**CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS**
**CAMPUS TIMÓTEO**

Elias Luiz da Silva Júnior

# O PROCESSADOR LEON3 COMO PLATAFORMA DE ESTUDOS SOBRE ARQUITETURA DE COMPUTADORES

**Timóteo**

**2019**

Elias Luiz da Silva Júnior

# O PROCESSADOR LEON3 COMO PLATAFORMA DE ESTUDOS SOBRE ARQUITETURA DE COMPUTADORES

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia de Computação do Centro Federal de Educação Tecnológica de Minas Gerais, campus Timóteo, como requisito parcial para obtenção do título de Engenheiro de Computação.

Trabalho aprovado. Timóteo, 09 de agosto de 2019

Prof. Dr. Bruno Rodrigues Silva
Orientador

Prof. Dr. Elder de Oliveira Rodrigues
Professor Convidado

Prof. Dra. Viviane Cota Silva
Professor Convidado

Timóteo
2019

Dedico a meus familiares e amigos
que me apoiaram durante essa longa jornada.

# Agradecimentos

Primeiramente gostaria de agradecer a meu orientador, Bruno Rodrigues Silva, por ter permanecido firme nessa empreitada apesar de todas as turbulências ocorridas neste período de tempo, tanto no desenvolvimento deste trabalho quanto em nossas vidas pessoais.

Agradeço também a meus pais que estiveram sempre presentes, mesmo quando distantes. A fé em minha capacidade e o apoio incondicional foram chave para que este trabalho pudesse ser concluído.

Por último mas não menos importante agradeço a meus amigos por toda a ajuda prestada em momentos chave do desenvolvimento deste trabalho e pela motivação constante para conseguir terminá-lo.

*“Mantenha seus olhos nas estrelas*
*e seus pés no chão”.*
*Theodore Roosevelt*

# Resumo

A criação de hardware especializado para uma determinada tarefa permite que aplicações antes inviáveis possam se tornar realidade, como pode ser notado pela ampla utilização de sistemas embarcados hoje em dia. Porém o processo para a criação de um chip personalizado é custoso, reduzindo o estímulo para tais empreitadas. Com o uso da tecnologia FPGA, é possível não só realizar a prototipação de hardware a baixo custo como também, para aplicações menos exigentes, realizar uma produção final com custo de entrada reduzido. Este trabalho consiste de um guia contendo informações básicas que possibilitam o desenvolvedor inexperiente com o processo de prototipação de hardware seja introduzido a projetos com tal característica em um processo de aprendizado gradual. Após levantamento bibliográfico envolvendo conteúdos teóricos sobre arquitetura de computadores e manuais práticos sobre as tecnologias utilizadas, foi realizada uma compilação com conhecimentos necessários para operar em um projeto de hardware customizado. Como plataforma para o desenvolvimento de processadores foi utilizado o LEON3, um processador de código aberto em VHDL. Foi então realizada uma análise descritiva sobre as ferramentas e técnicas aplicadas no desenvolvimento de hardware para um dispositivo Altera da família Cyclone II, desde conhecimentos gerais sobre arquitetura de computação e circuitos digitais a procedimentos para realização de determinadas tarefas utilizando os softwares necessários para tal projeto. Espera-se que este trabalho permita a entrada de mais pesquisadores na área de desenvolvimento de hardware a fim de permitir uma maior gama de aplicações de computação em processos especializados.

**Palavras-chave**: prototipação de hardware, FPGA, VHDL.

# Abstract

The creation of specialized hardware to a certain task allows applications before unfeasible can become real, as can be noted by the large utilization of embedded system nowadays. However the process for creating a custom chip is expensive, reducing the incentives to such enterprises. Using the FPGA technology, it is possible to not only prototype hardware at a low cost as also, for less demanding applications, make the final product at a lower entry value. This work consists of a guide containing basic informations that allow a developer new to the process of hardware prototyping to be introduced to projects that have such characteristic within a smooth learning process. After researching the literature regarding theoretical content about computer architecture and practical manuals about the employed technologies, a compilation with the necessary knowledge to operate in a custom hardware project was made. The LEON3, a open-source processor with code in VHDL, was used as an development platform. It was then made a descriptive analysis of the tool and techniques employed on the development of hardware to a device for the Altera Cyclone II family, from the general knowledge about computer architecture and digital circuitos to procedures to do certain tasks on the sofwares required for this project. It is expected that this work allow more researches to enter the area of hardware development so a larger range of computational applications in specialized processes can be done.

**Keywords**: hardware prototyping, FPGA, VHDL.

# Lista de ilustrações

# Lista de tabelas

# Lista de abreviaturas e siglas

| | |
|---|---|
| AMBA | Arquitetura avançada de barramento de microcontrolador (*Advanced Microcontroller Bus Architecture*) |
| AHB | Barramento avançado de alto desempenho (*Advanced High-performance Bus*) |
| APB | Barramento avançado de periféricos (*Advanced Peripheral Bus*) |
| ASIC | Circuitos integrados para aplicação específica (*Application Specific Integrated Circuits*) |
| CAD | Projeto auxiliado por computador (*Computer Aid Design*) |
| CI | Circuito integrado |
| CPLD | Dispositivo de lógica programável complexa (*Complex Programmable Logic Device*) |
| CPI | Clock por instrução |
| CPU | Unidade central de processamento (*Central Processing Unit*) |
| FPGA | Arranjo de portas programáveis em campo (*Field-Programmable Gate Array*) |
| GPL | Licença pública geral GNU (*GNU General Public License*) |
| GRFPU | Unidade de ponto-flutuante da Gaisler Research (*Gaisler Research Floating-Point Unit*) |
| GRLIB | Bilioteca da Gaisler Research (*Gaisler Research Library*) |
| HDL | Linguagem de descrição de hardware (*Hardware Description Language*) |
| IEEE | Instituto de Engenheiros Eletricistas e Eletrônicos |
| ISA | Conjunto de instruções da arquitetura (*Instruction Set Architecture*) |
| JTAG | Grupo de ação de teste conjunto (*Joint Test Action Group*) |
| LAB | Blocos de arranjo lógico (*Logic Array Blocks*) |
| LE | Elemento lógico (*Logic Element*) |
| LUT | Tabela de resultados (*Lookup Table*) |
| MMU | Unidade de gerenciamento de memória (*Memory Management Unit*) |
| RAR | Leitura após leitura (*Read after read*) |

| | |
|---|---|
| RAW | Leitura após escrita (*Read after write*) |
| RISC | Conjunto reduzido de instruções de computador (*Reduced Instruction Set Computer*) |
| RTL | Nível de transferência de registrador (*Register-Transfer Level*) |
| SDF | Formato padrão de atrasos (*Standard Delay Format*) |
| USB | Barramento serial universal (*Universal Serial Bus*) |
| VHDL | Linguagem de descrição de hardware VHSIC (*VHSIC Hardware Description Language*) |
| VHSIC | Circuito integrado de velocidades muito altas (*Very High Speed Integrated Circuit*) |
| WAR | Escrita após leitura (*Write after read*) |
| WAW | Escrita após escrita (*Write after write*) |

# Sumário

# 1 Introdução

*"Engenheiros gostam de solucionar problemas.*
*Se não houverem problemas facilmente acessíveis, eles criaram os seus próprios."*[1] .
*Scott Adams (Tradução nossa)*

No mundo de hoje, estamos cercados por circuitos lógicos a todo instante. Nos computadores, celulares, eletrodomésticos e até coisas que não notamos como, controladores de semáforos. Essa quase onipresença de circuitos digitais se deve ao poder dessa abstração, capaz de representar e processar qualquer tipo de informação utilizando sinais elétricos em condições restritas (AGARWAL; LANG, 2005). A diversidade de possíveis aplicações faz com que haja grande interesse em pesquisas direta ou indiretamente relacionadas com a produção de circuitos lógicos digitais.

Por exemplo, a automatização crescente de tarefas que podem ser descritas por algoritmos é uma área relacionada com amplo potencial que tem sido explorado por pesquisadores. Os avanços tecnológicos permitem hoje que sistemas embarcados estejam disponíveis a um custo muito mais acessível. Sendo assim, é possível que até mesmo pessoas físicas automatizem tarefas do dia a dia, seja em busca de eficiência ou comodidade. Essa forma nova de aplicar a computação em áreas não convencionais tem se tornado um próprio ramo dentro do estudo da computação, automação e circuitos: a chamada computação ubíqua, que quando possui um foco maior em tecnologias embarcadas é conhecida como internet das coisas.

Uma outra área de pesquisa que também se interessa na eficácia de circuitos lógicos digitais é a arquitetura de computadores. Por mais de 50 anos a Lei de Moore se manteve consistente com os produtos no mercado, observando um desempenho aproximadamente duas vezes maior a cada 18 meses corridos. Porém, limites físicos e de materiais fazem com que esse crescimento exponencial não possa prosseguir infinitamente (MACK, 2011). Sendo assim, o projetista de unidades de processamento não pode apenas confiar nos avanços na tecnologia de transistores para conseguir um desempenho capaz de se adequar às demandas modernas de processar rapidamente bases de dados cada vez maiores. A pesquisa para aperfeiçoamento do projeto do hardware é uma pedra fundamental no caminho do suprimento dessa demanda.

Para que a pesquisa possa ser realizada de forma efetiva, é necessário que seja possível o teste e a medição de performance de uma dada hipótese. Porém a produção de hardware personalizado pode se mostrar custosa. Uma tecnologia que torna acessível a criação de hardware customizado em um chip e, portanto, os testes em uma unidade real implementando a hipótese em voga são os chips de FPGA, do inglês *Field-Programmable Gate Array* ou Arranjo de Portas Programáveis em Campo. O FPGA é uma tecnologia que utiliza diversas unidades de memória de poucos bits e circuitos multiplexadores para simular a lógica nele programada

[1] *"Engineers like to solve problems. If there are no problems handily available, they will create their own problems."*

em um chip físico (MAXFIELD, 2004).

Tendo isto em mente, o FPGA pode se mostrar uma poderosa ferramenta para àqueles interessados em realizar pesquisas relacionadas a circuitos digitais e também a todos que buscam uma forma mais barata de aplicar esses conhecimentos a problemas do mundo real.

## 1.1 Justificativa

Durante o processo de aprendizado de um tema novo, o estudante passa por um processo de aumento gradual da complexidade do objeto de estudo. Apesar de nem sempre ser o caso, esse é o processo que tende a ser buscado naturalmente durante o aprendizado. O assunto tem então sua complexidade reduzida afim de que o aprendiz seja capaz de construir o conhecimento em etapas. A partir do aprendizado dos conceitos mais básicos, amplia-se a abrangência do estudo e são adicionadas mais camadas de abstração possibilitando a compreensão do assunto como um todo (PIAGET et al., 1977).

Tomemos como exemplo a estrutura do curso de Engenharia de Computação ministrado no Centro Federal de Educação Tecnológica de Minas Gerais, Campus VII em Timóteo, Minas Gerais (CEFET-MG, 2008). Seguindo a grade de disciplinas obrigatórias propostas e suas respectivas ementas, podemos ver que o aluno é introduzido ao *design* digital e à arquitetura de computadores de forma gradual, vendo a seguinte sequência de assuntos:

- Disciplina “Sistemas Digitais para Computação” (obrigatória):
  - Introdução aos princípios de Eletrônica Digital;
  - Conceitos e conhecimentos práticos referentes aos componentes utilizados em eletrônica digital;
  - Estudo dos circuitos integrados mais utilizados em eletrônica digital;
  - Montagem e simulação de circuitos digitais com análise de funcionamento;
  - Estudo da álgebra de Boole, circuitos aritméticos, circuitos seqüenciais (FLIP-FLOP), memórias e as famílias TTL e CMOS.
- Disciplina “Arquitetura e Organização de Computadores I” (obrigatória):
  - Histórico dos computadores digitais;
  - Níveis de abstração;
  - Blocos funcionais: processadores, memórias primária e secundária, entrada/saída;
  - Nível lógico-digital: circuitos digitais básicos, circuitos de memória, circuitos de microprocessadores e barramentos, interfaceamento;
  - Nível de microprogramação: micro-arquitetura, macro-arquitetura, micro-programas, exemplo de uma microarquitetura.
- Disciplina “Arquitetura e Organização de Computadores II” (obrigatória):

  - Nível de arquitetura convencional: formatos de instrução, endereçamento, tipos de instruções e controle de fluxo;
  - Nível de sistema operacional: memória virtual, instruções de entrada/saída virtuais, instruções virtuais usadas em processamento paralelo, exemplo de um sistema operacional;
  - Nível de linguagem montadora: linguagem montadora, o processo de montagem, macros, ligação e carregamento;
  - Introdução às arquiteturas não-convencionais de computadores.

- Disciplina “Arquitetura e Organização de Computadores III” (optativa):
  - Evolução das arquiteturas dos computadores; máquinas RISC: princípios de projeto de máquinas RISC, uso de registradores, exemplos de arquiteturas RISC: SPARC e MIPS;
  - Computadores paralelos: computadores MIMD, computadores SIMD, computadores vetoriais, computadores a fluxo de dados;
  - Multiprocessadores: multiprocessadores com memória compartilhada no barramento, multiprocessadores com memória compartilhada MIND.

- Disciplina “Microprocessadores e Microcontroladores” (optativa):
  - Breve histórico dos microprocessadores;
  - Arquitetura e organização de um microprocessador e um microcontrolador;
  - Conjunto básico de instruções;
  - Programação em linguagem montadora;
  - Modos de endereçamento, manipulação de registros, pilhas, subrotinas;
  - Métodos de transferência de dados: *polling*, interrupções, acesso direto a memória; organização de memórias, interfaces seriais e paralelas;
  - Dispositivos de entrada e saída; técnicas para acionamento e controle de periféricos.

Podemos observar que o plano de estudos parte de princípios básicos, simples e gerais e os utiliza como fundamentos para a construção de conhecimento mais específico.

Porém, ao fazer a transição para uma situação do mundo real o aluno pode se encontrar sem esses degraus iniciais. Ao se deparar com um escopo repleto de complexidade em um assunto desconhecido, o estudante não vê claramente como adentrar o assunto, dividir a complexidade em seções mais facilmente gerenciáveis e progredir rumo à compreensão do objeto de estudo. Essa barreira de entrada pode desestimular àqueles que desejam realizar pesquisa na área ou implementar de forma prática os conceitos aprendidos em sala de aula.

## 1.2 Problema

O problema que motiva este trabalho é o fato que todos aqueles que desejam realizar um projeto de pesquisa que envolva a prototipação de hardware em FPGA terão uma grande carga de trabalho direcionado a algo fundamental para a pesquisa mas que não a avança propriamente. Isso se deve ao fato de haver uma grande carga de conhecimento envolvido em realizar a transição do estado inicial em que nada está pronto até o estado em que o desenvolvimento passa a ser apenas uma forma de expressar o objeto da pesquisa e todas as ferramentas disponíveis atuam de forma complementar dando uma visão mais completa deste ao pesquisador.

Um dos principais fatores contribuindo para essa quantidade de trabalho é a acentuada curva de aprendizado relacionado à prototipação de hardware. Algumas das referências básicas sobre o funcionamento específico das tecnologias, como os manuais de usuário, possuem centenas de páginas sem um fluxo direcionado a ser seguido.

Já as referências mais gerais, como livros de desenvolvimento de hardware em VHDL e *design* digital em geral, não possuem os passos específicos devido à variabilidade de procedimentos para escolhas de tecnologias diferentes.

Além disso, a maioria absoluta das referências que alguém que aspira desenvolver circuitos utilizando alguma linguagem de descrição de hardware são em língua estrangeira, primariamente o inglês. Essa questão linguística fica mais exarcebada quando considera-se as referências que lidam com aspectos mais práticos e específicos do desenvolvimento, adicionando uma barreira quase intransponível para aqueles que não possuem domínio da linguagem.

Por causa disso, é necessário bastante esforço de pesquisa para realizar a ligação entre esses estados de descrição breve e generalizada e a minuciosa exploração de todos os detalhes técnicos relacionados à tecnologia, dos quais a maioria podem não ser relevantes para o pesquisador, que está mais interessado nas funcionalidades fundamentais.

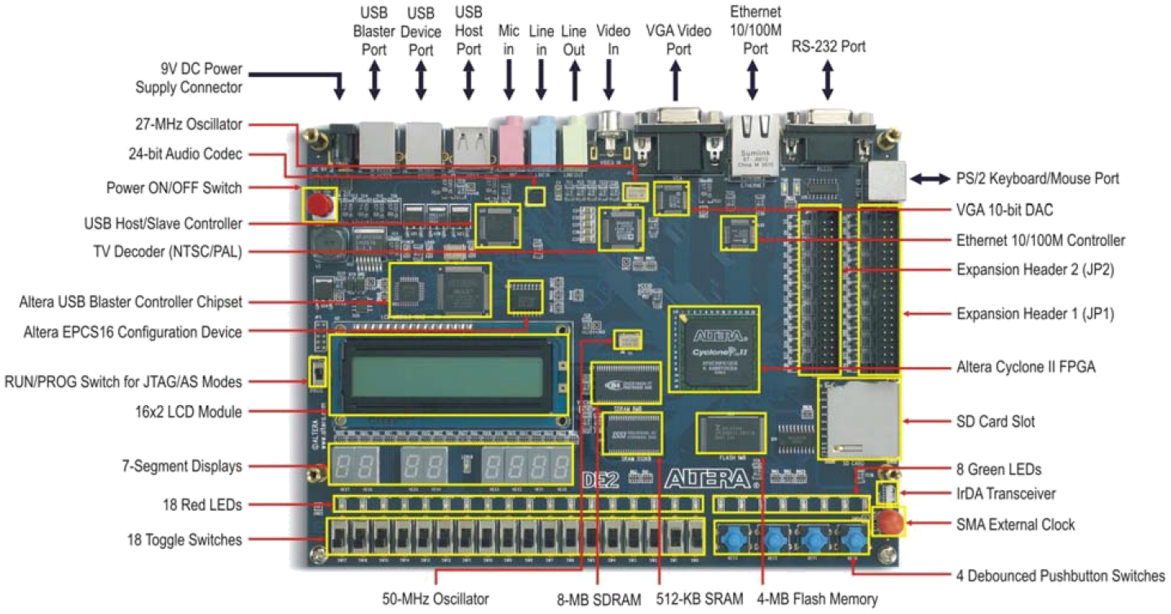
## 1.3 Objetivos

Este trabalho busca ser um guia compreensível em língua portuguesa para permitir que àqueles interessados em realizar pesquisas envolvendo o desenvolvimento de hardware usando lógica digital foquem seus esforços em sua área de pesquisa ao invés de gastar tempo e energia ajustando o ambiente necessário.

