Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

MODELAGEM E VALIDAÇÃO DE REDES INTRACHIP ATRAVÉS DE SÍNTESE COMPORTAMENTAL

Rosana Perazzolo Disconzi

Dissertação apresentada como requisito parcial à obtenção do grau de mestre em Ciência da Computação

Orientador: Prof. Dr. Ney Laert Vilar Calazans

Porto Alegre
2008

Substitua esta folha pela ficha de homologacao

"Pedras no caminho? Guardo todas, um dia vou construir um castelo..." *(Fernando Pessoa)*

Agradecimentos

Agradeço a todos que me ajudaram a aprender e a crescer com cada uma das dificuldades. Agradeço a minha família e ao Maicon pelo incentivo. Agradeço ao Ney pela paciência. Agradeço aos amigos do GAPH pela acolhida quando cheguei na PUC. Agradeço ao SERPRO pela oportunidade e pelas horas liberadas para que eu pudesse dar andamento a este trabalho. Agradeço aos amigos e colegas do SERPRO e prometo comparecer em todas as festas de agora em diante.

Resumo

A crescente demanda pela redução do *time-to-market* para SoCs (*System-on-chip*) leva a mudanças essenciais na maneira como esses sistemas são concebidos. Um dos componentes críticos em qualquer SoC é a arquitetura interna de comunicação entre módulos do sistema. Tradicionalmente, estas são implementadas como arquiteturas de comunicação baseadas em barramentos. Contudo, a medida que a complexidade de SoCs cresce com a evolução tecnológica, barramentos apresentam crescentes limitações com relação a escalabilidade, consumo de potência e paralelismo. Devido a estas limitações, estruturas do tipo redes intrachip ou NoCs (*Networks-on-Chip*) têm ganho crescente destaque como forma de permitir superar as limitações derivadas do uso de barramentos em SoCs. Tais redes ampliam o espaço de soluções de projeto de estruturas de comunicação intrachip e trazem como vantagem largura de banda escalável de forma mais sistemática, o uso de conexões ponto a ponto curtas com menor dissipação de potência e a capacidade de facilmente definir o grau de paralelismo da comunicação. O processo de projeto de NoCs tem sido alvo de esforços da indústria e do meio acadêmico e este trabalho contribui com a avaliação de um processo de projeto que está retomando força com ferramentas comerciais, a síntese comportamental. O processo de projeto avaliado aqui, especificamente aquele ao qual dá suporte o ambiente *Cynthesizer* da FORTE Design Systems, não foi concebido para dar suporte ao projeto de arquiteturas de comunicação intrachip e não possui associados arcabouços de projeto para tal tarefa. No entanto, a facilidade de modelagem dessas estruturas mostrou-se atraente para realizar tal avaliação. Para tanto, foram escolhidos estudos de caso de NoCs com topologia toro 2D bidirecionais, pouco exploradas na literatura. Como contribuições deste trabalho cita-se a avaliação da síntese comportamental para o projeto de NoCs e a adaptação de algoritmos livres de (*deadlocks*) da literatura. Tais algoritmos foram propostos para redes de topologia malha e para toro unidirecional, e neste trabalho realizaram-se adaptações para uso destes em redes toro bidirecionais. Como resultado da avaliação, conclui-se que o estado da arte da síntese comportamental ainda precisa avançar e incluir processos para a geração e otimização de arquiteturas de comunicação intrachip. Os resultados obtidos são significativamente inferiores àqueles derivados de codificação direta no estilo RTL em termos de área e velocidade, mesmo depois de aplicado esforços significativos de otimização de código e exploração do espaço de projeto. Este trabalho demonstrou, contudo um fator positivo da síntese comportamental, qual seja a facilidade de modelagem e avaliação de algoritmos de roteamento.

Palavras-chave: redes intrachip, Síntese Comportamental, NoCs, SoCs, FPGA, fluxos de projeto.

Abstract

The growing demand for system-on-Chip (SoC) time-to-market reduction leads to relevant changes in the way such systems are designed. One of the critical components in any SoC is the intra-chip architecture employed to enable communication among the SoC processing elements. Traditionally, intra-chip communication architectures are implemented based on multipoint structures such as shared busses. However, as SoC complexity grows following the silicon technology evolution, busses tend to display growing limitations related to figures like scalability, power consumption and degree of parallelism. Due to these limitations, structures like networks-on-chip (NoCs) have gained attention as ways to allow overcoming the limitations due to the use of shared busses. NoCs enlarge the design search space of intra-chip communication architectures and bring forward a set of advantages when compared to shared busses, including more systematic methods to scale communication bandwidth, reduction of global wiring, point-to-point multiple wires leading to power reduced interconnect and the capacity to easily define the degree of parallelism in communication. The NoC design process has been a target for academic and industrial efforts. This work contributes with an evaluation of a design process that has found growing acceptance, the behavioral synthesis. This is corroborated by the current availability of several commercial CAD tools that support it. The specific design process employed here is the one supported by the Cynthesizer environment of FORTE Design Systems. This environment was neither conceived with specific facilities for the design of intra-chip communication architectures nor has any associated design framework for this task. However, the easiness with which Cynthesizer allows the modeling of such structures rendered makes it interesting to perform the cited evaluation work. To do so, case studies of 2D torus topology NoCs were selected. This topology is not quite explored in the literature as a target of NoCs. The main contributions of this work are the evaluation results of using behavioral synthesis methods to produce intra-chip communication structures and the adaptation of deadlock-free algorithms for the chosen topology. These algorithms were originally proposed for networks with 2D mesh and unidirectional 2D torus topologies which have been adapted for a 2D bidirectional torus topology. As a result of the evaluation it is possible to conclude that the current state of the art in behavioral synthesis needs to evolve to include processes for the generation and optimization of intra-chip communication architectures. The obtained results are significantly inferior in terms of area overhead and speed when compared to implementations starting from handwritten RTL code, even after significant optimizations and design space exploration. This work demonstrated however a positive factor of using behavioral synthesis, which is the easiness to model and evaluating routing algorithms.

Keywords: Networks-on-Chip, NoCs, SoCs, FPGA.

Lista de Figuras

Figura 1 Fluxo de síntese RTL convencional, baseado em [24]. 28
Figura 2 Fluxo de síntese comportamental, baseado em [24]. 30
Figura 3 Exemplo de um MAG, retirado de [24]: (b) com pesos; (a) derivado do SDFG. 32
Figura 4 Derivação da especificação comportamental até a arquitetura RT, retirada de [24]: (a) exemplo de especificação comportamental; (b) um possível DFG da descrição; (c) uma possibilidade de arquitetura RT. . . 32
Figura 5 Escopos de síntese comportamental e RTL, retirado de [21]. 33
Figura 6 Diagramas de projeto do sistema MIMOLA, baseado em [29]. 36
Figura 7 Particionamento arquitetural, retirado de [23]. 38
Figura 8 Clusterização dos objetos A, B, C, D e E, retirado de [23]. 38
Figura 9 Dois estágios da clusterização multiestágio. No primeiro estágio, a árvore completa é construída e uma linha é escolhida para produzir os *clusters* L, M, N, O e P. No segundo estágio, os *clusters* do estágio anterior são utilizados como base para uma nova iteração. 39
Figura 10 Método de síntese de alto nível, conforme proposto em [3]. 40
Figura 11 Clusterização comportamental, retirado de [35]. 42
Figura 12 Inserção da ferramenta Cynthesizer em um fluxo de projeto. 46
Figura 13 Integração dos componentes do Cynthesizer. Módulos marcados com M são descritos manualmente pelo projetista. 47
Figura 14 Processo de verificação disponibilizado pelo Cynthesizer. O mesmo *testbench* é utilizado para simular a especificação inicial e todas as especificações derivadas do processo de síntese, tanto para ASICs quanto para FPGAs [39]. 49
Figura 15 Exemplo de interface de comunicação entre duas *threads*: (a) produzindo dados; (b) consumindo dados. 50
Figura 16 Organização dos arquivos de projeto no ambiente Cynthesizer. 50
Figura 17 Organização de um módulo de projeto em uma descrição comportamental. 51
Figura 18 Exemplo de descrição comportamental: (a) arquivo cubo.h, que contém a declaração do módulo, seus sinais, *threads* e seu construtor. (b) arquivo cubo.cc, com a implementação do comportamento do módulo. . 53
Figura 19 Arquivo project.tcl para o projeto exemplo dessa Seção. 54
Figura 20 Arquivo cubo_diretivas.h, que contém a definição das diretivas de síntese do projeto para as configurações LATENCY e BASIC, que foram especificadas no arquivo project.tcl. 55
Figura 21 Relatório preliminar com as estimativas iniciais baseadas na biblioteca de tecnologia caracterizada para o projeto (*vtvtlib25*). As áreas são apresentadas em microns quadrados (μm^2). 56

Figura 22 Visualização da simulação do código RTL SystemC da configuração BASIC. A cada 5 ciclos uma operação é realizada. 57
Figura 23 Visualização da simulação do código RTL SystemC da configuração LATENCY. A cada 4 ciclos uma operação é realizada. 57
Figura 24 Exemplo de implementação RTL em SystemC: (a) arquivo controle.h da especificação do controle do roteador da rede Hermes; (b) arquivo controle.cc, que implementa o módulo de controle do roteador da rede Hermes. 61
Figura 25 Exemplo de implementação comportamental em SystemC: (a) arquivo router.h que descreve o roteador da rede Hermes. O módulo de controle é uma das *threads* do roteador; (b) implementação da *thread* de controle de roteamento. 62
Figura 26 Exemplo de instanciação e acesso de componentes de memória em descrições comportamentais aceitas pelo Cynthesizer. 64
Figura 27 Exemplo de interface de comunicação entre duas *threads* : (a) produzindo dados; (b) consumindo dados. 65
Figura 28 Detalhes da simulação do módulo *buffer*, que controla a porta de entrada de uma das interfaces dos roteadores implementados nesse trabalho. . . 65
Figura 29 Sincronização com *handshake* leva em média 3 ciclos para concluir a troca de dados. 66
Figura 30 Sincronização com crédito leva dois ciclos para concluir a troca de dados. 66
Figura 31 Fluxo de projeto NetChip [5]. 72
Figura 32 Exemplos de grafos de quadrante de comunicação, obtidos no mapeamento da topologia com caminhos mínimos entre os nodos e e h. 73
Figura 33 Organização do modelo conceitual OCCN. 75
Figura 34 Exemplos de topologias diretas: (a) malha 3X3X3 (b) toro (2-cubo 3-ário) (c) hipercubo (4-cubo 2-ário). 80
Figura 35 Exemplos de redes com topologia indireta: (a) crossbar; (b) butterfly. . . 80
Figura 36 Processos disputando um recurso, tendo suas requisições avaliadas por um árbitro. 82
Figura 37 Em uma rede malha 9x9, o nodo 88 tem um pacote destinado ao nodo 77. Os canais que podem ser selecionados pelo algoritmo estão ocupados e o pacote é encaminhado adaptativamente para outra direção para não permanecer bloqueado. Assim, o pacote pode se afastar consideravelmente de seu destino, gerando risco de *livelock*. 83
Figura 38 Situação de *deadlock* ou impasse entre os processos A e B. 83
Figura 39 Representação de uma situação de impasse em uma rede anel unidirecional com quatro nodos usando filas de entrada. Cada nodo tem um pacote destinado ao nodo oposto. As mensagens não conseguem avançar porque os *buffers* dos nodos estão cheios e existe uma dependência cíclica que impede de esvaziá-los. 84
Figura 40 Grafo de interconexão e grafo de dependência de canais para uma rede anel de quatro nodos. 86
Figura 41 Grafo de interconexão e grafo de dependência entre canais para uma estrutura anel de quatro nodos com a adição de canais virtuais e quebra do ciclo de dependência entre os canais. 86
Figura 42 Rede toro 4x4 unidirecional com canais numerados de acordo com o método de Duato, Ni e Yalamanchili. 88

Figura 43 Algoritmo de roteamento para a rede toro 2D unidirecional com canais virtuais. 89
Figura 44 Exemplo da execução do algoritmo, onde a origem 22 envia uma pacote para o destino 11. 89
Figura 45 Curvas em uma rede malha de duas dimensões: (a) ciclos simples formados pelas direções permitidas para uma rede malha bidimensional; (b) curvas proibidas (pontilhadas) para evitar *deadlock*. 90
Figura 46 Situação de *deadlock* gerada na combinação de curvas: (a) proibir uma curva torna as curvas à esquerda restantes equivalentes a proibir uma curva à direita; (b) proibir uma curva torna as curvas à direita restantes equivalentes a proibir uma curva à esquerda; (c) os dois ciclos combinados geram uma situação de impasse. 91
Figura 47 Curvas permitidas (traço contínuo) e proibidas (traço pontilhado) nos algoritmos *turn model* de Glass e Ni: (a) west-first; (b) north-last; (c) negative-first. 91
Figura 48 Canais virtuais na rede anel bidirecional. 92
Figura 49 Rede toro 4x4, salientando os conjuntos de roteadores nas extremidades da rede. 96
Figura 50 Pseudo-código do algoritmo *west-first não-mínimo* elaborado para a rede toro 2D bidirecional. 97
Figura 51 Exemplos de operação do algoritmo de roteamento *west-first* para redes toro 2D: (a) pacote injetado no nodo 13 com destino ao nodo 30. Nesse caso, é vantajoso e possível utilizar o canal de *wraparound* como primeiro *hop*; (b) pacote injetado no nodo 23 com destino ao nodo 00. Não é possível utilizar o canal de *wraparound* devido à restrição do *west-first*. 98
Figura 52 Arquitetura do roteador para topologia toro 2D. 99
Figura 53 Interface de comunicação entre roteadores e estruturas responsáveis, sendo R1 o roteador transmissor e R2 o roteador receptor. 99
Figura 54 Interfaces dos módulos árbitro e controle do roteador para topologia toro 2D. 100
Figura 55 Esquema para utilização dos módulos e arquivos de entrada gerados pelo ambiente ATLAS na simulação em nível comportamental (SystemC) e RTL (SystemC e Verilog) e os arquivos de saída produzidos na simulação que serão utilizados para a extração dos relatórios no Atlas. 101
Figura 56 Função de roteamento. 103
Figura 57 Consulta e atualização das tabelas de roteamento. 104
Figura 58 Simulação do roteamento sem as diretivas de síntese. A resposta da requisição de roteamento leva de 5 a 16 ciclos. 105
Figura 59 Simulação do roteamento aplicando-se todas as diretivas marcadas no código. A resposta da requisição de roteamento sempre 4 ciclos. 105
Figura 60 Simulação modificando a diretiva 1 marcada no código para executar a operação em até 3 ciclos. A resposta da requisição de roteamento leva de 4 a 6 ciclos. 105
Figura 61 Simulação suprimindo a diretiva 1 marcada no código. A resposta da requisição de roteamento leva de 5 a 10 ciclos. 106
Figura 62 Simulação suprimindo as diretivas 2 e 3 assinaladas no código. A resposta da requisição de roteamento leva de 3 a 6 ciclos. 106

Figura 63 Simulação suprimindo a diretiva 2 assinalada no código. A resposta da requisição de roteamento leva de 4 a 6 ciclos. 106
Figura 64 Relatório da síntese dos módulos *buffer* e *buffer_c*, que são reponsáveis pelo armazenamento e controle das entradas no roteador, ambos com capacidade para 8 flits de 16 bits. 108
Figura 65 Mapeamento dos elementos extraídos da síntese comportamental do *buffer* com os elementos contidos na biblioteca de tecnologia. 108
Figura 66 Mapeamento dos elementos extraídos da síntese comportamental do buffer_c com os elementos contidos na biblioteca de tecnologia. 109
Figura 67 Relatório da síntese para ASIC e para FPGA para a rede 2x2 com buffers de 8 flits de 16 bits. 110
Figura 68 Rotas de alguns dos pacotes do exemplo de simulação. Cabe salientar que essas rotas foram obtidas na simulação com uma rede congestionada. 113
Figura 69 Toro 4x4 unidirecional com canais numerados de acordo com o método de Duato, Ni e Yalamanchili para o algoritmo complementar Sul-Oeste. 116
Figura 70 Algoritmo Sul-Oeste, complementar ao algoritmo Leste-Norte, respeitando a codificação estabelecida por Duato, Ni e Yalamanchili. 116
Figura 71 Exemplo da alocação dos canais na execução do algoritmo Sul-Oeste em uma rede toro 2D 4x4, com origem no nodo 31 e destino 12. 117
Figura 72 Exemplo de funcionamento do esquema de roteamento com os algoritmos Leste-Norte e Sul-Oeste: (a) rota do pacote que é injetado na rede pelo roteador 11 e tem como destino 32. O roteameto é Leste-Norte; (b) o pacote injetado na rede por 21 com destino 33 segue o caminho Sul-Oeste, devido ao congestionamento da porta de saída Leste do roteador 21 por outro pacote. 117
Figura 73 Rotas dos algoritmos Leste-Norte (setas escuras) e Sul-Oeste (setas claras) com origem em 21 e destino 02. 118
Figura 74 Algoritmo de roteamento implementado para o estudo de caso deste Capítulo. 119
Figura 75 Arquitetura do roteador da rede toro 2D com canais virtuais. 120
Figura 76 Interface entre roteadores, com R1 como transmissor e R2 como receptor. 121
Figura 77 Porta Leste do nodo 20, com dado a ser transmitido nos dois canais. . . 122
Figura 78 Detalhe do comportamento da porta quando a linha de dados é compartilhada pelos canais. 123

Lista de Tabelas

Tabela 1 Pontos de sincronização e modelos de entrada em diferentes níveis de abstração. 33
Tabela 2 Escopo da síntese comportamental e da síntese RTL [7]. 34
Tabela 3 Quadro comparativo de ferramentas de síntese comportamental modernas. 44
Tabela 4 Quadro comparativo de gerações de ferramentas de síntese comportamental. 44
Tabela 5 Lista dos pricipais arquivos de um projeto no ambiente Cynthesizer [39]. 49
Tabela 6 Ferramentas que podem ser integradas ao ambiente Cynthesizer [39]. . 58
Tabela 7 Resultado da utilização das diretivas COMPLETE, CONSERVATIVE e AGGRESSIVE para o desenrolamento parcial de laços. 68
Tabela 8 Quadro comparativo dos algoritmos revisados no Capítulo. 93
Tabela 9 Exploração de Espaço de projeto para diferentes sincronizações para a entrada de dados na fila de uma porta de entrada do roteador. 102
Tabela 10 Resumo do relatório apresentado pela ferramenta para os cenários aplicados a *thread* de roteamento. 104
Tabela 11 Informações relativas às áreas das NoCs geradas, extraídas do relatório disponibilizado pelo Cynthesizer. 108
Tabela 12 Ocupação do FPGA de cada rede em relação a sua dimensão, após a síntese comportamental visando o Virtex II XC2V4000(*speed grade* -5) da XILINX. 109
Tabela 13 Comparação entre resultados obtidos com a síntese comportamental e com um processo de síntese RTL tradicional. 110
Tabela 14 Comparação das latências médias com as obtidas no trabalho de Scherer [36]. 111
Tabela 15 As 10 menores latências obtidas na simulação do cenário com a rede 7x6 com *buffer* de 16 flits. 111
Tabela 16 As 10 maiores latências obtidas na simulação do cenário com a rede 7x6 com *buffers* de 16 flits. 112
Tabela 17 As 10 menores latências obtidas na simulação do cenário com a rede 3x3 com *buffer* de 8 flits. 124
Tabela 18 As 10 maiores latências obtidas na simulação do cenário com a rede 3x3 com *buffer* de 8 flits. 124
Tabela 19 Ocupação do dispositivo Virtex II XC2V4000(*speed grade* -5) da XILINX obtido através de síntese comportamental. 125

Lista de Siglas

TL	Transaction Level	27
RTL	Register Transfer Level	27
DFG	Data Flow Graph	29
SDFG	Scheduled Data Flow Graph	31
ASAP	As-Soon-As-Possible	31
ALAP	As-Late-As-Possible	31
MAG	Module Allocation Graph	31
E/S	Entrada/Saída	33
CFG	Control-Flow Graph	33
CDFG	Control Data Flow Graph	33
FSMD	Finite State Machine with Datapath	33
FSM	Finite State Machine	33
BDD	Binary Decision Diagram	33
VHDL	VHSIC Hardware Description Language	34
VHSIC	Very High Speed Integrated Circuit	34
BNG	Behavioral Network Graph	39
HLS	High Level Synthesis	39
DSP	Digital Signal Processing	41
QoR	Quality of Results	41
TLM	Transaction Level Modeling	43
EDIF	Eletronic Design Interchange Format	43
FPGA	Field-Programmable-Gate-Array	43
ASIC	Application Specific Integrated Circuit	43
ESL	Electronic System Level	45
RTOS	Real-Time Operating Systems	45
BDW	Behavioral Design Workbench	46
TSMC	Taiwan Semiconductor Manufacturing Company	51
VCD	Value Change Dump	57

OCCN	On-Chip Communication Network	71
CAFES	Communication Analysis for Embedded Systems	71
MCF	Multi-Commodity Flow	73
API	Application Programming Interface	74
OCCA	On-Chip Communication Architectures	74
BSP	Board Support Package	75
BIST	Built in Self Test	75
CWM	ommunication Weighted Model	76
ECWM	Extended Communication Weighted Model	76
CDM	Communication Dependence Model	76
CDCM	Communication Dependence and Computation Model	76
ACPM	Application Communication Pattern Model	76
CTM	Communication Task Model	76

Sumário

1 Introdução **23**
1.1 Motivação 24
1.2 Objetivos 25
1.3 Contribuições 25
1.4 Organização do Restante do Documento 26

2 Síntese Comportamental **27**
2.1 Comparação entre Síntese RTL e Síntese Comportamental 27
2.2 A Síntese Comportamental segundo Lee 29
2.3 Fronteiras entre a Síntese Comportamental e a Síntese RTL 32
2.4 Uma Classificação de Ferramentas de Síntese Comportamental 35
2.4.1 Primeira Geração 35
2.4.2 Segunda Geração 37
2.4.3 Terceira Geração 38
2.4.4 Ferramentas de Síntese de Alto Nível Modernas 41
2.4.5 Comparativo entre as Gerações de Síntese Comportamental 43

3 Síntese Comportamental com a Ferramenta Cynthesizer **45**
3.1 Estrutura da Ferramenta 46
3.1.1 Fases do Processo de Síntese Comportamental 47
3.2 Estrutura de um Projeto no Ambiente Cynthesizer 48
3.2.1 Exemplo de Projeto 51
3.3 Integração do Cynthesizer com Ferramentas Disponíveis no Mercado. 57

4 Codificação Comportamental **59**
4.1 FSMs e Funcionalidades Multiciclo 60
4.2 Acesso a Memória em Especificações Comportamentais 62
4.3 Interfaces de Comunicação em Descrições Comportamentais 63
4.4 Diretivas do Cynthesizer para Otimização da Qualidade dos Resultados 66
4.4.1 Protocolos para Sincronização entre Threads 66
4.4.2 Desdobramento de Laços 67
4.4.3 Redução do Tamanho dos Multiplexadores 67
4.4.4 Latência de Partes Específicas do Código 69

5 Arcabouços de Projeto Alto Nível para NoCs **71**
5.1 Netchip 71
5.2 OCCN - On-Chip Communication Network 74
5.3 CAFES - Communication Analysis for Embedded Systems 76

5.4 Abordagem deste Trabalho 77

6 Redes Toro e Algoritmos de Roteamento 79
6.1 Algoritmos de Roteamento para Redes Toro 85
6.1.1 Abordagem de Dally e Seitz 85
6.1.2 Abordagem de Duato, Ni e Yalamanchili 87
6.1.3 Abordagem de Glass e Ni 88
6.1.4 Abordagem de Draper e Petrini 92
6.1.5 Comparação de Algoritmos de Roteamento para Redes Toro 93

7 Rede Toro 2D Bidirecional com Roteamento Turn Model 95
7.1 Modelagem e Validação através de Síntese Comportamental 98
7.1.1 Exploração do Espaço de Projeto 102
7.1.2 Avaliação de Área da NoC Gerada com Síntese Comportamental 107
7.1.3 Avaliação de Latência da NoC Gerada com Síntese Comportamental 110

8 Rede Toro 2D com Canais Virtuais 115
8.1 Modelagem e Validação Através de Síntese Comportamental 119
8.1.1 Avaliação de Latência da NoC Gerada com Síntese Comportamental 122
8.1.2 Avaliação de Área da NoC Gerada com Síntese Comportamental 123

9 Conclusões e Trabalhos Futuros 127

Referências 129

1 Introdução

Um *System-on-chip*, ou SoC, incorpora um sistema complexo inteiro em um único circuito integrado (CI), podendo conter processadores, memórias, controladores de periféricos e outros elementos de processamento. A crescente complexidade desses sistemas e a demanda pela redução do tempo de chegada de produtos ao mercado (em inglês, *time-to-market*) leva a mudanças essenciais na maneira como esses sistemas são projetados [4]. Uma das estruturas críticas em qualquer SoC é a arquitetura interna de comunicação entre módulos do sistema. Tradicionalmente implementadas como arquiteturas de comunicação baseadas em interconexão multiponto como provido por barramentos, estas arquiteturas estão se transformando. Por força da necessidade de melhor escalabilidade destas estruturas, arquiteturas mais elaboradas estão surgindo, frequentemente denominadas de redes intrachip (em inglês, *Network-on-Chip* - NoC).

Os esquemas de comunicação com barramentos têm um baixo custo de implementação, já possuem uma longa história em reuso de componentes e em geral têm um bom desempenho em termos de consumo de potência e latência para conectar poucos módulos. Porém, tal organização tem um custo proibitivo com o aumento da quantidade de módulos a interconectar, o que acarreta aumento da latência de comunicação, aumento da dissipação de potência, em virtude do aumento do tamanho dos fios, e o aumento da complexidade da arbitragem de controle da comunicação. Dessa forma, estruturas do tipo redes intrachip ou NoCs ganham destaque como forma de aumentar a escalabilidade de comunicação em SoCs complexos [2] [15]. Uma rede intrachip pode ser definida como um conjunto de roteadores e canais ponto-a-ponto que interconectam os núcleos de um SoC de modo a suportar a comunicação entre esses núcleos. Tais redes ampliam o espaço de soluções de projeto de estruturas de comunicação e trazem como vantagem a largura de banda escalável, o uso de conexões ponto-a-ponto curtas, com menor dissipação de potência, e o paralelismo na comunicação.

Porém, tal expansão do espaço de projeto tem custo: é necessária a elaboração de métodos de projeto e construção de redes intrachip, um foco atual na área acadêmica e na indústria. Diversos arcabouços são apresentados na literatura para dar suporte ao projeto de estruturas de comunicação intrachip incluindo *NetChip* [1], OCCN [9], *Xpipes Compiler* [20], SUNMAP [31], *ATLAS* [34], etc. Em sua maioria, as soluções apresentadas baseiam-se no uso de codificação RTL que pode ser sintetizada para topologias específicas de estruturas de comunicação ou em modelagem em alto nível não sintetizável, utilizada para validação de estruturas de comunicação através de simulação apenas. Nesta última, não há um fluxo de projeto contínuo desde a modelagem em alto nível até a descrição sintetizável verificada. Quando se necessita avaliar uma proposta de modelo de estrutura de comunicação intrachip, o projetista recorre de ferra-

mentas que permitam descrever tal projeto em alto nível para realizar o trabalho de validação de forma mais rápida. Depois de validado, o modelo proposto precisa ser transcrito corretamente para uma arquitetura sintetizável, e validado com ferramentas adequadas.

Em busca de uma solução diferente para o projeto estruturas de comunicação intrachip do que é explorado na literatura, a síntese comportamental oferecida por ferramentas comerciais já disponíveis é avaliada neste trabalho como alternativa para o projeto de estruturas de comunicação do tipo redes intrachip. A utilização de síntese comportamental na implementação de redes intrachip permite que esses sistemas possam ser modelados e validados em uma linguagem de alto nível, como SystemC, e posteriormente traduzidos automaticamente para descrições sintetizáveis por ferramentas comerciais. Isso viabiliza modelar sistemas de comunicação intrachip bastante complexos de uma maneira mais intuitiva e em menor tempo com um fluxo de projeto contínuo desde a modelagem até o ASIC ou FPGA. Porém, tal tipo de síntese não foi definida especificamente para o projeto de estruturas de comunicação intrachip, que essencialmente têm pouca lógica de processamento e muita troca de sinais de controle, ocasionando a geração de hardware menos eficiente para garantir a sincronização perfeita do sistema e não comprometer seu funcionamento.

Neste trabalho, o processo de síntese comportamental oferecido pela ferramenta *Cynthesizer*, da FORTE Design Systems, é avaliado especificamente para o projeto de redes intrachip. Foram escolhidos estudos de caso com complexidade suficiente para avaliar o processo de modelagem e validação disponibilizados pela ferramenta. Os estudos de caso contemplaram redes com topologia toro 2D bidirecional e algoritmos de roteamento adaptados da literatura [11] [15] [17], com e sem utilização de canais virtuais. A adaptação dos algoritmos, que foram inicialmente desenvolvidos para redes de topologia malha [17] ou para topologia toro unidirecional [11] [15], demandou um esforço considerável de avaliação para comprovar a ausência de *deadlocks* antes da implementação e validação através de síntese comportamental. Todo o processo de modelagem, incluindo o estilo de codificação em SystemC aceito pela ferramenta, e o processo de validação foi estudado e aplicado nos estudos de caso. Foi explorada durante o projeto a integração com outras ferramentas de síntese RTL, lógica e física e simuladores.

1.1 Motivação

Como motivação para o desenvolvimento deste trabalho, está a constante evolução do estado de arte em projetos de CIs de larga escala. Tal cenário de evolução aponta cada vez mais para a inviabilidade do uso de estruturas de comunicação intrachip do tipo barramento em SoCs complexos, motivando assim o esforço na pesquisa de NoCs. Essas redes trazem junto com o aumento do espaço de soluções de projeto dessas estruturas de comunicação, um grau de complexidade de projeto bastante acentuado. Essa complexidade está associada a carência de

métodos de desenvolvimento práticos e com resultados realmente eficientes. O custo da complexidade do projeto desse tipo de infraestrutura de comunicação influencia diretamente o custo da produção de SoCs.

As pressões e necessidades de mercado têm influência na crescente complexidade desses SoCs e na redução do tempo de chegada desses componentes ao seu público consumidor. Assim, o processo de concepção desses sistemas necessita ser rápido e oferecer resultados satisfatórios em termos de custo-benefício. Isso indica cada vez mais mudanças na maneira como esses sistemas são projetados. Uma das conseqüências dessa mudança de concepção é que o uso de descrições RTL na captura inicial da descrição formal do sistema poderá se tornar inviável, mostrando a necessidade de elevar o nível de abstração de descrições iniciais executáveis do sistema.

1.2 Objetivos

O principal objetivo deste trabalho é dominar o processo de modelagem e validação de projeto oferecido pela síntese comportamental e utilizá-lo para desenvolver estruturas de comunicação intrachip. Para tanto, os seguintes objetivos específicos são determinantes:

- Dominar o fluxo de projeto da ferramenta *Cynthesizer* e sua interação com fluxos de ferramentas de síntese RTL para ASIC e FPGA e todo o processo de simulação e validação.

- Avaliar a sobrecarga causada pelo uso de ferramentas de síntese comportamental na descrição de arquiteturas de comunicação, quando comparado com a codificação direta em RTL.

- Dominar o estilo de descrição comportamental em SystemC aceito pela ferramenta e a utilização de diretivas de síntese para explorar espaço de projeto de descrições de NoCs.

- Desenvolver estudos de caso complexos o suficiente para avaliar o processo.

- Avaliar os resultados obtidos nas diversas fases do projeto para cada estudo de caso.

- Avaliar a factibiliade da utilização de tal síntese para o projeto de estruturas de comunicação intrachip.

1.3 Contribuições

Como contribuições principais deste trabalho estão uma avaliação específica da síntese comportamental oferecida pela ferramenta *Cynthesizer* para o projeto de redes intrachip e a adap-

tação e avaliação de algoritmos da literatura desenvolvidos para redes de topologia toro 2D unidirecional e topologia malha para a rede toro 2D bidirecional.

Também como contribuição deste trabalho está um método para utilização da ferramenta ATLAS [34] como auxiliar no processo de avaliação da modelagem comportamental de estruturas de comunicação intrachip.

1.4 Organização do Restante do Documento

O restante do documento está organizado como segue. O Capítulo 2 traz uma revisão de literatura sobre a síntese comportamental. O Capítulo 3 apresenta a ferramenta de síntese comportamental utilizada neste trabalho o ambiente *Cynthesizer*. O Capítulo 4 discute as principais diferenças de estilo entre codificações RTL e comportamental. O Capítulo 5 apresenta arcabouços de projeto em alto nível para modelar estruturas de comunicação intrachip e apresenta o estilo de modelagem adotada neste trabalho. O Capítulo 6 traz a revisão dos algoritmos de roteamento para redes com topologia toro. O Capítulo 7 apresenta um dos estudos de caso do trabalho com rede de topologia toro 2D bidirecional sem canais virtuais e algoritmo adaptado da literatura revisada. O Capítulo 8 apresenta a modelagem e validação de uma rede de topologia toro 2D bidirecional com a utilização de canais virtuais e algoritmo adaptado da literatura revisada. Por fim, o Capítulo 9 apresenta conclusões e uma proposta de trabalhos futuros.

2 Síntese Comportamental

Segundo Gajski et al. em [16], a atividade de síntese é todo processo automatizado que refina a descrição abstrata do circuito em termos de primitivas mais próximas da sua realização física. O processo de síntese pode ser aplicado a diferentes descrições de projeto em diversos níveis de abstração, podendo ser classificado justamente por tais critérios. Antes de diferenciar os tipos de síntese é preciso definir os níveis de abstração sobre os quais se trabalha, tais como: comportamental, nível de transação (TL), nível de transferência entre registradores (RTL) e físico. No nível *comportamental*, a descrição aborda a funcionalidade. Informações de temporização não fazem parte destas descrições. Nesse nível, também podem ser consideradas as restrições impostas ao projeto e o comportamento de suas interfaces. A modelagem *TL* é temporizada, porém sem estar amarrada a ciclos de relógio. Nesse tipo de descrição, os componentes são modelados como processos, portas e canais abstratos. Processos definem o comportamento particular de um módulo e a comunicação entre os módulos ocorre através dos canais abstratos. No nível *RTL*, são especificados os elementos estruturais do sistema, bem como as máquinas de estado que controlam sua funcionalidade a cada ciclo do relógio. A descrição possui informações de temporização e o tempo é dividido em intervalos chamados *passos de controle* [16]. No nível *físico* a descrição contém uma lista com todas as células físicas do sistema.

A *Síntese RTL* parte de um conjunto pré-determinado de estados e um conjunto de transferências entre os registradores e gera a estrutura correspondente em duas partes: caminho de dados (do inglês, *datapath*) e unidade de controle. O *datapath* é a estrutura contendo todos os elementos de armazenamento de informações e unidades funcionais que transferem e/ou transformam dados entre registradores. A unidade de controle dirige a sucessão dos estados durante a operação do sistema.

A *Síntese Comportamental* transforma uma descrição comportamental em uma representação arquitetural. Ela parte de um conjunto de processos que se comunicam através de mensagens ou variáveis compartilhadas, e gera a estrutura do sistema com componentes de uma biblioteca previamente caracterizada, que se adaptam a funcionalidade desejada.

2.1 Comparação entre Síntese RTL e Síntese Comportamental

Esta Seção introduz a comparação entre fluxos de projeto e síntese RTL e Comportamental apresentado por Lee em [24]. Para tanto, é necessário definir o que é *espaço de soluções* de um

projeto. Esse conceito refere-se ao conjunto de diferentes organizações que podem implementar uma determinada arquitetura. Tanto na síntese RTL como na Comportamental é possível explorar esse espaço, porém de maneira diferente em cada uma.

A síntese RTL automatizada tradicional inicia com uma descrição arquitetural na qual o escalonamento e alocação já estão determinados pelo projetista, que gera a descrição de entrada. Se algum dos requisitos de projeto não for atendido por essa descrição após a síntese, a maneira de atendê-lo é, tipicamente, modificar a arquitetura RTL [24]. Desse modo, o projetista deve modificar manualmente o escalonamento e a alocação para explorar o espaço de soluções de projeto. Para projetos muito complexos, isto acaba tornando-se um gargalo no desenvolvimento. A Figura 1 ilustra o fluxo de síntese RTL tradicional. Na Figura, as elipses, os quadrados e os losangos representam respectivamente, ações, repositórios e decisões. Os processos manuais são marcados com M e os automatizados com A.

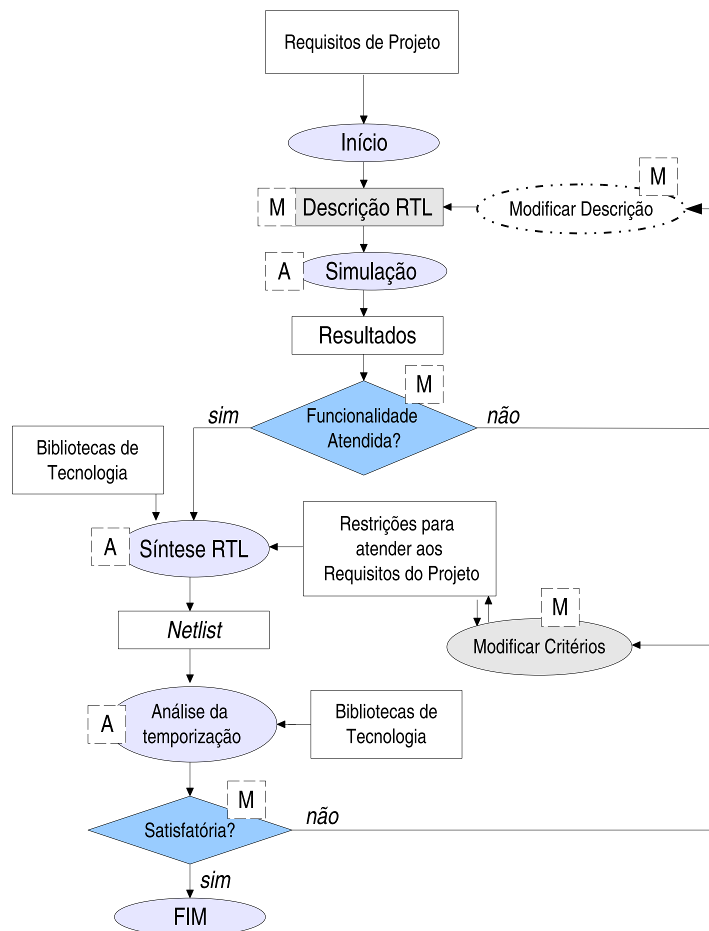

Figura 1 – Fluxo de síntese RTL convencional, baseado em [24].

Na síntese RTL, a descrição RTL pode ser verificada, por exemplo, através de simulação. Ela pode ser sintetizada quando atender os requisitos de projeto. A síntese RTL é automatizada por ferramentas capazes de associar a tecnologia alvo do projeto, utilizar critérios/diretivas de otimização e restrições à síntese e disponibilizar estimativas antecipadas de resultados. A

síntese associa os componentes arquiteturais da descrição aos disponibilizados por bibliotecas de tecnologia respeitando a temporização definida na descrição inicial. Assim, é obtido um *netlist* que contém a descrição da arquitetura em termos das primitivas físicas da tecnologia alvo do projeto.

No processo de síntese comportamental, a funcionalidade do circuito é especificada de forma algorítmica, a chamada *descrição comportamental*. Neste tipo de descrição não é definido o comportamento do circuito a cada ciclo de relógio nem os recursos necessários para executar o algoritmo. A síntese de alto nível transforma a descrição comportamental em uma micro-arquitetura, na qual, após fixado o comportamento em ciclos de relógio, podem ser satisfeitas as restrições de área, desempenho e potência. A síntese comportamental pode facilitar a exploração do espaço de soluções de projeto, caso um conjunto de arquiteturas RTL possa ser derivado automaticamente a partir de uma descrição do comportamento do circuito.

