UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA

JORGE LUCIO TONFAT SECLEN

Projeto, Verificação Funcional e Síntese de Módulos Funcionais para um Comutador Gigabit Ethernet

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Microeletrônica

Prof. Dr. Ricardo Augusto da Luz Reis
Orientador

Porto Alegre, maio de 2011

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Seclen, Jorge Lucio Tonfat

Projeto, Verificação Funcional e Síntese de Módulos Funcionais para um Comutador Gigabit Ethernet / Jorge Lucio Tonfat Seclen. – Porto Alegre: PPGMICRO da UFRGS, 2011.

128 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2011. Orientador: Ricardo Augusto da Luz Reis.

1. Comutador ethernet. 2. Comunicação de dados. 3. Verificação funcional. 4. SystemVerilog. 5. Síntese física. 6. Microeletrônica. I. Reis, Ricardo Augusto da Luz. II. Título.

Caminante, son tus huellas
el camino, y nada más;
caminante, no hay camino,
se hace camino al andar.
Al andar se hace camino,
y al volver la vista atrás
se ve la senda que nunca
se ha de volver a pisar.
Caminante, no hay camino,
sino estelas en la mar.
— ANTONIO MACHADO

AGRADECIMIENTOS

Agradezco a Dios primeramente, por darme la fuerza, tranquilidad y perseverancia para culminar esta etapa.

A mi toda mi familia, en especial a mis papas, por todo su amor y apoyo. Sin ellos no sería la persona que soy hoy. El ejemplo que me dan me da las fuerzas necesarias para continuar.

A mi orientador, Ricardo Reis, porque gracias a él pude realizar este proyecto llamado Maestría. La oportunidad que me dio al poder venir a Brasil es algo que siempre lo recordaré.

Al equipo que trabajó para el convenio TERACOM-UFRGS, Gustavo Neuberger, Érico Sawabe, Lucas Mizusaki y especial mención para Rafael Ramos porque en ese grupo aprendí mucho tanto en lo profesional como en lo personal. ¡Gracias!

A las personas que conocí en esta nueva ciudad llamada Porto Alegre, en especial al equipo que trabaja en los laboratorios 217, 219 y 232 de informática de la UFRGS porque gracias a su amistad, mi adaptación a esta nueva casa fue muy rápida.

A mis amigos que siempre estuvieron ahí para darme un consejo u opinión, en especial para Jimmy Tarrillo y Chris Tómas por sus críticas y comentarios que ayudaron a mejorar la calidad de este trabajo. No puedo dejar de mencionar a Gracieli Posser, Claudia Antunez y Tania Ferla porque gracias a ellas este trabajo se puede considerar que está escrito en idioma portugués. Obrigado!

Para finalizar, a los miembros del grupo de microelectrónica de la PUCP, porque fue ahí donde di mis primeros pasos en el mundo de la microelectrónica. Quiero agradecer especialmente a su director Carlos Silva porque él fue quien me dio la oportunidad de hacer mi primer viaje a Brasil y después conseguir hacer la maestría en la UFRGS.

A todos, ¡muchas gracias!

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 9

LISTA DE SíMBOLOS 13

LISTA DE FIGURAS 15

LISTA DE TABELAS 19

RESUMO 21

ABSTRACT 23

1 INTRODUÇÃO 25
1.1 Contribuição da dissertação de mestrado 26
1.2 Organização da dissertação 27

2 A TECNOLOGIA ETHERNET 29
2.1 História da Ethernet 29
2.2 A arquitetura das redes 29
2.3 Conceito de encapsulamento 31
2.4 Endereçamento 31
2.5 Quadro Ethernet 32
2.6 Modos de Operação 34
2.7 Outras Tecnologias LAN 34
2.8 Padrão IEEE 802.1D 34
2.9 Padrão IEEE 802.1Q 34

3 COMUTADOR GIGABIT ETHERNET 35
3.1 Principais Funções de um Comutador 35
3.1.1 Encaminhamento 35
3.1.2 Aprendizagem 36
3.1.3 Envelhecimento 36
3.2 Principais Arquiteturas 36
3.2.1 Plataforma NetFPGA 37

4 FLUXO DE IMPLEMENTAÇÃO PARA UM ASIC 41
4.1 Descrição das etapas do fluxo de implementação 41

5 DESENVOLVIMENTO DOS MÓDULOS FUNCIONAIS DO COMUTADOR GIGABIT ETHERNET 45
5.1 Árbitro da Entrada 46
5.1.1 Análise das características dos algoritmos *Round Robin* e *Deficit Round Robin* 48
5.1.2 Modificações feitas ao módulo original 50
5.2 Pesquisador da porta de saída 54
5.2.1 Blocos Extrator e Adicionador de VLAN 55
5.2.2 Motor de Classificação de Nível 2 60
5.3 Marcador de Quadros 72
5.4 Filas de Saída 76
5.4.1 Lógica das filas de saída 77
5.4.2 Bloco DDR2 de escrita e leitura de dados 80
5.4.3 Adaptação do controlador de memória DDR2 SDRAM 85

6 METODOLOGIA DE VERIFICAÇÃO FUNCIONAL 93
6.1 Introdução 93
6.2 Arquitetura do ambiente de verificação 94
6.2.1 Camadas de sinal e comando 95
6.2.2 Camada funcional 96
6.2.3 Modificações feitas ao modelo de referência 96
6.2.4 Camada de cenário 98
6.2.5 Camada de teste 99
6.3 Geração de estímulos restritos 99
6.4 Etapas da simulação 100
6.5 Métricas de cobertura 100
6.6 Caso de análise: Módulo filas de saída 101

7 ANÁLISE DOS RESULTADOS E COMPARAÇÕES 105
7.1 Resultados da Verificação Funcional 105
7.2 Resultados da Implementação Física 106
7.2.1 Árbitro de entrada 106
7.2.2 Pesquisador da porta de saída 106
7.2.3 Marcador de quadros 109
7.2.4 Filas de saída 109

8 CONCLUSÕES E TRABALHOS FUTUROS 113
8.1 Trabalhos Futuros 114

REFERÊNCIAS 117

APÊNDICE A *SCRIPTS* TCL PARA O CONTROLE DA SIMULAÇÃO EM MODELSIM 123
A.1 *Script* para o módulo Marcador de quadros 123
A.2 *Script* para o módulo Árbitro de entrada 124
A.3 *Script* para o módulo Pesquisador da porta de saída 125
A.4 *Script* para o módulo Filas de saída 126

LISTA DE ABREVIATURAS E SIGLAS

ASIC	*Application specific Integrated Circuit*
BFM	*Bus functional Model*
BRAM	*Block RAM*
CAS	*Column Address Strobe*
CAM	*Content-adressable Memory*
CBS	*Committed Burst Size*
CCITT	*Comité Consultatif International Téléphonique et Télégraphique*
CIR	*Committed Information Rate*
CRC	*Cyclic Redundancy Check*
CFI	*Canonical Format Indicator*
CoS	*Class of Service*
CPU	*Central Process Unit*
DDR	*Double Data Rate*
DEC	*Digital Equipment Corporation*
DFI	*DRR PHY Interface*
DFT	*Design for Testability*
DMA	*Direct Memory Access*
DLL	*Delay Locked Loop*
DRAM	*Dynamic Random Access Memory*
DRR	*Deficit Round Robin*
DUT	*Design Under Test*
DUV	*Design Under Verification*
EBS	*Excess Burst Size*
EDA	*Electronic Design Automation*
EDD	*Ethernet Demarcation Device*
EIR	*Excess Information Rate*

FCS	*Frame Check Sequence*
FDDI	*Fiber Distributed Data Interface*
FIFO	*First Input First Output*
FPGA	*Field Programmable Gate Array*
FT-FIFO	*Fallthrough FIFO*
FSM	*Finite State Machine*
Gbps	*Gigabits per second*
GEMAC	*Gigabit Ethernet MAC*
GigE	*Gigabit Ethernet*
HDL	*Hardware Description Language*
IBM	*International Business Machines*
IEEE	*Institute of Electrical and Electronics Engineers*
IP	*Internet Protocol*
IP-core	*Intellectual Property core*
LAN	*Local Area Network*
LEC	*Logic Equivalence Checking*
LLC	*Logical Link Control*
LUT	*Lookup Table*
MAC	*Media Access Control*
MAN	*Metropolitan Area Network*
Mbps	*Megabits per second*
MEN	*Metro Ethernet Network*
MIG	*Memory Interface Generator*
MTU	*Maximum Transmission Unit*
OSI	*Open System Interconnection*
OUI	*Organizationally Unique Identifier*
PBB	*Provider Backbone Bridge*
PCI	*Peripheral Component Interconnect*
PDU	*Protocol Data Unit*
PHY	*Physical Interface*
QoS	*Quality of Service*
RAS	*Row Address Strobe*
RFC	*Request For Comments*
RTL	*Register Transfer Level*

RR	*Round Robin*
RSTP	*Rapid Spanning Tree Protocol*
Rx	*Receiver*
SDU	*Service Data Unit*
SDRAM	*Synchronous Dynamic Random Access Memory*
SI	*Signal Integrity*
SLA	*Service Level Agreement*
SoC	*System on Chip*
SPN	*Service Provider Network*
srTCM	*single rate Three Color Marker*
SRAM	*Static Random Access Memory*
STA	*Static Timing Analysis*
STP	*Spanning Tree Protocol*
TCL	*Tool Command Language*
TSMC	*Taiwan Semiconductor Manufacturing Company*
trTCM	*two rate Three Color Marker*
Tx	*Transmitter*
UFRGS	Universidade Federal do Rio Grande do Sul
UNI	*User Network Interface*
VCD	*Value Change Dump*
VHDL	*Very high speed integrated circuits Hardware Description Language*
VLSI	*Very Large Scale Integration*
VLAN	*Virtual Local Area Network*
VMM	*Verification Methodology Manual*
WRR	*Weighted Round Robin*
ZBT	*Zero Bus Turnaround*

LISTA DE SÍMBOLOS

B	Byte
f	Femto
Hz	Hertz
K	Kilo
M	Mega
μ	Micron
m	Milli
n	Nano
Ω	Ohms
p	Pico
s	segundos
W	Watts

LISTA DE FIGURAS

Figura 1.1: Visão do ambiente de inserção do Comutador. 26

Figura 2.1: Modelo de referência OSI e sua relação com o padrão IEEE 802.3. . . 30
Figura 2.2: Encapsulamento dos dados através das camadas. 31
Figura 2.3: Formato do endereço MAC *Ethernet*. 32
Figura 2.4: Formato do quadro Ethernet MAC. 33
Figura 2.5: Formato do quadro Ethernet MAC com etiqueta VLAN. 33

Figura 3.1: Diagrama da plataforma NetFPGA. 37
Figura 3.2: Estrutura modular da plataforma NetFPGA. 38
Figura 3.3: Formato interno dos quadros *Ethernet* na plataforma NetFPGA. . . . 38
Figura 3.4: Diagrama da parte operativa do comutador *Gigabit Ethernet*. 39
Figura 3.5: Diagrama de tempos da comunicação entre os módulos da parte operativa. 39

Figura 4.1: Diagrama do fluxo de síntese em ASIC. 42
Figura 4.2: Diagrama do fluxo da síntese lógica. 43
Figura 4.3: Diagrama do fluxo de síntese física. 44

Figura 5.1: Diagrama do funcionamento do módulo Árbitro da Entrada. 46
Figura 5.2: Algoritmo DRR implementado. 47
Figura 5.3: Largura de banda (*data bandwidth*) por fila para cada um dos algoritmos analisados. 49
Figura 5.4: Diagrama de blocos do árbitro de entrada. 50
Figura 5.5: Estrutura de uma FIFO. 51
Figura 5.6: Operação de leitura e escrita de uma FIFO padrão. 51
Figura 5.7: Operação de leitura e escrita de uma FIFO *fallthrough*. 52
Figura 5.8: Máquina de estados do árbitro de entrada. 53
Figura 5.9: Diagrama de blocos do módulo Pesquisador da porta de saída. 54
Figura 5.10: Diagrama de blocos do Extrator de VLAN. 56
Figura 5.11: Diagrama de estados da máquina que procura pela informação VLAN do bloco Extrator de VLAN. 56
Figura 5.12: Diagrama de estados da máquina que adiciona o cabeçalho VLAN do bloco Extrator de VLAN. 57
Figura 5.13: Cabeçalho VLAN inserido no quadro na saída do bloco Extrator de VLAN. 57
Figura 5.14: Diagrama de blocos do Adicionador de VLAN. 58

Figura 5.15: Diagrama de estados da máquina que procura pelo cabeçalho VLAN do bloco Adicionador de VLAN. 58
Figura 5.16: Diagrama de estados da máquina que adiciona o identificador VLAN no quadro do bloco Adicionador de VLAN. 59
Figura 5.17: Função de *Hash*. 60
Figura 5.18: Diagrama de blocos da arquitetura 1. 61
Figura 5.19: Diagrama de estados da FSM do bloco analisador de cabeçalho. . . . 62
Figura 5.20: Diagrama de estados da FSM do bloco de encaminhamento e aprendizagem. 63
Figura 5.21: Organização da memória SRAM. 64
Figura 5.22: Organização do segmento da tabela VLAN na memória SRAM. . . . 65
Figura 5.23: Diagrama de estados da FSM do bloco de envelhecimento. 66
Figura 5.24: Diagrama de tempos de uma operação de escrita no árbitro SRAM. . 66
Figura 5.25: Diagrama de tempos de uma operação de leitura no árbitro SRAM. . 67
Figura 5.26: Diagrama de estados da primeira FSM do árbitro SRAM. 67
Figura 5.27: Diagrama de estados da segunda FSM do árbitro SRAM. 68
Figura 5.28: Diagrama de estados da FSM do motor de classificação. 69
Figura 5.29: Diagrama de blocos da arquitetura 2. 70
Figura 5.30: Estrutura de *pipeline* do árbitro SRAM. 70
Figura 5.31: Diagrama de tempos de uma operação de escrita na SRAM. 71
Figura 5.32: Diagrama de tempos de uma operação de leitura na SRAM. 71
Figura 5.33: Algoritmo de medição e classificação de quadros descrito no RFC 2697 e RFC 4115. 72
Figura 5.34: Algoritmo de medição e classificação de quadros descrito no RFC 2698. 73
Figura 5.35: Diagrama de blocos do módulo Marcador de Quadros. 73
Figura 5.36: Cabeçalho inserido no quadro pelo módulo Marcador de Quadros. . . 74
Figura 5.37: Máquina de estados do Marcador de Quadros. 75
Figura 5.38: Diagrama de blocos do módulo das Filas de Saída e sua interface com a memória DDR2 SDRAM externa. 77
Figura 5.39: Diagrama de blocos da lógica interna das filas de saída. 78
Figura 5.40: Diagrama de estados do bloco analisador de cabeçalho. 78
Figura 5.41: Arquitetura das Filas DRAM. 79
Figura 5.42: Diagrama de blocos do bloco interface entre o sistema e o controlador DDR2. 83
Figura 5.43: Operação de escrita na DRAM. 85
Figura 5.44: Operação de leitura na DRAM. 86
Figura 5.45: Diagrama de blocos de um controlador de memória SDRAM. 86
Figura 5.46: Hierarquia dos blocos do controlador original gerado pela Xilinx. . . 87
Figura 5.47: Hierarquia dos blocos do controlador modificado com a interface DFI. 88
Figura 5.48: Sinais da interface DFI implementada. 89
Figura 5.49: Parâmetro t_{ctrl_delay}. 89
Figura 5.50: Parâmetros t_{phy_wrlat} e t_{phy_wrdata}. 91
Figura 5.51: Parâmetros t_{rddata_en} e t_{phy_rdlat}. 91
Figura 5.52: Sinal $dfi\_init\_complete$. 92

Figura 6.1: O processo de projetar contra o processo de verificar. 93
Figura 6.2: O progresso da verificação aleatória *versus* verificação direcionada. . 94
Figura 6.3: Arquitetura do ambiente de verificação descrito em SystemVerilog. . 95

Figura 6.4: Hierarquia de classes dos *drivers* de dados e registradores, monitor e gerador de quadros. 96
Figura 6.5: Hierarquia de classes utilizadas para gerar o *scoreboard*. 97
Figura 6.6: Hierarquia de classes utilizadas para gerar a configuração do DUV. Nesta Figura, o DUV é o "marcador de quadros". 98
Figura 6.7: Classe Ambiente: Implementa todo o ambiente de verificação. 99
Figura 6.8: Hierarquia de classes para definir os quadros Ethernet e o conteúdo dos registradores de configuração. 100
Figura 6.9: Primeira etapa da simulação: construção ou *build*. 101
Figura 6.10: Segunda e terceira etapa da simulação: *Run* e *Wrap-up*. 102
Figura 6.11: Relação da cobertura de código e funcional. 102
Figura 6.12: Hierarquia de classes para criar a cobertura funcional. 103
Figura 6.13: Exemplo de um grupo de cobertura funcional definida em SystemVerilog. 103
Figura 6.14: Primeira etapa: verificação da lógica das filas de saída. 103
Figura 6.15: Segunda etapa: verificação da interface entre o sistema e o controlador DDR2 SDRAM. 104
Figura 6.16: Terceira etapa: verificação do controlador DDR2 SDRAM. 104

Figura 7.1: Leiaute do módulo Árbitro de entrada. 107
Figura 7.2: Leiaute da segunda proposta do módulo Pesquisador da porta de saída. 108
Figura 7.3: Distribuição de área do módulo Pesquisador da porta de saída. 109
Figura 7.4: Leiaute do módulo Marcador de quadros. 109
Figura 7.5: Leiaute do módulo Filas de saída. 110
Figura 7.6: Distribuição de área dos quatro módulos. 111
Figura 7.7: Distribuição de potência dos quatro módulos. 111

LISTA DE TABELAS

Tabela 3.1: Endereços *multicast* reservados. 36

Tabela 5.1: Interface de comunicação comum aos quatro módulos. 45
Tabela 5.2: Descrição dos parâmetros do ambiente de teste. 48
Tabela 5.3: Características dos cenários aplicados. 49
Tabela 5.4: Registradores do Árbitro de entrada. 53
Tabela 5.5: Interface do módulo pesquisador da porta de saída. 55
Tabela 5.6: Formato de cada entrada da tabela de endereços MAC. 64
Tabela 5.7: Registradores do motor de classificação. 69
Tabela 5.8: Tabela de ciclos da FSM mostrando as operações realizadas em cada ciclo. 70
Tabela 5.9: Tabela de ciclos da FSM do bloco de encaminhamento e aprendizagem. 72
Tabela 5.10: Formato do cabeçalho do Marcador de Quadros. 74
Tabela 5.11: Registradores do Marcador de Quadros. 76
Tabela 5.12: Registradores das filas de saída. 80
Tabela 5.13: Tabela de sinais na interface entre a lógica das filas de saída e o bloco DDR2 de escrita e leitura de dados. Os sinais são síncronos ao relógio das Filas de saída. 81
Tabela 5.14: Tabela de sinais na interface entre o bloco DDR2 de escrita e leitura de dados e o controlador da memória DDR2. Os sinais são síncronos ao relógio do controlador DDR2. 82
Tabela 5.15: Tabela de sinais na interface DFI entre o bloco do controlador lógico e a camada física ou PHY. 90

Tabela 7.1: Tabela de resultados da verificação funcional dos quatro módulos. . . 105
Tabela 7.2: Características do módulo Árbitro de entrada. 106
Tabela 7.3: Comparação dos resultados do módulo Pesquisador da porta de saída. 107
Tabela 7.4: Tabela de comparação de resultados com outros trabalhos na literatura. 108
Tabela 7.5: Características do módulo Pesquisador da porta de saída. 108
Tabela 7.6: Características do módulo Marcador de quadros. 110
Tabela 7.7: Características do módulo Filas de saída. 110

RESUMO

Este trabalho apresenta o projeto, a verificação funcional e a síntese dos módulos funcionais de um comutador Gigabit Ethernet. As funções destes módulos encontram-se definidas nos padrões IEEE 802.1D, IEEE 802.1Q, IEEE 802.3 e nos seguintes RFCs (*Request for Comments*): RFC 2697, RFC 2698 e RFC 4115. Estes módulos formam o núcleo funcional do comutador e implementam as principais funções dele. Neste trabalho quatro módulos são desenvolvidos e validados. Estes módulos foram projetados para serem inseridos na plataforma NetFPGA, formando o chamado "*User Data Path*". Esta plataforma foi desenvolvida pela universidade de Stanford para permitir a prototipagem rápida de hardware para redes. O primeiro módulo chamado de "Árbitro de entrada" decide qual das portas de entrada do comutador ele vai atender, para que os quadros que ingressam por essa porta sejam processados. Este módulo utiliza um algoritmo *Deficit Round Robin* (DRR). Este algoritmo corrige um problema encontrado no módulo original desenvolvido na plataforma NetFPGA. O segundo módulo é o "Pesquisador da porta de saída". O bloco principal deste módulo é o motor de classificação. A função principal do motor de classificação e aprendizagem de endereços MAC é encaminhar os quadros à suas respectivas portas de saída. Para cumprir esta tarefa, ele armazena o endereço MAC de origem dos quadros em uma memória SRAM e é associado a uma das portas de entrada. Este motor de classificação utiliza um mecanismo de *hashing* que foi provado que é eficaz em termos de desempenho e custo de implementação. São apresentadas duas propostas para implementar o motor de classificação. Os resultados da segunda proposta permite pesquisar efetivamente 62,5 milhões de quadros por segundo, que é suficiente para trabalhar a uma taxa *wire-speed* em um comutador Gigabit de 42 portas. O maior desafio foi conseguir a taxa de *wire-speed* durante o processo de "aprendizagem" usando uma memória SRAM externa. O terceiro módulo é o marcador de quadros. Este módulo faz parte do mecanismo de qualidade de serviço (QoS). Com este módulo é possível definir uma taxa máxima de transferência para cada uma das portas do comutador. O quarto módulo (*Output Queues*) implementa as filas de saída do comutador. Este módulo faz parte de plataforma NetFPGA, mas alguns erros foram encontrados e corrigidos durante o processo de verificação. Os blocos foram projetados utilizando Verilog HDL e visando as suas implementações em ASIC, baseado em uma tecnologia de 180 nanômetros da TSMC com a metodologia *Semi-Custom* baseada em *standard cells*. Para a verificação funcional foi utilizada a linguagem SystemVerilog. Uma abordagem de estímulos aleatórios restritos é utilizada em um ambiente de *testbench* com capacidade de verificação automática. Os resultados da verificação funcional indicam que foi atingido um alto porcentual de cobertura de código e funcional. Estes indicadores avaliam a qualidade e a confiabilidade da verificação funcional. Os resultados da implementação em ASIC amostram que os quatro módulos desenvolvidos atingem a freqüência de operação (125 MHz)

definida para o funcionamento completo do comutador. Os resultados de área e potência mostram que o módulo das Filas de saída possui a maior área e consumo de potência. Este módulo representa o 92% da área (115 K portas lógicas equivalentes) e o 70% da potência (542 mW) do "*User Data Path*".

Palavras-chave: Comutador ethernet, comunicação de dados, verificação funcional, systemVerilog, síntese física, microeletrônica.

Design, Functional Verification and Synthesis of Functional Modules for a Gigabit Ethernet Switch

ABSTRACT

This work presents the design, functional verification and synthesis of the functional modules of a Gigabit Ethernet switch. The functions of these modules are defined in the IEEE 802.1D, IEEE 802.1Q, IEEE 802.3 standards and the following RFCs (Request for Comments): RFC 2697, RFC 2698 and RFC 4115. These modules are part of the functional core of the switch and implement the principal functions of it. In this work four modules are developed and validated. These modules were designed to be inserted in the NetFPGA platform, as part of the "User Data Path". This platform was developed at Stanford University to enable the fast prototype of networking hardware. The first module called "input arbiter" decides which input port to serve next. This module uses an algorithm Deficit Round Robin (DRR). This algorithm corrects a problem found in the original module developed in the NetFPGA platform. The second module is the classification engine. The main function of the MAC address classification engine is to forward Ethernet frames to their corresponding output ports. To accomplish this task, it stores the source MAC address from frames in a SRAM memory and associates it to one of the input ports. This classification engine uses a hashing scheme that has been proven to be effective in terms of performance and implementation cost. It can search effectively 62.5 million frames per second, which is enough to work at wire-speed rate in a 42-port Gigabit switch. The main challenge was to achieve wire-speed rate during the "learning" process using external SRAM memory. The third module is the frame marker. This module is part of the quality of service mechanism (QoS). With this module is possible to define a maximum transmission rate for each port of the switch. The fourth module (Output Queues) implements the output queues of the switch. This module is part of the NetFPGA platform, but some errors were found and corrected during the verification process. These module were designed using Verilog HDL, targeting the NetFPGA prototype board and an ASIC based on a 180 nm process from TSMC with the Semi-custom methodology based on standard cells. For the functional verification stage is used the SystemVerilog language. A constrained-random stimulus approach is used in a layered-testbench environment with self-checking capability. The results from the functional verification indicate that it was reached a high percentage of functional and code coverage. These indicators evaluate the quality and reliability of the functional verification. The results from the ASIC implementation show that the four modules developed achieve the operation frequency (125 MHz) defined for the overall switch operation. The area and power results demonstrate that the Output Queues module has the largest area and power consumption. This module represents the 92% of area (115 K equivalent logic gates) and the 70% of power (542 mW) from the User Data Path.

Keywords: Ethernet switch, data communication, functional verification, SystemVerilog, physical synthesis, microelectronics.

1 INTRODUÇÃO

A informática tem revolucionado o modo de viver da sociedade atual. Este impacto não poderia ter sido tão grande sem as redes de comunicação de dados. Estas redes fazem com que todo dispositivo, com capacidade de processar informação, conectado a uma rede, possa interagir com outros dispositivos ligados à rede. Esta capacidade de interconexão, seja com ou sem fios (*wireless*), permite a execução de diversas aplicações como a Internet (*World Wide Web*), VoIP (voz sobre IP), vídeo *broadcasting*, etc.

A *Ethernet* (METCALFE; BOGGS, 1976) é o protocolo da segunda camada (camada de enlace, segundo o modelo OSI) mais empregado nas redes de comunicação de dados locais (LANs) e atualmente também nas redes metropolitanas (MANs). O rápido aumento do seu uso deve-se às características de baixo custo e alto desempenho assim como ao acelerado processo de padronização da taxa de transmissão: 10 Mbits/s em 1983, 100 Mbits/s em 1995, 1 Gbit/s em 1998, e 10 Gbits/s em 2002 (FRAZIER; JOHNSON, 1999). A *Ethernet* foi padronizada pela IEEE com a norma 802.3 (IEEE Std 802.3, 2005).

No começo, as LANs foram projetadas usando um canal de comunicação compartilhado. No final dos anos 80 e começo dos anos 90, dois fatores mudaram o jeito de projetar as LANs (SEIFERT; EDWARDS, 2008):

- A topologia mudou de um canal de comunicação compartilhado para um sistema de cabeamento estruturado centralizado em hubs;
- Os dispositivos de computação e as requisições das aplicações avançaram até ultrapassar a capacidade das LANs compartilhadas, limitando o desempenho global do sistema.

Esses fatores somados ao desenvolvimento da microeletrônica permitiram o desenvolvimento dos chamados "comutadores LAN", que permitem utilizar a estrutura de cabeamento já existente para criar uma rede microssegmentada. Essa mudança tem as seguintes vantagens:

- A possibilidade de eliminar as colisões de quadros (*frames*), se o modo de operação *full-duplex* for utilizado;
- Cada dispositivo tem largura de banda dedicada;
- A taxa de transmissão pode ser independente para cada dispositivo.

Estas características, bem como os baixos custos de implementação contribuíram para a ampla utilização destes dispositivos em redes corporativas e também em redes privadas.

1.1 Contribuição da dissertação de mestrado

O objetivo desta dissertação de mestrado é o projeto, verificação e implementação física em ASIC dos blocos funcionais para um comutador *Gigabit Ethernet*. Este trabalho está inserido no projeto EDDASIC que visa o desenvolvimento do protótipo em ASIC de um comutador EDD (*Ethernet Demarcation Device*). Este dispositivo serve como entidade controladora dos serviços de borda, entre os clientes e os provedores de serviço. A Figura 1.1 apresenta a visão do ambiente de inserção do comutador EDD. Os EDDs podem fornecer as funções de monitoramento e diagnóstico das SLAs, *loopback* remoto, limitação da banda, VLAN *stacking*, *priority queuing*, etc (BRAY, 2006).

O objetivo não é a implementação de um comutador *Gigabit Ethernet* completo portanto, neste trabalho não se detalham as características que deve ter um comutador com o protocolo RSTP (*Rapid Spanning Tree Protocol*), CoS ou *Class of Service*, *multiple tagging* (*Q-in-Q tagging*) (IEEE Std 802.1ad, 2006), *MAC-in-MAC* ou PBB (*Provider Backbone Bridge*) (IEEE Std 802.1ah, 2008).

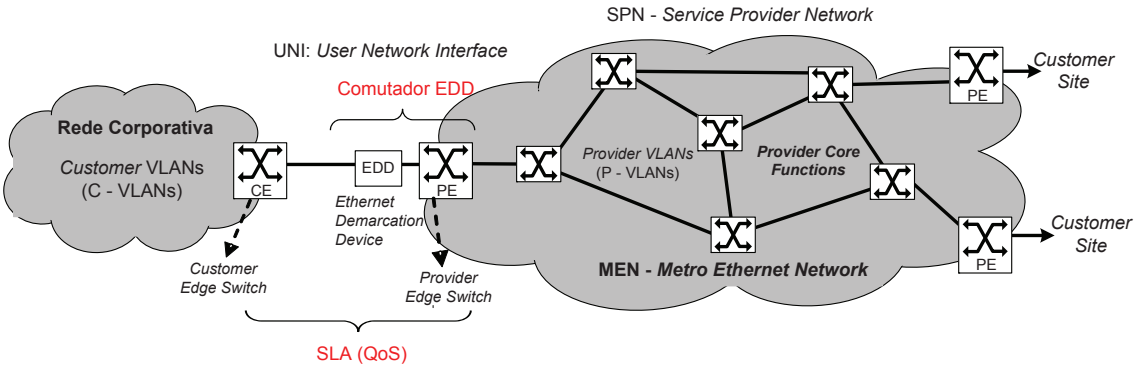

Figura 1.1: Visão do ambiente de inserção do Comutador (ROCHOL, 2009).

A arquitetura básica e as interfaces dos módulos deste trabalho são parte da plataforma NetFPGA (NAOUS et al., 2008). Porém, dois dos quatro módulos apresentados neste trabalho são projetos novos que adicionam novas funcionalidades ao comutador. Os outros dois módulos são projetos disponibilizados pela NetFPGA, mas foram modificados para melhorar a sua funcionalidade e foram adaptados para sua implementação em ASIC.

Considerando o objetivo do trabalho, as contribuições podem ser resumidas como segue:

- o projeto de módulos que implementam as funções de um comutador *Gigabit Ethernet*. Os resultados de desempenho do motor de classificação foram publicados em Tonfat e Reis (2010), Tonfat, Neuberger e Reis (2010), Tonfat e Reis (2011);
- A migração/adaptação para ASIC dos módulos projetados para FPGA;
- A aplicação de uma metodologia de verificação funcional aos módulos projetados que permite certificar que a funcionalidade do projeto está de acordo com sua especificação. Os resultados da verificação funcional e o método para criar os modelos de nível de transação (TLM) foram publicados em Tonfat, Neuberger e Reis (2011).

Durante o desenvolvimento deste trabalho muitos desafios foram encontrados. Para compreender a estrutura interna e as funções de um comutador *Ethernet* foi utilizado como referência o livro de Seifert e Edwards (2008). O processo de desenvolvimento

implicou muitas iterações até obter-se uma descrição que atendesse à especificação. Outro desafio foi trabalhar com códigos desenvolvidos por outras pessoas e depois ter que modificá-los para cumprir com novas especificações.

1.2 Organização da dissertação

O restante do trabalho está organizado como segue. No Capítulo 2, são apresentados os conceitos da tecnologia Ethernet que é o protocolo utilizado neste comutador.

O Capítulo 3 apresenta as funções de um comutador e as principais arquiteturas existentes. No Capítulo 4 é apresentado o fluxo de implementação física utilizado neste trabalho.

No Capítulo 5 é detalhada a arquitetura de cada um dos módulos desenvolvida nesta dissertação. No Capítulo 6 é mostrada a aplicação da metodologia de verificação funcional aos módulos projetados. No Capítulo 7 sao apresentados os resultados da implementação e da verificação funcional. Finalmente, no Capítulo 8 são mostradas as conclusões e os trabalhos futuros.

2 A TECNOLOGIA ETHERNET

2.1 História da Ethernet

A *Ethernet* foi criada nos laboratórios da Xerox Corporation em Palo Alto, Califórnia no ano de 1973. O Dr. Robert Metcalfe, chamado também o pai da *Ethernet*, criou o primeiro protótipo que trabalhava a 2,94 milhões de *bits* por segundo. Depois de alguns anos, as empresas Intel e DEC (*Digital Equipment Corporation*) juntaram-se com a Xerox para criar o padrão *Ethernet* com uma taxa de 10 Mb/s.

Em paralelo ao trabalho feito por DIX (DEC-Intel-Xerox), a IEEE formou o conhecido projeto 802 para criar uma maior infra-estrutura para a padronização das tecnologias LAN. Como não conseguiu-se um acordo relativo a tecnologia LAN a ser utilizada, criaram-se grupos de trabalho específicos para cada uma das tecnologias da época. Em 1983, o grupo de trabalho 802.3 apresentou o primeiro padrão da IEEE para redes locais baseadas em *Ethernet*. Salvo algumas diferenças, o padrão da IEEE e o padrão gerado por DIX são semelhantes. Posteriormente foram feitas algumas modificações no padrão, adicionando-se outros meios físicos para realizar a implementação, como a utilização de cabo coaxial e cabo de fibra ótica para conexões entre edificações.

Entre os anos 1991 e 1992 foi apresentada uma versão de *Ethernet* que trabalhava a uma taxa de 100 Mb/s e mantendo praticamente as mesmas características do padrão apresentado em 1983. O resultado foi novamente a padronização por parte da IEEE desta versão de *Ethernet* conhecida como *Fast Ethernet*, em 1995. Já no ano 1998 surge o padrão IEEE para redes *Ethernet* que trabalham a uma taxa de 1000Mb/s, conhecido como *Gigabit Ethernet*. Em 2002 foi apresentado o padrão para 10 *Gigabit Ethernet*. O trabalho continua em busca de redes *Ethernet* que trabalhem a taxas mais altas. Por isso, no ano 2010 foi apresentado o padrão para redes que trabalha com taxas de 40 Gb/s e 100 Gb/s.

2.2 A arquitetura das redes

Existem diferentes conceitos sobre uma rede. Por exemplo, os projetistas de interfaces físicas das redes não se preocupam com o formato dos dados que atravessam a rede. Por outro lado, um programador de aplicações baseadas na rede, conhece as interfaces e facilidades que oferece o sistema operativo, mas não se preocupa com o mecanismo utilizado para transportar os dados na rede. Outro fato, é que as aplicações e as tecnologias utilizadas nas redes estão em uma mudança constante e estas mudanças não devem afetar o funcionamento global da rede. A chave para conseguir isto é separar as diversas funções da rede em camadas. Esta organização permite a modificação de algumas funções sem comprometer o funcionamento das outras camadas.

No final dos anos 1970, a OSI definiu um modelo de camadas como é apresentado na Figura 2.1 (ISO/IEC Standard 7498-1, 1994).

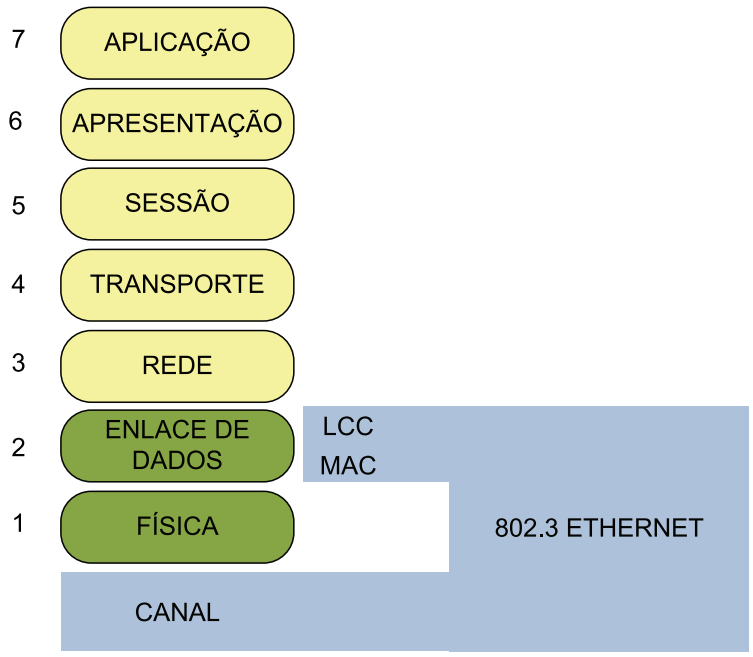

Figura 2.1: Modelo de referência OSI e sua relação com o padrão IEEE 802.3.

Os comutadores são dispositivos que permitem a ligação entre os elementos de uma rede, que trabalham basicamente nas duas primeiras camadas do modelo de referência OSI apresentadas na figura anterior.

A primeira é a camada física. Esta se encarrega das funções de codificação e decodificação dos sinais, sincronização dos sinais de relógio, entre outras. A implementação desta camada dependerá do canal de comunicação e do meio físico utilizado. Exemplos de interfaces da camada física são: *Token Ring*, *Ethernet* e FDDI. Para um dispositivo da rede, esta camada não mudará as funções de alto nível. Por exemplo, um comutador trabalhará do mesmo jeito tanto em uma rede *Fast Ethernet* (100 Mbps), que utiliza um cabo de par trançado, como em uma rede *Gigabit Ethernet* (1000 Mbps), que está implementada sobre cabos de fibra ótica.

A camada de enlace de dados (camada dois) oferece serviços que permitem a comunicação de dois dispositivos através de um canal físico. A comunicação pode ser ponto a ponto ou ponto a multiponto. Esta camada deve cumprir três funções principais:

- Delimitar os dados transmitidos em unidades discretas (*framing*);
- Identificar cada quadro com um endereço de origem e de destino;
- Oferecer um mecanismo de detecção de erros para evitar a transmisão de dados corrompidos aos níveis superiores.

Em geral, a tecnologia LAN trabalha nas duas primeiras camadas do modelo OSI e os comutadores realizam funções principalmente na camada de enlace. As camadas superiores oferecem os serviços necessários para estabelecer uma boa comunicação entre os dispositivos da rede. Alguns exemplos são: o encaminhamento dos pacotes através de várias redes (camada de rede), o ordenamento dos pacotes (camada de transporte),

mecanismos de autenticação (camada de sessão), a compressão dos dados (camada de apresentação) e, finalmente, as funções que o usuário final utiliza (camada de aplicação).

2.3 Conceito de encapsulamento

Na medida em que os dados cruzam através das camadas que formam os diferentes protocolos, estes são incrementados com alguma informação de controle para que a entidade correspondente no receptor possa interpretá-los e agir assim que os dados são recebidos.

As entidades de cada camada recebem os dados de um cliente em uma camada superior (também chamado SDU ou *Service Data Unit*) e adicionam informações de controle na forma de cabeçalhos ou fim de carga útil, formando o chamado PDU (*Protocol Data Unit*). No dispositivo receptor acontece o contrário, a entidade de uma camada recebe os dados de uma camada de nível inferior e processa a informação de controle para logo depois extrair esta informação e encaminhar os dados a uma camada superior. Este processo é chamado de desencapsulamento.

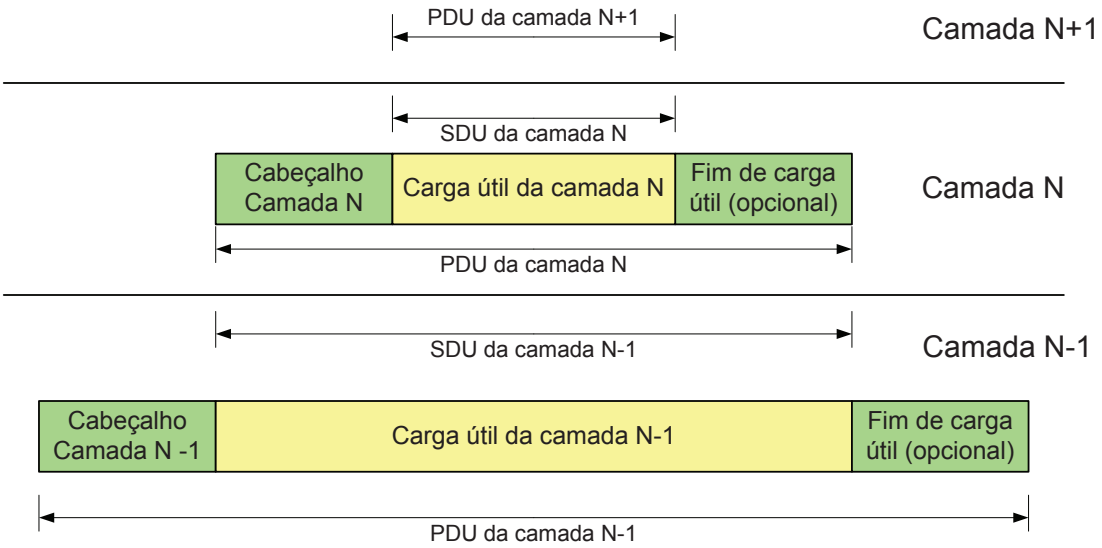

Figura 2.2: Encapsulamento dos dados através das camadas (SEIFERT; EDWARDS, 2008).

2.4 Endereçamento

Uma rede, por definição, envolve mais de um dispositivo. O objetivo da rede é habilitar a comunicação entre estes dispositivos. Um endereço é o meio para identificar um dispositivo na rede. Cada protocolo que suporta a troca de informação entre múltiplas estações deve ter algum mecanismo de identificação para as estações.

A única característica importante dos endereços é a singularidade (SEIFERT; EDWARDS, 2008), que permite identificar de modo inequívoco os dispositivos da rede. Cabe ressaltar que esta singularidade abrange somente o meio onde o protocolo está operando. Por exemplo, um endereço da camada de enlace de dados só precisa ser único dentro da rede local onde trabalha. Quando um dispositivo tem que se comunicar com um dispositivo em outra rede, terá que utilizar um endereço de um

protocolo da camada de rede.

Na tecnologia *Ethernet*, os endereços estão compostos por seis octetos de *bits*, formando um endereço de 48 *bits* que pode ser dividido em duas partes:

- Endereços que identificam só um dispositivo ou interface de rede. Estes também são chamados endereços *unicast*.
- Endereços que identificam um grupo de dispositivos. Estes também são chamados endereços *multicast*.

Como o endereço é utilizado pela sub-camada de controle de acesso ao meio (MAC) da camada de enlace de dados, o endereço também é chamado de endereço MAC. Para diferenciar os endereços *unicast* dos *multicast*, é utilizado o primeiro *bit* do primeiro *byte* do endereço, conforme mostra a Figura 2.3.

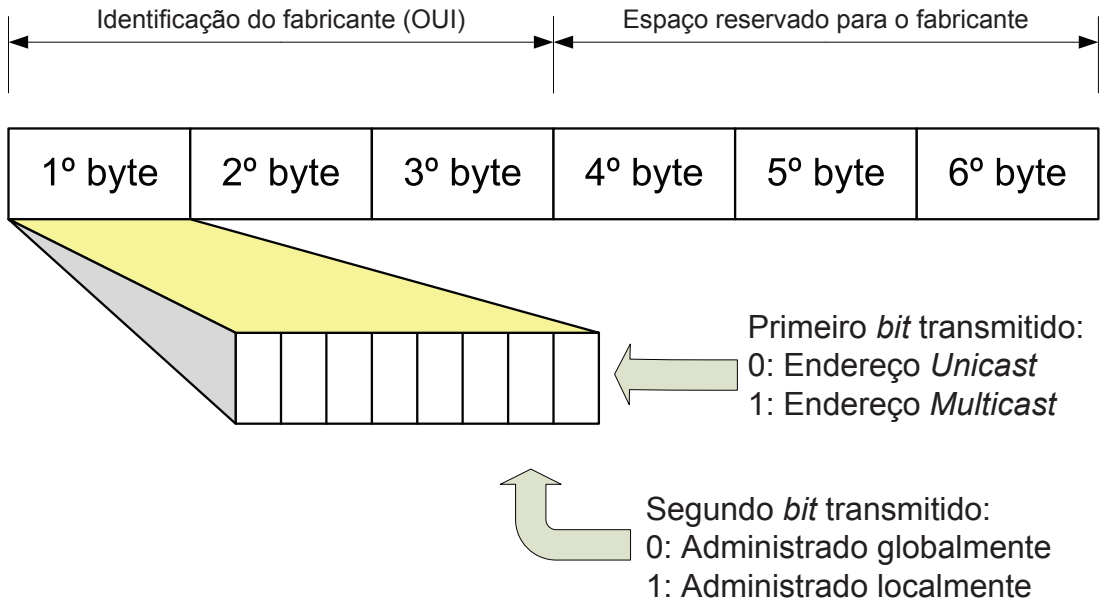

Figura 2.3: Formato do endereço MAC *Ethernet*.

O segundo *bit* do primeiro *byte* também é utilizado para diferenciar se o endereço é administrado localmente ou globalmente. Os endereços administrados globalmente são atribuídos pelos fabricantes na hora de produzir o dispositivo. Ao contrário, os endereços administrados localmente são atribuídos pelos administradores de rede no momento da configuração dos dispositivos. A IEEE é a encarregada de administrar a atribuição dos endereços MAC, atribuindo aos três primeiros *bytes* do endereço a identificação do fabricante ou OUI, deixando os outros três *bytes* seguintes para serem administrados pelo fabricante com um OUI atribuído. A IEEE só utiliza os endereços que são globalmente administrados, deixando os outros para uso livre pelos administradores de rede.

2.5 Quadro Ethernet

Um quadro (em inglês, *frame*) *Ethernet* é definido pela norma IEEE 802.3 (IEEE Std 802.3, 2005) no caso do quadro *Ethernet* padrão, e pela norma IEEE 802.1Q (IEEE Std

Figura 2.4: Formato do quadro Ethernet MAC.

