PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE ENGENHARIA
FACULDADE DE INFORMÁTICA
CURSO DE ENGENHARIA DA COMPUTAÇÃO

COMUNICAÇÃO ENTRE EMULADORES EM FPGAS E REDES ETHERNET

Autores
ADELCIO BIAZI
DOUGLAS MACIEL CARDOSO

VOLUME FINAL DO TRABALHO DE CONCLUSÃO DO CURSO DE ENGENHARIA DE COMPUTAÇÃO NA PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

Orientador
PROF. DR. FERNANDO GEHM MORAES

Porto Alegre 2011

RESUMO

O aumento na demanda por sistemas digitais em FPGAs faz com que seja necessário o desenvolvimento de técnicas que aumentem a velocidade e a confiabilidade dos testes nestes dispositivos. O uso de técnicas de emulação por cadeias de *scan* é a melhor opção quando se deseja apresentar boa controlabilidade e observabilidade.

O presente Trabalho de Conclusão de Curso relata o desenvolvimento de um ambiente de transmissão de dados entre um módulo de comunicação *Ethernet* e um sistema de emulação de *hardware*, este emulador é baseado no uso de cadeias de *scan*, que permite avaliar o comportamento de um dado sistema digital diretamente em um dispositivo FPGA. O trabalho consiste em capturar dados do módulo *Ethernet* e inserir os mesmos em um emulador de *hardware*, após a emulação estes dados são enviados novamente ao módulo *Ethernet* e recebidos por um *software* também desenvolvido neste trabalho.

Um *software* foi desenvolvido com a finalidade de estabelecer uma comunicação entre um computador hospedeiro e o FPGA, este programa permite conectar, desconectar e descobrir endereço *IP* da placa que contém o FPGA a ser utilizado.

AGRADECIMENTOS

Agradecemos primeiramente ao nosso orientador Fernando Gehm Moraes pelo apoio, pela paciência e pelas críticas durante o desenvolvimento deste trabalho, da mesma forma agradecemos ao Professor Ney Laert Vilar Calazans que gentilmente aceitou participar da banca de avaliação deste trabalho. De mesma forma agradecemos ao Professor Edson Moreno por participar da banca de avaliação. Também agradecemos a todos os participantes do grupo GAPH, pois a ajuda destes foi fundamental para a realização deste trabalho, em especial a Douglas Silva, Maurício Magnaguagno, Guilherme Medeiros, Dairan Severo Corrêa, não só pelo suporte técnico, mas também pelo excelente e agradável ambiente de trabalho e estudo.

Eu, Adelcio Biazi, agradeço também ao meu colega Douglas Maciel Cardoso pela parceria no desenvolvimento deste trabalho. Agradeço em especial a minha esposa pelo apoio e por ter aguentado as minhas reclamações, não só durante a resolução deste trabalho, mas de toda a graduação. E quero agradecer a todos os professores da graduação, principalmente aos que complicaram nas provas, pois desta forma posso declarar, que sou um engenheiro bem preparado para o mercado de trabalho e para um futuro mestrado.

Eu, Douglas Maciel Cardoso quero também agradecer ao meu colega e amigo durante a faculdade Adelcio Biazi, agradeço aos professores do curso de Engenharia de Computação por todo conhecimento a mim oferecido. E agradeço aos meus amigos pessoais que nas horas de lazer me fazem rir e esquecer um pouco das dificuldades que se pode enfrentar durante um curso de engenharia. Não poderia deixar de manifestar meu profundo agradecimento a minha noiva e futura psicóloga que esteve sempre ao meu lado me apoiando e me incentivando mesmo quando nem sabia de que tecnologia eu estava falando.

Sumário

Lista de Figuras

Lista de Siglas

ARP	-	*Address Resolution Protocol*
LCTCP	-	*Low Cost Transmission Control Protocol*
BIST	-	*Built-in Self-Test*
CAD	-	*Computer-aided design*
CE	-	*Chip Enable*
CTRL+NACK	-	*Control and negative acknowledgment*
DHCP	-	*Dynamic Host Configuration Protocol*
DNS	-	*Domain Name System*
DUT	-	*Design Under Test*
DF	-	*Don't Fragment*
FCS	-	*Frame Checksum*
FPGA	-	*Field Programmable Gate Array*
HL	-	*Header Length*
IEEE	-	*Institute of Electrical and Electronics Engineers*
IP	-	*Internet Protocol*
JTAG	-	*Joint Test Action Group*
LUT	-	*Look Up Table*
MAC	-	*Media Access Control*
MF	-	*More Fragment*
MII	-	*Media Independent Interface*
OSI	-	*Open Systems Interconnection*
PCIe	-	*Peripheral Component Interconnect Express*
RAM	-	*Random Access Memory*
SRL	-	*Shift Register LUT*
TCP	-	*Transmission Control Protocol*
TCP/IP	-	*Transmission Control Protocol / Internet Protocol*
TOS	-	*Type Of Service*
TTL	-	*Time To Live*
UDP	-	*User Datagram Protocol*

1 INTRODUÇÃO

Dispositivos *FPGA* (do inglês, *Field Programmable Gate Arrays*) são circuitos integrados que podem conter milhares de unidades lógicas programáveis idênticas. A utilização de FPGA vem crescendo na indústria de componentes eletrônicos, podendo ser utilizados nas mais variadas aplicações, tais como sistemas de comunicação de dados, automação, equipamentos hospitalares, etc. FPGAs foram criados pela *Xilinx* em 1985 e hoje são comercializados por várias empresas, sendo que os principais fornecedores são *Xilinx* e *Altera*.

Devido à elevada complexidade de projetos desenvolvidos em FPGAs, faz-se necessário o constante empenho na criação de novas ferramentas que reduzam o tempo gasto na verificação de circuitos nestes dispositivos. Seguindo essa tendência, juntamente com a necessidade de redução do *time-to-market*, o mercado disponibiliza ferramentas e técnicas para acelerar o processo de simulação e visualização de sinais internos do FPGA, como por exemplo, o *Chipscope Pro Analyzer* desenvolvido pela *Xilinx*. Porém, estas ferramentas têm um alto custo de aquisição, ficando viável apenas para um reduzido grupo de empresas.

A emulação implica em verificar um circuito durante a sua execução, a fim de observar sinais de um circuito, neste caso diretamente em FPGA, para tal utiliza-se um computador hospedeiro como auxiliar no controle dos sinais a serem observados. Uma interface de comunicação de alto desempenho, como o barramento *PCIe* (do inglês, *Peripheral Component Interconnect Express*) é uma opção indicada, dado que a mesma pode atingir taxas de transferências de até 400MB/s (*Mega Bytes por Segundo*) de dados úteis, sendo que existe a possibilidade de aumentar em até 16 vezes esta velocidade através de uma técnica conhecida como agrupamento de vias [CAS10].

Embora o barramento *PCIe* seja uma ótima solução devido à alta velocidade, ele apresenta uma interface difícil de ser desenvolvida, pois requer *drivers* específicos para cada fabricante de plataforma, além de ter pouca portabilidade, pois necessita de um *hardware* extra, ou seja, um computador que possua um conector padrão *PCIe*. Portanto torna-se necessário a exploração de novas tecnologias para facilitar a comunicação em FPGAs.

Neste trabalho foi desenvolvido um módulo de *hardware* capaz de controlar a comunicação entre um projeto de emulação de *hardware* em FPGA usando a tecnologia *Ethernet*. Para tal, foi utilizado o módulo de comunicação *LCTCP* (do inglês, *Low Cost Transmission Control Protocol*) proposto por [SIL11], que é encapsulado no protocolo de transporte *UDP* (do inglês, *User Datagram Protocol*) e que foi desenvolvido a fim de minimizar o custo da comunicação *TCP* convencional. *LCTCP* é uma alteração do módulo *Ethernet* desenvolvido por [REI09].

1.1 OBJETIVOS

Este trabalho tem por objetivo principal desenvolver um mecanismo de acesso e controle de um emulador de *hardware* (que aqui será referenciado por *Emulator*) diretamente em FPGA, através de redes *Ethernet*. Tanto o *Emulator* quanto o *Ethernet* são módulos de *hardware* desenvolvidos em trabalhos anteriores, e serão abordados mais adiante.

Para realização deste TCC foi necessário o domínio de: uma interface de alto desempenho entre uma estação de trabalho e uma placa de prototipação; técnicas de manipulação de arquivos *VHDL* (do inglês, *Very High Speed Integrated Circuits Hardware Description Language*); técnicas de prototipação e ferramentas de *CAD* (do inglês, *Computer-Aided Design)* para síntese; técnicas de comunicação de dados em uma interface de rede. Estas técnicas mostram a estreita relação deste trabalho com o curso de Engenharia da Computação, utilizando especificamente conteúdos das cadeiras de: Programação de Periféricos, Teste e Confiabilidade de Sistemas, Algoritmos e Estruturas de Dados, Laboratório de Programação, Organização e Arquitetura de Computadores, Redes de Comunicação, Microeletrônica e Projeto de Sistemas Integrados, desta forma atingindo o objetivo deste trabalho.

1.2 MOTIVAÇÃO

No mercado competitivo atual para que uma empresa consiga estabelecer uma vantagem sustentável é necessário que ela procure diminuir o tempo de colocação de um produto no mercado. Neste contexto, faz-se necessário a agilidade na verificação das funcionalidades do produto em questão. Ao se tratar de sistemas digitais o problema de *time-to-market* acentua-se ainda mais devido ao fato de que a não verificação de um determinado produto pode acarretar em perda total do mesmo.

A motivação do trabalho surgiu da necessidade de tornar flexível a comunicação no processo de verificação e teste de projetos de sistemas digitais. O presente trabalho estabelece uma comunicação *Ethernet* entre o *hardware* a ser verificado e a ferramenta de verificação, desta forma aumentando a flexibilidade na comunicação de projetos de sistemas digitais.

1.3 CONTRIBUIÇÕES

Entre as contribuições desse trabalho se destacam o desenvolvimento de um ambiente completo de comunicação entre a ferramenta de emulação de *hardware* em FPGA e uma infraestrutura de comunicação *Ethernet*, desta forma permitindo a validação e concepção de circuitos digitais mais confiáveis.

A ferramenta, além de prover o envio e recebimento de dados através do módulo *Ethernet*, é capaz de enviar e receber dados diretamente do circuito em teste, de modo a fazer simulação de um *DUT* (do inglês, *Design Under Test*) no ambiente de *emulação*. Para isto, fez-se necessário o desenvolvimento de um *software* capaz de parametrizar o *MAC* (do inglês, *Media Access Control*) da placa que contém os FPGAs, a fim de estabelecer uma conexão entre esta placa e um computador hospedeiro.

1.4 ORGANIZAÇÃO DO DOCUMENTO

Este documento está organizado como segue. O Capítulo 2 apresenta as fundamentações teóricas, que incluem o *Emulator*, *Ethernet*, FPGA, ferramentas de *CAD* e de monitoração de redes. O Capítulo 3 descreve o fluxo do projeto e a forma como foi feita a comunicação entre o *Ethernet* e o módulo de emulação de *hardware* em FPGA. O Capítulo 4 apresenta o *software* desenvolvido para prover suporte ao *hardware*. O Capítulo 5 descreve o processo de validação do sistema. O Capítulo 6 apresenta a conclusão do presente Trabalho e trabalhos futuros.

2 REFERENCIAL TEÓRICO

Este Capítulo descreve as principais tecnologias empregadas neste trabalho, desta forma aqui foram abordados temas como: FPGAs, seção 2.1; técnicas de emulação de *hardware*, seção 2.2; comunicação de redes, seção 2.3.

2.1 FPGA

FPGA é um circuito integrado que contém milhares de unidades lógicas idênticas que podem ser configuradas independentemente [KIL07], podendo ser interconectadas a partir de uma matriz de trilhas condutoras e chaves programáveis, como por exemplo, *PlanAhead*. A configuração das unidades lógicas do FPGA é dada por um arquivo binário que pode ser gerado através de ferramentas de *CAD*. Este arquivo contém informações necessárias para especificar a função de cada unidade lógica e para seletivamente fechar as chaves da matriz de interconexão.

Na Figura 1 os blocos lógicos configuráveis são formados por circuitos combinacionais e sequenciais. Os blocos de entrada e saída servem para definir as portas de entrada e saída dos sinais no FPGA, a matriz de roteamento é usada para fazer as interconexões entre os blocos lógicos que utiliza as linhas de roteamento.

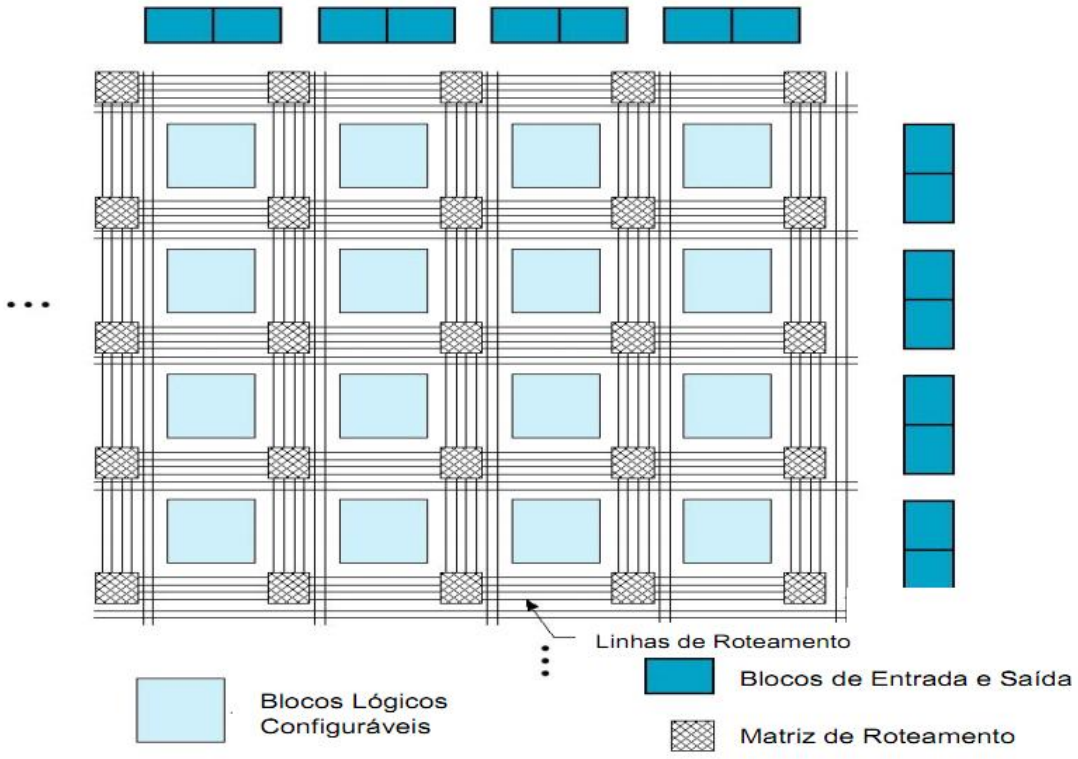


Figura 1 – Bloco básico do FPGA.

2.1.1 Fluxo de Projeto em FPGAs

O fluxo de projeto clássico de um circuito para um FPGA pode ser resumido em quatro etapas, especificação, verificação, implementação e depuração do sistema, sendo que cada fabricante fornece um ambiente integrado de desenvolvimento capaz de dar suporte a cada uma dessas etapas, como exemplo, *Quartus* da *Altera* e *ISE* da *Xilinx.*

Cada etapa do fluxo de projeto pode ser definida da seguinte forma: a especificação é a parte onde os projetistas definem o comportamento que o *hardware* deve ter, sendo que o resultado dessa etapa é um modelo de circuito descrito em linguagem de descrição de *hardware*, como *VHDL ou Verilog*; a verificação é feita por meio de simulações, que consiste na atribuição de valores aos sinais e verificação de suas respostas; a etapa de implementação é a parte onde a descrição de *hardware* é convertida em um arquivo binário, que será carregado no FPGA; a fase de depuração de sistema consiste em fazer uma verificação no comportamento do sistema, semelhante à fase de verificação, sendo que nesta os sinais observados estarão dentro do FPGA, ou seja, trata-se de verificar o comportamento do circuito durante sua operação.

2.2 **EMULATOR**

Com o objetivo de identificar se o sistema em desenvolvimento apresenta os requisitos de especificação, etapas de teste e verificação são utilizadas para auxiliar projetistas a encontrar erros antes de chegar à etapa de fabricação, evitando que o mesmo chegue à etapa final com erros de projeto, onde haveria desperdício de material, principalmente em arquiteturas que hoje possuem milhões de portas lógicas e que tem alto custo de fabricação.

Uma forma de auxiliar na verificação da existência de erros de projeto é através da verificação direta em FPGA através de emulação, com o circuito também implementado no FPGA, porém controlado por um computador hospedeiro.

O *Emulator* desenvolvido por [CAS10] utiliza duas máquinas de estados para controlar entradas e saídas de dados em cadeias de *scan*. A máquina de estados que controla a entrada de dados funciona da seguinte forma: ela inicia em um estado de repouso até que haja um dado disponível. Após o recebimento de um dado a máquina passa para um estado que avalia e interpreta este como um dado de configuração (que controla informações de controle das cadeias de *scan*) ou de transporte (que verifica se as cadeias de *scan* devem ser ignoradas ou não) ou de teste (quando os dados serão inseridos nas cadeias de *scan*).

Como ocorre na máquina que controla a entrada de dados nas cadeias de *scan*, a máquina que controla a saída de dados também inicia em um estado de repouso, em seguida a máquina atinge o

estado que habilita o registrador mestre das cadeias de *scan* para armazenar os dados. O circuito com as cadeias de *scan* de estrada e saída controlados pelas máquinas de estados pode ser visto na Figura 2.

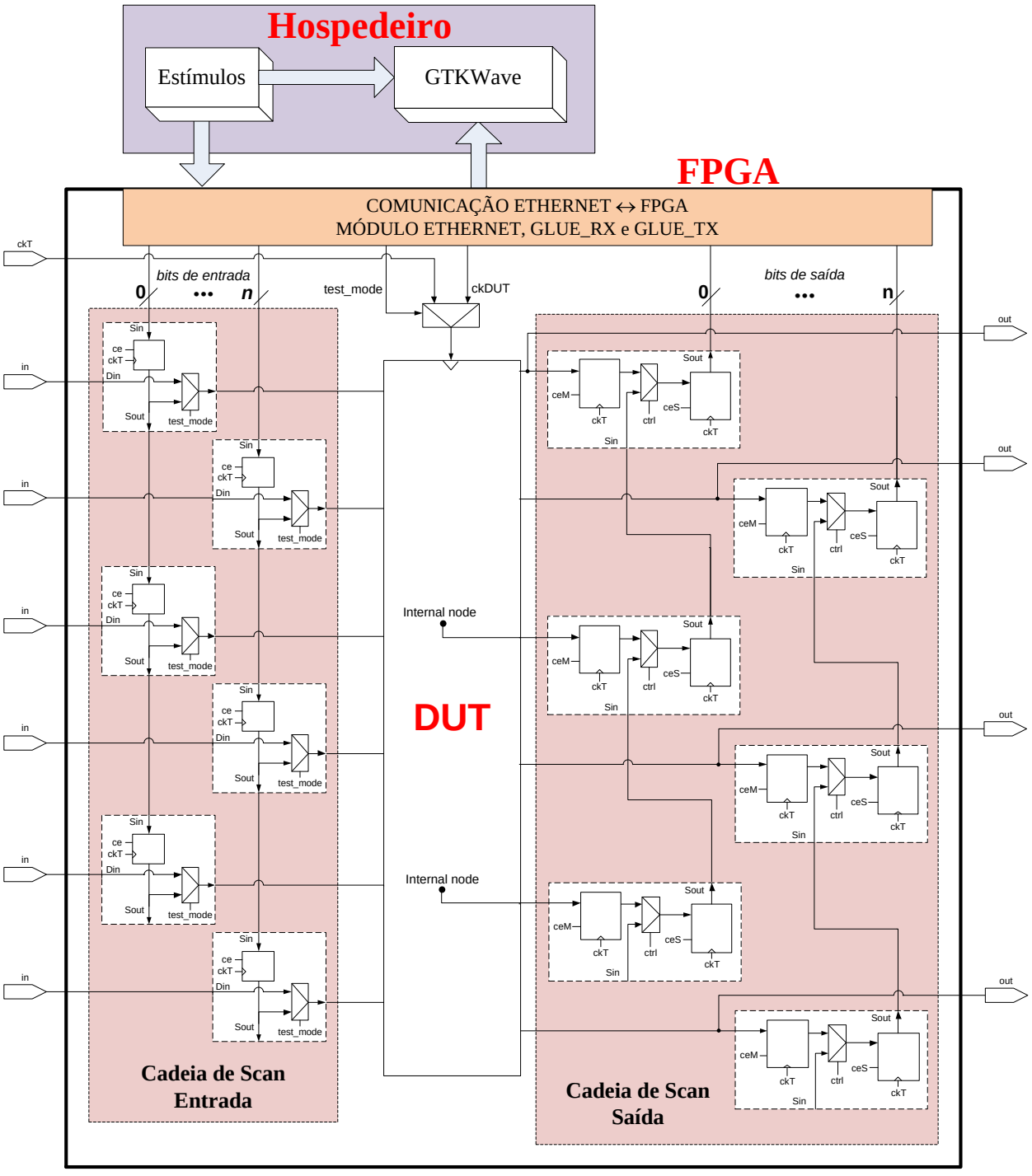


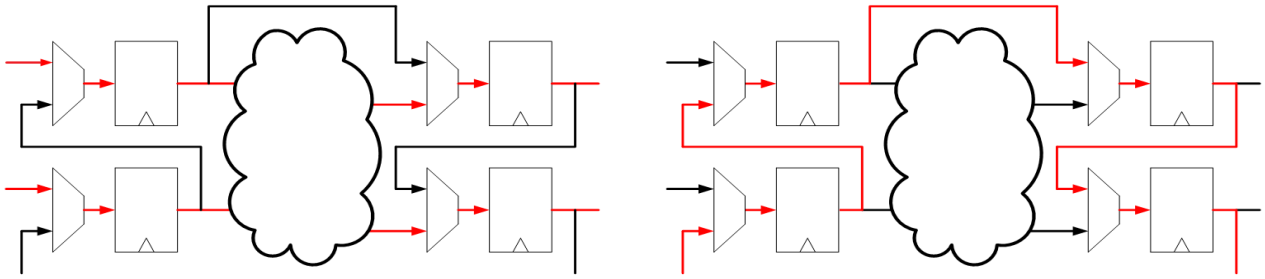
Figura 2 – Arquitetura do sistema de emulação proposto em [CAS10].

