UNIVERSIDADE DE SÃO PAULO
Escola de Engenharia de São Carlos

Analisador Lógico para Análise *On-Chip* de Sistemas Digitais Implementados em FPGA

Gabriel Santos da Silva

São Carlos - SP

Analisador Lógico para Análise *On-Chip* de Sistemas Digitais Implementados em FPGA

Gabriel Santos da Silva

Orientador: Maximiliam Luppe

Monografia de conclusão de curso apresentada a Escola de Engenharia de São Carlos - EESC-USP - para obtenção do título de Engenheiro Eletricista com Ênfase em Eletrônica.

USP – São Carlos
Junho de 2011

Ficha catalográfica preparada pela Seção de Tratamento
da Informação do Serviço de Biblioteca – EESC/USP

Silva, Gabriel Santos da
S586a Analisador lógico para análise *On-Chip* de sistemas digitais implementados em FPGA / Gabriel Santos da Silva ; orientador Maximiliam Luppe -- São Carlos, 2011.

Monografia (Graduação em Engenharia Elétrica com ênfase em Eletrônica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2011.

1. Dispositivos reconfiguráveis. 2. FPGA. 3. Analisador lógico. 4. Ferramentas de desenvolvimento. I. Titulo.

FOLHA DE APROVAÇÃO

Nome: Gabriel Santos da Silva

Título: "Analisador Lógico para Análise On-Chip de Sistemas Digitais Implementados em FPGA"

Trabalho de Conclusão de Curso defendido e aprovado em 10 / 06 / 2011 ,

com NOTA 9,0 (Nove , Zero), pela comissão julgadora:

Prof. Associado Evandro Luís Linhari Rodrigues - EESC/USP

Prof. Dr. Marcelo Andrade da Costa Vieira - EESC/USP

Prof. Associado Homero Schiabel
Coordenador da CoC-Engenharia Elétrica
EESC/USP

Dedicatória

Dedico este trabalho aos meus pais, Gilson e Valéria, pessoas essenciais em minha vida, responsáveis pela pessoa que sou hoje e por chegar aonde cheguei. Obrigado por tantas alegrias, por tantos ensinamentos, por serem meus amigos e por sempre me permitirem arriscar em novas jornadas na vida.

Aos meus amigos Ulisses e João, companheiros em todos os momentos de faculdade. Devo muito a vocês, meus irmãos. Obrigado pela amizade indispensável nesses anos.

E em especial à minha avó Tina, tão presente em meus pensamentos na conclusão de mais esta fase da minha vida.

Agradecimentos

Agradeço a Deus por todas as graças em minha vida.

À minha família, por todos os momentos vividos até hoje, pela companhia e apoio incontestáveis.

Ao Prof. Maximiliam Luppe, meu orientador, pela paciência nesses três anos de parceria, desde a Iniciação Científica até este Projeto de Graduação; pelos ensinamentos e pela dedicação.

Aos meus amigos de Pindamonhangaba, Bárbara, Bruna, Camila, Carol, Kenzo, Natália, Nícolas, Pedro e Rodrigo, pelos ótimos finais de semana tão esperados durante a faculdade, e pela amizade que se fortalece a cada dia, mesmo com a distância.

A todos os amigos conquistados durante a graduação. Em especial, a Átila e Bruno, grandes amigos de república e kitnete, pelo companheirismo e conversas "à toa" até altas horas; e a Fernando, pela parceria nas tentativas de estudo e consideração nos momentos difíceis. Espero que todas estas amizades possam durar independentemente dos caminhos escolhidos por cada um.

Resumo

Um problema existente na análise de sinais digitais em sistemas embarcados implementados em FPGA é a visualização dos sinais entre os diversos módulos (*On-Chip*). Mesmo os fabricantes de FPGA possuindo aplicativos para este tipo de análise, os mesmos são restritos apenas aos dispositivos dos seus próprios fabricantes, sendo ideal um módulo interno que fosse *open-source*. Baseando-se nestes pontos decidiu-se pelo desenvolvimento de um analisador lógico *On-Chip open-source*. Desta forma, este projeto desenvolve um *soft-core* referente ao analisador lógico, envolvendo o estudo de linguagens de *hardware*, além do estudo qualitativo das ferramentas de desenvolvimento dos principais fabricantes deste dispositivo no mercado atualmente. Os resultados obtidos permitem comprovar a eficiência da lógica adotada na implementação dos módulos; a capacidade da ferramenta de desenvolvimento Lattice Diamond em comparação com as demais ferramentas adotadas pelo mercado; e a dificuldade em superar as características intrínsecas das FPGAs de diferentes tecnologias, exemplificadas neste caso pelo uso dos elementos de memória embarcadas das FPGA's, para a implementação do módulo analisador lógico.

Palavras-chave: dispositivos reconfiguráveis, FPGA, analisador lógico, ferramentas de desenvolvimento.

Abstract

One issue in the analysis of digital signals in embedded systems implemented in FPGA is the visualization of signals between the various modules (On-Chip). Even the manufacturers of FPGA possessing applications for this type of analysis, they are restricted only to their own devices manufacturers, making it ideal an internal module that was open source. Based on these points decided by the development of an On-Chip Logic Analyzer open-source. Thus, this project develops a soft-core that refers to the logic analyzer, involving the study of hardware description languages, beyond the qualitative study of the development tools from leading manufacturers of this device on the market today. The results prove the efficiency of the logic adopted in the implementation of the modules, the ability of the Diamond Lattice development tool in comparison with other tools used by the market, and the difficulty in overcoming the inherent characteristics of the FPGA's from different technologies, exemplified in this case by the use of embedded memory elements of FPGA's, to implement the logic analyzer module.

Keywords: reconfigurable devices, FPGA, logic analyzer, development tools.

Sumário

Lista de Abreviaturas

ASIC: Circuito Integrado de Aplicação Específica (do inglês *Application-specific integrated circuit*)

CLB: Bloco Lógico Configurável (do inglês *Configurable Logic Block*)

CPLD: Dispositivo Lógico Programável Complexo (do inglês)

DCM: Gerenciador de Temporizador Digital (do inglês *Digital Clock Manager*)

DSP: Processamento de Sinais Digitais (do inglês *Digital Signal Processing*)

DUV: Entidade Sob Teste(do inglês *Design Under Verification*)

EBR: Bloco de RAM dedicada (do inglês *Embedded Block RAM*)

EDIF: Formato de Transição de Design Eletrônico (do inglês *Electronic Design Interchange Format*)

EEPROM: EPROM Elétrica (do inglês *Electrical EPROM*)

EPROM: Memória Somente de Leitura Apagável e Programável (do inglês *Erasable Programmable Read Only Memory*)

FIFO: Primeiro Dentro, Primeiro Fora (do inglês *First In, First Out*)

FPGA: Arranjo de Portas Programável em Campo (do inglês *Field-Programmable Gate Array*)

HCPLD: Dispositivo Lógico Programável de Alta Capacidade (do inglês *High Capacity Programmable Logic Device*)

HDL: Linguagem de Descrição de Hardware (do inglês *Hardware Description Language*)

IDE: Ambiente de Desenvolvimento Integrado (do inglês *Integrated Development Environment*)

IEEE: Insituto de Engenheiros Elétricistas e Eletrônicos (do inglês *Institute of Electrical and Electronics Engineers*)

IOB: Bloco de Entrada e Saída (do inglês In and Out Block)

IP-Core: Núcleo de propriedade intelectual (do inglês *Intellectual Property Core*)

ISP: Programação Em Sistema (do inglês *In System Programmability*)

LAB: Bloco de Arranjo Lógico (do inglês *Logic Array Block*)

LE: Elemento Lógico (do inglês *Logical Element*)

LUT: Tabela de Pesquisa (do inglês *Look-Up Table*)

MOS: Semicondutor de Óxido de Metal (do inglês *Metal Oxide Semiconductor*)

PAL: Matriz Lógica Programável (do inglês *Programmable Array Logic*)

PFF: Unidade Funcional Programável sem RAM (do inglês *Programmable Functional Unit without RAM*)

PFU: Unidade Funcional Programável (do inglês *Programmable Functional Unit*)

PIC: Célula Programável de Entrada e Saída (do inglês *Programmable I/O Cell*)

PLA: Arranjo Lógico Programável (do inglês *Programmable Logic Array*)

PLD: Dispositivo Lógico Programável (do inglês *Programmable Logic Device*)

PROM: Memória Programável Somente de Leitura (do inglês *Programmable Read-Only Memory*)

RAM: Memória de Acesso Aleatório (do inglês *Random Access Memory*)

RTL: Nível de Transferência entre Registradores (do inglês *Register Transfer Level*)

SoC: Sistema Operacional em Chip (do inglês *System on Chip*)

SPLD: Dispositivo Lógico Programável Simples (do inglês *Simple Programmable Logic Device*)

SRAM: RAM Estática (do inglês *Static RAM*)

ULA: Unidade Lógica Aritmética

USB: Barramento Serial Universal (do inglês *Universal Serial Bus*)

VHDL: HDL para Circuitos Integrados de Alta Velocidade (do inglês *Very High Speed Integrated Circuit HDL*)

Lista de Tabelas

Lista de Figuras

CAPÍTULO 1: INTRODUÇÃO

1.1. Contextualização e Motivação

Nos últimos anos o crescimento, tanto em diversidade, quanto em densidade, dos dispositivos reconfiguráveis e de suas ferramentas de desenvolvimento, tem favorecido a implementação de sistemas complexos em lógica integrada e programável, *System on Chip* (SoC), em um curto espaço de tempo. Altera, Lattice e Xilinx são exemplos de empresas que desenvolvem soluções na área de sistemas reconfiguráveis digitais, cada uma delas possuindo suas respectivas ferramentas de desenvolvimento: Quartus II, Diamond e ISE.

Trabalhar com *Field-programmable Gate Array* (FPGA) pode remeter ao estudo de *soft-cores* de microcontroladores e de periféricos para sistemas embarcados. Estes núcleos apresentam como principais vantagens o fato de poderem ser reutilizados (um mesmo *soft-core* pode ser utilizado em diversos projetos, sem custo adicional e sem gasto com tempo de projeto) e serem portáteis (podem ser adequados a diversas plataformas de desenvolvimento de dispositivos reconfiguráveis). O uso das ferramentas de desenvolvimento de dispositivos reconfiguráveis, aliadas ao uso de linguagens de descrição de *hardware* possibilita o desenvolvimento de novos *soft-cores* para a área de instrumentação.

Um problema existente na análise de sinais digitais em sistemas embarcados implementados em FPGA é a visualização dos sinais entre os diversos módulos (*On-Chip*). Mesmo os fabricantes de FPGA possuindo aplicativos para este tipo de análise, os mesmos são restritos apenas aos dispositivos dos seus próprios fabricantes, sendo ideal um módulo interno que fosse *open-source*. Baseando-se nestes pontos decidiu-se pelo desenvolvimento de um analisador lógico *On-Chip open-source*.

O analisador lógico é um instrumento de medida específico para análise e comparação de sinais digitais, oferece um grande número de canais de entrada, permitindo trabalhar com circuitos mais complexos. O inconveniente desse equipamento está em seu alto preço, justificado por apresentar mais recursos do que o necessário em pequenas aplicações.

Para garantir que o analisador lógico implementado seja *open-source* há a necessidade de verificar seu funcionamento em diferentes ferramentas de desenvolvimento. A escolha das mesmas tem como base o ambiente atual do mercado de dispositivos reconfiguráveis, optando pelas empresas que mais se destacam no ramo. Como mencionado, existe também a necessidade do estudo das linguagens de descrição de *hardware* (HDL), optando-se pela linguagem Verilog, devido à maior facilidade de aprendizagem em relação à "*Very High Speed Integrated Circuits*" HDL (VHDL), visto que esta opção se assemelha a linguagem C, amplamente trabalhada na área de eletrônica digital.

Este projeto foi elaborado a fim de solucionar os problemas citados neste item, criando o Analisador lógico para análise *On-Chip* de sistemas digitais implementados em FPGA, sendo uma solução barata e não condicionada ao seu fabricante.

1.2. Objetivos

Este trabalho dá continuidade ao trabalho de iniciação científica, tendo como objetivos principais o estudo das ferramentas de desenvolvimento de Dispositivos Reconfiguráveis existentes no mercado; a implementação de *soft-cores* de um analisador lógico para análise de sistemas digitais intra-FPGA integrado ao estudo de linguagens de descrição de *hardware* (Verilog); e a análise do desempenho dos *soft-cores* implementados, avaliando a relação de desempenho por célula lógica reconfigurável.

1.3. Organização do Trabalho

A fim de uma melhor compreensão das atividades realizadas neste projeto, esta monografia é organizada da seguinte maneira:

- No Capítulo 2 é apresentada uma revisão bibliográfica sobre os assuntos pertinentes a este projeto: tecnologia dos dispositivos reconfiguráveis, em especial FPGA, e no desenvolvimento de projetos envolvendo a mesma;
- No Capítulo 3 são descritas a metodologia adotada para a implementação dos módulos necessários para a execução do projeto;

- No Capítulo 4 são apresentados os resultados obtidos e uma análise dos mesmos. Também são expostas as principais dificuldades e limitações encontradas durante a realização deste projeto;
- O Capítulo 5 contém uma conclusão acerca do projeto de Graduação, além de possíveis trabalhos futuros que possam tomar como base este projeto e contribuições que o mesmo fornece ao autor.

CAPÍTULO 2: REVISÃO BIBLIOGRÁFICA

2.1. Considerações Iniciais

Este capítulo tem como objetivo apresentar os conteúdos técnicos e conceituais utilizados durante a elaboração do projeto em questão; entre eles encontram-se a descrição de um analisador lógico, do tipo de memória utilizada pelo mesmo, além de abordar a tecnologia de FPGA, suas arquiteturas e plataformas de desenvolvimento.

2.2. System on Chip e IP Core

O avanço da tecnologia digital em relação ao nível de integração de componentes em chip propiciou o desenvolvimento de sistemas complexos em uma única pastilha de silício, que podem incluir processadores, módulos de memória e controladores de entrada e saída; sistemas estes denominados *Systems on Chip* (SOCs). Por sua vez, os projetos de SOCs estão se tornando cada vez mais complexos e a necessidade de integração e comunicação entre diversos sistemas embarcados - sistemas com capacidade computacional dentro de um circuito integrado - está se tornando característica chave dos sistemas modernos.

Para conseguir atender as exigências de mercado, os projetistas de SOCs devem buscar novos métodos de projeto em nível de sistema. Estes métodos deverão possibilitar o desenvolvimento de *hardware* e *software* de forma eficaz, e também o reuso de blocos previamente projetados e verificados que executam tarefas específicas, os quais recebem o nome de *Intellectual Property Cores* (*IP-Cores*) [Moraes, 2004]. Podem-se destacar como principais vantagens associadas ao uso de SOCs e destes núcleos; baixo custo de fabricação em série; alta qualidade; baixa potência consumida; pequeno tamanho e alta velocidade. Os *cores,* segundo [Gupta, 1997], seguem a seguinte classificação:

Soft Core – Consiste de uma descrição em HDL, código fonte, que pode ser mapeada para diferentes processos de fabricação, independente da tecnologia;

Firm Core – Núcleo que contem mais estruturas, normalmente um *netlist* dependente da tecnologia, pronto para etapas mais avançadas do processo de desenvolvimento de projetos;

Hard Core - Inclui *layout* e informações referentes à temporização do circuito para uma determinada tecnologia, uma organização pré-definida que não pode ser modificado pelo projetista.

2.3. Dispositivos Reconfiguráveis

Os componentes de lógica programável são dispositivos que possuem em sua lógica interna centenas ou milhares de portas lógicas, *flip-flops* e registradores e são chamados de dispositivos lógicos programáveis, *Programmable Logic Devices* (PLD). Evoluem de acordo com a necessidade de implementação de funções mais complexas, sendo divididos, de acordo com [Brown, 1996] em duas classes: *Simple Programmable Logic Devices* (SPLDs) e *High Capacity Programmable Logic Devices* (HCPLDs).

SPLD, ou arranjo lógico programável, consiste de um circuito que possui uma estrutura interna baseada em um conjunto de portas AND-OR. Estes arranjos só podem ser programados uma vez, ou seja, definida sua função lógica ela não poderá ser mudada. Seus principais representantes são: *Programmable Read-Only Memory* (PROM), dispositivos onde o arranjo AND é pré-definido em fábrica e somente o arranjo OR é programável; *Programmable Array Logic* (PAL), dispositivos opostos a PROM, onde as portas AND são programáveis enquanto as portas OR são pré-conectadas em fábrica; e *Programmable Logic Arrays* (PLA), dispositivo que possui tanto a matriz de portas AND quanto a matriz de portas OR programáveis.

HCPLDs, ou arranjos de portas programáveis, são estruturas mais genéricas e versáteis que as baseadas na estrutura tradicional AND-OR. A principal vantagem deste tipo de circuito em comparação com os arranjos lógicos programáveis é a possibilidade de reprogramação do comportamento de um circuito quantas vezes forem necessárias. São representados por *Complex* PLD (CPLD), dispositivos que utilizam em sua estrutura vários PLD´s interligados através de conexões programáveis; e FPGA, dispositivo que possui uma arquitetura baseada em blocos lógicos configuráveis - *Configuration Logical Blocks* (CLB), blocos de entrada e saída - *In/Out Blocks* (IOB), e chaves de interconexão programáveis.

2.4. FPGA

Os dispositivos mais utilizados atualmente para computação reconfigurável são as FPGA's, projetadas inicialmente para prototipação de circuitos, apesar do seu custo e acesso. Apresenta como principal vantagem a possibilidade de modificação da estrutura de *hardware* de um sistema através de um processo denominado reconfiguração, o qual permite o desenvolvimento incremental, correção de erros de projeto, além da adição de novas funções de *hardware* [Moraes, 2004].

A FPGA se destaca na implementação de circuitos digitais, oferecendo uma boa relação custo/benefício, suportando a implementação de circuitos lógicos relativamente grandes. Apresenta maior flexibilidade que microprocessadores de uso geral e menor custo que os circuitos integrados de aplicação específica (ASIC) [Eskinazi, 2005]. Normalmente as aplicações implementadas em FPGA são mais lentas e consomem mais energias que as implementas em ASIC, mas as deste caso só se justificam pela produção em larga escala.

Focada em obter o resultado lógico final desejado em cada projeto implementado, a FPGA é composta por uma matriz de CLB's cercados por uma rede de interconexão programável e os IOB's [Martins 2003]. Estes IOB's são programáveis e servem como interface entre o mundo exterior e a lógica interna do dispositivo. São constituídos por *buffers* bidirecionais com saída em alta-impedância, logo, através de uma programação adequada pode-se configurar um pino da FPGA para funcionar como entrada, saída, bidirecional ou coletor-aberto. Os CLB's e a rede de interconexão serão descritos a seguir.

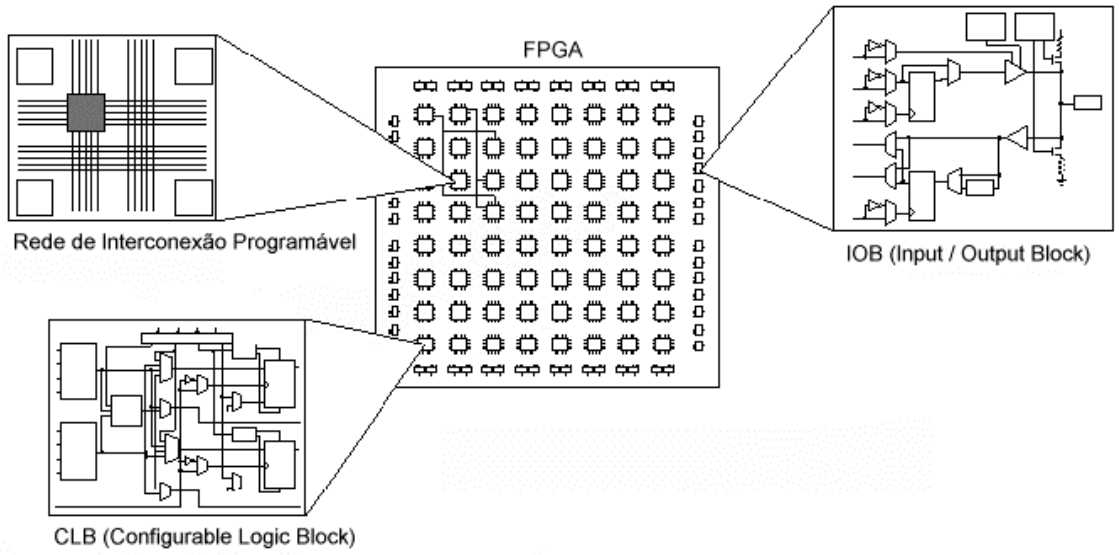

Figura 1 - Arquitetura de uma FPGA

2.4.1. Blocos Lógicos Configuráveis

Os CLB's formam um arranjo bi-dimensional de linhas e colunas. A arquitetura destes blocos lógicos varia de família para família e de fabricante para fabricante, mas basicamente são compostos de pontos de entrada e registradores. Os pontos de entrada se conectam a blocos que implementam funções puramente combinacionais como *Lookup Table* (LUT), multiplexadores que direcionam o fluxo dos sinais internos do CLB. Os registradores, tipicamente *flip-flops*, estão ligados às saídas e também podem realimentar as entradas dos geradores de funções combinacionais.