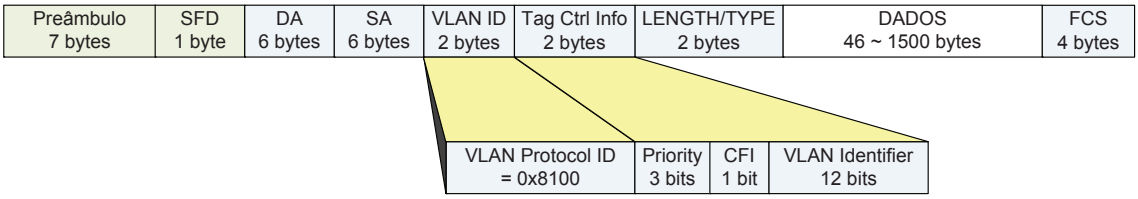

Figura 2.5: Formato do quadro Ethernet MAC com etiqueta VLAN.

802.1Q, 2006) para os quadros com etiquetas VLAN. Nas Figuras 2.4 e 2.5 são apresentados os dois formatos utilizados no presente trabalho.

A seguir é apresentada uma breve descrição dos campos do quadro *Ethernet* padrão e do quadro *Ethernet* com etiquetas VLAN das figuras anteriores:

- Preâmbulo: É um padrão alternado de '1's e '0's que permite aos receptores sincronizarem com o quadro recebido;
- *Start Frame Delimiter* (SFD): *Byte* responsável pela sinalização de início de quadro;
- Endereço de Destino (DA): Especifica o endereço do dispositivo que será o destino do quadro, ou o endereço que representa um grupo de dispositivos de destino (*multicast* ou *broadcast*);
- Endereço de Origem (SA): Especifica o endereço físico MAC do dispositivo que enviou o quadro. Este endereço sempre deve ser *unicast*;
- Tamanho do Quadro (LEN/TYP): Indica o valor do tamanho do quadro *Ethernet*. Se o valor representado neste campo for maior ou igual que 0x600 (hexadecimal), então o campo indica o tipo de quadro;
- Etiqueta VLAN (Opcional): A etiqueta VLAN de 12-bits permite a construção de um máximo de 4096 VLANs. Um identificador de protocolo (0x8100) é utilizado para detectar a etiqueta no quadro. O campo *Priority* serve para definir um nível de prioridade no quadro *Ethernet*. O campo CFI (*Canonical Format Indicator*) indica a ordem dos *bits* nos *bytes* (*Little or Big Endian*);
- Dados: Este campo possui um tamanho mínimo de 46 *bytes* e máximo de 1500 *bytes*. Se a quantidade de dados a ser enviada é menor que 46 *bytes*, então um conjunto de bits de enchimento (denominados em inglês *padding*) é acrescentado ao final dos dados para completar o tamanho mínimo do quadro;
- Sequência de Verificação de Quadro: O *Frame Check Sequence* (FCS) é o valor resultante do cálculo CRC-32 (32bit-Cyclic Redundancy Check), executado sobre o conteúdo dos campos de endereço de destino (DA), endereço de origem (SA), tamanho/tipo do quadro (LEN/TYP), e os dados (levando em conta o *padding*, se for gerado). O FCS é utilizado para verificação de erros no quadro recebido.

Além dos quadros Ethernet padrão também existem os chamados quadros *Jumbo*. Estes quadros possuem um tamanho superior ao máximo definido no padrão IEEE 802.3 (IEEE Std 802.3, 2005). Estes quadros são muito utilizados em interfaces *Gigabit Ethernet*.

2.6 Modos de Operação

As interfaces *Ethernet* podem trabalhar em dois modos de operação: *half-duplex* e *full-duplex*. Tradicionalmente as LANs compartilhavam um mesmo meio de comunicação. Portanto, não era possível transmitir e receber dados ao mesmo tempo. Este modo de operação é conhecido como *half-duplex*.

O modo *full-duplex* permite a transmissão e recepção simultânea de dados. Isto foi possível devido à utilização de comutadores que segmentam a rede e criam canais dedicados entre os dispositivos

2.7 Outras Tecnologias LAN

Além da *Ethernet*, existem outras tecnologias LAN difundidas, utilizadas ao nível comercial e que possuem seus próprios padrões. Amplamente conhecida é a tecnologia desenvolvida pela IBM chamada *Token Ring*.

A diferença entre a *Ethernet* e o *Token Ring* é que esta última conecta os dispositivos em laço. Cada dispositivo só pode transmitir diretamente para o dispositivo vizinho e só pode receber de um dispositivo vizinho. Para que o protocolo trabalhe corretamente, só um dispositivo pode transmitir dados num determinado momento. Essa autorização que recebe um dispositivo é chamada de *token*. O *token* vai circulando pelos dispositivos na rede, então quando um dispositivo quer transmitir, primeiro tem que esperar o *token* chegar.

Outro protocolo que surgiu junto com a *Ethernet* foi o FDDI ou *Fiber Distributed Data Interface*, o primeiro que conseguiu operar a 100 Mb/s. O seu uso não foi muito difundido nas redes locais devido à utilização exclusiva de fibra ótica como meio físico, apresentando alto custo de implementação. O funcionamento é similar ao *Token Ring* e a principal diferença é a utilização de duplo anel para interligar os dispositivos.

2.8 Padrão IEEE 802.1D

O padrão IEEE 802.1D descreve a operação dos comutadores. Além disso, contém uma descrição da tabela de endereços do comutador, os mecanismos de filtro e encaminhamento de quadros, o suporte de entradas dinâmicas e estáticas, assim como regras de encaminhamento e definições de filtros customizados. O padrão também estabelece um grupo de parâmetros que permitem a operação de comutadores interligados.

2.9 Padrão IEEE 802.1Q

O padrão IEEE 802.1Q é o padrão para redes virtuais sobre *Ethernet*. A tecnologia das redes virtuais permite separar a conectividade lógica da conectividade física. Isso significa que a conectividade das estações em uma rede não está mais restrita à topologia física. Este padrão é uma extensão do padrão IEEE 802.1D, mas com as especificações para que os comutadores trabalhem em um ambiente com redes virtuais.

3 COMUTADOR GIGABIT ETHERNET

Após apresentar a tecnologia *Ethernet*, neste capítulo serão explicadas as características de um comutador *Gigabit Ethernet*, que é o foco deste trabalho. Este capítulo está dividido em duas partes. A primeira descreve as principais funções de um comutador e a segunda parte expõe as principais arquiteturas existentes, com ênfase na plataforma NetFPGA que é a base deste trabalho.

3.1 Principais Funções de um Comutador

As seguintes funções descrevem principalmente o funcionamento do comutador quando recebe quadros. O tipo de operação do comutador é chamado de promíscuo. Chama-se assim porque recebe todos os quadros de cada porta sem levar em conta o endereço de destino do quadro. A operação típica de um dispositivo é receber somente os quadros que possuem o mesmo endereço de destino do dispositivo.

3.1.1 Encaminhamento

Quando um novo quadro aparece, o comutador analisa o endereço de destino e procura na tabela de endereços interna para escolher a porta por onde o quadro deve ser encaminhado. A seguir são apresentadas as diferentes possíveis respostas do comutador.

3.1.1.1 Unicast

Encaminhamento *unicast* é quando um quadro é encaminhado para uma única porta. Isto acontece quando o endereço de destino do quadro é do tipo *unicast* e o endereço estava armazenado na tabela de endereços.

3.1.1.2 Multicast

Encaminhamento *multicast* é quando um quadro é encaminhado para um grupo de portas. Isto acontece quando o endereço de destino é do tipo *multicast*. Os endereços *multicast* são utilizados por alguns protocolos específicos para poder enviar dados a mais de um dispositivo numa só transmissão. A Tabela 3.1 mostra alguns endereços *multicast* utilizados.

3.1.1.3 Flooding

O *Flooding* é quando um quadro é encaminhado por todas as portas exceto a porta de entrada do quadro. Isto acontece quando o comutador não encontra o endereço de destino armazenado na tabela de endereços.

Tabela 3.1: Endereços *multicast* reservados.

Endereço	Uso
01 - 80 - C2 - 00 - 00 - 00	Spanning Tree Protocol (STP)
01 - 80 - C2 - 00 - 00 - 01	IEEE 802.3 Full Duplex PAUSE
01 - 80 - C2 - 00 - 00 - 02	Link Aggregation Control e Marker Protocols
01 - 80 - C2 - 00 - 00 - 03	Reservado para uso no futuro
01 - 80 - C2 - 00 - 00 - 0F	Reservado para uso no futuro

3.1.1.4 Broadcast

O *broadcast* é um tipo de endereço *multicast*. Quando o quadro possui o endereço de destino FF - FF - FF- FF - FF - FF, o quadro é encaminhado para todas as portas do comutador exceto a porta de entrada do quadro.

3.1.1.5 Filtering

Filtragem de quadros ou *filtering* é restringir o encaminhamento de quadros que passam pelo comutador. O mais básico é aquele que impede que um quadro saia pela porta por onde entrou. Atualmente os comutadores permitem filtros customizados baseados nos endereços de origem/destino, o protocolo e outros critérios.

3.1.2 Aprendizagem

A chave para o correto funcionamento do comutador é a tabela de endereços MAC. Esta tabela relaciona endereços MAC com as portas físicas do comutador. No começo, as tabelas eram criadas manualmente pelos técnicos de rede. Atualmente, a tabela é criada automaticamente pelo comutador num processo chamado de aprendizagem.

3.1.3 Envelhecimento

Se o comutador só aprendesse endereços e nunca os apagasse, a maioria dos algoritmos de pesquisa na tabela demorariam mais para procurar e encontrar um endereço, a tabela chegaria a seu máximo e o comutador não poderia aprender mais endereços. Uma solução simples é apagar os endereços que não estão ativos. Este processo é chamado de envelhecimento. Um endereço fica ativo quando existem quadros com origem naquele endereço.

3.2 Principais Arquiteturas

Na implementação de comutadores encontram-se diferentes abordagens na literatura. O comutador pode-se dividir em três blocos principais: um que se encarrega de receber os quadros, processá-los e encontrar a porta de saída; a matriz de chaveamento ou em inglês, *switch matrix*, que se encarrega do mecanismo de encaminhamento dos quadros para as portas de saída e, por último, o bloco que se encarrega de ordenar os quadros que vão ser encaminhados em cada uma das portas de saída. As principais variantes encontram-se no projeto da matriz de chaveamento. As principais arquiteturas são: memória compartilhada, barramento compartilhado e comutador de malha ou *cross-point matrix*. O presente trabalho baseia-se na plataforma NetFPGA que é apresentada a seguir.

3.2.1 Plataforma NetFPGA

A plataforma NetFPGA foi desenvolvida por um grupo de pesquisa na universidade de *Stanford* (NetFPGA, 2010). Esta plataforma visa a rápida prototipação de *hardware* aplicado a redes. A Figura 3.1 mostra o diagrama da arquitetura. Possui tanto blocos de *hardware* descritos em *Verilog*, como o software necessário para interagir com o computador.

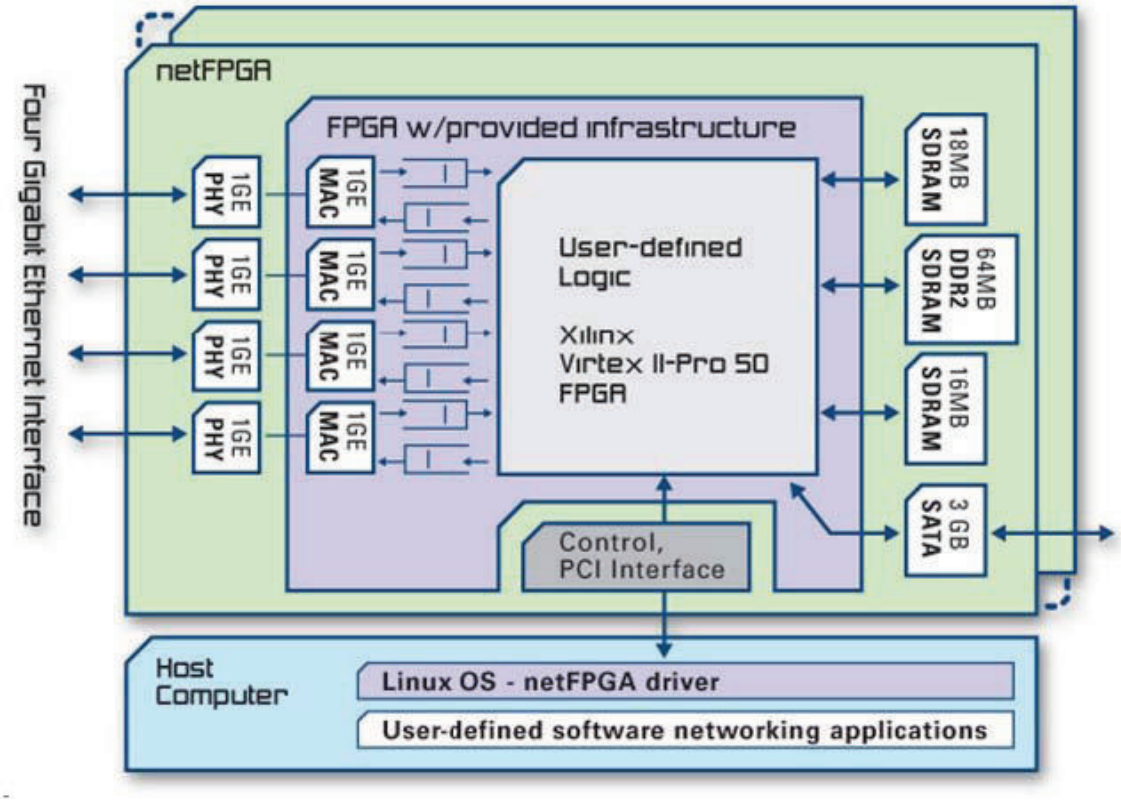

Figura 3.1: Diagrama da plataforma NetFPGA (NetFPGA, 2010).

A plataforma fornece os recursos necessários para a implementação de dispositivos de redes como um comutador. Basicamente está formado por um FPGA, os PHYs (*Physical Interfaces*) *Ethernet* e memórias externas.

Os blocos de *hardware* dividem-se em dois grupos: o caminho por onde vão circular os quadros ou parte operativa e os blocos que permitem a configuração desta parte operativa e a interação com um CPU externo. Na Figura 3.2 é apresentada a arquitetura da parte operativa. Esta arquitetura permite a adição de novos módulos sem muita complicação.

A parte operativa possui dois barramentos, um para dados e outro barramento para configurar e ler os registradores dos módulos. O barramento de dados possui uma largura de 64 bits para os dados do quadro *Ethernet* e 8 *bits* para controle.

Quando os quadros ingressam na parte operativa, estes têm um formato definido pela plataforma, ilustrado na Figura 3.3. Todos os quadros possuem um cabeçalho identificado com a palavra de controle `0xFF`. Este cabeçalho contém 4 dados do quadro: a porta de entrada, a porta de saída, e o tamanho em *bytes* e palavras (*words*).

Adicionalmente é possível a inserção de mais cabeçalhos, definidos pelos módulos da parte operativa. Os dados do quadro são definidos com a palavra de controle `0x00`. O fim do quadro é definido indicando-se com um dos *bits* de controle, a posição do último *byte*

do quadro. No exemplo da Figura 3.3, o último *byte* do quadro se encontra no segundo *byte* mais significativo. Portanto, a palavra de controle da última palavra do quadro é `0x40=0b0100_0000`.

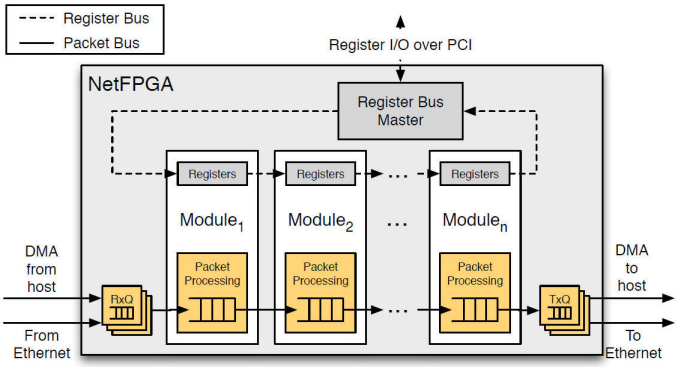

Figura 3.2: Estrutura modular da plataforma NetFPGA (NAOUS et al., 2008).

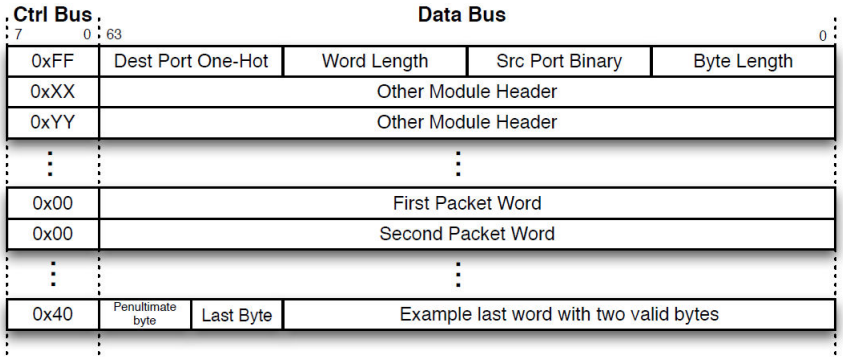

Figura 3.3: Formato interno dos quadros *Ethernet* na plataforma NetFPGA (NAOUS et al., 2008).

Na Figura 3.4 pode ser observado o diagrama de blocos da parte operativa dos módulos propostos. Os quadros *Ethernet* podem vir de duas fontes:

- Interfaces *Gigabit Ethernet* que são controladas através dos blocos GEMAC. Na plataforma NetFPGA é utilizado o núcleo IP da *Xilinx*. Neste trabalho é utilizado o núcleo desenvolvido no trabalho de mestrado de Tomás (2009).

- CPU: A plataforma NetFPGA possui uma interface PCI que permite que sejam inseridos quadros vindos da CPU utilizando controladores DMA.

O tempo de execução de cada bloco pode ser diferente, porque a comunicação entre os módulos é do tipo *handshake* com a ajuda de *buffers* (FIFOs) intermediários. O diagrama de tempos de uma comunicação de dados típica de dois módulos é apresentado na Figura 3.5. Quando um módulo precisa enviar dados para outro módulo, o módulo transmissor revisa o estado do sinal RDY (*Ready*) do módulo receptor. Se o módulo receptor estiver pronto para receber dados (RDY = 1), então o módulo transmissor ativa o sinal WR (*Write*) e envia os dados pelos sinais DATA e CTRL. Todos os sinais são síncronos com o sinal CLK.

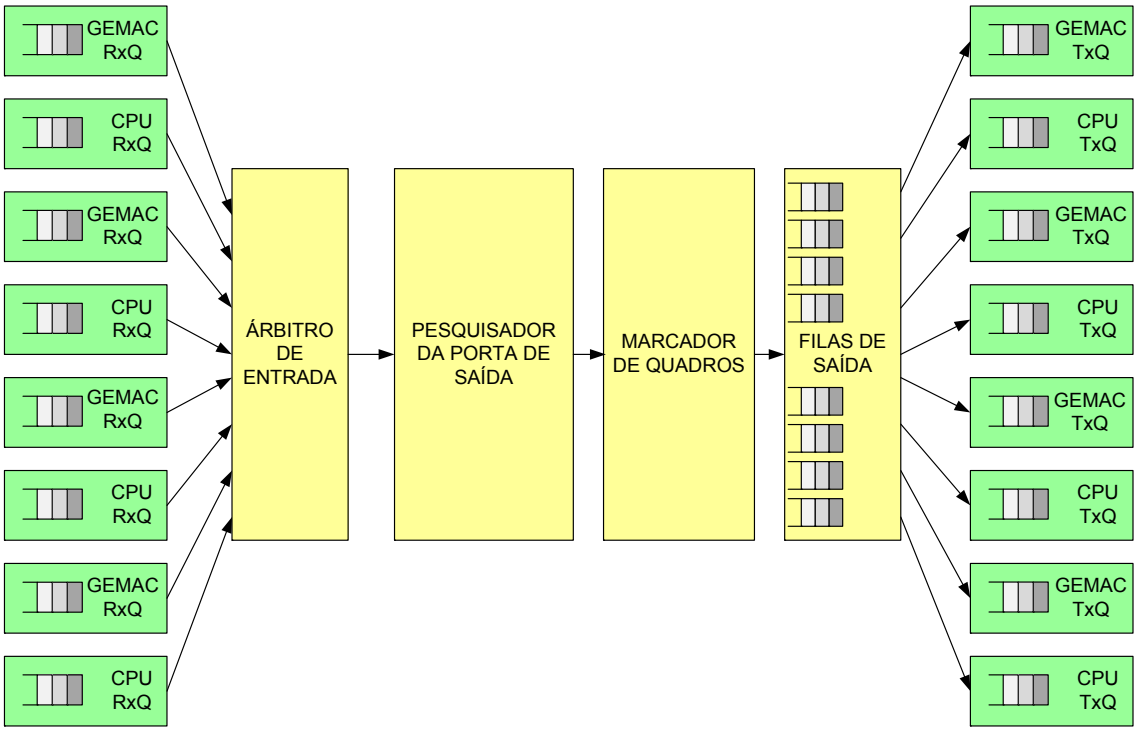

Figura 3.4: Diagrama da parte operativa do comutador *Gigabit Ethernet*.

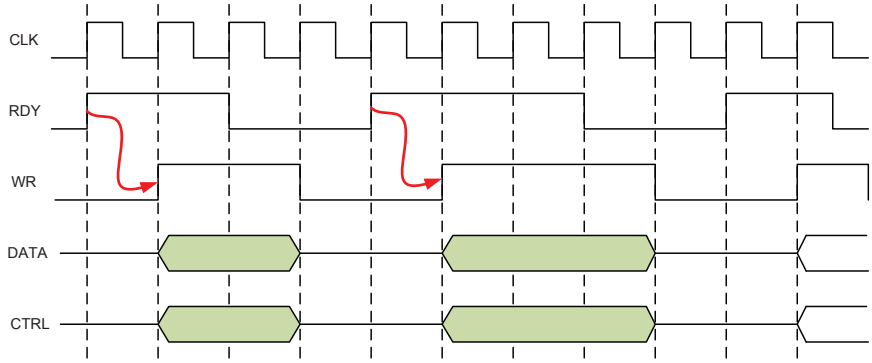

Figura 3.5: Diagrama de tempos da comunicação entre os módulos da parte operativa.

No capítulo seguinte é apresentado o desenvolvimento dos módulos funcionais da parte operativa do comutador *Gigabit Ethernet*.

4 FLUXO DE IMPLEMENTAÇÃO PARA UM ASIC

Neste capítulo é apresentado o fluxo de implementação utilizado para sintetizar os módulos projetados em descrição RTL para a implementação física (leiaute). O fluxo de implementação para um ASIC está fortemente ligado à metodologia utilizada.

Neste trabalho é utilizada a metodologia *semi-custom* baseada em *standard cells*. Esta metodologia consiste em utilizar um conjunto de células (biblioteca) que implementam funções lógicas, que são pré-projetadas para uma determinada tecnologia. São utilizadas ferramentas EDA (*Electronic Design Automation*) para automatizar algumas etapas do fluxo de projeto. Esta metodologia permite criar circuitos com milhões de transistores, reduzindo os tempos de desenvolvimento de um ASIC. A desvantagem é que devido a natureza restrita da biblioteca de células, a possibilidade de obter um circuito com o melhor desempenho é reduzida.

As etapas para a obtenção de um ASIC são similares ao fluxo para FPGA, tendo como principal diferença a criação do leiaute completo do *chip*, no caso do ASIC. Já no FPGA são utilizados blocos genéricos prontos disponíveis no *chip*. Esta diferença na flexibilidade do *chip* traz uma mudança significativa no consumo de potência e no desempenho.

Na implementação do circuito em ASIC, foi utilizada a biblioteca de células da *foundry* TSMC 180nm. Esta tecnologia possui oito camadas de metal e trabalha com uma tensão de alimentação de 1.8V.

4.1 Descrição das etapas do fluxo de implementação

O fluxo de implementação utilizado neste trabalho é apresentado na Figura 4.1. É muito semelhante ao fluxo utilizado em Silva (2010).

Este fluxo contém etapas de DFT (*Design for Testability*), sendo inseridas cadeias de *scan* em todo o circuito. Também é aplicada uma técnica para reduzir o consumo de potência dinâmica do circuito, chamado *clock-gating*. Esta técnica reduz o chaveamento dos elementos de memória, bloqueando o sinal de relógio.

O fluxo começa com a especificação do projeto. A especificação consiste num documento escrito que detalha as características do circuito ou um modelo do circuito descrito em alguma linguagem de alto nível (C, C++, SystemC).

O projetista transforma a especificação em uma descrição mais detalhada do circuito com uma linguagem de descrição de *hardware* (VHDL ou *Verilog*). Depois desta etapa é utilizada a primeira etapa de verificação, a verificação funcional. A verificação funcional visa certificar se a funcionalidade do projeto está de acordo com sua especificação. Para esta etapa é desenvolvido um ambiente de verificação que pode ser mais complexo do que o próprio circuito. A ferramenta utilizada para realizar a verificação funcional foi o

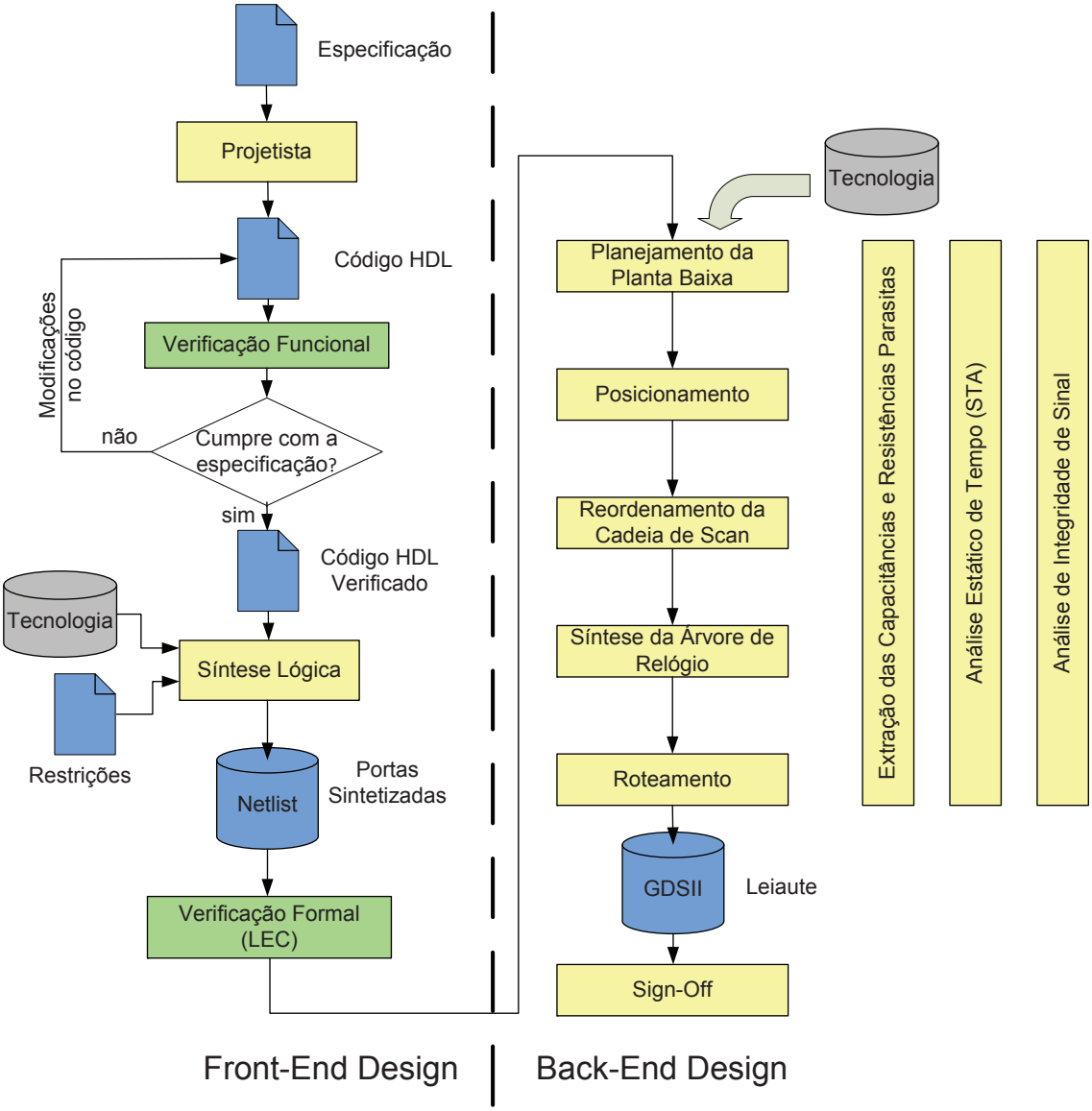

Figura 4.1: Diagrama do fluxo de síntese em ASIC.

Modelsim versão 6.6b (MODELSIM SE USER'S MANUAL, 2010). No Capítulo 6 são apresentados mais detalhes sobre a verificação funcional.

Somente quando o código HDL verificado cumpre com as especificações, a etapa seguinte é realizada. A síntese lógica transforma o código HDL verificado em um conjunto de células chamado *netlist*. Este *netlist* depende da tecnologia utilizada e das restrições definidas para o projeto, como a frequência de operação. Esta etapa envolve um conjunto maior de passos que são apresentados na Figura 4.2. A ferramenta para realizar a síntese lógica foi o RTL *Compiler* (USING ENCOUNTER RTL COMPILER, 2009).

A etapa de síntese lógica precisa de três entradas basicamente. A primeira são os arquivos HDL que descrevem o circuito. A segunda entrada são os arquivos da biblioteca de células da tecnologia que se pretende usar. Estes arquivos contém principalmente a informação de tempo, consumo e área de cada uma das células da biblioteca. A terceira entrada é o arquivo de restrições do projeto. Este arquivo contem informação dos objetivos do projeto como a frequência de operação, o consumo de potência, etc.

Geralmente é utilizado um *script* para executar todos os passos que envolvem a síntese

lógica. É importante mencionar que dentro destes passos temos a inserção da cadeia de *scan* para o DFT e também a inserção dos *clock gates* para reduzir o consumo de potência dinâmico.

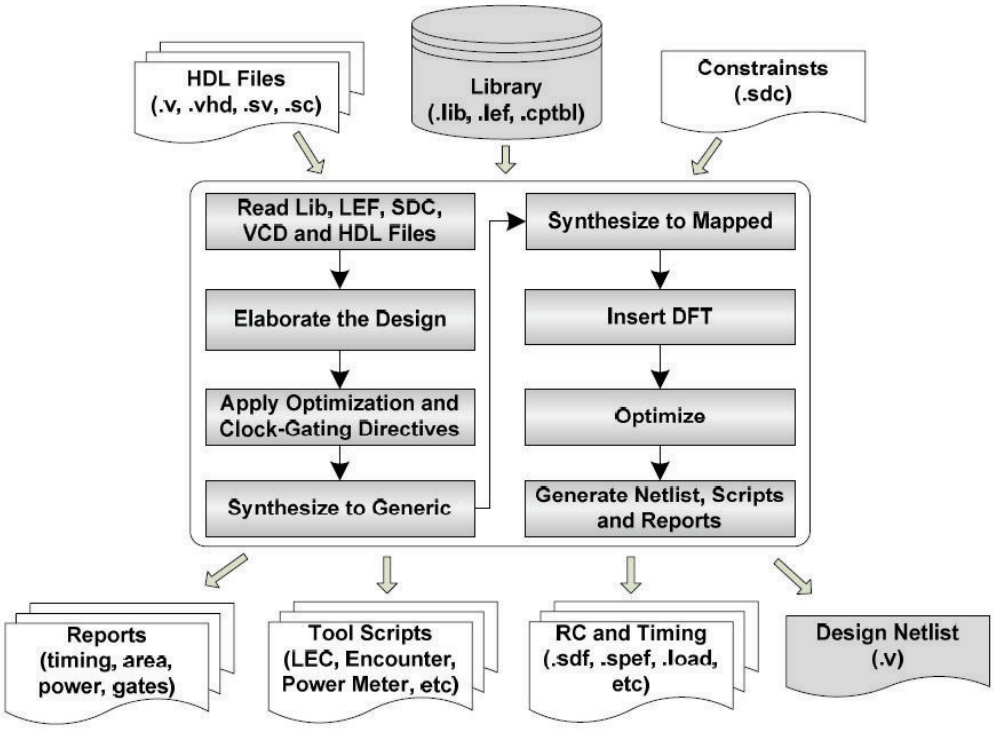

Figura 4.2: Diagrama do fluxo da síntese lógica (SILVA, 2010).

A etapa de síntese lógica precisa ser verificada com uma ferramenta de verificação formal (*Logic Equivalence Checking*). Na verificação formal pretende-se conferir que o *netlist* gerado tenha a mesma funcionalidade do código HDL original. A ferramenta para realizar a verificação formal foi o *Conformal* LEC (ENCOUNTER CONFORMAL EQUIVALENCE CHECKING USER GUIDE, 2008).

O conjunto de etapas feitas até este ponto é conhecido como *Front-end*. As etapas seguintes do fluxo de implementação são conhecidas como o *Back-end*. O *Back-end* inicia com todas as etapas envolvidas na síntese física do circuito. A ferramenta utilizada para a síntese física foi o *SoC Encounter* (ENCOUNTER USER GUIDE, 2009). As etapas da síntese física são apresentadas com maior detalhe na Figura 4.3. Estas etapas têm como objetivo gerar o leiaute final do circuito. Como apresentado na figura, temos dois fluxos definidos. O primeiro é o fluxo que define os passos para a geração do leiaute do circuito e o segundo é o fluxo que define as etapas de verificação do primeiro fluxo.

Nos capítulos seguintes, são detalhadas as arquiteturas de cada um dos módulos desenvolvidos neste trabalho e o processo de verificação funcional de cada um deles. Alguns dos módulos foram criados para serem implementados em FPGA. Estes códigos foram modificados para poderem ser implementados em ASIC. No capítulo seguinte são apresentadas com detalhe estas modificações.

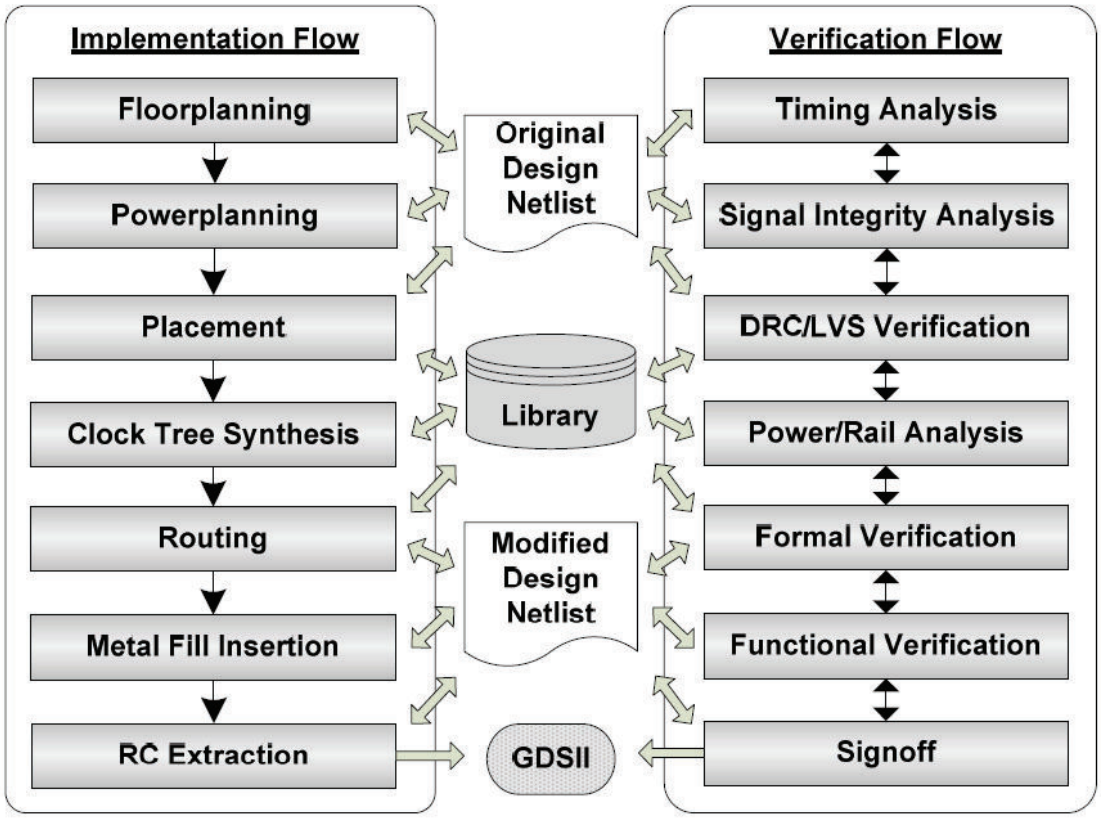

Figura 4.3: Diagrama do fluxo de síntese física (SILVA, 2010).

5 DESENVOLVIMENTO DOS MÓDULOS FUNCIONAIS DO COMUTADOR GIGABIT ETHERNET

Neste capítulo, é apresentada a descrição das arquiteturas de cada um dos quatro módulos funcionais do comutador. Estes módulos por sua vez são compostos por blocos menores. Dois dos módulos são produto da modificação dos módulos originais inclusos na plataforma NetFPGA e outros dois são arquiteturas totalmente novas.

Devido à estrutura modular dos blocos da plataforma NetFPGA apresentada no capítulo anterior, os módulos possuem interfaces de comunicação semelhantes. Os únicos módulos que possuem interfaces adicionais são aqueles que precisam acessar as memórias externas (SRAM e DDR2 SDRAM). O detalhe dos sinais da interface entre os módulos é apresentado na Tabela 5.1. Os quatro módulos precisam operar a uma frequência de 125 MHz. Estes quatro módulos, segundo a organização de módulos na plataforma NetFPGA, pertencem ao chamado "*User Datapath*".

Tabela 5.1: Interface de comunicação comum aos quatro módulos.

Sinal	Tipo	Descrição
Interface do módulo com o barramento de dados da parte operativa		
out_data [63:0]	Saída	Barramento de dados da parte operativa.
out_ctrl [7:0]	Saída	Barramento de controle da parte operativa.
out_wr	Saída	Indica que os dados nas saídas out_data e out_ctrl são válidos.
out_rdy	Entrada	Indica se o módulo seguinte pode receber dado.
in_data [63:0]	Entrada	Barramento de dados da parte operativa.
in_ctrl [7:0]	Entrada	Barramento de controle da parte operativa.
in_wr	Entrada	Indica que os dados nas entradas in_data e in_ctrl são válidos.
in_rdy	Saída	Indica ao módulo anterior que pode receber dado.
Interface do módulo com o barramento de registradores		
reg_req_in	Entrada	Requisição de acesso a um registrador.
reg_ack_in	Entrada	Resposta à requisição de acesso.

reg_rd_wr_L_in	Entrada	Indica se a operação é de leitura (alta) ou escrita (baixa).
reg_addr_in [22:0]	Entrada	Barramento de endereços dos registradores.
reg_data_in [31:0]	Entrada	Barramento de dados dos registradores.
reg_src_in [1:0]	Entrada	Sinal utilizado pelos iniciadores das requisições para identificar suas respectivas respostas.
reg_req_out	Saída	Requisição de acesso a um registrador.
reg_ack_out	Saída	Resposta à requisição de acesso.
reg_rd_wr_L_out	Saída	Indica se a operação é de leitura (alta) ou escrita (baixa).
reg_addr_out [22:0]	Saída	Barramento de endereços dos registradores.
reg_data_out [31:0]	Saída	Barramento de dados dos registradores.
reg_src_out [1:0]	Saída	Sinal utilizado pelos iniciadores das requisições para identificar suas respectivas respostas.

5.1 Árbitro da Entrada

Este módulo cumpre a função de multiplexar os quadros vindos das portas de entrada do comutador *GigE*. Estes quadros são multiplexados para serem processados em um único caminho de dados. Como foi mencionado anteriormente, a interface deste módulo é a padrão dos módulos do NetFPGA com a diferença que na entrada possui uma interface independente para cada porta do comutador. Este módulo decide qual fila de entrada (*Rx queue*) deve ser encaminhada para o seguinte módulo, baseado em um algoritmo de escalonamento (*scheduling*).

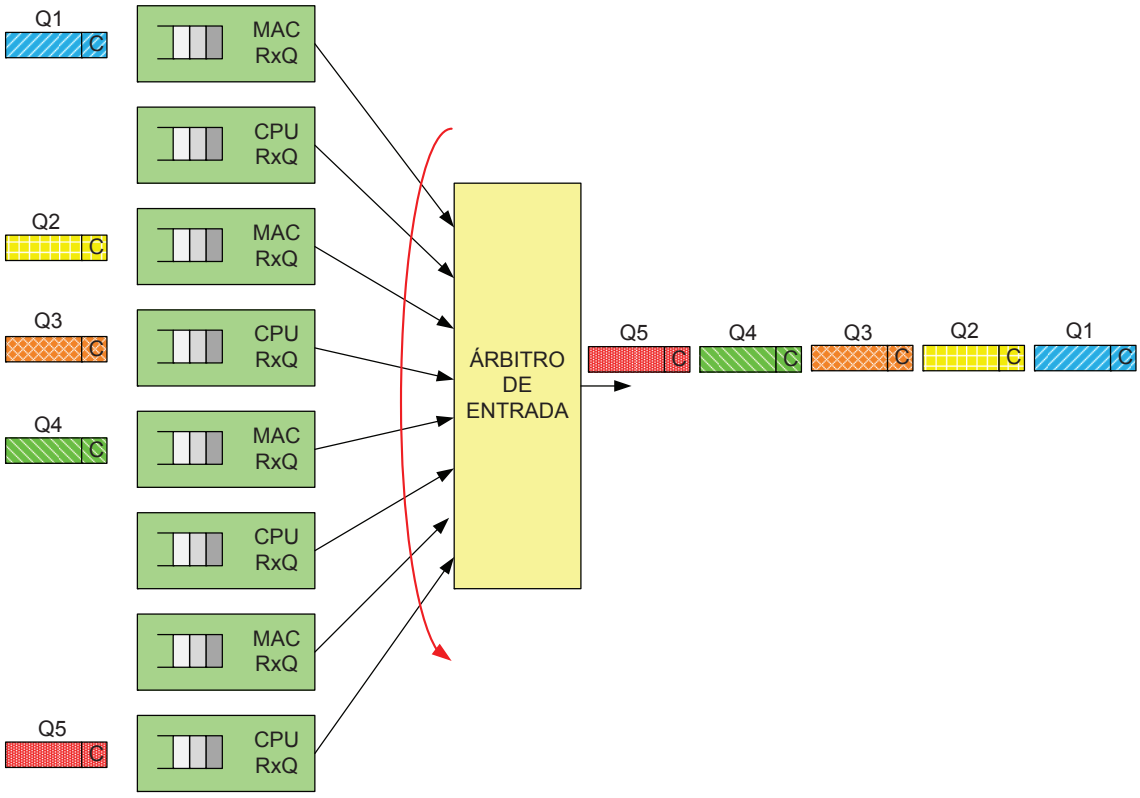

Figura 5.1: Diagrama do funcionamento do módulo Árbitro da Entrada.

O algoritmo utilizado no módulo original embarcado na plataforma NetFPGA é o *Round Robin* ou *RR*. Este algoritmo é um dos mais simples para a manipulação de um conjunto de entradas. O módulo percorre cada uma das filas de entrada e se encontra um quadro nesta fila, o quadro é retirado da fila e colocado na saída do módulo como se apresenta na Figura 5.1. Quando uma fila de entrada não possui nenhum quadro, o árbitro ignora esta fila e analisa a fila seguinte.

Este algoritmo possui um grande problema. O problema encontra-se na largura de banda (*bandwidth*) designada a cada uma das filas. A largura de banda de cada uma das filas é determinada pelo tamanho de cada um dos quadros que passam pelas filas. Ter uma largura de banda semelhante a cada fila nos permitirá oferecer uma qualidade de serviço (QoS) justa para cada porta do comutador. Então, controlar a largura de banda por fila é uma característica muito importante e desejável.

Para mostrar com maior clareza este problema, foi desenvolvido um ambiente de teste (*testbench*) para simular o módulo em dois cenários. O primeiro cenário envia quadros com diferentes tamanhos em cada uma das filas mas, designando uma faixa de tamanhos válidos diferente para cada fila. No segundo cenário, enviam-se quadros com diferentes tamanhos mas, a faixa de tamanhos válidos para cada fila é a faixa válida para um quadro *Ethernet* padrão (de 16 a 1522 *bytes*). Na subseção 5.1.1 é explicado com mais detalhes o procedimento realizado.

Uma alternativa para solucionar o problema da largura de banda por fila no algoritmo *Round Robin* é utilizar o algoritmo *Deficit Round Robin* (SHREEDHAR; VARGHESE, 1996). Este algoritmo encontra-se no grupo dos algoritmos que implementam o chamado "*fair queuing*" ou escalonamento justo. A diferença deste algoritmo para outros esquemas que possuem uma complexidade de $O(ln(n))$, onde n é o número de fontes de dados é que este algoritmo possui uma complexidade de $O(1)$. Isto significa que independentemente do número de entradas, o tempo para processar cada entrada será constante. O algoritmo define uma quantidade média de dados (*quantum*) a serem transferidos pelo escalonador. Cada fila possui uma quantidade de créditos, que é o número de dados que podem ser transferidos (contador de déficit).

```
Inicialização:
  for i = 0 to N      (onde N é o número de filas de entrada)
    créditos(i) = 0

  while {TRUE} {
    if FilaNoVazia(i)
      creditos(i) = creditos(i) + quantum(i)
      while { creditos(i) <= tamanhoDoQuadro } (O primeiro quadro da fila i)
        creditos(i) = creditos(i) - tamanhoDoQuadro
        enviaParaSaida(quadro da fila i)
        (Se houver outro quadro na fila esse quadro se torna o primeiro)
    else
      creditos(i) = 0
    end if
    i ++ (seleciona a fila seguinte)
  }
```

Figura 5.2: Algoritmo DRR implementado.

O escalonador percorre cada uma das filas e se ela estiver vazia, zera seus créditos. Caso contrário, o escalonador soma o valor do *quantum* aos créditos da fila. Se o tamanho do quadro na fila for menor que os seus créditos, então retira o quadro da fila e recoloca-o na saída, atualizando o valor dos créditos, e subtraindo o valor do tamanho do quadro. Se houver outro quadro na fila que possua um tamanho menor que a quantidade de créditos, ele também será encaminhado para a saída. Na Figura 5.2 é apresentado o algoritmo *DRR* implementado. Este algoritmo garante equidade (*fairness*) para cada uma das filas de entrada. Esta propriedade será verificada com o *testbench*.