Devem ser beneficiados por este trabalho não somente aqueles que desejam realizar pesquisa relacionada à arquitetura de computadores, mas também qualquer trabalho que possa envolver lógica digital. Os que buscam ter uma experiência prática com tecnologias de *design*, simulação e análise de hardware, juntamente com aqueles que desejam realizar alguma aplicação prática em outra área que possa se beneficiar de um circuito de lógica programável, também poderão fazer proveito das informações aqui contidas.

Por ser um trabalho de natureza técnica e altamente atrelado às tecnologias utilizadas,

Figura 1 – *Layout* da placa de desenvolvimento Altera DE2.




Fonte: (ALTERA, 2006)

limitou-se o escopo ao desenvolvimento voltado aos dispositivos FPGA disponibilizados no momento da escrita deste aos alunos do Centro Federal de Educação Tecnológica de Minas Gerais Campus VII, em Timóteo. Os equipamentos disponibilizados para os estudantes são as placas de desenvolvimento DE2 da fabricante Altera, cada uma contendo como chip de lógica programável o Cyclone II EP2C35F672C6 (ALTERA, 2006). O *layout* da placa pode ser visto na figura 1.

Todos com acesso à placa terão disponível uma licença de estudante para os softwares necessários que não são de código aberto. Estes softwares são o Quartus II e o Modelsim, ambos disponibilizados em um CD-ROM de instalação junto ao restante dos componentes do conjunto Altera DE2.

Espera-se que o leitor seja familiarizado com lógica digital, arquitetura de computadores, sistemas operacionais baseados em Linux e de programação de computadores.

## 1.4 Estrutura da monografia

Esta monografia está organizada em 5 capítulos. Esses capítulos foram ordenados em uma sequência lógica que facilitasse a compreensão do leitor, já que cronologicamente houve certo paralelismo durante a produção de algumas etapas.

- No capítulo 2 é demonstrado o processo metodológico adotado no desenvolvimento deste trabalho. Nele é descrito o processo de seleção da literatura base, a definição

das tecnologias envolvidas e como foi planejado a análise a fim de descrever um procedimento viável de prototipação de hardware.

- Sequencialmente, no capítulo 3 é apresentada a fundamentação que serve como base teórica para o trabalho, incluindo uma conceituação mais apurada sobre a arquitetura de processadores escolhida como referência e as tecnologias utilizadas para o desenvolvimento.

- No capítulo 4 é exposto todo o processo de desenvolvimento, demonstrando as técnicas e softwares utilizados. Esse capítulo contém o passo a passo da utilização das ferramentas, boas práticas de desenvolvimento e coisas que devem ser levadas em consideração ao projetar unidades de lógicas em linguagens de descrição de hardware.

- No capítulo 5 discorre-se sobre os resultados obtidos, algumas considerações sobre a abordagem aqui adotada e os resultados esperados e algumas sugestões de trabalhos futuros para expandir a pesquisa aqui realizada e a gama de opções daqueles que buscam prototipar hardware.

# 2 Procedimentos metodológicos

*"Se soubéssemos o que estamos fazendo, não seria chamado de pesquisa, seria?"* [1].
*Albert Einstein (Tradução nossa)*

A pesquisa aqui desenvolvida é básica, já que se propõe a agir como uma referência para o desenvolvimento de trabalhos futuros utilizando as tecnologias indicadas. O formato desenvolvido envolve uma pesquisa bibliográfica abrangendo desde a literatura geral sobre arquitetura de computadores até a documentação específica dos artifícios tecnológicos empregados. Também inclui uma análise qualitativa das tecnologias e práticas descritas na bibliografia, sendo a análise qualitativa a força motriz por trás da seleção dos tópicos abordados. O trabalho então explora de forma descritiva as conclusões obtidas e o processo percorrido para se chegar às mesmas.

Os procedimentos metodológicos adotados nessa pesquisa não seguiram uma sequência linear. Ao invés disso, houve um processo iterativo buscando uma forma viável de abordar o problema. Esse processo iterativo inclui a pesquisa sobre quais as tecnologias seriam aplicáveis dada uma determinada demanda, uma investigação mais aprofundada entre as possíveis candidatas e testes com a tecnologia escolhida afim de decidir por escolhê-la ou testar outra candidata. Esse processo foi aplicado em todos os níveis da pesquisa, desde a escolha da arquitetura que serviria como plataforma para o desenvolvimento de hardware até técnicas de implementação direta.

Os procedimentos metodológicos adotados podem ser descritos na seguinte sequência:

1. Decompor o problema em problemas menores de acordo com a capacidade de identificá-los em um primeiro momento;
2. Buscar sobre o assunto na literatura geral sobre o desenvolvimento relacionado à arquitetura de computadores e determinar um conjunto de possíveis soluções;
3. Explorar a documentação existente para as tecnologias envolvidas sobre como seria dada a melhor abordagem técnica, quais os benefícios e as desvantagens de tal uso e suas limitações;
4. Realizar testes práticos com as abordagens e tecnologias levantadas nas etapas anteriores e avaliar os resultados obtidos;
5. Decidir se o problema atual pode ser considerado solucionado dentro dos objetivos esperados e proceder para o próximo problema.

[1] *"If we knew what we were doing, it wouldn't be called research, would it?"*

Neste capítulo será expandido sobre como esse processo foi aplicado de forma macro sobre o escopo geral do trabalho. Pelo âmbito da pesquisa envolver explorar as soluções existentes para o problema proposto e compor um procedimento recomendado para pesquisadores futuros esses procedimentos não foram aplicados de forma clara e direta. Para cada nível de abstração em que um problema era identificado foi aplicada essa metodologia iterativamente até serem encontrados resultados satisfatórios, que foram compilados no capítulo 4.

## 2.1 Literatura sobre arquitetura

No início deste trabalho, realizou-se uma pesquisa na literatura geral sobre o desenvolvimento relacionado à arquitetura de computadores sobre o estado da arte de métodos de rápida prototipação. Esse fator é chave já que, diferente da versão final de um produto que utilizaria por exemplo uma tecnologia ASIC, as técnicas de prototipação buscam criar um ambiente dinâmico com custo reduzido para erro e de rápido desenvolvimento de ideias, não se preocupando tanto com a otimização de propriedades do material e do design.

Não somente ficou determinado que a tecnologia FPGA atende os requerimentos de prototipação, como esta também é de ampla utilização no mercado e possui amplo reconhecimento na literatura. Atualmente é prática comum que o ensino de arquitetura de computadores e de *design* digital incluam a descrição de hardware em alguma linguagem HDL e a prototipação utilizando um chip FPGA. Entre as várias referências que adotam tal prática, podem ser citadas Chu (2006), Oklobdzija (2007), Kleitz (2011) e Mano e Ciletti (2012).

Também buscou-se utilizar um processador completo como plataforma de desenvolvimento. Esse processador deveria ser simples o suficiente para ser compreendido e possivelmente alterado por um aluno de graduação que estivesse disposto a dedicar tempo em um esforço de pesquisa. Sendo assim, processadores que utilizam técnicas mais complexas, como execução fora de ordem e processamento superescalar foram descartados.

O LEON3, processador atualmente mantido pela empresa sueca Cobham Gaisler, se mostrou uma arquitetura robusta o suficiente para os propósitos desejados e de simples utilização, já que existem diversas ferramentas desenvolvidas pela própria Cobham para auxiliar seu uso e desenvolvimento, além de ser uma plataforma aberta com suporte a diversos chips FPGA existentes.

## 2.2 Literatura sobre as tecnologias

A placa FPGA disponibilizada para uso neste trabalho pelo CEFET-MG é a Altera DE2, com o chip Altera Cyclone II EP2C35F672C6. O manual de instruções da placa recomenda o uso do software Quartus II para o desenvolvimento, contendo já na caixa um CD-ROM com instaladores para Windows e Linux.

Devido à sua licensa aberta e suporte a outras ferramentas como o GCC e Make, optou-se por utilizar um sistema operacional Linux. O desenvolvimento ocorreu na distribuição

Linux Mint 17.

Procurou-se então a literatura disponível sobre as tecnologias utilizadas afim de melhor empregar os recursos disponíveis.

Para o uso do Quartus II as principais referências foram os próprios manuais da Altera. As base do uso do software foram compreendidas a partir de (ALTERA, 2007b), (ALTERA, 2007c), (ALTERA, 2007d), (ALTERA, 2007e), (ALTERA, 2007f). Para o simulador Modelsim, a principal referência foram (MENTOR GRAPHICS, 2012) e (MENTOR GRAPHICS, 2015).

Quanto ao LEON3 foram levantados, além de documentos oficiais de referência para o uso do LEON3 e outros componentes da biblioteca GRLIB, foram utilizadas também referências que envolvem o desenvolvimento envolvendo o processador LEON3. Como exemplo desta podemos citar (DANĚK et al., 2012) e (BENGTSSON; FÅNG, 2011), enquanto daquela, (GAISLER et al., 2007) e (GAISLER, 2010).

## 2.3 Testes, resolução de problemas e documentação

A cada etapa de desenvolvimento ao longo deste trabalho foram realizados testes empíricos a fim de averiguar a validade do método selecionado.

O método de desenvolvimento, que engloba as tecnologias desenvolvidas, as considerações a serem feitas durante o *design*, a forma de desenvolver e testar o protótipo, deve atender às seguintes características:

- Ser determinístico, ou seja, aqueles que aplicarem o mesmo método devem obter os mesmos resultados;
- Ser simples, já que o foco primário deste trabalho é permitir que alunos de graduação sem experiência com prototipação de hardware e com conhecimentos básicos de lógica digital e arquitetura de computadores sejam capazes de executar os procedimentos de forma satisfatória;
- Ser realista, fazendo com que o desenvolvedor entenda as etapas de desenvolvimento e seja capaz de as realizar completamente, desde o momento da concepção da ideia até à aplicação do circuito resultante aplicado em um chip de FPGA.

Espera-se que ao seguir o processo documentado neste trabalho, o estudante seja capaz de desenvolver seu projeto.

Com um procedimento documentado sobre como se dá o desenvolvimento, coisas que devem ser levadas em consideração e uma compilação de referências, deseja-se estimular a melhor compreensão do *design* digital, circuitos lógicos e arquitetura de computadores.

Com essa capacidade, espera-se despertar maior interesse relacionado às áreas acima citadas, resultando em maior volume de desenvolvimento de pesquisas e melhor capacitação de futuros profissionais que busquem uma carreira que envolva a prototipação de hardware.

# 3 Fundamentos teóricos

*“A ciência pode ser descrita como a arte da simplificação excessiva e sistemática.”*[1] .
*Karl Popper (Tradução nossa)*

Neste capítulo busca-se contextualizar o leitor dos fundamentos teóricos nos quais este trabalho se baseia, mais especificamente situando-o com relação às tecnologias utilizadas. Aqui são apresentados conceitos gerais sobre as técnicas atualmente envolvidas no desenvolvimento e prototipação de hardware, além de abordar brevemente alguns conceitos de arquitetura de computadores. Assume-se que o leitor já seja familiar com conceitos de programação de computadores, circuitos lógicos digitais e arquitetura de computadores.

Para melhor compreensão, este capítulo é dividido em cinco seções. Na seção 3.1 são apresentados alguns conceitos fundamentais de arquitetura de computadores que possuem grande impacto para aqueles que desejam utilizar o processo descrito neste trabalho para desenvolvimento de pesquisa relacionada à microprocessadores. Na seção 3.2 ocorre a demonstração de algumas peculiaridades de arquiteturas que seguem a especificação SPARC. Na seção 3.3 é apresentado o processador LEON3 juntamente com outras tecnologias auxiliares. Na seção 3.4 vemos uma breve explicação sobre o que são linguagens de descrição de hardwares e alguns aspectos do VHDL, linguagem utilizada ao longo deste trabalho. Por fim, na seção 3.5 é explicado como funciona o FPGA, tecnologia utilizada neste trabalho para gravação de circuitos em um chip físico.

## 3.1 Arquitetura de computadores

Segundo Tanenbaum e Zucchi (2009), o termo "organização de computadores"pode ser definido como a forma de estruturar computadores como uma sequência de abstrações projetadas uma sobre a anterior, permitindo o gerenciamento da complexidade inerente ao *design*. Já o termo "arquitetura de computadores"classicamente se refere somente à definição da ISA, o conjunto de instruções de máquina, e das funcionalidades visíveis ao programador em linguagem de máquina como quantidade e estrutura da memória, instruções disponíveis, etc. Porém, na prática, "organização de computadores"e "arquitetura de computadores"são utilizados de forma intercambiável, tratando tanto das definições da interface que está visível ao usuário do processador quanto da forma como esta foi implementada.

Como parte do escopo deste trabalho, o foco principal se dá na implementação de uma dada arquitetura. Alterações em características arquiteturais são realizadas da mesma forma a nível de implementação, mas o processo de criação, avaliação de prós e contras, decisão e testes de uma arquitetura está além do escopo deste trabalho. Sendo assim, aquele que desejar criar uma arquitetura nova sob demanda poderá fazer uso das mesmas técnicas aqui

[1] *“Science may be described as the art of systematic oversimplification.”*

descritas para o processo de desenvolvimento do circuito, porém entendendo que a definição dos requisitos arquiteturais e da interface externa requerem diferentes capacidades técnicas.

A seguir, serão abordados alguns conceitos de arquitetura de computadores de crítica importância para o desenvolvimento de qualquer alteração na implementação de um processador moderno.

### 3.1.1 Paralelismo a nível de instrução

O paralelismo a nível de instrução é o nome dado ao conjunto de técnicas que buscam explorar as características paralelizáveis do hardware para executar mais instruções dado um período de tempo, reduzindo a taxa CPI (clock por instrução) média.

Cada uma dessas técnicas busca reduzir o tempo ocioso de unidades do processador, fazendo com que múltiplos componentes trabalhem simultaneamente para explorar ao máximo as capacidades do hardware. Se explorando esse paralelismo há um ganho de desempenho, há também um aumento na complexidade do circuito, demandando maior atenção para os possíveis problemas causados pela necessidade de sincronia entre instruções. Essa possibilidade de transformar um programa criado para execução sequencial em um conjunto de instruções que podem ser executadas simultaneamente ou até mesmo fora de ordem é o cerne do paralelismo a nível de instrução.

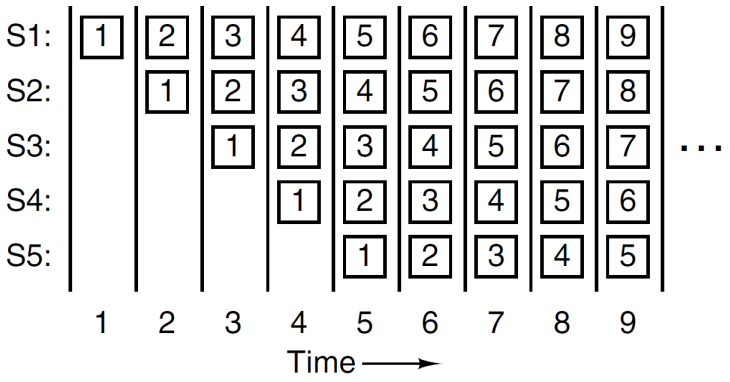
#### 3.1.1.1 *Pipeline*

O *pipeline*, termo em inglês que convém a ideia de "tubulação", é uma técnica que busca dividir a execução de cada instrução em etapas sequenciais padronizadas, chamados estágios. Assim que uma instrução finalizar a execução necessária dentro de um estágio, os dados necessários são transferidos para o próximo estágio e uma outra instrução começa a realizar o seu processamento no estágio atual. Esse processo, semelhante a um tubo no qual um fluido se desloca constantemente ocupando todo o comprimento do condutor, é demonstrado na figura 2.

Uma outra vantagem da utilização da técnica *pipeline* é a redução do caminho crítico em cada ciclo. Como cada instrução executa somente um estágio por ciclo de clock, a latência do caminho crítico deixa de ser o tempo de execução da instrução mais longa e passa a ser o tempo de execução do estágio mais longo. Quanto menor os estágios do *pipeline*, maior pode ser a frequência do clock (PATTERSON; HENNESSY, 2013).

Cabe aqui ressaltar que essa relação não é linear, já que o aumento da taxa de clock implica no aumento da dissipação de potência em forma de calor de uma mesma área de chip. Sendo assim, a não ser que as capacidades de resfriamento do conjunto sejam capazes de lidar com essa maior produção de energia térmica, o aumento da taxa de clock irá fazer com que o circuito falhe por super-aquecimento. O projetista da implementação deve então considerar esses fatores ao determinar o tamanho do estágio do *pipeline*, já que a redução do CPI médio obtida através do aumento da complexidade de um estágio pode causar uma melhoria de performance a despeito do efeito colateral de aumento de latência, forçando uma

Figura 2 – Diagrama representando o fluxo ideal de instruções ao longo do tempo em um *pipeline* de 5 estágios.




Fonte: Tanenbaum e Zucchi (2009)

redução da taxa de clock (HENNESSY; PATTERSON, 2011).

Como a maioria dos componentes utilizados em um processador são síncronos, estes iniciarão a computação baseada nas entradas disponíveis em suas portas no momento em que a borda do sinal de clock for atingida. Durante a execução de uma instrução que abrange vários estágios, é necessário então armazenar os valores computados no ciclo atual para a utilização como entradas do próximo estágio. Para essa tarefa normalmente são empregados registradores, denominados registradores de estágio. Qualquer alteração em um processador que utilize essas estratégias de paralelismo deve considerar como irá lidar com a transição entre estágios, seja expandindo os registradores de estágio para armazenar mais dados ou alterando dados que já seriam armazenados.

Um outro fator que pode causar problemas para o projetista são as restrições temporais dos componentes utilizados, incluindo os componentes do estágio atual, do próximo estágio e os próprios registradores de estágio. Essas restrições serão abordadas na seção 3.1.2.1.

#### 3.1.1.1.1 *Hazard* de dados

Segundo (PATTERSON; HENNESSY, 2013), os *hazards* de dados são os eventos em que duas instruções em execução simultaneamente compartilham dados. Um evento desse tipo pode ser classificado como *hazard*, do inglês significando risco, por poderem causar resultados incorretos na execução do programa.

