplicadores. Estes multiplicadores são parâmetros que em conjunto com outros parâmetros definem o funcionamento e o tipo de gerador que se pretende mapear.

Na figura 7.9 apresenta-se um trecho do código VHDL do sub-bloco Multiplicador a_2, o qual possui como entrada os sinais `clock`, `habilitacao` e `multiplicador_in` e como saída, o sinal `multiplicador_out`. Se houver uma transição lógica de baixo para alto no sinal `clock` e o sinal de entrada `habilitacao` estiver no nível lógico alto, então a etapa de configuração foi acionada e deve-se armazenar o conteúdo que está disponível no barramento de configuração, `multiplicador_in`. O sub-bloco mostra no sinal de saída `multiplicador_out` o conteúdo armazenado no registrador.

```
PROCESS(clock, habilitacao, multiplicador_in)
VARIABLE multiplicador: STD_LOGIC_VECTOR(INDICE_NUM DOWNTO 0);
BEGIN

IF((clock'EVENT) and (clock='1') and (habilitacao = '1'))
THEN
    multiplicador := multiplicador_in;
END IF;

multiplicador_out <= multiplicador;

END PROCESS;
```

Figura 7.9: Código VHDL do sub-bloco de armazenamento do parâmetro de multiplicação

7.5.3 Armazenamento do Controle do Sinal de Transporte

Durante a etapa de configuração da unidade de geração, este sub-bloco pertencente ao bloco Geração de Números Pseudo-Aleatórios é responsável por armazenar o parâmetro que controla a geração da variável de transporte c_i indicando se o gerador linear será do tipo multiplicação-com-transporte. Se o conteúdo deste registrador for igual a um, têm-se que $c_i = c_{i-1} =$ constante, caso contrário, $c_i = (a_1 * x_{i-1} + a_2 * x_{i-2} + c_{i-1}) \quad \text{div} \quad 2^{15}$.

Na figura 7.10 apresenta-se um trecho do código VHDL deste sub-bloco, o qual possui como entrada os sinais `clock`, `habilitacao` e `entrada_controle` e como saída, o sinal `saida_controle`. Se houver uma transição lógica de baixo para alto no sinal `clock` e o sinal de entrada `habilitacao` estiver no nível lógico alto, então a etapa de configuração foi acionada e deve-se armazenar o conteúdo que estiver disponível no barramento de configuração, `entrada_controle`. O sub-bloco mostra, no sinal `saida_controle`, o conteúdo armazenado no registrador.

```
PROCESS(clock, habilitacao, entrada_controle)
VARIABLE controle: STD_LOGIC;
BEGIN

IF((clock'EVENT) and (clock = '1') and habilitacao = '1')
THEN
    controle := entrada_controle;
END IF;

saida_controle <= controle;

END PROCESS;
```

Figura 7.10: Código VHDL do sub-bloco de armazenamento do controle do sinal de transporte

7.5.4 Armazenamento do Sinal de Transporte

Este sub-bloco, pertencente ao bloco Geração de Números Pseudo-Aleatórios, é responsável pelo armazenamento ou do sinal de transporte ou de uma constante. Dependendo do valor armazenado no registrador que controla o sinal de transporte, este sub-bloco pode armazenar o transporte gerado no processo de obtenção de um número pseudo-aleatório durante a etapa de execução da unidade ou pode atuar apenas como um registrador que disponibiliza uma constante, a qual é utilizada para a geração dos números.

Na figura 7.11 apresenta-se um trecho do código VHDL deste sub-bloco, o qual possui como entrada os sinais `clock`, `ha_config_menos_sign`, `ha_config_mais_sign`, `ha_exec`, `controle`, `numero_config` e `numero_exec` e como saída, o sinal `numero_out`.

```
PROCESS(clock, ha_config_menos_sign, ha_config_mais_sign, ha_exec,
        controle, numero_config, numero_exec)
VARIABLE numero: STD_LOGIC_VECTOR((2 * INDICE_NUM + 1) DOWNTO 0);
BEGIN

IF((clock'EVENT) and (clock = '1'))
THEN
    IF(ha_config_menos_sign = '1')
    THEN
        numero(INDICE_NUM DOWNTO 0) := numero_config;
    ELSIF(ha_config_mais_sign = '1')
    THEN
        numero((2 * INDICE_NUM + 1) DOWNTO (INDICE_NUM + 1)) := numero_config;
    ELSIF(ha_exec = '1' AND controle = '0')
    THEN
        numero((INDICE_NUM + 2) DOWNTO 0) := numero_exec;
    END IF;
END IF;

numero_out <= numero;

END PROCESS;
```

Figura 7.11: Código VHDL do sub-bloco de armazenamento do sinal de Transporte

Se houver uma transição lógica de baixo para alto no sinal `clock` e o sinal de entrada `ha_config_menos_sign` estiver no nível lógico alto, então a etapa de configuração foi acionada e deve-se armazenar, nas posições menos significativas, o conteúdo que estiver disponível no barramento de configuração, `numero_config`. Quando houver a transição do sinal `clock` e o sinal de entrada `ha_config_mais_sign` estiver no nível lógico alto então deve-se armazenar, nas posições mais significativas, o conteúdo que etiver disponível no barramento de configuração. Se a unidade de geração estiver na etapa de execução, este sub-bloco deverá fornecer ou uma constante ou um transporte. Portanto, se o sinal `clock` transitar do nível lógico baixo para alto e o sinal `controle` estiver no nível lógico baixo, então o sub-bloco deverá armazenar o transporte gerado na produção do número pseudo-aleatório. Caso o controle esteja no nível lógico alto, então este sub-bloco não armazenará o transporte gerado em cada iteração, atuando desta forma, como uma constante.

O sub-bloco mostra no sinal de saída `numero_out` o conteúdo armazenado em seu registrador.

7.5.5 Multiplicação de Operandos

Existem, no bloco Geração de Números Pseudo-Aleatórios, dois sub-blocos responsáveis pela multiplicação de dois operandos. Na figura 7.12 apresenta-se um trecho do código VHDL do sub-bloco Multiplicador, o qual possui como entrada os sinais `multiplicador` e `numero_aleatorio` e, como saída, o sinal `result_multiplicacao`. O sinal de saída é apenas a multiplicação dos dois sinais de entrada `multiplicador` e `numero_aleatorio`. O sinal de entrada `multiplicador` pode ser armazenado no registrador do sub-bloco Multiplicador a_1 ou do registrador do sub-bloco Multiplicador a_2 e o sinal de entrada `numero_aleatorio` pode ser armazenado no registrador do sub-bloco Número x_{i-2} ou do registrador do sub-bloco Número x_{i-1}.

```
PROCESS(multiplicador, numero_aleatorio)
BEGIN

result_multiplicacao <= multiplicador * numero_aleatorio;

END PROCESS;
```

Figura 7.12: Código VHDL do sub-bloco de multiplicação de dois operandos

7.5.6 Soma de Operandos

Existem, no bloco Geração de Números Pseudo-Aleatórios, dois sub-blocos responsáveis pela soma de dois operandos. Na figura 7.13 apresenta-se um trecho do código VHDL deste tipo de sub-bloco, o qual possui como entrada os sinais `operando_1` e `operando_2` e como saída, o sinal `result_soma`.

```
PROCESS(operando_1, operando_2)
BEGIN

result_soma <= ('0' & operando_1) + ('0' & operando_2);

END PROCESS;
```

Figura 7.13: Código VHDL do sub-bloco de soma de dois operandos

O sinal de saída é apenas a soma dos dois sinais de entrada `operando_1` e `operando_2`. Num sub-bloco Somador, o sinal de entrada `operando_1` é o resultado da multiplicação

entre a_1 e x_{i-1} e o sinal de entrada `operando_2` é o resultado da multiplicação entre a_2 e x_{i-2}. No outro sub-bloco Somador, o sinal de entrada `operando_1` é o resultado da soma realizada sobre $a_1 * x_{i-1}$ e $a_2 * x_{i-2}$ e o sinal de entrada `operando_2` é o conteúdo armazenado no registrador do sub-bloco de Transporte c_{i-1}.

7.5.7 Memória de Endereços

Este bloco armazena os endereços enviados pelos blocos BCERPs da arquitetura proposta. Uma vez gerado o número pseudo-aleatório, o endereço armazenado na memória de endereço é utilizado para enviar um pacote contendo este número gerado para o bloco BCERP que solicitou a sua criação.

Na figura 7.14 apresenta-se um trecho do código VHDL deste bloco, o qual possui como entrada os sinais `clock`, `endereco_escrita`, `posicao_mem_escrita`, `escrita` e `posicao_mem_leitura` e como saída, o sinal `endereco_leitura`. Se houver uma transição lógica de baixo para alto no sinal `clock` e o sinal de entrada `escrita` estiver no nível lógico alto, então a memória deve armazenar, na posição indicada pelo sinal de entrada `posicao_mem_escrita` o conteúdo que está disponível no sinal de entrada `endereco_escrita`. O bloco mostra no sinal de saída `saida_controle` o conteúdo da memória que está armazenado na posição indicada pelo sinal de entrada `posicao_mem_leitura`.

```
PROCESS(clock, endereco_escrita, posicao_mem_escrita,
        escrita, posicao_mem_leitura)
TYPE memoria_endereco IS ARRAY(0 TO INDICE_MEM)
        OF STD_LOGIC_VECTOR(INDICE_ENDERECO DOWNTO 0);
VARIABLE mem_endereco: memoria_endereco;
BEGIN

IF((clock'EVENT) and (clock = '1') and (escrita = '1'))
THEN
    mem_endereco(posicao_mem_escrita) :=  endereco_escrita;
END IF;

endereco_leitura <= mem_endereco(posicao_mem_leitura);

END PROCESS;
```

Figura 7.14: Código VHDL do bloco que memoriza os endereços enviados pelos blocos BCERPs da arquitetura proposta

7.5.8 Controlador da Memória de Endereços

Este bloco gerencia as posições da memória de endereços que podem estar ocupadas ou vazias. Na figura 7.15 apresenta-se um trecho do código VHDL deste bloco, o qual possui como entrada os sinais `clock`, `inicializar`, `reservar_pos`, `posicao_mem_reservar` `liberar_pos`, `posicao_mem_liberar` e `posicao_mem_leitura` e como saída, o sinal `estado_posicao`.

```
PROCESS(clock, inicializar, reservar_pos, posicao_mem_reservar,
        liberar_pos, posicao_mem_liberar, posicao_mem_leitura)
VARIABLE posicao: STD_LOGIC_VECTOR(INDICE_MEM DOWNTO 0);
BEGIN

IF((clock'EVENT) and (clock = '1'))
THEN
    IF(inicializar = '1')
    THEN
        FOR i IN INDICE_MEM DOWNTO 0 LOOP
            posicao(i):= '0';
        END LOOP;
    ELSE
        IF(liberar_pos = '1')
        THEN
            posicao(posicao_mem_liberar) :=  '0';
        END IF;
        IF(reservar_pos = '1')
        THEN
            posicao(posicao_mem_reservar) :=  '1';
        END IF;
    END IF;
END IF;
estado_posicao <= posicao(posicao_mem_leitura);

END PROCESS;
```

Figura 7.15: Código VHDL do bloco de controle da memória de endereços

Se houver uma transição lógica de baixo para alto no sinal `clock` e o sinal `inicializar` estiver no nível lógico alto, então a memória deve ser inicializada com todas as posições de memória sendo colocadas no estado de vazias. Se houver a transição do sinal `clock`, o sinal `inicializar` estiver no nível lógico baixo e o sinal `liberar_pos` estiver no nível lógico alto, então o controlador deverá liberar a posição de memória indicada no sinal `posicao_mem_liberar`. Caso o sinal `reservar_pos` estiver no nível lógico alto o controlador deverá reservar a posição de memória indicada no sinal de entrada `posicao_mem_reservar`. O bloco mostra no sinal `estado_posicao` o estado (vazia ou ocupada) em que se encontra a posição de memória indicada pelo sinal `posicao_mem_leitura`.

7.5.9 Contadores

A unidade de geração possui dois contadores que permitem percorrem as posições da memória de endereços. O contador de pacotes enviados percorre a memória em busca de posições ocupadas para identificar os endereços dos blocos BCERPs e enviar os números pseudo-aleatórios que estão sendo criados. O contador de endereços percorre a memória para armazenar os endereços dos blocos BCERPs que estão realizando pedidos de geração de números.

Na figura 7.16 apresenta-se um trecho do código VHDL do bloco Contador de Endereços, o qual possui como entrada os sinais `clock`, `inicializar` e `incrementa` e como saída, o sinal `cont_endereco`.

```
PROCESS(clock, inicializar, incrementa)
VARIABLE contador_ender: INTEGER RANGE 0 TO INDICE_MEM;
BEGIN

IF((clock'EVENT) and (clock = '1'))
THEN
    IF(inicializar = '1')
    THEN
        contador_ender := 0;
    ELSIF(incrementa = '1')
    THEN
        IF(contador_ender = INDICE_MEM)
        THEN
            contador_ender := 0;
        ELSE
            contador_ender := contador_ender + 1;
        END IF;
    END IF;
END IF;
cont_endereco <= contador_ender;

END PROCESS;
```

Figura 7.16: Código VHDL do bloco Contador de Endereços

Se houver uma transição lógica de baixo para alto no sinal `clock` e o sinal `inicializar` estiver no nível lógico alto, então o contador deve ser inicializado com o valor zero. Caso houver uma transição no sinal `clock`, o sinal `inicializar` estiver no nível lógico baixo e o sinal `incrementa` estiver no nível lógico alto, então o contador deve incrementar uma posição no valor atualmente armazenado. Se o contador tiver atingido o limite do tamanho da memória (`INDICE_MEM`) então o valor armazenado deve ser novamente zerado. O bloco mostra no sinal `cont_endereco` o valor atualmente armazenado no contador.

7.5.10 Armazenamento de Estados

Dois blocos da unidade de geração, um responsável pelo controle de pacotes recebidos e o outro responsável pelo controle de pacotes a enviar, precisam cada um de um registrador para armazenar os possíveis estados em que eles se encontram. O bloco responsável pelo controle de pacotes recebidos pode ter oito estados, dos quais sete estados referem-se à programação da unidade de geração e um estado é referente a execução da unidade. O bloco responsável pelo controle de pacotes a enviar precisa de apenas dois estados para enviar pacotes ao sistema de roteamento da arquitetura.

Na figura 7.17 apresenta-se um trecho do código VHDL do bloco Armazenamento de Estados do Controle de Pacotes Recebidos, o qual possui como entrada os sinais `clock`, `inicializar` e `prox_estado` e como saída, o sinal `estado_atual`.

```
PROCESS(clock, inicializar, prox_estado) VARIABLE estado:
STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN

IF((clock'EVENT) and (clock = '1'))
THEN
    IF(inicializar = '1')
    THEN
        estado := "000";
    ELSE
        estado := prox_estado;
    END IF;
END IF;

estado_atual <= estado;

END PROCESS;
```

Figura 7.17: Código VHDL de armazenamento de estados do controle de pacotes recebidos

Se houver uma transição lógica de baixo para alto no sinal `clock` e o sinal `inicializar` estiver no nível lógico alto, então o estado do bloco de controle de pacotes recebidos deve ser inicializado com o valor zero. Caso houver uma transição no sinal `clock` e o sinal `inicializar` estiver no nível lógico baixo, então o próximo estado do bloco de controle de pacotes recebidos deve ser o valor que está disponível no sinal `prox_estado`. O bloco mostra no sinal `estado_atual` o conteúdo armazenado no registrador que indica o estado atual do bloco de controle.

7.5.11 Controle de Pacotes Recebidos

Este bloco controla os pedidos realizados pelos blocos BCERPs da arquitetura proposta, disponibilizando os níveis lógicos corretos em alguns sinais de controle para possibilitar o armazenamento do endereço de cada bloco BCERP numa posição disponível da memória de endereços. Na figura 7.18 apresenta-se um trecho do código VHDL deste bloco de controle.

```
PROCESS(pap_in, carga_util, estado_atual)
BEGIN

numero_config <= carga_util(INDICE_NUM DOWNTO 0);

inicializa todos os sinais de controle no nível lógico zero

CASE estado_atual IS

-- Etapa de Configuração
    WHEN "000" =>
        IF(pap_in = '1')
        THEN
            paca_out <= '1';
            ha_multip_penult_num <= '1';
            prox_estado <= "001";
        ELSE
            prox_estado <= "000";
        END IF;

    WHEN "001" => armazena segundo pacote de configuração
    ...
    WHEN "110" => armazena sexto pacote de configuração

-- Etapa de Execução
    WHEN "111" =>
        IF(pap_in = '1')
        THEN
            paca_out <= '1';
            escrita <= '1';
            endereco_escrita <= carga_util(INDICE_ENDERECO DOWNTO 0);
            incrementa <= '1';
        END IF;
        prox_estado <= "111";

    WHEN OTHERS =>

END CASE;

END PROCESS;
```

Figura 7.18: Código VHDL do bloco de controle de pacotes recebidos

Este bloco possui duas etapas, a de configuração e a de execução. Na etapa de configuração, este bloco recebe pacotes de configuração que devem ser utilizados para programar a unidade de geração de números pseudo-aleatórios. Como indicado na figura 7.6, para a configuração da unidade de geração são necessários seis pacotes de configuração e por isso, este bloco possui seis estados dedicados ao recebimento de cada um desses pacotes de configuração. Na etapa de execução, o bloco ao receber um pedido de geração de um número, deverá armazenar na memória o endereço de destino do bloco BCERP que solicitou o serviço. O código VHDL deste bloco implementa o fluxograma mostrado na figura 7.5 e possui os sinais de entrada `pap_in`, `carga_util` e `estado_atual` e como saída, disponibiliza treze sinais responsáveis pela programação da unidade de geração e pela alocação dos endereços dos blocos BCERPs nas posições disponíveis da memória.

O sinal de saída `paca_out` indica ao sistema de roteamento o tratamento de um pacote enviado. O sinal de saída `prox_estado` indica o próximo estado do controlador. Sete sinais de saída são utilizados no processo de configuração de alguns registradores da arquitetura, como o sinal `ha_multip_penult_num` que habilita o armazenamento no registrador do multiplicador a_2. Os outros seis sinais habilitam o armazenamento de a_1, x_{i-2}, x_{i-1}, c_i menos significativos, c_i mais significativos e controle do sinal de transporte. O sinal de saída `numero_config` indica o valor que deve ser armazenado nos registradores habilitados. O sinal de saída `escrita` indica um processo de escrita na memória de endereços e o sinal de saída `endereco_escrita` indica o valor que deve ser armazenado nesta memória. A posição da memória aonde ocorrerá a escrita é informada pelo contador de endereços. Por último, o sinal de saída `incrementa` indica ao contador de endereços quando ele deve ser incrementado.

7.5.12 Controle de Pacotes a Enviar

Este bloco envia os números pseudo-aleatórios gerados para os blocos BCERPs que solicitaram os pedidos, disponibilizando os níveis lógicos corretos em alguns sinais de controle.

Na figura 7.19 apresenta-se um trecho do código VHDL deste bloco de controle. Durante a etapa de execução, este bloco percorre a memória de endereços na busca por solicitações que precisam ser executadas. Assim, ao se encontrar uma posição de memória ocupada, será executada a solicitação armazenada nesta posição. Para tanto, será enviado um pacote com o número pseudo-aleatório gerado para o bloco BCERP remetente, cujo endereço se encontra armazenado na posição de memória. Ao enviar o pacote no sistema

de roteamento, o bloco de controle espera a confirmação de recebimento proveniente do roteador.

```
PROCESS(paca_in, estado_atual, estado_posicao_mem_ender,
        numero_aleatorio, endereco_leitura)
BEGIN

inicializa todos os sinais de controle no nível lógico zero

CASE estado_atual IS
    WHEN '0' =>
        IF(estado_posicao_mem_ender = '1')
        THEN
            pap  <= '1';
            prox_estado <= '1';
        ELSE
            prox_estado <= '0';
        END IF;

    WHEN '1' =>
        pacote(INDICE_ENDERECO DOWNTO 0) <= endereco_leitura;
        pacote((INDICE_NUM + INDICE_ENDERECO + 1) DOWNTO
               (INDICE_ENDERECO + 1)) <= numero_aleatorio;
        pacote((INDICE_ENDERECO + INDICE_CARGA_UTIL + 1) DOWNTO
               (INDICE_ENDERECO + INDICE_CARGA_UTIL - 1)) <= "100";

        IF(paca_in = '1')
        THEN
            liberar_pos <= '1';
            ha_exec <= '1';
            incrementa <= '1';
            prox_estado <= '0';
        ELSE
            pap <= '1';
            prox_estado <= '1';
        END IF;

    WHEN OTHERS =>

END CASE;

END PROCESS;
```

Figura 7.19: Código VHDL do bloco de controle de pacotes a enviar

O código VHDL deste bloco implementa o fluxograma mostrado na figura 7.7 e possui os sinais de entrada `paca_in`, `estado_atual`, `estado_posicao_mem_ender`, `numero_aleatorio` e `endereco_leitura` e, como saída, disponibiliza `pap`, `pacote`, `prox_estado`, `liberar_pos`, `ha_exec` e `incrementa`.

O sinal `paca_in` é utilizado para confirmar o armazenamento de um pacote enviado. O

sinal `estado_atual` indica o estado do controlador. O sinal `estado_posicao_mem_ender` refere-se ao estado da posição da memória de endereços definida no contador de pacotes enviados. O sinal `numero_aleatorio` indica o número pseudo-aleatório que foi gerado no bloco de geração. O sinal `endereco_leitura` indica o valor que está armazenado na posição da memória definida no contador de pacotes enviados. O valor armazenado é utilizado como endereço de destino do pacote contendo o número pseudo-aleatório gerado.

O sinal `pap` é utilizado para realizar um pedido de armazenamento de pacote pelo sistema de roteamento. O sinal `pacote` indica o pacote que deve ser enviado ao roteador acoplado na unidade de geração. O sinal `prox_estado` indica o próximo estado do controlador de pacotes a enviar. O sinal `liberar_pos` é direcionado ao controlador da memória de endereços e indica a liberação de uma posição de memória quando esta já tiver sido processada. O sinal `ha_exec` é direcionado ao bloco Geração de Números Pseudo-Aleatórios e habilita a criação de um novo número pseudo-aleatório. O sinal `incrementa` indica ao contador de pacotes enviados quando ele deve ser incrementado.

7.5.13 Armazenamento do Sinal PAP

Este bloco armazena o sinal PAP gerado no bloco de controle de pacotes a enviar. Na figura 7.20 apresenta-se um trecho do código VHDL deste bloco, o qual possui como entrada os sinais `clock`, `inicializar` e `pap_in`, e como saída, o sinal `pap_out`.

```
PROCESS(clock, inicializar, pap_in)
VARIABLE pap: STD_LOGIC;
BEGIN

IF((clock'EVENT) and (clock = '1'))
THEN

    IF(inicializar = '1')
    THEN
        pap := '0';
    ELSE
        pap := pap_in;
    END IF;

END IF;

pap_out <= pap;

END PROCESS;
```

Figura 7.20: Código VHDL do bloco de armazenamento do sinal PAP

Se houver uma transição lógica de baixo para alto no sinal `clock` e o sinal `inicializar` estiver no nível lógico alto, então a etapa de inicialização foi acionada e o conteúdo do registrador deve ser zerado. Caso houver transição no sinal `clock` e o sinal `inicializar` estiver no nível lógico baixo então deve-se armazenar o valor indicado no sinal `pap_in` proveniente do bloco de controle de pacotes a enviar.

7.6 Testes e Resultados

A arquitetura do gerador de números pseudo-aleatórios apresentada anteriormente foi mapeada em um CYCLONE II, EP2C50F672C6, da corporação Altera. Vários testes foram realizados utilizando diferentes valores para os multiplicadores (a_1 e a_2) e para o controle do sinal de transporte. Os resultados obtidos pelo gerador em FPGA foram confrontados com os resultados esperados, gerados a partir de um computador por meio da linguagem de programação C.

Para motivos de comparação, a arquitetura foi recompilada quatro vezes utilizando o comando `GENERIC`, o que permitiu a síntese de números pseudo-aleatórios compostos de 32, 64, 96 e 128 bits. O tamanho da memória e a quantidade de bits que representa o endereço de destino do número foram mantidos constantes em todos os mapeamentos, ou seja, memória de 20 posições e endereço de destino composto por 12 bits. Na figura 7.21 apresenta-se o tempo gasto de processamento do FPGA para a geração de um, três e seis milhões de números compostos por 32, 64, 96 e 128 bits.

A geração de números pseudo-aleatórios de 32 bits ocupou apenas 1% do total de elementos lógicos disponíveis no FPGA utilizado e 9% do total de celúlas de multiplicação, trabalhando com um freqüência interna de 58,83MHz. A geração de números de 64 bits utilizou 4% do total de elementos lógicos e 37% do total de células de multiplicação, trabalhando em uma freqüência interna de 29,38MHz. Por sua vez, a geração de números de 96 bits utilizou 9% do total de elementos lógicos e 82% do total de células de multiplicação, trabalhando a uma freqüência interna de 25,17MHz. Por último, a geração de números pseudo-aleatórios de 128 bits utilizou 29% do total de elementos lógicos e 100% do total de células de multiplicação, trabalhando a uma freqüência interna de 16,48MHz.

Por motivos de comparação, na figura 7.22 apresenta-se a geração de números pseudo-aleatórios utilizando a unidade lógica e aritmética de um Pentium 4 de 3,2GHz de freqüência e 1MB de *cache*. A geração de números pseudo-aleatórios de 32 bits no FPGA é em torno de 94 vezes mais rápida do que a geração por meio do processador Pentium 4

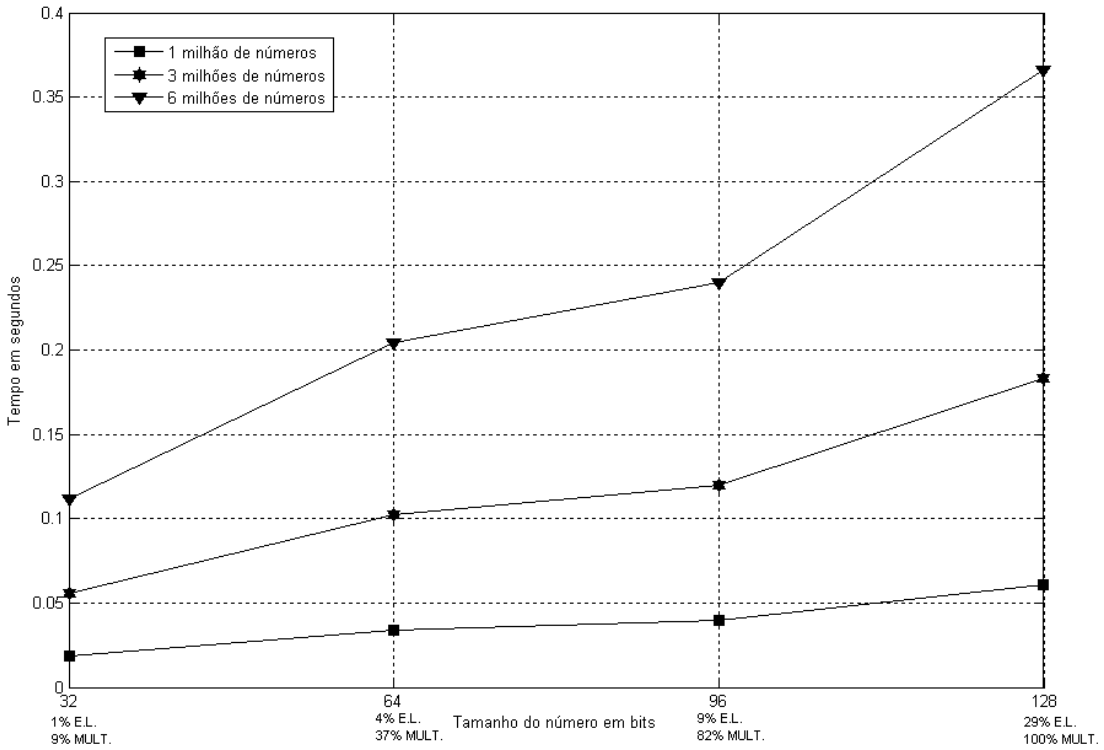


Figura 7.21: Tempo de geração no FPGA, onde E.L. significa **E**lementos **L**ógicos e MULT. significa **Mult**iplicadores

testado. Para números pseudo-aleatórios de 64 bits a proporção chega a ser de 102 vezes mais rápida; para 96 bits, a geração de números no FPGA é cerca de 300 vezes mais rápida e, para 128 bits, chega a ser 360 vezes mais rápida.

Foi utilizada a linguagem de programação C e o programa gratuito Dev-C++ para a edição e compilação do código gerado. A utilização da linguagem C, considerada de médio nível de abstração, ao invés de linguagens de mais alto nível como é o caso da linguagem FORTRAN, se deve ao fato de que na linguagem C é possível definir variáveis sem sinal e trabalhar diretamente sobre os bits dos registradores utilizando os operadores binários de deslocamento e lógico. Na figura 7.22 apresenta-se o tempo de geração de um, três e seis milhões de números pseudo-aleatórios de 32, 64, 96 e 128 bits. Vale ressaltar que a unidade lógica e arimética de inteiros do Pentium 4 trabalha com variáveis de no máximo 32 bits. Assim, foram projetados algoritmos para permitir a realização de operações de soma e multiplicação com uma quantidade maior de bits.

O bloco BCGN, com geração de números de 15 bits, foi mapeado em outros FPGAs para a verificação da quantidade de lógica gasta para a sua implementação e da quantidade de blocos BCGNs que podem ser incluídos em um FPGA. No FPGA EP1S10F780C5, da família STRATIX, um bloco BCGN utiliza apenas 235 elementos lógicos de 10570

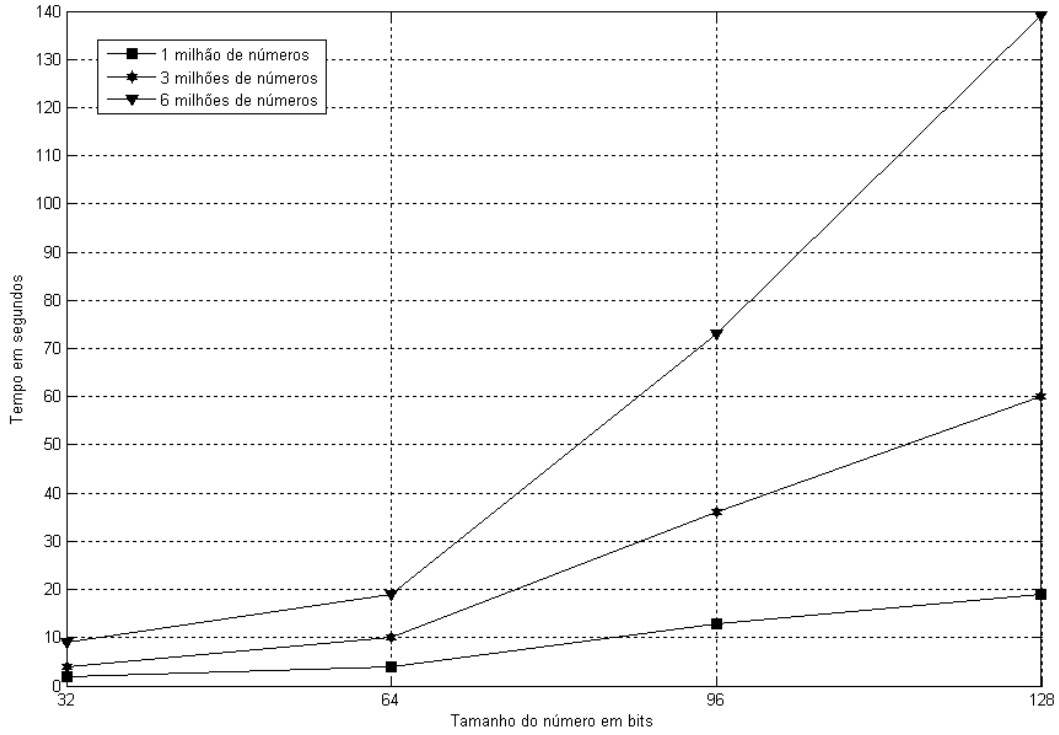


Figura 7.22: Tempo de geração no computador

disponíveis (2%). Neste FPGA foi possível o mapeamento de até 36 blocos BCGNs. Em FPGAs maiores, como o EP2C50F672C6, da família CYCLONE II, é possível o mapeamento de até 137 blocos BCGNs.

O processo de gerar números pseudo-aleatórios com uma quantidade de bits acima da quantidade implementada na unidade lógica e aritmética de inteiros de um processador é extremamente custoso, se tornando até mesmo inviável processar números com uma grande quantidade de bits ou gerar uma grande quantidade de números. Em contrapartida, a arquitetura do gerador implementada em um FPGA não utilizou uma grande quantidade de recursos totais disponíveis e foi capaz de gerar milhões de números em milésimos de segundo, mesmo para números compostos por uma grande quantidade de bits.

