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

EMULAÇÃO DE SISTEMAS DIGITAIS SÍNCRONOS EM DISPOSITIVOS FPGAS

GUILHERME MACHADO DE CASTILHOS
LEONARDO LUIGI SIMÕES PIRES GIACOMET
THOMAS GRECHI

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
2010

Emulação de Sistemas Digitais Síncronos em Dispositivos FPGAs

RESUMO

Durante o projeto de sistemas digitais, simulações são de suma importância para validar a descrição do hardware. Simulações realizadas em computadores convencionais podem demorar horas ou dias para executarem a simulação de um conjunto representativo de vetores que teste que caracterizem o dispositivo que está sendo avaliado. Alternativa à simulação é a prototipação em FPGAs, porém esta técnica possui baixa controlabilidade e observabilidade. A técnica de emulação alia a controlabilidade e a observabilidade obtida com a simulação, e o baixo tempo de validação obtido com a prototipação. O presente Trabalho de Conclusão de Curso apresenta o desenvolvimento um ambiente de emulação que permite avaliar o comportamento de um dado sistema digital diretamente em um dispositivo FPGA, controlando-o através de um hospedeiro. O projeto sob teste é sintetizado com cadeias de *scan* ligadas aos sinais de entrada e sinais selecionados a serem observados. Para garantir a elevada taxa de dados demandada, a comunicação com o hospedeiro é realizada através da interface *PCIe.* O hospedeiro executa um software de controle, que possibilita a visualização dos dados de entrada e saída através de formas de onda, acelerando a verificação das funcionalidades do projeto.

SUMÁRIO

Lista de Tabelas **6**

Lista de Figuras **7**

Lista de Siglas **9**

1. Introdução **11**

1.1 Objetivos 13

1.2 Motivação 13

1.3 Resultados Esperados 15

1.4 Organização do Documento 15

2. Referencial Teórico **16**

2.1 O padrão PCI Express 16

2.1.1 Placa DN9200k10PCIe-8T *18*

2.2 Teste Baseado em Cadeias de Scan 19

2.3 Ferramentas para Manipulação de Arquivos VHDL 20

3. Arquitetura do Ambiente de Emulação **22**

3.1 Bibliotecas de Cadeias de Scan 25

3.2 Controle da Comunicação com PCIe e Inserção/Extração de Dados das Cadeias de Scan 29

3.2.1 Máquinas de Controle Para Inserção de Dados *29*

3.2.2 Inserção/extração de Dados nas Cadeias de Scan *33*

3.3 Protocolo e Interface de Comunicação 36

3.3.1 Interface de Comunicação *37*

3.4 Paralelismo Cadeias de Scan Entrada e Saída 43

4. Fluxo de Emulação **45**

4.1 Ferramenta de Anotação do Código VHDL 46

4.2 Ferramenta de Integração 47

5. Avaliação do Emulador **52**

5.1 Validação em Software 52

5.2 Validação em Hardware 56

6. Conclusões e Trabalhos Futuros **58**

Referências Bibliográficas **59**

Anexo A – Estrutura dos Códigos HDL **60**

Anexo B – VHDL convertido para XML **61**

Anexo C – VHDL anotado **62**

Anexo D – Arquivo VCD **63**

LISTA DE TABELAS

Tabela 1 – Comparação entre a ferramenta *ChipScope* e a proposta de emulação. 14

Tabela 2 – Máxima taxa de transferência de interfaces *PCIe* [IBM05]..... 17

Tabela 3 – Resultados de síntese da cadeia de *scan* de entrada, dispositivo xc5vlx330-2ff1760, ISE 11.4..... 29

Tabela 4 – Resultados de síntese da cadeia de scan de saída, dispositivo xc5vlx330-2ff1760, ISE 11.4..... 29

Tabela 5 – Resultados de síntese do controle, dispositivo xc5vlx330-2ff1760, ISE 11.4 29

Tabela 6 – Descrição dos campos do protocolo de comunicação. 36

Tabela 7 – Descritores utilizados pelo controlador *DMA* da Dini Group [DIN09b]. 40

Tabela 8 – Cenários de teste com respectivos sinais de entrada e saída (sinais observados). 53

Tabela 9 – Resultados de síntese da validação em software, dispositivo xc5vlx330-2ff1760, ISE 11.4..... 54

Tabela 10 – Resultados de síntese da validação em hardware, dispositivo xc5vlx330-2ff1760, ISE 11.4..... 57

LISTA DE FIGURAS

Figura 1 – Hierarquia das atividades de análise incluídas no projeto e na fabricação de sistemas digitais [CAL98]. 11

Figura 2 – Camadas da arquitetura de interfaces *PCIe* [NAI09]. 17

Figura 3 – Placa DN9200k10PCIe-8T utilizada no projeto [DIN09a]. (A) *FPGA* do tipo Xilinx Virtex-5 XCVLX330, denominado *FPGA A*; (B) *FPGA* do tipo Xilinx Virtex-5 XCVLX330, denominado *FPGA B*; (Q) *FPGA* do tipo Xilinx Virtex-5 LX50T, denominado *FPGA Q*; (S) *FPGA* do tipo *Spartan* 3. 18

Figura 4 – Método de teste baseado em *scan* – os caminhos ativos estão destacados em vermelho [BER03]. 19

Figura 5 – Arquitetura do ambiente de emulação. 23

Figura 6 – Ciclos de relógios de teste e relógio do *DUT*. 25

Figura 7 – Circuito que compõe o bloco unitário da cadeia de *scan* de entrada. 26

Figura 8 – Circuito da cadeia de *scan* de entrada com profundidade 2. 26

Figura 9 – Simulação da cadeia de *scan* de entrada de profundidade 5. 27

Figura 10 – Circuito que compõe o bloco unitário da cadeia de *scan* de saída. 27

Figura 11 – Circuito da cadeia de *scan* de saída com profundidade 2. 28

Figura 12 – Simulação da cadeia de *scan* de saída de profundidade 5. 28

Figura 13 – Máquina de estados de controle da cadeia de *scan* de entrada. 30

Figura 14 – Máquina de estados de controle da cadeia de *scan* de saída. 32

Figura 15 – Procedimento de inserção de dados na cadeia de *scan* de entrada. 34

Figura 16 – Sentido de extração dos dados da cadeia de *scan* de entrada. 34

Figura 17 – Procedimento de inserção de dados na cadeia de *scan* de saída. 35

Figura 18 – Sentido de extração dos dados da cadeia de *scan* de entrada. 36

Figura 19 – Campos do protocolo de comunicação. 36

Figura 20 – Utilização das memórias ao FPGA, utilizadas como *buffers*. 42

Figura 21 – Simulação do ambiente de emulação. Cadeias de *scan* de profundidade 16 (entrada) por 8 (saída). 43

Figura 22 – Fluxo do ambiente de emulação. 45

Figura 23 – Fluxo da ferramenta analisadora léxica/sintática. 47

Figura 24 – Interface gráfica da *Ferramenta de Integração*. 48

Figura 25 – Solicitação do arquivo topo da hierarquia. 48

Figura 26 – *Script list.do* executado pelo usuário na ferramenta ModelSim 49

Figura 27 – Exemplo de um arquivo *list. lst*. 49

Figura 28 – Exemplo de um arquivo binário de entrada. (a) campo indicando pacote de configuração; (b) profundidade das cadeias de scan de entrada; (c) profundidade das cadeias de scan de saída; (d) quantidade de ciclos de relógio de teste com o *DUT* ativo; (e) campo indicando pacote de dados; (f) quantidade de frames; (g) dados úteis; (h) campo indicando final teste. 50

Figura 29 – Exemplo de um arquivo *wave.do*. 50

Figura 30 – Comunicação entre *DUT* no ambiente de emulação e o controlador *PCIe*. 52

Figura 31 – Simulação do *DUT* (Somador) no ModelSim. 54

Figura 32 – Dados de saída visualizados no GTKWave. 55

Figura 33 – Simulação do *DUT* (Sqrt) no ModelSim. 55

Figura 34 – Dados de saída visualizados no GTKWave. 56

Figura 35 – Simulação do *DUT* (Framer) no ModelSim. 56

Figura 36 – Dados de saída visualizados no GTKWave. 56

Figura 37 – Dados de saída do DUT (Somador) obtidos através da *PCIe*. 57

Figura 38 – Dados de saída do DUT (Sqrt) obtidos através da *PCIe*. 57

Figura 39 – Organização dos módulos utilizados no presente projeto. 60

LISTA DE SIGLAS

AGP – *Accelerated Graphics Port*
BAR – *Base Address Register*
BRAM – *Block Random Access Memory*
ckDUT – *Clock* do *DUT*
ckT – *Clock* de Teste
CLB – *Configurable Logic Block*
CPU – *Central Processing Unit*
CRC – *Cyclic Redundancy Check*
DFT – *Design for Testability*
DMA – *Direct Memory Access*
DUT – *Design Under Test*
FPGA – *Field Programmable Gate Array*
HDL – *Hardware Description Language*
IP – *Intellectual Property*
JTAG – *Joint Test Action Group*
LUT – *Look Up Table*
LVDS – *Low-voltage differential signaling*
NRE – *Non-recurring engineering*
OTN – *Optical Transport Network*
PCI – *Peripheral Component Interconnect*
PCIe – *Peripheral Component Interconnect Express*
PERL – *Practical Extraction and Report Language*
RTL – *Register Transfer Level*
SAS – *Serial Attached SCSI*
SCSI – *Small Computer System Interface*
SO – Sistema Operacional
SoC – *System-on-a-Chip*
VCD – *Value Change Dump*
VHDL – *Very High Speed Integrated Circuits Hardware Description Language*
XML – *Extensible Markup Language*

1. INTRODUÇÃO

Desde a introdução das linguagens de descrição de hardware e das ferramentas de síntese lógica, em meados dos anos 80, a preocupação das equipes de engenheiros voltou-se para as etapas de teste e verificação dos projetos, com a finalidade de constatar se o projeto em desenvolvimento está de acordo ou não com as especificações estabelecidas. O maior tempo despendido nas etapas de verificação deve-se ao aumento da complexidade dos sistemas digitais. O tempo e custo com retrabalho sobre um produto que não passou pelas etapas de teste e verificação pode acarretar prejuízos e afetar a colocação do mesmo no competitivo mercado tecnológico.

Na atual era dos projetos de sistemas computacionais contendo milhões de portas lógicas, reuso de módulos de Propriedades Intelectual (do inglês, *IP*), e integração de todos os elementos de um sistema complexo em um único *chip* (do inglês, *SoC*), as etapas de teste e verificação chegam a consumir de 60 a 70% do tempo de projeto. Devido a esse fato, as equipes de projeto incluem engenheiros inteiramente dedicados às etapas de teste e verificação [BER03].

Em razão do alto esforço direcionado às etapas de teste e verificação, faz-se necessário o constante desenvolvimento de novas ferramentas que busquem reduzir o tempo gasto na verificação. Seguindo esta tendência, juntamente com a necessidade de redução do *time-to-market*, o mercado atual disponibiliza diversas ferramentas e técnicas para acelerar simulações ou para visualizar sinais internos no *FPGA*, sempre com o objetivo de simplificar o processo de teste e verificação. Porém, as ferramentas mais eficientes têm um alto custo de aquisição, ficando viável apenas para um reduzido grupo de empresas.

A Figura 1 apresenta uma hierarquia das atividades de análise utilizadas no projeto e na fabricação de sistemas digitais, que serão explicadas a seguir, no intuito de introduzir os seguintes conceitos: simulação, prototipação e emulação. Os demais, não fazem parte do escopo deste trabalho.

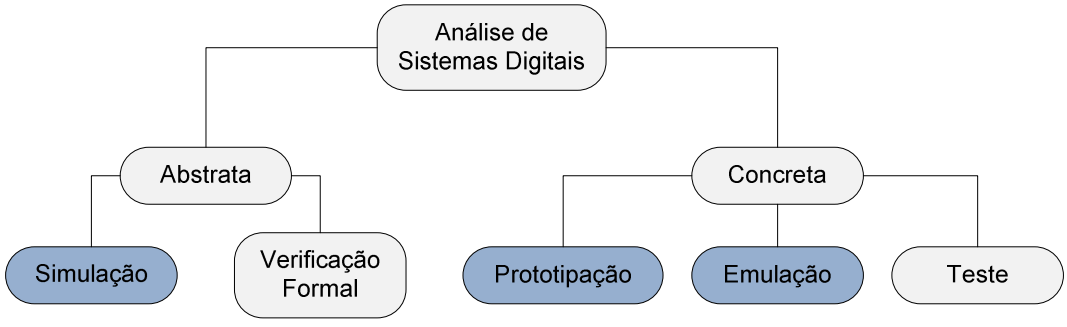


Figura 1 – Hierarquia das atividades de análise incluídas no projeto e na fabricação de sistemas digitais [CAL98].

A *simulação* de um sistema digital é um processo que consiste em exercitar uma descrição abstrata do sistema, aplicando um conjunto de padrões de entrada e computando as saídas associadas, ao longo de um intervalo de tempo pré-determinado. Assim, dois tipos de informação precisam ser elaborados para que se dê uma simulação: uma implementação do sistema e um padrão de entradas a ser aplicado à implementação [CAL98].

Um recurso utilizado na simulação é o *testbench*, que é um ambiente no qual o produto em desenvolvimento é testado através de um conjunto de entradas. Muitas vezes, esse ambiente é projetado especificamente para o produto em verificação.

O processo de validação da funcionalidade de um sistema digital muito complexo por simulação pode se tornar demorado a ponto de inviabilizar o uso deste método. Um método de prototipação em hardware pode acelerar a validação e ainda aumentar o grau de segurança sobre os resultados do processo de validação, por se tratar de um modelo concreto do sistema digital.

Por *prototipação* entende-se a construção de uma versão do sistema digital usando componentes que não serão necessariamente todos idênticos aos do produto final, mas que desempenham o mesmo papel de um ou de uma parte do componente que substituem. A validação de sistemas digitais por prototipação é feita pela construção do protótipo, seguida de aplicação de um conjunto de excitações. Um software é implementado em uma plataforma de desenvolvimento (hospedeiro), com capacidade de gerar um código específico para o sistema digital em implementação, que é descarregado para a memória do protótipo [CAL98].

A ferramenta *Chipscope* [XIL10a] desenvolvida pela empresa Xilinx, é um exemplo de ferramenta que dá suporte a métodos de prototipação. Apesar de prover ao projetista detalhes da operação do circuito no ambiente em que o mesmo será utilizado, esta ainda possui diversas limitações que serão detalhadas na Seção 1.2.

Métodos de *emulação* são métodos muito similares à prototipação, mas nestes existe uma conexão física entre o hospedeiro e o protótipo através de um subsistema denominado emulador. O software de emulação reside na plataforma de desenvolvimento (tipicamente um microcomputador, denominado hospedeiro), enquanto que o hardware de emulação conecta-se por um lado à plataforma e por outro ao protótipo. A principal diferença entre prototipação e emulação dá-se pela forma de estimular e controlar o protótipo. Na prototipação o circuito opera na freqüência nominal e os estímulos são fornecidos pelos circuitos conectados ao protótipo, não havendo um controle externo. Por outro lado, na emulação o protótipo é controlado pelo hospedeiro, o qual é responsável pelo envio de estímulos e pelo controle de operação do protótipo.

A vantagem de emuladores é que podem alcançar ganhos em termos de tempo de

desenvolvimento uma em uma ordem de grandeza com relação a abordagens de prototipação. Estes ganhos se devem ao aumento significativo na controlabilidade e observabilidade do componente substituído. Porém, a grande desvantagem de um emulador é que geralmente é uma ferramenta muito cara, como já citado anteriormente. Sua aquisição constitui em altos custos não-recorrentes de engenharia (do inglês, *NRE costs*) no desenvolvimento de um produto [CAL98].

1.1 Objetivos

O presente Trabalho de Conclusão de Curso (TCC) tem por objetivo geral apresentar o desenvolvimento de uma plataforma de hardware e software capaz de emular um sistema digital diretamente em *FPGA*, com a finalidade de verificá-lo de forma mais rápida.

Os objetivos específicos do TCC são listados abaixo. Relacionam-se para cada objetivo específico as disciplinas cursadas no curso de Engenharia de Computação, mostrando a estreita relação deste TCC com o mesmo.

- Domínio de técnicas de prototipação (Organização/Arquitetura de Computadores);
- Domínio de uma interface de alto desempenho entre uma estação de trabalho e uma placa de prototipação (Programação de Periféricos);
- Domínio da técnica de cadeias de scan para teste de circuitos digitais (Projetos de Sistemas Integrados I e II, Teste e Confiabilidade de Sistemas);
- Domínio de técnicas de manipulação de arquivos *VHDL* (Algoritmos e Estruturas de Dados).

1.2 Motivação

O tempo de colocação de um produto no mercado faz com que seja necessária 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 grandes prejuízos.

A necessidade de avaliar um sistema digital rapidamente, sem incorrer nos altos tempos requeridos pela simulação *VHDL/Verilog*, aliado aos altos custos das ferramentas já existentes que desempenham a função de verificação, motivou o desenvolvimento do presente projeto.

Uma ferramenta muito utilizada em prototipação, que tem por objetivo agilizar a verificação, é o *ChipScope* [XIL10a]. Esta ferramenta permite selecionar sinais de um código *HDL* a serem observados. No processo de síntese são acrescidos hardware de controle e blocos de memória no projeto para armazenar um determinado conjunto de amostras dos sinais selecionados. Após a

síntese o circuito pode ser carregado no *FPGA*, e o hospedeiro controla a leitura dos sinais a serem observados através de um cabo e interface JTAG. Nota-se que por se tratar de uma prototipação, o hospedeiro não controla o protótipo, apenas realiza a leitura dos dados observados.

A Tabela 1 compara a proposta de emulação deste trabalho com a ferramenta de depuração da Xilinx.

Tabela 1 – Comparação entre a ferramenta *ChipScope* e a proposta de emulação.

Pontos Comparados	Chipscope	Emulação
Inserção de estímulos de entrada	☹ não possui suporte a inserção de estímulos	☺ as entradas são providas pelo hospedeiro
Quantidade de recursos utilizados no *FPGA*	☹ alta, pois há o consumo de *BRAMs* para o armazenamento das saídas	☺ pequena, apenas as cadeias de *scan*, o controlador, e o *core PCIe*
Quantidade de sinais observados	☹ pequena, limitada pelo número de *BRAMs*	☺ alta, em função da profundidade das cadeias de *scan*
Processamento dos dados de saída e numero de ciclos amostrados	☹ *offline*, limitado pela interface *JTAG*. Uma condição de gatilho dispara o armazenamento das respostas em *BRAMs*, para posteriormente serem enviados pela interface *JTAG* ao hospedeiro	☺ *online*, com a execução do *DUT* durante um número arbitrário de ciclos, pode-se exibir as respostas à medida que o *DUT* processa os estímulos de entrada
Freqüência do *DUT*	☺ o *DUT* opera na freqüência de utilização	☹ o *DUT* opera em uma freqüência muito menor à freqüência normal de operação

Outra ferramenta disponível no mercado é o *ISim* [XIL10b], também desenvolvido pela empresa Xilinx. Ela é um simulador de *HDL*, que possui uma funcionalidade, denominada *Hardware Co-Simulation*, que permite fazer uma simulação executando porções em hardware e porções em software simultaneamente. Isso pode acelerar a simulação de um projeto complexo e verificar se o mesmo realmente funciona em hardware. A desvantagem dessa ferramenta, é que apenas um módulo do projeto pode ser simulado em hardware por vez. Portanto, quando um módulo é habilitado para exucutar em hardware, a instância habilitada anteriormente é desabilitada. Isto limita o desempenho da simulação, já que não é possível executar a simulação por completa em hardware.

Em relação à aceleração na simulação, citam-se os produtos *Veloce*, desenvolvido pela Mentor Graphics [MEN10], e o *Palladium*, criado pela Cadence [CAD06]. Ambas plataformas prometem um aumento significativo na velocidade da verificação em relação à simulação *RTL*. A desvantagem desses produtos é o elevado preço inicial, tornando sua aquisição viável apenas para empresas de grande porte.

Na área acadêmica citamos o *Hybrid Approach to Faster Functional Verification with Full Visibility* [CHU07], que é um exemplo de pesquisa relacionada à aceleração de verificação. Nesse trabalho foi criada uma plataforma capaz de emular o circuito e gravar o comportamento interno do mesmo, reproduzindo-o posteriormente. Com esta ferramenta, é possível ter total visibilidade de circuitos pequenos, em tempos comparáveis ao de simulações.

Considerando os pontos negativos da ferramenta *ChipScope* (Tabela 1), o elevado custo dos produtos *Veloce* e *Palladium*, e as restrições de tamanho dos circuitos do sistema apresentado em [CHU07], o presente trabalho busca desenvolver uma solução de baixo custo para verificação de sistemas digitais, baseado em emulação.

1.3 Resultados Esperados

Ao final do TCC é esperado o desenvolvimento de um ambiente completo de emulação, que permita validar circuitos digitais síncronos. Este ambiente será composto por um conjunto de ferramentas, responsáveis por: **(1)** modificar o *DUT* em função dos sinais a serem observados; **(2)** enviar estímulos ao *DUT*; **(3)** controlar a operação do *DUT*; **(4)** exibir os resultados de execução.