Os tipos de *hazard* de dados são:

- RAR (*Read after read*, ou leitura após leitura) - Ocorre quando uma instrução lê dados que foram lidas por outra instrução ainda em execução. Não apresenta riscos para o resultado final produzido pela computação e é incluído apenas por fatores didáticos;

- RAW (*Read after write*, ou leitura após escrita) - Ocorre quando uma instrução lê dados que serão sobrescritos por outra instrução ainda em execução. Como esta ainda não completou sua passagem por todos os estágios do *pipeline*, aquela pode realizar a leitura de um valor obsoleto. Existem duas possíveis soluções para esse *hazard* dependendo da distância entre os estágios atuais de cada instrução. Se a instrução contendo a escrita já tiver computado o valor a ser escrito, esse resultado pode ser passado para a instrução contendo a leitura. Caso contrário, acontece o *stall*, ou parada, em que a instrução que realiza a leitura e todas as outras subsequentes tem sua execução interrompida até que a instrução de escrita tenha terminado de computador o valor, permitindo então que ocorra o *forwarding*. É considerada uma "dependência verdadeira";

- WAR (*Write after read*, ou escrita após leitura) - Ocorre quando uma instrução sobrescreverá dados que foram lidos por outra instrução em execução. Desde que a ordem das instruções seja preservada, não possui impacto no resultado final e é citado por questões didáticas. É considerada uma "dependência falsa", a "anti-dependência";

- WAW (*Write after write*, ou escrita após escrita) - Ocorre quando uma instrução sobrescreverá dados que serão escritos por outra instrução em execução. Desde que a ordem das instruções seja preservada, não possui impacto no resultado final e é citado por questões didáticas. É considerada uma "dependência falsa", a "dependência de saída".

#### 3.1.1.1.2 *Hazard* de controle

Como descrito por (PATTERSON; HENNESSY, 2013), os *hazards* de controle são eventos em que uma decisão de controle do processador deve ser tomada considerando informações de instruções ainda em execução. A não ser que a arquitetura sendo considerada possua alguma característica especial, como é o caso da SPARC[2], esses *hazards* serão encontrados somente em instruções de desvio condicional e desvios relativos.

Segundo (HENNESSY; PATTERSON, 2011), a solução para esse *hazard* é a chamada predição de desvios. Usando essa abordagem, o processador carrega as próximas instruções a partir do endereço para que acredita que o fluxo do programa seguirá. De qualquer forma, a condição do desvio é avaliada e caso o processador tenha acertado o próximo endereço do contador de programas, a execução segue normalmente. Caso contrário, são enviados sinais especiais para neutralizar as instruções erroneamente colocadas em execução para evitar que estas causem efeitos colaterais no estado do programa, como uma alteração indesejada nos dados.

Existem duas metodologias para realizar a predição de desvios. A predição estática de desvios assume que os desvios nunca serão tomados, carregando a próxima instrução normalmente, ou assume que os desvios serão sempre tomados e desvia o fluxo do processador.

[2] Como detalhado na seção 3.2.1, a arquitetura SPARC trabalha com janelas de registradores. Uma instrução que altere a referência para a qual essa janela aponta pode se mostrar como um *hazard* de controle, já que todas as traduções de endereços de registradores utilizados terão seus valores finais alterados. Claro que essa possibilidade depende da forma em que foi dada a implementação.

O desempenho de cada uma dessas decisões depende do tipo de software que será primariamente executado sobre aquela plataforma. Por padrão, diz-se que um processador possui predição estática de desvios quando ele assume o desvio sendo tomado, já que não é necessária nenhuma alteração em um processador comum para que este assuma o desvio como não tendo sido tomado.

A outra metodologia de predição de desvios é a predição dinâmica. A implementação dessas unidades varia imensamente, de simples máquinas de estados a componentes complexos que levam em consideração diversos aspectos do estado do processador. De qualquer maneira, a ideia por trás de uma unidade de predição de desvios dinâmica é a utilização de informações sobre os desvios passados para realizar sua predição. Caso nas últimas instruções de desvio o desvio não tenha sido tomado, será assumido como não tomado e vice-versa.

Independente da técnica de previsão de desvios empregadas, algo que beneficia a todas é a criação de hardware dedicado para a previsão de desvios. Assim, ao remover a dependência da ALU (unidade lógico-aritmética) para a avaliação da condição de desvios e cálculo do endereço final, pode-se antecipar a resolução da instrução e agir caso a predição tenha sido equivocada. Quanto menos estágios necessários para a realização dessa conferência, menos instruções fora do fluxo do programa serão carregadas no *pipeline* e menor terá sido o número de ciclos gastos executando instruções descartadas.

### 3.1.2 Outras armadilhas

Existem alguns outros aspectos que devem ser considerados ao trabalhar com circuitos digitais de larga escala. Aqui serão citados alguns desses pontos e os impactos que estes podem ter no funcionamento geral do processador.

#### 3.1.2.1 Considerações temporais

Um processador é um circuito síncrono. Para garantir que o fluxo de informação seja constante, todos os componentes devem atender às restrições temporais impostas pela frequência de trabalho deste e pelo tempo de execução dos outros componentes que trabalham de forma síncrona com este.

##### 3.1.2.1.1 Latência

A primeira informação a ser avaliada para um componente projetado para trabalhar sob regime síncrono é sua latência. A latência é o tempo necessário para que o componente produza o resultado a partir do momento em que suas entradas são alteradas e este começa a processar os dados. Para calcular a latência avaliamos o caminho crítico do circuito, ou seja, o maior trajeto que pode ser percorrido por um sinal. A cada elemento de circuito que o sinal passa um atraso é adicionado e o somatório desses atrasos no pior caso possível é a latência do componente.

A latência é uma grandeza fundamental para qualquer circuito digital. Em circuitos que trabalham em frequências altas, como é o caso de microprocessadores, um componente com

alta latência pode aumentar o tempo de resposta do circuito como um todo, já que o atraso é medido no pior caso. Dependendo da situação, pode ser necessário ajustar a frequência de trabalho do circuito completo para que este possa acomodar o componente.

Uma alternativa para componentes de alta latência é dividir o processamento em etapas menores. Exceto em raras situações, um componente que agilize não agilize o processamento de todas as instruções é melhor projetado trabalhando ao longo de vários ciclos. Isso se deve ao fato que, por mais que haja uma perda de eficiência ao ser obrigado a esperar a próxima mudança de ciclo mesmo já tendo computado os resultados, um componente que atende às restrições temporais do circuito atual não causará impacto no desempenho do restante do processador. Como exemplo de componentes que realizam suas tarefas ao longo de múltiplos ciclos podemos citar algumas unidades de operações aritméticas mais complexas e unidades de memória (PATTERSON; HENNESSY, 2013).

A divisão do processamento de um componente em múltiplos ciclos ainda oferece uma outra vantagem. Em processadores capazes de trabalhar com mais de uma instrução em execução simultaneamente esta unidade de alta latência pode trabalhar independentemente do resto do circuito, enquanto outras instruções que não possuem dependências verdadeiras sobre os resultados desta continuam sendo executadas em paralelo sem mudança em seu desempenho esperado.

#### 3.1.2.1.2 Defasagem de clock, tempo de *setup* e *hold*

Apesar de ser um problema pouco comum em circuitos modernos, é importante que o projetista esteja ciente da possibilidade da ocorrência da defasagem de clock. A defasagem de clock acontece quando o sinal de clock chega em componentes sequencialmente adjacentes com uma diferença de tempo. Apesar de ter entre suas principais causas fatores relacionados às propriedades físicas do material utilizado no circuito e das tecnologias atuais terem tornado estes efeitos em grande parte irrelevantes, é possível causar a defasagem de clock através do design do circuito.

Caso o sinal de clock utilizado por parte do circuito seja derivado da saída de algum componente, como por exemplo um circuito para ajustar a frequência do sinal, poderá se perceber uma defasagem de clock de magnitude equivalente à latência deste componente. Essa defasagem pode interferir com o resultado produzido por um circuito síncrono. Portanto, o projetista deve sempre conectar componentes que devem trabalhar em fina sincronia em uma única fonte de clock.

Outras características que podem causar a dessincronização entre componentes adjacentes e sequenciais são os tempos de *setup* e *hold*. Os tempos de *setup* e *hold* são características de cada componente e que estão diretamente relacionadas à latência deste.

O tempo de *setup* informa com quanto tempo de antecedência os sinais de entrada devem estar estáveis antes do sinal de clock. Devido ao tempo necessário para que os sinais se propaguem pelos elementos lógicos, em componentes complexos existe a necessidade de se estabilizar os sinais de entrada antes que o sinal de clock chegue na borda que inicia o

próximo ciclo. Quando a restrição de tempo de *setup* é respeitada, os sinais de entrada que devem ser processados quando no início do ciclo já estão em todos os elementos de circuitos necessários. Caso essa restrição não seja respeitada, o resultado obtivo pode não condizer com o sinal de entrada. De forma semelhante funciona o tempo de *hold*. O tempo de *hold* se refere a por quanto tempo após o início do próximo ciclo as entradas devem permanecer estáveis. (CHU, 2006).

O tempo de *setup* e *hold* funcionam como abstrações para o exterior sobre o comportamento do componente externo. Sabendo essas informações, é possível garantir que as latências internas da unidade serão respeitadas sem conhecimento específico de sua composição interna. Assim é possível trabalhar com circuitos síncronos e gerenciar a complexidade da análise temporal.

### 3.1.3 Área de circuito e potência

Apesar do foco maior do projetista ser no desempenho de um dado circuito, não se deve deixar de lado um fator essencial em diversas aplicações: o consumo de potência do circuito. Em sistemas embarcados, o baixo consumo de energia é um ponto chave, já que muitos produtos podem estar dependendo de baterias para sua alimentação. Logo, um circuito com menor consumo energético implicará em uma autonomia maior de bateria.

Dois aspectos que possuem alta relação com a potência consumida por um circuito são a área de chip e a frequência de clock. Essa relação existe devido ao maior número de transistores trocando de estado. Na tecnologia CMOS, componentes digitais consomem potência apenas no momento da troca de valor, seja de tensão alta para baixa ou vice-versa (AGARWAL; LANG, 2005). Logo, quanto maior a área de circuito, o que implica em maior quantidade de transistores e outros elementos necessários para implementar a lógica digital, maior a potência dissipada. Da mesma forma, quanto maior a frequência de um circuito, mais trocas de posição das portas lógicas ocorreram em um determinado período de tempo, aumentando a potência dissipada no intervalo e, por definição, aumentando a potência consumida.

O ponto ideal no balanço entre potência consumida e desempenho do processador fica a cargo do projetista. Quanto maior a eficiência do circuito, menor o impacto nessa troca, por isso um projeto inteligente é de prima importância.

Existem algumas técnicas de engenharia que podem ser empregadas para tentar amenizar o consumo de um circuito. A primeira e mais simples é buscar que a alimentação do processador esteja, dentro do intervalo necessário para o funcionamento correto, com a tensão mais baixa possível. Como é possível concluir a partir da lei de Ohm e da definição de potência, quanto menor a diferença de potencial entre as duas extremidades de um elemento, menor a potência dissipada neste (AGARWAL; LANG, 2005).

Uma outra possibilidade é tentar reduzir o desempenho do circuito ao necessário. Por exemplo, um processador que está passando uma parte considerável de seu tempo ocioso pode ter sua taxa de clock reduzida, reduzindo assim seu consumo. Uma outra situação em que essa ideia se manifesta é em sistemas embarcados que realizam tarefas esporadica-

mente. Tirando componentes essenciais para o funcionamento correto durante o período ocioso, como por exemplo um temporizador, pode-se desligar partes do circuito para que não consumam energia. Essas implementações dinâmicas requerem uma complexidade maior de projeto, mas são possibilidades para o projetista de um sistema em que o consumo energético é um fator crítico (HOROWITZ; HILL, 2015).

Um último ponto relevante sobre a potência dissipada em um circuito é a relação com a produção de calor. A potência dissipada por um circuito digital é em grande parte transformada em energia térmica. O calor produzido, quando acumulado excessivamente, pode alterar as propriedades do material e causar comportamentos inesperados ou até mesmo a falha de elementos do circuito. Caso estime-se que um circuito será utilizado por longos períodos de tempos em um alto consumo de potência, é recomendado que o projetista leve em consideração a capacidade de dissipar o calor gerado, a fim de evitar danos durante a utilização.

## 3.2 Arquitetura SPARC

A arquitetura SPARC, *Scalable Processor Architecture* foi desenvolvida pela empresa Sun Microsystems em 1987. Atualmente é mantido pela SPARC International, que a fez uma arquitetura aberta e livre de *royalties*. Foi baseada principalmente no projeto RISC da Universidade de Berkeley, e portanto compartilha diversas características com outras linguagens RISC.

As duas principais diferenças entre a arquitetura SPARC e outras arquiteturas RISC são o arquivo de registradores e a forma com que trata o espaço de atraso de instruções de desvio (KADY; KHATER; ALHAFNAWI, 2014). Essas características serão exploradas a seguir.

### 3.2.1 Arquivo de registradores

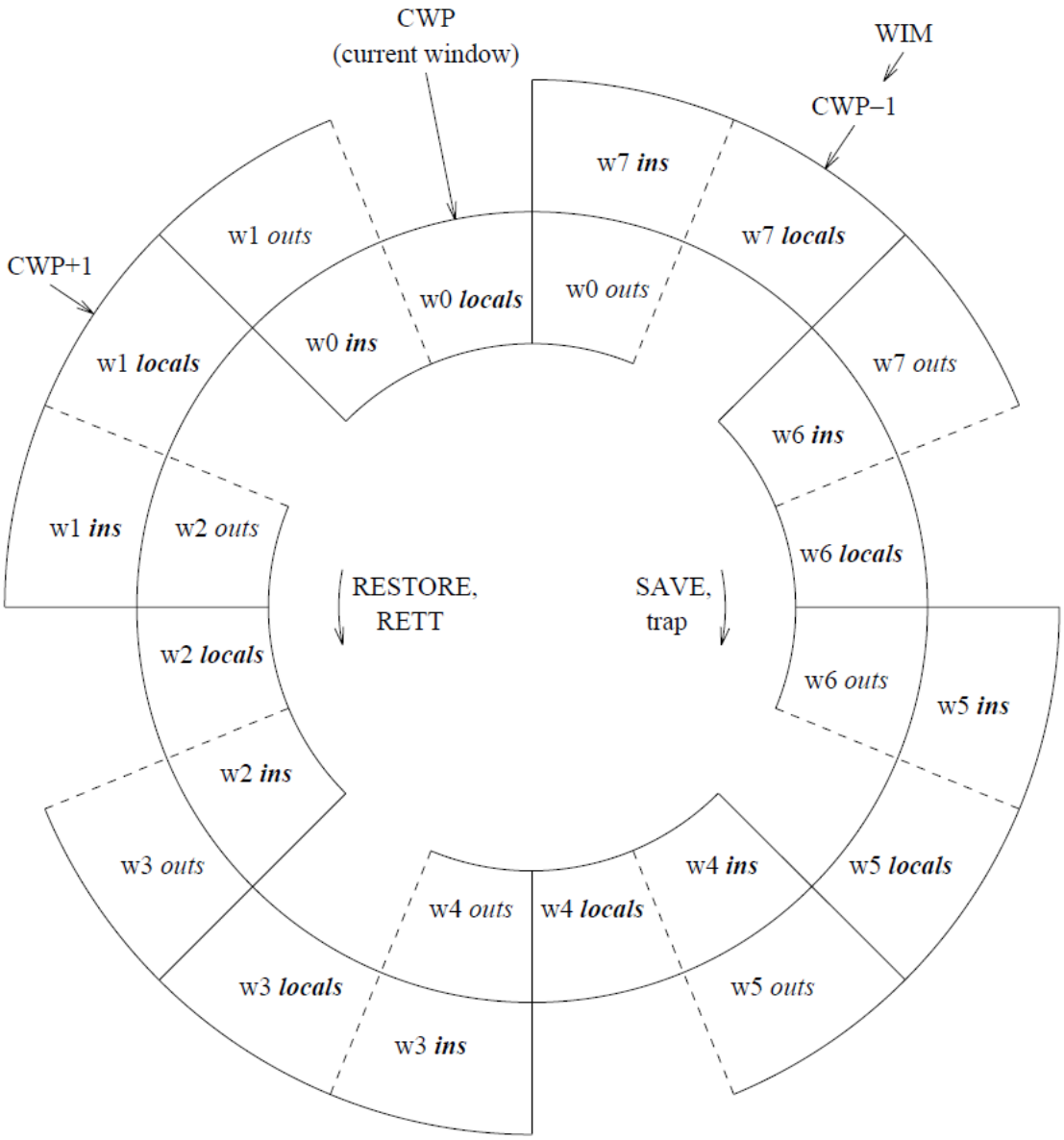
Dado um momento qualquer, uma instrução da ISA SPARC tem acesso à 32 registradores de dados que podem ser utilizados para armazenar valores arbitrários. Porém, a quantidade total de registradores varia de acordo com a implementação. Isso se deve ao fato da arquitetura trabalhar com "janelas de registradores". A quantidade de janelas varia de 2 a 32 de acordo com a implementação.

Dada a janela atual, existem 4 grupos de registradores, listados abaixo.

- Registradores 00 a 07 (*globals*, ou globais): compartilhados por todas as janelas.
- Registradores 08 a 15 (*out*, ou saída): compartilhados com a janela anterior.
- Registradores 16 a 23 (*local*, ou locais): exclusivos da janela atual.
- Registradores 24 a 31 (*in*, ou entrada): compartilhados com a próxima janela.

Todos esses registradores possuem as mesmas características, variando apenas em relação a seu acesso. O registrador especial CWP (*Current window pointer*, ou ponteiro para a janela atual) armazena em qual janela o processador se encontra atualmente. O CWP pode

Figura 3 – Diagrama representando conceitualmente o funcionamento das janelas de registradores SPARC. Registradores globais não representados. Na figura, o registrador CWP, que armazena qual janela está sendo atualmente usada, possui valor 0.




Fonte: SPARC International, Inc. (1992)

ser alterado somente por instruções especiais: a instrução *RESTORE* incrementa seu valor enquanto a instrução *SAVE* ou o lançamento de uma *trap* o decrementam. Esse funcionamento pode ser melhor visto na figura 3.

### 3.2.2 Instruções de desvio

Assim como algumas outras arquiteturas RISC, uma arquitetura SPARC executa a instrução posterior a uma instrução de desvio, exceto no caso da instrução de *trap* (lançamento de exceção) (SPARC International, Inc., 1992). Essa decisão foi tomada como um esforço para evitar o descarte de ciclos de execução, já que até ser possível determinar se o desvio deve ser tomado e para qual endereço o pipeline já está sendo carregado com as próximas instruções de uma execução sequencial. Com hardware específico para acelerar a decisão de desvios, o desvio pode ser efetuado já no segundo estágio, fazendo com que apenas uma instrução sequencial ao desvio seja iniciada no *pipeline*. Essa instrução é dita estar no espaço de atraso de instruções de desvio (em inglês *branch delay slot*). Esse espaço varia de acordo com a quantidade de ciclos que a arquitetura especifica serem levados para a resolução de uma instrução de desvio (HENNESSY; PATTERSON, 2011).