A Figura 2 mostra um fluxo de síntese comportamental, salientando como este gera automaticamente um conjunto de micro-arquiteturas RTL. Na Figura, as elipses, os quadrados e os losangos representam, respectivamente, ações, repositórios e decisões. Os processos manuais são marcados com M e os automatizados com A.

Nesse fluxo, após a descrição inicial ser validada segundo os critérios funcionais do projeto, pode-se executar a síntese comportamental. Essa deve ser capaz de associar diretivas para a exploração de espaço de projeto e a tecnologia alvo de forma automatizada. Como resultado é obtido um conjunto de micro-arquiteturas. A ferramenta que automatiza esse processo deve ser capaz de gerar estimativas inciais sobre o atendimento dos requisitos de projeto. Essa estimativa auxilia na escolha da micro-arquitetura que melhor satisfaz os requisitos. Com o resultado obtido, é possível realizar em seguida o processo de síntese RTL automatizada tradicional.

2.2 A Síntese Comportamental segundo Lee

Lee, em [24], apresenta um fluxo de atividades para realização de síntese Comportamental, para ferramentas que automatizam esse processo de síntese.

A entrada desse processo é uma descrição do comportamento do circuito. Esse comportamento pode ser especificado em uma linguagem de descrição de hardware de alto nível. Em uma especificação comportamental um *bloco básico* é definido como uma seqüência de atribuições que não contém testes nem desvios, exceto o teste de condição de parada. Desta descrição pode-se derivar um *grafo de fluxo de dados* (em inglês, *data flow graph* ou DFG) do bloco básico, onde cada vértice está associado a uma operação e cada aresta a uma variável. As dependências de dados estão implícitas nas direções das arestas do grafo.

As variáveis em um DFG podem ser classificadas em três subconjuntos: entradas primárias (V_I), saídas primárias (V_O) e variáveis intermediárias (V_M). O fluxo de dados em um DFG é denotado por $v1 \xrightarrow{O_1} v2 \xrightarrow{O_2} ... \xrightarrow{O_{n-1}} vn$, onde v_i, $1 \leq i \leq n$, é uma variável do conjunto

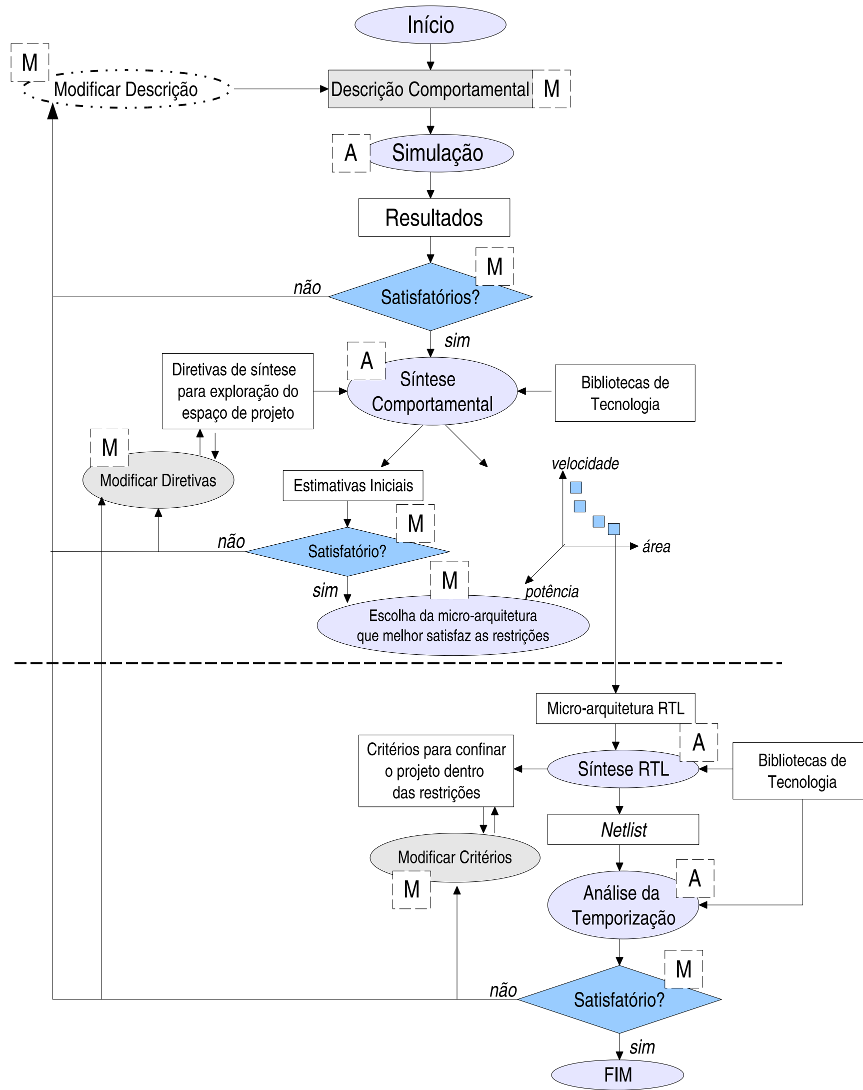

Figura 2 – Fluxo de síntese comportamental, baseado em [24].

$V_I \cup V_O \cup V_M$, e O_j associado a cada $\longrightarrow$, $1 \leq j \leq n$, é uma operação com v_j como operando e v_{j+1} como resultado. Ainda, se v_1 e v_2 representam o mesmo vértice, o DFG é *cíclico* e v_1 ou v_2 são chamadas *variáveis limitadas*. Um DFG cíclico é inferido de construções de linguagem do tipo laço, como comandos *for* e *while*.

Para cada operação O associada com um nó em um DFG, $O.earliest$ denota o ciclo mais cedo em que a operação pode executar e $O.latest$ denota o ciclo mais tardio em que a operação pode executar sem violar a dependência de dados expressa no DFG. A *folga* (do inglês, *slack*) de O é definida pelo intervalo $[O.earliest, O.latest]$, e a *mobilidade* de O é definida como $O.latest - O.earliest$. A seqüência de operações com mobilidade igual a zero é chamada *caminho crítico*. Folgas em mobilidade podem ser utilizadas para indicar a liberdade de escalonamento para uma operação.

A tarefa de *escalonamento* atribui um tempo de execução, ou passo de controle, para cada operação da especificação comportamental. O DFG obtido após o escalonamento é chamado

DFG *Escalonado* (em inglês *scheduled data flow graph* ou SDFG). Segundo Lee, os algoritmos de escalonamento podem ser classificados como de *transformação* ou *iterativos/construtivos*. Os escalonamentos com transformação partem de uma solução inicial e tentam realizar transformações, preservando o comportamento, para obter uma melhoria da estrutura inicial. A abordagem *iterativa/construtiva* escalona a operações de forma construtiva a partir da junção de componentes um a um. Algoritmos que partem dessa abordagem são o escalonamento *as-soon-as-possible* (ASAP), *as-late-as-possible* (ALAP), escalonamento do caminho crítico, etc.

ASAP e ALAP são tipos simples de algoritmos construtivos. Para cada operação O do DFG, o algoritmo ASAP escalona para o ciclo corrente $O.earliest$, enquanto ALAP escalona $O.latest$. Esses algoritmos são mais rápidos, porém não consideram o caminho crítico e alocam mais recursos do que os necessários. O algoritmo de escalonamento do caminho crítico escalona primeiramente as operações do caminho crítico do DFG. As operações restantes são escalonadas de acordo com sua mobilidade.

Dado um SDFG, a tarefa de *alocação de recursos* atribui elementos de hardware para realizar operações e produzir uma micro-arquitetura RTL. Para definir formalmente a tarefa de alocação de recursos é preciso definir o conceito de *tempo de vida de uma variável*. Segundo Lee, em um SDFG acíclico, o *instante de nascimento* de uma variável v, denotado por $v.birth$ é o ciclo em que ela é definida; o *instante de morte* de v, denotado por $v.death$, é o último ciclo em que a variável é utilizada. O *tempo de vida* de v é definido pelo intervalo $[v.birth, v.death]$.

A *alocação de recursos* é dividida em alocação de registradores, de módulos de processamento e de interconexões [24]. A alocação de registradores, denotada por **R**, pode ser considerada como o problema de encontrar uma partição $\{R_1, R_2, ..., R_r\}$ de $V_I \cup V_O \cup V_M$ tal que para quaisquer duas variáveis v_i e v_j em R_k, $1 \leq k \leq r$, seus tempos de vida não se sobrepõe. Similarmente, a alocação de módulos de processamento, denotada por **M**, pode ser considerada como o problema de encontrar uma partição $\{M_1, M_2, ..., M_m\}$ de O tal que quaisquer duas operações o_i e o_j em M_k, $1 \leq k \leq m$, não tenham tempos de execução conflitantes. A partir de um SDFG é derivado um grafo não dirigido, chamado *grafo de alocação de módulos* (em inglês, *module allocation graph* ou MAG), em que cada nodo corresponde a uma operação do SDFG e cada aresta corresponde à capacidade de compartilhamento de um mesmo módulo entre dois nodos conectados. O grau de capacidade de compartilhamento pode ser diferenciado atribuindo pesos às arestas. A Figura 3 apresenta um SDFG e o MAG com pesos correspondentes, supondo que dois somadores sejam reservados, e que o peso de compartilhamento entre $+_a$ e $+_b$ é -1 e entre $+_a$ e $+_c$, 3. Como o peso entre $+_a$ e $+_c$ é maior que aquele entre $+_a$ e $+_b$, um somador será compartilhado entre $+_a$ e $+_c$ e outro ($+_b$) será dedicado.

A alocação de interconexões associa fios a sinais de *vai-um* e variáveis entre registradores e módulos. Dois sinais podem ser associados para o mesmo fio, desde que seus tempos de vida não se sobreponham.

A Figura 4 (a) mostra um exemplo de especificação comportamental, onde cada variável representa um nome de sinal e cada operação representa uma computação que será realizada.

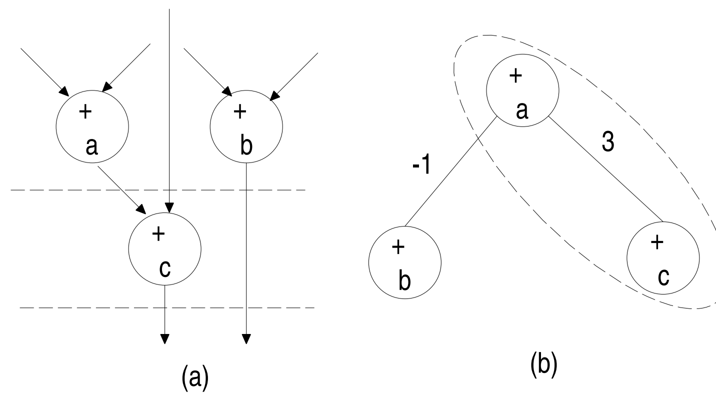

Figura 3 – Exemplo de um MAG, retirado de [24]: (b) com pesos; (a) derivado do SDFG.

A Figura 4(b) apresenta um DFG que pode ser derivado da especificação inicial. A Figura 4(c) apresenta uma possibilidade de arquitetura para o comportamento descrito no algoritmo inicial. Nessa arquitetura, três registradores são alocados pela síntese de alto nível para armazenar os valores das variáveis: $R1$ para a e d, $R2$ para b e $R3$ para c e k. Um multiplicador e um somador são mapeados para realizar as operações especificadas. O tempo de execução de cada operação também é escalonado pela síntese de alto nível. No primeiro ciclo, O multiplicador gera o produto em $R1$ a partir dos operandos de $R1$ e $R2$ (a e b). No segundo ciclo, o somador adiciona os operandos de $R1$ e $R3$ (c e d) e armazena o produto k em $R3$.

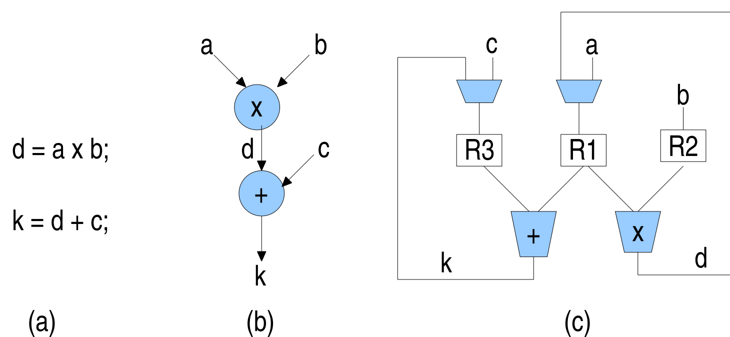

Figura 4 – Derivação da especificação comportamental até a arquitetura RT, retirada de [24]: (a) exemplo de especificação comportamental; (b) um possível DFG da descrição; (c) uma possibilidade de arquitetura RT.

2.3 Fronteiras entre a Síntese Comportamental e a Síntese RTL

Segundo e.g. [7] e [21], existe uma sobreposição de funcionalidades entre as tarefas de alocação e amarração de recursos entre as sínteses comportamental e RTL, conforme mostra a Figura 5. Devido a esse fato, ferramentas de síntese comportamental podem realizar apenas a

tarefa de escalonamento enquanto que as tarefas de alocação e amarração podem ser deixadas a cargo das ferramentas de síntese RTL.

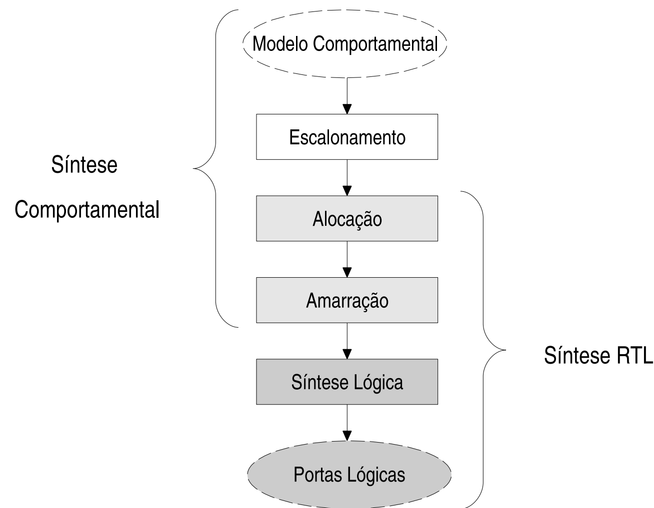

Figura 5 – Escopos de síntese comportamental e RTL, retirado de [21].

Segundo de Ku e de Micheli [22], *pontos de sincronização* são os pontos do tempo onde ocorre interação do sistema com o ambiente no qual ele está inserido. Em sistemas digitais síncronos a diferença essencial entre os níveis de abstração são os *pontos de sincronização* utilizados. A Tabela 1, retirada de [7], apresenta os diversos pontos de sincronização utilizados em diferentes níveis de abstração. No nível Sistema, processos são sincronizados através de trocas de mensagens. Processos podem ser representados no nível Algorítmico por grafos de dados/controle, sincronizados por eventos de entrada e saída (E/S). No nível Algorítmico, o grafo de fluxo de dados/controle pode ser representado utilizando máquinas de estados finitas com *datapath* (FSMD). No nível RTL, transferências de dados são realizadas dentro dos limites de cada ciclo de relógio e o modelo de entrada contempla as FSMs (Máquinas de Estados Finita, do inglês *Finit State Machines*), BDD (Diagrama de Decisão Binária, do inglês *Binary Decision Diagram*) e as Equações Booloeanas. No nível Físico, as mudanças de valores nos fios definem a sincronização.

Nível de Abstração	Pontos de Sincronização	Modelo de Entrada
Sistema	Mensagens entre processos	Processos com comunicação
Algorítmico	Eventos E/S	CFG , DFG, CDFG , FSMD
RTL	Ciclo de Relógio	FSM , BDD, Equações Booleanas
Físico	Mudança de valores nos fios	Netlist de portas e modelo de leiaute

Tabela 1 – Pontos de sincronização e modelos de entrada em diferentes níveis de abstração.

A Tabela 2, retirada de [7], mostra o escopo das tarefas realizadas pelas sínteses comportamental e RTL e os modelos de entrada associados a cada tarefa. Pode-se observar que existe uma

sobreposição de funcionalidades entre a síntese comportamental e RTL. O modelo de entrada da síntese RTL e a arquitetura produzida têm em comum a sincronização dentro dos limites dos ciclos de relógio, isto é, são *precisas em nível de ciclo*. Na síntese comportamental essa precisão somente é obtida após a tarefa de escalonamento.

Tarefas da Síntese	Comportamental		RTL	
	Pontos de sincronização	Modelo de entrada	Pontos de sincronização	Modelo de entrada
Escalonamento	Eventos E/S	CFG	-	-
Alocação de recursos	Relógio	FSMD	Relógio	FSMD
Amarração de recursos	Relógio	FSMD + Recursos	Relógio	FSMD + Recursos
Síntese Lógica	-	-	Relógio	FSM + Datapath
Síntese do Layout	-	-	-	-

Tabela 2 – Escopo da síntese comportamental e da síntese RTL [7].

A síntese comportamental traz um conjunto potencial de vantagens tais como o encurtamento do ciclo de desenvolvimento do projeto, antecipação das estimativas de desempenho, maior facilidade de reuso, independência de tecnologia, melhor relação dos compromissos assumidos com relação a área, tempo, potência e *desempenho*. Contudo, conforme [21], existem alguns problemas ainda hoje que dificultam que esse tipo de síntese seja utilizado pela indústria, incluindo: o resultado da síntese é difícil de entender, o modelo de entrada não é prático e é difícil integrar síntese comportamental nos fluxos de projeto existentes. Além disto, e em muitos casos, ferramentas RTL têm um resultado nitidamente melhor.

A referência [21] apresenta uma proposta de integração eficiente entre síntese comportamental e RTL. Aquele trabalho introduz um fluxo de síntese comportamental, cujas principais características são: a síntese é centrada na tarefa de escalonamento e sua saída é um código de alto nível preciso em nível de ciclo, que descreve uma FSM que pode ser explorada via síntese RTL; o modelo comportamental de entrada é um subconjunto de VHDL que permite misturar protocolos precisos e modelos comportamentais puros em uma mesma especificação. O código RTL produzido é fácil de entender, alegam ainda os autores do trabalho.

O escalonamento proposto consiste dividir operações complexas do sistema em operações mais simples, para que estas fiquem confinadas em um ciclo relógio. O método de escalonamento usado parte de FSMs complexas em nível comportamental e produz um conjunto de FSMs precisas em nível de ciclo. Dois passos são utilizados durante este processo. O primeiro passo manipula os estados da FSM implícitos na descrição comportamental. Isto se dá, principalmente, devido à dependência de dados e laços que incluem ou não pontos de sincronização.

Para isso, o algoritmo de escalonamento dinâmico de laços é utilizado. Esse algoritmo produz um novo modelo onde as transações são formadas por operações que podem ser executadas em paralelo em um passo de controle. Este passo assume que existe um período de relógio infinito e uma quantidade de recursos também infinita. O segundo passo realiza uma retemporização (do inglês, *retiming*) em nível de sistema para assegurar que todas operações simples, que foram anteriormente extraídas de uma descrição complexa, possam ser executadas em um único ciclo de relógio. Assim, novos estados de FSM podem ser incluídos para garantir a execução correta das operações do sistema. A descrição resultante da execução do escalonamento pode ser utilizada em uma ferramenta de síntese RTL tradicional para as etapas de alocação e amarração de recursos.

2.4 Uma Classificação de Ferramentas de Síntese Comportamental

A referência [21] classifica três gerações distintas de ferramentas de síntese comportamental, a partir da contribuição destas para o processo de síntese de alto nível e da evolução de seus algoritmos e técnicas de otimização.

2.4.1 Primeira Geração

A primeira geração de ferramentas foi desenvolvida principalmente pela comunidade de arquitetura de computadores e sua principal contribuição foi a definição precisa das tarefas de síntese. A síntese baseada em descrições RTL ainda não era realidade na época. O trabalho de maior destaque na época é o ambiente de projeto MIMOLA [29], baseado na linguagem MIMOLA (*Machine Independent Microprogramming Language*) .

Sistema MIMOLA

O ambiente de projeto MIMOLA ([29], foi concebido como uma ferramenta para o projeto de estruturas de hardware. Esse ambiente compreende uma série de ferramentas para descrever, simular e sintetizar circuitos. Ele contém um método de projeto para processadores digitais a partir de uma especificação de alto nível. A característica chave deste método é a síntese de um processador a partir dos programas que constituem aplicações típicas [29]. MIMOLA dá atenção especial à predição de desempenho, paralelismo e flexibilidade. A Figura 6 apresenta o diagrama do fluxo de projeto do ambiente.

A especificação de um conjunto de instruções é um exemplo de especificação funcional de projeto. Contudo, esse tipo de especificação restringe o espaço de projeto, porque implica a existência de registradores, multiplexadores, etc. Porém, especificações funcionais têm um

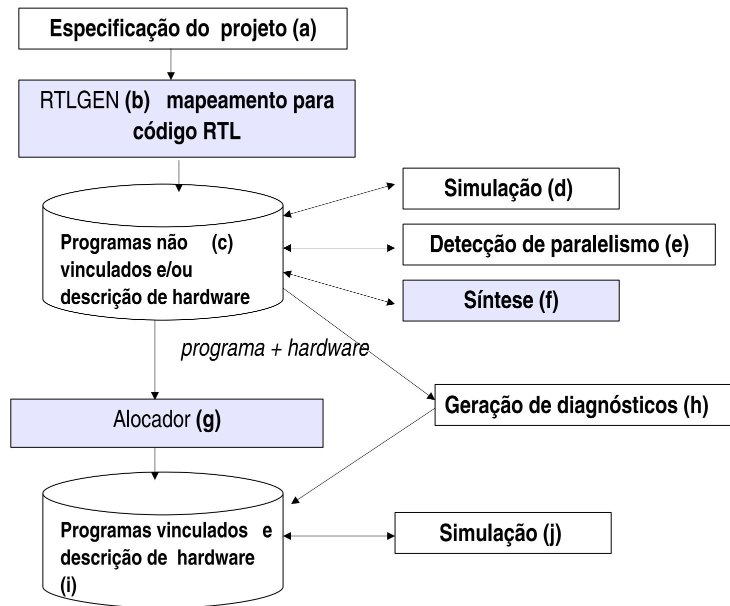

Figura 6 – Diagramas de projeto do sistema MIMOLA, baseado em [29].

espaço de projeto maior que especificações estruturais. A partir da especificação funcional do sistema (Figura 6(a)) é possível gerar uma série de soluções alternativas, todas elas formadas por um caminho de dados com componentes RTL e por uma unidade de controle representada por um microprograma (Figura 6(i)).

Dada a descrição funcional das aplicações que serão executadas no processador projetado, é feita a alocação por substituição de elementos RT de funcionalidade equivalente às operações em alto nível (Figura 6(b)). A troca de elementos da descrição de alto nível por elementos equivalentes em RTL é definida por regras de substituição. Essas regras são armazenadas em uma biblioteca. A partir do momento em que se obtém uma descrição com elementos RT (Figura 6(c)) é possível realizar uma simulação para verificar se o programa resultante está correto (Figura 6(d)).

A detecção das funcionalidades que possam ser executadas em paralelo é diagnosticada após esse ponto do processo e tudo é agrupado em blocos paralelos (Figura 6(e)). Essa transformação é realizada com base na avaliação do fluxo de controle e das dependências de dados.

Após o mapeamento para RTL e a detecção de paralelismos, é realizada a atividade de síntese (Figura 6(f)). Nessa fase, é gerada uma descrição de hardware para um processador e são computados os seguintes parâmetros arquiteturais: número de portas de entrada e saída, já que o procedimento de síntese computa o número máximo de operações de leitura e escrita em um bloco para criar os registradores das portas; número de *funções*, que são criadas quando a execução paralela de todas as funções dentro de cada bloco é possível; *caminhos* (fios), o número necessário de *caminhos* é computado; e formato das instruções, onde assume-se que as unidades de hardware são controladas diretamente por um campo da instrução. O processo de geração de diagnósticos (Figura 6(h)) é responsável pela geração automática de alguns tipos de

teste, como para detecção de erros do tipo *stuck-at*. Se não for possível gerar automaticamente teste para alguma estrutura do sistema, esta deve ser modificada manualmente antes de continuar o processo do projeto.

Para permitir a estimativa de desempenho e geração de código para o hardware projetado, os programas de aplicação são vinculados aos recursos de hardware. Isso é feito por um *alocador* (Figura 6(g)). Esse alocador é composto por dois componentes. O primeiro tenta encontrar conjuntos de *funções* que podem ser mapeadas para os operadores e portas de memória de operações de leitura e escrita. O segundo seleciona o conjunto que proporciona o tempo mínimo de execução. Após essa fase, são obtidos a descrição do hardware e o microprograma (Figura 6(i)), e é possível realizar uma nova simulação para validar o funcionamento do sistema gerado (Figura 6(j)).

2.4.2 Segunda Geração

A segunda geração de ferramentas se caracterizou pela escolha de *domínios de aplicação restritos* e a geração de arquiteturas com *bloco de controle* e *datapath*. Essas ferramentas se concentravam em poucas tarefas de síntese e eram baseadas em geração de máquinas de estados RTL e mapeamento tecnológico. Muitas ferramentas foram desenvolvidas nessa fase para uma variedade de aplicações: DSP, controladores embarcados, circuitos de comunicação, etc.

O principal inconveniente dessas ferramentas é a dificuldade de gerar eficientemente a descrição RTL, principalmente devido à falta de processos que fossem capazes de fazer otimizações durante o fluxo de síntese. O principal ambiente de projeto concebido nessa época foi o *System Architect's Workbench* [40], que engloba uma série de ferramentas para converter uma descrição algorítmica em um conjunto de componentes em nível RTL, cujo controle é especificado por uma tabela de transição de estados. Neste ambiente de síntese de alto nível foi introduzida a fase de *Particionamento Arquitetural* [23], que é uma fase do projeto em nível de sistema que determina o número de blocos que serão usados para implementar o projeto e o subconjunto do comportamento que será implementado em cada um deles. Essa fase acontece antes que a síntese RTL inicie. As características estruturais determinadas no particionamento podem ser utilizadas para conduzir as decisões de síntese de baixo nível.

O objetivo principal do particionamento arquitetural é descobrir a estrutura implícita na descrição de um comportamento, conforme ilustrado na Figura 7. Determinar a estrutura correta tem um efeito positivo na área e no desempenho do projeto. Boas partições podem reduzir o comprimento global dos fios através de agrupamentos de partes da arquitetura que se comunicam muito. Assim, são reduzidas as interconexões necessárias para implementar dependências de dados entre as partições.

O particionamento arquitetural fornece meios de coordenar todas essas fases da síntese RTL, já que providencia informações preliminares sobre a estrutura do projeto. A ferramenta apre-

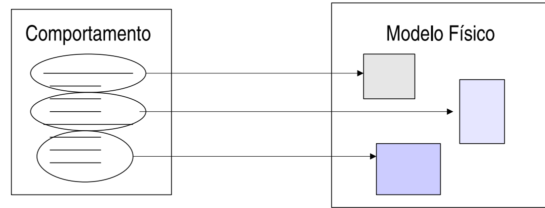

Figura 7 – Particionamento arquitetural, retirado de [23].

sentada no trabalho de [23] é o APARTY, um dos pacotes do *System Architect's Workbench*. Ela utiliza um algoritmo de *clusterização multiestágio* para extrair a estrutura de alto nível de um projeto, considerando critérios como área e interconexões.

A clusterização é utilizada para agrupar operadores em grupos significativos para síntese, ou seja, em grupos cuja funcionalidade pode ser combinada em uma unidade de hardware. O algoritmo de clusterização comum considera um conjunto de objetos e os agrupa acordo com algum critério de aproximação. A Figura 8 mostra os objetos que são agrupados de acordo com suas medidas de proximidade. A e B são clusterizados primeiro e considerados como um único objeto. Já a clusterização em múltiplos estágios é uma variação desse algoritmo. Múltiplos estágios facilitam a implementação e o desempenho da clusterização.

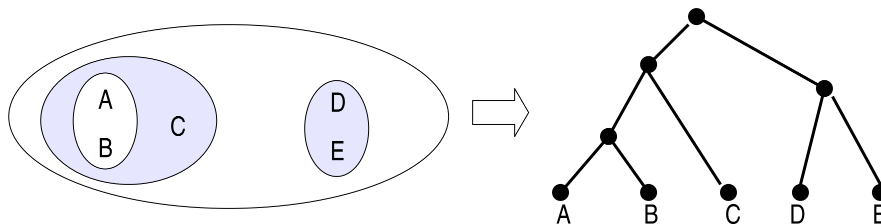

Figura 8 – Clusterização dos objetos A, B, C, D e E, retirado de [23].

No algoritmo de clusterização multiestágio ocorrem vários passos seqüenciais de clusterização, cada um tendo seu próprio critério de aproximação. Cada um desses passos é construído com os resultados do passo anterior. Em cada um deles é criada uma árvore de *clusters*. A partir disso, uma linha de corte é escolhida em algum nível dessa árvore e cada sub-árvore abaixo da linha torna-se um elemento que será considerado para a clusterização no próximo estágio. Após a construção da árvore completa é escolhida a melhor linha de corte para a clusterização final. Isso é ilustrado na Figura 9.

2.4.3 Terceira Geração

A terceira geração de ferramentas de síntese comportamental tenta fazer uso do investimento da indústria em síntese RTL, realizando a transformação da descrição comportamental para que

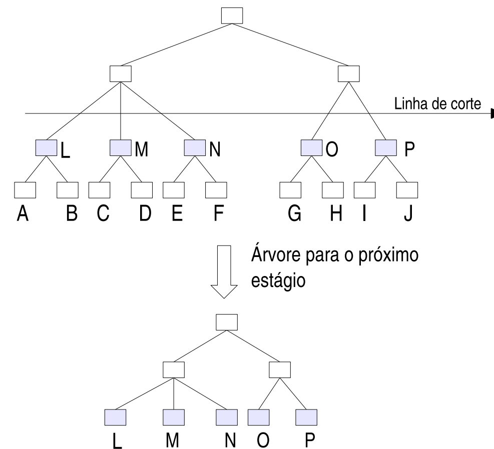

Figura 9 – Dois estágios da clusterização multiestágio. No primeiro estágio, a árvore completa é construída e uma linha é escolhida para produzir os *clusters* L, M, N, O e P. No segundo estágio, os *clusters* do estágio anterior são utilizados como base para uma nova iteração.

ela possa ser sintetizada por ferramentas de síntese RTL. Nesta Seção, são apresentadas breves descrições de trabalhos relacionados a esta geração de ferramentas.

No trabalho de Bergamaschi, descrito em [3], é apresentada uma abordagem alternativa para integrar as sínteses comportamental e RTL no mesmo fluxo de projeto. A síntese RTL é utilizada para executar parte das tarefas que seriam realizadas pela síntese comportamental e este modelo é chamado *Behavioral Network Graph*(BNG).

Na maioria das metodologias, a rede RTL gerada pelo processo de síntese de alto nível (em inglês *High Level Synthesis* ou HLS) é submetida à síntese lógica para otimização em nível de portas, que tenta satisfazer certas restrições de área e atrasos. A qualidade do resultado obtido depende da qualidade das duas sínteses utilizadas. Para produzir uma rede RTL eficiente, a síntese de alto nível deve estimar o efeito que cada decisão, tomada a partir da descrição algorítmica, terá sobre a rede de portas resultante. Este efeito é traduzido em estimativas de latência de operação, área ocupada ou potência consumida. Porém, alguns algoritmos ignoram aspectos importantes, como o tamanho e o atraso do bloco de controle, multiplexadores e registradores.

Segundo Bergamaschi [3], o principal problema para computar estes custos com precisão é que o modelo interno em que a síntese de alto nível opera é demasiado distinto da rede RTL final. Na maioria dos sistemas descritos em alto nível, os modelos internos são os grafos de controle e dados. Um CDFG representa uma especificação de um projeto de uma maneira muito diferente de uma implementação de hardware. Embora um CDFG contenha arestas e nós que representam valores e operadores (somadores, subtratores, etc.), geralmente ele não contém uma especificação explícita dos multiplexadores e lógica de controle necessários para a implementação do hardware.

No trabalho de [3], o BNG (em inglês, *Behavioral Network Graph*) é uma representação de uma rede RTL contendo descrições comportamentais não escalonadas. Para tanto, é preciso de-

terminar a rede RTL que pode representar todos os escalonamentos possíveis que uma descrição comportamental pode assumir. Outra questão importante é que não se deve criar representações ambíguas de um comportamento.

A Figura 10 ilustra a metodologia proposta em [3]. O CDFG é utilizado como uma árvore de análise estendida, representando a mesma semântica que uma linguagem de descrição de hardware. Além de otimizações do compilador, o CDFG pode ser submetido a transformações específicas, como extração de paralelismos e desdobramento de laços. O próximo passo é o mapeamento do CDFG em uma representação BNG . As tarefas de escalonamento, alocação e compartilhamento de recursos são realizadas sobre o BNG. Pode-se também realizar transformações lógicas e análise de *temporização* para avaliar com maior precisão os custos envolvidos durante a síntese de alto nível. Após essas tarefas, o BNG representa a rede RTL final.

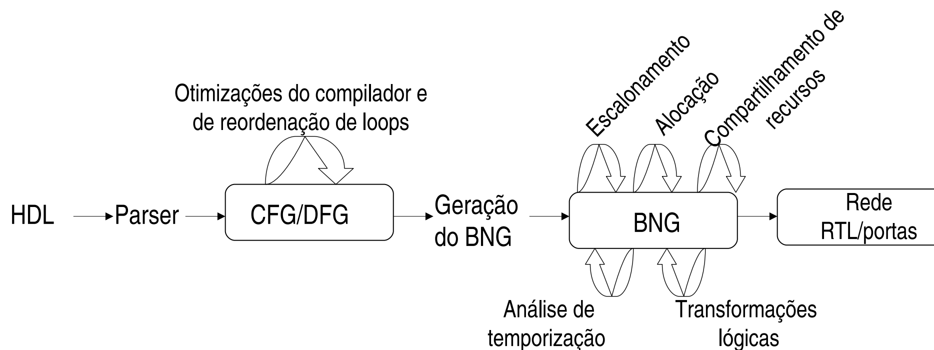

Figura 10 – Método de síntese de alto nível, conforme proposto em [3].

O método proposto por Dougherty e Thomas [12] unifica síntese comportamental e projeto físico, permitindo escalonamento, alocação, vinculação e posicionamento simultâneos. Isto ocorre através de um conjunto de transformações definidas em ambos domínios e levando a um sistema comportamental/físico.

Segundo [12], as ferramentas de síntese comportamental não são bem aceitas devido à baixa qualidade de seus resultados. O grande problema é que essas ferramentas têm pouca ou nenhuma base em projeto físico. Assim, alguns componentes do projeto são desenvolvidos com baixa qualidade desde o início do processo.

A proposta desenvolvida no trabalho citado tenta unificar projeto físico e síntese comportamental. Assim, decisões comportamentais e transformações são representadas como forças agindo em objetos do modelo comportamental (DFG). Do mesmo modo, transformações no projeto físico são representadas por forças agindo nos objetos *posicionáveis*. Nesta metodologia, os objetos do modelo comportamental e os objetos *posicionáveis* do modelo físico são considerados como sendo o mesmo objeto. Esta formulação permite que decisões comportamentais e físicas sejam tomadas simultaneamente.

O método parte de uma descrição HDL em alto nível. Este código é compilado para uma representação DFG. O DFG é modificado, seguindo o grafo da rede comportamental e permi-

tindo escalonamento e alocação simultâneos. A seguir, as operações são fisicamente dispostas no chip. Junto com a disposição, este processo combina operações sem o conhecimento do escalonador. Conforme esses eventos vão ocorrendo, restrições são geradas para assegurar que o resultado final seja um escalonamento válido. Após isso, todas as operações são vinculadas a unidades funcionais. Nesta fase ainda não se tem informações sobre o hardware e o escalonamento.

Baseado na fase de posicionamento, uma árvore parcial é construída para o projeto e utilizada para produzir uma série de escalonamentos e alocações que vão construindo simultaneamente o projeto físico.

2.4.4 Ferramentas de Síntese de Alto Nível Modernas

O processo de síntese com ferramentas modernas combina a automação com restrições específicas de alto nível, para que o projetista possa controlar a implementação do hardware e fazer com que seu projeto atinja uma qualidade significativamente melhor em menos tempo. As otimizações realizadas pelas ferramentas associadas são oriundas da evolução dos algoritmos e técnicas desenvolvidas nas ferramentas das gerações anteriores.

Entre os métodos e algoritmos que são utilizados nessas ferramentas está a técnica de *clusterização comportamental* [35]. Esta técnica é utilizada para aperfeiçoar desempenho, área e consumo de potência em projetos DSP (*Digital Signal Processing*). O método de clusterização comportamental permite combinar e otimizar processos de tal maneira que a qualidade dos resultados (QoR) seja significativamente melhor do que quando esses mesmos processos são implementados separadamente.

Algoritmos de clusterização, como os vistos na Seção 7, servem para determinar o particionamento de um projeto entre chips, *datapaths* ou entre hardware e software. Geralmente esses algoritmos têm como ponto de partida uma descrição comportamental monolítica e o objetivo de manter blocos que se intercomunicam juntos, ou próximos. Contudo, aplicações do tipo DSP têm um ponto de partida diferente. Geralmente, elas são especificadas como um conjunto de processos que se comunicam entre si. Cada processo corresponde a um bloco DSP. No processo tradicional de síntese RTL, cada bloco é implementado separadamente, em nível RTL, e depois combinados para formar o projeto completo. Já no processo de síntese comportamental, a clusterização comportamental pode ser utilizada para combinar os processos em conjuntos semelhantes antes do processo de síntese. A Figura 11 ilustra a clusterização comportamental em que os processos menores são agrupados em dois conjuntos.

Segundo Pursley [35], a clusterização comportamental permite trabalhar com blocos de tamanho maior para realizar otimizações de latência, reuso, blocos de dados e comunicação. A diminuição da latência é uma otimização de desempenho que reordena operações de tal forma que dois processos possam executar alguma computação ao mesmo tempo. Geralmente, essa

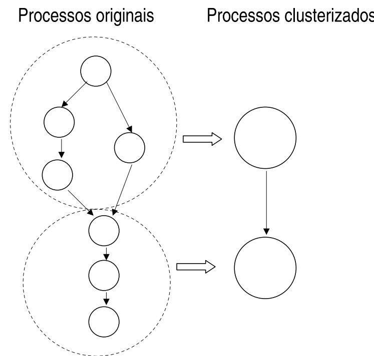

Figura 11 – Clusterização comportamental, retirado de [35].

transformação é aplicada a processos seqüenciais, ou seja, aqueles que possuem dependência um do outro. O tempo total de execução é reduzido, mas nem sempre com conservação da área. O reuso é uma otimização em que alguma parte da computação é compartilhada por alguns processos, como multiplicadores, somadores e subtratores. Essa transformação reduz a área total do circuito.

Assume-se que a técnica de clusterização comportamental tenta ir de encontro aos seguintes ojetivos, considerando que P_a e P_b processos; P_{aUb} denota o cluster formado pelos dois processos; $Area(P_a)$ é a área presumida para o processo a $Latency(P_a)$ é o tempo que o processo a leva para executar; $Throughput(P_a)$ é a taxa de dados que o processo a produz ou consome:

- Aperfeiçoar área se $Area(P_{aUb}) < Area(P_a) + Area(P_b)$;
- Aperfeiçoar latência se $Latency(P_{aUb}) < Latency(P_a) + Latency(P_b)$;
- Aperfeiçoar *throughput* se $Throughput(P_{aUb}) < Area(P_a)$ e $Throughput(P_{aUb}) < Area(P_b)$.

Porém, existem três limitações principais quando processos são agrupados em clusters: *(i)* distribuição do relógio, porque é preciso garantir que os processos de um mesmo cluster pertençam ao mesmo domínio de relógio; *(ii)* comunicação, porque o algoritmo tem maior impacto se todos os processos têm um grande volume de comunicação; *(iii)* escalabilidade, já que os blocos comportamentais englobam blocos maiores que o de uma síntese RTL.