7.7 Comentários

Neste capítulo descreveu-se a unidade de geração de números pseudo-aleatórios reconfigurável (BCGN) da arquitetura proposta. Esta unidade de geração de números pseudo-aleatórios pode ser configurada para executar um gerador linear congruente multiplicativo, um gerador linear congruente *mixed*, um gerador linear múltiplo convencional

com ordem de recorrência igual a dois, um gerador linear múltiplo *mixed* ou um gerador linear com transporte do tipo multiplicação-com-transporte. Desta forma, dependendo do sistema a ser implementado na arquitetura proposta pode-se configurar um gerador que atenda especificamente as necessidades deste sistema. A arquitetura do gerador descrita em VHDL e mapeada em um FPGA é capaz de gerar milhões de números em milésimos de segundo mesmo para números compostos por uma grande quantidade de bits.

No próximo capítulo apresenta-se a arquitetura do bloco básico de configuração dos elementos de uma Rede de Petri (BCERP).

8 *BCERP: Bloco Básico de Configuração dos Elementos de uma Rede de Petri*

Resumo

Um bloco BCERP pode implementar uma transição da Rede de Petri e até quatro lugares de entrada dessa transição. O bloco possui um banco de memória contendo pacotes de dados que devem ser enviados se a transição que este bloco implementa for disparada. Nesses pacotes de dados encontram-se a identificação da marca e a quantidade que deve ser adicionada nos respectivos lugares de saída da transição, os quais estão implementados em outros blocos BCERPs. No processo de resolução de conflito, o bloco BCERP envia pacotes contendo o número pseudo-aleatório internamente armazenado ou, se for o caso, envia pacotes indicando a impossibilidade de disparo da transição para os blocos BCERPs que implementam as outras transições em conflito. O bloco BCERP descrito em VHDL e implementado em FPGA não utiliza muita lógica, o que permite a inclusão de diversos blocos num único FPGA.

8.1 Introdução

Os blocos BCERPs, como comentado no capítulo 5, podem ser configurados para implementar as transições da Rede de Petri e seus respectivos lugares de entrada. Os blocos BCERPs se comunicam entre si para a realização do disparo das transições. Blocos BCGNs são utilizados pelos blocos BCERPs para a geração de números pseudo-aleatórios. Estes números podem ser usados no processo de resolução de conflito e também podem ser usados para determinar se uma transição habilitada pode ser disparada, caso tenha sido definida uma transição com probabilidade de disparo.

Nas próximas seções, explicam-se o funcionamento do bloco BCERP projetado, a

arquitetura desenvolvida, a forma de configuração deste bloco e a utilização de blocos BCERPs auxiliares. Por fim, são expostos os testes realizados e os resultados obtidos.

8.2 Funcionamento do Bloco BCERP

O bloco básico de configuração dos elementos de uma Rede de Petri (BCERP) possui quatro sinais diferentes, sendo eles: `PAP` (pedido de armazenamento de pacote), `PACA` (indica o armazenamento de pacote), `PACOTE` (sinaliza os bits representantes do pacote que está sendo enviado) e `CARGA_ÚTIL` (indica somente os bits da carga útil de um pacote). Os sinais `PAP` e `PACA` são utilizados para controlar a comunicação entre o bloco BCERP e o roteador que está acoplado a este bloco. Esta comunicação permite a extração da carga útil do pacote, representado pelo sinal de entrada `CARGA_ÚTIL`. O envio de mensagens ocorre por meio do sinal de saída `PACOTE`.

Na sua forma padrão, cada bloco BCERP pode implementar uma transição da Rede de Petri e até quatro lugares de entrada dessa transição. Basicamente, o bloco BCERP possui um banco de memória contendo os pacotes de dados que devem ser enviados se a transição que este bloco implementa for disparada. Nesses pacotes de dados encontram-se a identificação da marca e a quantidade que deve ser adicionada nos respectivos lugares de saída da transição. Dessa forma, no processo de disparo da transição, enviam-se os pacotes de dados da memória para os blocos BCERPs que armazenam os lugares de saída dessa transição. O bloco permite que a transição seja disparada após um determinado intervalo de tempo, assim, no disparo da transição, enviam-se também pacotes de sincronismo de tempo para todos os blocos BCERPs que podem modificar os lugares de entrada da transição disparada.

Como comentado no capítulo 5, a arquitetura proposta permite a implementação da Rede de Petri em sua forma mais original, ou seja, todas as transições habilitadas e sem conflito podem ser disparadas simultaneamente e com relação às transições em conflito, a arquitetura projetada é capaz de escolher aleatoriamente as transições que serão disparadas. A arquitetura proposta utiliza um algoritmo distribuído capaz de definir aleatoriamente as transições a serem disparadas.

No processo de resolução de conflito, o bloco BCERP envia pacotes contendo o número pseudo-aleatório internamente armazenado ou, se for o caso, envia pacotes indicando a impossibilidade de disparo da transição nele implementada para os blocos que implementam as outras transições em conflito. Desta forma, todos os blocos BCERPs que implemen-

tam transições em conflito se comunicam uns com os outros para definir as transições que poderão ser disparadas. Um bloco BCERP em conflito receberá pacotes de impossibilidade de disparo e pacotes contendo números pseudo-aleatórios, quando houver transições aptas a disparar. Entenda-se como bloco BCERP em conflito um bloco BCERP que implementa uma transição em conflito estrutural com outras transições da Rede de Petri. Internamente, o bloco possui dois registradores, um para armazenar o seu próprio número pseudo-aleatório e outro utilizado para armazenar o menor número pseudo-aleatório encontrado entre todos os blocos em conflito que possuem transições habilitadas e que, portanto, estão na disputa para a realização do disparo. Assim, quando o bloco receber um número pseudo-aleatório proveniente de outro bloco BCERP, o número é comparado com o atualmente armazenado. Se o número pseudo-aleatório for menor do que o atualmente armazenado, então este novo número é armazenado. O processo só se finaliza quando todos os blocos BCERPs em conflito enviarem os pacotes de informação da possibilidade de disparo e os seus respectivos números. Quando todos os pacotes forem recebidos, o bloco compara o número armazenado com o seu próprio número pseudo-aleatório. Se o número comparado entre os blocos em conflito for igual ao número armazenado, então a transição deste bloco venceu a disputa e irá disparar após o processo de resolução de conflito se encerrar. Para tanto, o bloco BCERP envia pacotes de dados contendo a marca e a quantidade que deve ser subtraída dos outros blocos BCERPs. Após essa etapa de subtração de marcas, o processo inicial recomeça, ou seja, novamente os blocos BCERPs em conflito trocam informação referente a possibilidade de disparo de cada um, observando que o bloco BCERP que ganhou a disputa anterior enviará apenas pacotes de impossibilidade de disparo, pois o bloco já ganhou a disputa e a transição implementada neste bloco irá disparar após a finalização do processo de resolução de conflito. Se houver mais alguma transição que possa disparar mesmo após a remoção de marcas realizada, então o bloco BCERP que possuir o menor número pseudo-aleatório irá ganhar a nova disputa, e a transição implementada neste bloco também irá disparar. O processo só é finalizado quando não houver mais nenhuma transição apta a disparar. Neste momento, o processo de resolução de conflito se encerra e inicia-se o processo de disparo de todas as transições que venceram a disputa. O processo de disparo, ocorre como comentado anteriormente, ou seja, os blocos BCERPs que possuirem transições a serem disparadas enviam pacotes de dados contendo as identificações das marcas e as quantidades que devem ser adicionadas nos respectivos lugares de saída das transições disparadas.

Se a transição que está implementada em um bloco BCERP possuir um conflito estrutural, ou seja, a transição possuir um ou mais lugares de entrada em comum com

outras transições, então o bloco que implementa a transição em conflito na arquitetura proposta fará uso de outros três blocos BCERPs auxiliares para a realização do processo de resolução de conflito, como apresentado na figura 8.1. O bloco BCERP em conflito, quando ganha a disputa pelo disparo da transição, precisa enviar pacotes de dados contendo as subtrações das marcas para todos os outros blocos em conflito. Para tanto, o bloco que implementa a transição a ser disparada, possui um registrador contendo o endereço de outro bloco BCERP, o qual possui no seu banco de memória os endereços de todos os blocos que estão em conflito estrutural com a transição que será disparada. De maneira análoga, utiliza-se outros dois blocos auxiliares, um para enviar pacotes de dados contendo a impossibilidade de disparo da transição para todos os blocos BCERPs em conflito e o outro para enviar a possibilidade de disparo e o número pseudo-aleatório armazenado para todos os blocos em conflito.

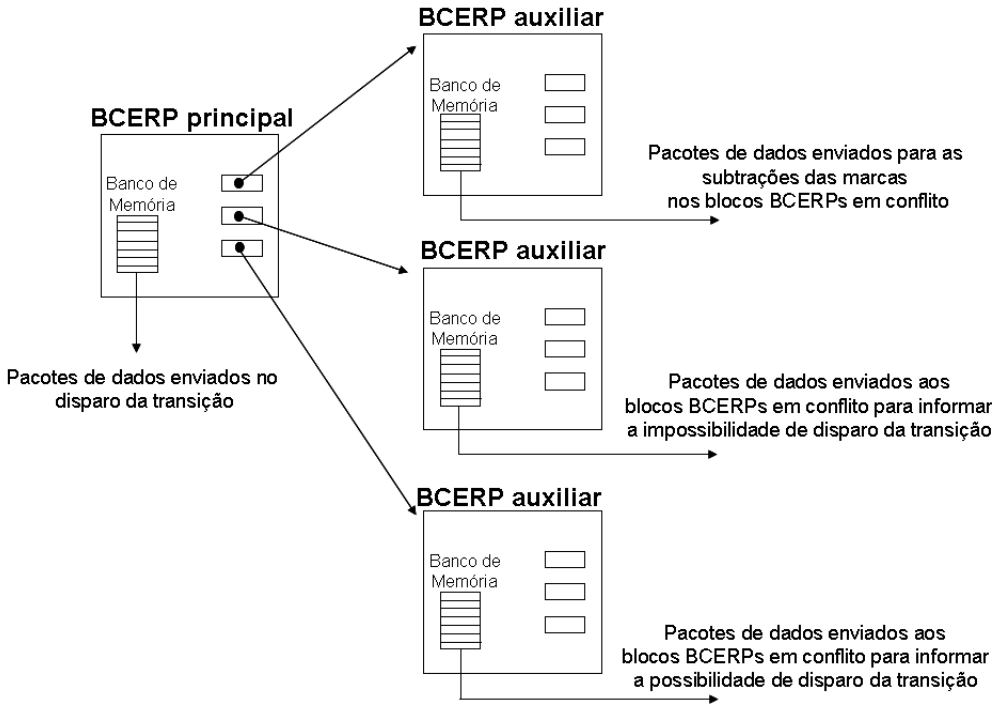


Figura 8.1: Mapeamento em blocos BCERPs de uma transição que possui conflito estrutural

Na realidade, cada bloco BCERP auxiliar implementa uma transição e um lugar de entrada. Estes elementos não existem na Rede de Petri original, são implementados na arquitetura apenas para a realização do processo de resolução de conflito do bloco BCERP principal, o qual implementa a transição original da Rede de Petri modelada. Desta forma, o bloco principal envia um pacote de dados contendo uma marca que será armazenada no bloco auxiliar. Este bloco auxiliar, ao receber esta nova marca, dispara a transição auxiliar nele implementada. Ao disparar, o bloco auxiliar envia os pacotes de dados que

se encontra no seu banco de memória, o qual poderá conter pacotes de dados referentes a impossilidade de disparo de uma transição, a possibilidade de disparo ou pacotes de dados que realizam somas ou subtrações.

Os blocos BCERPs projetados também são capazes de implementar transições com probabilidade de disparo, ou seja, o disparo pode não ocorrer quando a transição estiver habilitada. Para determinar se a transição está habilitada para o disparo, o bloco que a implementa, compara as quantidades de marcas disponíveis com as quantidades necessárias para um possível disparo. Se as marcas armazenadas forem maiores ou iguais as quantidades necessárias então a transição se encontra habilitada.

Se no processo de configuração for definido que a transição implementada no bloco BCERP possui uma probabilidade de disparo, então o número pseudo-aleatório armazenado é comparado com o conteúdo do registrador de probabilidade. Se o número pseudo-aleatório armazenado for menor do que o valor definido no registrador de probabilidade, então a transição, se não houver conflito, irá disparar. Uma vez disparada, o bloco BCERP envia um pacote de dados para um bloco BCGN solicitando a geração de um novo número pseudo-aleatório, o qual será armazenado no bloco BCERP para uma futura verificação da possibilidade de disparo. Desta forma, se for armazenado o valor zero no registrador de probabilidade, a transição habilitada nunca será disparada e se for armazenado o valor máximo, a transição sempre que habilitada será disparada. Valores intermediários no registrador de probabilidade definem a quantidade de vezes que a transição habilitada irá disparar ao longo do tempo de processamento da Rede de Petri.

8.3 Arquitetura do Bloco BCERP

O bloco de configuração BCERP possui três sinais de entrada, sendo: `PAP_IN`, `PACA_IN`, `CARGA_ÚTIL`; e três sinais de saída, sendo: `PAP_OUT`, `PACA_OUT` e `PACOTE`. O sinal `PACOTE` foi definido com 32 bits, o sinal `CARGA_ÚTIL` foi definido com 20 bits e os sinais restantes, denominados sinais de controle de comunicação, possuem apenas 1 bit cada um.

Na figura 8.2 encontra-se um esquema simplificado da arquitetura do bloco BCERP. A arquitetura deste bloco possui cinco componentes principais, quais sejam: decodificador de pacotes, análise de disparo, resolução de conflito, processo de disparo e lógica de envio de pacotes.

A arquitetura possui um banco de memória, utilizado ou para enviar pacotes de dados quando a transição implementada disparar ou para realizar o sincronismo de tempo. A

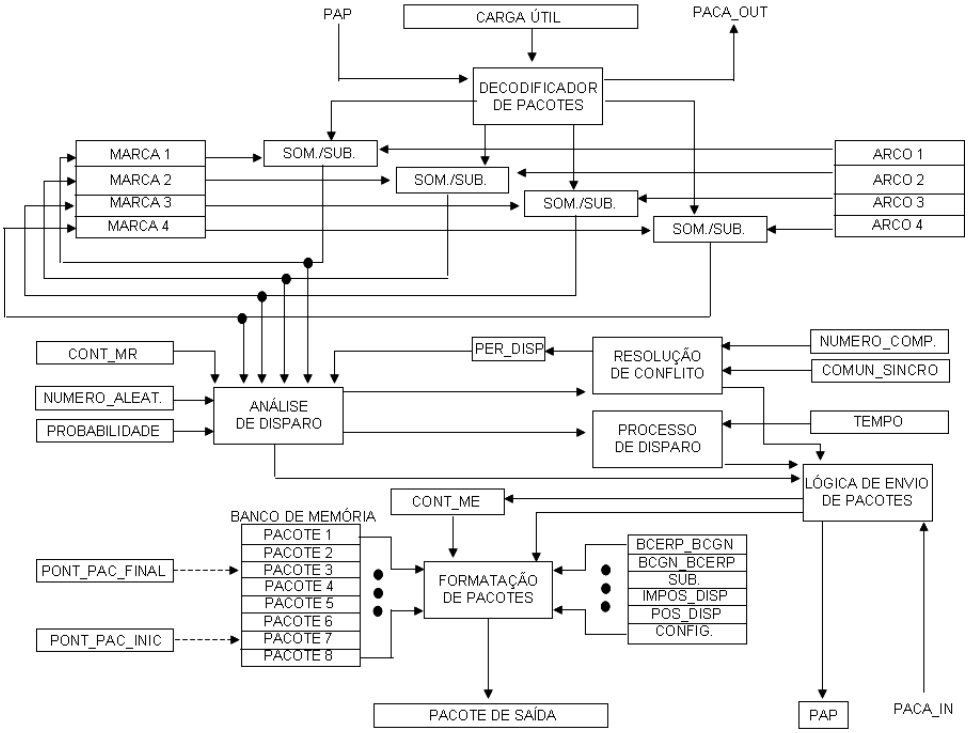


Figura 8.2: Diagrama simplificado da arquitetura do bloco BCERP

arquitetura também possui quatro somadores/subtratores, registradores para armazenar as quantidades de marcas dos lugares, registradores para armazenar as quantidades necessárias para o disparo da transição (arcos de entrada) e registradores para armazenar os endereços de blocos BCERPs auxiliares. A arquitetura possui ainda contadores de tempo, de mensagem a enviar, de mensagem a receber e de sincronismo/comunicação para a resolução de conflito. O bloco apresenta registradores para armazenar a permissão de disparo, a probabilidade de disparo da transição e os números pseudo-aleatórios provenientes de um bloco BCGN ou de outro bloco BCERP.

Os componentes de decodificação de pacotes, de análise de disparo, de resolução de conflito, de processo de disparo e de lógica de envio de pacotes, bem como todos os componentes auxiliares estão inteiramente descritos em VHDL. Ao todo foram desenvolvidos 70 arquivos VHDL, totalizando 7560 linhas de código, que constituem os 69 processos rodados na etapa de simulação. Devido a grande quantidade de código produzido nos principais componentes do bloco BCERP, foi anexado um CD contendo todos os códigos VHDL do bloco BCERP, do bloco BCGN e do roteador.

Na seqüência comenta-se sobre cada um dos cinco principais componentes do bloco BCERP.

8.3.1 Decodificador de Pacotes

O decodificador de pacotes, no processo de execução da arquitetura proposta, identifica e executa a instrução de um determinado pacote de dados. A carga útil entregue pelo roteador é dividido em três campos, como mostrado na figura 8.3.

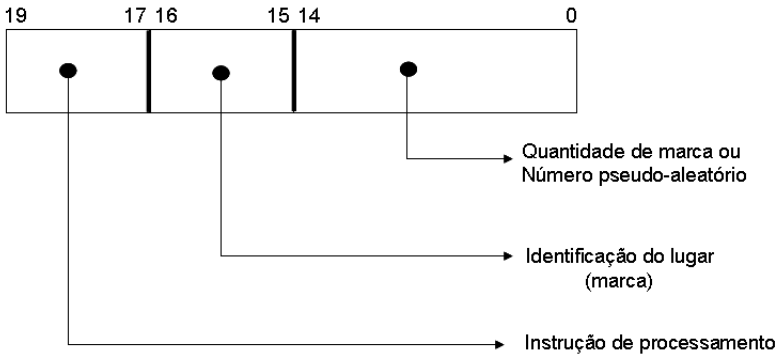


Figura 8.3: Composição da carga útil no processo de execução da Rede de Petri implementada

Os primeiros três bits indicam a instrução que deve ser realizada, os próximos dois bits referem-se a identificação do lugar (marca) e os últimos 15 bits podem indicar ou uma quantidade de marcas ou um número pseudo-aleatório. Na figura 8.4 apresentam-se as instruções de processamento. Desta forma, se o decodificador de pacotes receber, por exemplo, uma carga útil contendo a configuração "00001000000000000001", então será adicionada uma marca ("000000000000001") no registrador MARCA 2 (posição "01").

000	P	valor diferente de zero	marca(P) = marca(P) + valor, (soma)
001	P	número pseudo-aleatório	marca(P) = marca(P) + valor, (soma finalizadora)
001	—	zero	sincronismo de tempo
010	P	valor diferente de zero	marca(P) = marca(P) – valor, (subtração)
011	P	valor diferente de zero	marca(P) = marca(P) – valor, (subtração finalizadora)
011	—	zero	sincronismo de conflito
100	—	número pseudo-aleatório	número pseudo-aleatório proveniente do bloco BCGN
101	—	número pseudo-aleatório	número pseudo-aleatório proveniente do bloco BCERP
110	—	———	impossibilidade de disparo
111	—	dado	envio de dado para os endereços armazenados no banco de memória

Figura 8.4: Instruções de processamento dos pacotes de dados

Um bloco BCERP pode enviar diversos pacotes de dados para outro bloco BCERP com o intuito de adicionar ou subtrair marcas. Nestes casos, o último pacote de dados deve ser uma soma/subtração finalizadora, o que indica, para o decodificador de pacotes,

que o bloco BCERP que estava enviando os pacotes não possui mais marcas a serem computadas. Ao receber, por exemplo, uma soma finalizadora, o decodificador de pacotes decrementa o registrador Cont_MR. Este registrador indica a quantidade de blocos BCERPs que implementam as transições de entrada/saída do bloco. Desta forma, quando o registrador atingir o valor zero, sinalizando que todos os blocos BCERPs já enviaram os pacotes de dados necessários, o componente de análise de disparo será ativado para determinar a possibilidade da transição ser disparada.

Já no processo de configuração do bloco BCERP, o decodificador de pacotes identifica e realiza a programação que está sendo enviada no pacote de configuração. A carga útil entregue pelo roteador, no processo de configuração, é dividido em apenas dois campos, como mostrado na figura 8.5.

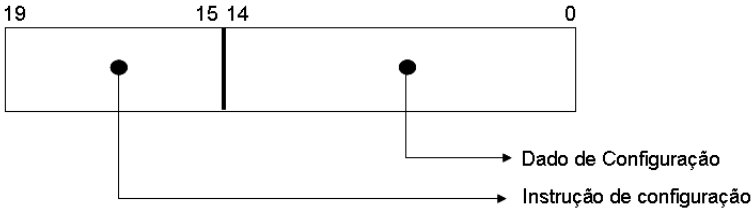


Figura 8.5: Composição da carga útil no processo de configuração da arquitetura proposta

O campo de 5 bits define a instrução de configuração e o campo de 15 bits traz os dados de configuração. Desta forma, se o decodificador receber o pacote de configuração contendo, por exemplo, a configuração "01110000000000000010", então o valor "000000000000010"será armazenado no registrador de probablidade de disparo da transição (instrução "01110").

Na tabela 8.1 apresenta-se a transição de estados do decodificador de pacotes na fase de configuração. Como pode ser observado, dependendo da instrução enviada, o decodificador armazena os dados de configuração em um determinado registrador. Atenção especial deve ser dado ao banco de memórias. Como o banco de memória possui, em cada entrada, 32 bits, não é possível configurar cada entrada de uma só vez, visto que a carga útil de um pacote possui apenas 20 bits. Desta forma, cada entrada da memória é armazenada em duas etapas. Na primeira, envia-se o cabeçalho (cab.) de 12 bits (endereço de destino dos pacotes). Na segunda etapa, armazena-se a carga útil, composta por 20 bits. Desta forma, são necessários o envio seqüencial de dois pacotes para a configuração de uma única entrada do banco de memória. Por exemplo, a instrução "10010", fará o decodificador de pacotes armazenar 12 bits do pacote recebido no cabeçalho de PAC 1 (Pacote 1) e, o próximo pacote enviado, será armazenado na carga útil de PAC 1.

Tabela 8.1: Decodificador de pacotes: tabela de transição de estados na fase de configuração

Finalizar Config.	**Estado Atual**	**Instrução de Config.**	**Ação**	**Próximo Estado**
0	R_0	00000	Marca 1 <= Dado de configuração	R_0
0	R_0	00001	Marca 2 <= Dado de configuração	R_0
0	R_0	00010	Marca 3 <= Dado de configuração	R_0
0	R_0	00011	Marca 4 <= Dado de configuração	R_0
0	R_0	00100	Arco 1 <= Dado de configuração	R_0
0	R_0	00101	Arco 2 <= Dado de configuração	R_0
0	R_0	00110	Arco 3 <= Dado de configuração	R_0
0	R_0	00111	Arco 4 <= Dado de configuração	R_0
0	R_0	01000	BCERP_BCGN <= Dado de configuração	R_0
0	R_0	01001	BCGN_BCERP <= Dado de configuração	R_0
0	R_0	01010	SUB <= Dado de configuração	R_0
0	R_0	01011	IMPOS_DISP <= Dado de configuração	R_0
0	R_0	01100	POS_DISP <= Dado de configuração	R_0
0	R_0	01101	CONFIG. <= Dado de configuração Enviar pacote de finalização de configuração	R_9
0	R_0	01110	Probabilidade <= Dado de configuração	R_0
0	R_0	01111	Numero_aleat <= Dado de configuração Numero_comp <= Dado de configuração	R_0
0	R_0	01111	MR <= Dado de configuração Tempo <= Dado de configuração	R_0
0	R_0	01111	Pont_PAC_Final <= Dado de configuração Pont_PAC_INIC <= Dado de configuração Comun_sincro <= Dado de configuração	R_0
0	R_0	10010	PAC 1(cab.) <= Dado de configuração	R_1
0	R_1	Inexistente	PAC 1(carga_util) <= Dado de configuração	R_0
0	R_0	10011	PAC 2(cab.) <= Dado de configuração	R_2
0	R_2	Inexistente	PAC 2(carga_util) <= Dado de configuração	R_0
0	R_0	10100	PAC 3(cab.) <= Dado de configuração	R_3
0	R_3	Inexistente	PAC 3(carga_util) <= Dado de configuração	R_0
0	R_0	10101	PAC 4(cab.) <= Dado de configuração	R_4
0	R_4	Inexistente	PAC 4(carga_util) <= Dado de configuração	R_0
0	R_0	10110	PAC 5(cab.) <= Dado de configuração	R_5
0	R_5	Inexistente	PAC 5(carga_util) <= Dado de configuração	R_0
0	R_0	10111	PAC 6(cab.) <= Dado de configuração	R_6
0	R_6	Inexistente	PAC 6(carga_util) <= Dado de configuração	R_0
0	R_0	11000	PAC 7(cab.) <= Dado de configuração	R_7
0	R_7	Inexistente	PAC 7(carga_util) <= Dado de configuração	R_0
0	R_0	11001	PAC 8(cab.) <= Dado de configuração	R_8
0	R_8	Inexistente	PAC 8(carga_util) <= Dado de configuração	R_0
0	R_0	11010	Cont_MR <= Dado de configuração	R_0
1	R_9	–	–	T_0

Para a finalização do processo de configuração, deve-se enviar um pacote de configuração contendo a instrução "01101". Esta instrução fará com que o bloco BCERP que está sendo programado envie um pacote de finalização de configuração para um determinado endereço de configuração, o qual foi definido no recebimento desta instrução. Desta forma, quando todos os blocos BCERPs forem configurados e retornarem os pacotes de

finalização de configuração, o sinal finalizar configuração pode ser mudado do nível lógico baixo para alto, informando a todos os blocos BCERPs da arquitetura que o processo de execução da Rede de Petri pode ser iniciado.

Na tabela 8.2 apresenta-se a transição de estados do decodificador de pacotes na fase de execução da arquitetura. Como pode ser observado, dependendo da instrução enviada, o decodificador realiza uma determinada tarefa. Quando se envia uma soma finalizadora, além de se realizar a adição da marca na posição P (definida no pacote recebido), o decodificador também decrementa o registrador Cont_MR. Um procedimento análogo ocorre para uma subtração finalizadora, que decrementa o registrador sincro. Quando o decodificador de pacotes receber a instrução "101", então o número pseudo-aleatório enviado será comparado com o número armazenado no registrador Numero_comp. Se o número enviado for menor que o armazenado, então o decodificador de pacotes armazenará o número enviado. Além disso, o registrador Comun, utilizado pelo processo de resolução de conflito, será decrementado em uma unidade. A instrução "111"fará com que o número pseudo-aleatório enviado seja retransmitido para todos os endereços armazenados no banco de memória.

Tabela 8.2: Decodificador de pacotes: tabela de transição de estados na fase de execução da Rede de Petri

Estado	**Instrução**	**Ação**	**Próximo**
T_0	000	Marca(P) <= Dado + Marca(P)	T_0
T_0	001	Marca(P) <= Dado + Marca(P) Cont_MR <= Cont_MR - 1	T_0
T_0	010	Marca(P) <= Dado - Marca(P)	T_0
T_0	011	Marca(P) <= Dado - Marca(P) Sincro <= Sincro - 1	T_0
T_0	100	chegada_num => 1 Numero_comp <= numero_chegou Numero_arm <= numero_chegou	T_0
T_0	101	Comun <= Comum - 1 Comparar numero_chegou e Numero_comp	T_1
T_1	–	Se numero_chegou ≤ Numero_comp então Numero_comp <= numero_chegou	T_0
T_0	110	Comun <= Comum - 1	T_0
T_0	111	Numero_comp <= numero_chegou Numero_arm <= numero_chegou Compor pacotes com o número e enviá-los	T_0

8.3.2 Análise de Disparo

Basicamente, o componente de análise de disparo verifica a possibilidade da transição ser disparada de acordo com os sinais gerados na arquitetura do bloco BCERP, ativando

determinados componentes para a realização ou do disparo, ou da resolução de conflito ou do sincronismo de tempo. Na tabela 8.3 encontra-se a transição de estados da análise de disparo referente ao processo de ativação dos componentes. Os sinais de entrada são a possibilidade de disparo (Pos_Disp) da transição, o estado do registrador Cont_MR, o estado do registrador de ativação de disparo (Ativ. Disp.), a indicação de probabilidade de disparo da transição e o estado atual do componente de análise. O registrador de ativação de disparo indica se a transição está em processo de disparo, pois está só será disparada após o intervalo de tempo definido ter se esgotado.

Tabela 8.3: Análise de Disparo: tabela de transição de estados referente à ativação de componentes

Estado Atual	**MR Zero**	**Ativ. Disp.**	**Conflito**	**Pos_Disp**	**Ação**	**Próx. Estado**
T_0	Não	–	–	–	–	T_0
T_0	Sim	Não	Não	–	Ativar resolução de conflito	T_4
T_0	Sim	Não	Sim	–	Liberar iteração de disparo	T_3
T_0	Sim	Sim	–	–	Verificar Pos_Disp	T_{17}
T_{17}	–	–	Não	Não	Ativar resolução de conflito	T_2
T_{17}	–	–	Sim	Não	Ativar/Liberar processo de disparo	T_1
T_{17}	–	–	Sim	Sim	Enviar sincronismo de tempo	T_8
T_{17}	–	–	Não	Sim	Ativar resolução de conflito	T_{15}

A análise de disparo também identica a necessidade de se buscar um novo número pseudo-aleatório, como mostrado na tabela 8.4. Desta forma, um novo número pseudo-aleatório será produzido pelo bloco BCGN e armazenado no bloco BCERP que o requisitou.

Tabela 8.4: Análise de Disparo: tabela de transição de estados referente a busca de um novo número pseudo-aleatório

Estado Atual	**Probabilidade**	**Marcas $\geq$ Arcos**	**Ação**	**Próximo Estado**
T_9	Sim	–	–	T_0
T_9	Não	–	Comparar Marcas e Arcos	T_{18}
T_{18}	–	Sim	Buscar novo número	T_{10}
T_{18}	–	Não	–	T_0
T_{11}	–	–	Buscar novo número	T_{12}
T_{13}	Sim	–	–	T_0
T_{13}	Não	–	Buscar novo número	T_{14}

A análise de disparo busca um novo número pseudo-aleatório quando:

- uma transição com probabilidade de disparo não estiver habilitada porque o número pseudo-aleatório armazenado é maior do que o conteúdo do registrador de probabilidade,

- uma transição com probabilidade de disparo e sem conflito estrutural disparar e,
- uma transição com conflito estrutural estiver habilitada para o disparo, independentemente do disparo ser efetivado.

Os pacotes de dados referentes ao sincronismo de tempo e a requisição de um novo número pseudo-aleatório são enviados, na realidade, pela lógica de envio de pacotes. Na tabela 8.5, encontram-se as etapas de comunicação entre os componentes da análise de disparo e da lógica de envio de pacotes.

Tabela 8.5: Análise de Disparo: tabela de transição de estados para a comunicação com a lógica de envio de pacotes

Estado Atual	**Pacotes Enviados**	**Novo Número Chegou**	**Próximo Estado**
T_7	Não	–	T_7
T_7	Sim	–	T_{11}
T_8	Não	–	T_8
T_8	Sim	–	T_9
T_{10}	–	Não	T_{10}
T_{10}	–	Sim	T_0
T_{12}	–	Não	T_{12}
T_{12}	–	Sim	T_0
T_{14}	–	Não	T_{14}
T_{14}	–	Sim	T_0
T_{16}	Não	–	T_{16}
T_{16}	Sim	–	T_9

A análise de disparo trabalha por meio da ativação e desativação de componentes. Na tabela 8.6, encontram-se as etapas de comunicação entre os componentes de análise de disparo, de resolução de conflito e de processo de disparo. Desta forma, quando a análise de disparo determinar, por exemplo, que a transição será disparada, então será ativado o componente de processo de disparo, interrompendo-se temporariamente o processamento no componente de análise de disparo. Após o processo de disparo executar a sua função, o controle é retornado para a análise de disparo que continuará o seu processamento. O mesmo procedimento ocorre para a ativação da resolução de conflito, ou seja, quando houver uma transição em conflito estrutural, a análise de disparo irá ativar o componente de resolução de conflito, o qual ao término da sua função devolve o controle de processamento para a análise de disparo.