A diferença da arquitetura SPARC neste quesito é a opção de anular a execução dessa instrução caso o desvio não seja tomado. Para cada tipo de instrução de desvio, há uma variação que altera a *flag annul*, que informa ao processador que a próxima instrução deve ser anulada caso o desvio não seja tomado (SPARC International, Inc., 1992). Sendo assim, no caso de um laço de repetição é possível colocar no espaço de atraso uma instrução pertencente ao laço, que não possui dependências verdadeiras sobre seu resultado dentro do laço nesse espaço e melhor aproveitar o tempo do processador, sem a preocupação que essa instrução seria executada após o laço de repetição se encerre. Essa preocupação com a execução após o término do laço de repetição é um dos motivos que leva programadores a preencherem esse espaço com instruções "NOP", efetivamente desperdiçando ciclos de processamento.

## 3.3 O LEON3

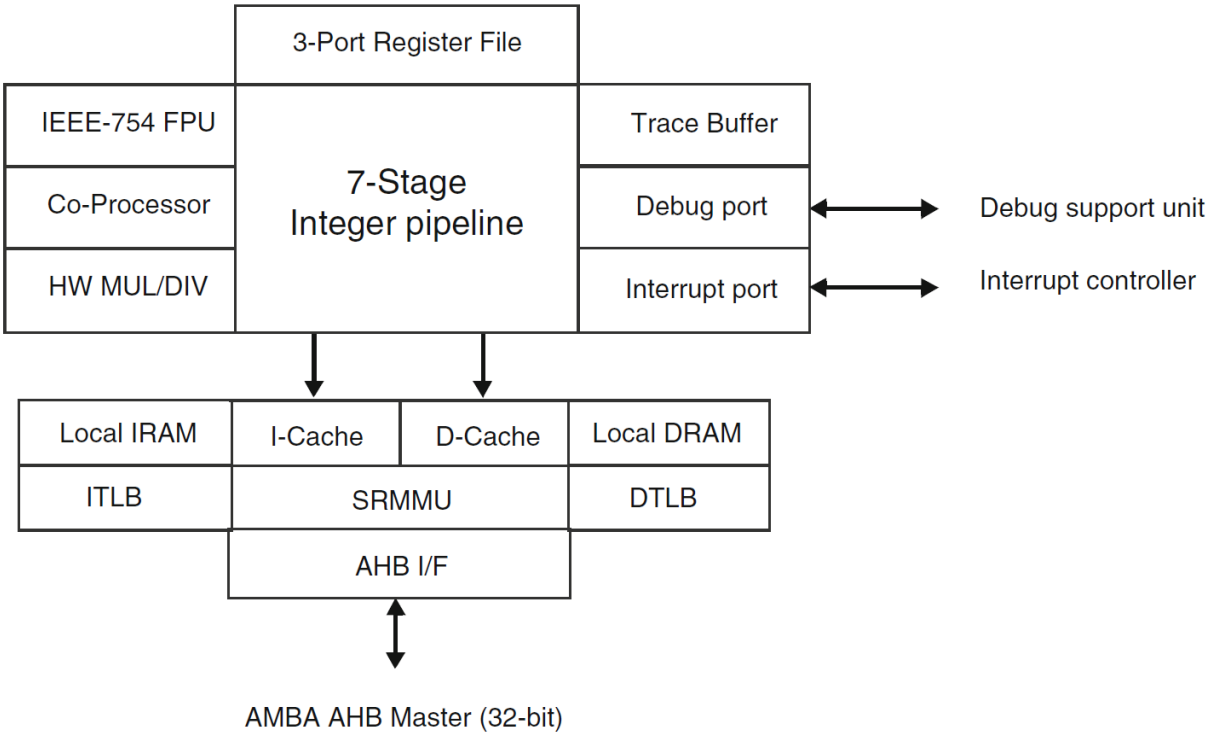
O LEON3 é um processador 32-bit baseado na arquitetura SPARC V8 atualmente mantido pela Cobham Gaisler. Disponibilizado sob a licensa GPL, o código-fonte é na linguagem VHDL e implementado utilizando *generics*[3], o que permite a configuração dos parâmetros que definem algumas características do processador (COBHAM GAISLER, 2016b).

O LEON3 possui um banco de registradores com suporte para 2 a 32 janelas. Cada janela tem acesso a 32 registradores:

- *Global*: 8 registradores acessíveis por todas as janela. Utilizados para armazenamento de valores globais. Registradores 0 a 7.
- *Out*: 8 registradores acessíveis pela janela atual e pela próxima janela. Utilizados para passagem de parâmetros e recebimento de valores de retorno. Registradores 8 a 15.
- *Local*: 8 registradores acessíveis somente pela janela atual. Utilizados para variáveis locais e valores temporários. Registradores 16 a 23.

[3] *Generics* são um recurso da linguagem VHDL que permite a definição de constantes em variáveis que podem ser mapeadas de forma diferente para instâncias de uma mesma entidade.

Figura 4 – Diagrama de blocos interno do LEON3.




Fonte: COBHAM GAISLER (2016a)

- *In*: 8 registradores acessíveis pela janela atual e pela janela anterior. Utilizados para recebimento de parâmetros e retorno de valores. Registradores 24 a 31.

A janela atual é determinada pelo *Current Window Pointer*, um contador de 5 bits contido no registrador *Processor State Register*. O contador é incrementado com a instrução RETT e decrementado com a SAVE. O objetivo dessas janelas é agilizar a troca de contexto, permitindo a mudança de vários registradores para um valor anterior em menos instruções (SPARC International, Inc., 1992).

Ao configurar o sistema antes da síntese do código-fonte também é possível optar pela inclusão de uma unidade de processamento de ponto-flutuante. Duas unidades estão disponíveis, a GRFPU e a GRFPU-Lite, ambas se adequando ao padrão IEEE-754.

Além dessas unidades, como demonstrado na figura 4 são incluídas outras unidades como suporte a depuração, multiplicadores e divisores em hardware, entre outros (COBHAM GAISLER, 2016b).

A unidade de inteiros, que lida com as instruções que não são de ponto-flutuante na arquitetura SPARC, é organizada em um pipeline de 7 estágios:

1. FE (*Instruction Fetch*): A instrução é buscada na cache de instruções. Caso haja um *miss* a devida instrução é buscada no barramento.

2. DE (*Decode*): A instrução é decodificada. O endereço de desvio é calculado.
3. RA (*Register Access*): Os registradores utilizados são lidos do banco ou de alguma unidade de *fowarding*.
4. EX (*Execute*): Operações lógicas e aritméticas são executadas. Endereços de acesso de memória e de retorno de chamada são calculados.
5. ME (*Memory*): Caso a instrução requisite, é feito um acesso à cache de dados. Na ocorrência de um *miss* ou de uma escrita (o tratamento de escritas é *write-through*) o comando é enviado ao barramento principal.
6. XC (*Exception*): Tratamento de excessões e interrupções.
7. WR (*Write*): O resultado da operação lógica ou aritmética é escrito no banco de registradores.

As memórias caches são separadas para instruções e dados. Seus tamanhos, formatos e associatividades são configuráveis. A presença de uma MMU para auxiliar o gerenciamento de memória é opcional. As requisições de memória são realizadas para o barramento AHB.

Um sistema GRLIB utiliza uma barramento segundo o padrão AMBA 2.0. O processador é conectado ao barramento AHB, juntamente com o controlador de memória e outras unidades como entradas para JTAG, USB, Ethernet, entre outros. Também está conectado neste barramento um controlador ponte para o APB. O APB é responsável por controle dos periféricos, como temporizadores e portas de entrada e saída de dados. Todos estes componentes são propriedades intelectuais disponíveis na GRLIB para uso e estudo (COBHAM GAISLER, 2016b).

## 3.4 Desenvolvimento em VHDL

As linguagens de descrição de hardware, ou HDLs, são linguagens de programação criadas com o propósito de permitir a descrição de hardware em texto, utilizando comandos e símbolos. Assim o processo de desenvolvimento de hardware passa a compartilhar várias características com a produção de software.

A grande diferença entre a programação de software e de hardware é a natureza sequencial do processamento de um software. Quando se trabalha com circuitos digitais existem tanto processos paralelos quanto sequenciais, que normalmente são encontrados operando juntamente. O projetista deve considerar os tempos de processamento e os tempos de *setup* e *hold* de cada componente para garantir que unidades lógicas sequenciais operem com os dados corretos. Isso faz com que seja necessário em certos momentos adicionar componentes a mais, como *buffers* e registradores, para que o tempo dos sinais coincida com o necessário.

Apesar de existirem diversas linguagens de descrição de hardware, duas dominam o mercado: VHDL e Verilog. Neste trabalho focaremos no uso da linguagem VHDL. Esta linguagem é a utilizada pelo processador LEON3, o que foi um dos fatores decisivos para sua

escolha. A principal diferença entre o Verilog e o VHDL é a rigidez imposta por este durante o processo de compilação. O VHDL é muito mais restrito em sua sintaxe que o Verilog, o que o torna mais trabalhoso para o desenvolvedor mas permite que muitos erros sejam detectados no momento da compilação, em vez de apenas durante o processo de simulação ou execução.

Além disso, o VHDL também possui melhor suporte para projetos parametrizados, o que torna o *design* mais complexo porém mais flexível, já que algumas alterações em certos valores permitem grandes mudanças no circuito gerado (CHU, 2006). Um exemplo disso será dado na seção 4.3.1.1, onde essa funcionalidade é explorada para configurar diversos aspectos sobre como o processador LEON3 deve ser gerado pelo compilador.

A linguagem pode ser utilizada tanto para a descrição do circuito em si quanto para a criação de *testbenches*, códigos que descrevem como as entradas devem ser definidas ao longo do tempo e que verifica as saídas, servindo para testar os componentes criados. Por isso existem certas funcionalidades da linguagem que não são diretamente ligados à produção de circuitos digitais em si, como tipos para a utilização de arquivos no sistema de arquivos de um sistema operacional.

Os aspectos mais técnicos da linguagem serão abordados na seção 4.2, com exemplos de código e algumas orientações gerais. Este trabalho não busca ser uma fonte extensiva sobre a linguagem, mas apenas um ponto de partida que permite uma compreensão inicial do assunto. A medida do necessário serão introduzidos alguns aspectos mais específicos da linguagem no que se aplicar ao assunto em questão, mas para referências mais completas sobre a linguagem em si, recomendamos a especificação da IEEE (1994)[4]. Algumas outras referências mais didáticas podem ser encontradas nos livros publicados por D'Amore (2005), Chu (2006) e Ashenden (2001).

## 3.5 Circuitos FPGA

Nesta seção será apresentado brevemente o funcionamento de um circuito de FPGA. A intenção é permitir ao leitor a compreensão que mesmo sendo diferente de um circuito impresso, um circuito gravado em FPGA compartilhará de quase todas as suas características apesar de sua diferente configuração.

Devido a sua versatilidade e relativo baixo custo, decidiu-se por utilizar a tecnologia de FPGA para realização dos testes de hardware. Para produção de poucas unidades, tecnologias ASIC (do inglês *Application Specific Integrated Circuits* ou Circuitos Integrados para Aplicação Específica) possuem um custo por unidade muito maior que o uso de FPGA, além desta oferecer a possibilidade de realização de modificações no hardware depois de pronto, o que é impossível naquela (MAXFIELD, 2004).

Chips de FPGA são componentes de hardware programáveis e, apesar de serem mais caros que um CI de produção em massa por unidade possuem um custo inicial praticamente nulo, sendo vantajoso para produção em pequena escala. Porém, apesar de se mostrarem

[4] Apesar de existir um padrão de 2008 para o VHDL que adicionou algumas funcionalidades, o padrão de 1993 atende à maioria das necessidades e é utilizado por possuir maior compatibilidade com diversas ferramentas.

mais lentos que circuitos usando tecnologias ASIC, os circuitos em FPGA possuem um comportamento semelhante a estes, já que placas de FPGA foram feitas para emularem as conexões entre elementos lógicos contidas em um circuito ASIC. Isso os torna ideais para prototipação e testes de hardware, já que é possível testar de forma barata e versátil a funcionalidade e características de um determinado circuito (MAXFIELD, 2004).

Além disso, a comparação de modelos gravados em circuito utilizando uma tecnologia como FPGA permite tirar conclusões para esses modelos, mesmo que sejam utilizados posteriormente de outras maneiras. Cabe ressaltar que a comparação de valores entre projetos de circuito deve considerar a proporcionalidade causada pela diferença de tecnologias (TANENBAUM; ZUCCHI, 2009).

Um circuito de FPGA é composto principalmente de *Lookup Tables* e interconexões programáveis. A replicação desses componentes e a possibilidade de programá-los independentemente dá ao FPGA uma capacidade de criar diversos circuitos lógicos de acordo com os valores dados. As LUTs são combinadas a outros elementos de circuito para compor uma unidade lógica. Essas unidades lógicas tem sua composição e até mesmo nomeclatura definida pelo fabricante. Por exemplo, a empresa Xilinx as denomina "células lógicas", enquanto a Altera as chama "elementos lógicos". Na figura 5 podemos ver um diagrama de um elemento lógico de chips da família Cyclone II produzidos pela Altera, hoje uma subsidiária da Intel.

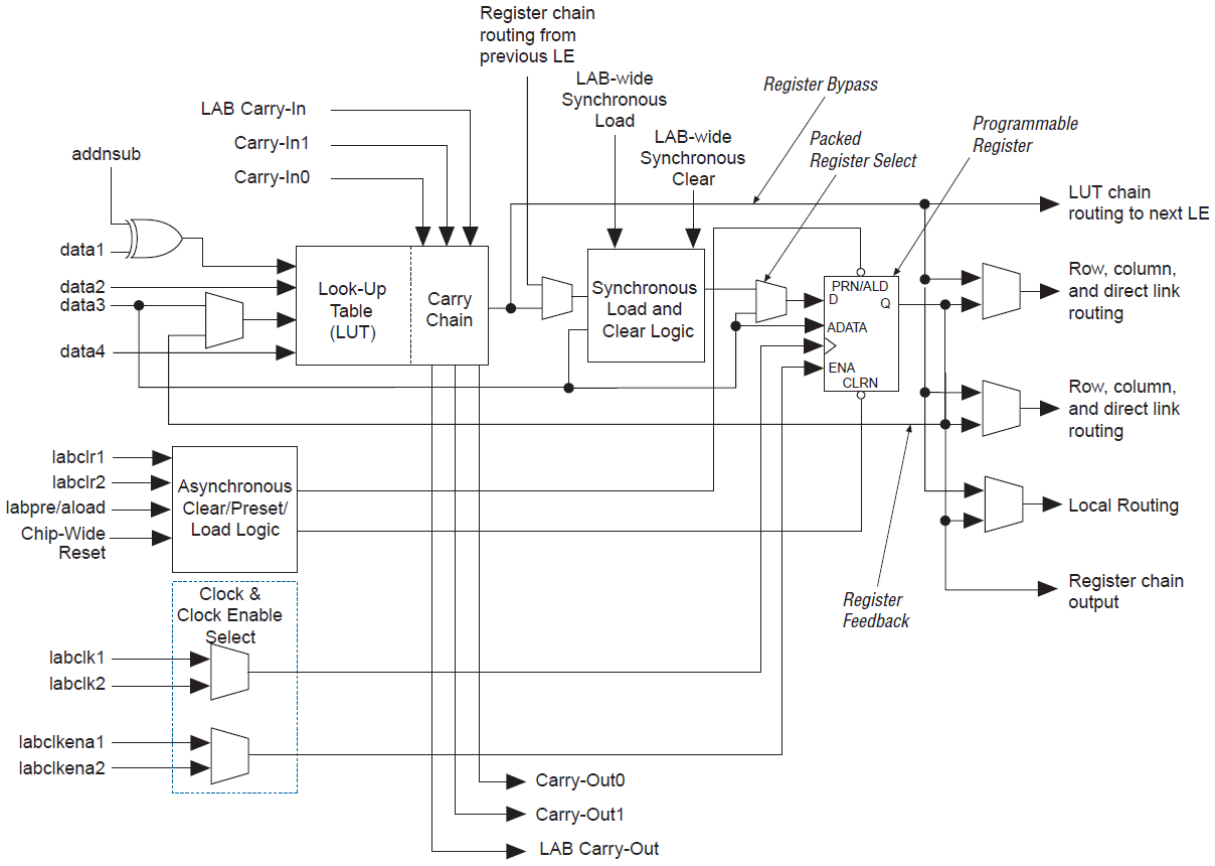
Uma LUT é uma pequena unidade de memória que armazena valores determinados pelo hardware a ser gerado. Ao combinar os pinos de endereçamento formando uma posição de memória, o valor gravado é lido e colocado nos pinos de saída. Dessa forma uma LUT é capaz de simular o funcionamento de qualquer função lógica que possua um número de entradas menor ou igual seu número de pinos de endereçamento. Combinando as entradas e saídas de unidades que contém uma LUT de forma programável permite a emulação de uma infinidade de circuitos lógicos, limitados apenas pela quantidade de elementos lógicos contidos no chip utilizado (TANENBAUM; ZUCCHI, 2009).

Adicionalmente às unidades lógicas, um circuito de FPGA pode conter outros componentes que auxiliem o desenvolvedor ao providenciar um componente de hardware especializado já implementado diretamente no chip, o tornando mais eficiente e garantindo a utilização de um componente garantido pelo fabricante. Como exemplo, podemos citar circuitos de PLL e unidades de memória RAM presentes nos chips da linha Altera Cyclone II (ALTERA, 2007a).

Uma outra tecnologia que buscam proporcionar essa flexibilidade ao projetista de hardware é o CPLD (*Complex Programmable Logic Device*). Porém o CPLD não é indicado para circuitos maiores e complexos por possuir uma estrutura mais centralizada nas ligações entre suas células. Essa característica é compensada pela presença de células mais complexas, capazes de realizar uma maior quantidade de computação (CHU, 2006). Porém em um circuito como o de um microprocessador, em que existem várias unidades lógicas trabalhando em paralelo, essas características são contra-produtivas, fazendo do FPGA uma melhor escolha para esses casos em específico.

Neste trabalho foi feito uso do dispositivo Altera Cyclone II. Neste as LUT possuem 16

Figura 5 – Diagrama de um elemento lógico de chips da família Cyclone II.