2.2.1 Emulação por Cadeias de Scan

Existem muitas técnicas para verificação de circuitos digitais, podendo ser destacado: *Ad Hoc Testing* que são coleções de ideias para redução do esforço relacionado ao teste, sendo útil para

circuitos de pequeno porte; *BIST* (do inglês, *Built-in Self-Test*), onde circuitos extras são adicionados ao projeto, tendo a vantagem de reduzir o custo no teste de manufatura, porém tem a desvantagem de *overhead* de área de silício e de possível degradação no desempenho; *BURN_IN* que tem a capacidade de testar a expectativa de vida de um CI.

Os testes baseados em cadeias *de scan* são testes do tipo varredura que permitem a *controlabilidade* e a *observabilidade* de cada registrador presente no circuito. Estes registradores são modificados para operarem em dois modos, conforme pode ser visto na Figura 3, observa-se que nesta figura os caminhos ativos estão destacados em vermelho: no modo normal os registradores desempenham suas funções regulares, ou seja, é como se o circuito não tivesse sofrido nenhuma alteração, Figura 3(a); no modo *scan*, os registradores são conectados para formar um registrador de deslocamento denominado cadeia de *scan*, Figura 3(b*)*. Os registradores das cadeias de *scan* são formados por um *flip-flop* e um *multiplexador*, que quando o sinal de *scan* está desabilitado eles se comportam como registradores convencionais e quando o sinal de *scan* está habilitado o dado do circuito em teste é carregado para a cadeia de *scan* de saída. Nas próximas seções será mais bem explicado o funcionamento de cadeias de *scan* de entrada e de saída.



(a) Modo Normal (b) Modo *Scan*

Figura 3 – Método de teste baseado em *scan*, [BER03].

2.2.2 Bibliotecas de Cadeias de Scan

O bloco básico construtivo das cadeias de *scan* de entrada são formadas por um flip-flop tipo D e um multiplexador 2x1, conforme pode ser visto na Figura 4, que são replicados tantas vezes quanto for a profundidade da cadeia. O multiplexador é utilizado para decidir se o circuito será alimentado com sinais que vêm do registrador das cadeias de *scan*, quando estiver em teste, ou de um circuito externo, quando não está sendo testado. É controlado pelo sinal *test_mode* que em nível lógico baixo utiliza como entrada os dados procedentes do circuito externo (sinal *Din*). Em nível lógico alto permite a passagem do dado oriundo do registrador da cadeia de *scan*. O flip-flop é utilizado para registrar o dado de entrada, proveniente do hospedeiro (sinal *Sin*), e desloca para o próximo registrador da cadeia através de *Sout*, desta forma preenchendo a profundidade total da cadeia de *scan*.

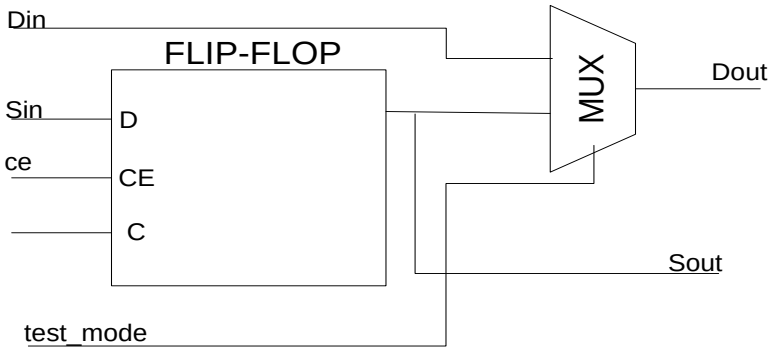


Figura 4 – Circuito que compõe o bloco unitário da cadeia de *scan* de entrada.

A Figura 5 apresenta os multiplexadores e registradores formando duas cadeias de *scan* de profundidade dois. Percebe-se que cada bloco têm os mesmos sinais *ce* (do inglês, *chip enable*), *ckT* e *test_mode*, desta forma possibilitando a replicação dos blocos. A Figura 5 também mostra que a saída do primeiro registrador (*Sout)* é a entrada do segundo registrador (*Sin)*, desta forma pode-se fazer a varredura de até quatro *bits* por vez (dois *bits* em paralelos) durante o nível lógico alto do sinal *test_mode*.

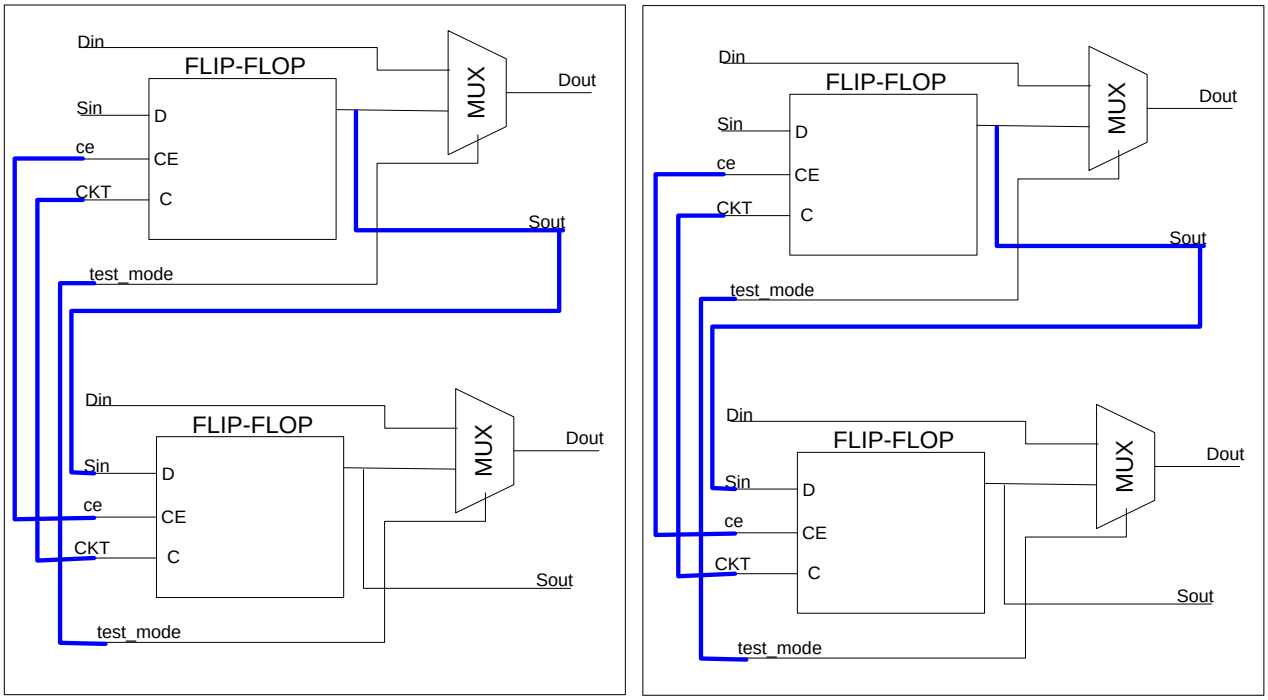


Figura 5 – Duas cadeias de *scan* de entrada com profundidade dois.

Conforme pode ser visto na Figura 6, diferentemente de *scan* de entrada, a *scan* de saída é composta por dois *flip-flop* tipo D e um multiplexador 2x1, que são replicados tantas vezes quanto for a profundidade da cadeia. O multiplexador tem a função de interligar os dois registradores e escolher se o dado que entra no segundo registrador vem do *Sin*, ou da saída do registrador anterior.

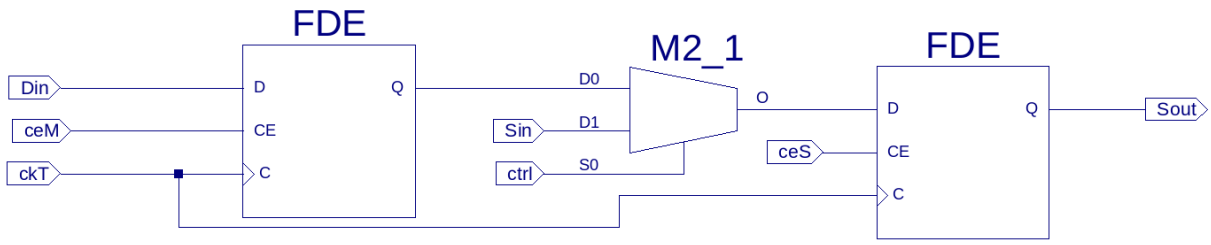


Figura 6 – Circuito que compõe o bloco unitário da cadeia de *scan* de saída [CAS10].

A Figura 7 apresenta o circuito da cadeia de *scan* de saída com profundidade dois. Segundo [CAS10] o circuito funciona da seguinte maneira: "*Uma vez que os blocos (1) e (2) estão replicados, os sinais ckT, ceM, ceS e ctrl são os mesmos para ambos. O sinal Din de ambos os blocos é o sinal que insere dados na cadeia de scan, alimentando o registrador mestre. Observa-se que o sinal de saída Sout do bloco (1) alimenta o sinal de entrada Sin do bloco (2), que faz com que os dados sejam deslocados através do controle do sinal ceS*". (Castilhos, G. M.; Giacomet, L. L. S. P; Grechi, T. "Emulação de Sistemas Digitais Síncronos em Dispositivos FPGAs", Monografia de Trabalho de Conclusão de Curso, PUCRS, 2010, p.27-28).

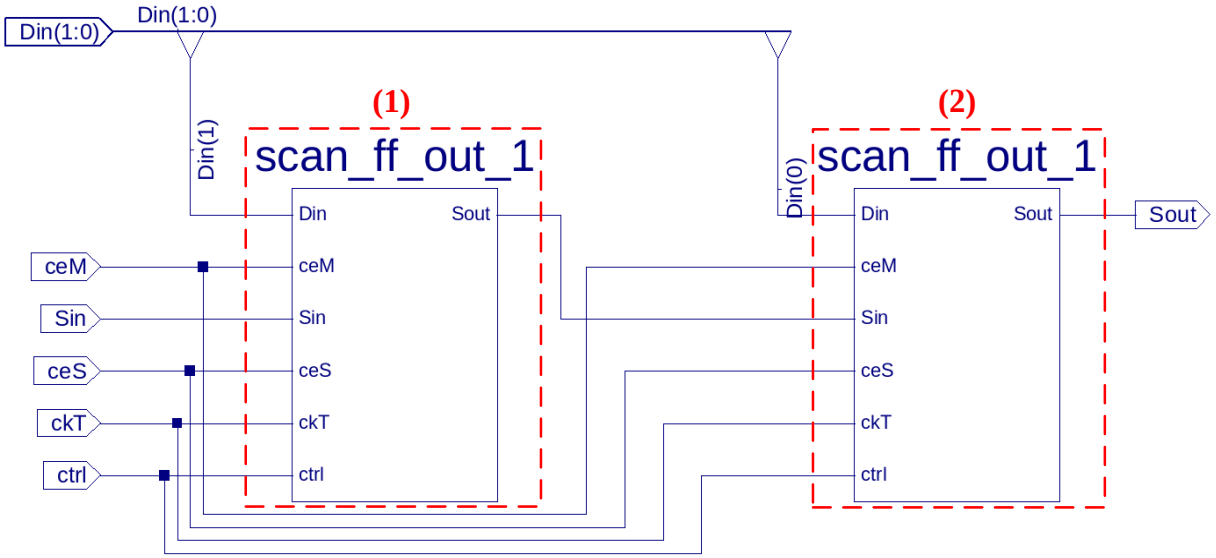


Figura 7 – Circuito da cadeia de *scan* de saída com profundidade 2 [CAS10].

Em resumo o funcionamento das cadeias de *scan* é dado da seguinte forma: um computador hospedeiro envia os dados a ser testado, que é carregado para as cadeias de *scan* de entrada, se o sinal *test_mode* estiver em nível lógico um, os dados serão carregados para os registradores de deslocamento (flip-flops), através do sinal *Sin*, depois de certo tempo, toda a profundidade das cadeias de *scan* serão preenchidas e o *DUT* avaliara as entradas. Após a avaliação do *DUT* os dados são armazenados nas cadeias de *scan* de saída. Estes dados são comparados com os valores esperados, desta forma, se consegue controlabilidade de todas as entradas e observabilidade de todas suas saídas.

2.2.3 PCI Express

Este tema é abordado neste tópico por se tratar de uma tecnologia utilizada na comunicação de dados entre um computador hospedeiro e o FPGA, onde o *Emulator* é executado.

Um exemplo de uso de *PCIe* é dado pela placa de prototipação *DN9200k10PCIe-8T,* fabricada pela *Dini Group*. A Figura 8 apresenta uma foto desta placa com destaque em vermelho dos principais módulos. Neste Capítulo será abordado apenas o módulo *PCIe*, por se tratar de um sistema de comunicação entre os FPGAs. O módulo *Ethernet* será abordado nas próximas seções, pois o mesmo merece destaque, por se tratar do módulo que substituirá o *PCIe* neste trabalho.

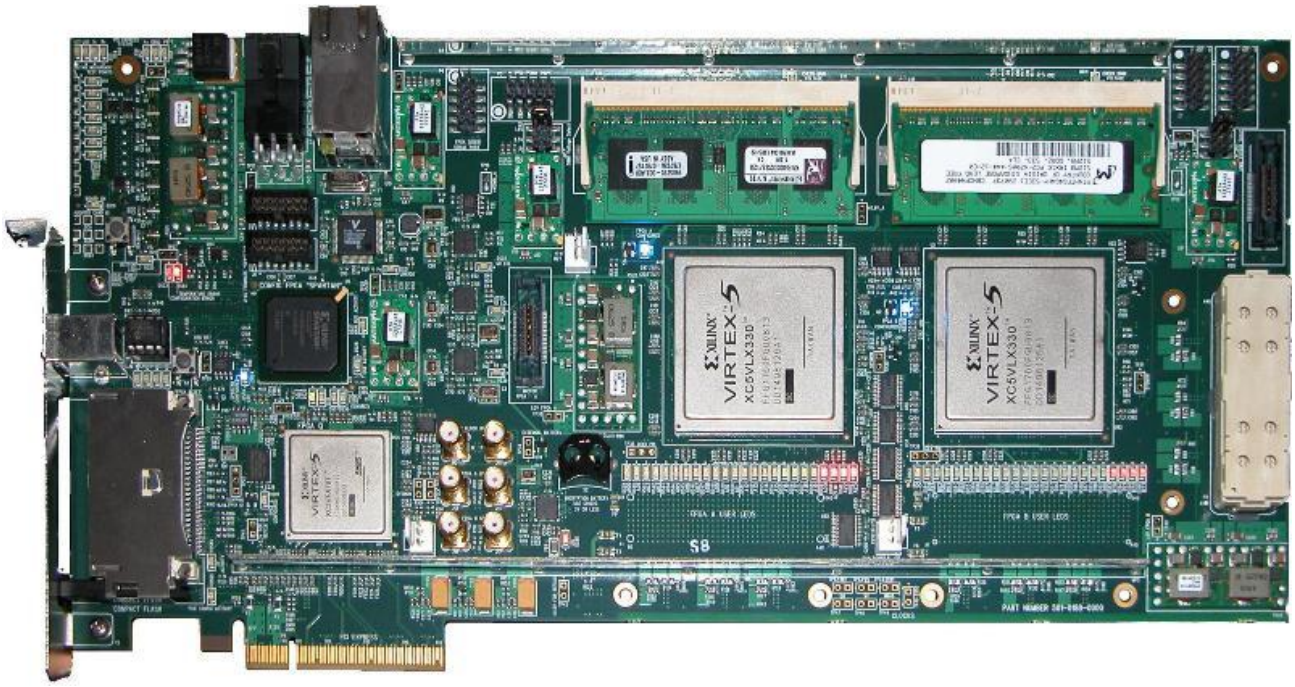


Figura 8 – Placa *DN9200k10PCIe-8T* utilizada no projeto [DIN09].

PCI Express é um barramento ponto a ponto com canal de comunicação bidirecional, suporta velocidades de 256MB/s até 8GB/s. Uma placa *PCIe* pode ser acoplada a um conector de seu tamanho ou de tamanho maior, mas não em tamanho menor.

2.3 REDES DE COMUNICAÇÕES

Para a realização deste trabalho, foi necessário o domínio de algumas tecnologias e protocolos de redes, em particular: *Ethernet*, pois o projeto foi implementado através desta tecnologia de rede; o protocolo *IP* utilizado para transmitir dados através de um pacote *IP*; O protocolo *UDP*, que utiliza um pacote *UDP* para transferir os dados na rede; o modelo *OSI* (do inglês, *Open Systems Interconnection*), sendo que este é um modelo de referência teórico que serve para melhor

compreensão de uma rede baseada em camadas.

Este trabalho foi desenvolvido sobre o modelo *TCP/IP*, que é um modelo em 4 camadas e será melhor explicado nos próximos tópicos.

2.3.1 Modelo TCP/IP

O modelo *TCP/IP* também conhecido como modelo de pilha de protocolos *TCP/IP*. Tem como origem de seu nome seus 2 principais protocolos: *TCP* (do inglês, *Transmission Control Protocol* - Protocolo de Controle de Transmissão) e o *IP* (do inglês, *Internet Protocol* - Protocolo de Interconexão). É importante não confundir o modelo de camadas *TCP*/*IP* com os protocolos *IP* e *TCP*, pois os mesmos fazem parte de um conjunto de protocolos utilizados em comunicação de redes.

O conjunto de protocolos pode ser visto como uma pilha de quatro camadas, onde cada uma é responsável por uma coleção de tarefas, fornecendo um conjunto de serviços para o protocolo da camada superior. A Figura 9 apresenta o modelo de camada *TCP*/*IP* com seus respectivos protocolos. Os protocolos sublinhados, em cada camada, são os que serão utilizados neste trabalho.