Todos os elementos lógicos que compõem uma FPGA são configuráveis e propiciam grande flexibilidade para a implementação de funções. Esta diferença em termos de simplicidade/complexidade destes elementos é chamada granularidade do dispositivo reconfigurável

Granularidade Fina: menor quantidade de lógica ou poder computacional nas unidades de reconfiguração [Hartenstein, 2001]. Oferece maior flexibilidade de implementação de algoritmos em *hardware*, melhor desempenho devido a uma maior aproximação entre a arquitetura pós-configuração e o algoritmo implementado, mas também um alto custo de roteamento e de energia durante a reconfiguração, além de maiores atrasos da propagação dos sinais. Usa portas lógicas como blocos básicos de

construção, além de blocos funcionais compostos por tabelas, *flip flops* e mux [Moraes, 2001].

Granularidade Grossa: maior quantidade lógica ou poder computacional nas unidades reconfiguraveis, possuindo blocos de construções maiores, Unidades Lógicas Aritméticas (ULAs), multiplicadores, deslocadores, etc. São indicados para aplicações que envolvem computações mais complexas como manipulações de imagens e outras típicas aplicações de caminho de dados, com manipulação de dados com largura (codificação) de vários bits.

2.4.2. Blocos de Interconexão

Os blocos de interconexão possuem chaves programáveis, comutadores que permitem conectar os blocos lógicos de maneira conveniente, em função da necessidade de cada projeto [Pontes, 2006]. A rede de interconexão programável é composta por diferentes tipos de segmentos de conexão, capazes de interligar a maioria das entradas e saídas dos CLB's entre si e aos IOB's. Isso tudo permite que circuitos complexos, máquinas de estado e algoritmos sejam implementados nos FPGA's [Martins, 2003]. As propriedades destes comutadores, tamanho, resistência em condução e capacitância parasita, definem a eficiência e o desempenho do dispositivo. As tecnologias mais usadas na implementação dos mesmos são as tecnologias de programação baseada em memória estática - *Static Random Access Memory* (SRAM), transistores e porta flutuante (*floating gate*).

SRAM - nessa tecnologia, a chave de roteamento ou comutador é um transistor de passagem ou um multiplexador controlado por uma memória estática de acesso aleatório SRAM. Ocupa muito espaço no circuito integrado e exige *hardware* externo auxiliar que deve ser montado junto com os blocos lógicos, entretanto tem como vantagem a possibilidade de ser rapidamente configurada.

Anti-fusível - essa tecnologia baseia-se num dispositivo de dois terminais que no estado não programado apresenta uma alta impedância (circuito aberto, modo de corte). Aplicando-se uma tensão, o dispositivo forma um caminho de baixa impedância entre seus terminais (circuito fechado, modo de condução). Essa é uma opção mais barata que a opção de RAM estática.

Porta flutuante - a tecnologia de porta flutuante (*floating gate*) baseia-se em transistores *Metal Oxide Semiconductor* (MOS), especialmente construído com dois *gates* flutuantes semelhantes aos usados nas memórias *Erasable Programmable Read Only Memory* (EPROM) e *Electrical EPROM* (EEPROM). A maior vantagem dessa tecnologia é a sua capacidade de programação e retenção de dados. Da mesma forma que em uma memória EEPROM, os dados podem ser programados com o circuito integrado instalado na placa, característica denominada *In System Programmability* (ISP).

2.5. Desenvolvimento em FPGA

Uma característica importante das FPGA's é sua capacidade de ser programada em campo pois sua funcionalidade não é definida na fundição do chip e sim pelo projetista da aplicação final, que usa métodos de reconfiguração

O processo de criação de uma lógica digital não é muito diferente do desenvolvimento de sistemas embarcados. Uma estrutura de descrição de *hardware* é escrita em uma linguagem de alto nível (usualmente VHDL e Verilog), o código é compilado e copiado para ser executado. Normalmente utilizam-se estas linguagens para realizar um modelo *Register Transfer Level* (RTL) do projeto, ou seja, uma descrição do projeto através do fluxo (transferência) de dados entre os seus registradores, controlado por um sinal de *clock* [Souza, 2008].

Descrever circuito como um esquemático digital é também possível. O esquemático é uma representação visual de portas e componentes lógicos combinacionais e seqüenciais do *hardware* que fazem parte da solução implementada no sistema reconfigurável, obrigando o desenvolvedor a ter conhecimento de componentes lógicos e desenvolvimento de *hardware*. A parte da solução implementada em *hardware* reconfigurável é desenvolvida utilizando ferramentas que a partir da captura do esquemático geram os bits de configuração de um determinado dispositivo reconfigurável. Essa abordagem geralmente não é aplicada a projetos grandes por causa da dificuldade que existe em se fazer uma representação gráfica de muitos componentes.

A principal diferença entre o design de *hardware* e *software* é a maneira que o desenvolvedor precisa pensar para resolver um problema. Desenvolvedores de *software* tendem a pensar seqüencialmente, mesmo quando estão desenvolvendo aplicações

multitarefa. As linhas de código são escritas para serem executadas em uma ordem, pelo menos dentro de uma tarefa em particular. Durante o projeto de *hardware* os designers precisam programar em paralelo, todos os sinais são processados desta maneira, pois trafegam através de um caminho de execução próprio até o destino do sinal de saída. Sendo assim, a descrição do *hardware* cria estruturas que podem ser "executadas" todas ao mesmo tempo, usualmente sincronizadas através de um sinal, como o de *clock*.

2.5.1. Etapas de Projeto em FPGA

Tipicamente, a etapa de início de um projeto em FPGA é a de compilação, que consiste em duas etapas. Primeiro uma representação intermediária do projeto do *hardware* é produzida, passo chamado de síntese (*synthesis*). A síntese consiste na tradução do código da linguagem HDL para uma linguagem mais próxima da implementação, uma representação chamada de *netlist*. *Netlist* independe de um FPGA ou CPLD em particular; ele é armazenado geralmente em um formato padrão, conhecido como *Electronic Design Interchange Format* (EDIF) [Barr, 1999]. A existência prévia de layouts para os componentes de *hardware* selecionados facilita bastante este processo.

A segunda etapa neste processo de tradução é chamada de "*place & route* (posicionar e rotear), este passo envolve traçar as estruturas lógicas descritas no *netlist* em macro células, interconexões e pinos reais de entrada e saída. Este processo é similar à etapa de desenvolvimento de uma placa de circuito impresso, permitindo otimizações manuais ou automáticas das disposições. O resultado deste processo é um *bitstream*, bits de configuração de um dispositivo, determinando a função que o mesmo irá desempenhar a partir do momento em que é configurado ou reconfigurado [Barr, 1999]. Neste momento são especificadas também as portas de entrada ou de saída de cada elemento reconfigurável da matriz, gerando a configuração do roteamento dos dados. Quanto melhor o roteamento, melhor a utilização da área do dispositivo reconfigurável e melhor o desempenho conseguido na execução das funções configuradas no dispositivo.

Seguem-se após estas etapas os períodos de simulação funcional. A verificação funcional tem o objetivo de checar todas as funcionalidades de projeto e assegurar que estas estão ocorrendo da maneira especificada. Os testes em projetos de *hardware* podem

ser realizados através de elementos chamados *testbenches*, usados para criar simulações para o modelo do *Design Under Verification* (DUV) ou Entidade Sob Teste (EST) que é representado em alguma linguagem de descrição de *hardware*. A função do *testbench* é criar estímulos que consigam ativar as funcionalidades desejadas no EST [Souza, 2008]. Os valores obtidos com a simulação da entidade de projeto podem ser observados através de uma janela própria da ferramenta de simulação, utilizando formas de onda.

Uma vez criado um *bitstream* para uma FPGA é necessário baixá-lo no dispositivo. Os dispositivos de lógica programáveis são como memória, as mesmas siglas são utilizadas: PROM (para os programáveis apenas uma vez), EPROM, flash, etc. As tecnologias EEPROM e flash incluem suporte à gravação "*in-circuit*, se assemelhando aos micro-controladores, podendo suportar inclusive as interfaces JTAG. Em adição às tecnologias de memória permanente, existem também dispositivos baseados na tecnologia SRAM com índices de memória temporários, necessitando ter seus dados recarregados após cada restauração do sistema ou do chip. Atualmente seu conteúdo pode ser manipulado *on-the-fly*; onde o *bitstream* é carregado de uma origem remota através de uma rede, de modo que o projeto do *hardware* pode ser atualizado de forma tão simples quanto acontece com *software*.

2.6. Ambiente Integrado de Desenvolvimento (IDE)

Toda a parte de projeto, processamento, simulação e programação de FPGA são feitas através de programas específicos, e cada fabricante disponibiliza programas que suportam suas próprias FPGA's ou fazem parcerias com empresas que produzam tais programas. Devido a isso, esses programas específicos – *Integrated Development Environment* (IDE) – possuem características distintas e diferentes formas de realizar os processos de desenvolvimento de FPGA, como síntese e simulação.

Este item tem como objetivo abordar partes do processo de desenvolvimento em FPGA em três ferramentas: Quartus II (Altera), ISE Design Suite (Xilinx) e Lattice Diamond (Lattice), a fim de demonstrar algumas de suas especificidades.

2.6.1. Quartus II 9.0

O Quartus II pertence à empresa Altera que lançou, em 1984, o primeiro CPLD e ocupa o segundo lugar no mercado de dispositivos lógicos reconfiguráveis. Ao inicializar este IDE encontra-se o ambiente de trabalho exibido pela Figura 2.

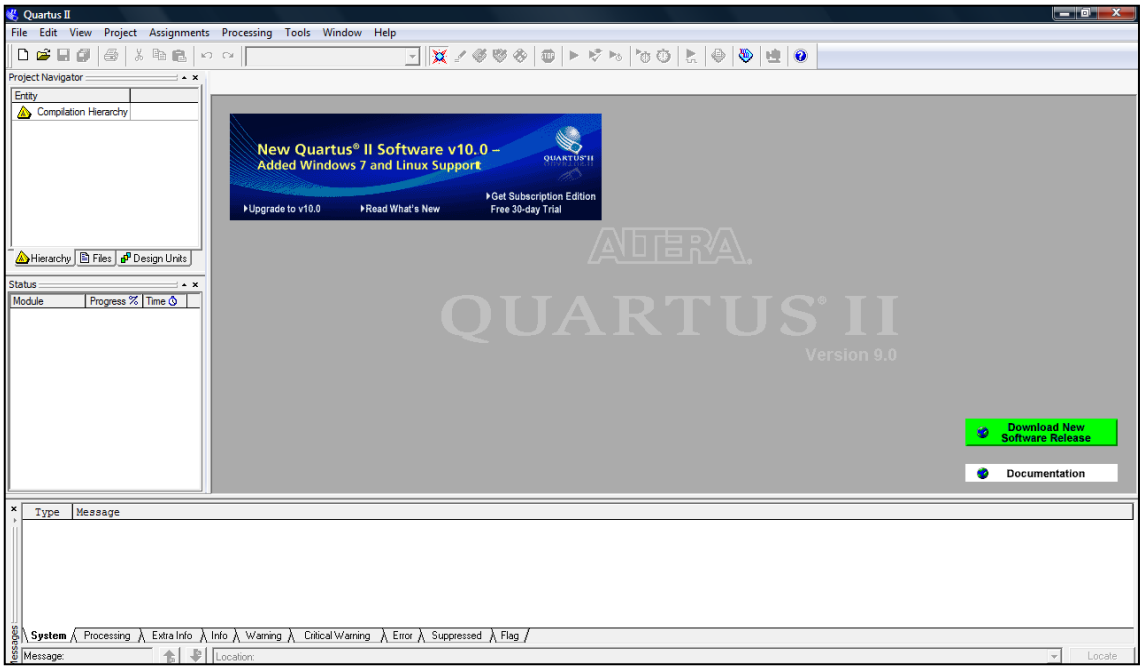

Figura 2 - Ambiente de Trabalho (Quartus II)

Primeiramente, cria-se um novo projeto, fornecendo, passo a passo, suas principais informações: o diretório onde ele será criado, o seu nome, o nome da entidade que ficará no topo de sua hierarquia (automaticamente será dado a este arquivo o nome do projeto), arquivos de design previamente implementados a serem adicionados, família, dispositivo e ferramentas opcionais de síntese, simulação e análise temporal. Os campos onde deverão ser inseridas essas informações, as diversas janelas onde os mesmos se encontram e um resumo geral das características do projeto é apresentado pela Figura 3.

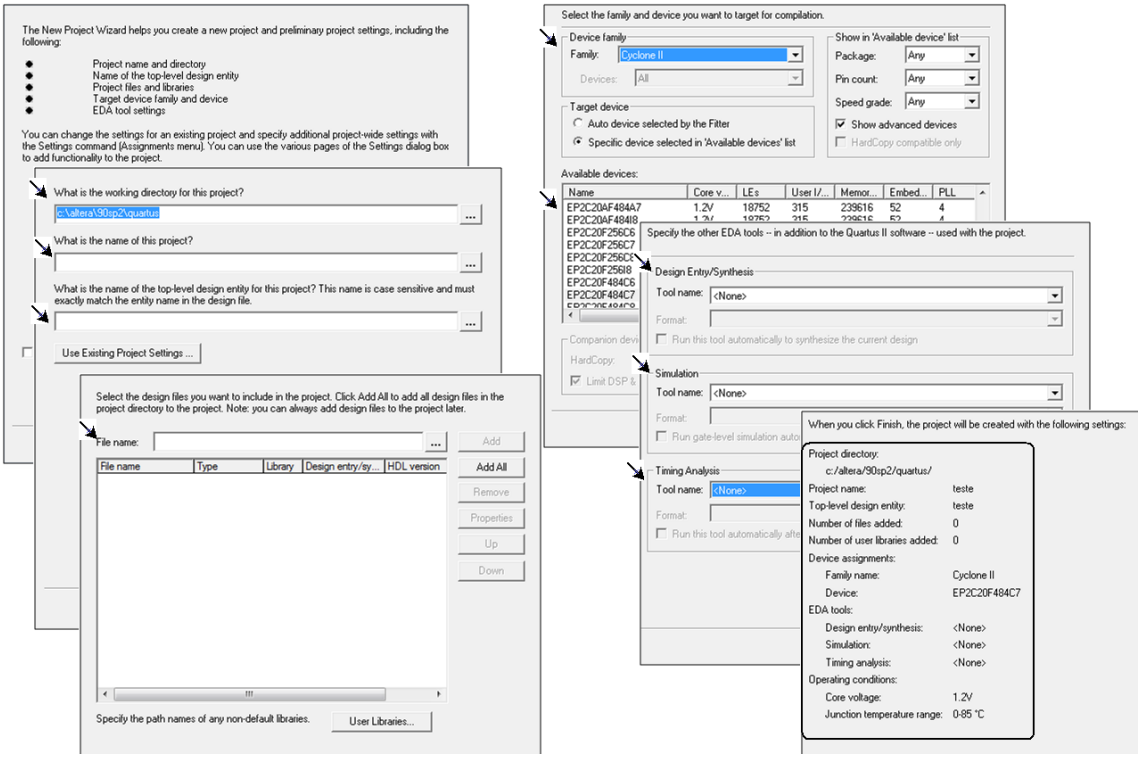

Figura 3 - New Project Wizard

Após a criação de um novo projeto e do desenvolvimento de um código-fonte, sintetiza o a fim de verificar sua sintaxe implementação na FPGA. O Quartus II 9.0 possui uma ferramenta de síntese integrada, logo, não é necessário o uso de outra ferramenta para este processo, mesmo oferecendo esta opção.

Na Figura 4 encontram-se, indicados por setas e círculos, alguns dos pontos relevantes para realizar o processo de síntese. A janela *Project Navigator*, contém, em suas três abas (*Hierarchy*, *Files* e *Design Units)*, informações a respeito do arquivo ou projeto aberto na área de trabalho, como por exemplo, endereço e hierarquia do arquivo em relação a um projeto. A janela *Status* identifica por meio de barras de 0 a 100%, a situação das etapas da compilação. Na região superior da Figura 4 encontra-se a barra de ferramentas da área de trabalho, contendo o botão para início da compilação (símbolo roxo). Na região inferior encontram-se as abas *Warnings* e *Error*, importantes para a visualização de possíveis erros ou ambigüidades na sintaxe do código implementado. Por último, na região referente à área de trabalho, depois de completada a síntese, encontra-se o *Flow Summary*, um resumo das informações deste processo.

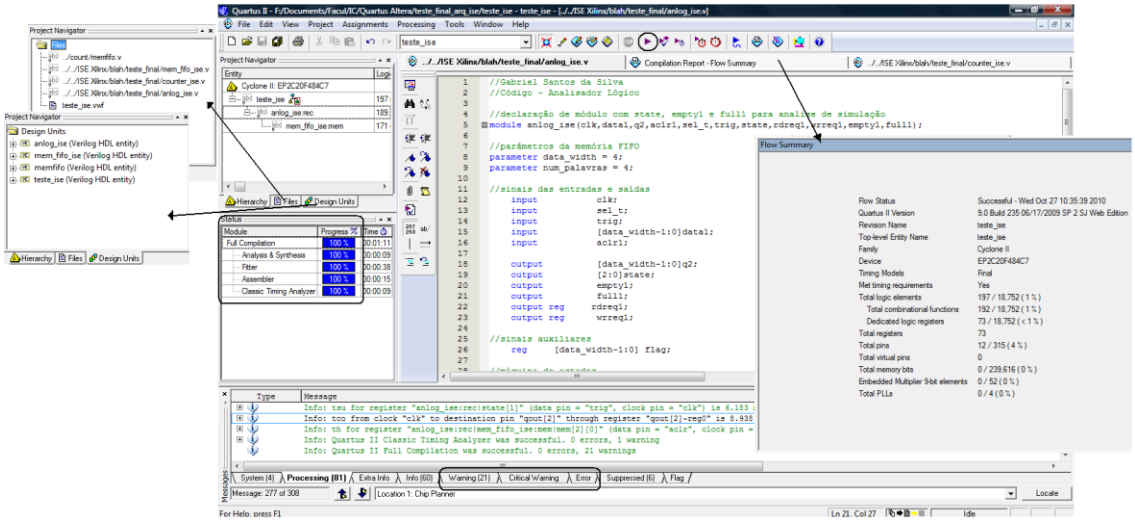

Figura 4 - Processo de Síntese

O Quartus II 9.0 também possui um simulador integrado, além de permitir o uso do simulador ModelSim. A partir de sua última versão, 10.0, o simulador integrado foi removido. Uma das maneiras mais simples de simular o projeto é por meio de um arquivo *Vector Waveform*. Este arquivo permite a inserção dos sinais a serem simulados que preenchem a área de trabalho de forma semelhante à Figura 5. Nesta figura a seta indica a barra de ferramentas de simulação, e o círculo a sua inicialização. A janela Status, que anteriormente indicava a situação do processo de síntese, passa a indicar a situação do processo de simulação.