5.1.1 Análise das características dos algoritmos *Round Robin* e *Deficit Round Robin*

Para a análise das características dos algoritmos utilizados para multiplexar os quadros vindos das portas de entrada do comutador *GigE* desenvolveu-se um ambiente de teste (*Testbench*) que visa mostrar claramente a largura de banda (*data bandwidth*) de cada uma das filas do árbitro.

Tabela 5.2: Descrição dos parâmetros do ambiente de teste.

Parâmetro	Descrição
MODE_PKT_QUANTITY	Escolher se a quantidade de dados inseridos em cada fila depende do número de pacotes ou do número de *bytes*. 0: depende do número de pacotes ,1: depende do número de *bytes*.
DRR	Escolher qual módulo vai ser simulado. (Round Robin ou Deficit Round Robin)
DATA_WIDTH	Define a quantidade de *bits* para o barramento de dados. (Default: 64)
CTRL_WIDTH	Define a quantidade de *bits* para o barramento de controle. (Default: DATA_WIDTH/8)
STAGE_NUMBER	Define um parâmetro da máquina de estados do árbitro. (Default: 2)
NUM_QUEUES	Define o número de filas.
PERIOD	Define o período do relógio.
NUM_PKTS	Define a quantidade de pacotes a serem inseridos. Depende do parâmetro MODE_PKT_QUANTITY.
NUM_BYTES	Define a quantidade de *bytes* a serem inseridos. Depende do parâmetro MODE_PKT_QUANTITY.
PKT_MIN_LENGTH_X	Define o tamanho mínimo do pacote em *bytes* da fila X. X é o número da fila.
PKT_MAX_LENGTH_X	Define o tamanho máximo do pacote em *bytes* da fila X. X é o número da fila.

O ambiente de teste foi desenvolvido utilizando-se Verilog e possui um conjunto de parâmetros que permite configurar os cenários da simulação. A lista dos parâmetros é mostrada na Tabela 5.2.

Na primeira etapa, o *testbench* gera os dados que serão inseridos em cada fila. Dependendo dos valores designados aos parâmetros PKT_MIN_LENGTH e PKT_MAX_ LENGTH de cada fila, o *testbench* gera pacotes de tamanho aleatório dentro da faixa determinada pelos parâmetros mencionados. Dependendo do parâmetro MODE_PKT_

QUANTITY é possível gerar-se a mesma quantidade de pacotes ou a mesma quantidade de *bytes* para cada fila.

Na segunda etapa, os pacotes são inseridos no módulo de maneira contínua e o *testbench* começa a analisar a saída do módulo. O *testbench* conta o número de pacotes e *bytes* que passam através da saída. É possível identificar-se a fila de origem do pacote, pois cada *word* de cada fila contém o número da fila de origem.

Foram aplicados dois cenários para cada algoritmo implementado. A Tabela 5.3 apresenta estas características.

Tabela 5.3: Características dos cenários aplicados.

Cenário 1	Cenário 2
- 80000 *bytes* inseridos em cada fila.	- 80000 *bytes* inseridos em cada fila.
- Tamanho dos pacotes (em *bytes*):	- Tamanho dos pacotes (em *bytes*):
fila 0: 16 - 150	fila 0: 16 - 1522
fila 1: 150 - 300	fila 1: 16 - 1522
fila 2: 300 - 500	fila 2: 16 - 1522
fila 3: 500 - 700	fila 3: 16 - 1522
fila 4: 700 - 900	fila 4: 16 - 1522
fila 5: 900 - 1100	fila 5: 16 - 1522
fila 6: 1100 - 1300	fila 6: 16 - 1522
fila 7: 1300 - 1522	fila 7: 16 - 1522

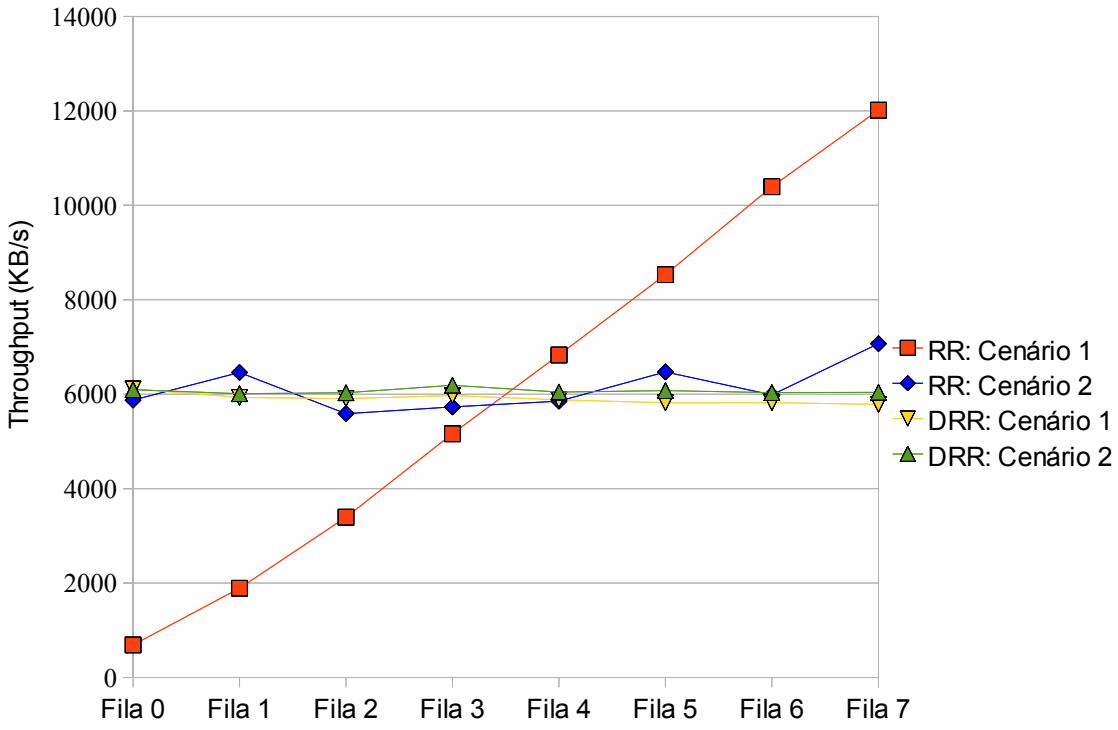

Figura 5.3: Largura de banda (*data bandwidth*) por fila para cada um dos algoritmos analisados.

Os resultados obtidos são apresentados na Figura 5.3. A Figura mostra que a largura de banda por fila no caso do algoritmo *Round Robin* muda bastante com a troca de cenário

entre 1 e 2. No cenário 1, as filas com quadros de maior tamanho obtém maior largura de banda. No cenário 2, quando os quadros tem uma distribuição uniforme, mantém-se a equidade da largura de banda entre as filas. No caso do algoritmo *Deficit Round Robin*, a equidade na largura de banda mantém-se em ambos os cenários. Isto valida a demonstração teórica apresentada em Shreedhar e Varghese (1996) sobre a equidade no serviço às entradas do módulo.

5.1.2 Modificações feitas ao módulo original

O diagrama de blocos do módulo é apresentado na Figura 5.4. Este módulo é composto por FIFOs em cada uma das entradas, uma máquina de estados e um módulo para gerenciar os registradores de configuração e estado.

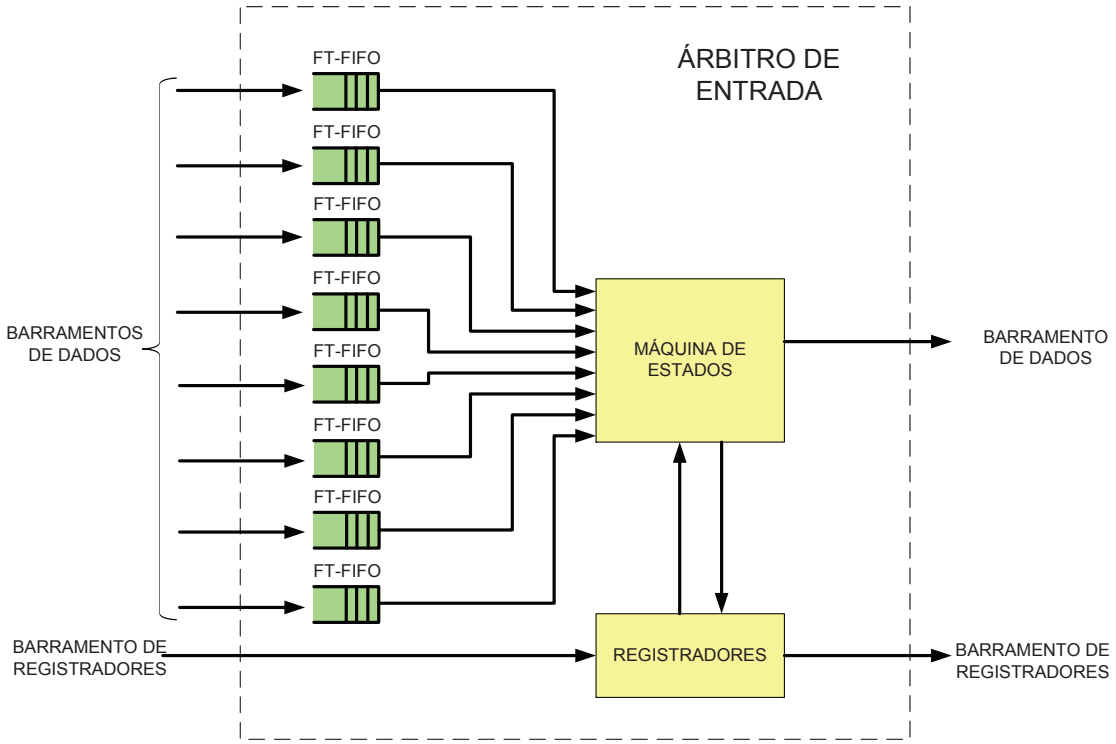

Figura 5.4: Diagrama de blocos do árbitro de entrada.

O módulo original possui FIFOs padrão, onde padrão refere-se ao modo de leitura. Este tipo de FIFOs são chamados neste trabalho de *small FIFOs*. Estes blocos são síncronos na leitura e escrita, e como é mostrado na Figura 5.5, possuem marcadores de vazio (*empty*), cheio (*full*), quase cheio (*almost_full*) e um cheio programavél pelo projetista (*prog_full*). Internamente uma FIFO é um registrador de *n bits* com *m* posições junto com ponteiros de escrita e leitura e lógica, que controla esses ponteiros e os marcadores de estado.

O modo de operação de leitura e escrita da FIFO padrão é apresentado na Figura 5.6. Segundo este diagrama de tempos, somente quando é feito uma requisição de leitura (READ_EN) o dado aparece na saída e, portanto, é retirado do registrador interno também porque o ponteiro de leitura é atualizado. Nesta FIFO o sinal de READ_EN significa "Quero ler uma palavra". Esta característica não interfere no funcionamento do módulo porque o algoritmo *Round Robin* transfere, sem interferência de nenhuma condição, um quadro de cada fila por vez.

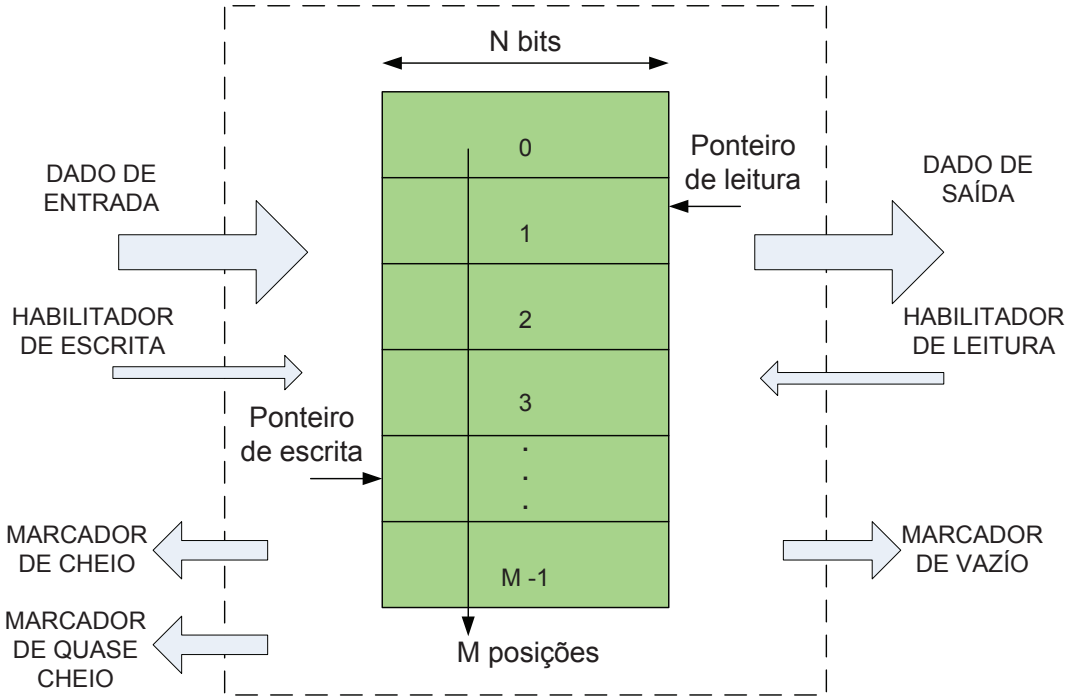

Figura 5.5: Estrutura de uma FIFO.

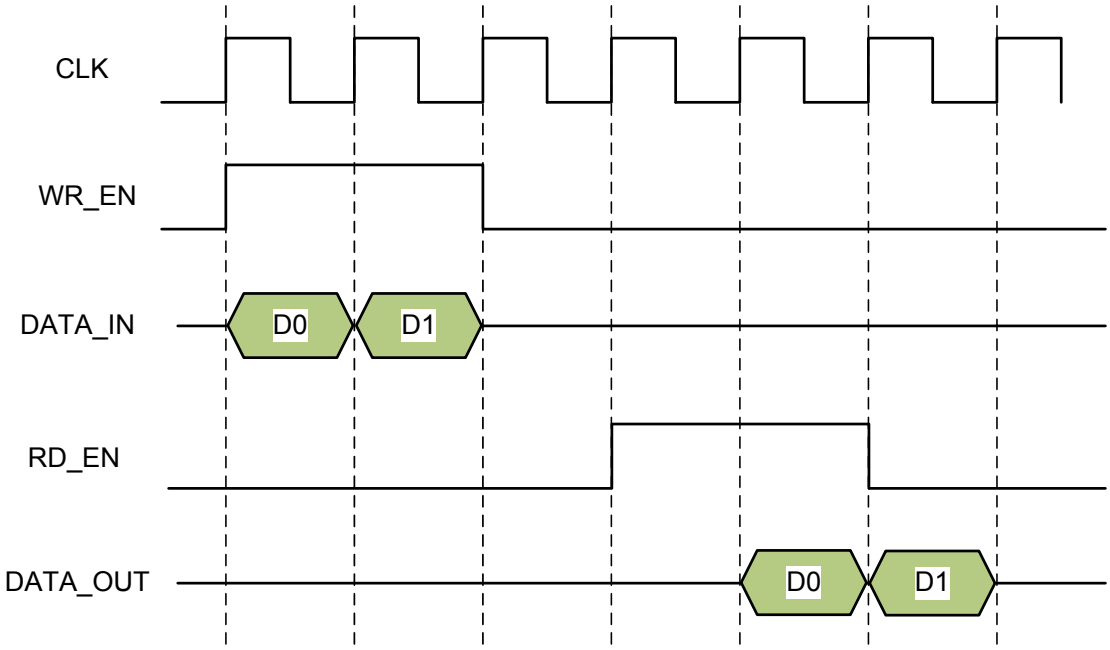

Figura 5.6: Operação de leitura e escrita de uma FIFO padrão.

A diferença do algoritmo *Round Robin* para o algoritmo *Deficit Round Robin*, consiste em que o segundo analisa primeiro se a fila tem os créditos necessários para poder transferir o quadro. Este requerimento é incompatível com o funcionamento da FIFO padrão. Uma solução é trocar a FIFO padrão por uma FIFO *fallthrough*. Este modo de operação é apresentado na Figura 5.7. Nesta FIFO, o primeiro dado aparece na saída imediatamente depois de ser armazenado. Esta característica permite visualizar o tamanho do quadro na fila sem retirar o valor do registrador interno. Nesta FIFO, o sinal de READ_EN significa

"Já li a palavra, agora quero ler a próxima". É importante lembrar que, de acordo com o formato dos quadros dentro da plataforma NetFPGA, o tamanho do quadro encontra-se na primeira palavra dos cabeçalhos do quadro.

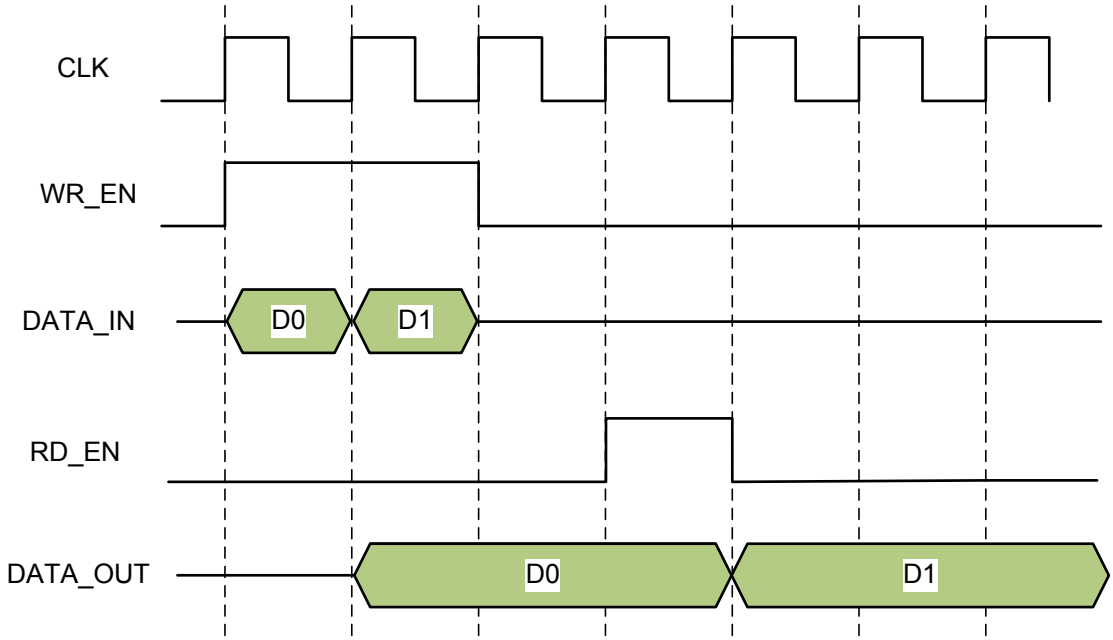

Figura 5.7: Operação de leitura e escrita de uma FIFO *fallthrough*.

Devido ao que foi exposto anteriormente, decidiu-se trocar as FIFOs padrão por FIFOs *fallthrough*. Estas FIFOs contêm sómente 4 posições, pois a maior retenção de dados encontra-se nas FIFOs que estão na saída dos blocos MAC (*Rx Queues*). Baseado no algoritmo DRR, foi implementada a máquina de estados (FSM) que controla este módulo. Esta FSM é apresentada na Figura 5.8. Esta máquina de estados substituiu a original do módulo contido na plataforma NetFPGA.

Esta FSM possui 3 estados. No estado *IDLE* a máquina revisa o estado das FIFOs de entrada. Se a FIFO estiver vazia, zera os créditos dessa entrada e muda para a entrada seguinte. Caso contrário, incrementa um *quantum* aos créditos da entrada e passa ao estado seguinte, *MEASURE_PKT*. Neste estado é verificado se a entrada tem os créditos suficientes para transmitir o quadro. Se o tamanho do quadro for maior que a quantidade de créditos, então muda-se para a proxima entrada e retorna ao estado *IDLE*. Caso contrário, se o tamanho do quadro for menor, atualiza-se os créditos subtraindo-se o valor do tamanho do quadro e inicia a transferência do quadro para a saída. A transferência do quadro é feita no estado *WR_PKT*.

O último bloco, ainda não descrito, gerencia a escrita e leitura dos registradores do módulo. Este bloco possui registradores para configurar o valor do *quantum* para cada entrada e registradores que armazenam o valor dos créditos de cada entrada para serem lidos por um agente externo (por exemplo, microprocessador ou CPU). O valor do *quantum* utilizado é o tamanho máximo de um quadro *Ethernet* padrão (1518 *bytes*). Na Tabela 5.4 são apresentados os registradores deste módulo. Todos os registradores são de 32 *bits*.

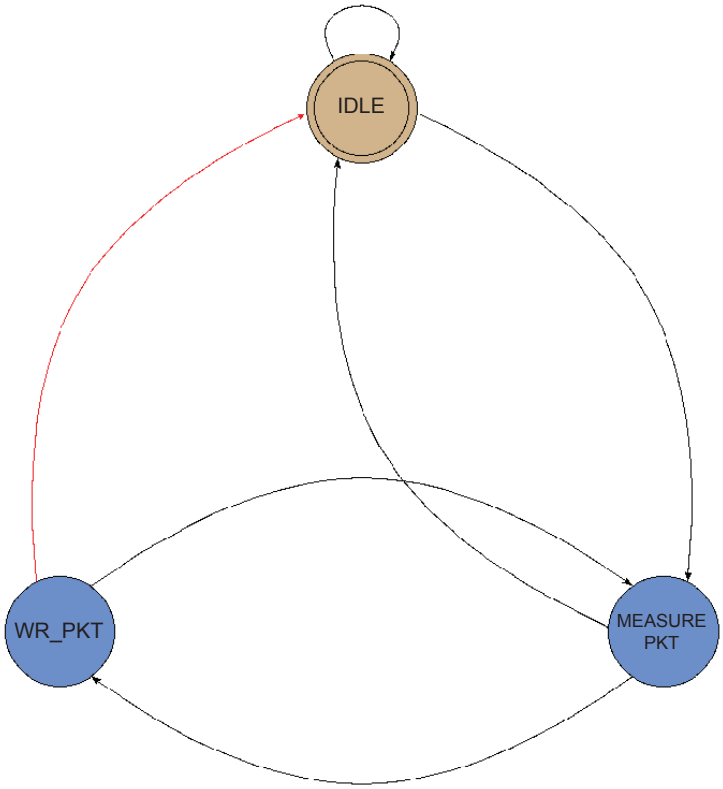

Figura 5.8: Máquina de estados do árbitro de entrada.

Tabela 5.4: Registradores do Árbitro de entrada.

Nome do Registrador	Descrição
IN_ARB_NUM_ PKTS_SENT_REG	Contém o número de quadros que têm sido atendidos pelo módulo.
IN_ARB_LAST_PKT_ WORD_0_LO_REG	Contém os 32 *bits* menos significativos da primeira palavra de 64 *bits* do último quadro que passou pelo módulo.
IN_ARB_LAST_PKT_ WORD_0_HI_REG	Contém os 32 *bits* mais significativos da primeira palavra de 64 *bits* do último quadro que passou pelo módulo.
IN_ARB_LAST_PKT_ CTRL_0_REG	Contém os primeiros 8 *bits* da palavra de controle do último quadro que passou pelo módulo.
IN_ARB_LAST_PKT_ WORD_1_LO_REG	Contém os 32 *bits* menos significativos da segunda palavra de 64 *bits* do último quadro que passou pelo módulo.
IN_ARB_LAST_PKT_ WORD_1_HI_REG	Contém os 32 *bits* mais significativos da segunda palavra de 64 *bits* do último quadro que passou pelo módulo.
IN_ARB_LAST_PKT_ CTRL_1_REG	Contém os segundos 8 *bits* da palavra de controle do último quadro que passou pelo módulo.
IN_ARB_STATE_REG	Contém o valor do estado da máquina de estados.
IN_ARB_FIFO_ CREDITS_X	Contém o valor atual dos créditos da porta X. X é o número da porta.
IN_ARB_FIFO_ QUANTUM_X	Configura o valor do *quantum* da porta X. X é o número da porta.

5.2 Pesquisador da porta de saída

Este módulo está encarregado de encontrar a porta ou as portas de saída para cada quadro que ingressa ao comutador. Para realizar esta tarefa, baseia-se principalmente em um motor de classificação que neste trabalho utiliza os dados da camada 2 (L2) ou de enlace segundo o modelo OSI. Além do motor de classificação de nível 2, este módulo possui dois blocos que processam quadros que tem identificadores VLAN ou VLAN IDs.

Na Figura 5.9 é apresentado o diagrama de blocos deste módulo. Os blocos Extrator de VLAN e Adicionador de VLAN ajudam a reduzir a complexidade do analisador de cabeçalho do motor de classificação. Estes blocos primeiro extraem o identificador VLAN e colocam este dado como um cabeçalho do formato de quadro do NetFPGA (bloco Extrator de VLAN) e depois inserem novamente o identificador dentro do quadro *Ethernet* em seu campo respectivo (bloco Adicionador de VLAN). Então o motor de classificação recebe todos os quadros como se fossem quadros *Ethernet* padrão.

Como foi mencionado no Capítulo 2, um quadro *Ethernet* com VLAN desloca o campo de *LENGTH/TYPE* para inserir a informação VLAN. Portanto, dessa forma, o analisador de cabeçalho do motor de classificação só precisa examinar as duas primeiras palavras do quadro para extrair os dados que precisa, já que a informação VLAN vem dentro de um cabeçalho proprietário. Cada palavra do quadro com o formato NetFPGA possui 8 *bytes* (64 *bits*), portanto nas duas primeiras palavras do quadro está a informação dos endereços de origem e destino MAC e o campo LENGTH/TYPE.

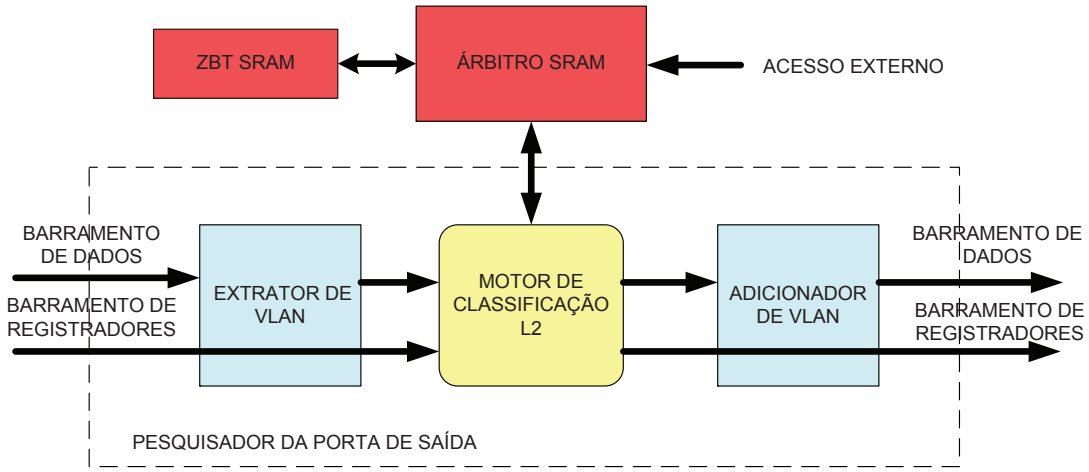

Figura 5.9: Diagrama de blocos do módulo Pesquisador da porta de saída.

Como foi mencionado anteriormente, a estrutura proposta na plataforma NetFPGA permite adicionar módulos utilizando uma interface padrão. A Tabela 5.5 mostra os sinais além da interface padrão. Estes sinais são da comunicação com o árbitro da memória SRAM. Os sinais do relógio e *reset* estão excluídos.

Tabela 5.5: Interface do módulo pesquisador da porta de saída.

Sinal	Tipo	Descrição
Interface com o árbitro SRAM para o bloco de encaminhamento e aprendizagem		
wr_0_req	Saída	Requisição de escrita na memória SRAM.
wr_0_addr [18:0]	Saída	Barramento de endereços para a escrita na memória SRAM.
wr_0_data [71:0]	Saída	Barramento de dados para a escrita na memória SRAM.
wr_0_ack	Entrada	Indica que se a requisição de escrita foi atendida.
rd_0_req	Saída	Requisição de leitura na memória SRAM.
rd_0_addr	Saída	Barramento de endereços para a leitura na memória SRAM.
rd_0_data	Entrada	Barramento de dados para a leitura na memória SRAM.
rd_0_ack	Entrada	Indica que a requisição de leitura foi atendida.
rd_0_vld	Entrada	Indica que os dados em rd_0_data são válidos.
Interface com o árbitro SRAM para o bloco de envelhecimento		
wr_1_req	Saída	Requisição de escrita na memória SRAM.
wr_1_addr [18:0]	Saída	Barramento de endereços para a escrita na memória SRAM.
wr_1_data [71:0]	Saída	Barramento de dados para a escrita na memória SRAM.
wr_1_ack	Entrada	Indica que a requisição de escrita foi atendida.
rd_1_req	Saída	Requisição de leitura na memória SRAM.
rd_1_addr	Saída	Barramento de endereços para a leitura na memória SRAM.
rd_1_data	Entrada	Barramento de dados para a leitura na memória SRAM.
rd_1_ack	Entrada	Indica que se atendeu a requisição de leitura.
rd_1_vld	Entrada	Indica que os dados em rd_1_data são válidos.

A seguir estes três módulos são descritos, salientando-se que para o motor de classificação tem-se duas propostas.

5.2.1 Blocos Extrator e Adicionador de VLAN

Estes dois blocos complementam-se para extrair e adicionar a informação VLAN dos quadros. Estes blocos estão inclusos na plataforma NetFPGA mas foram modificados para a correção de alguns problemas encontrados durante a etapa de verificação funcional. Algumas modificações visam reduzir a latência dos blocos, mudando-se as máquinas de estado. Durante a verificação funcional foi encontrado um problema, quando estes blocos processavam quadros com tamanhos menores ao mínimo (64 *bytes*). Estes quadros poderiam aparecer dentro do comutador, já que o *padding* é removido de todos os quadros nos blocos GEMAC nas entradas do comutador. A seguir são descritos brevemente estes dois blocos.

5.2.1.1 Bloco Extrator de VLAN

Este bloco encarrega-se de extrair o identificador VLAN do quadro e colocá-lo em um cabeçalho adicional. A Figura 5.10 mostra o respectivo diagrama de blocos. Possui uma FIFO *fallthrough* para reter os dados na entrada e uma FIFO padrão para reter a informação VLAN. Possui duas máquinas de estado, uma que encarrega-se de procurar

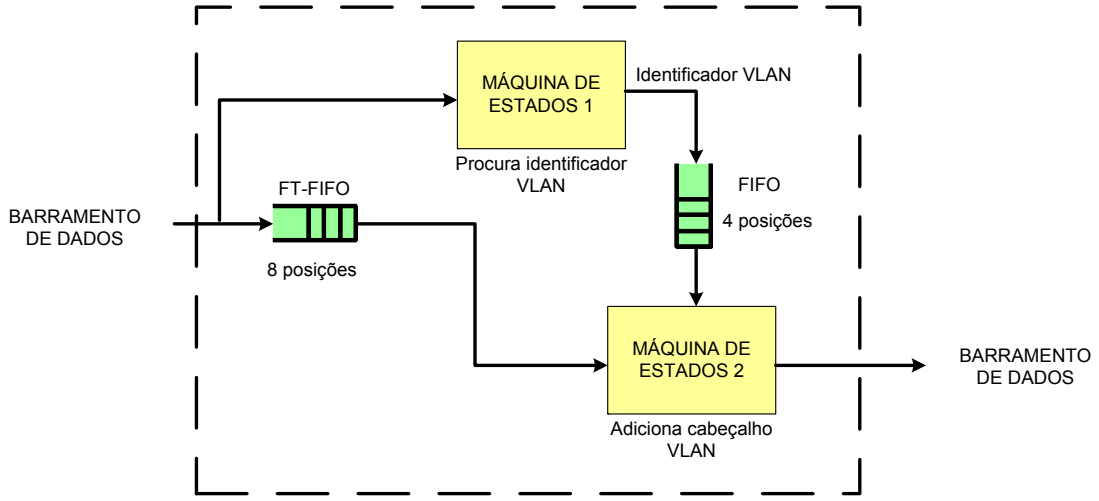

Figura 5.10: Diagrama de blocos do Extrator de VLAN.

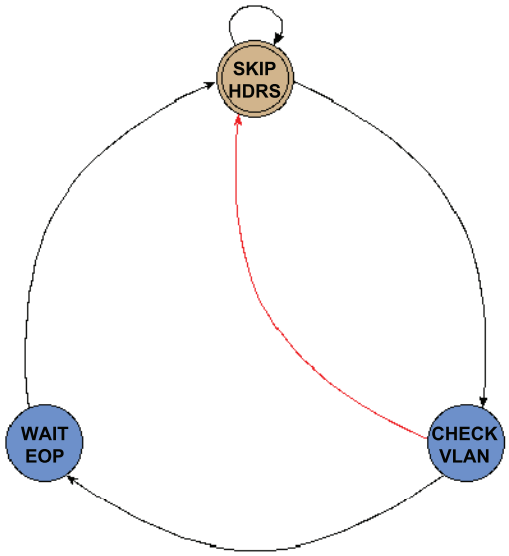

Figura 5.11: Diagrama de estados da máquina que procura pela informação VLAN do bloco Extrator de VLAN.

pelo identificador de protocolo VLAN (0x8100) e extrair o identificador VLAN e a outra máquina de estado que trata de adicionar o cabeçalho com a palavra de controle **0x42**.

Na Figura 5.11 é apresentado o diagrama de estados da primeira máquina. Esta máquina possui 3 estados. O estado inicial é o SKIP_HDRS. Neste estado, a máquina de estados espera que passem os outros cabeçalhos que podem conter os quadros e muda para o estado CHECK_VLAN quando a palavra de controle do quadro é **0x00**. No estado CHECK_VLAN, a máquina extrai o identificador VLAN quando o identificador de protocolo VLAN é encontrado. Por último, no estado WAIT_EOP é esperado o fim do quadro.

Na Figura 5.12 é mostrado o diagrama de estados da segunda máquina. Esta máquina possui 8 estados. A sequência de estados quando não é encontrado o identificador VLAN é a seguinte: WAIT PREPROCESS -> READ FIFO -> SEND UNMODIFIED PKT -> WAIT PREPROCESS. Quando o identificador VLAN é encontrado a sequência é: WAIT PREPROCESS -> READ FIFO -> ADD MODULE HEADER -> WRITE MODULE

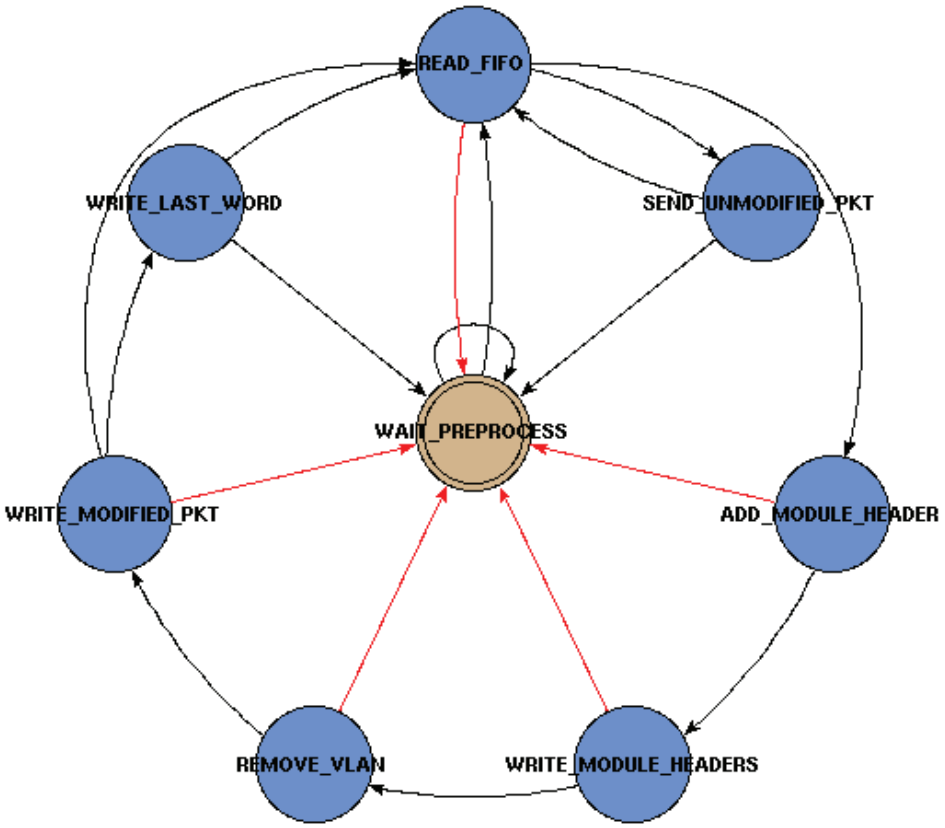

Figura 5.12: Diagrama de estados da máquina que adiciona o cabeçalho VLAN do bloco Extrator de VLAN.

HEADERS -> REMOVE VLAN -> WRITE MODIFIED PKT -> WRITE LAST WORD -> WAIT PREPROCESS.

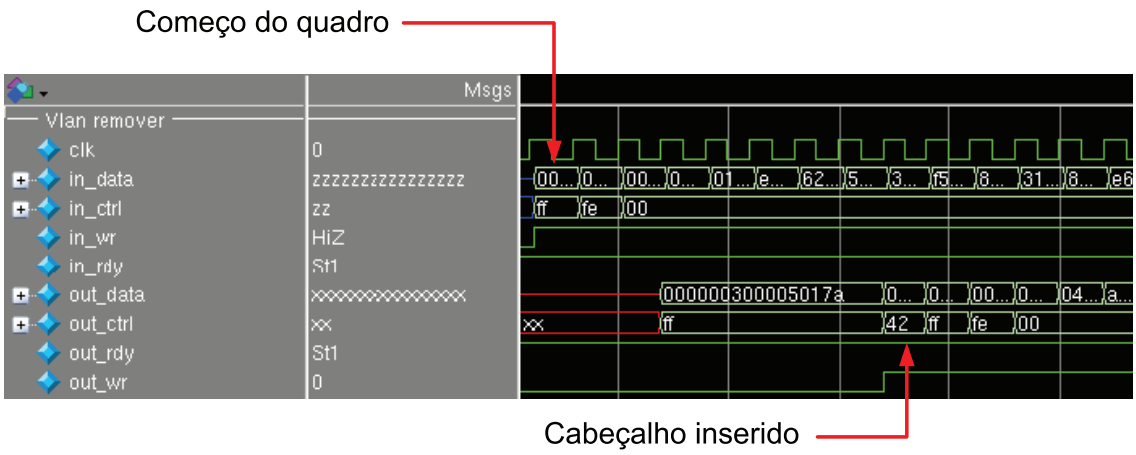

Figura 5.13: Cabeçalho VLAN inserido no quadro na saída do bloco Extrator de VLAN.

Na Figura 5.13 é mostrado um quadro com o cabeçalho VLAN (**0x42**) adicionado pelo bloco. Na interface de entrada do bloco (IN_DATA e IN_CTRL) é mostrado um quadro com identificador VLAN inserido no quadro *Ethernet*. Na interface de saída do bloco (OUT_DATA e OUT_CTRL) é mostrado o mesmo quadro com o cabeçalho VLAN inserido no começo do quadro.

5.2.1.2 *Bloco Adicionador de VLAN*

Este bloco é muito similar ao bloco Extrator. O diagrama de blocos do adicionador de VLAN pode ser observado na Figura 5.14. Possui uma FIFO *fallthrough* para reter os dados na entrada e duas máquinas de estado para procurar pelo cabeçalho VLAN e logo depois adicionar o identificador VLAN no quadro.

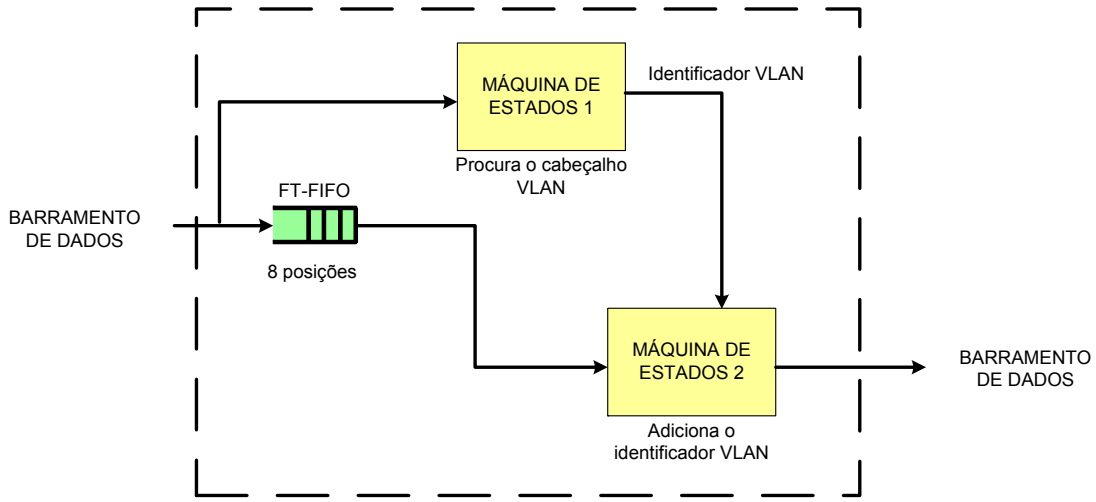

Figura 5.14: Diagrama de blocos do Adicionador de VLAN.

O diagrama de estado da primeira máquina apresenta-se na Figura 5.15. Esta máquina procura pelo cabeçalho VLAN (**0x42**). O estado inicial é o FIND_VLAN_HDR. Neste estado a máquina procura pelo cabeçalho VLAN e extrai o identificador VLAN. No estado WAIT_EOP a máquina espera pelo fim do quadro.

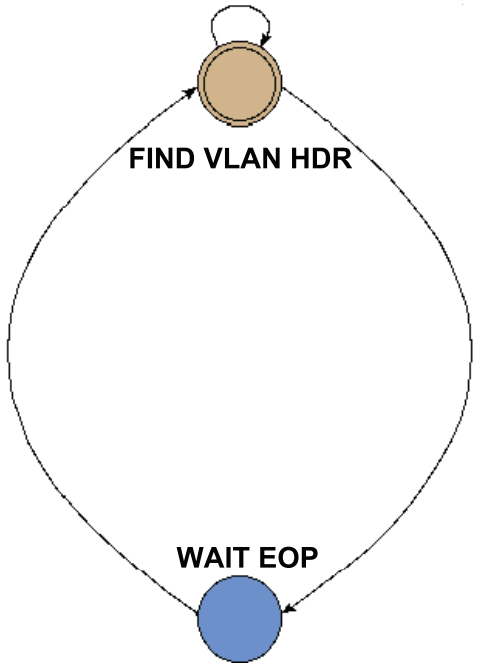

Figura 5.15: Diagrama de estados da máquina que procura pelo cabeçalho VLAN do bloco Adicionador de VLAN.

Na Figura 5.16 é mostrada a segunda máquina de estados. Esta máquina adiciona o

identificador VLAN dentro do quadro. A sequência de estados quando um quadro sem cabeçalho VLAN ingressa é: WAIT PREPROCESS -> SEND UNMODIFIED PKT -> WAIT PREPROCESS. Quando o cabeçalho é encontrado a sequência é: WAIT PREPROCESS -> REMOVE MODULE HEADER -> WAIT SOP -> ADD VLAN -> WRITE MODIFIED PKT -> WAIT PREPROCESS.

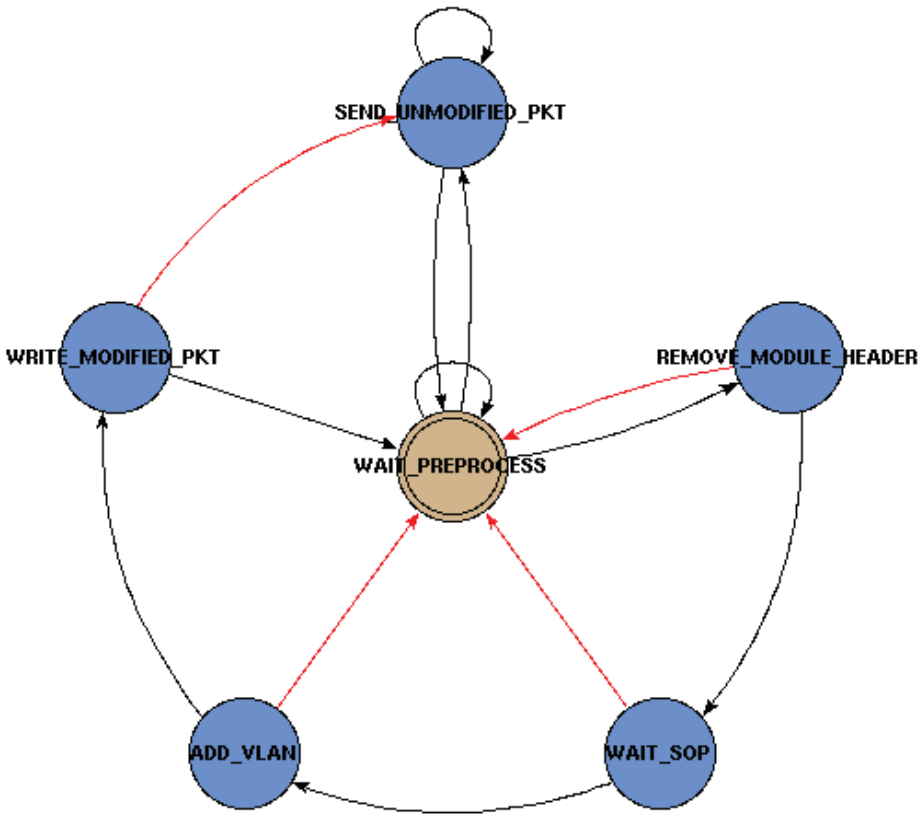

Figura 5.16: Diagrama de estados da máquina que adiciona o identificador VLAN no quadro do bloco Adicionador de VLAN.