1.4 Organização do Documento

Este documento está organizado da seguinte forma. O Capítulo 2 apresenta o referencial teórico, descrevendo os principais conceitos utilizados, tais como, *PCI Express*, teste baseado em cadeias de *scan* e ferramentas para manipulação de arquivos *VHDL*. O Capítulo 3 descreve o desenvolvimento do ambiente de emulação propriamente dito, apresentando as bibliotecas de cadeias de *scan*, controle da comunicação com a *PCIe*, inserção/extração de dados nas cadeias e interface de comunicação *PCIe* ↔ Hospedeiro ↔ *FPGA*. O Capítulo 4 apresenta o fluxo completo de emulação e também as ferramentas analisadora léxica/sintática e de integração. O Capítulo 5 contém a avaliação do ambiente de emulação, apresentando a validação do mesmo em software e em hardware. Por fim, o Capítulo 6 descreve as conclusões e trabalhos futuros.

2. REFERENCIAL TEÓRICO

O presente TCC envolve três temas: **(1)** comunicação entre o hospedeiro e o *FPGA* através de uma interface *PCIe*; **(2)** teste baseado em cadeias de *scan*, para envio de estímulos e recebimento de respostas do processamento; **(3)** ferramentas de edição automatizada de código *VHDL*, para capturar sinais primários e/ou internos que deverão ser observados.

Este Capítulo está organizado como segue. A Seção 2.1 faz uma rápida introdução ao padrão *PCI Express*. A Seção 2.1.1 apresenta a placa utilizada no projeto. A Seção 2.2 introduz o teste baseado em cadeias de *scan*. Por fim, a Seção 2.3 discute ferramentas para manipulação de arquivos *VHDL*.

2.1 O padrão PCI Express

O avanço da tecnologia de circuitos integrados conduziu também à necessidade de um aprimoramento dos padrões e recursos para comunicação entre dispositivos de forma a obter velocidades compatíveis às freqüências de operação. Para tal, foram desenvolvidos diversos padrões de comunicação com diferentes tecnologias, técnicas e focos. Citam-se como exemplos de padrões *AGP*, *HyperTransport* e *RapidIO*.

Na longa lista de padrões de comunicação, surgiu na década de 90 o padrão *PCI*, criado pela Intel. Neste padrão todos os dispositivos compartilham a mesma comunicação em um barramento. Ao longo do tempo, houve uma evolução do padrão, sempre com o objetivo de melhorar o desempenho global dos computadores, pois o barramento *PCI* pode chegar rapidamente a uma saturação, uma vez que os dispositivos ligados a ele disputam a banda disponível, podendo o barramento tornar-se um gargalo.

A busca pela evolução do padrão *PCI* deu origem ao *PCI Express* (*PCIe*) em 2004. No padrão *PCIe* cada periférico possui um canal exclusivo para comunicação com o principal controlador do sistema digital. Não vamos discutir se o *PCIe* pode ou não ser chamado de fato de barramento no sentido estrito da palavra, já que o termo barramento surgiu com o intuito de descrever um canal de comunicação compartilhado por vários dispositivos ou periféricos.

O padrão *PCIe* disponibiliza para cada ponto do barramento até 16 vias ou *lanes* que provêem acesso direto à outro dispositivo. Cada via possui uma comunicação independente, atingindo até 400MB/s de dados úteis. É possível agrupar as vias para atingir velocidades até 16 vezes superiores

à de uma via utilizada isoladamente. Esta velocidade torna o *PCIe* ideal para streaming contínuo de quantidades elevadas de dados, requisito básico neste TCC.

A Tabela 2 ilustra as taxas que podem ser obtidas com o barramento *PCIe*.

Tabela 2 – Máxima taxa de transferência de interfaces *PCIe* [IBM05].

Largura da Via	Velocidade do Relógio	Vazão (*duplex*, *bits*)	Vazão (*duplex*, *bytes úteis*)	Aplicações alvo
x1	2.5 GHz	5 Gbps	400 MBps	*Slots*, *Gigabit Ethernet*
x2	2.5 GHz	10 Gbps	800 MBps	
x4	2.5 GHz	20 Gbps	1.6 GBps	*Slots*, 10 *Gigabit Ethernet*, *SCSI*, *SAS*
x8	2.5 GHz	40 Gbps	3.2 GBps	
x16	2.5 GHz	80 Gbps	6.4 GBps	Graphics adapters

A arquitetura do *PCIe* é definida em camadas, conforme a Figura 2.

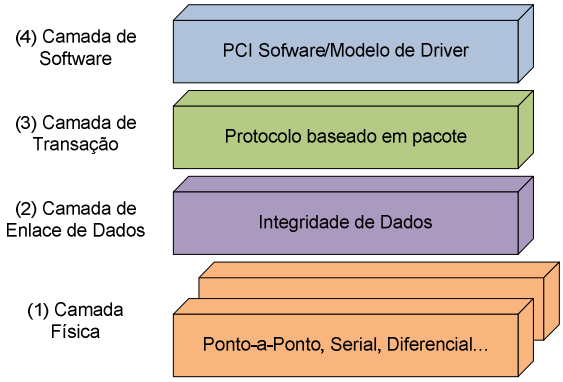


Figura 2 – Camadas da arquitetura de interfaces *PCIe* [NAI09].

(1) **Camada Física.** Um par de sinais diferenciais de baixa voltagem (do inglês, *LVDS*) transporta os dados de maneira serial. O sinal é codificado com um sinal de relógio próprio, utilizando o esquema de '8b/10b' que significa que cada 8 *bits* são representados em 10 *bits* (por esta razão as taxas em bits e bytes diferem em 20% conforme Tabela 2). A codificação é transparente às camadas superiores, como usual em protocolos de comunicação com camadas. A camada física está ligada às camadas de enlace de dados.

(2) **Camada de Enlace de Dados.** Responsável pela integridade dos dados, inserindo nos pacotes um número de seqüência e dados de *CRC*. É responsável também por enviar novamente qualquer pacote corrompido. O controle de fluxo utilizado é a técnica baseada em créditos, para garantir que só haja transmissão de pacotes quando há espaço disponível nos *buffers* de entrada e saída.

(3) **Camada de Transação.** Integra a camada do software com a camada de enlace, montando e desmontando os pacotes, que são utilizados para requisições de escrita e leitura. Esta camada

também realiza o controle de fluxo de dados utilizando créditos entre as Camadas de Transação nos dispositivos em comunicação. E, finalmente, recebendo os pacotes de resposta das requisições da camada de software, entregando-os a seus respectivos remetentes originais.

(4) **Camada de Software.** É onde o usuário interage com o Sistema Operacional através de *drivers* para se comunicar com a camada de transação.

O padrão *PCIe* também inclui em sua especificação registradores de configuração denominados de *BAR* (do inglês, *Base Address Register*) com a especificação permitindo até 6 *BAR* diferentes, nomeados *BAR0*, *BAR1*, *BAR2*, *BAR3*, *BAR4* e *BAR5*. Estes registradores indicam áreas da memória utilizadas para configuração dos dispositivos *PCIe*.

2.1.1 Placa DN9200k10PCIe-8T

A placa de prototipação utilizada neste projeto é a DN9200k10PCIe-8T, fabricada pela empresa Dini Group. A Figura 3 mostra uma foto desta placa.

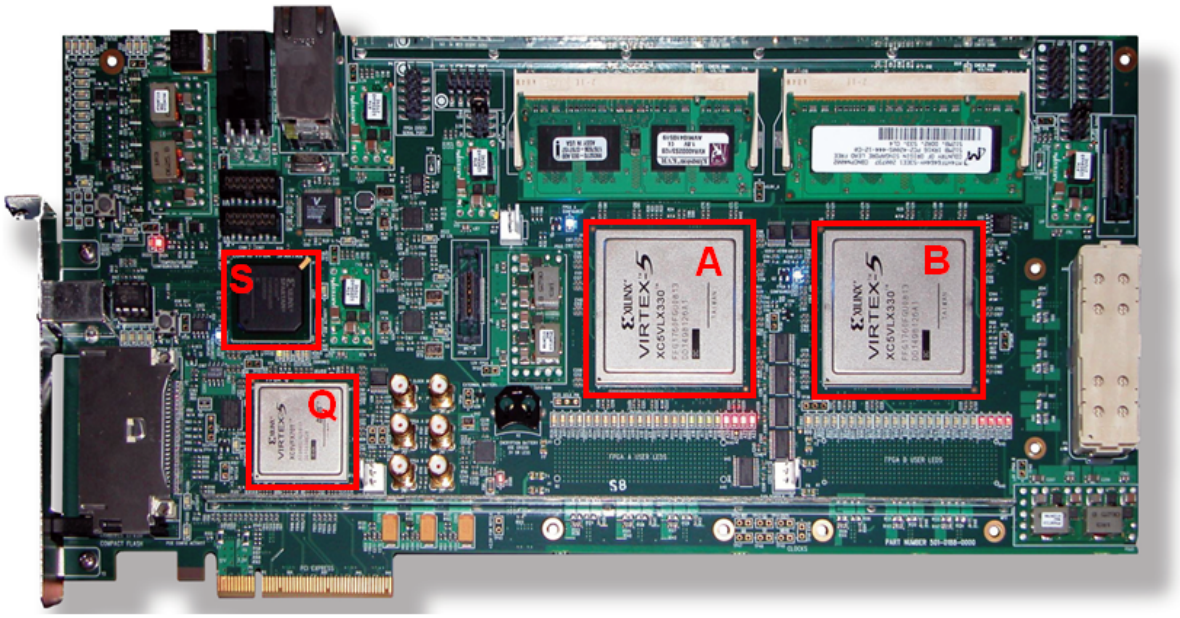


Figura 3 – Placa DN9200k10PCIe-8T utilizada no projeto [DIN09a]. (A) *FPGA* do tipo Xilinx Virtex-5 XCVLX330, denominado *FPGA A*; (B) *FPGA* do tipo Xilinx Virtex-5 XCVLX330, denominado *FPGA B*; (Q) *FPGA* do tipo Xilinx Virtex-5 LX50T, denominado *FPGA Q*; (S) *FPGA* do tipo *Spartan* 3.

A placa provê mecanismos de acesso direto a memória (do inglês, *DMA*). O *DMA* permite que os dados enviados/recebidos pela *PCIe* sejam gravados/lidos diretamente na/da memória da placa. Esta característica permite otimizar o desempenho dos circuitos desenvolvidos na placa de prototipação. A técnica de *DMA* não necessita de controle em tempo real. Por definição já se inclui buffers de entrada e saída inferidos (as próprias memórias acessadas pelo *DMA*). Este modo possui suporte do fabricante, e por tais razões será a técnica utilizada neste TCC.

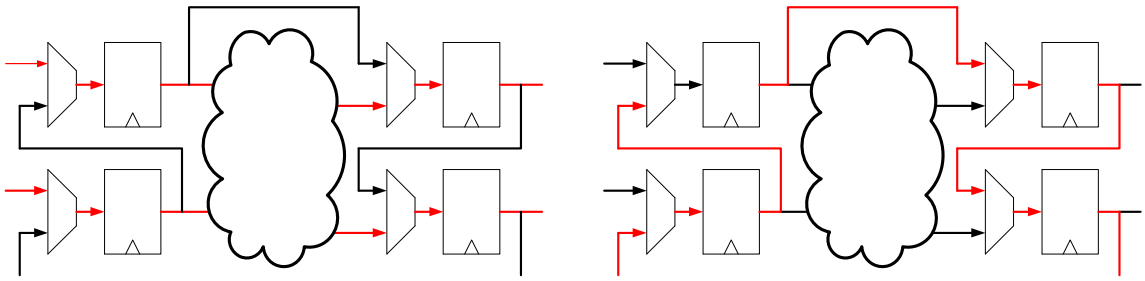
A placa inclui interface de comunicação *PCIe* 8 vias, podendo chegar a uma velocidade de 2,5 Gb/s por via com a Virtex-5 LXT ou 5,0 Gb/s por via com Virtex-5 FXT. Entre seus recursos, a placa possui quatro dispositivos *FPGAs*, onde três deles são configuráveis. Dos três configuráveis, apenas dois são dedicados ao projeto do usuário, ambos do tipo Xilinx Virtex-5 XCVLX330, denominados *FPGA A* (Figura 3(A)) e *FPGA B* (Figura 3(B)). O terceiro *FPGA* configurável é do tipo Xilinx Virtex-5 LX50T (denominado *FPGA Q* – Figura 3(Q)) dedicado exclusivamente à controladora do *PCIe*, que pode ser modificado caso haja necessidade. Por fim, o quarto *FPGA*, do tipo Spartan 3 (Figura 3(S)), é encarregado do controle da placa, inclusive pela configuração dos outros *FPGAs*. [DIN09a].

2.2 Teste Baseado em Cadeias de Scan

A etapa de teste de um projeto é muitas vezes confundida com a etapa de verificação. O objetivo da primeira é verificar se o projeto foi fabricado corretamente, enquanto da segunda é garantir que um projeto cumpre as funcionalidades desejadas [BER03].

Entre os métodos de teste existentes, o método baseado em cadeias de *scan* é a técnica de projeto para testabilidade (do inglês, *DFT*) mais comumente utilizada em projetos complexos. Neste método, todos os registradores dentro do projeto são interconectados em uma longa cadeia. A simplicidade do método permite desenvolver algoritmos que automaticamente insiram cadeias de *scan* no projeto, tornando-o aplicável a grandes circuitos, e apresenta ainda a vantagem de aumento da controlabilidade do projeto.

A Figura 4 apresenta a utilização do método de teste baseado em *scan*. No modo normal **(a)**, os registradores operam como se a cadeia de *scan* não estivesse presente no circuito, enquanto no modo de *scan* **(b)**, os registradores operam como um longo e único registrador de deslocamento, que contém todos os *flip-flops* do hardware.



(a) Modo Normal (b) Modo Scan

Figura 4 – Método de teste baseado em *scan* – os caminhos ativos estão destacados em vermelho [BER03].

Para testar um projeto utilizando o método baseado em *scan*, a unidade sob teste é colocada

em modo de teste, e um padrão de teste é carregado serialmente nos registradores internos. Em seguida, o circuito é colocado no modo normal e pulsos de relógio são aplicados, gravando-se o resultado da operação normal nos registradores de *scan* de saída. O circuito é então colocado novamente em modo de teste, deslocando-se o resultado para fora do circuito, comparado-se o valor obtido com o valor esperado. Paralelamente, o próximo vetor de dados é inserido na cadeia de *scan* de entrada [BER03].

Esse método aumenta a observabilidade e a controlabilidade do projeto, pois fornece um mecanismo para colocar o circuito em qualquer estado possível. Porém, determinadas restrições são colocadas no projeto para permitir a inserção de uma cadeia de *scan* e a geração automática de padrões de teste. Algumas destas restrições incluem, mas não se limitam a: projetos inteiramente síncronos, sem sinais de relógio derivados ou chaveados, e uso de apenas uma borda de relógio.

A inserção de cadeias de *scan* no circuito a ser testado implica o aumento de área e esforço de projeto. No entanto, estes dois fatores são superados no momento em que se tem uma ferramenta que facilite a inserção de uma ou mais cadeias de scan no projeto, juntamente com a geração automática de padrões de teste. Neste caso, a economia de tempo e maior confiança para colocar um produto no mercado superam o custo da utilização do método de cadeias de *scan*.

No presente projeto, a técnica de teste baseado em cadeias de *scan* será utilizada para verificação. O procedimento adotado será o mesmo do teste, utilizando-se um computador hospedeiro para envio de estímulos ao *DUT*, controle, e exibição dos sinais internos.

2.3 Ferramentas para Manipulação de Arquivos VHDL

Observar sinais internos de um projeto pode não ser uma tarefa simples devido às hierarquias presentes no mesmo. Criar uma ferramenta de manipulação de arquivos *VHDL* para este fim é uma tarefa complexa, justamente pela dificuldade do tratamento das hierarquias de um projeto.

A idéia da ferramenta consiste em modificar o código original levando os sinais até o topo da hierarquia do projeto. Desta forma, os sinais podem ser conectados às cadeias de *scan.*

Alternativas podem ser utilizadas para contornar a dificuldade de tratamento das hierarquias presentes no projeto. Dentre elas, destacamos um analisador léxico/sintático apresentado em [REH09] e um analisador sintático apresentado em [LON02].

A primeira traduz automaticamente descrições *VHDL* em código *XML*. A linguagem *XML* é uma linguagem estruturada hierarquicamente, o que facilita o caminhamento através da hierarquia de projetos.

A segunda é um módulo que define a sintaxe necessária para analisar um código *VHDL*. Como o [LON02] é somente um analisador sintático seria necessário criar *scripts* na linguagem *PERL* (sugerida pelo autor da ferramenta) para percorrer o código *VHDL* e realizar funções específicas a fim de se desenvolver um analisador léxico.

Desta forma, o presente projeto faz uso da ferramenta desenvolvida por [REH09] por ser tanto um analisador léxico como sintático. Fazendo uso desta ferramenta desenvolveu-se no contexto deste TCC uma ferramenta que disponibiliza os sinais a serem observados no topo da hierarquia do projeto. A Seção 4.1 do Capítulo 4, detalha o funcionamento e utilização desta ferramenta.

3. ARQUITETURA DO AMBIENTE DE EMULAÇÃO

O presente Capítulo descreve o ambiente de emulação. Uma introdução do funcionamento geral do ambiente é apresentada no início do Capítulo. Em seguida, a Seção 3.1 descreve as bibliotecas de cadeias de *scan* implementadas no presente projeto. A Seção 3.2 apresenta o controle da comunicação com a *PCIe* e a inserção/extração de dados nas cadeias de *scan*. A Seção 3.3 descreve a interface de comunicação *PCIe* ↔ Hospedeiro ↔ *FPGA*. Por fim, a Seção 3.4 apresenta uma simulação exemplo para demonstrar o funcionamento paralelo entre as cadeias de *scan* de entrada e saída.

A Figura 5 apresenta a arquitetura do ambiente de emulação. Nela observam-se as duas principais entidades do ambiente de emulação: o hospedeiro e o dispositivo reconfigurável responsável por receber o *DUT* (*FPGA*).

Na transmissão, o hospedeiro é responsável pela geração do conjunto de padrões de entrada para teste, que serão transmitidos por *drivers* da *PCIe* e entregues ao *DUT* emulado via cadeias de *scan* de entrada. Na recepção, o hospedeiro receberá os dados retirados das cadeias de *scan* de saída, também através da *PCIe.* O software hospedeiro também será responsável por gerar as formas de onda dos resultados da emulação, utilizando o software GTKWave.

Por sua vez, o *FPGA* é responsável por: (1) receber os dados vindos do Hospedeiro e colocá-los na cadeia de *scan* de entrada; (2) controlar a execução do *DUT*; (3) enviar os dados processados pelo *DUT* ao hospedeiro.

O *DUT* pode operar tanto em modo normal quanto em modo de teste. Em modo normal o sinal de controle *test_mode* é colocado em nível lógico zero, e o *DUT* opera com as entradas fornecidas pelo circuito conectado ao *DUT*, com o relógio nominal – *ckT*. Em modo teste o sinal *test_mode* é colocado em nível lógico um, e o *DUT* opera com as entradas fornecidas pelo hospedeiro, com o relógio fornecido pelo controlador – *ckDUT*. As cadeias de *scan*, Figura 5(a) e Figura 5(b), são detalhas na Seção 3.1.

A principal restrição adotada neste trabalho é que o *DUT* seja síncrono, com apenas um domínio de relógio.