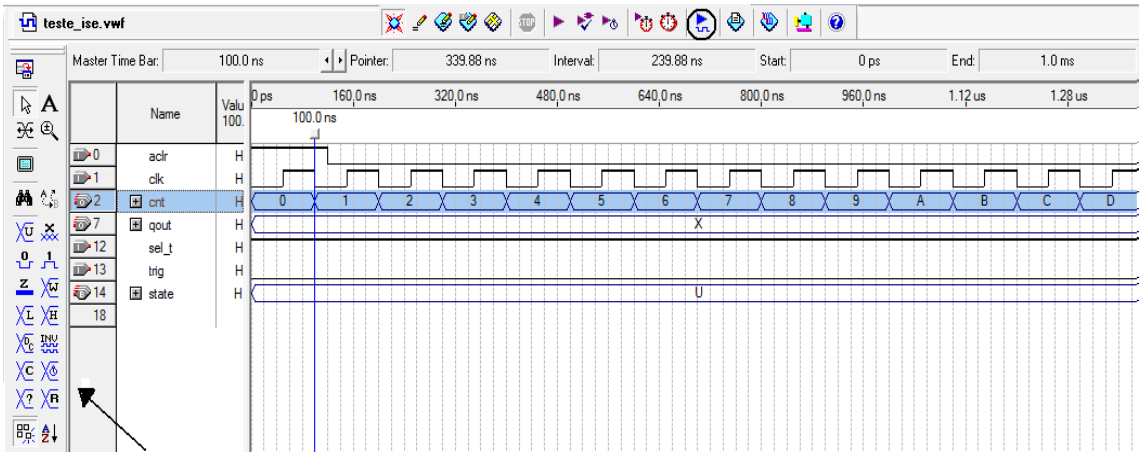

Figura 5 - Processe de Simulação (Quartus II)

Este simulador integrado permite ao usuário modificar, de forma simples, os sinais a serem simulados, fornecendo a eles novos valores em intervalos de tempo específicos. Além disto, as modificações nos códigos-fontes fornecem um retorno de visualização fácil e dinâmico.

Este IDE possui um gerador de *IP-cores*, o Mega Wizard Plug-In Manager. A Figura 6 indica duas janelas referentes a este aplicativo. A janela inferior seleciona qual o *IP-core*, a ferramenta e a linguagem de descrição de *hardware* a serem utilizadas, campos indicados por círculos. A janela superior indica uma das etapas pelas quais o usuário deve percorrer para configurar seu *IP-core*, no caso a memória FIFO. Seus círculos exemplificam algumas destas configurações: largura de dados, número de palavras e sinal de *clock*. Nas demais etapas o usuário pode escolher os sinais de controle, o tipo de acesso de leitura, o tipo de bloco de memória, entre outras opções para a implementação. Esta ferramenta é muito prática, fornece ao usuário, na janela superior, uma simbologia equivalente a memória configurada.

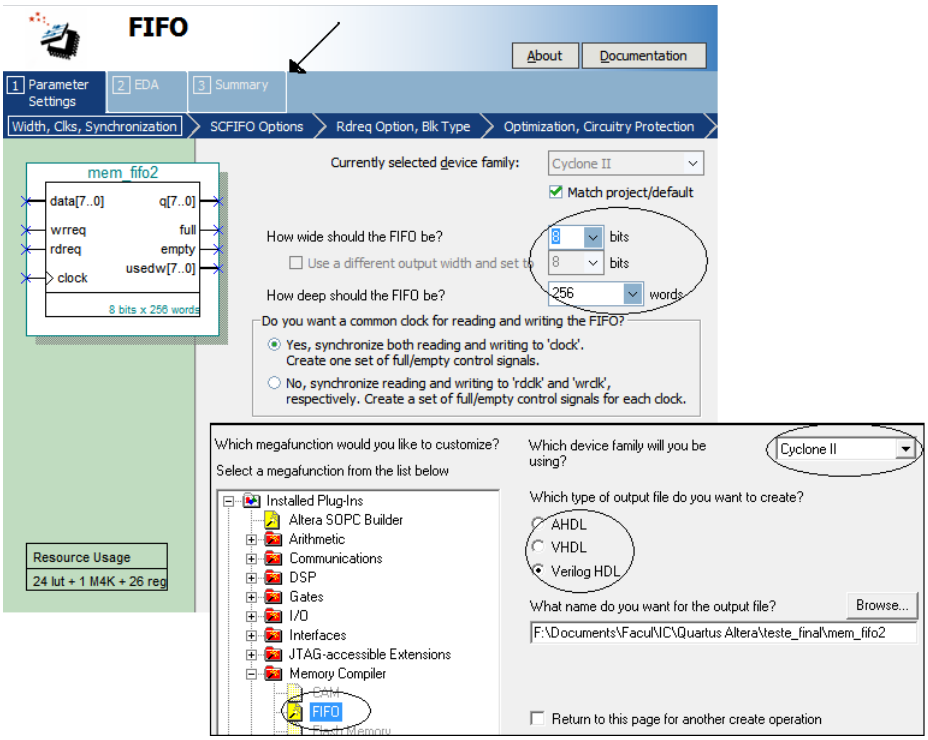

Figura 6 - Mega Wizard Plug-In Manager

2.6.2. ISE Design Suite

O ISE pertence à empresa Xilinx, maior fabricante de dispositivos lógicos reprogramáveis, liderando este mercado desde a década de 90. Ao iniciar este IDE, o mesmo apresenta uma janela à esquerda da sua tela inicial para que o usuário inicie seu trabalho, podendo optar por criar um novo projeto, abrir um existente, entre outros, conforme indicado na Figura 7.

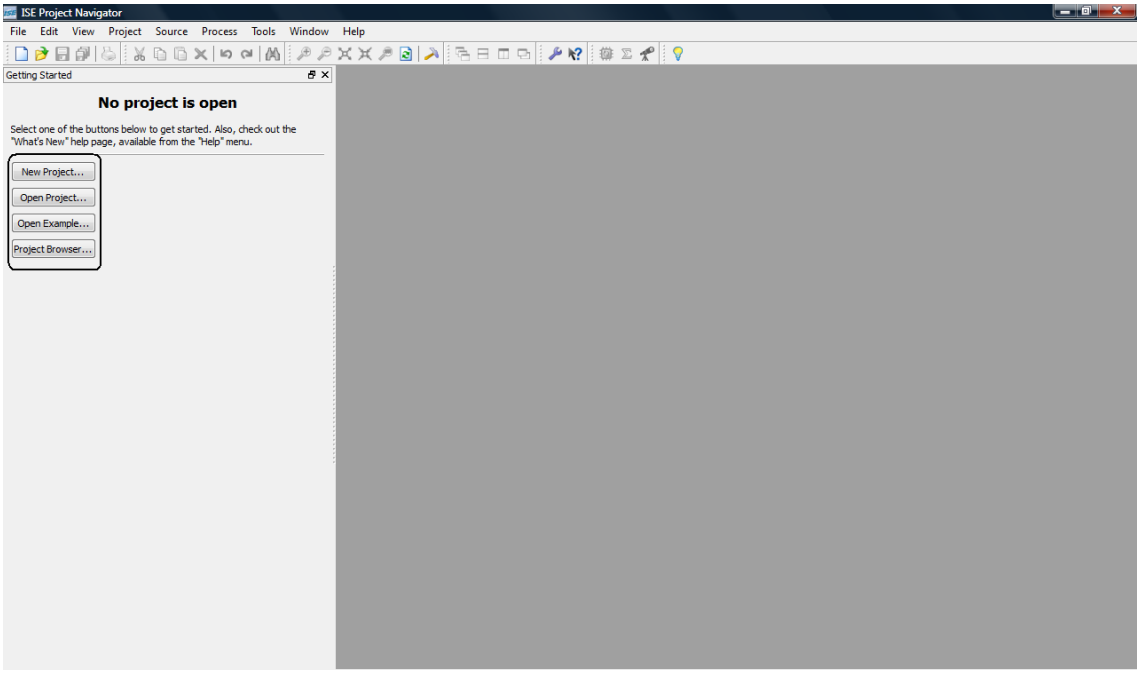

Figura 7 - Ambiente de Trabalho (ISE)

Optando pela criação de um novo projeto, novas janelas se abrirão para que o usuário forneça as características desejadas para o mesmo (semelhante ao Quartus II), sendo elas demonstradas na Figura 8. As várias informações necessárias são indicadas por setas e estão circuladas as opções de ferramenta de síntese e simulação. Diferentemente do Quartus II, a ferramenta de simulação não é integrada ao IDE, apenas pertencente à mesma fabricante, Xilinx.

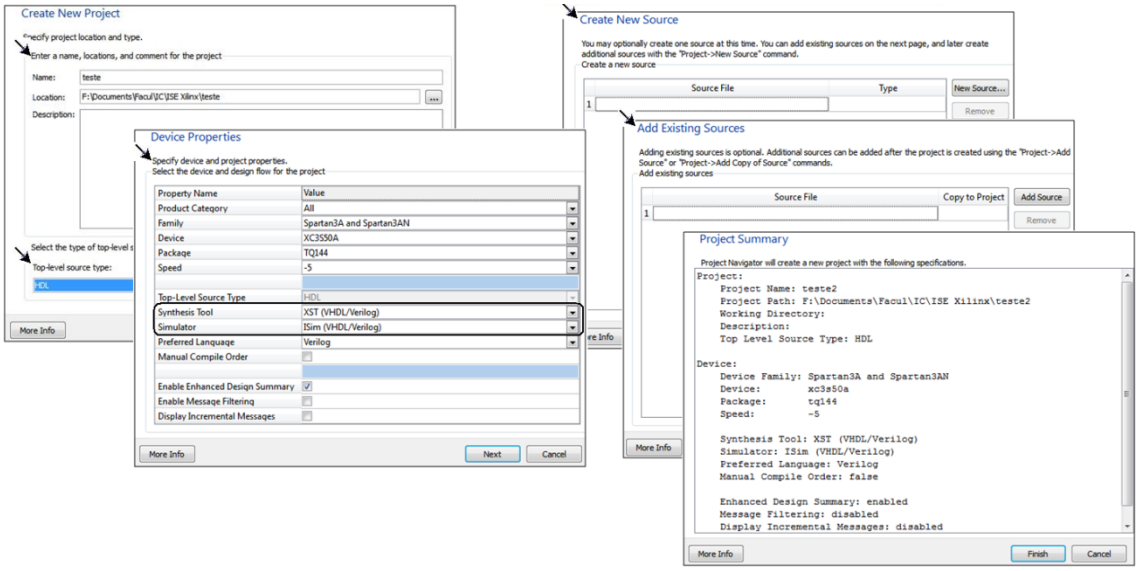

Figura 8 - New Project

O processo de síntese utiliza a ferramenta *Synthesize-XST*, opção *default* para este processo, aplicativo integrado pertencente à própria Xilinx.

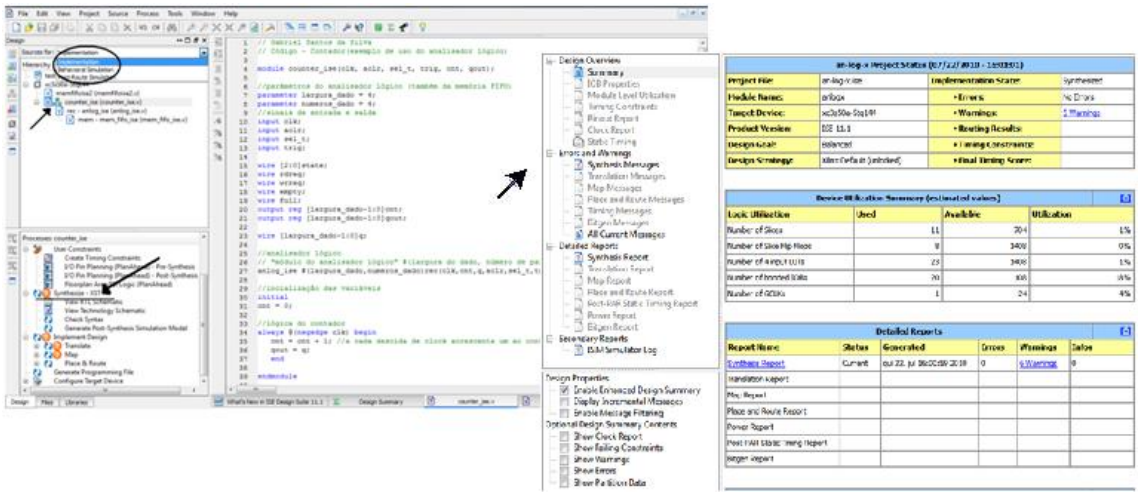

Figura 9 - Processo de Síntese (ISE)

Pode-se dividir a Figura 9 em três janelas principais, a área de trabalho - parte maior, *Hirearchy* - superior à esquerda, e *Processes* - inferior à esquerda. A seta em *Hirearchy* indica qual arquivo está no topo da hierarquia do projeto, enquanto que a seta em *Processes* indica a inicialização da síntese dos módulos, entre outros processos. O círculo indica as categorias dos processos encontrados em *Processes*, implementação (opção para o processo atual), simulação comportamental e simulação *post-route*. A análise de *warnings* e *errors* se dá por meio da janela *Design Summary*, janela sobreposta à

área de trabalho indicada pela seta mesma, que surge após o término do processo de síntese. Vale ressaltar que para a compilação de alguns códigos deve-se modificar a opção *loop iteration limit* nas propriedades de síntese, visto que a mesma é pequena em sua condição *default*.

Para realizar a simulação do projeto este IDE utiliza-se um arquivo de *testbench*, juntamente ao aplicativo ISim, simulador não integrado a ISE mas pertencente a própria Xilinx. Antes de se inicializar este processo, deve-se modificar a categoria de processo de *Implementation* para *Behavioral Simulation*, o que modifica a janela *Processes*, apresentando as seguintes ações: *Behavioral Check Syntax* e *Simulate Behavioral Model*. A primeira ação verifica a sintaxe do arquivo de *testbench* relacionado ao projeto e a segunda inicia o processo de simulação em si. Se a sintaxe do *testbench* estiver correta, ao se inicializar a simulação, uma nova janela se abrirá automaticamente com a ferramenta ISim, mostrando, por meio de formas de onda, a resposta do processo desejado, conforme a Figura 10.

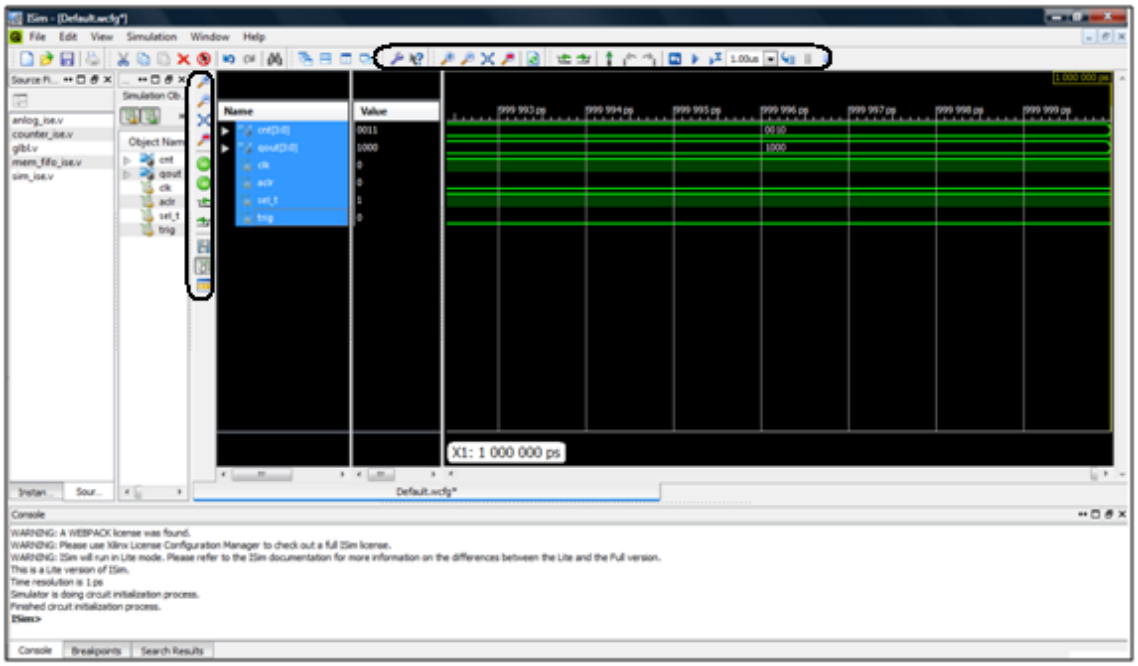

Figura 10 - ISim

Esta figura apresenta as duas barras de ferramentas do ISim, com funções que facilitam a visualização do resultado da simulação. Devido ao uso de *testbench*, menos pratico que um arquivo *waveform*, e do ISim, este processo se torna bem mais trabalhoso.

A realização e visualização de pequenas alterações a serem simuladas é bem menos dinâmica, sendo necessário retornar ao ISE ao invés de realizá-las no próprio ISim.

Como o Quartus II, o ISE também possui uma ferramenta de geração de *IP-core*, (IP *CORE Generator & Architecture Wizard*). A Figura 11 mostra duas janelas referentes ao aplicativo, a inferior, para seleção de qual *IP-core* se deseja trabalhar, e a superior, para sua configuração. Esta janela por sua vez se divide em duas áreas, a da esquerda, com um esquemático do *IP-core* (memória FIFO), modificando-o à medida que ele é configurado; e a da direita, possuindo etapas de configuração semelhantes ao o Mega Wizard Plug-In Manager.

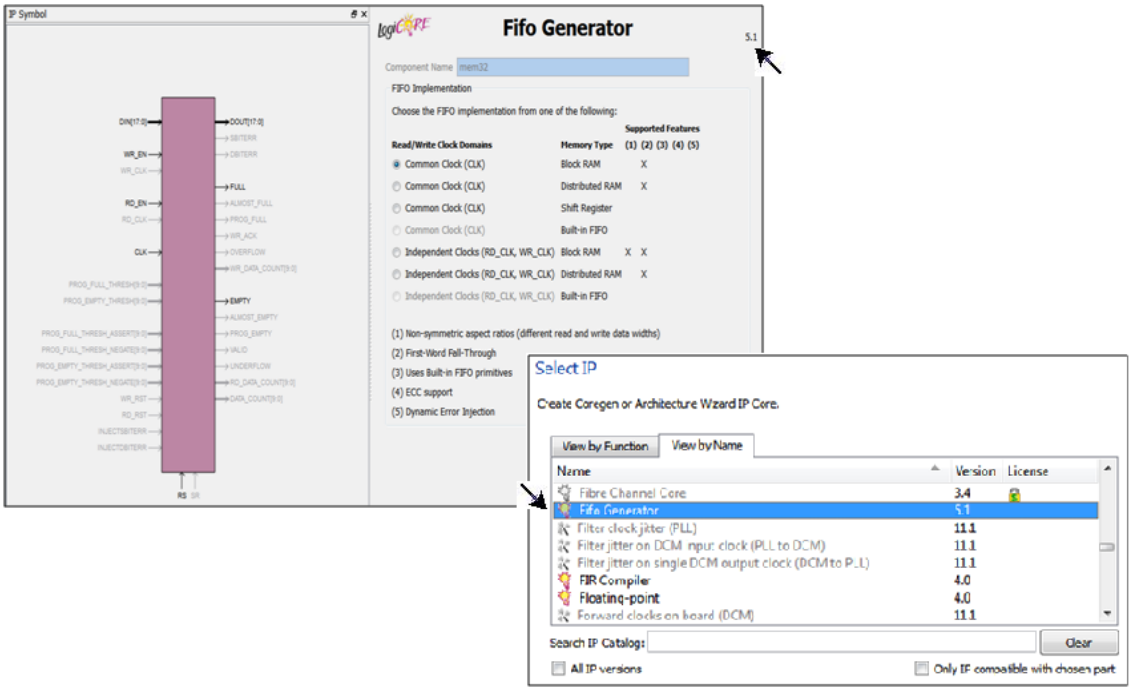

Figura 11 - IP (CORE Generator & Architecture Wizard)

2.6.3. Lattice Diamond

O Lattice Diamond pertence à empresa Lattice Semicondutor, pioneira do sistema de programação ISP e uma das três maiores fabricantes de dispositivos reconfiguráveis de todo o mercado internacional. Em sua inicialização, o Diamond apresenta uma página inicial contendo suas opções iniciais e diversos *links* de acesso rápido a projetos trabalhados anteriormente e a guias de usuário e tutoriais, como mostra a Figura 12.