5.2.2 Motor de Classificação de Nível 2

5.2.2.1 Trabalhos relacionados

Na literatura podemos encontrar diferentes abordagens ao problema de classificação de quadros. Algumas soluções utilizam memórias de conteúdo endereçável ou CAMs binárias ou ternárias, mas o custo por *bit* e o consumo de potência apresentado em Mcauley e Francis (1993), faz inviável seu uso em comutadores com tabelas de endereços MAC com tamanhos que estão em torno das centenas de milhares de entradas. Outras propostas tentaram implementar o motor de classificação em *software*, mas os testes feitos em Luo et al. (2007) confirmam que esta tarefa deve ser implementada em *hardware*. Uma solução muito empregada é utilizar uma função *hash* para armazenar os endereços MAC numa memória SRAM. Na Figura 5.17 apresenta-se o esquema de uma função *hash*. Utiliza um *hardware* mais simples em comparação às outras soluções, mas possui algumas desvantagens como a redução da capacidade da memória devido às colisões de endereços. Para lidar com o problema das colisões, a tabela de endereços está organizada em *buckets* que contém mais de uma entrada. Uma função *hash* escolhida com uma distribuição relativamente uniforme de valores de saída reduzirá as colisões e melhorará a capacidade da tabela por consequencia.

A seguir apresentam-se as duas soluções propostas para o projeto do motor de classificação. Ambas propostas possuem a mesma interface mostrada na Tabela 5.5. A principal diferença encontra-se no projeto dos blocos que encarregam-se do encaminhamento e aprendizagem dos quadros e no árbitro SRAM. Esta diferença faz com que o desempenho mude significativamente. Os resultados de cada uma das propostas são expostos no Capítulo 7.

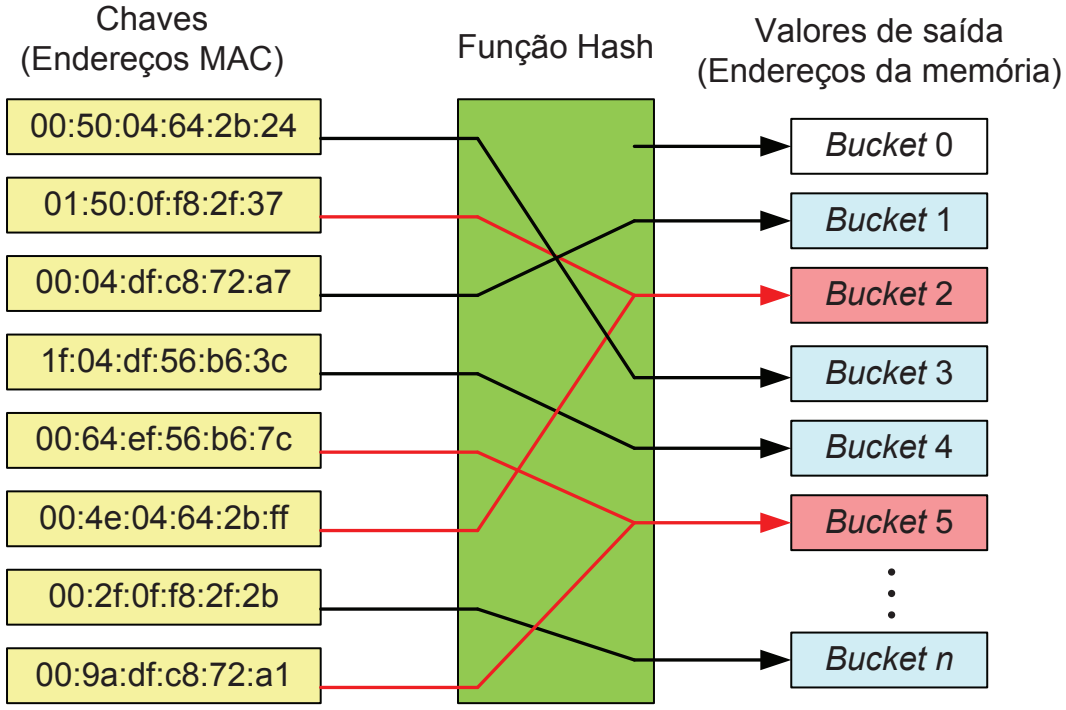

Figura 5.17: Função de *Hash*.

5.2.2.2 *Primeira Proposta (TONFAT J.; NEUBERGER; REIS, 2010)*

A primeira arquitetura foi feita baseada nos blocos inclusos na plataforma NetFPGA. Este diagrama de blocos é mostrada na Figura 5.18.

Este bloco cumpre as seguintes características definidas no padrão IEEE 802.1D (IEEE Std 802.1D, 2004):

- Encaminhamento de quadros *unicast*, *multicast* e *broadcast*;
- Tabela de endereços MAC para o encaminhamento de quadros;
- Escritura estática de endereços MAC;
- Aprendizagem dinâmica de endereços MAC;
- Envelhecimento dos endereços dinamicamente apreendidos com tempo configurável.

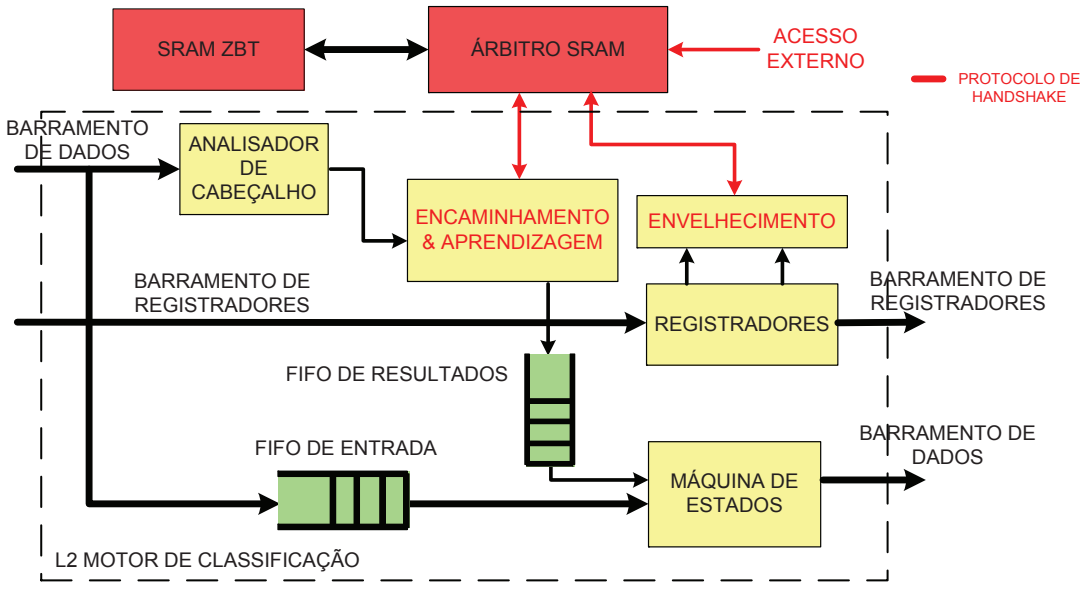

Figura 5.18: Diagrama de blocos da arquitetura 1.

A seguir cada um dos blocos integrantes do motor de classificação são descritos.

5.2.2.2.1 Analisador de cabeçalho

Este bloco encarrega-se de extrair quatro dados do quadro: o endereço MAC de destino, o endereço MAC de origem, a porta de entrada e o VLAN ID, quando existir. Para realizar esta função, este bloco possui uma máquina de estados, que é mostrada na Figura 5.19.

5.2.2.2.2 FIFO de entrada

Este bloco encarrega-se de armazenar as primeiras palavras de cada quadro e permitir que o bloco que analisa o cabeçalho consiga extrair os campos importantes. As características desta FIFO são as seguintes:

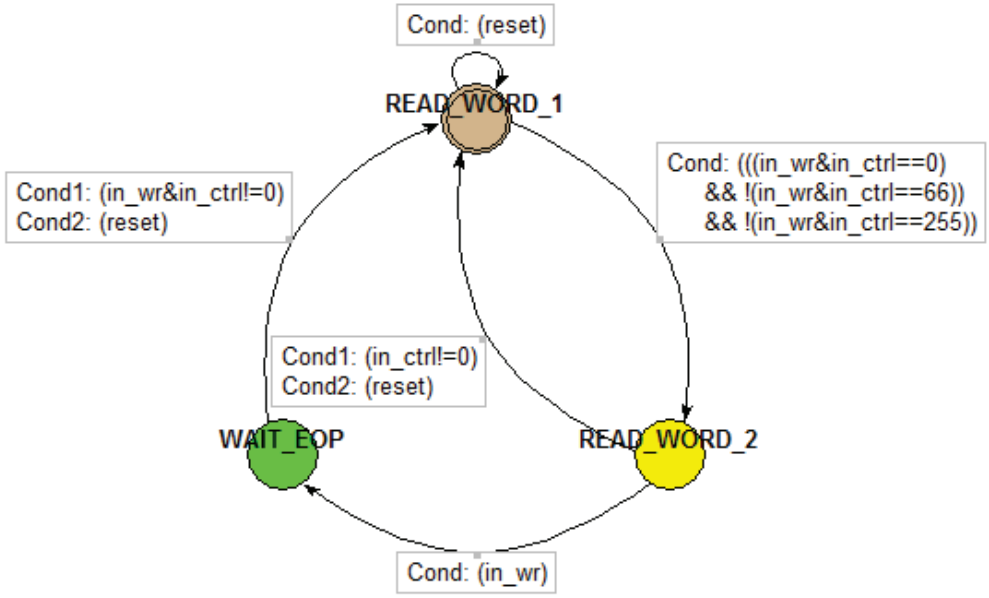

Figura 5.19: Diagrama de estados da FSM do bloco analisador de cabeçalho.

- Tamanho da palavra: 72 *bits*;
- Profundidade: 16 posições;
- FIFO síncrona;
- Desempenho de leitura: padrão.

O desempenho de leitura padrão se refere ao modo de ler os valores da FIFO.

5.2.2.2.3 FIFO de resultados

Este bloco armazena o resultado do bloco de encaminhamento e aprendizagem. Este resultado consiste das portas de destino e, quando o quadro contiver informação de VLAN, também será incluso a informação sobre quais serão as portas *tagged* daquela VLAN. As características desta FIFO são as seguintes:

- Tamanho da palavra: 16 *bits*;
- Profundidade: 4 posições;
- FIFO síncrona;
- Desempenho de leitura: padrão.

5.2.2.2.4 Encaminhamento e aprendizagem

Este bloco é o principal do motor de classificação, portanto o mais complexo. A função principal é encontrar as portas de saída de cada quadro. Dependendo das características do quadro, este poderia ter mais de um destino (ex. quadros *multicast* e *broadcast*). Para cumprir esta função o bloco possui uma máquina de estados que é mostrada na Figura 5.20.

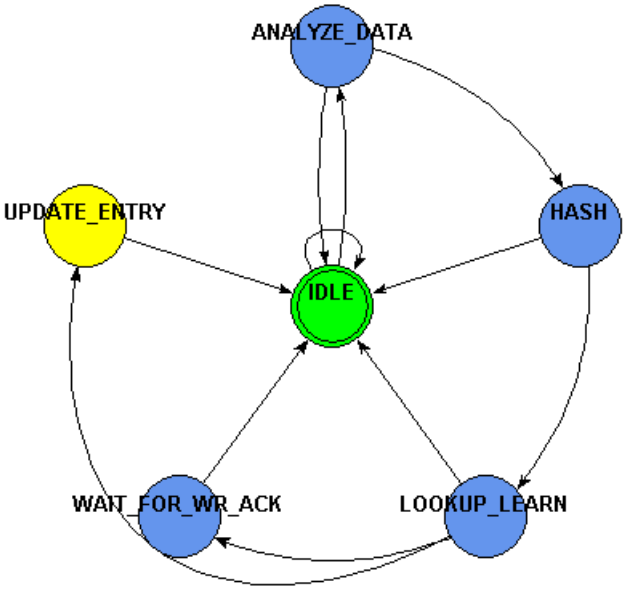

Figura 5.20: Diagrama de estados da FSM do bloco de encaminhamento e aprendizagem.

No primeiro estado (IDLE), o bloco espera a chegada de dados de um quadro. No segundo estado (ANALIZE_DATA), o bloco vai analisar se o endereço de destino é *unicast*, *multicast* ou *broadcast*. Se o quadro for *multicast* ou *broadcast*, as portas de saída já encontram-se definidas, então não é necessário procurar o endereço de destino na tabela. Neste estado também é analisado se o quadro possui um endereço de origem inválido (*multicast* ou *broadcast*). Quando isto acontecer, estes quadros serão descartados.

No estado (HASH) calculam-se os endereços na tabela SRAM dos endereços MAC do quadro utilizando a função *Hash* apresentada anteriormente e, também, se o quadro possuir o VLAN ID, é procurado na tabela SRAM quais são as portas membros *Tag* e *Untag* daquele ID. Os membros *Tag* são aqueles que devem passar o VLAN *Tag* ao seguinte elemento na rede, e os membros *Untag* são aqueles que pertencem a um determinado grupo VLAN, mas não passam o quadro com o VLAN *Tag*.

No estado (LOOKUP_LEARN), tem-se duas opções: se o quadro tiver um endereço MAC de destino do tipo *multicast* ou *broadcast*, então, neste estado, só é efetuada a operação de aprendizagem. No caso contrário, primeiro é efetuada a operação de pesquisa e depois a operação de aprendizagem. A operação de pesquisa compreende a procura do endereço MAC de destino na tabela SRAM. Paralelamente, a operação de aprendizagem abrange a procura do endereço MAC de origem na tabela e depois a escrita do mesmo, caso o endereço não seja encontrado. As operações de pesquisa e aprendizagem são feitas juntas para aproveitar a redução do tempo de acesso na SRAM. Mais informação é apresentada na descrição do Árbitro SRAM.

No estado (UPDATE_ENTRY), aguarda-se pela confirmação da escrita do endereço

MAC de origem e também pela resposta da operação de leitura na tabela do endereço MAC de destino (operação de pesquisa).

Por último, no estado (WAIT_FOR_WR_ACK) só aguarda-se pela confirmação da escrita do endereço MAC de origem.

Para armazenar os endereços MAC na SRAM, utiliza-se uma função *Hash* definida pelo seguinte polinômio:

$$x^{16} + x^{12} + x^5 + 1$$

Este polinômio também é conhecido como CRC-CCITT e também é utilizado por protocolos como X.25 ou *Bluetooth*. Foi escolhido devido aos resultados apresentados em Jain (1992) que demonstram a efetividade deste polinômio na tarefa de pesquisa de endereços MAC. A tabela SRAM está segmentada conforme mostra a Figura 5.21.

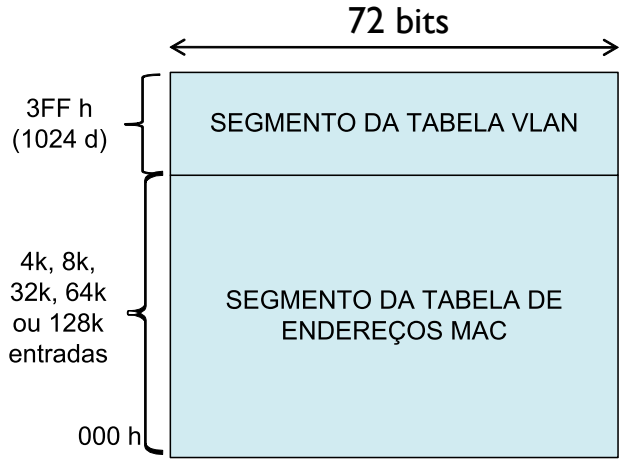

Figura 5.21: Organização da memória SRAM.

Os primeiros endereços da memória são reservados para armazenar a tabela de endereços MAC. O formato utilizado para cada um dos endereços armazenados é apresentado na Tabela 5.6.

Tabela 5.6: Formato de cada entrada da tabela de endereços MAC.

Bits	Campo	Descrição
71:60	VLAN ID	12-*bit* VLAN ID associado ao endereço MAC.
59:12	MAC ADDR	48-*bit* endereço MAC.
11:4	Port ID	8-*bit* Porta associada ao endereço MAC utilizando codificação *one-hot*.
3	Reservado	Não utilizado.
2	*Valid bit*	1: entrada válida, 0: entrada vazia.
1	*Static bit*	1: entrada estática, 0: entrada dinamicamente aprendida.
0	*Age bit*	1: entrada atualizada desde o último processo de envelhecimento, 0: entrada não atualizada desde o último processo de envelhecimento.

O formato utilizado para armazenar a informação de cada VLAN é mostrado na Figura 5.22. Cada VLAN possui 16 *bits* que estão divididos em dois campos: um com a

informação das portas membros e o outro campo tem a informação das portas membros *tagged*. Um membro *tagged* além de encaminhar os quadros dos outros membros tem que inserir o VLAN *tag* em cada quadro. Esta operação deve ser feita no bloco MAC de cada porta.

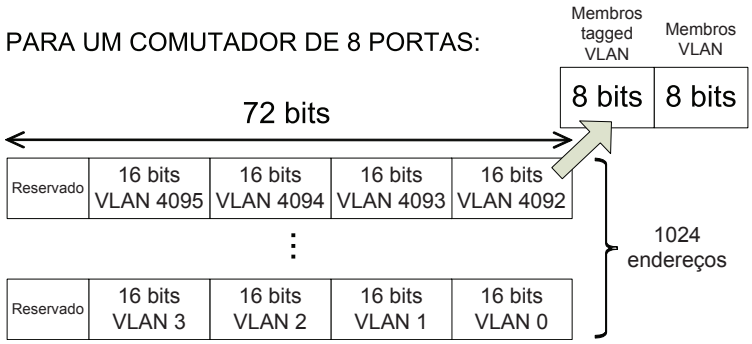

Figura 5.22: Organização do segmento da tabela VLAN na memória SRAM.

5.2.2.2.5 Envelhecimento

Este bloco realiza a operação de envelhecimento da tabela de endereços MAC. Envelhecimento da tabela refere-se a apagar os endereços MAC armazenados na tabela que não são utilizados por um determinado intervalo de tempo. Este intervalo de tempo, para executar o processo de envelhecimento, é programável através de um registrador de configuração. Se o intervalo de tempo não é alterado, o valor padrão é 300 ms. Também é possível configurar a quantidade de entradas lidas numa só operação de leitura (*Read Burst*). A leitura em rajada diminui o tempo de execução da operação de envelhecimento, pois a operação de leitura na tabela tem uma latência de menos de seis ciclos de relógio, mas também poderia atrapalhar a operação de pesquisa na tabela. O bloco vai pesquisar cada uma das entradas da tabela de endereços MAC e procurará nas entradas válidas (utilizando o *valid bit* da entrada) se a entrada foi utilizada desde a última operação de envelhecimento (utilizando o *age bit* da entrada). Quando a entrada não for visitada, esta é marcada como inválida apagando o *valid bit*. O bloco possui uma máquina de estados que é mostrada na Figura 5.23.

5.2.2.2.6 Árbitro SRAM

O bloco árbitro SRAM encarrega-se de encaminhar as solicitações de acesso (leitura e escrita) na memória SRAM, onde encontram-se armazenadas as tabelas de endereços MAC e VLAN. Neste caso, a memória SRAM possui três fontes de acesso: o bloco de encaminhamento e aprendizagem; o bloco de envelhecimento e o acesso externo através de registradores. Este bloco estava incluso na plataforma NetFPGA, mas foi modificado para aceitar três fontes de acesso independentes.

Nas Figuras 5.24 e 5.25 são apresentados os diagramas de tempos das operações de escrita e leitura na SRAM feita pelo árbitro. Estes diagramas mostram que para uma operação de escrita precisa-se de no mínimo três ciclos e para uma operação de leitura seis ciclos. Com estas características, as operações de leitura e escrita em rajada favorecem a redução do tempo de acesso.

O árbitro utiliza o algoritmo WRR (*Weighted Round Robin*) para servir às fontes de acesso. O bloco de encaminhamento e aprendizagem tem a prioridade com respeito aos

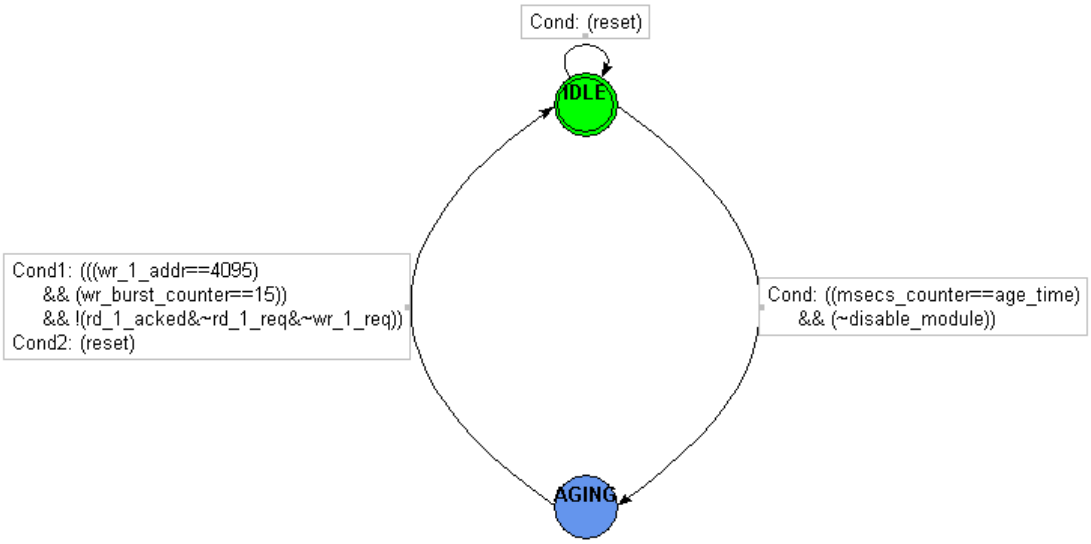

Figura 5.23: Diagrama de estados da FSM do bloco de envelhecimento.

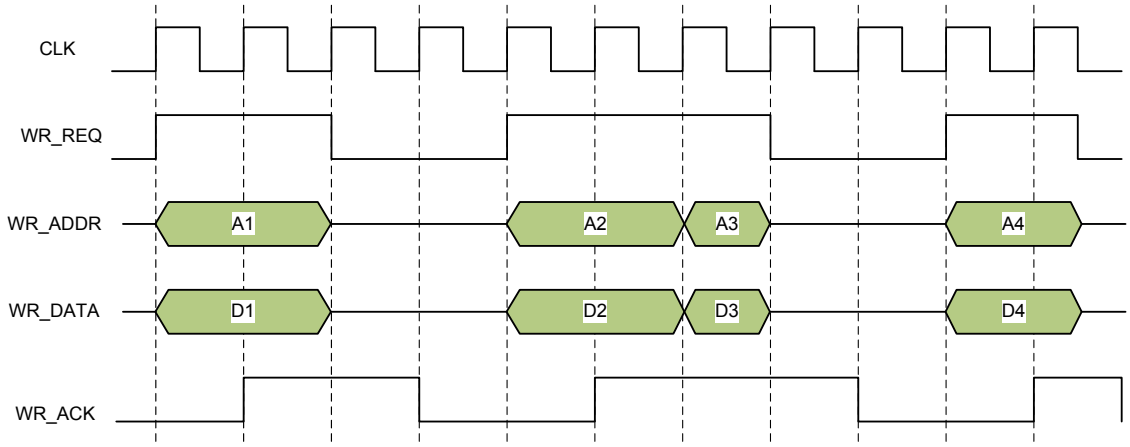

Figura 5.24: Diagrama de tempos de uma operação de escrita no árbitro SRAM.

outros dois porque este cumpre a principal função do comutador. Para implementar o algoritmo, o árbitro possui duas máquinas de estado finito.

A primeira máquina de estados é mostrada na Figura 5.26 e encarrega-se de receber as requisições de acesso e reservar a utilização da SRAM por uma quantidade de ciclos. Deste modo, se define a prioridade às fontes.

A segunda máquina de estados que é apresentada na Figura 5.27 encarrega-se de encaminhar os sinais de cada fonte de acesso à memória SRAM. Também encarrega-se de selecionar a fonte seguinte que será atendida, dependendo do estado da requisição seguinte. A nomenclatura dos nomes dos estados é a seguinte: RD para uma operação de leitura e WR para uma operação de escrita. Os números que aparecem ao lado identificam a fonte de acesso. A fonte número 0 é o bloco de encaminhamento e aprendizagem. A fonte número 1 é o bloco de envelhecimento e o bloco número 2 é o acesso externo através de registradores. No diagrama observa-se que de cada um dos estados pode-se mudar para qualquer outro, pois não existe nenhuma restrição com respeito à ordem das requisições.

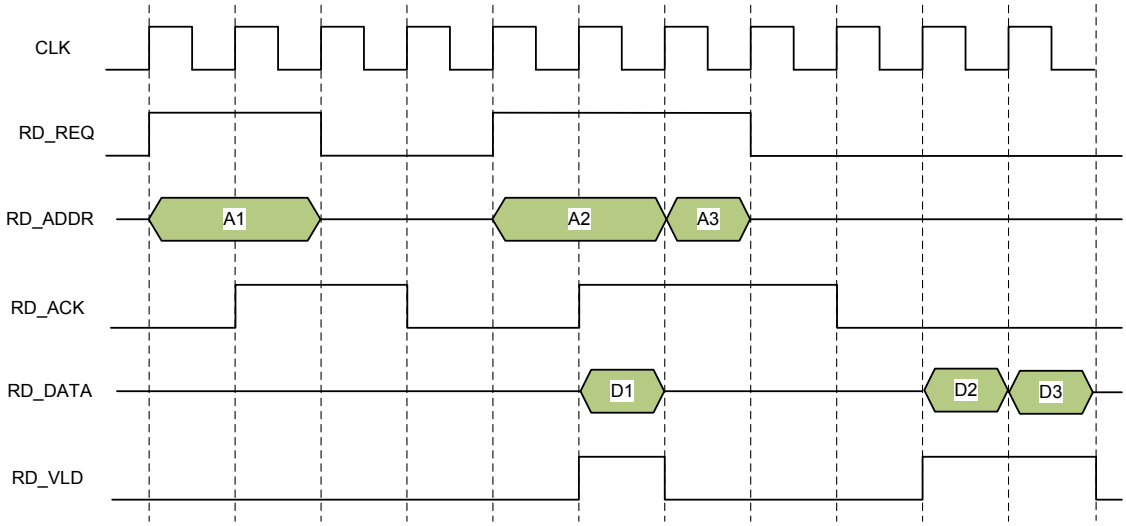

Figura 5.25: Diagrama de tempos de uma operação de leitura no árbitro SRAM.

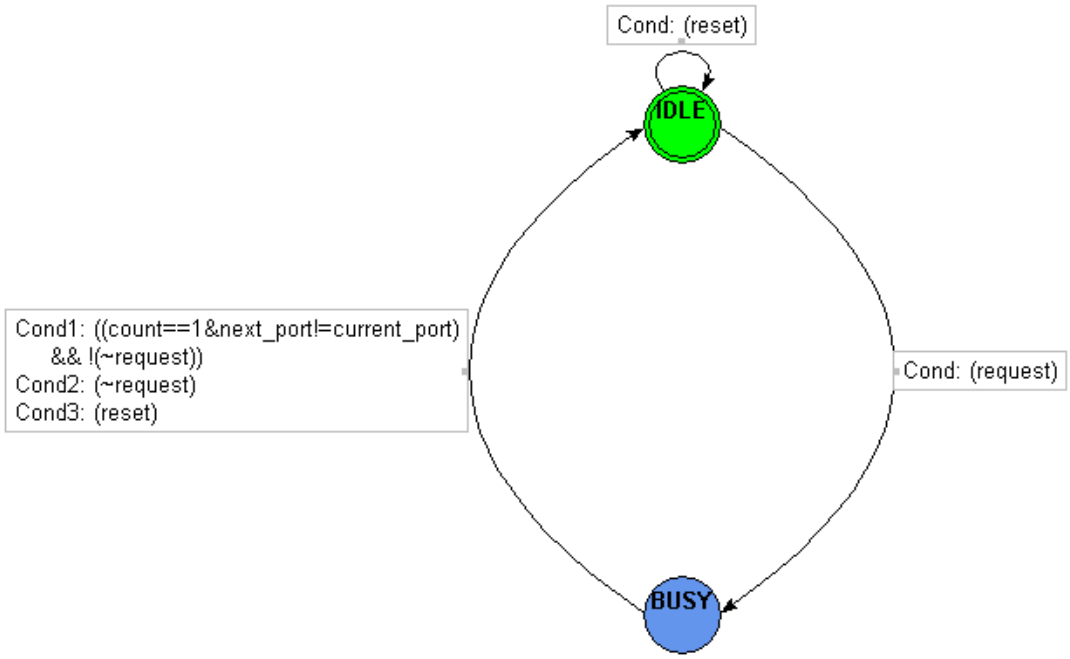

Figura 5.26: Diagrama de estados da primeira FSM do árbitro SRAM.

5.2.2.2.7 Bloco de Registradores

Este bloco contém a lógica que implementa o protocolo de comunicação com o barramento de registradores do comutador e também possui os registradores de configuração do motor de classificação assim como também os registradores de estado do bloco. A descrição dos registradores é mostrada na Tabela 5.7.

5.2.2.2.8 Máquina de estados do motor de classificação

O motor de classificação possui uma máquina de estados finitos geral para controlar o fluxo dos quadros no bloco. O diagrama de estados é apresentado na Figura 5.28. No

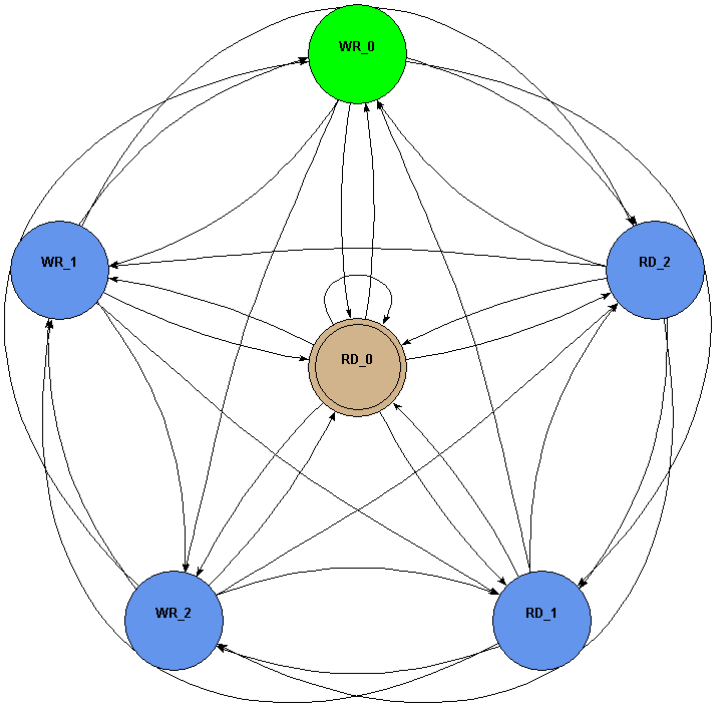

Figura 5.27: Diagrama de estados da segunda FSM do árbitro SRAM.

estado inicial (WAIT_TILL_DONE_DECODE), espera-se pela conclusão do processo de pesquisa. O indicador deste evento é o marcador de vazio da FIFO de resultados. No estado seguinte (WRITE_HDR), escrevem-se os resultados da pesquisa (a porta de saída e as portas *tagged* no caso de quadros VLAN) no cabeçalho do quadro. O estado seguinte (SKIP_HDRS) vai encaminhar os outros cabeçalhos no quadro para a saída. No último estado (WAIT_EOP), espera-se pelo indicador de fim de quadro no *byte* de controle para assegurar que todo o quadro foi encaminhado ao bloco seguinte na parte operativa.

5.2.2.3 Segunda Proposta (TONFAT; REIS, 2010)

A segunda proposta possui a mesma arquitetura que a proposta anterior, porém as mudanças estão na arquitetura interna dos blocos de encaminhamento e aprendizagem e no árbitro SRAM. A Figura 5.29 apresenta o digrama de blocos. A principal diferença está na comunicação entre o bloco de encaminhamento e aprendizagem com o árbitro SRAM. Esta comunicação não é mais do tipo "*handshake*" e vai passar a ser uma comunicação sincronizada com o árbitro SRAM. Para conseguir isto, os dois blocos devem ter suas máquinas de estado sincronizadas.

A seguir são apresentados somente os blocos que mudaram com respeito à primeira arquitetura.

5.2.2.3.1 Árbitro SRAM

Este bloco cumpre as mesmas funções do bloco na proposta 1 na subseção 5.2.2.2. A mudança mais significativa está no protocolo de comunicação com o bloco de encaminhamento e aprendizagem. Na proposta anterior todas as fontes de acesso utilizavam o protocolo de "*handshake*". Agora, só os blocos de envelhecimento e o acesso externo possuem este protocolo. Para eliminar o protocolo de "*handshake*", as máquinas de estado

Tabela 5.7: Registradores do motor de classificação.

Nome do Registrador	Descrição
LUT_HIT	Registrador de estado que indica a quantidade de acertos que teve o motor de classificação na pesquisa de endereços MAC na tabela.
LUT_MISS	Registrador de estado que indica a quantidade de erros que teve o motor de classificação na pesquisa de endereços MAC na tabela.
COLLISION_COUNTER	Registrador de estado que indica a quantidade de colisões que teve o motor de classificação na aprendizagem de endereços MAC na tabela.
TIME_TO_AGE	Registrador de configuração do tempo de envelhecimento da tabela de endereços MAC. Este valor está em milissegundos (ms).
AGE_CHANGE	Registrador de configuração que indica se o tempo de envelhecimento vai ser diferente do tempo padrão (300 ms).

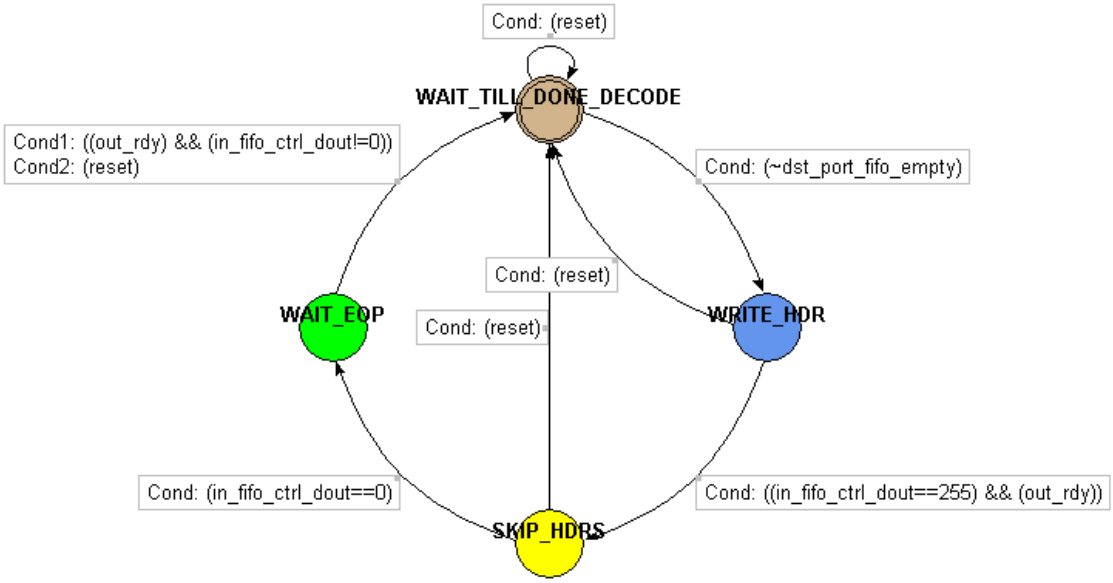

Figura 5.28: Diagrama de estados da FSM do motor de classificação.

do árbitro SRAM e do bloco de encaminhamento e aprendizagem estão sincronizadas.

Assim, quando o bloco de encaminhamento e aprendizagem faz uma requisição de leitura, o árbitro SRAM está pronto para receber a requisição. O mesmo acontece nas operações de escrita. Na Tabela 5.8 são apresentadas as operações feitas em cada um dos ciclos da FSM. A máquina de estados possui 16 estados, 1 ciclo por estado. Os ciclos de cor azul estão reservados para processar leituras ou escritas das fontes de acesso externo e do envelhecimento. Os outros 14 ciclos restantes, são utilizados pelo bloco de encaminhamento e aprendizagem.

Outra mudança significativa está na capacidade de processamento das requisições de acesso. Na proposta anterior, é atendida a uma fonte por vez e só é processada outra requisição quando a operação (de leitura ou escrita) era terminada. Nesta proposta, o ár-

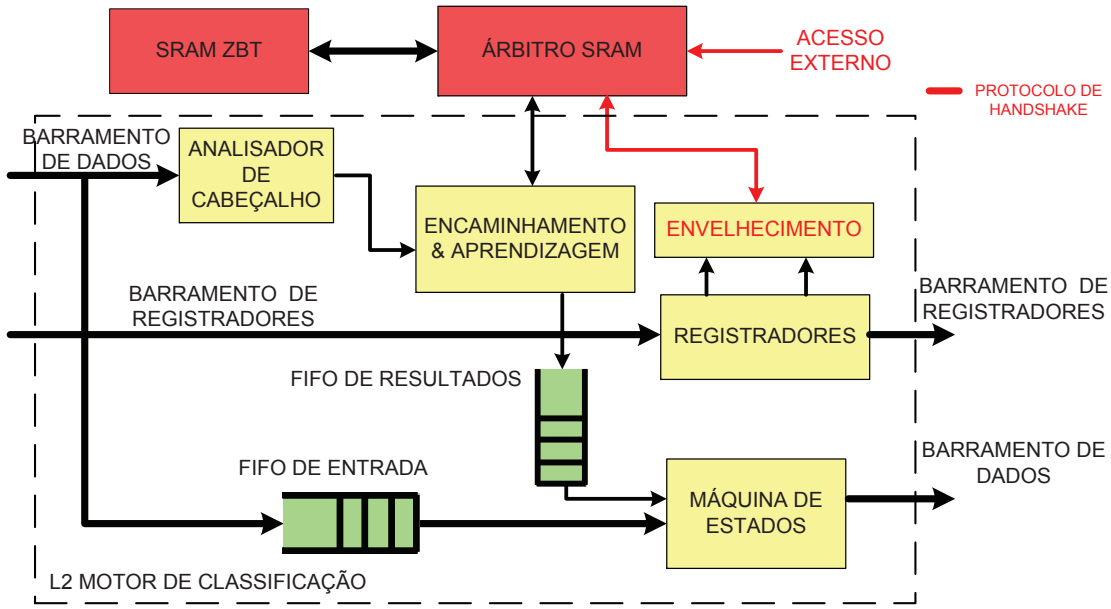

Figura 5.29: Diagrama de blocos da arquitetura 2.

Tabela 5.8: Tabela de ciclos da FSM mostrando as operações realizadas em cada ciclo.

Ciclo 0	Ciclo 1	Ciclo 2	Ciclo 3	Ciclo 4	Ciclo 5	Ciclo 6	Ciclo 7	Ciclo 8	Ciclo 9	Ciclo 10	Ciclo 11	Ciclo 12	Ciclo 13	Ciclo 14	Ciclo 15
		F0 RD REQ	F0 RD REQ	F0 RD REQ	F0 RD REQ	F0 RD REQ	ACESSO REGS	F0 WR REQ	F0 WR REQ						ACESSO ENVELHEC.
F1 WR REQ	F1 WR REQ									F1 RD REQ	F1 RD REQ	F1 RD REQ	F1 RD REQ	F1 RD REQ	

bitro tem a capacidade de receber uma requisição por ciclo de relógio. As requisições e os sinais respectivos para fazer as operações de leitura e escrita estão sendo processados em uma estrutura de *pipeline* para poder cumprir com a característica anterior. Na Figura 5.30 é apresentada a estrutura de *pipeline* para cada uma das operações do árbitro. Nas operações de leitura são utilizadas 5 etapas, enquanto nas operações de escrita só se utilizam 4 etapas.

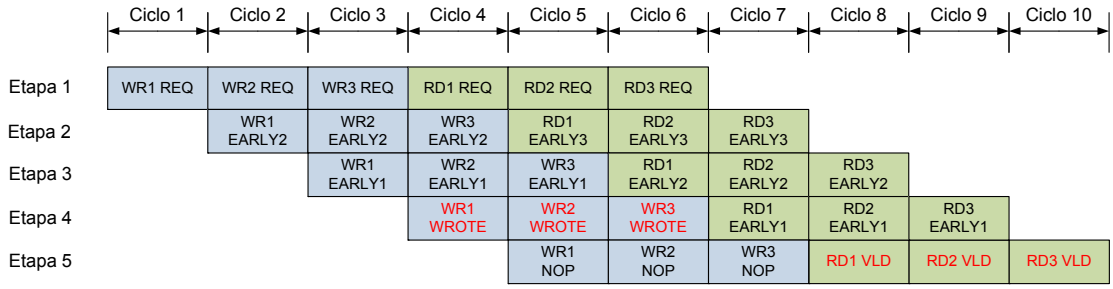

Figura 5.30: Estrutura de *pipeline* do árbitro SRAM.

Esta estrutura de *pipeline* foi projetada baseada nas características das operações de leitura e escrita na memória SRAM externa. Nas figuras 5.31 e 5.32 são apresentados os diagramas de tempo do acesso na memória SRAM externa. O barramento de dados é

bidirecional e o sinal de habilitação de escrita (WE) ativa-se em baixa.

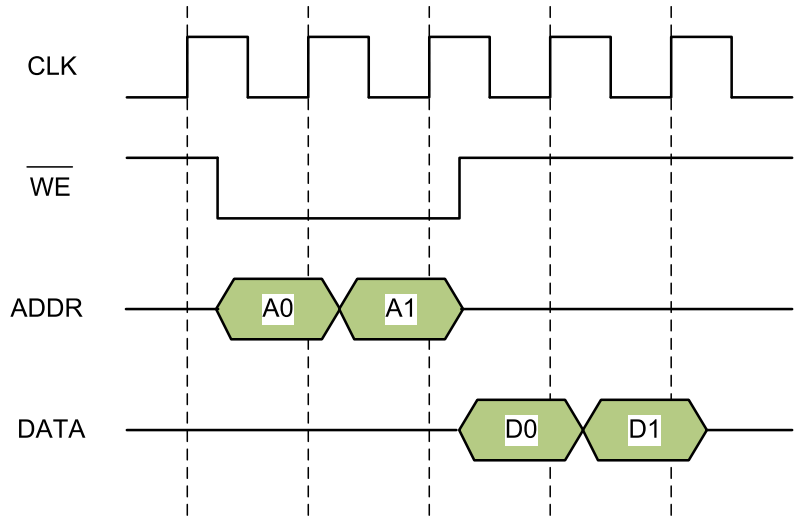

Figura 5.31: Diagrama de tempos de uma operação de escrita na SRAM.

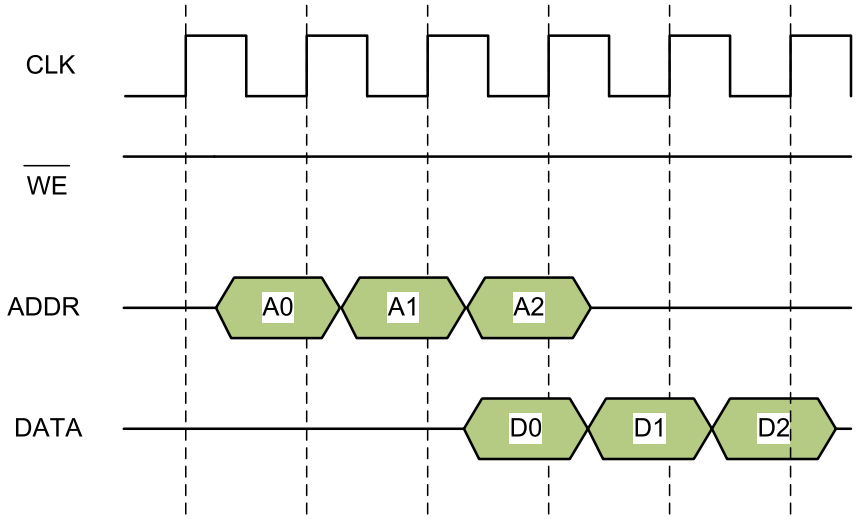

Figura 5.32: Diagrama de tempos de uma operação de leitura na SRAM.

5.2.2.3.2 Encaminhamento e aprendizagem

Este bloco também foi modificado com o objetivo de melhorar o desempenho geral do motor de classificação. A máquina de estados deste bloco também foi modificada para que possa ser sincronizada com a máquina de estados do árbitro SRAM. Agora a máquina de estados deste bloco possui 16 estados com 1 ciclo de relógio por estado. Nos 16 estados são processados 2 quadros *Ethernet*. Para processar um quadro *Ethernet* são necessários 11 ciclos de relógio. Portanto, para poder processar 2 quadros em 16 ciclos, foi colocada cada uma das etapas do processamento do quadro de um modo no qual ambos os processos não interfiram mutuamente durante o acesso da memória SRAM. Na Tabela 5.9 são apresentadas as operações feitas em cada um dos ciclos da FSM. As duas filas representam os dois quadros que são possíveis de processar.

Tabela 5.9: Tabela de ciclos da FSM do bloco de encaminhamento e aprendizagem.

Ciclo 0	Ciclo 1	Ciclo 2	Ciclo 3	Ciclo 4	Ciclo 5	Ciclo 6	Ciclo 7	Ciclo 8	Ciclo 9	Ciclo 10	Ciclo 11	Ciclo 12	Ciclo 13	Ciclo 14	Ciclo 15
F0	F0	F0 RD REQ	F0 RD REQ	F0 RD REQ	F0 RD REQ	F0 RD REQ	F0	F0 WR REQ	F0 WR REQ	F0	F0				
F1 WR REQ	F1 WR REQ	F1	F1					F1	F1	F1 RD REQ	F1 RD REQ	F1 RD REQ	F1 RD REQ	F1 RD REQ	F1

5.3 Marcador de Quadros

Este módulo permite a classificação dos quadros e atribui-lhes uma prioridade. Esta prioridade define uma taxa de quadros por segundo em cada uma das portas. Assim, junto a um mecanismo de qualidade de serviço ou *QoS* se poderia restringir as portas a uma taxa máxima de transmissão. Este módulo foi apresentado em Tonfat, Neuberger e Reis (2010).

O controle de taxa de transmissão funciona com um mecanismo baseado em *créditos* que utiliza depósitos ou *buckets* para controlar a taxa em cada porta do comutador. Os créditos são continuamente incrementados de acordo a uma taxa configurada de incremento e os créditos são decrementados cada vez que um quadro atravessa uma determinada porta. Se nenhum quadro entra por uma determinada porta, os créditos são incrementados até um determinado tamanho configurado do *bucket*.