Ferramentas disponíveis no mercado

Nesta Seção são apresentadas resumidamente as ferramenta: *Catapult C* da *Mentor Graphics*, *Agility Compiler* da *Celoxica* e *Cynthesizer* da *Forte Design System*.

A ferramenta Catapult C é a ferramenta comercializada pela *Mentor Graphics* para síntese comportamental a partir de uma descrição em C++ *não temporizada* para uma descrição RTL otimizada [18]. Essa ferramenta permite a exploração de múltiplas micro-arquiteturas e interfaces com uma qualidade superior a uma exploração manual. O resultado é garantido por um

fluxo que produz descrições RTL precisas a nível de ciclo de relógio ajustadas para uma tecnologia alvo. A ferramenta não exige que as interfaces do projeto da descrição comportamental sigam algum tipo de protocolo, como no Cynthesizer. Ela habilita análise do tipo *what-if* para que o projetista escolha interfaces como FIFO, *handshaking*, *single* ou *dual port*.

A ferramenta permite o controle do projetista sobre o processo de síntese através de restrições que são inseridas na descrição inicial. As restrições que podem ser aplicadas são: desdobramento de laços, *pipeline*, fusão de laços, mapeamento para RAM, ROM e FIFO, alocação de recursos e compartilhamento e utilização simplificada de recursos de memória.

A ferramenta permite que um *testbench* escrito em C++ possa ser reutilizado para verificar o RTL (Verilog ou VHDL) gerado pelo processo de síntese. Esse processo é automatizado através da geração dos *wrappers* e *transatores* que permitem a cossimulação de todo o projeto.

A ferramenta *Agility Compiler* [6], comercializada pela *Celoxica*, é um compilador para síntese comportamental que aceita como entrada descrições TL (*Transaction Level Modeling*) em SystemC. Essa ferramenta gera automaticamente código RTL otimizado para a síntese RTL com a ferramenta *Design Compiler* da *Synopsys* e *netlists EDIF* para FPGAs da *Altera, Actel* e *Xilinx*. Essa ferramenta permite explorar algoritmos complexos e micro-arquiteturas nos passos iniciais do fluxo de projeto. Permite também analisar o desempenho e fazer estimativas de *temporização* antes da prototipação ou verificação do ASIC fabricado.

A ferramenta *Cynthesizer* da *Forte Design Systems* transforma uma implementação comportamental SystemC em uma descrição RTL Verilog ou SystemC. Partindo de um modelo de alto nível sem temporização descrito em SystemC, essa ferramenta constrói micro-arquiteturas RTL temporizadas baseadas em um conjunto de diretivas especificadas pelo projetista. Essas diretivas podem ser relacionadas à latência de operação, paralelismo temporal (pipeline), área, desenrolamento de laços, etc. Essa ferramenta também disponibiliza um processo eficiente de verificação e simulação das micro-arquiteturas obtidas. O mesmo *testbench* SystemC pode ser utilizado em todas as fases do projeto.

Comparação entre as Ferramentas Apresentadas

Nesta Seção é apresentado um quadro contendo uma comparação inicial das ferramentas de síntese comportamental modernas comerciais. O resumo desta comparação aparece na Tabela 3.

2.4.5 Comparativo entre as Gerações de Síntese Comportamental

Esta Seção apresenta uma comparação entre as gerações de ferramentas de síntese comportamental conforme proposto em [7]. A Tabela 4 apresenta a contribuição de cada geração, as dificuldades de seu emprego e ferramentas representativas da geração.

Ferramenta	Vendedor	Entrada	Saída
CatapultC	Mentor Graphics	C++ não temporizado	RTL Otimizado
Agility	Celoxica	SystemC TLM	RTL para o Design Compiler e EDIF para FPGAs
Cynthesizer	Forte Design Systems	SystemC comportamental	RTL: SystemC ou Verilog para ASICs ou FPGAs

Tabela 3 – Quadro comparativo de ferramentas de síntese comportamental modernas.

Geração	Contribuição	Dificuldades	Representante
Primeira	Definição das tarefas de síntese	Síntese RTL não era realidade na época	MIMOLA [29]
Segunda	Domínios de aplicação mais restritos e geração de arquiteturas com controle e caminho de dados	Dificuldade devido à falta de algoritmos de otimização para o processo de síntese	System Architect's Workbench [40]
Terceira	Escalonamentos mais eficientes e preocupação com o resultado físico do projeto	Tentativa de integrar o fluxo de projeto HLS com o fluxo de projeto RTL	VOTAN [41], Hiasynth [3]
Moderna	Melhorias nas técnicas de otimização, Síntese para FPGA	Qualidade dos resultados inferior aos que são obtidos na síntese RTL	Catapult C, Agility Compiler, Cynthesizer

Tabela 4 – Quadro comparativo de gerações de ferramentas de síntese comportamental.

3 Síntese Comportamental com a Ferramenta Cynthesizer

A necessidade de um método prático que permita gerenciar de maneira eficiente projetos de sistemas digitais de grande complexidade em um ciclo de desenvolvimento mais curto, faz com que a síntese em alto nível seja retomada como opção de processo de concepção de sistemas digitais. Neste capítulo, será apresentada em detalhe a funcionalidade da ferramenta *Cynthesizer* da *Forte Design Systems*. Essa ferramenta permite que o projeto de um sistema digital seja capturado em nível comportamental de maneira formal e refinado até atender restrições de desempenho e área. O método proposto pelo *Cynthesizer* habilita a exploração do espaço de soluções de projeto aliado à síntese de alto nível e verificação baseada na utilização da linguagem C++ e de SystemC. A ferramenta *Cynthesizer* disponibiliza a automação do processo de transformação de algoritmos em alto nível em descrições RTL otimizadas incluindo síntese, verificação e simulação, permitindo investigar múltiplas implementações RTL. Essa investigação é baseada no uso de diretivas sem a modificação da descrição original do projeto. A ferramenta inclui: alocação de recursos, temporização, escalonamento de operações, particionamento de *datapath* e blocos de controle, implementação de máquinas de estados e tradução de SystemC para Verilog. O ambiente do *Cynthesizer* contém ainda ferramentas de automação de geração de *Makefiles*, configuração do ambiente de desenvolvimento, integração com outras ferramentas para síntese para ASIC ou FPGA e simulação HDL.

A Figura 12 mostra onde a ferramenta pode ser encaixada em um fluxo de projeto. Primeiramente, é realizada uma etapa de *projeto em nível de sistema*, ESL (do inglês, *Electronic System Level Design*) em que são especificados os requisitos funcionais do sistema. Isso é apoiado pela escolha de uma plataforma (ASIC ou FPGA) e pelo particionamento das funcionalidades em *hardware/software*. Na fase seguinte, de *especificação*, a plataforma é especificada em detalhes e as funcionalidades resultantes do particionamento anterior são descritas em especificações executáveis. Na fase de *implementação*, a especificação do *software* será refinada, otimizada e compilada sobre uma base de sistema operacional, tipicamente de tempo real (em inglês, *Real Time Operating System* ou RTOS). A especificação de *hardware* será refinada, otimizada e sintetizada, obtendo-se, ao final do processo, a descrição física dessa partição. Na fase final, de *validação*, o sistema é integrado à plataforma e validado. A ferramenta *Cynthesizer* é encaixada na fase de especificação e implementação da partição de *hardware* do sistema. O digrama do fluxo da ferramenta (Figura 12) mostra, simplificadamente: (*i*) a descrição inicial em SystemC do *hardware* do sistema, com sua respectiva simulação comportamental; (*ii*) a geração da descrição RTL pela ferramenta com a utilização de uma biblioteca de tecnologia, que deve ter relação com a plataforma escolhida para o sistema, seja essa baseada em ASIC ou FPGA;

(*iii*) A simulação funcional do código RTL obtido, com o mesmo *testbench* utilizado na simulação comportamental; (*iv*) a síntese lógica, com alguma ferramenta associada ao *Cynthesizer*(tal como o *Synplify-Pro* da *Symplicity*), do código RTL obtido, gerando um *netlist* de portas; (*v*) simulação em nível de portas do *netlist* final com o mesmo *testbench* utilizado nas simulações anteriores.

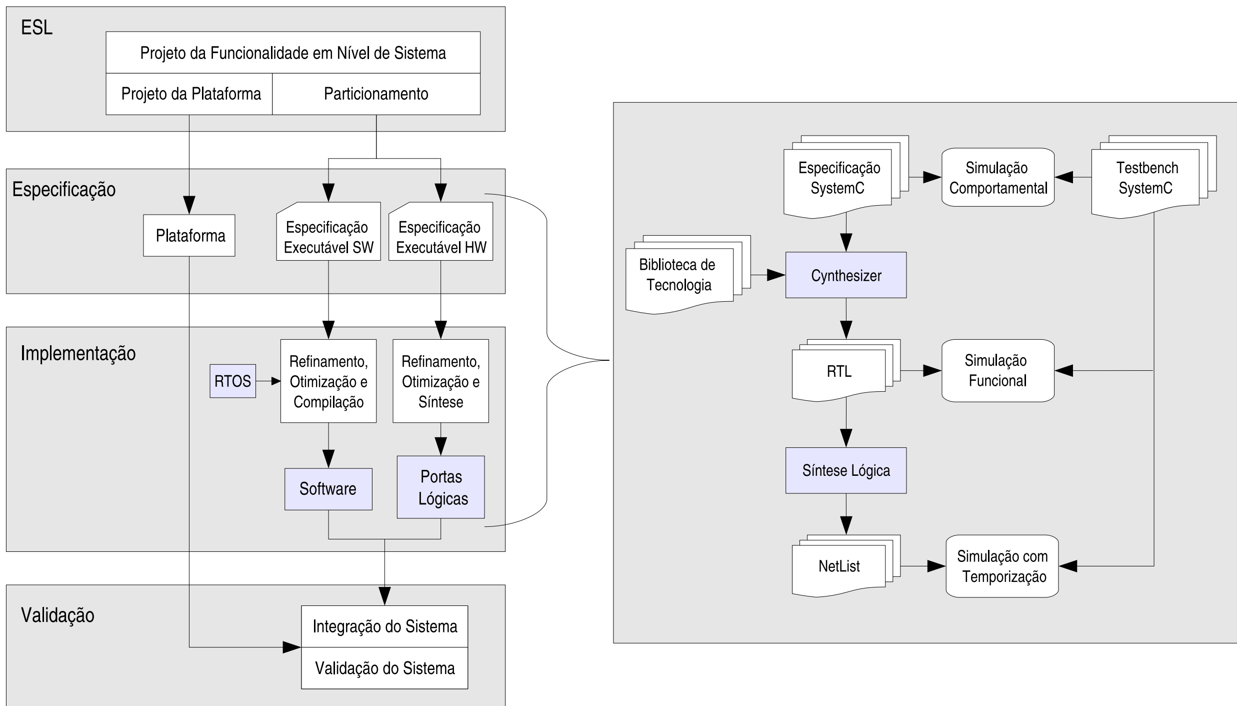

Figura 12 – Inserção da ferramenta Cynthesizer em um fluxo de projeto.

3.1 Estrutura da Ferramenta

O Cynthesizer é formado por dois componentes de software principais: o mecanismo de síntese comportamental e um ambiente chamado *Behavioral Design Workbench* (BDW) . O mecanismo de síntese comportamental é realizado por duas ferramentas: *cynthHL*, que faz a síntese da descrição comportamental C++/SystemC em uma descrição RTL C++; e *cynthVLG*, que transforma o código RTL C++ em RTL Verilog. O BDW possui ferramentas para automação de geração de *Makefiles*, configuração do ambiente de desenvolvimento, integração com ferramentas síntese para ASIC e FPGA, depurador de código, gerador de componentes de memória e simulação HDL.

A Figura 13 apresenta a relação entre os componentes do Cynthesizer. O BDW é responsável por comandar todo o processo. O projetista precisa definir a especificação comportamental e suas diretivas de síntese, os arquivos de controle de simulação (*testbenchs*) e o arquivo de regras

de projeto (*project.tcl*). O arquivo de regras de projeto será utilizado pelo BDW para construir todos os *Makefiles* para todos os processos executados pela ferramenta. O BDW automatiza a caracterização da biblioteca de tecnologia e a sua associação na transcrição da especificação comportamental para RTL. Essa transcrição é realizada pelas ferramentas *cynthHL* e *cynthVLG*, que são coordenadas pelo BDW com auxílio de compilador C++/SystemC. Esse ambiente automatiza a simulação de todas as descrições, integrando algum simulador (tal como o *Modelsim* da Mentor) quando necessário, e gerando os *wrappers* necessários para simulação do código Verilog RTL. A simulação da descrição comportamental e a da descrição RTL SystemC gerada é realizada pelo próprio ambiente. O ambiente também faz a integração com ferramentas de síntese lógica disponíveis no mercado, para ASIC ou FPGA, disponibilizando também a simulação para os resultados desses processos de síntese. A ferramenta disponibiliza também a geração de relatórios de todos os processos de síntese em páginas HTML.

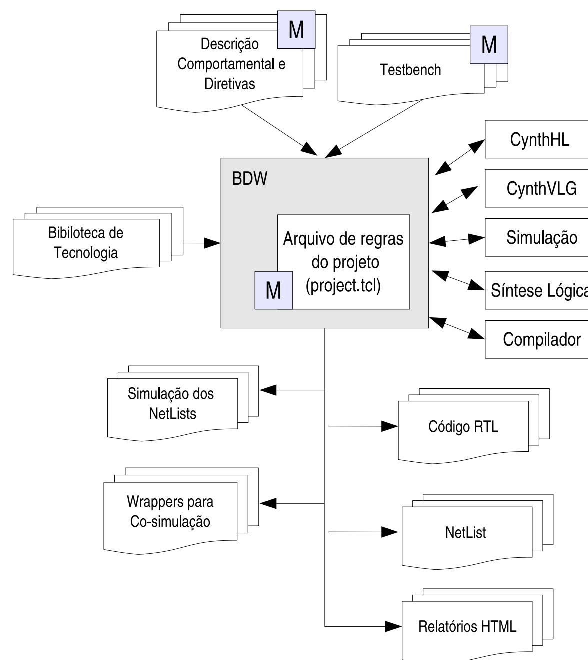

Figura 13 – Integração dos componentes do Cynthesizer. Módulos marcados com M são descritos manualmente pelo projetista.

3.1.1 Fases do Processo de Síntese Comportamental

O ambiente *Cynthesizer* executa vários processos durante a síntese comportamental. Primeiramente, é realizada uma *compilação* da descrição comportamental SystemC para garantir que nenhum erro de sintaxe existe no código fonte do projeto. Após a compilação, a ferramenta realiza a *análise do fluxo de dados* do projeto. Nesse passo determina-se como os dados de entrada serão conduzidos através das operações *datapath* para implementar a funcionalidade da

descrição inicial. Esta análise determina todos os operadores (de forma independente de tecnologia), multiplexadores e a lógica de controle necessários para a implementação do projeto. A fase seguinte é a *alocação*, onde cada operador identificado no passo anterior é mapeado para um operador da biblioteca de tecnologia. Nessa fase, são extraídas informações de temporização para cada operador da biblioteca. Na fase seguinte, de *escalonamento*, cada operador alocado na fase anterior recebe um ciclo de relógio específico onde será executado, baseado no fluxo do projeto e em seu atraso relativo. Durante o escalonamento introduz-se paralelismo na execução e as diretivas definidas para latência e *throughput* são utilizadas para derivar o escalonamento final. Todos os operadores alocados são mapeados para uma descrição RTL na fase de *vinculação* e a saída resultante é um código RTL, SystemC ou Verilog.

Simulação no Ambiente Cynthesizer

A ferramenta disponibiliza uma metodologia para simulações, onde o mesmo *testbench* é utilizado para verificar a implementação comportamental e todas as implementações derivadas pela ferramenta, conforme apresentado na Figura 14. As simulações podem ser feitas pelo próprio ambiente, no caso da verificação funcional da especificação comportamental, ou por ferramentas integradas como o *Modelsim* da Mentor, no caso verificação funcional do resultado derivado da Síntese Comportamental e da verificação com temporização dos resultados da Síntese Lógica para ASIC ou FPGA.

Para que o mesmo *testbench* possa ser aproveitado, é preciso que o projetista especifique protocolos de sincronização para cada sinal de interface. A Figura 27 apresenta um exemplo de comunicação entre *threads* de uma descrição comportamental, com uma *thread* produzindo dados (Figura 27(a)) e a outra consumindo (Figura 27(b)). Em (a), a *thread* informa que tem um dado através do sinal tx, assinalando-o com 1, disponibiliza o dado no sinal $data$ e espera enquanto o sinal ack_tx, escrito pela *thread* consumidora, é igual a 0. Quando ack_tx for igual a 1 a *thread* produtora assinala tx com 0. Em (b), a *thread* consumidora espera enquanto tx é igual a 0. Quando tx passa a ser 1 o dado do sinal $data$ é consumido e o sinal ack_tx é mantido por um ciclo no valor 1 para concluir a sincronização.

3.2 Estrutura de um Projeto no Ambiente Cynthesizer

Um projeto aceito pelo ambiente é composto, basicamente, pelos arquivos listados na Tabela 5. O projetista é responsável pela criação dos arquivos da descrição comportamental e suas diretivas (*<modulo>.cc*, *<modulo>.h* e *<modulo>_directives.h*), pelos arquivos que compõe o *testbench* (*tb.h* e *tb.cc*), pelo arquivo de regras do projeto (*project.tcl*) e pelo *Makefile* principal de automação do projeto. São gerados, de maneira automatizada pela própria ferramenta, os arquivos de instanciação dos módulos do sistema (*system.h* e *system.cc*), o arquivo que instancia

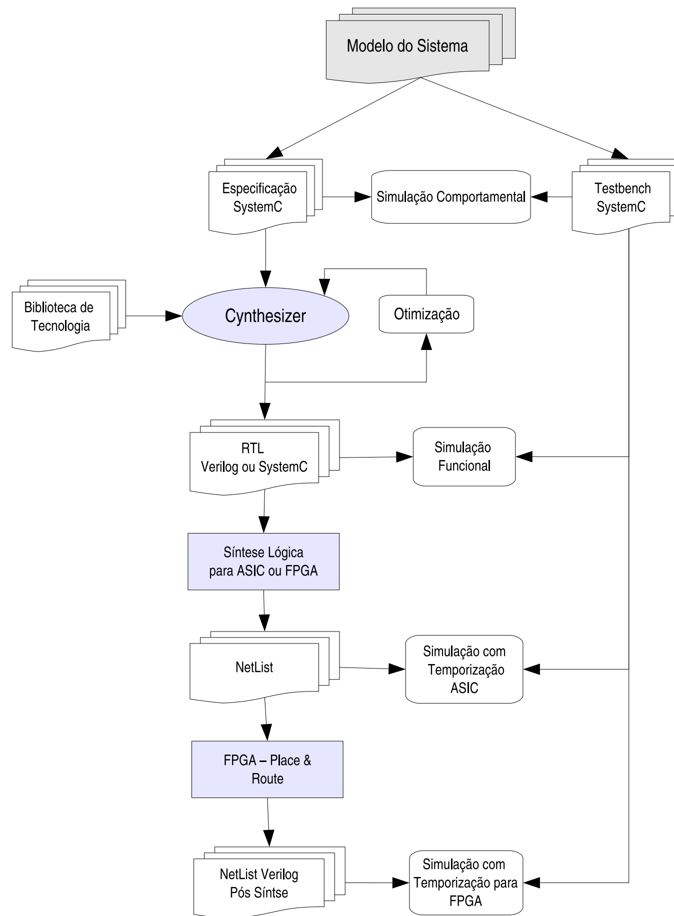

Figura 14 – Processo de verificação disponibilizado pelo Cynthesizer. O mesmo *testbench* é utilizado para simular a especificação inicial e todas as especificações derivadas do processo de síntese, tanto para ASICs quanto para FPGAs [39].

o sistema como um todo (*main.cc*) e o arquivo *makefile* que automatiza todo o processo da ferramenta (*Makefile.prj*).

Arquivo	Descrição	Geração
<modulo>.cc	Descrição do comportamento do projeto	Manual
<modulo>.h	Declarações referentes ao módulo do projeto	Manual
<modulo>_directives.h	Diretivas de síntese referentes ao projeto	Manual
tb.cc	Funções de produção e consumo de sinais	Manual
tb.h	Interface para simulação do projeto	Manual
system.cc	Ligação entre o módulo e o testbench	Automatizada
system.h	Instanciação dos módulos do sistema	Automatizada
main.cc	Instancia o sistema como um todo	Automatizada
project.tcl	Arquivos de regras do projeto	Manual
Makefile	Makefile principal	Manual
Makefile.prj	Makefile de regras do projeto	Automatizada

Tabela 5 – Lista dos pricipais arquivos de um projeto no ambiente Cynthesizer [39].

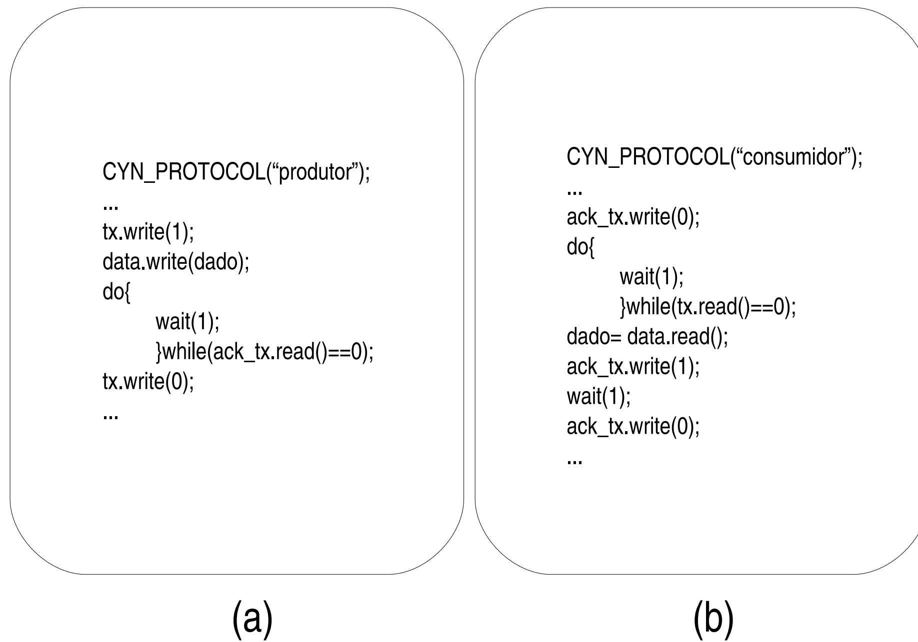

Figura 15 – Exemplo de interface de comunicação entre duas *threads*: (a) produzindo dados; (b) consumindo dados.

Esses arquivos se relacionam conforme apresentado na Figura 16. O módulo de projeto, seu *testbench* e todos os sinais de interface entre eles são instanciados nos arquivos de sistema (*system.cc* e *system.h*). O sistema como um todo é instanciado no arquivo *main.cc*. Todos eles são descritos em SystemC/C++. Os arquivos de regras de projeto e diretivas são scripts declarativos escritos em TCL. No arquivo *project.tcl* são especificados os arquivos do projeto, a tecnologia alvo, os arquivos de diretivas de síntese, as configurações da simulação e da síntese, etc. No arquivo de diretivas, estão as definições de síntese utilizadas em cada uma das configurações de exploração de espaço de projeto aceitas pela ferramenta.

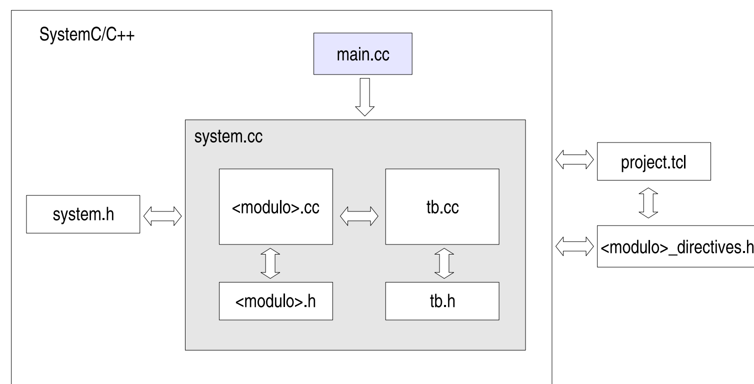

Figura 16 – Organização dos arquivos de projeto no ambiente Cynthesizer.

3.2.1 Exemplo de Projeto

Nesta Seção será apresentado um exemplo simplificado, baseado no apresentado por Ober [33], da elaboração de uma especificação comportamental aceita pelo Cynthesizer, através da descrição de um módulo que calcula o cubo de um número.

Primeiramente, é necessária atenção à organização do código da descrição comportamental do projeto. Um módulo do projeto deve ser descrito em SystemC, representado como um SC_MODULE com uma ou mais *threads* do tipo SC_CTHREAD. Esse módulo tem obrigatoriamente uma porta para o sinal de relógio e outra para o sinal de *reset*. A descrição do comportamento engloba as interfaces de entrada e saída do módulo, que devem ser descritas a nível de ciclo de relógio e com protocolo de sincronização explícito. A descrição do comportamento em si deve ser na forma de um algoritmo não temporizado, conforme explicado na Seção 3.1.1. A Figura 17 apresenta a organização da descrição de um módulo de projeto.

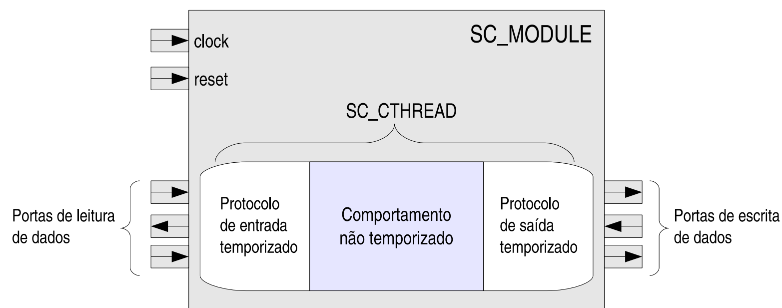

Figura 17 – Organização de um módulo de projeto em uma descrição comportamental.

Para o projetista, é disponibilizado um esqueleto básico do conteúdo dos arquivos necessários para o projeto. A primeira ação do projetista deve ser a definição de todas as portas do módulo. Assim, poderão ser geradas de maneira automatizada, com um *script* em *Perl* disponibilizado pela ferramenta, a declaração das portas de entrada e saída nos arquivos da descrição do módulo, no *testbench* e nos arquivos de sistema (*system.h* e *system.cc*). A partir de então, o projetista pode iniciar a descrição comportamental e demais tarefas sob sua responsabilidade.

Uma biblioteca de tecnologia deve ser utilizada em conjunção com as ferramentas de síntese comportamental. Para o exemplo, foi escolhida a *vtvtlib25*, desenvolvida e mantida pelo grupo Virginia Tech VLSI for Telecommunication [38] [37] para a tecnologia TSMC $0.25\ \mu$, com valor de lambda igual a $0,12\mu$m. Esta biblioteca é compatível com as regras de projeto da MOSIS (SCN5M-DEEP) para a fabricação junto à fundição TSMC. A biblioteca deve ser previamente caracterizada para a síntese comportamental. A ferramenta Cynthesizer pode assim realizar as estimativas de área e desempenho baseado nas características extraídas dos elementos da biblioteca que serão alocados para a geração da micro-arquitetura do projeto.

A Figura 18 (a) e (b) apresentam, respectivamente, os arquivos *cubo.h* e *cubo.cc*, que descrevem o módulo de exemplo desta Seção. Em *cubo.h* é feita a declaração do módulo, a declaração

de seus sinais de entrada, da *thread* de comportamento e do construtor do módulo. Os sinais declarados são: *clk*, que é o sinal de relógio; *rst*, sinal de *reset* do módulo; *in*, sinal de entrada de oito bits; e *out*, sinal de saída de 24 bits. Para os sinais *in* e *out* foram adicionados mais dois sinais que são necessários para a sincronização na comunicação com o módulo. Para a porta de entrada *in*: *in_rdy* assinala quando o módulo está pronto para receber dados e *in_vld* assinala quando o dado na entrada é um dado válido. Para a porta de saída *out*: *out_rdy* assinala quando o parceiro de comunicação do módulo está pronto para receber um dado e *out_vld* quando o módulo tem um dado válido em sua saída.

Em *cubo.cc*, o comportamento do módulo é descrito na thread *operacao()*. Ela contém os *blocos de protocolo*, que descrevem a comunicação e a sincronização dos sinais das interfaces, e a descrição do algoritmo funcional do módulo. Primeiramente, é descrito o comportamento do módulo quando o sinal de reset (*rst*) é ativado. Esse comportamento é expresso por um *bloco de protocolo* e mostra quais sinais são escritos durante o reset (Figura 18 (b) item 1). Após, é descrito o laço infinito que descreve a operação realizada pelo módulo. Nesse laço, inicialmente, é apresentado o comportamento da porta de entrada do módulo, também descrita como um bloco de protocolo preciso a nível de ciclo (Figura 18 (b) item 2). Após, é descrito o algoritmo da operação do módulo (Figura 18 (b) item 3). Esse algoritmo não é temporizado, nem descreve uma máquina de estados finitos, ficando livre para o escalonamento realizado pela ferramenta. Por fim, o laço contém a descrição da interface de saída do módulo, com a devida sincronização descrita também em um protocolo (Figura 18 (b) item 4).

Quando o módulo *cubo* começa sua operação, o sinal de *in_rdy* é ativado ($in_rdy = 1$). Com isso, seu parceiro de comunicação, que pode ser outro módulo ou um *testbench*, é informado de que a especificação já está pronta para receber os dados. O algoritmo espera até que o parceiro na comunicação assinale *in_vld*, indicando que o valor apresentado na porta de entrada é válido. Quando isto ocorre, o módulo lê a entrada, desativa o sinal *in_rdy* para informar ao parceiro que não pode receber dados e processa a operação de cálculo do cubo. Quando acaba de processar, o algoritmo espera que sinal *out_rdy* seja ativado pelo parceiro, indicando que o mesmo está pronto para receber os dados. Quando isso ocorre, o módulo escreve os resultados na porta de saída e assinala *out_vld*, para informar que está apresentando dados válidos nessa interface. Após isso, espera por um ciclo de relógio e baixa *out_vld* para que possa ser iniciada iniciada uma nova execução.

Ainda é preciso descrever o arquivo de projeto, com os detalhes necessários para a automatização do processo. A Figura 19 apresenta um exemplo desse arquivo para o projeto do módulo cubo. É necessário definir: uma biblioteca de síntese, *cynthLib*; uma biblioteca de tecnologia, *techLib*; opções para o Compilador GCC, *ccOptions*; opções globais para a transcrição do projeto em RTL, *cynthHLoptions*; opções de simulação, como o simulador para Verilog, o tipo de *log* de simulação gerado, o tempo de início da simulação, etc; os módulos não sintetizáveis, *systemModule*; os módulos sintetizáveis, *cynthModule*, e suas respectivas configurações de exploração de espaço de projeto, *cynthConfigs*; e as configurações de simulação, *simConfig*.

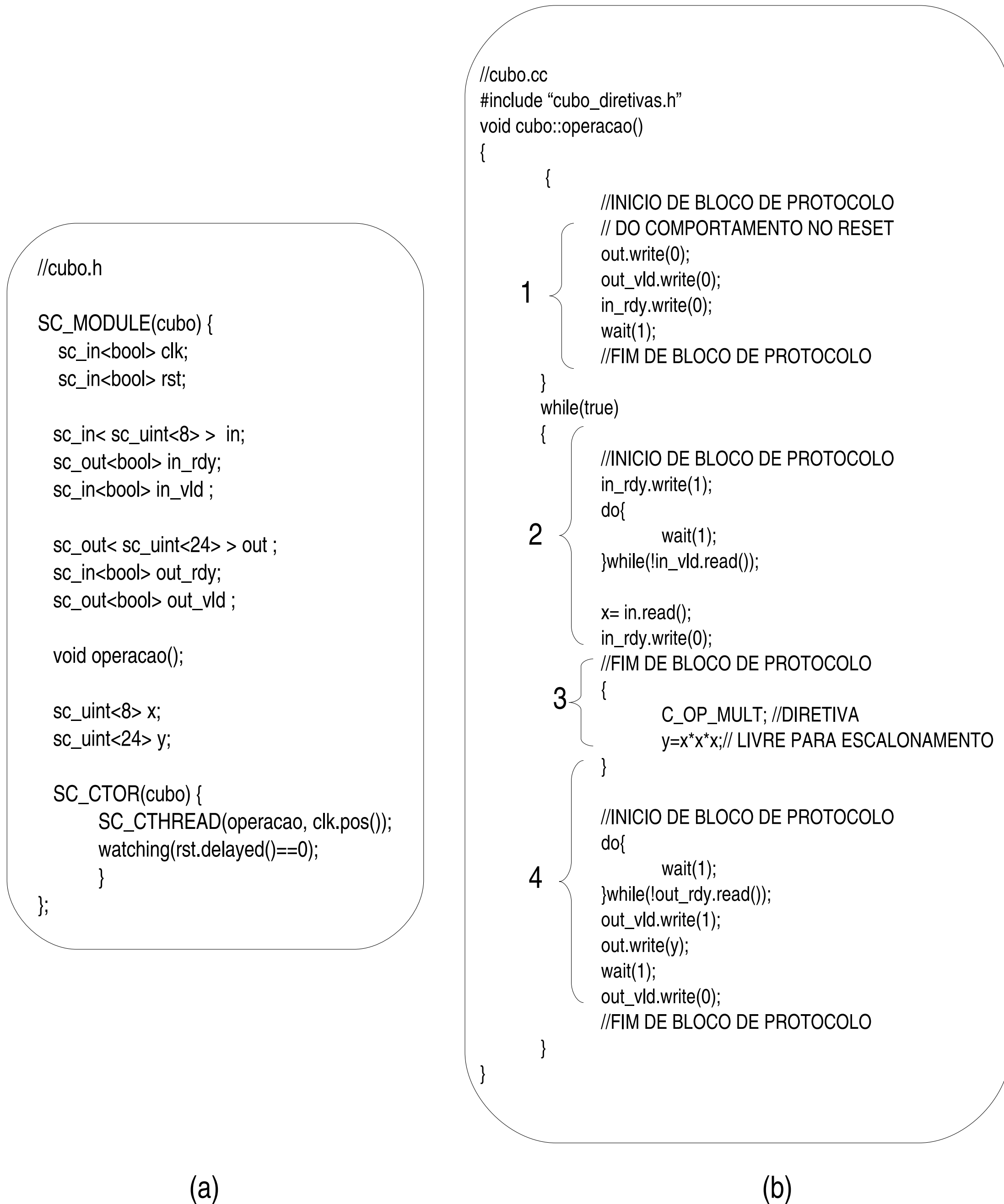

Figura 18 – Exemplo de descrição comportamental: (a) arquivo cubo.h, que contém a declaração do módulo, seus sinais, *threads* e seu construtor. (b) arquivo cubo.cc, com a implementação do comportamento do módulo.

Outro aspecto importante é a definição das diretivas de projeto, que irão habilitar a exploração do espaço de soluções do projeto. Elas podem ser inseridas diretamente no código ou encapsuladas em macros. No Cynthesizer são permitidas as seguintes diretivas:

- CYN_FLATTEN - que permite que os *arrays* sejam implementados como registradores individuais ao invés de serem implementados como memórias.
- CYN_LATENCY - determina uma latência máxima e mínima em ciclos de relógio para

```
# BDW Project File

# Biblioteca de Tecnologia
cynthLib   "./techlib"
techLib    "techlib/vtvtlib25.db"

# Opções do compilador GCC
 ccOptions      "-DCLOCK_PERIOD=10000.0"

# Opções globais para geração de código RTL
cynthHLOptions "--clock_period=10000.0 --chain=on"

# Opções para a simulação
verilogSimulator mti
startTime       5
endOfSimCommand  "\@make --no-print-directory compare"
logOptions vcd

# Testbench e módulos de sistema
systemModule main.cc system.cc tb.cc

# SC_MODULES que serão sintetizados e suas configurações de
# exploração de espaço de projeto

cynthModule cubo cubo.cc \
   [cynthConfigs {
      BASIC
      LATENCY
   }] \
   [vlogFiles {}]

# Configurações para a simulação
simConfig BEH     { cubo BEH   BASIC }
simConfig BASIC_C { cubo RTL_C BASIC }
simConfig BASIC_V { cubo RTL_V BASIC }
simConfig LATENCY_C {cubo  RTL_C LATENCY}
simConfig LATENCY_V {cubo RTL_V LATENCY}
```

Figura 19 – Arquivo project.tcl para o projeto exemplo dessa Seção.

o escalonamento de certas operações do projeto.

- CYN_UNROLL - desenrola os laços, para que mais operações sejam realizadas em paralelo.
- CYN_PIPELINE - transforma os laços em operações *pipeline*.
- CYN_BASIC - é utilizada por omissão, quando nenhuma diretiva é definida.

Como exemplo da utilização de diretivas, considerando $TAPS$ como uma constante previamente definida, pode-se utilizar CYN_UNROLL para desenrolar um laço *for*:

```
for(i=0; i<TAPS; i++){
```

```
CYN_UNROLL(COMPLETE, TAPS, "loop_for");
...
}
```

Assim, o laço é desenrolado e todas as suas iterações ocorrem em paralelo. Para melhor modularização da implementação, a diretiva pode ser encapsulada com definição de uma macro, que deve ser incluída no arquivo de diretivas de projeto:

```
#define   C_FOR_LOOP  CYN_UNROLL(COMPLETE, TAPS, "loop_for")//macro

for(i=0; i<TAPS; i++){
    C_FOR_LOOP;
  ...
  }
```

Para o módulo exemplo, o arquivo de diretivas com a definição das macros de síntese é apresentado na Figura 20.

```
#ifndef CUBO_DIRECTIVES_INC
#define CUBO_DIRECTIVES_INC

#if defined LATENCY
#     define  C_OP_MULT              CYN_LATENCY(0,1,"operação de multiplicação")
#else /* BASIC  */
#     define  C_OP_MULT

#endif /*  */

#endif /*  */
```

Figura 20 – Arquivo cubo_diretivas.h, que contém a definição das diretivas de síntese do projeto para as configurações LATENCY e BASIC, que foram especificadas no arquivo project.tcl.

Com a especificação do módulo pronta, é preciso criar o *testbench* para as simulações de verificação do módulo. É necessário dar atenção às interfaces, que têm de ter os sinais complementares aos do módulo a ser testado. O *testbench* deve implementar as *threads* produtoras e consumidoras de estímulos, com a interface sincronizada da mesma maneira que as interfaces do módulo.

Ainda, é necessário criar o arquivo *Makefile* principal do projeto. A ferramenta oferece um arquivo pronto, que pode ser alterado pelo projetista. Nesse *Makefile* são especificadas as ações tomadas em relação aos resultados das simulações. Dependendo do tipo de projeto, os

resultados das simulações podem ser comparados aos resultados esperados para aqueles dados que foram entrada da simulação.

Nesse ponto, toda a especificação a cargo do projetista está pronta e pode ser iniciado o processo de verificação e síntese. A ferramenta não oferece interface gráfica e todos os comandos devem ser disparados manualmente através de um *shell*. Os comandos estão especificados no *Makefile.prj*. O primeiro passo é gerar esse arquivo com o o comando *bdw_makegen project.tcl*. Os comandos são derivados dos nomes dados às *simConfigs* no arquivo *project.tcl*, do tipo *<operação>_<nome da configuração>*. São comandos de simulação resultantes: *sim_BEH*, simulação comportamental; *sim_BASIC_C*, para a simulação do RTL SystemC da configuração BASIC_C; *sim_BASIC_V*, para a simulação do RTL Verilog da configuração BASIC_V; *sim_LATENCY_C*, para simular o RTL SystemC da configuração LATENCY; e *sim_LATENCY_V*, para simular o RTL Verilog da configuração LATENCY. Os comandos de síntese são do tipo *cynth_<nome da configuração>*.