8.3.3 Resolução de Conflito

Se a transição que está sendo implementada em um BCERP possuir um conflito estrutural, ou seja, a transição possuir um ou mais lugares de entrada em comum com outras

Tabela 8.6: Análise de Disparo: tabela de transição de estados para a comunicação com os componentes de resolução de conflito e de processo de disparo

Estado Atual	**Terminou Resol. Conf.**	**Terminou Iter. Disp.**	**Permissão Disparo**	**Ação**	**Próximo Estado**
T_1	–	Não	–	–	T_1
T_1	–	Sim	–	–	T_{13}
T_2	Não	–	–	–	T_2
T_2	Sim	–	–	–	T_3
T_3	–	–	Não	Enviar sincronismo de tempo	T_7
T_3	–	–	Sim	Ativar/Liberar processo de disparo	T_6
T_4	Não	–	–	–	T_4
T_4	Sim	–	–	Liberar iteração de disparo	T_5
T_5	–	Não	–	–	T_5
T_5	–	Sim	–	–	T_0
T_6	–	Não	–	–	T_6
T_6	–	Sim	–	–	T_{11}
T_{15}	Não	–	–	–	T_{15}
T_{15}	Sim	–	–	Enviar sincronismo de tempo	T_{16}

transições, então o componente de análise de disparo ativa o componente de resolução de conflito para a definição das transições que poderão disparar. Na ativação do componente de resolução de conflito, o registrador de permissão de disparo é zerado e o estado atual de processamento deixa de ser T_0 e passa a ser T_1. Neste instante, este componente irá executar o algoritmo distribuído comentado na seção 8.2 e apresentado na tabela 8.7.

Inicialmente, verifica-se a possibilidade interna de disparo da transição (Pos. Disp.), que ocorre quando as marcas armazenadas são maiores ou iguais às quantidades necessárias para o disparo e quando o número pseudo-aleatório armazenado é menor ou igual ao conteúdo do registrador de probabilidade, caso tenha sido definida uma transição com probabilidade de disparo. Se, internamente, a transição puder disparar, então o componente de resolução de conflito irá enviar pacotes que informam aos outros blocos BCERPs em conflito a possibilidade de disparo. A carga útil dos pacotes enviados deve conter o número pseudo-aleatório armazenado. Se, internamente, não existir a possibilidade de disparo, o componente de resolução de conflito enviará pacotes indicando a impossibilidade de disparo. Após o envio desses pacotes, o componente de resolução de conflito espera o recebimento dos pacotes de dados provenientes dos outros blocos BCERPs que também estão em conflito. A quantidade de blocos BCERPs em conflito é definida no processo de configuração e armazenada nos registradores Comun e Sincro. Desta forma, a resolução de conflito só irá para a próxima etapa após o recebimento de todos os pacotes, ou seja, quando o registrador Comun for zerado pelo componente de decodificação

Tabela 8.7: Resolução de Conflito: tabela de transição de estados

Estado Atual	**Pos. Disp.**	**Comun = 0**	**Sincro = 0**	**Num_comp = Num_arm**	**Pac. Env.**	**Ação**	**Próx. Estado**
T_1	–	–	–	–	–	Verificar Pos. Disp.	T_{15}
T_{15}	Sim	–	–	–	–	–	T_{12}
T_{15}	Não	–	–	–	–	Enviar pacote com número (pos. disp.)	T_6
T_2	–	–	–	–	Não	–	T_2
T_2	–	–	–	–	Sim	–	T_3
T_3	–	Não	–	–	–	–	T_3
T_3	–	Sim	–	–	–	Atualizar Comun Comparar Num_comp e Num_arm	T_{13}
T_{13}	–	–	–	Não	–	Num_com <= Num_arm Enviar pacote de sincronismo de conflito	T_4
T_{13}	–	–	–	Sim	–	Desativar resol. conf.	T_0
T_4	–	–	–	–	Não	–	T_4
T_4	–	–	–	–	Sim	–	T_5
T_5	–	–	Não	–	–	–	T_5
T_5	–	–	Sim	–	–	Atualizar Sincro	T_{12}
T_6	–	–	–	–	Não	–	T_6
T_6	–	–	–	–	Sim	–	T_7
T_7	–	Não	–	–	–	–	T_7
T_7	–	Sim	–	–	–	Atualizar Comun Comparar Num_comp e Num_arm	T_{14}
T_{14}	–	–	–	Não	–	Num_com <= Num_arm Enviar pacote de sincronismo de conflito	T_8
T_{14}	–	–	–	Sim	–	Num_com <= Num_arm Enviar subtrações Permissão Disp. <= 1	T_{10}
T_8	–	–	–	–	Não	–	T_8
T_8	–	–	–	–	Sim	–	T_9
T_9	–	–	Não	–	–	–	T_9
T_9	–	–	Sim	–	–	Atualizar Sincro	T_1
T_{10}	–	–	–	–	Não	–	T_{10}
T_{10}	–	–	–	–	Sim	–	T_{11}
T_{11}	–	–	Não	–	–	–	T_{11}
T_{11}	–	–	Sim	–	–	Atualizar Sincro	T_{12}
T_{12}	–	–	–	–	–	Enviar pacote de imposs. disparo	T_2

de pacotes. Nesta próxima etapa, o registrador Comun é novamente carregado com o valor inicialmente configurado e o número pseudo-aleatório armazenado (Numero_arm) é comparado com o número disponível no registrador Numero_comp. Vale ressaltar, que o decodificador de pacotes ao receber pacotes com números pseudo-aleatórios de outros blocos BCERPs compara o número enviado com o armazenado no registrador Numero_comp e o armazena se for menor do que o atualmente disponível. Se, internamente, a transição puder disparar e se Numero_arm for igual a Numero_comp, então o bloco BCERP ganhou

a disputa e a transição poderá ser disparada, para tanto, no registrador de permissão de disparo será armazenado o nível lógico alto. No término da execução do componente de resolução de conflito, a análise de disparo irá verificar o valor armazenado neste registrador e definir se a transição irá disparar. Se, internamente, a transição não puder disparar e se Numero_arm for igual a Numero_comp, então não há mais transições entre os blocos BCERPs que possam disparar, e assim encerra-se o processo de resolução de disparo e o estado atual passa a ser T_0.

No entanto, se o Numero_arm for diferente de Numero_comp, outro bloco BCERP ganhou a disputa no disparo da transição e então envia-se pacotes de dados de sincronismo de conflito e armazena-se novamente o conteúdo de Numero_arm em Numero_comp. Após está etapa, o componente de resolução de conflito espera o recebimento de pacotes de todos os outros blocos BCERPs em conflito (Sincro igual a zero). O bloco BCERP que venceu a disputa para o disparo da transição, ao invés de enviar pacotes de sincronismo de conflito, enviará pacotes de dados para a subtração das marcas em todos os outros blocos BCERPs em conflito. Quando Sincro for igual a zero, então o estado do componente de resolução de conflito irá novamente para T_1 e o processo recomeça.

8.3.4 Processo de Disparo

Quando o componente de análise de disparo identifica que a transição pode ser disparada, então armazena-se no registrador de ativação do processo de disparo e no registrador de iteração de disparo (Inter. Disp.) um nível lógico alto. Desta forma, ativa-se o componente de processo de disparo, o qual é mostrado na tabela 8.8.

Tabela 8.8: Processo de Disparo: tabela de transição de estados

Estado Atual	**Ativação = 0**	**Iter. Disp. = 0**	**Tempo = 0**	**Pac. Env.**	**Ação**	**Próximo Estado**
T_0	Não	–	–	–	Atualizar Tempo Subtrair Marcas	T_1
T_0	Sim	–	–	–	–	T_0
T_1	–	Não	–	–	–	T_1
T_1	–	Sim	–	–	–	T_2
T_2	–	–	Não	–	Enviar sincronismo de tempo Tempo <= Tempo − 1	T_4
T_2	–	–	Sim	–	Enviar pacote de disparo	T_3
T_3	–	–	–	Não	–	T_3
T_3	–	–	–	Sim	Iter. Disp. <= 0 Desativar disparo	T_0
T_4	–	–	–	Não	–	T_4
T_4	–	–	–	Sim	Iter. Disp. <= 0	T_1

Uma vez ativado, o componente de processo de disparo subtrai as marcas interna-

mente armazenadas nas quantidades necessárias para o disparo. Depois dessa etapa, o componente de processo de disparo verifica o conteúdo do registrador de tempo. Se o conteúdo do registrador de tempo for zero, então serão enviados os pacotes armazenados no banco de memória e o processo de disparo se encerra, ou seja, o conteúdo dos registradores de ativação e de iteração de disparo passam a ser zero. Se o conteúdo do registrador de tempo for diferente de zero, então pacotes de sincronismo de tempo são enviados para blocos BCERPs, o registrador de tempo é decrementado em uma unidade e o processo de iteração não se encerra, mas retorna o controle de processamento para o componente de análise de disparo, ou seja, o conteúdo do registrador de iteração de disparo é zerado. Posteriormente, o componente de análise de disparo armazenará novamente um nível lógico alto no registrador de iteração de disparo, o que fará com que o processo de disparo seja novamente executado. Este procedimento ocorre até que o conteúdo do registrador de tempo seja zerado, ocasionando o disparo da transição.

8.3.5 Lógica de Envio de Pacotes

A lógica de envio de pacotes é responsável por toda a comunicação do bloco BCERP com outros blocos BCERPs ou BCGNs. Esta lógica recebe tarefas (funções) provenientes dos componentes anteriormente explicados, como esquematizado na tabela 8.9. Dependendo da tarefa solicitada, este componente realiza a formatação dos pacotes de saída e os envia para o roteador que está acoplado ao bloco BCERP. O registrador Cont_ME é utilizado para enviar dados armazenados em posições diferentes do banco de memória.

Tabela 8.9: Lógica de Envio de Pacotes: tabela de transição de estados para a funções solicitadas

Estado Atual	**Função**	**Ação**	**Próximo Estado**
T_0	Enviar pacotes de disparo	Cont_ME <= Pont_Pac_Final PAP <= 1	T_1
T_0	Enviar sincronismo de tempo	Cont_ME <= Pont_Pac_Inic PAP <= 1	T_{10}
T_0	Enviar pacotes de subtrações	PAP <= 1	T_3
T_0	Enviar sincronismo de conflito	PAP <= 1	T_4
T_0	Buscar número pseudo-aleatório	PAP <= 1	T_5
T_0	Enviar possibilidade de disparo	PAP <= 1	T_6
T_0	Enviar impossibilidade de disparo	PAP <= 1	T_7
T_0	Enviar pacotes com número recebido	Cont_ME <= Pont_Pac_Inic PAP <= 1	T_8
T_0	Enviar finalização de configuração	PAP <= 1	T_{12}

Na tabela 8.10 encontra-se o processo de formatação dos pacotes de dados, os quais são enviados para o roteador acoplado ao bloco BCERP.

Tabela 8.10: Lógica de Envio de Pacotes: tabela de transição de estados para a formatação dos pacotes

Estado Atual	**PACA = 0**	**Cont_ME = 0**	**Ação**	**Próximo Estado**
T_1	Não	–	–	T_1
T_1	Sim	–	Pacote de saída: 1, PAP <= 0	T_2
T_2	–	Não	PAP <= 1, Cont_ME <= Cont_ME − 1	T_1
T_2	–	Sim	Finalizar processo	T_0
T_3	Não	–	–	T_3
T_3	Sim	–	Pacote de saída: 3, PAP <= 0, Finalizar processo	T_0
T_4	Não	–	–	T_4
T_4	Sim	–	Pacote de saída: 4, PAP <= 0, Finalizar processo	T_0
T_5	Não	–	–	T_5
T_5	Sim	–	Pacote de saída: 5, PAP <= 0, Finalizar processo	T_0
T_6	Não	–	–	T_6
T_6	Sim	–	Pacote de saída: 6, PAP <= 0, Finalizar processo	T_0
T_7	Não	–	–	T_7
T_7	Sim	–	Pacote de saída: 7, PAP <= 0, Finalizar processo	T_0
T_8	Não	–	–	T_8
T_8	Sim	–	Pacote de saída: 8, PAP <= 0,	T_9
T_9	–	Não	PAP <= 1, Cont_ME <= Cont_ME − 1	T_6
T_9	–	Sim	Finalizar processo	T_0
T_{10}	–	Não	–	T_{10}
T_{10}	–	Sim	Pacote de saída: 2, PAP <= 0	T_{11}
T_{11}	–	Não	PAP <= 1, Cont_ME <= Cont_ME − 1	T_{10}
T_{11}	–	Sim	Finalizar processo	T_0
T_{12}	Não	–	–	T_{12}
T_{12}	Sim	–	Pacote de saída: 9, PAP <= 0, Finalizar processo	T_0

Para se enviar, por exemplo, os pacotes de disparo, a lógica envia o pacote que está armazenado na posição Cont_ME do banco de memória. Após o recebimento do pacote de dados pelo roteador, o registrador Cont_ME é decrementado em uma unidade e o pacote armazenado na posição Cont_ME do banco de memória é então enviado ao roteador. O processo se repete até que o conteúdo do registrador Cont_ME seja zero.

Os pacotes de saída foram definidos com números de 1 até 9 e cada número representa uma formatação diferente, como apresentado na figura 8.6.

Desta forma, o pacote de saída 6, por exemplo, indica que o campo de instrução de processamento deve ser “111”, que o endereço do pacote a ser enviado está armazenado no registrador Pos_disp e que a carga útil deve ter o número pseudo-aleatório internamente armazenado.

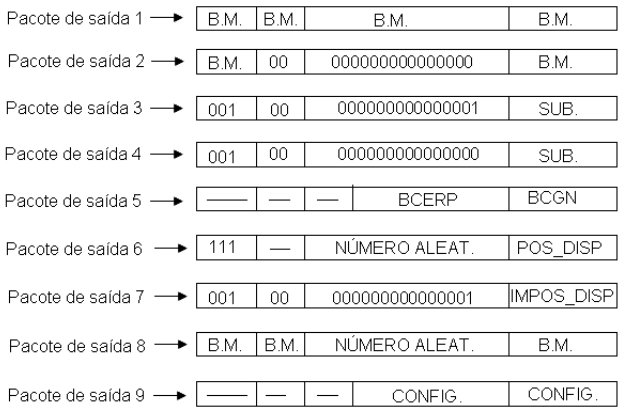


Figura 8.6: Formatação dos pacotes de saída, onde B.M. indica um dado proveniente do **B**anco de **M**emória

8.4 Utilização de Blocos BCERPs Auxiliares

Cada bloco BCERP, mostrado na figura 8.2, possui um banco de quatro registradores de 15 bits cada (Marca 1 até Marca 4) para o armazenamento das marcas geradas devido ao disparo de determinadas transições da Rede de Petri. Cada registrador é responsável pelo armazenamento de um tipo diferente de marca. Assim sendo, cada bloco BCERP pode armazenar até quatro tipos diferentes de marcas e cada tipo poderá ter de 0 até $2^{15} - 1$ marcas. O bloco BCERP também possui outro banco de quatro registradores de 15 bits cada (Arco 1 até Arco 4) para o armazenamento dos valores dos arcos de entrada do modelo de Rede de Petri que está sendo mapeado. O banco de memória armazena até oito palavras de 32 bits cada.

Apesar de cada bloco BCERP possuir um banco de registradores capaz de armazenar apenas quatro tipos diferentes de marcas de entrada, a arquitetura pode ser configurada para possibilitar o mapeamento de transições sem conflito e sem probabilidade de disparo que possuam mais do que quatro tipos diferentes de marcas de entrada. Para isso, faz-se uso de blocos BCERPs auxiliares que implementam transições intermediárias denomeadas transições de convergência, como mostrado na figura 8.7.

Na realidade, cada bloco BCERP auxiliar implementa uma transição e um ou mais lugares de entrada. Estes elementos não existem na Rede de Petri original, são implementados na arquitetura apenas para permitir o mapeamento de transições da Rede de Petri que não podem ser mapeadas em um único bloco BCERP.

Como mostrado na figura 8.7, cada transição de convergência do nível N dispara após receber uma determinada quantia de marcas. O disparo de quatro transições de convergência do nível N irá disparar uma transição de convergência de um nível imedia-

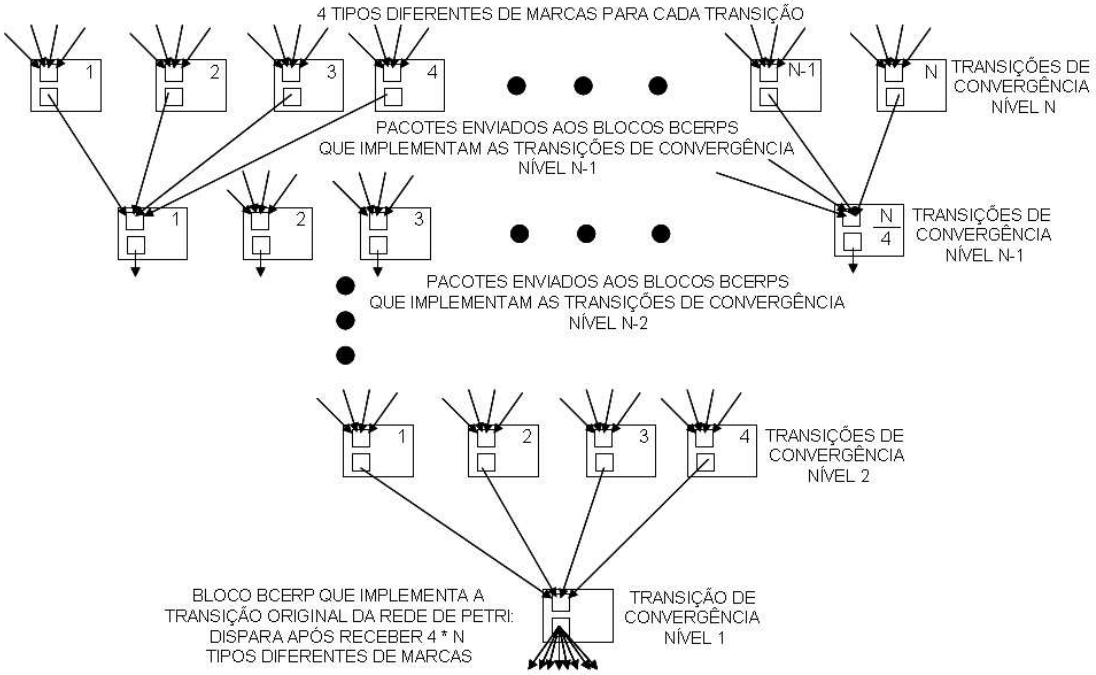


Figura 8.7: Blocos BCERPs auxiliares que implementam transições de convergência para possibilitar o mapeamento de um número maior de tipos diferentes de marcas

tamente inferior (nível N-1) e assim sucessivamente até que se atinja a última transição de convergência de nível 1. Desta forma, a transição de convergência de nível 1, na realidade, só irá disparar após o recebimento de $4 * N$ tipos diferentes de marcas nas suas quantidades pré-estabelecidas.

Originalmente, a arquitetura também teria uma restrição quanto ao número de pacotes que cada bloco BCERP poderia enviar, visto que o banco de memória é capaz de armazenar até oito pacotes de dados. Porém, a arquitetura dos blocos BCERPs permite o mapeamento de transições que precisem enviar um número maior de pacotes para o sistema de roteamento. Para isso, faz-se uso de blocos BCERPs auxiliares que implementam transições intermediárias denomeadas transições escravas, como mostrado na figura 8.8.

O bloco BCERP que implementa a transição original da Rede de Petri, quando tiver marcas em quantidades e qualidades suficientes para disparar envia oito pacotes para os blocos BCERPs auxiliares de nível 1, os quais implementam as transições escravas de nível 1. Quando um bloco BCERP auxiliar de nível 1 receber um pacote de dados, a transição escrava será disparada e o bloco BCERP enviará oito pacotes para os blocos BCERPs de nível imediatamente inferior (nível 2). Assim, o bloco BCERP que implementa a transição original juntamente com os blocos BCERPs auxiliares de nível 1 até N são capazes de enviar até 8^N pacotes de dados.

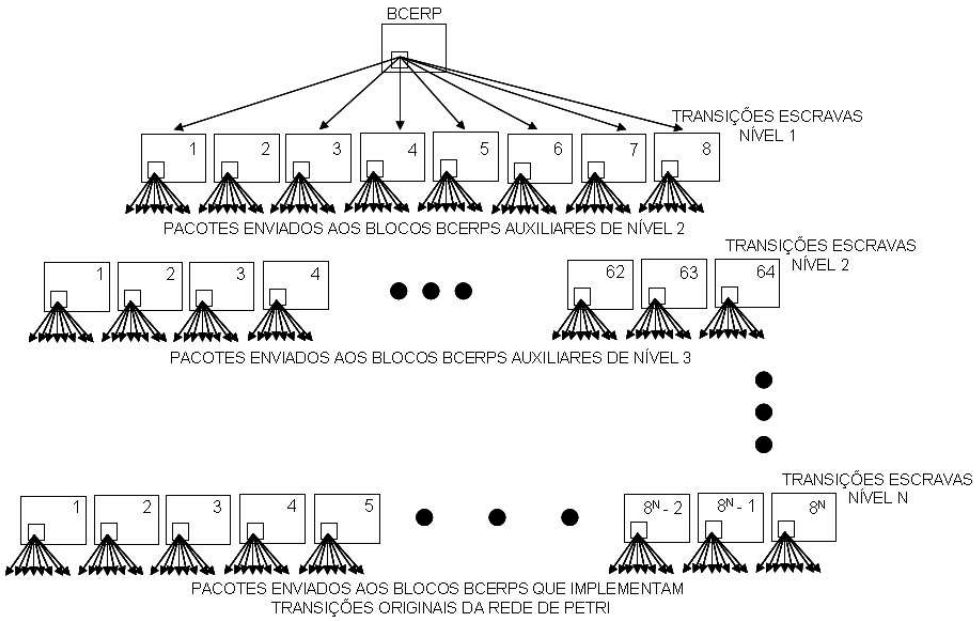


Figura 8.8: Blocos BCERPs escravos para possibilitar o envio de um número maior de pacotes de dados

No processo de resolução de conflito, os blocos BCERPs auxiliares também podem utilizar outros blocos BCERPs auxiliares para o envio de pacotes. Na figura 8.9 apresenta-se a configuração de um bloco BCERP auxiliar que faz uso de outro bloco auxiliar para o envio de pacotes de dados que realizam as subtrações das marcas.

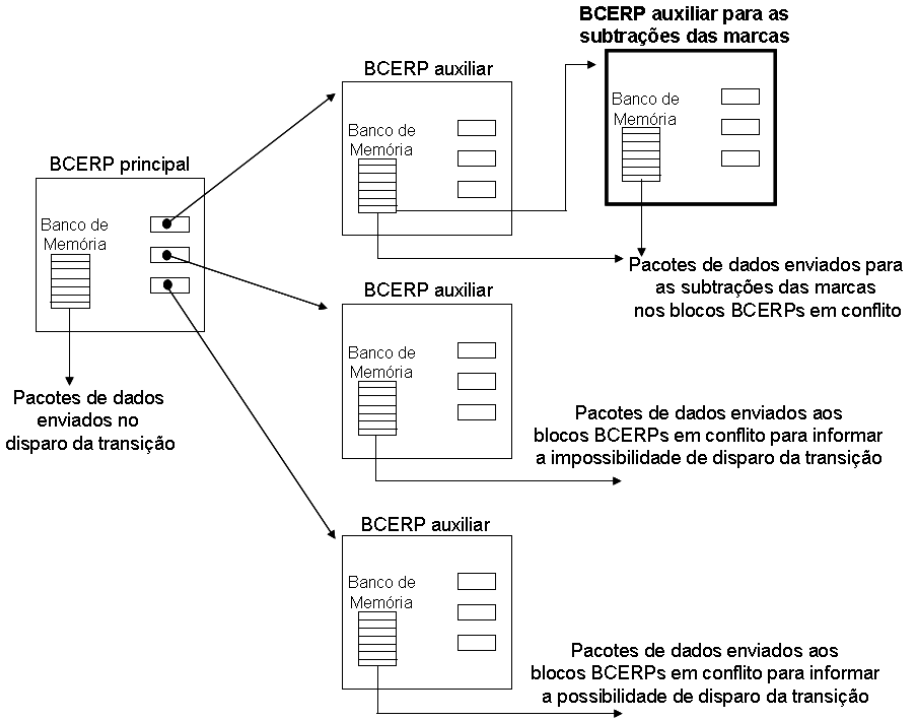


Figura 8.9: Bloco BCERP auxiliar para as subtrações das marcas no processo de resolução de conflito

8.5 Síntese, Simulação e Teste do Bloco BCERP

Utilizando o *software* Quartus II Web Edition versão 5.0, o bloco BCERP foi mapeado no FPGA EPF10K70RC240. De 3744 elementos lógicos disponíveis, 1625 (43%) foram usados, e de 189 pinos, utilizaram-se 59 (31%). No processo de simulação, foi realizado o procedimento definido a seguir. Deve-se gerar um sinal de relógio e inicializar o bloco BCERP (sinal inicializar no nível lógico alto). Depois, deve-se enviar os pacotes de configuração para que o bloco seja devidamente programado. Após o envio de todos os pacotes de configuração, deve-se esperar o retorno de um pacote de finalização de configuração. Ao receber este pacote, deve-se enviar um nível lógico alto no sinal `finalizar_configuracao`, indicando o final da etapa de configuraçao. Terminada a etapa de configuração, deve-se enviar pacotes de dados de sincronismo de tempo ou pacotes de dados para a adição de marcas nos lugares de entrada implementados no bloco BCERP e, observar o comportamento do bloco com relação ao disparo da transição mapeada, ou seja, deve-se esperar pacotes de dados contendo informações referentes ao disparo da transição, a busca de número pseudo-aleatório, ao sincronismo de conflito, entre outras possibilidades.

No processo de teste do bloco BCERP, utilizou-se a placa educacional UP2 que possui o FPGA EPF10K70RC240. No processo de síntese realizado no bloco BCERP, o código VHDL descrito foi mapeado nesta placa UP2. Devido a quantidade de pinos de entrada do bloco BCERP (25 pinos), foram confeccionados módulos em VHDL que implementam duas memórias RAMs, como apresentado na figura 8.10.

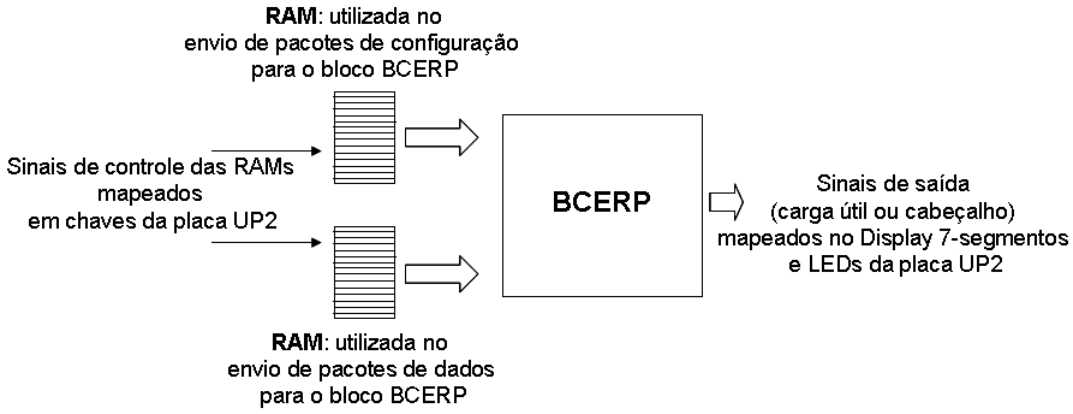


Figura 8.10: Sistema utilizado para o teste do bloco BCERP no FPGA

As memórias são utilizadas para o envio de pacotes de configuração e de dados nas etapas de programação e de processamento da arquitetura proposta. Os módulos foram devidamente implementados no FPGA e interligados ao bloco BCERP. No exemplo da tabela 8.11 encontram-se os pacotes de configuração enviados, por meio da memória RAM,

para a programação do bloco BCERP. Esta configuração refere-se a um lugar de entrada que possui uma marca armazenada e a uma transição que não está em conflito estrutural com outras transições e não possui probabilidade de disparo. No processo de análise de disparo da transição, o bloco BCERP configurado se comunica com três blocos BCERPs.

Tabela 8.11: Pacotes de configuração armazenados na memória RAM

Instrução	**Dado de Configuração**	**Ação**
00000	000000000000001	Marca 1 <= 1
00001	000000000000000	Marca 2 <= 0
00010	000000000000000	Marca 3 <= 0
00011	000000000000000	Marca 4 <= 0
00100	000000000000010	Arco 1 <= 2
00101	000000000000000	Arco 2 <= 0
00110	000000000000000	Arco 3 <= 0
00111	000000000000000	Arco 4 <= 0
01110	111111111111111	Sem probab.
01111	000000000000110	Num_arm <= 6
10000	000000100000100	MR <= 2 Tempo <= 1
11010	000000010000000	Cont_MR <= 1
10001	000000000100010	Sem conflito Pont_Pac_Final <= 1 Pont_Pac_Inic <= 2
10010	000000000010000	Pac 1(cab.) <= 2 Pac 1(carga útil) <= 4
10011	000000000001000	Pac 2(cab.) <= 1 Pac 2(carga útil) <= 5
10100	000000000011000	Pac 3(cab.) <= 3 Pac 3(carga útil) <= 6
01101	000000000111000	Config. <= 7

Na etapa de execução do bloco BCERP, deve-se enviar pacotes de dados de sincronismo de tempo e pacotes de dados para a adição de marcas no lugar de entrada implementado e observar o comportamento do bloco com relação ao disparo da transição mapeada. No exemplo da tabela 8.12 encontram-se os pacotes de dados que foram enviados, por meio da memória RAM, para a realização deste teste no bloco BCERP configurado.

Tabela 8.12: Pacotes de dados armazenados na memória RAM

Inst. Proc.	**Ident. Lugar**	**Dado**	**Ação**
001	00	000000000000001	Marca 1 <= Marca 1 + 1 (finalizador)
000	00	000000000000001	Marca 1 <= Marca 1 + 1
001	00	000000000000000	Sincronismo de tempo

Vários testes foram realizados no bloco BCERP modificando-se os valores armazenados nas posições das memórias RAMs para o mapeamento de transições diferentes, ou seja, com/sem probabilidade de disparo e com/sem conflito estrutural. Diferentes pacotes

também foram armazenados para a verificação do comportamento do bloco BCERP ao receber pacotes de sincronismo de tempo, de conflito, pacotes de disparo, de impossibilidade de disparo, pacotes que carregam números pseudo-aleatórios, entre outras possibilidades.

O bloco BCERP foi mapeado em outros FPGAs para a verificação da quantidade de lógica gasta para a sua implementação e da quantidade de blocos BCERPs que podem ser incluídos em um FPGA. No FPGA EP1S10F780C5, da família STRATIX, um bloco BCERP utiliza apenas 10% de elementos lógicos disponíveis e em FPGAs maiores, como o EP2C70F896C8, da família CYCLONE II, utilizou-se apenas 1116 elementos lógicos de 68416 disponíveis (1%). Neste FPGA é possível o mapeamento de até 60 blocos BCERPs.

8.6 Comentários

Neste capítulo foi descrito o bloco básico de configuração dos elementos de uma Rede de Petri (BCERP). A arquitetura do bloco BCERP possui cinco componentes digitais capazes de realizar a decodificação de pacotes, a análise de disparo, a resolução de conflito, o disparo da transição e o envio de pacotes para o sistema de comunição. A arquitetura possui um banco de memória, utilizado ou para enviar pacotes de dados quando a transição implementada disparar ou para realizar o sincronismo de tempo. A arquitetura também possui quatro somadores/subtratores, registradores para armazenar as quantidades de marcas dos lugares, registradores para armazenar as quantidades necessárias para o disparo da transição (arcos de entrada) e registradores para armazenar os endereços de blocos BCERPs auxiliares. A arquitetura possui ainda contadores de tempo, de mensagem a enviar, de mensagem a receber e de sincronismo/comunicação para a resolução de conflito. O bloco apresenta registradores para armazenar a permissão de disparo, a probabilidade de disparo da transição e os números pseudo-aleatórios provenientes de um bloco BCGN ou de outro bloco BCERP. A implementação do bloco BCERP em FPGAs modernas não utiliza muita lógica, o que permite a inclusão de diversos blocos BCERPs num único FPGA.

No próximo capítulo comentam-se sobre os processos de simulação e de teste da arquitetura proposta.

9 *Simulação e Teste da Arquitetura Proposta*

Resumo

Uma arquitetura com nove roteadores interligando oito blocos BCERPs e um bloco BCGN foi mapeada no FPGA EPF10K70RC240 e utilizou um total de 3224 elementos lógicos disponíveis (86%). Devido a pequena quantidade de lógica do FPGA disponível para a realização dos testes, os pacotes de dados e de configuração foram reduzidos de 32 bits para 17 bits. Alguns modelos de Redes de Petri contendo transições com/sem conflito estrutural e com/sem probabilidade de disparo foram descritos e implementados na arquitetura. Pacotes de configuração foram enviados, por meio de uma memória ROM, a todos os blocos BCERPs e ao bloco BCGN da arquitetura. Pacotes de sincronismo de tempo e de adição de marcas de tipos diferentes foram enviados, por meio de outra memória ROM, para a verificação do comportamento da arquitetura de topologia 3x3 definida.