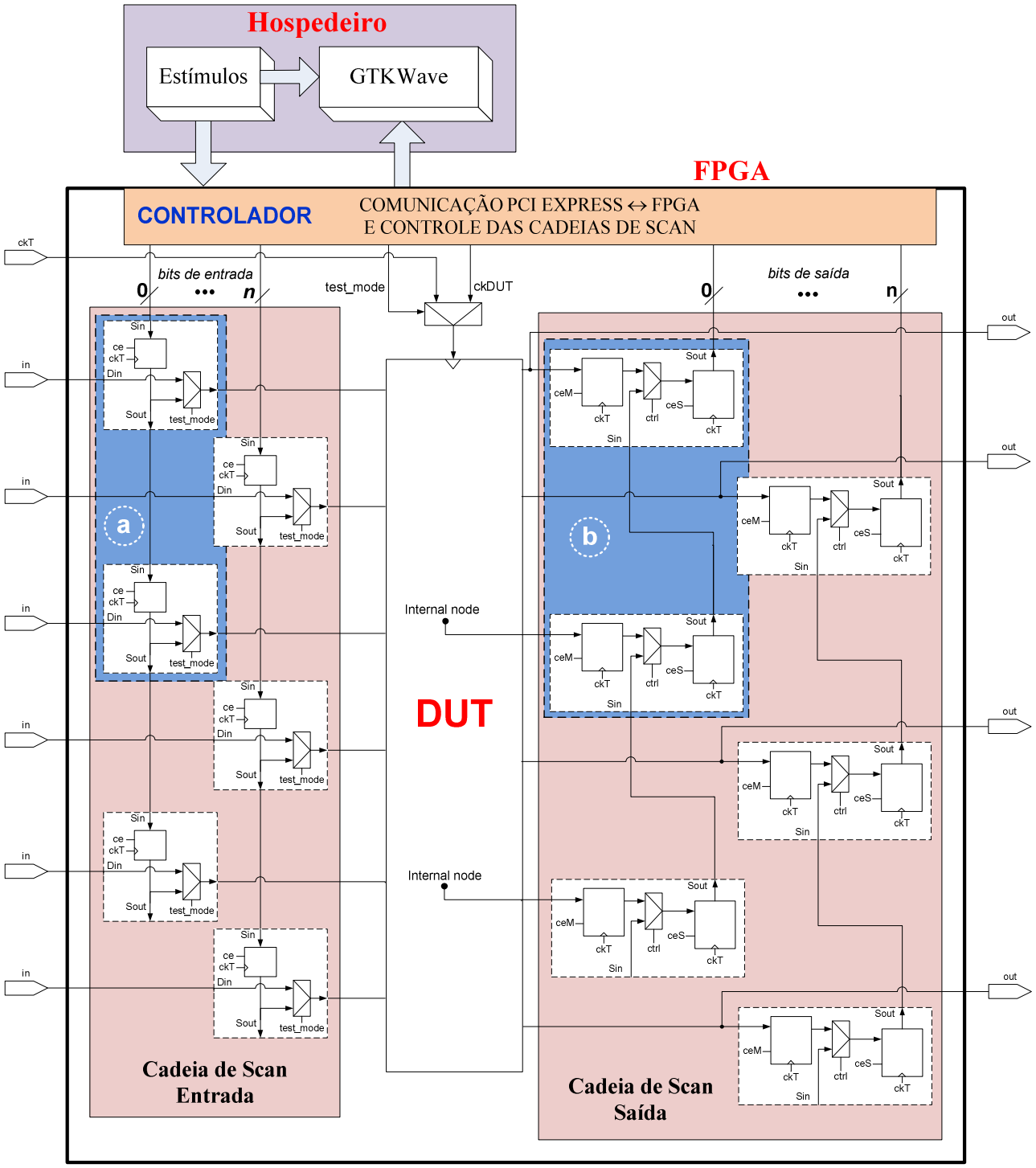


Figura 5 – Arquitetura do ambiente de emulação.

O fluxo de ações para realizar a emulação consiste em (sinal *test_mode* ativado):

1. Inicialmente o sinal de relógio do *DUT*, *ckDUT*, está em repouso (nível lógico zero);

2. O hospedeiro envia através da interface *PCIe* o conjunto de vetores de entrada através de cadeias de *scan*. Assume-se como condição de contorno do projeto o envio de até 512 bits de entrada, havendo no máximo 32 cadeias de *scan* em paralelo, cada uma com profundidade de até 16 bits. A Seção 3.3 deste Capítulo apresenta a justificativa para esta condição.

3. Uma vez as entradas armazenadas nas cadeias de *scan*, o *ckDUT* sobe, fazendo com que o *DUT* avalie as entradas. O *DUT* é controlado por apenas um sinal global de relógio, síncrono e é sensível à borda de subida deste relógio.

4. Aguarda-se um número parametrizável de ciclos do relógio de teste, de forma a permitir a propagação das entradas até saídas primárias.

5. Uma vez transcorrido este número de ciclos, o *ckDUT* desce, e os dados a serem amostrados (internos e saídas primárias) são armazenados nas cadeias de *scan* de saída. Aqui, obedece-se às mesmas condições de contorno: observar até 512 sinais, tanto externos quanto internos.

6. Em seguida, os dados armazenados nas cadeias de *scan* de saída são enviados ao hospedeiro. Esta etapa é realizada em paralelo com a recepção dos dados (etapa 2), para otimizar o desempenho da emulação. No caso da recepção dos dados ser mais rápida que o envio dos mesmos para o hospedeiro, existe um mecanismo de controle para evitar perda de dados. Este mecanismo está detalhado na Seção 3.2.1 deste Capítulo.

7. Por fim, o hospedeiro exibe em uma interface gráfica (GTKWave) tanto os estímulos de entrada quanto os estímulos de saída.

Notar que durante o teste a freqüência do *DUT* é inferior à freqüência nominal de operação do *FPGA*. A razão para isto é detalhada na Figura 6:

(1) são necessários 7 ciclos para estabelecer a comunicação entre o controlador *PCIe* e configurar a máquina de controle de inserção de dados;

(2) até 3 ciclos multiplicados pela profundidade da cadeia de scan de entrada para que as mesmas sejam carregadas;

(3) 1 ciclo - comando para ativar o *ckDUT*;

(4) 1 a 4 ciclos para propagar os dados de entrada no *DUT*;

(5) 1 ciclo - comando para ativar a máquina de controle da cadeia de *scan* de saída;

(6) o número total de ciclos para extração dos dados presentes na cadeia de scan de saída depende da seguinte equação: 3 + 2 * profundidade *scan* out;

(7) 3 ciclos para finalizar o modo teste.

A seqüência de eventos de emulação é semelhante a um *pipeline*. O segundo envio de dados compreende os eventos 2-7, superpostos a partir do evento 6. O tempo médio para o envio e recebimento dos dados compreende os eventos 2-5, totalizando 54 ciclos (48 ciclos para o evento 2, 1 para o evento 3, 4 para o evento 4, e 1 para o evento 5).

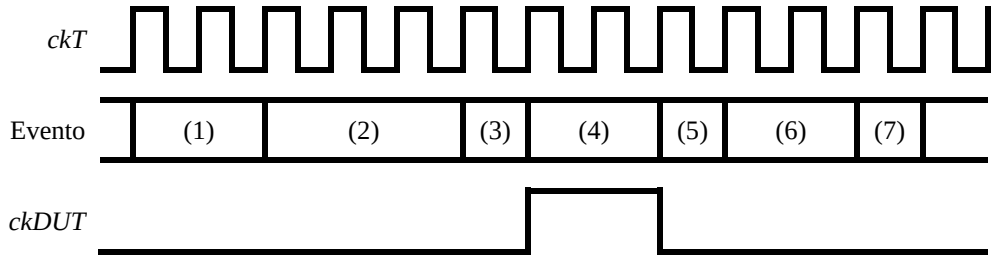


Figura 6 – Ciclos de relógios de teste e relógio do *DUT*.

A título de comparação, considere-se a simulação de um circuito de média complexidade no ModelSim, com f_{DUT}=200 MHz. Tipicamente, o tempo para simular 10ms (2 milhões de ciclos de f_{DUT}) de operação do *DUT* é de 100 segundos de simulação (em estação de trabalho). Para executar 2 milhões de ciclos com f=3,7 MHz (200 MHz / 54 ciclos) são necessários 0,54 segundos. Esta avaliação aponta para uma aceleração potencial da emulação em relação à simulação na ordem de 185 vezes.

Sendo necessário enviar 512 bits em 300 ns (16 ciclos de relógio para f = 200MHz), é preciso uma largura de banda de 300 Mbps com o hospedeiro. Esta largura de banda pode ser alcançada com o barramento *PCIe,* conforme Tabela 2.

3.1 Bibliotecas de Cadeias de Scan

Para o escopo do presente projeto, foi necessário fazer uma implementação das cadeias de *scan* diferente da habitual encontrada na literatura. Enquanto o método convencional interconecta todos os registradores do projeto, a cadeia desenvolvida para o presente ambiente de emulação apenas baseia-se neste método, inserindo somente cadeias de entrada e de saída.

A biblioteca implementada inclui as 16 possíveis diferentes profundidades de cadeias de *scan*, tanto de entrada quanto de saída. As cadeias de *scan* de entrada são formadas por um *flip-flop* tipo D e um multiplexador 2x1, replicados tantas vezes quanto for a profundidade da cadeia.

A Figura 7 apresenta o bloco composto pelos elementos citados no parágrafo anterior. O multiplexador tem a função de selecionar se o circuito irá ser alimentado com dados oriundos do hospedeiro, ou de uma entrada externa. Ele é controlado pelo sinal *test_mode* que quando em nível lógico baixo utiliza como entrada os dados procedentes do circuito externo (sinal *Din*). Em nível lógico alto o multiplexador permite a passagem do dado oriundo do registrador da cadeia de *scan*.

O *flip-flop* tipo D tem a função de registrar o dado de entrada, proveniente do hospedeiro (sinal *Sin*), e deslocá-lo para o próximo registrador da cadeia através do sinal *Sout*, a fim de preencher a profundidade total da cadeia de *scan*.

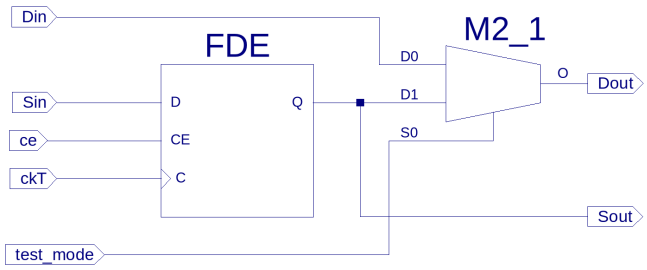


Figura 7 – Circuito que compõe o bloco unitário da cadeia de *scan* de entrada.

A Figura 8 apresenta a forma com que um bloco é conectado a outro formando uma cadeia de *scan* de profundidade 2. Os sinais *ce* (do inglês, *chip enable*), *ckT* e *test_mode* são os mesmos para os dois blocos uma vez que eles estão replicados. Observa-se também que o sinal de saída *Sout* do bloco (1) alimenta a entrada *Sin* do bloco (2), formando assim, um registrador único no momento em que o sinal *test_mode* encontra-se no nível lógico alto.

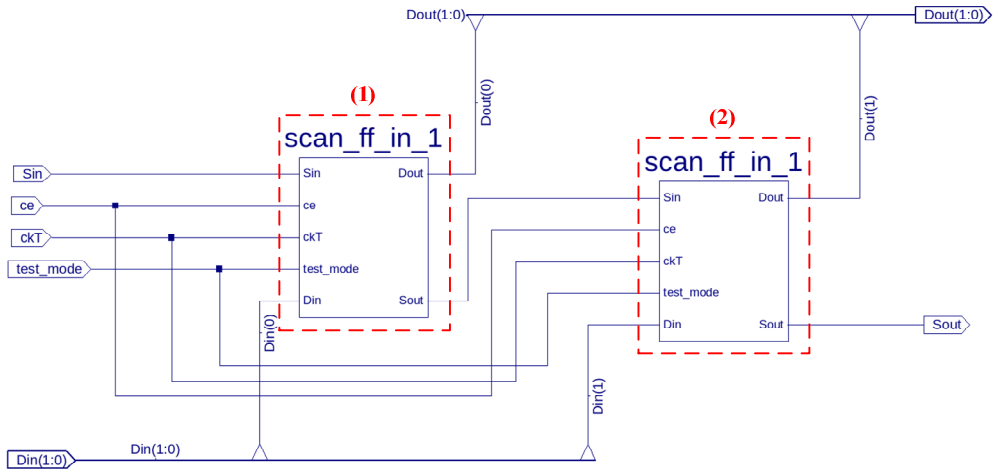


Figura 8 – Circuito da cadeia de *scan* de entrada com profundidade 2.

A Figura 8 corresponde à cadeia de *scan* detalhada na Figura 5(a). Uma cadeia de *scan* com profundidade 3 fará uso de um bloco como o circuito apresentado na Figura 8 interligado ao bloco que compõe o circuito da Figura 7, e assim por diante.

Para exemplificar o funcionamento das cadeias de *scan* de entrada, foi simulado uma cadeia de profundidade 5. Esta simulação é apresentada na Figura 9. Inicialmente o sinal *test_mode* encontra-se em nível lógico alto. No momento em que o sinal *ce* transiciona para nível lógico alto, os dados provenientes do hospedeiro começam a ser inseridos na cadeia através do sinal *Sin*. A entrada, que é inserida bit a bit, tem como primeiro valor ‘1’ e os subseqüentes ‘0’. A inserção desta entrada pode ser observada no sinal de saída *Dout* que a cada ciclo do relógio, vai sendo deslocada. Após todos os dados serem inseridos na cadeia de *scan*, o sinal *test_mode* foi transicionado para

nível lógico baixo, o que fez com que a saída *Dout* fosse sobrescrita com dados provenientes do sinal *Din*, ou seja, não provenientes do hospedeiro.

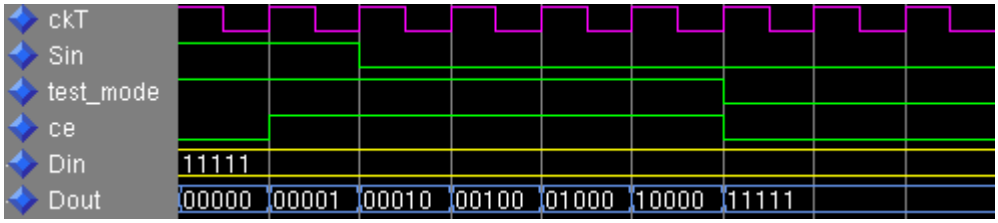


Figura 9 – Simulação da cadeia de *scan* de entrada de profundidade 5.

A arquitetura da cadeia de *scan* de saída é apresentada na Figura 10. Similar a cadeia de *scan* de entrada, é formada por um bloco único replicado tantas vezes quanto for sua profundidade, porém seu circuito é composto por dois *flip-flops* tipo D e apenas um multiplexador 2x1. O multiplexador interliga os *flip-flops*.

O primeiro *flip-flop* do circuito é denominado ‘Mestre’. Ele tem a função de armazenar o dado atual até que todo o dado anterior presente no segundo *flip-flop* (chamado de ‘Escravo’) tenha sido consumido pelo hospedeiro. O registrador escravo desloca o dado através dos demais registradores escravos da cadeia, para que possam ser consumidos serialmente pelo hospedeiro. Esse mecanismo permite o paralelismo entre recepção e processamento de novos dados, com o envio dos dados processados no ciclo anterior.

A comunicação entre os dois registradores presentes no bloco que compõe a cadeia de *scan* de saída é realizada pelo multiplexador e controlada através do sinal *ctrl*. Quando este sinal está em nível lógico baixo, o registrador escravo é alimentado pelo dado proveniente do registrador mestre. Caso contrário, em nível lógico alto, o registrador escravo recebe dados oriundos de outro registrador escravo através do sinal *Sin*.

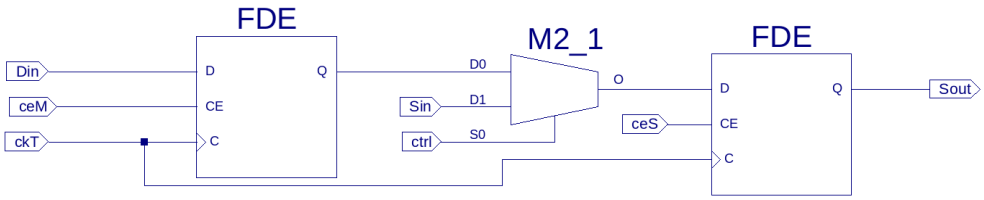


Figura 10 – Circuito que compõe o bloco unitário da cadeia de *scan* de saída.

A Figura 11 apresenta um circuito da cadeia de *scan* de saída com profundidade 2, com o objetivo de descrever a interconexão entre dois blocos. 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*.

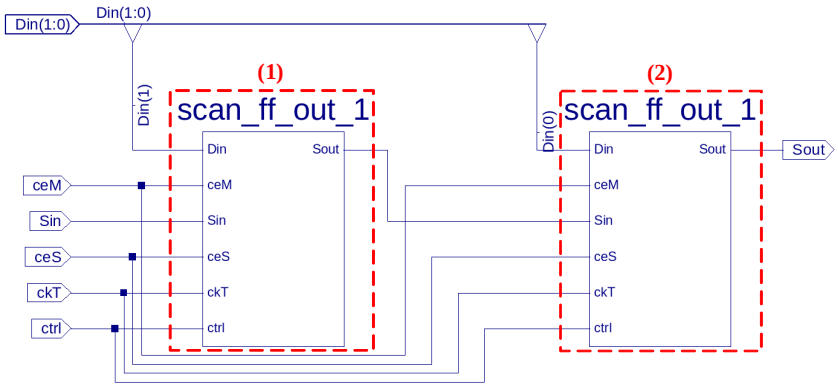


Figura 11 – Circuito da cadeia de *scan* de saída com profundidade 2.

A Figura 11 corresponde à cadeia de *scan* detalhada na Figura 5(b). Portanto, uma cadeia de scan com profundidade 3 fará uso de um bloco como o circuito apresentado na Figura 11 interligado ao bloco que compõe o circuito da Figura 10, e assim por diante.

Para exemplificar o funcionamento da cadeia de *scan* de saída, foi simulado uma cadeia de profundidade 5. Esta simulação é apresentada na Figura 12. Inicialmente o sinal *ceM* encontra-se em nível lógico baixo. No momento em que este sinal transiciona para nível lógico alto ocorre a carga do primeiro dado de entrada (sinal *Din* contendo o valor “10101”) nos registradores mestre da cadeia. Em seguida, o sinal *ceM* transiciona para nível lógico baixo para que os valores registrados não sejam sobrescritos antes de serem enviados para o registrador escravo. Como o sinal *ctrl* está em nível lógico baixo, os valores de entrada presentes nos registradores mestre são deslocados até a entrada do registrador escravo. Então, o sinal *ceS* é transicionado para nível lógico alto, carregando o valor nos registradores escravo. Como já mencionado anteriormente, o sinal *ctrl* é responsável por selecionar a entrada do registrador escravo. No momento em que o sinal transiciona para o nível lógico alto, observa-se que o sinal de saída *Sout* amostra o valor que será enviado bit a bit ao hospedeiro. Estes valores são deslocados de registrador para registrador escravo.

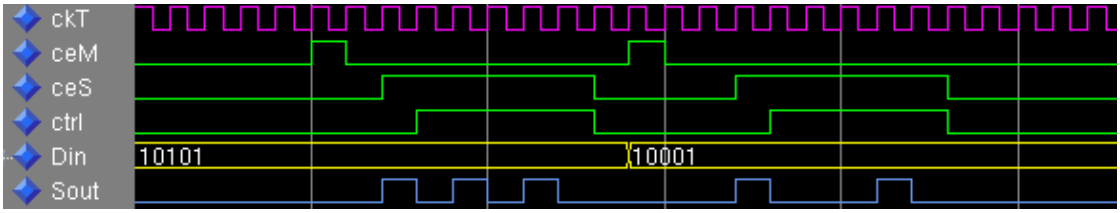


Figura 12 – Simulação da cadeia de *scan* de saída de profundidade 5.

Para fins de análise, foram sintetizadas as 32 cadeias de *scan* (16 de entrada e 16 de saída)

separadamente. Os resultados de síntese são apresentados na Tabela 3 para as cadeias de *scan* de entrada e Tabela 4 para as cadeias de *scan* de saída. Notar que a área total ocupada pelas cadeias de *scan* deve ser multiplicada por 32 (largura da palavra). Mesmo assim, a área ocupada é muito pequena. Para a maior profundidade das cadeias de *scan* teríamos o seguinte consumo de *LUTs*: (16 + 32) * 32 = 1536, correspondendo a 0,74% da área do *FPGA*.

Tabela 3 – Resultados de síntese da cadeia de *scan* de entrada, dispositivo xc5vlx330-2ff1760, ISE 11.4

SCAN IN	Profundidade das *Scans*																Total Disp.
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
Número Registradores	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	207360
Número *LUTs*	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	207360
Número *BUFG/BUFGCTRLs*	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	32

Tabela 4 – Resultados de síntese da cadeia de scan de saída, dispositivo xc5vlx330-2ff1760, ISE 11.4

SCAN OUT	Profundidade das *Scans*																Total Disp.
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
Número Registradores	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	207360
Número *LUTs*	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	207360
Número *BUFG/BUFGCTRLs*	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	32

3.2 Controle da Comunicação com PCIe e Inserção/Extração de Dados das Cadeias de Scan

O controle da comunicação com a interface *PCIe* e inserção/extração de dados das cadeias de *scan* é realizado por duas máquinas de estados. Estas máquinas trabalham paralelamente e serão detalhadas na Seção 3.2.1.

Para fins de análise de custo, o controle da comunicação com a *PCIe* foi sintetizado separadamente das cadeias de *scan* e do *DUT*. Os resultados de síntese são apresentados na Tabela 5. A área total consumida pelas *LUTs* corresponde somente a 0,12% da área do *FPGA*.

Tabela 5 – Resultados de síntese do controle, dispositivo xc5vlx330-2ff1760, ISE 11.4

Controle	Usado	Total disponível
Número Registradores	119	207360
Número *LUTs*	257	207360
Número *BUFG/BUFGCTRLs*	1	32

3.2.1 Máquinas de Controle Para Inserção de Dados

A Figura 13 apresenta a máquina de estados de controle de inserção de dados. A máquina de controle de inserção de dados inicia no estado *INIT*, que garante que o sinal do relógio do *DUT* e demais sinas estejam em repouso (nível lógico baixo). Seu próximo estado é denominado *IDLE*.

Nele é aguardada a disponibilização de dado válido através do sinal *data_av*. Esta informação é fornecida pela *PCIe* de modo que os dados possam ser consumidos nos próximos estados.