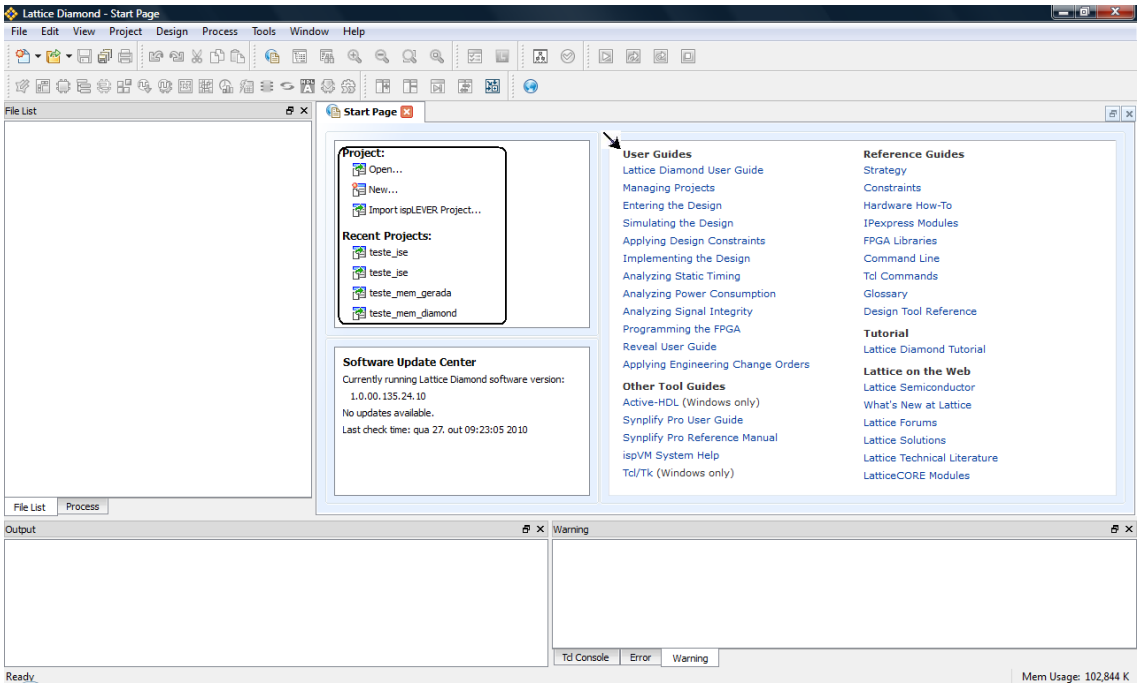

Figura 12 - Área de Trabalho (Diamond)

A seqüência de criação de um projeto é semelhante as já abordadas, com diversas janelas para seleção das configurações desejadas, conforme mostra a Figura 13. As configurações iniciais do projeto estão indicadas por setas. A última janela apresenta um resumo das configurações do projeto.

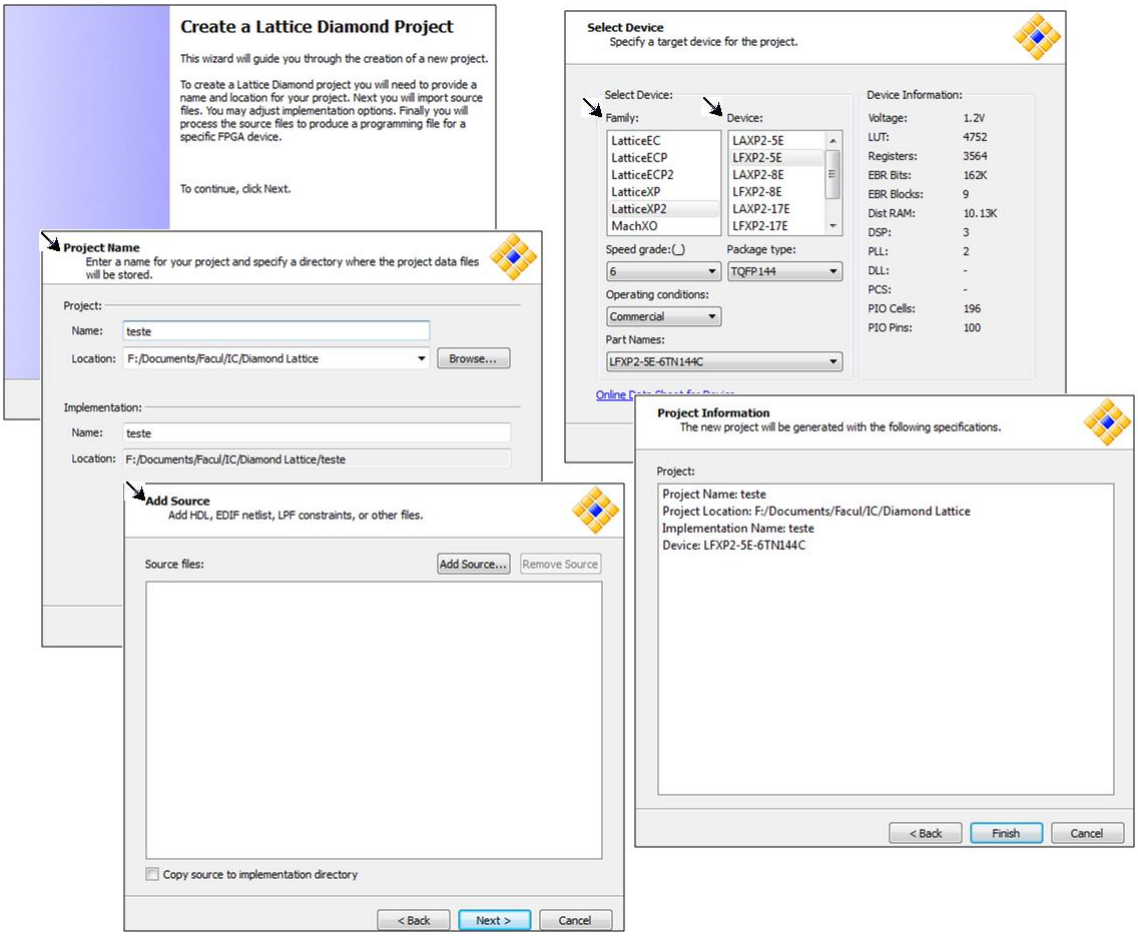

Figura 13 - Criando um Novo Projeto

Após a criação de um novo projeto e do desenvolvimento dos códigos desejados, a área de trabalho assume a forma encontrada na Figura 14, iniciando o processo de síntese. Nesta figura destacam-se duas abas, *File List* e *Process*, encontradas à esquerda do ambiente de trabalho. A primeira apresenta os arquivos incorporados ao projeto e a segunda os processos pelos quais ele pode ser submetido. Para iniciar a síntese do projeto basta um duplo clique na região circulada, contendo o nome do processo e da sua ferramenta integrada, *Synplify Pro*, que, diferentemente dos demais IDE's, é um aplicativo de outra empresa, empresa Synopsys.

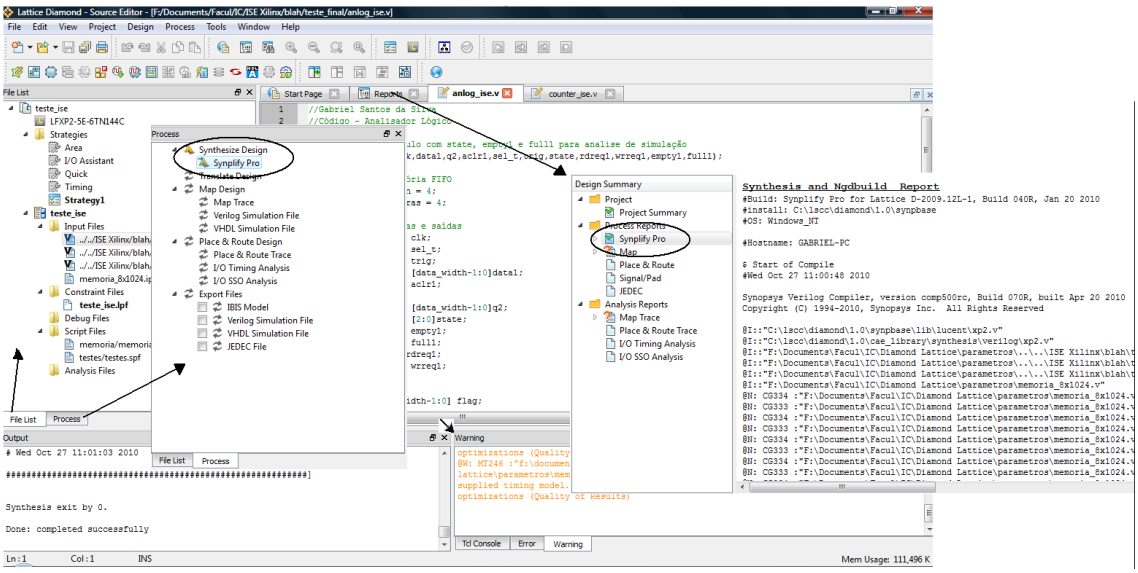

Figura 14 - Processo de Síntese (Diamond)

Com a síntese do projeto realizada, o usuário pode analisar o *report* deste processo, indicado pela seta presente na área de trabalho do Diamond. Para a visualização de possíveis erros e *warnings* utiliza-se a janela mais inferior da Figura 14, também indicada por uma seta.

Para o processo de simulação utiliza-se uma ferramenta externa que necessita de projeto próprio, *Active-HDL Lattice WebEdition 8.2*, aplicativo também pertencente a outra empresa, empresa Aldec. Para a criação deste projeto existem etapas de criação semelhantes as demais, com suas configurações: nome, ferramenta de simulação (a ferramenta *ModelSim* pode ser utilizada), tipo de processo e arquivos-fontes de sinais. Após a criação deste novo projeto, abre-se o *Active-HDL*, com uma área de trabalho conforme apresentada pela Figura 15.

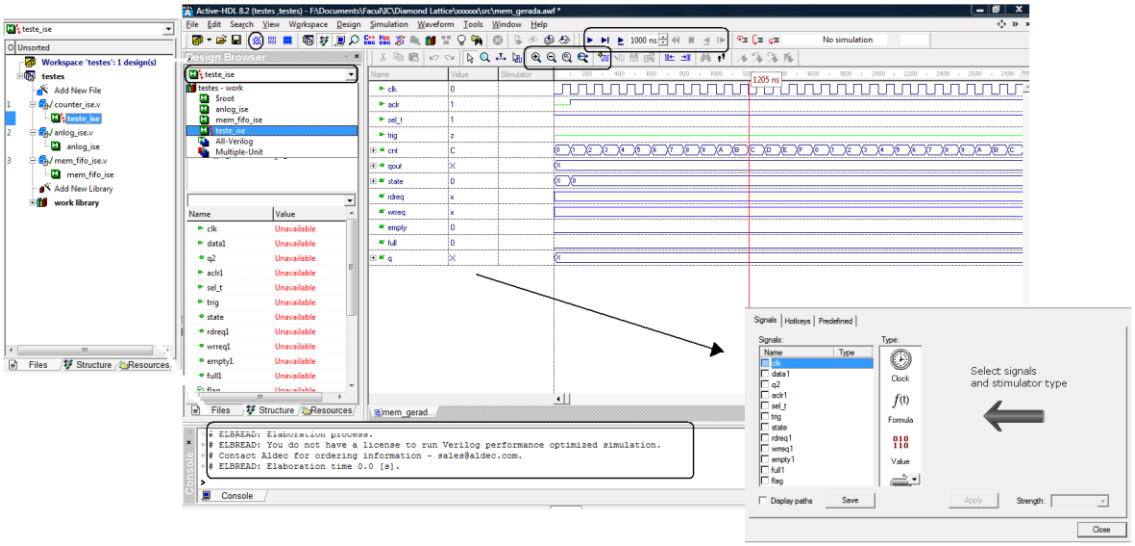

Figura 15 – Active-HDL

O Active-HDL possui uma janela central onde podem ser vistos os arquivos abertos e simulados. Ademais da central, possui também uma janela inferior, *Console*, contendo a descrição de possíveis erros e *warnings*, e uma janela lateral à esquerda, *Design Browser*. Esta possui abas úteis para o usuário, sua aba *Files* mostra os arquivos já contidos no projeto e fornece a opção de se criar um novo, enquanto que a aba *Structure*, apresenta a escolha do arquivo "raiz" do projeto. O arquivo "raiz" fornece sinais para a simulação, recebendo a palavra *top* em vermelho ao lado do seu nome Esta escolha deve ser feita pelo campo *top-level selection*, região circulada da janela *Design Browser*. Os atalhos mais utilizados durante o processo de simulação também estão circulados.

Mesmo a simulação sendo feita por um arquivo *waveform*, a forma de atribuição de valores aos sinais a serem simulados é diferente da anterior, sendo necessário o uso da janela sobreposta às demais, responsável pela seleção de um sinal e o tipo de valor a ser atribuído. O Active-HDL não é uma ferramenta de simulação muito simples devido à forma como se atribui valores aos sinais, sendo visualizados somente após a finalização do processo. Apesar disto, possui como diferencial o fato do usuário poder modificar e compilar os códigos do projeto original no próprio *Active-HDL*, permitindo uma rápida visualização das respostas a pequenas alterações.

Sua ferramenta de geração de *IP-core* é a *IPexpress*, mais simples que as demais, possuindo apenas uma etapa de configuração. A Figura 16 indica duas janelas, a inferior com opções de qual *IP-core*, linguagem de descrição de *hardware* e família deseja-se

trabalhar, e a opção *customize*, responsável pela abertura da janela superior. Esta, por sua vez, possui as possíveis configurações da memória, indicadas pelo círculo, e sua representação, indicada pela seta.

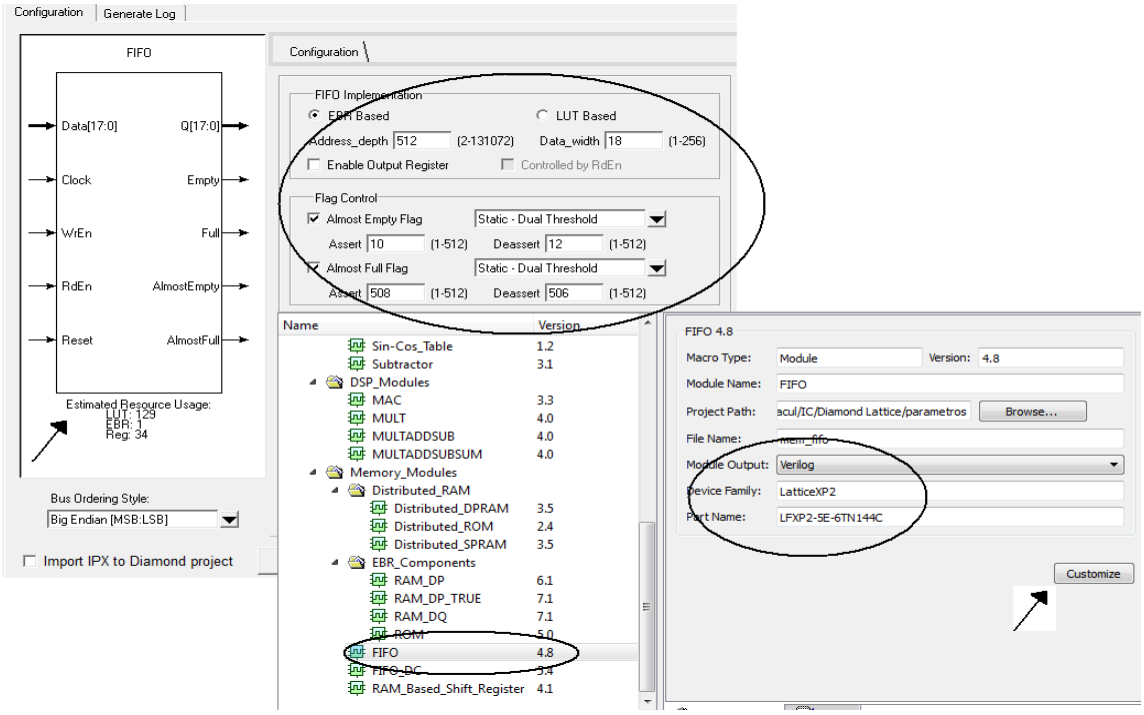

Figura 16 - IPexpress

Vale ressaltar que, para a simulação do código gerado referente à memória FIFO utilizando o *Active-HDL*, foi necessário adicionar ao mesmo duas linhas de comando obtidos em fóruns do próprio Diamond:

```
PUR PUR_INST(.PUR(1'b1));

GSR GSR_INST(.GSR(1'b1));
```

2.7. Arquitetura das Famílias de FPGA

Os principais fabricantes de FPGA possuem várias famílias de dispositivos, cada qual voltada para tipos diferentes de aplicações. Devido à disponibilidade do laboratório que apoiou o desenvolvimento deste projeto, foram escolhidas as seguintes famílias e arquiteturas: Cyclone II (EP2C20F484C7) [Altera, 2008], Spartan 3A(N) (XC3S50A-5TQ144) [Xilinx, 2009] e LatticeXP2 (LFXP2-5E-6TN144C) [Lattice, 2010].

Na análise destes dispositivos um aspecto muito importante é o estudo de suas arquiteturas internas, considerando os blocos lógicos e os de interconexão. A seguir apresenta-se a análise das arquiteturas dos dispositivos reconfiguráveis citados.

2.7.1. Cyclone II (EP2C20F484C7)

Dispositivos da família Cyclone II contêm uma arquitetura bi-dimensional para implementar a lógica personalizada, responsável pelas conexões de *Logic Array Blocks* (LAB's), blocos de memórias, multiplicadores, etc. Estes LAB's contêm 16 elementos lógicos, pequenas unidades responsáveis pela implementação de funções lógicas. Estão presentes também nessa arquitetura blocos de memória denominados M4K, dispostos em colunas entre alguns LAB's; e os blocos de multiplicadores embutidos, dispostos da mesma maneira.

Como já citado, a menor unidade lógica da arquitetura do Cyclone II é o elemento lógico, *Logical Element* (LE). Ele é compacto e fornece diversos recursos como: LUT de quatro entradas, podendo implementar qualquer função de quatro variáveis; um registrador programável; um bloco *carry chain;* entre outros. O Quartus II é responsável por adequá-lo para seu melhor modo de operação: modo normal, apropriado para funções gerais de lógica e funções combinacionais, e modo aritmético, ideal para implementar contadores, por exemplo.

Além dos 16 LE's, cada LAB consiste de: sinais de controle, cadeia de registradores, locais de interconexão, etc. Estas conexões locais realizam a transferência de dados entre LE's de um mesmo LAB, enquanto que a cadeia de registradores realiza essa transferência entre LE's de diferentes LAB's. Essas conexões garantem a eficiência de área e o desempenho do processo de síntese.

A memória embutida do Cyclone II consiste de colunas de blocos de memória M4K. Estes blocos possuem registradores de entrada e de saída, além de serem capazes de implementar vários tipos de memória (single-port RAM, ROM, FIFO *buffers*, etc.).

Dispositivos Cyclone II possuem ainda blocos multiplicadores otimizados para multiplicação intensiva de funções de processamento digital de sinais (DSP), podendo operar como um multiplicador de 18-bit ou dois independentes de 9-bit.

2.7.2. Spartan 3A(N) (XC3S50A-5TQ144)

A arquitetura da geração Spartan-3 consiste de cinco elementos programáveis fundamentais: CLB's que podem operar para implementação lógica e armazenamento de dados por meio de LUT's; blocos de RAM que armazenam dados na forma de blocos de 18-Kbit; blocos multiplicadores; e os *Digital Clock Manager* (DCM's). Esta geração possui uma rica rede de traços que interconectam esses elementos funcionais e transmitem sinais entre os mesmos, possuindo uma chave matricial associada que permite múltiplas conexões no roteamento.

Os DCM's fornecem capacidades avançadas de *clock* para aplicações das FPGA's da geração Spartan-3. Estes blocos eliminam distorções do sinal de *clock*, multiplicam ou dividem a freqüência do *clock* de entrada para sintetizar uma nova freqüência de *clock*, realiza mudanças de fase do sinal, entre outras aplicações.

Toda a geração Spartan-3 apresenta blocos de RAM's de 18 Kbits organizados em colunas. Usando as várias opções de configuração destes blocos, os mesmos podem criar RAM, ROM, FIFO, grandes LUT's, conversores de largura de dados, *buffers* circulares e registradores de deslocamento.