9.1 Introdução

Para a simulação e teste da arquitetura proposta foi definida, no editor de símbolos do Quartus II, uma matriz 3x3 de roteadores para interligar oito blocos BCERPs e um bloco BCGN. Na figura 9.1 encontra-se o diagrama de blocos da topologia utilizada.

Os blocos BCERPs foram utilizados para permitir a implementação de 8 transições e até 32 lugares de uma Rede de Petri. O bloco BCGN permitiu a simulação e teste de transições que possuem probabilidade de disparo e/ou que estejam em conflito comportamental.

Pacotes de configuração foram enviados aos blocos BCERPs e BCGN por meio do canal de comunicação Oeste do roteador(3,1). Os pacotes de dados, na etapa de execução

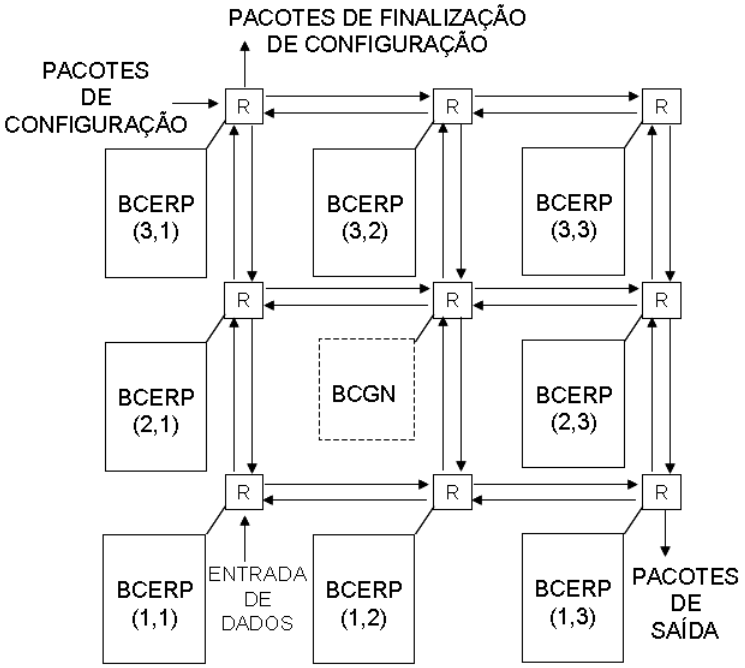


Figura 9.1: Definição de uma arquitetura 3x3 utilizada nos processos de simulação e de teste

da Rede de Petri mapeada, foram enviados por meio do canal de comunicação Sul do roteador(1,1). O comportamento da arquitetura foi observado pelo recebimento de pacotes de saída, os quais eram disponibilizados no canal de comunicação Sul do roteador(1,3).

Nas próximas seções, comentam-se sobre os processos de simulação e de teste do sistema de roteamento e do sistema completo, composto pelo conjunto de roteadores, pelo conjunto de blocos BCERPs e pelo bloco BCGN da arquitetura 3x3 definida.

9.2 Simulação e Teste do Sistema de Roteamento

Antes da configuração dos blocos BCERPs e do bloco BCGN, o sistema de comunicação da arquitetura 3x3 da figura 9.1 foi simulado no *software* Quartus II, mapeado e testado no FPGA EPF10K70RC240, família FLEX 10K, da placa UP2. Vários pacotes de dados com diferentes endereços de destino foram inseridos em alguns roteadores da arquitetura. Na figura 9.2, encontram-se alguns pacotes de dados que foram inseridos no sistema de roteamento da arquitetura proposta.

Neste exemplo, cinco pacotes foram enviados a partir de diferentes roteadores. Os endereços de destino foram escolhidos de tal forma que houvesse concorrência entre os pacotes pela utilização de um mesmo canal de comunicação. Desta forma, no roteador

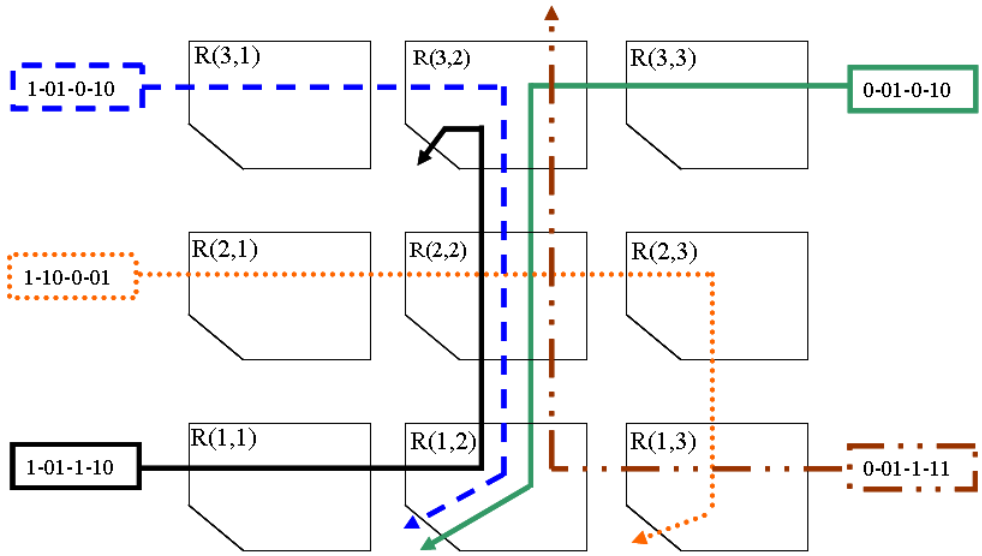


Figura 9.2: Pacotes inseridos no sistema de comunicação da arquitetura 3x3

R(3,2), por exemplo, os dois pacotes com os endereços de destino "101010"e "001010", disputaram o acesso ao canal de comunicação do Sul e, os dois pacotes com os endereços de destino "101110"e "001110"disputaram o canal de comunicação do Norte.

9.3 Exemplo de uma Rede de Petri Mapeada na Arquitetura Proposta

Na figura 9.3 apresenta-se a descrição da Rede de Petri utilizada nos processos de simulação e de teste da arquitetura implementada no FPGA. A Rede de Petri possui nove lugares e seis transições. Definiu-se, para cada transição, um determinado tempo lógico de disparo: T1 = 1 unidade de tempo, T2 = 2 un., T3 = 3 un., T4 = 1 un., T5 = 2 un. e T6 = 3 unidades. Para a transição T1 definiu-se uma probabilidade de disparo de 98%. Apesar da transição quase sempre disparar, essa probabilidade de disparo foi utilizada para verificar o processo de comunicação entre o bloco BCERP e o bloco BCGN na busca por números pseudo-aleatórios. Neste exemplo, as transições, T2, T3, T4, T5 e T6 foram definidas sem probabilidade de disparo, ou seja, essas transições, sempre que habilitadas, poderiam disparar.

A Rede de Petri da figura 9.3 foi mapeada nos blocos BCERPs da arquitetura, como mostrado na figura 9.4. A transição T1 e o seu respectivo lugar de entrada L1 foram mapeados no bloco BCERP(1,1). A transição T2 e o seu respectivo lugar de entrada L2 foram mapeados no bloco BCERP(1,2).

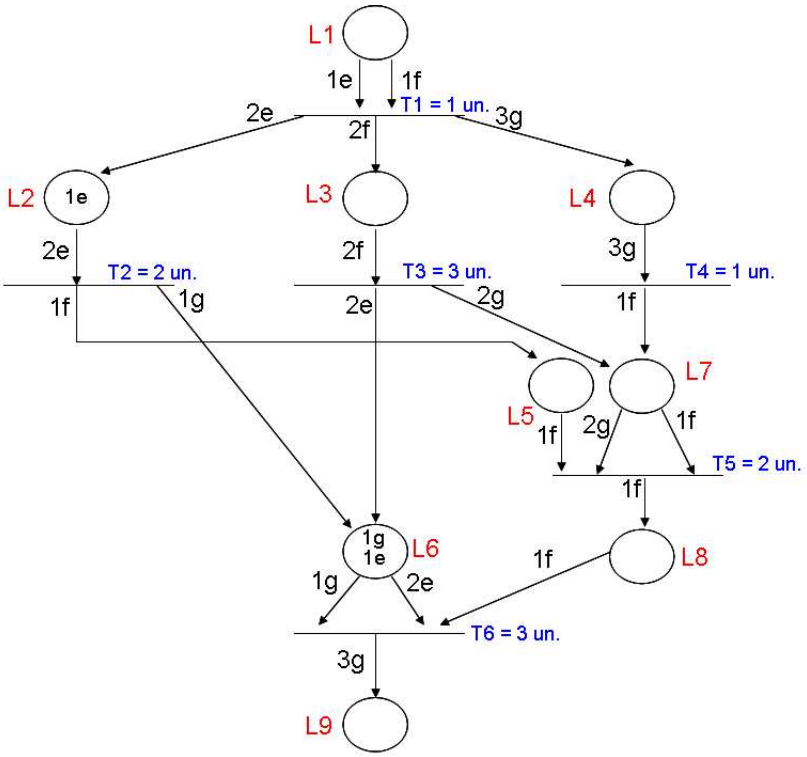


Figura 9.3: Descrição de uma Rede de Petri utilizada nos processos de simulação e de teste da arquitetura proposta

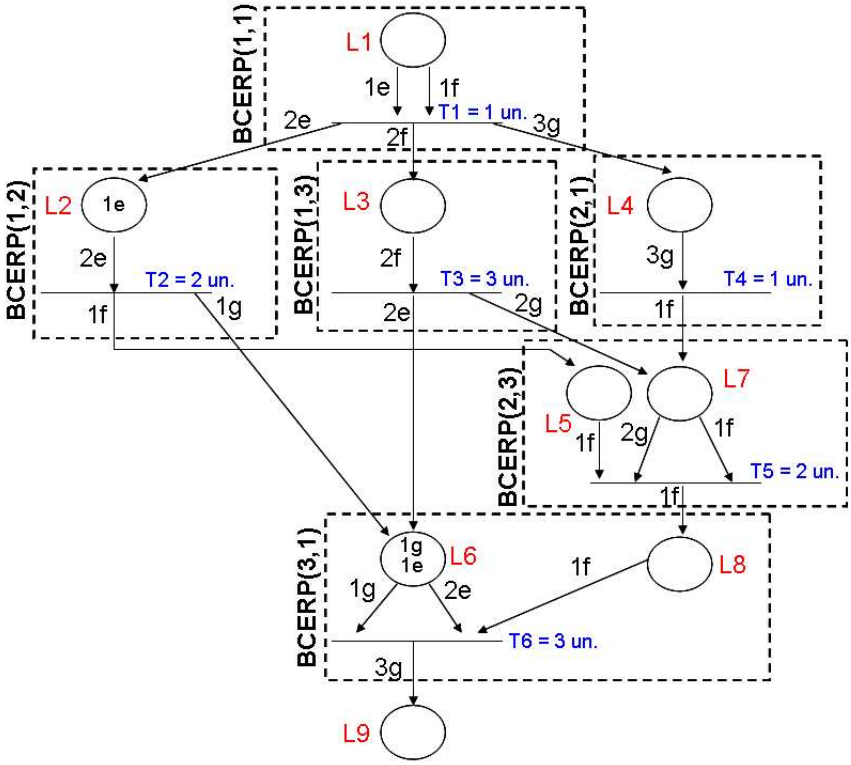


Figura 9.4: Mapeamento da Rede de Petri na arquitetura proposta

No bloco BCERP(1,3), foram mapeados a transição T3 e o lugar L3. A transição T4 e o lugar L4 foram alocados no bloco BCERP(2,1). No bloco BCERP(2,3), foram alocados a transição T5 e os seus respectivos lugares de entrada L5 e L7. Por último, o bloco BCERP(3,1) foi configurado para implementar a transição T6 e os seus respectivos lugares de entrada L6 e L8.

9.3.1 Configuração do Bloco BCERP(1,1)

No bloco BCERP(1,1) foram mapeados a transição T1 e o lugar L1. O lugar L1 pode conter até dois tipos diferentes de marcas, denominadas *e* e *f*. A transição T1 estará habilitada para o disparo se o lugar L1 possuir uma marca do tipo *e* e uma marca do tipo *f*. O disparo da transição produz duas marcas do tipo *e*, no lugar L2, duas marcas do tipo *f*, no lugar L3, e três marcas do tipo *g*, no lugar L4. Na tabela 9.1 apresentam-se os pacotes de configuração que foram enviados ao bloco BCERP(1,1).

Tabela 9.1: Pacotes de configuração do BCERP(1,1)

Instrução	**Dado de Configuração**	**Definição**
00000	000000	Marca 1
00001	000000	Marca 2
00010	000000	Marca 3
00011	000000	Marca 4
00100	000001	Arco 1
00101	000001	Arco 2
00110	000000	Arco 3
00111	000000	Arco 4
01110	111110	Com probab.
01111	000001	Num_arm
01000	101101	BCERP - BCGN
01001	001001	BCGN - BCERP
10000	010001	MR Tempo
11010	000100	Cont_MR
10001	001111	Sem conflito Pont_Pac_Final Pont_Pac_Inic
10010	101000	Pac 1(cab.) Pac 1(carga útil)
10011	110000	Pac 2(cab.) Pac 2(carga útil)
10100	000101	Pac 3(cab.) Pac 3(carga útil)
10101	000001	Pac 4(cab.) Pac 4(carga útil)
01101	000111	Config.

O conteúdo dos registradores que armazenam as marcas produzidas no lugar L1 é zerado, pois não há nenhuma marca em L1 na marcação inicial da Rede de Petri. Os

pesos dos arcos de entrada são armazenados nos registradores Arco 1 e Arco 2. O Arco 1 indica a necessidade de uma marca do tipo *e* e o Arco 2 indica a necessidade de uma marca do tipo *f*. A instrução "01110", na tabela 9.1, armazena, no registrador de probabilidade, o valor "111110", o que indica uma probabilidade de disparo de aproximadamente 98%. As instruções "01000"e "01001"são utilizadas para armazenar, respectivamente, o endereço de destino do bloco BCGN ("101101") e o endereço de destino do bloco BCERP em relação ao bloco BCGN ("001001"). O bloco BCERP(1,1) precisa se comunicar com três blocos BCERPs, quais sejam: BCERP(1,2), BCERP(1,3) e BCERP(2,1). Além de se comunicar com esses blocos, o BCERP(1,1) também se comunica com o bloco BCGN, para a busca de números pseudo-aleatórios, e com a interface de entrada. Para que essas comunicações ocorram, o registrador de comunicação (MR) deve receber o valor 4 por meio de um pacote de configuração que contenha a instrução "10000". Nesta mesma instrução, os dois últimos bits do Dado de Configuração refere-se ao tempo de disparo da transição. Como foi definida uma unidade de tempo para a transição T1, então o registrador de tempo recebe o valor 1. O banco de memória deve conter quatro pacotes. No cabeçalho de Pac 1, deve-se armazenar o valor "101000", o que indica o bloco BCERP(1,2) como endereço de destino. Na carga útil de Pac 1 deve-se colocar o valor "00100000010". Este pacote, quando enviado, produz duas marcas do tipo *e* no bloco BCERP(1,2). No cabeçalho de Pac 2, deve-se armazenar o valor "110000", o que indica o bloco BCERP(1,3) como endereço de destino. Na carga útil de Pac 2 deve-se colocar o valor "00101000010". Este pacote, quando enviado, produz duas marcas do tipo *f* no bloco BCERP(1,3). No cabeçalho de Pac 3, deve-se armazenar o valor "000101", o que indica o bloco BCERP(2,1) como endereço de destino. Na carga útil de Pac 3 deve-se colocar o valor "00110000011". Este pacote produz três marcas do tipo *g* no bloco BCERP(2,1). No cabeçalho de Pac 4, deve-se armazenar o valor "000001", o que indica a interface de entrada como endereço de destino. Na carga útil de Pac 4 deve-se colocar o valor "00100000000". Este pacote, quando enviado, realiza o sincronismo de tempo. Por fim, deve-se enviar a instrução "01101"de finalização de configuração, obrigando o bloco BCERP configurado a enviar um pacote para o endereço de destino "000111".

9.3.2 Configuração do Bloco BCERP(1,2)

No bloco BCERP(1,2) foram mapeados a transição T2 e o lugar L2. O lugar L2 pode conter marcas do tipo *e*. A transição T2 estará habilitada para o disparo se o lugar L2 possuir duas marcas do tipo *e*. O disparo da transição produz uma marca do tipo *f* no lugar L5 e uma marca do tipo *g* no lugar L6. Na tabela 9.2 apresentam-se os pacotes de

configuração que foram enviados ao bloco BCERP(1,2).

Tabela 9.2: Pacotes de configuração do BCERP(1,2)

Instrução	**Dado de Configuração**	**Definição**
00000	000001	Marca 1
00001	000000	Marca 2
00010	000000	Marca 3
00011	000000	Marca 4
00100	000010	Arco 1
00101	000000	Arco 2
00110	000000	Arco 3
00111	000000	Arco 4
01110	111111	Sem probab.
01111	000001	Num_arm
10000	001110	MR Tempo
11010	000100	Cont_MR
10001	001010	Sem conflito Pont_Pac_Final Pont_Pac_Inic
10010	101101	Pac 1(cab.) Pac 1(carga útil)
10011	001110	Pac 2(cab.) Pac 2(carga útil)
10100	001000	Pac 3(cab.) Pac 3(carga útil)
01101	001111	Config.

O registrador Marca 1 armazena as marcas produzidas no lugar L2. Neste registrador deve-se armazenar o valor 1, pois há uma marca do tipo *e* em L2 na marcação inicial da Rede de Petri. O peso do arco de entrada é armazenado no registrador Arco 1. O Arco 1 indica a necessidade de duas marcas do tipo *e*. A instrução "01110", na tabela 9.2, armazena, no registrador de probabilidade, o valor "111111", o que indica uma transição sem probabilidade de disparo, ou seja, quando habilitada, a transição poderá disparar. O bloco BCERP(1,2) precisa se comunicar com três blocos BCERPs, quais sejam: BCERP(2,3), BCERP(3,1) e BCERP(1,1). Para que essas comunicações ocorram, o registrador de comunicação (MR) deve receber o valor 3 por meio de um pacote de configuração que contenha a instrução "10000". Nesta mesma instrução, os dois últimos bits do Dado de Configuração refere-se ao tempo de disparo da transição. Como foi definida duas unidades de tempo para a transição T2, então o registrador de tempo recebe o valor 2. O banco de memória deve conter três pacotes. No cabeçalho de Pac 1, deve-se armazenar o valor "101101", o que indica o bloco BCERP(2,3) como endereço de destino. Na carga útil de Pac 1 deve-se colocar o valor "00111000001". Este pacote, quando enviado, produz uma marca do tipo *f* no bloco BCERP(2,3). No cabeçalho de Pac 2, deve-se armazenar o valor "001110", o que indica o bloco BCERP(3,1) como endereço de destino. Na carga útil de

Pac 2 deve-se colocar o valor "00110000001". Este pacote produz uma marca do tipo *g* no bloco BCERP(3,1). No cabeçalho de Pac 3, deve-se armazenar o valor "001000", o que indica o bloco BCERP(1,1) como endereço de destino. Na carga útil de Pac 3 deve-se colocar o valor "00100000000". Este pacote, quando enviado, realiza o sincronismo de tempo. Por fim, deve-se enviar a instrução "01101"de finalização de configuração, obrigando o bloco BCERP configurado a enviar um pacote para o endereço de destino "001111".

9.3.3 Configuração do Bloco BCERP(1,3)

No bloco BCERP(1,3) foram mapeados a transição T3 e o lugar L3. O lugar L3 pode conter marcas do tipo *f*. A transição T3 estará habilitada para o disparo se o lugar L3 possuir duas marcas do tipo *f*. O disparo da transição produz duas marcas do tipo *e*, no lugar L6, e duas marcas do tipo *g*, no lugar L7. Na tabela 9.3 apresentam-se os pacotes de configuração que foram enviados ao bloco BCERP(1,3).

Tabela 9.3: Pacotes de configuração do BCERP(1,3)

Instrução	**Dado de Configuração**	**Definição**
00000	000000	Marca 1
00001	000000	Marca 2
00010	000000	Marca 3
00011	000000	Marca 4
00100	000000	Arco 1
00101	000010	Arco 2
00110	000000	Arco 3
00111	000000	Arco 4
01110	111111	Sem probab.
01111	000001	Num_arm
10000	001111	MR Tempo
11010	000100	Cont_MR
10001	001010	Sem conflito Pont_Pac_Final Pont_Pac_Inic
10010	010110	Pac 1(cab.) Pac 1(carga útil)
10011	000101	Pac 2(cab.) Pac 2(carga útil)
10100	010000	Pac 3(cab.) Pac 3(carga útil)
01101	010111	Config.

O registrador Marca 2 armazena as marcas produzidas no lugar L3. O conteúdo deste registrador é zerado, pois não há marcas do tipo *f* em L3 na marcação inicial da Rede de Petri. O peso do arco de entrada é armazenado no registrador Arco 2. O Arco 2 indica a necessidade de duas marcas do tipo *f*. A instrução "01110", na tabela 9.3, armazena,

no registrador de probabilidade, o valor "111111", o que indica uma transição sem probabilidade de disparo, ou seja, quando habilitada, a transição poderá disparar. O bloco BCERP(1,3) precisa se comunicar com três blocos BCERPs, quais sejam: BCERP(2,3), BCERP(3,1) e BCERP(1,1). Para que essas comunicações ocorram, o registrador de comunicação (MR) deve receber o valor 3 por meio de um pacote de configuração que contenha a instrução "10000". Nesta mesma instrução, os dois últimos bits do Dado de Configuração refere-se ao tempo de disparo da transição. Como foi definida três unidades de tempo para a transição T3, então o registrador de tempo recebe o valor 3. O banco de memória deve conter três pacotes. No cabeçalho de Pac 1, deve-se armazenar o valor "010110", o que indica o bloco BCERP(3,1) como endereço de destino. Na carga útil de Pac 1 deve-se colocar o valor "00100000010". Este pacote, quando enviado, produz duas marcas do tipo *e* no bloco BCERP(3,1). No cabeçalho de Pac 2, deve-se armazenar o valor "000101", o que indica o bloco BCERP(2,3) como endereço de destino. Na carga útil de Pac 2 deve-se colocar o valor "00110000010". Este pacote produz duas do tipo *g* no bloco BCERP(2,3). No cabeçalho de Pac 3, deve-se armazenar o valor "010000", o que indica o bloco BCERP(1,1) como endereço de destino. Na carga útil de Pac 3 deve-se colocar o valor "00100000000". Este pacote, quando enviado, realiza o sincronismo de tempo. Por fim, deve-se enviar a instrução "01101"de finalização de configuração, obrigando o bloco BCERP configurado a enviar um pacote para o endereço de destino "010111".

9.3.4 Configuração do Bloco BCERP(2,1)

No bloco BCERP(2,1) foram mapeados a transição T4 e o lugar L4. O lugar L4 pode conter marcas do tipo *g*. A transição T4 estará habilitada para o disparo se o lugar L4 possuir três marcas do tipo *g*. O disparo da transição produz uma marca do tipo *f* no lugar L7. Na tabela 9.4 apresentam-se os pacotes de configuração que foram enviados ao bloco BCERP(2,1).

O registrador Marca 3 armazena as marcas produzidas no lugar L4. O conteúdo deste registrador é zerado, pois não há marcas do tipo *g* em L4 na marcação inicial da Rede de Petri. O peso do arco de entrada é armazenado no registrador Arco 3. O Arco 3 indica a necessidade de três marcas do tipo *g*. A instrução "01110", na tabela 9.4, armazena, no registrador de probabilidade, o valor "111111", o que indica uma transição sem probabilidade de disparo, ou seja, quando habilitada, a transição poderá disparar. O bloco BCERP(2,1) precisa se comunicar com dois blocos BCERPs, quais sejam: BCERP(2,3) e BCERP(1,1). Para que essas comunicações ocorram, o registrador de comunicação (MR)

Tabela 9.4: Pacotes de configuração do BCERP(2,1)

Instrução	**Dado de Configuração**	**Definição**
00000	000000	Marca 1
00001	000000	Marca 2
00010	000000	Marca 3
00011	000000	Marca 4
00100	000000	Arco 1
00101	000000	Arco 2
00110	000011	Arco 3
00111	000000	Arco 4
01110	111111	Sem probab.
01111	000001	Num_arm
10000	001001	MR Tempo
11010	000100	Cont_MR
10001	000101	Sem conflito Pont_Pac_Final Pont_Pac_Inic
10010	110000	Pac 1(cab.) Pac 1(carga útil)
10011	000001	Pac 2(cab.) Pac 2(carga útil)
01101	000110	Config.

deve receber o valor 2 por meio de um pacote de configuração que contenha a instrução "10000". Nesta mesma instrução, os dois últimos bits do Dado de Configuração refere-se ao tempo de disparo da transição. Como foi definida uma unidade de tempo para a transição T4, então o registrador de tempo recebe o valor 1. O banco de memória deve conter dois pacotes. No cabeçalho de Pac 1, deve-se armazenar o valor "110000", o que indica o bloco BCERP(2,3) como endereço de destino. Na carga útil de Pac 1 deve-se colocar o valor "00101000001". Este pacote produz uma marca do tipo *f* no bloco BCERP(2,3). No cabeçalho de Pac 2, deve-se armazenar o valor "000001", o que indica o bloco BCERP(1,1) como endereço de destino. Na carga útil de Pac 2 deve-se colocar o valor "00100000000". Este pacote, quando enviado, realiza o sincronismo de tempo. Por fim, deve-se enviar a instrução "01101"de finalização de configuração, obrigando o bloco BCERP configurado a enviar um pacote para o endereço de destino "000110".

9.3.5 Configuração do Bloco BCERP(2,3)

No bloco BCERP(2,3) foram mapeados a transição T5 e os lugares L5 e L7. O lugar L5 pode conter marcas do tipo *f* e, o lugar L7 pode conter marcas dos tipos *f* e *g*. A transição T5 estará habilitada para o disparo se o lugar L5 possuir uma marca do tipo *f* e, o lugar L7 possuir uma marca do tipo *f* e duas marcas do tipo *g*. O disparo da transição

produz uma marca do tipo *f* no lugar L8. Na tabela 9.5 apresentam-se os pacotes de configuração que foram enviados ao bloco BCERP(2,3).

Tabela 9.5: Pacotes de configuração do BCERP(2,3)

Instrução	**Dado de Configuração**	**Definição**
00000	000000	Marca 1
00001	000000	Marca 2
00010	000000	Marca 3
00011	000000	Marca 4
00100	000000	Arco 1
00101	000001	Arco 2
00110	000010	Arco 3
00111	000001	Arco 4
01110	111111	Sem probab.
01111	000001	Num_arm
10000	010010	MR Tempo
11010	001100	Cont_MR
10001	001111	Sem conflito Pont_Pac_Final Pont_Pac_Inic
10010	010101	Pac 1(cab.) Pac 1(carga útil)
10011	001001	Pac 2(cab.) Pac 2(carga útil)
10100	000001	Pac 3(cab.) Pac 3(carga útil)
10101	010000	Pac 4(cab.) Pac 4(carga útil)
01101	010110	Config.

O registrador Marca 4 é utilizado para armazenar as marcas do tipo *f* produzidas no lugar L5. O registrador Marca 2 é utilizado para armazenar as marcas do tipo *f* produzidas no lugar L7 e, o registrador Marca 3 é utilizado para armazenar as marcas do tipo *g* produzidas no lugar L7. O conteúdo dos três registradores é zerado, pois não há marcas armazenadas em L5 e L7 na marcação inicial da Rede de Petri. Os pesos dos arcos de entrada são armazenados nos registradores Arco 2, Arco 3 e Arco 4. O Arco 2 indica a necessidade de uma marca do tipo *f* no lugar L7. O Arco 3 indica a necessidade de duas marcas do tipo *g* no lugar L7 e, o Arco 4 indica a necessidade de uma marca do tipo *f* no lugar L5. A instrução "01110", na tabela 9.5, armazena, no registrador de probabilidade, o valor "111111", o que indica uma transição sem probabilidade de disparo, ou seja, quando habilitada, a transição poderá disparar. O bloco BCERP(2,3) precisa se comunicar com quatro blocos BCERPs, quais sejam: BCERP(3,1), BCERP(1,2), BCERP(1,3) e BCERP(2,1). Para que essas comunicações ocorram, o registrador de comunicação (MR) deve receber o valor 4 por meio de um pacote de configuração que contenha a instrução "10000". Nesta mesma instrução, os dois últimos bits do Dado de Configuração refere-se

ao tempo de disparo da transição. Como foi definida duas unidades de tempo para a transição T5, então o registrador de tempo recebe o valor 2. O banco de memória deve conter quatro pacotes. No cabeçalho de Pac 1, deve-se armazenar o valor "010101", o que indica o bloco BCERP(3,1) como endereço de destino. Na carga útil de Pac 1 deve-se colocar o valor "00101000001". Este pacote, quando enviado, produz uma marca do tipo *f* no bloco BCERP(3,1). No cabeçalho de Pac 2, deve-se armazenar o valor "001001", o que indica o bloco BCERP(1,2) como endereço de destino. Na carga útil de Pac 2 deve-se colocar o valor "00100000000". Este pacote, quando enviado, realiza o sincronismo de tempo. No cabeçalho de Pac 3, deve-se armazenar o valor "000001", o que indica o bloco BCERP(1,3) como endereço de destino. Na carga útil de Pac 3 deve-se colocar o valor "00100000000". Este pacote realiza o sincronismo de tempo. No cabeçalho de Pac 4, deve-se armazenar o valor "010000", o que indica o bloco BCERP(2,1) como endereço de destino. Na carga útil de Pac 4 deve-se colocar o valor "00100000000". Este pacote, quando enviado, realiza o sincronismo de tempo. Por fim, deve-se enviar a instrução "01101"de finalização de configuração, obrigando o bloco BCERP configurado a enviar um pacote para o endereço de destino "010110".

9.3.6 Configuração do Bloco BCERP(3,1)

No bloco BCERP(3,1) foram mapeados a transição T6 e os lugares L6 e L8. O lugar L8 pode conter marcas do tipo *f* e, o lugar L6 pode conter marcas dos tipos *e* e *g*. A transição T6 estará habilitada para o disparo se o lugar L8 possuir uma marca do tipo *f* e, o lugar L6 possuir duas marcas do tipo *e* e uma marca do tipo *g*. O disparo da transição produz três marcas do tipo *g* no lugar L9. Na tabela 9.6 apresentam-se os pacotes de configuração que foram enviados ao bloco BCERP(3,1).

O registrador Marca 1 é utilizado para armazenar as marcas do tipo *e* produzidas no lugar L6. O registrador Marca 2 é utilizado para armazenar as marcas do tipo *f* produzidas no lugar L8 e, o registrador Marca 3 é utilizado para armazenar as marcas do tipo *g* produzidas no lugar L6. No registrador Marca 1 é armazenado o valor 1, no registrador Marca 2 o valor 0 e, no registrador Marca 3 o valor 1, assim, a marcação inicial da Rede de Petri é estabelecida para os lugares L6 e L8. Os pesos dos arcos de entrada são armazenados nos registradores Arco 1, Arco 2 e Arco 3. O Arco 1 indica a necessidade de duas marcas do tipo *e* no lugar L6. O Arco 2 indica a necessidade de uma marca do tipo *f* no lugar L8 e, o Arco 3 indica a necessidade de uma marca do tipo *g* no lugar L6. A instrução "01110", na tabela 9.6, armazena, no registrador

Tabela 9.6: Pacotes de configuração do BCERP(3,1)

Instrução	**Dado de Configuração**	**Definição**
00000	000001	Marca 1
00001	000000	Marca 2
00010	000001	Marca 3
00011	000000	Marca 4
00100	000010	Arco 1
00101	000001	Arco 2
00110	000001	Arco 3
00111	000000	Arco 4
01110	111111	Sem probab.
01111	000001	Num_arm
10000	001111	MR Tempo
11010	001100	Cont_MR
10001	001111	Sem conflito Pont_Pac_Final Pont_Pac_Inic
10010	110011	Pac 1(cab.) Pac 1(carga útil)
10011	101010	Pac 2(cab.) Pac 2(carga útil)
10100	110010	Pac 3(cab.) Pac 3(carga útil)
10101	110001	Pac 4(cab.) Pac 4(carga útil)
01101	000101	Config.

de probabilidade, o valor "111111", o que indica uma transição sem probabilidade de disparo, ou seja, quando habilitada, a transição poderá disparar. O bloco BCERP(3,1) precisa se comunicar com três blocos BCERPs, quais sejam: BCERP(1,2), BCERP(1,3) e BCERP(2,3). Além disso, o bloco BCERP(3,1) também se comunica com a interface de saída. Para que essas comunicações ocorram, o registrador de comunicação (MR) deve receber o valor 3 por meio de um pacote de configuração que contenha a instrução "10000"(não há necessidade de sincronismo de tempo com a interface de saída). Nesta mesma instrução, os dois últimos bits do Dado de Configuração refere-se ao tempo de disparo da transição. Como foi definida três unidades de tempo para a transição T6, então o registrador de tempo recebe o valor 3. O banco de memória deve conter quatro pacotes. No cabeçalho de Pac 1, deve-se armazenar o valor "110011", o que indica a interface de saída como endereço de destino. Na carga útil de Pac 1 deve-se colocar o valor "00110000011". Este pacote indica, para a interface de saída, a produção de três marcas do tipo *g*. No cabeçalho de Pac 2, deve-se armazenar o valor "101010", o que indica o bloco BCERP(1,2) como endereço de destino. Na carga útil de Pac 2 deve-se colocar o valor "00100000000". Este pacote, quando enviado, realiza o sincronismo de tempo. No cabeçalho de Pac 3, deve-se armazenar o valor "110010", o que indica o bloco BCERP(1,3)

como endereço de destino. Na carga útil de Pac 3 deve-se colocar o valor "00100000000". Este pacote realiza o sincronismo de tempo. No cabeçalho de Pac 4, deve-se armazenar o valor "110001", o que indica o bloco BCERP(2,3) como endereço de destino. Na carga útil de Pac 4 deve-se colocar o valor "00100000000". Este pacote realiza o sincronismo de tempo. Por fim, deve-se enviar a instrução "01101"de finalização de configuração, obrigando o bloco BCERP configurado a enviar um pacote para o endereço de destino "000101".