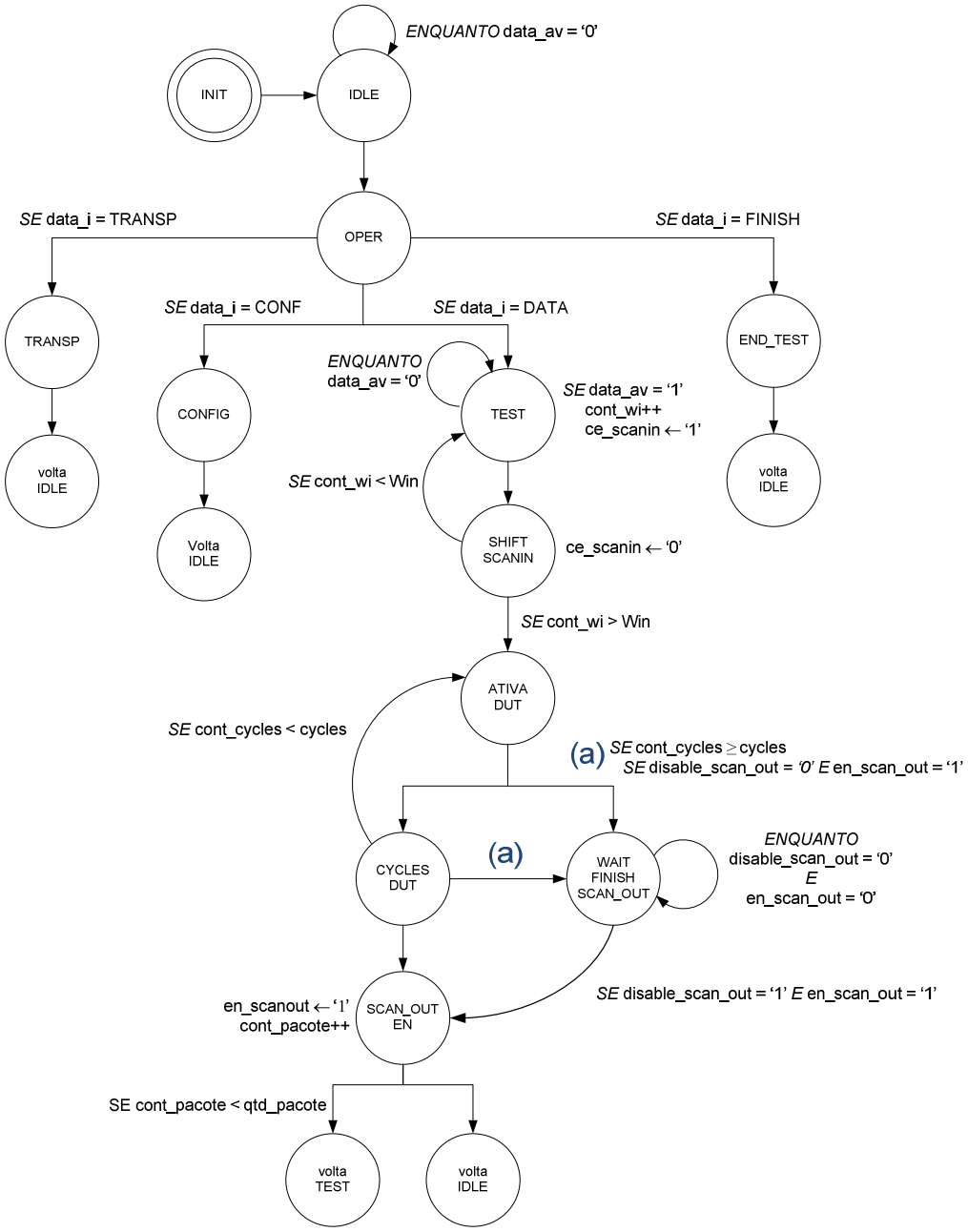


Figura 13 – Máquina de estados de controle da cadeia de *scan* de entrada.

Com o dado disponível, o estado *OPER* fará a avaliação da informação contida no pacote recebido, para então seguir para os próximos estados possíveis. São eles: *TRANSP* – os dados são oriundos do circuito externo, e as cadeias de *scan* são ignoradas; *CONFIG* – atribuição de informações de controle, tais como profundidade das *scans* e ciclos de execução do *DUT*; *TEST* – inserção dos dados nas cadeias de *scan* de entrada; *END_TEST* – finalização do modo de teste. É importante ressaltar que um *ack* é enviado ao o hospedeiro sempre que um dado é consumido.

No caso do próximo estado ser *TRANSP*, o sistema emulado irá receber dados do circuito externo, ignorando assim a existência das cadeias de *scan* de entrada. Se o próximo estado for *CONFIG*, as informações de profundidade das cadeias de entrada e saída serão registradas, bem como a quantidade de ciclos que o *DUT* ficará ativo para avaliar os dados provenientes da cadeia. É neste estado também, que o sistema emulado é configurado para operar em modo de teste, ou seja, fará uso das cadeias de *scan*.

Sendo o estado *TEST* o subseqüente de *OPER*, os dados úteis serão inseridos nas cadeias de *scan* de entrada. Para que isto aconteça, o pacote de dados lido no estado *OPER* deve conter a informação que os próximos '*n*' pacotes serão de dados úteis. A partir deste ponto, os dados provenientes do hospedeiro serão deslocados (através do estado *SHIFT SCAN IN*) a fim de preencher a profundidade total da cadeia.

Uma vez preenchida as cadeias de *scan* de entrada os próximos quatro estados possíveis são: *ATIVA DUT*, *CYCLES DUT*, *SCAN OUT EN* e *WAIT FINISH SCAN OUT*. O primeiro é o estado que ativa o sinal de relógio do *DUT*, iniciando a contagem dos ciclos de processamento dos dados procedentes das cadeias de *scan* de entrada. Enquanto o número de ciclos de relógio do *DUT* não atingir o valor pré-determinado pela configuração (feita no estado *CONFIG*), os estados irão se alternar entre *CYCLES DUT* e *ATIVA DUT*, incrementando a contagem de ciclos. Atingido o valor total de ciclos do *DUT*, o sinal de relógio do *DUT* é colocado em repouso no estado *CYCLES DUT*.

Ambos os estados *ATIVA DUT* e *CYCLES DUT* possuem condições de controle – detalhe (a) da Figura 13 – que garantem que os dados presentes nas cadeias de *scan* de saída não sejam sobrescritos, pois a mesma pode ser mais lenta do que as cadeias de *scan* de entrada. Este controle visa verificar, nos dois estados, se o sinal *disable_scan_out* esta em nível lógico alto. Caso contrário, a máquina de controle permanece no estado *WAIT FINISH SCAN OUT*.

Com o sinal do relógio do *DUT* em repouso, a máquina de controle das cadeias de *scan* de saída é ativada (estado *SCAN OUT EN*), dando início a extração dos dados.

Como a máquina de controle de extração de dados trabalha paralelamente a de inserção, o próximo estado possível ao de *SCAN OUT EN* pode ser o estado *TEST* novamente, pois podem existir mais pacotes de dados úteis a serem inseridos na cadeia de entrada.

Por fim, o último estado possível seguinte ao estado *OPER*' é o *END_TEST*. É neste estado que o sistema emulado é configurado para finalizar o modo teste, voltando a aguardar novas informações em *IDLE*.

A Figura 14 apresenta a máquina de estados de controle das cadeias de *scan* de saída. Assim como a máquina de inserção de dados, a de extração também inicia no estado *INIT*, que garante o nível lógico baixo dos sinais por ela controlados. Seu próximo estado é denominado *IDLE*. Nele a máquina encontra-se em repouso, aguardando a transição do sinal *en_scanout* (oriundo da máquina de inserção) para o nível lógico alto, o que garante que a extração de dados só começará quando houver dados validos na saída do *DUT*.

Com o dado disponível para ser extraído, caberá ao estado *MASTER* habilitar o registrador mestre da *scan* (sinal *ceM* em nível lógico alto), no intuito de armazená-lo. O estado seguinte, *SLAVE*, desabilita o registrador mestre e ativa o escravo (sinal *ceS* em nível lógico alto), deslocando os dados de um para o outro.

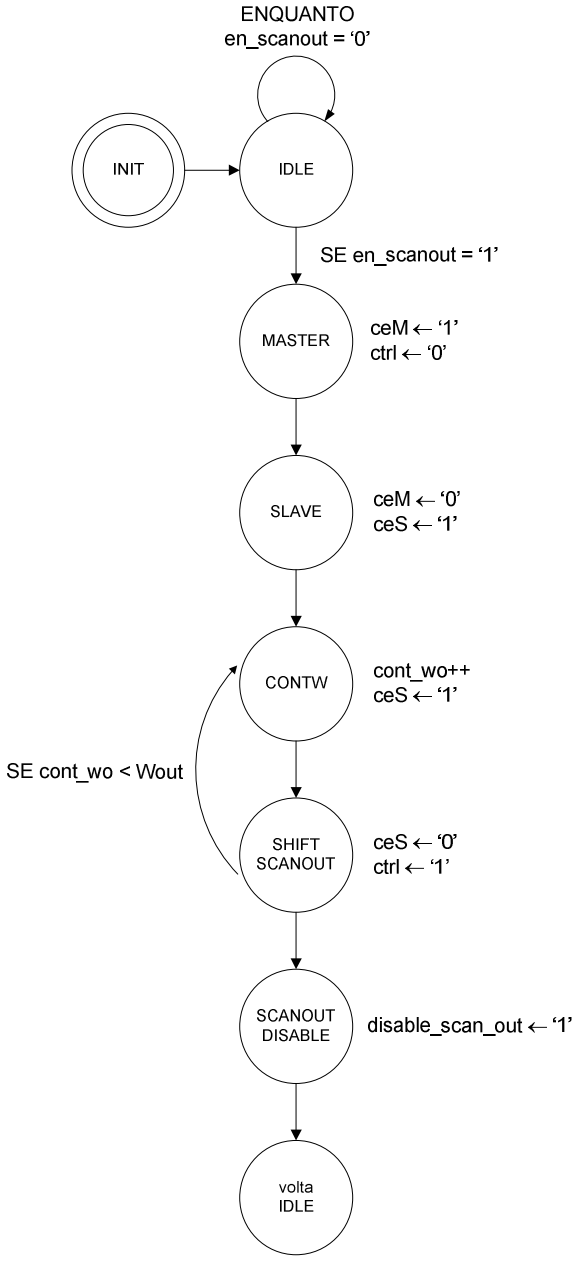


Figura 14 – Máquina de estados de controle da cadeia de *scan* de saída.

Com o dado armazenado no registrador escravo, a máquina entra no estado *CONTW*, estado no qual será alternado com o *SHIFT SCAN OUT*, deslocando os dados entre os registradores escravos até atingir a profundidade total da cadeia de saída. O deslocamento do dado de um registrador escravo para outro de diferente profundidade é permitido apenas quando o sinal *ctrl* transiciona para o nível lógico alto, e isto é feito no estado *SHIFT SCAN OUT*.

A máquina de controle de extração de dados termina com o estado *SCANOUT_DISABLE*, no qual o sinal *disable_scan_out* transiciona para nível lógico alto indicando que todos os dados da cadeia de *scan* de saída foram enviados ao hospedeiro. Portanto, a máquina retorna ao estado *IDLE* e aguarda novamente uma transição do sinal *en_scanout* que irá habilitá-la.

3.2.2 Inserção/extração de Dados nas Cadeias de Scan

Esta Seção explica através de um exemplo o método de inserção/extração de dados das cadeias de *scan* implementadas no ambiente de emulação. Porém, antes do detalhamento do método é importante ressaltar que os dados são tratados em pacotes de 32 bits, o que é justificado na Seção 3.3 deste Capítulo. Portanto, tanto as cadeias de *scan* de entrada, como as de saída possuem uma largura de 32 para comportar os 32 bits do pacote de dados.

A Figura 15 exemplifica através da utilização de três sinais, a inserção de dados nas cadeias de *scan* de entrada. Os sinais utilizados no exemplo possuem diferentes tamanhos (sinal a – 32 bits; sinal b – 16 bits; sinal c – 22 bits), totalizando 70 bits. Desta forma, observa-se que se faz necessário uma cadeia com profundidade 3 (96 bits) para que seja possível comportar os três sinais. A profundidade das cadeias de scan é calculada conforme Equação (1).

$$profundidade = \left\lfloor \frac{total\ bits}{32} \right\rfloor + 1 \quad (1)$$

A profundidade da cadeia é preenchida de forma serial e no sentido apresentado pela Figura 15. A parte superior da figura apresenta a concatenação dos bits de entrada em um vetor denominado *linha*, e os respectivos índices (de 0 a 69). A parte inferior da figura apresenta as 32 cadeias de *scan* de profundidade 3. A primeira cadeia de *scan* recebe os bits 0-2 do vetor *linha*, a segunda cadeia de *scan* recebe os bits 3-5 do vetor *linha*, e assim sucessivamente até a cadeia de *scan* 23. Os bits não utilizados são preenchidos com zeros, correspondendo a parte da cadeia de *scan* 23 até a última cadeia de *scan*.

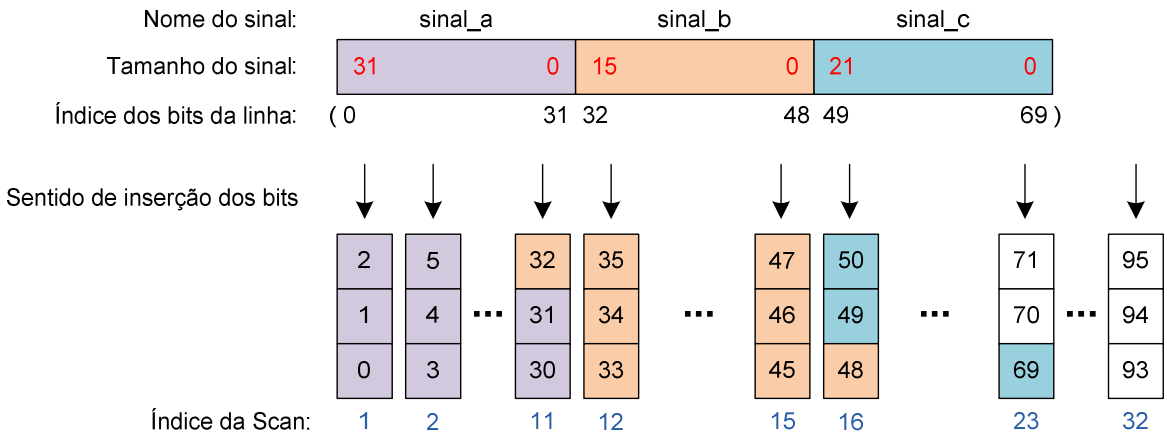


Figura 15 – Procedimento de inserção de dados na cadeia de *scan* de entrada.

Ao contrário da inserção, a extração dos dados da cadeia de entrada é feita de forma paralela para então ser entregue ao *DUT*. A forma como os bits são extraídos é apresentada na Figura 16. O controle não considera a parte da cadeia de *scan* 23 até a última cadeia, não extraindo seus bits.

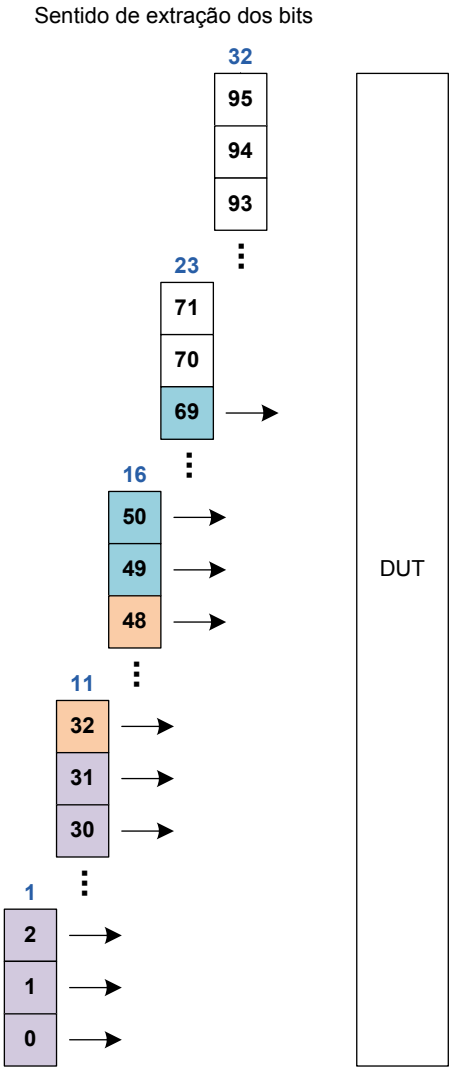


Figura 16 – Sentido de extração dos dados da cadeia de *scan* de entrada.

Da mesma forma que foram utilizados três sinais para exemplificar a inserção/extração de dados das cadeias de *scan* de entrada, a Figura 17 apresenta a inserção/extração da cadeia de saída destes mesmos dados.

Utiliza-se a Equação (1) para calcular a profundidade das cadeias de *scan* de saída. Os dados são enviados do *DUT* para as cadeias de forma paralela, como é possível se observar na Figura 17. Os bits de índice 0 a 69 preenchem as cadeias de índice 1 a 23 e a partir do bit 70 até o último bit da *scan* 32 a cadeia é preenchida com dados não válidos.

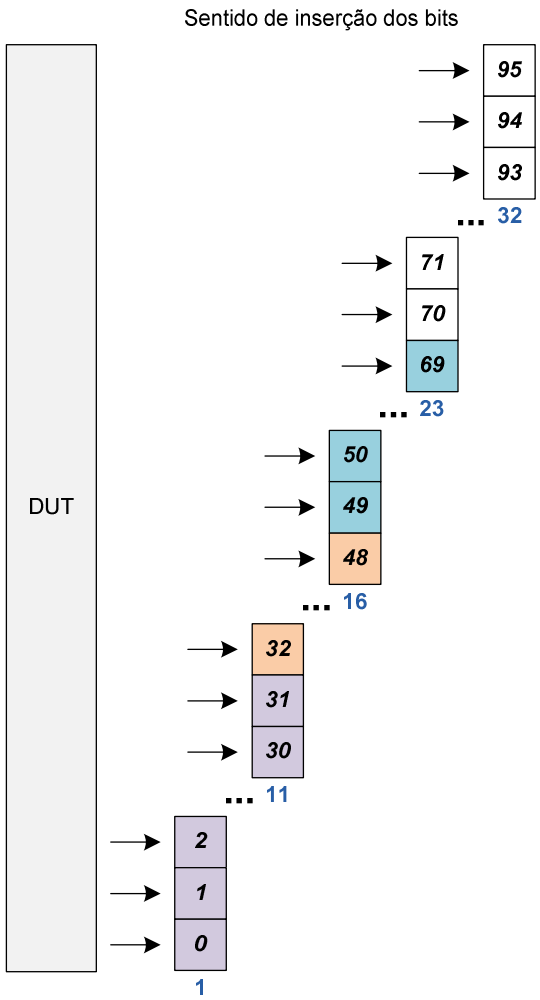


Figura 17 – Procedimento de inserção de dados na cadeia de *scan* de saída.

A extração dos dados das cadeias de *scan* de saída é feita de forma serial, como se pode observar na Figura 18.

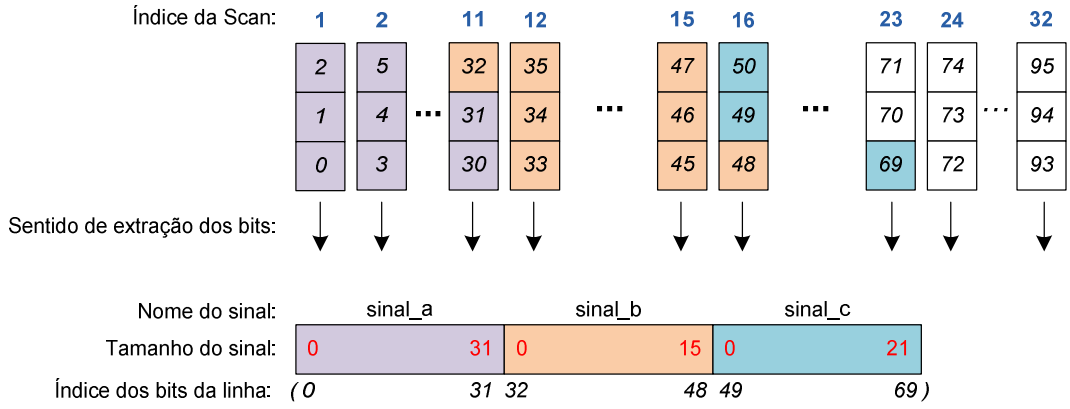


Figura 18 – Sentido de extração dos dados da cadeia de *scan* de entrada.

3.3 Protocolo e Interface de Comunicação

A comunicação entre o hospedeiro e o ambiente emulado no *FPGA*, via interface *PCIe*, é realizada através de um protocolo de comunicação. Para tanto, foi definido um protocolo específico para este fim. O protocolo é composto por palavras de 32 bits, de forma a casar o protocolo com a largura da palavra na interface *PCIe*. Sua estrutura permite a configuração do ambiente de emulação através de pacote de dados, possibilitando a reconfiguração do ambiente sem a necessidade de uma nova síntese do projeto. A Figura 19 apresenta os campos do protocolo. Observa-se que cada pacote contém uma informação de 8 bits que indica o tipo de pacote que esta sendo transmitido.