Os CLB's constituem o principal recurso lógico para implementação síncrona e combinatória de circuitos e estão dispostos em uma matriz regular de linhas e colunas. Cada um destes elementos possuem quatro *slices*, que por sua vez, possuem duas LUT's para implementação lógica e armazenamento de dados, dois *flip-flops* ou *latches*, multiplexadores, *carry-in* e *carry-out*

Os *slices* são agrupados em pares e divididos em dois grupos: os pares da esquerda, que suportam funções lógicas e de memória, e os da direita que suportam somente funções lógicas. Ambos os grupos possuem duas LUT's de quatro entradas, dois elementos armazenadores, dois multiplexadores, além de elementos aritméticos e de *carry*. Os pares da esquerda podem ser utilizados como RAM de 16x1 ou registradores de deslocamento. A combinação de uma LUT e um elemento armazenador é definida como uma célula lógica, e um *slice* equivale a 2,25 células lógicas.

Os multiplicadores têm um funcionamento semelhante aos blocos multiplicadores do item anterior, sendo que neste caso, eles implementam apenas multiplicadores de 18-bit.

2.7.3. LatticeXP2 (LFXP2-5E-6TN144C)

Cada dispositivo LatticeXP2 possui uma matriz de blocos lógicos cercada por *Programmable I/O Cells* (PIC's). Entre as fileiras de blocos lógicos se encontram linhas de *Embedded Block RAM* (EBR's), blocos de memórias, e uma fileira de *Digital Signal Processing* (DSP).

Existem dois tipos de blocos lógicos, o *Programmable Functional Unit* (PFU), responsável por funções lógicas, aritméticas, RAM e ROM, e o *Programmable Functional Unit without RAM* (PFF), responsável pelas funções lógicas, aritméticas e ROM. Todos os blocos são alocados em uma matriz bi-direcional, mas cada fileira possui apenas um dos tipos, além disso, ambos possuem quatro *slices* interligados. Três dos quatro *slices* existentes nos blocos lógicos possuem duas LUT's de quatro entradas e dois registradores, enquanto o ultimo *slice* não possui registradores. Para o tipo PFU, os três primeiros *slices* podem trabalhar como memória distribuída.

Os *slices* podem operar em quatro modos distintos: modo Lógico, onde suas LUTs são configuradas com quatro entradas; modo *Ripple*, permite uma eficiente implementação de pequenas funções aritméticas, gerando dois sinais de *carry* para funções que necessitem concatenar *slices*; modo RAM, onde podem ser construídas memórias 16x4-bit de porta única ou uma pseudo 16x2-bit com porta dupla; e modo ROM.

Dispositivos LatticeXP2 possuem uma ou mais fileiras de blocos EBR's, blocos de memórias de 18 Kbits, sendo que cada bloco pode ser configurado como RAM ou ROM de diversos parâmetros. FIFO's podem ser implementadas com blocos EBR's utilizando PFU's como lógica suporte.

Além destes elementos, esta geração contém uma ou mais fileiras de blocos DSP's, possuidores de multiplicadores e somadores/acumuladores para funções complexas de processamento de sinal.

2.8. Linguagem de Descrição de *Hardware*

Ao projetar o *hardware* de um sistema embutido, HDL's podem ser utilizadas para realizar a descrição dos circuitos eletrônicos, executando sua lógica não somente de acordo com o fluxo do algoritmo, mas também seguindo a temporização de um ou mais *clocks*.

Essas linguagens permitem descrever a forma como os circuitos operam, possibilitando também a sua simulação antes mesmo de sua fabricação. As linguagens mais populares são VHDL e Verilog, que podem ser utilizadas como entrada para simulação e síntese automática de circuitos, através da utilização de ferramentas comerciais bastante difundidas [Carro 2003].

Um programa utilizando HDL pode ser escrito basicamente usando dois tipos (modelos) de descrição: estrutural e comportamental. Na descrição estrutural, é apresentada a organização física e topológica do sistema, ou seja, são especificadas as entradas e/ou saídas, os componentes lógicos, a interligação deles e os sinais que compõem o sistema. Esta descrição pode ser usada como entrada para o processo de simulação da mesma forma que uma entrada esquemática.

Na descrição comportamental é necessária somente a descrição do comportamento do circuito, o funcionamento de cada um de seus componentes. Um programa que utiliza esse tipo de descrição possui o mesmo formato de um programa fonte escrito em uma linguagem de programação de alto nível. Essa abordagem diminui a necessidade de conhecimento em projeto de *hardware*, aumentando a facilidade de desenvolvimento do sistema, no entanto, os sistemas gerados a partir desse tipo de descrição podem não ser tão otimizados em questões de desempenho e área de dispositivo ocupada.

Portanto o diferencial de uma HDL é a sua capacidade de descrever simultaneamente o comportamento de componentes individuais e como estão interligados [Tala], sendo processada seqüencialmente nas estruturas individuais e de forma concorrente entre estas estruturas. Em termos gerais o objetivo principal da HDL é caracterizar a síntese e propiciar a simulação de projetos eletrônicos.

2.8.1. Verilog

Verilog é uma das duas principais HDL's juntamente da linguagem VHDL. VHDL foi definido com um padrão do *Institute of Electrical and Electronics Engineers* (IEEE) em 1987 [Amore, 2005]. Por sua vez Verilog foi aceito por este processo de padronização apenas em 2001, apesar de ter sido criada em 1985 pela *Gateway Design System Corporation*, agora pertencente a *Cadence Design Systems*, *Inc's Systems Division*. Verilog, por sua grande semelhança com a linguem C, é preferida pela maioria dos

profissionais de engenharia elétrica e da computação. Este também foi o quesito decisivo para a escolha do Verilog para a implementação deste projeto.

A linguagem Verilog fornece ao designer digital um meio de descrever um sistema digital em uma ampla gama de níveis de abstração, e, ao mesmo tempo, fornece acesso a ferramentas de projeto auxiliado por computador para ajudar no processo de design a estes níveis. Uma representação abstrata ajuda o designer explorar alternativas de arquitetura através de simulações e detectar pontos críticos do projeto antes de começar o projeto detalhado.

2.9. Analisador Lógico

Na análise de circuitos digitais é importante saber o que ocorre simultaneamente com níveis lógicos em diversos pontos de um circuito [Ribeiro, 1989]. Normalmente os sistemas digitais operam com um fluxo contínuo de dados, usualmente apresentados em um período de tempo controlado por um sinal de temporização. Para verificar se o funcionamento do sistema está correto, há a necessidade de testar os dados e analisar os valores encontrados. O uso do analisador lógico possibilita visualizar e analisar esses dados.

O Analisador Lógico é um instrumento de medida que captura os dados de um circuito digital para posterior análise, de modo similar a um osciloscópio, mas difere deste por ser capaz de visualizar os sinais de múltiplos canais. Além de verificar o correto funcionamento do sistema digital, os sinais capturados podem medir tempos entre mudanças de nível, número de estados lógicos, etc. Utiliza-se um analisador lógico quando é necessária a análise de uma determinada condição lógica, copiando uma grande quantidade de dados digitais do sistema ao que está ligado, e por último, disponibilizando a visualização destes dados e o diagrama de fluxo do sistema.

Os dados podem ser apresentados através de um monitor, em forma de onda ou em valores numéricos (base decimal, hexadecimal ou binária). Isto pode ser realizado através de circuitos lógicos habilitados em um determinado intervalo de tempo. O sinal amostrado é uma palavra binária que deve ser armazenada em uma memória possuindo o menor tempo de acesso possível, para não interferir na velocidade de aquisição de dados. Esta memória será melhor abordada no item seguinte.

Um circuito de temporização interno controla a amostragem e a armazenagem dos sinais de entrada, possibilitando o dispositivo ver a resposta do sistema após uma ocorrência específica [Silva 2002]. Esta temporização normalmente é disparada por um sinal externo, um sinal de temporização do circuito a ser testado. Muitos analisadores lógicos também possuem um temporizador interno que pode ser usado para manusear funções de temporização. Estes sinais, tanto o externo quanto o interno, definem uma das principais características de um analisador lógico, sua capacidade de disparo (*trigger*).

A lógica de funcionamento do analisador lógico é definida através das opções do modo de captura e do modo de disparo. O modo de captura possui duas formas: o modo de sincronismo, onde são colhidas amostras em intervalos regulares, com base em um *clock* interno ou externo; e o modo de estado, onde um ou mais sinais são definidos como *clock*, e os dados são amostrados nas bordas destes sinais. Uma vez que o modo de captura do analisador lógico é escolhido, a condição de disparo pode ser ajustada, podendo variar de uma simples borda de sinal a um conjunto de condições que devem ser cumpridas. Após definir ambas as configurações, o analisador lógico pode ser executado uma única vez, ou repetidamente.

Atualmente, dentre as categorias de analisadores lógicos disponíveis no mercado encontra-se a categoria *PC-based*, onde o *hardware* se conecta a um computador através do cabo *Universal Serial Bus* (USB) ou Ethernet e retransmite os sinais capturados para o *software* no computador. Esta categoria de dispositivos é geralmente muito menor e mais barata, não necessitando de displays externos ou entradas de *hardware*, como teclados ou botões, sendo a categoria utilizada no projeto.

2.9.1. Memória FIFO

Para o armazenamento dos dados adquiridos pelo analisador lógico pode-se adotar uma memória do tipo *First In, First Out* (FIFO). Este tipo de memória é comumente usado em eletrônica para circuitos de controle de fluxo, usualmente partindo do *hardware* em direção ao *software*. Sua nomenclatura, *First In, First Out*, faz referência a sua forma de organização e manipulação de dados relativos à prioridade, o dado que vem em primeiro lugar é também lido em primeiro lugar, análogo ao comportamento de pessoas em uma fila.

A memória FIFO é caracterizada por um conjunto de ponteiros de leitura e escrita, armazenamento e lógica de controle. O armazenamento pode ser feito por meio de SRAM, *flip-flops*, *latches* ou outras formas adequadas. Sua lógica de controle é baseada em seus vários sinais de controle, os mais utilizados são: *full*, *empty*, *write_enable* e *read_enable*. Há também o sinal de *clock*, este define se a FIFO é síncrona, onde o mesmo *clock* é usado tanto para leitura e escrita, ou assíncrona, onde dois sinais de *clock* distintos são usados para estes processos.

Em *hardware*, a memória FIFO é usada para fins de sincronização, geralmente implementada utilizando um ponteiro de leitura e outro de escrita. Inicialmente a memória se encontra vazia e estes ponteiros possuem o endereço da sua primeira posição. Quando o ponteiro de leitura iguala o seu endereço ao endereço do ponteiro de escrita a memória fornece um sinal *empty* em nível alto. Para o caso contrario, quando o ponteiro de escrita alcança o endereço do ponteiro de leitura, a memória fornece um sinal *full* em nível alto. O controle de fluxo gera os sinais *empty* e *full* para que os dados da entrada não substituam o conteúdo já armazenado na memória.

A Figura 17 representa os processos de escrita e leitura, respectivamente, de uma memória FIFO, indicando os principais sinais de controle.

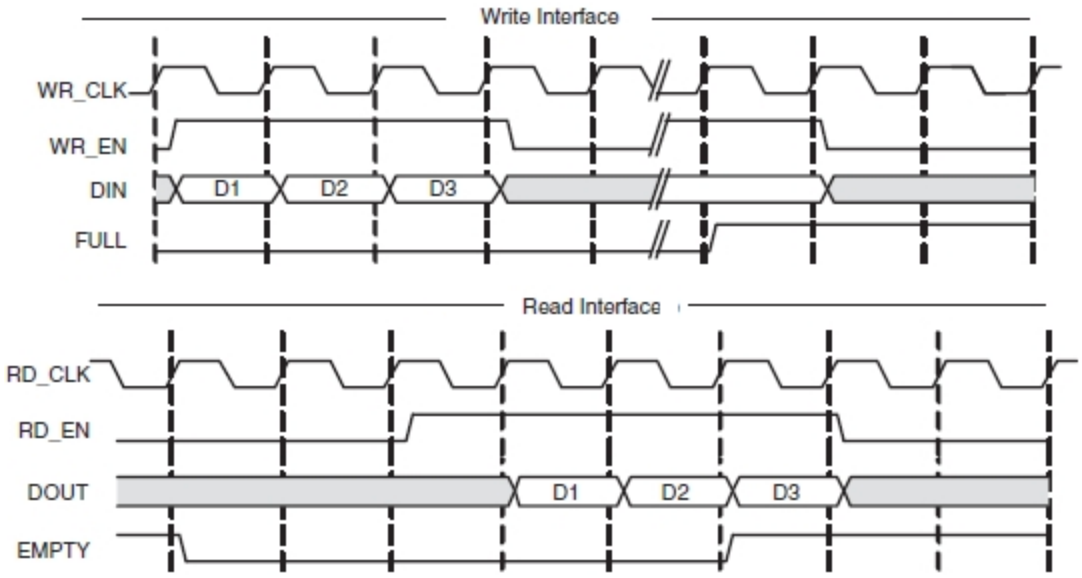

Figura 17 - Ciclos de Escrita e Leitura da Memória FIFO

2.10. Considerações Finais

Como mencionado anteriormente, foram introduzidos, de forma sucinta, os conceitos e as ferramentas adotadas na elaboração do projeto, necessários para o seu melhor entendimento. Deseja-se, portanto, que a leitura dos próximos capítulos deste documento possa ser feita de forma mais satisfatória

CAPÍTULO 3: DESENVOLVIMENTO DO TRABALHO

3.1. Considerações Iniciais

Todo o desenvolvimento das atividades realiza-se nas ferramentas computacionais Quartus II, da Altera, ISE, da Xilinx e Diamond, da Lattice. A elaboração de uma versão inicial dos códigos referentes ao Analisador lógico e sua memória é feita utilizando o Quartus II, devido ao fato deste projeto ser baseado no uso de dispositivos reconfiguráveis da própria Altera, utilizados nos trabalhos anteriores.

Após a conclusão do desenvolvimento dos códigos mencionados, as outras duas ferramentas citadas, ISE e Diamond, são utilizadas para generalização dos códigos elaborados, a fim de se obter um código aberto. As simulações, tanto para a análise dos *cores*, quanto para a validação do funcionamento e medidas de desempenho dos mesmos, são executadas no Quartus II, e nos *software*s integrados para simulação, ISim Simulator, para o ISE, e Active-HDL, para o Diamond.

São utilizadas as linguagens de descrição de *hardware* VHDL e Verilog, conforme a necessidade, para a análise dos *cores* de micro controladores. A escolha da utilização de linguagens de descrição de *hardware*, ao contrário da implementação em esquemático, facilita a implementação e a modificação, tanto dos módulos, quanto da própria arquitetura.

3.2. Projeto

A proposta do projeto todo pode ser descrita em três etapas distintas:

1. Estudo da ferramenta de desenvolvimento Quartus II – empresa Altera – para elaboração dos módulos do Analisador Lógico e de sua memória de armazenamento de dados, além de simulações posteriores a fim de validar seu funcionamento.
2. Estudo das demais ferramentas de desenvolvimento, ISE – empresa Xilinx – e Diamond – empresa Lattice, para que, em posse dos módulos

citados, possam ser realizadas supostas modificações necessárias para generalizá-los, tornando-os códigos abertos.

3. Estudo das arquiteturas das FPGA's adotadas pelos dispositivos em questão, e simulação dos módulos generalizados, variando seus parâmetros de entrada a fim de efetuar uma análise comparativa e qualitativa destes dispositivos.

3.3. Descrição das Atividades Realizadas

Tendo como base as três etapas descritas na seção anterior, descreve-se nesta a implementação dos módulos referentes ao analisador lógico e sua memória FIFO. As demais etapas – simulação dos módulos e análise dos processos de síntese – estão descritas no capítulo seguinte, CAPÍTULO 4: RESULTADOS OBTIDOS.

3.3. Implementação dos *Cores*

Para a implementação do projeto do Analisador Lógico é necessário o desenvolvimento de dois módulos distintos: do próprio analisador e de sua memória FIFO. Em ambos os módulos, para definição de suas lógicas, utilizam-se máquina de estados. Esta modelagem requer registradores para armazenar o estado das variáveis e um bloco de lógica combinacional que determina o estado de máquina de estado.

Ademais destes dois módulos implementa-se um código exemplo afim de definir a atuação do analisador lógico. Este código exemplo é um simples contador síncrono com a mesma largura de dados do analisador lógico. Desta forma podem-se comprovar quais dados são capturados pelo analisador lógico e se a seqüencia dos mesmos está correta. Vale ressaltar que todos os códigos desenvolvidos se encontram nos Apêndices ao final deste documento.

3.3.1. Implementação do Analisador Lógico

A lógica funcional de um analisador lógico pode ser descrita conforme a Figura 18. Os blocos nomeados como Estágio de Disparo e Base de Tempo representam o modo de captura e a condição de disparo, configurações abordadas anteriormente. O bloco memória representa a memória FIFO responsável pelo armazenamento dos dados adquiridos e o

último bloco, interface, é responsável pela forma que os dados são apresentados ao usuário, não implementada neste projeto.

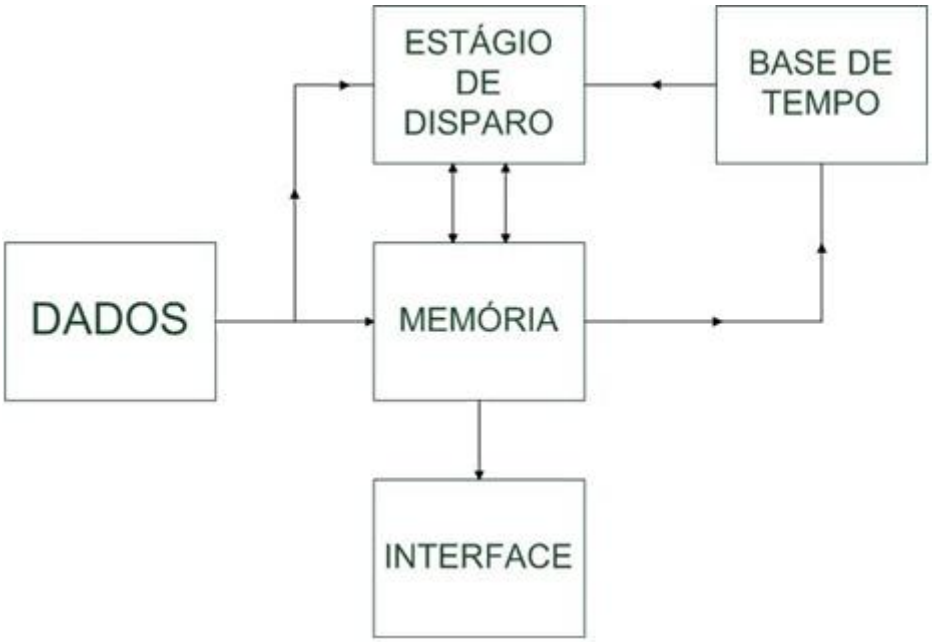

Figura 18 - Diagrama de Blocos do Analisador Lógico

A **Figura 19** a seguir representa uma máquina de estado do analisador lógico e a Figura 20 a um fluxograma com os principais sinais utilizados para a implementação do código referente ao analisador lógico.