9.3.7 Configuração do Bloco BCGN

O bloco BCGN foi configurado como sendo um gerador linear congruente multiplicativo, como explicado na seção 4.2. Na figura 9.7 apresentam-se os pacotes de configuração que foram enviados ao bloco BCGN para a sua programação. Os pacotes de configuração definiram o seguinte gerador: $x_i = (4 * x_{i-1}) \mod 2^6$, com a semente ($x_{i-1}$ inicial) igual a 6.

Tabela 9.7: Pacotes de configuração do bloco BCGN

Dado de Configuração	**Definição**
00000000000	a_2
00000000011	x_{i-2}
00000000100	a_1
00000000110	x_{i-1}
00000000000	c_{i-1} (menos significativos)
00001000000	controle transporte c_{i-1} (mais significativos)

9.4 Simulação e Teste da Arquitetura 3x3

Com o auxílio do *software* Quartus II Web Edition versão 5.0, a arquitetura proposta foi mapeada no FPGA EPF10K70RC240. De 3744 elementos lógicos disponíveis, foram utilizados 3224 (86%). Devido a pequena quantidade de lógica do FPGA disponível para a realização dos testes na arquitetura proposta, os blocos de configuração BCERP e BCGN tiveram que ser adaptados. Desta forma, para que a arquitetura 3x3 pudesse ser inteiramente mapeada no FPGA, os pacotes de dados e de configuração foram reduzidos de 32 bits para 17 bits, distribuídos da seguinte forma:

- 1 bit para o sentido leste-oeste ou oeste-leste;
- 2 bits para a variação no eixo x;

- 1 bit para o sentido norte-sul ou sul-norte;
- 2 bits para a variação no eixo y;
- 3 bits para a instrução de processamento;
- 2 bits para a identificação do lugar;
- 6 bits para a quantidade de marcas.

No processo de simulação, foi realizado o procedimento definido a seguir. Deve-se inicializar a arquitetura com um nível lógico alto no sinal inicializar dos blocos BCERPs, BCGNs e roteadores. Depois, deve-se enviar os pacotes de configuração para o bloco BCGN e para todos os blocos BCERPs. Após o envio de todos os pacotes de configuração, deve-se esperar o retorno dos pacotes de finalização de configuração de todos os blocos BCERPs. Ao receber esses pacotes, deve-se enviar um nível lógico alto no sinal `finalizar_configuracao`, indicando o final da etapa de configuraçao. Terminada a etapa de configuração, deve-se enviar pacotes de dados de sincronismo de tempo ou pacotes de dados para a adição de marcas no lugar de entrada L1 implementado no bloco BCERP(1,1) e, observar o comportamento da arquitetura com relação ao disparo das transições mapeadas, ou seja, deve-se esperar pacotes de dados contendo informações referentes ao disparo das transições, a busca de números pseudo-aleatórios, ao sincronismo de conflito, entre outras possibilidades.

No processo de teste da arquitetura proposta, utilizou-se a placa educacional UP2 que possui o FPGA EPF10K70RC240. No processo de síntese realizado na arquitetura, o código VHDL descrito foi mapeado nesta placa UP2. Devido a quantidade de pinos de entrada da arquitetura (43 pinos), foram confeccionados módulos em VHDL que implementam duas memórias ROMs, como apresentado na figura 9.5.

As memórias são utilizadas para o envio de pacotes de configuração e de dados nas etapas de programação e de processamento da arquitetura. Os módulos foram devidamente implementados no FPGA e interligados ao sistema de roteamento. Os pacotes de configuração mostrados nas tabelas de 9.1 até 9.7, foram armazenados na memória ROM e enviados, posteriormente, para a programação dos blocos BCERPs e do bloco BCGN.

Na etapa de execução da rede implementada na arquitetura 3x3, foram enviados pacotes de dados de sincronismo de tempo e pacotes de dados para a adição de marcas no lugar L1 da Rede de Petri.

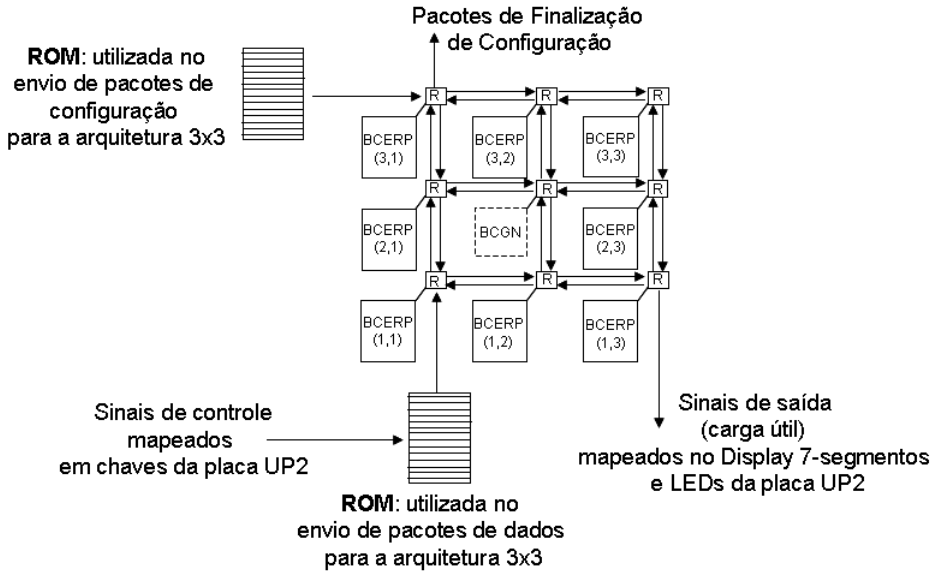


Figura 9.5: Sistema utilizado para o teste da arquitetura 3x3 no FPGA

No exemplo da tabela 9.8 encontram-se alguns pacotes de dados que foram armazenados na memória ROM e enviados para a arquitetura proposta na realização do teste da Rede de Petri apresentada na figura 9.3.

Tabela 9.8: Pacotes de dados armazenados na memória ROM e utilizados no processo de execução da Rede de Petri

Inst. Proc.	**Ident. Lugar**	**Dado**	**Ação**
000	00	000001	Produção de uma marca do tipo e
001	01	000001	Produção de uma marca do tipo f (soma finalizadora)
001	00	000000	Sincronismo de tempo

No processo de mapeamento da Rede de Petri da figura 9.3, a arquitetura levou $3,42\mu s$ para enviar os 133 pacotes ou 4256 bits de configuração armazenados na memória ROM.

A Rede de Petri implementada na arquitetura levou em torno de $5,35\mu s$ para a produção de três marcas do tipo g no lugar L9, após o envio de uma marca do tipo e e uma marca do tipo f para o lugar L1. Na tabela 9.9 apresenta-se a marcação da Rede de Petri em cada tempo lógico até a produção de marcas no lugar de saída L9. Dois pacotes de dados foram enviados logo após a configuração da arquitetura. Estes pacotes adicionaram duas marcas no lugar L1, uma do tipo e e outra do tipo f, como mostrado na tabela 9.9 no tempo lógico 0. Após $3,48\mu s$, contando a partir do início da configuração, a arquitetura armazenou estas duas marcas. Na tabela 9.9 encontra-se toda a dinâmica da rede (marcação da rede em cada tempo lógico) e o seu correspondente tempo físico no qual a arquitetura atingiu a marcação esperada.

Tabela 9.9: Marcação da Rede de Petri

Tempo Lógico	L1	L2	L3	L4	L5	L6	L7	L8	L9	Tempo Arq.	Descrição
0	$1e$ $1f$	$1e$	$0f$	$0g$	$0f$	$1e$ $1g$	$0f$ $0g$	$0f$	$0g$	$3,48\mu s$	Armazenamento das marcas enviadas no lugar L1
1	$0e$ $0f$	$1e$	$0f$	$0g$	$0f$	$1e$ $1g$	$0f$ $0g$	$0f$	$0g$	$3,53\mu s$	Transição T1 habilitada T1 = 1un.
2	$0e$ $0f$	$3e$	$2f$	$3g$	$0f$	$1e$ $1g$	$0f$ $0g$	$0f$	$0g$	$4,2\mu s$	Disparo da transição T1
3	$0e$ $0f$	$1e$	$0f$	$0g$	$0f$	$1e$ $1g$	$0f$ $0g$	$0f$	$0g$	$4,47\mu s$	Transições T2, T3 e T4 habilitadas T2 = 2un. / T3 = 3un. / T4 = 1un.
4	$0e$ $0f$	$1e$	$0f$	$0g$	$0f$	$1e$ $1g$	$1f$ $0g$	$0f$	$0g$	$5,00\mu s$	Disparo da transição T4 T2 = 1un. / T3 = 2un.
5	$0e$ $0f$	$1e$	$0f$	$0g$	$1f$	$1e$ $2g$	$1f$ $0g$	$0f$	$0g$	$6,07\mu s$	Disparo da transição T2 T3 = 1un.
6	$0e$ $0f$	$1e$	$0f$	$0g$	$1f$	$3e$ $2g$	$1f$ $2g$	$0f$	$0g$	$6,78\mu s$	Disparo da transição T3
7	$0e$ $0f$	$1e$	$0f$	$0g$	$0f$	$3e$ $2g$	$0f$ $0g$	$0f$	$0g$	$6,82\mu s$	Transição T5 habilitada T5 = 2un.
8	$0e$ $0f$	$1e$	$0f$	$0g$	$0f$	$3e$ $2g$	$0f$ $0g$	$0f$	$0g$	–	T5 = 1un.
9	$0e$ $0f$	$1e$	$0f$	$0g$	$0f$	$3e$ $2g$	$0f$ $0g$	$1f$	$0g$	$8,46\mu s$	Disparo da transição T5
10	$0e$ $0f$	$1e$	$0f$	$0g$	$0f$	$1e$ $1g$	$0f$ $0g$	$0f$	$0g$	$8,5\mu s$	Transição T6 habilitada T6 = 3un.
11	$0e$ $0f$	$1e$	$0f$	$0g$	$0f$	$1e$ $1g$	$0f$ $0g$	$0f$	$0g$	–	T6 = 2un.
12	$0e$ $0f$	$1e$	$0f$	$0g$	$0f$	$1e$ $1g$	$0f$ $0g$	$0f$	$0g$	–	T6 = 1un.
13	$0e$ $0f$	$1e$	$0f$	$0g$	$0f$	$1e$ $1g$	$0f$ $0g$	$0f$	$3g$	$8,77\mu s$	Disparo da transição T6

Vários testes foram realizados na arquitetura 3x3 modificando-se os valores armazenados nas posições da memória ROM para o mapeamento de diferentes Redes de Petri, as quais possuiam transições com/sem probabilidade de disparo e com/sem conflito estrutural. Diferentes pacotes de dados também foram armazenados para a verificação do comportamento da arquitetura ao receber pacotes de sincronismo de tempo, pacotes de adição de marcas de tipos diferentes, pacotes contendo somas finalizadoras, entre outras possibilidades. Alguns destes testes estão disponíveis no CD-ROM anexado a esta tese.

9.5 Comentários

Foram comentados, neste capítulo, os processos utilizados para a simulação e teste da arquitetura proposta. Uma arquitetura 3x3 composta de 9 roteadores interligando oito blocos BCERPs e um bloco BCGN foi mapeada num FPGA. Alguns modelos de Redes de Petri foram descritos e mapeados na arquitetura. Pacotes de configuração foram enviados, por meio de uma memória ROM, para todos os blocos BCERPs e para o bloco BCGN da arquitetura 3x3 e, pacotes de dados foram enviados, por meio de outra memória ROM, para a verificação do comportamento da arquitetura diante da adição de diferentes marcas em diferentes lugares das Redes de Petri implementadas.

No próximo capítulo encontram-se as considerações finais sobre a arquitetura proposta e algumas sugestões para trabalhos futuros.

10 Considerações Finais

10.1 Tópicos Desenvolvidos

A proposta apresentada e o desenvolvimento dos projetos do roteador e dos blocos BCGN e BCERP possuem algumas características inovadoras tanto em relação à idéia da proposta quanto à própria arquitetura adotada. Os pontos mais relevantes são:

- O desenvolvimento de uma arquitetura reconfigurável e multiprocessada que permite o mapeamento tecnológico de sistemas descritos em Redes de Petri diretamente no nível comportamental, sem a necessidade de uma descrição em níveis menos abstratos;
- A possibilidade de implementação física de sistemas descritos em Redes de Petri T-temporizadas que possuem diferenciação entre as marcas e probabilidade de disparo entre as transições;
- A utilização de um sistema de comunicação composto por roteadores ao invés de um barramento composto por chaves reconfiguráveis;
- A possibilidade de extensão da arquitetura proposta para uma estrutura 3-D, utilizando dois tipos diferentes de roteadores, com cinco ou seis canais de comunicação;
- O desenvolvimento em *hardware* de um algoritmo distribuído para a resolução de conflito comportamental entre as transições de uma Rede de Petri; e
- A possibilidade de se enviar vários pacotes de configuração ao mesmo tempo, distribuindo paralelamente e agilizando o processo de configuração dos blocos BCGNs e BCERPs.

10.2 Sugestões para Trabalhos Futuros

Na etapa de programação da arquitetura proposta é necessário o envio de pacotes de configuração para os blocos BCGNs e BCERPs. Para automatizar o processo de configuração da arquitetura, seria útil o desenvolvimento de um programa capaz de gerar esses pacotes de configuração a partir da descrição em Rede de Petri do sistema a ser implementado.

As transições de entrada e saída de uma Rede de Petri devem ser alocadas em blocos BCERPs próximos uns dos outros. Se as transições de entrada e saída forem alocadas em blocos BCERPs muito distantes uns dos outros, os pacotes de dados, no processo de execução da arquitetura, terão que atravessar uma grande quantidade de roteadores até chegarem aos seus destinos. Por isso, transições que se comunicam diretamente devem ser mapeadas em blocos BCERPs próximos uns dos outros para que se reduza o tempo de comunicação entre eles. O desenvolvimento de um algoritmo de otimização para mapear adequadamente as transições nos blocos BCERPs poderia reduzir o custo de comunicação entre os blocos de configuração da arquitetura proposta.

A arquitetura proposta pode ser adaptada para implementar Redes de Petri temporais (WANG, 1998) e redes estocásticas (MARSAN, 1990) (WANG, 1998), o que aumentaria o poder de modelagem dos sistemas. Nas redes temporais, pode-se utilizar o próprio bloco BCGN projetado para a geração de números pseudo-aleatórios que serão utilizados pelos blocos BCERPs para definir, dentro do intervalo de disparo, o momento em que a transição será disparada. Em redes estocásticas, deve-se projetar um bloco de configuração capaz de gerar números pseudo-aleatórios com uma distribuição não uniforme, como, por exemplo, a exponencial ou a gaussiana. Números pseudo-aleatórios com distribuição não uniforme podem ser gerados a partir de números pseudo-aleatórios com distribuição uniforme. Para isso, aplica-se uma determinada fórmula, definida de acordo com a distribuição pretendida, no número uniformemente gerado. Portanto, a inclusão, no próprio bloco BCGN projetado, de uma lógica capaz de implementar essa fórmula viabilizaria o tratamento de Redes de Petri estocásticas.

Utilizando o programa de geração de pacotes de configuração e o algoritmo de mapeamento dos elementos da Rede de Petri nos blocos BCERPs pode-se desenvolver uma plataforma integrada de trabalho que permitirá ao projetista descrever um sistema graficamente por meio de uma Rede de Petri, analisar e verificar erros da rede modelada e implementá-la fisicamente na arquitetura proposta.

Uma possibilidade interessante para a continuidade deste projeto é a implementação da arquitetura proposta em um circuito integrado e as respectivas análises da arquitetura.

10.3 Conclusão

O primeiro bloco desenvolvido foi o roteador do sistema de comunicação. Posteriormente, foi elaborado o bloco BCGN para a geração de números pseudo-aleatórios. Por último, foi desenvolvido o bloco BCERP, responsável pela implementação de lugares e transição da Rede de Petri. Foi feita uma escolha minuciosa dos circuitos digitais adequados para cada bloco da arquitetura proposta e, posteriormente, foi realizada uma descrição detalhada em VHDL dos blocos projetados, uma vez que a arquitetura previa um processamento da Rede de Petri estritamente por *hardware*.

Devido à complexidade da arquitetura proposta, foi adotada a estratégia de se buscar a resposta de cada bloco individualmente prevendo posteriores interligações para a formação de blocos mais complexos. Os resultados obtidos nos processos de simulação e de teste dos blocos BCERP e BCGN e do roteador, bem como do sistema completo, foram muito satisfatórios e mostraram a viabilidade da arquitetura proposta. Tendo como base a família de FPGAs STRATIX, o roteador projetado possui 464 elementos lógicos, o bloco BCGN possui 235 e o bloco BCERP utiliza 1100 elementos lógicos. Várias Redes de Petri foram utilizadas para a validação e verificação do tempo de execução da arquitetura. Em média, uma Rede de Petri com 8 transições e 32 lugares é processada em torno de $5,35\mu s$, levando em consideração uma Rede de Petri T-temporizada; com a mesma estrutura da rede, porém sem a realização de temporização, a rede é processada em $0,71\mu s$, conseguindo-se com isso, um ganho de velocidade de 7,53 vezes em relação à uma rede T-temporizada. No processo de configuração da Rede de Petri, a arquitetura levou, no pior caso, $3,42\mu s$ para enviar 133 pacotes ou 4256 bits de configuração.

A arquitetura foi mapeada em alguns FPGAs para a verificação da quantidade de lógica gasta para a sua implementação e da quantidade de blocos lógicos que podem ser incluídos em um FPGA. Em FPGAs EP1S40, da família STRATIX, é possível o mapeamento de uma arquitetura 5x4, com 19 blocos BCERPs, 20 roteadores e 1 bloco BCGN. Nesta arquitetura 5x4 é possível o mapeamento de até 19 transições e 76 lugares. Em FPGAs EP1S80, é possível o mapeamento de uma arquitetura 7x6, com 41 blocos BCERPs, 42 roteadores e 1 bloco BCGN. Nesta arquitetura 7x6 é possível o mapeamento de até 41 transições e 164 lugares.

Referências

ALTERA CORPORATION. *Introduction to QUARTUS II*. San Jose, 2004. 213 p.

ANZAI, F. et al. Hardware implementation of a multiprocessor system controlled by petri nets. In: INTERNATIONAL CONFERENCE ON INDUSTRIAL ELECTRONICS, CONTROL AND INSTRUMENTATION, 19., 1993, Hawaii. *Proceedings...* Piscataway: IEEE Computer Society Press, 1993. p. 121–126.

ARAUJO, J. *Dominando a linguagem C*. Rio de Janeiro: Ciência Moderna, 2004.

BAREL, M. Fast hardware random number generator for the tausworthe sequence. In: ANNUAL SYMPOSIUM ON SIMULATION, 16., 1983, Tampa. *Proceedings...* Los Alamitos: IEEE Computer Society Press, 1983. p. 121–135.

BARTIC, T. et al. Highly scalable network on chip for reconfigurable systems. In: INTERNATIONAL SYMPOSIUM ON SYSTEM-ON-CHIP, 5., 2003, Tampere. *Proceedings...* Washington: IEEE Computer Society Press, 2003. p. 79–82.

BAYS, C.; DURHAM, S. D. Improving a poor random number generator. *ACM Trans. Math. Softw.*, New York, v. 2, n. 1, p. 59–64, 1976.

BENINI, L.; MICHELI, G. D. Networks on chips: A new soc paradigm. *Computer*, Los Alamitos, v. 35, n. 1, p. 70–78, 2002.

BHATIA, D. Reconfigurable computing. In: INTERNATIONAL CONFERENCE ON VLSI DESIGN, 10., 1997, Hyderabad. Washington: IEEE Computer Society Press, 1997. p. 356–359.

BRENT, R. P. Uniform random number generators for supercomputers. In: AUSTRALIAN SUPERCOMPUTER CONFERENCE, 5., 1992, Canberra. *Proceedings...* Melbourne: ACM Press, 1992. p. 95–104.

BUNDELL, G. A. An fpga implementation of the petri net firing algorithm. In: AUSTRALASIAN CONFERENCE ON PARALLEL AND REAL-TIME SYSTEMS, 1997, Newcastle. Singapore: Springer-Verlag, 1997. p. 434–445.

BüYüKSAHIN, K. M.; NAJM, F. N. High-level power estimation with interconnect effects. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, 2000, Rapallo. *Proceedings...* New York: ACM Press, 2000. p. 197–202.

BüYüKSAHIN, K. M.; NAJM, F. N. High-level area estimation. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, 2002, Monterey. *Proceedings...* New York: ACM Press, 2002. p. 271–274.

CARTA, D. F. Two fast implementations of the minimal standard random number generator. *Commun. ACM*, New York, v. 33, n. 1, p. 87–88, 1990.

CHAIYAKUL, V.; WU, A. C.-H.; GAJSKI, D. D. Timing models for high-level synthesis. In: CONGRESS CENTRUM HAMBURG, 1992, Hamburg. *Proceedings...* Los Alamitos: IEEE Computer Society Press, 1992. p. 60–65.

CHANG, N.; KWON, W. H.; PARK, J. Fpga–based implementation of synchronous petri nets. In: INTERNATIONAL CONFERENCE ON INDUSTRIAL ELECTRONICS, CONTROL, AND INSTRUMENTATION, 22., 1996, Taipei. *Proceedings...* Piscataway: IEEE Computer Society Press, 1996. p. 934–939.

CHIU, T.-W. Shift-register sequence random number generators on the hypercube concurrent computers. In: CONFERENCE ON HYPERCUBE CONCURRENT COMPUTERS AND APPLICATIONS, 3., 1988, Pasadena. *Proceedings...* New York: ACM Press, 1988. p. 1421–1429.

CODDINGTON, P. *Random number generators for parallel computers*. Syracuse: [s.n.], 1997. Relatório técnico.

COLLINGS, B. J.; HEMBREE, G. B. Initializing generalized feedback shift register pseudorandom number generators. *J. ACM*, New York, v. 33, n. 4, p. 706–711, 1986.

COUTURE, R.; L'ECUYER, P. Linear recurrences with carry as uniform random number generators. In: CONFERENCE ON WINTER SIMULATION, 27., 1995, Arlington. *Proceedings...* New York: ACM Press, 1995. p. 263–267.

CSERTÁN, G. et al. Hardware accelerators for petri net analysis. In: WORKSHOP ON DISTRIBUTED AND PARALLEL SYSTEMS, 2., 1997, Budapest. *Proceedings...* Budapest: Institut für angewandte Informatik und Informationssysteme, 1997. p. 99–104.

CULLER, D. E.; GUPTA, A.; SINGH, J. P. *Parallel computer architecture*: a hardware/software approach. San Francisco: Morgan Kaufmann, 1997.

DALLY, W. J. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, Piscataway, v. 3, n. 2, p. 194–205, 1992.

DALLY, W. J.; TOWLES, B. Route packets, not wires: on-chip interconnection networks. In: CONFERENCE ON DESIGN AUTOMATION, 38., 2001, Las Vegas. *Proceedings...* New York: ACM Press, 2001. p. 684–689.

DAVID, R.; ALLA, H. *Petri nets and Grafcet*: tools for modelling discrete event systems. Upper Saddle River: Prentice-Hall, 1992.

DENG, L.-Y.; XU, H. A system of high-dimensional, efficient, long-cycle and portable uniform random number generators. *ACM Trans. Model. Comput. Simul.*, New York, v. 13, n. 4, p. 299–309, 2003.

ENTACHER, K. *A collection of selected pseudorandom number generators with linear structures*. [S.l.: s.n.], 1997. Relatório técnico.

ENTACHER, K. Bad subsequences of well-known linear congruential pseudorandom number generators. *ACM Trans. Model. Comput. Simul.*, New York, v. 8, n. 1, p. 61–70, 1998.

ERCEGOVAC, M.; LANG, T.; MORENO, J. H. *Introdução aos sistemas digitais*. Porto Alegre: Bookman, 2000.

GELOSH, D. S.; STELIFF, D. E. Modeling layout tools to derive forward estimates of area and delay at the rtl level. *ACM Trans. Des. Autom. Electron. Syst.*, New York, v. 5, n. 3, p. 451–491, 2000.

GOMES, L. *Redes de petri e sistemas digitais*: uma introdução. Lisboa: FCT, 1999.

GORESKY, M.; KLAPPER, A. Efficient multiply-with-carry random number generators with maximal period. *ACM Trans. Model. Comput. Simul.*, New York, v. 13, n. 4, p. 310–321, 2003.

GUERRIER, P.; GREINER, A. A generic architecture for on-chip packet-switched interconnections. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, 2000, Paris. *Proceedings...* New York: ACM Press, 2000. p. 250–256.

HANSELMAN, D.; LITTLEFIELD, B. *MATLAB 5*: versão do estudante: guia do usuário. São Paulo: Makron Books do Brasil, 1997.

HULL, T. E.; DOBELL, A. R. Mixed congruential random number generators for binary machines. *J. ACM*, New York, v. 11, n. 1, p. 31–40, 1964.

HUTCHINSON, D. W. A new uniform pseudorandom number generator. *Commun. ACM*, New York, v. 9, n. 6, p. 432–433, 1966.

JENSEN, K. *Coloured petri nets*: basic concepts, analysis methods and practical use. 2nd. ed. London: Springer-Verlag, 1997.

KAMAKURA, T. et al. Implementation of a large petri net by a group of petri net controller. In: INTERNATIONAL CONFERENCE ON INDUSTRIAL ELECTRONICS, CONTROL AND INSTRUMENTATION, 23., 1997, New Orleans. *Proceedings...* Piscataway: IEEE Computer Society Press, 1997. p. 1210–1215.

KARIM, F.; NGUYEN, A.; DEY, S. An interconnect architecture for networking systems on chips. *IEEE Micro*, Los Alamitos, v. 22, n. 5, p. 36–45, 2002.

KATZ, R. H. *Contemporary logic design*. Redwood City: Benjamin, 1994.

KERNIGHAN, B. W.; RITCHIE, D.; RITCHIE, D. M. *C programming language*. 2nd. ed. Upper Saddle River: Prentice Hall, 1988.

KUBÁTOVÁ, H. Petri net simulation using fpga. In: INTERNATIONAL AUTUMN COLLOQUIUM, 23., 2001, Ostrava. *Proceedings...* Ostrava: MARQ, 2001. p. 129–134.

KUBÁTOVÁ, H. Direct hardware implementation of petri net based model. In: NEW WAVES IN SYSTEM ARCHITECTURE, 29., 2003, Belek-Antalya. *Proceedings...* Canberra: IEEE Computer Society Press, 2003. p. 56–57.

KUBÁTOVÁ, H. Petri net models in hardware. *ECMS 2003*, Liberec, v. 1, n. 4, p. 158–162, 2003.

KUMAR, S. et al. A network on chip architecture and design methodology. In: ANNUAL SYMPOSIUM ON VLSI, 2002, Pittsburgh. *Proceedings...* Washington: IEEE Computer Society Press, 2002. p. 117.

L'ECUYER, P. Efficient and portable combined random number generators. *Commun. ACM*, New York, v. 31, n. 6, p. 742–751, 1988.

L'ECUYER, P. Uniform random number generators: a review. In: CONFERENCE ON WINTER SIMULATION, 29., 1997, Atlanta. *Proceedings...* New York: ACM Press, 1997. p. 127–134.

L'ECUYER, P. Uniform random number generators. In: CONFERENCE ON WINTER SIMULATION, 30., 1998, Washington. *Proceedings...* Los Alamitos: IEEE Computer Society Press, 1998. p. 97–104.

L'ECUYER, P.; BLOUIN, F.; COUTURE, R. A search for good multiple recursive random number generators. *ACM Trans. Model. Comput. Simul.*, New York, v. 3, n. 2, p. 87–98, 1993.

L'ECUYER, P.; PANNETON, F. A new class of linear feedback shift register generators. In: CONFERENCE ON WINTER SIMULATION, 32., 2000, Orlando. *Proceedings...* San Diego: Society for Computer Simulation International, 2000. p. 690–696.

L'ECUYER, P.; SIMARD, R. Beware of linear congruential generators with multipliers of the form $a = \pm 2^q \pm 2^r$. *ACM Trans. Math. Softw.*, New York, v. 25, n. 3, p. 367–374, 1999.

L'ECUYER, P.; TOUZIN, R. Fast combined multiple recursive generators with multipliers of the form $a = \pm 2^q \pm 2^r$. In: CONFERENCE ON WINTER SIMULATION, 32., 2000, Orlando. *Proceedings...* San Diego: Society for Computer Simulation International, 2000. p. 683–689.

LEWIS, T. G.; PAYNE, W. H. Generalized feedback shift register pseudorandom number algorithm. *J. ACM*, New York, v. 20, n. 3, p. 456–468, 1973.

LIU, J.; ZHENG, L.-R.; TENHUNEN, H. Interconnect intellectual property for network-on-chip (noc). *J. Syst. Archit.*, New York, v. 50, n. 2, p. 65–79, 2004.

MACLAREN, M. D.; MARSAGLIA, G. Uniform random number generators. *J. ACM*, New York, v. 12, n. 1, p. 83–89, 1965.

MANGIONE-SMITH, W. H. Application design for configurable computing. *Computer*, Los Alamitos, v. 30, n. 10, p. 115–117, 1997.

MANO, M. M. *Digital design*. 2nd. ed. Upper Saddle River: Prentice Hall, 1991.

MANO, M. M.; KIME, C. R. *Logic and computer design fundamentals*. 2nd. ed. Upper Saddle River: Prentice Hall, 1999. 1 CD–ROM.

MARSAGLIA, G. A current view of random number generators. In: SYMPOSIUM ON THE INTERFACE, 16., 2000, Lexington. *Proceedings...* North-Holland: Elsevier Science, 1985. p. 3–10.

MARSAN, M. A. Stochastic petri nets: an elementary introduction. In: EUROPEAN WORKSHOP ON APPLICATIONS AND THEORY IN PETRI NETS-SELECTED PAPERS, 9., 1990, London. *Advances...* London: Springer-Verlag, 1990. p. 1–29.

MATSUMOTO, M.; KURITA, Y. Twisted gfsr generators ii. *ACM Trans. Model. Comput. Simul.*, New York, v. 4, n. 3, p. 254–266, 1994.

MATSUMOTO, M.; NISHIMURA, T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, New York, v. 8, n. 1, p. 3–30, 1998.

MAZOR, S.; LANGSTRAAT, P. *A guide to VHDL*. 2nd. ed. Boston: Kluwer Academic, 1993.

MCHUGH, J. A. *Algorithmic graph theory*. Upper Saddle River: Prentice-Hall, 1990.

MORRIS, J.; BUNDELL, G. A.; THAM, S. A scalable re-configurable processor. In: AUSTRALIAN COMPUTER ARCHITECTURE CONFERENCE, 5., 2000, Canberra. *Proceedings...* Canberra: IEEE Computer Society Press, 2000. p. 64–73.

MURATA, T. Petri nets: properties, analysis and applications. In: PROCEEDINGS OF THE IEEE, 4., 1989, New York. *Proceedings...* Canberra: IEEE Computer Society Press, 1989. p. 541–580.

NEMANI, M.; NAJM, F. High-level power estimation and the area complexity of boolean functions. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, 1996, Monterey. *Proceedings...* Piscataway: IEEE Computer Society Press, 1996. p. 329–334.

NEMANI, M.; NAJM, F. N. High-level area and power estimation for vlsi circuits. In: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1997, San Jose. *Proceedings...* Washington: IEEE Computer Society Press, 1997. p. 114–119.