Cada *bucket* pode absorver rajadas de quadros limitado ao seu tamanho configurado. Quando um *bucket* está vazio, a taxa de transmissão desse *bucket* está limitada à taxa de incremento do *bucket*. Neste ponto, dependendo do método utilizado para a supressão de quadros, estes podem ser eliminados (*dropped*) ou ativa-se o mecanismo de controle de fluxo forçando a porta a reduzir sua taxa de transmissão utilizando quadros de pausa ou *pause frames*.

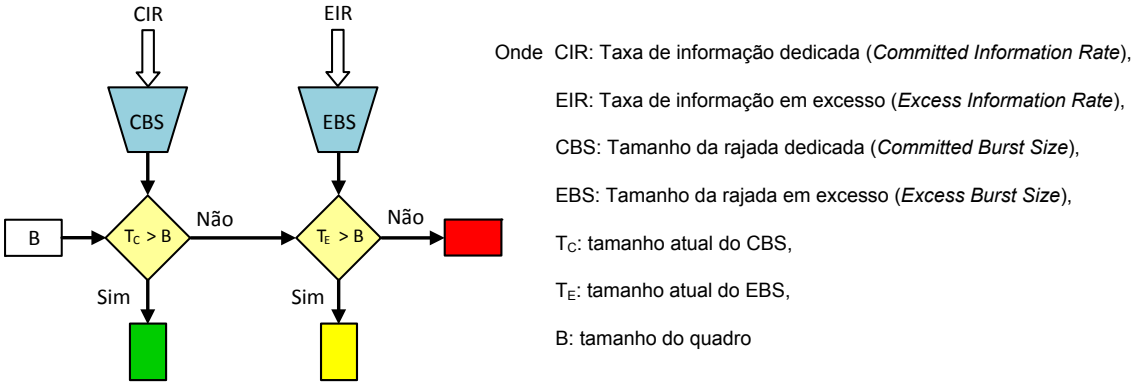

Figura 5.33: Algoritmo de medição e classificação de quadros descrito no RFC 2697 e RFC 4115.

Este módulo utiliza um sistema de créditos de dois depósitos ou *buckets* que possibilitam a implementação de dois sistemas de medição e classificação: *single rate Three Color Marker(srTCM)* e *two rate Three Color Marker (trTCM)*. Estes dois sistemas encontram-se definidos nos seguintes *request for comments* ou *RFCs*: RFC 2697 (HEINANEN; GUERIN, 1999a), RFC 2698 (HEINANEN; GUERIN, 1999b), RFC 4115 (ABOUL-MAGD; RABIE, 2005).

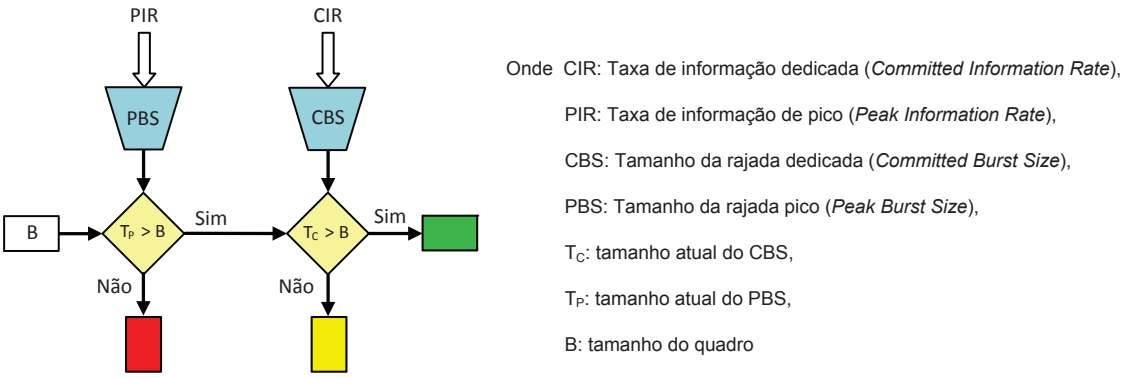

Figura 5.34: Algoritmo de medição e classificação de quadros descrito no RFC 2698.

O sistema descrito no RFC 2698 é um pouco diferente do descrito no RFC 2697 e RFC 4115. A diferença pode ser observada nas Figuras 5.33 e 5.34. A Figura 5.33 mostra o diagrama de fluxo dos algoritmos descritos nos RFC 2697 e RFC 4115. A diferença entre estes dois é que o RFC 2697 determina que EIR = CIR, portanto, define somente uma taxa de informação (*single information rate*). Estas taxas aumentam o tamanho dos *buckets* até um valor máximo definido por CBS e EBS. Nesta Figura, são apresentados os passos para marcar um quadro com três diferentes cores. Os quadros marcados com a cor verde representam aqueles que não ultrapassaram o limite de taxa estabelecido (CIR). Os quadros marcados com a cor amarela representam aqueles quadros que ultrapassaram o limite de taxa estabelecido (CIR) mas não ultrapassaram o limite de taxa de excesso estabelecido (EIR). Os quadros marcados com a cor vermelha ultrapassaram ambos os limites. Cada vez que um quadro passa a ser medido por algum dos dois limites e o tamanho do quadro não ultrapassa o tamanho atual do *bucket* (T_C, T_E), então o *bucket* é decrementado pelo tamanho do quadro.

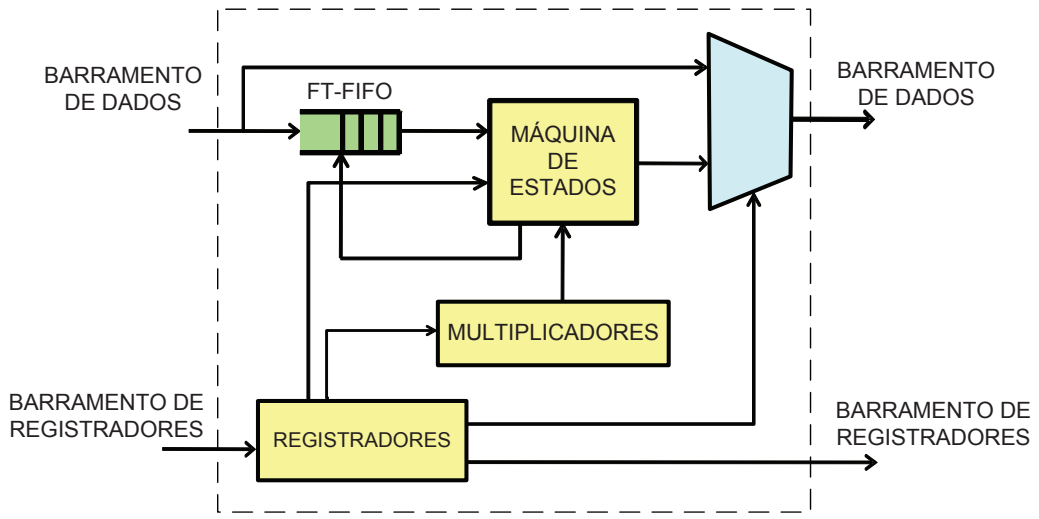

Figura 5.35: Diagrama de blocos do módulo Marcador de Quadros.

Já no RFC 2698, apresentado na Figura 5.34, a marcação dos quadros é um pouco diferente. Os quadros com cor vermelha são aqueles que ultrapassaram o limite de taxa de pico estabelecido (PIR). Os quadros amarelos são aqueles que não ultrapassaram o limite

de taxa pico estabelecido (PIR), mas que ultrapassaram o limite de taxa (CIR). Os quadros verdes são aqueles que não ultrapassaram nenhum dos dois limites. A recomendação é que o valor da taxa PIR seja igual ou maior que a taxa CIR. PBS e CBS definem os tamanhos máximos dos *buckets*. Cada vez que um quadro passa por estes *buckets* e o tamanho do quadro é menor que o tamanho atual dos *buckets* (T_C, T_P), então os *buckets* são decrementados pelo tamanho do quadro.

Para os três RFCs recomenda-se que os tamanhos dos *buckets* seja maior ao tamanho máximo dos quadros processados. Nos três algoritmos, as taxas de transmissão (CIR, EIR, PIR) estão definidas em *bytes* por segundo (B/s).

Para obter uma maior flexibilidade e ainda atender a todas as recomendações desejadas, foram implementados os três RFCs e a escolha destes é feita através de um registrador de configuração. O diagrama de blocos deste módulo é mostrado na Figura 5.35.

Tabela 5.10: Formato do cabeçalho do Marcador de Quadros.

Barramento de controle (8 *bits*)	Barramento de dados (64 *bits*)	
0xFE	Reservado [63:2]	Cor do quadro [1:0]

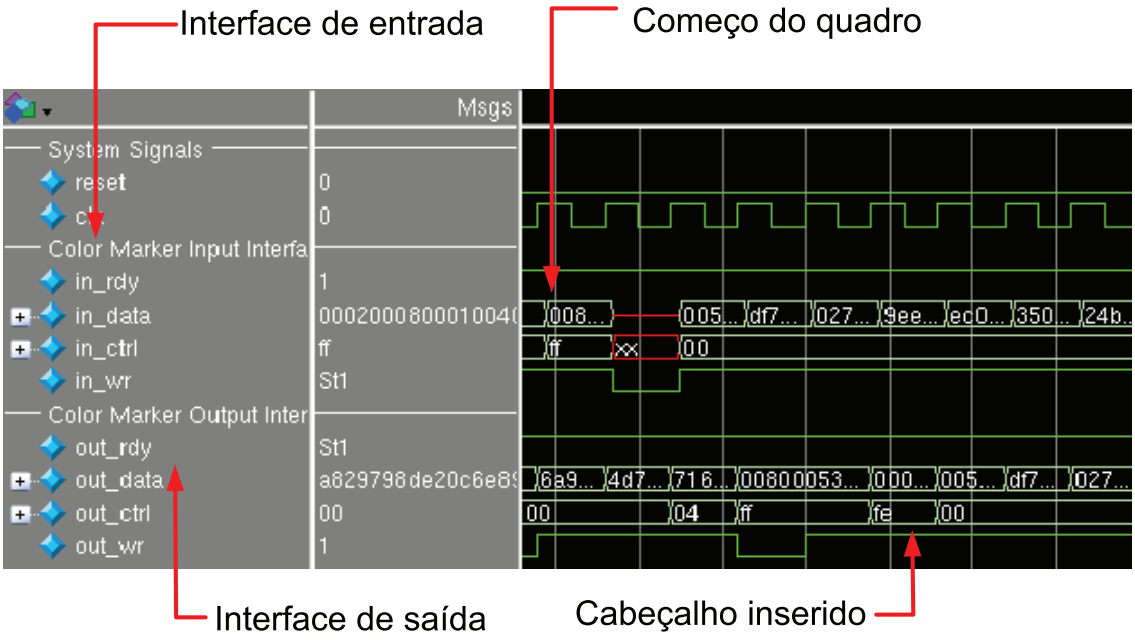

Figura 5.36: Cabeçalho inserido no quadro pelo módulo Marcador de Quadros.

O módulo basicamente é uma máquina de estados que controla o fluxo de dados e implementa os três algoritmos descritos nos RFCs. O módulo possui uma FIFO *fallthrough* (FT-FIFO) de 4 posições para reter as primeiras palavras do quadro até a máquina de estados decidir a prioridade (cor) do quadro. Utilizou-se uma FT-FIFO porque permite ler o tamanho e a porta de entrada do quadro que encontra-se na primeira palavra, segundo o formato NetFPGA, sem extrair esta palavra da FIFO. De posse do tamanho do quadro e o modo de operação vindo dos registradores é atribuído uma cor ao quadro. Como este dado não tem um campo específico nem na trama *Ethernet* nem o cabeçalho do NetFPGA, decidiu-se criar um novo cabeçalho para armazenar este dado. Na Tabela 5.10 é mostrado

o formato do novo cabeçalho inserido por este módulo. Este cabeçalho é inserido depois do cabeçalho padrão do NetFPGA.

Na Figura 5.36 é apresentado um quadro na saída deste módulo. Aqui pode-se observar que na interface de entrada do módulo, o quadro só apresenta o cabeçalho padrão do NetFPGA (CTRL = **0xFF**). Na interface de saída observa-se o mesmo quadro com o novo cabeçalho inserido (CTRL = **0xFE**). Este cabeçalho é inserido depois do cabeçalho padrão.

A latência do módulo depende do modo de operação de cada porta. Se o módulo está desabilitado para todas as portas, então a latência do circuito é zero. Isto obtem-se com o uso de um multiplexor que possui o módulo na saída. Quando está desabilitado para algumas portas, a latência é um ciclo de relógio para as portas em modo desabilitado, e três ciclos de relógio para as portas habilitadas.

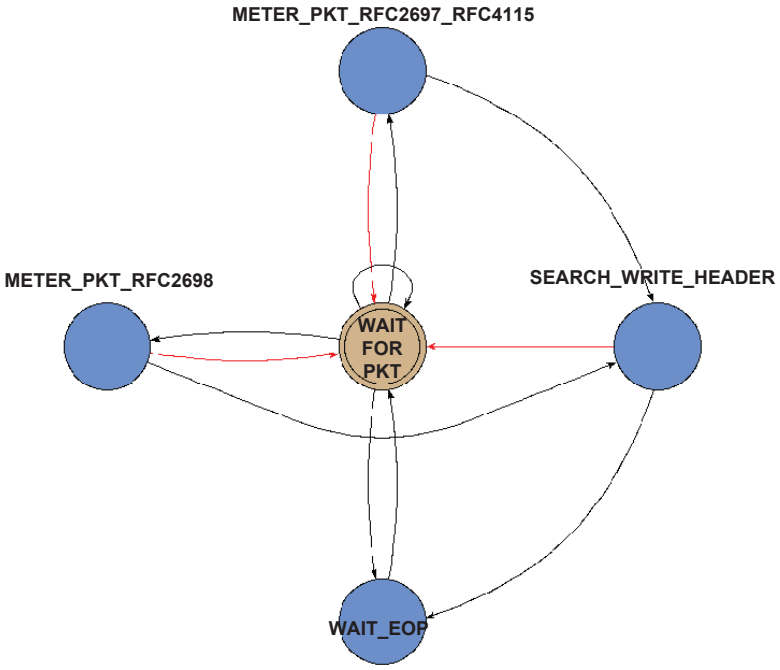

Figura 5.37: Máquina de estados do Marcador de Quadros.

O diagrama de estados é apresentado na Figura 5.37. A máquina FSM possui 5 estados usando codificação binária. Dependendo do modo de operação atribuído a cada porta, a máquina calcula o cor de cada quadro. A máquina de estados também prevê se o quadro foi classificado em um módulo anterior. O módulo mudará a cor do quadro somente quando a cor calculada eleva o nível do quadro (em direção à cor verde).

A Tabela 5.11 apresenta os registradores de configuração deste módulo. Todos os registradores são de 32 *bits*. O registrador COLOR_MARKER_CONFIG permite a configuração do modo de operação de todas as portas atribuindo 2 *bits* para cada porta. Os outros quatro registradores configuram os tamanhos dos *buckets* e as taxas de incremento dos *buckets*. As taxas estão configuradas para serem incrementadas em passos de 8KB/s. Os multiplicadores adaptam os valores dos registradores a este passo mínimo.

Tabela 5.11: Registradores do Marcador de Quadros.

Nome do Registrador	Descrição
COLOR_ MARKER_CONFIG	Dois *bits* por porta são utilizados para configurar o modo de operação: 00: módulo desabilitado, 01: RFC2698, 10: RFC2697, 11: RFC4115 *bit* [31:16] = Reservados, *bit* [15:14]: Modo de operação Porta 7, *bit* [13:12]: Modo de operação Porta 6, *bit* [11:10]: Modo de operação Porta 5, *bit* [9:8]: Modo de operação Porta 4, *bit* [7:6]: Modo de operação Porta 3, *bit* [5:4]: Modo de operação Porta 2, *bit* [3:2]: Modo de operação Porta 1, *bit* [1:0]: Modo de operação Porta 0.
COLOR_MARKER BUCKET0_RATE_X	Configura a taxa de transmissão do primeiro *bucket* da porta X. X representa o número de porta.
COLOR_MARKER BUCKET1_RATE_X	Configura a taxa de transmissão do segundo *bucket* da porta X. X representa o número de porta.
COLOR_MARKER BUCKET0_SIZE_X	Configura o tamanho máximo do primeiro *bucket* da porta X. X representa o número de porta.
COLOR_MARKER BUCKET1_SIZE_X	Configura o tamanho máximo do segundo *bucket* da porta X. X representa o número de porta.

5.4 Filas de Saída

Este é o último módulo do "*User Datapath*" e encarrega-se de implementar as filas de saída do comutador. Estas filas servem para armazenar os quadros *Ethernet* na saída de cada porta. Se a fila de transmissão (*Tx queue*) de uma porta encontra-se ocupada, o quadro será colocado na sua respectiva fila de saída, esperando a liberação da fila de transmissão. Estas filas estão implementadas utilizando uma memória DDR2 SDRAM externa de 64 *MBytes*. O módulo está ligado à memória externa conforme mostra a Figura 5.38. Nesta Figura, além do módulo que implementa toda a lógica das filas de saída, está o bloco que faz a interface entre o domínio de relógio do sistema e o domínio de relógio da memória DDR2 SDRAM e o controlador da memória DDR2 SDRAM.

Este módulo faz parte da plataforma NetFPGA, mas teve que ser modificado para poder ser implementado em ASIC. A mudança mais significativa foi a troca das FIFOs nas entradas e saídas das filas SDRAM de cada porta. No módulo original, estas são implementadas em blocos de BRAM da FPGA Virtex-IIPRO da Xilinx (VIRTEX-II PRO AND VIRTEX-II PRO X PLATFORM FPGAS: COMPLETE DATASHEET, 2011). Para a implementação em ASIC, as FIFOs foram implementadas com blocos de memória de células de oito transistores (8T-SRAM) e de dupla porta. Estas FIFOs foram geradas com a ferramenta fornecida pela empresa criadora da biblioteca de células padrão (*standard-cells*). Estas FIFOs não podem ser implementadas com *flip-flops* pois utilizam memórias muito grandes. Isto geraria maior área (uma célula 8T-SRAM tem menor área que um *flip-flop*) e maior consumo de potência. A seguir é descrito o módulo e sua interface com

a memória.

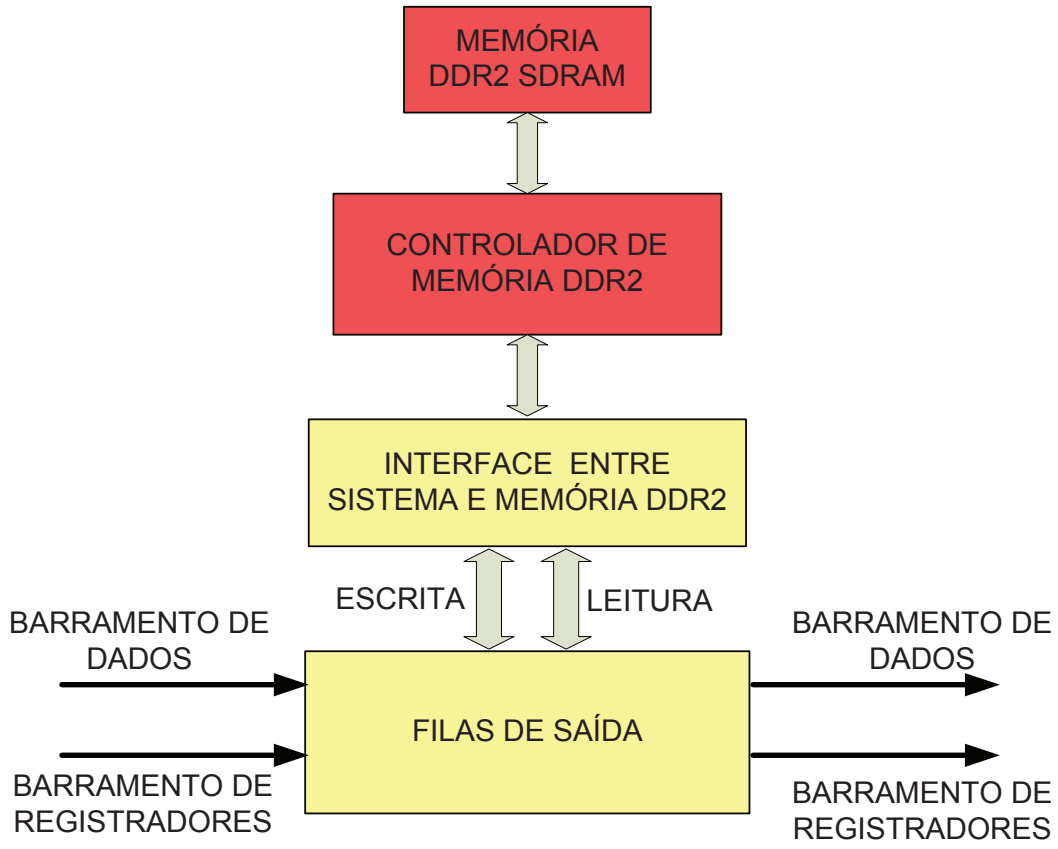

Figura 5.38: Diagrama de blocos do módulo das Filas de Saída e sua interface com a memória DDR2 SDRAM externa.

5.4.1 Lógica das filas de saída

Este módulo implementa toda a lógica das filas de saída. O diagrama de blocos é apresentado na Figura 5.39, e está composto por um grupo de blocos que serão explicados a seguir.

5.4.1.1 FIFO de entrada

Esta FIFO de entrada é do tipo *fallthrough*. Possui 8 posições e encarrega-se de armazenar as primeiras palavras de cada quadro e permite que o bloco que analisa o cabeçalho consiga extrair os campos importantes.

5.4.1.2 Analisador de cabeçalho

Este bloco encarrega-se de extrair os dados importantes do quadro para o módulo, a saber: a(s) porta(s) de saída do quadro que foi calculado no módulo pesquisador da porta de saída e o tamanho do quadro em *bytes* e em palavras (*words*). Para cumprir com esta tarefa, utiliza uma máquina de estados simples e uma FIFO *fallthrough* para armazenar os dados extraídos dos quadros e esperar que sejam solicitados pelo bloco "Árbitro das Filas DRAM". O diagrama de estados deste bloco é apresentado na Figura 5.40.

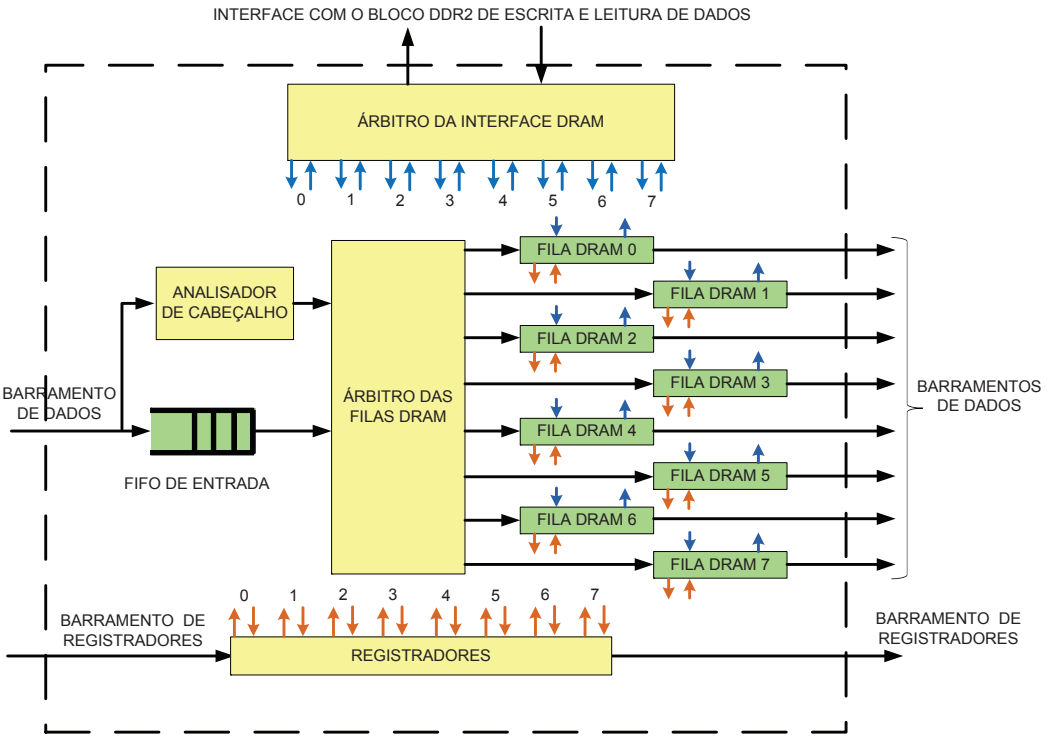

Figura 5.39: Diagrama de blocos da lógica interna das filas de saída.

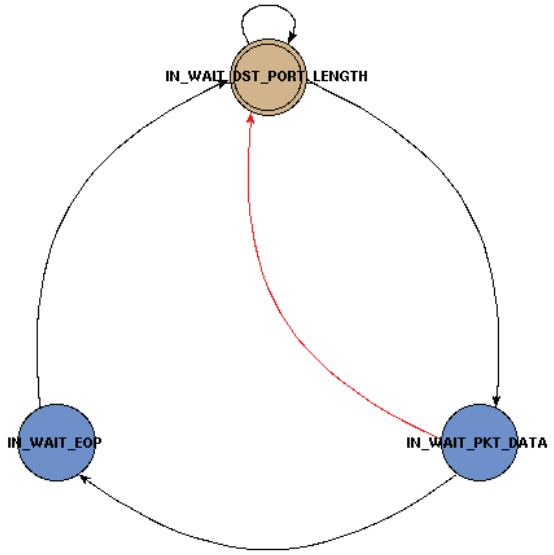

Figura 5.40: Diagrama de estados do bloco analisador de cabeçalho.

5.4.1.3 Árbitro das Filas DRAM

Este bloco tem a função de encaminhar os quadros para as filas de saída. Quando um quadro tem que ser encaminhado a mais de uma porta de saída (Ex: endereço de destino *multicast* ou *broadcast*), este bloco copia o quadro para todas as portas necessárias. Este bloco recebe os dados do quadro do bloco "Analisador de cabeçalho". Antes de enviar o quadro para a respectiva fila, este bloco calcula se a fila de saída tem espaço suficiente para receber o quadro. Com o tamanho do quadro e o estado atual da fila de saída, consegue

determinar se a fila vai conseguir armazenar o quadro. Se a fila não tem o espaço suficiente para armazenar o quadro então o quadro é descartado (*dropped*). A quantidade de quadros descartados são atualizados no bloco “Registradores”.

5.4.1.4 Filas DRAM

Neste trabalho são implementadas oito filas DRAM. Estas filas encaminham os quadros para a interface com a DRAM. Cada fila possui um espaço de endereços do mesmo tamanho na memória DRAM. Esta configuração pode ser modificada com os registradores “LOW_ADDR” e “HIGH_ADDR” de cada fila, e neste caso, deve-se verificar se não existe uma sobreposição do espaço de endereços, pois o módulo não faz esta verificação. A arquitetura das filas é mostrada na Figura 5.41. Este bloco implementa duas FIFOs e a interface com a memória externa DDR2. Estas FIFOs na entrada e saída servem como *buffers*.

Para atingir as diferentes larguras de dados entre este módulo (72 *bits*) e o bloco DDR2 de escrita e leitura de dados (144 *bits*), estas FIFOs têm diferentes larguras na entrada e na saída. Para que a FIFO de entrada consiga passar um quadro da interface de 72 *bits* para 144 *bits*, o quadro deve possuir um número de palavras par. Para evitar que a última palavra de um quadro com número de palavras ímpar fique na FIFO de entrada (72 *bits* a 144 *bits*), uma palavra extra (*dummy*) é adicionada ao quadro. Esta palavra extra será retirada quando sair deste bloco na FIFO de saída.

Cada FIFO consegue armazenar até 4 KB. Quando os dados na FIFO da entrada da fila DRAM excedem os 2 KB, o conteúdo é transferido à memória externa. Do mesmo modo, quando a FIFO de saída tem espaço para mais de 2 KB, o conteúdo é transferido da memória externa à FIFO de saída.

Como observação, se o conteúdo na FIFO de entrada é menor que o limite de 2 KB, estes dados ficarão presos na FIFO para sempre. Para evitar isto, existe um caminho direto entre as duas FIFOs. Quando a FIFO de saída não consegue receber um quadro da FIFO de entrada, o quadro é enviado para a memória externa DDR2 SDRAM. Então a DRAM só é utilizada quando tem-se pelo menos 2 KB de dados acumulados na FIFO de entrada.

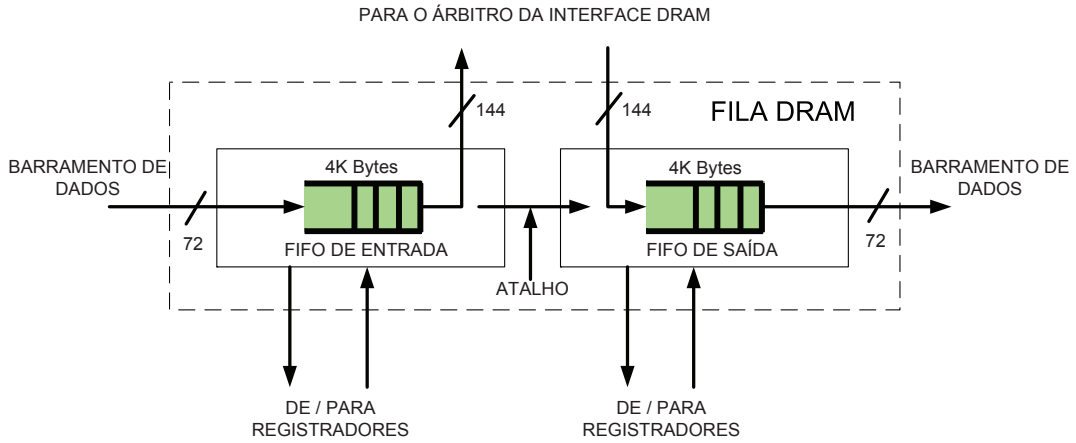

Figura 5.41: Arquitetura das Filas DRAM.

5.4.1.5 Árbitro da interface DRAM

Este bloco encarrega-se do encaminhamento dos pedidos de acesso à memória DDR2 SDRAM de cada uma das filas. Utiliza um algoritmo *Round Robin* para servir a cada fila, uma por vez.

5.4.1.6 Registradores

Este bloco contém todos os registradores deste módulo. Na Tabela 5.12 é apresentada a lista dos registradores deste módulo. *X* representa o número da fila.

Tabela 5.12: Registradores das filas de saída.

Nome do Registrador	Descrição
PKTS_STORED_X	Registrador que indica a quantidade de quadros da fila X armazenados na DRAM.
PKTS_DROPPED_X	Registrador que indica a quantidade de quadros da fila X descartados.
PKTS_REMOVED_X	Registrador que indica a quantidade de quadros da fila X retirados da DRAM.
SHORTCUT_WORDS_X	Registrador que indica a quantidade de palavras que passaram pela fila X sem usar a DRAM.
INPUT_WORDS_X	Registrador que indica a quantidade de palavras que entraram na fila X.
OUTPUT_WORDS_X	Registrador que indica a quantidade de palavras que saíram na fila X.
DRAM_WR_WORDS_X	Registrador que indica a quantidade de palavras da fila X que são escritas na DRAM.
DRAM_RD_WORDS_X	Registrador que indica a quantidade de palavras da fila X que são lidas da DRAM.
BLK_ADDR_LO_X	Registrador que configura o começo do espaço de memória da fila X.
BLK_ADDR_HI_X	Registrador que configura o fim do espaço de memória da fila X.
SHORTCUT_DISABLE_X	Registrador que configura o caminho entre a FIFO de entrada e saída da fila X.
CTRL_X	Registrador que configura a reinicialização da fila X.
RD_ADDR_X	Registrador que indica o endereço de leitura na DRAM da fila X.
WR_ADDR_X	Registrador que indica o endereço de escrita na DRAM da fila X.

5.4.2 Bloco DDR2 de escrita e leitura de dados

Este bloco fornece uma interface entre a lógica das filas de saída e o controlador da memória DDR2 SDRAM. Todos os sinais que utilizam a lógica das filas para acessar à memória DDR2 SDRAM são síncronas ao relógio dos módulos do "*user datapath*" ou do sistema, portanto isola a lógica das filas de saída do domínio de relógio da DDR2 SDRAM (relógio de 200 MHz). Os sinais são detalhados nas Tabelas 5.13 e 5.14. Também as operações típicas da DRAM, como o ciclo de *refresh* são invisíveis à lógica das filas de saída.

Tabela 5.13: Tabela de sinais na interface entre a lógica das filas de saída e o bloco DDR2 de escrita e leitura de dados. Os sinais são síncronos ao relógio das Filas de saída.

Grupo de Sinais	Nome do sinal	Fonte	Bits	Descrição
Pedido de transação	p_wr_req	Filas de saída	1	1 = pedido de escrita, 0 = caso contrário.
Pedido de transação	p_wr_ptr	Filas de saída	22	Endereço inicial da DRAM para a transferência de dados.
Pedido de transação	p_wr_ack	Bloco DDR2 de escrita e leitura de dados	1	1 = O árbitro confirma que a operação de escrita pode ser feita, 0 = caso contrário.
Transferência de dados	p_wr_data_vld	Filas de saída	1	1 = os dados de escrita são válidos, 0 = caso contrário.
Transferência de dados	p_wr_data	Filas de saída	144	Os dados que são transferidos para a DRAM.
Transferência de dados	p_wr_full	Bloco DDR2 de escrita e leitura de dados	1	1 = notifica às filas de saída para suspender a transferência no próximo ciclo de relógio até o sinal ser desativado, 0 = caso contrário.
Transferência de dados	p_wr_done	Bloco DDR2 de escrita e leitura de dados	1	1 = indica que este será o último dado, 0 = caso contrário.
Pedido de transação	p_rd_req	Filas de saída	1	1 = pedido de leitura, 0 = caso contrário.
Pedido de transação	p_rd_ptr	Filas de saída	22	Endereço inicial da DRAM para a transferência de dados.
Pedido de transação	p_rd_ack	Bloco DDR2 de escrita e leitura de dados	1	1 = O árbitro confirma que a operação de leitura pode ser feita, 0 = caso contrário.
Transferência de dados	p_rd_rdy	Bloco DDR2 de escrita e leitura de dados	1	1 = o bloco possui dados que podem ser lidos, 0 = caso contrário.
Transferência de dados	p_rd_en	Filas de saída	1	1 = o módulo das filas de saída está lendo uma palavra de dados, 0 = caso contrário.
Transferência de dados	p_rd_data	Bloco DDR2 de escrita e leitura de dados	144	Os dados transferidos desde a DRAM.
Transferência de dados	p_rd_done	Bloco DDR2 de escrita e leitura de dados	1	1 = indica que este será o último dado, 0 = caso contrário.

As operações de leitura e escrita são feitas somente sobre blocos de dados completos. Estes pedidos de acesso na DDR2 SDRAM são arbitrados utilizando um algoritmo *Round Robin*. Quando uma das operações (leitura ou escrita) faz acesso à DDR2 SDRAM, este acesso não é interrompido até a operação terminar.

Este sistema precisa que os acessos em rajada no domínio de relógio da memória não sejam detidos. Por isso é necessário que a lógica do domínio de relógio do sistema escreva mais rápido do que a operação de escrita em rajada no domínio de relógio da memória. Da mesma forma, a lógica do domínio de relógio do sistema deve ler mais rápido que a operação feita no domínio de relógio da memória. Para garantir este cenário precisamos que a seguinte inequação se cumpra:

Tabela 5.14: Tabela de sinais na interface entre o bloco DDR2 de escrita e leitura de dados e o controlador da memória DDR2. Os sinais são síncronos ao relógio do controlador DDR2.

Grupo de Sinais	Nome do sinal	Fonte	Bits	Descrição
Informação	init_val_180	Controlador DDR2	1	1 = Inicialização da DDR2 DRAM completa, 0 = caso contrário.
Informação	cmd_ack_180	Controlador DDR2	1	1 = Controlador DDR2 lendo ou escrevendo na DDR2 DRAM, 0 = caso contrário.
Informação	auto_ref_req_180	Controlador DDR2	1	1 = a memória DDR2 DRAM precisa de um ciclo de *refresh* , 0 = caso contrário.
Informação	ar_done_180	Controlador DDR2	1	1 = ciclo de refresh completado, 0 = caso contrário.
Transferência de dados	rd_data_90	Controlador DDR2	64	Dados lidos da memória DDR2 DRAM.
Transferência de dados	rd_data_valid_90	Controlador DDR2	1	1 = o dado em rd_data_90 é válido, 0 = caso contrário.
Vários	reset_0	Controlador DDR2	1	reset síncrono a clk_0.
Vários	clk_0	Controlador DDR2	1	relógio de referência da DDR2 DRAM.
Vários	clk_90	Controlador DDR2	1	relógio da DDR2 DRAM com fase de 90 graus.
Configuração do controlador DDR2	cmd_180	Bloco DDR2 de escrita e leitura de dados	4	Registrador de comandos do controlador DDR2.
Transferência de dados	bank_addr_0	Bloco DDR2 de escrita e leitura de dados	2	Endereço do banco da DDR2 DRAM.
Transferência de dados	addr_0	Bloco DDR2 de escrita e leitura de dados	22	Endereço de fila e coluna da DDR2 DRAM
Informação do controlador DDR2	burst_done_0	Bloco DDR2 de escrita e leitura de dados	1	1 = transferência de dados completada, 0 = caso contrário.
Configuração do controlador DDR2	config1	Bloco DDR2 de escrita e leitura de dados	15	Registrador 1 de configuração dos paramêtros da DDR2 DRAM.
Configuração do controlador DDR2	config2	Bloco DDR2 de escrita e leitura de dados	13	Registrador 2 de configuração dos paramêtros da DDR2 DRAM.
Transferência de dados	wr_data_90	Bloco DDR2 de escrita e leitura de dados	64	Dados escritos na memória DDR2 DRAM.
Transferência de dados	wr_data_mask_90	Bloco DDR2 de escrita e leitura de dados	8	Máscara para a escrita de dados na DDR2 DRAM.

$$Freq_{sistema} * (larguradosdadosdoquadro) > Freq_{DDR2} * (64bits)$$

A largura do barramento de dados é 144 *bits* e, portanto, o tamanho do bloco que a lógica das filas de saída deve ler ou escrever é de 2034 *bytes* (113 palavras de 144 *bits* cada). Esta configuração deve trabalhar com uma frequência de relógio de sistema de 125

MHz para alcançar a taxa de transferência necessária da memória DDR2 SDRAM. Esta taxa deve ser maior que 9,9 *Gbps*.

A plataforma NetFPGA suporta até 4 controladores MAC *Gigabit* trabalhando a velocidade *full-duplex* e uma interface PCI de 32 *bits* (com até 4 CPU DMA) com um relógio de 33 MHz. A taxa mínima de dados de leitura/escrita para suportar esta configuração deve ser:

$$4Gbps * 2 + 32bits * 33MHz = 8Gbps + 1,056Gbps = 9,056Gbps$$

A largura do barramento de dados para a DDR2 é 64 *bits*. A frequência do relógio é 200 MHz. Cerca de 15% dos ciclos são perdidos devido à latencia CAS/RAS e à arbitragem da DDR2. Outros 2,6% são perdidos devido ao "*refresh*" da memória. Além disso, para cada 8 *bytes* de dados, temos 1 *byte* de controle que também é armazenado na memória. O tamanho dos blocos de dados que gera a lógica das filas de saída é 2304 *bytes*, mas é preenchido até 2048 *bytes* que é o tamanho dos blocos da memória DDR2. Com isto, a taxa de dados efetiva da memória DDR2 SDRAM é:

$$200MHz * 64bits * [(100 - 15 - 2,6)\%] * 8/9(bytes) * 2034/2048(pad) = 9,31Gbps$$

Este bloco está composto basicamente de duas máquinas de estado e algumas FIFOs síncronas e assíncronas. A função principal é trocar dados entre dois domínios de relógio diferentes. Na Figura 5.42 é apresentado o diagrama de blocos simplificado do bloco.

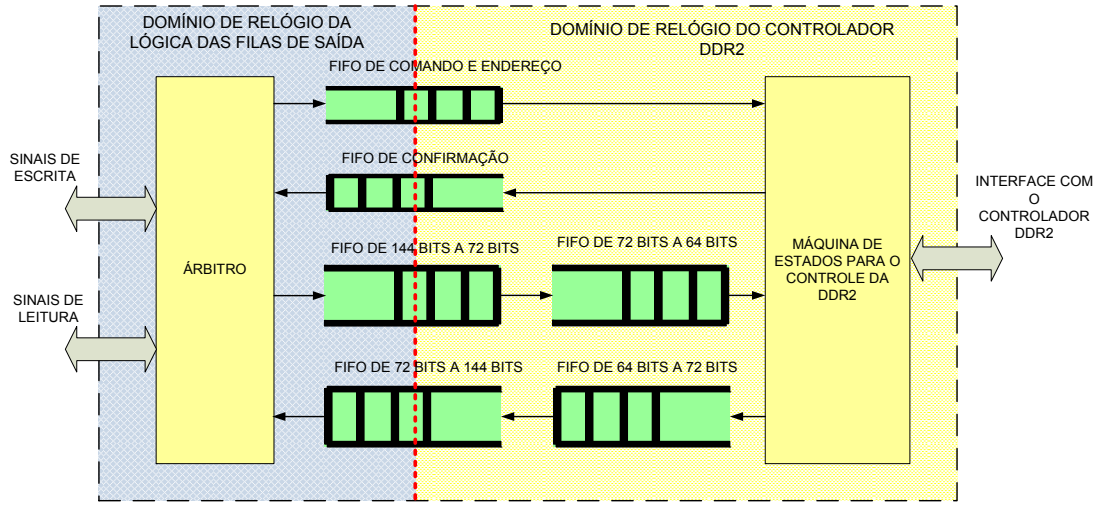

Figura 5.42: Diagrama de blocos do bloco interface entre o sistema e o controlador DDR2.

As caracteristicas destas FIFOs são apresentadas a seguir:

- **FIFO de comando e endereço:** Esta FIFO envia o comando (leitura ou escrita) e o endereço inicial da memória do domínio de relógio da lógica das filas de saída para o domínio de relógio do controlador DDR2.
  - Tipo: Assíncrona;
  - Largura dos dados: 22 + 2 *bits*: 22 *bits* para o endereço e 2 *bits* para o comando;
  - Número de posições: 4;

  - Tipo de leitura: *Fallthrough*.

- **FIFO de confirmação (*acknowledge*):** Esta FIFO envia o sinal de confirmação do domínio de relógio do controlador DDR2 para o domínio de relógio da lógica das filas de saída.
  - Tipo: Assíncrona;
  - Largura dos dados: 1 *bit*;
  - Número de posições: 4;
  - Tipo de leitura: *Fallthrough*.

- **FIFO de 144 *bits* a 72 *bits*:** Esta FIFO envia dados do domínio de relógio da lógica das filas de saída para o domínio de relógio do controlador DDR2.
  - Tipo: Assíncrona;
  - Largura dos dados: 144 *bits* na entrada e 72 *bits* na saída;
  - Número de posições na entrada: 16;
  - Número de posições na saída: 32;
  - Tipo de leitura: *Fallthrough*.

- **FIFO de 72 *bits* a 144 *bits*:** Esta FIFO envia dados do domínio de relógio do controlador DDR2 para o domínio de relógio da lógica das filas de saída.
  - Tipo: Assíncrona;
  - Largura dos dados: 72 *bits* na entrada e 144 *bits* na saída;
  - Número de posições na entrada: 16;
  - Número de posições na saída: 16;
  - Tipo de leitura: *Fallthrough*.

- **FIFO de 72 *bits* a 64 *bits*:** Esta FIFO converte dados de 72 *bits* a 64 *bits* no domínio de relógio do controlador DDR2.
  - Tipo: Síncrona;
  - Largura dos dados: 72 *bits* na entrada e 64 *bits* na saída;
  - Número de posições: 4;
  - Tipo de leitura: *Fallthrough*.

- **FIFO de 72 *bits* a 64 *bits*:** Esta FIFO converte dados de 72 *bits* a 64 *bits* no domínio de relógio do controlador DDR2.
  - Tipo: Síncrona;
  - Largura dos dados: 72 *bits* na entrada e 64 *bits* na saída;
  - Número de posições: 4;
  - Tipo de leitura: *Fallthrough*.

- **FIFO de 64 *bits* a 72 *bits*:** Esta FIFO converte dados de 64 *bits* a 72 *bits* no domínio de relógio do controlador DDR2.

- Tipo: Síncrona;
- Largura dos dados: 64 *bits* na entrada e 72 *bits* na saída;
- Número de posições: 4;
- Tipo de leitura: *Fallthrough*.

Nas Figuras 5.43 e 5.44 são mostrados os diagramas de tempo de uma operação de leitura e escrita entre a lógica das saídas e o bloco DDR2 de escrita e leitura de dados. O sinal de relógio mostrado nessas Figuras é o relógio do módulo “Filas de saída”.

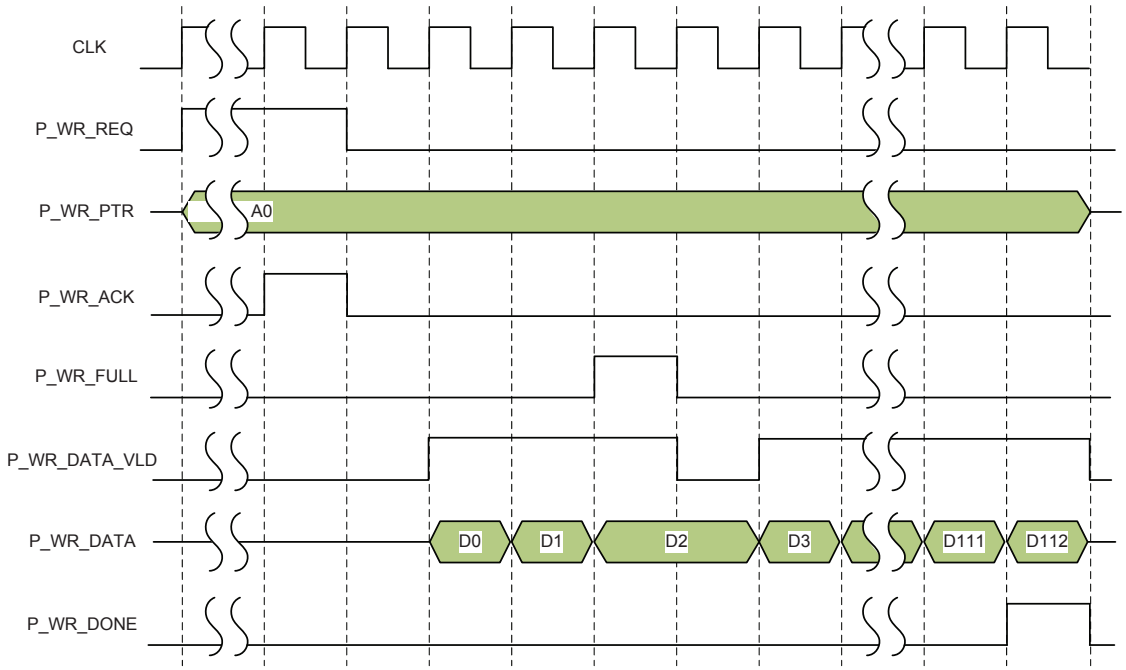

Figura 5.43: Operação de escrita na DRAM.