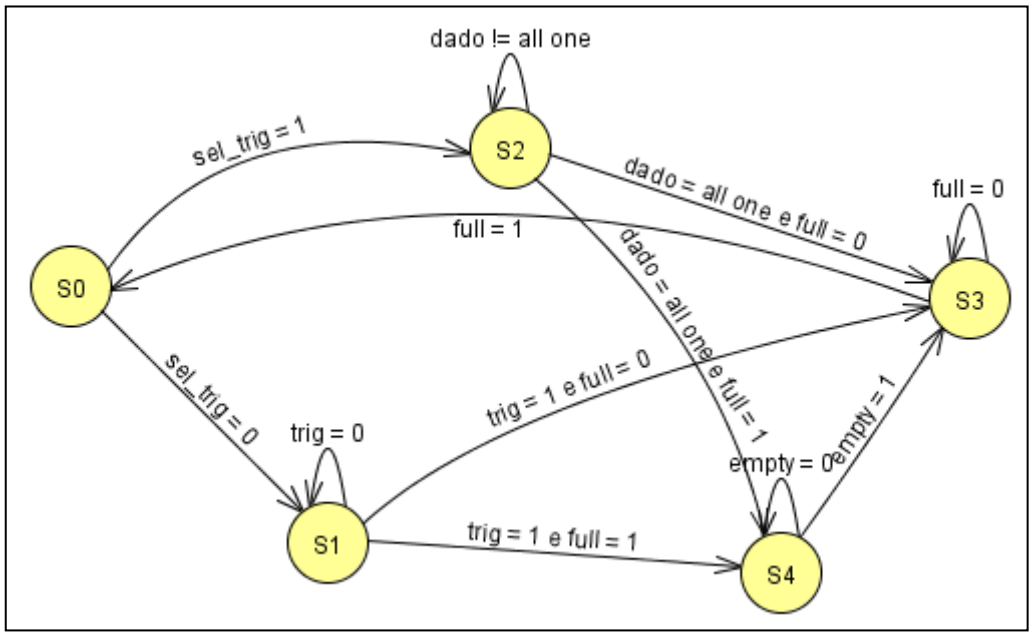

Figura 19 - Diagrama de Máquina de Estados do Analisador Lógico

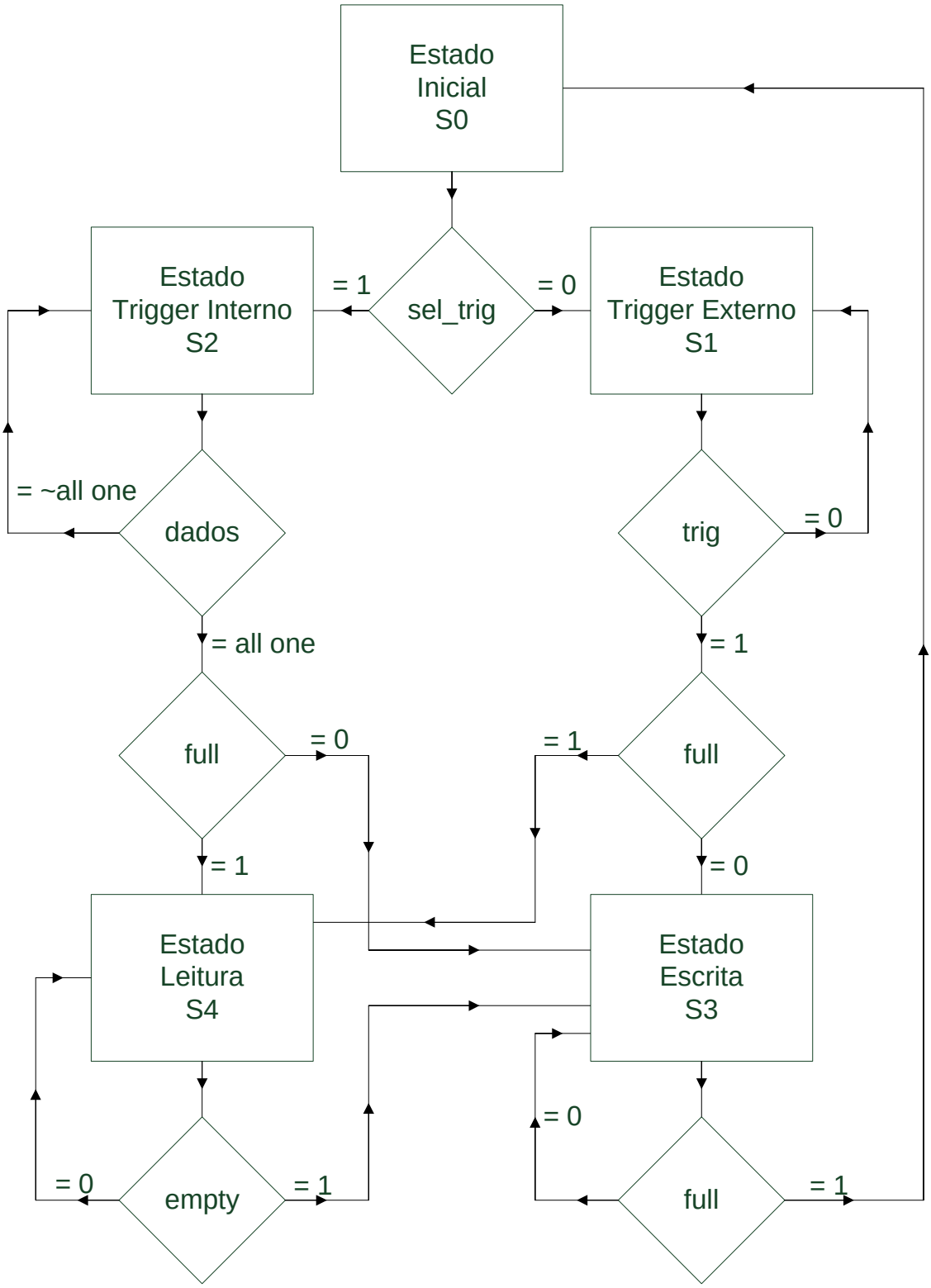

Figura 20 - Fluxograma do Analisador Lógico

Vale ressaltar que o fluxograma não leva em consideração a influência do sinal de entrada `clear`. Independente do estado que o dispositivo se encontra ao ocorrer uma

transição para nível alto deste sinal, o fluxo de dados se dirige para o estado S0, reiniciando todo o processo. O estado S0 representa a inicialização das funções do analisador lógico, zerando os pedidos de leitura e escrita e verificando a opção de *trigger* – externo ou interno - adotado pelo usuário.

Caso a escolha tenha sido pelo *trigger* externo (`sel_trig = 0`), o fluxo de dados se dirige para o estado S1. Neste estado o analisador se mantém em um laço de espera enquanto não ocorrer uma transição positiva do sinal de entrada trig. Este sinal representa o *trigger* externo, sendo o mesmo associado a algum outro sinal advindo do código que se deseja analisar. Caso a memória FIFO esteja toda preenchida, a mesma envia um sinal de aviso (`full = 1`) e o próximo estado será S4, caso contrário (`full = 0`) o estado seguinte será S3.

Caso a escolha tenha sido pelo *trigger* interno (`sel_trig = 1`) o próximo estado será S2. Como acontece no estado S1, o analisador se mantém em um laço de espera enquanto a entrada de dados não apresentar a palavra de *trigger* escolhida, no caso, a palavra *"all ones*" - palavra com todos seus bits em nível alto. Depois de reconhecida a palavra de *trigger* (`dado = all one`), o nível do sinal `full` da memória FIFO é analisado, seguindo os mesmos fluxos descritos anteriormente.

O estado S3 é o ciclo de escrita da memória, nele a memória recebe o sinal `write request` em nível alto e começa a gravar os dados presentes em sua entrada. Pela implementação, a gravação se mantém enquanto a memória não estiver totalmente preenchida, ou seja, enquanto o sinal `full` não estiver em nível alto. Quando esta transição ocorrer (`full = 1`) o fluxo retorna para S0, reiniciando o processo. Este retorno para S0 ocorre a fim de evitar que a memória, ao ser totalmente preenchida, comece a ler os dados armazenados automaticamente, necessitando assim de outro sinal de disparo.

O estado S4 é o ciclo de leitura da memória, nele a memória recebe o sinal `read request` em nível alto e começa a enviar para a sua saída os sinais previamente armazenados no estado S3. Esta leitura se mantém enquanto a memória não estiver completamente vazia, ou seja, enquanto o sinal `empty` não estiver em nível alto. Após isso ocorrer (`empty = 1`) o analisador se dirige para S3, onde se reinicia o ciclo de escrita.

3.3.2. Implementação da Memória FIFO

Este item tem como objetivo explicar a lógica adotada para a implementação de uma memória FIFO, descrita por meio do diagrama de máquina de estados presente na Figura 21 ou por meio do fluxograma presente na Figura 22. O estado q0 é o estado inicial da memória FIFO, onde todos seus sinais de controle apresentam resposta em nível baixo; q1 é o estado "esvaziar", estado de espera pelo sinal habilitador de leitura `rd_en`; q2 é o estado de leitura, onde todas as posições da memória têm seus conteúdos direcionados para a saída da memória, q3 é o estado "preencher", estado de espera pelo sinal habilitador de escrita `wr_en`, e q4 é o estado de escrita, onde os dados presentes na entrada da memória são enviados, um a um, para as posições da memória.

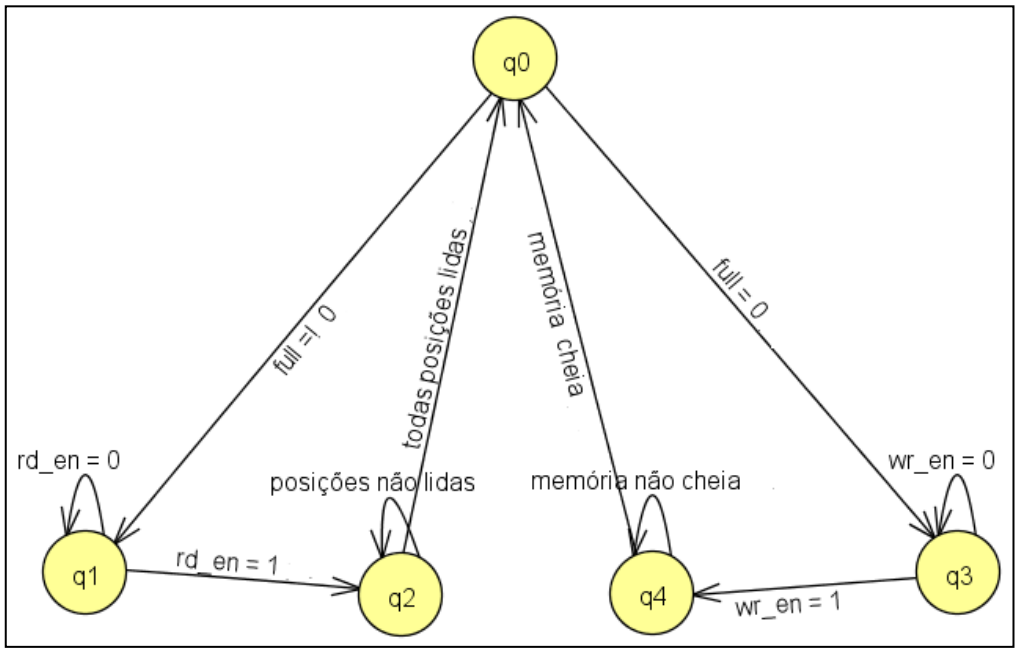

Figura 21 - Diagrama de Máquina de Estados da Memória FIFO

Considerando que a memória se inicia vazia (`full = 0` e `empty = 1`), o fluxo segue de q0 para q3. Neste estado, o fluxo entra em loop, esperando por uma transição positiva do sinal habilitador de escrita (`write_en = 1`), e, ao recebê-la, o fluxo segue para q4. O dispositivo se mantém no estado de escrita até todas as posições de memória serem preenchidas com os dados recebidos em sua entrada, apresentando o sinal `full` em nível alto e `empty` em nível baixo. Ao ser completamente preenchida, a memória muda seu fluxo de q4 para q0.

Com a memória cheia, o fluxo segue de q0 para q1, neste estado a máquina entra em loop, até receber uma transição positiva do sinal habilitador de leitura (`read_en = 1`), quando se direciona de q1 para q2. O dispositivo se mantém neste último estado esperando que o conteúdo de todas as posições de memória seja lido, esvaziando novamente a memória, apresentando o sinal `full` em nível baixo e `empty` em nível alto.

Vale ressaltar que esta lógica possui também um sinal de *clear*, independente do estado que o dispositivo se encontre ao receber uma transição positiva do sinal `all_clear`, ele se direciona para o estado q0, estado inicial, reiniciando o processo.

Outra maneira de se obter um código referente à memória FIFO necessária para a aquisição de dados é por meio dos aplicativos de geração de *IP-cores* encontrados nas ferramentas de desenvolvimento. Foi gerada por meio da ferramenta Quartus II um *IP-core* referente à memória FIFO, este módulo combinado com o módulo do analisador lógico forneceu os resultados esperados. Idealmente, um *IP-core* deve ser totalmente portátil, isto é, ser facilmente inserido em tecnologias de qualquer fornecedor ou em qualquer metodologia de projeto, mas estes códigos não foram aceitos nas demais ferramentas abordadas, ISE e Diamond. Este fato justifica a implementação deste código próprio para a memória FIFO, adequando-o a cada ferramenta de desenvolvimento trabalhada.

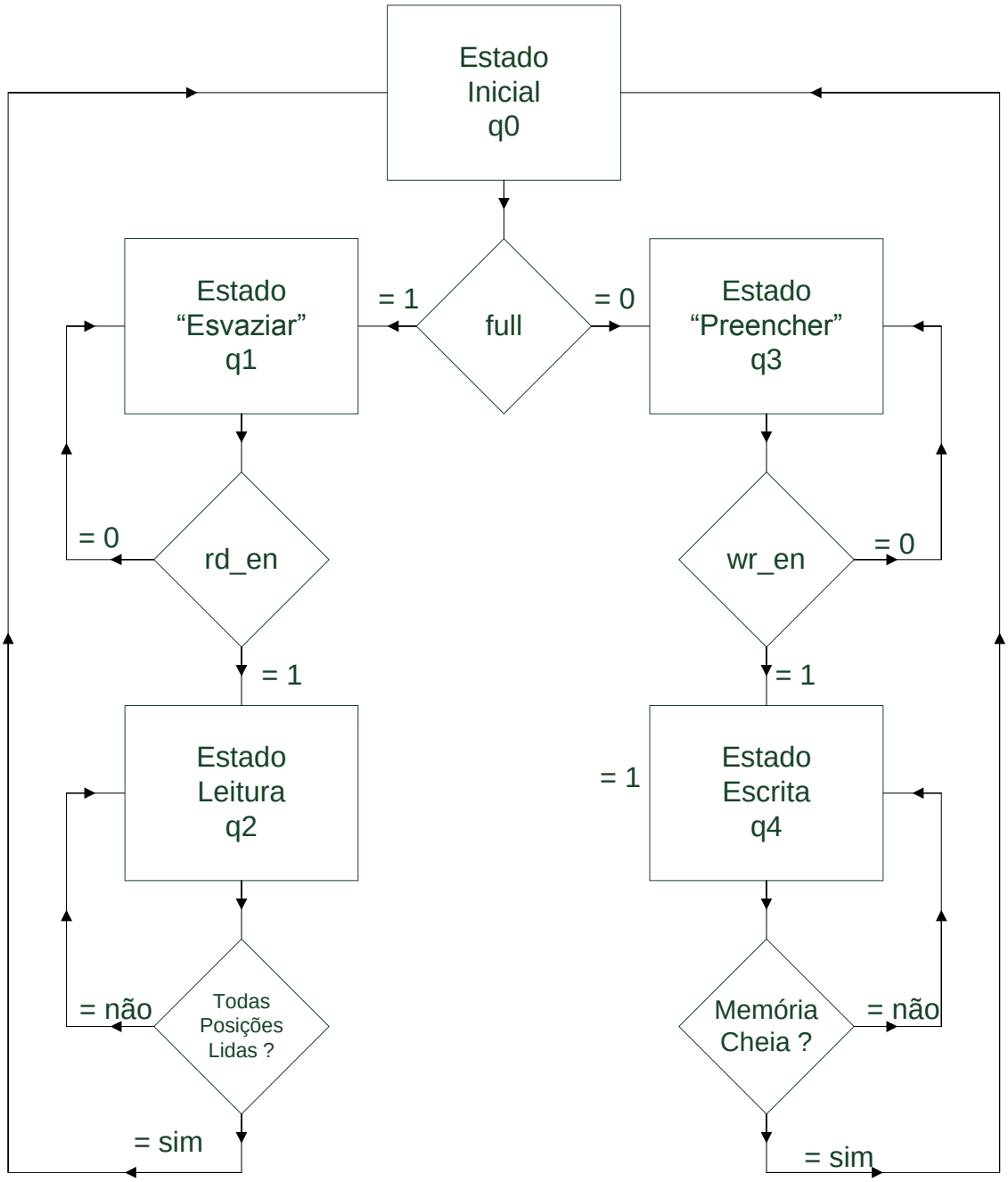

Figura 22 - Fluxograma Memória FIFO

3.4. Considerações Finais

Neste capítulo foi apresentado o projeto desenvolvido e as implementações realizadas, focando nas considerações adotadas para as mesmas.

CAPÍTULO 4: RESULTADOS OBTIDOS

4.1. Considerações Iniciais

Neste capítulo encontram-se as simulações dos *cores* implementados e a análise qualitativa e quantitativa dos três dispositivos. Consta também com a implementação física do projeto utilizando o kit da Altera fornecido pelo laboratório.

4.2. Simulações do Analisador Lógico

A fim de garantir o correto funcionamento dos módulos previamente implementados, foi feita a simulação dos mesmos nos três IDE's escolhidos, realizando as alterações de sintaxe exigidas a cada novo dispositivo de desenvolvimento. A princípio, os três *soft-cores* foram implementados e simulados utilizando o IDE Quartus II, e como exemplo de alterações na sintaxe inicial tem-se: forma da declaração do sinal de saída da memória FIFO ao utilizar o ISE, e forma de declaração dos módulos como componentes dentro de um código ao utilizar o Diamond.

Após ter garantido que a sintaxe dos mesmos fosse aceita para as três IDE's, simulou-se o projeto todo, adotando como parâmetros para o analisador lógico o máximo de quatro palavras de quatro bits de largura. Observam-se os resultados que se seguem, ressaltando que, para os três casos, o projeto foi simulado com suas duas opções de *trigger*, interno e em seguida externo.

A Figura 23apresenta a simulação realizada por meio do Quartus II e os sinais por ele apresentados. Para o primeiro caso, *trigger* interno, tem-se a indicação de cada um dos estados que o fluxo de dados percorre, das ocorrências da palavra definida como disparo, dos dados salvos e lidos da memória. Para o segundo caso, *trigger* externo, tem-se novamente a indicação dos estados percorridos, dados salvos e lidos da memória, e das ocorrências de pulsos externos de disparo. Como se pode observar, o fluxo dos dados obedece a lógica implementada do analisador lógico. O sinal `aclr` em nível alto mantém o processo no seu estado inicial, sendo direcionado para os estados S1 ou S2, dependendo do tipo de disparo. Estes disparos iniciam o processo de escrita, S3, encerrado após a memória ter adquirido a quantidade máxima de palavras, e de leitura, S4, encerrado após o último dado salvo na memória ser enviado para a saída. Vale ressaltar que antes de se iniciar o

estado de leitura, o fluxo de dados retorna ao estado inicial e em seguida ao estado de espera, aguardando pelo segundo sinal de disparo.

A Figura 24 apresenta a simulação realizada por meio do ISE, e os sinais por ele apresentados, com exceção do sinal `state`. Como se pode analisar, a resposta das simulações é praticamente a mesma resposta apresentada pela Figura 23, seguindo as mesmas variações do fluxo de dados. A única mudança considerável nesta simulação se dá pela resposta do analisado lógico ao sinal `aclr` em nível alto, forçando os sinais `qout`, `sel_t` e `trig` a assumirem valores desconhecidos.

A Figura 25 apresenta a simulação do projeto realizada por meio do Diamond, esta inclui além dos sinais apresentados pelo Quartus II, os sinais `rdreq`, `wrreq`, `empty` e `full`, sinais de controle da memória FIFO. As saídas apresentadas nesta simulação são as mesmas apresentadas nos dois casos anteriores. Da mesma forma que ocorreu com a simulação realizada por meio do ISE, enquanto o sinal `aclr` se mantém em nível alto, algumas saídas desta simulação assumem valores desconhecidos, no caso, os sinais `qout`, `rdreq` e `wreq`. Devido aos sinais de controle da memória FIFO, a Figura 25 apresenta outras quatro análises, a relação dos estados de escrita e leitura com os sinais `wrreq` e `rdreq`, respectivamente; e as transições dos sinais `empty` e `full`, representando o momento em que a memória começa a ser preenchida com dados e o momento em que a mesma inicia o envio destes dados para sua saída. Estas análises são indicadas por círculos e retas tracejadas, respectivamente.