NILSSON, E. *Design and implementation of a hot-potato switch in a network on chip*. 2002. 66 f. Tese (M. S.) — Royal Institute of Technology, Stockholm, 2002.

NILSSON, E. et al. Load distribution with the proximity congestion awareness in a network on chip. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, 2003, Munich. *Proceedings...* Washington: IEEE Computer Society Press, 2003. p. 11126.

PANDE, P. P. et al. Design of a switch for network on chip applications. In: IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, 2003, Bangkok. *Proceedings...* Bangkok: ISCAS, 2003. p. 217–220.

PARK, S. K.; MILLER, K. W. Random number generators: good ones are hard to find. *Commun. ACM*, New York, v. 31, n. 10, p. 1192–1201, 1988.

RAMACHANDRAN, C. et al. Accurate layout area and delay modeling for system level design. In: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1992, Santa Clara. *Proceedings...* Los Alamitos: IEEE Computer Society Press, 1992. p. 355–361.

REISIG, W. *A primer in Petri net design*. Secaucus: Springer-Verlag, 1992.

ROKYTA, P.; FENGLER, W.; HUMMEL, T. Electronic system design automation using high level petri nets. *Hardware Design and Petri Nets*, Boston, v. 1, n. 4, p. 193–204, 2000.

ROSA, F. H. F. P. da; JUNIOR, V. A. P. *Gerando números aleatórios*. São Paulo: [s.n.], 2002. Relatório técnico.

SAKAMOTO, M.; MORITO, S. Combination of multiplicative congruential random-number generators with safe prime modulus. In: CONFERENCE ON WINTER SIMULATION, 27., 1995, Arlington. *Proceedings...* New York: ACM Press, 1995. p. 309–315.

SCHOLLMEYER, M. F.; TRANTER, W. H. Noise generators for the simulation of digital communication systems. In: ANNUAL SYMPOSIUM ON SIMULATION, 24., 1991, New Orleans. *Proceedings...* Los Alamitos: IEEE Computer Society Press, 1991. p. 264–275.

SOTO, E.; PEREIRA, M. Implementing a petri net specification in a fpga using vhdl. In: INTERNATIONAL WORKSHOP ON DISCRETE-EVENT SYSTEM DESIGN, 1., 2001, Zielona Góra. Zielona Góra: Oficyna Wydaw, 2001. p. 180–185.

SOTO, E.; PEREIRA, M. Implementing a petri net specification in a fpga using vhdl. In: ADAMSKI, M. A.; KARATKEVICH, A.; WEGRZYN, M. (Ed.). *Design of embedded control systems*. New York: Springer, 2005. p. 167–174.

SRINIVASAN, A.; HUBER, G.; LAPOTIN, D. Accurate area and delay estimation from rtl descriptions. *IEEE Transactions on VLSI Systems*, Canberra, v. 6, n. 1, p. 168–172, 1998.

STRUM, M.; CHAU, W. J.; NETO, J. V. V. Síntese de alto nível: plana e hierárquica. In: ROZO, A. G. (Ed.). *Sistemas digitales*: elementos para un diseño a alto nivel. Santafé de Bogotá: Uniandes, 1999. p. 181–279.

TANENBAUM, A. S. *Computer networks*. 3th. ed. Upper Saddle River: Prentice-Hall, 1996.

TEZUKA, S.; L'ECUYER, P. Analysis of add-with-carry and subtract-with-borrow generators. In: CONFERENCE ON WINTER SIMULATION, 24., 1992, Arlington. *Proceedings...* New York: ACM Press, 1992. p. 443–447.

TEZUKA, S.; L'ECUYER, P.; COUTURE, R. On the lattice structure of the add-with-carry and subtract-with-borrow random number generators. *ACM Trans. Model. Comput. Simul.*, New York, v. 3, n. 4, p. 315–331, 1993.

THESEN, A.; SUN, Z.; WANG, T.-J. Some efficient random number generators for micro-computers. In: CONFERENCE ON WINTER SIMULATION, 16., 1984, Dallas. *Proceedings...* Piscataway: IEEE Computer Society Press, 1984. p. 186–196.

THIAGARAJAN, P. S. Elementary net systems. In: BRAUER, W.; REISIG, W.; ROZENBERG, G. (Ed.). *Advances in Petri nets 1986*. London: Springer-Verlag, 1987. p. 26–59.

WANG, J. *Timed Petri nets*: theory and application. Boston: Kluwer Academic, 1998. (The Kluwer international series on discrete event dynamic systems).

WU, A. C.-H.; CHAIYAKUL, V.; GAJSKI, D. Layout-area models for high-level synthesis. In: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1991, Santa Clara. *Proceedings...* Washington: IEEE Computer Society Press, 1991. p. 34–37.

WU, J. A deterministic fault-tolerant and deadlock-free routing protocol in 2-d meshes based on odd-even turn model. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 16., 2002, New York. *Proceedings...* New York: ACM Press, 2002. p. 67–76.

WU, P.-C. Multiplicative, congruential random-number generators with multiplier $\pm 2^{k1} \pm 2^{k2}$ and modulus $2^p - 1$. *ACM Trans. Math. Softw.*, New York, v. 23, n. 2, p. 255–265, 1997.

ZEFERINO, C. A. *Introdução às Redes-em-Chip*. Itajaí: [s.n.], 2003. Relatório técnico.

ZEFERINO, C. A. et al. A study on communication issues for systems-on-chip. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 15., 2002, Porto Alegre. *Proceedings...* Washington: IEEE Computer Society Press, 2002. p. 121.

ZEFERINO, C. A.; SANTO, F. G. M. E.; SUSIN, A. A. Paris: a parameterizable interconnect switch for networks-on-chip. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 17., 2004, Pernambuco. *Proceedings...* New York: ACM Press, 2004. p. 204–209.

ZEFERINO, C. A.; SUSIN, A. A. Socin: a parametric and scalable network-on-chip. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 16., 2003, Sao Paulo. *Proceedings...* Washington: IEEE Computer Society Press, 2003. p. 169.

ANEXO A – Uma Abordagem para Análise de Desempenho no Projeto de um Roteador

Resumo

Os roteadores projetados para o sistema de comunicação da arquitetura proposta possuem decrementadores que podem ser implementados de diferentes formas: transporte propagado, transporte antecipado, transporte selecionado entre outras possibilidades. Para determinar a melhor arquitetura do decrementador que deve ser incorporada ao roteador foi desenvolvida uma abordagem baseada em equações matemáticas que computam as quantidades de portas lógicas e de níveis de lógica dos decrementadores e do roteador. Uma fórmula de desempenho foi estabelecida para realizar uma análise comparativa da arquitetura do roteador com cada um dos decrementadores. O decrementador com transporte em cascata obteve a melhor relação de desempenho desde que seu tamanho esteja entre 4 e 7 bits. A partir de 8 bits, os decrementadores que utilizam a técnica de transporte selecionado entre os blocos de decremento obtiveram os melhores resultados.

A.1 Introdução

Devido a um mercado atual bastante competitivo, o projeto de sistemas digitais tem se tornado cada vez mais complexo e exigindo um tempo de confecção bastante curto. Diante deste panorama, vem surgindo uma necessidade crescente de ferramentas CAD que possam auxiliar os projetistas na tomada de decisão durante o desenvolvimento de um sistema. Análises realizadas antes da etapa de implementação física podem economizar custos e melhorar o desempenho do sistema em relação ao consumo de energia, tempo de resposta entre outros fatores. Com isso, algumas técnicas têm sido propostas para a estimativa da área, tempo de resposta e consumo de energia de um dado sistema digital (WU;

CHAIYAKUL; GAJSKI, 1991; CHAIYAKUL; WU; GAJSKI, 1992; RAMACHANDRAN et al., 1992; SRINIVASAN; HUBER; LAPOTIN, 1998; NEMANI; NAJM, 1996, 1997; BüYüKSAHIN; NAJM, 2000, 2002; GELOSH; STELIFF, 2000). Em (WU; CHAIYAKUL; GAJSKI, 1991; CHAIYAKUL; WU; GAJSKI, 1992; RAMACHANDRAN et al., 1992) os autores realizam uma estimativa da área e do tempo de resposta de um sistema baseando-se em modelos com arquiteturas de *layouts* pré-definidas. Em (SRINIVASAN; HUBER; LAPOTIN, 1998) os autores estimam a área e o tempo de resposta total do sistema utilizando informações provenientes da síntese lógica de apenas um subconjunto de circuitos do projeto. Em (NEMANI; NAJM, 1996, 1997; BüYüKSAHIN; NAJM, 2000) os autores realizam uma estimativa da área e do consumo de energia baseando-se na quantidade e no tamanho dos implicantes primos das funções booleanas que compõem o sistema. Em (BüYüKSAHIN; NAJM, 2002) os autores descrevem o sistema por meio de uma rede booleana e, a extração de determinadas características dessa rede, como a quantidade de nós, a quantidade de arcos e o grau dos nós permite uma estimativa da área do sistema. Em (GELOSH; STELIFF, 2000) os autores utilizam técnicas de aprendizado de máquina para modelar a própria ferramenta de síntese e assim utilizar esse modelo para a obtenção da área e do tempo de resposta do sistema.

Agora, imagine-se um sistema digital composto por vários blocos digitais como mostrado na figura A.1. Imagine-se que o sistema precise executar uma determinada tarefa que pode ser implementada por meio de arquiteturas diferentes. A questão que se configura neste panorama é a seguinte: "Qual é a melhor arquitetura que deve ser implementada no bloco funcional para que o sistema como um todo atinja o seu máximo desempenho?".

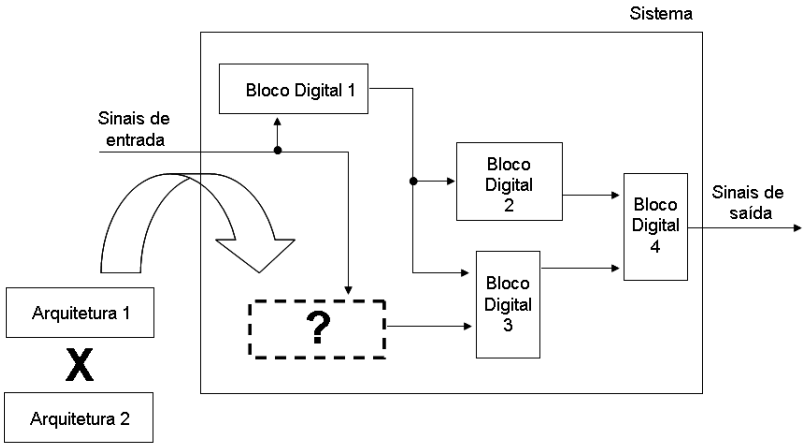


Figura A.1: Sistema digital com duas possibilidades diferentes de implementação

O desempenho depende de quais parâmetros pretende-se levar em consideração, podendo ser o custo de uma possível implementação física, tempo de resposta, consumo de energia entre outros. A pergunta é pertinente visto que nem sempre a inserção do

bloco funcional que individualmente possui o melhor desempenho acarretará no melhor desempenho para o sistema. Isto acontece porque um sistema razoavelmente complexo possui vários blocos digitais sendo executados simultaneamente o que gera vários caminhos distintos que podem ser percorridos a partir dos sinais de entrada até os sinais de saída. Desta forma, a inclusão de um bloco poderá, por exemplo, modificar o tempo de resposta de um determinado caminho mas não modificar o desempenho do caminho crítico do sistema. Portanto, desde que o caminho crítico do sistema permaneça inalterado, vale mais a pena colocar um bloco digital cujo tempo de resposta é mais demorado mas que possua uma quantidade de portas lógicas reduzida do que colocar um bloco cujo tempo de resposta é extremamente rápido mas exige uma quantidade maior de portas lógicas.

Além disso, imagine-se que os sinais de entrada do sistema apresentado na figura A.1 sejam de 32 bits e que agora estes sinais precisam ser expandidos para 64, 128 ou 256 bits. As perguntas decorrentes são: "O bloco digital que proporcionou o melhor desempenho para o sistema de 32 bits continua sendo o mais adequado no processo de expansão?" e "Qual é a melhor arquitetura que deve ser implementada no bloco em cada uma dessas expansões?".

A motivação para o desenvolvimento de uma abordagem capaz de responder a estas questões se deve aos motivos comentados a seguir.

Como explicado no capítulo 6, foi desenvolvido em VHDL um roteador capaz de determinar o trajeto de um pacote de dados sobre uma rede com topologia de malha bidimensional. Na arquitetura do roteador projetado foi necessário o uso de estruturas aritméticas (decrementadores) para computar o endereço de destino de um pacote. Arquiteturas diferentes podem ser utilizadas nessas decrementadores levando em consideração a forma de obtenção do bit de transporte. Dependendo da arquitetura utilizada para esses decrementadores o roteador pode alcançar uma velocidade maior de processamento ou/e uma área menor numa possível implementação física da arquitetura proposta num *chip* ou num dispositivo lógico programável. Uma maior velocidade de execução do sistema de comunicação, composto por um conjunto de roteadores, aumentará o tempo de resposta da arquitetura proposta no capítulo 5. Por sua vez, a redução da área de cada roteador pode reduzir o custo da fabricação em um circuito integrado dessa arquitetura proposta.

A abordagem apresentada neste anexo tem como propósito comparar o desempenho do roteador com as principais e mais conceituadas técnicas de obtenção do sinal de transporte, quais sejam, decrementadores com transporte em cascata, decrementadores com transporte antecipado e decrementadores com transporte selecionado e identificar a me-

lhor técnica que deve ser utilizada no projeto do roteador.

Na seqüência, são formulados os conceitos fundamentais e as equações matemáticas que possibilitam a comparação de desempenho da arquitetura do roteador. Posteriormente, apresentam-se os resultados obtidos sobre o desempenho do roteador para cada decrementador, finalizando, com alguns comentários e discussões sobre os resultados obtidos.

A.2 Desenvolvimento das Fórmulas de Comparação

Nesta seção são desenvolvidas as fórmulas matemáticas que possibilitam a comparação de desempenho do roteador utilizando diferentes tipos de decrementadores em sua composição. Inicialmente, expõem-se os conceitos fundamentais adotados e as operações básicas formuladas para a extração dos parâmetros utilizados na comparação, quais sejam: a quantidade de portas lógicas e a quantidade de níveis de lógica de um sistema digital. Na seqüência, especificam-se as equações matemáticas que computam as quantidades de portas lógicas necessárias e de níveis de lógica (profundidade do caminho crítico) de quatro estruturas aritméticas, quais sejam: decrementador com transporte em cascata, decrementador com transporte antecipado, uma modificação do decrementador com transporte antecipado e um decrementador com transporte selecionado. Posteriormente, é estabelecida uma fórmula capaz de definir o desempenho da arquitetura do roteador com cada uma das estruturas aritméticas por meio dos parâmetros definidos.

A.2.1 Conceitos Fundamentais

A escolha da quantidade de portas lógicas e da quantidade de níveis de lógica como parâmetros para realizar a comparação de desempenho se deve aos motivos explicados a seguir.

A quantidade de portas lógicas pode ser utilizada para determinar a quantidade de recursos que serão utilizados em um FPGA ou no caso de uma possível implementação em um circuito integrado, a quantidade de portas lógicas pode ser utilizada para estimar o tamanho da área que será ocupada, e por conseqüência, determinar o custo da concepção deste circuito.

Por sua vez, a quantidade de níveis de lógica pode ser utilizada para determinar a velocidade de operação do circuito. Pode ser utilizada inclusive para definir a taxa

teórica máxima de vazão do roteador. O roteador possui cinco portas de comunicação de entrada/saída, envia e recebe pacotes compostos por 32 bits, assim, uma estimativa da taxa teórica máxima de vazão do roteador pode ser equacionada da seguinte forma: 5 * 32 * velocidade de operação. Desta forma, as quantidades de portas lógicas e de níveis de lógica podem ser utilizadas para estimar importantes características do circuito implementado, como custo, velocidade de processamento e taxa de vazão.

A abordagem apresentada aqui realiza o cálculo das quantidades de portas lógicas e de níveis de lógica levando em consideração portas de no máximo duas entradas visto que toda função booleana, simples ou complexa, pode ser implementada por meio de portas contendo no máximo duas entradas. Assim, uma operação que envolva um número de operandos maior do que dois deve ser desmembrada em mais de uma porta lógica contendo cada uma, duas entradas.

Na figura A.2 apresenta-se o cálculo da quantidade de níveis de lógica para portas lógicas *and* com entradas de 2 até 5 operandos.

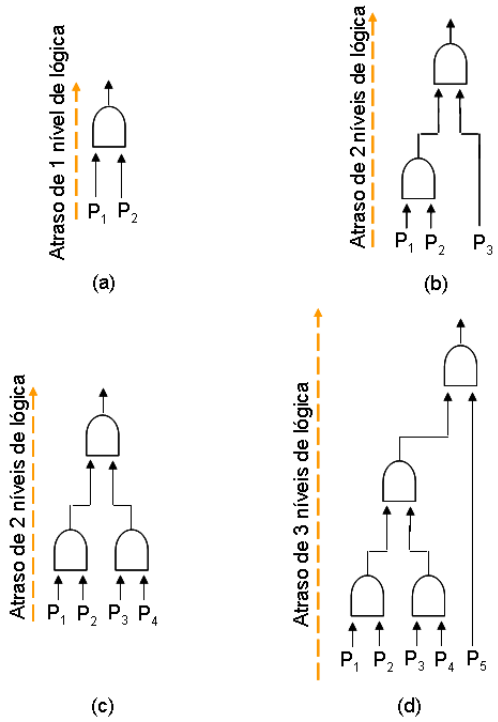


Figura A.2: Cálculo da quantidade de níveis de lógica para portas com (a) dois operandos, (b) três operandos, (c) quatro operandos e (d) cinco operandos

A realização de uma operação com 2 operandos exigirá um atraso de apenas uma porta lógica, como é o caso mostrado na figura A.2 (a) para uma operação lógica *and*. Para uma operação lógica com 3 operandos, a operação deve ser desmembrada em duas portas lógicas, como mostrado na figura A.2 (b) para a operação *and*, sendo necessário um

atraso de dois níveis de lógica. Com 4 operandos, a operação é desmembrada em 3 portas lógicas, no entanto, duas portas lógicas são processadas simultaneamente e portanto será necessário um atraso de apenas dois níveis de lógica, como mostrado na figura A.2 (c) para uma operação *and*. Uma operação contendo 5 operandos deve ser desmembrada em 4 portas lógicas, como é o caso mostrado na figura A.2 (d), duas portas lógicas *and* são processadas simultaneamente e, portanto, será necessário um atraso de 3 níveis de lógica.

Generalizando, uma operação lógica com n operandos será desmembrada em $(n-1)$ portas lógicas, sendo necessário um atraso de $\lceil \log_2 n \rceil$ níveis de lógica, como mostrado na figura A.3 para uma operação lógica *and*. O símbolo $\lceil x \rceil$ indica o menor inteiro maior ou igual a x.

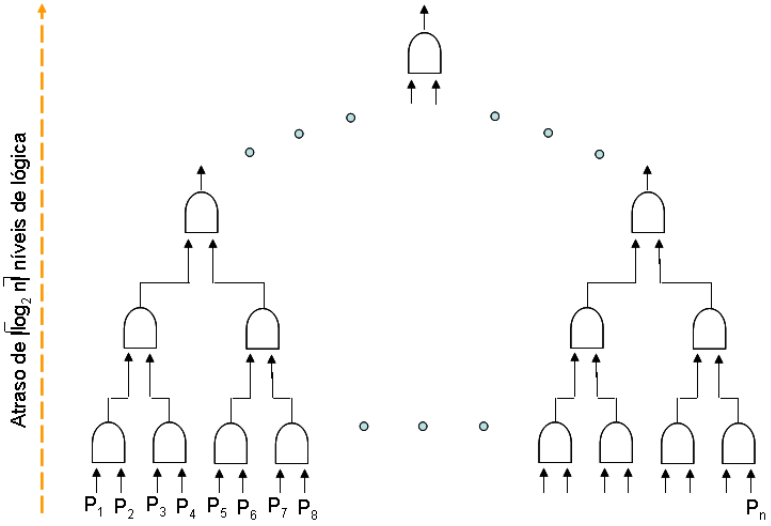


Figura A.3: Cálculo da quantidade de níveis de lógica para portas com n operandos

No cálculo da quantidade de níveis de lógica de um sistema digital pode também ser necessário o uso das funções de máximo e soma, como mostrado no diagrama de blocos da figura A.4. Um conjunto de sinais de entrada é utilizado em três módulos paralelos (Módulos 1, 2 e 4). O terceiro módulo utiliza os sinais de saída do Módulo 2 para realizar o seu processamento. Neste esquema, deve-se determinar o atraso de cada caminho existente e relacioná-los por meio da função de máximo. No sistema da figura A.4 três caminhos distintos são formados visto que os mesmos sinais de entrada são direcionados para três módulos diferentes.

O primeiro caminho possui apenas o Módulo 1 e portanto, o atraso neste caminho é equivalente ao atraso deste módulo, ou seja NL(Módulo 1), onde NL representa a quantidade de níveis de lógica. O terceiro caminho possui apenas o Módulo 4 e assim, o atraso neste caminho será de NL(Módulo 4).

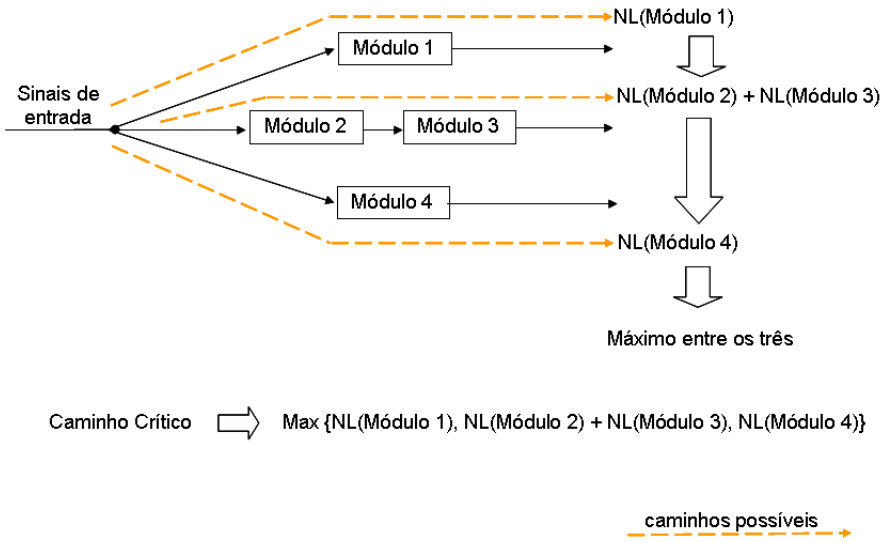


Figura A.4: Uso das funções de máximo e soma no cálculo da quantidade de níveis de lógica, onde NL é o número de níveis de lógica

O segundo caminho possui dois módulos (2 e 3) que são processados seqüencialmente. Desta forma, o atraso deste caminho é equivalente à soma dos atrasos individuais dos módulos 2 e 3. Assim, o atraso do segundo caminho será de NL(Módulo 2) + NL(Módulo 3).

Como cada um dos três caminhos são realizados em paralelo, o atraso total do sistema, ou melhor, o caminho crítico será dado pelo caminho que possuir o maior atraso, ou seja, por MAX {NL(Módulo 1), NL(Módulo 2) + NL(Módulo 3), NL(Módulo 4)}.

Resumidamente, para módulos que são processados em paralelo deve-se fazer uso da função de máximo para englobar todos os caminhos e, para módulos que são processados seqüencialmente a operação de soma deve ser utilizada para a computação de todos os atrasos que ocorrem em um determinado caminho.

Na abordagem apresentada aqui, o cálculo da quantidade de portas lógicas leva em consideração o uso de portas lógicas que possuam no máximo duas entradas. Na figura A.5 apresenta-se o cálculo da quantidade de portas lógicas *and* para operações contendo de 2 até 5 operandos. A realização de uma operação com 2 operandos necessita de apenas uma porta lógica, como é o caso mostrado na figura A.5 (a) para uma operação lógica *and*. Uma operação lógica com 3 operandos deve ser desmembrada em duas portas lógicas, como mostrado na figura A.5 (b) para a operação *and*. Com 4 operandos, a operação é desmembrada em três portas lógicas, como mostrado na figura A.5 (c). Uma operação contendo 5 operandos deve ser desmembrada em quatro portas lógicas, como é o caso mostrado na figura A.5 (d).

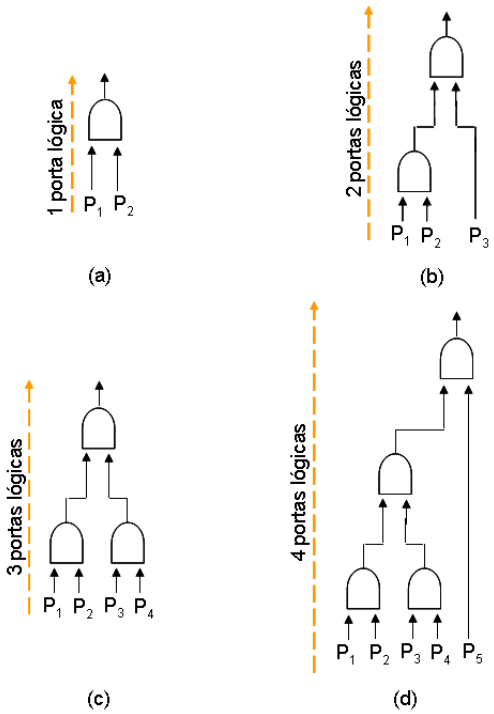


Figura A.5: Cálculo da quantidade de portas lógicas *and* com (a) dois operandos, (b) três operandos, (c) quatro operandos e (d) cinco operandos

Generalizando, uma operação lógica com n operandos será desmembrada em $(n-1)$ portas lógicas, como mostrado na figura A.6 para uma operação lógica *and*.

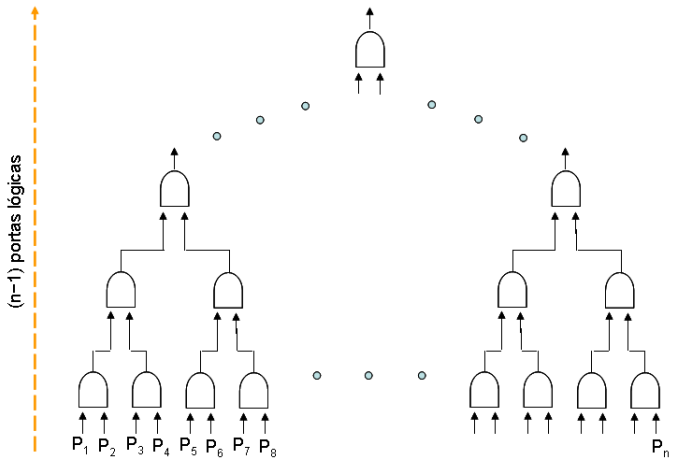


Figura A.6: Cálculo da quantidade de portas lógicas com n operandos

Nas próximas quatro seções os conceitos apresentados aqui são aplicados a cada um dos decrementadores considerados, para a obtenção dos respectivos parâmetros de desempenho.

A.2.2 Decrementador com Transporte em Cascata

Na figura A.7 apresenta-se um somador com o bit de transporte (vai-um) em cascata (MANO, 1991) para operandos (X e Y) de n bits. O sinal de transporte (C) se propaga desde o bit menos significativo até o mais significativo para a produção da soma (S). Com esta estrutura é possível construir um somador com qualquer número de bits para os operandos, bastando para isso ligar em cascata tantos somadores completos quantos forem necessários. Na figura A.8 apresenta-se o circuito somador completo de 1 bit, utilizado na composição do somador mostrado na figura A.7.

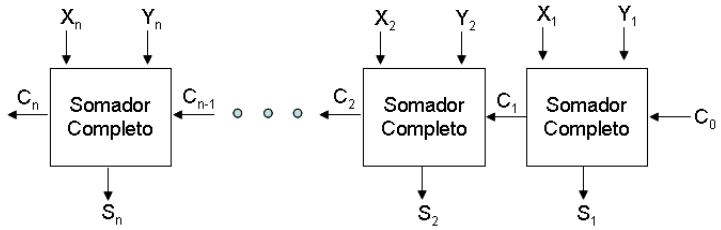


Figura A.7: Somador com transporte em cascata

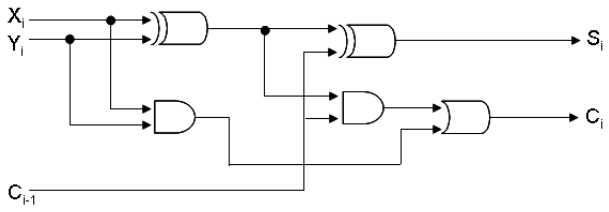


Figura A.8: Circuito somador completo utilizado na composição do somador com transporte em cascata

Para o projeto do roteador, a arquitetura do somador com transporte em cascata mostrada na figura A.9 pode ser reduzida tendo em vista que a operação a ser realizada é um decremento de uma unidade e que no último bit não há a necessidade da produção do bit de transporte. Assim sendo, e utilizando o complemento de um, o operando Y passa a ser Y = 11...10 e o sinal vem-um do primeiro bit será $C_{in} = 1$. A adaptação do somador para o projeto do roteador pode ser observada na figura A.9. Na figura A.9 (a) apresenta-se o circuito de soma do primeiro bit. Na figura A.9 (b) apresenta-se o circuito de soma dos bits intermediários e na figura A.9 (c) apresenta-se o circuito de soma do último bit.

Embora práticos, estes somadores ou decrementadores apresentam muito atraso na propagação do sinal de transporte, o que os torna inadequados para aplicações que precisam de velocidade no processo de obtenção da soma.

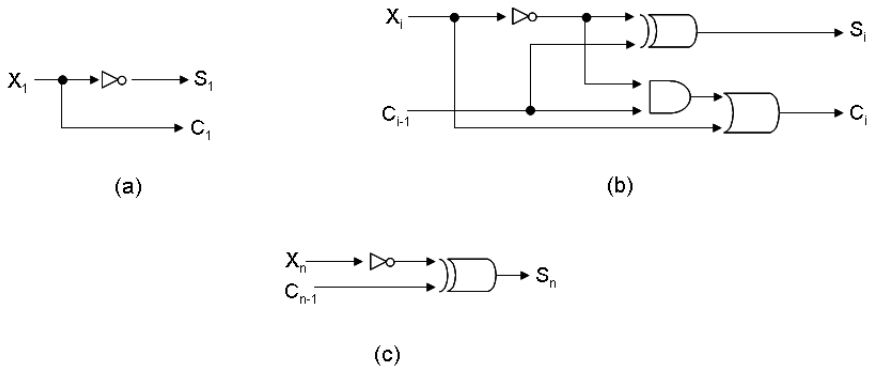


Figura A.9: Circuito decrementador com transporte em cascata para o projeto do roteador

Na seqüência, são apresentadas as fórmulas que descrevem a quantidade de portas lógicas e de níveis de lógica da arquitetura do decrementador com transporte em cascata baseando-se na figura A.9.

Com relação à quantidade de portas lógicas, tem-se que no primeiro bit de soma, mostrado na figura A.9 (a), é necessária apenas uma porta lógica *not*. No último bit, duas portas lógicas são necessárias enquanto os bits intermediários precisam de quatro portas lógicas. Assim, levando em consideração que Δx representa a quantidade de bits do decrementador, a fórmula que define a quantidade de portas lógicas do decrementador com transporte em cascata é dada por:

$$QP = 1 + 4 * (\Delta x - 2) + 2$$

Desde que $\Delta x \geq 2$.

O sinal de transporte em cascata é o principal elemento de atraso na obtenção do resultado de soma. De acordo com a figura A.10, não há necessidade de nenhuma porta lógica para a produção do primeiro vai-um. No segundo bit de soma, a produção do sinal de transporte passa, no caminho crítico, por uma porta lógica *not*, uma porta lógica *and* e por uma porta lógica *or*. Portanto, no segundo bit, o atraso na produção do vai-um é de três níveis de lógica. No último bit, o atraso ocorre apenas na produção do sinal de soma $S_{\Delta x}$, sendo de um nível de lógica. Para os bits restantes (intermediários), o atraso na propagação do vai-um é de dois níveis de lógica, sendo uma porta *and* e uma porta *or*. Desta maneira, a fórmula que define a quantidade de níveis de lógica do decrementador com transporte em cascata é determinada por:

$$\begin{aligned} NL &= 0 + 3 + 2 * (\Delta x - 3) + 1 \\ NL &= 2 * \Delta x - 2 \end{aligned}$$

Desde que $\Delta x \geq 3$.