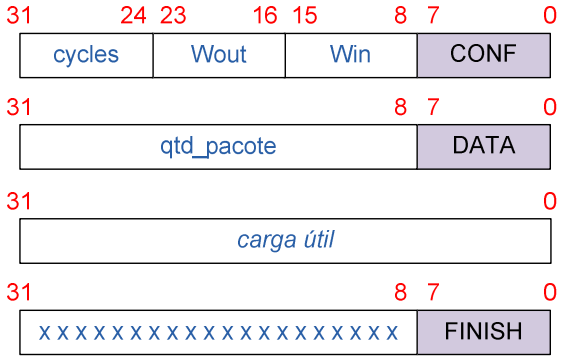


Figura 19 – Campos do protocolo de comunicação.

Existem três tipos de informações que são apresentadas na Tabela 6.

Tabela 6 – Descrição dos campos do protocolo de comunicação.

Campo	Identificador	Descrição
CONFIG	“00000001”	Configura o emulador.
DATA	“00000010”	Indica o começo da transmissão de dados úteis, e a quantidade de pacotes do mesmo.
FINISH	“00000011”	Indica o termino da transmissão.

Os pacotes a serem transmitidos ao ambiente de emulação são gerados através da *ferramenta de integração* (apresentada na Seção 4.2 do Capítulo 4). O primeiro pacote a ser transmitido é o de configuração. A única informação solicitada ao usuário para que este pacote seja montado, é a quantidade de ciclos de relógio que se deseja manter o *DUT* ativo para processamento dos dados. Este campo, denominado *cycles*, compreende os bits 24 a 31.

O pacote de configuração contém ainda mais dois campos. São eles: *Wout* – com a informação da profundidade da cadeia de *scan* de entrada (bits 16 a 23); *Win* – com a informação da profundidade da cadeia de *scan* de saída (bits 8 a 15).

O pacote seguinte é o que indica que o próximo pacote a ser enviado ao ambiente de emulação é de carga útil. Ele contém o campo chamado 'qtd_pacote' (bits 8 a 31) que carrega a informação de quantos *frames* de carga útil serão transmitidos na seqüência. Define-se *frame* como o número de palavras de 32 bits para preencher as cadeias de *scan* de entrada. Assim, um *frame* corresponde a um conjunto completo de dados para ser tratado em um ciclo de relógio do *DUT*.

Após o recebimento de toda a carga útil, o último pacote enviado ao ambiente de emulação é o pacote contendo a operação que indica que não haverá mais transmissão de dados.

3.3.1 Interface de Comunicação

A comunicação do hospedeiro com o ambiente de emulação é realizada através do barramento *PCIe*. Esta comunicação utiliza bibliotecas e módulos de hardware desenvolvidos pelo fabricante da placa (detalhada na Seção 2.1.1 do Capítulo 2), com modificações para servirem os propósitos do presente projeto. A comunicação por software e hardware será detalhada separadamente nos itens seguintes desta seção.

Software

O fabricante da placa utilizada neste projeto disponibilizou *drivers* para acesso à mesma no sistema operacional Linux, assim como um programa exemplo. Ambos são descritos em linguagem de programação 'C', que possui diretivas de baixo nível compatíveis com os requisitos desta aplicação.

O programa exemplo inclui funcionalidades para programar os *FPGAs*, ler dados do funcionamento de toda a placa, configurar parâmetros de funcionamento da mesma e comunicar-se por *DMA* com o dispositivo. Este aplicativo demonstra alguma das funcionalidades da placa utilizando apenas a *PCIe* para comunicação.

O programa deve ser compilado para o sistema operacional do hospedeiro. Para tal, é preciso

indicar o SO no arquivo *Makefile.make* e então executar o comando *make* que compila e liga todos arquivos fontes. Assim é gerado o programa teste: '*aetest_cmd_linux*' (hospedeiro está utilizando Linux).

Este programa possui funções que fogem do contexto do trabalho, sendo aqui apresentadas apenas as funcionalidades relacionadas com o presente projeto. São elas: configuração dos *FPGAs*, escrita por *DMA* e leitura por *DMA*. Para utilizá-las passa-se por parâmetro qual o comando desejado, conforme será explicado a seguir.

Para a configuração dos *FPGAs* por *PCIe*, deve-se executar o aplicativo desta maneira: "*./aetest_cmd_linux –F <FPGA> <arquivo_bitstream>*". O arquivo indicado por <arquivo_bitstream> será descarregado no *FPGA* indicado em <*FPGA*>, sendo os *FPGAs* configuráveis por *PCIe* são o "A" e "B" (vide descrição da placa na Seção 2.1.1).

Utilizando o *bitstream* de exemplo fornecido com a placa, é possível testar o funcionamento básico do *DMA* pela *PCIe*. O programa é utilizado da seguinte forma: "*./aetest_cmd_linux –DMAWR <arquivo_de_entrada> <deslocamento>*". Este comando lê o <*arquivo_de_entrada*> e escreve seu conteúdo na memória do *FPGA* a partir da posição indicada pelo <*deslocamento*>.

O arquivo pode ser lido invocando o programa da seguinte forma: "*./aetest_cmd_linux –DMARD <arquivo_de_saida> <deslocamento> <tamanho>*" que lê o conteúdo da memória a partir da posição <*deslocamento*> com tamanho ditado pela informação em <*tamanho*> e então escreve-o no arquivo que será criado com o nome <*arquivo_de_saida*>.

Utilizando este programa exemplo como base, foi adicionado outro parâmetro que pode ser chamado da seguinte forma: "*./aetest_cmd_linux –EMUL <arquivo_entrada.bin> <arquivo_saida.bin> <n_clocks>*" Este último parâmetro é a quantidade de ciclos de relógio de teste que o *DUT* ficará ativo para processar as informações oriundas das cadeias de *scan* de entrada.

O aplicativo desenvolvido lê do arquivo de entrada a profundidade das cadeias de *scan*, tanto de entrada como de saída, assim como o número de ciclos do *DUT* que serão emulados. Desta maneira o programa sabe a quantidade de dados que será escrita a cada ciclo de relógio de teste. Isto mantém o fluxo de dados proporcional à velocidade que o ambiente de emulação consome dados de entrada.

Os dados são escritos através do *DMA* em iterações que seguem a Equação (2). Salvo exceção no primeiro ciclo, aonde são escritas 2 palavras de 32 bits que contém as configurações do emulador.

$$num_{iteracoes} = \frac{Tam_{arquivoentrada}}{32 \times num_{scansentrada}} \quad (2)$$

Onde:

- *Num_interacoes*: Número de iterações de escrita;
- *Tam_arquivoentrada*: Tamanho em bits do arquivo de entrada;
- *Num_scansentrada*: Profundidade das cadeias de *scan* de entrada.

A cada iteração são escritos e lidos dados, conforme demonstra as equação (3) e (4) respectivamente:

$$num_{escritos} = 32 \times num_{scansentrada} \times C \quad (3)$$

$$num_{lidos} = 32 \times num_{scanssaida} \times C \quad (4)$$

Onde:

- *Num_escritos*: Número de bits escritos nas cadeias de scan de entrada;
- *Num_lidos*: Números de bits lidos das cadeias de scan de saída;
- *Num_scansentrada*: Profundidade das cadeias de scan de entrada;
- *Num_scanssaida*: Profundidade das cadeias de scan de saída;
- *C*: Constante que define quantas palavras são enviadas em rajada, de forma a reduzir o tempo com o protocolo de controle.

A escrita no *PCIe* é realizada utilizando buffers de entrada encontrados nos drivers de acesso à controladora *PCIe*, utilizando a função *ioctl* para interagir com dispositivos de entrada e saída do Sistema Operacional Linux.

Como este controle de fluxo não é suficiente para garantir que não haja erros de fluxo de dados, é utilizado também um registrador BAR (do inglês, *Base Address Register*). Este registrador é lido a cada iteração para garantir que os *buffers* de entrada e saída não sejam sobrescritos ou duplamente lidos. O registrador possui sinais indicando com ‘0’ ou ‘1’ se as memórias estão em um ou mais dos seguintes estados: memória de entrada quase vazia; memória de entrada quase cheia; memória de saída quase vazia; memória de saída quase cheia. Caso o sinal de “memória de entrada quase cheia” esteja habilitado, o controle interrompe temporariamente o envio de estímulos, analogamente, caso o sinal “memória de saída quase vazia” encontre-se ativo, a leitura de resultados do *FPGA* é interrompida temporariamente.

Processo de DMA da Dini Group

O *DMA* é uma técnica de movimentação de dados empregada para evitar que uma CPU seja sobrecarregada na escrita/leitura de grandes quantidades de dados em modo rajada.

Controladores de *DMA* usam descritores para orientá-los. Um descritor contém: tamanho, endereço de origem, endereço de destino da transferência de dados, indicadores de *status* e indicadores de modos de transferência. O controlador do *DMA* utiliza estas informações para executar as requisições de leitura ou escrita para movimentar os dados. Um bit é utilizado para representar se o descritor é válido. O software escreve o valor '1' no bit quando o descritor contém dados válidos. O hardware escreve o valor '0' quando acabar de mover os dados.

A Tabela 7 apresenta a estrutura dos descritores utilizados pelo controlador *DMA* do fabricante da placa. Os descritores são armazenados na memória do hospedeiro em uma fila circular. O software apenas informa ao controlador do *DMA* o endereço base e uma máscara de bits para o tamanho da fila circular, e desta forma o hardware tem informação suficiente para capturar descritores e processá-los.

Tabela 7 – Descritores utilizados pelo controlador *DMA* da Dini Group [DIN09b].

Palavra	Bit	Nome	Descrição
0	31	Válido	Bit indicando se o descritor é válido. Uma vez configurado para '1', o software não pode mudar nada no descritor.
0	30	Direção	'0': *do hospedeiro* (move dados do hospedeiro para o endereço na placa) '1': *para hospedeiro* (move dados do endereço da placa para o endereço do hospedeiro)
0	29	Gera Interrupção	Se '1', hardware interromperá o software uma vez que o descritor tenha sido completamente processado. O bit é ignorado quando o campo Configurar Tipos dos Registradores possuir valor '1'
0	28	Configurar Tipos dos Registradores	Valor '1' é utilizado para usar endereço como parte alta. Utilizado apenas caso queira acessar mais que 16GB de dados.
0	27	Tarefa de configuração do *FPGA*	Se '1', indica que é uma transferência de dados de/para o *FPGA* da Spartan (*FPGA* de configuração). Se '0', os dados são direcionados para o *FPGA* do usuário.
0	26	Reservado	Não utilizado
0	25	*Overflow*	Quando '1', o usuário retornou mais dados do que esperado.
0	24	Transferência completa	Bit de *status* modificado pelo hardware. Apenas para direção *para hospedeiro* e indica que é o fim desta transferência de dados.
0	23 .. 0	Tamanho em bytes	Valores válidos são: (4 até 64 MB) - 4 bytes. Este comprimento deve indicar um bloco contíguo de memória.
1	31..0	Endereço da placa	Endereço para ler ou escrever da placa.
2	31 .. 0	Endereço Hospedeiro (bit 31 até 0)	Endereço físico na memória do hospedeiro. Notar que os dados necessitam estar contíguos durante toda a transferência.
3	31 .. 0	Endereço Hospedeiro Parte Alta (bit 63 até bit 32)	Parte alta do endereço do hospedeiro. Caso endereçamento de 32 bits seja utilizado, este valor deve ser preenchido com zeros.

Registradores *BAR0* são utilizados no funcionamento do *DMA*, enquanto os demais ficam livres para implementações do usuário. No presente projeto o registrador *BAR4* foi utilizado para controle de fluxo da comunicação.

Hardware

Acompanhando a placa, os desenvolvedores da mesma incluíram módulos de hardware referente às suas funcionalidades, incluindo módulos de interação com a *PCIe*. Toda estrutura da placa foi descrita em *Verilog* com alguns módulos de exemplo em *VHDL*. Assim como no software, o hardware fornecido implementa um *DMA* simples. Este exemplo foi utilizado como base para o desenvolvimento do módulo do presente projeto.

Os módulos fornecidos para interagir com a *PCIe* são: *pcie_x8 _user_interface.vhd* e *user_blockram_example.vhd* (vide Anexo A – estrutura dos módulos). O primeiro é responsável pela interface com a *PCIe*. Este é o arquivo que deve ser alterado caso se queira modificar a forma de comunicação com a *PCIe*. Sua configuração inicial opera com DMA (outros modos incluem: envio direto de dados e pipeline). O segundo módulo, é o que implementa o *DMA*. Este foi o arquivo modificado para receber o ambiente de emulação. Ambas as entidades são instanciadas uma única vez no arquivo *fpga.v* , o topo do projeto, mas apenas o arquivo *pcie_x8_user_interface.vhd* tem ligação direta com a interface *PCIe, oriundas do FPGA Q*. O exemplo fornecido recebe os sinais do *pcie_x8_user_interface.vhd* e os trata de forma a escrever e ler os dados em duas memórias de 512 posições. O hardware descrito no arquivo *user_blockram_exemple.vhd* trabalha com o mesmo *DMA* apresentado na Tabela 7. Este projeto exemplo apenas realiza a escrita e leitura dos dados via *DMA* nestas memórias, tratando-as como uma memória de 64 bits.

As requisições de leitura e escrita são realizadas sem levar em consideração sobrescrita de dados ou leituras repetidas, deixando este controle a cargo do software. O exemplo também implementa funcionalmente os registradores *BAR* como memórias de 512 posições e 64 bits.

O protótipo proposto no presente trabalho utilizou este exemplo como ponto de partida para comunicação entre o hospedeiro e *FPGA*. Foi decidido utilizar o sistema já validado do fornecedor da placa, para economizar tempo de desenvolvimento, testes, depuração, entre outros.

O módulo desenvolvido para o presente TCC substitui, inclusive com as mesmas entradas e saídas, o *user_blockram_example.vhd*. Este módulo foi nomeado de *interface_dma.vhd*. Em essência o funcionamento é o mesmo, porém as memórias de leitura e escrita foram duplicadas. São instanciadas quatro memórias *RAM* de 512 posições com 32 bits cada, agrupadas de forma a agir como duas memórias de 64 bits. Ambas as memórias são responsáveis pela comunicação entre o

hospedeiro e o ambiente de emulação, porém uma para cada direção. Ou seja, uma memória é preenchida pelo hospedeiro e lida pelo ambiente de emulação, enquanto outra é preenchida pelo ambiente e lida pelo hospedeiro, como se pode observar na Figura 20. Esta Figura mostra que futuramente pode-se modificar os *drivers* de comunicação, de forma que os mesmos trabalhem com palavras de 64 bits, aumentando-se assim o desempenho da comunicação.

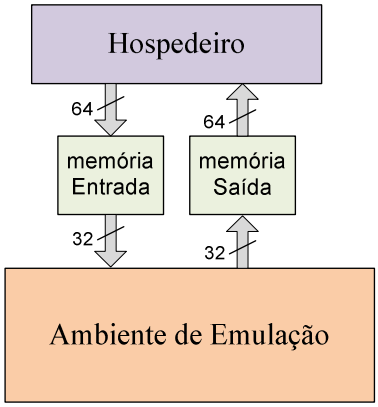


Figura 20 – Utilização das memórias ao FPGA, utilizadas como *buffers*.

As memórias operam como buffers circulares, constantemente sendo preenchidas e lidas por ambas as partes. Para evitar erros na comunicação devido a natureza cíclica do processo, a máquina de controle mantém os deltas de escrita e leitura armazenados. Isto evita que em situações como a memória quase vazia, sejam enviados dados repetidos e inválidos; assim como na possibilidade análoga de ter a memória quase cheia e sobrescrever dados não lidos ainda.

Os sinais dentro do modulo *interface_emulator* (arquivo *interface_emulator.vhd*) possuem prefixos que indicam a qual sentido da comunicação o sinal esta relacionado. Por exemplo, sinais precedidos de "*hte_*" são relacionados à comunicação Hospedeiro – Emulador, e "*eth_*" são responsáveis pela comunicação Emulador – Hospedeiro.

Os deltas de escrita e leitura são chamados de *hte_diferenca* e *eth_diferenca.* Seus funcionamentos são análogos, portanto apenas será detalhado o funcionamento de um deles. O sinal *hte_diferenca* é inicializado no estado de *reset* com o valor 0, a partir de então, para cada palavra de 32 bits escrita pelo Hospedeiro, é incrementado 1 ao *hte_diferenca* e a cada palavra lida pelo emulador, decrementa-se 1 deste sinal. Isto garante que enquanto o sinal *hte_diferenca* for positivo e menor que o número de palavras de 32 bits máximo da memória, os dados não serão sobrescritos, ou lidos repetidamente.

Para evitar que as situações de memória vazia ou memória cheia aconteçam, foram definidas constantes que indicam estados de *quase vazio* e *quase cheio.* Inicialmente estes valores foram estipulados em 20, para *quase vazio* e 400 para *quase cheio*. Enquanto o sinal de diferença estiver

abaixo de *quase vazio* nenhum dado pode ser lido, apenas escrito. E quando o valor de diferença for maior que *quase cheio* nenhum dado pode ser escrito, apenas lido. Um estudo mais aprofundado sobre estes valores deve garantir um uso otimizado dos buffers de entrada e saída, conforme será descrito na Seção 6.

Como os dados chegam em palavras de 64 bits para o *FPGA*, os dados são armazenados em duas memórias separados em parte baixa e parte alta. Uma vez que a quantidade de dados escritos ultrapasse o limite escolhido para caracterizar a memória como *quase vazia*, a memória de entrada sai do estado de *quase vazio* e a interface disponibiliza dados para o emulador começar a ler palavras de 32 bits.

Uma vez prontos os dados nas cadeias de *scan*, o ambiente de emulação inicia a comunicação com a interface, que extrai os dados com um protocolo de *handshake* de três fases. A memória é preenchida, e quando sai do estado de *quase vazia*, o hospedeiro começa a ler os dados. A leitura de dados, também é realizada utilizando os *drivers* disponibilizados pelo fabricante, através de diretivas do SO. Os dados são então armazenados da maneira que saem da cadeia de *scan* de saída em um arquivo binário que posteriormente será rearranjado e transformado em um *VCD* (vide Seção Arquivo *VCD* e GTKWave do Capítulo 4).

3.4 Paralelismo Cadeias de Scan Entrada e Saída

A simulação apresentada na Figura 21 visa exemplificar o paralelismo das cadeias de *scan*, utilizando cadeias com profundidade 16 (entrada) por 8 (saída). Os detalhes presentes na figura serão detalhados a seguir. Desconsidera-se o *DUT* utilizado na simulação.

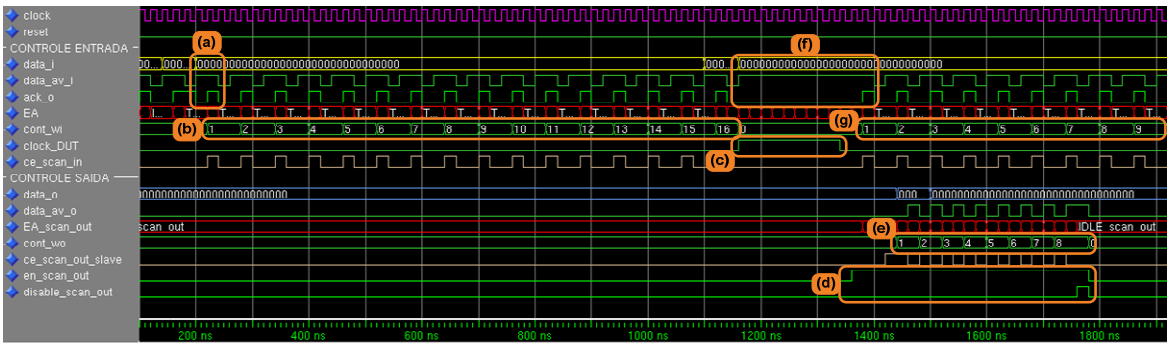


Figura 21 – Simulação do ambiente de emulação. Cadeias de *scan* de profundidade 16 (entrada) por 8 (saída).

Em (a) observa-se o sinal *data_av_i*, oriundo da *PCIe*, transicionando de nível lógico baixo para alto, indicando disponibilidade de dado válido a ser consumido pela cadeia de *scan* de entrada. Ainda em (a), o sinal *ack_o* transiciona de nível lógico baixo para alto informando a *PCIe* que o dado foi consumido, pois em (b) o dado é inserido na cadeia de *scan* de entrada. Nos ciclos

seguintes, o dado é deslocado até preencher a profundidade total da cadeia.

Com a cadeia preenchida, o sinal *clock_DUT* transiciona para nível lógico alto (detalhe (c)), e permanece assim enquanto não atingir a quantidade pré-configurada de ciclos de relógio de teste. Então o sinal *en_scan_out* transiciona para nível lógico alto habilitando a máquina de controle da cadeia de *scan* de saída (detalhe (d)). O sinal *en_scan_out* permanece em nível lógico alto até ser desabilitado pelo sinal *disable_scan_out* indicando que a máquina de controle de saída foi finalizada.