A partir de então, o projetista pode iniciar o processo de verificação e síntese da descrição comportamental. Para o projeto exemplo, a verificação da descrição comportamental é realizada com a utilização do comando *make sim_BEH*, que realiza a compilação, vinculação, e execução da simulação da descrição. Caso erros de sintaxe e semântica sejam encontrados, ou a simulação não ocorra da forma esperada o projetista deve modificar o código da descrição ou *testbench* para acertar os eventuais erros.

Se a simulação comportamental foi concluída com sucesso, o projetista pode iniciar a síntese. Para esse projeto, existem quatro opções: síntese para RTL SystemC e Verilog para a configuração BASIC; e síntese para RTL SystemC e Verilog para a configuração LATENCY. Para exemplificar, realiza-se a síntese RTL SystemC para BASIC e LATENCY, respectivamente, com os comandos *make cynth_BASIC_C* e *make cynth_LATENCY_C*. Pode-se então gerar um relatório preliminiar de área e demais informações de síntese para o projeto na biblioteca de tecnologia escolhida. A Figura 21 apresenta o relatório das estimativas de área iniciais para as duas configurações de síntese.

Synthesis Report

cynthModule cubo

cynthConfig name	last run	area (square µm)				register bits	total run time
		combinational	memory	register	total		
BASIC	Nov-22 16:23	119141.3	0.0	24984.3	144125.6	85	00:00:23
LATENCY	Nov-22 16:26	122064.4	0.0	20281.4	142345.7	69	00:00:23

Figura 21 – Relatório preliminar com as estimativas iniciais baseadas na biblioteca de tecnologia caracterizada para o projeto (*vtvtlib25*). As áreas são apresentadas em microns quadrados (μm^2).

Após a síntese, é possível realizar a simulação dos códigos RTL SystemC gerados pela ferramenta, para as duas configurações. Com os comandos *sim_BASIC_C* e *sim_LATENCY_C*, realiza-se a simulação para as configurações geradas, respectivamente, BASIC e LATENCY.

No arquivo *project.tcl* foi definido que o *log* de simulação seria do formato VCD (*logOptions vcd*), o que permite que a simulação seja exibida por alguma ferramenta gráfica. A simulação do RTL SystemC da configuração BASIC é apresentada na Figura 22 e a da RTL SystemC da configuração LATENCY é apresentada na Figura 23. Na configuração BASIC, a operação completa demora 5 ciclos para completar e na configuração LATENCY um ciclo a menos é consumido para realizar a operação.

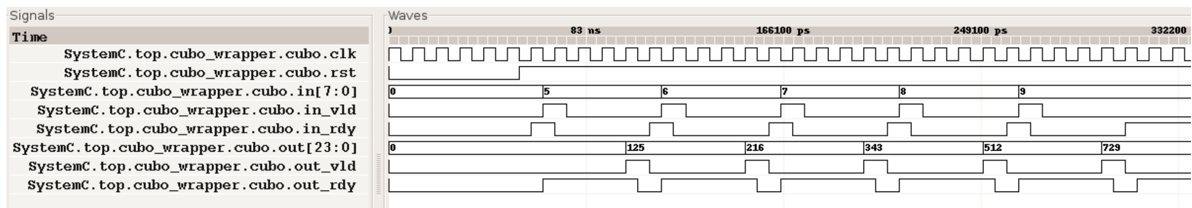

Figura 22 – Visualização da simulação do código RTL SystemC da configuração BASIC. A cada 5 ciclos uma operação é realizada.

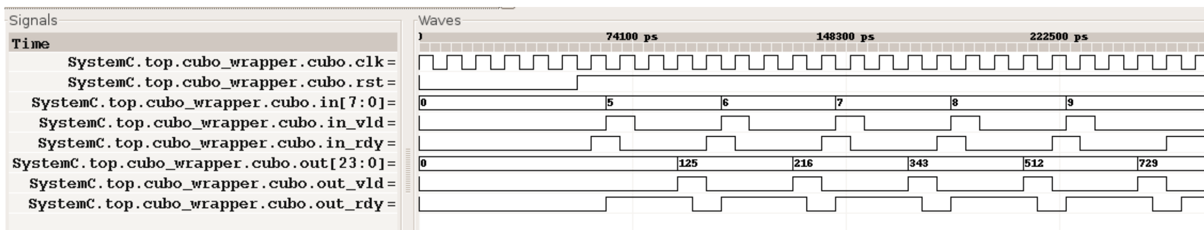

Figura 23 – Visualização da simulação do código RTL SystemC da configuração LATENCY. A cada 4 ciclos uma operação é realizada.

3.3 Integração do Cynthesizer com Ferramentas Disponíveis no Mercado.

O fluxo de projeto oferecido pelo Cynthesizer operar com diversas ferramentas integradas integradas a ele. A Tabela 6 apresenta as ferramentas de diversos fornecedores e um descrição de sua utilização.

Neste trabalho as ferramentas utilizadas em conjunto com o Cynthesizer foram ModelSim, para as simulações RTL e Synplify Pro, para a síntese de FPGA.

Fornecedor	Produto	Descrição
Altera	Quartus II	Síntese para FPGA
Atrenta	Spyglass	RTL Checker
Axiom	Designer	Visualizador de esquemáticos e sinais
Cadence	NC-Verilog	Simulação RTL Verilog
Cadence	BuildGates	Síntese Física
Cadence	Conformal	RTL-to-Gate Equivalence Checker
Calypto	SLEC	Behavioral-to-RTL Equivalence Checker
CoWare	ConvergenSC	Ambiente de desenvolvimento com simulador Systemc
Mentor	ModelSim	Simulação RTL
Novas	Debussy	Visualizador de esquemáticos e sinais
Open Source	OSCI	Simulação SystemC
Open Source	Insight	Depurador HDL
Sequence	PowerTheater	Analisador de Potência RTL/Portas Lógicas
Summit	Vista	Ambiente de Desenvolvimento
Synopsys	VCS	Simulação RTL
Synopsys	Design Compiler	Síntese Física
Synplicity	Synplify ASIC	Síntese Física
Synplicity	Synplify Pro	Síntese para FPGA
Xilinx	ISE	Síntese para FPGA

Tabela 6 – Ferramentas que podem ser integradas ao ambiente Cynthesizer [39].

4 Codificação Comportamental

Este Capítulo apresenta uma breve comparação entre os estilos de codificação para as descrições Comportamental e RTL.

Conforme visto no Capítulo 2, Seção 2.3, a principal diferença entre as descrições RTL e Comportamental é a maneira como é realizada a *sincronização* do sistema. Em uma descrição Comportamental a sincronização dos módulos/processos/*threads* é realizada através de troca de mensagens. Assim, os eventos de entrada/saída têm aqui um papel fundamental. Em uma descrição RTL, os módulos/processos/*threads* sincronizam sua comunicação com um sinal de relógio. Todo o controle dessa atividade é realizada por máquinas de estados finitas (FSM). Porém, outras diferenças podem ser observadas entre descrições Comportamentais sintetizáveis e descrições RTL.

Em uma especificação RTL, independentemente da linguagem, cada processo deve estar confinado em ciclos de relógio. O projetista é responsável por decompor funcionalidades mais complexas, que custariam vários ciclos, em um conjunto de passos simples, que executam em poucos ciclos, quase sempre controlados por uma FSM. Em uma especificação Comportamental sintetizável com as ferramentas disponíveis atualmente, uma funcionalidade que custa vários ciclos pode ser descrita sem a necessidade de decomposição em passos. Isso torna a descrição mais simples e natural.

Um fator bastante importante para a comparação entre os estilos de codificação é o acesso à memória. A implementação do acesso à memória em descrições RTL é realizada através da construção explícita de FSMs. Os componentes de memória devem ser instanciados e o acesso deve ser controlado pela FSM, observando a interface oferecida pelo componente. Já em descrições comportamentais, isso é realizado de maneira mais intuitiva, como um acesso a um vetor, utilizando a construção de indexação da linguagem. O processo de Síntese Comportamental se encarrega de instanciar os elementos de memória, mapeando-os para componentes disponíveis nas bibliotecas, e de inferir a FSM de acesso a esses elementos.

Outra facilidade da codificação Comportamental é a liberdade na utilização de *laços*. Em SystemC, laços do tipo *while*, *do while* e *for* podem ser utilizados com índices dependentes e condições de parada com construções do tipo *break*. Ferramentas de síntese Comportamental como o Cynthesizer permitem que esses laços sejam desenrolados para obter paralelismo de execução. Para descrições RTL as ferramentas de síntese impõe restrições à utilização de laços. Exemplos dessas restrições são índices fixos nas iterações e o corpo do laço dever consumir apenas um ciclo de relógio. Essas condições permitem que as ferramentas possam inferir paralelismo com mais eficiência. Porém, obrigam o projetista a criar FSMs complexas para garantir

a operação do sistema.

O restante do Capítulo é organizado: a Seção 4.1 apresenta exemplos de implementações RTL e comportamental para o módulo de controle do roteador da NoC Hermes; a Seção 4.2 apresenta exemplos de acesso a elementos de memória em especificações comportamentais; a Seção 4.3 mostra como devem ser organizadas as interfaces de comunicação entre os processos de uma descrição comportamental; a Seção 4.4 trata de métodos para otimizar os resultados no *Cynthesizer*

4.1 FSMs e Funcionalidades Multiciclo

Um exemplo de especificação RTL em SystemC é o mostrado na Figura 24, que implementa o módulo de controle dos roteadores da rede Hermes. O módulo de controle é responsável por realizar o chaveamento entre as portas. No arquivo *controle.h* (Figura 24(a)) são declarados os sinais de interface do módulo, os sinais internos, a enumeração dos estados da FSM de controle e os métodos da classe. Na declaração do construtor da classe, cada método tem associada a sua lista de sensitividade.

No arquivo *controle.cc* (Figura 24(b)) são implementados os métodos da classe. Na Figura são apresentados, simplificadamente, apenas três dos métodos da classe: *upd_SC*, *upd_PSC* e *main_action*. O método *upd_SC* é responsável por informar ao método *main_action* o estado atual da operação da FSM. O método *upd_PSC* determina qual o próximo estado da FSM. E por fim, *main_action* realiza as decisões de roteamento, baseado no estado atual do processo, e no modelo de chaveamento utilizado (XY, *west-first*, *negative-first*, etc).

A versão comportamental do módulo controle do roteador da rede Hermes é apresentada na Figura 25. A codificação apresentada é aceita pela ferramenta Cynthesizer. Nessa implementação cada módulo do roteador é descrito como uma *thread*. A *thread t_roteamento* implementa a função de chaveamento das portas. Esse processo se comunica com a *thread* de arbitragem, responsável pela ordem do atendimento dos pedidos, e com as portas de saída, informando o estabelecimento e o fim das conexões entre as portas de entrada e saída do roteador.

Na descrição, para cada *thread* é necessário incluir o comportamento da mesma quando ocorre o *reset*, ou seja, as inicializações. No laço principal é incluída a inicialização das variáveis auxiliares para o algoritmo de roteamento. A comunicação com a arbitragem e com as portas de saída é realizada através de protocolos de comunicação. O processo de arbitragem informa se existe alguma requisição (sinal *req_rot*), para atender uma determinada porta (sinal *incoming*) e o flit de cabeçalho do pacote recebido (sinal *header*).

Sempre que existe alguma requisição, o algoritmo de roteamento é executado e a *thread* responde à requisição de arbitragem através do sinais *ack_rot_vld* e *ack_rot* que informam, respectivamente, se o resultado da tentativa de conexão está disponível e se foi possível realizar a conexão. O processo de controle, também informa às portas de saída sobre os estados de suas

```
//controle.h
SC_MODULE(controle){

  sc_in<bool > clock;
  sc_in<bool > reset_n;
 sc_in<bool > req_rot;
  sc_out<bool > ack_rot;
  sc_in<reg3 > dir_incoming;
  sc_in<regflit > header;
  sc_in<regNport > sender;
  sc_out<regNport > free_d;
  sc_out<reg_mux > mux_in;
  sc_out<reg_mux > mux_out;
  ...
  enum state_controle {SC0, SC1, SC2, SC3, SC4, SC5};
  sc_signal<state_controle > SC,PSC;

 sc_signal<reg3 > dirx, diry;
  sc_signal<regmetadeflit > lx, ly, tx, ty;
  sc_signal<regNport > auxfree;
  sc_signal<reg_mux > source;
  sc_signal<regNport > sender_ant;

  void upd_free();
  void upd_muxin();
  void upd_l();
  void upd_t();
  void upd_dir();
  void upd_SC();
  void upd_PSC();
  void main_action();

  ...
SC_CTOR(controle){
   ....
   SC_METHOD(upd_SC);
   sensitive_neg << reset_n << req_rot;
   sensitive_pos << clock;

   SC_METHOD(upd_PSC);
   sensitive << SC << req_rot;
   sensitive << lx << ly;
   sensitive << tx << ty;
   sensitive << auxfree << dirx << diry;

   SC_METHOD(main_action);
   sensitive_neg << clock << reset_n;
  }
```

```
//controle.cc
void controle::upd_SC(){
  if(reset_n.read()==false){ SC.write(SC0); }
  else if(req_rot.read()==false) { SC.write(SC0); }
  else { SC.write(PSC.read()); }
}
void controle::upd_PSC(){
  regNport localauxfree;
  localauxfree = auxfree.read();
  switch(SC.read()){
    case SC0:
      if(req_rot.read()==true) PSC.write(SC1);
      else PSC.write(SC0);
      break;
    case SC1:
      if(/*CONDIÇÃO 1*/.){
        PSC.write(SC2);
      }
      else if(/*CONDIÇÃO 2*/){
        PSC.write(SC3);
      }
      else if(/*CONDIÇÃO 3*/){
        PSC.write(SC4);
      }
      else{
        PSC.write(SC1);
      }
      break;
    case SC2:
      PSC.write(SC5);
      break;
    case SC3:
      PSC.write(SC5);
      break;
    case SC4:
      PSC.write(SC5);
      break;
    case SC5:
      PSC.write(SC5);
      break;
  }
}
void controle::main_action(){
...
  if(reset_n.read()==false){
  ...
  }
  else{
    if(req_rot.read()==true){
      switch(SC.read()){
        case SC2:
        ...
        case SC3:
          ...
        case SC4:
          ...
      }
    }
   ...
}
```

(a) (b)

Figura 24 – Exemplo de implementação RTL em SystemC: (a) arquivo controle.h da especificação do controle do roteador da rede Hermes; (b) arquivo controle.cc, que implementa o módulo de controle do roteador da rede Hermes.

tabelas de roteamento e sobre o estabelecimento de conexões.

```
//router.h
SC_MODULE(router)
{
 public:
  sc_in< bool >        clk;
  sc_in< bool >        rst;

  sc_in< sc_uint<16> >  address;
  sc_in< sc_uint<1> >   add_vld;
  sc_in< sc_uint<1> >   rx_o;
  sc_in< sc_uint<16> >  data_in_o;
  sc_out< sc_uint<1> >  ack_rx_o;
  ...
  /* Declaração dos demais sinais  e váriávies
   internos ao roteador*/

  //Tabelas de roteamento
  sc_uint<TAM_P> in[TAM_TAB], out[TAM_TAB];
  sc_uint<TAM_FREE> free_d[TAM_TAB];

  SC_CTOR(router) {

  CYN_FLATTEN(in);// diretiva
  CYN_FLATTEN(out);// diretiva
  CYN_FLATTEN(free_d);// diretiva

   ...
   SC_CTHREAD(t_roteamento, clk.pos());
   watching(rst.delayed() == false);

  }

 private:
  void t_roteamento();
  ...
};
```

(a)

```
//router.cc
void router::t_roteamento()
{
  reset_rot();

  while(1)
  {
    inicializacao_loop_main_rot();
    {
      CYN_PROTOCOL("REQ_ROT");
      rot_s=req_rot.read();
    }
    if(rot_s==1)
    {
      {
        CYN_PROTOCOL("REQ_ROT_HEADER_INCOMING");
        header_rot=header.read();
        inc=incoming.read();
      }

      d_rot=0;
      /*
       Algoritmo de decisão de roteamento
       d_rot recebe a informação: 1 se a conexão
       foi estabelecida, 0 se não.
      */
      {
        CYN_PROTOCOL("ACK_ROT");
        ack_rot.write(d_rot);
        ack_rot_vld.write(1);
        wait(1);
        ack_rot.write(0);
        ack_rot_vld.write(0);
      }
    }
    {
      CYN_PROTOCOL("TAB_ATUALIZA");
      /*
        Atualização da tabela de roteamento
        para as portas de saída.
      */
    }
  }
}
```

(b)

Figura 25 – Exemplo de implementação comportamental em SystemC: (a) arquivo router.h que descreve o roteador da rede Hermes. O módulo de controle é uma das *threads* do roteador; (b) implementação da *thread* de controle de roteamento.

4.2 Acesso a Memória em Especificações Comportamentais

Instanciar e utilizar elementos de memória em especificações comportamentais é mais simples do que em descrições RTL, já que não é preciso declarar explicitamente a FSM de controle de acesso e os componentes podem ser inferidos pelo próprio processo de síntese comportamental. Porém, o projetista pode decidir pelo reuso de componentes de memória, que podem ser descritos em nível comportamental, RTL ou *netlist*, e para isso precisa criar *wrappers* para que a

simulação seja possível. Isso vale também para qualquer componente que possa ser reutilizado no projeto.

A Figura 26 apresenta a instanciação e o acesso de elementos de memória em uma descrição comportamental. Os componentes MEM1, MEM2, e MEM3 são instâncias de *wrappers* de componentes de memória que foram reusados para essa descrição. O vetor MEM4 instanciado será mapeado pela síntese comportamental para uma memória disponível em alguma biblioteca caracterizada para o projeto, ou para uma memória genérica, derivada pela própria síntese comportamental. Ainda, a critério das diretivas utilizadas pelo projetista, o mapeamento pode ser feito para um conjunto de registradores, que podem ser acessados individualmente, de forma paralela.

O acesso aos elementos dessas memórias, independentemente se são componentes reusados, derivados da síntese ou um conjunto de registradores, é realizado através da indexação permitida na linguagem de captura. Ainda na Figura 26, o exemplo de acesso das memórias externas dentro de um laço *for* que percorre todas as posições de cada uma é $MEM1[i] = MEM2[i] + MEM3[i]$. A síntese comportamental irá derivar uma FSM de acesso que permita que cada iteração do laço de acesso seja realizada dentro de um número finito de ciclos de relógio. A Figura apresenta ainda uma operação com acesso a uma memória externa e um componente de memória derivado pela própria síntese comportamental. Dentro de um laço *for*, para a operação $MEM4[j] = MEM1[j] * x$ é gerada uma FSM para que cada iteração do laço aconteça em um número finito de ciclos de relógio.

4.3 Interfaces de Comunicação em Descrições Comportamentais

Em uma especificação comportamental, a troca de informações entre as *threads* que descrevem o comportamento de um módulo é sincronizada através de troca explícita de sinais e controle, conforme apresentado no Capítulo 3 Seção 3.1.1. Tais trocas inserem ciclos de relógio na especificação RTL derivada da síntese comportamental. A Figura 27 retoma o exemplo da Seção citada acima de comunicação entre *threads* de uma descrição comportamental, com uma *thread* produzindo dados (Figura 27(a)) e a outra consumindo (Figura 27(b)). Em (a), a *thread* informa que tem um dado através do sinal tx, atribuindo 1 a este, disponibiliza o dado no sinal $data$ e espera até que o sinal ack_tx, escrito pela *thread* consumidora, seja igual a 1. Quando ack_tx for igual a 1 a *thread* produtora assinala tx com 0. Em (b), a itthread consumidora espera enquanto tx for igual a 0. Quando tx passa a ser 1, o dado do sinal $data$ é consumido e o sinal ack_tx é mantido por um ciclo no valor 1, para concluir a sincronização.

O comportamento descrito no parágrafo anterior ocorre com o módulo *buffer* da porta de entrada dos roteadores implementados nesse trabalho. Esse módulo possui um controle de entrada, que recebe os *flits* da interface e os armazena na fila, e um controle de saída, que envia uma requisição para realizar conexão com alguma porta de saída e, após a conexão ser

```
SC_MODULE(M) {
  sc_in <bool> clk;
  sc_in <bool> reset;

  sc_in < sc_uint<32> >  op1;
  ...
  //Instâncias de memórias externas
  ramA_256x32::port MEM1;
  ramA_256x32::port MEM2;
  ramA_256x32::port MEM3;
  ...
  sc_uint<36> MEM4[256];//vetor que
  //pode ser desenrolado ou
  //mapeado automaticamente para
  //uma classe de memória

  sc_uint<32> i, j;
  ...
  SC_CTOR(M) {
  ...
  SC_CTHREAD(t,clk.pos());
 }
};
/**********************/

void t() {
 ...
 while (1) {
          ...
          //Acesso às memórias externas
          for(i=0; i< 256; i++)
          {
                   MEM1[i] = MEM2[i] + MEM3[i];
          }
          //Acesso ao vetor MEM4: memória ou array
          for(j=0; j< 256; j++)
          {
                   MEM4[j] = MEM1[j] * x;
          }
          ...
 }
}
```

Figura 26 – Exemplo de instanciação e acesso de componentes de memória em descrições comportamentais aceitas pelo Cynthesizer.

confirmada, retira os *flits* da fila. As duas interfaces desse módulo têm sincronização para troca de dados e com isso a inserção de ciclos adicionais pela síntese para garantir a precisão de seu funcionamento. A Figura 28 apresenta um detalhe do funcionamento do módulo *buffer*: (1): o parceiro da comunicação informa que tem um *flit* na interface ($rx = 1$) e disponibiliza o dado ($data_in$); (2) e (3): o controle da entrada do módulo *buffer* grava o dado na fila e confirma a operação ($ack_rx = 1$ por um ciclo); (4): o parceiro da comunicação confirma o fim da transação ($rx = 0$); (5): nova transmissão é iniciada; (6) e (7): quando o *buffer* tem dados na fila, pede uma conexão ($h = 1$) e espera até que a conexão seja atendida ($h = 0$); (8): após a conexão ser garantida, a interface de saída informa que tem dado à disposição ($data_av = 1$); (9) e (10): confirmação da conclusão da troca do *flit*; e (11): nova transmissão é iniciada na interface de saída do *buffer*.

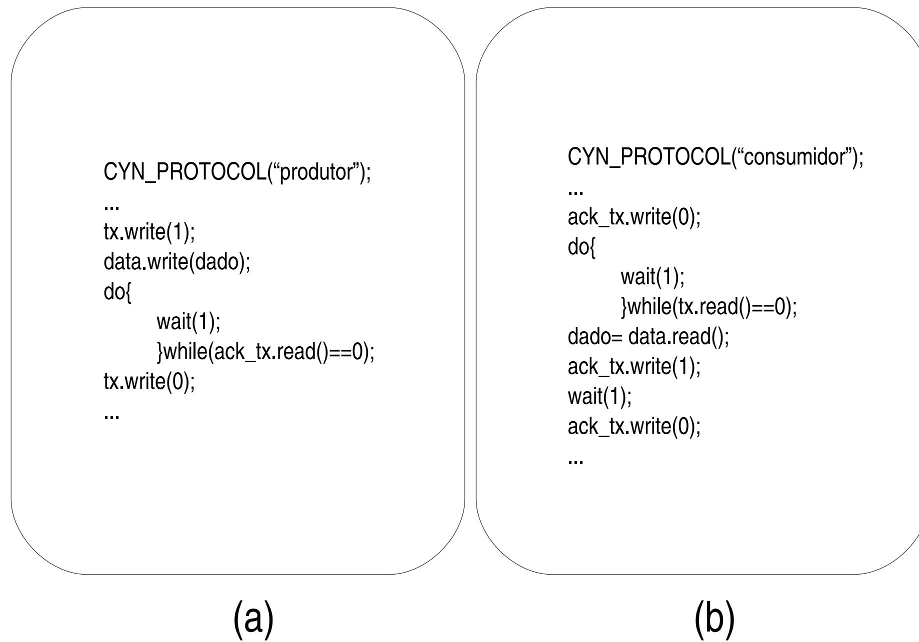

Figura 27 – Exemplo de interface de comunicação entre duas *threads* : (a) produzindo dados; (b) consumindo dados.

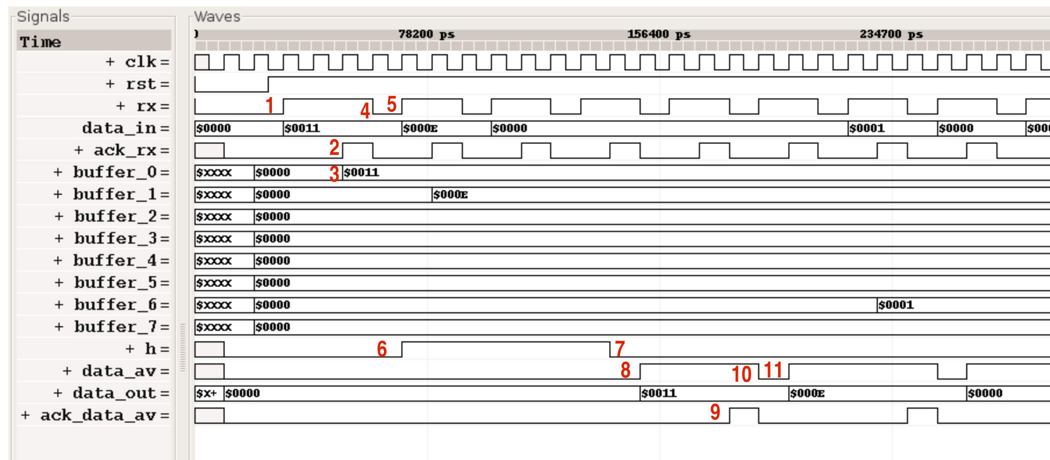

Figura 28 – Detalhes da simulação do módulo *buffer*, que controla a porta de entrada de uma das interfaces dos roteadores implementados nesse trabalho.

A qualidade em latência da descrição de hardware em nível RTL, extraída da síntese comportamental disponibilizada pela ferramenta utilizada neste trabalho, está diretamente ligada ao volume da troca de sinais entre as *threads* que implementam um determinado comportamento, porque essas trocas precisam ser sincronizadas com sinais adicionais. Para cada sinal que precisar de protocolo para a comunicação, a síntese comportamental irá derivar os ciclos que forem necessários para tal sincronização. Quanto mais *threads* forem incluídas na implementação de um módulo e quanto mais sinais essas *threads* trocarem, menos eficiente em latência será o hardware derivado.

4.4 Diretivas do Cynthesizer para Otimização da Qualidade dos Resultados

4.4.1 Protocolos para Sincronização entre Threads

Grande parte da degradação do desempenho em descrições comportamentais de estruturas de comunicação intrachip está relacionada à necessidade da troca de sinais para sincronização da comunicação. Portanto, a escolha do protocolo para sincronização pode reduzir os ciclos gastos nas troca de mensagens entre *threads*. A Figura 29 apresenta um exemplo da sincronização por *handshake* na interface de entrada de uma porta do roteador. A Figura 30 mostra a sincronização com mecanismo de crédito. A sincronização com *handshake* consome em média 3 ciclos para trocar um dado e armazenar no *buffer*. A sincronização com mecanismo de créditos leva 2 ciclos para realizar o mesmo processo.

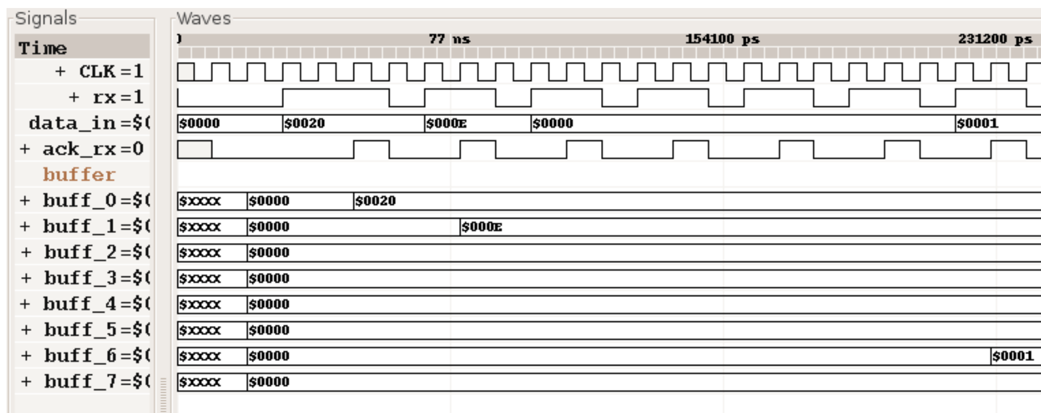

Figura 29 – Sincronização com *handshake* leva em média 3 ciclos para concluir a troca de dados.

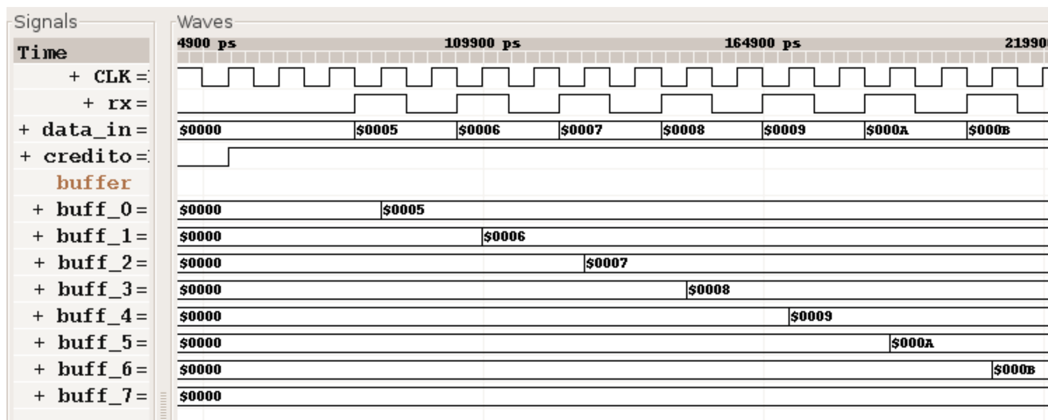

Figura 30 – Sincronização com crédito leva dois ciclos para concluir a troca de dados.

Quanto menos operações de sincronização forem necessárias para descrever um determinado comportamento melhor será a eficiência em termos de latência do resultado obtido. Sempre que for preciso utilizar sincronização, o projetista deve avaliar o protocolo de comunicação que garanta o funcionamento correto do sistema para que o mesmo não consuma ciclos desnecessariamente.

4.4.2 Desdobramento de Laços

A diretiva CYN_UNROLL é utilizada para desdobrar laços para que cada iteração, ou um conjunto destas, possa ser executada em paralelo com outras. A diretiva pode ser usada de duas maneiras: (1) CYN_UNROLL(ON|ALL|OFF, NOME PARA RELATÓRIO) para desenrolar o laço em quantas iterações ele possui (ON), todos os laços internos ao laço principal (ALL) ou desabilitar o desdobramento do laços (OFF); e (2) CYN_UNROLL(COMPLETE | CONSERVATIVE | AGRESSIVE, NÚMERO DE VEZES QUE O LAÇO PODE SER DESENROLADO, NOME PARA RELATÓRIO) para desenrolar parcialmente o laço.

A diferença entre COMPLETE, CONSERVATIVE e AGRESSIVE está na maneira como a ferramenta deriva a descrição comportamental para RTL. Considere o laço, em que OP pode ser uma das três opções:

```
for ( int i = 0; i < 4; i++){
          CYN_UNROLL(OP, 2, "laço");
         array[i] = func(i);
}
```

A Tabela 7 apresenta o código RTL derivado pela ferramenta para cada uma das opções para desenrolar parcialmente o laço.

Desenrolar complementante o laço, em geral melhora a latência da operação. Contudo em certos casos, como para acesso de vetores de muitas posições, isso pode implicar um aumento significativo de área, já que tal elemento pode ser mapeado para um conjunto de registradores ao invés de uma memória. Desenrolar parcialmente é uma opção intermediária entre um hardware com latência aceitável para a operação e uma área menor do que a obtida com o desenrolamento completo.

4.4.3 Redução do Tamanho dos Multiplexadores

A diretiva CYN_FLATTEN aplicada a um vetor transforma-o em um conjunto de registradores. Se esses registradores são vinculados a variáveis como índice, e esta não pode ser reduzida a uma constante, a ferramenta deriva um multiplexador para realizar esse acesso. Em

<table>
<tr><th>Diretiva</th><th>RTL SystemC Derivado pela Ferramenta</th></tr>
<tr><td>COMPLETE</td><td><pre>array[0] = func(0); // desenrolado para i=0
array[1] = func(1); // desenrolado para i=1
for (int i = 2; i < 4; i++) {
      array[i] = func(i);
}</pre></td></tr>
<tr><td>CONSERVATIVE</td><td><pre>for (int i = 0; i < 4; i += 2) {
     if (i >= 4) break;
     array[i] = func(i);
     (i+1 >= 4) break;
     array[i+1] = func(i+1);
}</pre></td></tr>
<tr><td>AGGRESSIVE</td><td><pre>for (int i = 0; i < 4; i += 2) {
    array[i] = func(i);
    array[i+1] = func(i+1);
}</pre></td></tr>
</table>

Tabela 7 – Resultado da utilização das diretivas COMPLETE, CONSERVATIVE e AGGRESSIVE para o desenrolamento parcial de laços.

alguns casos, o tamanho desses multiplexadores pode ser reduzido, com algumas alterações na própria descrição comportamental. Considere o código abaixo, em que $in1$ é uma porta de entrada com 1 bit, e a operação consiste em atribuir a y o valor da posição do vetor arr indexado por $I * 2$ (I vindo da iteração do laço) mais o valor lido de $in1$ (que pode assumir os valores 0 ou 1):

```
sc_uint<8> arr[128];
CYN_FLATTEN(arr);
sc_in< sc_uint<1> > in1;
for(sc_uint<6> I=0; I<32; I++){
        CYN_UNROLL( COMPLETE, 32, "laço" );
        y = arr[I*2+in1.read()];
}
```

No código acima, a ferramenta infere um multiplexador 128 para 1. Porém, com alguma modificação na descrição, retirando a leitura da porta $in1$ da operação de indexação, pode-se obter apenas um multiplexador 2 para 1. Tal modificação está descrita no código abaixo:

```
for(sc_uint<6> I=0; I<32; I++){
```

```
    CYN_UNROLL( COMPLETE, 32, "laço" );
    y = in1.read() ? arr[I*2+1] : arr[I*2];
}
```

4.4.4 Latência de Partes Específicas do Código

A diretiva CYN_LATENCY é utilizada para otimização de latência em certas partes de algoritmos. A estutura de uso da diretiva é: CYN_LATENCY(MÍNIMO EM CICLOS, MÁXIMO EM CICLOS, NOME PARA RELATÓRIO). Abaixo dá-se um exemplo:

```
...
{
CYN_LATENCY(1, 3, "Latencia_operacao");
x = y**3 + u;
}
```

Aqui, o projetista especifica que a operação $x = y^3 + u$ deve ser escalonada para conter 1, 2 ou 3 ciclos de relógio. Durante a síntese, componentes das bibliotecas caracterizadas para o projeto são escolhidos para compor a operação e ocorre a verificação do atendimento da restrição de latência. A ferramenta pode ou não confinar a operação nos limites especificados pelo projetista e gera um relatório sobre esse processo. No caso da operação acima, uma latência menor implica em adição de área para atender à restrição de 1 ciclo.

5 Arcabouços de Projeto Alto Nível para NoCs

Este Capítulo apresenta uma revisão dos arcabouços de projeto alto nível para redes intrachip e apresenta também a abordagem de modelagem adotada neste trabalho.

Bertozzi e outros, em [5] ressaltam a importância do suporte das ferramentas de CAD para que o projeto de redes intrachip seja viável. É necessário desenvolver bibliotecas especializadas, ferramentas de mapeamento de aplicações, fluxos de síntese específicos para NoCs, etc. Desenvolver sistemas de síntese específicos para NoCs é crucial para encontrar soluções de projeto viáveis em termos de desempenho, consumo de energia e personalização para aplicações específicas [5]. Para ser realmente efetivo, tal processo depende de um método de projeto que parta de uma aplicação, descrita em alto nível, e derive uma configuração otimizada da distribuição dos módulos e de sua correspondente arquitetura de comunicação.

A Seção 5.1 apresenta um fluxo específico para síntese de redes intrachip, conforme relatado por Bertozzi e outros [5]. A Seção 5.2 apresenta a proposta de Coppola e outros para modelagem de estruturas de comunicação intrachip baseado em um modelo conceitual disposto em níveis de abstração OCCN [9]. A Seção 5.3 apresenta os modelos propostos por Marcon [26] e sua implementação no arcabouço CAFES. Por fim, a Seção 5.4 descreve como foi realizada a modelagem das estruturas de comunicação que foram estudo de caso deste trabalho.

5.1 Netchip

Bertozzi e outros [5] apresentam um fluxo de projeto chamado *NetChip* como uma alternativa para personalizar arquiteturas de comunicação intrachip. Esse fluxo pressupõe o mapeamento da aplicação em núcleos durante uma fase de *projeto integrado de hardware e software* , com a utilização de alguma ferramenta específica para esse passo. A ferramenta utilizada nesse processo deve ser capaz de gerar um grafo (*core graph*) com os respectivos núcleos e a informação das demandas de suas comunicações. Esse grafo é a entrada do fluxo *NetChip* e passa por três fases de operação: *mapeamento da topologia*, *seleção da topologia* e *geração da topologia*. Essas fases são realizadas por duas ferramentas integradas no fluxo *NetChip*: *SUNMAP* [31], que realiza as fases de mapeamento e seleção, e o *xpipesCompiler* [20] que gera a topologia selecionada. A Figura 31 apresenta o fluxo de projeto *NetChip*.

Tendo como entrada o *core graph*, a função objetivo do projeto e as restrições que devem ser satisfeitas, a fase de *mapeamento da topologia* associa o grafo de entrada a várias topologias

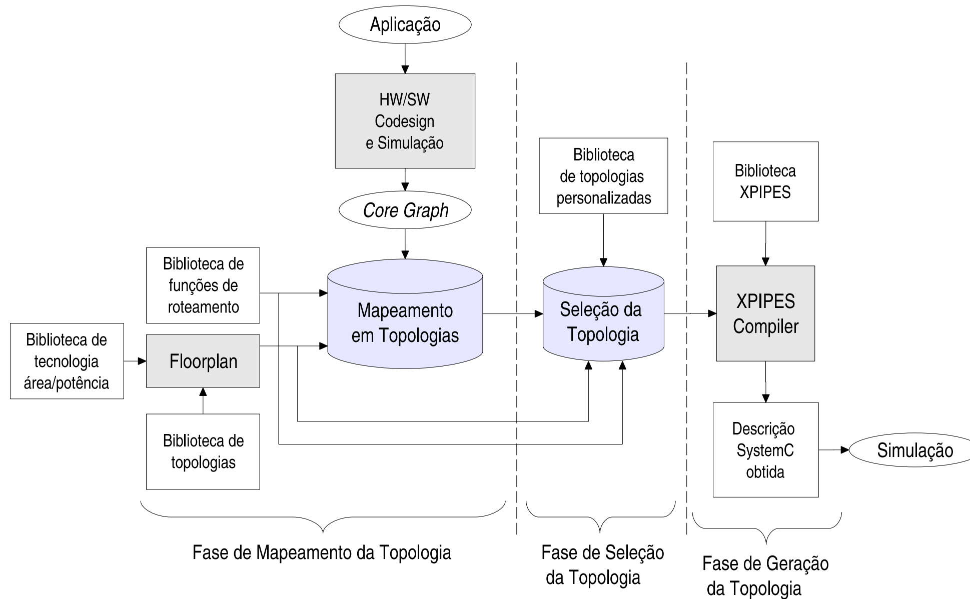

Figura 31 – Fluxo de projeto NetChip [5].

padrão (malha, toro, hipercubo, etc.) definidas na biblioteca de topologias, e a diferentes esquemas de roteamento, de acordo com a topologia. Para cada mapeamento, é avaliado o ajuste da micro-arquitetura obtida a restrições como largura de banda e área. Somente os mapeamentos viáveis são selecionados. A ferramenta também permite incorporar a essa fase uma estimativa de potência inicial para o projeto.