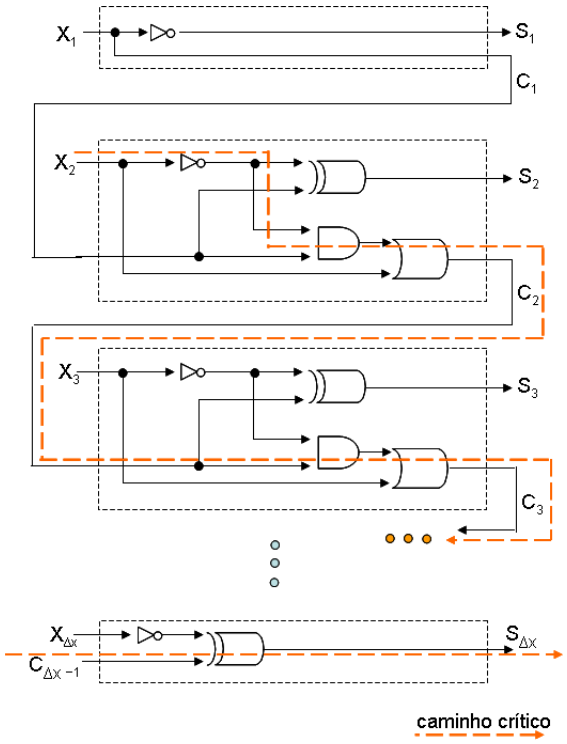


Figura A.10: Caminho crítico para o decrementador com transporte em cascata

A.2.3 Decrementador com Transporte Antecipado

Os somadores em cascata apresentados anteriormente são mais simples e mais econômicos em termos de *hardware*. No entanto, são também os mais lentos, pois o caminho crítico do circuito é definido pela propagação do sinal de transporte de bit para bit. Portanto, para operandos com um número de bits elevado este atraso poderá se tornar inaceitável em muitas aplicações.

Uma alternativa mais rápida do que o somador com transporte em cascata pode ser obtida utilizando-se um maior número de portas lógicas. Para este propósito, o somador com transporte antecipado (ERCEGOVAC; LANG; MORENO, 2000) (MANO, 1991) é capaz de determinar os bits de transporte de todos módulos de soma utilizando apenas os sinais de entrada dos operandos (X e Y) e o vem-um inicial (C_0). Com isso, o cálculo dos bits de transporte é realizado simultaneamente, não necessitando realizar a propagação do vai-um como acontece no somador com transporte em cascata.

Na figura A.11 apresenta-se a arquitetura de um somador com transporte antecipado. O circuito gerador de transporte antecipado determina os valores de todos os bits de transporte intermediários antes que os bits correspondentes da soma sejam computados.

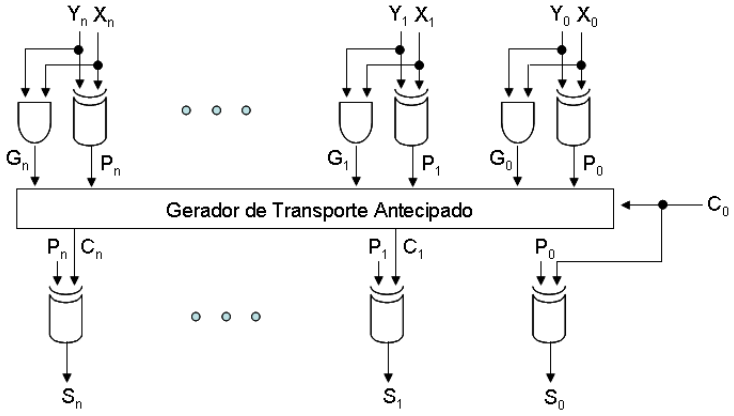


Figura A.11: Somador com transporte antecipado

Na seqüência, estes sinais de transporte pré-computados são utilizados para determinar o valor dos bits de soma. Para o cálculo do bit de transporte, dois sinais auxiliares são gerados (ERCEGOVAC; LANG; MORENO, 2000):

- G_i: saída que indica a geração de transporte no nível i. A única situação em que a soma de dois bits gera necessariamente transporte é quando os dois são 1. Assim, $G_i = X_i.Y_i$, onde i representa um estágio (bit) do somador,

- P_i: saída que indica a propagação de transporte para o nível seguinte caso venha transporte do nível anterior. Esta situação ocorre quando um dos bits dos operandos de entrada for 1, ou seja, $P_i = X_i \oplus Y_i$.

A expressão computada pelo circuito gerador de transporte antecipado e que define os transportes pode ser apresentada da seguinte forma:

$$C_{i+1} = G_i + P_i.C_i$$

Por substituição pode-se obter as expressões dos transportes utilizando apenas os sinais de entrada, representados pelos sinais auxiliares G e P, como mostrado no exemplo abaixo para C_1, C_2 e C_3:

$$\begin{aligned} C_1 &= G_0 + P_0.C_0 \\ C_2 &= G_1 + P_1.C_1 \\ &= G_1 + G_0.P_1 + P_0.P_1.C_0 \\ C_3 &= G_2 + P_2.C_2 \\ &= G_2 + G_1.P_2 + G_0.P_1.P_2 + P_0.P_1.P_2.C_0 \end{aligned}$$

Como mostrado na figura A.11, após o cálculo dos transportes pelo circuito gerador de transporte antecipado, a soma é então computada pela expressão $S_i = P_i \oplus C_i$.

Para o projeto do roteador, a arquitetura do somador com transporte antecipado mostrada na figura A.11 pode ser modificada tendo em vista que a operação a ser realizada é um decremento de uma unidade. Assim sendo, e utilizando o complemento de um, o operando Y passa a ser Y = 11...10 e o sinal vem-um do primeiro bit será $C_0 = 1$. Desta forma, os sinais auxiliares G e P, e a soma podem ser adaptados para o projeto do roteador, assim: $P_i = X_i \oplus Y_i = X_i \oplus 1 = \overline{X_i}$; $G_i = X_i.Y_i = X_i.1 = X_i$ e $S_i = P_i \oplus C_i$, sendo $i \geq 1$. Para $i = 0$, tem-se que: $P_0 = X_0 \oplus Y_0 = X_0 \oplus 0 = X_0$; $G_0 = X_0.Y_0 = X_i.0 = 0$ e $S_0 = P_0 \oplus C_0 = X_0 \oplus 1 = \overline{X_0}$.

Quanto ao circuito gerador de transporte antecipado, tem-se a seguinte adaptação:

$$
\begin{aligned}
C_1 &= 0 + P_0 \\
C_2 &= G_1 + 0 + P_1.P_0 \\
C_3 &= G_2 + P_2.G_1 + 0 + P_2.P_1.P_0 \\
C_4 &= G_3 + P_3.G_2 + P_3.P_2.G_1 + 0 + P_3.P_2.P_1.P_0 \\
C_5 &= G_4 + P_4.G_3 + P_4.P_3.G_2 + P_4.P_3.P_2.G_1 + 0 + P_4.P_3.P_2.P_1.P_0 \\
&\vdots \quad\quad \vdots \\
C_{n+1} &= G_n + P_n.G_{n-1} + P_n.P_{n-1}.G_{n-2} + P_n.P_{n-1}.P_{n-2}.G_{n-3} + \ldots + \\
&\quad + P_n.P_{n-1}.P_{n-2} \ldots P_2.G_1 + 0 + P_n.P_{n-1} \ldots P_0
\end{aligned}
$$

Com relação à quantidade de níveis de lógica, pode ser observado que os sinais auxiliares P e G precisam de no máximo um nível de lógica, correspondente ao atraso de uma porta lógica *not*. Para a produção da soma, de acordo com a fórmula $S_i = P_i \oplus C_i$, é necessário o atraso de mais uma porta lógica. Na figura A.12 apresenta-se o caminho crítico do decrementador com transporte antecipado.

O circuito gerador de transporte antecipado realiza o cálculo dos sinais C_i em paralelo e, portanto, apenas o atraso do bit de transporte do último nível deve ser levado em consideração, visto que este sinal de transporte possui a expressão booleana que gera o maior atraso. Tomando $n + 1$ como o índice de maior nível, tem-se:

$$
\begin{aligned}
C_{n+1} &= G_n + P_n.G_{n-1} + P_n.P_{n-1}.G_{n-2} + P_n.P_{n-1}.P_{n-2}.G_{n-3} + \ldots + \\
&\quad + P_n.P_{n-1}.P_{n-2} \ldots P_2.G_1 + 0 + \underbrace{P_n.P_{n-1} \ldots P_0}_{\text{maior atraso}}
\end{aligned}
$$

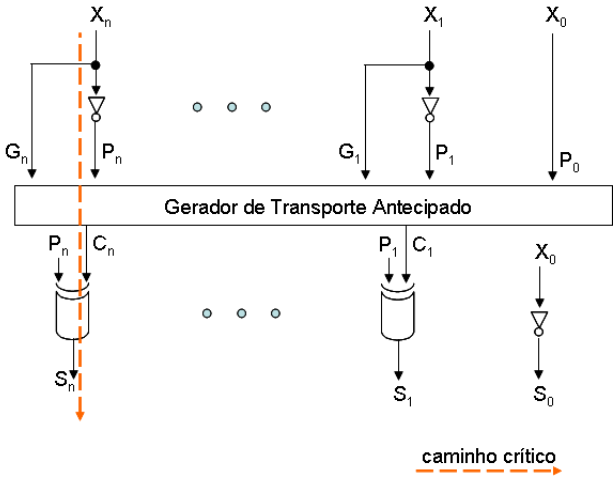


Figura A.12: Caminho crítico para o decrementador com transporte antecipado

Note-se que para a geração do sinal C_{n+1}, os mintermos (termos com somente operações *and*) são realizados simultaneamente e, portanto, deve-se levar em consideração apenas o mintermo com maior atraso, ou seja, o mintermo que possuir um maior número de operandos. Na expressão anterior, o mintermo de maior atraso, possui $(n+1)$ entradas, que corresponde a um atraso de $\lceil \log_2(n+1) \rceil$ níveis de lógica. Após a execução da operação lógica *and*, deve-se computar o atraso máximo gerado pela realização das operações lógicas *or*, visto que os mintermos e as operações booleanas *or* são realizadas seqüencialmente. Na expressão anterior para o sinal C_{n+1}, as operações lógicas *or* terão $(n+1)$ operandos de entrada, visto que $C_{n+1} = G_n + \ldots + G_{n-1} + \ldots G_{n-2} + \ldots\ G_1 + P_n.P_{n-1} \ldots P_0$. Portanto, para a realização das operações *or* será necessário um atraso máximo de $\lceil \log_2(n+1) \rceil$ níveis de lógica.

Assim, a fórmula completa indicando a quantidade de níveis de lógica da arquitetura do decrementador com transporte antecipado será:

$$\begin{aligned} NL &= \lceil \log_2(n+1) \rceil + \lceil \log_2(n+1) \rceil + 2 \\ &= 2 * \lceil \log_2(n+1) \rceil + 2 \end{aligned}$$

Desde que $n \geq 0$.

Colocando a fórmula em termos de Δx $(n = \Delta x - 2)$, tem-se:

$$NL = 2 * \lceil \log_2(\Delta x - 1) \rceil + 2$$

Desde que $\Delta x \geq 2$.

Com relação à quantidade de portas lógicas do decrementador com transporte anteci-

pado, primeiro, será calculada a quantidade de portas do gerador de transporte antecipado e depois, será computada as lógicas de produção dos sinais auxiliares P e G, e soma.

Para o cálculo da quantidade de portas lógicas do gerador de transporte, tem-se:

$$\begin{aligned}
C_1 &= 0 + P_0 \\
C_2 &= G_1 + 0 + P_1.P_0 \\
C_3 &= G_2 + P_2.G_1 + 0 + P_2.P_1.P_0 \\
C_4 &= G_3 + P_3.G_2 + P_3.P_2.G_1 + 0 + P_3.P_2.P_1.P_0 \\
&\vdots \quad \vdots \\
C_n &= G_{n-1} + P_{n-1}.G_{n-2} + P_{n-1}.P_{n-2}.G_{n-3} + P_{n-1}.P_{n-2}.P_{n-3}.G_{n-4} + \ldots + \\
&\quad + P_{n-1}.P_{n-2}.P_{n-3} \ldots P_2.G_1 + 0 + P_{n-1}.P_{n-2} \ldots P_0
\end{aligned}$$

Da expressão anterior, a quantidade de portas lógicas necessárias para implementar a lógica do sinal C_1 é zero, para o sinal C_2 é 2 (uma operação lógica *and* entre P_1 e P_0 e uma operação lógica *or* entre os operandos G_1 e o resultado da operação *and*), assim:

$$\begin{aligned}
QP(C_1) &= 0 \\
QP(C_2) &= 2 \\
QP(C_3) &= 5 \\
QP(C_4) &= 9 \\
QP(C_5) &= 14
\end{aligned}$$

O sinal C_n possui n operandos para a realização da operação binária *or*, visto que:

$$C_n = \underbrace{G_{n-1} + \ldots G_{n-2} + \ldots G_{n-3} + \ldots G_{n-4} + \ldots + \ldots G_1}_{n-1 \text{ operandos}} + P_{n-1} \ldots P_0$$

Assim, para a realização da operação *or* são necessárias $(n-1)$ portas lógicas *or* de duas entradas.

O sinal C_n ainda possui $(n-1)$ termos contendo operações *and*:

$$C_n = G_{n-1} + \underbrace{\ldots G_{n-2} + \ldots G_{n-3} + \ldots G_{n-4} + \ldots + \ldots G_1}_{n-2 \text{ operandos}} + P_{n-1} \ldots P_0$$

A cada termo *and*, da esquerda para a direita da expressão de C_n, acrescenta-se um operando a mais, como mostrado na tabela A.1.

Tabela A.1: Acréscimo de portas lógicas de um termo para outro

Número do Termo	**Quantidade de portas *and***	**Correspondência**
1	1	$P_{n-1}.G_{n-2}$
2	2	$P_{n-1}.P_{n-2}.G_{n-3}$
3	3	$P_{n-1}.P_{n-2}.P_{n-3}.G_{n-4}$
4	4	$P_{n-1}.P_{n-2}.P_{n-3}.P_{n-4}.G_{n-5}$
$\vdots$	$\vdots$	$\vdots$
$(n-1)$	$(n-1)$	$P_{n-1}.P_{n-2}\ldots P_0$

Assim, de acordo com a tabela A.1, para a realização de todas as operações *and* em todos os termos do sinal C_n são necessárias $\sum_{i=1}^{n-1} i$ portas lógicas *and* de duas entradas. Desta forma, tem-se:

$$QP(C_n) = \underbrace{(n-1)}_{\text{portas } or} + \underbrace{\sum_{i=1}^{n-1} i}_{\text{portas } and}$$

A quantidade total de portas lógicas do gerador de transporte antecipado (GTA) pode então ser expressa da seguinte maneira:

$$QP(\text{GTA}) = \underbrace{\sum_{j=1}^{n} \overbrace{\left[(j-1) + \sum_{i=1}^{j-1} i\right]}^{QP(C_j)}}_{QP(C_1)+QP(C_2)+\ldots+QP(C_n)}$$

Expandindo, tem-se:

$$QP(\text{GTA}) = \sum_{j=1}^{n}\sum_{i=1}^{j-1} i + \sum_{j=1}^{n} j - n$$

Para finalizar o cálculo da quantidade de portas do decrementador com transporte antecipado, é necessário computar a quantidade de portas lógicas dos sinais auxiliares P e G, e soma.

Como comentado anteriormente, $P_0 = X_0$, $G_0 = 0$ e $S_0 = \overline{X_0}$ e assim, para a geração dos sinais auxiliares e soma do primeiro bit, será necessário apenas uma porta lógica.

Para os demais P_i, G_i e S_i, tem-se $P_i = \overline{X_i}$, $G_i = X_i$ e $S_i = P_i \oplus C_i$. Assim, para cada (P_i, G_i e S_i) precisa-se de duas portas lógicas.

Portanto, tomando n como sendo o tamanho (quantidade de bits) do decrementador com transporte antecipado, a quantidade de portas lógicas para todos os sinais auxiliares

e somas é dada por:

$$QP(\text{P, G e S}) = 1 + 2 * n$$

A quantidade de portas lógicas do decrementador com transporte antecipado, é dada por:

$$QP = QP(\text{GTA}) + QP(\text{P, G e S})$$

Substituindo, tem-se:

$$QP = \left(\sum_{j=1}^{n} \sum_{i=1}^{j-1} i + \sum_{j=1}^{n} j - n \right) + (1 + 2 * n)$$

Desde que $n \geq 0$.

Colocando a fórmula em termos de Δx ($n = \Delta x - 1$), tem-se:

$$QP = \sum_{j=1}^{\Delta x - 1} \sum_{i=1}^{j-1} i + \sum_{j=1}^{\Delta x - 1} j + \Delta x$$

Desde que $\Delta x \geq 1$.

A.2.4 Modificação do Decrementador com Transporte Antecipado

É possível realizar uma diminuição na quantidade de portas lógicas do gerador de transporte antecipado, observando que:

$$\begin{aligned}
C_1 &= 0 + P_0 \\
C_2 &= G_1 + 0 + P_1.P_0 \\
C_3 &= G_2 + P_2.G_1 + 0 + P_2.\underbrace{P_1.P_0}_{repetida} \\
C_4 &= G_3 + P_3.G_2 + P_3.\underbrace{P_2.G_1}_{repetida} + 0 + P_3.\underbrace{P_2.P_1.P_0}_{repetida} \\
&\vdots \quad \vdots \\
C_{n-1} &= G_{n-2} + P_{n-2}.G_{n-3} + P_{n-2}.P_{n-3}.G_{n-4} + P_{n-2}.P_{n-3}.P_{n-4}.G_{n-5} + \ldots + \\
&\quad + P_{n-2}.P_{n-3}.P_{n-4} \ldots P_2.G_1 + 0 + P_{n-2}.P_{n-3} \ldots P_0 \\
C_n &= G_{n-1} + P_{n-1}.G_{n-2} + P_{n-1}.\underbrace{P_{n-2}.G_{n-3}}_{repetida} + P_{n-1}.\underbrace{P_{n-2}.P_{n-3}.G_{n-4}}_{repetida} + \ldots + \\
&\quad + P_{n-1}.\underbrace{P_{n-2}.P_{n-3} \ldots P_2.G_1}_{repetida} + 0 + P_{n-1}.\underbrace{P_{n-2} \ldots P_0}_{repetida}
\end{aligned}$$

Da expressão anterior, a quantidade de portas lógicas necessárias para implementar a lógica do sinal C_1 é zero, para o sinal C_2 é 2 (uma operação lógica *and* entre P_1 e P_0 e uma operação lógica *or* entre os operandos G_1 e o resultado da operação *and*), para o sinal C_3 é 4, visto que a operação lógica *and* $P_1.P_0$ já foi calculada no bit de transporte anterior C_2 e não será incluída novamente. Assim:

$$\begin{aligned} QP(C_1) &= 0 \\ QP(C_2) &= 2 \\ QP(C_3) &= 4 \\ QP(C_4) &= 6 \\ QP(C_5) &= 8 \end{aligned}$$

Para a contagem da quantidade de portas lógicas do sinal C_n deve-se levar em consideração que cada operação repetida é considerada um único operando. Desta forma, atribuiu-se a cada operação repetida uma determinada variável: $x = P_{n-2}.G_{n-3}$, $y = P_{n-2}.P_{n-3}.G_{n-4}$, $z = P_{n-2}.P_{n-3}\ldots P_2.G_1$ e $h = P_{n-2}\ldots P_0$. Portanto, a quantidade de portas lógicas *and* necessárias para cada mintermo do sinal C_n é dada por:

$$C_n = G_{n-1} + \underbrace{P_{n-1}.G_{n-2}}_{1\text{ porta}} + \underbrace{P_{n-1}.x}_{1\text{ porta}} + \underbrace{P_{n-1}.y}_{1\text{ porta}} + \ldots + \underbrace{P_{n-1}.z}_{1\text{ porta}} + 0 + \underbrace{P_{n-1}.h}_{1\text{ porta}}$$

Como demonstrado na seção anterior, o sinal C_n possui $(n-1)$ mintermos e como cada mintermo precisa de apenas uma porta lógica, então são necessárias $(n-1)$ portas lógicas *and* de duas entradas.

O sinal C_n possui n operandos para a realização da operação binária *or*, visto que:

$$C_n = \underbrace{G_{n-1} + \ldots G_{n-2} + \ldots G_{n-3} + \ldots G_{n-4} + \ldots + \ldots G_1}_{n-1\text{ operandos}} + P_{n-1}\ldots P_0$$

Assim, para a realização da operação *or* são necessárias $(n-1)$ portas lógicas *or* de duas entradas.

Portanto, a quantidade de portas lógicas *and* e *or* necessárias para implementar o sinal de transporte C_n é dada por:

$$\begin{aligned} QP(C_n) &= \underbrace{(n-1)}_{\text{portas } and} + \underbrace{(n-1)}_{\text{portas } or} \\ QP(C_n) &= 2*(n-1) \end{aligned}$$

A quantidade total de portas lógicas do gerador de transporte antecipado (GTA) pode então ser expressa da seguinte maneira:

$$QP(\text{GTA}) = \underbrace{\sum_{i=1}^{n} \overbrace{2 * (i-1)}^{QP(C_i)}}_{QP(C_1)+\ldots+QP(C_n)}$$

Expandindo, tem-se:

$$QP(\text{GTA}) = \sum_{i=1}^{n} 2*(i-1) = 2*\sum_{i=1}^{n}(i-1) = 2*\sum_{i=1}^{n} i - 2*\sum_{i=1}^{n} 1 = 2*\sum_{i=1}^{n} i - 2*n = 2*\sum_{i=1}^{n-1} i$$

Para finalizar o cálculo da quantidade de portas do decrementador é necessário computar a quantidade de portas lógicas dos sinais auxiliares P e G, e soma. Esses cálculos já foram realizados na seção anterior e, portanto, a quantidade de portas lógicas do decrementador com transporte antecipado modificado, é definida por:

$$QP = QP(\text{GTA}) + QP(\text{P, G e S})$$

Substituindo, tem-se:

$$QP = 2 * \sum_{i=1}^{n-1} i + 1 + 2 * n$$

Desde que $n \geq 0$.

Colocando a fórmula em termos de Δx ($n = \Delta x - 1$), tem-se:

$$QP = 2 * \sum_{i=1}^{\Delta x-2} i + 1 + 2 * (\Delta x - 1) = 2 * \sum_{i=1}^{\Delta x-2} i + 2 * \Delta x - 1$$

Desde que $\Delta x \geq 1$.

Com relação à quantidade de níveis de lógica, os sinais auxiliares P e G precisam de no máximo um nível de lógica e a soma precisa também de um nível de lógica, como comentado na seção anterior.

O circuito gerador de transporte antecipado realiza o cálculo dos sinais C_i em paralelo e, portanto, apenas o atraso do bit de transporte do último nível deve ser levado em consideração, visto que este sinal de transporte possui a expressão booleana que gera o maior atraso. Tomando $n + 1$ como o índice de maior nível, tem-se:

$$C_{n+1} = G_n + P_n.G_{n-1} + P_n.P_{n-1}.G_{n-2} + P_n.P_{n-1}.P_{n-2}.G_{n-3} + \ldots +$$

$$+P_n.P_{n-1}.P_{n-2}\ldots P_2.G_1 + 0 + \underbrace{P_n.P_{n-1}\ldots P_0}_{\text{maior atraso}}$$

Como comentado na seção anterior, na geração do sinal C_{n+1} os mintermos são realizados simultaneamente e, portanto, deve-se levar em consideração apenas o mintermo com maior atraso, ou seja, o mintermo que possuir um maior número de operandos. Na expressão anterior, o mintermo de maior atraso, possui $(n+1)$ entradas.

No caso do decrementador com transporte antecipado, a realização das operações *and* tem um atraso de $\lceil \log_2(n+1) \rceil$ níveis de lógica, como mostra a figura A.13.

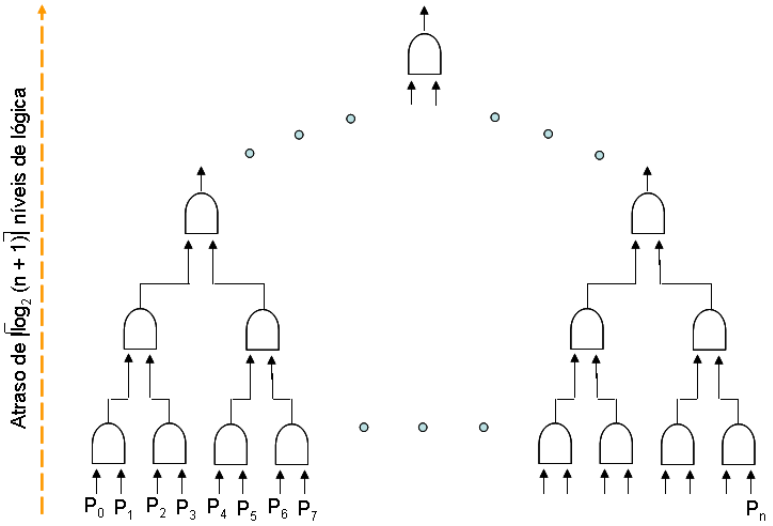


Figura A.13: Disposição dos operandos lógicos *and* no decrementador com transporte antecipado

No entanto, no caso do decrementador com transporte antecipado modificado, a diminuição do número de portas lógicas para a obtenção dos sinais de transporte acarreta um aumento de níveis de lógica relacionados ao tempo de resposta do decrementador. Como as operações lógicas *and* repetidas não são incluídas novamente, a operação lógica $P_n.P_{n-1}\ldots P_0$ passa a ser construída de maneira diferente, como mostrado na figura A.14. Nesta figura, há $(n+1)$ operandos e, portanto, o atraso causado por essa operação será de n níveis de lógica.

Após a execução da operação lógica *and*, deve-se computar o atraso máximo gerado pela realização das operações lógicas *or*, visto que os mintermos e as operações booleanas *or* são realizadas seqüencialmente. Na expressão anterior para o sinal C_{n+1}, as operações lógicas *or* terão $(n+1)$ operandos de entrada, visto que $C_{n+1} = G_n + \ldots + G_{n-1} + \ldots G_{n-2} + \ldots\ G_1 + P_n.P_{n-1}\ldots P_0$. Portanto, para a realização das operações *or* será necessário um atraso máximo de $\lceil \log_2(n+1) \rceil$ níveis de lógica.

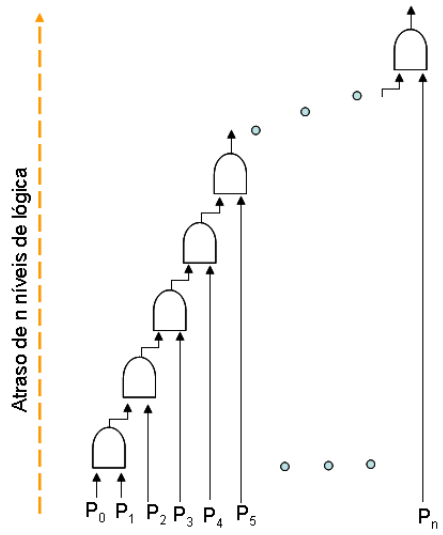


Figura A.14: Disposição dos operandos lógicos *and* no decrementador com transporte antecipado modificado

Assim, a fórmula completa indicando a quantidade de níveis de lógica do decrementador com transporte antecipado modificado é:

$$NL = \lceil log_2(n+1) \rceil + n + 2$$

Desde que $n \geq 0$.

Colocando a fórmula em termos de Δx ($n = \Delta x - 2$), tem-se:

$$NL = \lceil log_2(\Delta x - 1) \rceil + (\Delta x - 2) + 2 = \lceil log_2(\Delta x - 1) \rceil + \Delta x$$

Desde que $\Delta x \geq 2$.

A.2.5 Decrementador com Transporte Selecionado

Na figura A.15 está representado um somador com transporte selecionado (KATZ, 1994). A estratégia que esta arquitetura usa é duplicar o número de circuitos somadores para calcular em paralelo as duas possibilidades para o valor do bit de transporte. Assim, um destes circuitos tem $C_{in} = 0$ e o outro $C_{in} = 1$. O bit C_{out} do bloco anterior, quando estiver disponível, será utilizado para fazer a seleção da alternativa correta.

No exemplo apresentado na figura A.15, cada bloco calcula em paralelo dois resultados, um contendo o vai-um ($C_{in} = 1$) realizado pelo circuito Somador 2 e o outro não leva em consideração o vai-um ($C_{in} = 0$) sendo realizado pelo circuito Somador 1. Quando o sinal de transporte do bloco anterior (C_M, por exemplo) estiver disponível, um multiplexador

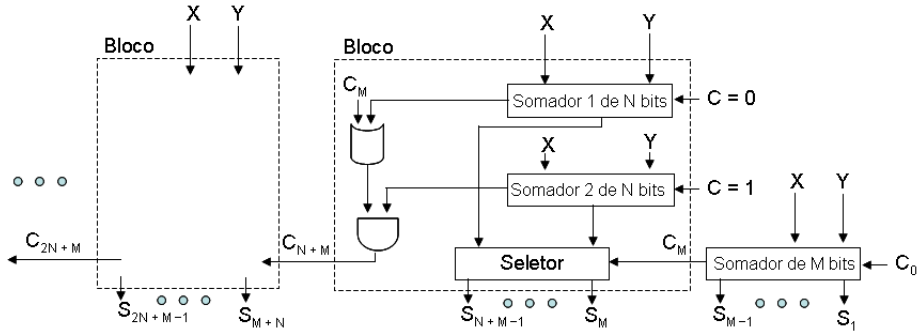


Figura A.15: Somador com transporte selecionado

selecionará a soma correta, que será proveniente ou do circuito Somador 1 ou do circuito Somador 2. O circuito responsável pela seleção ou multiplexação entre o Somador 1 e o Somador 2 é mostrado na figura A.16, para o sinal de transporte C_M.

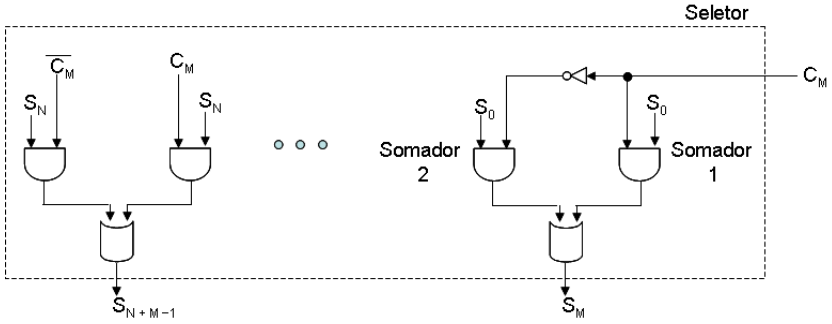


Figura A.16: Processo de multiplexação do somador com transporte selecionado para C_M

Para o projeto do roteador, a arquitetura do somador com transporte selecionado mostrada na figura A.15 pode ser reduzida tendo em vista que a operação a ser realizada é um decremento de uma unidade. Assim sendo, e utilizando o complemento de dois, o operando Y de cada bloco passa a ser Y = 11...11. Desta forma, o resultado da soma em cada bloco no Somador 2 ($C_{in} = 1$) será:

Vai-Um	=	1	...	1	1
Y	=	1	...	1	1
X	=	X_j	...	X_1	X_0
Soma	=	X_j	...	X_1	X_0

Ou seja, em cada bloco o resultado da soma no circuito Somador 2 é igual ao operando de entrada X.

Por sua vez, o circuito Somador 1 de N bits tem, em cada bloco, as entradas fixas Y = 11...11 e $C_{in} = 0$, o que implica em um circuito decrementador. O mesmo ocorre com o circuito Somador de M bits que também pode ser considerado um circuito decrementador.

Na figura A.17 apresenta-se a arquitetura do decrementador com transporte selecionado para o projeto do roteador.

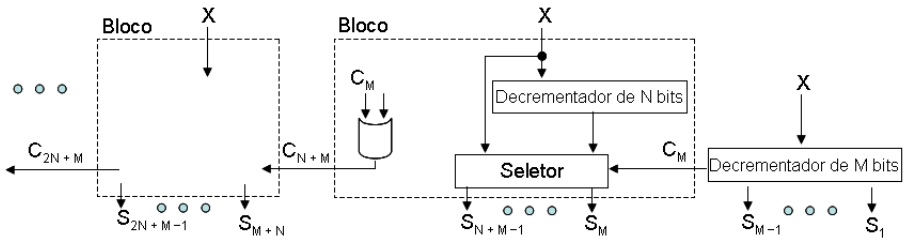


Figura A.17: Decrementador com transporte selecionado para o projeto do roteador

Na seqüência serão apresentadas as fórmulas que descrevem as quantidades de portas lógicas e de níveis de lógica do decrementador com transporte selecionado mostrado na figura A.17. As fórmulas levam em consideração as seguintes definições: Δx = quantidade total de bits do decrementador, N = quantidade de bits de um decrementador pertencente a um bloco, num_bloc = $\lfloor \frac{\Delta x}{N} \rfloor - 1$, M = $\Delta x - (num\_bloc) * (N)$. O símbolo $\lfloor x \rfloor$ corresponde ao maior inteiro menor ou igual a x.