5.4.3 Adaptação do controlador de memória DDR2 SDRAM

O controlador de memória DDR2 SDRAM permite o armazenamento e leitura de dados na memória DDR2 SDRAM. Na plataforma NetFPGA é utilizado o controlador gerado pela ferramenta MIG (XILINX MEMORY INTERFACE GENERATOR , MIG 007) da *Xilinx* para seu FPGA VirtexII-PRO (VIRTEX-II PRO AND VIRTEX-II PRO X PLATFORM FPGAS: COMPLETE DATASHEET, 2011). Este controlador está projetado para ser usado especificamente em FPGAs da *Xilinx*, por esse motivo não pode ser utilizado para uma implementação em ASIC.

Atualmente os controladores de memória SDRAM são compostos de três blocos: o controlador lógico, a parte operativa (*datapath*) e a camada física ou PHY. Na Figura 5.45 é apresentada esta organização.

O controlador lógico é encarregado pelo controle do fluxo de dados entre a memória e a lógica do usuário. Também é responsável pela tradução dos comandos gerados pela lógica do usuário para os comandos da memória e pela tradução dos endereços para o esquema de linha e coluna. Faz o controle da latência dos dados, da sequência de inicialização da memória SDRAM e de tarefas de manutenção como o ciclo periódico de *refresh*.

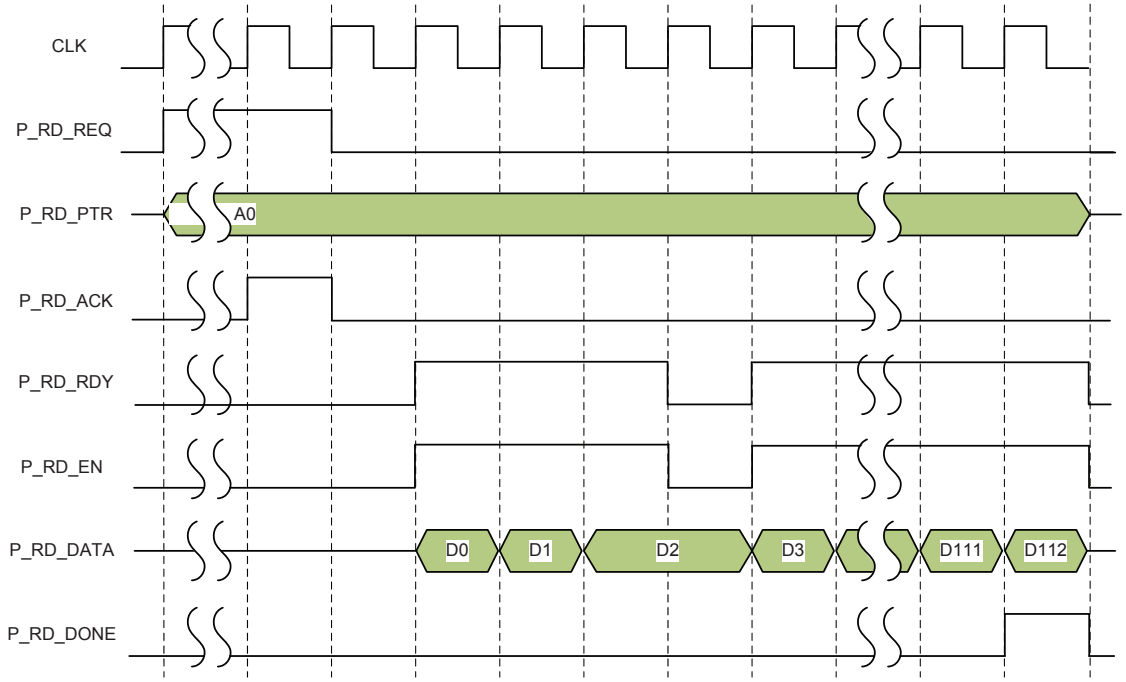

Figura 5.44: Operação de leitura na DRAM.

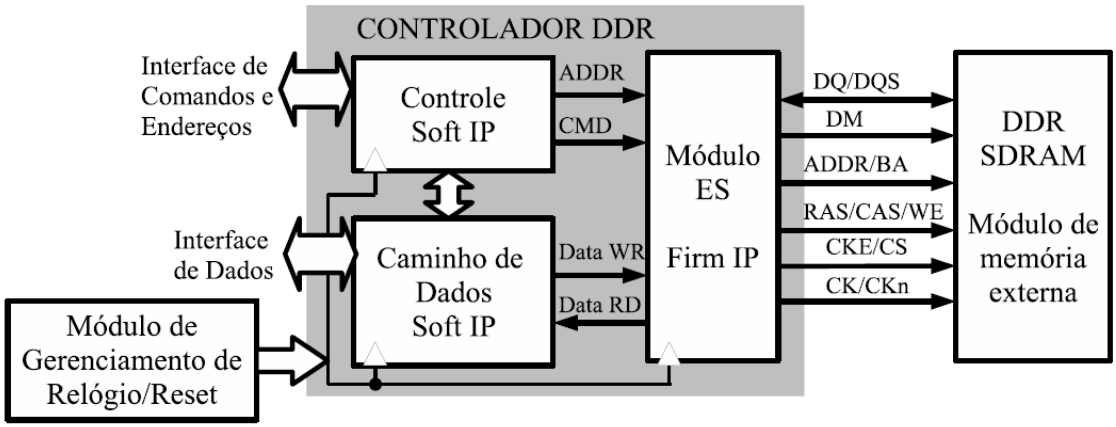

Figura 5.45: Diagrama de blocos de um controlador de memória SDRAM. (BONATTO, 2009)

A parte operativa contém o caminho dos dados que serão armazenados ou lidos na/da memória DDR2 SDRAM. Por último, a camada física implementa os elementos da interface física com a memória externa como *buffers*, registradores DDR (*Double Data Rate*), buffers tri-estado e elementos de controle do atraso e fase dos sinais como DLLs (*Delay locked loops*).

O uso destes elementos não permite que este bloco seja projetado facilmente com *standard-cells* e geralmente este bloco seja adquirido como um IP-core para o desenvolvimento de um ASIC.

Um consórcio de empresas de fabricantes de PHYs para memórias SDRAM, desenvolveu uma especificação para a interface entre o PHY e o resto dos blocos do controlador. Esta interface é chamada como DFI (DDR PHY INTERFACE (DFI) SPECIFICATION 2.1.1, 2010). Esta especificação visa aumentar o potencial reuso dos blocos que compõem

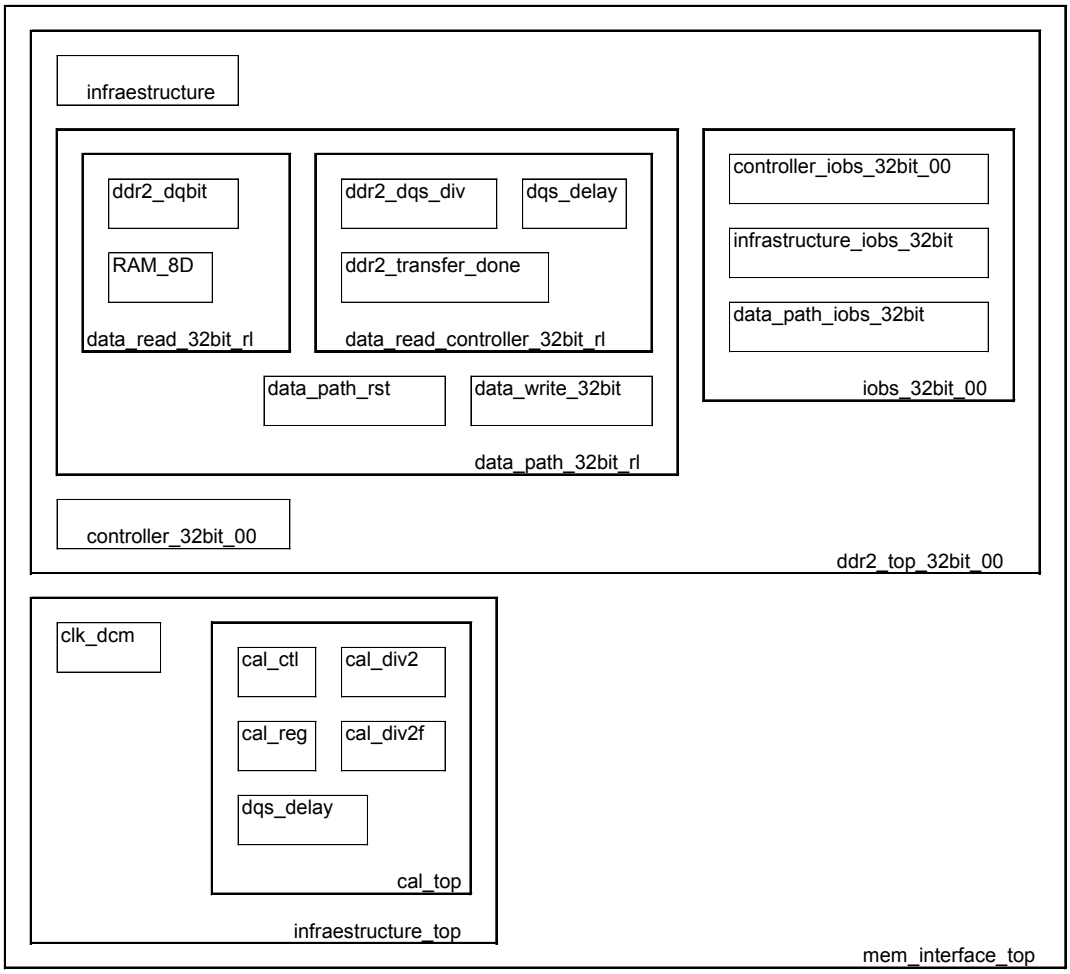

Figura 5.46: Hierarquia dos blocos do controlador original gerado pela Xilinx.

o controlador e reduzir o tempo de integração dos blocos que podem ter sido projetados por diferentes fabricantes. Esta especificação define os sinais, as relações entre estes sinais e os parâmetros de tempo necessários para trocar dados e informações de controle. Esta especificação não define valores absolutos para os atrasos nos sinais mas sim, uma relação de tempo entre eles.

Como o controlador da *Xilinx* foi projetado antes do surgimento da especificação DFI, este controlador não possui uma separação clara entre o PHY, a parte operativa e o controlador lógico. Então, foi analisado o código fonte do controlador para achar essa divisão. Além de encontrar a divisão, teve-se que fazer uma adaptação dos sinais para que sejam compatíveis com o DFI. Isto nos permitiria utilizar o mesmo controlador lógico para a implementação em ASIC e FPGA.

Na Figura 5.46 é apresentado a organização dos blocos no controlador gerado pela ferramenta MIG da *Xilinx*. Esta hierarquia foi modificada para ser compatível com a especificação DFI.

Na Figura 5.47 é apresentada a hierarquia modificada do controlador. Nesta hierarquia é implementado os sinais da interface DFI entre o controlador lógico e o PHY que são mostrados na Figura 5.48. Na Tabela 5.15 são detalhadas as dependências dos sinais com os parâmetros definidos no DFI. Os parâmetros são explicados a seguir:

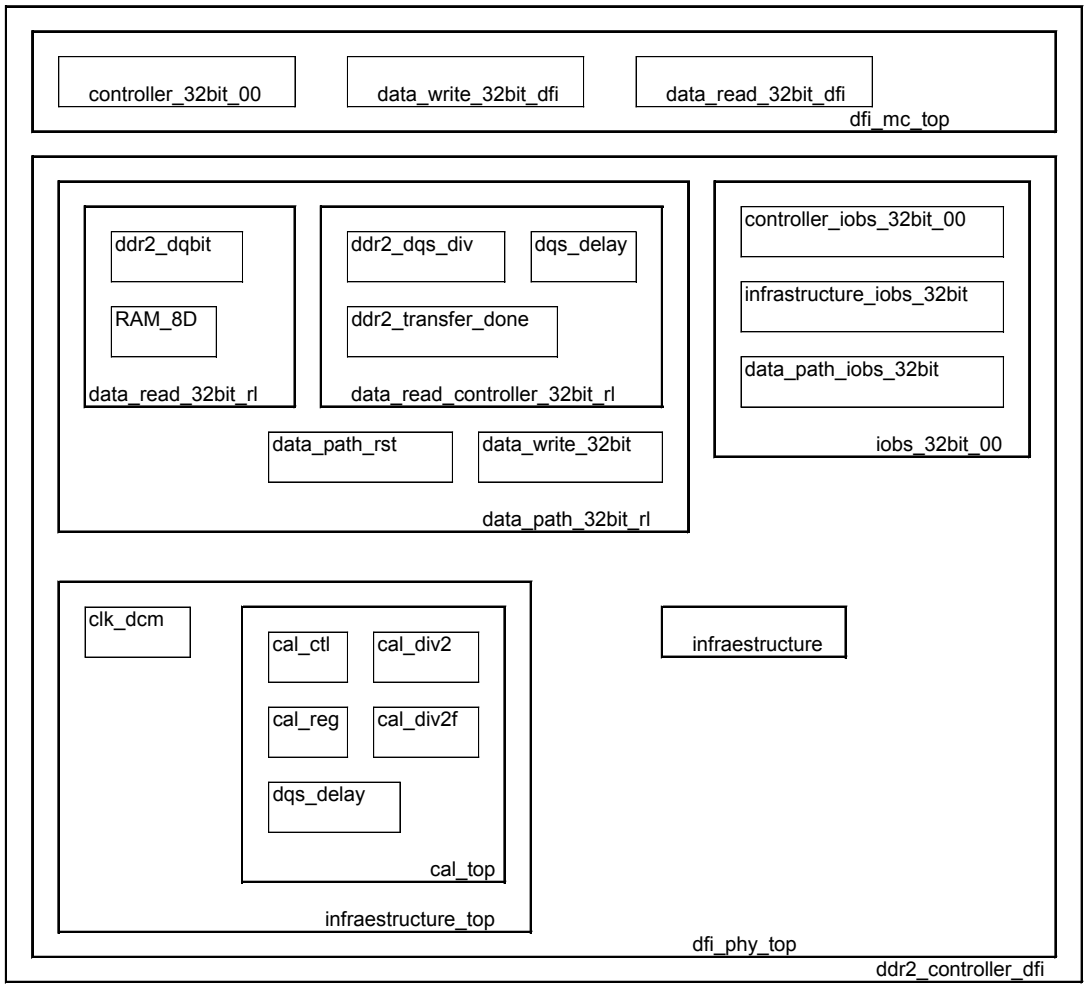

Figura 5.47: Hierarquia dos blocos do controlador modificado com a interface DFI.

- t_{ctrl_delay}: Este parâmetro indica a latência que deve manter todos os sinais do grupo de controle ao passar pelo PHY. Este atraso é medido em função do número de ciclos de relógio do controlador DDR2. Um exemplo é mostrado na Figura 5.49;

- t_{phy_wrdata}: Este parâmetro indica a quantidade de ciclos que o sinal dfi_wrdata deve esperar para colocar os dados depois de ativar o sinal dfi_wrdata_en. Na Figura 5.50 apresenta-se um exemplo;

- t_{phy_wrlat}: Este parâmetro indica a quantidade de ciclos que o sinal dfi_wrdata_en deve esperar para ativar-se depois de inserir o comando de escrita. Na Figura 5.50 é apresentado um exemplo;

- t_{rddata_en}: Este parâmetro indica a quantidade de ciclos que o sinal dfi_rddata_en deve esperar para ativar-se depois de inserir o comando de leitura. Na Figura 5.51 é apresentado um exemplo;

- t_{phy_rdlat}: Este parâmetro define o número máximo de ciclos de latência entre a ativação do sinal dfi_rddata_en e a ativação dos sinais dfi_rddata_valid e dfi_rddata. Na Figura 5.51 é mostrado um exemplo;

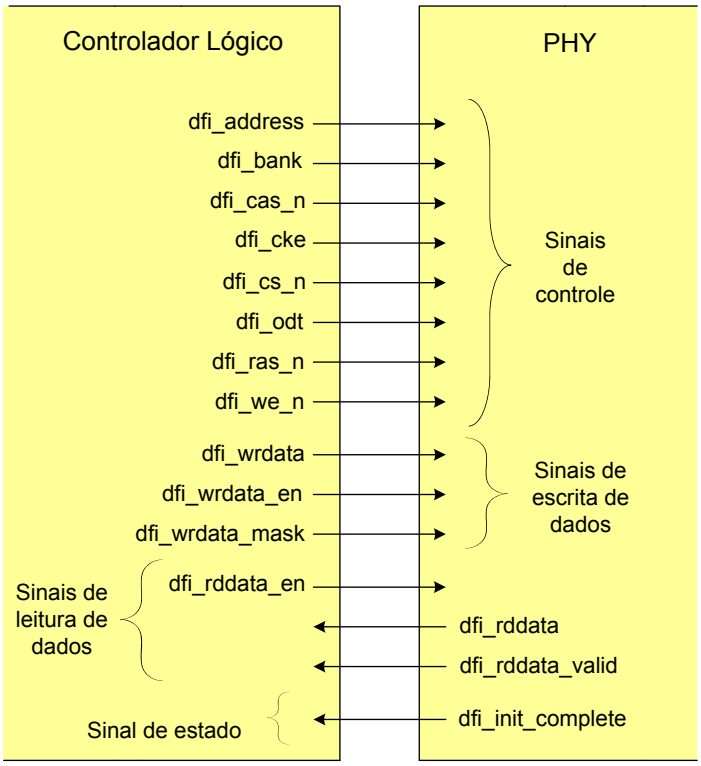

Figura 5.48: Sinais da interface DFI implementada.

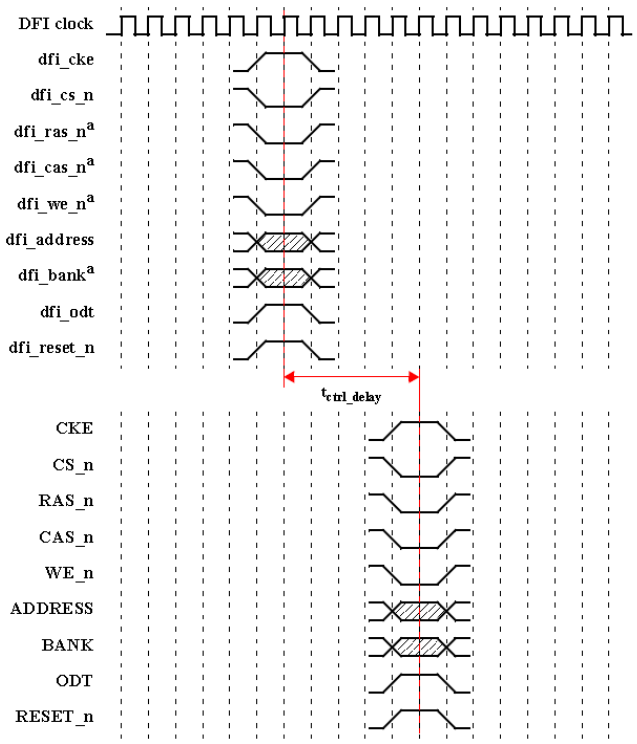

Figura 5.49: Parâmetro t_{ctrl_delay}. (DDR PHY INTERFACE (DFI) SPECIFICATION 2.1.1, 2010)

O sinal $dfi\_init\_complete$ não tem um parâmetro de tempo, mas limita aos outros sinais conforme a Figura 5.52 mostra.

Tabela 5.15: Tabela de sinais na interface DFI entre o bloco do controlador lógico e a camada física ou PHY.

Nome do sinal	Fonte	Bits	Parâmetro de tempo
Grupo de sinais de controle			
dfi_address	Controlador Lógico	13	tctrl_delay
dfi_bank	Controlador Lógico	2	tctrl_delay
dfi_cas_n	Controlador Lógico	1	tctrl_delay
dfi_cke	Controlador Lógico	1	tctrl_delay
dfi_cs_n	Controlador Lógico	1	tctrl_delay
dfi_odt	Controlador Lógico	1	tctrl_delay
dfi_ras_n	Controlador Lógico	1	tctrl_delay
dfi_we_n	Controlador Lógico	1	tctrl_delay
Grupo de sinais de escritura de dados			
dfi_wrdata	Controlador Lógico	64	tphy_wrdata
dfi_wrdata_en	Controlador Lógico	1	tphy_wrlat
dfi_wrdata_mask	Controlador Lógico	8	tphy_wrdata
Grupo de sinais de leitura de dados			
dfi_rddata_en	Controlador Lógico	1	trddata_en, tphy_rdlat
dfi_rddata	PHY	64	tphy_rdlat
dfi_rddata_valid	PHY	1	tphy_rdlat
Grupo de sinais de estado			
dfi_init_complete	PHY	1	n/a

Desta forma, conclui-se a descrição de cada um dos módulos desenvolvidos neste trabalho e a seguir é apresentada a metodologia de verificação utilizada para validar o funcionamento destes módulos.

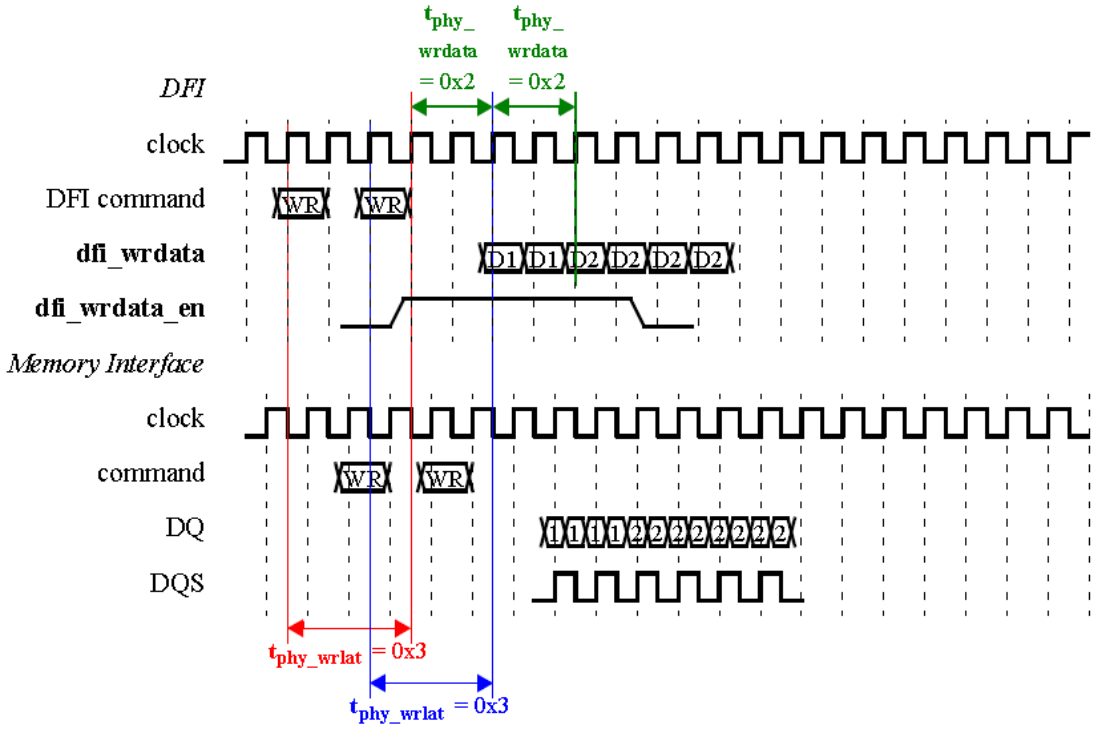

Figura 5.50: Parâmetros t_{phy_wrlat} e t_{phy_wrdata}. (DDR PHY INTERFACE (DFI) SPECIFICATION 2.1.1, 2010)

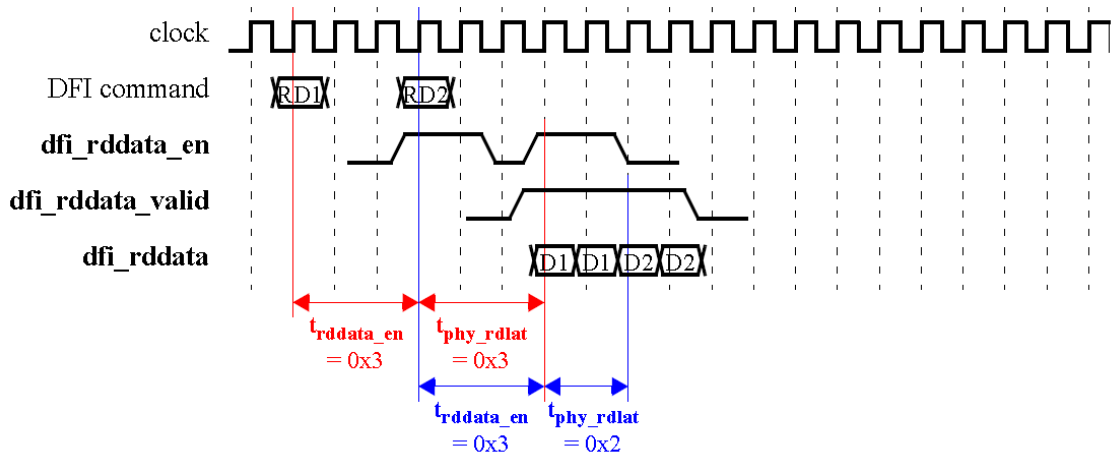

Figura 5.51: Parâmetros t_{rddata_en} e t_{phy_rdlat}. (DDR PHY INTERFACE (DFI) SPECIFICATION 2.1.1, 2010)

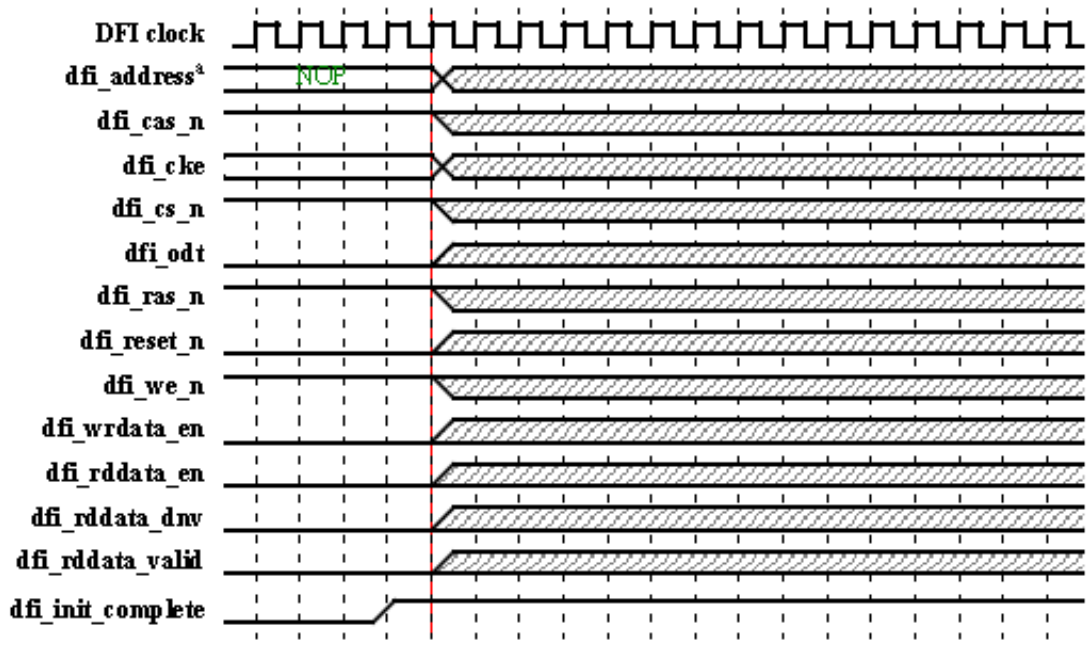

Figura 5.52: Sinal $dfi\_init\_complete$. (DDR PHY INTERFACE (DFI) SPECIFICATION 2.1.1, 2010)

6 METODOLOGIA DE VERIFICAÇÃO FUNCIONAL

Este capítulo apresenta a metodologia de verificação funcional utilizada neste trabalho. Baseia-se na metodologia VMM (*Verification Methodology Manual*) (BERGERON et al., 2005), mas não são utilizadas as classes propostas na metodologia VMM. São detalhado o ambiente de verificação desenvolvido e as métricas de cobertura utilizadas. Como contribuição, é proposto um método para sincronizar modelos de referência com o respectivo circuito e assim evitar o uso de um modelo com informação de tempo que aumentaria o tempo de simulação (TONFAT J.; NEUBERGER; REIS, 2011).

6.1 Introdução

O progresso nos processos de fabricação de semicondutores e as metodologia de projeto levaram à possibilidade de maiores e mais complexos projetos digitais. Este cenário, juntamente com tempos de projeto mais reduzidos, diminui a possibilidade de sucesso na primeira tentativa de fabricação.

Neste contexto, a verificação funcional do projeto adquire atenção relevante nos ambientes acadêmicos e industriais. Uma boa verificação de projeto significa que os projetos terão alta qualidade, e menos erros funcionais do circuito serão observados pelo usuário final.

A verificação de um circuito é o processo pelo qual certificar-se que a funcionalidade do projeto está de acordo com sua especificação. É o processo inverso do projeto, como é mostrado na Figura 6.1. Dois métodos são comumente utilizados: verificação funcional e formal.

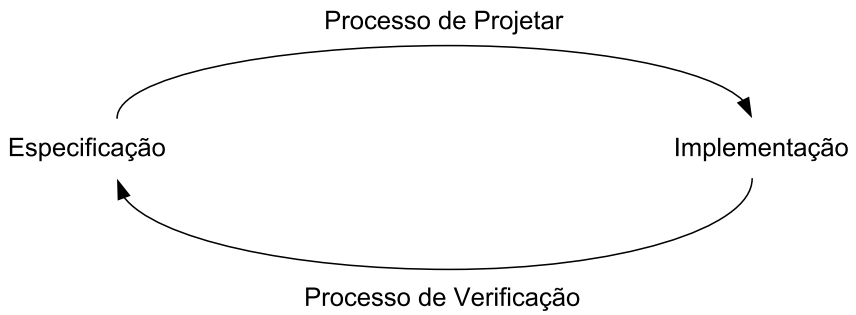

Figura 6.1: O processo de projetar contra o processo de verificar.

A verificação funcional ou verificação baseada em simulação é a técnica mais utilizada em aplicações industriais. Esta técnica é fácil de tratar, mas quase sempre é a fase

que consome mais recursos e que é o "gargalo" do fluxo do projeto. Alguns métodos de verificação formal também foram propostos, mas só aplicam-se a circuitos de baixa complexidade. Nesse cenário, poucas experiências de verificação funcional de sistemas de comunicações foram reportados em Strum, Chau e Romero (STRUM; CHAU; ROMERO, 2005). A maioria deles foram aplicados a processadores (WU et al., 2009) (GU et al., 2002).

A metodologia utilizada é a VMM. Esta metodologia utiliza estímulos restritos aleatórios. Este método permite encontrar erros no projeto de forma mais rápida que utilizando a metodologia de verificação direcionada, como é mostrado na Figura 6.2.

A metodologia de verificação direcionada consiste em criar vetores específicos para testar cada uma das funcionalidades do projeto. O tempo de verificação aumenta linearmente com a complexidade do projeto. Na metodologia VMM, precisa-se criar uma infraestrutura que permita inserir os estímulos e depois possa verificar o resultado. Isto leva um tempo, mas depois é compensado pela reutilização desta infraestrutura para verificar outros projetos.

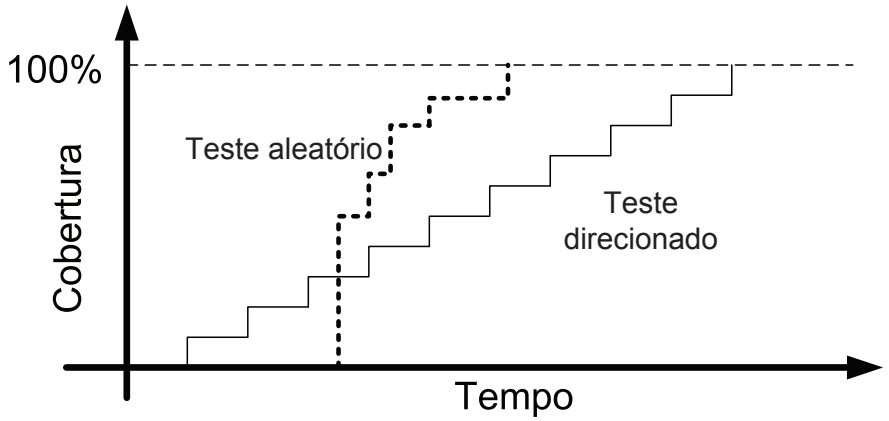

Figura 6.2: O progresso da verificação aleatória *versus* verificação direcionada (SPEAR, 2008).

O ambiente utilizado para verificar este bloco foi descrito utilizando a linguagem *SystemVerilog* (IEEE STANDARD FOR SYSTEMVERILOG–UNIFIED HARDWARE DESIGN, SPECIFICATION, AND VERIFICATION LANGUAGE - REDLINE, 2009). O *Systemverilog* é uma combinação de um HDL ou (*Hardware Description Language*) com um HVL ou (*Hardware Verification Language*). Criado como uma extensão do *Verilog* (IEEE STANDARD VERILOG HARDWARE DESCRIPTION LANGUAGE, 2001) , *SystemVerilog* utiliza técnicas de programação orientada a objetos para criar ambientes de verificação. A ferramenta utilizada para executar a verificação funcional foi o *Modelsim* 6.6b SE (MODELSIM SE USER'S MANUAL, 2010).

6.2 Arquitetura do ambiente de verificação

Na Figura 6.3 é apresentada a arquitetura completa do ambiente de verificação ou *testbench*. Esta arquitetura foi criada baseada nas recomendações encontradas em Spear (2008). Esta arquitetura foi desenvolvida dentro do projeto EDDASIC da UFRGS/DATACOM e nesta dissertação de mestrado apresenta-se a aplicação desta arquitetura aos

módulos desenvolvidos.

Este ambiente está organizado hierarquicamente em camadas. Essa estrutura ajuda a manter e reutilizá-lo com diferentes DUVs ou *Designs Under Verification*. As seguintes subseções detalham a funcionalidade de cada camada do ambiente de verificação.

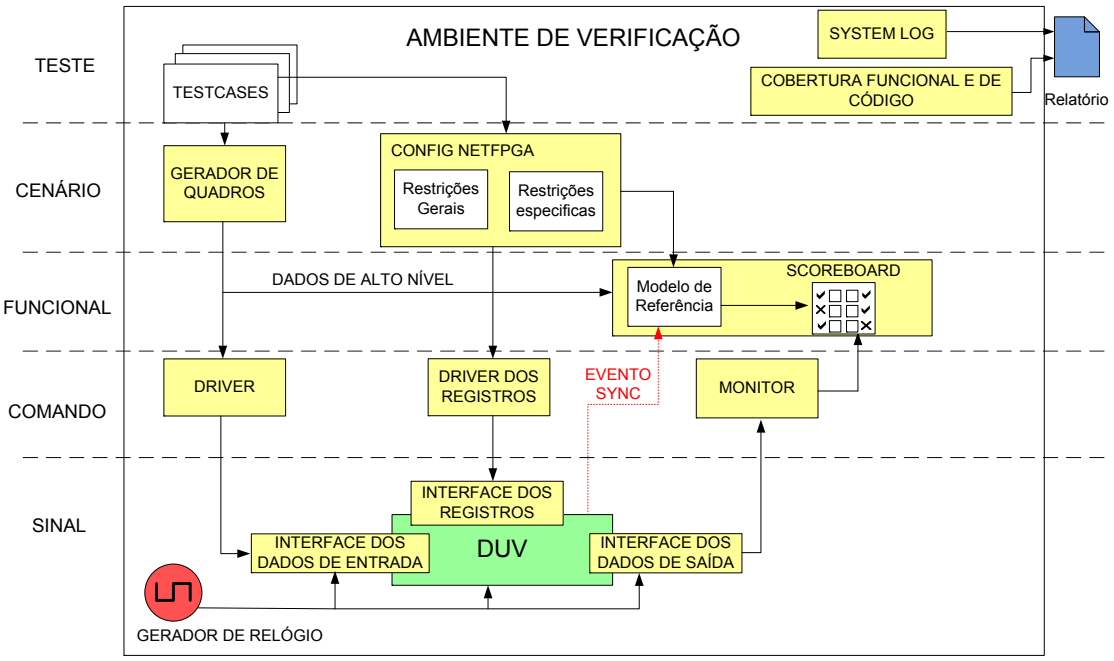

Figura 6.3: Arquitetura do ambiente de verificação descrito em SystemVerilog.

6.2.1 Camadas de sinal e comando

A camada de sinal implementa a interface entre o ambiente de verificação e o DUV. Como todos os módulos têm a mesma interface, estas interfaces são compartilhadas entre todos os ambientes de verificação desenvolvidos. Modificações mínimas são necessárias para os módulos "Árbitro de entrada" e "Filas de saída" porque multiplexam e demultiplexam, respectivamente, os quadros das portas *Ethernet*.

No caso do módulo "árbitro de entrada", as interfaces de entrada são expandidas para o número de portas de entrada do comutador *GigE*. E no caso do módulo de "filas de saída", as interfaces de saída também são expandidas.

A camada de comando implementa as funções dos *drivers* e do monitor. O *driver* recebe dados de alto nível e os transforma em sinais, que são enviados para o DUV através das interfaces. Outro *driver* (*driver* dos registradores) recebe a configuração dos registradores do módulo e cria os sinais para configurar o DUV.

O monitor implementa a função complementar. Ele recebe os sinais gerados pelo DUV e compõe um conjunto de dados de alto nível que serão usados para comparar com os resultados do modelo de referência.

Na Figura 6.4 é mostrada a hierarquia das classes que implementam os *drivers*, monitor e gerador de quadros. Todos derivam da classe `basic_transactor`. Um *transactor* é feito por um simples laço que recebe dados, os transforma e depois os encaminha para o bloco seguinte . A classe `netfpga_dp_in_transactor` implementa o *driver* de dados. A classe `netfpga_dp_out_transactor` implementa o monitor de dados. A classe `netfpga_reg_transactor` implementa o *driver* dos registradores.

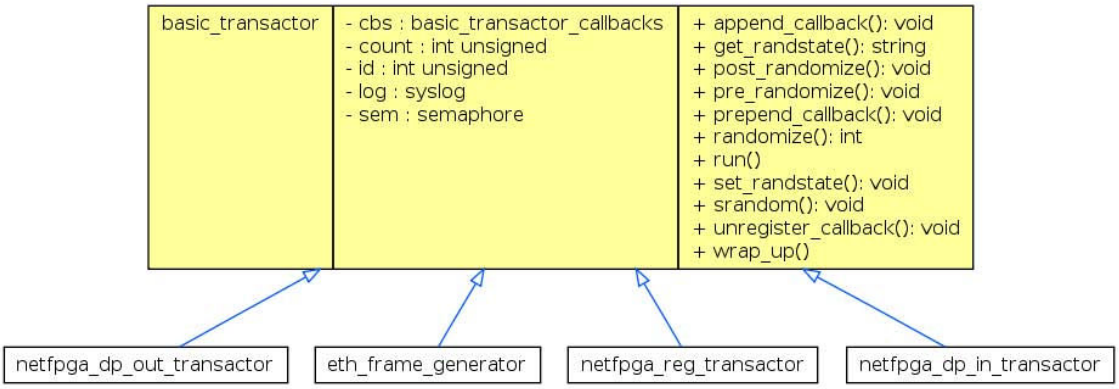

Figura 6.4: Hierarquia de classes dos *drivers* de dados e registradores, monitor e gerador de quadros.

Por último, a classe `eth_frame_generator` implementa o gerador de quadros. O objeto criado por esta classe não recebe os dados de um bloco superior, por isso, ele os cria aleatoriamente segundo a configuração recebida.

6.2.2 Camada funcional

A camada funcional implementa o "*scoreboard*". Na literatura, tem-se diferentes definições para o *scoreboard*. Neste trabalho, o *scoreboard* inclui o modelo de referência, a estrutura de dados que contém os resultados do modelo de referência, e a função de comparação de saída.

O modelo de referência recebe dados de alto nível e calcula o resultado esperado para o circuito. Este resultado é armazenado em uma estrutura de dados. Quando o monitor recebe um resultado gerado pelo DUV, este será comparado com os resultados armazenados que são gerados pelo modelo de referência. Se a comparação for bem-sucedida, então o circuito produziu o resultado esperado, caso contrário, um relatório de erro é gerado.

A função de comparação de saída depende muito do módulo que está sendo verificado. Em alguns casos, o que interessa é que o valor seja igual ao produzido pelo modelo e, em outros casos, é importante também a ordem na qual aparecem esses valores.

Na Figura 6.5 são apresentadas as classes necessárias para implementar o *scoreboard*. Nessa Figura, apresenta-se a classe que implementa o *scoreboard* do módulo "Marcador de quadros". Todos os *scoreboards* derivam da classe `basic_scoreboard`.

6.2.3 Modificações feitas ao modelo de referência (TONFAT J.; NEUBERGER; REIS, 2011)

Como foi mencionado no inicio deste capítulo, o modelo de referência utilizado é diferente dos modelos tradicionais (*Transaction-level models* ou TLMs). A seguir é apresentado o mecanismo usado para sincronizar o modelo de referência com o DUV.

Quando é mencionado o uso de modelos em nível de transação (TLM), a tendência é relacioná-los com a linguagem de modelagem de alto nível de *hardware*, *SystemC*, como mostrado em You, Oh e Song (2009). Nesta dissertação, todos os modelos em nível de transação (TLMs) foram escritos em *SystemVerilog*. Como explicado em Bergeron (2006), não há nenhuma razão técnica para que um modelo descrito no mesmo nível de abstração execute mais rápido se ele é descrito em outra linguagem como *SystemC*.

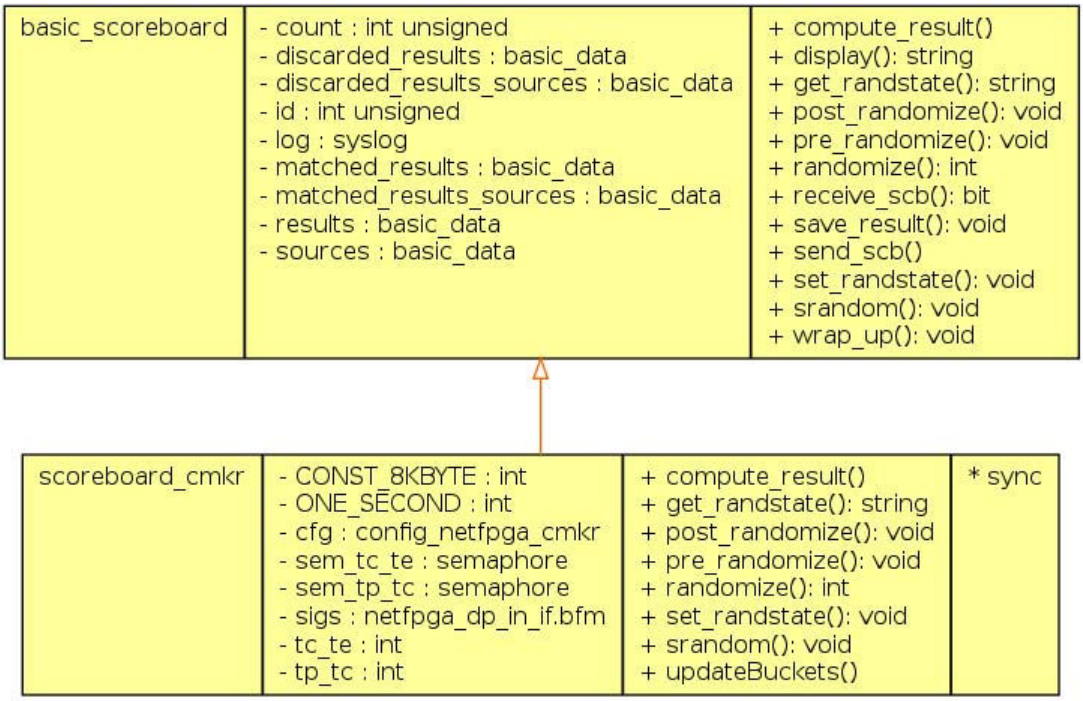

Figura 6.5: Hierarquia de classes utilizadas para gerar o *scoreboard*.

SystemVerilog e *SystemC* pertencem ao mesmo nível de abstração.

Numa primeira abordagem, para modelar o comportamento do circuito, um modelo em nível de transação tradicional foi criado para cada um dos módulos. Esta abordagem mostrou que, se a latência do circuito não é levada em conta, alguns resultados esperados são diferentes dos resultados do modelo em RTL. Essas diferenças aparecem porque estes circuitos têm uma relação estreita entre o tempo e a funcionalidade, de modo que o modelo de referência clássico não pode ser usado.

Numa segunda abordagem, tentou-se modelar a latência do circuito, sem alterar a característica de nível de transação do modelo de referência. Mas, a complexidade e a variedade dos modos de operação nestes módulos levaram a uma modelagem complexa, que reduz a confiabilidade do modelo de referência.

A aproximação final para modelar a funcionalidade do circuito foi sincronizar o cálculo do resultado esperado com alguns pontos-chave (estados) no modelo RTL. Outras abordagens para resolver o problema da precisão do modelo em ambientes mais complexos, como por exemplo em SoCs (*system-on-chips*), são apresentados em Cornet, Maraninchi e Maillet-Contoz (2008) e Salimi-Khaligh e Radetzki (2008).

Essa sincronização foi realizada com um dos recursos da linguagem *SystemVerilog*. Os “eventos” em *SystemVerilog* são objetos usados para acionar o cálculo do modelo de referência quando determinadas condições são cumpridas no modelo RTL. A observação do DUV é feita sem modificar o código original do modelo RTL. O modelo de referência continua sendo um modelo em nível de transação simples e o resultado esperado é computado no tempo de execução do mesmo modelo RTL.