Fonte: Altera (2007a)

posições de memória, com quatro pinos para endereçamento. Essas LUT estão contidas em unidades chamadas *Logic Elements*, que são a unidade lógica mínima na arquitetura. Uma LE também possui um registrador programável e suporte para sinal de *feedback*, *clock* e *clear*. Por estarem agrupadas em conjuntos denominados *Logic Array Blocks*, cada LE possui um sinal de entrada *carry in* e um sinal de saída *carry out*, facilitando o uso de conjuntos de LE para operações aritméticas (ALTERA, 2007a).

# 4 Processo de desenvolvimento

*"Quanto mais simples, melhor.*
*Complicações levam a cadeias multiplicativas de efeitos não antecipados."* [1].
*Nassim Nicholas Taleb (Tradução nossa)*

O atual capítulo contém o conteúdo prático sobre o desenvolvimento de hardware utilizando linguagens de programação de hardware, desde a configuração do ambiente à depuração do circuito final. Por ser um capítulo com foco prático, não é necessário conhecimento teórico para sua aplicação direta, porém o desenvolvimento requer conhecimentos prévios de sistemas de lógica digital e de arquitetura de computadores para que seja bem sucedido.

Na seção 4.1 é descrito como se dá a preparação do ambiente necessário para desenvolvimento, compilação e testes dos projetos desenvolvidos. Nesta são listados os softwares necessários para o desenvolvimento, o procedimento de instalação para os mesmos e possíveis dificuldades que o desenvolvedor pode encontrar.

O método de desenvolvimento de unidades lógicas em linguagem de descrição de hardware é descrito na seção 4.2. Nessa seção são abordados conceitos básicos da linguagem VHDL, demonstra-se alguns pontos importantes para o desenvolvimento envolvendo o processador LEON3 e são listadas as unidades lógicas de propriedade intelectual da Altera que auxiliam no desenvolvimento de circuitos mais complexos, explorando melhor certas características do chip alvo.

Na seção 4.3 discorre-se sobre o processo de compilação, demonstrando as configurações necessárias para cada forma de compilação e quais os objetivos cumpridos por cada. Também é demonstrado como transferir o circuito compilado para o dispositivo FPGA.

Por fim, na seção 4.4 é demonstrado como se dá o processo de testes e simulações com o circuito gerado. Nela são apresentadas ferramentas para visualizar o circuito gerado de formas diferentes, como realizar simulações lógicas para averiguar se o *design* proposto produz os resultados esperados, verificar se a implementação realizada atinge às restrições externamente impostas, como tamanho máximo de circuito, tempo de resposta e consumo de energia e simular o circuito em um caso real de uso.

## 4.1 Preparação do ambiente

Nessa seção será coberto como deve ser feito o setup do ambiente com os programas necessários para a realização do desenvolvimento e dos testes de componentes criados em VHDL para serem executados em placas de FPGA Altera, mais especificamente o chip EP2C35F672C6, da linha Cyclone II. Também será abordado como obter o código fonte e fer-

[1] *"The simpler, the better. Complications lead to multiplicative chains of unanticipated effects"*

ramentas adicionais disponibilizadas pela Cobham Gaisler para o desenvolvimento utilizando o processador LEON3.

Para a configuração do ambiente, foi utilizada uma máquina executando o sistema operacional Linux Mint 17.3 Rosa 32 bits, uma distribuição Linux baseada no Ubuntu 14.04.6 LTS. As instruções aqui passadas se aplicam a distribuições Linux baseadas em Debian, como o Ubuntu e o Mint com 32 bits.

Apesar de estar fora do escopo deste trabalho, é possível realizar o desenvolvimento descrito neste trabalho em ambiente Windows. Porém, para que seja possível executar os programas necessários para a utilização do LEON3 é necessário a instalação de um emulador de terminal Linux. O emulador testando e validado pela COBHAM GAISLER (2016a) é o Cygwin.

### 4.1.1 Quartus II

O Quartus II é o IDE (*Integrated Development Environment*, ou ambiente de desenvolvimento integrado) oficial da Altera para o desenvolvimento de soluções envolvendo seus dispositivos e propriedades intelectuais. O Quartus permite ao desenvolvedor realizar a configuração, desenvolvimento, compilação, análise e transferência para o chip através de sua interface gráfica.

Para trabalhar com a família Quartus II é precisa utilizar a versão 13.0 do Quartus II, já que versões posteriores não possuem suporte para estes dispositivos. O download deste pode ser realizado no caminho <http://fpgasoftware.intel.com/13.0sp1/> ou através de um disco pelo fabricante. O kit de desenvolvimento DE2 foi utilizado neste trabalho e a instalação foi realizada a partir do disco contido.

#### 4.1.1.1 Instalação do Quartus II

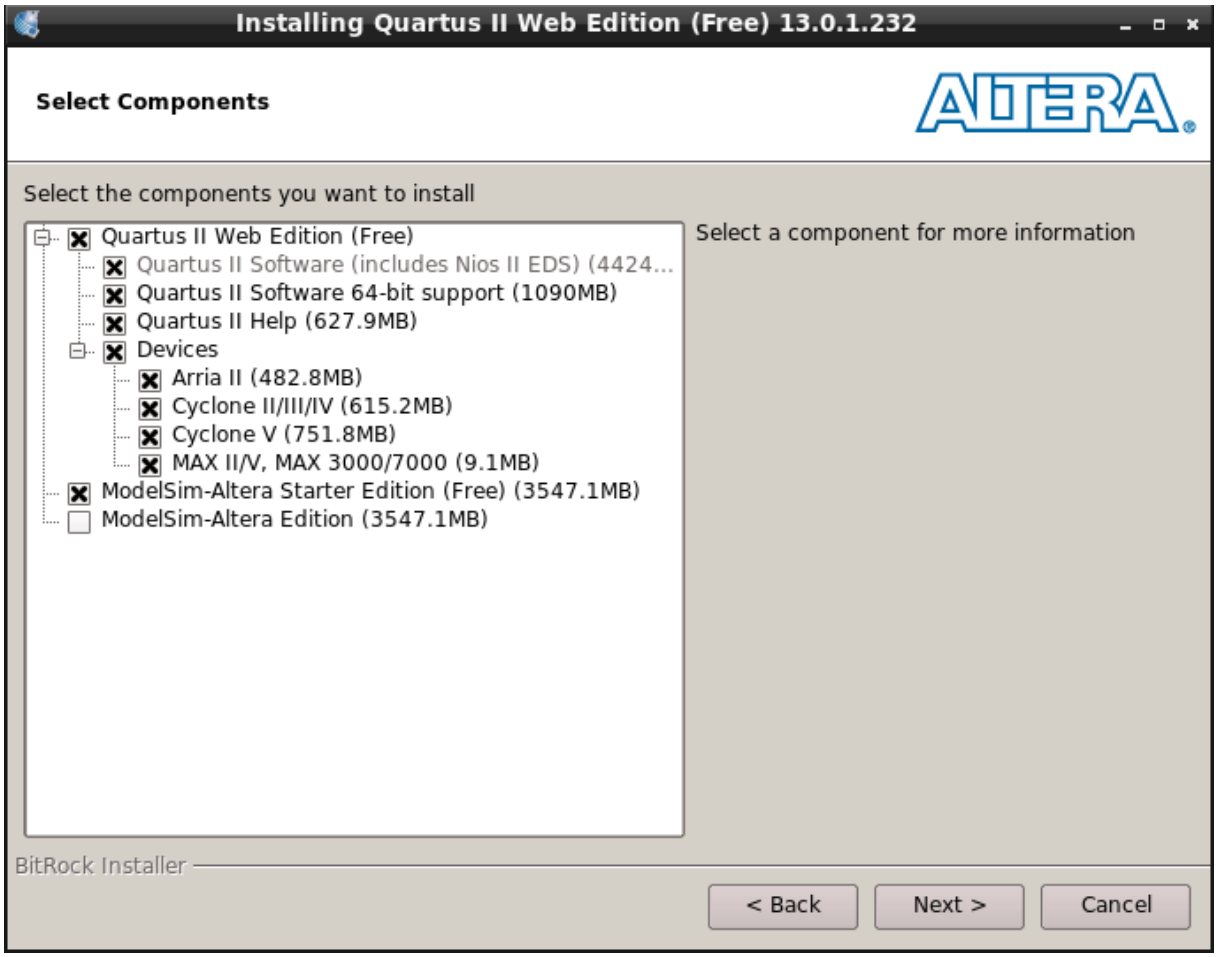
Para realizar a instalação do Quartus II, primeiramente deve se extrair o arquivo contendo o instalador para uma pasta local. Após realizar a extração, deve-se executar o script `setup.sh`. Para realizar a execução deste, é necessário primeiro dar as permissões de leitura, escrita e execução necessárias. Isso é realizado executando o comando a seguir:

```
chmod 744 components/QuartusSetupWeb-13.0.1.232.run
chmod 744 components/QuartusHelpSetup-13.0.1.232.run
chmod 744 components/ModelSimSetup-13.0.1.232.run
chmod 744 setup.sh
```

Durante a instalação do Quartus, entre outras opções o usuário será questionado sobre as funcionalidades a serem adicionadas pelo instalador, como pode ser visto na figura 6. Além das funcionalidades básicas do Quartus, é necessário instalar o suporte aos dispositivos Altera a serem utilizados, que no caso deste trabalho é a família Cyclone. Também deve ser instalado o Modelsim-Altera para a realização de simulações mais complexas. O *Modelsim-Altera Starter Edition* possui limitações no desempenho e no tamanho dos *scripts* a serem executados porém pode ser utilizado gratuitamente, diferente do *Modelsim-Altera Edition* que requer uma licença.

Caso o usuário esteja utilizando um sistema operacional de 64 bits, é necessário

Figura 6 – Captura de tela com as opções durante a instalação do Quartus II. Além das funcionalidades básicas, estão selecionados os suportes aos dispositivos Altera disponíveis e a versão gratuita do Modelsim-Altera.




Fonte: Altera (2013)

instalar algumas dependências do software em sua versão 32 bits. Para isso, basta executar o comando abaixo:

```
sudo apt-get install libstdc++6:i386 libc6:i386 libx11-dev:i386
libxext-dev:i386 libxau-dev:i386 libxdmcp-dev:i386 libfreetype6:i386
fontconfig:i386 expat:i386 libsm6:i386.
```

#### 4.1.1.2 Adição dos caminhos ao PATH

Para o correto funcionamento do software, é necessário adicionar alguns caminhos à variável de ambiente `PATH` do sistema operacional. Essa tarefa é realizada executando os comandos abaixo, onde o valor `<CAMINHO DA INSTALAÇÃO>` deve ser substituído pela pasta raiz da do Quartus especificada no momento da instalação.

```
export PATH=$PATH:<CAMINHO DA INSTALAÇÃO>/quartus/bin
export PATH=$PATH:<CAMINHO DA INSTALAÇÃO>/modelsim_ase/bin
```

```
export QUARTUS_ROOTDIR=<CAMINHO DA INSTALAÇÃO>/quartus/
```

Devido às características de sistemas operacionais Linux, estes comandos serão aplicados somente para a sessão atual. Em distribuições baseadas em Debian é possível designar comandos para serem executados automaticamente ao iniciar uma nova sessão ao os adicionar ao arquivo `~/.bashrc`.

### 4.1.2 Modelsim

O Modelsim é um software desenvolvido pela Mentor Graphics para a simulação de códigos desenvolvidos em HDL, como VHDL e Verilog (MENTOR GRAPHICS, 2012). Para a simulação temporal de circuitos e simulação de circuitos complexos seu uso é recomendado em vez do uso do simulador contido no Quartus (ALTERA, 2007d).

A Altera disponibiliza uma versão do Modelsim com suas propriedades intelectuais adicionadas em forma de bibliotecas por padrão, chamado Modelsim Altera. Esse software acompanha o Quartus e pode ser invocado de dentro deste para a realização de simulações lógicas e temporais (ALTERA, 2013).

O Modelsim-Altera pode ser instalado pelo instalador do Quartus II. Porém, algumas peculiaridades do sistema Linux faz com que ajustes adicionais sejam necessários para seu correto funcionamento.

#### 4.1.2.1 Resolução de problemas

Para o funcionamento correto do Modelsim, é necessário realizar a instalação da fonte `FreeType 2.5.0.1-1`. Para isso, primeiro deve ser feito o download do código fonte desta, disponível em <https://download.savannah.gnu.org/releases/freetype/>.

Após extrair o código-fonte, vá até o diretório extraído e execute os seguintes comandos para realizar a compilação da fonte:

```
./configure -build=i686-pc-linux-gnu "CFLAGS=-m32CXXFLAGS=-m32LDFLAGS=-m32"
make -j8
```

O resultado da compilação estará no diretório `./objs/.libs`. Como em sistemas baseados em Linux pastas iniciadas em '.' são ocultas é possível que a pasta não seja visível ao acessar via interface gráfica ou no terminal dependendo das configurações do sistema. Para acessar os arquivos compilados da fonte, deve se alterar as opções do visualizador de arquivos para exibir arquivos ocultos ou realizar a cópia pelo terminal utilizando o comando `cp`.

É necessário copiar os arquivos `libfreetype.so`, `libfreetype.so.6` e `libfreetype.so.6.10.1` para a pasta `lib32` dentro do diretório de instalação do Quartus. Após realizar estes passos deve-se criar uma variável de ambiente chamada `LD_LIBRARY_PATH` com o caminho onde foram colocados os arquivos compilados. Essa variável pode ser criada com o comando `export`. Este comando pode ser adicionado ao arquivo `~/.bashrc` como realizado na seção 4.1.1.2.

Por fim, para garantir o funcionamento da conexão com o dispositivo via JTAG, é ne-

cessário criar um arquivo de extensão `.rules` no diretório `/etc/udev/rules.d/`. O conteúdo desse arquivo deve ser preenchido com a seguinte instrução:

```
SUBSYSTEM=="usb",\
ENVDEVTYPE=="usb_device",\
ATTRidVendor=="09fb",\
ATTRidProduct=="6001",\
MODE="0666",/
NAME="bus/usb/$envBUSNUM/$envDEVNUM",\
RUN+="/bin/chmod 0666 %c"
```

#### 4.1.2.2 Adição dos caminhos ao PATH

O Modelsim-Altera instalado possui seus arquivos executáveis dentro do diretório de instalação do Quartus. O caminho com os binários do Modelsim deve ser adicionado à variável de ambiente `PATH` para a execução do comando `vsim`, utilizado para realizar compilações com o Modelsim. O caminho dos executáveis é `<DIRETÓRIO DO QUARTUS>/modelsim_ase/bin`.

Para que o Quartus possa iniciar o Modelsim-Altera para realizar as simulações, é necessário alterar uma configuração neste. Dentro do Quartus, no menu `Tools`, na opção `EDA Tool Options` o campo `Modelsim-Altera` deve ser preenchido com o diretório `<DIRETÓRIO DO QUARTUS>/modelsim_ase/linuxaloem/`.

### 4.1.3 Instalação das ferramentas de desenvolvimento do LEON

A Cobham Gaisler disponibiliza uma série de ferramentas com o objetivo de permitir o desenvolvimento de software para o processador LEON3. Para o desenvolvimento utilizando o LEON em um dispositivo físico de hardware não é necessário a obtenção de simuladores para o mesmo, se fazendo necessário então os softwares GRMON, BCC e MKPROM.

O GRMON é um depurador para o LEON que permite a comunicação de um computador com o hardware contendo o processador através do barramento AMBA. Essa conexão pode ser feita via utilizando uma gama de entradas e protocolos, entre os quais cabe destacar o USB e o JTAG. Utilizando o GRMON é possível carregar programas à memória, executar sua depuração passo a passo, verificar o valor de registradores e de seções de memória, enviar e receber entradas e saídas de dados e alterar informações no processador em tempo real (COBHAM GAISLER, 2018b).

O GRMON pode ser obtido através do endereço <https://www.gaisler.com/anonftp/grmon/>. A licença gratuita deste software permite somente a utilização para pesquisas acadêmicas ou avaliação antes da compra, sendo esta válida por 21 dias.

A instalação do GRMON consiste em extrair os arquivos obtidos no site da Cobham Gaisler em um diretório da preferência do usuário. Recomenda-se extrair o conteúdo para o diretório `/opt/` e adicionar o diretório contendo os binários do programa à variável de ambiente `PATH`, como realizado na seção 4.1.1.2.

O BCC é um conjunto de ferramentas baseadas no GNU GCC, G++ e GDB para per-

mitir a compilação e depuração de códigos escritos em C e C++ para um binário compatível com a arquitetura SPARC utilizada no processador LEON. Sendo assim, um código C11 ou C++11 pode ser compilado e executado no LEON, seja através do GRMON para execução simples e depuração ou execução no boot através de gravação em armazenamento persistente (COBHAM GAISLER, 2017).

O BCC pode ser obtido através do endereço <https://www.gaisler.com/anonftp/bcc2/>. Ele é distribuído sob a licença GPL, portanto pode ser utilizado gratuitamente considerando que qualquer código-fonte que utilize parte do fonte disponibilizado do BCC também deve ter seu código-fonte distribuido livremente como requerido pela licença.

O MKPROM é uma ferramenta que permite a conversão de um binário compilado para o LEON em uma imagem bootável que pode ser gravada em uma memória ROM, PROM, MRAM ou FLASH, que será carregado para a memória RAM ao inicializar o circuito, permitindo o funcionamento independente do processador para execução do programa gravado (COBHAM GAISLER, 2018c).

O MKPROM pode ser obtido através do endereço <https://www.gaisler.com/anonftp/mkprom2/>. Ele é distribuído sob a licença GPL, seguindo então as mesmas restrições às quais o BCC está sujeito.

Tanto o BCC quanto o MKPROM possuem o processo de instalação semelhante ao descrito para o GRMON. Os mesmos passos e observações se aplicam ao processo de preparação para todos os três softwares.