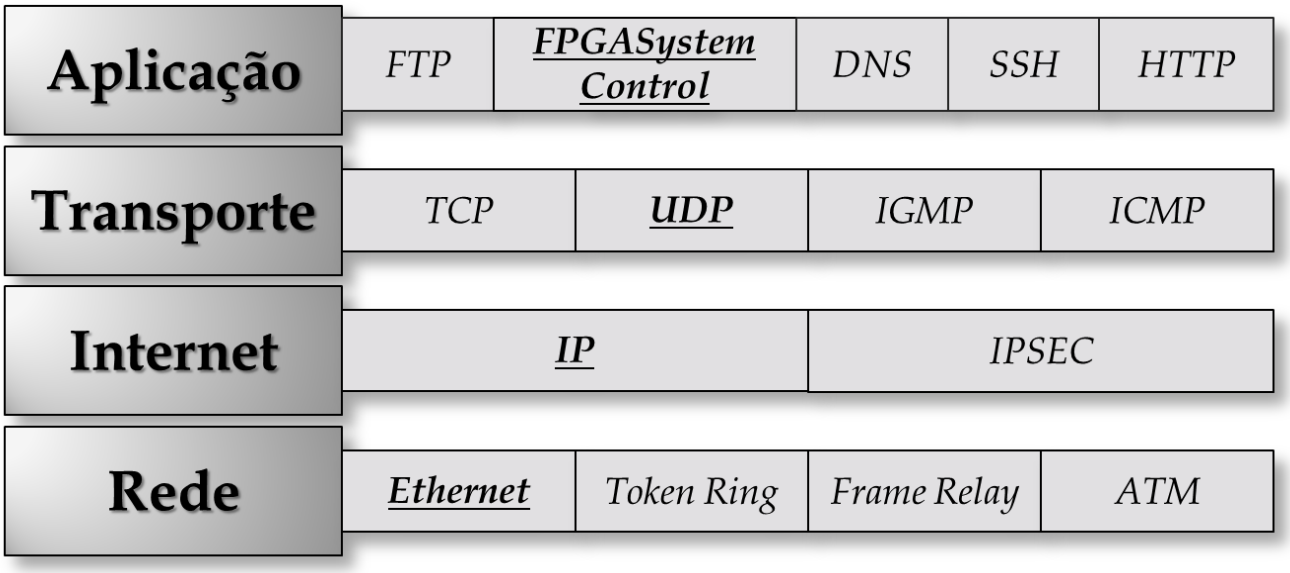


Figura 9 – Modelo da camada *TCP*/*IP*.

Na camada 1 ou camada de rede tem como um de seus protocolos o *Ethernet*, que se utiliza de seu endereço *MAC* para localizar o host na rede. A camada *Internet* utiliza o protocolo *IP*, que em seu pacote possui o endereço lógico *IP*, que serve para encontrar a rede que contém o endereço *MAC*. Na camada de *Transporte* o protocolo utilizado neste trabalho é o *UDP*, neste protocolo será informado o endereço a que serviço pertence os dados enviados, este endereço é conhecido como endereço de porta. A camada de aplicação é a camada onde ficam os programas que fazem uso dos

dados transportados, nesta camada foi definido como nome provisório *FPGASystem Control*. Nos próximos tópicos serão detalhados os protocolos mais importantes do modelo *TCP*/*IP*.

2.3.2 Ethernet

Ethernet é uma tecnologia desenvolvida por *Robert Metcalfe* na década de 1970. Trata-se de um padrão de transmissão de dados para redes locais, baseado no principio de que todas as máquinas da mesma rede estão ligadas em um mesmo meio de comunicação. De acordo com o meio físico de interconexões.

A Figura 10 representa o pacote de dados Ethernet, que é composto por informações de endereçamento físico, protocolos e correção de erros.

Figura 10 – Cabeçalho do pacote *Ethernet*.

O endereço Ethernet, *MAC* Destino e *MAC* Origem, vêm definidos com as placas de redes, são chamados de endereços físicos e cada placa de rede possui um endereço distinto. Na Figura 10 os primeiros 6 bytes são para definir o endereço físico da máquina que vai receber os dados e os próximos 6 bytes são destinados a armazenar o endereço físico da máquina que vai receber os dados. O campo tipo formado pelos próximos 2 bytes serve para informar que tipo de dados é transmitido. Este pacote pode transmitir de 0 a 1500 bytes no campo dados, lembrando que este campo é composto por outros pacotes com seus respectivos cabeçalhos e protocolos, como por exemplo, o pacote *IP* e o pacote *UDP* ou *TCP* que serão comentados nas próximas seções. Os últimos 4 bytes servem para definir o campo FCS (*Frame Checksum*) que é um cálculo de verificação de erro de todos os dados.

2.3.3 Protocolo TCP

O protocolo *TCP* especifica um formato de pacotes e utiliza mecanismos de sincronização para que dois equipamentos possam se comunicar de forma segura e confiável. O *TCP* é um protocolo orientado à conexão e funciona da seguinte forma: se o cliente quer enviar um pacote de dados para o servidor, (1) envia um pacote *SYN* (pedido de conexão com o servidor, do inglês, *Synchronizing*), (2) o servidor reconhece o pedido de conexão enviando um segmento do tipo *SYN* com bit de reconhecimento *ACK* (do inglês, *Acknowledgment*), (3) o cliente responde ao servidor com o pacote de dados.

2.3.4 Protocolo IP

Ao contrario do *TCP* o protocolo *IP* não é orientado a conexão, ou seja, quando um pacote precisa ser enviado, o outro lado pode estar desconectado. Por este motivo é o protocolo adotado pelos servidores de *e-mail*. Por não ser orientado à conexão, possui um conjunto de regras que envolvem a ideia da expedição de pacotes não confiáveis. Juntamente com *TCP,* forma o núcleo dos protocolos da rede mundial de computadores que chamamos de *Internet*.

A Figura 11 apresenta o conjunto de protocolos do pacote *IP*, lembrando que o pacote *IP* é um pacote inserido no pacote Ethernet.

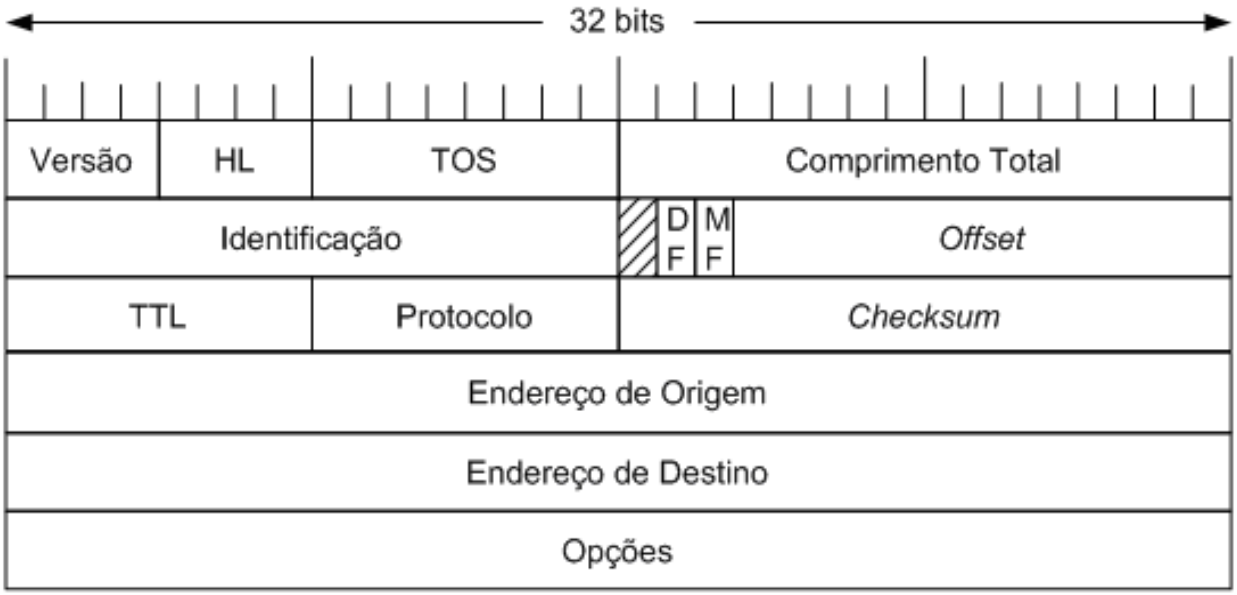


Figura 11 – Cabeçalho do pacote *IP*.

Especificação dos campos do pacote *IP*:

- Versão: informa a versão do protocolo a quem o pacote pertence, a versão atual é 4, devido ao *IPv4*.
- HL (*Header Length*): tamanho do cabeçalho em palavras de 32 *bits*.
- TOS (*Type Of Service*): informa o tipo de rede que o pacote necessita.
- Comprimento Total: tamanho total do pacote (cabeçalho+dados).
- Identificação: identifica a qual pacote pertence um fragmento recebido.
- Fragmentação: controle de fragmentos, DF (*Don't Fragment*), MF (*More Fragments*): se o pacote é o último fragmento ou se existe mais. Se o campo DF estiver em nível lógico alto não é permitido a fragmentação do pacote. Se fragmentado, o campo MF coloca todos os fragmentos em nível lógico alto, exceto o último.
- *Fragment offset*: indica local do fragmento no pacote.
- TTL (*Time To Live*): controle do tempo de vida de um pacote, decrementado à cada roteador.

- Protocolo: Informa o processo de transporte que deve ser aplicado ao pacote, ou seja, se vai utilizar o protocolo *TCP*, *UDP* ou outro protocolo da camada de transporte.
- *Header checksum*: confere as informações presentes no cabeçalho, para verificar na origem se teve algum erro durante a transmissão.
- Endereço de origem: endereço *IP* origem.
- Endereço de destino: endereço *IP* destino.
- Opções: projetado para permitir a inclusão de informações inexistentes no projeto original do protocolo.

2.3.5 Protocolo DHCP

Trata-se de um protocolo de serviço *TCP/IP* que é utilizado para alocar endereços *IP* automaticamente. No início de uma conexão, a estação de origem da comunicação não sabe qual é o seu endereço *IP*, muito menos o servidor *DHCP* da rede. Então, o processo de atribuição de *IP* ocorre da seguinte maneira: inicialmente a estação envia um pacote em *broadcast* denominado *DHCPdiscover*; os servidores *DHCP* disponíveis enviam um pacote *broadcast* denominado *DHCPoffer*, que apresentam ao solicitante suas propostas de configurações; ao receber os pacotes *DHCPoffer*, o cliente *DHCP* seleciona uma das opções e envia um pacote broadcast denominado *DHCPrequest*, que indica a opção e o servidor escolhido; ao receber um quadro *DHCPrequest*, o servidor *DHCP* confirma a atribuição dos parâmetros de configuração oferecidos através de um pacote *DHCPack*; de posse da configuração estabelecida, o cliente *DHCP* envia um pacote broadcast *ARPrequest* para verificar se nenhum outro computador já possui este endereço *IP* configurado, se não houver resposta é por que este endereço está disponível.

2.3.6 Protocolo UDP

Da mesma forma que o protocolo *IP*, o *UDP* não é orientado a conexão, por isto não é um protocolo confiável. Para resolver este problema, ele implementa uma série de estruturas de controle, tais como, *timeouts*, retransmissão, *ACK* e controle de fluxo. Os dados são enviados ao destino e não existe confirmação de entrega, além de poder ser entregue fora da ordem de envio.

Apesar de não garantir integridade no transporte de dados, é bastante utilizado em sistemas de comunicação sem conexão como é o caso da negociação *DHCP*, mencionada no tópico anterior. Outra vantagem deste protocolo é que, por não ter grandes estruturas de controle, seu pacote é praticamente composto por dados.

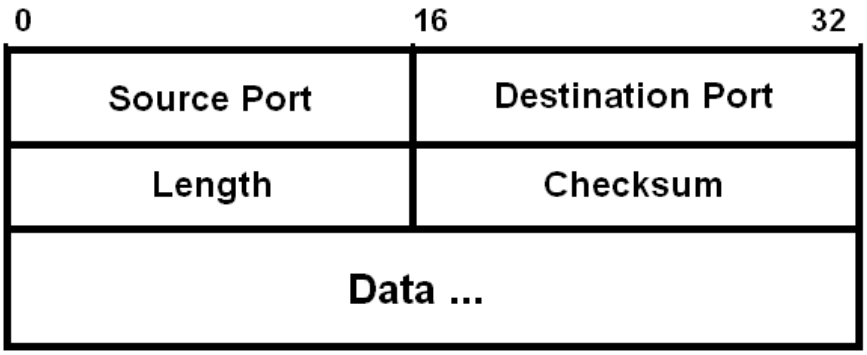


Figura 12 – Cabeçalho do pacote *UDP*.

A Figura 12 apresenta o pacote de dados *UDP*. Os primeiros 32 *bits*, *Source Port* e *Destination Port*, são utilizados para especificar a que serviço pertence o pacote, o primeiro é opcional. O campo *Length* informa o tamanho de todo o pacote, cabeçalho e dados. O campo *Checksum* é utilizado para verificar a ocorrência de dados corrompidos.

2.3.7 Protocolo LCTCP

Em [SIL11] é implementado um protocolo baseado no *TCP*, de baixo consumo de área e executado na camada de transporte dentro do protocolo *UDP*.

O controle de fluxo e de erros do *LCTCP* é baseado no cálculo do *checksum* do pacote *UDP* e no número de sequência de pacotes. Em suma o que *LCTCP* implementa é um campo de sequência de pacotes e um campo de *flag* no *UDP. O* campo *flag* serve para informar se é um pacote de sincronização (*SYN*), controle (*CTRL*), confirmação (*ACK*) ou finalização de conexão (*FIN*), conforme pode ser visto na Figura 13.

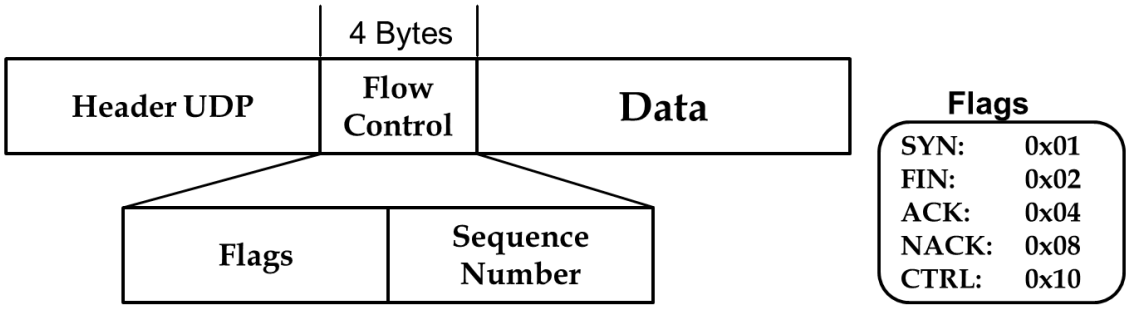


Figura 13 – Pacote *LCTCP*.

A configuração e a transmissão dos dados entre o *host* e o FPGA ocorrem conforme ilustração da Figura 14:

(1) O *hospedeiro* (*computador ligado à Internet*) envia um pacote de requisição de conexão (*SYN*).

(2) O FPGA responde (*SYN+ACK*), caso não esteja conectado a outra máquina a conexão é estabelecida.

(3) Ao receber a confirmação do FPGA o *hospedeiro* envia uma sequência de pacotes de dados (*ACK+data*), enquanto o FPGA os processa e soma uma unidade ao número de sequência.

(4) Em caso de erro o FPGA envia uma mensagem *CTRL+NACK* (do inglês, *Control and negative acknowledgment*).

(5) O envio de dados do FPGA para o *hospedeiro* se da através de (*ACK+data*) que espera um pacote de controle *CTRL+ACK* para garantir que o pacote foi recebido pela máquina hospedeira.

(6) Se houver um erro de *checksum* o *host* envia um pacote informando que ocorreu um erro *CTRL+ACK* e o FPGA retransmite o pacote.

(7) Quando não tem mais dados para enviar o *host* envia um pacote *FYN* e o FPGA responde com FYN+ACK, desta forma finalizando a conexão.

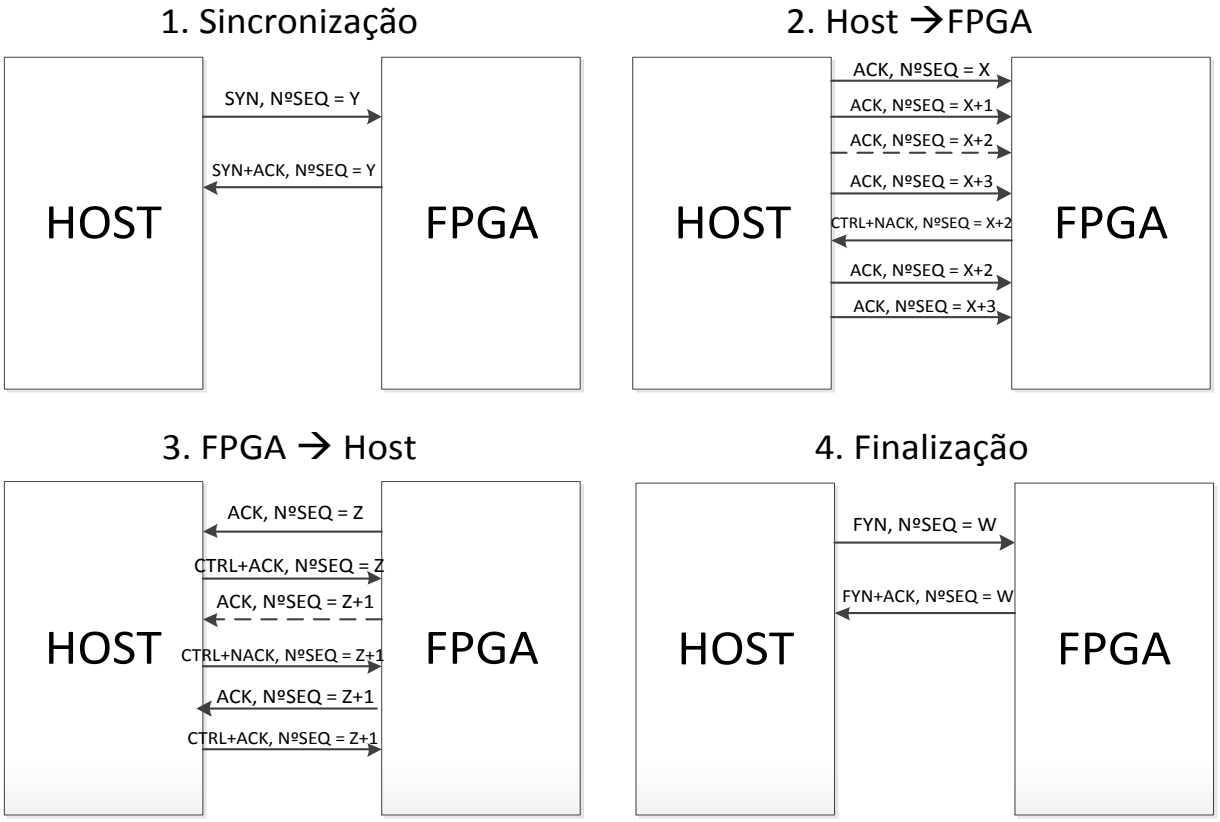


Figura 14 – Modelos de Comunicação *Host* ↔ FPGA.

2.4 **WIRESHARK**

O *Wireshark* é um capturador de pacotes que trafegam em redes. Trata-se de um aplicativo *GPL*, ou seja, é um *software* de licença livre, e está disponível em todos os sistemas operacionais

com base para *Unix* assim como para *Windows* [WIR11]. É utilizado para inspecionar redes, gerenciar pacotes, desenvolver protocolos, e etc. Através deste *software* é possível analisar dados que entram e que saem de uma determinada rede.

O principal objetivo de usar esta ferramenta neste trabalho é devido ao fato de que com ela é possível controlar o tráfego de um determinado dispositivo conectado na rede, que pode ser desde uma impressora, um computador, ou até mesmo uma placa que possua um módulo *Ethernet* acoplado, neste caso a placa da *Dini Group* que contém os FPGA*s*. O *Wireshark* foi usado para capturar os dados que entram e que saem da placa, sendo possível visualizar como é o processo de atribuir um endereço *IP* da placa e como ele encontra o endereço físico (*MAC*) desta mesma placa.