Na fase de *seleção da topologia*, os resultados obtidos anteriormente são avaliados de acordo com os objetivos do projeto e uma topologia para a aplicação é selecionada pelo projetista. A ferramenta *SUNMAP* [31], que engloba as duas fases apresentadas, gera como saída a descrição da topologia e dos roteadores da microrede.

A ferramenta *xpipesCompiler* [20], responsável pela fase de *geração da topologia*, transforma a descrição obtida na fase anterior em uma descrição SystemC. Essa descrição é gerada a partir dos elementos disponíveis na biblioteca *xpipes* [10]. Essa biblioteca é composta de macros (roteadores, interfaces, enlaces) descritas em SystemC com precisão a nível de ciclo.

Na fase inicial, de *mapeamento da topologia*, uma biblioteca de funções de roteamento é associada ao processo. A ferramenta dá suporte a diferentes funções de roteamento: *determinísticas* tais como com ordenamento de dimensões e outros algoritmos de caminhos mínimos; e *não determinísticas*, tais como divisão de tráfego entre os caminhos mínimos e divisão de tráfego entre todos os caminhos. A ferramenta utiliza uma heurística de três fases para mapear o *core graph* no grafo inicial da topologia da rede:

1. Obter um mapeamento inicial dos núcleos da rede através de um algoritmo guloso.

2. Para roteamento com caminhos mínimos, esses caminhos e os custos são computados como produtos origem/destino. Quando o roteamento é por divisão de tráfego, os cami-

nhos são obtidos resolvendo um sistema de equações MCF (do inglês, *Multi-Commodity Flow*), obtendo os produtos origem/destino.

3. A solução é melhorada iterativamente, invocando o segundo passo da heurística para cada mapeamento produzido com a troca origem/destino dos vértices.

Dessa forma, no mapeamento inicial o nodo que tem a máxima demanda de comunicação é mapeado para uma posição na rede em que tenha o número máximo de vizinhos [26]. Uma vez definido esse mapeamento, os produtos (origem/destino) são classificados em ordem, e assim, para cada produto forma-se um *grafo de quadrante* entre a origem e o destino. Dois exemplos de grafos de quadrante entre origem e destino são apresentados na Figura 32, formados entre os nodos e e h. Para cada grafo de quadrante, se forem considerados apenas os caminhos mínimos no roteamento, é aplicado o algoritmo de Dijkstra para obter os caminhos mínimos desse grafo. Após esses passos, pesos apropriados são colocados nas arestas. Isso é realizado para cada um dos produtos origem/destino na ordem inicialmente estabelecida.

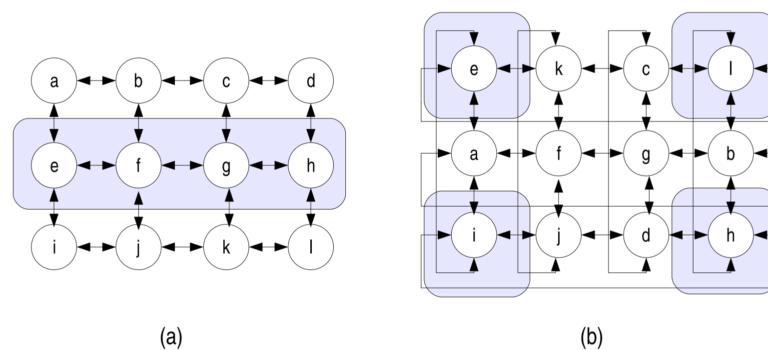

Figura 32 – Exemplos de grafos de quadrante de comunicação, obtidos no mapeamento da topologia com caminhos mínimos entre os nodos e e h.

Após realizado o roteamento para todos os produtos, *NetChip* dispõe de uma funcionalidade de geração de planta baixa do circuito, baseada nos módulos do Xpipes Compiler e associada à tecnologia escolhida, que é utilizada para realizar estimativas iniciais de área. Se as restrições de área e largura de banda forem atendidas, o custo da comunicação é calculado. A restrição de largura de banda é satisfeita se no resultado do mapeamento o tráfego que passa por um *enlace* é menor ou igual à capacidade do mesmo. Essa capacidade é oriunda da tecnologia associada à implementação. A restrição de área é satisfeita quando a área do mapeamento está dentro dos limites permitidos associados à implementação. Todo esse processo é repetido no terceiro passo da heurística, quando os pares origem/destino são invertidos. No final, o melhor mapeamento de todos os gerados é selecionado. Com essa heurística, o problema é particionado e os caminhos são selecionados dentro de partições e não sobre o grafo completo da rede.

Após essa seleção uma configuração personalizada da NoC pode ser gerada pelo *xpipes-Compiler*. Esse compilador utiliza a biblioteca *xpipes*, que contém componentes parametrizáveis para estruturas de comunicação que podem ser associados à arquitetura descrita em alto

nível obtida nos passos anteriores. A saída é uma descrição SystemC hierárquica de todos os roteadores, enlaces, nodos de processamento da rede e interfaces.

5.2 OCCN - On-Chip Communication Network

Coppola e outros [9] propõem outra abordagem para o projeto de estruturas de comunicação intrachip, chamada OCCN (do inglês, *On-Chip Communication Network*). Os Autores propõem um *arcabouço* conjuntamente com uma metodologia para especificação, modelagem, simulação e exploração do espaço de projeto de arquiteturas de comunicação intrachip. É disponibilizada uma API (do inglês, *Application Programming Interface*) SystemC orientada a objetos, aberta e flexível, que habilita a criação e o reuso de modelos executáveis de OCCAs (do inglês, *On-Chip Communication Architectures*) [8] [9]. Essas arquiteturas de comunicação intrachip englobam desdes barramentos, estruturas do tipo *crossbar* e NoCs, disponibilizando o mecanismo de comunicação entre os elementos de processamento distribuídos [8].

Segundo Coppola e outros [9], modelar uma NoC consiste em um conjunto de passos:

- Modelar os níveis de abstração - *(i) modelo funcional* - definido como o modelo que não considera o compartilhamento de recursos nem informações sobre o tempo, ou o tipo de sincronização ocorrendo em uma sequência de eventos; *(ii) modelo transacional comportamental* - são modelos mapeados para um domínio discreto de tempo, cuja sincronização é realizada por operações atômicas chamadas transações; *(iii) modelo transacional preciso a nível de ciclo* - este mapeia as transações para ciclos de relógio, estes modelos nem sempre são modelos sintetizáveis; *(iv) modelo RTL* - são modelos precisos em nível de ciclo de relógio, sintetizáveis com caminho de dados e controle bem definidos; *(v) modelo de portas* - este descreve o sistema em termos de primitivas lógicas.

- Ortogonalização - separar as funcionalidades de comunicação e computação da arquitetura de comunicação.

- Determinar um modelo de protocolo de comunicação - como o OSI, definindo as funções de cada camada de protocolos.

O método OCCN estabelece um modelo conceitual para comunicação de intermódulos baseado em níveis de abstração. São três níveis: *comunicação*, *adaptação* e *aplicação*. A Figura 33 apresenta a estrutura desse modelo conceitual.

O *nível de comunicação* é o inferior e implementa uma ou mais camadas do modelo OSI, a partir da camada física. O nível intermediário é o de *adaptação*, que mapeia uma ou mais camadas intermediárias do modelo OSI. Ele inclui componentes de adaptação de *software* e *hardware* para disponibilizar os *drivers* necessários para computação, comunicação, sincroni-

zação e serviços disponíveis para aplicações. O nível superior é o de *aplicação*, que mapeia diretamente a camada de aplicação OSI que lhe corresponde.

Para esses níveis de abstração, o modelo conceitual OCCN disponibiliza duas APIs. A *API de comunicação*, associada à interface entre os níveis *comunicação* e *adaptação*, disponibiliza uma interface que simplifica a implementação de vários níveis de *drivers* de comunicação em diferentes níveis de abstração. Esta é baseada em modelos genéricos, com reuso de componentes e separação entre comunicação e computação, oferecendo um conjunto de métodos para troca de dados e sincronização. A *API de aplicação*, que fica entre os níveis de *aplicação e adaptação*, especifica os métodos necessários para que a aplicação possa requisitar e utilizar serviços do nível de adaptação, e para que esse nível possa disponibilizar seus serviços para a aplicação.

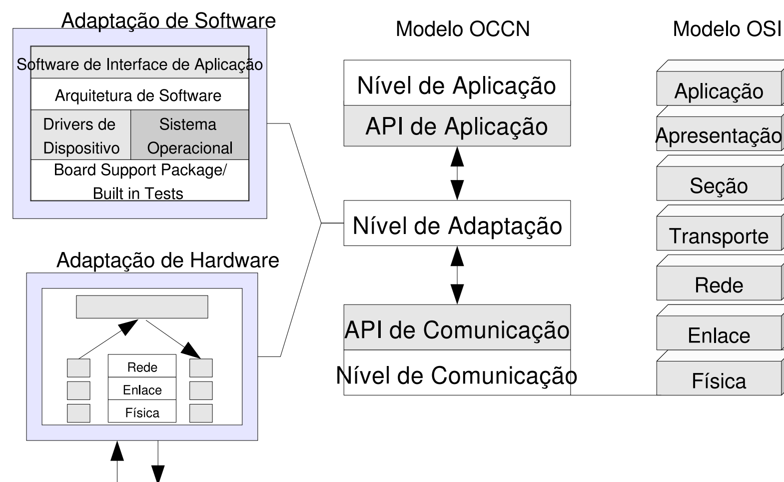

Figura 33 – Organização do modelo conceitual OCCN.

O nível de adaptação tem vários subníveis, é definido pelo projetista e deve incluir os elementos de adaptação de *software* e *hardware*, implementando funções como: interrupções, compartilhamento de memória, escalonamento de tarefas, etc. Na adaptação de *software*, o nível mais baixo é composto pelo BSP (do inglês, *Board Support Package*), que permite que todo *software*, inclusive o sistema operacional, seja carregado em memória para começar a executar, e pelo BIST (do inglês, *Built in Self Test*), que detecta e reporta erros de *hardware*. Logo acima, está o nível dos *drivers* de dispositivo e do sistema operacional. A camada seguinte é a de arquitetura, que contém a estrutura dos serviços oferecidos. No topo da adaptação de *software* está a interface de troca de requisições/respostas oferecida ao nível de aplicação do modelo OCCN.

5.3 CAFES - Communication Analysis for Embedded Systems

Marcon [26] [27] propõe modelos em alto nível para avaliar o *problema do mapeamento*, que consiste em associar os núcleos de propriedade intelectual (IPs) da aplicação nos nodos da infraestrutura de comunicação. Os modelos permitem capturar as características da rede intrachip a ser utilizada e o comportamento da aplicação a qual a rede deve dar suporte para avaliar os resultados em termos de consumo de energia, com base em modelos propostos em seu trabalho e na literatura. Os modelos explorados no trabalho de Marcon são:

- O *modelo de comunicação com pesos* [28] (CWM, do inglês *Communication Weighted Model*) modela uma aplicação em função da *quantidade de comunicação* que ocorre entre os pares de núcleos. A quantidade de comunicação considerada é a soma de todos os bits de todos os pacotes transmitidos entre núcleos durante a execução da aplicação.

- O *modelo estendido de comunicação com pesos* (ECWM, do inglês *Extended Communication Weighted Model*) acrescenta ao CWM as transições que ocorrem entre bits consecutivos na comunicação para computar o consumo de energia da comunicação de forma mais precisa.

- O *modelo de dependência da comunicação* (CDM, do inglês *Communication Dependence Model*) modela a aplicação em função da quantidade de comunicação e das dependências na disputa por recursos da infra-estura da rede. Tal modelo permite avaliar a contenção da rede e suas implicações no cálculo do consumo de energia e descobrir quais mapeamentos são menos custosos em relação a isso.

- O *modelo de computação e dependência da comunicação* (CDCM, do inglês *Communication Dependence and Computation Model*) acrescenta ao CDM a *quantidade de computação* dos núcleos da aplicação. A *quantidade de computação* é o tempo decorrente entre a ativação do envio da mensagem e o do real envio da mesma. Assim, podem ser considerados os períodos de ociosidade da infra-estrutura de comunicação no consumo de energia.

- O *modelo do padrão de comunicação da aplicação* (ACPM, do inglês*Application Communication Pattern Model*) modela uma aplicação através do ordenamento total dos eventos de forma que seja associada à cada mensagem um marcador de tempo. Esse marcador indica o instante em que uma mensagem é enviada de um núcleo para outro. Esse modelo não considera a computação dos núcleos da aplicação no cálcuo do consumo de energia.

- O *modelo de tarefas de comunicação* [19](CTM, do inglês *Communication Task Model*) considera a computação e a comunicação da aplicação e modela o sistema baseado no escalonamento das tarefas e da taxa de comunicação entre as mesmas. Esse modelo

contempla as aplicações de tempo real, já que permite a inserção de restrições de tempo para a execução das tarefas da aplicação.

Marcon [26] porpõe o arcabouço CAFES (do inglês, *Communication Analysis for Embedded Systems*), que inicialmente foi projetado para analisar as estimativas de consumo de energia de comunicação em aplicações embarcadas, em que os modelos propostos foram incorporados para avaliar o consumo de energia em comunicações aplicadas em redes intrachip. Para utilizar a ferramenta, o projetista deve informar como entrada os dados da infra-estrutura de comunicação alvo da aplicação: *parâmetros de topologia*, que contempla NoCs com topologia 2D malha e toro com roteamento XY e chaveamento *wormhole*, tamanho de *buffers*, tamanho da rede e comprimento entre as conexões da rede parametrizáveis; *parâmetros de tempo*, que são freqüência de relógio, número de ciclos para transmissão de 1 *phit* e número de ciclos para executar o roteamento; *parâmetros de consumo de energia*, como potência dissipada em um roteador e consumo de energia para transmitir 1 *phit*, considerando ou não transições em bits consecutivos. A partir daí, o projetista pode descrever a aplicação graficamente usando um formato particular de cada modelo e extrair o mapeamento de menor custo em termos de energia e tempo pela aplicação dos algoritmos adequadamente disponíveis.

5.4 Abordagem deste Trabalho

O processo de modelagem e validação de estruturas de comunicação intrachip deste trabalho foi realizado com uma ferramenta de síntese comportamental que não possui um arcabouço de projeto específico para NoCs ou outra estrutura de comunicação intrachip. A modelagem de cada estudo de caso foi realizada em SystemC, com o estilo de codificação aceito pela ferramenta Cynthesizer. Todas as classes que compõem a descrição de projeto foram elaboradas sem auxílio de outros arcabouços de projeto e validadas segundo o fluxo de verificação disponibilizado na ferramenta. Para as simulações foi necessária a utilização do ambiente ATLAS [34]. Cabe salientar que essa modelagem não considerou a aplicação para a qual a rede servirá e contemplou apenas redes de topologia toro 2D bidirecional. Os passos que seguem foram realizados para cada estudo de caso:

1. Avaliação de algoritmo da literatura para adaptação para topologia toro 2D bidirecional. Essa avaliação contemplou a verificação de que a adaptação do algoritmo é livre de *deadlock*.

2. Descrição comportamental das classes de projeto que compõe a rede. Elaboração de processo automatizado para parametrizar o tamanho de *buffer* e as dimensões da rede.

3. Síntese comportamental e posterior validação funcional oferecida pelo Cynthesizer. Nesse passo, ficou evidente a necessidade de auxílio para gerar tráfego para a simulação das re-

des. Para tanto, foi realizada uma adaptação utilizando a ferramenta ATLAS [34] para gerar tráfego e analisar resultados do desempenho da rede. Esta adaptação está descrita no Capítulo 7.

4. Síntese para ASIC ou FPGA dos resultados obtidos no passo anterior e nova simulação funcional.

5. Coleta e análise dos resultados apresentados nos relatórios disponibilizados pelo Cynthesizer e dos obtidos com a ferramenta ATLAS.

6 Redes Toro e Algoritmos de Roteamento

Uma rede intrachip é um conjunto de roteadores e canais ponto-a-ponto que interconectam elementos de um sistema de forma que estes possam comunicar-se de maneira eficiente.

Uma rede de interconexão qualquer, e uma rede intrachip em particular, pode ser caracterizada pela maneira com que seus nodos são interligados, ou seja sua topologia. Quanto ao tipo de topologia de interconexão, tais redes podem ser classificadas em *diretas* ou *indiretas*. Nas redes de topologia direta, cada nodo de processamento possui um nodo de chaveamento associado. As topologias diretas mais comumente utilizadas são a malha *n*-dimensional, toro, n-cubo k-ário e hipercubo [15]. Glass e Ni, em [17], apresentam definições formais para cada umas dessas topologias, reproduzidas a seguir:

- Uma rede malha *n*-dimensional (Figura 34 (a)) tem $k_0 \times k_1 \times ... \times k_{n-1}$ nodos, k_l nodos ao longo de cada dimensão l, onde $k_l \geq 2$. Cada nodo X em uma malha *n*-dimensional é identificado por n coordenadas, $(x_0, x_1, ..., x_{n-1})$, onde $0 \leq x_i \leq k_l - 1$ para cada dimensão i. Dois nodos X e Y são vizinhos se e somente se $x_l = y_l$ para todo i, exceto um j, onde $y_j = x_j \pm 1$. Dependendo de sua localização na rede um nodo tem de n a $2n$ vizinhos.

- A rede toro *d*-dimensional (Figura 34 (b)) contém $k_{d-1} \times k_{d-2} \times ... \times k_0$ dimensões, onde $d \geq 1$ e $k_i \geq 2$ para todo i, $0 \leq i < d$. Uma rede toro contém $n = \prod_{i=0}^{d-1} k_i$ nodos. Todo nodo da rede é identificado da forma $(a_{d-1}, a_{d-2}...a_0)$, onde $0 \leq a_i < k_i$ para todo i, $0 \leq i < d$. Cada nodo $(a_{d-1}...a_{i+1}, a_i, a_{i-1}...a_0)$ é conectado a outros nodos da rede da forma $(a_{d-1}...a_{i+1}, a_i \pm 1 \mod k_i, a_{i-1}...a_0)$, onde $0 \leq i < d$. As arestas que conectam os nodos identificados por $(a_{d-1}...a_{i+1}, k_i - 1, a_{i-1}...a_0)$ e $(a_{d-1}...a_{i+1}, a_i + 1, a_{i-1}...a_0)$ são chamadas arestas de retorno (do inglês, *wraparound*) e as outras são chamadas arestas internas.

- Redes n-cubo k-ário são um subconjunto da rede do tipo toro. Essas redes contêm k^n nodos e todos os nodos possuem o mesmo número de vizinhos. Se $k = 2$ cada nodo tem n vizinhos. Se $k > 2$ cada nodo tem $2n$ vizinhos. Se $n = 1$ o n-cubo k-ário é também chamada uma rede anel de k nodos.

- O hipercubo é um caso especial de rede malha *n*-dimensional e n-cubo k-ário (Figura 34 (c)). Essa topologia corresponde a uma malha *n*-dimensional com $k_i = 2$ para todo $0 \leq i \leq n - 1$, ou seja, um n-cubo 2-ário.

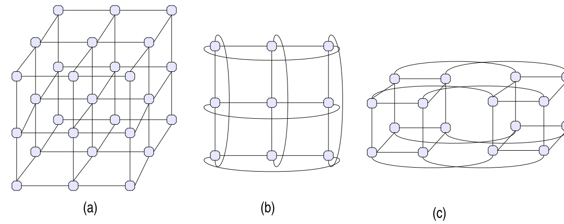

Figura 34 – Exemplos de topologias diretas: (a) malha 3X3X3 (b) toro (2-cubo 3-ário) (c) hipercubo (4-cubo 2-ário).

Nas redes de topologia indireta, os nodos de chaveamento não possuem necessariamente um nodo de processamento associado. O principal exemplo desse tipo de rede é o *crossbar*, como o apresentado na Figura 35 (a). Com N entradas e M saídas, um *crossbar* necessita $N \times M$ pontos de chaveamento na rede. Redes como o *crossbar* têm um custo proibitivo para grandes dimensões. Outras organizações de topologia indireta são as redes multiestágios, como a *butterfly* (Figura 35 (b)), em que os pacotes são roteados através de vários estágios de chaves.

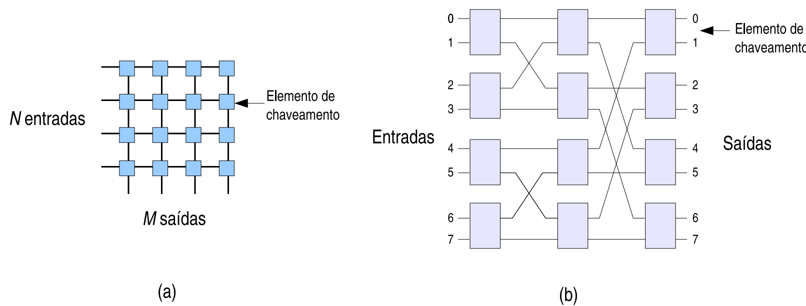

Figura 35 – Exemplos de redes com topologia indireta: (a) crossbar; (b) butterfly.

Uma rede intrachip também pode ser definida por seus mecanismos de comunicação. Esses mecanismos definem as formas como as mensagens são transferidas ao longo da rede. Esses mecanismos são: modo de chaveamento e algoritmo de roteamento.

O modo de *chaveamento* define como uma mensagem é transferida da entrada de um roteador para um de seus canais de saída. O chaveamento pode ser de *circuito* ou de *pacotes*. No chaveamento de circuito um caminho físico é reservado da origem até o destino, antes do início da transmissão dos dados. No chaveamento de pacotes, a mensagem pode, ou não, ser dividida mais de um pacote e cada um deles é encaminhado individualmente da origem até o destino. As técnicas de chaveamento de pacotes são: *store-and-foward*, em que o pacote é inteiramente armazenado em cada nodo de chaveamento antes de ser enviado para o próximo nodo; *virtual cut-through*, em que um pacote só é armazenado localmente se o canal de saída estiver indisponível, mas para isso o roteador deve possuir capacidade para armazenar todo o pacote;

wormhole, no qual os pacotes são divididos em unidades menores, ou *flits*. Assim não é necessário que o roteador tenha disponibilidade de armazenamento para pacotes inteiros, porque o pacote fica armazenado pelos *buffers* do caminho.

O *algoritmo de roteamento* define a seqüência de canais que uma mensagem percorre para chegar ao seu destino. Duato et al. [15] definem uma taxonomia para a classificação dos algoritmos de roteamento. Segundo estes Autores, vários critérios podem ser utilizados para classificá-los, incluindo:

- *Número de destinos*: os pacotes tem múltiplos destinos (*multicast*) ou um único destino (*unicast*).

- *Local das decisões de roteamento*: para roteamentos *unicast* as decisões de roteamento podem ocorrer de maneira centralizada, com um controle central da rota dos pacotes; no roteador de origem do pacote; de maneira distribuída, ou seja, em cada nodo do caminho do pacote é tomada alguma decisão em relação encaminhamento daquele pacote; ou de maneira híbrida, combinando os roteamentos anteriores.

- *Implementação*: independente do local das decisões de roteamento a implementação pode ser realizada através de uma tabela de roteamento ou de máquinas de estados finitas.

- *Adaptatividade*: tanto a implementação com tabela de roteamento quanto com máquinas de estados permitem que o roteamento seja determinístico ou adaptativo. No roteamento determinístico, pacotes com mesma origem e destino percorrem sempre o mesmo caminho. No roteamento adaptativo, é permitido que esses pacotes possam prosseguir por rotas diferentes para alcançar o destino.

- *Progressividade*: os algoritmos adaptativos podem ser classificados como progressivos ou regressivos (do inglês, *backtracking*). Os progressivos reservam um canal a cada operação de roteamento. Os regressivos permitem que o pacote retorne, liberando canais reservados anteriormente.

- *Minimalidade*: algoritmos podem ser classificados quanto a sua minimalidade em mínimos ou não mínimos. Um roteamento é mínimo quando o pacote chega ao seu destino no menor número de *hops* possíveis.

- *Número de caminhos*: algoritmos de roteamento adaptativos podem ser classificados como parcialmente adaptativos, em que apenas parte dos caminhos disponíveis pode ser utilizada, ou totalmente adaptativos, em que todos os caminhos possíveis do roteamento podem ser utilizados.

Segundo Glass e Ni em [17], um bom algoritmo de roteamento deve proporcionar uma baixa latência de comunicação, uma boa vazão de mensagens e ser fácil de implementar em VLSI. Os algoritmos de roteamento devem garantir que os pacotes sejam entregues a seus destinatários.

Para isso, é preciso garantir que o algoritmo seja livre de condições de *starvation*, *livelock* e *deadlock*.

Starvation é uma condição em que um processo solicitando acesso a um recurso pode ter sua requisição indefinidamente postergada, devido a uma política de atendimento não igualitária de alocação. Considerando o exemplo da Figura 36, em que os processos P_1, P_2 e P_3 disputam um recurso R_1 e suas requisições são avaliadas por um árbitro. Sendo $V = \{P_1, P_2, P_3\}$, o vetor do árbitro que armazena as requisições dos processos, a ordem de percurso desse vetor para atendimento das requisições pode acarretar *starvation*. Se o percurso inicia sempre pelo primeiro elemento do vetor e P_1 sempre faz requisições, P_2 e P_3 correm o risco de nunca serem atendidos. Se a ordem do percurso for que o último processo atendido recebe mínima prioridade, um critério mais igualitário, todos as requisições serão eventualmente atendidas.

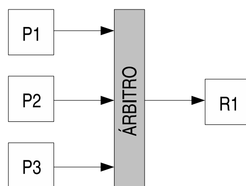

Figura 36 – Processos disputando um recurso, tendo suas requisições avaliadas por um árbitro.

No caso de redes intrachip, quando o cabeçalho de uma mensagem chega a um roteador, ele pode ser processado pelo mecanismo de arbitragem para que possa requisitar roteamento. Se esse mecanismo não for suficientemente justo, é possível que ocorra *starvation* em fluxos de mensagens.

Caso as mensagens que trafegam na rede possam ter seu percurso entre fonte e destino definido de tal forma que jamais atinjam o destino, ocorre a condição de *livelock*. Tipicamente, algoritmos de roteamento não-determinísticos e não mínimos mal elaborados podem apresentar riscos de gerar esta condição. A Figura 37 apresenta um exemplo, com uma rede malha 9X9 o nodo 88 tem um pacote destinado ao nodo 77. Com a utilização de um algoritmo adaptativo não mínimo é possível que ocorra uma sitação de *livelock* se for permitido que o pacote se aproxime e se afaste sem jamais atingir seu destino.

Deadlock é uma condição em que ocorre pelo menos uma dependência cíclica quando dois ou mais processos requisitam recursos que estão alocados a outros processos participando das solicitações, e isso causa um bloqueio cíclico de todos ou de um subconjunto dos processos envolvidos.

Quatro condições, isoladamente ou em conjunto, podem levar um conjunto de processos a essa situação: (*i*) exclusão mútua, ou seja, cada recurso só pode estar alocado para um processo em um determinado momento; (*ii*) posse e espera, em que processos que já possuem algum recurso podem solicitar um novo recurso; (*iii*) não-preempção, ou seja, recursos já alocados não podem ser desalocados do processo que o alocou sem a sua permissão explícita; (*iv*) espera

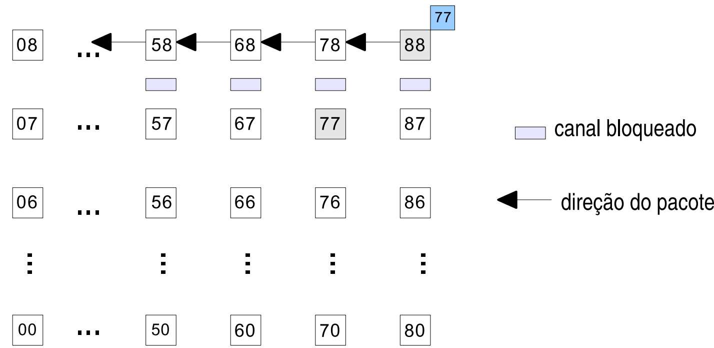

Figura 37 – Em uma rede malha 9x9, o nodo 88 tem um pacote destinado ao nodo 77. Os canais que podem ser selecionados pelo algoritmo estão ocupados e o pacote é encaminhado adaptativamente para outra direção para não permanecer bloqueado. Assim, o pacote pode se afastar consideravelmente de seu destino, gerando risco de *livelock*.

circular, em que cada processo pode estar aguardando um recurso retido pelo processo seguinte na cadeia.

A Figura 38 ilustra uma situação de *deadlock* ou impasse entre os processos A e B, que disputam os recursos R1 e R2. Inicialmente, o recurso R1 é alocado pelo processo A e o recurso R2 é alocado pelo processo B. Antes que o recurso R1 seja liberado, o processo A requisita o recurso R2 e não libera R1 antes que tenha alocado R2 para si. O mesmo acontece com o processo B, que requisita o recurso R1 antes de liberar R2 e não libera R2 antes de alocar R1.

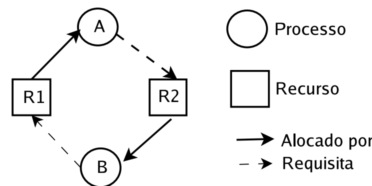

Figura 38 – Situação de *deadlock* ou impasse entre os processos A e B.

No caso de redes de intrachip, considere-se uma rede com técnica de chaveamento *wormhole* e topologia anel unidirecional. Considere-se ainda que a cada roteador, a lógica de roteamento disputa os canais do caminho que cada pacote deve seguir. *Deadlock* pode ocorrer quando as mensagens não podem avançar porque os *buffers* intermediários do caminho estão ocupados e forma-se um ciclo de dependência. A Figura 39, adaptada de Dally [11], apresenta um exemplo. Nela estão representadas as filas de uma rede anel de quatro nodos usando filas de entrada, cujo conteúdo indica o índice do nodo destino. Cada nodo da rede tem um pacote que deve ser encaminhado ao nodo oposto, isto é: o nodo n_0 tem um pacote endereçado para o nodo n_2; n_1 tem um pacote para n_3; n_2 para n_0; n_3 tem um pacote destinado a n_1. As filas representadas estão cheias e nenhuma mensagem consegue avançar e isso permanecerá assim até que uma ação excepcional quebre o ciclo do impasse.

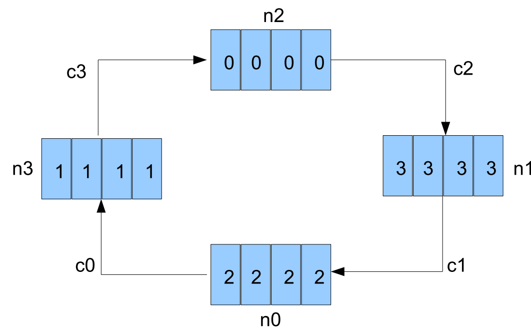

Figura 39 – Representação de uma situação de impasse em uma rede anel unidirecional com quatro nodos usando filas de entrada. Cada nodo tem um pacote destinado ao nodo oposto. As mensagens não conseguem avançar porque os *buffers* dos nodos estão cheios e existe uma dependência cíclica que impede de esvaziá-los.

Garantir que um algoritmo de roteamento seja globalmente livre de *starvation*, *livelock* e *deadlock* é imprescindível para assegurar que pacotes sejam entregues a seus destinos. Para cada uma destas situações é possível associar técnicas de prevenção:

- *Starvation* - é um problema relativamente simples de resolver com alguma técnica de escalonamento de processos/recursos que ofereça algum grau de justiça, como por exemplo, o escalonamento *round-robin*, também chamado de *prioridade rotativa dinâmica*. Nesta, a abitragem das requisições dos canais de cada roteador é baseada no último canal atendido. Caso se use arbitragem baseada em prioridades, alguma banda deve ser reservada para os pacotes de baixa prioridade, seja limitando o número de pacotes com alta prioridade ou reservando canais virtuais para os pacotes de baixa prioridade.

- *Livelock* - a utilização de algoritmos de roteamento mínimos é uma solução. Porém, se existe a necessidade de tolerância a falhas, caminhos não mínimos devem ser utilizados, mas com algum limite.

- *Deadlock* - existem três tipos de estratégias para essa situação: prevenir, evitar e recuperar. Para prevenir o sistema de entrar em uma situação de impasse é preciso garantir que a alocação dos recursos (canais, *buffers*) nunca vai levar a essa situação. Para isso, é preciso reservar todos os recursos antes de começar a transmitir o pacote. Esse é o caso do chaveamento de circuito. No caso de outros tipos de chaveamento, em que os recursos são alocados conforme o pacote avança na rede, utiliza-se prevenção e/ou recuperação de impasses. Na prevenção de *deadlock*, recursos são alocados se levarem um estado global seguro. Para a recuperação de *deadlock* é preciso estabelecer um mecanismo de detecção de *deadlock* e recursos podem ser retirados de processos para quebrar as situações de impasse.

Na Seção 6.1 a seguir são explorados alguns algoritmos de roteamento específicos para

redes com topologia toro com chaveamento *wormhole* e suas soluções para evitar situações de *deadlock*.

6.1 Algoritmos de Roteamento para Redes Toro

No caso específico de redes toro, a inserção de canais de *wrapparound* é problemática porque os algoritmos disponíveis para a rede malha não podem ser diretamente usados para realizar o roteamento livre de *deadlock*.

A Seção 6.1.1 apresenta o método proposto por Dally para gerar algoritmos de roteamento para redes toro unidirecionais com a inserção de canais virtuais para evitar *deadlock*, ilustrando o mesmo para redes anel. A Seção 6.1.2 mostra a adaptação e simplificação que Duato, Ni e Yalamanchili realizaram no algoritmo de Dally para uma redes toro unidirecionais em geral. A Seção 6.1.3, apresenta um modelo geral que serve para a elaboração de algoritmos de roteamento livres de *deadlock*, para várias redes diretas sem adição de canais virtuais proposto por Glass e Ni. Este modelo é denominado *turn model*. Por fim, a Seção 6.1.4 mostra o algoritmo de Draper e Petrini para redes toro bidirecionais.

6.1.1 Abordagem de Dally e Seitz

Dally e Seitz descrevem em [11] a implementação de uma rede toro dentro do TRC (do inglês, *Torus Routing Chip*) utilizando canais virtuais para evitar *deadlocks*. Através da divisão de canais físicos em canais virtuais a dependência cíclica de canais em uma dimensão é transformada em uma espiral acíclica.

Segundo Dally e Seitz [11], uma rede de interconexão direta pode ser descrita por um grafo direcionado $I = G(N, C)$, cujos vértices (N) representam os nodos de processamento e as arestas (C) representam os canais de comunicação. Assume-se que o chaveamento da rede é do tipo *wormhole*. A cada canal c_i é associada uma fila com capacidade $cap(c_i)$. O nodo de origem de um pacote é chamado s_i e o de destino d_i.

A essa rede associa-se uma função de roteamento $R : C \times N \mapsto C$, que a cada roteador mapeia o canal de entrada de um pacote (c_c) para um canal de saída (c_n) baseado no seu nodo de destino (n_d). Essa função de roteamento permite descrever roteamentos determinísticos.

A função de roteamento R pode ser usada para gerar um grafo de dependências entre canais $D = G(C, E)$, em que o conjunto dos vértices de D são os canais de I e as arestas de D são os pares $E = \{(c_i, c_j)|R(c_i, n) = c_j\}$ produzidos pela função R para todo $n \in N$. Tal função tem como resultado uma configuração de atribuição de canais em cada vértice $n \in N$ do caminho que o pacote deve seguir. Uma configuração é suscetível a *deadlock*, dada a função de roteamento R e considerando $size(c_j)$ como o número de flits de um pacote, d_i como o nodo de

destino e $member(n, c_i)$ indicando que o canal c_i tem um flit destinado ao nodo n, definida por Dally como: $\forall c_i \in C$, ($\forall n \in member(n, c_i)$, $n \neq d_i$ and $c_j = R(c_i, n) \Rightarrow size(c_j) = cap(c_j)$). Nessa configuração, nenhum flit está a apenas um *hop* de seu destino e nenhum flit pode avançar porque a fila do próximo canal está cheia.

Para solucionar o problema do*deadlock* para arquiteturas de comunicação, Dally apresenta teorema a seguir:

Teorema 1 *Uma função de roteamento R, para uma rede de interconexão I, é livre de impasse se não existirem ciclos no grafo de dependência de canais D.*

Dally e Seitz provam esse teorema propondo uma forma de garantir que não haverão ciclos através de um *ordenamento dos canais*. Para garantir que os ciclos do grafo sejam quebrados, são introduzidos *canais virtuais* para cada porta de comunicação do roteador.

Considere-se como exemplo o caso de uma rede anel unidirecional de quatro nodos, conforme Figura 40, com $N = \{n_0, ..., n_3\}$ e $C = \{c_0, ...c_3\}$. Na Figura, o grafo de interconexão I é mostrado à esquerda e o grafo de dependência D à direita.

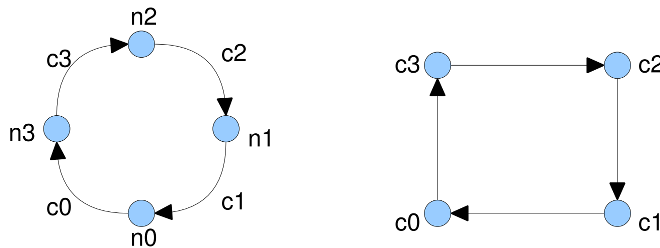

Figura 40 – Grafo de interconexão e grafo de dependência de canais para uma rede anel de quatro nodos.

Dally e Seitz substituem cada canal por dois canais virtuais, um canal baixo c_{0x} e um canal alto c_{1x}. Assim, o ciclo de dependência entre os canais pode ser quebrado. A Figura 41 apresenta à esquerda o novo grafo de interconexão I e à direita o grafo de dependência D, correspondentemente.

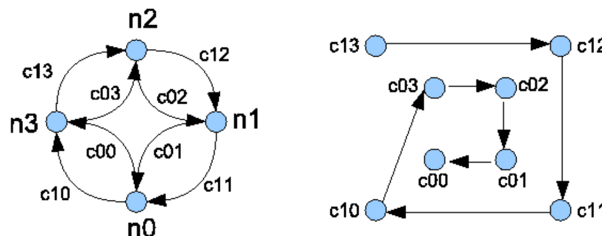

Figura 41 – Grafo de interconexão e grafo de dependência entre canais para uma estrutura anel de quatro nodos com a adição de canais virtuais e quebra do ciclo de dependência entre os canais.

O novo grafo de dependências pressupõe que pacotes partindo de um nodo com numeração menor que o destino percorrem seu caminho através de canais altos e os que partem de um nodo com numeração maior que o destino são encaminhados pelos canais baixos. Os canais são percorridos em ordem decrescente de seus índices e o roteamento é então livre de impasse. Essa técnica pode ser estendida de redes anel para redes toro em geral.

6.1.2 Abordagem de Duato, Ni e Yalamanchili

Duato, Ni e Yalamanchili, em [15], realizaram um estudo sobre algoritmos de roteamento e apresentaram uma codificação adaptada do algoritmo de Dally e Seitz [11], descrito na Seção 6.1.1, para redes n-cubos k-ários unidirecionais.

Nesta codificação, cada canal da rede foi dividido em dois canais virtuais, c_{0i} e c_{1i}. Após, cada canal recebeu um nome c_{dvi}, onde: $d = \{0, 1, ..., n-1\}$, é a dimensão atravessada pelo canal; $v = \{0, 1\}$, indica o canal virtual; $i = \{0, 1, ..., k-1\}$, indica a posição dentro do anel correspondente. A Figura 42 apresenta uma rede 2-cubo 4-ária (toro 4x4) unidirecional e seus canais numerados de acordo com o método de Duato, Ni e Yalamanchili.