Na arquitetura do decrementador com transporte selecionado os circuitos Decrementador de N bits e Decrementador de M bits podem ser de qualquer tipo, ou seja, podem ser decrementadores com transporte em cascata, com transporte antecipado entre outras possibilidades.

Com relação à quantidade de portas lógicas, pode-se notar que em cada bloco existe um circuito Decrementador e, portanto, $(num\_bloc) * QP(N \quad bit \quad \text{decrementador})$ portas lógicas são necessárias. Em cada bloco também existe um circuito seletor. Como mostrado na figura A.16, cada seletor possui três portas lógicas para cada bit mais uma porta lógica referente à operação booleana de complemento (*not*). Desta forma, para a implementação dos seletores em todos os blocos são necessárias $(3 * N + 1) * (num\_bloc)$ portas lógicas. Finalizando, para a realização do bit de transporte de cada bloco, uma porta lógica é necessária. Levando em consideração que no último bloco não é preciso identificar o vai-um de saída, tem-se que $(num\_bloc - 1)$ portas lógicas são necessárias para implementar o circuito responsável pela definição do transporte em todos os blocos. Assim, a fórmula da quantidade de portas lógicas do decrementador com transporte selecionado é:

$$QP = QP(\text{decrementador} \quad M \quad \text{bits}) + (num\_bloc) * QP(\text{decrementador} \quad N \quad \text{bits}) + \\ +3 * N * (num\_bloc) + 2 * (num\_bloc) - 1$$

Desde que $(num\_bloc) \geq 1$.

Com relação à quantidade de níveis de lógica, pode-se notar que todos os decrementadores de N bits e o decrementador de M bit são executados simultaneamente e como $M \geq N$, apenas o atraso do decrementador de M bits deve ser levado em consideração. Além disso, o processo de seleção ou multiplexação deve ser realizado seqüencialmente. Como apenas uma porta lógica é necessária para o cálculo do bit de transporte em cada bloco, tem-se um atraso de $(num\_bloc - 1)$, visto que no último bloco não há a necessidade do cálculo do bit de transporte. No entanto, no último bloco, o vai-um do bloco anterior deve passar pelo circuito de seleção. O circuito de seleção leva três níveis de lógica para a definição do resultado da soma, como pode ser observado na figura A.16. O caminho crítico do decrementador com transporte selecionado foi traçado na figura A.18 e de acordo com esta figura a fórmula completa para a quantidade de níveis de lógica é:

$$NL = NL(\text{decrementador de } M \text{ bits}) + (num\_bloc) + 2$$

Desde que $(num\_bloc) \geq 1$.

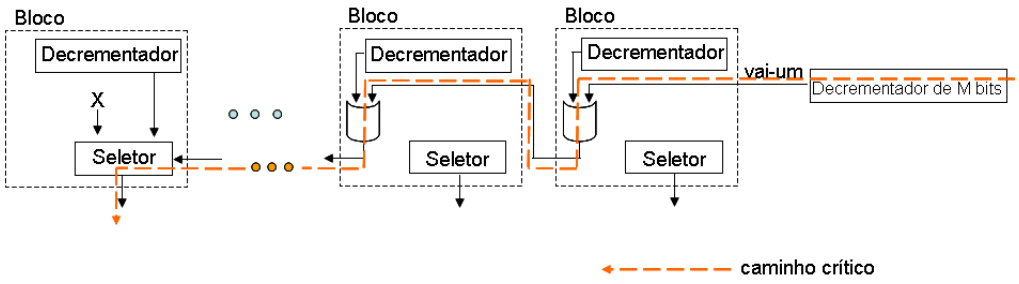


Figura A.18: Caminho crítico para o decrementador com transporte selecionado

A.2.6 Quantidades de Portas e de Níveis de Lógica do Roteador

Utilizando os mesmos procedimentos realizados acima, podem-se inferir as quantidades de portas lógicas(QP) e de níveis de lógica (NL) utilizados para implementar o roteador. O roteador possui uma série de registradores, *flip-flops*, multiplexadores, decrementadores e decodificadores, além da lógica de roteamento.

As variáveis são o tamanho do pacote (TAM), Δx, Δy e QP/NL do decrementador. O tamanho do pacote refere-se à quantidade de bits que trafegam paralelamente de um roteador a outro. A variável Δx indica a quantidade de bits utilizada para o endereçamento de roteadores no eixo x do plano carteziano. A variável Δy indica a quantidade de bits utilizada para o endereçamento de roteadores no eixo y.

Os decrementadores utilizados no processo de obtenção do endereço de destino de um

pacote podem ser implementados de diferentes formas: transporte propagado, transporte antecipado, transporte selecionado entre outras possibilidades. Por isso, no desenvolvimento das fórmulas para o roteador as quantidades de portas lógicas e de níveis de lógica do decrementador são colocados como variáveis. Portanto, dependendo do tipo de decrementador utilizado no projeto, o número de portas lógicas e a quantidade de níveis de lógica do roteador podem variar, o que modificaria o seu desempenho geral. As expressões a seguir definem a quantidade de portas e o número de níveis de lógica do roteador:

$$\begin{aligned} QP_{rot} &= 57 * TAM - 10 * \Delta x - 9 * \Delta y + 354 + 3 * QP(\text{decrementador} \quad X) + \\ &\quad +5 * QP(\text{decrementador} \quad Y) \\ NL_{rot} &= max\{\lceil log_2(\Delta max)\rceil\} + 12, \quad NL(\text{decrementador} \quad Y) + 3, \\ &\quad NL(\text{decrementador} \quad X) + 2\} \end{aligned}$$

Onde $\Delta max = max\{\Delta x, \Delta y\}$.

A.2.7 Fórmula de Desempenho

Anteriormente foram descritas as equações para a obtenção das quantidades de portas lógicas e de níveis de lógica de circuitos decrementadores implementados com arquiteturas diferentes. Também foram inferidas a quantidade de portas lógicas e a quantidade de níveis de lógica do roteador. Devido ao processo de obtenção do endereço de destino, o sistema de roteamento possui algumas estruturas que realizam o decremento de uma posição. Portanto, no desenvolvimento das fórmulas para o roteador, as quantidades de portas lógicas e de níveis de lógica dos decrementadores são colocadas como variáveis. Assim, dependendo do tipo de decrementador utilizado, o número de portas lógicas e a quantidade de níveis de lógica do roteador podem variar, o que modificaria o seu desempenho. A questão que surge diante deste panorama é: "Qual é a melhor arquitetura para o decrementador que deve ser utilizada no sistema de roteamento?"

A seguir especifica-se a fórmula de desempenho capaz de analisar um determinado sistema diante de estruturas digitais que realizam a mesma tarefa porém com arquiteturas diferentes. Desta forma, pode-se automatizar o processo que identificará a estrutura digital mais adequada para ser implementada no sistema proposto.

O desempenho em relação à quantidade de portas lógicas (QP) pode ser assim esquematizado:

$$DESEMP_{qp} = \frac{QP_{min}}{QP}$$

Onde QP_{min} indica a menor quantidade de portas lógicas atingida dentre todos os circuitos digitais envolvidos na comparação e com parâmetros específicos (quantidade de bits do circuito, tamanho dos blocos internos, entre outros) e QP indica a quantidade de portas lógicas do circuito digital que está sendo analisado.

Com isso, quando QP for mínimo, o desempenho em relação à quantidade de portas lógicas atinge o seu máximo:

$$DESEMP_{qp} = 1$$

O desempenho em relação à quantidade de níveis de lógica (NL) pode ser assim expresso:

$$DESEMP_{nl} = \frac{NL_{min}}{NL}$$

Onde NL_{min} indica a menor quantidade de níveis de lógica atingida dentre todos os circuitos digitais envolvidos na comparação e com parâmetros específicos e NL indica a quantidade de níveis de lógica do circuito digital que está sendo analisado.

Quando NL for mínimo, o desempenho em relação à quantidade de níveis de lógica atinge o seu máximo:

$$DESEMP_{nl} = 1$$

A fórmula de desempenho envolvendo tanto a quantidade de portas lógicas (QP) quanto a quantidade de níveis de lógica (NL) pode ser assim esquematizada:

$$DESEMP = P_{qp} * \left(\frac{QP_{min}}{QP}\right) + P_{nl} * \left(\frac{NL_{min}}{NL}\right)$$

Onde P_{qp} indica o peso dado ao desempenho em relação à quantidade de portas lógicas (QP) e P_{nl} indica o peso dado ao desempenho em relação à quantidade de níveis de lógica (NL). Os valores atribuídos aos pesos P_{qp} e P_{nl} devem satisfazer a seguinte relação:

$$P_{qp} + P_{nl} = 1$$

Esses pesos permitem um ajuste fino na análise matemática do sistema, atribuindo maior importância para determinados parâmetros do sistema (quantidade de portas lógicas ou quantidade de níveis de lógica). A escolha de determinados valores para os pesos, pode permitir, por exemplo, que durante o processo de análise haja uma maior preocupação com a quantidade de níveis de lógica do sistema correspondente do que com a quantidade de portas lógicas resultante, ou vice-versa.

Vale a pena ressaltar que a fórmula de desempenho aqui estabelecida leva em consideração dois parâmetros, quais sejam: o número de portas lógicas e a quantidade de níveis

de lógica dos circuitos. Contudo, a fórmula pode ser estendida para que o sistema seja analisado com parâmetros adicionais, como consumo de energia por exemplo. A inclusão do parâmetro de consumo de energia na fórmula geral de desempenho deve ocorrer de forma análoga aos parâmetros especificados anteriormente.

A.3 Resultados

As fórmulas de desempenho mostradas na seção A.2.7 e as fórmulas que indicam as quantidades de portas lógicas e de níveis de lógica dos decrementadores e do roteador foram descritas no MATLAB (HANSELMAN; LITTLEFIELD, 1997) por meio de 17 funções com o intuito de realizar uma análise comparativa da inserção desses decrementadores na arquitetura do roteador. A função principal realiza o cálculo do desempenho do roteador com cada uma das estruturas aritméticas apresentadas neste artigo.

A função inicia o cálculo do desempenho com decrementadores de 4 bits, depois o tamanho da entrada é incrementado e o cálculo de desempenho é refeito. O processo só é finalizado quando o limite previamente estabelecido for atingido. Além do limite referente ao tamanho máximo (quantidade de bits) do decrementador sob o qual é realizada a análise de desempenho, a função principal recebe como parâmetros, a quantidade de bits que representa a carga útil do pacote, o peso dado ao desempenho em relação à quantidade de portas lógicas, o peso dado ao desempenho em relação à quantidade de níveis de lógica e a quantidade de bits de um bloco para decrementadores que foram subdivididos em blocos, como é o caso do decrementador com transporte selecionado apresentado na seção A.2.5. O limite referente ao tamanho máximo dos decrementadores deve ser maior ou igual a duas vezes o tamanho do bloco.

Uma vez executada, a função gera dois gráficos referentes à comparação dos decrementadores. No primeiro gráfico, denominado Panorama 1, a função identifica o desempenho dos decrementadores sem levar em consideração a arquitetura do roteador na qual eles foram incluídos. Na figura A.19 apresenta-se o gráfico do Panorama 1, considerando um peso de 70% em relação à quantidade de níveis de lógica, 30% em relação à quantidade de portas lógicas e um tamanho de bloco contendo 4 bits.

Foram analisados seis tipos de decrementadores com técnicas diferentes para a obtenção dos bits de transporte, quais sejam: transporte em cascata, transporte antecipado, uma modificação do transporte antecipado, transporte em cascata nos blocos de decremento e transporte selecionado entre os blocos (transporte cascata–selecionado),

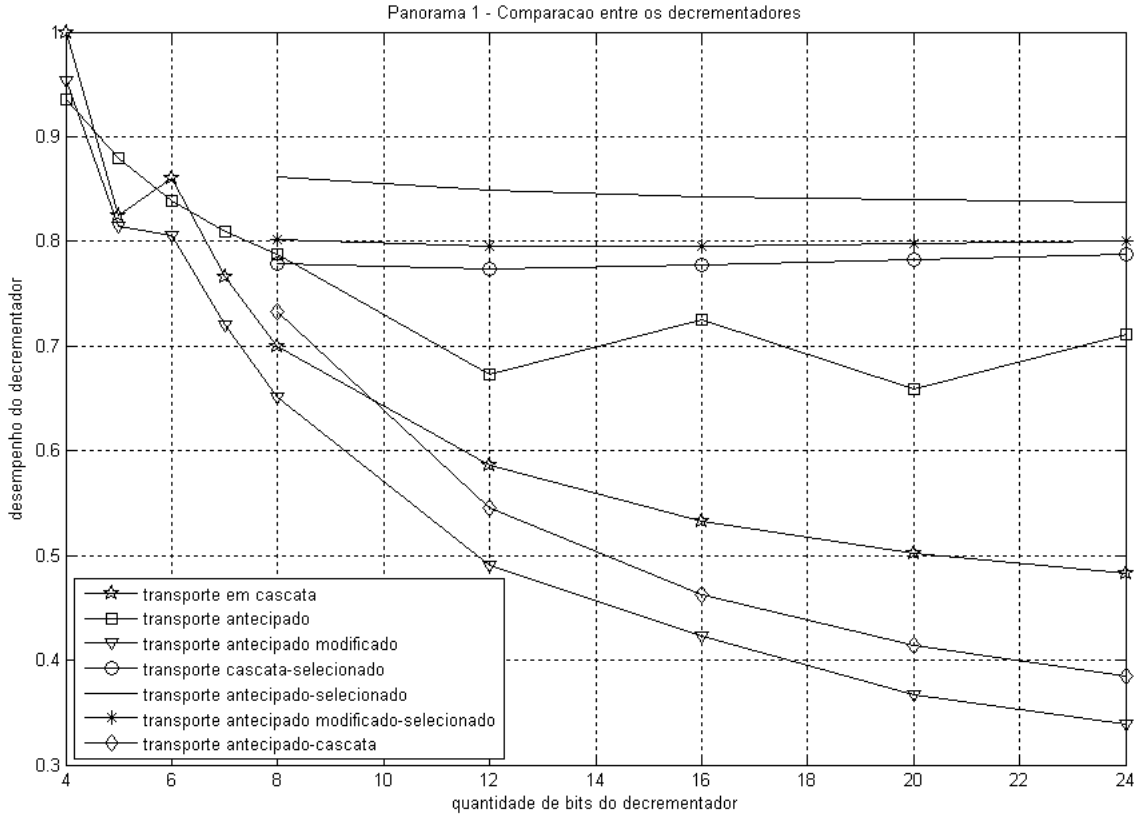


Figura A.19: Panorama 1 – Comparação de desempenho entre os decrementadores

transporte antecipado nos blocos e transporte selecionado entre os blocos (transporte antecipado–selecionado), transporte antecipado modificado nos blocos e transporte selecionado entre os blocos (transporte antecipado modificado–selecionado) e, por fim, um decrementador com transporte antecipado nos blocos e transporte em cascata entre os blocos (transporte antecipado–cascata). Este último decrementador possui um atraso maior de resposta do que o decrementador com transporte antecipado, porém possui uma menor quantidade de portas lógicas.

Pode-se notar que o decrementador com transporte antecipado–selecionado obteve o melhor desempenho entre 8 e 24 bits permanecendo na faixa de 0.8 e 0.9 de desempenho. Vale ressaltar que este resultado leva em consideração um peso de 70% em relação à quantidade de níveis de lógica e apenas 30% em relação à quantidade de portas lógicas. Na faixa logo abaixo, entre 0.7 e 0.8 de desempenho, o segundo melhor desempenho foi do decrementador com transporte antecipado modificado–selecionado. O decrementador com transporte cascata–selecionado obteve a terceira melhor relação de desempenho. O pior desempenho foi do decrementador com transporte antecipado modificado atingindo um desempenho de aproximadamente 0.2 para 24 bits.

O segundo gráfico, denominado Panorama 2, computa o desempenho da arquitetura do roteador projetada com cada um dos decrementadores. Na figura A.20 apresenta-se o gráfico do Panorama 2, considerando um peso de 70% em relação à quantidade de níveis de lógica, 30% em relação à quantidade de portas lógicas, uma carga útil composta por 20 bits e um tamanho de bloco contendo 4 bits.

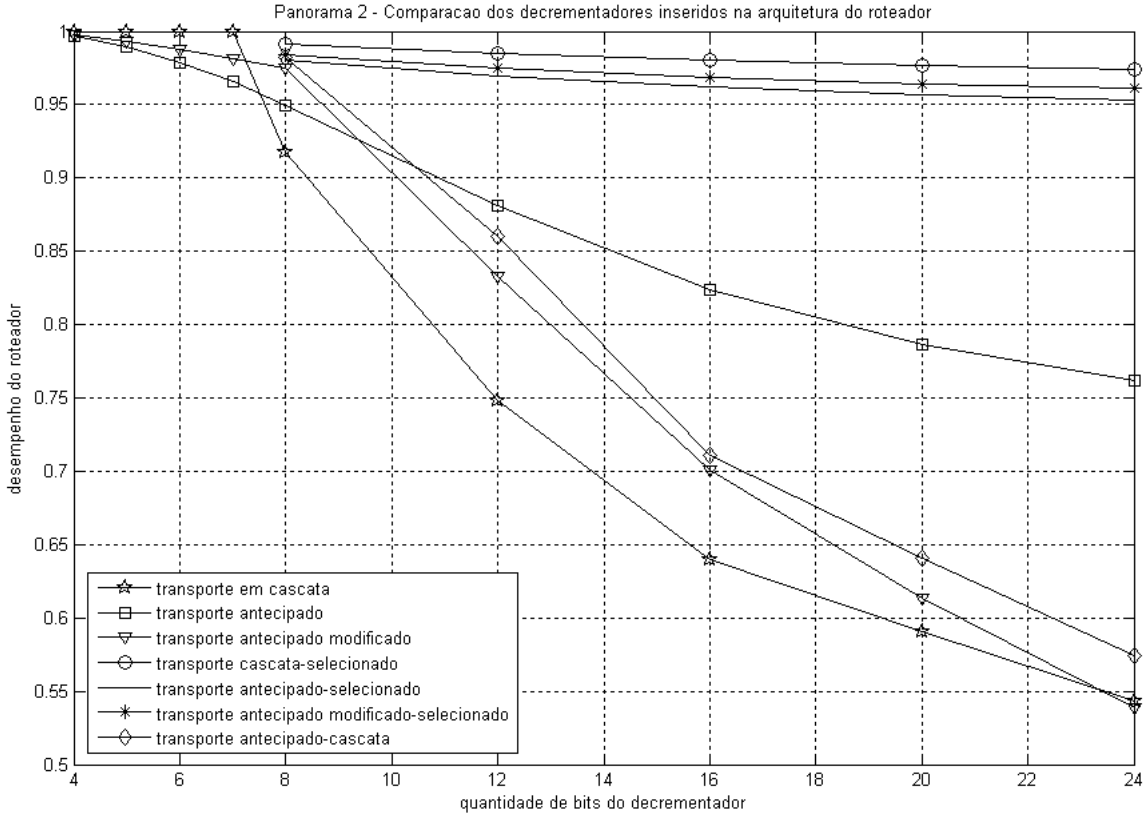


Figura A.20: Panorama 2 – Desempenho do roteador com os decrementadores inseridos na arquitetura

Pode-se notar que o decrementador com transporte em cascata possui a melhor relação de desempenho entre 4 e 7 bits, obtendo o valor máximo, $DESEMP = 1$. A partir de 8 bits, três decrementadores permaneceram na faixa de desempenho entre 0.95 e 1.00. Todos os três possuem como base a técnica de transporte selecionado entre os blocos, se diferenciando com relação à implementação interna do bloco. Assim, o decrementador com transporte cascata–selecionado obteve o melhor desempenho. Seguido de perto, o decrementador com transporte antecipado modificado–selecionado apresentou o segundo melhor desempenho. O terceiro melhor desempenho é do decrementador com transporte antecipado–selecionado. De modo geral, o pior desempenho entre 8 e 24 bits ficou com o decrementador com transporte em cascata, atingindo um desempenho de aproximadamente 0.55 para 24 bits.

A.4 Discussão

Como pode ser observado na figura A.19, o decrementador com transporte antecipado–selecionado apresentou um desempenho melhor do que o decrementador com transporte cascata–selecionado. Porém, como mostrado no Panorama 2, figura A.20, o roteador projetado com o decrementador com transporte cascata–selecionado obteve a melhor relação de desempenho, mesmo este decrementador tendo um desempenho individual inferior ao decrementador com transporte antecipado–selecionado.

Isso acontece porque na arquitetura do roteador, além de caminhos de dados existem também alguns caminhos relacionados ao controle da arquitetura. Com isso, a escolha de decrementadores mais rápidos pode melhorar o tempo de resposta de determinados caminhos de dados, mas contudo, pode não interferir no tempo de resposta dos caminhos de controle e de alguns caminhos de dados. Desta forma, apesar do decrementador com transporte antecipado–selecionado ser individualmente mais rápido, o desempenho em relação à quantidade de níveis de lógica do roteador projetado com qualquer um dos dois decrementadores será o *mesmo* pois o caminho crítico do roteador se mantém inalterado para os dois decrementadores. Contudo, o decrementador mais rápido que utiliza o transporte antecipado–selecionado gasta uma quantidade de portas lógicas um pouco maior do que o decrementador com transporte cascata–selecionado, e por este motivo a arquitetura do roteador utilizando o decrementador com transporte cascata–selecionado apresentou um desempenho melhor.

Também pode-se notar, na figura A.20, que de modo geral os decrementadores que utilizam transporte selecionado entre os blocos apresentaram os melhores resultados. Convencionalmente, o somador com transporte selecionado tem um atraso relativamente pequeno, contudo, possui um número elevado de portas lógicas visto que, em cada bloco, dois somadores são necessários para produzir o resultado. Porém, o somador com transporte selecionado foi adaptado para o projeto do roteador (realizando apenas o decremento de um) como mostrado na figura A.17, e esta adaptação permitiu a inclusão de apenas um somador (decrementador) em cada bloco, reduzindo significativamente o número de portas lógicas. Com isso, estes decrementadores além de serem rápidos passaram a ter uma pequena quantidade de portas lógicas, tornando-se extremamente atrativos.

A escolha de um peso de 70% em relação à quantidade de níveis de lógica e de 30% em relação à quantidade de portas lógicas se deve aos motivos explicados na seqüência.

A velocidade de operação dos roteadores se torna muito importante visto que paco-

tes gerados pela arquitetura poderão atravessar vários roteadores até chegarem ao seus destinos finais. Se a velocidade de operação for muito baixa, os pacotes levarão muito tempo para ser entregues às transições do sistema e, por conseqüência, o tempo de resposta da arquitetura poderá aumentar muito. Assim, para se conseguir uma arquitetura eficiente, os roteadores que compõem o sistema de comunicação devem ser os mais rápidos possíveis. Porém, a quantidade de lógica utilizada para o projeto de cada roteador influencia no tamanho do *chip* a ser construído, o que acarretaria um custo maior na fabricação dessa arquitetura. Além disso, roteadores com uma quantidade muito grande de lógica tomaria muito espaço no *chip*, o que diminuiria a quantidade de circuitos que poderiam ser colocados na implementação física dessa arquitetura. Com uma menor quantidade de circuitos no *chip*, o algoritmo de mapeamento seria capaz de alocar uma quantidade menor de lugares e transições das Redes de Petri.

Contudo, foram realizadas análises utilizando outros valores para os pesos em relação às quantidades de portas lógicas e de níveis de lógica. Uma função em MATLAB foi especialmente desenvolvida para retornar o desempenho do roteador com cada um dos decrementadores variando os pesos em relação à porta lógica de 0% até 100%, com uma taxa de incremento de 10%. Na figura A.21 apresenta-se o gráfico gerado por esta função, considerando uma carga útil composta por 20 bits, um tamanho de bloco contendo 4 bits e decrementadores de 12 bits de entrada.

Pode-se notar que para a maioria dos pesos em relação à quantidade de portas lógicas, de 0 até 0.8, ou de 0% até 80%, os três decrementadores que utilizam transporte selecionado entre os blocos apresentaram os melhores resultados, quais sejam, o decrementador com transporte cascata–selecionado, o decrementador com transporte antecipado modificado–selecionado e o decrementador com transporte antecipado–selecionado, em ordem de desempenho. A partir de um peso em relação à quantidade de portas lógicas de aproximadamente 87%, o decrementador com transporte em cascata ultrapassa o decrementador com transporte cascata–selecionado, se tornando o mais adequado. O decrementador com transporte em cascata possui uma quantidade reduzida de portas lógicas. Portanto, nesses casos, onde o nível de lógica não é quase levado em consideração, o decrementador com transporte em cascata leva maior vantagem apesar de apresentar o pior tempo de resposta.

A abordagem aqui apresentada foi desenvolvida para realizar uma análise comparativa da arquitetura do roteador com cada um dos decrementadores em uma rede-em-*chip*. Contudo, esta abordagem também pode ser utilizada para a realização de análises em

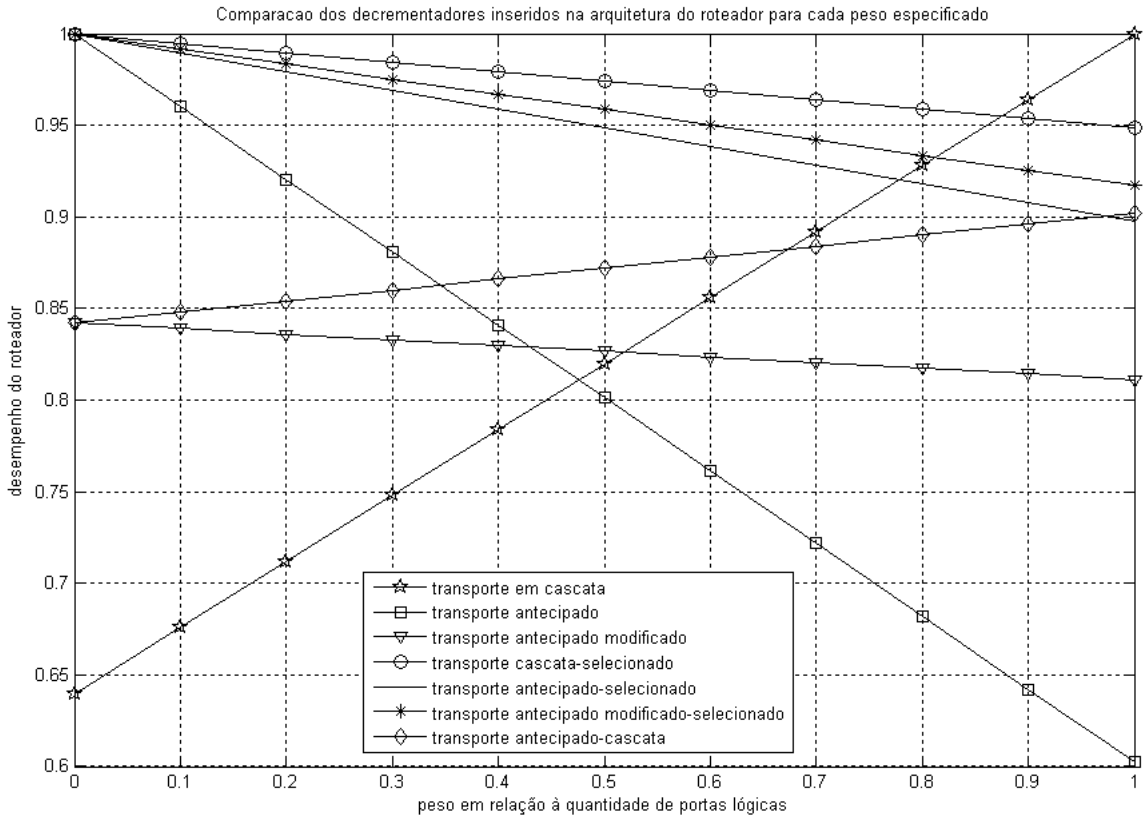


Figura A.21: Comparação dos decrementadores inseridos na arquitetura do roteador para cada peso especificado

outros tipos de projetos. Na realidade, todo projeto que possuir um sistema digital composto por decrementadores sem o bit de transporte final poderá fazer uso das fórmulas aqui desenvolvidas para definir a estrutura que melhor se adequará às condições estabelecidas pelo projetista. Desta forma, será possível automatizar o processo de identificação do circuito digital mais apropriado a ser incorporado em um determinado sistema. Assim, pode-se economizar tempo e dinheiro na realização de um projeto, bem como auxiliar os projetistas na obtenção de sistemas com melhores desempenhos.

A.5 Comentários

Foram formuladas equações matemáticas que computam as quantidades de portas lógicas e de níveis de lógica de algumas estruturas aritméticas. Uma fórmula de desempenho foi desenvolvida para realizar uma análise comparativa da arquitetura do roteador com diferentes tipos de decrementadores. O decrementador com transporte em cascata obteve a melhor relação de desempenho desde que seu tamanho esteja entre 4 e 7 bits. A partir de 8 bits, três decrementadores que utilizam a técnica de transporte selecionado ob-

tiveram desempenho semelhante, se diferenciando com relação à implementação do bloco de subtração. Dos três, o decrementador com transporte cascata–selecionado obteve o melhor desempenho.

Todo projeto que possuir um sistema digital composto por decrementadores sem o bit de transporte final poderá fazer uso das fórmulas aqui desenvolvidas para definir a estrutura que melhor se adequará às condições estabelecidas pelo projetista. Desta forma, será possível automatizar o processo de identificação do circuito digital mais apropriado a ser incorporado em um determinado sistema. Assim, pode-se economizar tempo e dinheiro na realização de um projeto, bem como auxiliar os projetistas na obtenção de sistemas com melhores desempenhos.

ANEXO B – Composição do CD-ROM Anexado a esta Tese

Este trabalho de doutorado vem acompanhado de um CD-ROM cujo conteúdo foi dividido em quatro diretórios, quais sejam:

- ROTEADOR: contém 20 arquivos VHDL que descrevem o sistema de comunicação da arquitetura proposta. O arquivo principal possui o nome `roteador.vhd` e contém todas as instâncias das entidades descritas (19 arquivos).

- BCGN: contém 21 arquivos VHDL que descrevem o bloco de configuração do gerador de números pseudo-aleatórios da arquitetura proposta. O arquivo principal possui o nome `BCGN.vhd` e contém todas as instâncias das entidades definidas (20 arquivos).

- BCERP: contém 76 arquivos VHDL que descrevem o bloco básico de configuração dos elementos de uma Rede de Petri. O arquivo principal possui o nome `BCERP.vhd` e contém todas as instâncias das entidades descritas (75 arquivos).

- ARQUITETURA_3X3: contém os arquivos VHDL que descrevem a estrutura de simulação e teste adotada (figura 9.5) e também possui alguns vetores de simulação que validam a arquitetura projetada. Este diretório é composto por quatro diretórios:

  - REDE_COM_TEMPO_LOGICO: contém os arquivos VHDL que descrevem o sistema 3x3 e um módulo de configuração que mapeia a Rede de Petri com tempo lógico da figura 9.4 neste sistema. O arquivo principal possui o nome `arq_top_3x3.vhd`. Neste diretório encontra-se o arquivo de simulação utilizado para validar a implementação(SIMULACAO_ARQ_3X3/ARQ_TOP_3X3.VWF).

– REDE_COM_TRANSICAO_CONFLITO: contém os arquivos VHDL que descrevem o sistema 3x3 e um módulo de configuração que mapeia uma Rede de Petri com transições em conflito neste sistema. O arquivo principal possui o nome `arq_top_3x3.vhd`. Neste diretório encontra-se o arquivo de simulação utilizado para validar a implementação(SIMULACAO_ARQ_3X3/ARQ_TOP_3X3.VWF).

– REDE_SEM_TEMPO_LOGICO: contém os arquivos VHDL que descrevem o sistema 3x3 e um módulo de configuração que mapeia uma Rede de Petri com temporização física neste sistema. Neste diretório encontra-se o arquivo de simulação utilizado para validar a implementação desta Rede de Petri na arquitetura(SIMULACAO_ARQ_3X3/ARQ_TOP_3X3.VWF).

– REDE_SEM_TEMPORIZACAO: contém os arquivos VHDL que descrevem o sistema 3x3 e um módulo de configuração que mapeia uma Rede de Petri sem temporização alguma neste sistema. Neste diretório encontra-se o arquivo de simulação utilizado para validar a implementação desta Rede de Petri na arquitetura(SIMULACAO_ARQ_3X3/ARQ_TOP_3X3.VWF).

O *software* Quartus II Web Edition contém a maioria das características da versão comercial, permitindo a programação de modelos em VHDL e AHDL (linguagem de descrição de *hardware* desenvolvida pela própria Altera). A versão Web Edition também possui o ambiente de compilação e análise de tempo, tendo suporte a algumas famílias de FPGAs, como a EPF10K70, a STRATIX e a CYCLONE II, incluindo os mais variados dispositivos de cada família. Normalmente, a instalação desse *software* é extremamente simples, bastando executar o arquivo de instalação de extensão `.exe` para que o programa seja devidamente instalado. Uma vez instalado, o *software* exigirá um código de autorização, o qual pode ser obtido pela internet no *site* da Altera cujo URL é http://www.altera.com.