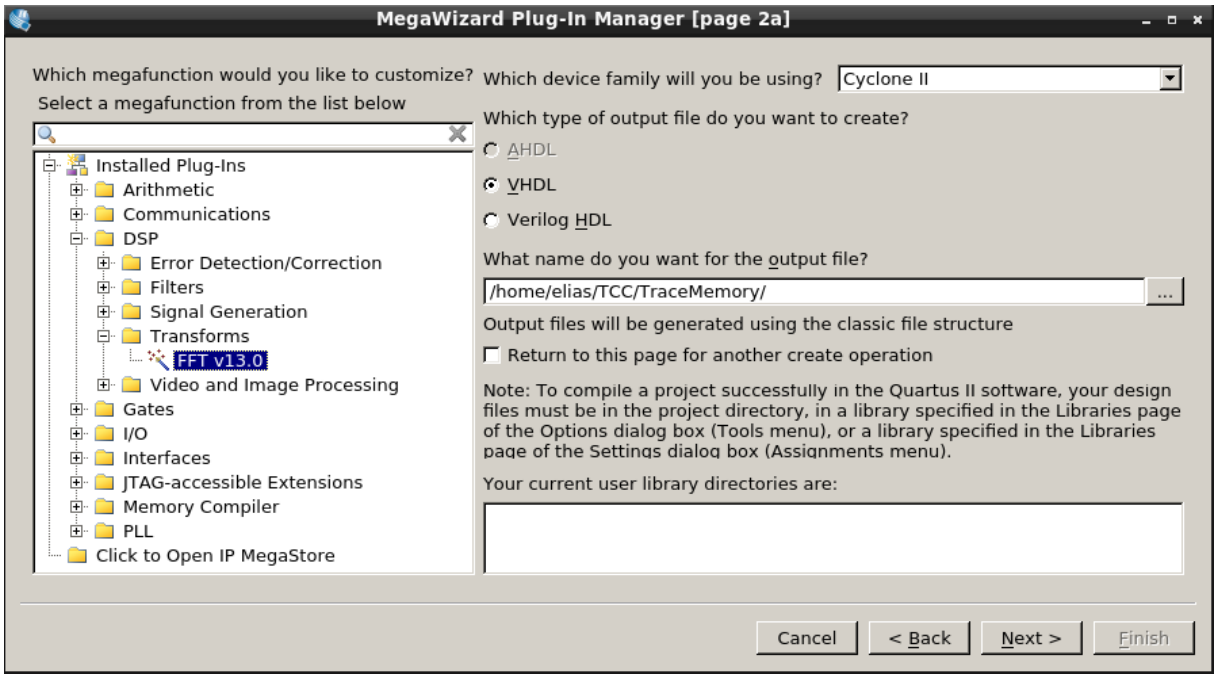
### 4.1.4 LEON3

O LEON3 é disponibilizado através de um arquivo compactado contendo todos os arquivos necessários para sua alteração e compilação. Esse arquivo pode ser obtido no endereço <https://www.gaisler.com/index.php/downloads/leongrlib>. Para a utilização da unidade de pontos flutuantes, é necessário obter um pacote de arquivos separados, disponibilizados na mesma página. Após realizar o *download* dos arquivos, basta extraí-los em um diretório qualquer para que possam ser utilizados.

## 4.2 Desenvolvimento

Nessa seção serão abordados alguns conceitos importantes para o desenvolvimento de circuitos em linguagem de descrição de hardware, explorando alguns recursos da linguagem VHDL e pontos que devem ser considerados ao implementar um projeto de circuito. Essa seção não possui como objetivo ser um guia completo e extensivo da linguagem, tocando somente alguns conceitos de nível mais alto na implementação. Uma listagem de referências bibliográficas específicas e aprofundadas sobre o desenvolvimento em VHDL pode ser encontrada na seção 3.4.

Figura 7 – Captura de tela do assitente para *megafuntions* do Quartus II. No exemplo, dentre os componentes disponíveis foi selecionado o componente que realiza a transformada rápida de Fourier em um sinal.




Fonte: Altera (2013)

### 4.2.1 Biblioteca de componentes Altera

A Altera disponibiliza para os desenvolvedores um conjunto de componentes comuns em sistemas mais complexos. Esses componentes, chamados *megafunctions* são propriedades intelectuais da Altera, desenvolvidos e otimizados especificamente para o hardware por ela produzido. A vantagem da utilização dessas *megafunctions* é, além do ganho de tempo ao já estar pronto um componente testado e otimizado, o fato desses componentes serem altamente parametrizados.

O software Quartus II possui um assistente para o uso de *megafunctions*. No menu *Tools*, a opção *MegaWizard Plug-in Manager*. A figura 7 demonstra a etapa de seleção de componente no assistente, enquanto a figura 8 contém o código fonte gerado para a *megafunction altsyncram* da Altera, um componente parametrizado que funciona como uma unidade de memória síncrona. Ao utilizar esse componente, o sintetizador tenta otimizar o circuito e utilizar unidades de memória presentes no chip FPGA Altera se possível, como as unidades de memória M4K contidas em alguns de seus dispositivos. Uma lista completa das *megafunctions* disponíveis no Quartus II 13.0 por padrão pode ser vista na figura 30, contida nos anexos deste trabalho.

Para a utilização do componente, o usuário deve mapear tanto as portas quanto os valores genéricos afim de que o compilador possa instanciar o componente de acordo com as especificações. Para saber como devidamente preecher esses valores é necessária a leitura

Figura 8 – Exemplo de código mapeando uma unidade de memória a uma *megafunction* existente, a *altsyncram* da Altera.

```
ENTITY altsyncram_example IS
   PORT (
      Clock      : IN    STD_LOGIC := '1';
      WAddress   : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
      WData      : IN    STD_LOGIC_VECTOR (31 DOWNTO 0);
      WEnable    : IN    STD_LOGIC := '0';
      RAddress   : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
      RData      : OUT   STD_LOGIC_VECTOR (31 DOWNTO 0)
   );
END altsyncram_example;

ARCHITECTURE SYN OF altsyncram_example IS
   COMPONENT altsyncram
      GENERIC (
         address_reg_b                       : STRING;
         clock_enable_input_a                : STRING;
         clock_enable_input_b                : STRING;
         clock_enable_output_a               : STRING;
         clock_enable_output_b               : STRING;
         intended_device_family              : STRING;
         lpm_type                            : STRING;
         numwords_a                          : NATURAL;
         numwords_b                          : NATURAL;
         operation_mode                      : STRING;
         outdata_aclr_b                      : STRING;
         outdata_reg_b                       : STRING;
         power_up_uninitialized              : STRING;
         read_during_write_mode_mixed_ports  : STRING;
         widthad_a                           : NATURAL;
         widthad_b                           : NATURAL;
         width_a                             : NATURAL;
         width_b                             : NATURAL;
         width_byteena_a                     : NATURAL
         );
         PORT (
         address_a  : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
         clock0     : IN    STD_LOGIC;
         data_a     : IN    STD_LOGIC_VECTOR (31 DOWNTO 0);
         q_b        : OUT   STD_LOGIC_VECTOR (31 DOWNTO 0);
         wren_a     : IN    STD_LOGIC;
         address_b  : IN    STD_LOGIC_VECTOR (7 DOWNTO 0)
      );
   END COMPONENT;
```

da documentação do componente. O trecho de código a seguir demonstra um exemplo de mapeamento do componente *altsyncram*, da própria Altera. As ideias por trás da criação dessa entidade serão expandidas na seção 4.2.2.

Figura 8 – (continuação)

```
BEGIN
   altsyncram_component : altsyncram
   GENERIC MAP (
      address_reg_b                       => "CLOCK0",
      clock_enable_input_a                => "BYPASS",
      clock_enable_input_b                => "BYPASS",
      clock_enable_output_a               => "BYPASS",
      clock_enable_output_b               => "BYPASS",
      intended_device_family              => "Cyclone II",
      lpm_type                            => "altsyncram",
      numwords_a                          => MemoTableT_WayLenght,
      numwords_b                          => MemoTableT_WayLenght,
      operation_mode                      => "DUAL_PORT",
      outdata_aclr_b                      => "NONE",
      outdata_reg_b                       => "CLOCK0",
      power_up_uninitialized              => "FALSE",
      read_during_write_mode_mixed_ports => "OLD_DATA",
      widthad_a                           => 8,
      widthad_b                           => 8,
      width_a                             => 32,
      width_b                             => 32,
      width_byteena_a                     => 1
   )
   PORT MAP (
      address_a  => WAddress,
      clock0     => Clock,
      data_a     => WData,
      wren_a     => WEnable,
      address_b  => RAddress,
      q_b        => RData
   );
END SYN;
```

Fonte: Elaborado pelo autor.

### 4.2.2 Criação de unidade lógica

A estrutura básica de um programa VHDL é a entidade (*entity*). Assim como a classe em uma linguagem orientada a objetos, a entidade exprime as propriedades básicas de um componente de circuito. O objetivo da entidade, assim como o da classe, é expressar um trecho lógico em um sistema de caixa preta, onde a complexidade da implementação é abstraída para os elementos externos, que devem se preocupar apenas com a interface do componente.

Uma entidade VHDL é composta por três elementos:

- Nome da entidade: O nome da entidade define como ela será referenciada em outros lugares do código, tanto para definir uma implementação desta como para a instanciar como componente dentro de outro elemento;
- Portas de entrada e saída: Define todas as informações compartilhadas entre a implementação interna e externa ao componente. Essas informações podem ser sinais ló-

Figura 9 – Exemplo de código declarando uma entidade chamada “Coincidencia". Esta possui duas entradas, a e b, que representam sinais digitais de dois bits e uma saída, q, que representa um sinal digital de um bit. O VHDL não permite caracteres especiais em seu código, por isso a acentuação foi omitida dos comentários.

```
library ieee ;                -- Biblioteca com diversas definicoes basicas
use ieee.std_logic_1164.all;  -- Especifica o uso dos tipos std_logic

entity Coincidencia is                        -- Declara entidade
   port (                                     -- Declara interface do componente
      a : in std_logic_vector(1 downto 0);    -- Sinal logico de 2 bits (1 a 0)
      b : in std_logic_vector(1 downto 0);    -- Sinal logico de 2 bits (1 a 0)
      q : out std_logic                       -- Sinal logico de 1 bit
   );
end Coincidencia;                             -- Encerra a declaracao da entidade
```

Fonte: Elaborado pelo autor.

gicos, valores ou entidades abstratas. Para a síntese física é necessário que essas informações sejam conversíveis em sinais lógicos utilizando alguma codificação. Caso contrário, o componente só poderá ser utilizado em simulações;

- Parâmetros genéricos: Define constantes a serem utilizadas pela implementação interna que podem variar de acordo com o valor associado pela implementação externa. Essas constantes permitem a criação de hardware configurável, permitindo o maior reuso de componentes.

A implementação da entidade definida nessa declaração será demonstrada a seguir. Em VHDL existem duas abordagens para a implementação de um circuito: implementação estrutural e implementação comportamental. Essas serão demonstradas nas seções 4.2.2.1 e 4.2.2.2 respectivamente. Ambas as abordagens podem ser utilizadas em um mesmo componente, o que faz dessa distinção não algo inerente à linguagem mas sim uma maneira de conceitualizar formas de descrição de hardware.

#### 4.2.2.1 Implementação estrutural

A implementação estrutural de uma unidade lógica busca descrever o circuito em si. O projetista deve realizar um esforço maior neste formato, já que precisa projetar o hardware em seu estado final, em forma de circuito.

A figura 10 contém um exemplo de código VHDL implementando a entidade “Coincidência” declarada na seção 4.2.2 em uma implementação estrutural[2]. Pode-se observar como o foco da implementação se dá na organização do hardware. O projetista criou o circuito neces-

[2] O operador de igualdade em VHDL já implementa um circuito concidência e deve ser utilizado preferencialmente a uma implementação customizada. Este componente é utilizado apenas como exemplo de um circuito representado em VHDL.

Figura 10 – Exemplo de código implementando a entidade ”Coincidencia” utilizando a abordagem estrutural.

```
library ieee ;                -- Biblioteca com diversas definicoes basicas
use ieee.std_logic_1164.all;  -- Especifica o uso dos tipos std_logic

entity Coincidencia is                        -- Declara entidade
   port (                                     -- Declara interface do componente
      a : in std_logic_vector(1 downto 0);    -- Sinal logico de 2 bits (1 a 0)
      b : in std_logic_vector(1 downto 0);    -- Sinal logico de 2 bits (1 a 0)
      q : out std_logic                       -- Sinal logico de 1 bit
   );
end Coincidencia;                             -- Encerra a declaracao da entidade

-- Declara uma implementacao do componente Coincidencia
architecture Coincidencia_arch of Coincidencia is
   -- Declara 2 sinais de 1 bit a serem utilizados internamente na implementacao
   signal p0, p1 : std_logic;
begin                                    -- Inicia a implementacao do circuito
   -- Associa ao sinal ‘p0’ a funcao ‘a(0) exor b(0)’
   p0 <= (a(0) and b(0)) or ((not a(0)) and (not b(0)));
   -- Associa ao sinal ‘p1’ a funcao ‘a(1) exor b(1)’
   p1 <= (a(1) and b(1)) or ((not a(1)) and (not b(1)));
   -- Associa ao sinal ‘q’ a funcao ‘p1 and p2’
   q  <= p0 and p1;
end Coincidencia_arch;                   -- Encerra a declaracao da implementacao
```

Fonte: Elaborado pelo autor.

sário para implementar determinada lógica e transcreveu o circuito resultante para a linguagem de descrição.

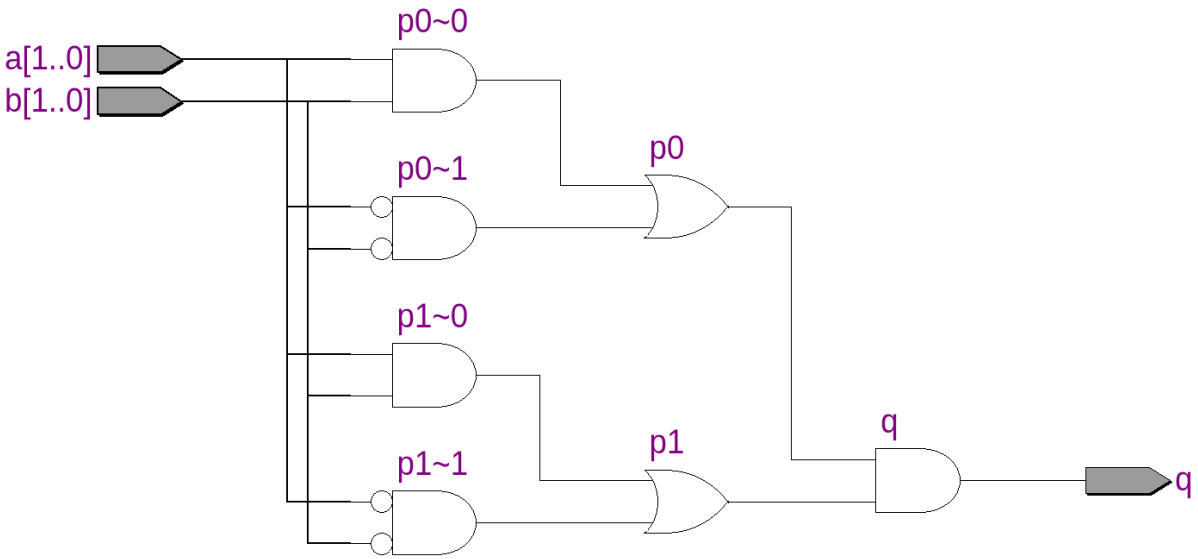
Na figura 11 podemos ver a representação esquemática utilizando portas lógicas do circuito gerado pelo projetista. Apesar do otimizador atuar sobre o circuito, seu comportamento é mais limitado já que alterações no circuito podem ter efeitos colaterais não desejados pelo programador. Por exemplo, a substituição do código `"(a(0) and b(0)) or ((not a(0)) and (not b(0)))"` por `"not(a(0) xor b(0))"` possui consequências no comportamento temporal do circuito.

#### 4.2.2.2 Implementação comportamental

A implementação comportamental de uma unidade lógica busca descrever o comportamento do circuito e deixa a definição específica do hardware a cargo do sintetizador. O projetista aqui se preocupa mais com o algoritmo necessário para que o componente processe os dados corretamente, deixando a transformação da lógica em circuito a cargo do sintetizador.

Essa implementação é util por permitir ao programador expressar ideias de forma mais enxuta. Por exemplo, em vez de descrever por completo o funcionamento de um somador, ele pode usar o operador de soma (por exemplo, `"q <= a + b"`). Outra vantagem é que essa

Figura 11 – Circuito gerado pelo sintetizador a partir do código contido na figura 10.




Fonte: Elaborado pelo autor.

descrição permite uma maior otimização pelo sintetizador, já que este possui um foco maior nos resultados produzidos pelo algoritmo.

A figura 12 contém um exemplo de código VHDL implementando a entidade “Coincidência” declarada na seção 4.2.2 em uma implementação comportamental. Pode-se observar como o foco da implementação se dá no comportamento esperado do componente. O projetista criou o algoritmo necessário para implementar determinada lógica, o descreveu utilizando os recursos para lógica sequencial da linguagem de descrição e o sintetizador converteu os resultados esperados em um circuito lógico. Por ser mais focado em resultados, vemos na figura 13 um circuito mais compacto capaz de produzir os mesmos resultados do circuito criado na seção 4.2.2.1.

Porém alguns pontos importantes devem ser observados com relação à descrição comportamental de um componente. A primeira questão é a capacidade limitada do sintetizador em converter lógica sequencial para um circuito digital. Em componentes voltados para a síntese o desenvolvedor deve evitar a utilização de algoritmos com lógica complexa, como por exemplo diversas operações em laços de repetição. Caso contrário haverá uma falha, com o compilador informando ao usuário que o circuito descrito não é sintetizável (CHU, 2006).

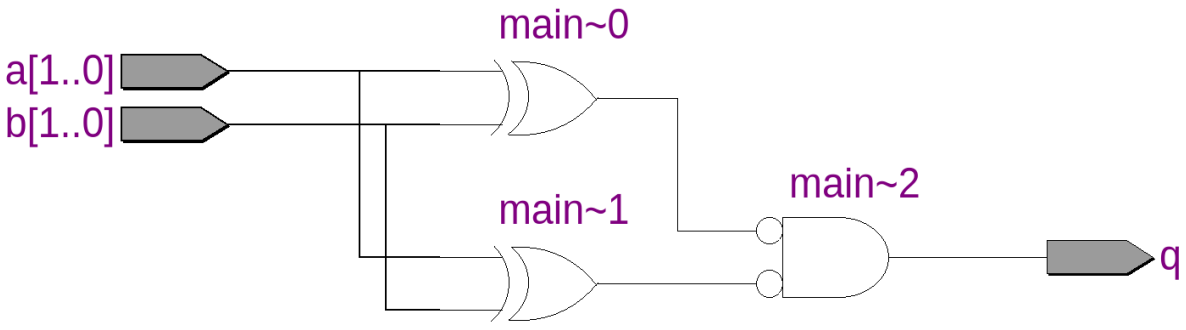
O segundo ponto é que em uma implementação comportamental o programador possui menor controle sobre o comportamento temporal do circuito. Apesar da linguagem VHDL permitir ao usuário especificar relações de delay entre sinais com o comando `"after"`, esse comando possui aplicação somente durante simulações, sendo ignorado no momento da síntese de circuito físico (CHU, 2006).