No detalhe (e) os dados são deslocados na cadeia de *scan* de saída, enquanto em (f) novos dados estão disponíveis para serem consumidos pela cadeia de *scan* de entrada. Por fim, em (g) novos dados são inseridos na cadeia de entrada e deslocados até o preenchimento da mesma, caracterizando assim o paralelismo das cadeias de *scan* de entrada e saída.

4. FLUXO DE EMULAÇÃO

O presente Capítulo apresenta o fluxo do ambiente de emulação, bem como o detalhamento de cada etapa. A Seção 4.1 introduz a ferramenta analisadora léxica/sintática utilizada para possibilitar a observação de sinais específicos do circuito em emulação. Por fim, a Seção 4.2 descreve a ferramenta de integração utilizada para unir o *DUT*, ao controle entre *PCIe* e cadeias de *scan*, e bibliotecas de cadeias de *scan*.

A Figura 22 apresenta o fluxo completo do ambiente de emulação, assim como as ferramentas e módulos de hardware criados.

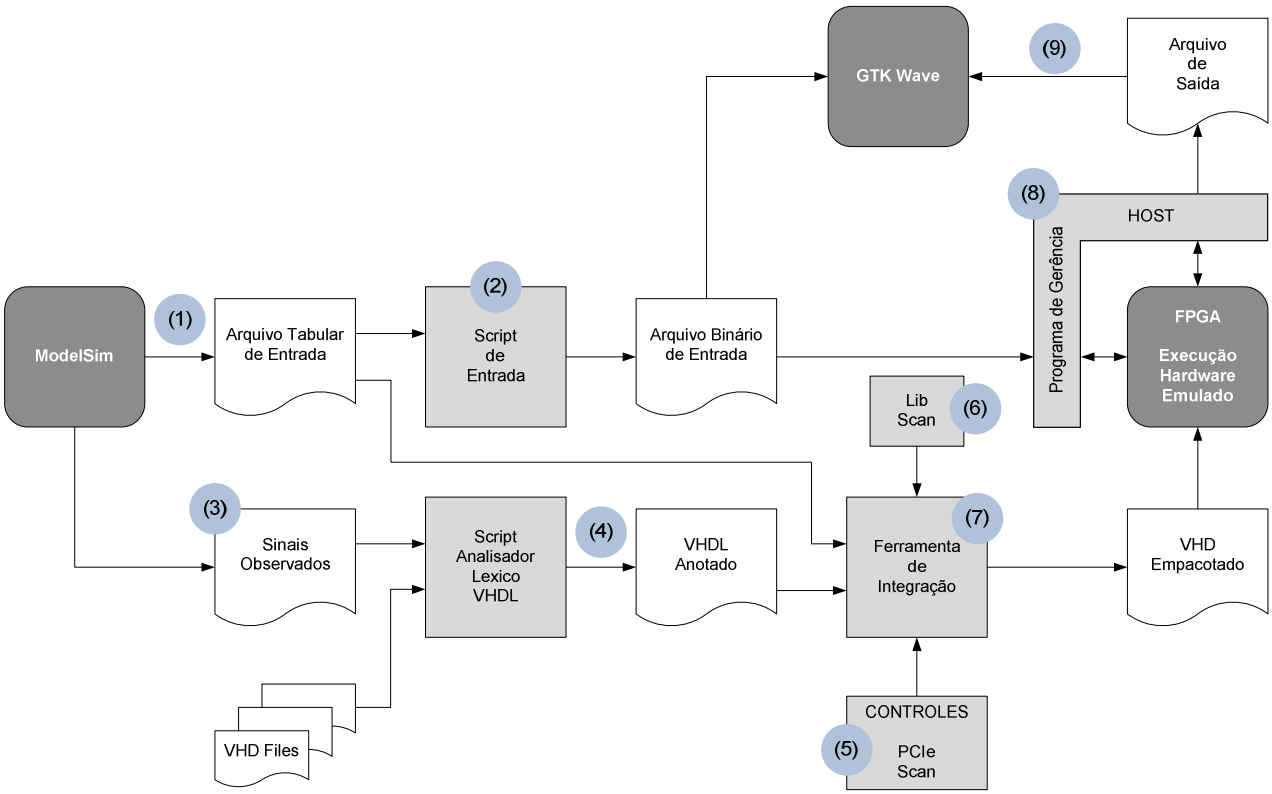


Figura 22 – Fluxo do ambiente de emulação.

(1) A ferramenta ModelSim é utilizada para gerar um arquivo tabular contendo os estímulos dos sinais de entrada do *DUT*.

(2) O arquivo tabular de entrada gerado na simulação do ModelSim é transformado em um arquivo binário através de um *script*. Este arquivo binário será usado como entrada do ambiente de emulação.

(3) A ferramenta ModelSim é também utilizada para gerar um arquivo com os sinais a serem observados na emulação do sistema.

(4) A ferramenta denominada ‘Parser’ é responsável por modificar os arquivos *VHDL* do *DUT* de acordo com os sinais a serem observados. No início deste TCC foram consideradas duas formas de realizar esta atividade: pré-síntese ou pós-síntese. A anotação pós-síntese apresenta como desvantagem a manipulação de formatos proprietários (e.g. NGC), além de diversos sinais terem os nomes alterados ou serem suprimidos de acordo com as otimizações adotadas pela ferramenta de síntese. A anotação de sinais pré-síntese envolve manipulação de arquivos *VHDL* e a geração de novos arquivos *VHDL*, com os sinais observados no topo da hierarquia. Apesar da complexidade na manipulação dos arquivos *VHDL*, esta foi a estratégia adotada neste TCC, devido à maior flexibilidade de anotação de sinais.

(5) Módulos de hardware: controlador *PCIe* e controlador das cadeias de *scan*. Utilizou-se o controlador *PCIe* desenvolvido pelo fabricante da placa, adaptando-o ao projeto. O controlador das cadeias de *scan* é responsável por receber os pacotes oriundos do hospedeiro e determinar a seqüência de operações realizadas sobre o *DUT*.

(6) Biblioteca que contém as cadeias de *scan* de entrada e saída.

(7) De posse de: (i) *VHDL* anotado; (ii) controladores *PCIe* e das cadeias de *scan*; (iii) biblioteca com as cadeias de *scan*; a ferramenta que integra todos estes módulos gera o *DUT* para emulação. Este novo projeto é sintetizado e enviado à placa de prototipação.

(8) Software a ser executado no hospedeiro. Este software executa o protocolo de comunicação com o barramento *PCIe*, sendo responsável por: (i) enviar os estímulos obtidos na etapa 2 para o *FPGA*; (ii) receber os dados processados do *DUT*. Os dados processados são armazenados em um arquivo.

(9) Software que lê os arquivos de entrada e saída, disponibilizando as informações da emulação em formas de ondas a serem analisadas pelo projetista.

4.1 Ferramenta de Anotação do Código VHDL

O propósito de se modificar o código original de um projeto para levar os sinais ao topo da hierarquia do mesmo se deve à necessidade de acesso a estes sinas, ou seja, prover *observabilidade* durante a emulação. Desta forma, os sinais podem ser conectados às cadeias de *scan* com o objetivo de se observar seu comportamento. Por este motivo foi desenvolvida uma ferramenta, denominada *Parser*, para atender as necessidades do presente trabalho. A Figura 23 apresenta o fluxo de execução da ferramenta.

(1) De posse dos arquivos fonte do projeto, o usuário deve utilizar a ferramenta ModelSim para gerar um arquivo com os sinais a serem observados na emulação do sistema. Exemplo de arquivo apresentado posteriormente na Figura 27.

(2) A ferramenta apresentada em [REH09] é utilizada para traduzir as descrições *VHDL* do *DUT*, em código *XML*. O Anexo B contém um exemplo de parte de um arquivo convertido para *XML*. O objetivo é apenas apresentar a estrutura do arquivo. O detalhamento da mesma não faz parte do escopo deste trabalho.

(3) A ferramenta *Parser* utiliza os arquivos gerados nas etapas (1) e (2) e ainda os arquivos originais do *DUT*, para gerar arquivos *VHDL* anotados. Os arquivos originais do *DUT* são modificados de acordo com os sinais a serem observados, fornecidos pelo arquivo *wave.do*.

(4) Por fim, o Anexo C apresenta um exemplo de parte de um arquivo anotado, gerado pela ferramenta *Parser*. Nele é possível se observar sinais internos que a ferramenta subiu na hierarquia do *DUT* até o topo do mesmo. Os sinais são precedidos por '*annotated_*' (destacado em vermelho no anexo) + hierarquia até chegar ao nome do sinal.

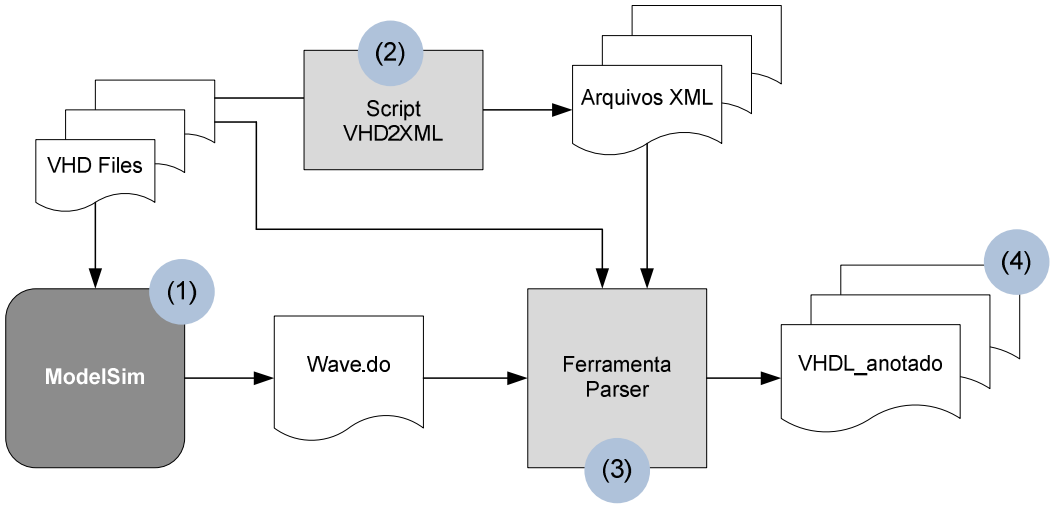


Figura 23 – Fluxo da ferramenta analisadora léxica/sintática.

4.2 Ferramenta de Integração

Uma ferramenta foi criada com o objetivo de integrar os módulos de hardware desenvolvidos e facilitar a inserção de dados, bem como a manipulação de arquivos. A ferramenta necessita de algumas informações de entrada que são apresentadas na etapa (7) da Figura 22 deste Capítulo e melhor detalhadas na presente seção.

A interface gráfica da *Ferramenta de Integração* apresenta inicialmente um tutorial sobre os passos que devem ser seguidos pelo usuário, como pode ser observado na Figura 24.

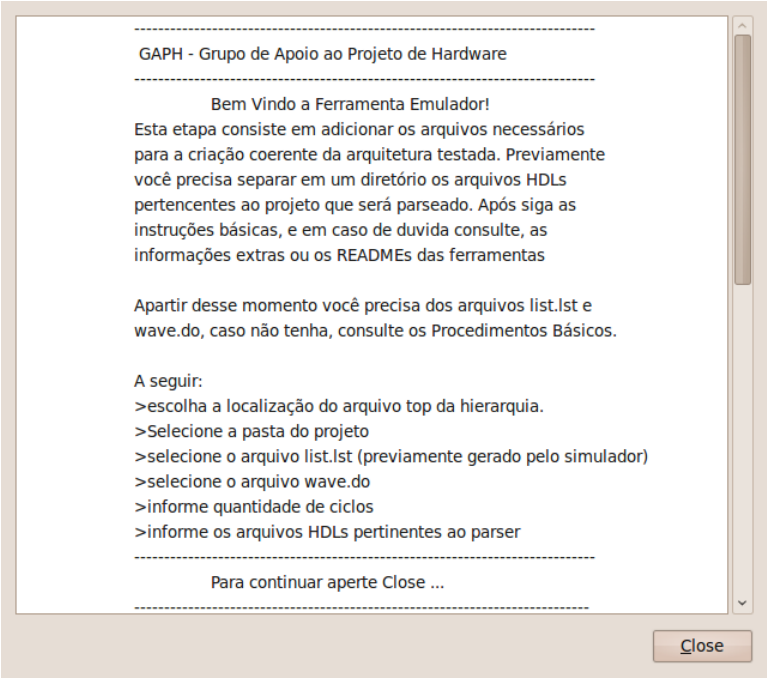


Figura 24 – Interface gráfica da *Ferramenta de Integração*.

A primeira informação solicitada pela ferramenta é o arquivo que esta no topo da hierarquia, conforme Figura 25. As demais informações necessárias para utilização da *ferramenta* seguem na seguinte ordem: arquivo tabular com dados de entrada (*list.lst*), arquivo com sinais a serem observados (*wave.do*), número de ciclos que o *DUT* ficará ativo, diretório que possui os arquivos do *DUT*. Estas informações são solicitadas da mesma forma que na Figura 25. Portanto, é válido ressaltar que o usuário gere antes de utilizar a *ferramenta*, os arquivos *list.lst* e *wave.do*.

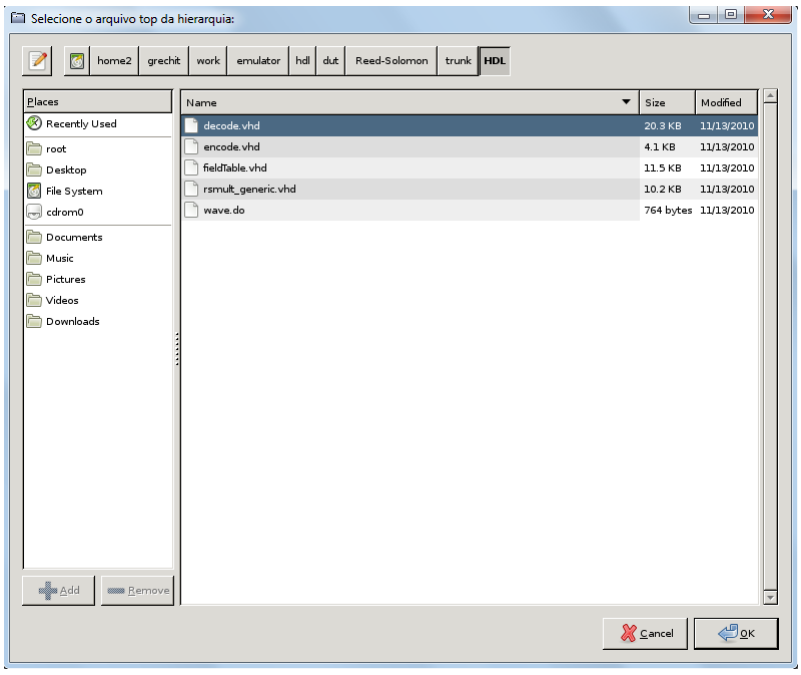


Figura 25 – Solicitação do arquivo topo da hierarquia.

O arquivo tabular *list.lst* é gerado através de uma simulação do *DUT* na ferramenta ModelSim. O usuário deve selecionar os sinais de *clock* e *reset* do circuito, bem como todas suas entradas. Portanto, o arquivo tabular irá conter os estímulos de entrada a serem inseridos no ambiente de emulação.

O usuário deve executar um *script*, denominado *list.do* (estrutura do arquivo apresentada na Figura 26), na ferramenta ModelSim que tem por objetivos: (i) retirar o caminho de cada sinal, deixando apenas seu nome; (ii) deixar os sinais em uma mesma linha. A Figura 27 apresenta um exemplo de arquivo tabular pós execução do *script list.do*.

```
#Configuracao da list
configure list -usestrobe 0
configure list -strobestart {0 ns} -strobeperiod {0 ns}
configure list -usesignaltrigger 1
configure list -delta none
configure list -signalnamewidth 1
configure list -datasetprefix -1
configure list -namelimit 1
#Grava arquivo list
write list -window .main_pane.list list.lst
```

Figura 26 – *Script list.do* executado pelo usuário na ferramenta ModelSim

```
ns                    clock  reset        X1         Y1         X2         Y2

  0                       0      0 0000000000 0000000000 0000000000 0000000000
 25                       1      0 0000000000 0000000000 0000000000 0000000000
 50                       0      1 0000000000 0000000000 0000000000 0000000000
 75                       1      1 0000000000 0000000000 0000000000 0000000000
100                       0      1 0000100111 0000000000 0001001000 0000000101
125                       1      1 0000100111 0000000000 0001001000 0000000101
150                       0      1 0000100111 0000000000 0001001000 0000000101
175                       1      1 0000100111 0000000000 0001001000 0000000101
200                       0      1 0000100111 0000000000 0001001000 0000000101
225                       1      1 0000100111 0000000000 0001001000 0000000101
250                       0      1 0000100111 0000000000 0001001000 0000000101
275                       1      1 0000100111 0000000000 0001001000 0000000101
300                       0      1 0000100111 0000000000 0001001000 0000000101
325                       1      1 0000100111 0000000000 0001001000 0000000101
350                       0      1 0000100111 0000000000 0001001000 0000000101
375                       1      1 0000100111 0000000000 0001001000 0000000101
400                       0      1 0000100111 0000000000 0001001000 0000000101
425                       1      1 0000100111 0000000000 0001001000 0000000101
450                       0      1 0000100111 0000000000 0001001000 0000000101
475                       1      1 0000100111 0000000000 0001001000 0000000101
500                       0      1 0000100111 0000000000 0001001000 0000000101
525                       1      1 0000100111 0000000000 0001001000 0000000101
550                       0      1 0000100111 0000000000 0001001000 0000000101
575                       1      1 0000100111 0000000000 0001001000 0000000101
600                       0      1 0000100111 0000000000 0001001000 0000000101
625                       1      1 0000100111 0000000000 0001001000 0000000101
650                       0      1 0000100111 0000000000 0001001000 0000000101
675                       1      1 0000100111 0000000000 0001001000 0000000101
700                       0      1 0000100111 0000000000 0001001000 0000000101
```

Figura 27 – Exemplo de um arquivo *list. lst*.

A *ferramenta de integração* é responsável por inserir o protocolo de comunicação (apresentado na Seção 3.3). Para tanto, ela faz uso da Equação (1) e da definição de *frame* (ambos apresentados na Seção 3.3). Com a Equação (1) é definida a profundidade das cadeias de *scan* e com a definição de *frame* a quantidade de pacotes de dados úteis a serem processados. Além disso, a *ferramenta* ainda insere a quantidade de ciclos de relógio de teste que o *DUT* deve ficar ativo, informação esta fornecida pelo usuário através da própria *ferramenta*.

O arquivo binário contendo o protocolo e dados úteis gerado pela *ferramenta* é apresentado na

Figura 28. É importante ressaltar que cada linha do arquivo representa um ciclo de relógio de teste.

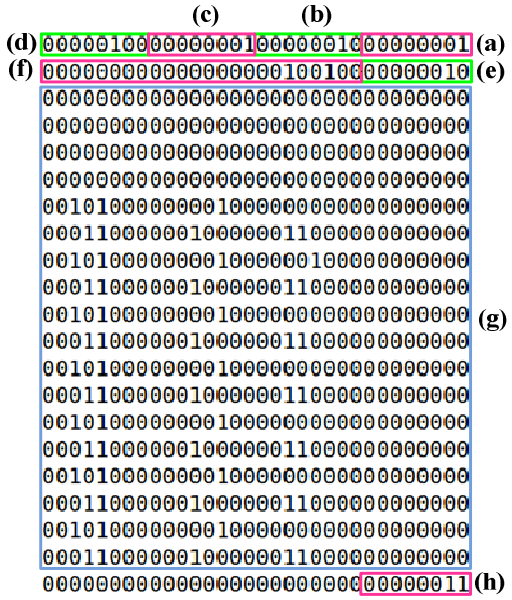


Figura 28 – Exemplo de um arquivo binário de entrada. (a) campo indicando pacote de configuração; (b) profundidade das cadeias de scan de entrada; (c) profundidade das cadeias de scan de saída; (d) quantidade de ciclos de relógio de teste com o *DUT* ativo; (e) campo indicando pacote de dados; (f) quantidade de frames; (g) dados úteis; (h) campo indicando final teste.

O arquivo *wave.do* que contém os sinais a serem observados também é gerado através da ferramenta ModelSim. Apenas as informações relativas aos sinais (em destaque na Figura 29(a) serão utilizadas pela *Ferramenta de Integração.*

```
onerror {resume}
quietly WaveActivateNextPane {} 0
add wave -noupdate /DUT/deltaX
add wave -noupdate /DUT/deltaY
add wave -noupdate /DUT/dispX
add wave -noupdate /DUT/dispY
add wave -noupdate /DUT/dispXAux
add wave -noupdate /DUT/loop_cnt
add wave -noupdate /DUT/X
add wave -noupdate /DUT/Y
TreeUpdate [SetDefaultTree]
WaveRestoreCursors {{Cursor 1} {0 ns} 0}
configure wave -namecolwidth 150
configure wave -valuecolwidth 100
configure wave -justifyvalue left
configure wave -signalnamewidth 0
configure wave -snapdistance 10
configure wave -datasetprefix 0
configure wave -rowmargin 4
configure wave -childrowmargin 2
configure wave -gridoffset 0
configure wave -gridperiod 1
configure wave -griddelta 40
configure wave -timeline 0
configure wave -timelineunits ns
update
WaveRestoreZoom {0 ns} {1 us}
```

(a)

Figura 29 – Exemplo de um arquivo *wave.do*.