6.2.4 Camada de cenário

A camada de cenário gera a configuração de cada um dos módulos no ambiente de verificação. Esta configuração define os parâmetros do DUV, como o tempo de envelhecimento no motor de classificação de nível 2 ou a taxa de dados PIR no módulo marcador de quadros. Esta configuração, conhecida como *testcase*, também é enviada para o modelo de referência para calcular o resultado esperado.

Com base nos parâmetros gerados para este *testcase*, o gerador criará dados aleatórios restritos (quadros *Ethernet*), que irão estimular o DUV. Dependendo do módulo que é verificado, o gerador irá criar diferentes tipos de quadros. Por exemplo, no ambiente criado para o módulo "árbitro de entrada", os quadros gerados têm portas de entrada diferentes porque o módulo utiliza este dado para processar os quadros. No caso do módulo de "filas de saída", a porta de saída é mais interessante porque o módulo utiliza este dado para encaminhar os quadros para a fila de saída respectiva. Neste caso, o gerador não precisa gerar portas de entrada diferentes porque este módulo não utiliza esse dado.

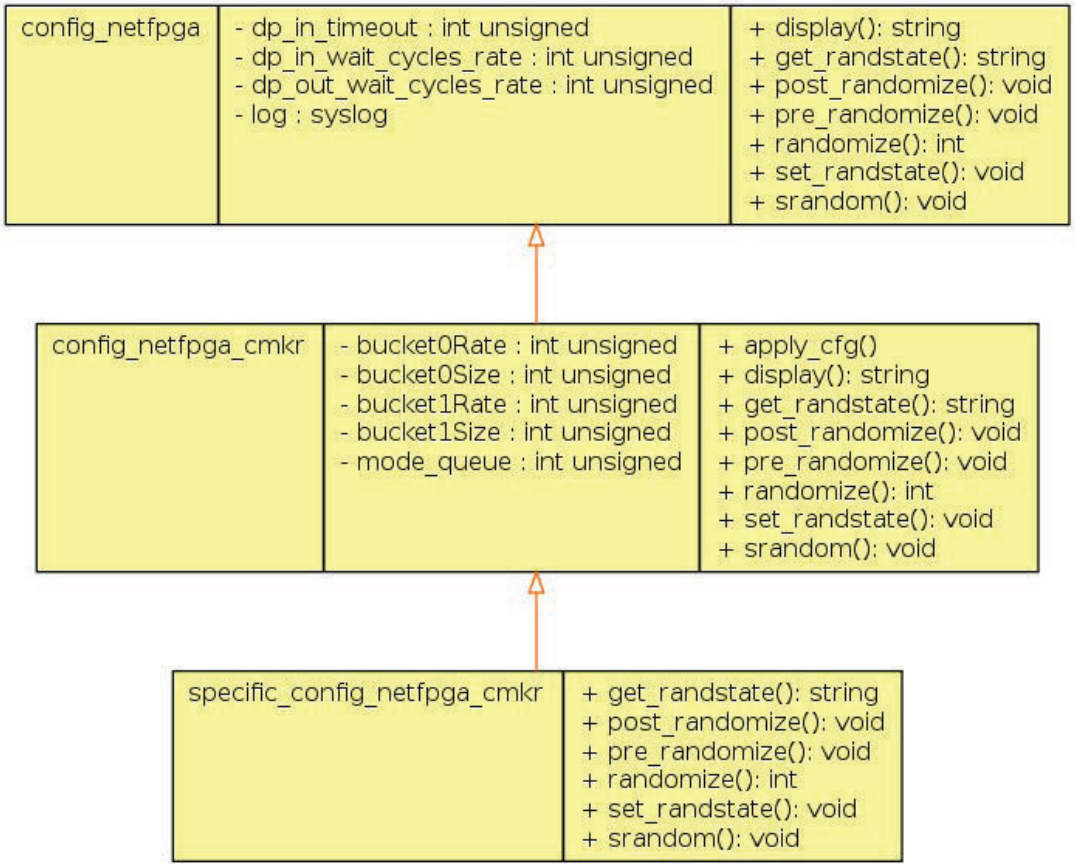

Figura 6.6: Hierarquia de classes utilizadas para gerar a configuração do DUV. Nesta Figura, o DUV é o "marcador de quadros".

Na Figura 6.6 é mostrada a hierarquia de classes para gerar a configuração do DUV. Todas as configurações derivam da classe *config_netfpga*. Esta classe contém dados como a porcentagem de tempo que a entrada dos módulos fica indisponível (sem dados).

6.2.5 Camada de teste

A camada de teste inclui os *testcases* que serão aplicados ao DUV. A maioria deles serão gerados aleatoriamente e, alguns casos direcionados, foram adicionados. Um exemplo de caso direcionado é a adição de configurações ruins nos registradores. Estes casos direcionados são necessários para verificar a resposta do módulo em situações incomuns que são possíveis que aconteçam. Esta camada irá controlar a execução da simulação atual e gerenciar os relatórios de cada função no ambiente de verificação.

Esta camada define a tolerância de erro, e classifica o comportamento inesperado com diferentes níveis de eventos. Os eventos são classificados em seis categorias diferentes, começando com os eventos de informação, passando por eventos de depuração (*debug*), eventos normais, eventos de advertência e de erro, e terminando com eventos fatais.

Um número máximo de erros é definido para cada *testcase*, esse recurso permitirá à simulação continuar funcionando na presença de erros e irá ajudar a encontrar mais erros no DUV. Um evento fatal irá necessariamente parar a simulação, pois este tipo de eventos torna inviável a execução do *testcase* atual.

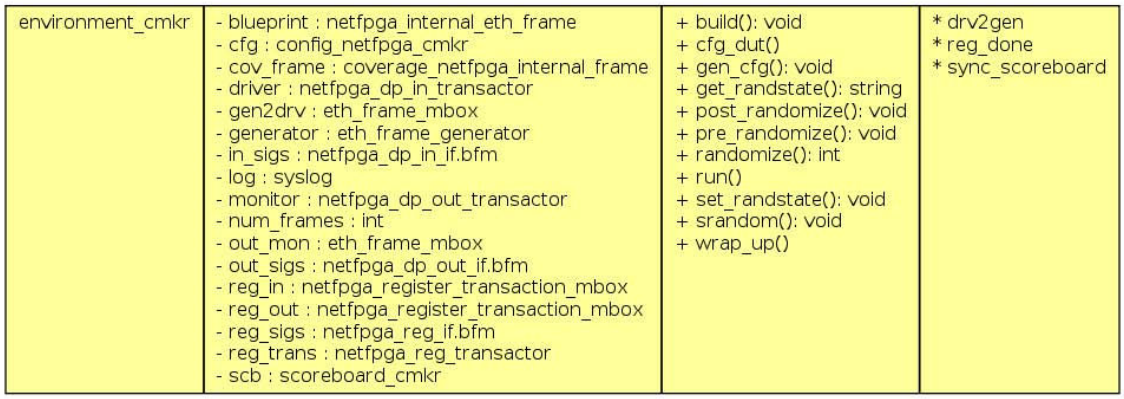

Figura 6.7: Classe Ambiente: Implementa todo o ambiente de verificação.

Na Figura 6.7 é mostrada a classe que implementa os ambientes de verificação de cada um dos módulos. É possível apreciar que esta classe possui os outros elementos do ambiente de verificação como o gerador, *drivers*, monitor e *scoreboard*.

6.3 Geração de estímulos restritos

O DUV precisa de estímulos para poder analisar seu funcionamento. Os estímulos gerados, no caso destes quatro módulos verificados, são quadros Ethernet que possuem o formato da plataforma NetFPGA. Cada um dos módulos recebe diferentes tipos de quadros e em diferentes quantidades, para testar cada uma das funcionalidades dos módulos.

Estes estímulos não podem ser totalmente aleatórios porque devem cumprir com os formatos definidos. Para conseguir isto, são definidas restrições específicas que permitem a geração de estímulos aleatórios, válidos para o formato definido.

A hierarquia de classes para definir os dois tipos de estímulos que o DUV precisa: os quadros *Ethernet* e a configuração do módulo, é mostrada na Figura 6.8. A classe `eth_frame` implementa um quadro *Ethernet* padrão. A classe `netfpga_internal_eth_frame` implementa um quadro *Ethernet*, mas com o formato NetFPGA. A classe `specific_netfpga_internal_eth_frame` permite adicionar algumas restrições para o *testcase* atual. Esta classe ajuda na depuração dos erros do DUV.

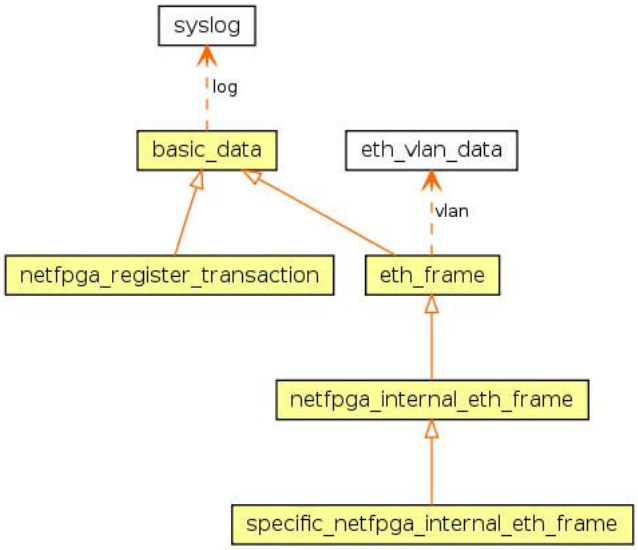

Figura 6.8: Hierarquia de classes para definir os quadros Ethernet e o conteúdo dos registradores de configuração.

6.4 Etapas da simulação

A simulação possui três etapas primárias: construção (*build*), execução (*run*) e revisão (*wrap-up*). Estas etapas são executadas em sequência, visando a sequência correta de operação do ambiente de verificação como um todo.

Na Figura 6.9 é mostrado o código da primeira etapa da simulação: Construção. Esta etapa inclui a criação do ambiente de verificação, a inicialização (*reset*) do DUV, a definição da configuração do DUV e a definição do tipo de estímulo que será inserido.

Na Figura 6.10 são mostradas as etapas de execução e revisão. Na etapa de execução começa a injeção de estímulos no DUV. Depois se espera até a finalização da injeção. Na etapa de revisão são gerados os relatórios finais da simulação.

Adicionalmente foi criado um *script* (ver apêndice A) em linguagem Tcl (*Tool Command language*) para controlar a execução de vários *testcases*, acumulando a informação de cobertura funcional e de código.

6.5 Métricas de cobertura

A qualidade da verificação funcional é avaliada utilizando duas métricas: a cobertura de código e a cobertura funcional. Na Figura 6.11 é mostrada a relação dos dois tipos de cobertura. Ambas as coberturas são necessárias para ter uma maior certeza da correta verificação do circuito.

A cobertura de código ou do tipo estrutural inclui: cobertura de linha (*statement coverage*), cobertura de máquinas de estado (*FSM coverage*), cobertura de saltos (*branch coverage*) e cobertura de variáveis (*toggle coverage*). Geralmente a ferramenta de simulação faz a análise do código e gera os respectivos relatórios.

A cobertura funcional é a porcentagem de funções do bloco que foram exercitadas na simulação. Os elementos funcionais que vão ser verificados tem que ser definidos utilizando a sintaxe da linguagem. Na Figura 6.12 é apresentada a hierarquia de classes para criar a cobertura funcional. Na Figura 6.13 é mostrado um exemplo de definição de

```
105
106   syslog log;
107   basic_data fr;
108   specific_netfpga_internal_eth_frame gen_fr;
109   environment_cmkr env;
110   specific_config_netfpga_cmkr cfg;
111   event trigger_compute_scoreboard;
112   int unsigned normal_exit;
113
129      log = new("test_color_marker", "Program");
130
131      gen_fr = new;
132
133      cfg = new;
134
135      // creates the environment
136      env = new(tb_top_color_marker.dp_in_if,
137                tb_top_color_marker.dp_out_if,
138                tb_top_color_marker.reg_if,
139                trigger_compute_scoreboard,
140                NUM_PACKETS,
141                cfg);
142      env.blueprint = gen_fr;
143      env.gen_cfg();
144      env.build();
145
146      reset_dut(5);
147
148      env.cfg_dut();
```

Primeira Etapa: Build

Figura 6.9: Primeira etapa da simulação: construção ou *build*.

um grupo de cobertura funcional. Nessa Figura é definida a cobertura do processamento de quadros com cabeçalho IP e VLAN junto com os diferentes tamanhos que o quadro pode ter.

6.6 Caso de análise: Módulo filas de saída

A verificação do módulo filas de saída foi a mais complexa, pois envolvia vários blocos. A verificação foi realizada em três etapas.

A primeira etapa envolvia a verificação somente da lógica das filas de saída. Isto foi possível porque o restante dos blocos foram modelados como um TLM. A Figura 6.14 mostra a primeira etapa da verificação. O modelo desenvolvido é menos complexo que o RTL original, portanto possui menos probabilidade de erro na codificação.

A segunda etapa visa a verificação da interface entre o controlador da memória DDR2 e o módulo das filas de saída. Para esta etapa foi modelado o controlador da memória DDR2. Na Figura 6.15 é mostrado o ambiente de verificação.

A última etapa consiste na verificação do controlador da memória DDR2. O único modelo utilizado foi o modelo fornecido pelo fabricante da memória SDRAM. Na Figura 6.16 é apresentado o ambiente de verificação.

Nas três etapas utilizaram-se os mesmos estímulos. Estes propiciaram uma alta porcentagem de cobertura funcional e do código. No capítulo seguinte são apresentados os

```
150     fork
151        forever begin
152           wait(tb_top_color_marker.color_marker_dut.state == 3'h1 || tb_top_color_marker.color_marker_dut.state == 3'h2);
153           -> trigger_compute_scoreboard;
154           wait(tb_top_color_marker.color_marker_dut.state == 3'h0);
155        end
156     join_none
157
158     env.run();
159
160     repeat(NUM_PACKETS) begin
161        env.out_mon.get(fr);
162     end
163
164     #100ns;
165
166     env.wrap_up();
167     #100ns;
168     // Static method from syslog class
169     syslog::report();
170     normal_exit = 1;
171     $stop;
```

Segunda Etapa: Run

Terceira Etapa: Wrap-up

Figura 6.10: Segunda e terceira etapa da simulação: *Run* e *Wrap-up*.

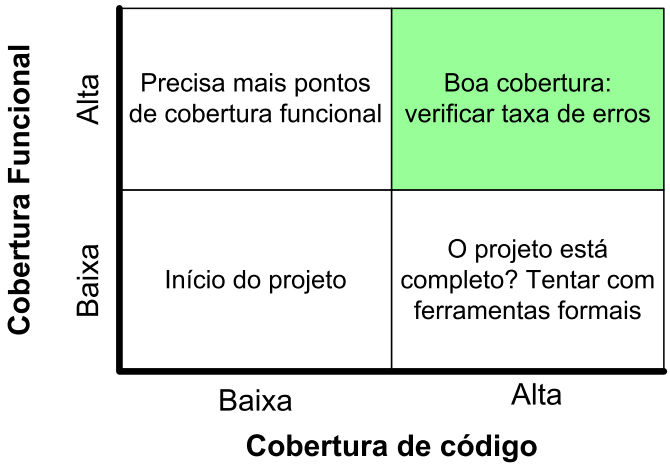

Figura 6.11: Relação da cobertura de código e funcional (SPEAR, 2008).

resultados da implementação e da verificação funcional.

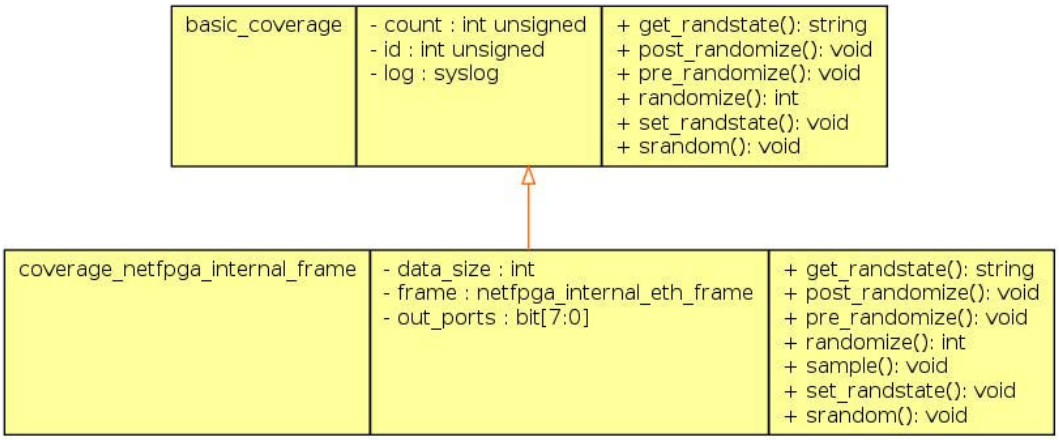

Figura 6.12: Hierarquia de classes para criar a cobertura funcional.

```
204  // this coverage makes a cross between IP, VLAN and data size
205  covergroup cov_type_length (int min_d_size, int max_d_size, int max_jumbo);
206     IP_frame: coverpoint frame.is_ip {
207        // these are the bins for frames with len_typ field equal to 0x8000 (IP)
208        bins IP_frame_yes = {1};
209        bins IP_frame_no = {0};
210        option.weight = 0; // Don't contribute to the total coverage metric
211     }
212     VLAN_frame: coverpoint frame.is_vlan {
213        // these are the bins for frames with len_typ field equal to 0x8100 (VLAN)
214        bins VLAN_frame_yes = {1};
215        bins VLAN_frame_no = {0};
216        option.weight = 0; // Don't contribute to the total coverage metric
217     }
218     length: coverpoint data_size {
219        // these bins are for the interesting lengths of frame
220        bins under_min_size = {[15:min_d_size-1]}; // 15: 6 bytes of src addr, 6 bytes of dst addr, 2 bytes of typ/len, 1 bytes of data
221        bins min_data_size = {min_d_size};
222        bins normal_data_size = {[min_d_size+1:max_d_size-1]};
223        bins max_data_size = {max_d_size};
224        bins vlan_size = {[max_d_size+1:max_d_size+3]};
225        bins max_vlan_size = {max_d_size+4};
226        bins jumbo_size   = {[max_d_size+5:max_jumbo-1]};
227        bins max_jumbo_size = {max_jumbo};
228        bins other          = default;
229        option.weight = 0; // Don't contribute to the total coverage metric
230     }
231     IP_VLAN_data_size: cross IP_frame, VLAN_frame, length {
232        option.weight = 2 * 2 * 2;
233     }
234  endgroup: cov_type_length
```

Figura 6.13: Exemplo de um grupo de cobertura funcional definida em SystemVerilog.

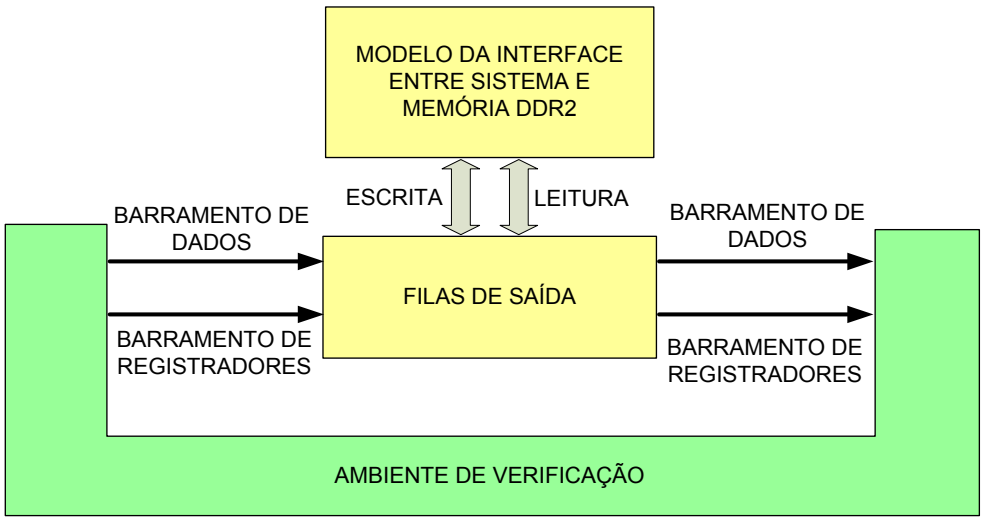

Figura 6.14: Primeira etapa: verificação da lógica das filas de saída.

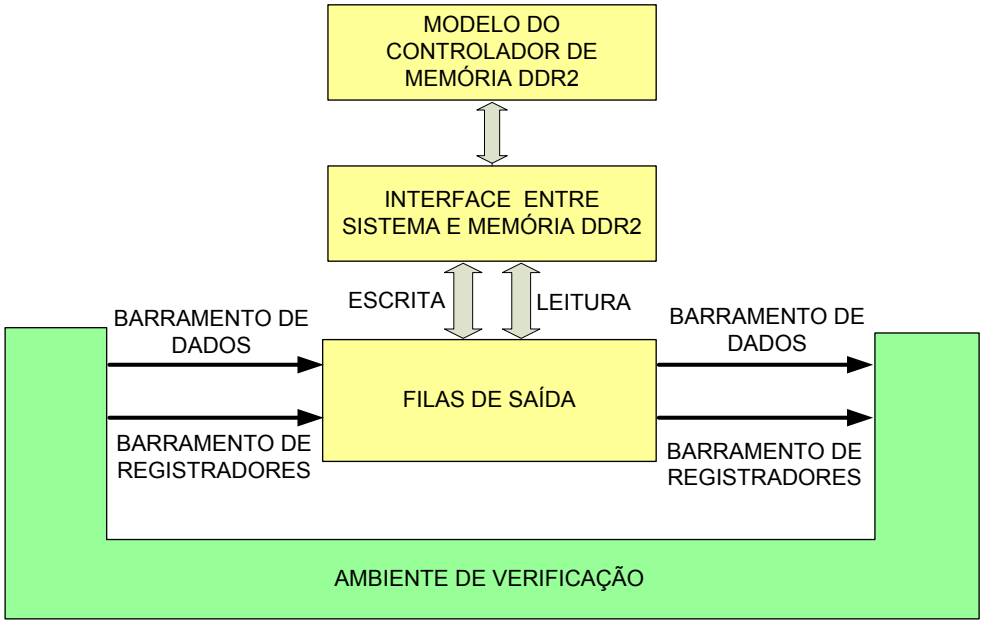

Figura 6.15: Segunda etapa: verificação da interface entre o sistema e o controlador DDR2 SDRAM.

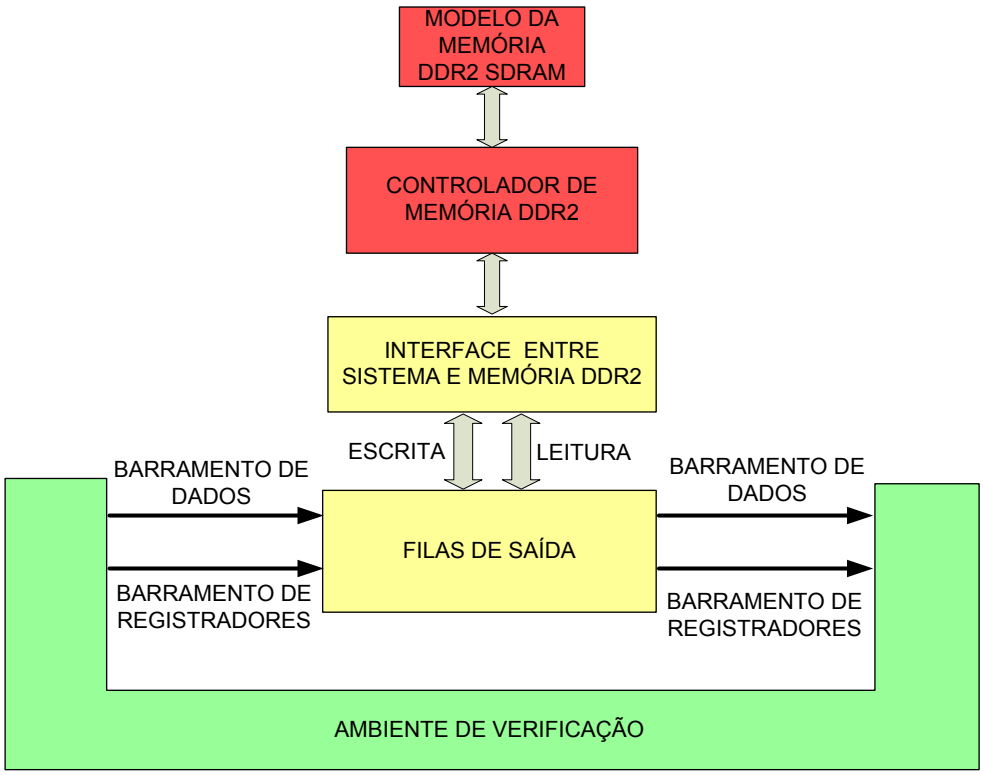

Figura 6.16: Terceira etapa: verificação do controlador DDR2 SDRAM.

7 ANÁLISE DOS RESULTADOS E COMPARAÇÕES

Este capítulo apresenta os resultados obtidos na verificação funcional dos quatro módulos apresentados neste trabalho. Também são apresentados os resultados da implementação física em ASIC destes quatro módulos. Aqui são apresentados os leiautes de cada um dos módulos e, adicionalmente, são feitas comparações com trabalhos relacionados encontrados na literatura.

7.1 Resultados da Verificação Funcional

Na Tabela 7.1 são apresentados os resultados da verificação funcional dos quatro módulos. Os resultados obtidos foram calculados pela própria ferramenta de simulação, Modelsim 6.6b (MODELSIM SE USER'S MANUAL, 2010). As simulações foram feitas em um servidor com um processador Intel Xeon @ 2.5 GHz com 32GB de memória RAM. No caso do módulo das filas de saída, os resultados da verificação apresentados devem-se à ultima etapa da verificação deste modulo, mostrado na Figura 6.16 do capítulo anterior.

Tabela 7.1: Tabela de resultados da verificação funcional dos quatro módulos.

Módulo	# estímulos para cada *testcase*	# *Testcases*	Tempo de simulação 1 (horas)	Tempo de simulação 2 (horas)	Cobertura Funcional (%)	Cobertura de código (%)
Árbitro de entrada	1600x8	1000	10.28*	8.8	94.44	91.70
Pesquisador da porta de saída	10000	1000	93.83*	76.66	99.60	90.18
Marcador de quadros	10000	1000	54.8	35.18	99.21	85.71
Filas de saída	10000	1000	64.68*	48.52	91.74	86.57

* = Estes valores foram aproximados tomando como referência o tempo de execução promedio de 1 *testcase* .

Algumas características comuns podem ser extraídas. Uma delas é a alta porcentagem de cobertura funcional em cada um dos quatro módulos. Estes resultados permitem a obtenção de uma medida quantitativa da qualidade da verificação. É importante mencionar que a cobertura funcional definida para cada um dos módulos está configurada de acordo a sua respectiva especificação. Os resultados de cobertura de código mostram que os *testcases* podem ser melhorados para aumentar a confiabilidade da verificação.

A informação mais importante concentra-se nas colunas do tempo de simulação. Na primeira coluna do tempo de simulação, o modelo de referência foi substituído pelo mesmo modelo RTL do circuito. O objetivo aqui é saber o tempo de simulação quando o modelo de referência é do tipo *cycle-accurate*. Como observação, o tempo da simulação médio de um *testcase* foi calculado utilizando o resultado de 30 simulações. A segunda coluna de tempo de simulação, mostra o tempo de simulação para o modelo de referência

proposto no Capítulo 6 que possui sincronia em tempo com o modelo RTL. O objetivo foi demonstrar que, embora o modelo de referência tenha a mesma resposta em tempo que o modelo RTL, o tempo de simulação é muito menor em comparação à utilização de um modelo com informação de tempo tal como o modelo RTL. Nesta tabela é mostrada uma clara melhoria no tempo de execução, com uma média de 23,37% de melhora para estes quatro módulos. Estes resultados confirmam que devido ao uso de um modelo em nível de transação, o tempo de execução da simulação é menor.

7.2 Resultados da Implementação Física

A seguir são apresentados os resultados da implementação física em ASIC dos quatro módulos. Segundo a plataforma NetFPGA, estes quatro módulos formam o "*user datapath*" e implementam as funções principais do comutador. São apresentados principalmente os resultados de área e potência de cada módulo. Para calcular a área foi utilizado o parâmetro de portas lógicas equivalentes. Este parâmetro é calculado dividindo a área total das células pela área de uma porta *NAND* de duas entradas de tamanho X1. O cálculo da potência foi feito com a ferramenta de análise embutida na ferramenta de síntese física SoC Encounter (ENCOUNTER USER GUIDE, 2009). A frequência utilizada para realizar o cálculo de potência foi 125 MHz.

Os leiautes apresentados a seguir não possuem anel de *pads* porque estes blocos são o núcleo funcional de comutador *Gigabit Ethernet*. Os módulos apresentados neste trabalho não interagem diretamente com os sinais das portas *Gigabit Ethernet*. Entre estes módulos e as portas *Gigabit Ethernet* estão os blocos *Gigabit Ethernet* MAC. Estes blocos foram projetados em Tomás (2009) para FPGA. Durante o convênio do projeto EDDA-SIC - UFRGS/DATACOM, foi feita a adaptação destes blocos para a implementação em ASIC. Mas foram encontrados problemas para realizar a síntese física deste bloco, especificamente na etapa de verificação formal entre o RTL original e o *netlist* gerado pela ferramenta de síntese lógica. Devido a estes problemas, este bloco não foi integrado junto aos módulos desenvolvidos neste trabalho e portanto não foi utilizado o anel de *pads*.

7.2.1 Árbitro de entrada

Na Figura 7.1 é mostrado o leiaute do módulo Árbitro de entrada e a Tabela 7.2 apresenta suas características. Este módulo representa 2% da área do *user datapath* e 8% da sua potência.

Tabela 7.2: Características do módulo Árbitro de entrada.

Tecnologia	TSMC 180 nm 6 Metal Layers 1.8V
Área	745 x 794 μm^2
Portas Lógicas	35,47 K
Frequência	125 MHz
Potência	60.23 mW

7.2.2 Pesquisador da porta de saída

Na Tabela 7.3 são apresentados os resultados de potência e área obtidos para as duas propostas apresentadas. A diferença que existe em potência e área entre as duas propostas não é muito significativa, porém o incremento no desempenho da segunda proposta chega

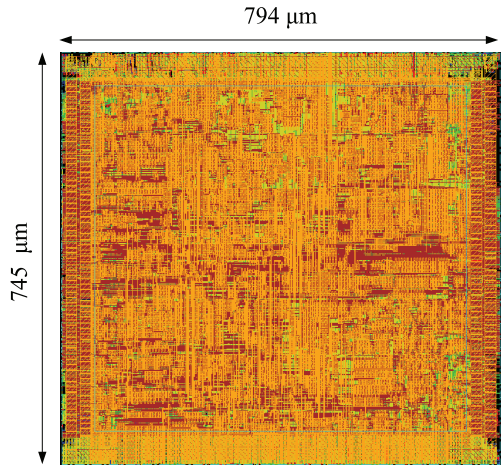

Figura 7.1: Leiaute do módulo Árbitro de entrada.

a ser 175% melhor. Este incremento deve-se principalmente às mudanças feitas no nível arquitetural dos blocos internos deste módulo, apresentados na seção 5.2.

Tabela 7.3: Comparação dos resultados do módulo Pesquisador da porta de saída.

	Consumo de Potência (mW)	Área
	Freq = 125 Mhz.	(portas lógicas equivalentes)
Primeira Proposta		
Motor de classificação	53,24	40127
Árbitro SRAM	6,1	3070
Total	59,34	43197
Segunda Proposta		
Motor de classificação	88,52	34733
Árbitro SRAM	6,68	3155
Total	95,2	37888

Na Tabela 7.4 apresenta-se uma comparação em termos de desempenho com respeito a outros trabalhos encontrados na literatura. A coluna de largura de banda refere-se à quantidade de quadros que motor de pesquisa pode processar o . Este valor é calculado a partir da quantidade de ciclos necessários para processar-se um quadro e a frequência máxima que atinge o circuito. Os resultados mostram que Papaefstathiou e Papaefstathiou (2006) possui a maior largura de banda, mas é importante observar que nesse trabalho a largura de banda está condicionada ao tamanho da tabela de endereços MAC. Se a tabela de endereços MAC fosse maior aos 128K entradas, a segunda proposta teria um desempenho melhor. No caso da segunda proposta, a largura de banda não depende do tamanho da tabela de endereços MAC. Cabe a observação também que Papaefstathiou e Papaefstathiou (2006), usa uma tecnologia de 130 nm que contribui para a obtenção de uma maior frequência de operação.

Na Figura 7.2 é mostrado o leiaute da segunda proposta do módulo Pesquisador da porta de saída e a Tabela 7.5 apresenta suas características. Este módulo representa 3% da área do *user datapath* e 11% da sua potência.

Tabela 7.4: Tabela de comparação de resultados com outros trabalhos na literatura.

Solução	Frequência de operação Max. (Mhz)	Largura de Banda (Gbps)	Tecnologia
Primeira Proposta	250	15.27	TSMC 180nm
Segunda Proposta	500	42	TSMC 180nm
(LAU et al., 2003)	125	22	180nm
(MISHRA et al., 2003)		10	180nm
(PAPAEFSTATHIOU; PAPAEFSTATHIOU, 2006)	400	103.5	UMC 130nm

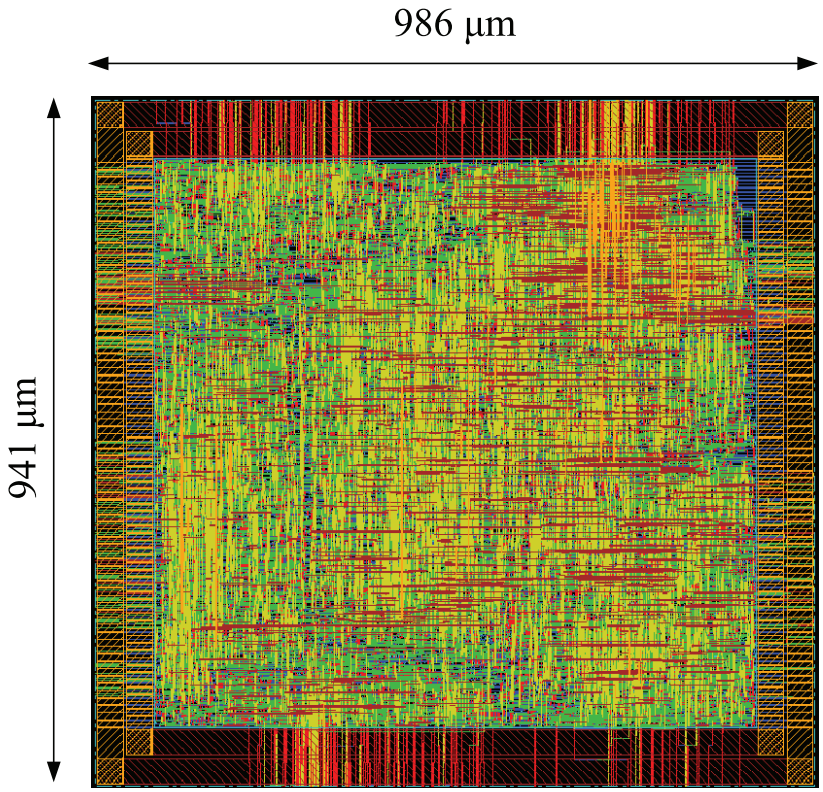

Figura 7.2: Leiaute da segunda proposta do módulo Pesquisador da porta de saída.

Tabela 7.5: Características do módulo Pesquisador da porta de saída.

Tecnologia	TSMC 180 nm 6 Metal Layers 1.8V
Área	986 x 941 μm^2
Portas Lógicas	48,26 K
Frequência	125 MHz
Potência	88,52 mW

Na Figura 7.3 é mostrada a distribuição de área do módulo Pesquisador da porta de saída. O motor de classificação representa 66% da área do módulo.

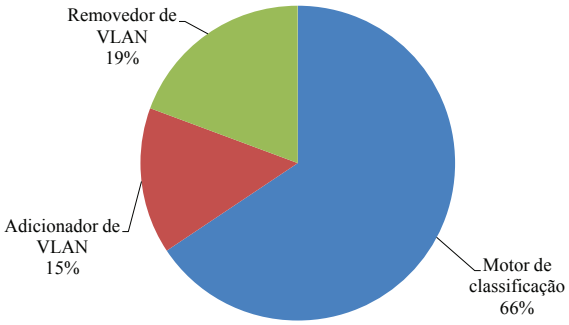

Figura 7.3: Distribuição de área do módulo Pesquisador da porta de saída.

7.2.3 Marcador de quadros

Na Figura 7.4 é mostrado o leiaute da segunda proposta do módulo Marcador de quadros e a Tabela 7.6 apresenta suas características. Este módulo representa 3% da área do *user datapath* e 11% da sua potência.

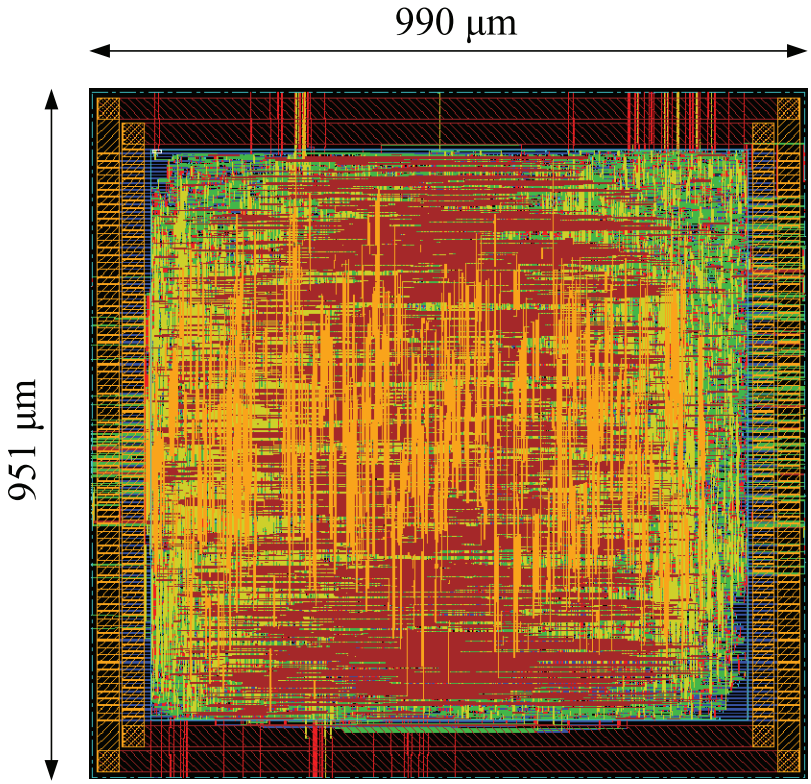

Figura 7.4: Leiaute do módulo Marcador de quadros.

7.2.4 Filas de saída

Na Figura 7.5 é mostrado o leiaute do módulo Filas de saída e a Tabela 7.7 apresenta suas características. As memórias SRAM são responsáveis por aproximadamente 52,3% da área e 47% da potência deste módulo. Para realizar o cálculo da potência dos blocos de memória SRAM foi utilizada a ferramenta HP-CACTI (WILTON; JOUPPI, 1996) e para

Tabela 7.6: Características do módulo Marcador de quadros.

Tecnologia	TSMC 180 nm 6 Metal Layers 1.8V
Área	990 x 951 μm^2
Portas Lógicas	52,48 K
Frequência	125 MHz
Potência	84,87 mW

o cálculo da potência das celúlas foi utilizada a ferramenta da Cadence Soc Encounter (ENCOUNTER USER GUIDE, 2009). Este módulo representa 92% da área do *user datapath* e 70% da sua potência.

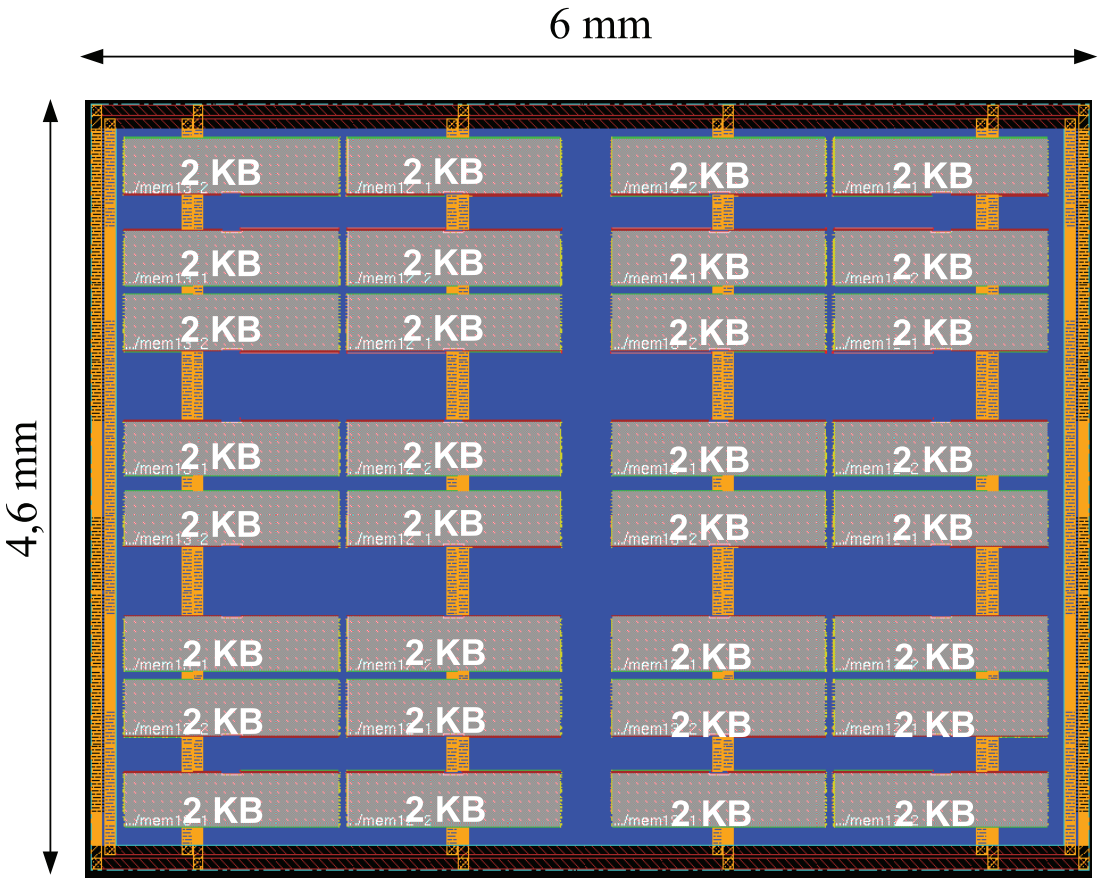

Figura 7.5: Leiaute do módulo Filas de saída.

Tabela 7.7: Características do módulo Filas de saída.

Tecnologia	TSMC 180 nm 6 Metal Layers 1.8V
Área	6 x 4,6 mm^2
Portas Lógicas	115,1 K
Memória	64 KB
Frequência	125 MHz
Potência	542.97 mW

Finalmente, nas Figuras 7.6 e 7.7 são apresentadas as distribuições de área e potência dos quatro módulos. Os resultados mostram que o módulo das filas de saída ocupa a maior área e consome mais potência, devido principalmente às memórias SRAM.

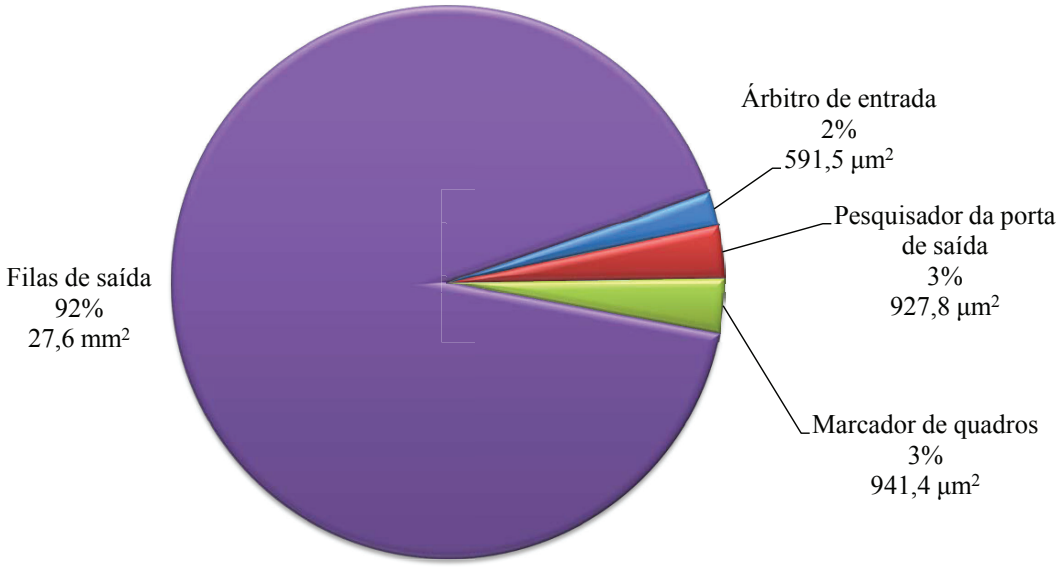

Figura 7.6: Distribuição de área dos quatro módulos.