A partir desta codificação dos canais, o algoritmo de roteamento encaminha pacotes seguindo sempre uma ordem crescente dos canais. Duato apresenta um algoritmo específico para redes toro 2D. Considera-se: (*Xcurrent, Ycurrent*) como as coordenadas do nodo em que o pacote se encontra; (*Xdest, Ydest*) como as coordenadas do nodo de destino do pacote; *Canal* como o canal selecionado para encaminhar o pacote, a estrutura do algoritmo proposto é apresentada na Figura 43.

O terceiro índice da codificação não aparece no algoritmo porque ele é utilizado para diferenciar um roteador do outro dentro do anel correspondente da rede. Esse algoritmo é muito simples de ser implementado e não é necessário para o roteador conhecer as dimensões específicas da rede e todos os roteadores são equivalentes para o roteamento.

A Figura 44 mostra um exemplo de execução do algoritmo. Em uma rede toro 3x3 unidirecional, o nodo 22 tem um pacote destinado ao nodo 11. O valor de *Xoffset* calculado no nodo 22 é negativo e o canal baixo c_{002} é escolhido. No nodo seguinte, 32, novamente *Xoffset* é negativo e o canal baixo do *wraparound* c_{003} é selecionado. Chegando ao nodo 02, *Xoffset* passa a ser positivo e o canal alto c_{010} é escolhido. Assim o pacote atravessa a dimensão X e chega ao nodo 12, em que *Xoffset* é igual a zero e *Yoffset* passa a ser considerado. No nodo 12, *Yoffset* é negativo, assim o canal baixo c_{102} é selecionado. No nodo 13, novamente *Yoffset* é negativo e o canal baixo de *wraparound* c_{103} é selecionado. No nodo seguinte, 10, o *Yoffset* passa a ser positivo e o canal alto c_{110} é selecionado. No nodo 11, destino do pacote, *Xoffset* e *Yoffset* são zero e o pacote é encaminhado para o módulo de processamento local.

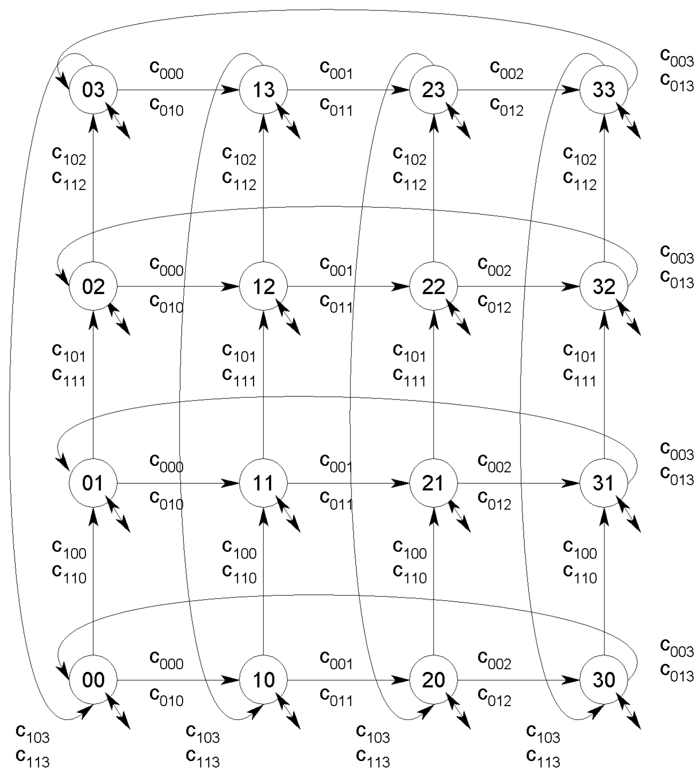

Figura 42 – Rede toro 4x4 unidirecional com canais numerados de acordo com o método de Duato, Ni e Yalamanchili.

6.1.3 Abordagem de Glass e Ni

Glass e Ni [17] realizaram um extenso estudo sobre roteamento adaptativo e propuseram um modelo chamado *turn model* para gerar algoritmos de roteamento que evitem a situações de *deadlock* em redes com chaveamento do tipo *wormhole* com topologias malha, n-cubo k-ário e hipercubos, sem a adição da canais virtuais. Esse modelo pressupõe o uso de redes ortogonais, ou seja, aquelas onde a posição de cada roteador por ser definida por uma tupla de valores, uma para cada dimensão da rede.

O modelo consiste em analisar as *curvas* (*turns*) que um pacote pode tomar, ou seja, as trocas de direção ortogonal ao longo do caminho do pacote entre fonte e destino, e os ciclos que essas mudanças de direção podem causar. Com base nisso, os Autores propõem algoritmos de roteamento baseados na proibição de algumas curvas para quebrar os ciclos do grafo de dependência, eliminando assim a possibilidade de *deadlock*. Os algoritmos de roteamento assim produzidos são em geral parcialmente adaptativos, porque sem a utilização de canais virtuais não é possível obter algoritmos totalmente adaptativos livres de *deadlock* [32].

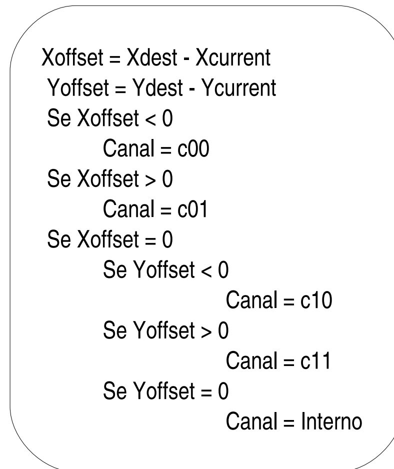

Figura 43 – Algoritmo de roteamento para a rede toro 2D unidirecional com canais virtuais.

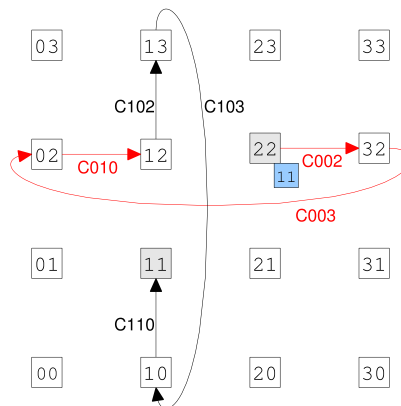

Figura 44 – Exemplo da execução do algoritmo, onde a origem 22 envia uma pacote para o destino 11.

Uma situação de *deadlock* não ocorre se não existir dependência cíclica entre os canais. No *turn model*, uma curva ocorre quando o pacote passa de uma dimensão para outra. Nesse modelo, para evitar *deadlock* é preciso proibir tipos de curva em número suficiente para quebrar todos os ciclos da rede. Para redes do tipo malha *n*-dimensional e n-cubo k-ário, seis passos devem ser seguidos para desenvolver um algoritmo baseado no *turn model*:

1. Classificar os canais de acordo com as direções que podem ser tomadas pelo roteamento dos pacotes. No caso de n-cubo k-ário, os canais de *wrapparound* não devem ser considerados nesse passo.

2. Identificar as curvas que ocorrem entre duas dimensões, omitindo as curvas de 0 e 180 graus. Curvas de 180 graus mudam o pacote de direção, mas não de dimensão. Curvas de 0 graus ocorrem quando existem canais virtuais em uma direção e o pacote pode trocar de canal virtual sem mudar a direção ou sentido do movimento.

3. Identificar os ciclos mais simples que essas curvas podem formar.

4. Proibir pelo menos uma curva em cada ciclo.

5. No caso de n-cubo k-ário, incorporar as curvas que envolvem canais de *wrapparound*, desde que estes não introduzam novos ciclos.

6. Adicionar curvas de 0 e 180 graus, desde que sem reintroduzir ciclos. Isso somente é necessário quando existem canais virtuais e para roteamentos não-mínimos.

Considerado uma rede malha de duas dimensões, as quatro direções que podem ser tomadas permitem oito curvas, ou seja, direita e esquerda para leste, oeste, norte e sul. Essas oito curvas formam dois ciclos simples, conforme mostrado na Figura 45 (a). O algoritmo de roteamento XY, em que primeiro o pacote atravessa a dimensão X e depois a dimensão Y, proíbe quatro curvas, conforme Figura 45 (b). O *deadlock* é evitado, porém sem garantir adaptatividade ao algoritmo. Dentro dos ciclos, é possível evitar impasse proibindo apenas uma curva, o que ocasionaria uma certa adaptatividade ao algoritmo. Porém, nem sempre o *deadlock* seria evitado se os dois ciclos fossem combinados. A Figura 46 apresenta essa situação. Na Figura 46 (a) as três curvas à esquerda permitidos equivalem a curva à direita proibido na Figura 46 (b). Na Figura 46 (b) as três curvas à direita permitidas equivalem a curva à esquerda proibida na Figura 46 (a). Se ambos os ciclos coexistirem, conforme o exemplo da Figura 46 (c), haverá uma situação de *deadlock*.

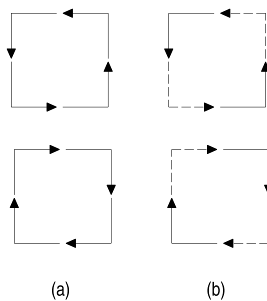

Figura 45 – Curvas em uma rede malha de duas dimensões: (a) ciclos simples formados pelas direções permitidas para uma rede malha bidimensional; (b) curvas proibidas (pontilhadas) para evitar *deadlock*.

Para redes de topologia malha, de Glass e Ni propõem os algoritmos: *west-first*, *north-last* e *negative-first*. No algoritmo *west-first*, são proibidas duas curvas de 90 graus para quebrar os ciclos, conforme Figura 47(a). Nesse algoritmo, o pacote é encaminhado primeiramente a direção Oeste, se necessário, e então adaptativamente é encaminhado para Norte, Sul ou Leste. O algoritmo *north-last* é outra maneira de proibir duas curvas de 90 graus, conforme Figura 47(b). Por esse algoritmo, primeiramente o pacote é encaminhado adaptativamente para o Leste, Sul ou Oeste e a última direção tomada é para o Norte. O algoritmo *negative-first* também proíbe duas curvas de 90 graus, conforme Figura 47(c). O pacote é adaptativamente encaminhado primeiro às direções negativas (Sul e Oeste) e depois, adaptativamente, às direções positivas (Norte e Leste). Para rede malha de 2D todos estes algoritmos são livre de *deadlock*.

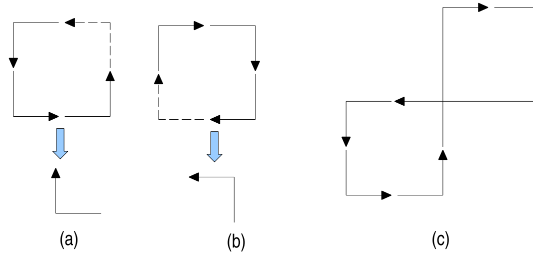

Figura 46 – Situação de *deadlock* gerada na combinação de curvas: (a) proibir uma curva torna as curvas à esquerda restantes equivalentes a proibir uma curva à direita; (b) proibir uma curva torna as curvas à direita restantes equivalentes a proibir uma curva à esquerda; (c) os dois ciclos combinados geram uma situação de impasse.

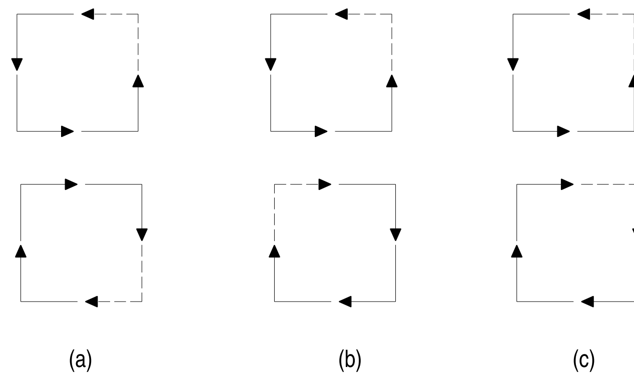

Figura 47 – Curvas permitidas (traço contínuo) e proibidas (traço pontilhado) nos algoritmos *turn model* de Glass e Ni: (a) west-first; (b) north-last; (c) negative-first.

Para redes do tipo toro bidirecional, Glass e Ni afirmam que é possível estender os algoritmos de roteamento da rede malha. O algoritmo *negative-first* pode ser estendido classificando cada canal *wraparound* como negativo ou positivo, de acordo com a posição relativa da rede malha. Os canais de retorno só são utilizados uma vez no caminho do pacote. O algoritmo *negative-first*, para esse caso, é não-minímo.

Para redes toro bidirecionais com $k > 4$, é impossível gerar algoritmos mínimos sem adicionar canais virtuais [32]. Glass e Ni citam o trabalho de Linder e Harden [25], que trata do particionamento da rede n-cubo k-ário bidirecional em 2^{n-1} redes virtuais com $n+1$ níveis por rede virtual e $(n+1)k^n$ canais virtuais por nível. Assim, cada canal físico é particionado em $(\frac{(n+2)^2}{n})2^{n-2}$ canais virtuais [25]. Essa abordagem é claramente impraticável para NoCs. Glass sugere a utilização do algoritmo proposto por Dally e Seitz (Seção 6.1.1) para obter algoritmos adaptativos e mínimos e praticáveis para toro ou n-cubo k-ário unidirecional.

6.1.4 Abordagem de Draper e Petrini

Uma abordagem de roteamento específica para rede n-cubo k-ário bidirecional, é apresentada nos trabalhos de Draper [13] e de Draper e Petrini [14]. Essa abordagem também utiliza canais virtuais e ordenação dos mesmos para a execução da função de roteamento.

Draper, em [13], apresenta o algoritmo *Red Rover* para redes de anéis bidirecionais (1-cubo k-ário). Cada canal físico do anel é dividido em dois canais virtuais e os canais da rede são separados em dois conjuntos A e B. Nesse algoritmo, a rede anel é organizada em dois conjuntos de nodos contíguos: os nodos de 0 a $(\frac{k}{2} - 1)$ como um grupo e os nodos $\frac{k}{2}$ até $k - 1$ outro grupo. Mensagens que são injetadas na rede por um dos grupos de nodos são sempre encaminhadas pelos canais do conjunto A, independente do destino. Mensagens injetadas na rede pelo outro grupo de nodos percorrem sempre os canais do grupo B. A Figura 48 apresenta o mapeamento dos canais virtuais na rede anel. Esse mapeamento forma quatro redes distintas: A_+, A_-, B_+ e B_-. Os dois canais não utilizados no roteamento estão tracejados na Figura.

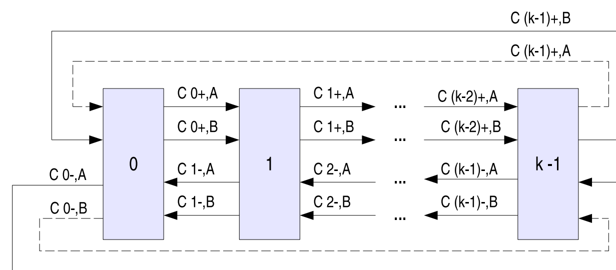

Figura 48 – Canais virtuais na rede anel bidirecional.

Para provar que o algoritmo é livre de *deadlock* basta mostrar que não existem ciclos se os pacotes forem encaminhados em um ordenamento coerente. Para cada uma das redes virtuais é possível mostrar um ordenamento de canais válido. Nas redes A_+ e B_+ o ordenamento é crescente e nas redes A_- e B_- decrescente. Por exemplo, para a rede A_+ cada um dos nodos, entre 0 e $(\frac{k}{2} - 1)$, pode enviar dados para todos os nodos entre seu nodo subseqüente até o nodo $k - 1$ pelos canais A_+. O ordenamento de canais para a rede A_+ é $c_{0+,A} > c_{1+,A} > ... > c_{(k-2)+,A}$. O canal $c_{(k-1)+,A}$ nunca será utilizado, porque sua utilização não resultaria em um encaminhamento com ordenamento válido. Já para a rede B_-, o ordenamento dos canais é $c_{(k-1)-,B} < c_{(k-2)-,B} ... < c_{2-,B} < c_{1-,B}$. O canal $c_{0-,B}$ não é utilizado porque sua inclusão não resulta em um ordenamento válido.

Draper e Petrini, em [14], estendem o algoritmo para redes toro bidirecionais. O algoritmo deve ser combinado com o algoritmo de ordenamento de dimensões. Primeiramente, o pacote é encaminhado seguindo o ordenamento das dimensões. Quando alcança a última dimensão é então encaminhado segundo o algoritmo Red Rover.

6.1.5 Comparação de Algoritmos de Roteamento para Redes Toro

A Tabela 8 apresenta um quadro comparativo entre os algoritmos revisados no capítulo. São considerados como critérios: a solução para *deadlocks*, a utilização de canais virtuais, o tipo de roteamento e as redes que abrangem.

Abordagem	Solução para Deadlock	Uso de Canais Virtuais	Características de Roteamento	Redes em que se aplicou
Dally e Seitz	Quebra dos ciclos de dependência de canais através da inserção de canais virtuais	Sim	Determinístico, não adaptativo com ordenamento dos canais	n-cubo k-ário unidirecionais, toro unidirecional
Duato, Ni e Yalamanchili	Codificação simplificada do modelo de Daly e Seitz, com a utilização de canais virtuais para quebrar os ciclos de impasse	Sim	Determinístico, não adaptativo com ordenamento dos canais	n-cubo k-ário unidirecionais, toro unidirecional
Glass e Ni	Construção de um modelo que leva em consideração as direções tomadas no roteamento e a proibição das que levam a ciclos de dependência	Não, mas podem ser adicionados para aumentar a adaptatividade	Determinístico, parcialmente adptativo	malhas n-dimensionais, n-cubo k-ário, p-cubos, todas bidirecionais
Draper e Petrini	Pacotes ficam confinados em redes virtuais	Sim	Determinístico, não adaptativo	anel bidirecional e n-cubo k-ário bidirecionais

Tabela 8 – Quadro comparativo dos algoritmos revisados no Capítulo.

7 Rede Toro 2D Bidirecional com Roteamento Turn Model

Como um estudo de caso de implementação prática para esse trabalho foi escolhida uma rede toro 2D bidirecional com modo de chaveamento *wormhole* e algoritmo de roteamento *Turn Model*. Para tanto, foram estudados algoritmos de roteamento com técnicas de prevenção de *deadlocks*, conforme Capítulo 6, Seção 6.1. O trabalho de Dally e Seitz, apresentado na Seção 6.1, Subseção 6.1.1 apresenta um algoritmo de roteamento para rede toro unidirecional utilizando canais virtuais para evitar *deadlocks*. A solução de Duato, Ni e Yalamanchili, apresentada na Seção 6.1, Subseção 6.1.2 é uma simplificação do modelo de Dally com uma codificação e algoritmo mais simples, também para toro unidirecional. Uma solução para roteamento em toro bidirecional com canais virtuais é apresentada por Draper e Petrini na Seção 6.1, Subseção 6.1.4. Um abordagem sem utilização de canais virtuais é mostrada no trabalho de Glass e Ni, na Seção 6.1, Subseção 6.1.3 em que restrições de certas curvas em certas direções no roteamento pode evitar impasses. Porém essa última abordagem não é suficientemente explorada para redes com topologia toro.

Nesse trabalho foi realizado um estudo sobre a adaptação do algoritmo *west-first não-mínimo* para redes toro 2D bidirecionais, sem a utilização de canais virtuais. Devido à inserção de canais de *wraparound*, quando se migra da topologia malha para a topologia toro, é preciso considerar o comportamento do algoritmo nas bordas da rede. A Figura 49 apresenta uma rede toro 4x4 2D, onde I representa o conjunto dos roteadores da extremidade esquerda, II o conjunto da extremidade direita, III o da extremidade norte e IV o da extremidade sul. No *turn model*, algumas direções devem ser proibidas para evitar ciclos de impasse. Com os canais *wraparound* isso complica-se e as restrições de roteamento devem incluir a consideração das particularidades desses canais. Por exemplo, seguir um enlace de *wraparound* é continuar na mesma direção ou realizar uma curva de 180 graus?

Uma maneira de garantir que não ocorrerá *deadlock* é fazer com que os canais de *wraparound* sejam usados apenas uma vez no roteamento [17]. Para evitar transportar informações dinâmicas de roteamento no pacote, pode-se ainda forçar que a utilização desses canais possa ocorrer apenas no primeiro *hop* do pacote. Assim, o roteador deve ser capaz de reconhecer que tem um ou mais canais de retorno e se é vantajoso utilizá-lo quando um pacote é injetado na sua porta local. Outra questão importante é garantir que a utilização desse canal não entre em conflito com as restrições do algoritmo *west-first*.

Na adaptação aqui proposta, o algoritmo *west-first* é utilizado sempre que um pacote é injetado em um roteador que não é da borda. Se o pacote for injetado em um roteador de um dos conjuntos das extremidades, deve-se realizar uma operação baseada no destino do pacote.

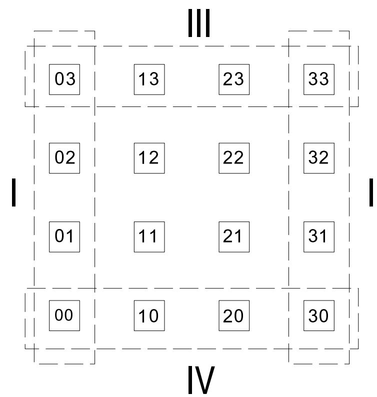

Figura 49 – Rede toro 4x4, salientando os conjuntos de roteadores nas extremidades da rede.

Se for vantajoso, utiliza-se o canal de *wraparound* para o primeiro *hop*. Caso contrário, o *west-first* deve ser seguido a risca. Após utilizar o canal de *wraparound* um pacote. segue o algoritmo *west-first* até alcançar o destino.

A Figura 50 apresenta resumidamente um pseudo-código da descrição comportamental do algoritmo de roteamento *west-first não-mínimo* adaptado para a rede toro 2D bidirecional. Considere-se: DIM_M e DIM_N como as dimensões em X e em Y da rede; add_x e add_y como o endereço (X, Y) de cada nodo na rede; $header_x$ e $header_y$ como o endereço de destino em X e Y do pacote que chega ao nodo; $incoming$ como a fila que fez a requisição para tal roteamento; $controle$ como uma variável auxiliar para o processo de decisão; $canais[n]$ como o vetor que contém a lista de portas válidas para o caminho daquele pacote; e X_{offset} eY_{offset} como variáveis auxiliares no cálculo da escolha do canal de destino do pacote. A requisição do árbitro é capturada com a função *captura_requisicao()*, e esta atribui os valores para $incoming$, $header_x$ e $header_y$. Primeiramente, se o pacote é injetado na rede no roteador, ou seja, pela porta Local deste, um conjunto de condições é testado para verificar se o nodo pertence a algum dos conjuntos de borda da rede. Se pertencer, verifica-se se vale a pena utilizar o canal de *wraparound*. Caso contrário, se não se trata de um roteador de borda e/ou não vale a pena utilizar o *wraparound*, a variável $controle$ é setada para 1. A partir daí, se $controle = 1$, devido ao não atendimento das condições anteriores, ou se o pacote chega ao nodo pelas outras portas (Leste, Oeste, Norte ou Sul), o algoritmo *west-first* puro é seguido. Nesse algoritmo, quando o pacote alcança sua dimensão X ($X_{offset} = 0$), vindo na direção Leste-Oeste, e tem os canais nas direções de Y bloqueados (Norte ou Sul) pode-se contornar essa situação continuando na direção Oeste. Disso decorre a *não-minimalidade* do algoritmo. A função $atualiza_tabela_roteamento(incoming, canais[])$ percorre o vetor $canais$ verificando, em ordem, se entre os canais escolhidos algum está disponível e garante que o pacote não retorne pela porta em que chegou ao roteador. Essa função também atualiza os valores das tabelas de roteamento, responde ao árbitro e informa as portas de saída do estabelecimento das conexões. O encerramento das conexões foi omitido nesse pseudo-código.

```
DIM_M = constante com tamanho em X da rede
DIM_N = constante com tamanho em Y da rede
add_x = endereço X do nodo na rede
add_y = endereço Y do nodo  na rede
incoming = porta da requisição
header_x = endereço X do pacote
header_y = endereço Y do pacote
canais[2] =  vetor com a ordem de atribuição de canais.
controle=0

captura_requisicao( );
Se requisição vem da porta  Local : incoming==local
        Condição 1: Se add_x  Є  I e  header_x > DIM_M/2
                canais[0] =  oeste
        Condição 2: Se add_x  Є  II e  header_x <  DIM_M/2
                canais[0] = leste
        Condição 3: Se add_y  Є  III  e (header_x>= add_x &  header_y<(DIM_N/2))
                canais[0] = norte
        Condição 4: Se add_y  Є  IV  e  (header_x>= add_x &  header_y >(DIM_N/2))
                canais[0] = sul
        Default - nenhuma das condições atendidas:
                controle=1

Se (controle=1 | ( incoming == leste | oeste | norte | sul ))
        x_offset=header_x – add_x
        y_offset=header_y – add_y
        Se x_offset<0
                canais[0] = oeste
        Se x_offset>0
                Se y_offset<0
                        canais[0] = leste
                        canais[1] = sul
                Se y_offset>0
                        canais[0] = leste
                        canais[1] = norte
                Se y_offset==0
                        canais[0] = leste
        Se x_offset==0
                 Se y_offset<0
                        canais[0] = sul
                        canais[1] = oeste - se  veio de leste e não chegou ao extremo oeste da rede
                Se y_offset>0
                        canais[0] = norte
                        canais[1] = oeste  - se  veio de leste e não chegou ao extremo oeste da rede
                Se y_offset==0
                        canais[0] = local
    atualiza_tabela_roteamento( incoming, canais[ ] )
```

Figura 50 – Pseudo-código do algoritmo *west-first não-mínimo* elaborado para a rede toro 2D bidirecional.

Dois exemplos são apresentados na Figura 51 (a) e (b). Em (a), o roteador 13 tem um pacote injetado com destino ao nodo 30. Qualquer uma das rotas adaptativas do *west-first* usando os caminhos de uma malha equivalente levaria 5 *hops* para chegar ao destino. Com a utilização do canal de retorno levar-se-ia 3 *hops*. Nesse caso, é vantajoso utilizar o canal de *wraparound*.

Já na situação (b), em que o roteador 23 tem um pacote injetado com destino ao nodo 00, não é possível utilizar o canal de retorno porque a condição inicial do algoritmo *west-first* seria quebrada, pois o pacote não poderia seguir na direção Oeste depois de utilizar um canal de qualquer outra direção.

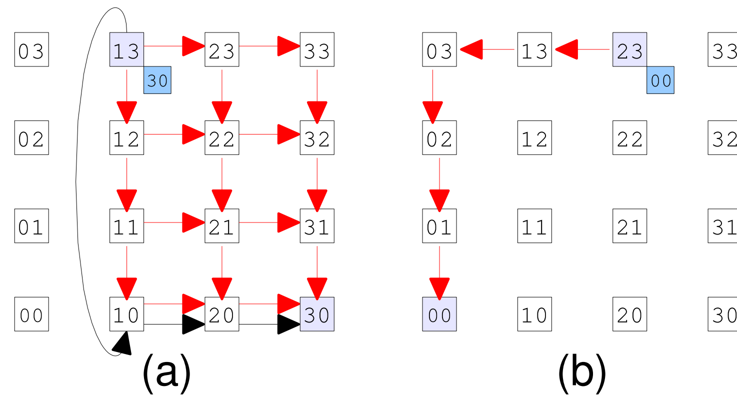

Figura 51 – Exemplos de operação do algoritmo de roteamento *west-first* para redes toro 2D: (a) pacote injetado no nodo 13 com destino ao nodo 30. Nesse caso, é vantajoso e possível utilizar o canal de *wraparound* como primeiro *hop*; (b) pacote injetado no nodo 23 com destino ao nodo 00. Não é possível utilizar o canal de *wraparound* devido à restrição do *west-first*.

7.1 Modelagem e Validação através de Síntese Comportamental

A rede descrita anteriormente foi modelada através de descrições comportamentais SystemC com a metodologia de projeto oferecida pela ferramenta Cynthesizer. A Figura 52 apresenta simplificadamente a arquitetura desse roteador. Todos os roteadores da rede são similares e possuem cinco interfaces de comunicação: Local, Leste, Oeste, Norte e Sul. Todas essas interfaces possuem memorização na entrada e mecanismo de controle de fluxo do tipo *handshake*, na interface Local, para a compatibilidade de interface para a simulação com a adaptação feita na ferramenta ATLAS, e mecanismo de crédito nas demais interfaces. O roteador implementa ainda lógica de arbitragem e roteamento, usando, respectivamente, arbitragem rotativa dinâmica e o algoritmo de roteamento *west-first* para toro 2D, conforme descrito acima. O tamanho do flit para a rede e o tamanho das filas de entrada pode ser variável. A lógica de arbitragem e roteamento são duas *threads* SystemC, que descrevem o comportamento do módulo árbitro e do módulo de roteamento; as portas de saída são *threads* que descrevem comportamentos individuais. O *crossbar* do centro da arquitetura é um conjunto de *threads* que realiza a lógica de cola entre os sinais trocados pelas Entradas, Saídas e a Lógica de Arbitragem e Roteamento.

A interface de saída de cada porta possui um módulo de controle, que recolhe as informações vindas das filas de entrada e transmite os dados para os roteadores vizinhos baseado nas

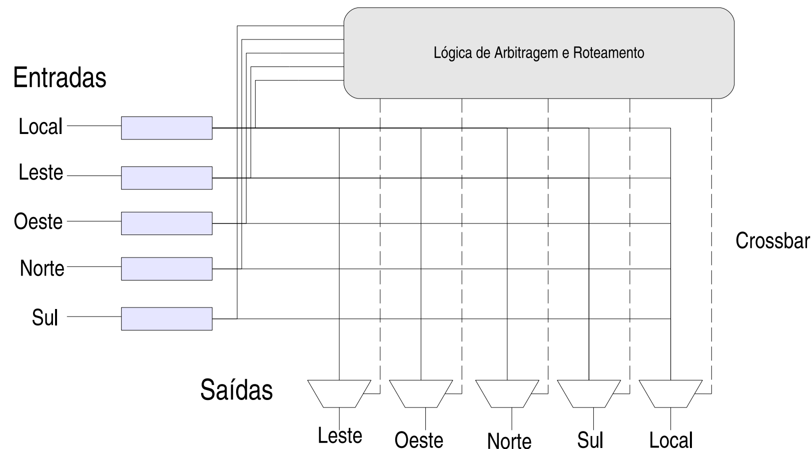

Figura 52 – Arquitetura do roteador para topologia toro 2D.

informações de roteamento vindas do módulo de controle. A Figura 53 apresenta simplificadamente as interfaces de entrada e saída e a conexão entre roteadores vizinhos com *flit* de 16 *bits*, das portas Leste, Oeste, Norte e Sul. Cada interface troca três informações: *tx_rx*, de um bit, que mapeia a sinalização do transmissor *tx* para o sinal de entrada da fila do receptor *rx*, definindo a disponibilidade de um dado para a transmissão; *data*, de 16 bits, que mapeia o dado da interface do transmissor *data_out* para a interface do receptor *data_in* e carrega o flit a ser transportado; *credito*, de um bit, que mapeia a disponibilidade de espaço na fila do receptor *credito_o* para o sinal *credito_i* do transmissor.

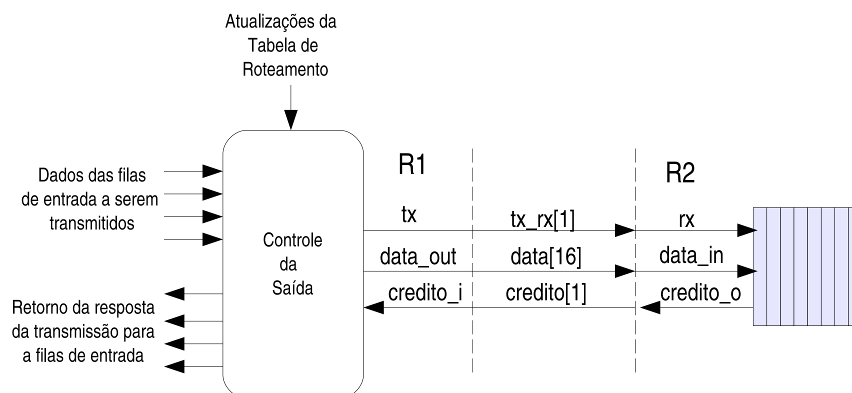

Figura 53 – Interface de comunicação entre roteadores e estruturas responsáveis, sendo R1 o roteador transmissor e R2 o roteador receptor.

Cada uma das filas de entrada se comunica com o módulo árbitro. Sempre que um novo pacote chega a uma fila uma nova requisição de roteamento é gerada. A fila informa que existe um novo pacote e informa o cabeçalho desse pacote ao árbitro e aguarda até receber uma resposta afirmativa para iniciar a transmissão. Os sinais trocados entre a fila e o árbitro são: *h*, em que a fila informa ao árbitro que tem um novo pacote; *h_header*, que a fila informa o cabeçalho

do pacote ao árbitro; *ack_h*, em que o árbitro informa para a fila que a transmissão pode ser iniciada.

Os módulos de arbitragem e roteamento são responsáveis pelo atendimento das requisições e pelo estabelecimento de conexões entre uma porta de entrada e uma porta de saída do roteador. O módulo árbitro recolhe as requisições vindas das interfaces das filas das entradas e garante uma ordem de atendimento justa a todas as requisições. A arbitragem implementada é do tipo rotativa dinâmica, em que a escolha da requisição a ser atendida será baseada em um ordenamento, na qual toda a fila que acaba de ter seu pedido atendido passa a ser a última desse ordenamento. Escolhida uma requisição de roteamento, o árbitro informa ao módulo de controle que tem uma requisição, qual a fila do origem e qual o nodo de destino do pacote. A Figura 54 mostra a interface entre os módulos árbitro e controle. Os sinais trocados entre os módulos são: *req_rot*, em que o árbitro assinala uma requisição ao controle; *incoming*, em que o árbitro informa ao controle qual a fila de origem da requisição; *header*, em que o árbitro envia o cabeçalho do pacote ao controle; *ack_rot*, resposta da requisição dada pelo controle ao árbitro; *ack_rot_vld*, necessário para a sincronização da informação trocada com o árbitro, porque todo esse processo é descrito em nível comportamental.

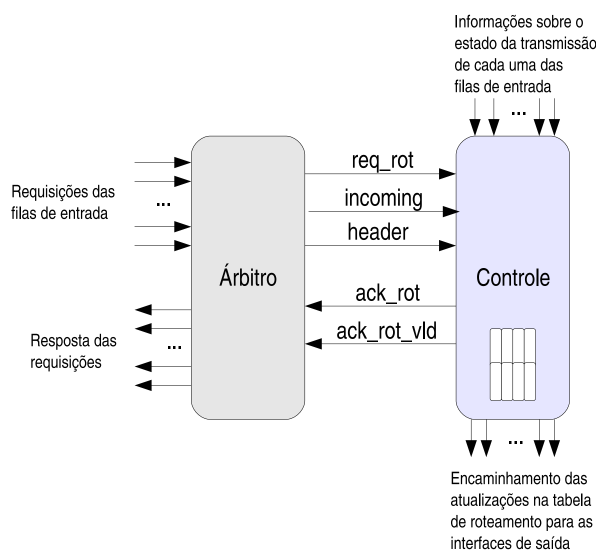

Figura 54 – Interfaces dos módulos árbitro e controle do roteador para topologia toro 2D.

O módulo de controle implementa o algoritmo de roteamento da rede. Ele mantém uma tabela com informações sobre a situação das conexões. Esse módulo atende os pedidos de conexão das filas que são ordenados pelo módulo árbitro, monitora o término das transmissões das filas de entrada e atualiza as informações de roteamento para as interfaces de saída.

Para esse estudo de caso foi elaborado um processo automatizado que instancia uma rede toro 2D de dimensões quaisquer. Essa rede pode ser simulada em nível comportamental e RTL, seja em SystemC, seja em Verilog, gerados pela pela ferramenta *Cynthesizer*. A geração dos

pacotes para simulação foi realizada com o ambiente ATLAS [34]. Com essa ferramenta é possível gerar automaticamente configurações para a rede Hermes [30], HermesTU [36], HermesTB [36] e Mercury [42], e criar cenários de tráfego variados. A ferramenta integra ainda um fluxo de simulação da rede gerada e um método de avaliação baseada na simulação com os cenário de tráfego ao qual a rede foi submetida. Cabe salientar que nesse trabalho o ATLAS é utilizado para gerar os cenários de tráfego e para realizar a avaliação do resultado da simulação. A simulação da rede implementada nesse estudo de caso em si foi realizada no ambiente disponibilizado no Cynthesizer.

Além da rede descrita em VHDL, dos pacotes a serem injetados, a ferramenta produz três módulos SystemC para a avaliação do tráfego da rede: *InputModule*, que injeta pacotes na rede; *OutputModule*, que consome os pacotes da rede; *OutputModuleRouter*, que monitora as interfaces entre os roteadores da rede. Depois da geração do cenário de tráfego, esses três módulos são utilizados na composição do *testbench* e da rede para a simulação no Cynthesizer. A Figura 55 apresenta a organização do roteador e do *testbench* para utilizar os módulos gerados pelo ambiente ATLAS no Cynthesizer. O *testbench* instancia os módulos *InputModule* e *OutputModule* e associa suas interfaces com a interface de simulação da rede toro. A rede instancia o *OutputModuleRouter*, que deve ser marcado no arquivo de projeto (*project.tcl*) como um módulo não sintetizável, e associa as interfaces entre os roteadores aos sinais consumidos pelo módulo.

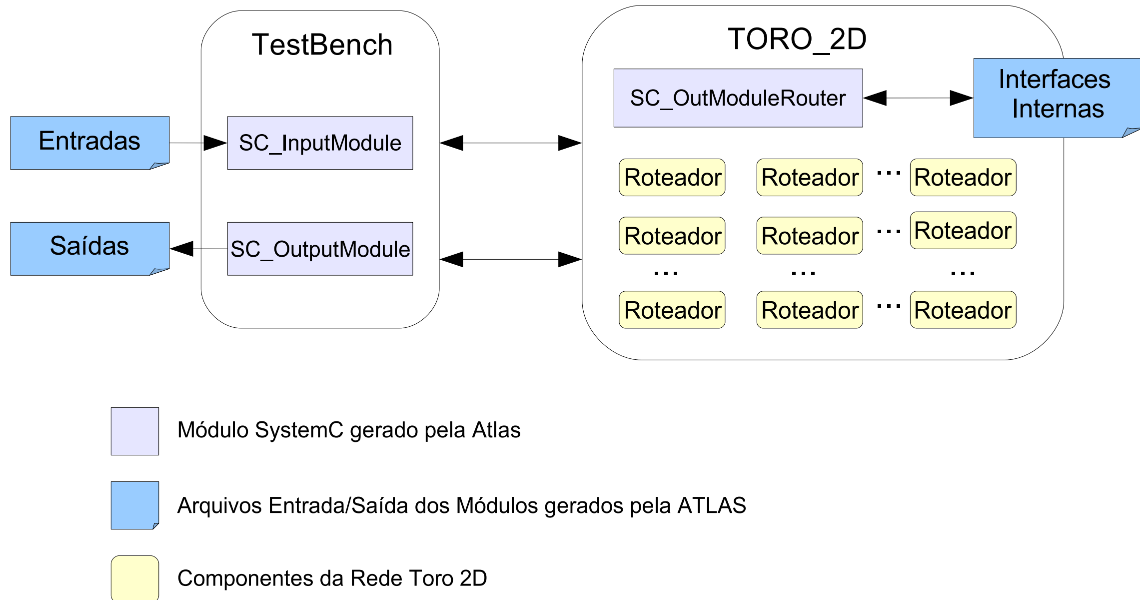

Figura 55 – Esquema para utilização dos módulos e arquivos de entrada gerados pelo ambiente ATLAS na simulação em nível comportamental (SystemC) e RTL (SystemC e Verilog) e os arquivos de saída produzidos na simulação que serão utilizados para a extração dos relatórios no Atlas.