3 INTEGRAÇÃO ETHERNET COM EMULATOR

Esse trabalho consiste no desenvolvimento de um sistema de comunicação entre dois *hardwares*, o protocolo *LCTCP* desenvolvido por [SIL11] e o módulo de emulação de *hardware* baseado em cadeias de *scan* [CAS10]. Este Capítulo apresenta assim a primeira *contribuição* do trabalho, correspondendo à integração entre os mesmos. A Figura 15 apresenta, de forma resumida, a arquitetura de todo o projeto, sendo que as contribuições deste trabalho são: *Glue_RX*, *Glue_TX* e o *Software* de Controle. O Módulo *Ethernet* foi desenvolvido por [REI09] com aperfeiçoamento por [SIL11] e o Emulador foi desenvolvido por [CAS10].

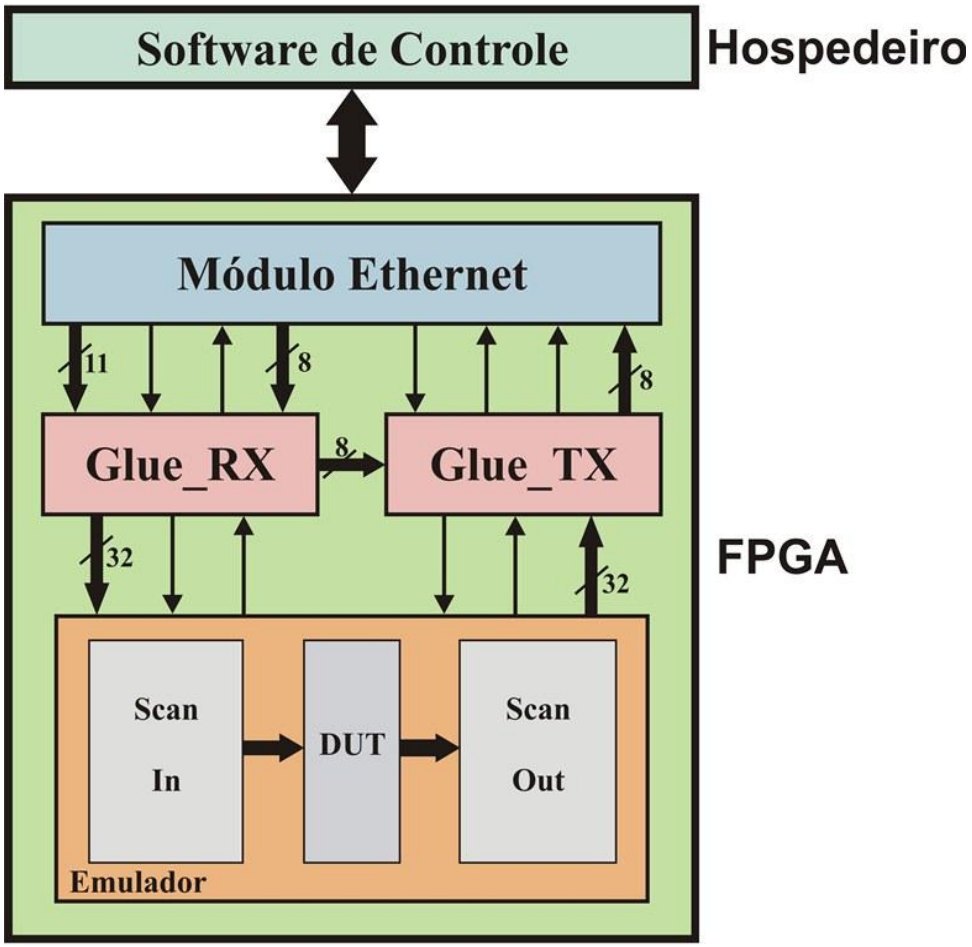


Figura 15 – Arquitetura do projeto.

Este Capítulo descreve características da arquitetura de *hardware* desenvolvida durante a elaboração desse trabalho de conclusão de curso, *Glue_TX* e *Glue_RX* apresentados na Figura 15. A seção 3.1 apresenta a máquina de estados do módulo *Glue_RX*, que manipula os dados oriundos do *LCTCP* e como são encaminhados para as cadeias de *scan* de entrada do *Emulator*. A seção 3.2 descreve as máquinas de estados do módulo *Glue_TX* e como estas manipulam os dados que vêm das cadeias de *scan* de saída do *Emulator* e envia para o *LCTCP*.

3.1 COMUNICAÇÃO ETHERNET-EMULADOR

Como cada camada do protocolo *TCP/IP* já tem por objetivo tratar partes do cabeçalho separadamente, o que chega até o bloco de *hardware Glue_RX* é somente dado. O bloco *Glue_RX* tem a função de dividir os pacotes de dados em palavras de 32 *bits* cada e enviá-los para o *Emulator* baseado em cadeias de *scan*. Sendo assim o recebimento dos dados oriundos do módulo *Ethernet* é feito pelo bloco de *hardware Glue_RX*, desenvolvido durante a elaboração desse TCC. A máquina de estados que controla esse *hardware* é ilustrada na Figura 16.

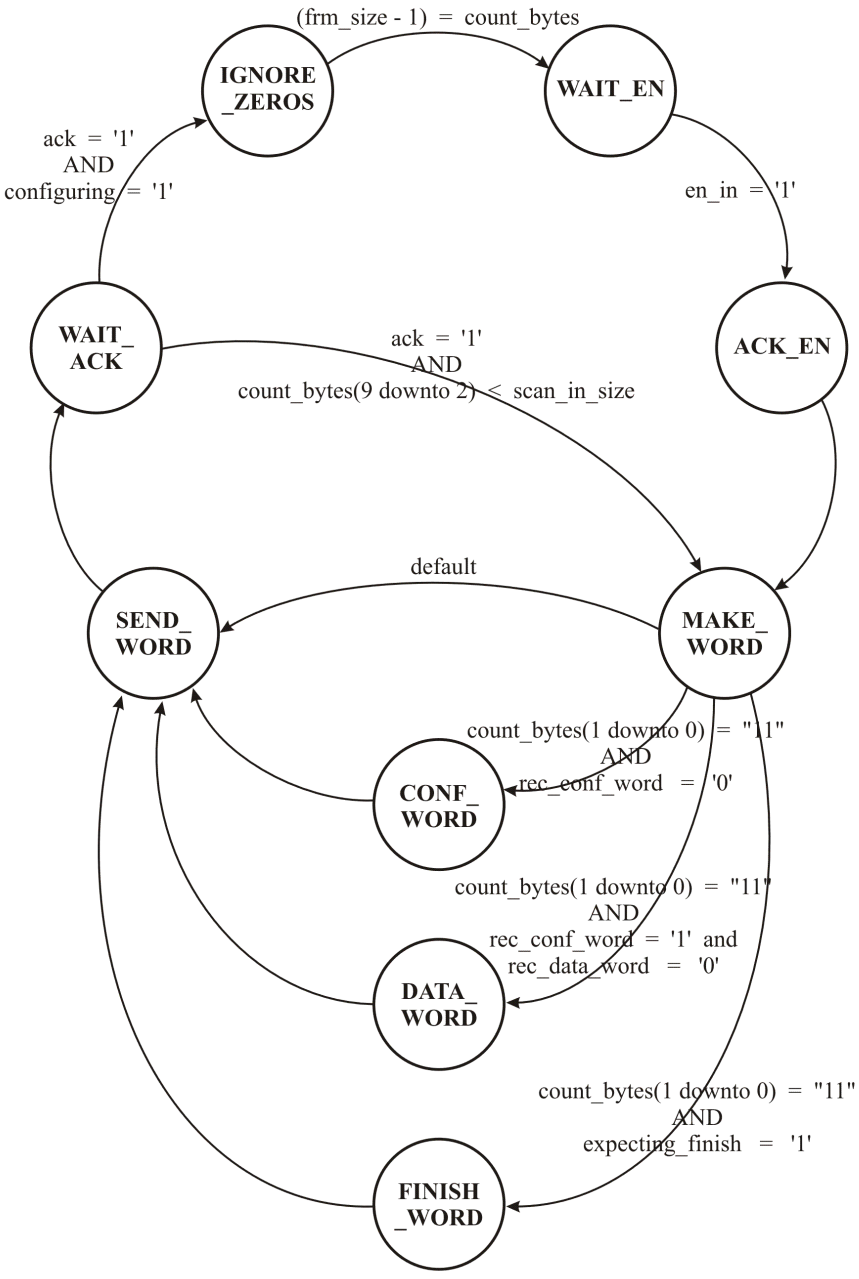


Figura 16 – Máquina de estados que recebe dados do *LCTCP* e envia para o *Emulator*.

A máquina de estados que controla o recebimento dos dados do *Ethernet* e a transmissão desses para o *Emulator,* representada na Figura 16, inicia no estado *WAIT_EN* onde fica aguardando o recebimento de um pacote válido. O primeiro pacote válido precisa necessariamente conter informações de configuração das cadeias de *scan*, estas configurações são: (i) quantidade de ciclos de clock que o *DUT* ficará ativo; (ii) tamanho das cadeias de *scan* de saída; e (iii) tamanho das cadeias de *scan* de entrada. O segundo pacote válido deve conter a quantidade de pacotes com estímulos que o *Emulator* irá receber durante o processamento do *DUT*.

Após esses dois pacotes com configurações, sinais são ativos para indicar que o *Emulator* já recebeu as configurações necessárias. Logo todos os demais pacotes são estímulos para testar o *DUT*, com exceção do último pacote que serve para indicar final de estímulos para o *DUT*.

Para cada pacote válido que chega ao bloco de *hardware Glue_RX*, a máquina passa para o estado *ACK_EN*. Nesse estado é verificado se a quantidade de pacotes recebidos já atingiu a quantidade de pacotes esperada, indicada no segundo pacote. Caso isso ocorra significa que todos os estímulos que o *DUT* esperava receber já foram recebidos. Então para esse caso o estado *ACK_EN* desativa os sinais que indicavam que o *Emulator* já teria recebido as palavras de configuração e ativa o sinal que indica final de estímulos para o *DUT*, possibilitando assim que novo *DUT* seja inserido para teste.

Depois do estado *ACK_EN* a máquina passa para o estado *MAKE_WORD*, onde tem a função de concatenar *byte* a *byte* até que uma palavra de 32 *bits* seja formada e assim possa ser enviada para o *Emulator*. Também nesse estado a máquina verifica se a informação que acabara de chegar é: (i) uma palavra com estímulos que será enviada para o *Emulator*, então a máquina passa para o estado *SEND_WORD*; (ii) o primeiro pacote, ou seja, de configuração das cadeias de *scan*, então o próximo estado passa a ser *CONF_WORD*; (iii) o segundo pacote, ou seja, uma palavra com a informação de quantos pacotes de dados *Ethernet* serão enviados ao *Emulator*, então o próximo estado será *DATA_WORD*; (iv) o último pacote, então o próximo estado passa a ser *FINISH_WORD*.

A partir dos três últimos estados a máquina passa para o estado *SEND_WORD* onde o último byte da palavra que estava sendo montada é concatenado com os outros três *bytes*, isso é necessário para poder descer o sinal de leitura da fila um ciclo de relógio mais cedo, e assim evitar problemas com recebimento de dados inválidos. Após a cópia do último *byte* e consequentemente uma palavra de 32 *bits* formada, nesse estado também ocorre o envio dessa palavra para o *Emulator*.

Com a palavra de 32 *bits* já enviada ao *Emulator*, a máquina passa para o estado *WAIT_ACK*. Nesse estado a máquina fica aguardando que o *Emulator* execute o *DUT* pela quantidade de ciclos de *clock* que o usuário escolheu, e avise quando seu processamento estiver terminado. A partir desse

estado a máquina pode seguir para o estado: (i) *MAKE_WORD*, onde a máquina inicia a montagem da próxima palavra com dados a ser enviada ao *Emulator*. Observe que se isso ocorrer a máquina não terá passado pelo estado *WAIT_EN*, isso significa que a nova palavra a ser montada ainda faz parte do mesmo pacote ao qual pertencia a última; (ii) *IGNORE_ZEROS*, nesse estado são ignorados os zeros acrescentados de *padding* no pacote *Ethernet*. Esses zeros fazem parte do pacote apenas para ter a quantidade mínima de 64 bytes que a *Ethernet* suporta, após ignorar os zeros a máquina volta para o estado *WAIT_EN* para aguardar um novo pacote com dados válidos.

Então pode se dizer que essa máquina de estados completa um ciclo para cada palavra de dados que é enviada para o *Emulator*, e está sempre ativa enquanto o módulo *Ethernet* não terminar o envio de todos os pacotes.

3.2 COMUNICAÇÃO EMULADOR-ETHERNET

Após o processamento dos estímulos do *DUT*, as cadeias de *scan* de saída enviam os resultados em palavra de 32 *bits* para o módulo *Glue_TX*. Este módulo captura os dados das cadeias de *scan* e os envia ao módulo *Ethernet*. O *Glue_TX* utiliza uma máquina de estados de recepção, *s_receive*, para separa a palavra de 32 *bits* em palavras de 8 *bits*, um *buffer* de memória é utilizado para facilitar o sincronismo, pois a quebra da palavra de 32 *bits* consumirá alguns ciclos de *clock*. Após a quebra em quatro partes estas são enviadas ao módulo *Ethernet*, controlado por outra máquina de estados, *s_send*.

A Figura 17 apresenta o esquemático deste módulo, com seus respectivos sinais de dados e controles do *Glue_TX*. Este esquemático foi extraído da ferramenta ISE durante a prototipação.

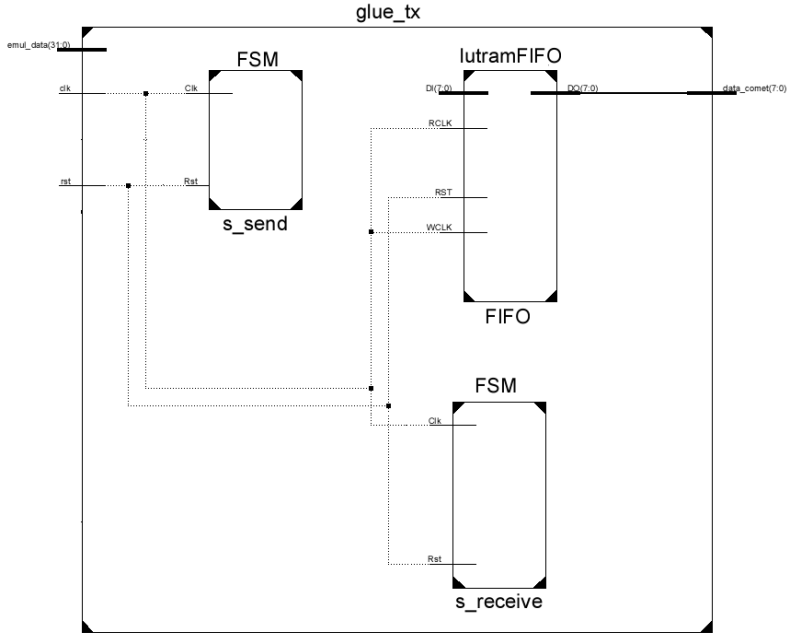


Figura 17 – Detalhamento do módulo *Glue_TX*.

O esquemático da Figura 17 apresenta de forma global o funcionamento do *Glue_TX*. Os dados entram através da porta *emul_data* e são armazenados na *lutramFIFO*, sendo o controle realizado através da máquina de estado *s_receive*, que será mencionada mais adiante. Os dados entram no *buffer* através da porta *DI* da *FIFO* e são retirado através da porta *DO*, que é controlado pela máquina de estado *s_send*, estes dados são colocados na porta de saída, *data_comet*, do *Glue_TX*. Os 3 módulos são controlados pelo mesmo sinal de relógio *clk* e pelo mesmo sinal *rst*, sendo que a *lutramFIFO* utiliza dois sinais de relógio um para leitura, *rclk*, e outro para escrita, *wcrk.*

A Figura 18 apresenta a máquina de estado responsável pelo recebimento dos dados que vêm das cadeias de *scan*, separa em palavras de 8 *bits* e insere num *buffer*. A recepção funciona da seguinte forma: a máquina de estado inicia no estado *IDLE* e passa incondicionalmente para o estado *WAIT_EN*. Neste estado ela desabilita a escrita na fila e fica aguardando até receber um dado válido. Um vetor de 32 bit, *emul_data*, acompanhado pelo sinal *data_av_i*, que indica dado válido, chegam ao *Glue_TX*, se a fila não estiver cheia este estado habilita a escrita na fila e passa para o estado *WF1*. Este estado, por sua vez, coloca os oito *bits* menos significativos, do bit 24 até o bit 31, na fila e passa para o estado *WF2*. Da mesma forma que o estado anterior, este estado, coloca os próximos 8 *bits* na fila, desta forma vai armazenando todos os 32 *bits* em vetores de oito *bits* na fila.

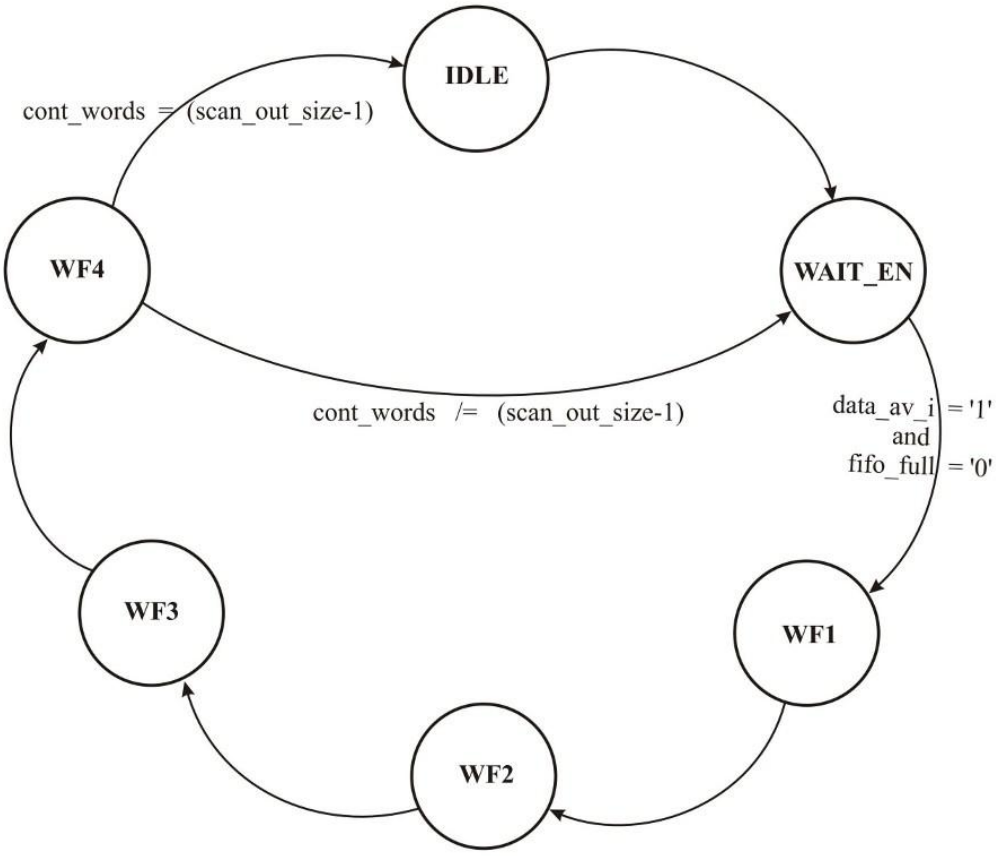


Figura 18 – Máquina de estados de recebimento dos dados do *Emulator.*

Quando chega ao último estado de escrita, *SW4*, a máquina escreve os últimos *bits* na fila, desabilita a escrita e manda um *ACK_O* para o *Emulator* avisando que pode enviar mais um vetor de 32 *bits*, pois o mesmo já foi armazenado. Das cadeias de *scan* vem um vetor que contém o número de palavras de 32 *bits*, se a contagem de palavras já atingiu este número então passa para o estado IDLE, que coloca o contador em zero, e o processo se repete novamente, porém se a contagem não atingiu este valor deste vetor o contador *cont_words* é incrementado e o próximo estado será o estado WAIT_EN.