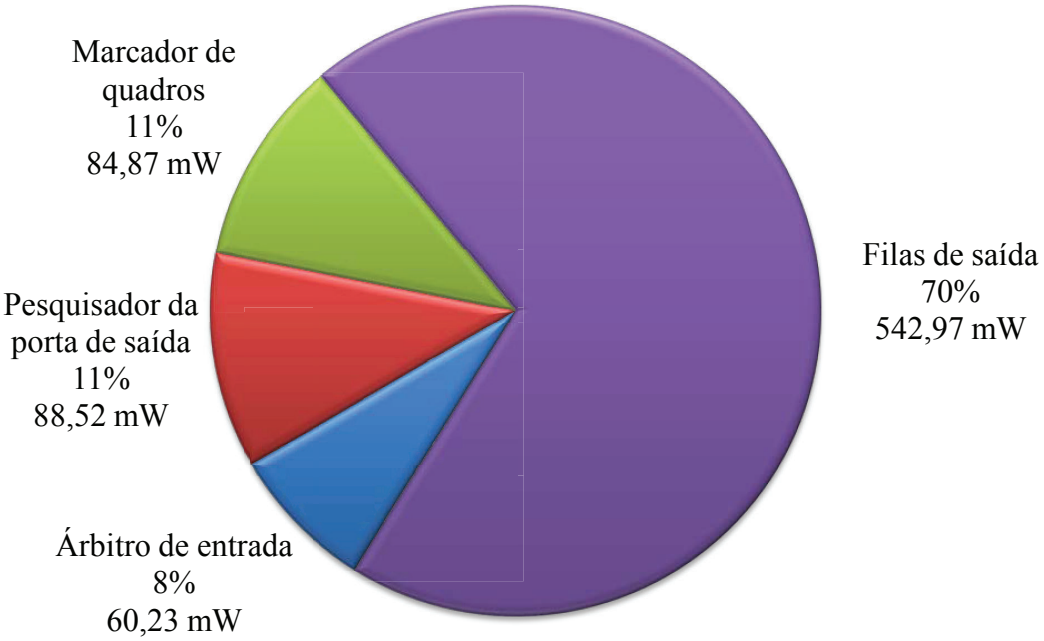

Figura 7.7: Distribuição de potência dos quatro módulos.

8 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou o projeto, a verificação funcional e a implementação em ASIC de quatro módulos funcionais para um comutador *Gigabit Ethernet*.

O primeiro módulo apresentado foi o Árbitro de entrada. Originalmente, este módulo utilizava um algoritmo *Round Robin*, que não permitia fixar a largura de banda de dados (*data bandwidth*), e foi substituído pelo algoritmo *Deficit Round Robin*, que não apresenta a mesma limitação. Isso foi demonstrado comparando-se ambos os algoritmos com um mesmo *testbench*.

O segundo módulo apresentado é o pesquisador da porta de saída. Este é o módulo mais importante, porque implementa a principal função do comutador, que é encontrar a porta de saída dos quadros que ingressam. O principal bloco deste módulo é o motor de classificação. Duas arquiteturas para o motor de classificação foram apresentadas, sendo que as mesmas atendem com as funções definidas no padrão IEEE 802.1D (IEEE Std 802.1D, 2004). Segundo os resultados da síntese em ASIC, a segunda proposta poderia processar até 42 *Gbps*, atingindo uma frequência de 500 MHz.

A diferença principal da segunda proposta reside na forma e método de acesso à memória externa SRAM, por meio de um protocolo de comunicação otimizado entre o motor de classificação e o árbitro SRAM, assim como também a reestruturação do árbitro SRAM que permite diminuir o tempo de espera numa operação de leitura ou escrita na memória. Esta diferença incrementa o desempenho em 175% na segunda proposta, sem apresentar um incremento significativo em área e potência.

Outra característica importante da segunda proposta é que a largura de banda dos dados é constante em relação ao tamanho da tabela de endereços. Esta característica torna este módulo importante para o desenvolvimento de comutadores de núcleo de empresas (*enterprise core*), onde o tamanho da tabela de endereços é maior (mais de 100 mil entradas).

O terceiro módulo implementa um marcador de quadros segundo os RFCs 2697, 2698 e 4115. Este módulo é a primeira etapa de um mecanismo de priorização de quadros, que permite oferecer uma qualidade de serviço (QoS) diferenciada para cada porta do comutador.

O último módulo implementa as filas de saída do comutador. As filas de saída do comutador são implementadas na memória SDRAM DDR2. Este módulo teve que ser adaptado para sua síntese em ASIC. A mudança principal encontra-se na utilização de blocos de memória SRAM gerados para a síntese em ASIC. No bloco original, recursos do FPGA (blocos BRAM) são utilizados.

A verificação funcional dos quatro módulos funcionais de um comutador *Gigabit Ethernet* foi apresentada. A metodologia utilizada permite encontrar erros no projeto que só apareceriam na presença de estímulos aleatórios. A abordagem utilizada neste tra-

balho é mais eficiente, em tempo, para atingir o objetivo de cobertura que outros métodos mais simples, como o teste direcionado (*direct-test*).

A verificação funcional dos módulos é muito importante e complexa. Sua complexidade deve-se à falta de um modelo de referência algorítmico. Portanto, a criação de um conjunto de quadros aleatórios que conseguisse estimular todas as funções dos blocos foi uma tarefa desafiadora para o desenvolvimento deste trabalho.

Estas experiências mostraram que a utilização de modelos de referência clássicos (que não possuem informação temporal), não é adequada para este tipo de circuitos, onde o tempo influencia na resposta do circuito. Um método simples para sincronizar o modelo de referência com o DUV foi proposto. Este método mantém os benefícios dos modelos em nível de transação, bem como prevê corretamente a saída do circuito.

8.1 Trabalhos Futuros

No aspecto das funcionalidades do comutador, algumas características adicionais deveriam ser implementadas para que este dispositivo tenha a flexibilidade de equipamentos comerciais. Uma delas é a utilização de SLAs mais complexas das que foram implementadas neste trabalho, como o CoS (*Class of Service*), QoS (*Quality of Service*). Se este dispositivo for utilizado na rede do provedor, então outras características devem ser adicionadas como o *multiple tagging* ou o *MAC-in-MAC*.

Durante o desenvolvimento dos módulos, os desenvolvedores da plataforma NetFPGA mudaram o sub-módulo que implementa os registradores. Na nova versão, os registradores possuem uma descrição HDL mais genérica. Como uma futura modificação, sugere-se a troca do sub-módulo dos registradores dos módulos "Árbitro de entrada", "Pesquisador da porta de saída" e "Marcador de quadros". O módulo "Filas de saída" já possui a nova versão do sub-módulo dos registradores.

Propõem-se as seguintes modificações ao módulo "Filas de saída":

- Modificar o módulo para que o mesmo seja capaz de detectar a sobreposição dos espaços de memória DRAM de cada fila de saída. E em uma segunda etapa acrescentar-se a utilização de algum método que permita a colocação dinâmica de quadros na memória DRAM;

- Adaptar o módulo para que este seja capaz de processar quadros com tamanho Jumbo. Atualmente, devido às limitações do módulo (utiliza memórias de 2KB), somente é possível processar quadros de até 2KB em conformidade com o padrão IEEE 802.3 (IEEE Std 802.3, 2005). Isto incrementaria consideravelmente o tamanho do circuito, porque seria necessário incrementar o tamanho das memórias de cada fila do comutador;

- Modificar o módulo de forma a possibilitar a utilização da informação do módulo "marcador de quadros", com o objetivo de colocação dos quadros nas filas de saída com prioridade aplicada.

Uma próxima etapa no desenvolvimento do motor de classificação, é habilitar o processamento de protocolos de nível 3 como o IP (*Internet Protocol*). Para conseguir isto, é necessário implementar um algoritmo de busca como o algoritmo LPM (*Longest Prefix Match*).

Na etapa de verificação funcional, deve-se adicionar um mecanismo que permita a redução de estímulos redundantes. Um estímulo redundante é definido como aquele estímulo que não fornece nenhum incremento na cobertura funcional. A eliminação dos estímulos redundantes reduz o tempo de simulação. Algumas soluções são apresentadas em Strum, Chau e Romero (2005) e em Bose (2001).

No fluxo de implementação em ASIC propõe-se a otimização da lógica, através do uso de redes de transistores (REIS, 2011), e com isto possibilitar a diminuição do consumo do circuito, devido principalmente à redução do número de transistores quando é usado uma rede de transistores em vez de células lógicas de uma biblioteca de células.

Finalmente propõe-se o projeto do *chip* completo, com a adição os blocos *Gigabit Ethernet* MAC e os outros blocos que formam parte da plataforma NetFPGA, como o controlador do barramento PCI, possibilitando a configuração dos registradores de cada um dos módulos projetados neste trabalho, tendo-se assim, um comutador *Gigabit Ethernet* completo.

REFERÊNCIAS

ABOUL-MAGD, O.; RABIE, S. **A Differentiated Service Two-Rate, Three-Color Marker with Efficient Handling of in-Profile Traffic**. Disponível em: `http://www.ietf.org/rfc/rfc4115.txt`. Acesso em: 23 mai 2009, RFC 4115 (Informational).

BERGERON, J. **Writing Testbenches using SystemVerilog**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

BERGERON, J. et al. **Verification Methodology Manual for SystemVerilog**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

BONATTO, A. C. **Núcleos de Interface de Memória DDR SDRAM para Sistemas-em-chip**. 2009. Mestrado em Engenheria Elétrica — UFRGS.

BOSE, M. et al. A Genetic Approach to Automatic Bias Generation for Biased Random Instruction Generation. In: CONGRESS ON EVOLUTIONARY COMPUTATION, 2001. **Proceedings...** [S.l.: s.n.], 2001. v.1, p.442 –448 vol. 1.

BRAY, A. **Webinar**: Ethernet OAM and Demarcation Devices: Boxing Clever. [S.l.]: ADVA Optical Networking, 2006.

CORNET, J.; MARANINCHI, F.; MAILLET-CONTOZ, L. A Method for the Efficient Development of Timed and Untimed Transaction-Level Models of Systems-on-Chip. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE 2008. **Proceedings...** [S.l.: s.n.], 2008. p.9 –14.

DDR PHY Interface (DFI) Specification 2.1.1. [S.l.]: Denali, 2010. Disponível em: `http://www.ddr-phy.org`. Acesso em: 17 jul 2009.

ENCOUNTER Conformal Equivalence Checking User Guide. [S.l.]: Cadence, 2008. Disponível em: `http://www.cadence.com`. Acesso em: 10 nov 2009.

ENCOUNTER User Guide. [S.l.]: Cadence, 2009. Disponível em: `http://www.cadence.com`. Acesso em: 10 nov 2009.

FRAZIER, H.; JOHNSON, H. Gigabit Ethernet: from 100 to 1,000 mbps. **IEEE Internet Computing**, Piscataway, NJ, USA, v.3, n.1, p.24–31, 1999.

GEORGE, M. **DDR2 SDRAM Memory Interface for Virtex-II Pro FPGAs**. [S.l.]: Xilinx, 2007. Disponível em: `http://www.xilinx.com/support/documentation/application_notes/xapp549.pdf`. Acesso em: 15 set 2009.

GU, Z. et al. Functional Verification Methodology of a 32-bit RISC Microprocessor. In: IEEE 2002 INTERNATIONAL CONFERENCE ON COMMUNICATIONS, CIRCUITS AND SYSTEMS AND WEST SINO EXPOSITIONS. **Proceedings...** [S.l.: s.n.], 2002. v.2, p.1454 – 1457 vol.2.

HEINANEN, J.; GUERIN, R. **A Single Rate Three Color Marker**. Disponível em: `http://www.ietf.org/rfc/rfc2697.txt`. Acesso em: 23 mai 2009, RFC 2697 (Informational).

HEINANEN, J.; GUERIN, R. **A Two Rate Three Color Marker**. Disponível em: `http://www.ietf.org/rfc/rfc2698.txt`. Acesso em: 23 mai 2009, RFC 2698 (Informational).

HUNTLEY, C.; ANTONOVA, G.; GUINAND, P. Effect of Hash Collisions on the Performance of LAN Switching Devices and Networks. In: IEEE CONFERENCE ON LOCAL COMPUTER NETWORKS, 2006, 31. **Proceedings...** [S.l.: s.n.], 2006. p.280 –284.

IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language - Redline. **IEEE Std 1800-2009 (Revision of IEEE Std1800-2005) - Redline**, [S.l.], p.1 –1346, 11 2009.

IEEE Standard Verilog Hardware Description Language. **IEEE Std 1364-2001**, [S.l.], p.1–856, 2001.

IEEE Std 802.1ad. IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks, Amendment 4: provider bridges. **(Amendment to IEEE Std 8021Q-2005)**, [S.l.], p.0 –60, 2006.

IEEE Std 802.1ah. IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks - Amendment 7: provider backbone bridges. **(Amendment to IEEE Std 802.1Q-2005)**, [S.l.], p.C1 –109, 14 2008.

IEEE Std 802.1D. IEEE Standard for Local and Metropolitan Area Networks Media Access Control (MAC) Bridges. , [S.l.], p.269, 2004.

IEEE Std 802.1Q. IEEE Standard for Local and Metropolitan Area Networks Virtual Bridged Local Area Networks. , [S.l.], p.285, 2006.

IEEE Std 802.3. IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications. , [S.l.], v.Section5, p.1 –417, 2005.

ISO/IEC Standard 7498-1. Information Technology - Open System Interconnection - Basic Reference Model: The Basic Model. , [S.l.], 1994.

JAIN, R. A Comparison of Hashing Schemes for Address Lookup in Computer Networks. **IEEE Transactions on Communications**, [S.l.], v.40, n.10, p.1570 –1573, oct 1992.

LAU, M. et al. Gigabit Ethernet Switches Using a Shared Buffer Architecture. **IEEE Communications Magazine**, [S.l.], v.41, n.12, p.76 – 84, dec. 2003.

LUO, J. et al. Prototyping Fast, Simple, Secure Switches for Ethane. In: ANNUAL IEEE SYMPOSIUM ON HIGH-PERFORMANCE INTERCONNECTS, HOTI 2007, 15. **Proceedings...** [S.l.: s.n.], 2007. p.73 –82.

MCAULEY, A.; FRANCIS, P. Fast Routing Table Lookup Using CAMs. In: INFOCOM '93. TWELFTH ANNUAL JOINT CONFERENCE OF THE IEEE COMPUTER AND COMMUNICATIONS SOCIETIES. NETWORKING: FOUNDATION FOR THE FUTURE. IEEE. **Proceedings...** [S.l.: s.n.], 1993. p.1382 –1391 vol.3.

METCALFE, R. M.; BOGGS, D. R. Ethernet: distributed packet switching for local computer networks. **Commun. ACM**, New York, NY, USA, v.19, n.7, p.395–404, 1976.

MISHRA, S. et al. Wire-speed Traffic Management in Ethernet Switches. In: THE 2003 INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, ISCAS 2003. **Proceedings...** [S.l.: s.n.], 2003. v.2, p.II–105 – II–108 vol.2.

MODELSIM SE User's Manual. [S.l.]: Mentor Graphics, 2010. Disponível em: `http://www.model.com`. Acesso em: 10 nov 2009.

NAOUS, J. et al. NetFPGA: reusable router architecture for experimental research. In: PRESTO '08: ACM WORKSHOP ON PROGRAMMABLE ROUTERS FOR EXTENSIBLE SERVICES OF TOMORROW, New York, NY, USA. **Proceedings...** ACM, 2008. p.1–7.

NetFPGA. **NetFPGA User Guide**. Disponível em: `http://netfpga.org/foswiki/bin/view/NetFPGA/OneGig/Guide`. Acceso em: 03 jun 2009.

PAPAEFSTATHIOU, V.; PAPAEFSTATHIOU, I. A Hardware-Engine for Layer-2 Classification in Low-storage, Ultra High Bandwidth Environments. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE 2006. **Proceedings...** [S.l.: s.n.], 2006. v.2, p.1 –6.

REIS, R. Design Tools and Methods for Chip Physical Design. In: HüBNER, M.; BECKER, J. (Ed.). **Multiprocessor System-on-Chip**: Hardware Design and Tool Integration. [S.l.]: Springer Science, 2011. p.155–166. ISBN 978-1-4419-6459-5, DOI 10.1007/978-1-4419-6459-5.

ROCHOL, J. **5. Metro Ethernet (Carrier Ethernet)**. Curso Extensão Metro Ethernet - FATEC 2009.

SALIMI-KHALIGH, R.; RADETZKI, M. A Latency, Preemption and Data Transfer Accurate Adaptive Transaction Level Model for Efficient Simulation of Pipelined Buses. In: FORUM ON SPECIFICATION, VERIFICATION AND DESIGN LANGUAGES, FDL 2008. **Proceedings...** [S.l.: s.n.], 2008. p.37 –42.

SEIFERT, R.; EDWARDS, J. **The All-New Switch Book**: The Complete Guide to LAN Switching Technology. Hoboken, NJ: Wiley, 2008.

SHREEDHAR, M.; VARGHESE, G. Efficient Fair Queuing Using Deficit Round-Robin. **IEEE/ACM Transactions on Networking**, [S.l.], v.4, n.3, p.375 –385, June 1996.

SILVA, L. M. d. L. **Implementação Física de Arquiteturas de Hardware para a Decodificação de Vídeo Digital Segundo o Padrão H.264/AVC**. 2010. Dissertação (Mestrado em Ciência da Computação) — UFRGS.

SPEAR, C. **SystemVerilog for Verification, Second Edition**: A Guide to Learning the Testbench Language Features. [S.l.]: Springer Publishing Company, Incorporated, 2008.

STRUM, M.; CHAU, W. J.; ROMERO, E. Comparing Two Testbench Methods for Hierarchical Functional Verification of a Bluetooth Baseband Adaptor. In: THIRD IEEE/ACM/IFIP INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN AND SYSTEM SYNTHESIS, CODES+ISSS 2005. **Proceedings...** [S.l.: s.n.], 2005. p.327 –332.

TOMÁS, C. **Arquiteturas para um Dispositivo de Demarcação Ethernet**. 2009. Dissertação (Mestrado em Ciência da Computação) — UFRGS.

TONFAT J.; NEUBERGER, G.; REIS, R. Design and Verification of a Layer-2 Ethernet MAC Search Engine and Frame Marker for a Gigabit Ethernet Switch. In: SOUTH SYMPOSIUM ON MICROELECTRONICS (SIM), 25. **Proceedings...** SBC, 2010. p.201–204. ISSN: 2177-5176.

TONFAT J.; NEUBERGER, G.; REIS, R. Functional Verification of Logic Modules for a Gigabit Ethernet Switch. In: IEEE LATIN AMERICAN TEST WORKSHOP, LATW 2011., 12., Porto de Galinhas, Pernambuco, Brazil. **Proceedings...** IEEE, 2011.

TONFAT, J.; REIS, R. Design and Verification of a Layer-2 Ethernet MAC Classification Engine for a Gigabit Ethernet Switch. In: IEEE INTERNATIONAL CONFERENCE ON ELECTRONICS, CIRCUITS, AND SYSTEMS (ICECS), 17., Athens, Greece. **Proceedings...** IEEE, 2010. p.146–149. ISBN: 978-1-4244-8156-9.

TONFAT, J.; REIS, R. Diseño y Verificación de un Motor de Clasificación de Capa 2 MAC Ethernet para un Conmutador Gigabit Ethernet. In: XVII IBERCHIP WORKSHOP, Bogotá, Colombia. **Proceedings...** [S.l.: s.n.], 2011.

USING Encounter RTL Compiler. [S.l.]: Cadence, 2009. Disponível em: `http://www.cadence.com`. Acesso em: 10 nov 2009.

VASUDEVAN, S. **Effective Functional Verification**. [S.l.]: Springer, 2006.

VIRTEX-II Pro and Virtex-II Pro X Platform FPGAs: complete datasheet. [S.l.]: Xilinx, 2011. Disponível em: `http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf`. Acesso em: 15 set 2009.

WILTON, S.; JOUPPI, N. CACTI: an enhanced cache access and cycle time model. **IEEE Journal of Solid-State Circuits**, [S.l.], v.31, n.5, p.677 –688, May 1996.

WU, Y. et al. A Coverage-Driven Constraint Random-Based Functional Verification Method of Pipeline Unit. In: EIGHTH IEEE/ACIS INTERNATIONAL CONFERENCE ON COMPUTER AND INFORMATION SCIENCE, ICIS 2009. **Proceedings...** [S.l.: s.n.], 2009. p.1049 –1054.

XILINX Memory Interface Generator (MIG 007) User Guide DDR SDRAM / DDR2 SDRAM Compiler. [S.l.]: Xilinx, 2005. Disponível em: `http://www.xilinx.com`. Acesso em: 15 set 2009.

YOU, M.-K.; OH, Y.-J.; SONG, G.-Y. Implementation of a Hardware Functional Verification System Using SystemC Infrastructure. In: IEEE REGION 10 CONFERENCE TENCON 2009. **Proceedings...** [S.l.: s.n.], 2009. p.1 –5.

APÊNDICE A *SCRIPTS* TCL PARA O CONTROLE DA SIMULAÇÃO EM MODELSIM

A.1 *Script* para o módulo Marcador de quadros

```
puts {
  ModelSimSE 6.6b general compile script version 1.0
  Copyright (c) Jorge Tonfat 2010
}

vlib work
vmap work work

vlog -work work -lint +incdir+defines defines/eddasic_xml_defines.v
vlog -work work -lint +incdir+defines defines/cpci_defines.v
vlog -work work -lint +incdir+defines defines/global_defines.v
vlog -work work -lint +incdir+defines defines/NF_2.1_defines.v
vlog -work work -lint +incdir+defines defines/udp_defines.v

vlog -work work -lint +cover=bcesf +incdir+defines color_marker/color_marker_regs.v
vlog -work work -lint +cover color_marker/small_fifo.v
vlog -work work -lint +cover color_marker/fallthrough_small_fifo.v
vlog -work work -lint +cover=bcesf color_marker/pipeline_mult.v
vlog -work work -lint +cover color_marker/const_multiplier.v
vlog -work work -lint +cover +incdir+defines color_marker/color_marker.v

vlog -work work -lint +incdir+defines tb_top_color_marker.sv
vlog -work work -lint syslog.sv
vlog -work work -lint basic_data.sv
vlog -work work -lint basic_transactor.sv
vlog -work work -lint config_netfpga.sv
vlog -work work -lint eth_frame.sv
vlog -work work -lint netfpga_internal_eth_frame.sv
vlog -work work -lint +incdir+defines netfpga_register_transaction.sv
vlog -work work -lint netfpga_dp_in_if.sv
vlog -work work -lint netfpga_dp_out_if.sv
vlog -work work -lint +incdir+defines netfpga_reg_if.sv
vlog -work work -lint +incdir+defines netfpga_sram_reg_if.sv
vlog -work work -lint netfpga_dp_in_transactor.sv
vlog -work work -lint netfpga_dp_out_transactor.sv
vlog -work work -lint +incdir+defines netfpga_reg_transactor.sv
vlog -work work -lint eth_frame_generator.sv
vlog -work work -lint basic_scoreboard.sv
vlog -work work -lint +incdir+defines scoreboard_cmkr.sv
vlog -work work -lint basic_coverage.sv
vlog -work work -lint coverage_netfpga_internal_frame.sv
vlog -work work -lint +incdir+defines config_netfpga_cmkr.sv
vlog -work work -lint +incdir+defines environment_cmkr.sv
vlog -work work -lint +incdir+defines test_color_marker.sv

onbreak {resume}

set f [open start_time.txt w]
puts $f "Start time was [clock seconds]"
close $f

for {set x 0} {$x<1000} {incr x} {

    puts "Running iteration Number: $x"

    quietly set seed1 [expr {int(1 + 1000*rand())}]; # [1, 1000]
    quietly set num_packets 10000

    vsim -cvg63 -coverage -gSEED=$seed1 -gNUM_PACKETS=$num_packets -sv_seed $seed1 -voptargs="+acc"
         -permit_unmatched_virtual_intf -t ps work.test_color_marker work.tb_top_color_marker

    do wave_mti.do

    coverage attribute -test $x

    set SolveArrayResizeMax 0
    if {$x != 0} {
      exec rm -f cov_temp.ucdb
     }
```

```
69
70      run -all
71
72      if { [examine /test_color_marker/normal_exit] == 0} {
73         break
74      }
75
76      coverage save cov_temp.ucdb
77
78      quit -sim
79
80      if {$x == 0} {
81         exec cp cov_temp.ucdb cov_final.ucdb
82      } else {
83         vcover merge -totals -verbose -out cov_final.ucdb cov_temp.ucdb cov_final.ucdb
84      }
85   }
86
87   vcover report -html cov_final.ucdb
88
89   # How long since project began?
90   if {[file isfile start_time.txt] == 0} {
91     set f [open start_time.txt w]
92     puts $f "Start time was [clock seconds]"
93     close $f
94   } else {
95     set f [open start_time.txt r]
96     set line [gets $f]
97     close $f
98     regexp {\d+} $line start_time
99     set total_time [expr ([clock seconds]-$start_time)/60]
100    set f [open total_time.txt w]
101    puts $f "Project time is $total_time minutes"
102    close $f
103  }
104  # Compress the generated transcript
105  exec tar zcvf transcript.tar.gz transcript
```

A.2 *Script* para o módulo Árbitro de entrada

```
1   puts {
2     ModelSimSE 6.6b general compile script version 1.0
3     Copyright (c) Jorge Tonfat 2010
4   }
5
6   vlib work
7   vmap work work
8
9   vlog -work work -lint +incdir+defines defines/eddasic_xml_defines.v
10  vlog -work work -lint +incdir+defines defines/cpci_defines.v
11  vlog -work work -lint +incdir+defines defines/global_defines.v
12  vlog -work work -lint +incdir+defines defines/NF_2.1_defines.v
13  vlog -work work -lint +incdir+defines defines/udp_defines.v
14
15  vlog -work work -lint +cover +incdir+defines input_arb/in_arb_regs.v
16  vlog -work work -lint +cover input_arb/small_fifo.v
17  vlog -work work -lint +cover input_arb/fallthrough_small_fifo.v
18  vlog -work work -lint +cover +incdir+defines input_arb/input_arbiter.v
19
20  vlog -work work -lint syslog.sv
21  vlog -work work -lint basic_data.sv
22  vlog -work work -lint basic_transactor.sv
23  vlog -work work -lint config_netfpga.sv
24  vlog -work work -lint eth_frame.sv
25  vlog -work work -lint netfpga_internal_eth_frame.sv
26  vlog -work work -lint +incdir+defines netfpga_register_transaction.sv
27  vlog -work work -lint netfpga_dp_in_if.sv
28  vlog -work work -lint netfpga_dp_out_if.sv
29  vlog -work work -lint +incdir+defines netfpga_reg_if.sv
30  vlog -work work -lint +incdir+defines netfpga_sram_reg_if.sv
31  vlog -work work -lint netfpga_dp_in_transactor.sv
32  vlog -work work -lint netfpga_dp_out_transactor.sv
33  vlog -work work -lint +incdir+defines netfpga_reg_transactor.sv
34  vlog -work work -lint eth_frame_generator.sv
35  vlog -work work -lint basic_scoreboard.sv
36  vlog -work work -lint +incdir+defines scoreboard_inArb.sv
37  vlog -work work -lint basic_coverage.sv
38  vlog -work work -lint coverage_netfpga_internal_frame.sv
39  vlog -work work -lint +incdir+defines config_netfpga_inArb.sv
40  vlog -work work -lint +incdir+defines environment_inArb.sv
41  vlog -work work -lint +incdir+defines tb_top_input_arbiter.sv
42  vlog -work work -lint +incdir+defines test_input_arbiter.sv
43
44  onbreak {resume}
45
46  set f [open start_time.txt w]
47  puts $f "Start time was [clock seconds]"
48  close $f
49
50  for {set x 0} {$x<1000} {incr x} {
51     puts "Running iteration Number: $x"
52     quietly set seed1 [expr {int(1 + 1000*rand())}]; # [1, 1000]
53     quietly set num_packets 200
54     vsim -coverage -gSEED=$seed1 -sv_seed $seed1 -gNUM_PACKETS=$num_packets -voptargs="+acc"
           -permit_unmatched_virtual_intf -t ps work.tb_top_input_arbiter
55     do wave_mti.do
56     coverage attribute -test $x
```

```
57      set SolveArrayResizeMax 0
58      if {$x != 0} {
59        exec rm -f cov_temp.ucdb
60       }
61      run -all
62      if { [examine tb_top_input_arbiter/test1/normal_exit] == 0} {
63         break
64      }
65      coverage save cov_temp.ucdb
66      quit -sim
67      if {$x == 0} {
68         exec cp cov_temp.ucdb cov_final.ucdb
69      } else {
70         vcover merge -totals -verbose -out cov_final.ucdb cov_temp.ucdb cov_final.ucdb
71      }
72  }
73
74  vcover report -html cov_final.ucdb
75
76  # How long since project began?
77  if {[file isfile start_time.txt] == 0} {
78    set f [open start_time.txt w]
79    puts $f "Start time was [clock seconds]"
80    close $f
81  } else {
82    set f [open start_time.txt r]
83    set line [gets $f]
84    close $f
85    regexp {\d+} $line start_time
86    set total_time [expr ([clock seconds]-$start_time)/60]
87    set f [open total_time.txt w]
88    puts $f "Project time is $total_time minutes"
89    close $f
90  }
91  # Compress the generated transcript
92  exec tar zcvf transcript.tar.gz transcript
```

A.3 *Script* para o módulo Pesquisador da porta de saída

```
1   puts {
2     ModelSimSE 6.6b general compile script version 1.0
3     Copyright (c) Jorge Tonfat 2010
4   }
5
6   vlib work
7   vmap work work
8
9   vlog -work work -lint +incdir+defines defines/eddasic_xml_defines.v
10  vlog -work work -lint +incdir+defines defines/cpci_defines.v
11  vlog -work work -lint +incdir+defines defines/global_defines.v
12  vlog -work work -lint +incdir+defines defines/NF_2.1_defines.v
13  vlog -work work -lint +incdir+defines defines/udp_defines.v
14
15  vlog -work work -lint +cover +incdir+defines lookup/sram_arbiter.v
16  vlog -work work -lint +define+sb166 lookup/cy7c1370d.v
17
18  vlog -work work -lint +cover +incdir+defines lookup/ethernet_parser_64bit.v
19  vlog -work work -lint +cover lookup/crc_func.v
20  vlog -work work -lint +cover lookup/header_hash.v
21  vlog -work work -lint +cover lookup/exact_match_learning.v
22  vlog -work work -lint +cover lookup/aging.v
23  vlog -work work -lint +cover lookup/small_fifo.v
24  vlog -work work -lint +cover +incdir+defines lookup/op_lut_regs.v
25  vlog -work work -lint +cover +incdir+defines lookup/output_port_lookup.v
26
27  vlog -work work -lint +cover lookup/fallthrough_small_fifo.v
28  vlog -work work -lint +cover +incdir+defines lookup/vlan_adder.v
29  vlog -work work -lint +cover +incdir+defines lookup/vlan_remover.v
30
31  vlog -work work -lint +cover +incdir+defines lookup/output_port_lookup_plus_sram.v
32
33  vlog -work work -lint syslog.sv
34  vlog -work work -lint basic_data.sv
35  vlog -work work -lint basic_transactor.sv
36  vlog -work work -lint config_netfpga.sv
37  vlog -work work -lint eth_frame.sv
38  vlog -work work -lint netfpga_internal_eth_frame.sv
39  vlog -work work -lint +incdir+defines netfpga_register_transaction.sv
40  vlog -work work -lint +incdir+defines netfpga_sram_reg_transaction.sv
41  vlog -work work -lint netfpga_dp_in_if.sv
42  vlog -work work -lint netfpga_dp_out_if.sv
43  vlog -work work -lint +incdir+defines netfpga_reg_if.sv
44  vlog -work work -lint +incdir+defines netfpga_sram_reg_if.sv
45  vlog -work work -lint netfpga_dp_in_transactor.sv
46  vlog -work work -lint netfpga_dp_out_transactor.sv
47  vlog -work work -lint +incdir+defines netfpga_reg_transactor.sv
48  vlog -work work -lint +incdir+defines netfpga_sram_reg_transactor.sv
49  vlog -work work -lint eth_frame_generator.sv
50  vlog -work work -lint basic_scoreboard.sv
51  vlog -work work -lint +incdir+defines scoreboard_opl.sv
52  vlog -work work -lint basic_coverage.sv
53  vlog -work work -lint coverage_netfpga_internal_frame.sv
54  vlog -work work -lint +incdir+defines config_netfpga_opl.sv
55  vlog -work work -lint +incdir+defines environment_opl.sv
56  vlog -work work -lint +incdir+defines tb_top_output_port_lookup.sv
57
58  vlog -work work -lint +incdir+defines test_output_port_lookup.sv
```

```
59
60  onbreak {resume}
61
62  set f [open start_time.txt w]
63  puts $f "Start time was [clock seconds]"
64  close $f
65
66  for {set x 0} {$x<1000} {incr x} {
67     puts "Running iteration Number: $x"
68     quietly set seed1 [expr {int(1 + 1000*rand())}]; # [1, 1000]
69     quietly set num_packets 10000
70     vsim -cvg63 -coverage -gSEED=$seed1 -sv_seed $seed1 -gNUM_PACKETS=$num_packets  -voptargs="+acc"
           -permit_unmatched_virtual_intf -t ps work.tb_top_output_port_lookup
71     do wave_mti.do
72     coverage attribute -test $x
73     set SolveArrayResizeMax 0
74     if {$x != 0} {
75       exec rm -f cov_temp.ucdb
76      }
77     run -all
78     if { [examine /tb_top_output_port_lookup/test1/normal_exit] == 0} {
79        break
80     }
81     coverage save cov_temp.ucdb
82     quit -sim
83     if {$x == 0} {
84        exec cp cov_temp.ucdb cov_final.ucdb
85     } else {
86        vcover merge -totals -verbose -out cov_final.ucdb cov_temp.ucdb cov_final.ucdb
87     }
88  }
89
90  vcover report -html cov_final.ucdb
91
92  # How long since project began?
93  if {[file isfile start_time.txt] == 0} {
94    set f [open start_time.txt w]
95    puts $f "Start time was [clock seconds]"
96    close $f
97  } else {
98    set f [open start_time.txt r]
99    set line [gets $f]
100   close $f
101   regexp {\d+} $line start_time
102   set total_time [expr ([clock seconds]-$start_time)/60]
103   set f [open total_time.txt w]
104   puts $f "Project time is $total_time minutes"
105   close $f
106 }
107 # Compress the generated transcript
108 exec tar zcvf transcript.tar.gz transcript
```

A.4 *Script* para o módulo Filas de saída

```
1   puts {
2     ModelSimSE 6.6b general compile script version 1.0
3     Copyright (c) Jorge Tonfat 2010
4   }
5
6   ##vmap unisims_ver C:/Xilinx/10.1/ISE/verilog/mti_se/unisims_ver
7   vlib work
8   vmap work work
9
10  vlog -work work -lint +incdir+defines defines/eddasic_xml_defines.v
11  vlog -work work -lint +incdir+defines defines/cpci_defines.v
12  vlog -work work -lint +incdir+defines defines/global_defines.v
13  vlog -work work -lint +incdir+defines defines/NF_2.1_defines.v
14  vlog -work work -lint +incdir+defines defines/udp_defines.v
15
16  vlog -work work FIFOs/async_72x512_fifo/async_72x512_fifo.v
17  vlog -work work FIFOs/async_144x256_fifo/async_144x256_fifo.v
18
19  vlog -work work FIFOs/async_72x512_fifo/fifo12.v
20  vlog -work work FIFOs/async_72x512_fifo/fifomem12.v
21  vlog -work work FIFOs/async_72x512_fifo/rptr_empty12.v
22  vlog -work work FIFOs/async_72x512_fifo/sync_r2w12.v
23  vlog -work work FIFOs/async_72x512_fifo/sync_w2r12.v
24  vlog -work work FIFOs/async_72x512_fifo/wptr_full12.v
25
26  vlog -work work FIFOs/async_144x256_fifo/fifo13.v
27  vlog -work work FIFOs/async_144x256_fifo/fifomem13.v
28  vlog -work work FIFOs/async_144x256_fifo/rptr_empty13.v
29  vlog -work work FIFOs/async_144x256_fifo/sync_r2w13.v
30  vlog -work work FIFOs/async_144x256_fifo/sync_w2r13.v
31  vlog -work work FIFOs/async_144x256_fifo/wptr_full13.v
32
33  vlog -work work -lint -cover bcesf +incdir+defines generic_regs/generic_table_regs.v
34  vlog -work work -lint -cover bcesf +incdir+defines generic_regs/generic_hw_regs.v
35  vlog -work work -lint -cover bcesf +incdir+defines generic_regs/generic_sw_regs.v
36  vlog -work work -lint -cover bcesf +incdir+defines generic_regs/generic_cntr_regs.v
37  vlog -work work -lint -cover bcesf +incdir+defines generic_regs/generic_regs.v
38
39  vlog -work work -lint -cover bcesf out_queues/small_fifo.v
40  vlog -work work -lint -cover bcesf out_queues/fallthrough_small_fifo.v
41  vlog -work work -lint -cover bcesf +incdir+defines out_queues/remove_pkt_dram.v
42  vlog -work work -lint -cover bcesf +incdir+defines out_queues/store_pkt_dram.v
43  vlog -work work -lint -cover bcesf +incdir+defines out_queues/dram_queue.v
```

```
vlog -work work -lint -cover bcesf +incdir+defines out_queues/oq_header_parser.v
vlog -work work -lint -cover bcesf +incdir+defines out_queues/dram_queue_arbiter.v
vlog -work work -lint -cover bcesf +incdir+defines out_queues/dram_interface_arbiter.v
vlog -work work -lint -cover bcesf +incdir+defines out_queues/dram_queue_regs.v
vlog -work work -lint -cover bcesf +incdir+defines out_queues/output_queues.v

vlog -work work -lint ddr2_blk_rdwr/glbl.v
vlog -work work -lint -cover bcesf ddr2_blk_rdwr/small_fifo.v
vlog -work work -lint -cover bcesf ddr2_blk_rdwr/fallthrough_small_fifo.v

vlog -work work -lint ddr2_blk_rdwr/FIFOs/async_fifo_144b/async_fifo_144b.v
vlog -work work -lint ddr2_blk_rdwr/FIFOs/async_fifo_144b/fifo14.v
vlog -work work -lint ddr2_blk_rdwr/FIFOs/async_fifo_144b/fifomem14.v
vlog -work work -lint ddr2_blk_rdwr/FIFOs/async_fifo_144b/rptr_empty14.v
vlog -work work -lint ddr2_blk_rdwr/FIFOs/async_fifo_144b/sync_r2w14.v
vlog -work work -lint ddr2_blk_rdwr/FIFOs/async_fifo_144b/sync_w2r14.v
vlog -work work -lint ddr2_blk_rdwr/FIFOs/async_fifo_144b/wptr_full14.v

vlog -work work -lint ddr2_blk_rdwr/FIFOs/async_fifo_in_144b_out_72b/async_fifo_in_144b_out_72b.v
vlog -work work -lint ddr2_blk_rdwr/FIFOs/async_fifo_in_144b_out_72b/fifo15.v
vlog -work work -lint ddr2_blk_rdwr/FIFOs/async_fifo_in_144b_out_72b/fifomem15.v
vlog -work work -lint ddr2_blk_rdwr/FIFOs/async_fifo_in_144b_out_72b/rptr_empty15.v
vlog -work work -lint ddr2_blk_rdwr/FIFOs/async_fifo_in_144b_out_72b/sync_r2w15.v
vlog -work work -lint ddr2_blk_rdwr/FIFOs/async_fifo_in_144b_out_72b/sync_w2r15.v
vlog -work work -lint ddr2_blk_rdwr/FIFOs/async_fifo_in_144b_out_72b/wptr_full15.v

vlog -work work -lint -cover bcesf ddr2_blk_rdwr/async_fifo_in_72b_out_144b.v
vlog -work work -lint -cover bcesf ddr2_blk_rdwr/ddr2_blk_rdwr_fifo_72b_2_64b.v
vlog -work work -lint -cover bcesf ddr2_blk_rdwr/ddr2_blk_rdwr_fifo_64b_2_72b.v
vlog -work work -lint -cover bcesf ddr2_blk_rdwr/small_async_fifo.v
vlog -work work -lint -cover bcesf ddr2_blk_rdwr/ddr2_blk_rdwr.v

vlog -work work -lint +define+XILINX +incdir+ddr2_controller/include ddr2_controller/controller/controller_32bit_00.v
vlog -work work -lint ddr2_controller/controller/ffd.v
vlog -work work -lint +incdir+ddr2_controller/include ddr2_controller/controller/data_read_32_bit_dfi.v
vlog -work work -lint +incdir+ddr2_controller/include ddr2_controller/controller/data_write_32bit_dfi.v
vlog -work work -lint +incdir+ddr2_controller/include ddr2_controller/controller/dfi_mc_top.v
vlog -work work -lint ddr2_controller/phy/cal_ctl.v
vlog -work work -lint ddr2_controller/phy/cal_div2.v
vlog -work work -lint ddr2_controller/phy/cal_div2f.v
vlog -work work -lint ddr2_controller/phy/cal_reg.v
vlog -work work -lint ddr2_controller/phy/cal_top.v
vlog -work work -lint ddr2_controller/phy/clk_dcm.v
vlog -work work -lint +incdir+ddr2_controller/include ddr2_controller/phy/controller_iobs_32bit_00.v
vlog -work work -lint +incdir+ddr2_controller/include ddr2_controller/phy/data_path_32bit_rl.v
vlog -work work -lint +incdir+ddr2_controller/include ddr2_controller/phy/data_path_iobs_32bit.v
vlog -work work -lint ddr2_controller/phy/data_path_rst.v
vlog -work work -lint ddr2_controller/phy/data_read_32bit_rl.v
vlog -work work -lint ddr2_controller/phy/data_read_controller_32bit_rl.v
vlog -work work -lint ddr2_controller/phy/data_write_32bit.v
vlog -work work -lint ddr2_controller/phy/dcmx3y0_2vp50.v
vlog -work work -lint ddr2_controller/phy/ddr2_dm_32bit.v
vlog -work work -lint ddr2_controller/phy/ddr2_dqbit.v
vlog -work work -lint ddr2_controller/phy/ddr2_dqs_div.v
vlog -work work -lint ddr2_controller/phy/ddr2_transfer_done.v
vlog -work work -lint ddr2_controller/phy/ddr_dq_iob.v
vlog -work work -lint +incdir+ddr2_controller/include ddr2_controller/phy/ddr_dqs_iob.v
vlog -work work -lint ddr2_controller/phy/dqs_delay.v
vlog -work work -lint ddr2_controller/phy/infrastructure.v
vlog -work work -lint ddr2_controller/phy/infrastructure_iobs_32bit.v
vlog -work work -lint ddr2_controller/phy/infrastructure_top.v
## in infrastructure_top.v Ln: 133 is a sentence for artificially bump the counter
## to reduce the waiting time during simulations.
## It generates the memory model WARNING: 200 us is required before CKE goes active.
vlog -work work -lint +incdir+ddr2_controller/include ddr2_controller/phy/iobs_32bit_00.v
vlog -work work -lint ddr2_controller/phy/mybufg.v
vlog -work work -lint ddr2_controller/phy/RAM_8D.v
vlog -work work -lint +incdir+ddr2_controller/include ddr2_controller/phy/dfi_phy_top.v
vlog -work work -lint +incdir+ddr2_controller/include ddr2_controller/ddr2_controller_dfi.v

## DEBUG parameter disabled in ddr2_parameters.vh
## defined MAX_MEM according to Micron memory model readme file for memory representation
vlog -work work +define+MAX_MEM ddr2_model/ddr2.v
##vopt ddr2 -o ddr2_opt -G DEBUG=0

## To change the level of abstraction of the models
##vlog -work work -lint verilog_testbench/dram_model.v
vlog -work work -lint verilog_testbench/dram_ctrl_model.v

vlog -work work -lint +incdir+defines+testbench testbench/syslog.sv
vlog -work work -lint +incdir+defines+testbench testbench/basic_data.sv
vlog -work work -lint +incdir+defines+testbench testbench/basic_transactor.sv
vlog -work work -lint +incdir+defines+testbench testbench/config_netfpga.sv
vlog -work work -lint +incdir+defines+testbench testbench/eth_frame.sv
vlog -work work -lint +incdir+defines+testbench testbench/netfpga_internal_eth_frame.sv
vlog -work work -lint +incdir+defines+testbench testbench/netfpga_register_transaction.sv
vlog -work work -lint +incdir+defines+testbench testbench/netfpga_dp_in_if.sv
vlog -work work -lint +incdir+defines+testbench testbench/netfpga_dp_out_if.sv
vlog -work work -lint +incdir+defines+testbench testbench/netfpga_reg_if.sv
vlog -work work -lint +incdir+defines+testbench testbench/netfpga_sram_reg_if.sv
vlog -work work -lint +incdir+defines+testbench testbench/netfpga_dp_in_transactor.sv
vlog -work work -lint +incdir+defines+testbench testbench/netfpga_dp_out_transactor.sv
vlog -work work -lint +incdir+defines+testbench testbench/netfpga_reg_transactor.sv
vlog -work work -lint +incdir+defines+testbench testbench/eth_frame_generator.sv
vlog -work work -lint +incdir+defines+testbench testbench/basic_scoreboard.sv
vlog -work work -lint +incdir+defines+testbench testbench/scoreboard_out_queues.sv
vlog -work work -lint +incdir+defines+testbench testbench/basic_coverage.sv
vlog -work work -lint +incdir+defines+testbench testbench/coverage_netfpga_internal_frame.sv
vlog -work work -lint +incdir+defines+testbench testbench/config_netfpga_out_queues.sv
```

```
vlog -work work -lint +incdir+defines+testbench testbench/environment_out_queues.sv
vlog -work work -lint +incdir+defines+testbench testbench/tb_top_output_queues.sv
vlog -work work -lint +incdir+defines+testbench testbench/test_output_queues.sv

onbreak {resume}

set f [open start_time.txt w]
puts $f "Start time was [clock seconds]"
close $f

for {set x 0} {$x<10} {incr x} {
   puts "Running iteration Number: $x"
   quietly set seed1 [expr {int(1 + 1000*rand())}]; # [1, 1000]
   quietly set num_packets 1000
   vsim -L unisims_ver -coverage -gSEED=$seed1 -sv_seed $seed1 -gNUM_PACKETS=$num_packets -voptargs="+acc"
        -permit_unmatched_virtual_intf -t ps work.glbl work.tb_top_output_queues
   do wave_mti.do
   coverage attribute -test $x
   set SolveArrayResizeMax 0
   if {$x != 0} {
     exec rm -f cov_temp.ucdb
    }
   run -all
   if { [examine tb_top_output_queues/test1/normal_exit] == 0} {
      break
   }
   coverage save cov_temp.ucdb
   quit -sim
   if {$x == 0} {
      exec cp cov_temp.ucdb cov_final.ucdb
   } else {
      vcover merge -totals -verbose -out cov_final.ucdb cov_temp.ucdb cov_final.ucdb
   }
}

vcover report -html cov_final.ucdb

# How long since project began?
if {[file isfile start_time.txt] == 0} {
  set f [open start_time.txt w]
  puts $f "Start time was [clock seconds]"
  close $f
} else {
  set f [open start_time.txt r]
  set line [gets $f]
  close $f
  regexp {\d+} $line start_time
  set total_time [expr ([clock seconds]-$start_time)/60]
  set f [open total_time.txt w]
  puts $f "Project time is $total_time minutes"
  close $f
}
# Compress the generated transcript
exec tar zcvf transcript.tar.gz transcript
```