Após a simulação, que é realizada no ambiente do Cynthesizer, são obtidos arquivos com informações da utilização dos canais internos e das interfaces de saída da rede. Esses arquivos são transportados para o ambiente ATLAS para que os relatórios de utilização de canais e

latência possam ser gerados e a NoC avaliada.

7.1.1 Exploração do Espaço de Projeto

Conforme apresentado no Capítulo 4, a exploração do espaço de projeto pode ser realizada modificando construções algorítmicas, diretivas de síntese permitidas pela ferramenta de síntese utilizada e/ou alterando o processo de sincronização entre as threads que compõe as funcionalidades. Um exemplo de exploração de sincronização de sinais pode ser observado no mecanismo de inclusão de dados na fila de entrada de uma porta do roteador. Foram implementados dois mecanismos de sincronização: créditos e *handshake*. Na Tabela 9, é apresentada a codificação em SystemC, aceita pela ferramenta utilizada neste trabalho, para esses mecanismos de sincronização. Também é apresentado na tabela um resumo do relatório disponibilizado pela ferramenta para cada uma das implementações para os critérios: número de ciclos para incluir um dado na fila e LUTS ocupadas no dispositivo alvo da síntese (Virtex II XC2V4000), considerando uma fila de 8 *flits* de 16 *bits*. Quanto ao primeiro critério, observa-se o ganho de um ciclo na sincronização por mecanismo de créditos. Quanto ao segundo, lembrando que o relatório apresenta o número de LUTS do componente *buffer* que contém a entrada e o consumo de dados da fila, o mecanismo de crédito torna o hardware um pouco mais custoso.

Critério	Crédito	Handshake

Código da sincronização da entrada da fila da porta do roteador.	`if(espaco()==1){`
	`   credito_o.write(1);`
	`   do{`
	`       wait(1);`
	`   }while(rx.read()==0);`
	`   i_data=data_in.read();`
	`   buf[last]=i_data;`
	`   credito_o.write(0);`
	`   incrementa_last();`
	`   do{`
	`       wait(1);`
	`   }while(rx.read()==1);`
	`}else{wait(1);}`
		`if(espaco()==1){`
		`   do{`
		`       wait(1);`
		`   }while(rx.read()==0);`
		`   i_data=data_in.read();`
		`   buf[last]=i_data;`
		`   ack_rx.write(1);`
		`   wait(1);`
		`   ack_rx.write(0);`
		`   incrementa_last();`
		`   do{`
		`      wait(1);`
		`   }while(rx.read()==1);`
		`}else{ wait(1);}`

Ciclos para incluir dado na fila	2	3
LUTS ocupadas	243	237

Tabela 9 – Exploração de Espaço de projeto para diferentes sincronizações para a entrada de dados na fila de uma porta de entrada do roteador.

Outro ponto do trabalho em que foi realizado o estudo de diretivas para exploração do espaço de projeto foi o componente de roteamento. O algoritmo de roteamento implementado é composto por duas funções principais: a lógica de roteamento e a consulta e atualização das tabelas de roteamento. A primeira é apresentada na Figura 56 a segunda na Figura 57. As seguintes diretivas foram marcadas no código: 1) diretiva de latência para a parte da decisão de roteamento; 2) diretiva de latência para a função de consulta e atualização das tabelas de roteamento; 3) diretiva para desdobramento do laço para que as operações sejam realizadas em paralelo.

```
CYN_LATENCY(0,1,"T_ROUTER");  (1)
if(inc == 4 ){
	if(add_x==LIM_M &  (h_x==MIN | h_x<(DIM_M/2)))//borda direita
		canais[0]=1;//leste
	else if(add_x==MIN & h_x==LIM_M  )//borda esquerda da rede
		canais[0]=0;//oeste
	else if(add_y==LIM_N & (h_x>= add_x & (h_y==MIN | h_y<(DIM_N/2))) )//extremo norte da rede
		canais[0]=3;//norte
	else if(add_y==MIN & (h_x>= add_x & (h_y==LIM_N | h_y >(DIM_N/2))))//extremo sul da rede
		canais[0]=2;//sul
	else
		controle=1;
}
if(controle==1 | inc==0 | inc==1| inc==2 | inc==3){
	x_offset=h_x.to_int()-add_x.to_int();
	y_offset=h_y.to_int()-add_y.to_int();

	if(x_offset<0)
		canais[0]=0;//sempre para oeste
	if(x_offset>0){
		if(y_offset<0)//leste ou sul
		{
			canais[0]=1;
			canais[1]=2;
		}
		if(y_offset>0)//leste ou norte
		{
			canais[0]=1;
			canais[1]=3;
		}
		if(y_offset==0)//leste
			canais[0]=1;
	}
	if(x_offset==0){
		if(y_offset<0)
		{
			canais[0]=2;
			if(inc==1 & add_x!=MIN)// se veio do leste -> oeste
				canais[1]=0;
		}
		if(y_offset>0){
			canais[0]=3;
			if(inc==1 & add_x!=MIN)// se veio do leste -> oeste
				canais[1]=0;
		}
	}
}
```

Figura 56 – Função de roteamento.

A Tabela 10 apresenta algumas combinações das diretivas que foram realizadas para a exploração do espaço de projeto do componente de roteamento. No primeiro cenário, sem a aplicação de qualquer diretiva, a ferramenta escalonou as operações em 469 LUTS, mas com uma operação que responde a requisição de roteamento entre 5 a 16 ciclos de relógio. Com a aplicação de todas as diretivas assinaladas no código (cenário 2), a operação é escalonada em 350 LUTS e responde sempre em 4 ciclos de relógio. Aplicando-se todas as diretivas, porém, modificando a diretiva 1 para operar em até 3 ciclos de relógio, obtem-se um gasto significativo em LUTS, em relação ao cenário anterior, para atender a diretiva. Com isso, para o terceiro

```
CYN_LATENCY(0,1,"CANAIS_R_LATENCY");  2
for(sc_uint<3> i; i<N; i++)
{
        CYN_UNROLL(COMPLETE,N,"CANAIS_R_UNROLL");  3
        if(canais[i]!=7){
                if(canais[i]!=inc){
                        if(f_livre(canais[i])==1){
                                f_tab_rot(inc, canais[i]);
                                a_rot=1;
                                break;
                        }
                }
        }
}
```

Figura 57 – Consulta e atualização das tabelas de roteamento.

cenário, existe um incremento de até dois ciclos em relação ao segundo cenário para responder a requisição de roteamento. Para o quarto cenário apresentado, sem a inclusão da diretiva de latência no cálculo do roteamento, a ferramenta escalona as operações em 392 LUTS, só que com um custo de 5 a 14 ciclos de relógio para responder à requisição de roteamento. No quinto cenário apresentado, é aplicada apenas a diretiva 1 e a ferramenta escalona as operações em 421 LUTS com tempos de resposta entre 3 e 10 ciclos de relógio. Para o sexto cenário, é suprimida a diretiva 2, e a ferramentas escalona as operações em 357 LUTS, com um custo de 4 a 6 ciclos de relógio para responder às requisições de roteamento. Para ilustrar cada cenário apresentado, foi realizada a simulação do comportamento da rede e recortado o comportamento do roteador de endereço 00 em uma rede toro 4x4. Nesse roteador, acontece uma requisição da porta Local(4) com destino ao roteador 11; outra requisição da porta Norte (3) com destino à 00; uma requisição vinda do canal de *wraparound*, correspondente a porta Oeste (0) do roteador, com destino ao roteador 22; e outra requisição vinda da porta Leste (1) com destino à 00.

Cenário	Combinação das Diretivas	LUTS ocupadas para o dispositivo Virtex II XC2V4000	Ciclos para resposta da requisição de roteamento	Simulação
1	sem diretivas	469	de 5 a 16 ciclos	Fig. 58
2	1, 2 e 3	350	sempre em 4 ciclos	Fig. 59
3	1(LATENCY de 0 a 3 ciclos), 2 e 3	564	de 4 a 6 ciclos	Fig. 60
4	2 e 3	392	de 5 a 14 ciclos	Fig. 61
5	1	421	de 3 a 10 ciclos	Fig. 62
6	1 e 3	357	de 4 a 6 ciclos	Fig. 63

Tabela 10 – Resumo do relatório apresentado pela ferramenta para os cenários aplicados a *thread* de roteamento.

Para a função de roteamento do roteador dessa implementação foi escolhida a opção do segundo cenário apresentada, que ocupa 350 LUTS e consegue responder à requisição de rote-

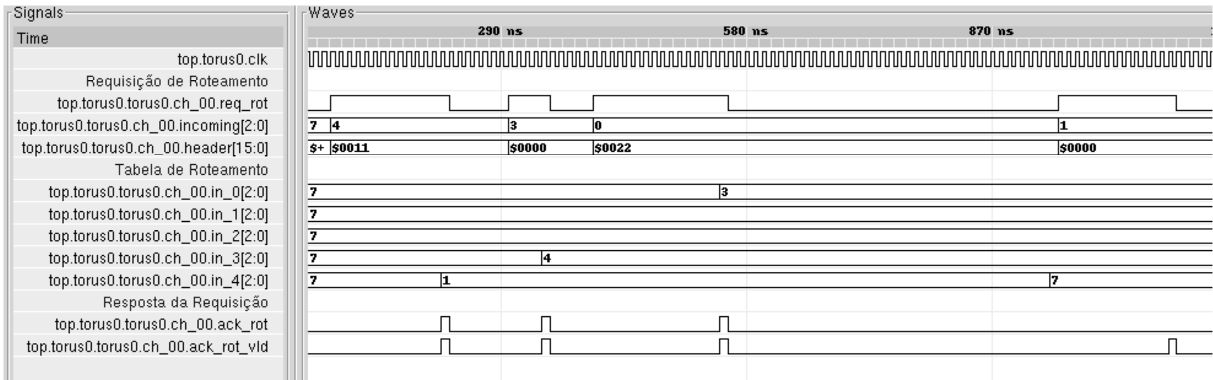

Figura 58 – Simulação do roteamento sem as diretivas de síntese. A resposta da requisição de roteamento leva de 5 a 16 ciclos.

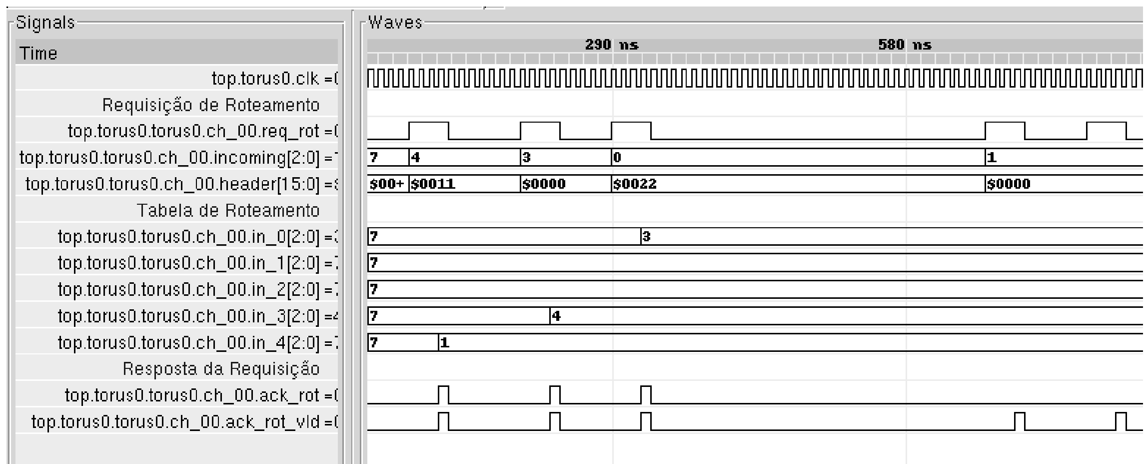

Figura 59 – Simulação do roteamento aplicando-se todas as diretivas marcadas no código. A resposta da requisição de roteamento sempre 4 ciclos.

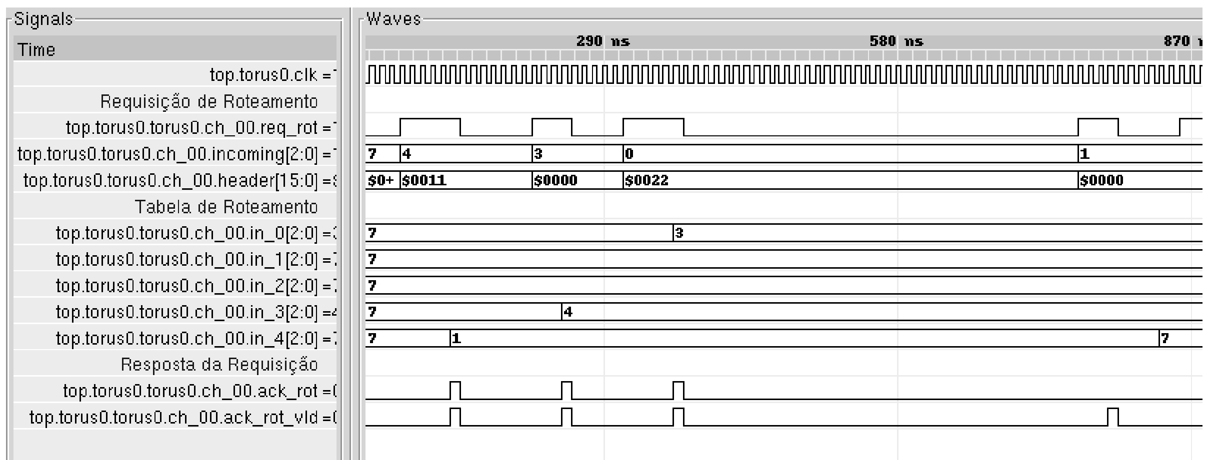

Figura 60 – Simulação modificando a diretiva 1 marcada no código para executar a operação em até 3 ciclos. A resposta da requisição de roteamento leva de 4 a 6 ciclos.

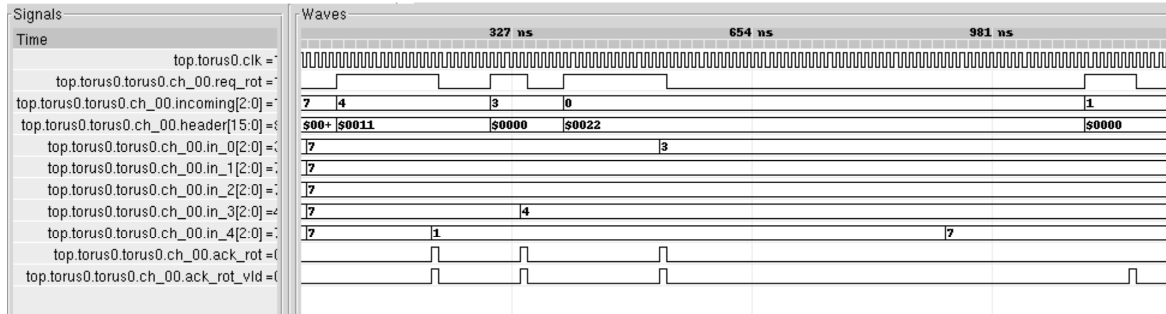

Figura 61 – Simulação suprimindo a diretiva 1 marcada no código. A resposta da requisição de roteamento leva de 5 a 10 ciclos.

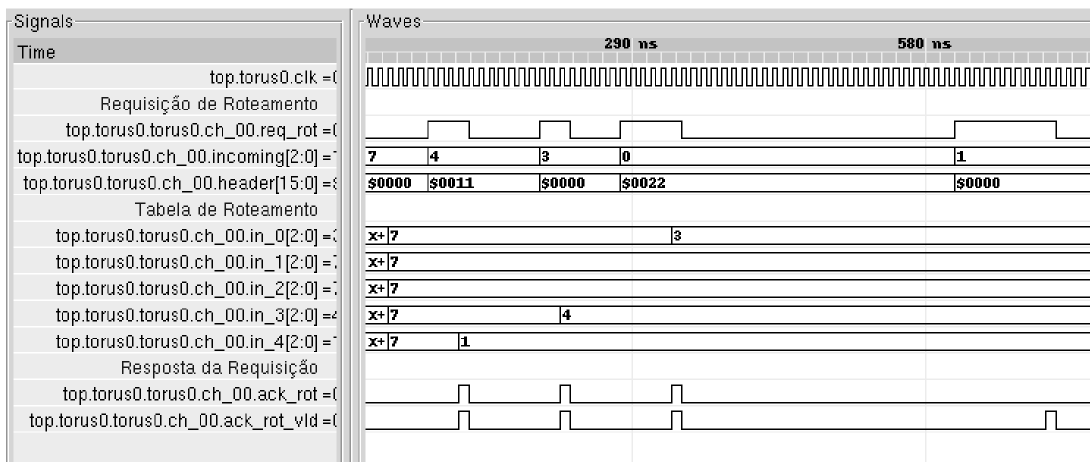

Figura 62 – Simulação suprimindo as diretivas 2 e 3 assinaladas no código. A resposta da requisição de roteamento leva de 3 a 6 ciclos.

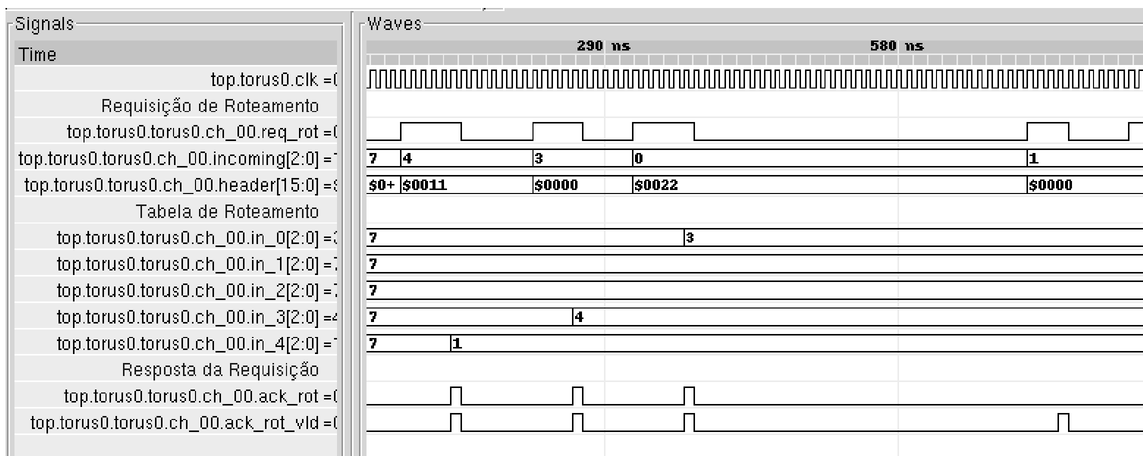

Figura 63 – Simulação suprimindo a diretiva 2 assinalada no código. A resposta da requisição de roteamento leva de 4 a 6 ciclos.

amento sempre em quatro ciclos. A porta Local do roteador tem sincronização de entrada de dados do tipo *handshake*, somando ao roteador 237 LUTS. As demais portas possuem sincroni-

zação por mecanismo de crédito e adicionam ao roteador um total de 1.209 LUTS. O roteador possui em torno de 2.138 LUTS no total (considerando *buffers* com 8 *flits* de 16 *bits*), com 1.559 distribuídas nas portas de entrada e na função de roteamento. As 579 LUTS restantes correspondem a sincronização da *thread* de arbitragem, a sincronização das portas de saída e a lógica de cola entre os sinais internos das portas de entrada com as portas de saída. Conforme apresentado, a codificação em SystemC para garantir o funcionamento adequado do componente descrito resulta em processos que levam vários ciclos para conseguir sincronizar. Assim, um flit desde que entra no roteador leva em média 6 ciclos para chegar a uma das interfaces de saída do roteador. Toda essa lógica para sincronização dos sinais consome muitos recursos de hardware. A ferramenta realiza otimizações muito boas em algoritmos puros, como é o caso do cálculo de roteamento, mas não consegue realizar otimizações para as interfaces de comunicação entre *threads* ou componentes. Por conseguinte, a utilização da síntese comportamental oferecida pelo *Cynthesizer* não é adequada para descrever processos típicos de redes de comunicação intrachip, uma vez que esses componentes tem como característica um volume grande de sincronização de sinais e pouca lógica de roteamento. Contrastando com esse fato, está a facilidade para modelagem e validação oferecida pela ferramenta para que estruturas de comunicação intrachip possam ser modeladas e que algoritmos possar ser testados e validados, como o que foi realizado neste trabalho. As próximas Seções deste Capítulo tratam dos resultados obtidos para a rede modelada neste estudo de caso.

7.1.2 Avaliação de Área da NoC Gerada com Síntese Comportamental

Com o processo automatizado implementado para esse estudo de caso, foram geradas redes de dimensões 2x2 e 3x3 com filas de 8 e 16 *flits*. Na implementação foram marcadas as diretivas para condução da síntese comportamental, de forma que os laços sejam desenrolados e que as restrições de latência específicas para certos pontos do projeto sejam atendidas.

As redes foram sintetizadas para ASIC com a biblioteca de tecnologia TSMC 0.18μm e foram geradas as descrições RTL SystemC e Verilog e os relatórios com as informações relativas ao processo de síntese. Para ilustrar simplificadamente que informações são apresentadas nos relatórios gerados pelo Cynthesizer tem-se: a Figura 64 apresenta um resumo das informações contidas no relatório de síntese para os módulos *buffer* e *buffer_c*, que implementam respectivamente o controle de entrada com *handshake* e com mecanismo de créditos, com capacidade de 8 flits de 16 bits. Ambos implementam uma fila linear para armazenamento dos *flits* do pacote. As Figuras 65 e 66 apresentam o relatório do mapeamento dos módulos extraídos na síntese comportamental dos módulos *buffer* e *buffer_c* com os elementos disponíveis na biblioteca de tecnologia e os que foram gerados automaticamente pela ferramenta.

A Figura 67 apresenta o relatório resumido disponibilizado pela ferramenta para a rede 2x2 com *buffers* de 8 *flits* de 16 bits. Para a síntese comportamental são apresentadas duas

Synthesis Report

cynthModule buffer

cynthConfig name	last run	area (square µm)				register bits	total run time
		combinational	memory	register	total		
BASIC	Jul-28 14:15	14026.1	0.0	19489.4	33515.5	247	00:00:21

cynthModule buffer_c

cynthConfig name	last run	area (square µm)				register bits	total run time
		combinational	memory	register	total		
BASIC	Jul-28 14:16	13686.1	0.0	19412.9	33099.0	246	00:00:21

Figura 64 – Relatório da síntese dos módulos *buffer* e *buffer_c*, que são reponsáveis pelo armazenamento e controle das entradas no roteador, ambos com capacidade para 8 flits de 16 bits.

Resources Used

Resources used by all threads

module name	resource kind	library	total area (square µm)	number used	unit area (square µm)	delay	pipeline latency
buffer_DataMux_16U_0	AUTOGEN	buffer_DataMux_16U_0	2571.3	1	2571.3	2.30	none
Add16	FUNCTION	typ	1167.6	1	1167.6	2.45	none
Eq16	FUNCTION	typ	582.1	1	582.1	0.50	none
And32	FUNCTION	typ	532.2	1	532.2	0.08	none
Add4	FUNCTION	typ	499.0	2	249.5	0.59	none
Add5	FUNCTION	typ	329.3	1	329.3	0.75	none
Eq4	FUNCTION	typ	286.1	2	143.0	0.34	none
Eq5	FUNCTION	typ	173.0	1	173.0	0.36	none
Ne4	FUNCTION	typ	136.4	1	136.4	0.27	none
RedOr4	FUNCTION	typ	26.6	1	26.6	0.18	none
RedNor1	FUNCTION	typ	6.7	1	6.7	0.01	none
reg_bit	REGISTER	NONE	19489.4	247	78.9	0.00	none

Figura 65 – Mapeamento dos elementos extraídos da síntese comportamental do *buffer* com os elementos contidos na biblioteca de tecnologia.

configurações: BASIC, que traz informações sobre a síntese para ASIC e FLATTENED, que traz informações sobre a síntese para FPGA. Também está na Figura o relatório da Síntese Lógica, que foi realizada com Synplify Pro e do P&R realizado com o ISE.

A Tabela 11 apresenta as informações gerais sobre a área da rede implementada com a síntese realizada para ASIC (TSMC 0.18 μm). Para a rede de dimensão 2x2, o aumento do *buffer* de 8 para 16 *flits* causou um acréscimo de 303567,56 μm^2 em área, que corresponde a um aumento de 34,55%. Aumento de proporção semelhante ocorreu na rede de dimensões 3x3, com o aumento do *buffer* de 8 para 16 *flits*.

Dimensão	Tamanho dos buffers	Tamanho total em μm^2
2x2	8	878624,30
2x2	16	1182191,86
3x3	8	1976904,71
3x3	16	2659931,69

Tabela 11 – Informações relativas às áreas das NoCs geradas, extraídas do relatório disponibilizado pelo Cynthesizer.

Para a síntese para FPGA foi escolhido o dispositivo Virtex II XC2V4000(*speed grade* -

Resources Used

Resources used by all threads

module name	resource kind	library	total area (square μm)	number used	unit area (square μm)	delay	pipeline latency
buffer_c_DataMux_16U_0	AUTOGEN	buffer_c_DataMux_16U_0	2571.3	1	2571.3	2.30	none
Add16	FUNCTION	typ	1167.6	1	1167.6	2.45	none
Eq16	FUNCTION	typ	582.1	1	582.1	0.50	none
And32	FUNCTION	typ	532.2	1	532.2	0.08	none
Add4	FUNCTION	typ	499.0	2	249.5	0.59	none
Add5	FUNCTION	typ	329.3	1	329.3	0.75	none
Eq4	FUNCTION	typ	286.1	2	143.0	0.34	none
Eq5	FUNCTION	typ	173.0	1	173.0	0.36	none
Ne4	FUNCTION	typ	136.4	1	136.4	0.27	none
RedOr4	FUNCTION	typ	26.6	1	26.6	0.18	none
RedNor1	FUNCTION	typ	6.7	1	6.7	0.01	none
reg_bit	REGISTER	NONE	19412.9	246	78.9	0.00	none

Figura 66 – Mapeamento dos elementos extraídos da síntese comportamental do buffer_c com os elementos contidos na biblioteca de tecnologia.

5) da XILINX para caracterização da biblioteca para a síntese lógica. A rede foi gerada com dimensões 2x2, 3x3 e 4x4 com *buffers* de profundidade 8 e 16 *flits*. A partir daí foi realizada a síntese comportamental e a síntese lógica das redes para o dispositivo escolhido. A Tabela 12 apresenta os resultados relativos à ocupação do dispositivo para cada rede em relação a sua dimensão. Para a rede 2x2 o aumento dos *buffers* causou um acréscimo de 19,62% da ocupação de LUTS do dispositivo. Para a rede 3x3, o aumento do tamanho dos *buffers* gerou um acréscimo de 18,3% em LUTS. Para a rede de dimensão 4x4 o aumento do tamanho dos *buffers* de 8 para 16 *flits* gerou um acréscimo de 17.48% em LUTS.

Dimensão da rede	Tamanho dos *buffers*	Ocupação do dispositivo
2x2	8	18% (8457 LUTS)
2x2	16	21% (10116 LUTS)
3x3	8	41% (19185 LUTS)
3x3	16	48,5%(22696 LUTS)
4x4	8	74% (34222 LUTS)
4x4	16	87% (40205 LUTS)

Tabela 12 – Ocupação do FPGA de cada rede em relação a sua dimensão, após a síntese comportamental visando o Virtex II XC2V4000(*speed grade* -5) da XILINX.

O trabalho de Scherer [36] apresenta uma avaliação da implementação em VHDL de uma rede toro, com chaveamento *wormhole* e algoritmo de roteamento *west-first* adaptado para a rede toro bidirecional, semelhante à implementada neste trabalho. No trabalho é avaliada a ocupação em LUTS da rede descrita em relação às redes Hermes, Hermes com 2 canais virtuais e Hermes Toro unidirecional, para flits de 8, 16, 32 e 64 bits e tamanho de fila de 4, 8, 16 e 32 *flits*. A Tabela 13 apresenta uma comparação entre os resultados obtidos entre o processo com descrições e síntese RTL obtidos no trabalho de Scherer [36] e os obtidos através do processo de síntese comportamental para rede 4x4 com flit de 16 bits e tamanhos de fila 8 e 16 *flits*

Synthesis Report

cynthModule torus

cynthConfig name	last run	area (square µm)				register bits	total run time
		combinational	memory	register	total		
BASIC	Jul-30 09:23	400504.9	0.0	478123.4	878624.3	6008	00:03:59
C_FLATTENED	Jul-30 08:44	7670.6	0.0	0.0	7666.6	6676	00:03:53

FPGA Logic Synthesis Report

logicSynthConfig name	last run	module	Resources	worst case slack	total run time
FPGA_FLAT	Jul-30 08:57	torus	Device: xc2v4000ff1152-6I/O Register bits 136Register bits 6337 (13.0)%Total LUTs 8457 (18.0)%	9.595ns	0:02:18

Place and Route Report

pnrConfig name	last run	module	Resources	Timing	total run time
PNR	Jul-30 09:08	torus	Device: xc2v4000ff1152-6Total LUTs 8407 of 46080 (18.0)%Occupied Slices 6527 of 23040 (28.0)%Slice Flops 6337 of 46080 (13.0)%IOB Flops 136	Requested period 20.0nsMinimum period 11.203nsMinimum input->clk 12.481nsMinimum clk->output 6.734ns	0:09:44

Figura 67 – Relatório da síntese para ASIC e para FPGA para a rede 2x2 com buffers de 8 flits de 16 bits.

sintetizados para o mesmo dispositivo (Virtex II XC2V4000(*speed grade* -5)). Para a fila com tamanho 8 *flits* o resultado obtido através do processo de síntese comportamental apresenta um acréscimo de 114,96% em relação à ocupação em LUTS do dispositivo obtida através do processo de descrição e síntese RTL tradicional. Para a fila com tamanho 16 *flits* o acréscimo é de 145,14% em LUTS.

Implementação	Tamanho dos *buffers* - 8 flits	Tamanho dos *buffers* - 16 flits
Comportamental	34222 LUTS	40205 LUTS
RTL	15920 LUTS	16401 LUTS

Tabela 13 – Comparação entre resultados obtidos com a síntese comportamental e com um processo de síntese RTL tradicional.

7.1.3 Avaliação de Latência da NoC Gerada com Síntese Comportamental

Para auxiliar a avaliação da qualidade do resultado obtido pela síntese comportamental em termos de latência da rede gerada foi utilizado o ambiente ATLAS, conforme exposto anteriormente. Para efeitos de comparação dos resultados, a Tabela 14 mostra a comparação das latências médias obtidas de cenário de tráfego com NoC 4x4, com distribuição uniforme, trá-

fego do tipo complemento e tamanho do pacote de 16 *flits*. Tal cenário foi simulado com a NoC Hermes TB do trabalho de Scherer [36] e com a NoC produzida neste trabalho. Claramente observa-se que a qualidade do resultado gerado não tem bom desempenho em termos de latência. Conforme mostrados no Capítulo 4, Seção 4.3, toda a troca de dados entre as *threads* que compõe o roteador da NoC inserem ciclos de relógio e sinais de controle para sincronização da comunicação, o que acarreta em custo de latência para a transmissão de dados. Para cada sincronização entre as *threads* internas do roteador para a transmissão de um *flit* são perdidos entre 2 e 3 ciclos de relógio, o que gera um grande impacto na latência final da transmissão do pacote.

Pares Origem Destino	Número de *hops* entre origem e destino	Número médio de ciclos de relógio	
		Hermes TB	Implementação deste Trabalho
33-00	4	63	163
00-33	4	66	159
01-32	2	36	129
11-22	2	33	131
13-20	2	38	131
20-13	4	66	167

Tabela 14 – Comparação das latências médias com as obtidas no trabalho de Scherer [36].

Para a avaliação, também foi gerada uma rede toro 7x6, com tamanho de fila de 16 flits. Para a simulação foi gerado um cenário de tráfego com distribuição normal e destino aleatório, com cada nodo enviando 10 pacotes, cada pacote com 16 *flits*. A Tabela 15 apresenta os valores para as 10 menores latências obtidas na simulação e a Tabela 16 apresenta as 10 maiores latências para o mesmo cenário.

Latência	Origem	Destino
115	10	15
117	01	00
117	32	22
117	22	32
117	25	35
119	61	51
119	02	12
119	61	62
119	54	44
121	04	14

Tabela 15 – As 10 menores latências obtidas na simulação do cenário com a rede 7x6 com *buffer* de 16 flits.

Latência	Origem	Destino
691	51	05
571	04	62
543	41	02
527	61	32
513	65	21
511	51	42
489	05	60
481	31	03
481	53	32
473	42	23
457	45	63

Tabela 16 – As 10 maiores latências obtidas na simulação do cenário com a rede 7x6 com *buffers* de 16 flits.

A Figura 68 apresenta os caminhos seguidos por alguns dos pacotes dessa simulação. O pacote com maior latência da simulação, que é injetado no nodo 51 e tem como destino o nodo 05, passa por 9 *hops* até chegar a seu destino. Os dois pacotes que tem como origem-destino, respectivamente, 53 para 32 e 05 para 60, levam 5 e 7 *hops* para chegar ao seu destino. O pacote com menor latência, com origem no nodo 10 e destino 15, leva 1 *hop* para alcançar seu destino.

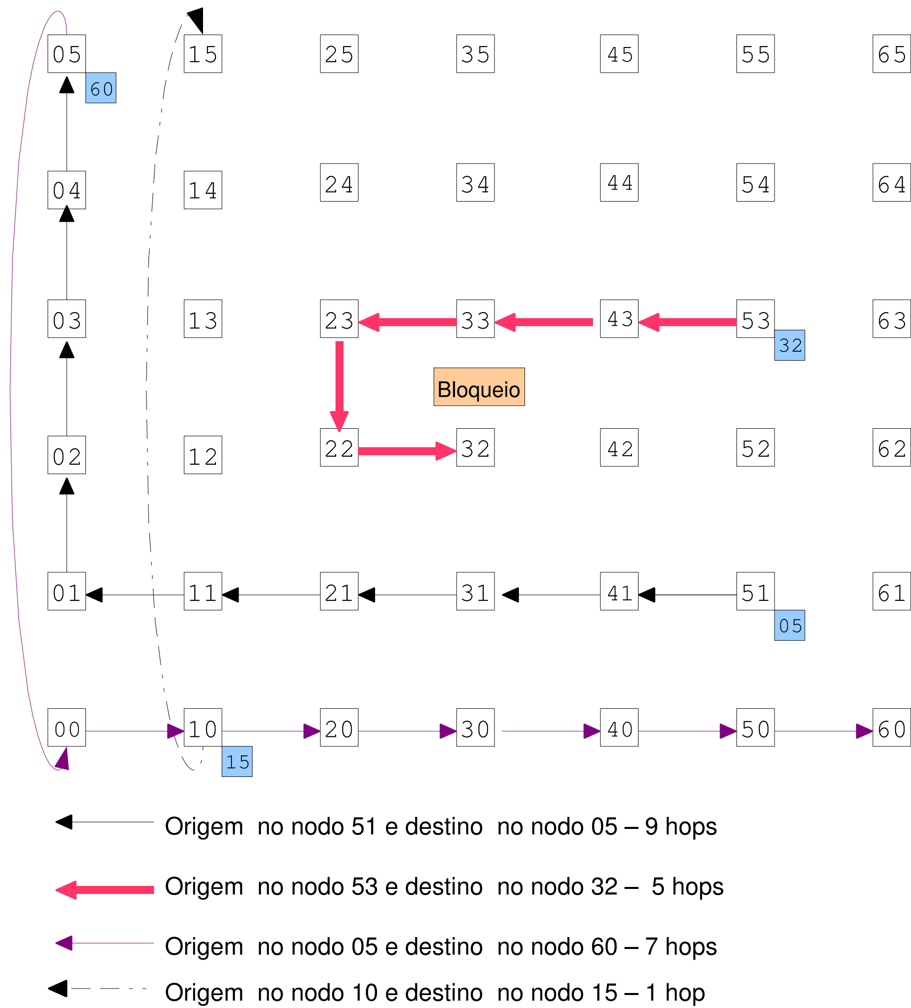

Figura 68 – Rotas de alguns dos pacotes do exemplo de simulação. Cabe salientar que essas rotas foram obtidas na simulação com uma rede congestionada.

8 Rede Toro 2D com Canais Virtuais

Este Capítulo apresenta um projeto de rede toro 2D usando um algoritmo de roteamento com a codificação proposta por Duato, Ni e Yalamanchili. Embora tal adaptação resulte em estruturas de comunicação intrachip bastante custosas, já que o gerenciamento de canais virtuais insere um custo em área e latência adicional, o estudo de caso agrega a complexidade necessária para a avaliação da síntese comportamental como instrumento de modelagem e validação dessas estruturas de comunicação.

Para tanto foi necessária a criação do algoritmo de roteamento Sul-Oeste, complementar ao Leste-Norte proposto por Duato e apresentado na Seção 6.1, Subseção 6.1.2. O algoritmo Leste-Norte encaminha os pacotes primeiro para os canais do Leste, se for o caso, e depois para os canais do Norte, se for necessário, até alcançar o seu destino, percorrendo canais em ordem crescente. O algoritmo complementar Sul-Oeste encaminha os pacotes primeiro para os canais do Sul, se necessário, e depois para Oeste, com ordenamento decrescente de canais. No algoritmo Sul-Oeste a codificação proposta por Duato, Ni e Yalamanchili é respeitada e a prova de que o mesmo é livre de *deadlock* é análoga a do algoritmo Leste-Norte. A codificação dos canais para o algoritmo Sul-Oeste é apresentada no exemplo da Figura 69. O algoritmo Sul-Oeste é apresentado na Figura 70. A Figura 71 apresenta um exemplo da execução do algoritmo Sul-Oeste para uma rede toro 4x4.

Na adaptação realizada para a rede toro 2D bidirecional, os dois algoritmos são utilizados para compor o processo de roteamento da rede. Porém, um pacote que é injetado na rede só pode trafegar obedecendo a apenas um dos algoritmos. A decisão de qual algoritmo seguir deve ser tomada no roteador de origem do pacote. Sendo assim, a prova de que a solução é livre de *deadlock* é a prova de que cada um dos algoritmos, Leste-Norte e Sul-Oeste, é livre de *deadlock*. Como os roteamentos ocorrem como se fossem em redes virtuais diferentes, os pacotes ficam confinados na rede virtual do algoritmo escolhido na origem, não disputando os canais da rede virtual do algoritmo complementar.