A máquina de estados utilizada para envio de dados para o *Ethernet* pode ser vista na Figura 19. Esta máquina é composta por apenas três estados e se comporta da seguinte forma: assim que o sinal de *reset* baixar a máquina atinge o estado *IDLE* e permanece até que o sinal *done* fique em nível lógico baixo e que o sinal *can_transmit* fique em nível lógico alto, onde *done* é um sinal que vem do módulo *Ethernet* avisando que pode transmitir dado e *can_transmit* é um sinal gerado pela máquina de estado de recepção *s_receive* e serve para avisar a esta máquina que pode transmitir. Desta forma a transmissão é feita somente se tem dado para transmitir e se o receptor está preparado para receber. Estabelecida as condições mencionadas à máquina vai para o estado *SEND_WORD* e permanece enviando dados até que a fila fique vazia, *fifo_empty* com nível lógico alto. O dado é retirado automaticamente da fila e transmitido para o *Ethernet* através da habilitação da leitura da fila. No momento que a fila fica vazia a máquina automaticamente atinge o estado *WAIT_DONE*, onde permanece até que *done* fique em nível lógico alto, ou seja, que o receptor mande um aviso que já pode enviar dados. Durante a permanência neste estado um sinal *ACK* é enviado ao receptor, *Ethernet*, solicitando a liberação do *done* para começar o ciclo novamente, se o *done* aparecer (nível lógico alto) a máquina atinge o estado *IDLE* e o ciclo se repete novamente.

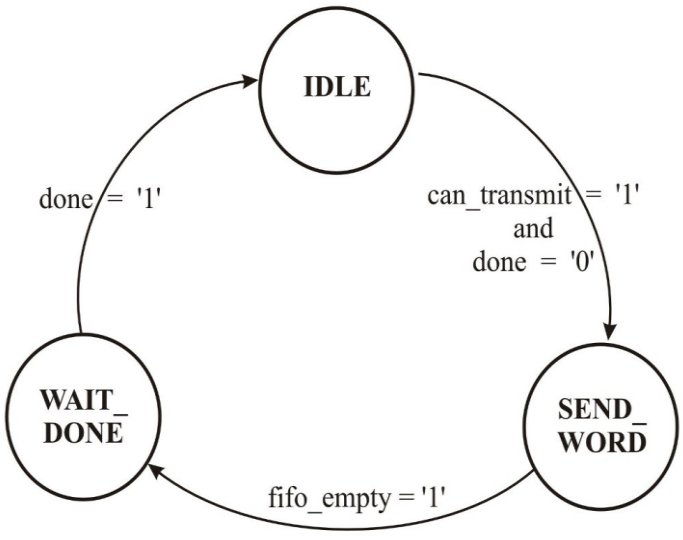


Figura 19 – Máquina de estados de envio de dados para o *Ethernet*.

4 SOFTWARE DE CONTROLE

Este Capítulo descreve o *software* desenvolvido em linguagem de alto nível *C#* e que estabelece uma comunicação entre o hospedeiro e o FPGA. A Figura 20 ilustra o processo de conexão, envio e recebimento de dados entre o FPGA e o computador hospedeiro. Estes processos são controlados pelo *software* em questão e que pertence à camada de aplicação do protocolo *TCP/IP*. Esse *software* permite conectar, desconectar, descobrir o endereço *IP* da placa que contém o FPGA que está sendo utilizado, enviar e receber pacotes.

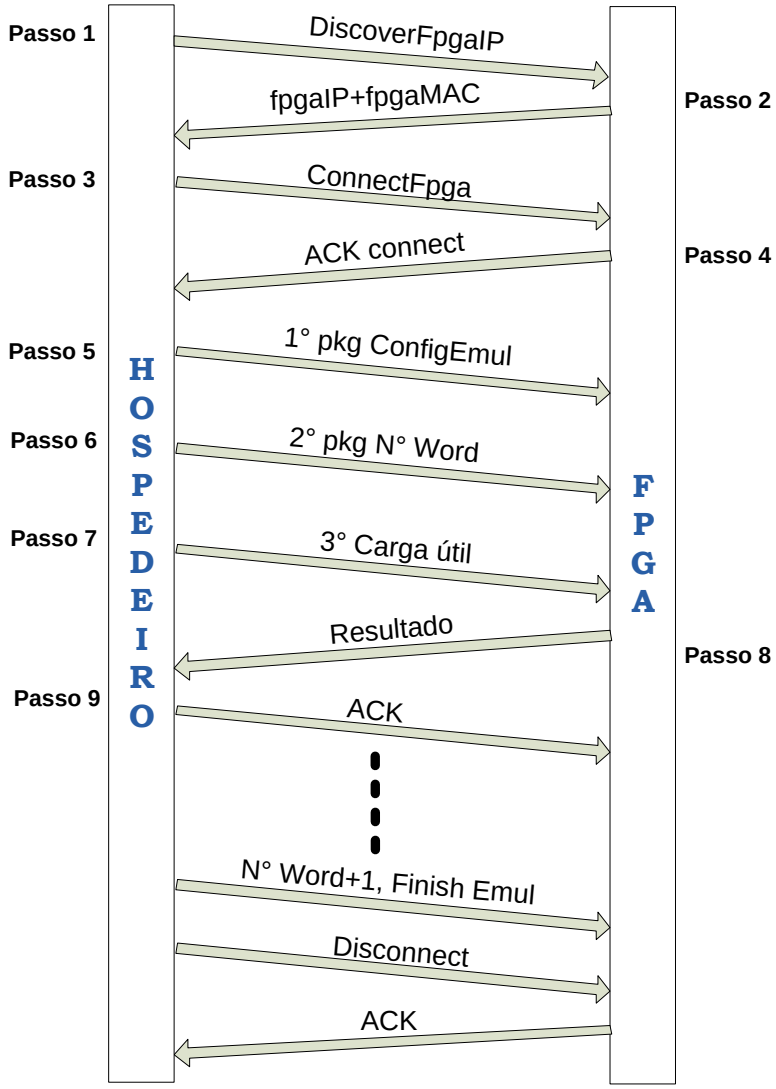


Figura 20 – Processo de conexão e transferência de dados com FPGA.

Para que uma conexão entre uma máquina transmissora e uma receptora funcione, tanto em protocolo *UDP* quanto em protocolo *TCP*, é necessário a definição de pontos terminais (neste caso computador hospedeiro e placa com FPGA), estes pontos serão definidos com uso de soquetes. Cada soquete tem um número que consiste no endereço *IP* do host mais o número da porta para este

host. Em cada extremidade existe um soquete que possui um número específico identificando o soquete do serviço requisitado. O serviço, neste caso, é a aplicação responsável por transmitir e receber dados do FPGA.

A transmissão e o recebimento dos dados são feitos de forma concorrente, para tal é feito uso de *threads,*. O sincronismo entre as duas *threads* é feito com uso de semáforos, que tem como objetivo tratar o problema de exclusão mútua, ou seja, que as duas *threads* possam usar os mesmos recursos.

Após a conexão as tarefas se dividem em duas *threads* para que as funções de envio e recepção possam ocorrer simultaneamente. O programa envia pacotes de dados, faz controle de sequência e controle de *flags*. O controle de sequência é feito para que quando ocorrer um erro no FPGA o *software* possa retransmitir o pacote, as *flags* são utilizadas para que o receptor saiba que tipo de pacote se trata, conforme pode ser visto na Figura 13.

Um arquivo texto armazena os dados que serão manipulados pelo *DUT,* conforme estes dados são lidos, linha-a-linha, os mesmos são concatenados conforme os protocolos de cada camada, ou seja, no pacote *IP* deve constar todo o seu cabeçalho, assim como o pacote Ethernet e o *UDP.* Após a montagem do pacote o mesmo é enviado para seu destino. Esse programa também faz o controle de quantos dados foram enviados e quantos foram recebidos. Para que seja possível uma retransmissão de pacotes quando ocorrer um erro, foi criado um vetor de retransmissão. Quando ocorrer um erro e o FPGA requisitar um novo envio, o programa envia o conteúdo deste vetor.

A fim de estabelecer uma conexão, o *software* se comporta da seguinte maneira: no início o *MAC* destino não está sendo parametrizado, o programa manda em *broadcast* uma requisição de descoberta de *IP,* esta requisição é acompanhada de seu *IP*, porta de destino, porta de origem, e um identificador "*HEMPFPGAGAPH",* e aguarda a máquina que tem esse *IP* (neste caso o FPGA) responder com o seu *MAC.* O passo 1 da Figura 20 mostra o pacote de requisição de *IP* sendo enviado. Esta mesma função retorna o *IP* e o *MAC* da placa, passo 2 da Figura 20. Depois disso o computador hospedeiro deve armazenar esse *MAC* em uma tabela *ARP* interna para que na próxima vez que precisar enviar o pacote não seja necessário fazer uma nova requisição.

No passo 3 o *software* envia um pacote de conexão para o FPGA e este responde com um pacote de *ACK*, passo 4, desta forma é estabelecida uma conexão entre o computador hospedeiro e a placa que contém o FPGA.

Após a conexão estabelecida o hospedeiro envia 3 pacotes passos 5, 6 e 7 da Figura 20, sendo que o primeiro pacote (*pkgConfigEmul*) serve para informar as configurações do *Emulator* (quantidade de ciclos de clock que o *DUT* fica ativo, profundidade das cadeias de *scan* de saída e profundidade das cadeias de *scan* de entrada). O segundo pacote *pkgNWord* informa além do início

da transmissão de dados para o *DUT*, a quantidade de pacotes com estímulos que o *DUT* irá receber. No passo 7 então é enviado o primeiro pacote com estímulos para testar o *DUT*.

O passo 8 consiste no retorno dos dados do FPGA para o computador hospedeiro, com resultados do processamento do *DUT*. O retorno desses resultados já indica que o FPGA teria recebido os dados do passo 7, desta forma dispensando a necessidade de um pacote com *ACK*. Após receber os resultados, o hospedeiro envia um *ACK* confirmando o recebimento dos resultados, passo 9 da Figura 20. Deste ponto os passos 7, 8 e 9 se repetem até que todos os pacotes de dados sejam transmitidos e os resultados recebidos, sendo que o recebimento e o envio são controlados por *threads*. Ao término do envio de dados o hospedeiro retira o *ARP* da tabela, assim possibilitando que a placa possa ser utilizada por outro hospedeiro.

Na ocorrência de um erro, durante o envio de pacotes de dados, o FPGA transmite um pacote tipo *CTRL+NACK* (soma a *flag* que identifica o *CTRL* com a *flag* que identifica o *NACK*), o programa identifica a *flag* e retransmite o pacote conforme o número de sequência que fora solicitado no pacote *CTRL+NACK*.

A Figura 21 apresenta o diagrama de classe do *software*. Este diagrama tem por objetivo apresentar, de forma resumida, as principais atividades durante seu funcionamento. O programa começa na classe *MainClass*, onde o método *Main()* chama a método construtor da classe *ThreadManager()*, que tem como parâmetros de entrada ("*entrada_tratada.txt*", "*saida_destratada.emul*", *2*, "*eth0*", "*HEMPFPGAGAPH*", *9002, 9001*), onde *entrada_tratada.txt* é o arquivo que contém os estímulos que devem ser enviados para o *DUT*, *saida_destratada.emul* é o arquivo que conterá os dados de retorno, ou seja, o resultado do processamento do *DUT*, esse é o arquivo de saída que é utilizado para gerar formas de ondas no *GTKWave*, desta forma podendo fazer uma comparação com a simulação.

O parâmetro *eth0* é utilizado para fornecer a interface de rede onde se encontra o *software*, o parâmetro *HEMPFPGAGAPH* é mais um identificador de controle, os últimos *2* parâmetros são respectivamente porta de descoberta (utilizada para estabelecer a conexão) e porta de dados (porta de entrada de dados).

Para realizar a conexão será feito uso do método *DiscoverFPGAIp()*, que em sua execução atribui ao atributo *_FPGAIP* o endereço *IP* da placa que contém o FPGA e atribui o endereço *MAC*, da mesma placa, ao atributo *_FPGAMAC*, isto será feito na classe *communicator*. Ainda na classe *communicator***,** o método *connectFPGA()* permite verifica se não existe uma conexão, através da execução do método *isConnected()*, se não existir então faz a sincronização entre o computador hospedeiro e o FPGA. Esta sincronização é feita através do envio de um pacote *SYN* e um número de sequência, que serve para reenvio no caso de perda de pacote, conforme Figura 14 do Capítulo 2,

o número de sequência é o único parâmetro do método *connectFPGA()*.

Após o estabelecimento da conexão o método *run()* é executado na *Main()*, este método executa a *Thread receiver* e a *Thread sender* e estas, por sua vez, executam os métodos *receiver()* (responsável por receber dados) e *sender()* (responsável por transmitir dados) respectivamente nesta ordem.

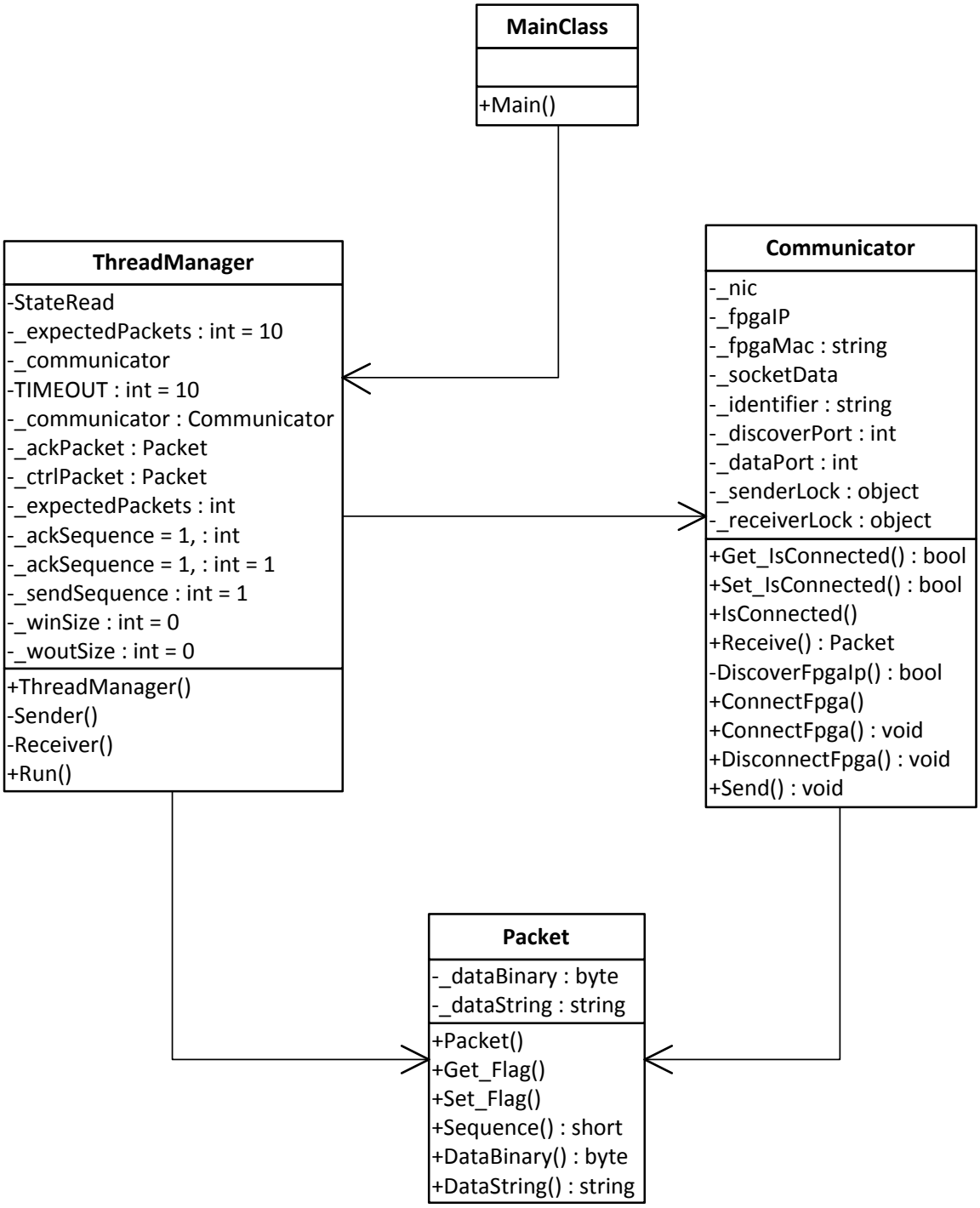


Figura 21 – Diagrama de classes do *software*.

5 TESTES DE VALIDAÇÃO

Este Capítulo tem por objetivo apresentar um processo de validação do projeto, através de simulação com o auxilio da ferramenta *ModelSim*, e por análise dos resultados diretamente no FPGA utilizando o *Chipscope Pro Analyzer*.

Antes de prosseguir com o teste de validação é preciso compreender o fluxo dos dados no projeto desenvolvido, conforme ilustrado na Figura 22. Um *software* envia dados para a rede *Ethernet* que chega ao módulo de *hardware Ethernet*, este módulo envia os dados para as cadeias de *scan* de entrada. Os mesmos dados chegam ao *DUT* que processa essas informações e entrega resultados às cadeias de *scan* de saída, que passa para o módulo *Ethernet* que envia para a rede *Ethernet*, e essa devolve para o *software*.

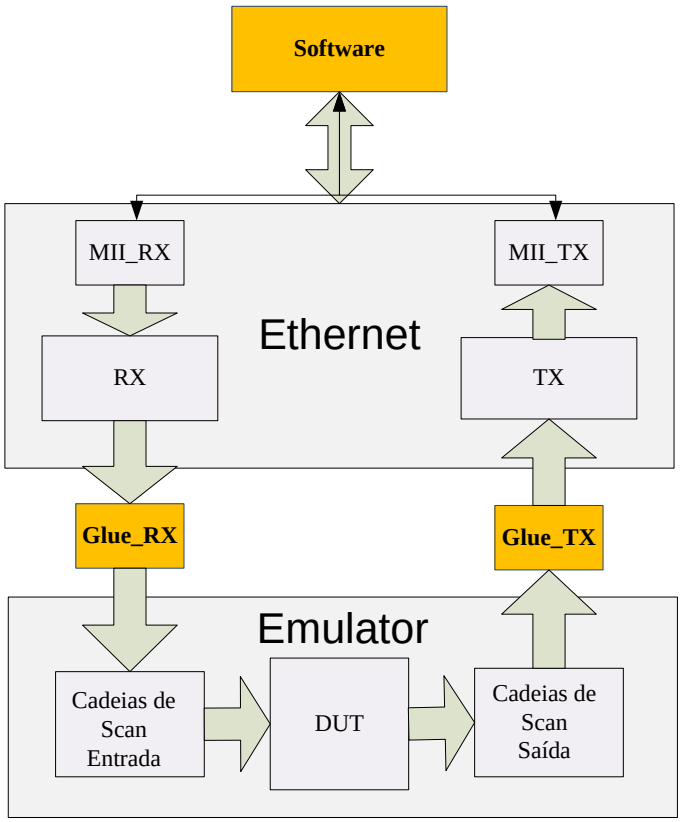


Figura 22 – Fluxo de dados do projeto.

O *software* faz leitura de um arquivo texto que contém os pacotes *Ethernet*, onde cada linha corresponde a um pacote, e envia pela rede. No módulo *Ethernet*, quem recebe os dados é o *MII_RX*, que é a interface de conexão do módulo *Ethernet* para recepção de dados. O *MII_RX* entrega o pacote *Ethernet* para as camadas *TCP/IP* que tratam da recepção, este por sua vez, é responsável por tratar os cabeçalhos dos pacotes conforme vai descendo um nível da camada *TCP/IP*, ou seja, quando o pacote chega ao *Glue_RX* só restam dados. Este é responsável por formar

pacotes de 32 *bits* e enviá-los para as cadeias de *scan* de entrada. Por fim o *DUT* é testado, e os resultados do teste são devolvidos para as cadeias de *scan* de saída, depois o processo segue no sentido contrario até chegar ao *software*.

5.1 VALIDAÇÃO POR SIMULAÇÃO DO VHDL

A ferramenta *ModelSim* (desenvolvida pela *Mentor Graphics*) foi utilizada para ver se os dados conferem com a especificação do projeto. Um pacote de dados que simula um pacote *Ethernet* foi criado com o objetivo de analisar os dados do sistema.