Com as informações solicitadas pela *Ferramenta de Integração* atendidas, a mesma esta pronta pra realizar o processo de inserir as cadeias de *scan* no *DUT*. A *ferramenta* invoca a

ferramenta *Parser* (descrita na seção 4.1 deste Capítulo) para que os sinais contidos no arquivo *wave.do* subam na hierarquia, sendo então acessíveis no topo do projeto. Desta forma, é possível conectar as cadeias de *scan* ao *DUT*.

A *Ferramenta de Integração* consulta o arquivo binário de entrada para ler as informações de profundidade das cadeias de *scan* de forma a empregar a correta profundidade disponibilizada pela biblioteca de cadeias de *scan*.

Por fim, a *ferramenta* realiza o mapeamento do circuito de controle, do *DUT* e das cadeias de *scan* gerando um novo *VHDL* em um diretório denominado *emulador_finalizado*. Este diretório conterá os seguintes arquivos: *emulador_dut_scan.vhd* – controle da comunicação com a *PCIe* e inserção/extração de dados das cadeias de *scan*; *types_pkg.vhd* – pacote com as constantes utilizada pelo *emulador_dut_scan.vhd*; *scan_ff_in.vhf* – cadeia de *scan* de entrada conectada ao *DUT*; *scan_ff_out.vhf* – cadeia de *scan* de saída conectada ao *DUT*; *informações_de_entrada.txt* – arquivo com o protocolo de comunicação e dados úteis; *informações_de_saida.txt* – arquivo utilizado para geração do *VCD* (explicado na Seção seguinte).

Arquivo *VCD* e GTKWave

Para atender a necessidade de comparação dos dados de entrada com os de saída, foi criado um *script* para gerar um arquivo *VCD* [IEE01]. O *script* denominado *vcd_generator.py*, necessita dos seguintes parâmetros para gerar o arquivo *VCD*: informações de entrada – arquivo contendo os nomes dos sinais de entrada com seus respectivos tamanhos; informações de saída – arquivo contendo os nomes dos sinais observados com seus respectivos tamanhos; arquivo tabular – arquivo contendo os dados úteis de entrada; arquivo de saída – arquivo contendo os dados úteis de saída

O arquivo *VCD* é gerado ao final da execução da emulação e para visualizá-lo foi utilizada a ferramenta GTKWave [BYB10]. Esta ferramenta exibe os resultados dos dados úteis de entrada e saída em formas de onda.O Anexo D ilustra, de forma parcial, um arquivo *VCD*, e a Figura 32 (da Seção 5.2) apresenta a visualização deste mesmo arquivo *VCD* pela ferramenta GTKWave.

5. AVALIAÇÃO DO EMULADOR

O presente Capítulo apresenta a validação do ambiente de emulação. A Seção 5.1 contém a validação em software, enquanto a 5.2 em hardware.

5.1 Validação em Software

De forma a se validar o hardware desenvolvido, foram criados três cenários de teste, onde cada um foi simulado na ferramenta ModelSim. A validação se procedeu como segue: (i) simulação do *DUT*; (ii) simulação do *DUT* no ambiente de emulação; (iii) avaliação dos dados resultantes na ferramenta GTKWave.

A simulação do *DUT* fez-se necessária para fins de validação. A simulação do *DUT* no ambiente de emulação compreende o envio e a recepção dos dados diretamente nas cadeias de *scan*, sem a utilização do controlador *PCIe*. Um terceiro ambiente de simulação poderia conter o controlador *PCIe*, porém julgou-se desnecessário sua realização, dada a simplicidade do protocolo de comunicação entre o módulo gerado pela ferramenta de integração e o controlador *PCIe*, conforme descrito no Anexo A.

A Figura 30 apresenta uma simulação ilustrando a comunicação entre o *DUT* no ambiente de emulação e o controlador *PCIe*. Os detalhes presentes na figura serão detalhados a seguir.

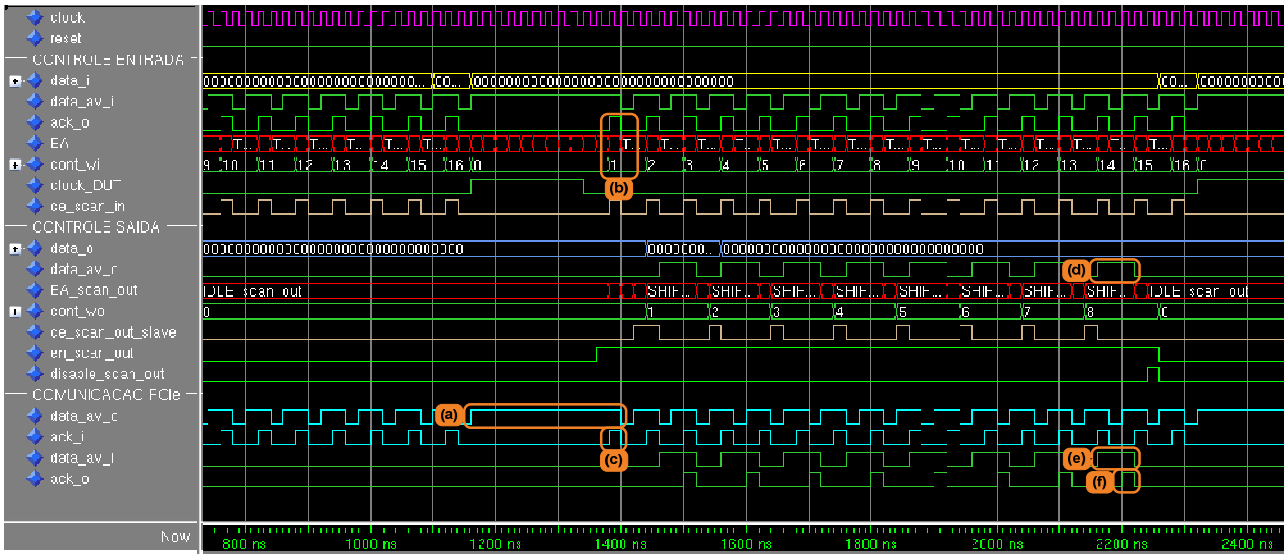


Figura 30 – Comunicação entre *DUT* no ambiente de emulação e o controlador *PCIe*.

O controlador *PCIe* indica a disponibilidade de dados para o ambiente de emulação através do

sinal *data_av_o*. No detalhe (a) é possível visualizar a transição deste sinal de nível lógico baixo para alto, indicando dado disponível. O sinal *data_av_o* permanece em nível lógico alto até que a cadeia de *scan* de entrada consuma o dado – detalhe (b). Em (c) o sinal *ack_i* transiciona para nível lógico alto indicando que o dado foi consumido, e o sinal *data_av_o* retorna para nível lógico baixo. As transições seguintes dos sinais *data_av_o* e *ack_i* funcionam da mesma forma, disponibilizando mais dados ao ambiente de emulação.

De forma análoga o ambiente de emulação também possui os sinais *data_av_o* e *ack_i* para indicar disponibilidade de dados extraídos das cadeias de *scan* de saída. Em (d) o sinal *data_av_o* indica dado pronto a ser enviado para o controlador *PCIe*. O sinal do *data_av_i* do controlador *PCIe* transiciona para nível lógico alto – detalhe (e) – e se mantém assim até o dado ser consumido. Em (f) o sinal *ack_o* transiciona para nível lógico baixo alto indicando que o dado foi consumido, e o sinal *data_av_i* retorna para o nível lógico baixo – detalhe (e).

A Tabela 8 apresenta os *DUTs* utilizados nos cenários de teste, bem como seus sinais de entrada e sinais observados. A escolha destes cenários deu-se pelo motivo de iniciarmos a validação por um circuito de menor complexidade para outros de maior. Os sinais a serem observados foram selecionados no intuito de se ter diferentes configurações de profundidade das cadeias de *scan*.

Tabela 8 – Cenários de teste com respectivos sinais de entrada e saída (sinais observados).

DUT	Descrição	Sinais Entrada	Sinais Observados
Somador	Somas parciais dos seguintes sinais: a + b; a+b+c+d; e+f+g+h; soma de todos os valores de entrada.	a (64 bits) b (64 bits) c (64 bits) d (64 bits) e (64 bits) f (64 bits) g (64 bits) h (64 bits)	annotated_somador_saida_a_b (64 bits) annotated_somador_saida_a_b_c_d (64 bits) annotated_somador_saida_e_f_g_h (64 bits) saida_all (64 bits)
Sqrt	Raiz quadrada de um valor	valor (64 bits)	annotated_sqrt_iteracoes (9 bits) annotated_sqrt_raiz (64 bits) annotated_sqrt_reg (128 bits) result (32 bits)
Framer	Módulo interno de um *transponder OTN*	entrada(64 bits)	annotated_framer_regin(64 bits) annotated_framer_estagio1 64 bits) annotated_framer_estagio2 64 bits) annotated_framer_estagio3 64 bits) annotated_framer_alinhador1out 64 bits) annotated_framer_comparadorfull_compara (1 bit) annotated_framer_comparadorfull_deslocado (48 bits) annotated_framer_enderecador_controle (64 bits) annotated_framer_enderecador_endereco (6 bits)

Para fins de análise de custo, os *DUTs* foram sintetizados utilizando a ferramenta ISE 11.4, da empresa Xilinx. Os resultados de síntese dos DUTS com as cadeias de *scan* e o controlador das mesmas, são apresentados na Tabela 9.

Tabela 9 – Resultados de síntese da validação em software, dispositivo xc5vlx330-2ff1760, ISE 11.4

DUT	Profundidade *Scans*		Original			DUT + Cadeias de Scan			*Overhead*		
	Entrada	Saída	*LUTs*	Regs.	*BUFGs*	*LUTs*	Regs.	*BUFGs*	*LUTs*	Regs.	*BUFGs*
Somador	16	8	692	0	1	1908	1437	2	1216	1437	1
Sqrt	2	8	311	237	1	1119	944	2	808	237	1
Framer	2	14	904	849	3	1685	1538	3	781	689	0

Cenário de Teste (1): Somador

O *DUT* utilizado neste primeiro cenário de teste foi um somador que realiza a soma de 8 valores de 64 bits cada, configurando assim a utilização de uma cadeia de *scan* de entrada de profundidade 16. O *DUT* de baixa complexidade, comparado aos demais, foi criado para dar início à validação do ambiente, realizando somas parciais atribuídas a sinais internos, com o objetivo de possibilitar a observação destes sinais internos e não só a saída primária do circuito.

A quantidade de bits dos sinais observados configurou a utilização de uma cadeia de *scan* de saída de profundidade 8. A Figura 31 apresenta a simulação do *DUT* na ferramenta ModelSim. Esta simulação corresponde à aplicação direta dos estímulos no *DUT* e a correspondente visualização.

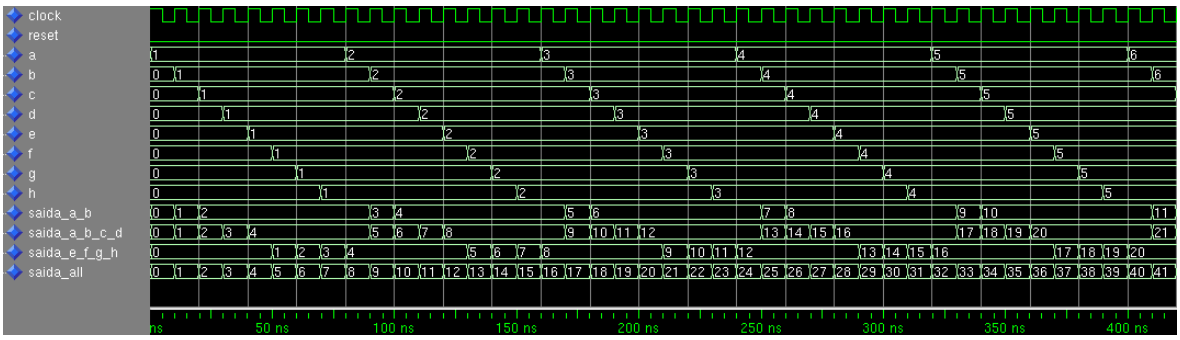


Figura 31 – Simulação do *DUT* (Somador) no ModelSim.

A partir dos vetores de teste da simulação apresentada na Figura 31, e dos sinais que desejamos observar, executamos o fluxo de emulação para a geração do *VHDL* anotado. Uma vez obtido o *VHDL* anotado, fez-se a simulação com as cadeias de *scan*. A Figura 32 apresenta a interface do GTKWave com os sinais lidos do arquivo *VCD*. Ambas as simulações apresentam o mesmo resultado, demonstrando a correta funcionalidade do ambiente de emulação.

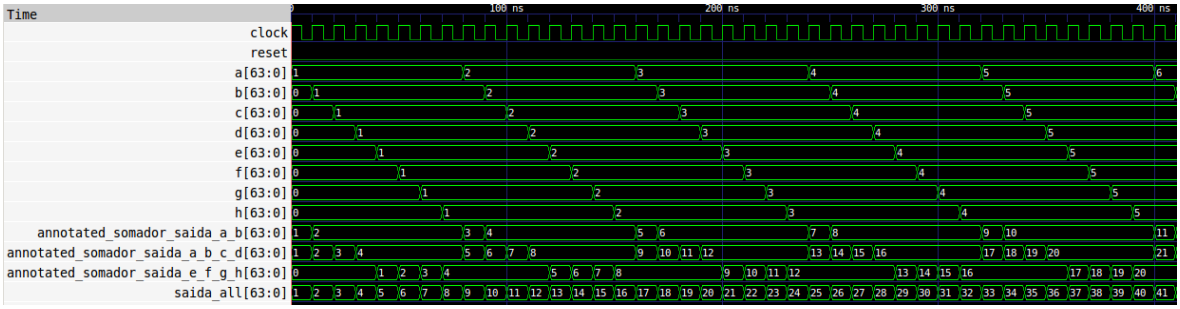


Figura 32 – Dados de saída visualizados no GTKWave.

Observando ainda a Figura 32, nota-se que a mesma apresenta os sinais de entrada do *DUT*, bem como os sinais observados. Conforme definição da ferramenta *Parser*, sinais observados são anotados, sendo precedidos por '*annotated_*' + hierarquia até chegar ao nome do sinal. Porém, observa-se que a saída primária *saída_all* também é observada, mas não foi anotada. Isto por que este sinal se encontra no topo da hierarquia do projeto.

Cenário de Teste (2): Sqrt

O segundo cenário de teste compreende o *DUT* Sqrt que faz a raiz quadrada de um valor de 64 bits, resultando em outro de 32 bits. Neste cenário as cadeias de *scan* utilizada possuem profundidade 2 para entrada e 8 para saída.

Do mesmo modo que no cenário de teste (1), foi realizado a simulação do *DUT* na ferramenta ModelSim, conforme apresenta a Figura 33. Esta simulação corresponde à aplicação direta dos estímulos no *DUT* e a correspondente visualização.

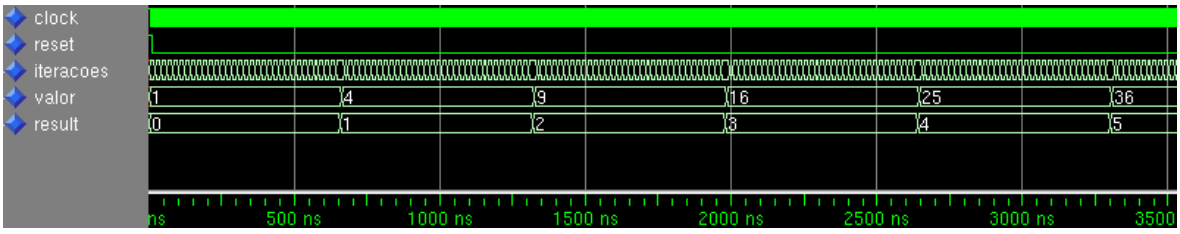


Figura 33 – Simulação do *DUT* (Sqrt) no ModelSim.

Seguindo o mesmo fluxo de emulação do cenário de teste (1), obtemos o arquivo *VCD*. A leitura dos sinais contidos neste arquivo é apresentada na Figura 34, através da ferramenta GTKWave.

Ambas as simulações apresentaram o mesmo resultado, demonstrando novamente a correta funcionalidade do ambiente de emulação.

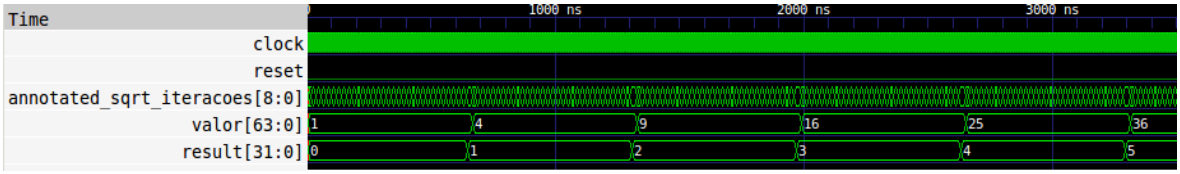


Figura 34 – Dados de saída visualizados no GTKWave.

Cenário de Teste (3): Framer

O terceiro e último cenário de teste, o Framer, compreende um circuito mais complexo comparado aos anteriores. Neste cenário as cadeias de *scan* utilizada possuem profundidade 2 para entrada e 8 para saída.

A Figura 35 apresenta a simulação do *DUT* na ferramenta ModelSim. Esta simulação corresponde à aplicação direta dos estímulos no *DUT* e a correspondente visualização.

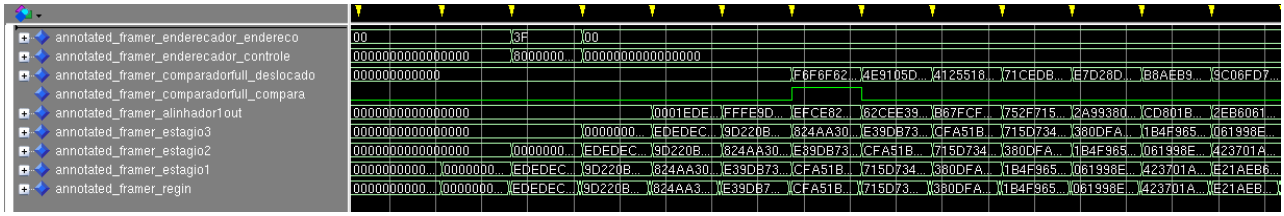


Figura 35 – Simulação do *DUT* (Framer) no ModelSim.

De posse do arquivo *VCD*, a interface GTKWave lê os sinais do arquivo e os exibe na simulação conforme figura Figura 36.

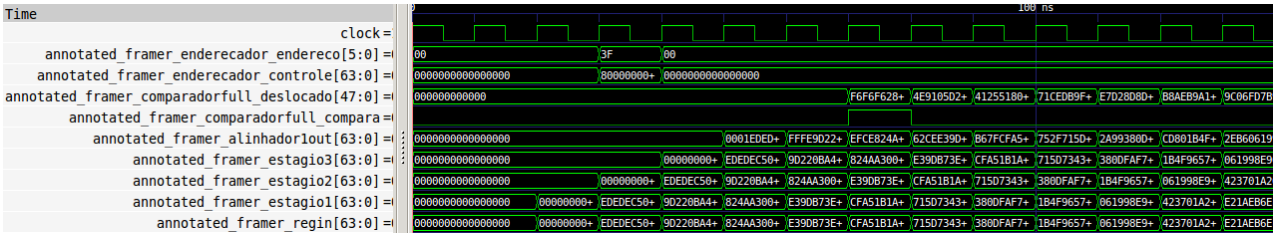


Figura 36 – Dados de saída visualizados no GTKWave.

Ambas as simulações apresentaram o mesmo resultado, demonstrando a correta funcionalidade do ambiente de emulação.

5.2 Validação em Hardware

Uma vez o ambiente de emulação funcional em simulação, foi desenvolvido cenários de teste para validar o funcionamento do ambiente em hardware, no *FPGA*. Os mesmos *DUTs* (Tabela 8) da validação em software foram utilizados para demonstrar a emulação em sua forma final.

Para fins de análise de custo, os *DUTs*, já inseridos no ambiente de emulação, foram sintetizados utilizando a ferramenta ISE 10.2, da empresa Xilinx. A síntese é realizada utilizando um *script* fornecido pela Dini Group, que garante as *constraints* necessárias para o *timing* da *PCIe*. Uma vez completo este processo, os *bitstreams* são utilizados para configurar ambos *FPGAs* via *PCIe*, utilizando o programa também fornecido pela Dini Group. Os resultados da síntese são apresentados na Tabela 10.