Figura 12 – Exemplo de código declarando uma entidade chamada "Coincidencia". Esta possui duas entradas, a e b, que representam sinais digitais de dois bits e uma saída, q, que representa um sinal digital de um bit. O VHDL não permite caracteres especiais em seu código, por isso a acentuação foi omitida dos comentários.

```
library ieee ;                -- Biblioteca com diversas definicoes basicas
use ieee.std_logic_1164.all;  -- Especifica o uso dos tipos std_logic

entity Coincidencia is                          -- Declara entidade
   port (                                       -- Declara interface do componente
      a : in std_logic_vector(1 downto 0);  -- Sinal logico de 2 bits (1 a 0)
      b : in std_logic_vector(1 downto 0);  -- Sinal logico de 2 bits (1 a 0)
      q : out std_logic                         -- Sinal logico de 1 bit
   );
end Coincidencia;                               -- Encerra a declaracao da entidade

-- Declara uma implementacao do componente Coincidencia
architecture Coincidencia_arch of Coincidencia is
begin                                    -- Inicia a implementacao do circuito
   -- Inicia um processo sequencial ativado por mudancas em 'a' e 'b'
   main: process(a, b)
      -- Declara uma variavel 'q0' que armazena um valor de sinal logico
      variable q0: std_logic;
   begin           -- Inicia a implementacao do processo sequencial
-- Inicia uma estrutura condicional com a condicao dos sinais a e b serem iguais
      if a(0) = b(0) and a(1) = b(1) then
         q0 := '1';       -- Associa o valor logico alto a variavel 'q0'
      else                -- Caso contrario
         q0 := '0';       -- Associa o valor logico baixo a variavel 'q0'
      end if;             -- Encerra a estrutura condicional
      q <= q0;            -- Associa o valor resultante de 'q0' a saida 'q'
   end process main;      -- Encerra o processo sequencial
end Coincidencia_arch;    -- Encerra a declaracao da implementacao
```

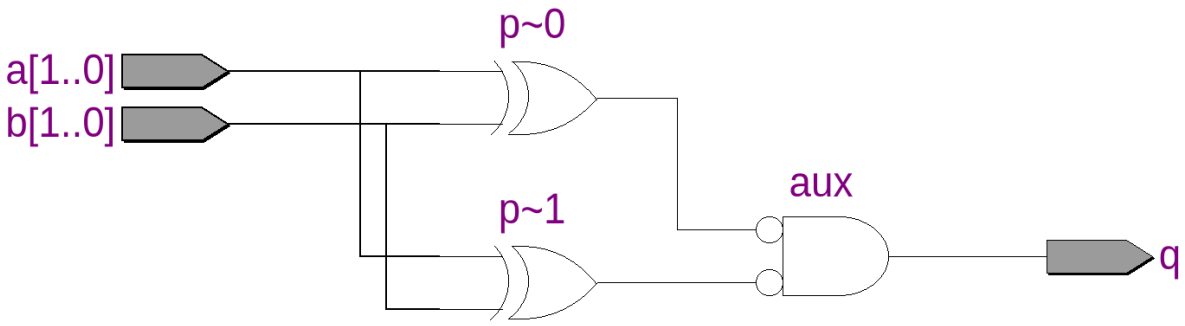
Fonte: Elaborado pelo autor.

Figura 13 – Circuito gerado pelo sintetizador a partir do código contido na figura 12.




Fonte: Elaborado pelo autor.

Figura 14 – Circuito coincidência de 2 bits gerado pelo sintetizador a partir do código contido na figura 15. O componente instanciador definiu o parâmetro genérico ‘n’ com valor 2.




Fonte: Elaborado pelo autor.

#### 4.2.2.3 Implementação mista

Tanto a implementação estrutural descrita na seção 4.2.2.1 quanto a implementação comportamental descrita na seção 4.2.2.2 possuem aplicações em que se mostram mais práticas para o programador. Essas abordagens representam apenas duas formas de descrever o circuito utilizando os recursos da linguagem.

Durante o desenvolvimento, o programador deve utilizar a ferramenta que lhe proporcionar melhor resultado dentro das suas necessidades. Isso pode significar a utilização de ambas as abordagens em um mesmo componente, sendo cada uma aplicada a um trecho da lógica que possuia melhor capacidade para descrever.

#### 4.2.2.4 Desenvolvimento parametrizado

Uma característica da linguagem VHDL que a torna extremamente versátil é a possibilidade de gerar projetos parametrizados. Para tal feito, é utilizado um conceito da linguagem denominado *generics*. Os *generics* são constantes que podem ser utilizadas ao longo de toda a definição e implementação de um componente e que podem ser alterados pelo componente instanciador.

Esse recurso, apesar de deixar a implementação de um componente menos clara e em alguns casos menos otimizada, é uma poderosa ferramenta para o projetista. Com a utilização de *generics* é possível a criação de componentes mais robustos e que atendem a circunstâncias diversas, aumentando o reuso e a confiabilidade do código por reduzir a quantidade de pontos de falha, desde que o componente tenha sido testado de forma satisfatória.

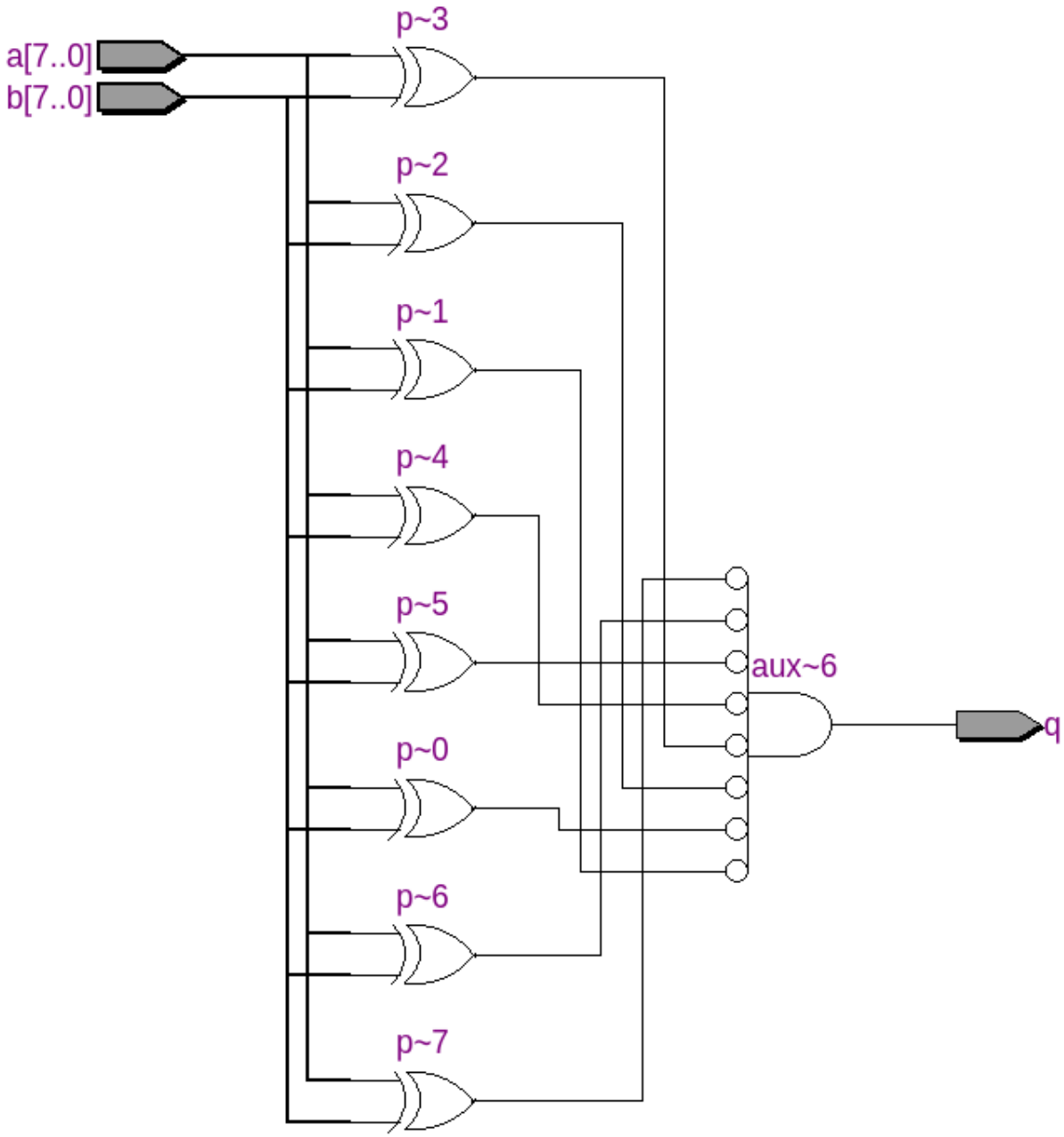
A figura 15 contém uma implementação parametrizada do componente “Coincidencia”. As implementações realizadas nas figuras 10 e 12 são eficazes em sua proposta, porém possuem a limitação de serem capazes de realizar a operação somente para parâmetros de entrada de 2 bits. Caso o usuário deseje reutilizar o componente, porém com entradas de

Figura 15 – Exemplo de código declarando uma entidade chamada "CoincidenciaGenerico". Esta entidade possui funcionamento semelhante às criadas no código contido nas figuras 10 e 12, com a diferença de poder se utilizada para comparar vetores de entrada de qualquer tamanho, já que o tamanho é parametrizado e o tamanho é mapeado de acordo com o valor definido ao instanciar o componente.

```
library ieee ;
use ieee.std_logic_1164.all;

entity CoincidenciaGenerico is
   -- Inicia o mapeamento de variaveis genericas
   generic (
      -- Declara uma variavel inteira com valor padrao igual a 2
      n : integer := 2
   );
   port (
      -- Sinal logico de 'n' bits (numerados n-1 a 0)
      a : in std_logic_vector(n-1 downto 0);
      b : in std_logic_vector(n-1 downto 0);
      q : out std_logic
   );
end CoincidenciaGenerico;

architecture CoincidenciaGenerico_arch of CoincidenciaGenerico is
   -- Cria n sinais auxiliares em um vetor
   signal p : std_logic_vector(n-1 downto 0);
begin
   -- Gera o circuito a seguir n vezes variando o valor de i para cada
       iteracao
   components: for i in n-1 downto 0 generate
      -- Verifica a coincidencia para cada bit dos dois vetores
      p(i) <= not(a(i) xor b(i));
   end generate components;

   process(a, b) is
      variable aux: std_logic;
   begin
      -- Inicializa a variavel 'aux'
      aux := '1';
      -- Laco de repeticao para realizar a operacao logica 'and' em todos os
          resultados p(i)
      bigand: for i in n-1 downto 0 loop
         aux := aux and p(i);
      end loop bigand;
      -- Salva o resultado final da operacao 'and' no sinal 'q'
      q <= aux;
   end process;
end CoincidenciaGenerico_arch;
```

Fonte: Elaborado pelo autor.

Figura 16 – Circuito coincidência de 8 bits gerado pelo sintetizador a partir do código contido na figura 15. O componente instanciador definiu o parâmetro genérico ‘n’ com valor 8.




Fonte: Elaborado pelo autor.

tamanho diferente, seria necessário realizar outra implementação.

Com um componente parametrizado, além do maior reuso, alterações no código são mais facilmente propagadas. A figura 14 contém o circuito gerado para entradas de 2 bits. Para realizar a expansão desse componente afim de que este seja capaz de realizar operações em entradas de tamanho 8 bits, o circuito que o instancia definiu a constante 'n' com valor 8. A figura 16 contém o circuito gerado após essa única alteração.

### 4.2.3 Criação de módulo

A medida que o desenvolvimento progride e a quantidade de código-fonte aumenta, pode ser interessante para a organização do projeto dividir trechos de código em grupos de acordo com alguma característica, seja o tipo de elemento declarado naquela seção ou a funcionalidade a qual pertence. Uma maneira de realizar esses agrupamentos é com a criação de módulos.

Um módulo permite a criação de código-fonte isolado de outros trechos de um mesmo programa. Isso evita conflitos com outras definições do programa. Para utilizar elementos de um pacote, utiliza-se o comando `use`. O comando `use` importa ao escopo atual os elementos nele nomeados, ou todos os elementos se a palavra-chave `all` tenha sido utilizada (IEEE, 1994).

O escopo em VHDL se aplica à um elemento declarado e à(s) sua(s) implementações. Por exemplo, suponha um arquivo "Entidades.vhd" e um "Arquitetura.vhd". Aquele contém a declaração de múltiplas entidades, enquanto este contém as implementações de todas as múltiplas entidades lá declaradas. Um pacote utilizado por todas as entidades declaradas em "Entidades.vhd" precisa ser adicionado com o comando `use` antes de cada entidade, já que seus escopos são separados. Porém, em nenhum momento este pacote precisa ser declarado no arquivo "Arquitetura.vhd", já que as implementações compartilham o escopo da declaração da entidade. Esse mesmo comportamento se aplica a pacotes e outros elementos da linguagem.

Uma outra vantagem da utilização de pacotes é a possibilidade de reutilizar código em outros projetos. Por padrão, a biblioteca que contém o código declarado no programa atual é acessada através da palavra-chave `work`. Pacotes externos podem ser adicionados ao processo de compilação como uma biblioteca, podendo ser adicionados com o comando `library`. A biblioteca `ieee` é um conjunto de pacotes padrão do VHDL com diversas definições de tipos, constantes e funções que não pertencem à especificação da linguagem mas são ubíquos em qualquer implementação (IEEE, 1994).

Alguns exemplos de pacotes definidos na biblioteca `ieee` são os pacotes `math_real`, que auxilia com operações utilizando números de ponto flutuante, `numeric_std` que auxilia com operações matemáticas e `std_logic_1164`, que implementa tipos robustos para representação de sinais lógicos, aceitando valores como 'Z' que representando alta impedância e 'X' representando valor desconhecido (CHU, 2006).

A figura 17 contém um exemplo de uma definição de pacote. Este pacote contém a

Figura 17 – Exemplo de código criando definições em um pacote. As definições são expressas dentro da estrutura `package` enquanto os valores e implementações são definidos dentro da estrutura `package body`. Caso não haja declaração de código, as definições podem ser realizadas diretamente na estrutura `package`. Ao fim do código vemos um exemplo de como adicionar o pacote ao escopo atual utilizando a biblioteca padrão `work`.

```
library ieee;
use ieee.math_real.all;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- Declaracao do pacote
package Pacote is
   -- Declaracao de uma funcao
   function BitsParaAderecar(constant n : in integer) return integer;
   -- Declaracao de duas constantes
   constant TamanhoDaPalavra: integer := 32;
   constant TamanhoDoEndereco: integer;
   -- Declaracao de um tipo composto
   type Registro is record
      Identificador: std_logic_vector(TamanhoDoEndereco-1 downto 0);
      Valor: std_logic_vector(TamanhoDaPalavra-1 downto 0);
   end record;
end Pacote;

-- Implementacao do pacote
package body Pacote is
   -- Implementacao da funcao declarada
   function BitsParaAderecar(constant n : in integer) return integer is
   begin
      return integer(ceil(log2(real(n))));
   end BitsParaAderecar;
   -- Definicao de uma constante declarada
   constant TamanhoDoEndereco: integer := BitsParaAderecar(TamanhoDaPalavra);
end package body;

-- Para incluir o pacote em outro escopo
use work.Pacote.all;
```

Fonte: Elaborado pelo autor.

declaração de constantes, um tipo e uma função. A implementação da função (e das eventuais entidades declaradas) se encontram na estrutura `package body`. Vale notar que apesar dos valores das constantes terem sido declarados no corpo do pacote, constantes que não dependem da implementação do pacote para receberem seu valor podem ter o valor associado na própria declaração do pacote, como é o caso da constante `TamanhoDaPalavra` no código da figura 17.

### 4.2.4 Integração com o circuito LEON3

O código-fonte do LEON3 foi desenvolvido em VHDL, fazendo uso portanto das técnicas abordadas neste capítulo. Algumas características específicas da implementação do LEON3 é a ampla utilização de tipos personalizados e de implementações parametrizadas. Os tipos personalizados são utilizados devido à grande quantidade de sinais passados como parâmetros para outros componentes. Ao utilizar tipos definidos pelo usuário é possível agrupar esses sinais de acordo com sua funcionalidade e passar o conjunto como um todo.

O uso abundante de *generics* se dá pelo fato do processador ser parametrizado. Todas as definições que podem ser alteradas ao executar o comando `make xconfig` na pasta do *design* são armazenados no arquivo `config.vhd` como constantes, que são passados para os componentes instanciados através do `generic map`. A alteração dessas configurações será abordada na seção 4.3.1.1.

Devido a essas alterações desses parâmetros realizadas ao configurar o processador é recomendado que este seja configurado com as definições desejadas antes de iniciar o processo de implementação. Assim o projeto não será influenciado posteriormente por alterações estruturais no processador, já que as configurações não só mudam o comportamento e tamanho de certas unidades, como pode alterar a implementação utilizada ou até mesmo adicionar componentes novos.

A maior parte dos arquivos `.vhd` utilizados para a compilação do LEON3 estão contidos dentro da pasta `lib`, na raiz do código fonte disponibilizado pela Gaisler. Esses arquivos são comuns a todas as diferentes implementações contidas na pasta `designs`, apesar de cada design ser configurado e compilado separadamente. Portanto, caso esteja o desenvolvedor esteja trabalhando com múltiplos *designs* simultaneamente sua implementação deve receber como parâmetro qual tecnologia está sendo utilizada ou ser implementada de forma que não é afetada pelas características de um chip específico.

Segue abaixo uma lista com os arquivos contendo a implementação alto-nível dos principais componentes do processador. A partir desses arquivos o projetista pode facilmente encontrar outros componentes que deseje observando a implementação, seja no código fonte ou através de algum visualizador de *netlists* disponibilizado pelo software Quartus II.

- `/designs/[chip]/leon3mp.vhd`: Componente raiz. Abriga os controladores de pinos (entradas e saídas), controlador de barramento e os núcleos do processador.
- `/lib/gaisler/leon3v3/leon3s.vhd`: Interface para um núcleo do processador.

- `/lib/gaisler/leon3v3/leon3x.vhd`: Componente com um núcleo do processador. Contém a memória cache, o arquivo de registradores e o processador principal.
- `/lib/gaisler/leon3v3/proc3.vhd`: Processador principal. Contém o pipeline/processador de inteiros, controlador da cache e unidades para operações aritméticas especiais, como multiplicação e divisão.
- `/lib/gaisler/leon3v3/iu3.vhd`: Pipeline de processamento de inteiros.
- `/lib/gaisler/leon3v3/regfile_3o_l3.vhd`: Banco de registradores. Contém uma unidade de memória por porta de leitura (2).
- `/lib/gaisler/leon3v3/cachemem.vhd`: Contém as memórias de cache de dados e instrução.