Na Figura 23 são apresentados vários pacotes de dados a serem simulados. Este arquivo é formado por várias linhas, onde cada linha representa um pacote *Ethernet* é composto pelos seguintes protocolos: Ethernet para a camada Física, *IP* para a camada de Rede, *UDP* para a camada de transporte.

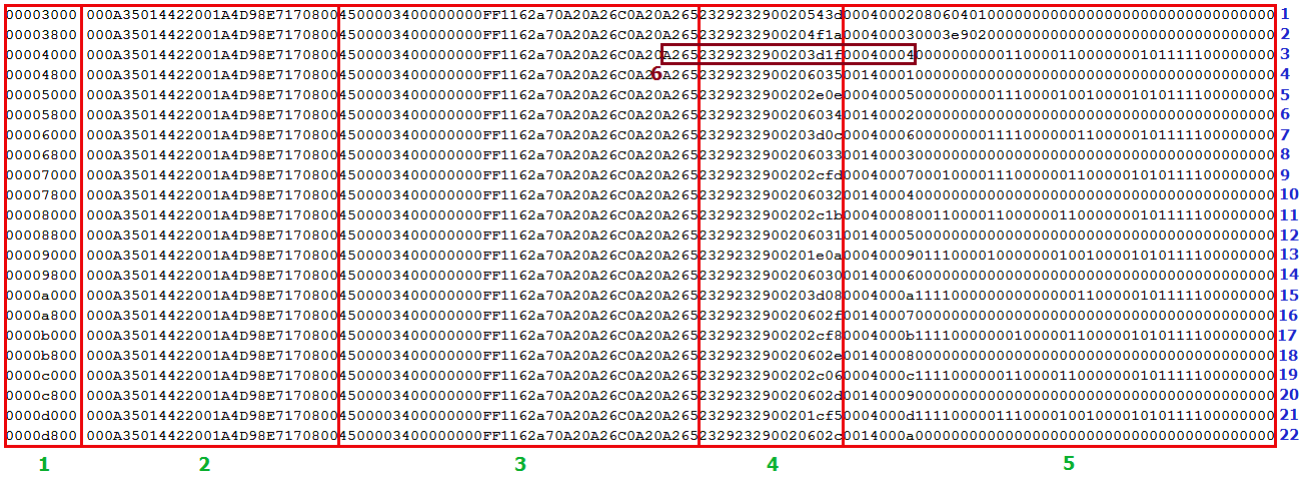


Figura 23 – Pacote *Ethernet* de entrada.

Os dados representados pela coluna sinalizada com um retângulo de número 1 servem para definir a posição de cada pacote na simulação, os dados vistos no retângulo 2 são dados do cabeçalho do pacote *Ethernet*, os primeiros 12 *bytes* servem para representar o endereço *MAC* destino e endereço *MAC* origem, respectivamente nesta ordem. Os últimos 2 *bytes* deste cabeçalho representam o tipo do pacote *Ethernet*.

O conteúdo do pacote *Ethernet* é formado pelo pacote *IP* e pelo pacote *UDP*. O pacote *IP* representado no retângulo de número 3, que é composto por 20 *bytes*, tem o seguinte formato: o primeiro *nibble* informa a versão do protocolo *IP*; o segundo *nibble* representa o tamanho do cabeçalho em 32 *bits*, que será fixo em 5; o segundo *byte* informa como o pacote deve ser tratado, ficará em zero, pois neste trabalho não será feito tratamento diferenciado de pacotes; os próximos 2 *bytes* serão utilizados para informar o tamanho de todo o pacote *IP*; os *bytes* das posições 5 a 8 servem para identificar a que “datagrama” pertence um fragmento recebido e assim controlar a

fragmentação, como os pacotes são de tamanho reduzido não há necessidade de fragmentar, ou seja, preencher com zeros; o *byte* de número 9 é utilizado para controlar o tempo de vida de um pacote, desta forma se não encontrar seu destino após um determinado número de saltos (cada roteador representa um salto), o mesmo é destruído; o décimo *byte* é usado para informar o processo de transporte que deve ser aplicado ao pacote *IP*; os *bytes* de número 11 e 12 são utilizados para armazenar o *Header Checksum*, cálculo de verificação de erro nos dados do cabeçalho do pacote *IP*; por fim os próximos 8 *bytes* são utilizados para informar o endereço *IP* origem e endereço *IP* destino respectivamente.

Os próximos dados estão representados por 2 retângulos, o de número 4 e o de número 5, que são respectivamente o cabeçalho do pacote *UDP* e os dados deste pacote. O cabeçalho *UDP* é formado pelos campos, porta *UDP* de origem, porta *UDP* destino, comprimento do pacote *UDP* e *Checksum* do pacote *UDP*. Os primeiros 4 bytes são usados para definir endereçamento por portas do *UDP*, os próximos 2 *bytes* são usados para definir o comprimento total do pacote e os *bytes* na posição 7 e 8 são usados para definir o cálculo de verificação de erro no pacote *UDP*, ou seja, o cálculo de *Checksum*.

Por fim, na coluna de número 5 temos os dados válidos, ou seja, os dados que serão utilizados para teste. Ainda dentro do pacote de dados *UDP*, encontra-se o protocolo *LCTCP* desenvolvido por [SIL11], onde o primeiro *byte* é preenchido com zeros e o segundo serve para definir se o dado é de controle, *ACK*, *NACK* ou dados. Neste caso se percebe que os dados estão intercalados por 2 tipos, 0x04 representando pacote de dados e 0x14 representando pacote de *ACK*. Para cada pacote de dado válido tem-se um *ACK* de resposta, por este motivo eles são intercalados, o pacote de *ACK* na linha 4 pertence ao pacote de dado na linha 3. Por fim, os últimos 2 *bytes* de protocolo são utilizados para sequenciar os dados, ou seja, cada linha da Figura 23 tem um número que corresponde a sequência do dado ou a sequencia do *ACK*. Resta, portanto, neste teste, 20 *bytes* que correspondem a dados propriamente.

É importante destacar que antes de começar a enviar dados existe um processo de sincronização, que funciona da seguinte forma: o *hospedeiro* envia um pacote de requisição de conexão (*SYN*) e o FPGA responde (*SYN+ACK*), caso não esteja conectado a outra máquina. Ao receber a confirmação do FPGA, o *hospedeiro* começa a enviar dados. O processo de sincronização, que pode ser visto na Figura 14, é feito através da inserção de pacotes que antecedem os dados da Figura 23. Os 2 primeiros pacotes parecem ser dois pacotes de dados, pois o segundo byte é 0x04, no entanto o primeiro pacote serve para configuração do *Emulator* e o segundo serve para determinar a quantidade de pacotes que serão enviados. Somente após esses dois pacotes é que se inicia a transmissão de dados para o *DUT*, intercalando entre eles os pacotes de *ACK*.

Na Figura 24 temos as formas de ondas com as portas dos blocos desenvolvidos e alguns dos seus sinais internos, bem como sinais da interface *MII*, esta interface controla o recebimento e envio de pacotes pela rede, percebe-se que na linha representada pelo vetor *MII_RXD_O* entre as duas tarjas verticais (letra “A” e letra “B”) têm 10 pacotes de dados e na linha representada pelo vetor *MII_TXD_O* têm apenas 5 pacotes, isto se deve ao fato de que no primeiro, por se tratar de um dado de recepção, tem um pacote de dados e outro de *ACK*, enquanto que no segundo só tem dados sendo enviado para a rede.

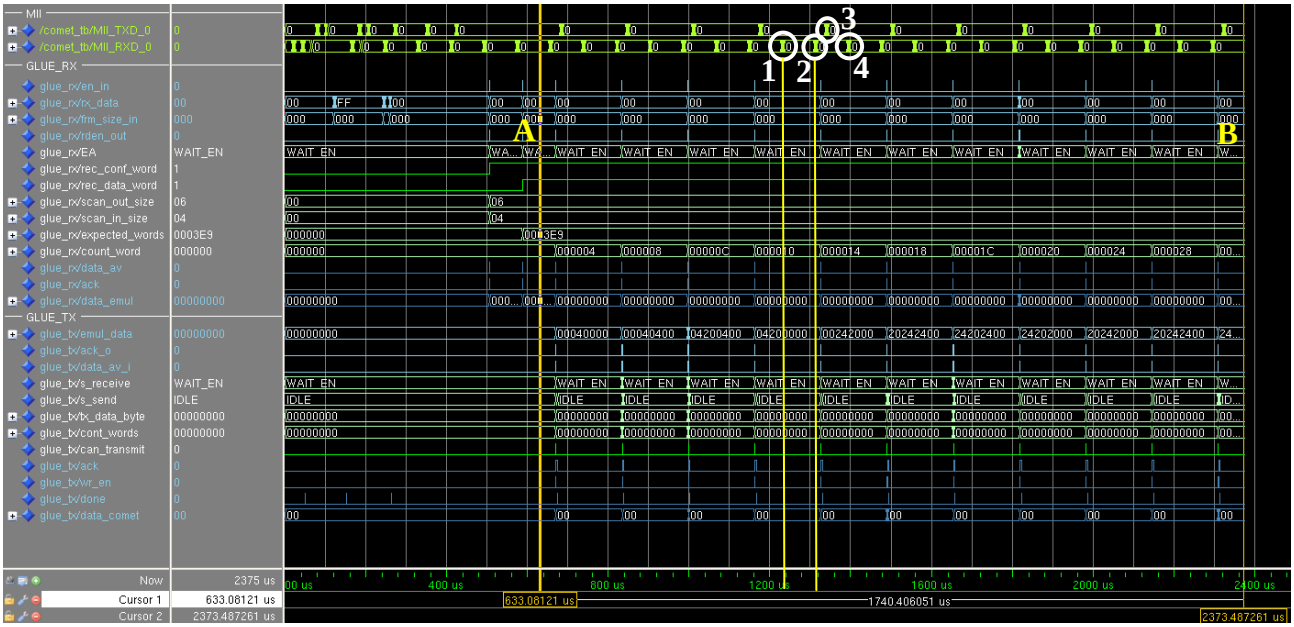


Figura 24 – Formas de ondas com sinais dos blocos desenvolvidos.

Para dar maior compreensão, na Figura 24 foram numerados 4 pacotes representando chegada e saída no *MII*. O pacote de número 1 representa um *ACK* correspondente ao pacote de dados anterior. O pacote de número 2 é um pacote de dados que chega no *MII*, ou seja, que chega ao módulo *Ethernet*, no número 3 é representado os dados de retorno no *MII* (enviado para o mundo externo), no número 4 é apresentado um pacote de *ACK*, confirmando o recebimento do pacote de número 3. Percebe-se que na linha vertical do número 1 o pacote que chega no *MII* é desconsiderado em *Glue_RX* e *Glue_TX*, pois não houve nenhuma alteração nas formas de ondas, ao passo que na linha de número 2 da para perceber a manipulação de dados, tanto em *Glue_RX* quanto em *Glue_TX*, isto se deve ao fato de que pacotes de *ACK*, no caso de número 1, não são transmitidos para o emulador, estes são tratados pela camada de enlace do módulo *Ethernet*, enquanto que pacotes de dados, no caso do número 2, são encaminhados para o emulador.

Na Figura 25, na Figura 26 na Figura 27 e na Figura 28 são apresentados os detalhes da Figura 24, observa-se que os pacotes são menores nos *Glue_RX* e *Glue_TX* (detalhado na Figura 26 e na Figura 27 respectivamente), pois os protocolos dos pacotes *Ethernet*, *IP* e *UDP* não entram no

Glue_RX e obviamente não são passados para o *Glue_TX*, porém ao chegar ao *MII_TX* os protocolos são inseridos novamente, desta forma o resultado pode ser enviado para a rede.

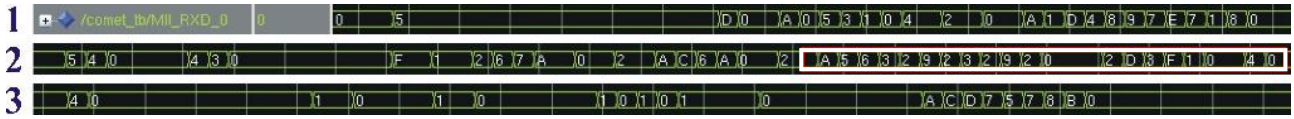


Figura 25 – Detalhes do *MII_RXD_o*, as linha 2 e 3 são continuações da linha 1.

Na Figura 25 é apresentado o detalhamento do pacote *MII_RXD_o*, vetor de dados de recepção na interface *MII*. Como exemplo, foi utilizado o primeiro pacote de dados, este pacote de dados corresponde à linha 3 da Figura 23. Para comparar os dados das duas figuras é importante notar que cada byte tem seus *nibbles* invertidos, observa-se que o primeiro byte demarcado pelo retângulo de número 6 (Figura 23) é *0xA2* e que na Figura 25 corresponde a *0x2A*.

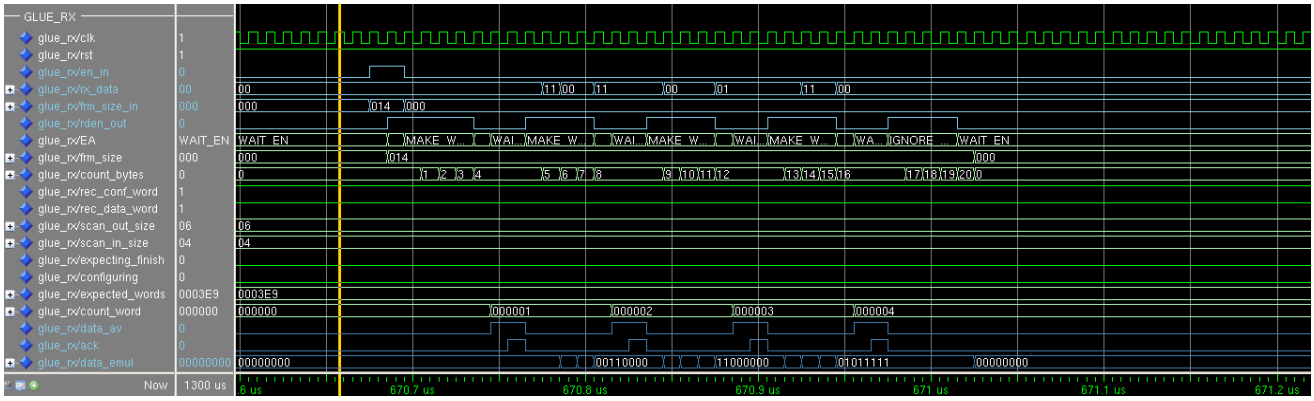


Figura 26 – Formas de ondas do *Glue_RX*.

A Figura 25 apresenta os sinais da máquina de estados e os dados manipulados por ela, conforme Glue_RX apresentado no Capítulo 3. Na Figura 25 é apresentado o pacote de dados correspondente a linha 3 da Figura 23, onde pode ser observado que os 20 *bytes* de dados são divididos em 5 pacotes de 4 bytes. O vetor *rx_data* é o receptor dos dados vindos da camada superior, ele recebe palavras de 1 byte e converte em palavras 4 bytes antes de enviar para as cadeias de *scan*. o registrador *data_emul* contém as palavras de 4 bytes que será enviadas para as cadeias de *scan*.

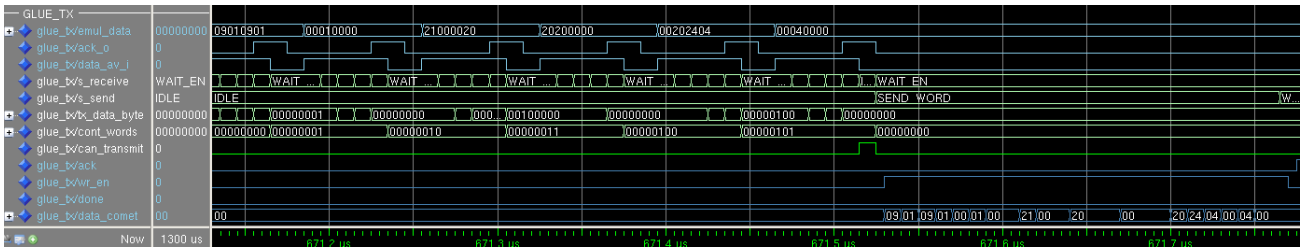


Figura 27 – Formas de ondas do *Glue_TX*.

A Figura 27 apresenta os dados e os sinais de manipulação do *Glue_TX*, conforme Capítulo 3. O vetor de recepção das cadeias de *scan* é um vetor de 32 *bits*. Percebe-se que os vetores *emul_data*, *tx_data_byte* e *data_comet* contêm os mesmos dados, porém com tamanhos diferentes e com deslocamento no tempo. O *emul_data* é um vetor de recepção (32 *bits*), *tx_data_byte* é um vetor de 8 *bits* (32 *bits* do *emul_data* dividido em 4 blocos de 8 *bits*) que será armazenado numa fila (*lutramFIFO*) e *data_comet* é um vetor de dados retirado da fila e enviados para as camadas superiores, que devem ser vistos no *MII* de transmissão, Figura 28.

Figura 28 – Detalhes do *MII_TXD_o*, as linhas 2 e 3 são continuações da linha 1.

Diferentemente da Figura 25 que constava de um pacote de dados e outro de *ACK*, em *MII_TXD_o* só têm pacotes de dados com seus respectivos protocolos. A primeira elipse destaca *0x04* que indica pacote de dados, a segunda elipse destaca *0x04* indica a sequência dos pacotes e por último a sequência de dados.

Por fim, para concluir este teste de validação por verificação em *VHDL*, é feito uma comparação entre os dados obtidos do *MII* e os dados previamente definido como dados de saída esperada, desta forma confirmando a funcionalidade do sistema.

5.2 VALIDAÇÃO POR DEBUG COM CHIPSCOPE

O processo de validação por debug consiste em analisar o resultado da operação diretamente em FPGA, para tal, será utilizado como ferramenta de verificação de sinais dentro do FPGA o *Chipscope pro Analyzer*. Também será utilizado o *software* desenvolvido neste projeto, que servira para enviar dados a partir da camada de aplicação do modelo de camadas *TCP*/IP. O visualizador de trafego, *Wireshark*, será utilizado como ferramenta para verificação dos pacotes de dados entrando e saindo no dispositivo FPGA. O *software Planahead* será utilizado para gerar o arquivo binário, visto que dentro do FPGA só é executado arquivos com esta extensão.

A Figura 29 apresenta a placa de prototipação fornecida pela *Hitech Global* que contém o FPGA da *Xilinx* da família *Virtex-5*. O arquivo binário contendo o módulo *Ethernet*, o *Emulator* e o *DUT* são descarregados neste FPGA através de uma comunicação *JTAG-USB*, retângulo A. Após o *download* do arquivo binário o *software* começa enviar dados através da comunicação *Ethernet*, retângulo B.

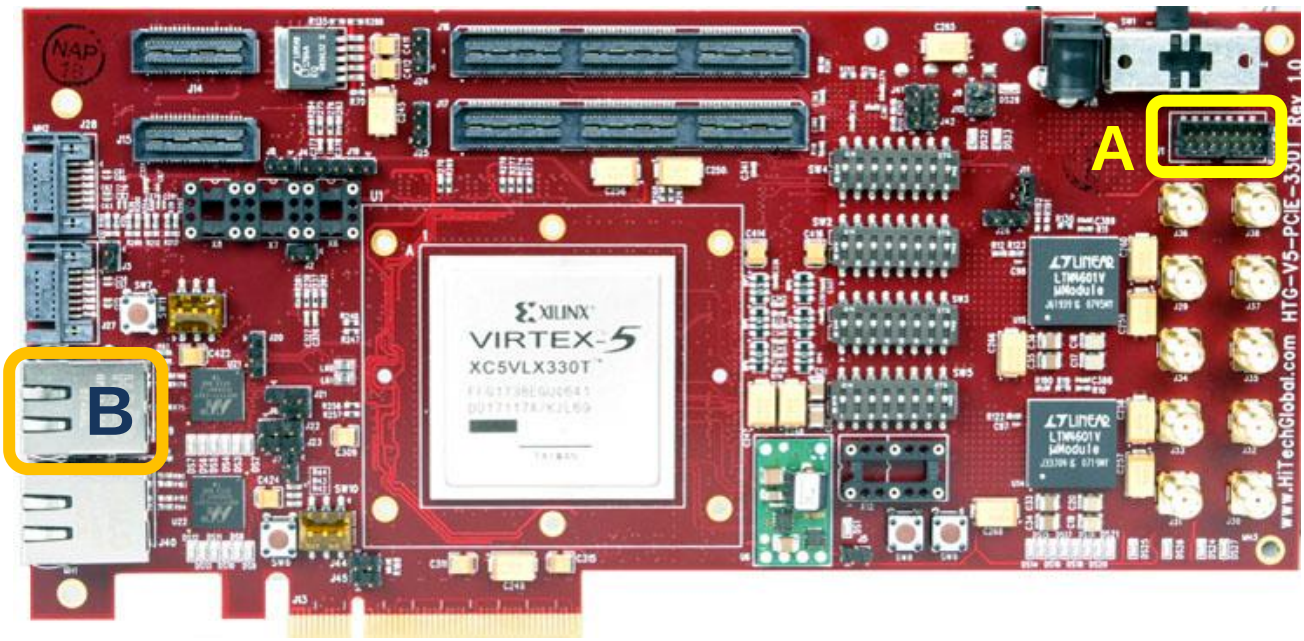


Figura 29 – Placa utilizada para prototipação com FPGA *Virtex-5 XC5VLX330T*.