A decisão tomada na origem de qual algoritmo o pacote deve respeitar pode ser realizada em função de diferentes critérios como congestionamento na origem, menor distância, nodos com falhas, etc. Essa decisão insere uma certa adaptabilidade ao roteamento. Como exemplo, considere-se uma rede toro 4x4 bidirecional usando como critério de decisão o grande congestionamento do nodo de origem: se não há congestionamento dos canais, o pacote pode seguir segundo o algoritmo Leste-Norte. Se houver congestionamento segue-se pela rede virtual do algoritmo Sul-Oeste. Isto é ilustrado na Figura 72 (a). Se o nodo 11 tem um pacote injetado na rede com destino ao nodo 32 e os canais da rede virtual Leste-Norte não estão bloqueados, o

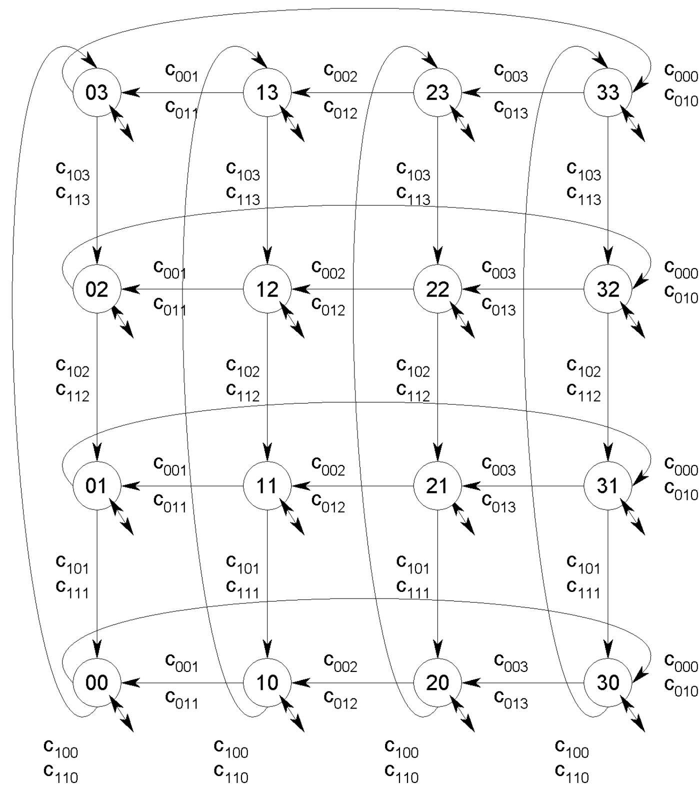

Figura 69 – Toro 4x4 unidirecional com canais numerados de acordo com o método de Duato, Ni e Yalamanchili para o algoritmo complementar Sul-Oeste.

```
Xoffset = Xdest - Xcurrent
Yoffset = Ydest - Ycurrent
Se Yoffset < 0
        Canal = 10
Se Yoffset > 0
        Canal = 11
Se Yoffset = 0
        Se Xoffset < 0
                        Canal = 00
        Se Xoffset > 0
                        Canal = 01
        Se Xoffset = 0
                        Canal = Interno
```

Figura 70 – Algoritmo Sul-Oeste, complementar ao algoritmo Leste-Norte, respeitando a codificação estabelecida por Duato, Ni e Yalamanchili.

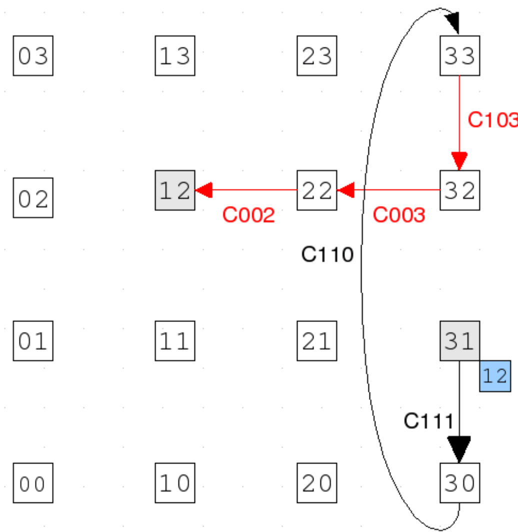

Figura 71 – Exemplo da alocação dos canais na execução do algoritmo Sul-Oeste em uma rede toro 2D 4x4, com origem no nodo 31 e destino 12.

pacote segue por esse caminho. O pacote percorre seu caminho em X e depois em Y. Porém, se um pacote é injetado e o canal da saída Leste do roteador origem está ocupado, o pacote pode seguir pela rede virtual Sul-Oeste. É o caso apresentado na Figura 72 (b), em que o nodo 21 tem um pacote injetado com destino ao nodo 33 após iniciar a transmissão mostrada na Figura 72 (a). Como o canal do caminho Leste-Norte está alocado para outra comunicação, o pacote injetado no nodo 21 segue pelo caminho determinado no algoritmo Sul-Oeste. Caso o congestionamento ocorra nas duas redes virtuais, o algoritmo de roteamento vai encaminhar o pacote assim que um dos caminhos da origem for liberado.

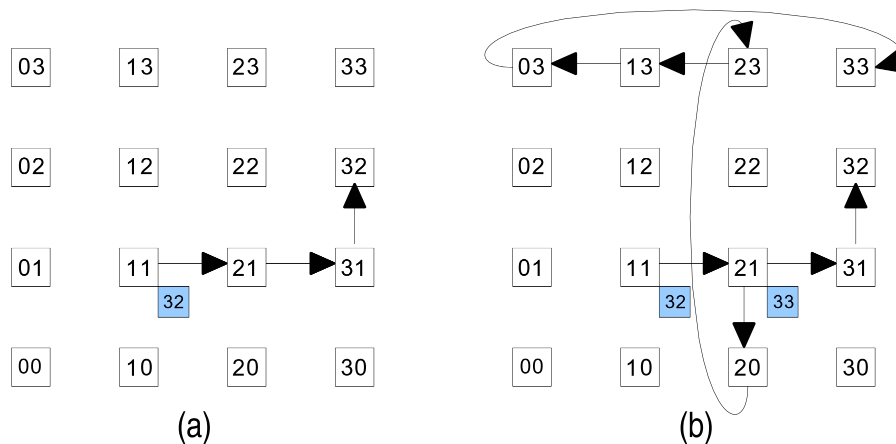

Figura 72 – Exemplo de funcionamento do esquema de roteamento com os algoritmos Leste-Norte e Sul-Oeste: (a) rota do pacote que é injetado na rede pelo roteador 11 e tem como destino 32. O roteameto é Leste-Norte; (b) o pacote injetado na rede por 21 com destino 33 segue o caminho Sul-Oeste, devido ao congestionamento da porta de saída Leste do roteador 21 por outro pacote.

Porém, se o critério de decisão para o algoritmo adaptado de Duato for a menor distância entre os roteamentos disponíveis, o roteador tem o custo adicional para realizar tal cálculo. É necessário que o mesmo seja capaz de calcular a menor distância relativa ao destino para cada um dos roteamentos, Leste-Norte e Sul-Oeste, e escolher a mais vantajosa. A Figura 73 apresenta essa situação. O nodo 21 tem um pacote injetado com destino 02. Se o pacote seguir a rota de Leste-Norte chegará ao destino em 3 *hops*. Se o pacote seguir pela rota de Sul-Oeste alcançará o destino em 5 *hops*.

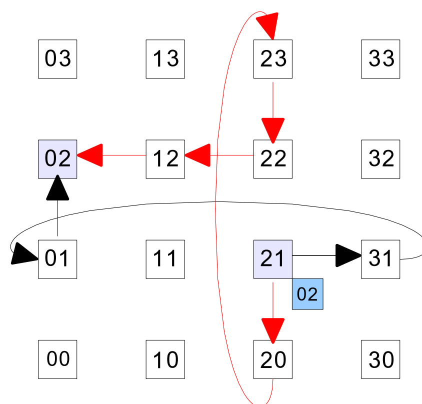

Figura 73 – Rotas dos algoritmos Leste-Norte (setas escuras) e Sul-Oeste (setas claras) com origem em 21 e destino 02.

A decisão tomada na origem deve ser escolhida de acordo com a aplicação a qual a rede deverá dar suporte. Cabe salientar que se for inserida alguma adaptatividade, como o caso da decisão baseada no congestionamento da origem, os pacotes de uma determinada origem e mesmo destino podem chegar fora de ordem ao nodo final.

Neste trabalho, o critério para a implementação da escolha do caminho do pacote foi o congestionamento no nodo de injeção do pacote na rede. A Figura 74 apresenta o pseudo-código simplificado do processo de roteamento implementado. Neste processo, após a requisição do árbitro, se o canal de entrada do pacote for um dos canais de injeção na rede (canais da porta local) é obtido o canal de saída do pacote através do algoritmo Leste-Norte. Porém, se o canal retornado por esse algoritmo estiver ocupado, o canal de saída do pacote é dado pelo algoritmo Sul-Oeste. Se o pacote chegar ao roteador por qualquer canal que não seja um canal de injeção, ele continua trafegando na rede segundo o algoritmo que foi estabelecido em seu nodo de origem até chegar ao seu destino. Ou seja, se o pacote chegar ao roteador por canais do Sul e do Oeste, continua o caminho dado pelo algoritmo Leste-Norte, sempre indo para Norte ou Leste. Se o pacote chegar ao roteador por canais do Norte ou Leste, continua sua rota pelo algoritmo Sul-Oeste, indo sempre para Sul ou para Oeste.

```
...
requisicao_arbitro() // recebe as informações da requisição do árbitro
c = canal_entrada_pacote() // retorna o canal de entrada do pacote
Se  c = =  canal_injecao()  // canal da porta local
        canal_saida = x_y()  //retorno da função XY
        Se canal_saida ocupado
                canal_saida = y_x()  // retorno da função YX
Senão
        Se c = = canais de entrada do  sul ou do oeste //  pacote em rotemento XY
                canal_saida = x_y() // continua respeitando XY
        Se c= = canais de entrada do norte ou do leste // pacote em roteamento YX
                canal_saida = y_x() // continua respeitando YX

atualiza_tabela_roteamento()
responde_requisicao_arbitro()
informa_tabela_controle_saidas()
...
```

Figura 74 – Algoritmo de roteamento implementado para o estudo de caso deste Capítulo.

8.1 Modelagem e Validação Através de Síntese Comportamental

A rede toro 2D com canais virtuais foi modelada através de descrição comportamentais em SystemC, utilizando a metodologia de projeto oferecida pela ferramenta Cynthesizer. Todos os roteadores da rede são similares e possuem cinco interfaces de comunicação: Local, Leste, Oeste, Norte e Sul. As interfaces Leste, Oeste, Norte e Sul possuem dois canais virtuais com memorização somente na entrada e controle de fluxo através do mecanismo de créditos. A interface Local possui apenas um canal de entrada com memorização na entrada e mecanismo de controle de fluxo do tipo *handshake*, para manter a compatibilidade com a interface da rede Hermes para a utilização das mesmas ferramentas de avaliação. O roteador contém ainda lógica de arbitragem e roteamento que implementam, respectivamente, arbitragem rotativa dinâmica e o algoritmo de roteamento para toro 2D adaptado do algoritmo de Duato. O tamanho do *flit* para a rede é de 16 bits e o tamanho da fila nas entradas é de 8 *flits*. A Figura 75 apresenta simplificadamente a arquitetura desse roteador.

Cada módulo do roteador foi descrito em nível comportamental: interfaces de comunicação, arbitragem e roteamento e demais *threads* necessárias a sincronização dos sinais de comunicação entre esses módulos. Para as interfaces de comunicação com memorização na entrada, duas filas do tipo FIFO foram implementadas: uma para controle de fluxo com mecanismo de aperto de mão e outra para o controle de fluxo com mecanismo de crédito.

A interface de comunicação de uma porta para comunicação entre roteadores, ou seja, inter-

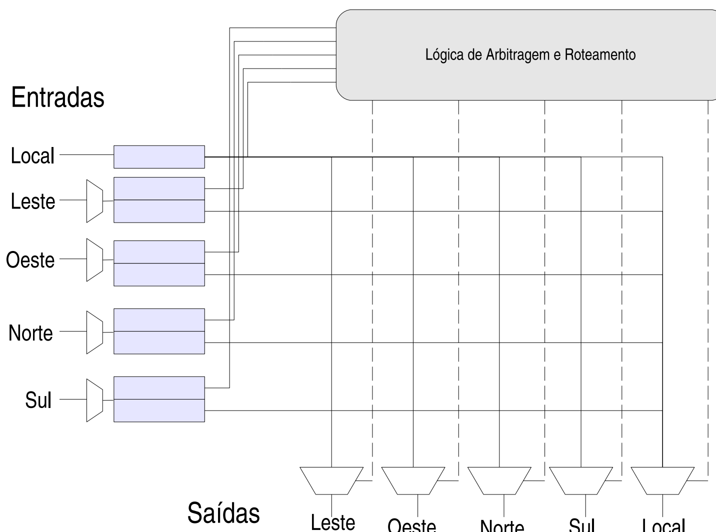

Figura 75 – Arquitetura do roteador da rede toro 2D com canais virtuais.

faces Leste, Oeste, Norte e Sul, é ilustrada simplificadamente na Figura 76. São trocados três sinais: *credito*, de dois bits de largura, que informa ao transmissor se o receptor tem disponibilidade em cada fila; *lane*, de dois bits de largura, que informa ao receptor qual é a fila de destino do *flit*; *dado*, de 16 bits, que é a linha de dados compartilhada pelos canais para transmissão do *flit*. A interface de entrada de dados tem um módulo de controle que gerencia a troca dos sinais da interface com os sinais da fila. A interface de saída tem um módulo de controle que transfere os dados que vem das filas de entrada e dá um retorno de cada dado transferido a partir das informações sobre a atualização da tabela de roteamento, realizada pelo módulo de controle. Essa interface implementa ainda um mecanismo de arbitragem que alterna a transmissão de dados entre os dois canais virtuais quando ambos requisitam transmissão de forma simultânea. Como exemplo dessa arbitragem, considerando uma NoC 3x3, a Figura 77 apresenta uma simulação que ilustra o comportamento da saída Leste do nodo 10, quando existem dois pacotes que são encaminhados para os dois canais virtuais da referida porta. Na Figura: (1) a porta de saída Leste recebe a atualização da tabela de roteamento; (2) e (3) começa a transmissão pelo *lane* baixo; (4) o dado gravado no *buffer* do canal virtual baixo da porta de entrada Oeste do nodo vizinho; (5) a porta recebe nova atualização da tabela de roteamento e agora a linha de dados precisa ser compartilhada; (6) e (7) transmissão realizada pelo *lane* baixo e dado gravado no *buffer* do canal baixo do nodo vizinho; (8) e (9) transmissão realizada pelo *lane* alto e dado gravado do *buffer* do canal alto nodo do vizinho.

A Figura 78 apresenta o detalhe da simulação e do compartilhamento da linha de dados *dado* entre os canais virtuais da porta de saída. Na Figura: (1) é feita a atualização da tabela de roteamento; (2) como o sinal de crédito está ativo para o canal baixo, *lane* baixo é ativado ; (3) o dado é disponibilizado na linha de dados; (4) o dado é gravado no *buffer* do canal baixo do vizinho; (5) como o crédito está ativo para o canal alto, o sinal *lane* alto é ativado; (6) o dado é

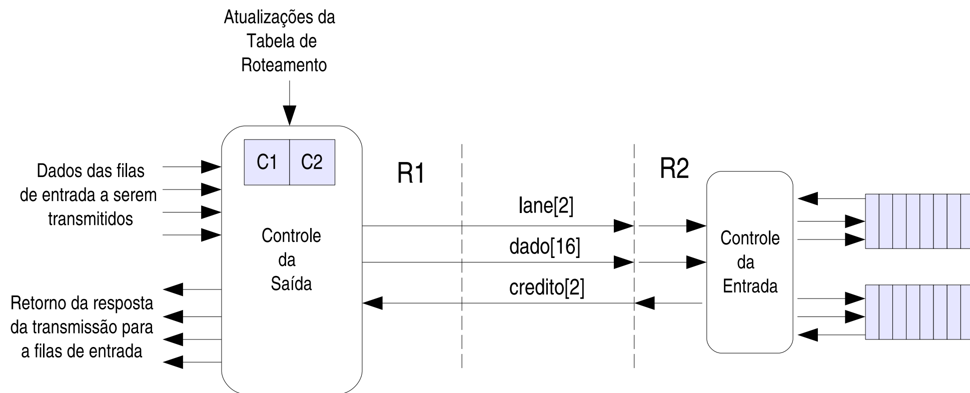

Figura 76 – Interface entre roteadores, com R1 como transmissor e R2 como receptor.

disponibilizado na linha de dados; (7) o dado é gravado no *buffer* do canal alto do vizinho.

A interface de comunicação da porta Local do roteador tem apenas um canal para manter a interface de acordo com a adaptação feita para simulação com a ferramenta ATLAS, explicada no Capítulo 7. Na interface de entrada utiliza-se uma fila com controle de fluxo por aperto de mão. Na interface de saída é necessário gerenciar apenas um canal de saída e recolhe-se as informações vindas do módulo de controle para transferir os dados que vem das filas das entradas de outras portas.

A requisição das filas de entrada pelo roteamento do pacote é semelhante ao apresentado no estudo de caso anterior (7). Os módulos de arbitragem e roteamento trocam os mesmos sinais que são apresentados naquele estudo de caso, com o diferencial do algoritmo de roteamento implementado e do número de canais atendidos pela arbitragem rotativa dinâmica.

Para o roteador deste estudo de caso, um processo semelhante ao apresentado no estudo de caso anterior (Capítulo 7, Seção 7.1.1) foi desenvolvido para explorar o espaço de projeto. O roteador desta descrição ocupa em média 4.722 LUTS (considerando *buffers* com 8 *flits* de 16 *bits*), sintetizado para o dispositivo alvo Virtex II XC2V4000. Com o acréscimos de *buffers* nas portas de entrada, temos um custo de 2.469 LUTS nesses componentes. A *thread* de roteamento, em seu melhor cenário custo/benefício, custa 433 LUTS. Existe um acréscimo significativo de sinais de sincronização desta implementação em relação a implementação do estudo de caso apresentado anteriormente. Tal acréscimo de componentes e sinais repercute na área obtida pela síntese realizada pelo Cynthesizer. Conforme, relatado no Capítulo 7, Seção 7.1.1, a síntese comportamental oferecida pelo *Cynthesizer* não apresentou-se adequada para síntese de processos como os de redes de comunicação intrachip, visto que não consegue otimizar componentes que possuem muitas *threads* trocando informações. Porém, o processo de modelagem e validação oferecido permitiu que esta adptação, com a utilização de redes virtuais para utilizar um algoritmo de rede toro unidirecional para toro bidirecional, pudesse ser implementada e validada. Assim, uma rede toro 3x3 foi instanciada e simulada em nível comportamental no ambiente da ferramenta Cynthesizer. As sínteses Comportamental para ASIC

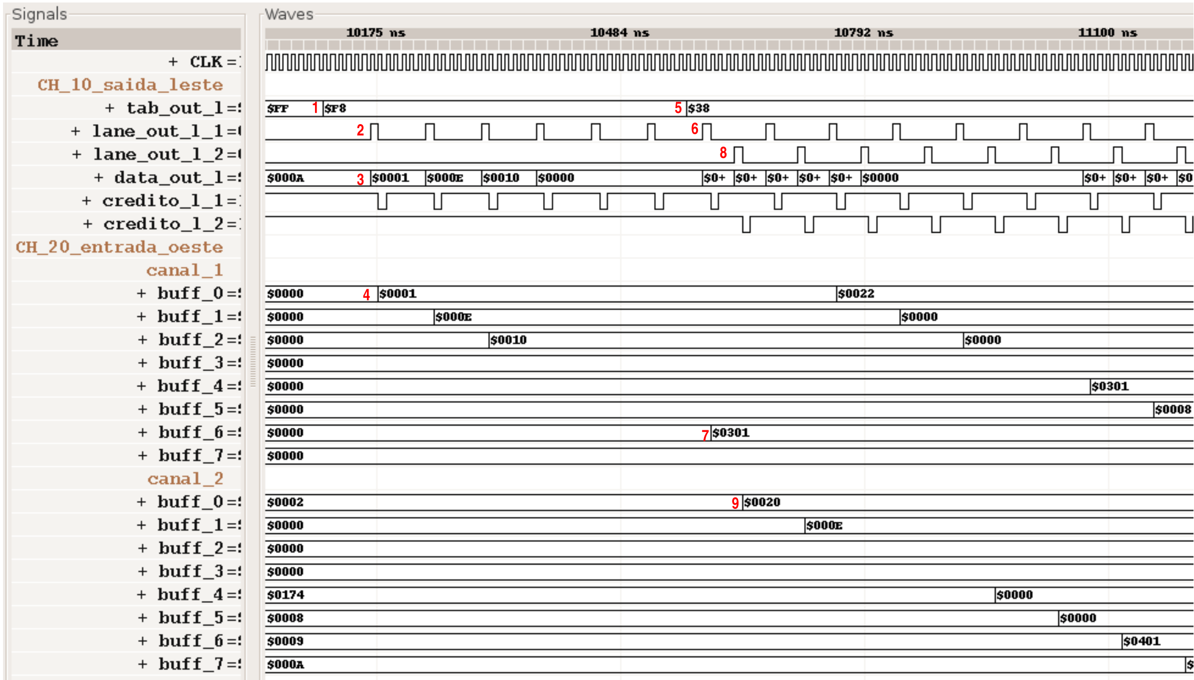

Figura 77 – Porta Leste do nodo 20, com dado a ser transmitido nos dois canais.

usando a tecnologia TSMC 0.18 μm e para FPGA foram realizadas e as arquiteturas geradas foram validadas através do processo de simulação funcional oferecido pelo mesmo ambiente. Para validar a rede foi elaborado uma esquema de comunicação em que cada roteador envia dez pacotes de tamanhos aleatórios para para cada um dos outros roteadores da rede. A verificação do sucesso desse processo foi realizada de forma automatizada, através da elaboração de um programa que confere se os pacotes na saída dos roteadores estão de acordo com os pacote das entradas e se a contagem das mensagens destinadas a cada roteador está correta. As Seções seguintes apresentam resultados adicionais de latência e área dos resultados de síntese obtidos com a ferramenta.

8.1.1 Avaliação de Latência da NoC Gerada com Síntese Comportamental

Para o presente estudo de caso também foi utilizada a ferramenta ATLAS [34] para a avaliação de latência do resultado obtido nas sínteses realizadas. Como exemplo de avaliação, foi instanciada uma rede toro 2D 3x3, com tamanho de *buffer* de 8 *flits*, e gerado na ATLAS um cenário de tráfego onde cada chave envia 30 pacotes de 16 *flits* a um destino aleatório. As Tabelas 17 e 18 mostram, respectivamente, as 10 menores e as 10 maiores latências da simulação deste cenário de tráfego.

Os resultados de latência obtidos demonstram que, quanto mais processos precisarem de sincronização, mais ciclos de relógio serão gastos para garantir a integridade do comportamento

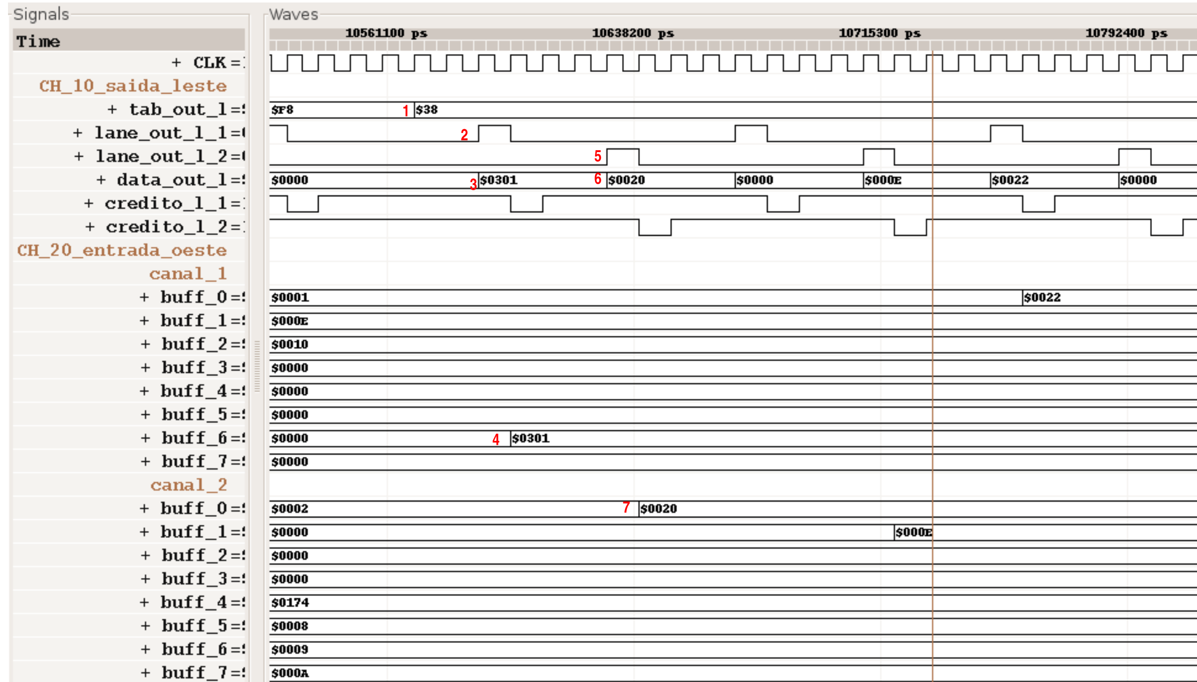

Figura 78 – Detalhe do comportamento da porta quando a linha de dados é compartilhada pelos canais.

especificado na descrição inicial. No caso da infraestrutura de comunicação intrachip descrita nesse estudo de caso isso realmente interfere na qualidade do hardware obtido, já que cada roteador possui um conjunto de processos que operam em paralelo e precisam ser sincronizados de alguma forma. Para cada sincronização entre as *threads* internas do roteador para a transmissão de um *flit* são perdidos entre 2 e 4 ciclos de relógio, o que gera um grande impacto na latência final de transmissão do pacote.

8.1.2 Avaliação de Área da NoC Gerada com Síntese Comportamental

Para avaliação de área da NoC gerada, foram extraídos relatórios da síntese para FPGA visando o dispositivo Virtex II XC2V4000(*speed grade* -5) da XILINX. A Tabela 19 apresenta o resultado do processo de síntese lógica para FPGA, a partir do resultado obtido na síntese comportamental, para NoCs de dimensão 3x3 e 4x4, com tamanho dos *buffers* de 4 e 8 *flits*. Para as redes de dimensão 3x3, o aumento do *buffer* gerou um acréscimo de 7,21 % em LUTS ocupadas. Para as redes 4x4 o aumento do *buffer* gerou um acréscimo de 7,77% em LUTS ocupadas. O dispositivo escolhido não comporta a rede de dimensão 4x4.

Como elemento de comparação, o trabalho de Scherer [36] avalia a ocupação em LUTS da rede descrita em relação a rede Hermes com 2 canais virtuais e Hermes Toro Unidirecional, para *flits* de 8, 16, 32 e 64 bits e tamanho de fila de 4, 8, 16 e 32 *flits*. A rede Hermes 2CV 4x4 com *flits* de 16 bits ocupa, para o mesmo dispositivo deste trabalho, 24019 LUTS para tamanho

Latência	Origem	Destino
157	20	21
159	12	11
159	11	12
161	01	11
163	00	01
165	12	10
165	00	10
165	20	21
165	22	02
165	01	02

Tabela 17 – As 10 menores latências obtidas na simulação do cenário com a rede 3x3 com *buffer* de 8 flits.

Latência	Origem	Destino
855	21	10
803	10	22
763	02	20
753	01	20
733	10	02
689	20	11
681	12	02
663	02	11
649	01	20
643	01	22

Tabela 18 – As 10 maiores latências obtidas na simulação do cenário com a rede 3x3 com *buffer* de 8 flits.

de *buffer* 4 *flits* e 24483 para tamanho de *buffer* de 8 *flits*. A comparação dos resultados obtidos por Scherer [36] com a rede toro 2D 4x4 gerada nesse estudo de caso revela que: (1) para o tamanho de *buffer* de 4 *flits* a rede do estudo de caso ocupa 193,49% mais LUTS que a Hermes 2CV; (2) para o tamanho de *buffer* de 8 *flits* a rede do estudo de caso ocupa 210,45% mais LUTS que a Hermes 2CV.

Dimensão	Tamanho do buffer em flits	Ocupação do Dspositivo
3x3	4	86% (39826 LUTS)
3x3	8	92% (42498 LUTS)
4x4	4	152% (70494 LUTS)
4x4	8	164% (76007 LUTS)

Tabela 19 – Ocupação do dispositivo Virtex II XC2V4000(*speed grade* -5) da XILINX obtido através de síntese comportamental.

9 Conclusões e Trabalhos Futuros

A avaliação do uso de um processo de síntese comportamental como alternativa para o projeto de redes intrachip foi realizada quanto a modelagem de projeto e a qualidade dos resultado obtidos.

Pelo que foi demonstrado nos Capítulos 7 e 8, o processo de síntese comportamental avaliado em termos de modelagem mostrou-se bastante intuitivo e simples, principalmente pelo uso da linguagem SystemC. O estilo de codificação em SystemC aceito pela ferramenta *Cynthesizer* foi dominado rapidamente e permitiu que os estudos de caso fossem implementados em pouco tempo, inclusive com reuso de codificação. Também devido a esse estilo simples foi possível criar facilmente programas em C que gerassem automaticamente a instanciação das redes implementadas. Quanto à validação, o processo integrado de simulação oferecido pela ferramenta torna transparente e fácil para o projetista a realização da simulação em vários níveis de abstração, o que permite verificar funcionalmente a validade do sistema desenvolvido. Outra vantagem observada foi a facilidade de utilizar ferramentas integradas ao ambiente de desenvolvimento, tais como Modelsim, Synplify Pro e ISE, habilitando um fluxo de projeto com bom grau de integração entre diferentes etapas, partindo da especificação inicial e chegando ao FPGA ou ASIC devidamente verificados.

Porém, conforme foi mostrado no Capítulo 4, e comprovado nos Capítulos 7 e 8, tal síntese não foi desenvolvida para contemplar projetos de estrutura de comunicação intrachip, que em geral, possuem pouco processamento e muita troca de sinais. A qualidade em latência da descrição de hardware em nível RTL está diretamente ligada ao volume da troca de sinais entre *threads* que implementam um determinado comportamento, porque essas trocas precisam ser sincronizadas com sinais adicionais. Quanto mais *threads* forem incluídas na implementação de um módulo e quanto mais sinais essas *threads* trocarem, pior será a latência do hardware derivado. Também em consequência disso, a ferramenta gera mais hardware para conseguir controlar a sincronização especificada pelo projetista. Comparado com a codificação direta em estilo RTL, a ferramenta gera hardware com baixa eficiência em área e latência para projetos de estruturas de comunicação intrachip.

Como resultado adicional desse trabalho os estudos de caso de adaptação de algoritmos de roteamento para redes toro bidirecionais, com ou sem canais virtuais foram concluídos a contento. Tais estudos de caso inseriram a complexidade necessária para realizar as avaliações que eram objeto deste trabalho, já que redes toro bidirecional 2D não são exploradas suficientemente na literatura. O algoritmo de roteamento apresentado no Capítulo 7, que foi desenvolvido em parceria com o colega Carlos Scherer [36], trata de uma adaptação sugerida pelos proponentes

do modelo *turn model*, porém sem registros de implementação na literatura. O algoritmo de roteamento apresentado no Capítulo 8 é uma contribuição deste trabalho e trata do suporte que a rede física oferece para permitir redes virtuais e executar dois algoritmos diferentes na mesma infra-estrutura de comunicação. Conforme mostrado, tal suporte tem alto custo em termos de área e latência.

Também, como contribuição deste trabalho, cita-se a utilização dos produtos da ferramenta ATLAS integrados ao fluxo de verificação do *Cynthesizer*, para que as redes pudessem ser simuladas com tráfego sintético complexo e avaliadas mais facilmente.

Como trabalhos futuros, a sugestão é a avaliação dos algoritmos de roteamento implementados em RTL para que a comparação no mesmo nível com outras redes, como a HERMES, seja realizada em termos de área e latência. Também, como sugestão coloca-se a utilização do *Cynthesizer* como ambiente para integração e simulação, já que a ferramenta permite simulação em vários níveis de abstração através da iteração com outras ferramentas comerciais, de forma transparente para o projetista.

Referências

[1] Luca Benini and Giovani De Micheli. Networks on Chips: a New Paradigm for Component-Based MPSoC Design. *A. Jerraya and W. Wolf Editors, Multiprocessors Systems on Chips, Morgan Kaufman*, pages 49–80, 2004.

[2] Luca Benini and Giovanni De Micheli. Networks on Chips: A New SoC Paradigm. *IEEE Computer*, 35(1):70–78, 2002.

[3] Reinaldo Bergamaschi. Behavioral network graph: Unifying the domains of high-level and logic synthesis. In *Proceedings of the ACM/IEEE Conference on Design Automation - DAC*, pages 213–218, 1999.

[4] Reinaldo Bergamaschi and John Cohn. The A to Z of SoCs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 790–798, New York, NY, USA, 2002. ACM Press.

[5] Davide Bertozzi, Antoine Jalabert, Srinivasan Murali, Rutuparna Tamhankar, Stergios Stergiou, Luca Benini, and Giovanni De Micheli. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):113–129, February 2005.

[6] Celoxica. *Datasheet: Agility Compiler for SystemC Electronic*. Disponível em: http://www.celoxica.com/techlib/files/CEL-W0602151E4L-335.pdf. Último acesso em 01/08/2007.

[7] Wander Cesario, Zoltan Sugar, Rodolphe Suescun, and Ahmed Jerraya. Overlap and frontiers between behavioral and RTL synthesis. In *User's Forum, Design, Automation and Test in Europe - DATE*, March 1999.

[8] Marcello Coppola, Stephane Curaba, Miltos Grammatikakis, Riccardo Locatelli, Giuseppe Maruccia, and Francesco Papariello. OCCN: a NoC modeling framework for design exploration. *Journal of Systems Architecture*, 50(2-3):129–163, 2004.

[9] Marcello Coppola, Stephane Curaba, Miltos Grammatikakis, Giuseppe Maruccia, and Francesco Papariello. OCCN: A network-on-chip modeling and simulation framework. In *Proceedings of the Conference on Design, Automation and Test in Europe - DATE*, pages 174–179. IEEE Computer Society, 2004.

[10] Matteo Dall'Osso, Gianluca Biccari, Luca Giovannini, Davide Bertozzi, and Luca Benini. XPIPES: A latency insensitive parameterized network-on-chip architecture for multiprocessor SoCs. In *Proceedings of the International Conference on Computer Design - ICCD*, pages 536–539, 2003.

[11] William Dally and Charles Seitz. The torus routing chip. *Distributed Computing*, 1(4):187–196, 1986.

[12] William Dougherty and Donald Thomas. Unifying behavioral synthesis and physical design. In *Proceedings of the ACM/IEEE Conference on Design Automation - DAC*, pages 756–761, 2000.

[13] Jeffrey Draper. The red rover algorithm for deadlock-free routing on bidirectional rings. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications -PDPTA*, pages 345–54, August 1996.

[14] Jeffrey Draper and Fabrizio Petrini. Routing in bidirectional k-ary n-cubes with the red rover algorithm. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications -PDPTA*, pages 1184–1193, June 1997.

[15] José Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., USA, 2002.

[16] Daniel Gajski, Nikil Dutt, Allen Wu, and Steve Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[17] Christopher Glass and Lionel Ni. The turn model for adaptive routing. *Journal of ACM*, 41(5):874–902, 1994.

[18] Mentor Graphics. *Datasheet: Catapult C Synthesis*. Disponível em: http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/upload/Catapult_DS_0107.pdf. Último acesso em 01/08/2007.

[19] Jingcao Hu and Radu Marculescu. Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In *Proceedings of the Conference on Design, Automation and Test in Europe - DATE*, pages 234–239. IEEE Computer Society Press, 2004.

[20] Antoine Jalabert, Srinivasan Murali, Luca Benini, and Giovanni De Micheli. XpipesCompiler: a tool for instantiating application specific networks on chip. In *Proceedings of the Conference on Design, Automation and Test in Europe - DATE*, pages 884–889, 2004.

[21] Ahmed Jerraya, WanderCesario, Zoltan Sugar, and Issam Moussa. Efficient integration of behavioral synthesis within existing design flows. In *Proceedings of the International Symposium on System Synthesis - ISSS*, pages 85–90, Madrid, Espanha, September 2000.

[22] David Ku and Giovanni De Micheli. *High level synthesis of ASICs under timing and synchronization constraints*. Kluwer Academic Publishers, EUA, 1992.

[23] Elizabeth Lagnese and Donald Thomas. Architectural partitioning for system level synthesis of integrated circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(7):847–860, 1991.

[24] Mike Lee. *High-Level Test Synthesis of Digital VLSI Circuits*. Artech House, 1997.

[25] Daniel Linder and Jim Harden. An adaptive and fault tolerant wormhole routing strategy for k-ary n-cubes. *IEEE Transactions on Computers*, 40(1):2–12, 1991.

[26] César Marcon. *Modelos para Mapeamento de Aplicações em Infra-estruturas de Comunicação Intrachip*. PhD thesis, Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul - UFRGS, December 2005.

[27] César Marcon, Márcio Kreutz, Altamiro Susin, and Ney Calazans. Models for embedded application mapping onto NoCs: Timing analysis. In *Proceedings of the IEEE International Workshop on Rapid System Prototyping - RSP*, pages 17–23. IEEE Computer Society, 2005.

[28] Radu Marculescu and Jingcao Hu. Energy-aware mapping for tile-based NoC architectures under performance constraints. In *Proceedings of the Design Automation Conference Asia and South Pacific - ASP-DAC*, January 2003.

[29] Peter Marwedel. The MIMOLA design system: Detailed description of the software system. In *Proceedings of the ACM/IEEE Conference on Design Automation - DAC*, pages 59–63. IEEE Press, 1979.

[30] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *Integration VLSI Journal - Elsevier*, 38(1):69–93, 2004.

[31] Srinivasan Murali and Giovanni De Micheli. SUNMAP: a tool for automatic topology selection and generation for NoCs. In *Proceedings of the ACM/IEEE Conference on Design Automation - DAC*, pages 914–919. ACM Press, 2004.

[32] Lionel Ni and Philip McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 26(2):62–76, 1993.

[33] Kirk Ober. Doing behavioral design the right way minimizes verification. In *Proceedings of the Design and Verification Conference and Exhibition - DVCon*, March 2004. Disponível em: http://www.forteds.com/products/paper_030304_bdrightway.pdf.

[34] Luciano Ost, Aline Mello, José Palma, Fernando Moraes, and Ney Calazans. MAIA: a framework for networks on chip generation and verification. In *Proceedings of the Conference on Asia South Pacific Design Automation - ASP-DAC*, pages 49–52, 2005.

[35] David Pursley. Using Behavioral Clustering to Improve Quality of Results for DSP Designs. In *International Signal Processing Conference - Global Signal Processing Expo and Conference - GSPx/ISPC*, 2004. Disponível em: http://www.forteds.com/products/paper_092704_behclustering.pdf.

[36] Carlos Scherer. Redes intrachip com topologia toro e modo de chaveamento wormhole: Projeto, geração e avaliação. Master's thesis, Programa de Pós-Graduação em Ciência da Computação, Pontifícia Universidade Católica PUCRS, March 2007.

[37] Jos Sulistyo and Dong Ha. A new characterization method for delay and power dissipation of standard library cells. *VLSI Design*, 15(3):667–678, 2002. Disponível em: http://www.ee.vt.edu/%7Eha/cell_library/references/vlsidesign_charac.pdf. Último acesso em 10/11/2006.

[38] Jos Sulistyo, Jonathan Perry, and Dong Ha. Developing standard cells for tsmc 0.25um technology under mosis deep rules. Technical Report VISC-2003-01, Department of Electrical and Computer Engineering, Virginia Tech, 2003. Disponível em: http://www.ee.vt.edu/%7Eha/cell_library/references/tech_report.pdf. Último acesso em 10/11/2006.

[39] Forte Design Systems. *Cynthesizer - Complete Manual Set*, 2006. Disponível a partir de: http://kb.forteds.com/cynthkb.

[40] Donald. E. Thomas, E. M. Dirkes, Robert A. Walker, Jayanth V. Rajan, John A. Nestor, and Robert L. Blackburn. The system architect's workbench. In *Proceedings of the ACM/IEEE Conference on Design Automation - DAC*, pages 337–343. IEEE Computer Society Press, 1988.

[41] Norbet Wehn, Jörg Biesenack, Peter Duzy, Anton Langmaier, Michael Pilsl, and Steffen Rumler. Scheduling of behavioral VHDL by retiming techniques. In *Proceedings of the European Design Automation Conference - EURO-DAC*, pages 546–551. IEEE Computer Society Press, 1994.

[42] Érico Bastos. Mercury: uma rede intrachip com topologia tor 2D e roteamento adaptativo. Master's thesis, Programa de Pós-Graduação em Ciência da Computação, Pontifícia Universidade Católica PUCRS, January 2006.