Tabela 10 – Resultados de síntese da validação em hardware, dispositivo xc5vlx330-2ff1760, ISE 11.4

DUT	Profundidade *Scans*		Original			Emulado			*Overhead*		
	Entrada	Saída	*LUTs*	Regs.	*BUFGs*	*LUTs*	Regs.	*BUFGs*	*LUTs*	Regs.	*BUFGs*
Somador	16	8	692	0	1	1003	12141	16	311	12141	15
Sqrt	2	8	311	237	1	9129	11671	16	8818	11434	15

O arquivo contendo os dados de entrada, gerados durante a validação por software, é informado para o módulo de comunicação desenvolvido (vide Seção 3.3). Inicia-se então o envio de dados pela *PCIe* e subseqüentemente, quando os primeiros dados válidos chegam nos buffers de entrada, o emulador inicia sua execução.

Os resultados obtidos foram analisados utilizando a ferramenta GTKWave para visualização de formas de onda, conforme a Figura 37 e Figura 38.

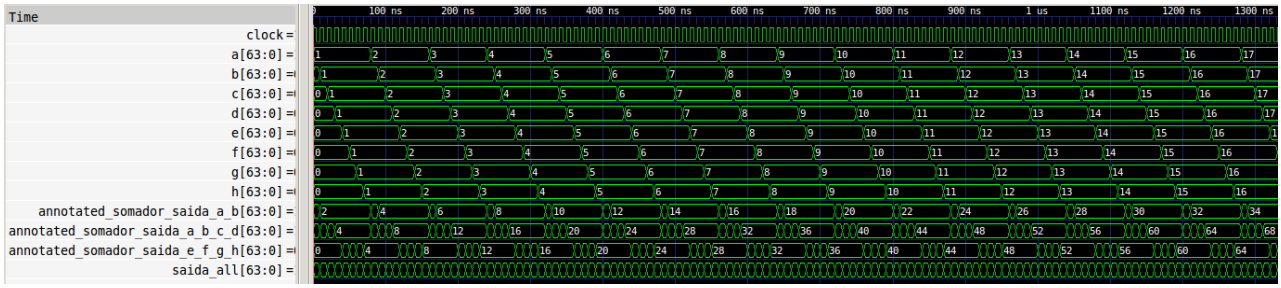


Figura 37 – Dados de saída do DUT (Somador) obtidos através da *PCIe*.

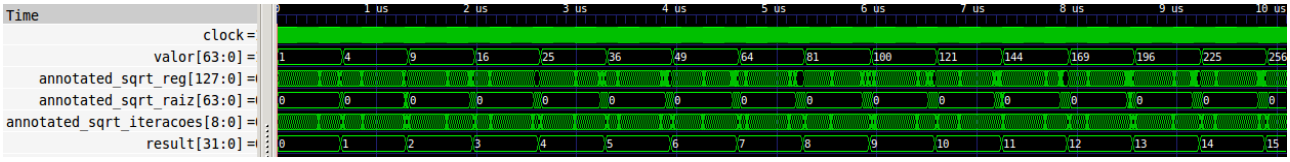


Figura 38 – Dados de saída do DUT (Sqrt) obtidos através da *PCIe*.

A título de comparação a Figura 37 e Figura 38 equivalem respectivamente a Figura 31 e Figura 33, que apresentam a simulação isolada dos respectivos *DUT*s em software. Ambas as emulações apresentaram o mesmo resultado comparados às suas simulações análogas, demonstrando a correta funcionalidade do ambiente de emulação.

6. CONCLUSÕES E TRABALHOS FUTUROS

O presente Trabalho de Conclusão apresentou o desenvolvimento de um ambiente de emulação para verificação de sistemas digitais síncronos. O estudo dos temas relacionados ao trabalho desenvolveu conhecimentos em várias áreas relacionadas à engenharia. Destas, as de maior influência sobre o projeto desenvolvido são as áreas de interfaces de alto desempenho, sistemas digitais e prototipação, tornando o presente trabalho multidisciplinar e de fundamental importância para a solidificação dos conhecimentos adquiridos durante o curso de Engenharia de Computação.

O desenvolvimento das cadeias de scan, do controlador de emulação e das ferramentas de integração que automatizam o processo de emulação foram finalizados com êxito, atingindo os objetivos almejados. A comunicação por PCIe não foi inteiramente finalizada devido ao tempo reduzido do TCC. Porém, este trabalho na presente forma, alcança o propósito inicial, prova de conceito do ambiente proposto. O presente projeto será continuado em conjunto com o Grupo de Apoio ao Projeto de Hardware (GAPH) de forma a concretizar uma plataforma robusta de emulação, capaz de comportar projetos reais.

Os trabalhos futuros compreendem em:

(i) Aperfeiçoamento do desempenho, comportamento e otimização do ambiente para atingir um resultado ótimo na emulação;

(ii) Criação de macros das cadeias de *scan* em um único *CLB* para aperfeiçoar o roteamento;

(iii) Integração dos módulos de controle com os módulos da *PCIe* automaticamente;

(iv) Integração com o Modelsim, para a exibição dos resultados da emulação, substituindo o GTKWave;

(v) Reavaliar o processo de análise léxica/sintática do projeto a ser testado;

(vi) Realizar testes com benchmarks reais, para avaliar o real ganho da emulação em relação à simulação.

(vii) Iniciar a emulação de circuitos com múltiplos domínios de relógio.

(viii) Aprimoramento da comunicação por *PCIe*, desenvolvendo módulos específicos para esta aplicação, em hardware e software.

(ix) Dimensionamento mais preciso e robusto dos buffers da comunicação por *PCIe*.

(x) Aperfeiçoamento da comunicação entre emulador e software hospedeiro.

(xi) Geração das formas de onda de saída dinamicamente, durante a emulação.

(xii) Geração de estímulos de entrada em tempo de execução a partir de *testbenches*.

(xiii) Execução do ambiente de emulação por tempo indefinido

REFERÊNCIAS BIBLIOGRÁFICAS

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

[CAD06] Cadence Design Systems Inc. “Incisive Enterprise Palladium”. Capturado em: http://www.cadence.com/rl/Resources/datasheets/incisive_enterprise_palladium.pdf, 2006.

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

[CHU07] Chuang, C-L; Cheng, W-H.; Liu, C-N.; Lu, D-J. “Hybrid Approach to Faster Functional Verification with Full Visibility”. IEEE Design & Test of Computers, vol. 24(2), 2007, pp. 154-162.

[DIN09a] Dini Group. “User Guide DN9200K10PCIe-8T”, Manual do Usuário, San Diego, Março 2009, 224p.

[DIN09b] Dini Group. “PCIE8T Dinigroup Board Family Full Design User Interface Manual”, Manual do Usuário, San Diego, Maio 2009, 19p.

[BYB10] Bybell, T. “GTKWave”. Capturado em: http://gtkwave.sourceforge.net, 2010.

[IBM05] IBM Corp. “Introduction to PCI Express” Capturado em: http://www.redbooks.ibm.com/abstracts/tips0456.html, 2005.

[LON02] London, G. “Hardware VHDL Parser”. Capturado em: http://search.cpan.org/~gslondon/Hardware-Vhdl-Parser-0.12/Parser.pm, 2002.

[MEN10] Mentor Graphics. “Emulation Systems”. Capturado em: http://www.mentor.com/products/fv/emulation-systems, 2010.

[NAI09] National Instruments. “PCI Express - An Overview of the PCI Express Standard”. Capturado em: http://zone.ni.com/devzone/cda/tut/p/id/3767, 2009.

[REH09] Rehák, Z. “Translation of VHDL programs into XML”. Capturado em: http://vhdlverif.sourceforge.net/vhd2xml.html. Masaryk University, Janeiro 2009.

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

[XIL10a] XILINX. “ChipScope Pro and the Serial I/O Toolkit”. Capturado em: http://www.xilinx.com/tools/cspro.htm, 2010.

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

ANEXO A – ESTRUTURA DOS CÓDIGOS HDL

A Figura 39 apresenta como os códigos desenvolvidos pelo fabricante da placa e os códigos desenvolvidos no contexto de TCC são integrados.

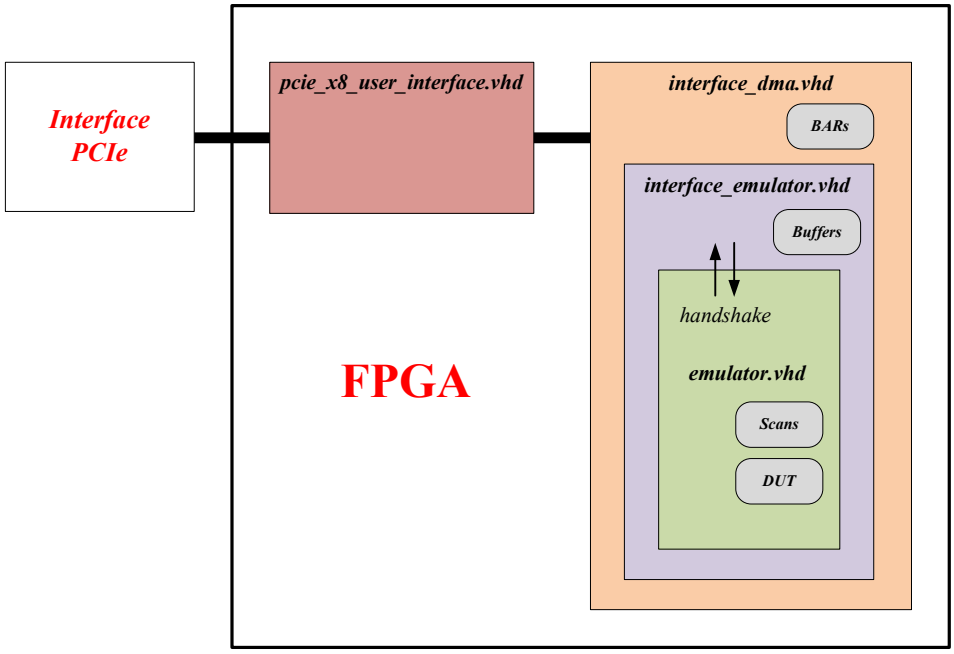


Figura 39 – Organização dos módulos utilizados no presente projeto.

A integração entre os dois grandes módulos, *pcie_x8_user_interface.vhd* e *interface_dma.vhd*, correspondem aos códigos fornecidos pelo fabricante. O módulo *interface_dma* instancia e controla os registradores *BAR*, e instancia o módulo *interface_emulator*, desenvolvido no escopo deste trabalho. O módulo *interface_emulator* instancia e controla os *buffers* de comunicação, e instancia o módulo *emulator*, o qual é gerado pela ferramenta de integração. Esse módulo *emulator* contém a controladora das cadeias de *scan*, as cadeias de *scan* e o *DUT*.

A interface de comunicação do módulo *emulator* é feita por dois barramentos de dados, com controle de fluxo baseado em *handshake*.

ANEXO B – VHDL CONVERTIDO PARA XML

```
<?xml version="1.0" ?>
<vhdl file="3des.vhd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.liberouter.org/formal_verification/tools/vhd2xml/vhdl.
xsd">
  <useClause library="ieee" line="26">
    <use id="ieee.std_logic_1164.all" line="27"/>
    <use id="ieee.std_logic_unsigned.all" line="28"/>
  </useClause>
  <entity id="tdes" line="30">
    <ports>
      <port id="rst_n" io="in" line="32">
        <type id="std_logic" line="32"/>
      </port>
      <port id="clk" io="in" line="33">
        <type id="std_logic" line="33"/>
      </port>
      <port id="enable_in" io="in" line="34">
        <type id="std_logic" line="34"/>
      </port>
      <port id="function_in" io="in" line="35">
        <type id="std_logic_vector" line="35">
          <range direction="downto">
            <constantExpression id="2" line="35"/>
            <constantExpression id="0" line="35"/>
          </range>
        </type>
      </port>
      <port id="data_in" io="in" line="36">
        <type id="std_logic_vector" line="36">
          <range direction="downto">
            <constantExpression id="63" line="36"/>
            <constantExpression id="0" line="36"/>
          </range>
        </type>
      </port>
      <port id="busy" io="out" line="37">
        <type id="std_logic" line="37"/>
      </port>
      <port id="data_ready" io="out" line="38">
        <type id="std_logic" line="38"/>
      </port>
      <port id="data_out" io="out" line="39">
        <type id="std_logic_vector" line="39">
          <range direction="downto">
            <constantExpression id="63" line="39"/>
            <constantExpression id="0" line="39"/>
          </range>
        </type>
      </port>
    </ports>
  </entity>
  <architecture entity="tdes" id="tdes" line="43">
    <declarations>
      <typeDeclaration id="state" line="45">
        <ids>
          <id id="s0" line="45"/>
          <id id="s1" line="45"/>
          <id id="s2" line="45"/>
          <id id="s3" line="45"/>
        </ids>
      </typeDeclaration>
      <signalDeclaration line="46">
        <ids>
          <id id="ea" line="46"/>
        </ids>
        <type id="state" line="46"/>
      </signalDeclaration>
      <signalDeclaration line="48">
        <ids>
          <id id="key1" line="48"/>
          <id id="key2" line="48"/>
          <id id="key3" line="48"/>
        </ids>
        <type id="std_logic_vector" line="48">
          <range direction="downto">
            <constantExpression id="63" line="48"/>
            <constantExpression id="0" line="48"/>
```

ANEXO C – VHDL ANOTADO

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity TDES is
port(
  annotated_tdes_des_initial_permutation_data_out : out std_logic_vector( 63 downto 0 ); -- Port observed by emul
  annotated_tdes_des_f_func                       : out std_logic;                       -- Port observed by emul
  annotated_tdes_des_f_C_out                      : out std_logic_vector( 27 downto 0 ); -- Port observed by emul
  annotated_tdes_des_f_second_xor                 : out std_logic_vector( 31 downto 0 ); -- Port observed by emul
  annotated_tdes_des_f_permutation_p_data_out     : out std_logic_vector( 31 downto 0 ); -- Port observed by emul
  rst_n                                           : in  std_logic;
  clk                                             : in  std_logic;
  enable_in                                       : in  std_logic;
  function_in                                     : in  std_logic_vector (2 downto 0);
  data_in                                         : in  std_logic_vector (63 downto 0);
  busy                                            : out std_logic;
  data_ready                                      : out std_logic;
  data_out                                        : out std_logic_vector (63 downto 0)
  );
end TDES;

architecture TDES of TDES is

type state is (S0, S1, S2, S3); --controle
signal EA : state;

signal key1, key2, key3                  : std_logic_vector (63 downto 0); --buffers p/ chaves
signal des_busy, des_ready, des_start    : std_logic;                      --fios p/ o des
signal des_data_in, des_data_out, des_key : std_logic_vector (63 downto 0); --fios p/ o des
signal des_func                          : std_logic;                      --funcao p/ o des

begin
  des: entity work.des --instacia 1 des
    port map(
      annotated_tdes_des_initial_permutation_data_out => annotated_tdes_des_initial_permutation_data_out, --by emul
      annotated_tdes_des_f_func                       => annotated_tdes_des_f_func,                       --by emul
      annotated_tdes_des_f_C_out                      => annotated_tdes_des_f_C_out,                      --by emul
      annotated_tdes_des_f_second_xor                 => annotated_tdes_des_f_second_xor,                 --by emul
      annotated_tdes_des_f_permutation_p_data_out     => annotated_tdes_des_f_permutation_p_data_out,     --by emul
      rst_n                                                => rst_n,
      clk                                                  => clk,
      enable_in                                       => enable_in,
      start                                                => des_start,
      function_in                                     => des_func,
      data_in                                              => des_data_in,
      key_in                                               => des_key,
      busy                                                 => des_busy,
      data_ready                                      => des_ready,
      data_out                                             => des_data_out
  );
--inicio do 3des (controle de 1 des)
  process(rst_n, clk)
  begin
    if rst_n='0' then --hard reset
      EA         <= S0;
      des_key    <= (others=>'0');
      key1       <= (others=>'0');
      key2       <= (others=>'0');
      key3       <= (others=>'0');
      des_func   <= '0';
      des_start  <= '0';
      des_data_in <= (others=>'0');
      busy       <= '0';
      data_ready  <= '0';
      data_out   <= (others=>'0');
    elsif clk'event and clk='1' then
      if enable_in = '1' then                    --recebeu sinal p/ iniciar
        case EA is
          when S0 =>                                                  --idle
            data_ready  <= '0';
            des_data_in <= data_in;                   --prepara barramento do des
            des_func    <= not function_in(0); --define funcao crypt "xx1" decrypt "xx0"
            case function_in is
              when "000" =>                          --idle
                EA <= S0;
              when "001" =>                          --recebe chave 1
                key1 <= data_in;
                EA <= S0;
              when "010" =>                          --recebe chave 2
                key2 <= data_in;
                EA <= S0;
              when "011" =>                          --recebe chave 3
                key3 <= data_in;
                EA <= S0;
              when others =>                         --crypt/decrypt
```

ANEXO D – ARQUIVO VCD

```
$comment
Generated through VCD Generator
$end

$timescale
1ns
$end

$scope module DUT $end

$scope module DUT_0 $end
$var wire 1 !0 clock $end
$var wire 1 !1 reset $end
$var wire 64 !2 a $end
$var wire 64 !3 b $end
$var wire 64 !4 c $end
$var wire 64 !5 d $end
$var wire 64 !6 e $end
$var wire 64 !7 f $end
$var wire 64 !8 g $end
$var wire 64 !9 h $end
$var wire 64 !10 saida_e_f $end
$var wire 64 !11 saida_c_d $end
$var wire 64 !12 saida_a_b $end
$var wire 64 !13 saida_all $end
$upscope $end

$upscope $end

$enddefinitions $end
#0
b0 !1
b1 !0
#50
b0 !0
#100
b1 !0
b0000000000100000000000000000000000000000000000100000000000000000000000 !2
b0000000000100000000000000000000000000000000000100000000000000000000000 !3
b0000000000100000000000000000110000000000000001000000000000000000000000 !4
b0000000000100000000000000000000000000000000000100000000000000000000000 !5
b0000000000100000000000000000110000000000000001000000000000000000000000 !6
b0000000000100000000000000000001000000000001000000000000000000000000000 !7
b0000000000100000000100000000000000000000000100000000000000000000000000 !8
b0000000000100000000000000000000000000100000100000000000000000000000000 !9
b0000000001000000000000000001101000000000010000000000000000000000000000 !10
b0000000001000000000000000000110000000000010000000000000000000000000000 !11
b0000000001000000000000000000000000000000000100000000000000000000000000 !12
b0000001000000000000100000110010000010010000000000000000000000000000000 !13
#150
b0 !0
#200
b1 !0
b0000000000100000000000000000000000000000000000100000000000000000000000 !2
b0000000000100000000000000000000000000000000000100000000000000000000000 !3
b0000000000100000000000000000110000000000000001000000000000000000000000 !4
b0000000000100000000000000000000000000000000000100000000000000000000000 !5
b0000000000100000000000000000110000000000000001000000000000000000000000 !6
b0000000000100000000000000000001000000000001000000000000000000000000000 !7
b0000000000100000000100000000000000000000000100000000000000000000000000 !8
b0000000000100000000000000000000000000100000100000000000000000000000000 !9
b0000000001000000000000000001101000000000010000000000000000000000000000 !10
b0000000001000000000000000000110000000000010000000000000000000000000000 !11
b0000000001000000000000000000000000000000000100000000000000000000000000 !12
b0000001000000000000100000110010000010010000000000000000000000000000000 !13
#250
b0 !0
#300
b1 !0
b0000000000100000000000000000000000000000000000100000000000000000000000 !2
b0000000000100000000000000000000000000000000000100000000000000000000000 !3
b0000000000100000000000000000110000000000000001000000000000000000000000 !4
b0000000000100000000000000000000000000000000000100000000000000000000000 !5
b0000000000100000000000000000110000000000000001000000000000000000000000 !6
b0000000000100000000000000000001000000000001000000000000000000000000000 !7
b0000000000100000000100000000000000000000000100000000000000000000000000 !8
b0000000000100000000000000000000000000100000100000000000000000000000000 !9
b0000000001000000000000000001101000000000010000000000000000000000000000 !10
b0000000001000000000000000000110000000000010000000000000000000000000000 !11
b0000000001000000000000000000000000000000000100000000000000000000000000 !12
b0000001000000000000100000110010000010010000000000000000000000000000000 !13
#350
b0 !0
#400
b1 !0
b0000000000100000000000000000000000000000000000100000000000000000000000 !2
b0000000000100000000000000000000000000000000000100000000000000000000000 !3
b0000000000100000000000000000110000000000000001000000000000000000000000 !4
b0000000000100000000000000000000000000000000000100000000000000000000000 !5
b0000000000100000000000000000110000000000000001000000000000000000000000 !6
b0000000000100000000000000000001000000000001000000000000000000000000000 !7
b0000000000100000000100000000000000000000000100000000000000000000000000 !8
b0000000000100000000000000000000000000100000100000000000000000000000000 !9
b0000000001000000000000000001101000000000010000000000000000000000000000 !10
b0000000001000000000000000000110000000000010000000000000000000000000000 !11
b0000000001000000000000000000000000000000000100000000000000000000000000 !12
b0000001000000000000100000110010000010010000000000000000000000000000000 !13
#450
b0 !0
#500
b1 !0
b0000000000100000000000000000000000000000000000100000000000000000000000 !2
b0000000000100000000000000000000000000000000000100000000000000000000000 !3
```