O procedimento para geração de um arquivo binário é dado pela síntese lógica e síntese física que funciona da seguinte forma: (i) através do *Planahead* é realizado a síntese lógica, transformação de um código RTL (*Register Transfer Level*) para um nível de portas lógicas; (ii) com o *Chipscope Pro*, sendo executado dentro do *Planahead*, são especificados os sinais internos do FPGA a serem analisados, estes sinais são salvos em um arquivo com extensão *cdc*; (iii) após a especificação dos sinais que serão analisados é feito a síntese física; (iv) após a síntese física é gerado o arquivo binário; (v) por fim, este arquivo é baixado para o FPGA.

Com o arquivo binário no FPGA, utiliza-se o analisador lógico *Chipscope Pro Analyzer* para visualizar o comportamento dos sinais especificados no arquivo *cdc*. A Figura 30, a Figura 31 e a Figura 32 apresentam os mesmos dados de configuração e de carga útil. Na Figura 30 são apresentados os pacotes saindo do *Software* da camada de aplicação, na Figura 31 são apresentados os pacotes no *Wireshark* sendo enviados para o FPGA e na Figura 32 são apresentados os pacotes no FPGA.

O pacote “08060401”, visto como primeiro pacote na Figura 30 e no primeiro quadrado na Figura 31 corresponde a um pacote de configuração do *Emulator*, onde: “08” determina a quantidade de ciclos que o *DUT* fica ativo para processar os dados, “06” determina a profundidade das cadeias de *scan* de saída, “04” determina a profundidade das cadeias de *scan* de entrada e “01” é o campo de configuração. O segundo pacote visto nas três figuras é um pacote que informa que o próximo pacote a ser enviado ao ambiente de emulação será de carga útil, sendo que “$0003e9_{16}=1001_{10}$” serve para determinar quantos pacotes de carga útil será transmitido e “02” serve para indicar o começo da transmissão de dados úteis, estas informações podem ser melhores

observadas em [CAS10]. Os próximos “n” pacotes são intercalados entre dados (carga útil) e *ACK* respectivamente nesta ordem (ver Figura 20), onde “n” corresponde ao número de pacotes de com carga útil.

```
dbg -> ACK

dbg -> 08060401

dbg -> 0003e902

dbg -> 00000000001100001100000001011111

dbg -> ACK

dbg -> 00000000011100001001000010101111

dbg -> ACK

dbg -> 00000000111100000011000001011111

dbg -> ACK

dbg -> 00010000111000000110000010101111

dbg -> ACK

dbg -> 00110000110000001100000001011111

dbg -> ACK
```

Figura 30 – Dados enviados pelo *software* no computador hospedeiro.

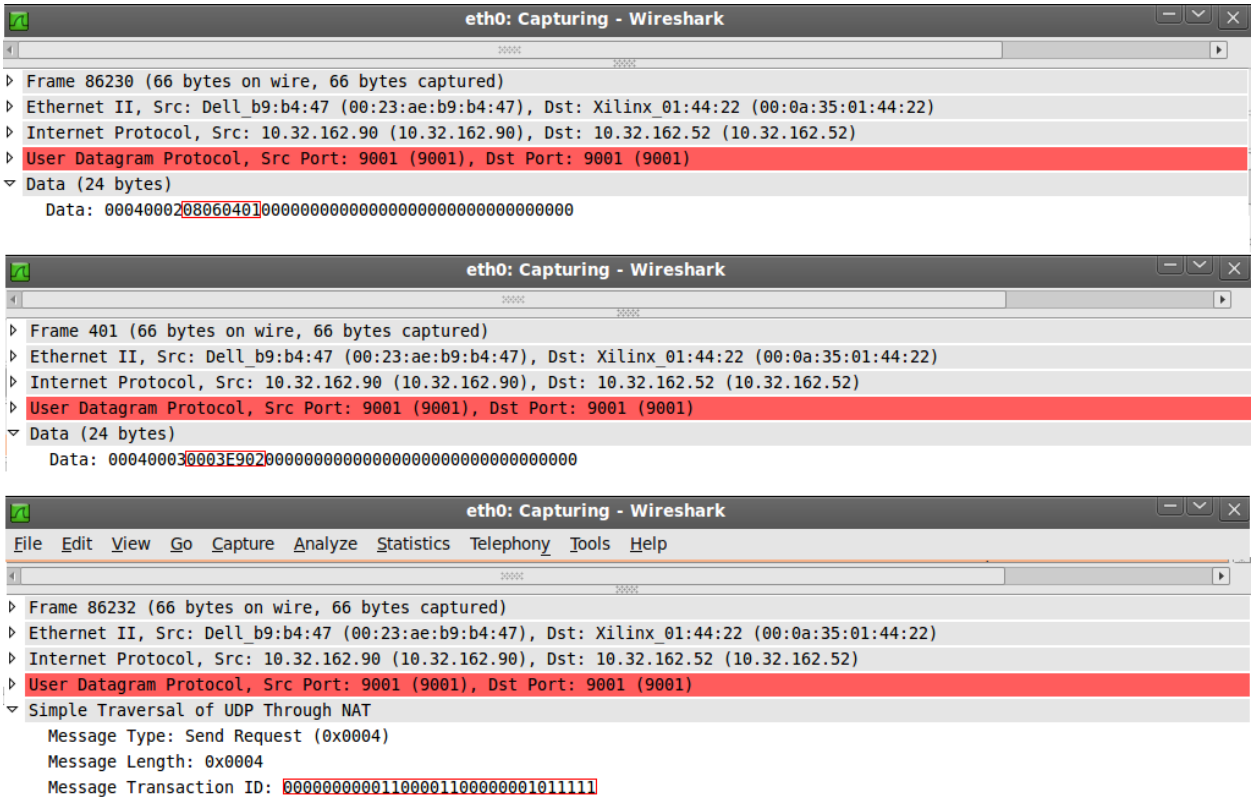


Figura 31 – Captura do *Wireshark* dos pacotes sendo enviado ao FPGA.

A Figura 31 representa a captura de dados do *Wireshark*, de cima para baixo os retângulos nos representam respectivamente os dados de: configuração, confirmando o segundo pacote da Figura 30; quantidade de pacotes, idem ao terceiro pacote da Figura 30; o primeiro pacote de carga útil, conforme o quarto pacote da Figura 30.

Nestas figuras do *Wireshark* pode ser visto que todos os pacotes são enviados para o mesmo o mesmo *IP* de destino (10.32.162.52), a mesma porta de destino (9001) e o mesmo *MAC* de destino (00:0a:35:01:44:22), confirmando que os dados são enviados para a placa da *Xilinx* que contém o FPGA.

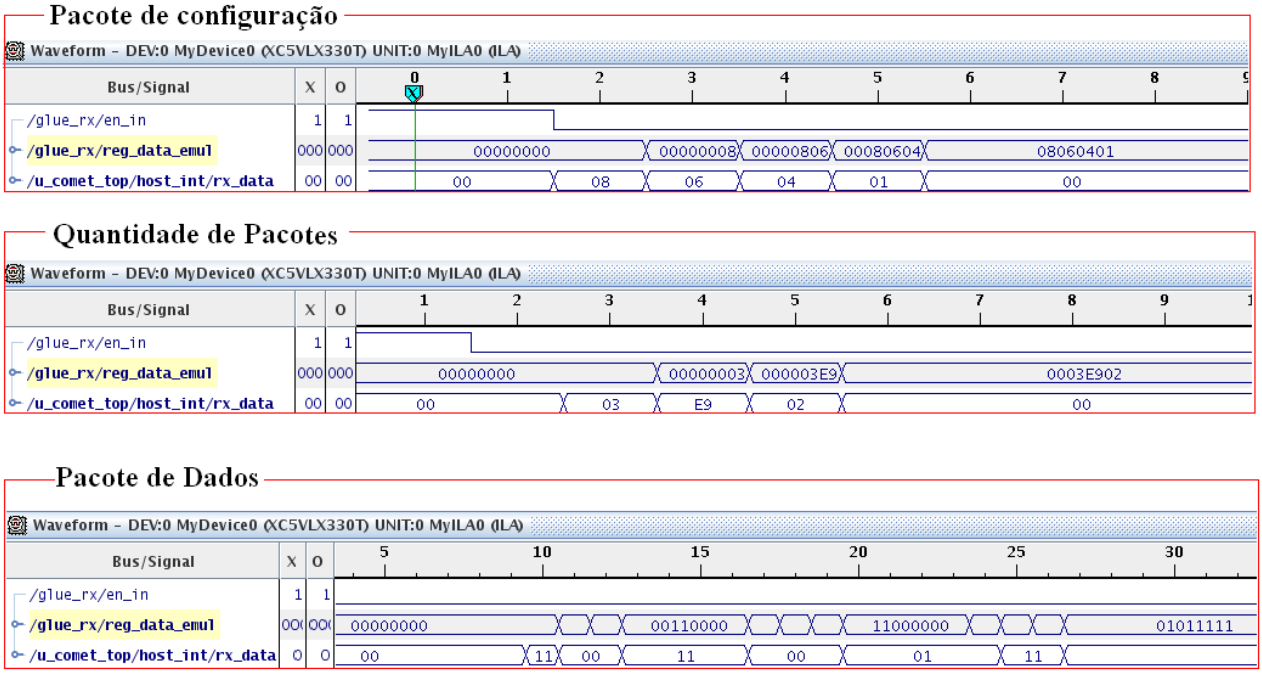


Figura 32 – Forma de ondas do *Glue_RX*, usando *Chipscope Pro Analyzer*.

O vetor *rx_data*, representado na Figura 32, corresponde aos dados vindos do *MII* em palavras de 8 *bits*, enquanto que o vetor *reg_data_emul* concatena de oito em oito *bits* em palavras de 32 *bits*, quando a palavra está completa a mesma é enviada às cadeias de *scan*. Observa-se que o vetor *rx_data* da Figura 32 é igual ao *rx_data* da Figura 26 e que o mesmo pode ser aplicado entre *data_emul* e *reg_data_emul* das duas figuras, desta forma confirmando a relação entre o teste por simulação e o teste por análise no próprio FPGA.

O *DUT* utilizado para este teste de validação é um somador de 64 *bits*, veja Figura 33. Os dados de entrada do somador (*entrada1* e *entrada2*) são os dados vindos do *Glue_RX* (*reg_data_emul* da Figura 32). Observa-se que quando o sinal de *clock* sobe a soma é realizada. O resultado da somada é colocada nas cadeias de *scan*. Para melhor compreender qual a relação de "entrada1" e "entrada2" da Figura 33 com o *reg_data_emul* da Figura 32 é preciso compreender a Figura 34, que ilustra como são inseridos os dados nas cadeias de *scan* de saída e como são retirados

das mesmas antes de enviar para o *DUT* (entrada1 e entrada2). Em [CAS10] é explicado com maiores detalhes a Figura 34 e Figura 36.

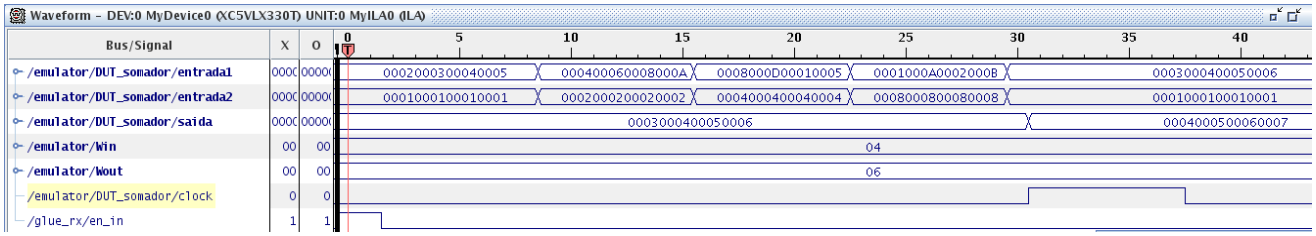


Figura 33 – Formas de ondas do *DUT*, Somador de 64 *bits*.

O somador representado na Figura 33 só serve para validar o sistema, pois se torna fácil de observar os sinais devido à simplicidade do *hardware*.

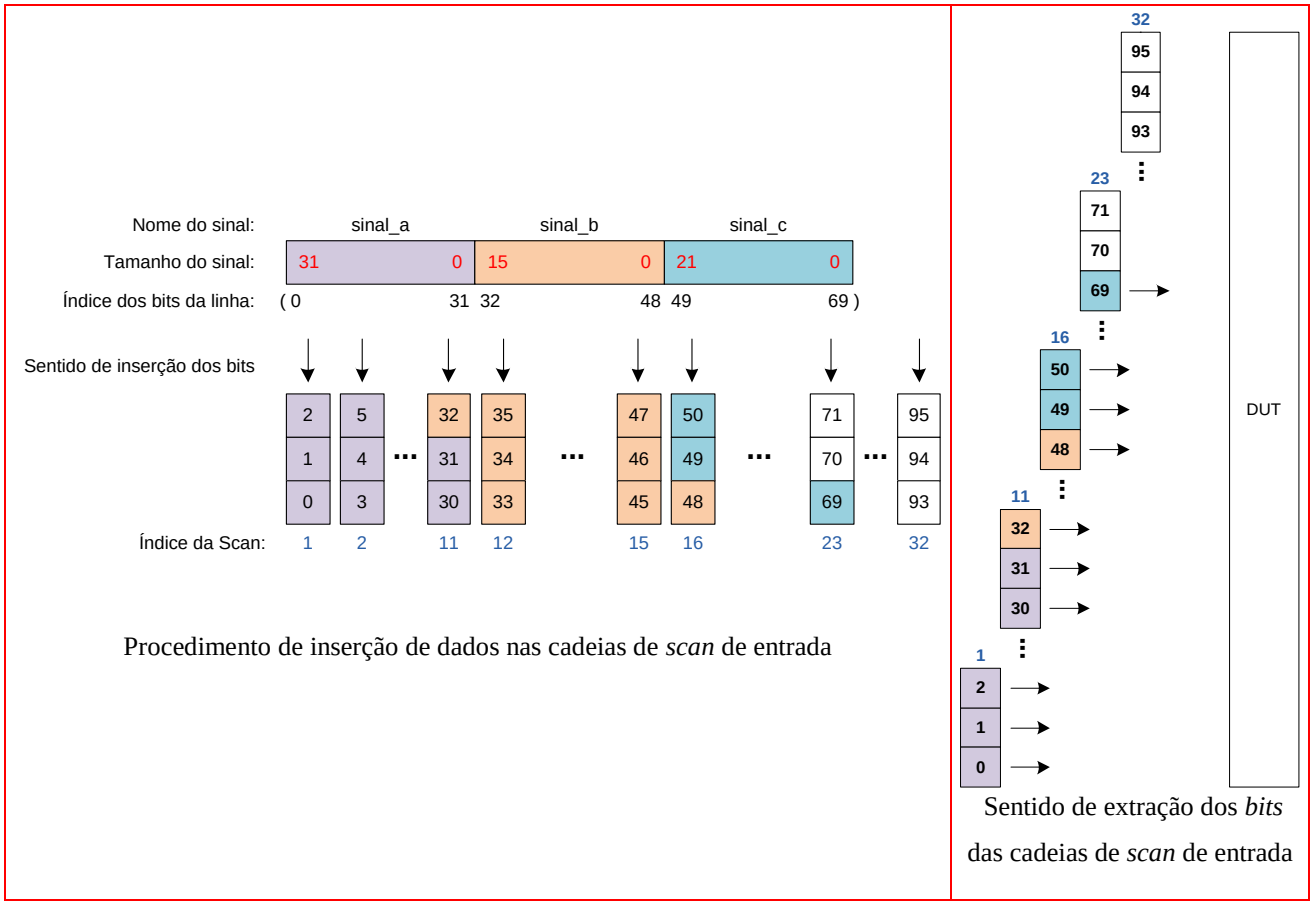


Figura 34 – Extração de dados das *scan* de entrada e inserção no *DUT*, [CAS10].

Na Figura 35 são apresentados os dados vindos do *Glue_TX*, percebe-se que o vetor *emul_data* é o resultado da soma do *DUT* (Figura 33), sendo que os dados estão embaralhados, efeito das cadeias de *scan*. A Figura 36 ilustra o procedimento de inserção e extração dos dados nas cadeias de *scan* de saída, convém uma consulta em [CAS10] para melhor compreender como os dados são embaralhados e como são desembaralhados. Ainda na Figura 35, os dados no vetor *data_comet* são enviados em palavras de 8 *bits* para o *MII*.

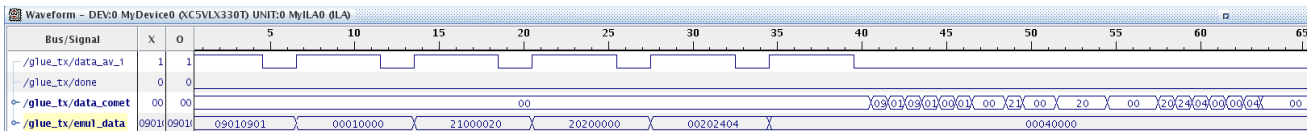
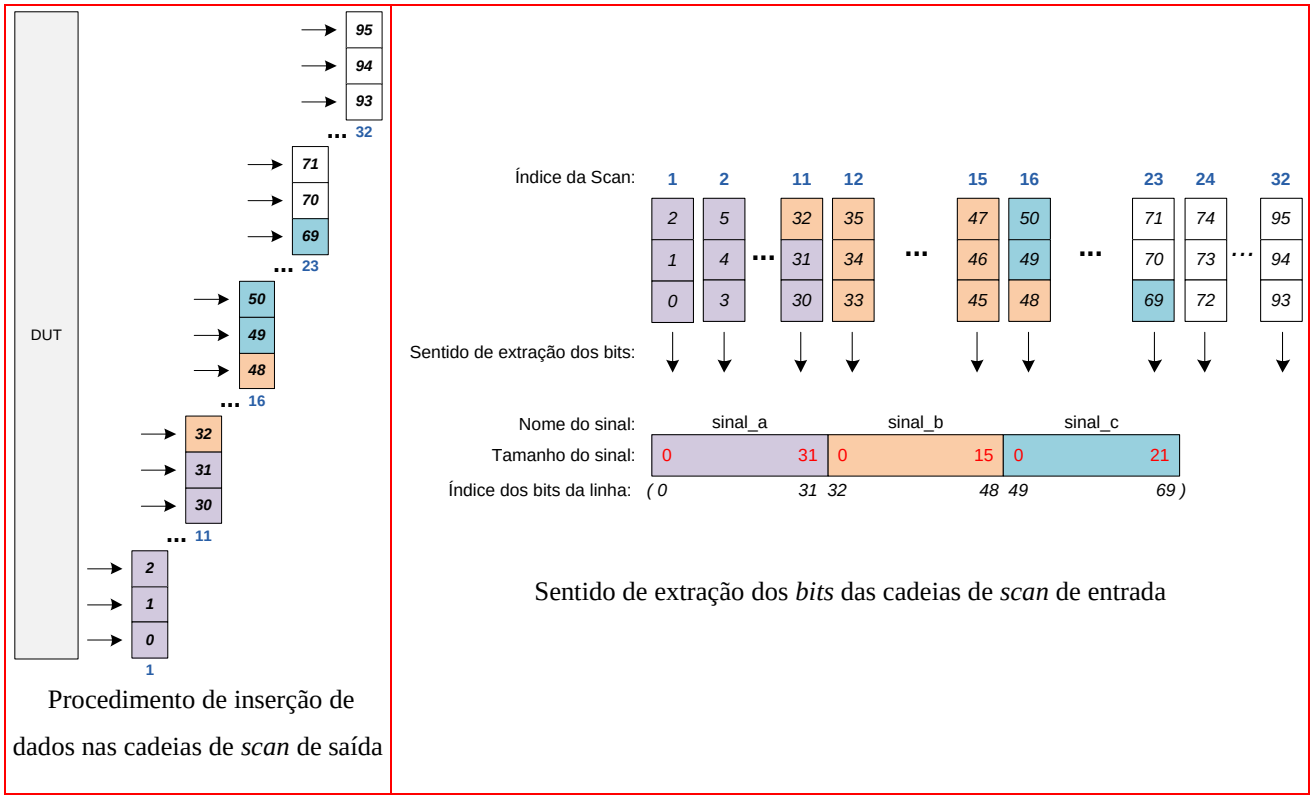


Figura 35 – Formas de ondas do *Glue_TX*, usando *Chipscope Pro Analyzer*.