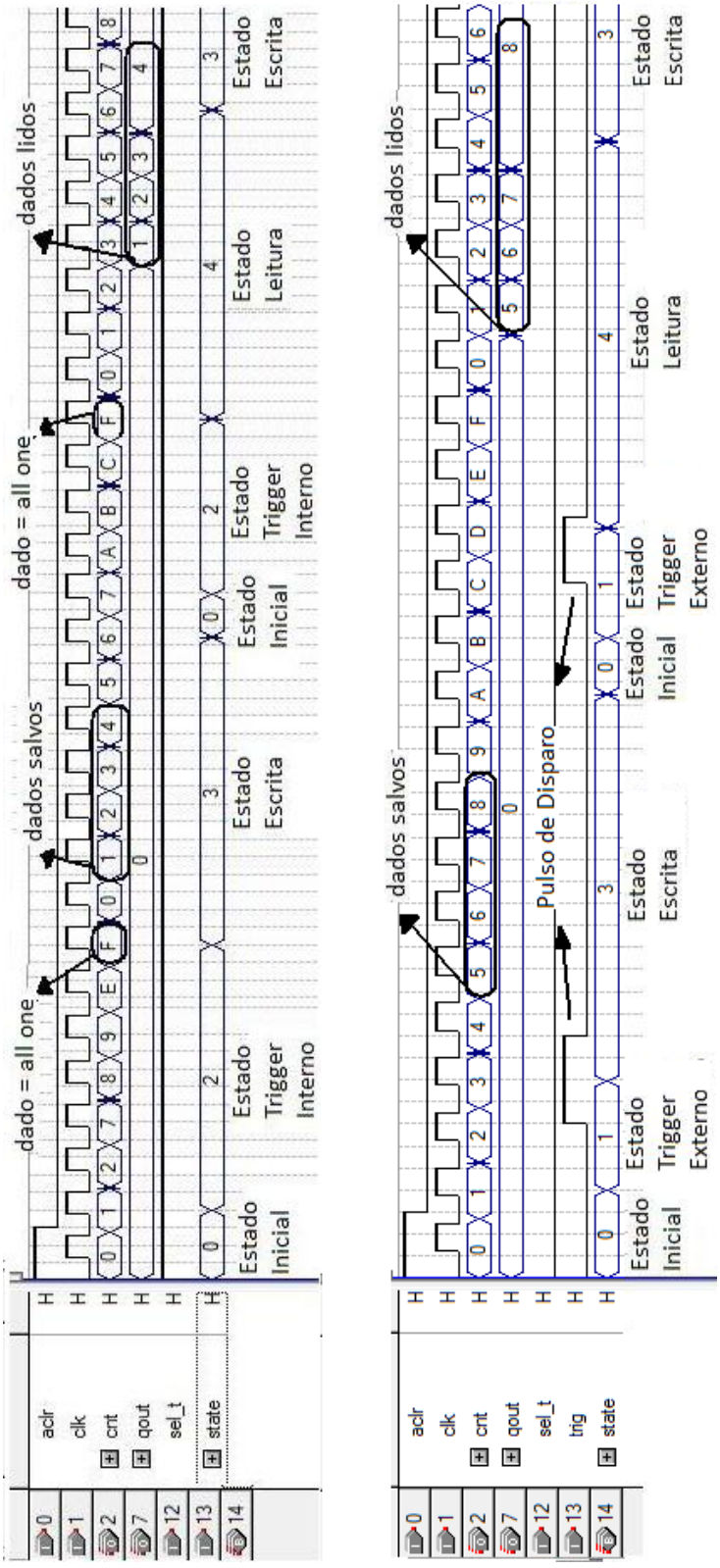

Figura 23 - Simulação do Projeto utilizando Quartus II (*Trigger* Interno e Externo)

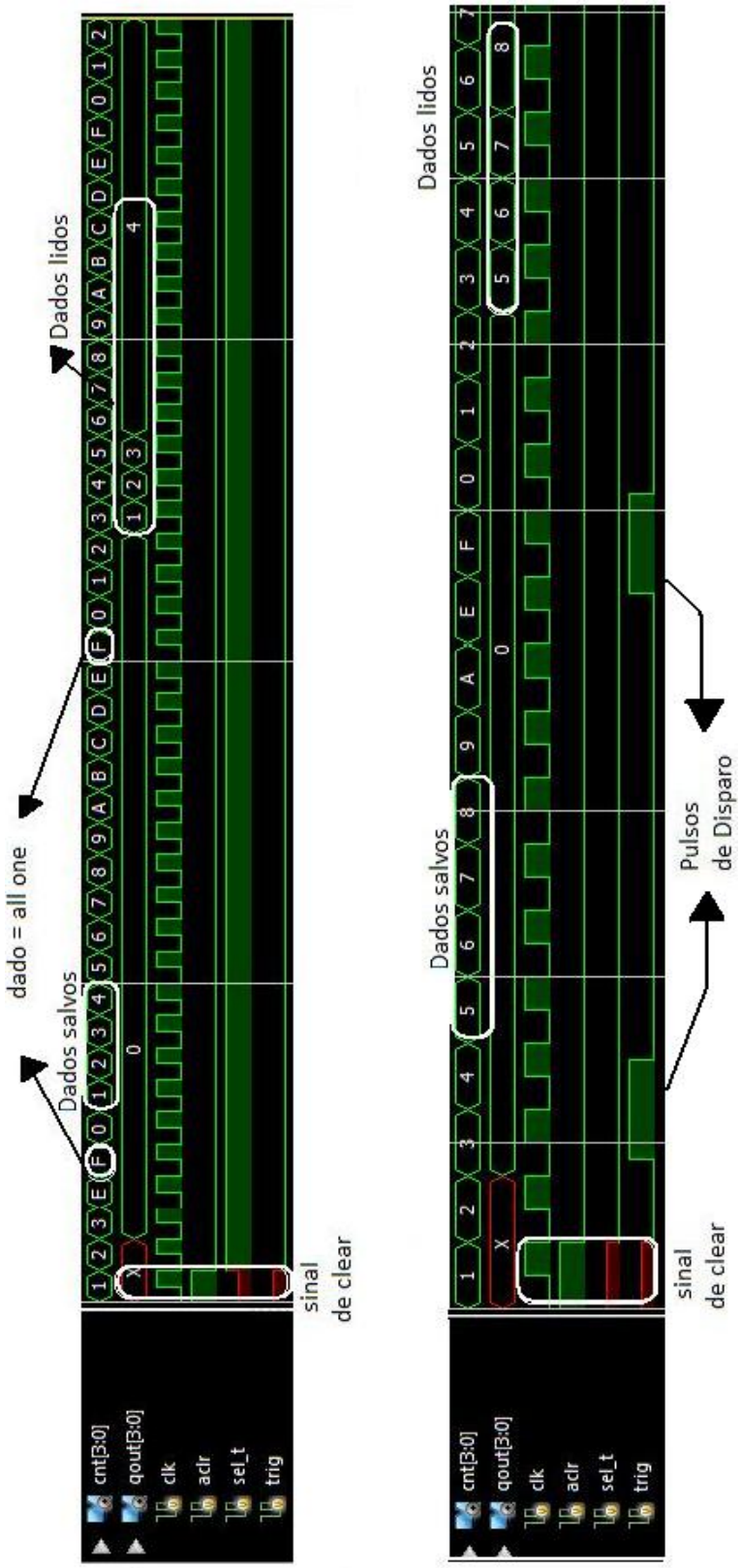

Figura 24 - Simulação do Projeto utilizando ISE (*Trigger* Interno e Externo)

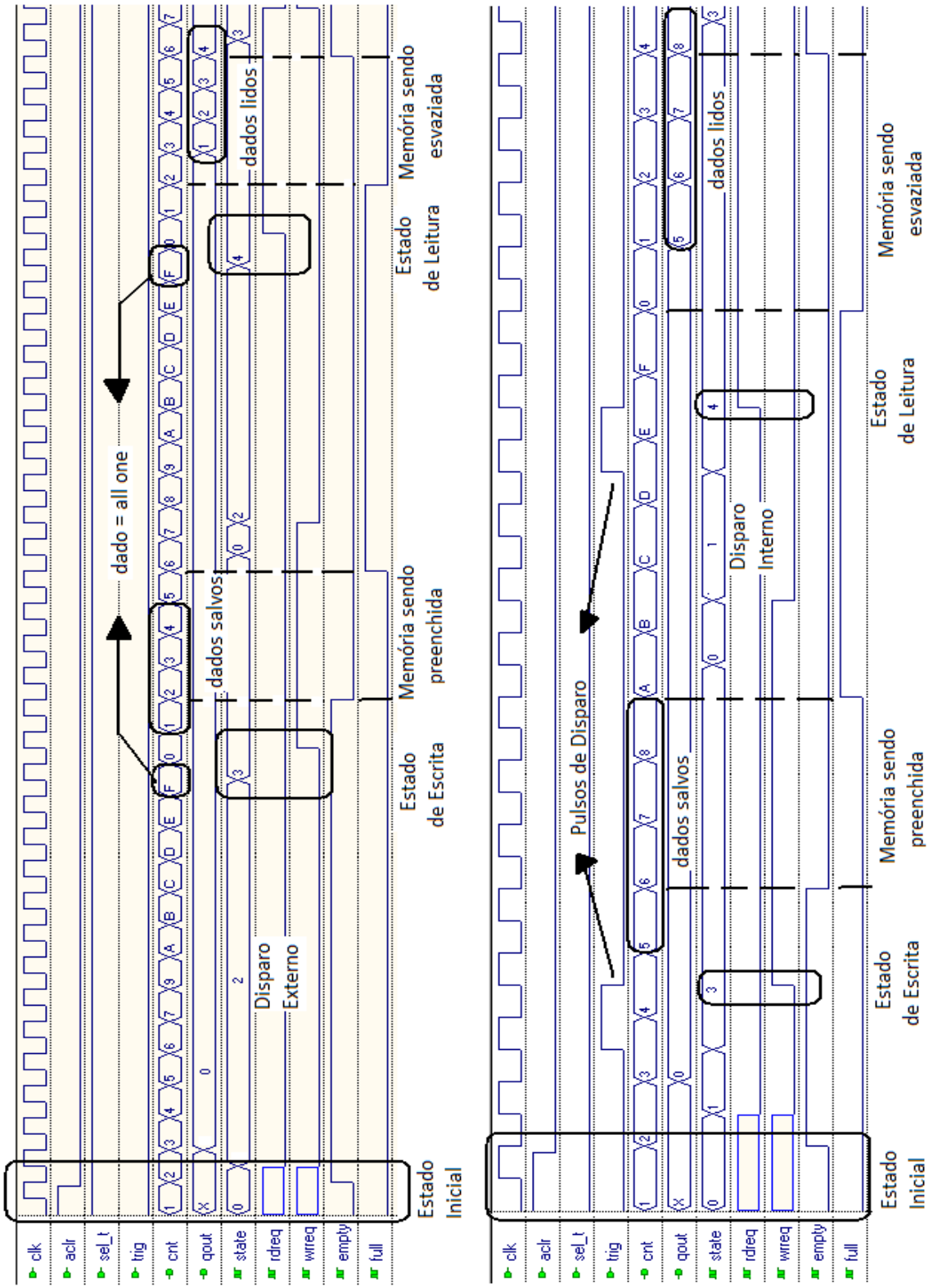

Figura 25 - Simulação do Projeto utilizando Diamond (*Trigger* Interno e Externo)

4.3. Teste Operacional Referente à Implementação Física

A implementação física dos módulos previamente simulados foi realizada no kit de desenvolvimento DK-CYCII-2C20N da empresa Altera, pertencente à família Cyclone II. O kit pode ser visto na Figura 24.

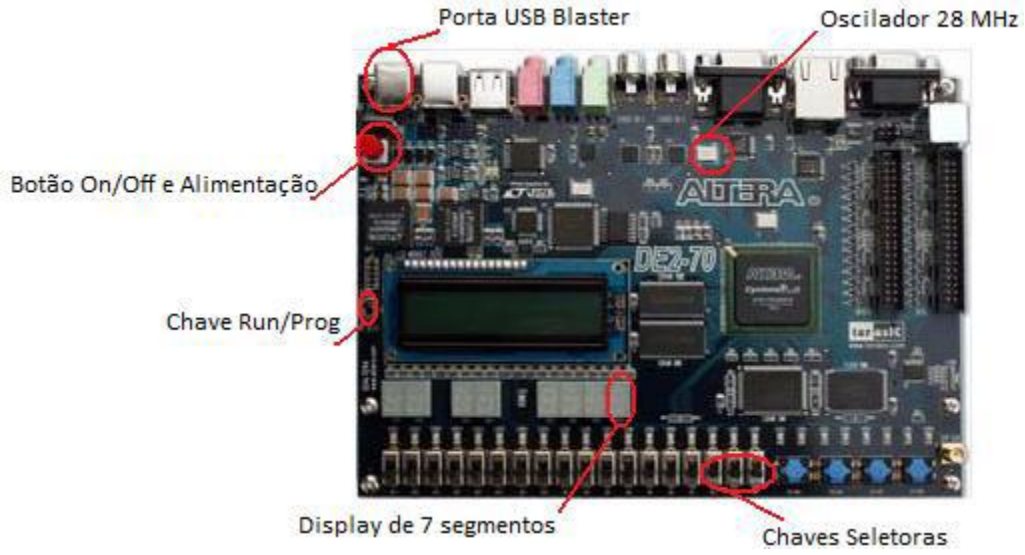

Figura 26 - Kit DK-CYCII-2C20N

Esta figura indica, dentre os vários componentes do kit, os utilizados nesta prática. Para a realização da mesma foi necessário alterar o código referente ao contador, relacionando as suas entradas `clk`, `trig`, `sel_trig` e `aclr` ao oscilador de 28 MHz e as chaves seletoras, respectivamente, e a sua saída `qout` ao display de 7 segmentos. O uso do display de 7 segmentos exigiu uma lógica de conversão do sinal advindo de `qout`.

Os demais itens indicados na Figura 26 representam os componentes responsáveis pela programação da FPGA existente no kit. O acionamento da placa é feita pelo botão ON/OFF, a conexão com o IDE se dá por meio da porta USB Blaster, e a chave Run/Prog habilita o envio das informações.

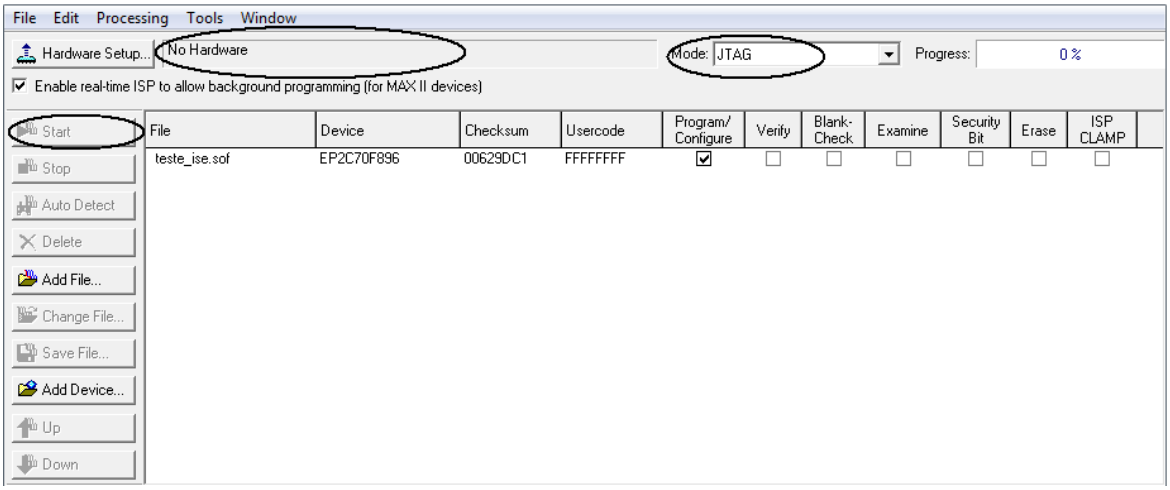

Figura 27 - Ferramenta Programmer

Após a configuração do kit para a programação da FPGA, posiciona-se a chave na opção Run e utiliza-se a ferramenta *Programmer* indicada pela **Figura 27**. As marcações representam os campos: *hardware*, responsável pela conexão (USB Blaster), modo de comunicação (JTAG) e início de gravação.

Primeiramente realizaram-se simulações com o código modificado, a fim de verificar se ele apresenta as mesmas respostas da Figura 21, em seguida, programou-se a FPGA com os módulos já abordados. Devido a alta freqüência do oscilador presente no kit não foi possível analisar com precisão o funcionamento do analisador lógico trabalhando na opção *trigger* interno, a saída apresentada pelo display de 7 segmentos se modificava constantemente. Em contra partida pode-se comprovar o funcionamento do dispositivo em sua opção de *trigger* externo, já que, adotando uma chave seletora como sinal de disparo, fica a cargo do usuário reiniciar o ciclo de captura de dados.

4.4. Análise do Processo de Síntese

Para aplicações reais do analisador lógico, o mesmo necessita de um maior número de bits de dados e de palavras, adotaram-se então os parâmetros: 1024 palavras de 8 bits. Ao tentar realizar a síntese do projeto com estes novos parâmetros obtiveram-se erros em todos os três casos. Estes erros se devem ao fato de que as três ferramentas de síntese utilizaram elementos lógicos ao invés de blocos de memória na implementação do *soft-core* referente à memória FIFO, sendo eles insuficientes para estes novos parâmetros.

A fim de solucionar este problema, iniciaram-se testes utilizando as memórias fornecidas pelas ferramentas geradoras de *IP-cores*, Mega Wizard Plug-In Manager do Quartus II, IP (Core Generator & Architecture Wizard) do ISE e IPexpress do Diamond. Estes códigos gerados, ao serem submetidos ao processo de simulação, apresentam o mesmo comportamento que o código implementado referente à memória FIFO e utilizam blocos de memória em suas sínteses. Possuem como desvantagem o fato de funcionarem, unicamente, com os IDE's dos seus próprios fabricantes.

Para realizar uma análise comparativa destes três processos de síntese, verificou-se os *reports* fornecidos pelos mesmos ao utilizar tanto o *soft-core* implementado quanto o *IP-core* gerado. Esta análise apresenta um alto nível de dificuldade devido às diferentes nomenclaturas adotadas por cada arquitetura. Para a melhor compreensão do que estes termos representam, sintetizaram-se ambos os tipos de memória com diferentes parâmetros, comparando item a item, além de estudar a arquitetura das FPGA's contidas nos dispositivos escolhidos. Vale ressaltar que a escolha dos dispositivos se deu pela disponibilidade em laboratório, sendo os três pertencentes à categoria *Low Cost FPGA*.

O *report* fornecido pelo IDE da empresa Altera, *Analysis & Synthesis Summary Reports*, possui, dentre as suas diversas informações, o número total de elementos lógicos, incluindo o total de funções combinacionais e de unidades lógicas; de registradores; e de bits de memória utilizados e disponíveis. Na Tabela 1 encontram-se os dados referentes aos elementos lógicos, registradores e bits de memória.

O *report* fornecido pelo ISE, *Synthesis Report*, possui uma forma diferente de abordagem. Analisa o uso de células no processo de síntese, dividindo-as entre *BELS*, elementos lógicos básicos como inversores, LUT's, *flip-flops*/*latches* e *buffers*. Adota-se, ao verificar todo o documento, que os *flip-flops*/*latches* são considerados como registradores, enquanto que os LUT's são considerados os elementos lógicos. Estas considerações se devem ao fato de que cada LE da arquitetura Cyclone II possui apenas uma LUT. Para analisar a utilização de blocos de memória, é importante verificar os *reports* gerados para o processo de *Map*. Na – Tabela 1 encontram-se os dados referentes às LUT's, aos registradores e aos blocos de memória. Os *reports* desta ferramenta definem como bloco de memória o conjunto de 18 Kbits de memórias.

O Diamond fornece o documento *Resource Usage Report*, que também indica por meio de LUT's e bits de registradores os itens a serem comparados. Da mesma forma que acontece com o ISE, utilizaram-se os *reports* gerados no processo de *Map* para a análise dos blocos de memória. Na Tabela 1 encontram-se os dados referentes às LUT's, aos bits de registradores e aos blocos de memória. Nota-se que neste caso o fator limitante é representado pelos *slices* do bloco lógico PFU e não as LUT's como nos demais, utilizando 480 *slices* de um total de 405 (119%) ao sintetizar o projeto com a memória implementada. Os reports desta ferramenta definem como bloco de memória o conjunto de 18 Kbits de memórias.