Procedimento de inserção de dados nas cadeias de *scan* de saída

Sentido de extração dos *bits* das cadeias de *scan* de entrada

Figura 36 – Inserção de dados nas *scan* de saída a partir do *DUT*, [CAS10].

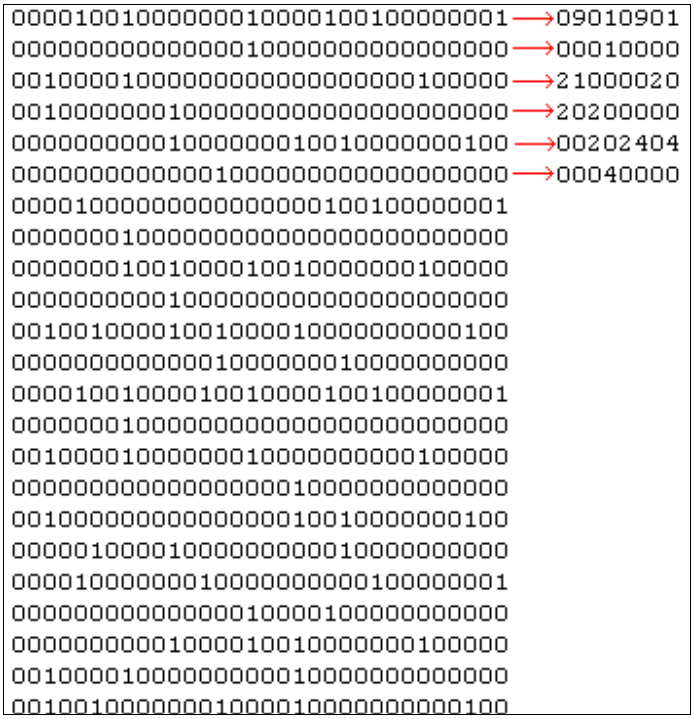


Figura 37 – Dados de retorno.

A Figura 37 apresenta os dados de retorno, percebe-se que estes dados são os mesmos dados visto na Figura 35, no vetor *emul_data*. Portanto comprovada a comunicação entre um computador hospedeiro e o *Emulator* de *hardware* em FPGA.

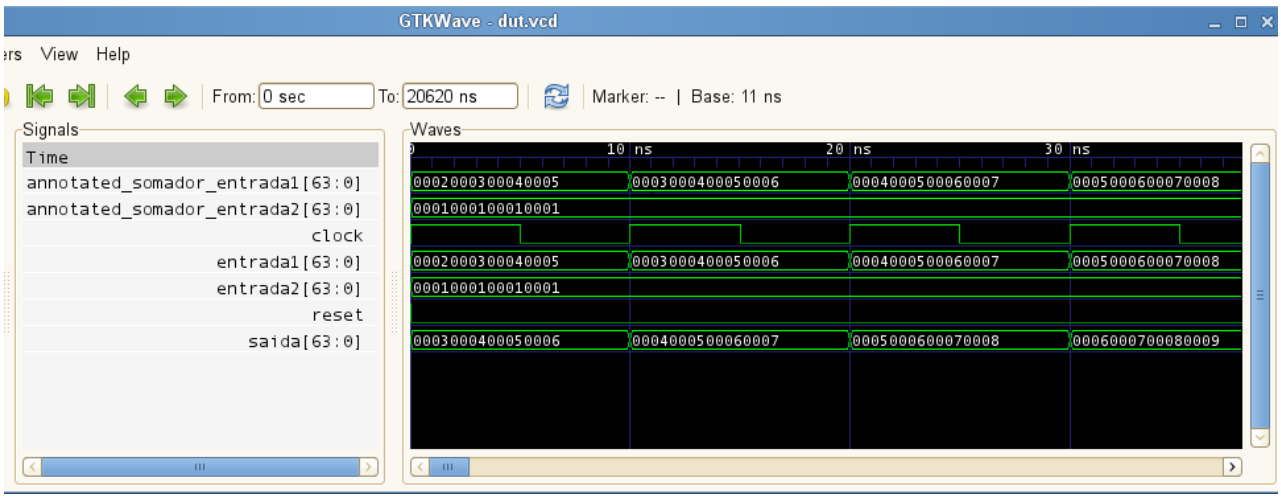


Figura 38 – Forma de Onda do *GTKWave*.

Por fim na Figura 38 é representado a formas de ondas dos dados que se encontra na Figura 37. O que pode ser observado é que as formas de ondas da Figura 38 equivalem-se as formas de ondas da Figura 33, ou seja, as entradas do somador são as mesmas e o resultado da soma é o mesmo, comprovando que o *hardware* foi executado e não sofreu nenhum dano no processo de verificação.

6 CONCLUSÃO E TRABALHOS FUTUROS

O presente Trabalho de Conclusão descreveu o desenvolvimento de uma interface de comunicação entre um ambiente de emulação de *hardware* e um módulo de comunicação de rede *Ethernet*, desta forma possibilitando melhor controle de projetos de *hardware.*

Para o desenvolvimento deste trabalho foi necessário o conhecimento de várias áreas da engenharia de computação, principalmente domínio sobre redes de comunicação, projeto de sistemas integrados e desenvolvimento de *softwares*, programação em linguagem de descrição de *hardware*.

Durante o desenvolvimento deste projeto algumas dificuldades foram enfrentadas, dando maior destaque para prototipação (por falta de domínio das ferramentas utilizadas), domínio sobre a hierarquia já desenvolvida, pois se trata de um projeto integrando outros dois projetos desenvolvidos em trabalhos anteriores.

Por fim a comunicação entre os dois módulos foi concluída com êxito, desta forma atingindo os objetivos propostos e obtendo bons resultados no sentido de ter maior e observabilidade de um projeto de *hardware* em FPGA.

Segue abaixo uma lista de possíveis trabalhos futuros.

- Integração com o *ModelSim*, para a exibição dos resultados da emulação, substituindo o *GTKWave*. Etapa que visa excluir o uso de mais um *software* (*GTKWave*), deixando então restrito ao uso apenas do *ModelSim*.
- Iniciar a emulação de circuitos com múltiplos domínios de relógio;
- Aperfeiçoamento do *software* de controle, deixando-o com uma interface mais amigável.
- Geração das formas de onda dinamicamente, durante a emulação. No presente trabalho, o emulador baseado em cadeias de *scan* exibe os resultados em formas de ondas somente após a emulação, então a ideia seria gerar estes resultados não após, e sim durante a emulação.
- Portar para outras plataformas de FPGAs da *Xilinx*. O *hardware* deve ser adaptado para outras famílias de FPGAs da *Xilinx*, tais como, *Virtex*-4 ou *Virtex*-6.
- Aumentar a velocidade de comunicação *Ethernet* para 1 *Gbit/seg*, já que a plataforma utilizada neste trabalho dispõe dessa velocidade. No presente trabalho foi explorado uma comunicação a 100 *Mbits/seg*.

REFERÊNCIAS BIBLIOGRÁFICAS

[CAD06] Cadence Design Systems Inc. "Incisive Enterprise Palladium". Capturado em: http://www.cadence.com/rl/Resources/datasheets/incisive_enterprise_palladium.pdf, Agosto 2011.

[CAL98] Calazans, N. L. V. "Projeto Lógico Automatizado de Sistemas Digitais Sequenciais". 11ª Escola de Computação, UFRJ, 1998, 318p.

[CAS10] Castilhos, G. M.; Giacomet, L. L. S. P; Grechi, T. "Emulação de Sistemas Digitais Síncronos em Dispositivos FPGAS", Monografia de Trabalho de Conclusão de Curso, PUCRS, 2010, 63p.

[BER03] Bergeron, J. "Writing Testbenches - Functional Verification of HDL Models". Kluwer Academic Publishers, 2003, 354p.

[DIN09] Dini Group. "User Guide DN9200K10PCIe-8T", Manual do Usuário, San Diego, Março 2009, 224p.

[KIL07] Kilts, S. "Advanced FPGA Design: Architecture, Implementation, and Optimization". John Wiley & Sons, 1st edition, 2007, 352p.

[MEN10] Mentor Graphics. "Emulation Systems". Capturado em: http://www.mentor.com/products/fv/emulation-systems/, 2011.

[IEE01] .IEEE. "IEEE Standard Verilog Hardware Description Language". IEEE Design & Test of Computers, 2001.

[SIL11] Da Silva, J. S.; "Infraestrutura Para Controle de Projetos em FPGAs Através do Protocolo Ethernet", Artigo de Trabalho de Conclusão de Curso, PUCRS, 2011, 11p.

[REI09] Reinbrecht, C. R. C.; Scartezzini, G.; Rosa, T. R.; "Desenvolvimento de um Ambiente de Execução de Aplicações Embarcadas para a Plataforma Multiprocessada HeMPs", Monografia de Trabalho de Conclusão de Curso, PUCRS, 2009, 109p.

[TAN10a] Tanenbaum, Andrew S. Redes de Computadores. Editora Campus, 3ª edição, 1999, 948p., ISBN 8535201572.

[TAN10b] Tanenbaum, Andrew S. Computer Networks. Prentice Hall, 3rd edition, 1996, 820 p., ISBN 0133499456.

[XIL10a] XILINX. "ChipScope Pro and the Serial I/O Toolkit". Capturado em: http://www.xilinx.com/tools/cspro.htm, 2011.

[XIL10b] XILINX. "ISim User Guide". Capturado em: http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_2/plugin_ism.pdf, Julho 2010.

[WIR11] WIRESHARK. "Wireshark User's Guide". Capturado em: http://www.wireshark.org/download/docs/user-guide-a4.pdf, Setembro 2011.

ANEXO A

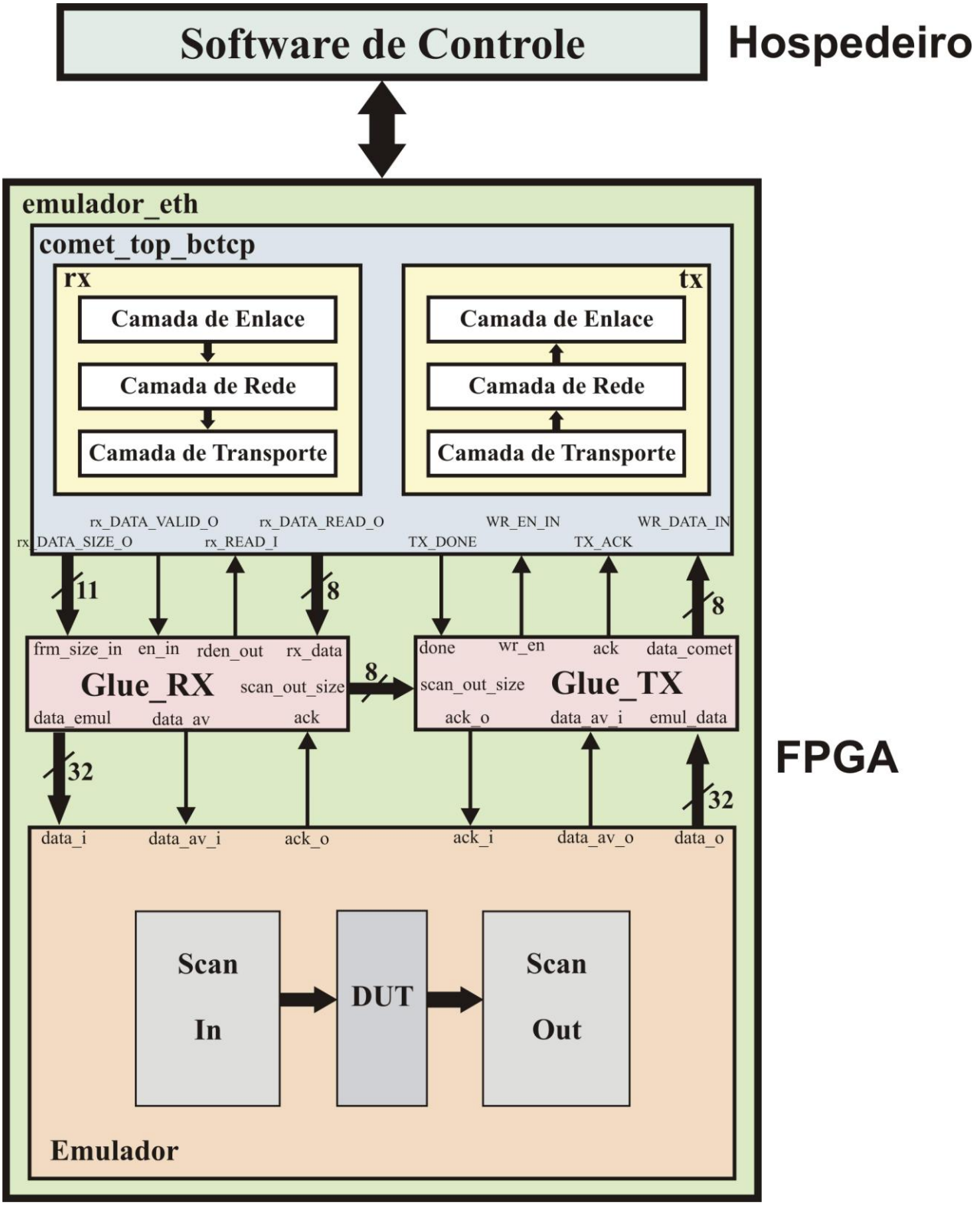
Arquivo de pacotes utilizado para envio de dados do *hospedeiro* para o FPGA.

```
00000000 000A350144222233223322330800450000320000000FF11cccc1A1B1C1DAABBCCDD00000000001eccc c00010002000000000000000000000000000000000000|
00000400
ffffffffffff11aa22bbcc0008004500014800000000FF11ccccA1B1C1D10000000000430044013 4cccc0201060000000000000008000000000000a20a2650000000000000000224401
350A0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000063825363350102000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000ff|
00001000
ffffffffffff11aa22bbcc0008004500014800000000FF11ccccA1B1C1D10000000000430044013 4cccc0201060000000000000008000000000000a20a2650000000000000000224401
350A0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000063825363350105000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000ff|
00001800 ffffffffffff001A4D98E71708004500003400000000ff110f2d0a20a26cffffffff232a232a0020720a00137200c1720a20a26c48454d504650474147415048 0000|
00002000 000A35014422001A4D98E71708004500003200000000ff1162a90a20a26c0a20a26523292329001e9f5f000100010000000000000000000000000000000c0ec|
00002800 000A35014422001A4D98E717080600010800060400020013 7200c1720a20a26c000a350144220a20a26c000000000000000000000000000000000000000000|
00003000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329000 20543d000400020806040100000000000000000000000000000000|
00003800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A2652329232900204f1a00040003002711020000000000000000000000000000000|
00004000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 03d1f000400040000000001100001100000001011111 00000000|
00004800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A265232923290020603500140001000000000000000000000000000000000000000|
00005000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 02e0e000400050000000001110000100100001010111100000000|
00005800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A265232923290020603400140002000000000000000000000000000000000000000|
00006000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 03d0c000400060000000011110000000110000010111110 0000000|
00006800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A265232923290020603300140003000000000000000000000000000000000000000|
00007000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 02cfd000400070001000011100000011000001010111100000000|
00007800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A265232923290020603200140004000000000000000000000000000000000000000|
00008000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 02c1b000400080011000011000000110000000101111100000000|
00008800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A265232923290020603100140005000000000000000000000000000000000000000|
00009000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 01e0a000400090111000010000000100100001010111100000000|
00009800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A265232923290020603000140006000000000000000000000000000000000000000|
0000a000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 03d080004000a111100000000000000011000001011111 00000000|
0000a800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 0602f00140007000000000000000000000000000000000000000|
0000b000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 02cf80004000b111100000000100000110000010101111 00000000|
0000b800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 0602e00140008000000000000000000000000000000000000000|
0000c000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 02c060004000c111100000001100001100000001011111 00000000|
0000c800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 0602d00140009000000000000000000000000000000000000000|
0000d000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 01cf50004000d111100000111000010010000101011110 0000000|
0000d800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 0602c0014000a000000000000000000000000000000000000000|
0000e000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 02bf30004000e111100001111000000011000001011111 00000000|
0000e800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 0602b0014000b000000000000000000000000000000000000000|
0000f000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 01be40004000f111000001110000000110000010121111 00000000|
0000f800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 0602a0014000c000000000000000000000000000000000000000|
00010000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 01b020004001011000000110000001100000001231111 00000000|
00010800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A265232923290020602900140000d000000000000000000000000000000000000000|
00011000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 00cf1000400111000000001000000100100001232111100000000|
00011800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A265232923290020602800140000e000000000000000000000000000000000000000|
00012000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 02bef00040012000000000000000000110000232311110 0000000|
00012800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A265232923290020602700140000f000000000000000000000000000000000000000|
00013000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 01bdf00040013000000000001000001100000323211110 0000000|
00013800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A265232923290020602600140010000000000000000000000000000000000000000|
00014000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 01aed00040014000000000011000011000000232311110 0000000|
00014800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A265232923290020602500140011000000000000000000000000000000000000000|
00015000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 00bdc00040015000000000111000010010000323211110 0000000|
00015800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A265232923290020602400140012000000000000000000000000000000000000000|
00016000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 01ada00040016000000001111000000110000232311110 0000000|
00016800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A265232923290020602300140013000000000000000000000000000000000000000|
00017000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 00acb00040017000100001110000000110000323211110 0000000|
00017800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A265232923290020602200140014000000000000000000000000000000000000000|
00018000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 009e900040018001100001100000011000000232311110 0000000|
00018800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A265232923290020602100140015000000000000000000000000000000000000000|
00019000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A2652329232900 20fbd700040019011100001000000010010000323211110 0000000|
00019800 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A265232923290020602000140016000000000000000000000000000000000000000|
0001a000 000A35014422001A4D98E71708004500003400000000FF1162a70A20A26C0A20A26523292329002 01ad60004001a111100000000000000011000023231111 00000000|
```

Este arquivo é formado por várias linhas, onde cada linha representa um pacote *Ethernet* é composto pelos seguintes protocolos: Ethernet para a camada Física, *IP* para a camada de Rede, *UDP* para a camada de transporte. Este é um exemplo de arquivo utilizado para simulação.

ANEXO B

Arquitetura do projeto com mais detalhes.



ANEXO C

Exemplo de *DUT* somador com sinais anotados, utilizado para simular o funcionamento do projeto.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity somador is
  port (
  annotated_somador_entrada2 : out std_logic_vector( 63 downto 0 );
  annotated_somador_entrada1 : out std_logic_vector( 63 downto 0 );
    clock       : in  std_logic;
    reset       : in  std_logic;
    entrada1    : in  std_logic_vector(63 downto 0);
    entrada2    : in  std_logic_vector(63 downto 0);
    saida       : out std_logic_vector(63 downto 0)
  );
end somador;
architecture rtl of somador is
  begin
   annotated_somador_entrada2 <= entrada2;
   annotated_somador_entrada1 <= entrada1;
     process(clock, reset)
     begin
       if reset='1' then
           saida <= (others=>'0');
       elsif clock='1' and clock' event then
           saida <= entrada1 + entrada2;
       end if;
     end process;
end rtl;
```