De posse dos *reports* devidamente analisados constrói-se a tabela 1. Nesta tabela são apresentados os dados referentes aos elementos reconfiguráveis e de memória utilizados na síntese da memória obtida por *soft-core* e da obtida pela ferramenta de geração de *IP-core*.

Tabela 1 – Elementos configuráveis e bits de memória utilizados na síntese de um *soft-core* implementado e de um *IP-core* gerado

	Software Família Dispositivo	Quartus II Cyclone II EP2C20F484C7	ISE Spartan3A(N) XC3S50A-5TQ144	Diamond LatticeXP2 LFXP2-5E-6TN144C
Memória Implementada	Elementos Configuráveis	Logic Element 23.078/18.752 (123%)	Slice (M + L) 7.917/704 (1124%)	Slice (PFU + PFF) 979/2.376 (41%)
Memória Gerada (IP_Core)	Elementos Configuráveis	Logic Element 79/18.752 (<1%)	Slice (M + L) 29/704 (4%)	Slice (PFU + PFF) 97/4.752 (4%)
	Bits de Memória	8.192/23.9616 (3%)	18.432/55.296 (33%)	18.432/165.888 (11%)

Para melhor entendimento desta tabela deve-se focar nas seguinte relações entre os elementos configuráveis de cada uma das três arquiteturas:

LAB = 16 LE’s = 16 * (LUT + registrador)

CLB = 4 Slices (M + L) = 4*(2 LUT’s + 2 registradores)

PFU/PFF = 4 Slices = 3*(2 LUT’s + 2 registradores) + 2LUT’s

4.5. Discussão dos resultados

Apesar do sucesso no funcionamento do analisador lógico, ou seja, os resultados obtidos nas simulações dos códigos implementados foram de acordo com o esperado, verificou-se a dificuldade de se obter um código genérico (*open-source*) compatível com os diversos fabricantes de FPGA’s. Isto se deve, principalmente, às características intrínsecas das mesmas, no caso deste projeto, exemplificadas pelos *IP-cores* e *reports* gerados, já que o uso de código gerado por uma ferramenta de desenvolvimento se mostrou altamente limitado.

A respeito do uso dos IDE’s conclui-se que o ISE exige maior esforço devido a sua interface não muito amigável, ainda que não se possa negar a grande gama de recursos por ele apresentado. O Diamond se mostra como a melhor opção, apresenta uma interface amigável, semelhante à fornecida pelo IDE da empresa Altera, além de inúmeros recursos tais quais os do ISE, mas de uso mais simples. Mesmo seu processo de simulação apresentando uma forma complicada de atribuição de valores aos sinais a serem analisados, ele é mais simples do que o processo que se utiliza de um *testbench.* Por sua vez, seu processo de síntese também se mostrou eficiente, visto que a divisão dos blocos lógicos em PFF e PFU permitiu que o seu fator limitante ultrapassasse apenas 19% do total permitido. Pelo fato deste IDE ser um produto recente da empresa Lattice Semiconductor, estando em sua primeira versão, acredita-se que o mesmo sofrerá diversas atualizações em um futuro próximo, tornando-se um forte competidor neste mercado, sendo válido acompanhar o seu desenvolvimento.

4.6. Dificuldades e Limitações

No decorrer do desenvolvimento deste projeto foram encontradas algumas dificuldades relacionadas ao uso e instalação dos *software*s abordados e à análise dos *reports* fornecidos pelos mesmos.

Para a instalação do ISE Design Suite foi necessário realizar alterações na configuração dos computadores utilizados no desenvolvimento deste projeto, além de alterar algumas configurações especificas do próprio *software*. Para a simulação do *IP-core* gerado pelo aplicativo *IPexpress*, do IDE Diamond, foi necessário acrescentar duas linhas de código no mesmo. Em todos os casos foi de grande ajuda pesquisar possíveis soluções para estes problemas em fóruns onlines. Com relação à análise dos *reports*, esta dificuldade foi descrita anteriormente.

Além destas dificuldades, este projeto é limitado devido às próprias FPGA's, suas diferentes arquiteturas, etc. Esta limitação se deve também a incompatibilidade entre os *IP-cores* gerados pelas suas ferramentas de desenvolvimento e aos processos de síntese.

4.7. Considerações Finais

Neste capítulo foram apresentados os principais resultados obtidos. Os resultados mostraram que, mesmo com a evolução das aplicações envolvendo FPGA's, o desenvolvimento do projeto foi limitado pelas características intrínsecas das mesmas e pelas técnicas de síntese de seus dispositivos.

No próximo capítulo será apresentada a conclusão deste trabalho, além de contribuições desse projeto, considerações sobre o curso de graduação, e também possíveis trabalhos futuros em relação a esse projeto.

CAPÍTULO 5: CONCLUSÃO

5.1. Contribuições

A utilização de diferentes ferramentas de desenvolvimento é uma prática incomum, normalmente o projetista utiliza tecnologias – FPGA e IDE – de apenas um fabricante. Logo este documento fornece várias informações a respeito destes *software*s e aplicativos que podem facilitar o trabalho de outros projetistas inexperientes em determinada tecnologia. Este projeto permite também, por meio de suas análises, escolher uma tecnologia específica para determinada aplicação.

Num âmbito pessoal, o projeto contribuiu para que o autor pudesse se aprofundar na teoria dos dispositivos reconfiguráveis, FPGA especificamente. A maior parte da teoria apresentada neste documento foi abordada de forma superficial durante a graduação, e pertence à área de eletrônica digital, área de grande evolução atualmente. Com base no projeto, o autor foi capaz de desenvolver outro trabalho relacionado ao tema, o artigo "Análise Comparativa e Qualitativa de Ferramentas de Desenvolvimento de FPGA's", publicado no *VII Southern Programmable Logic Conference*, categoria *Designer Forum*.

5.2. Trabalhos Futuros

Ao completar este projeto observa-se que o mesmo pode ser continuado a fim de que o analisador lógico possa ter seus sinais apresentados externamente, utilizando recursos como JTAG ou mesmo comunicação I2C. Em posse de uma interface de fácil uso, como um display de LCD, os sinais amostrados de um código exemplo, como o contador neste projeto, podem ser visualizados de maneira direta por qualquer usuário, sem a necessidade de uma ferramenta de simulação. Logo, os desafios se baseiam na maneira de adquirir estes sinais, enviá-los por meio de um barramento e amostrá-los corretamente em alguma interface.

Outro aspecto que pode ser melhor trabalhado em projetos futuros é a elaboração de módulos referentes a memórias que possam ser sintetizados utilizando blocos de memória ao invés de elementos lógicos, maior fator limitante neste projeto.

REFERÊNCIAS

ALTERA, *Cyclone II Device Handbook, Volume 1*. CII5V1-3.3, 2008. Disponível em: *www.altera.com/literature/hb/cyc2/cyc2_cii5v1.pdf.*

AMORE, R. VHDL *Descrição e Síntese de Circuitos Digitais*. LTC Editora , 2005.

BARR, M. *Programmable Logic: What's it to Ya?,* Embedded Systems Programming, pp. 75-84, June, 1999.

BROWN, S; ROSE, J. *Architecture of FPGAs and CPLDs: A Tutorial, IEEE Design and Test of Computers*, Vol. 13, No. 2, pp. 42-57, 1996.

CARRO, L.; WAGNER, F. R. *Sistemas Computacionais Embarcados*. In: JAI'03 – XXII Jornadas de Atualização em Informática, 2003, Campinas.

ESKINAZI, R. LIMA, M. E. NASCIMENTO, P. S. e GUILHERMINO, A. *FPGAS dinamicamente reconfiguráveis: fluxo de projeto e vantagens na concepção de circuitos integrados*, Congresso Brasileiro de Tecnologia – CONBRATEC, 2005

GUPTA, R; ZORIAN, Yervant. *Introduction to core-based design*. IEEE Design and Test of Computers. 1997

HARTENSTEIN, R. *A Decade of Reconfigurable Computing:A Visionary Retrospective.* In: *Proceedings of the Conference on Design, Automation and Test in Europe,* p. 642-649, Piscataway, NJ, USA. IEEE Press, 2001.

LATTICE Semiconductor. *Lattice XP2 Family Handbook.* HB1004 Version 02.5, 2010. Disponível em: *http://www.latticesemi.com/documents/HB1004.pdf.*

MARTINS, C. A. P. S; ORDONEZ, E. D. M; CORRÊA, J. B. T; e CARVALHO, M. B. *Computação Reconfigurável: Conceitos, Tendências e Aplicações.* In: *Jornada de Atualização em Informática (JAI, 2003)*,p. 339-388, Campinas, SP, Brasil, 2003.

MORAES, F., MESQUITA, D. *Tendências em Reconfiguração dinâmica de FPGAs.* SCR'2001 - Seminário de Computação Reconfigurável. Belo Horizonte, MG, Setembro, 2001.

MORAES, F; CALAZANS, N; MOLLER, L; BRIAO, E; CARVALHO, E. *Dynamic and Partial Reconfiguration in FPGA SoCs:* Requirements Tools and a Case Study. In: ROSENSTIEL, Wolfgang. Reconfigurable Computing. New York, USA. 2004.

PONTES, D.F; MADEIRA, C.S; *Aplicação Da Linguagem Descritiva De Hardware No Ensino De Circuitos Digitais*, I Congresso de Pesquisa e Inovação da Rede Norte Nordeste de Educação Tecnológica. Natal-RN - 2006.

RIBEIRO, C. H. *Projeto de um Analisador Lógico Baseado em PC.* Dissertação de Mestrado no Ita, 1989.

SILVA, L.C; *Aplicação de Computador Pessoal como Equipamento de Bancada Multifuncional de Baixo Custo para Laboratórios de Eletrônica e Microprocessadores*. Dissertação de Mestrado: Universidade Federal de Itajubá. Itajubá, 2002.

SOUZA, A.R.C; *Desenvolvimento e Implementação em FPGA de um Sistema Portátil para Aquisição e Compressão sem Perda de Eletrocardiogramas.* Dissertação de Pós-Graduação: Universidade Federal do Paraíba. João Pessoa, 2008.

TALA, D. K, *Verilog Tutorial*. Disponível em: *www.asic-world.com*

XILINX. *Spartan-3AN FPGA FamilyData Sheet* DS557-1 (v3.2), 2009. Disponível em: *www.xilinx.com/support/documentation/data_sheets/ds557.pdf*.

APÊNDICE A – Código Contador

```
//Gabriel Santos da Silva
//Código - Contador(exemplo de uso do analisador lógico)

module count(clk, aclr, sel_t, trig, cnt, qout);

//parâmetros do analisador lógico (também da memória FIFO)
parameter largura_dado = 4;
parameter numeros_dado = 8;

//sinais de entrada e saída
input clk;
input aclr;
input sel_t;
input trig;
wire [2:0]state;
wire rdreq;
wire wrreq;
wire empty;
wire full;
wire [largura_dado-1:0]q;
output reg [largura_dado-1:0]cnt;
output reg [largura_dado-1:0]qout;

//analisador lógico
// "módulo do analisador lógico"
// #(largura do dado, número de palavras)"nome"(entradas e saídas)
anlog#(largura_dado,numeros_dado)
rec(clk,cnt,q,aclr,sel_t,trig,state,rdreq,wrreq,empty,full);

//incialização das variáveis
initial
cnt = 0;

//lógica do contador
always @(negedge clk) begin
cnt = cnt + 1; //a cada descida de clock acrescenta um ao contador
qout = q;
end

endmodule
```

APÊNDICE B – Analisador Lógico

```
//Gabriel Santos da Silva
//Código - Analisador Lógico

//declaração de módulo com state, empty1 e full1 para analise de
simulação
module
anlog(clk,data1,q2,aclr1,sel_t,trig,state,rdreq1,wrreq1,empty1,full1);

//parâmetros da memória FIFO
parameter data_width = 3;
parameter num_palavras = 6;

//sinais das entradas e saídas
input clk;
input sel_t;
input trig;
input [data_width-1:0]data1;
input aclr1;
output [data_width-1:0]q2;
output [2:0]state;
output empty1;
output full1;
output reg rdreq1;
output reg wrreq1;

//sinais auxiliares
reg [data_width-1:0] flag;

//máquina de estados
reg [2:0]state;
parameter s0 = 0, s1 = 1, s2 = 2, s3 = 3, s4 = 4;

//memória FIFO
//"módulo da memória"
//#(largura do dado, número de palavras)
//"nome"(entradas e saídas)
mem_fifo#(data_width,num_palavras)
mem(clk,aclr1,data1,rdreq1,wrreq1,full1,empty1,q2);

//lógica do analisador
always @(posedge clk or posedge aclr1)
begin
if (aclr1 == 1)begin //aclr além de reiniciar a gravação na memória
também retorna ao estado 0, ideal
state = s0; //sempre que iniciar o programa ele enviar um único pulso de
inicialização.
end
else begin
case (state)
s0: begin //estado de inicialização
rdreq1 = 0;
wrreq1 = 0;
if (sel_t == 0) //seleciona entre trigger externo ou interno
state = s1;
```

```
else
state = s2;
end
s1: begin //estado de trigger externo
if (trig == 1'b1) // se o trigger receber algum sinal externo inicia o
processo
if (full1 == 1) //caso já tenha ocorrido o ciclo de escrita
state = s4;
else //caso ainda nao tenha ocorrido o ciclo de escrita
state = s3;
else
state = s1; //loop de espera pelo sinal externo de trigger
end
s2: begin //estado de trigger interno
flag = data1 + 1'b1; //flag para auxílio da palavra de trigger devido
variação do seu tamanho
if (flag == 0) // palavra de triger compatível
if (full1 == 1) //caso já tenha ocorrido o ciclo de escrita
state = s4;
else
state = s3; //se a palavra for a de trigger inicia o processo
else
state = s2; //loop de espera pela palavra de trigger compatível
end
s3: begin //estado de escrita na memória
rdreq1 = 0;
wrreq1 = 1; //solicitação de escrita
if (full1 == 1) // memória atinge seu nível máximo
state = s0;
else //enquanto a memória não estiver preenchida mantém os mesmos pedidos
e gravando
state = s3;
end
s4: begin //estado de leitura da memória
rdreq1 = 1; //solicitação de leitura
wrreq1 = 0;
if (empty1 == 1) //memória vazia
state = s3;
else //enquanto não estiver completamente vazia mantém os pedidos e lendo
state = s4;
end
endcase
end
end
endmodule
```

APÊNDICE C – Memória FIFO

```
// Gabriel Santos da Silva
// Código - Memória FIFO

//OBS: todos os parametros, entradas e saídas são os encontrados em uma
memória FIFO síncrona, tanto do QUARTUS quanto do ISE.

module mem_fifo_ise(clk, all_clr, data, rd_en, wr_en, full, empty,q);

//parâmetros da memória
parameter width = 4;
parameter num_word = 7;

//sinais de entrada e saída
input clk;
input all_clr;
input [width-1:0]data;
input rd_en;
input wr_en;
output reg full;
output reg empty;
output reg [width-1:0]q;

//sinais auxiliares
//[a:0]vetor[b:0] - a indica a posição do vetor e b o vetor dentro do
//vetor de vetores
reg [width-1:0]mem[num_word-1:0];
integer i;

//máquina de estados
reg [2:0]state;
parameter s_inicial = 0, s_write = 1, s_read = 2, s_clear = 3,
s_preencher = 4, s_esvaziar = 5;

//inicialização das variáveis
initial begin
full = 0;
empty = 0;
i = 0;
state = s_inicial;
end

//lógica da memória FIFO
always @(negedge clk or posedge all_clr) begin
if (all_clr == 1)
state = s_clear;
else begin
case (state)
s_inicial: begin //estado para verificar se a memória já está cheia ou
não (obs: vale ressaltar que para efeito de testes, a memória se
inicializará sempre vazia)
if(full == 0)
state = s_preencher;
else
state = s_esvaziar;
```

```
end
s_preencher: begin //estado apenas para espera do write enable.
if (wr_en == 1)
state = s_write;
else
state = s_preencher;
end
s_write: begin //estado para escrita na memória. Ao inicializar a escrita
ela continuará até preencher por completo a memória
empty = 0;
if (i !== num_word) begin
mem[i]=data;
i = i + 1;
state = s_write;
end
else begin
full=1;
state = s_inicial;
end
end
s_esvaziar: begin //estado apenas para espera do read enable.
if (rd_en == 1)
state = s_read;
else
state = s_esvaziar;
end
s_read: begin //estado para leitura da memória. Ao inicializar a leitura,
ela continuará até ler todos os dados da memória
full = 0;
if (i !== 0) begin
i = i-1;
q = mem[num_word-1-i];
state = s_read;
end
else begin
empty = 1;
state = s_inicial;
end
end
s_clear: begin //estado de reset da memória, tem q ser testado.
i = 0;
full = 0;
empty = 1;
q = 0;
while (i !== num_word) begin //preenche a memória com 0's
mem[i] = 0;
i = i+1;
end
i=0;
state = s_inicial;
end
endcase
end
end
endmodule
```

APÊNDICE D – *Testbench* (Simulação ISE)

```
//Gabriel Santos da Silva
//Código - Testebench para o contador (utilizando o analisador e a
//memória como componentes)

module sim_ise;

//sinais de entrada
reg clk;
reg aclr;
reg sel_t;
reg trig;

//sinais de saída
wire [3:0] cnt;
wire [3:0] qout;

// Instantiate the Unit Under Test (UUT)
//módulo da Unidade em Teste (UUT)
counter_ise uut (
.clk(clk),
.aclr(aclr),
.sel_t(sel_t),
.trig(trig),
.cnt(cnt),
.qout(qout)
);

//período do clock para simulação
always
#10 clk = ~clk;
initial begin

//inicialização das variáveis
clk = 0;
aclr = 1;
#20 aclr = ~aclr;
trig = 0;

//trigger interno
sel_t = 1;

//trigger externo
/*sel_t = 0;

#30 trig = ~trig;
#20 trig = ~trig;
#150 trig = ~trig;
#20 trig = ~trig;
#300 trig = ~trig;
#20 trig = ~trig;*/
end
endmodule
```

APÊNDICE E –Código Contador Modificado

```
// Gabriel Santos da Silva
// Código - Contador(exemplo de uso do analisador lógico)

module counter_ise(iCLK_28,cnt, qout,state,oHEX0_D, iSW);

//parâmetros do analisador lógico (também da memória FIFO)
parameter largura_dado = 4;
parameter numeros_dado = 4;

//sinais de entrada e saída
input iCLK_28; //atribuição a pinagem do kit
input [3:0]iSW; //atribuição a pinagem do kit
output [2:0]state;
wire rdreq;
wire wrreq;
wire empty;
wire full;
output reg [largura_dado-1:0]cnt;

//analisador lógico
// "módulo do analisador lógico" #(largura do dado, número de
palavras)"nome"(entradas e saídas)
anlog_ise
#(largura_dado,numeros_dado)rec(iCLK_28,cnt,q,aclr,sel_t,trig,state,rdreq
,wrreq,empty,full);

//incialização das variáveis
initial cnt = 0;

//lógica do contador
always @(negedge iCLK_28) begin
cnt = cnt + 1; //a cada descida de clock acrescenta um ao contador
qout = q;
sel_t = iSW[0];//associação dos sinais de entrada com sinais externos
advindos do kit
trig = iSW[1];
aclr = iSW[2];

case (qout) //lógica de conversão para o display de 7 segmentos
4'h0: oHEX0_D = 7'b1000000;
4'h1: oHEX0_D = 7'b1111001;
4'h2: oHEX0_D = 7'b0100100;
4'h3: oHEX0_D = 7'b0110000;
4'h4: oHEX0_D = 7'b0011001;
4'h5: oHEX0_D = 7'b0010010;
4'h6: oHEX0_D = 7'b0000010;
4'h7: oHEX0_D = 7'b1111000;
4'h8: oHEX0_D = 7'b0000000;
4'h9: oHEX0_D = 7'b0011000;
4'hA: oHEX0_D = 7'b0001000;
4'hB: oHEX0_D = 7'b0000011;
4'hC: oHEX0_D = 7'b1000110;
4'hD: oHEX0_D = 7'b0100001;
4'hE: oHEX0_D = 7'b0000110;
```

```
4'hF: oHEX0_D = 7'b0001110;
endcase
end
endmodule
```