## 4.3 Compilação

Seja ele detalhado ao nível de portas lógicas ou abstraído ao nível de algoritmo, o código-fonte não pode ser inserido diretamente em um chip de FPGA já que este necessita de uma programação específica para seu hardware. Para que ocorra essa tradução é necessário compilar o código-fonte. A compilação é uma etapa crucial para a verificação de erros no código, além de permitir a simulação do circuito resultante e a utilização de diversas análises realizadas por ferramentas especializadas, como visualizações, análise temporal, análise de área de circuito, análise de potência, entre outros.

### 4.3.1 Compilação do LEON3

O LEON3 possui uma variedade de *designs* preparados para compilação focada em certas famílias de dispositivos. Esses projetos, contidos na pasta `designs` já contém as definições de certas configurações e alguns outros arquivos, como o `Makefile`, para possibilitar a compilação e síntese do projeto para aquele dispositivo.

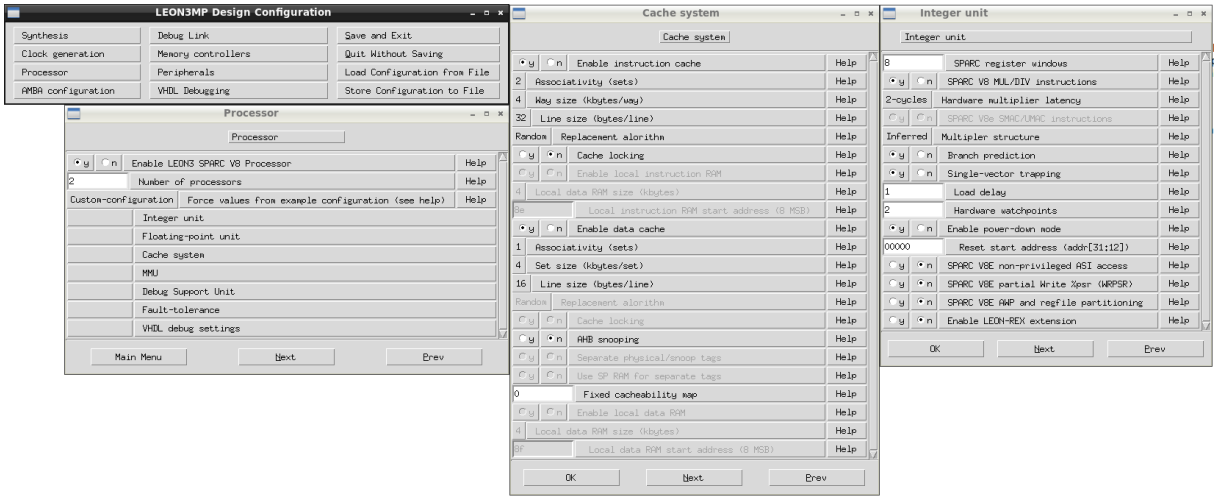
Dentro da pasta do design escolhido, que no caso da placa utilizada neste trabalho é a pasta `designs/leon3-altera-de2-ep2c35`, podemos realizar a configuração e a compilação do projeto utilizando o comando `make`. O comando `make` também será utilizado para gerar os arquivos de projeto necessários para abrir a o código-fonte de forma apropriada no editor utilizado, no caso deste trabalho o Quartus II.

#### 4.3.1.1 Configuração do processador com XConfig

O processador LEON3 foi projetado para ser altamente configurado. Sendo assim, muitos pontos críticos do design são definidos em arquivos de configurações como constantes e passados como *generics* para os componentes, definindo como esses devem se estruturar internamente para atender aos requisitos.

Para alterar essas configurações, o GRLIB disponibiliza uma interface gráfica, que pode ser acessada ao executar o comando *make xconfig* na linha de comando dentro do dire-

Figura 18 – Captura de tela de algumas telas de configuração da interface XConfig do GRLIB. Essa captura contém a janela inicial, `LEON3MP Design Configuration`, a janela `Processor` contendo as configurações do processador em si, ativadas clicando no botão `Processor` na janela `LEON3MP Design Configuration`, e as janelas `Cache system` e `Integer unit`, contendo respectivamente as definições da memória cache e do processador de inteiros, ativadas clicando nos botões de mesmo nome na janela `Processor`.




Fonte: COBHAM GAISLER (2018a)

tório de algum *design*. A figura 18 contém um exemplo com algumas das janelas do programa e suas opções de configurações.

Ao finalizar e salvar as configurações, basta compilar o projeto para que estas sejam aplicadas. Essas configurações se aplicam somente ao *design* contido na pasta local. Sendo assim, se o usuário deseja utilizar outro *design*, este também deve ser configurado de acordo com suas necessidades.

Caso deseje-se utilizar uma unidade de ponto flutuante no processador, é necessário realizar o *download* de arquivos adicionais como indicado na seção 4.1.4. Esses arquivos devem ser extraídos na pasta raiz do GRLIB. Recomenda-se que a primeira compilação após essa preparação seja realizada pelo `Makefile`, como demonstrado na seção 4.3.1.2.

#### 4.3.1.2 Compilação utilizando o Make

O `Makefile` disponível junto aos designs da GRLIB para a compilação do LEON3 contém diversas opções de compiladores, variando o formato dos arquivos gerados e os comandos a serem invocados de acordo com o software utilizado.

Para compilar utilizando o software Quartus II, deve-se executar o comando `make quartus` no mesmo diretório de *design* em que se executou o comando para permitir a configuração do XConfig. Da mesma forma, para compilar com o Modelsim o comando a ser executado é `make vsim`. Essas diretivas de compilação assumem que os comandos `quartus`

e `vsim` estejam devidamente mapeados aos inicializadores dos softwares Quartus e Modelsim, respectivamente. Para garantir a compatibilidade do LEON3 com as várias tecnologias utilizadas pela Altera, é recomendado executar o comando `make install-altera` antes da compilação.

### 4.3.2 Compilação com o Quartus

Ao executar o comando `make quartus` no diretório do *design* a ser utilizado, o `Makefile` irá gerar todos os arquivos necessários para a utilização do projeto com o ambiente Quartus. Isso inclui os arquivos de projeto `.qpf` e `.qsf`, com as definições do projeto como nome, versão, entidade principal, arquivos a incluir, atribuição de pinos, entre outros. Após a criação dos arquivos de projeto, este pode também ser aberto diretamente pelo menu do próprio Quartus ou através do comando `make quartus-launch`, que inicia uma instância nova do Quartus II com o projeto aberto.

Ao abrir o projeto no Quartus passa-se a ter acesso a todas as ferramentas de desenvolvimento disponíveis neste. Através das configurações do software é possível customizar a compilação, como adicionando restrições de recursos, alteração da distribuição dos pinos de entrada e saída de dados ou focar a otimização do circuito final em redução de área, melhor performance, etc.

Utilizando o menu de tasks, é possível ter um rápido acesso aos recursos disponíveis pela ferramenta ao longo do fluxo de trabalho envolvido no desenvolvimento de um circuito para FPGA. Essas ferramentas envolvem não somente configurações, mas também relatórios sobre as características do circuito gerado. Um exemplo de relatório pode ser visto na figura 19, que mostra a utilização de recursos por entidade de acordo com a compilação realizada.

É possível visualizar desde aspectos lógicos, como o visualizadores de componentes representados como máquinas de estados finitos e visualizadores do circuito equivalente representado utilizando portas lógicas e outros componentes eletrônicos, utilizado para a criação das figuras 11, 13, 14 e 16.

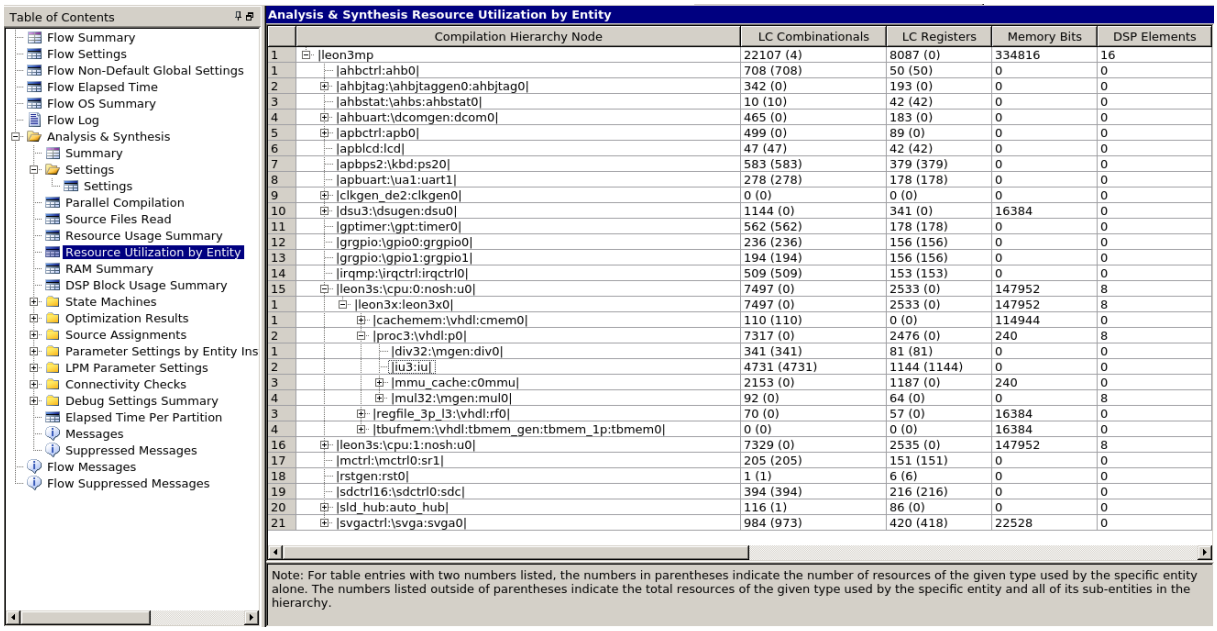
Além dessas, existem ferramentas de relatório após o processo de *fitting*, que transporta o circuito gerado pela compilação para a tecnologia alvo. Relatórios sobre a área de circuito, atrasos temporais e visualização do circuito dentro do chip podem ser gerados nessa etapa.

Após a compilação, o Quartus possui opções para ativar o Modelsim automaticamente se configurado, tanto para uma simulação lógica quanto temporal. Essas simulações serão abordadas na seção 4.4

### 4.3.3 Transferência para FPGA

Após realizar o processo de análise, síntese e ajuste ao dispositivo é possível gravar o circuito resultante no FPGA. Para transferir o hardware é utilizada uma ferramenta do Quartus chamada *Programmer*, acessada através do menu `Tools`.

Figura 19 – Captura de tela mostrando os relatórios pós-compilação do Quartus II. Em foco está o relatório exibindo a quantidade de recursos do chip utilizados e a distribuição desses recursos ao longo das entidades que compõe o projeto.




Fonte: Altera (2013)

A janela da ferramenta *Programmer* permite a seleção de qual a entidade a ser gravada, por padrão a entidade raiz do projeto, e qual o dispositivo alvo. Ao conectá-la ao computador esta será listada como `USB-Blaster` na lista de dispositivos de hardware disponíveis. A transferência será realizada com um clique no botão `Start`.

Caso a transferência seja configurada para utilizar o modo JTAG a chave `RUN/PROG` na placa deve estar em `RUN`. Essa gravação é mais simples e permite o teste instantâneo do hardware gravado e depuração, porém não é persistente. Ou seja, ao programar o FPGA utilizando JTAG a programação será perdida assim que o dispositivo for desligado.

Para gravação persistente, a chave `RUN/PROG` deve estar em `PROG`. Ao configurar o dispositivo no *Programmer*, o modo de transferência *Active Serial Programming* deve ser selecionado. Neste modo, a programação será gravada em uma memória *flash* contida no dispositivo e este será carregado ao inicializar.

## 4.4 Testes

Nesta seção será abordado o processo de verificação do projeto desenvolvido. Devido à complexidade e interdependência dos projetos desenvolvidos, pequenas coisas podem causar grandes impactos no resultado final. Para garantir que o resultado final estará então dentro do esperado é necessário dar a devida ênfase à etapa de testes.

As etapas aqui descritas seguem o fluxo padrão de desenvolvimento de um projeto de

hardware (ALTERA, 2007d). As formas de teste abrangem diferentes níveis de abstração do projeto, cada uma revelando diferentes aspectos do *design* produzido. A ideia por trás desse fluxo é a cada etapa de testes garantir certas características do circuito a fim de permitir que as próximas etapas possam focar em outras características.

### 4.4.1 Análise lógica

Após a implementação de uma unidade lógica, é necessário se assegurar de seu funcionamento correto. Afinal, um pequeno descuido pode impactar completamente o comportamento de um circuito, e quanto maior a complexidade do projeto maior a quantidade de pontos onde falhas podem ocorrer.

Analisar o projeto final em representações diferentes é uma forma de avaliar se o que buscou-se expressar foi de fato transmitido através da linguagem de programação. Porém nenhuma análise substitui o valor e a simplicidade de testes direcionados, enviando entradas à unidade e verificando as saídas produzidas, seja essa conferência manual ou automatizada.

#### 4.4.1.1 Análise lógica de onda com Quartus II

A maneira mais simples e rápida de testar a relação entre entradas e saídas de um componente de circuito é com a análise de forma de onda. Nessa forma de testes, uma onda é produzida para cada sinal de entrada variando os valores ao longo do tempo. Assim, diferentes combinações de valores para os diversos sinais de entrada diferentes são combinados com o tempo, permitindo testar as possibilidades de entrada e avaliar se as saídas produzidas se adéquam ao esperado do componente.

O Quartus possui um simulador de forma de onda que pode ser utilizado através de sua interface gráfica. Simples e intuitivo, esse simulador é indicado para testes menos complexos e permite uma rápida visualização do comportamento da unidade.

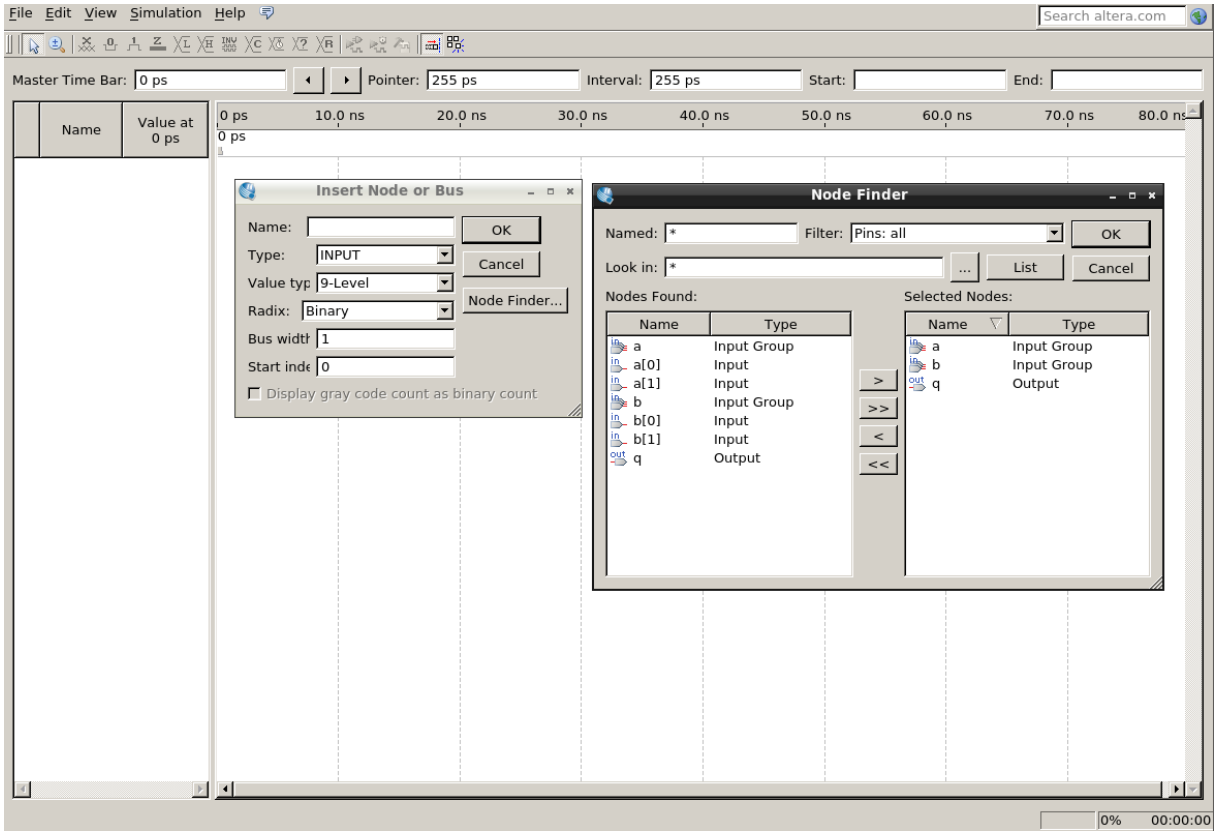
Para realizar um teste de resposta à forma de onde, primeiramente deve ser criado um arquivo do tipo `University Program VWF`. Ao editar esse arquivo, o Quartus abrirá uma nova janela contendo o *Simulation Waveform Editor*, que pode ser visto na figura 20 para uma simulação do componente *Coincidencia*, cujo código-fonte está disponível na figura 10.

No painel esquerdo são listados os sinais envolvidos na simulação, que na figura está vazio. Para adicionar sinais, deve-se clicar com o botão direito no painel, clicar em `Node Finder`, preencher os parâmetros da busca e clicar no botão `List`. Em seguida devem ser escolhidos os sinais e para finalizar clica-se no botão `OK`, confirmando a seleção.

Com os sinais visíveis, o usuário deve utilizar as ferramentas disponibilizadas na interface gráfica para adequar a forma da onda ao desejado. O procedimento padrão para geração de ondas de teste para sinais digitais com apenas os estados alto, representado por ‘1’, e baixo, representado por ‘0’, é atribuir a cada sinal uma onda quadrada, em que a frequência da onda de cada sinal varia com relação ao sinal anterior por um fator de 2.

Para executar a simulação lógica, deve-se clicar no botão `Run Functional Simulation`. Caso o usuário encontre problemas na simulação, deve-se alterar o simulador utilizado atra-

Figura 20 – Adicionando sinais ao simulador de forma de onda do Quartus.




Fonte: Elaborado pelo autor.

vés do menu `Simulations`, na opção `Options`. A figura 21 ilustra o resultado produzido após a execução da simulação.

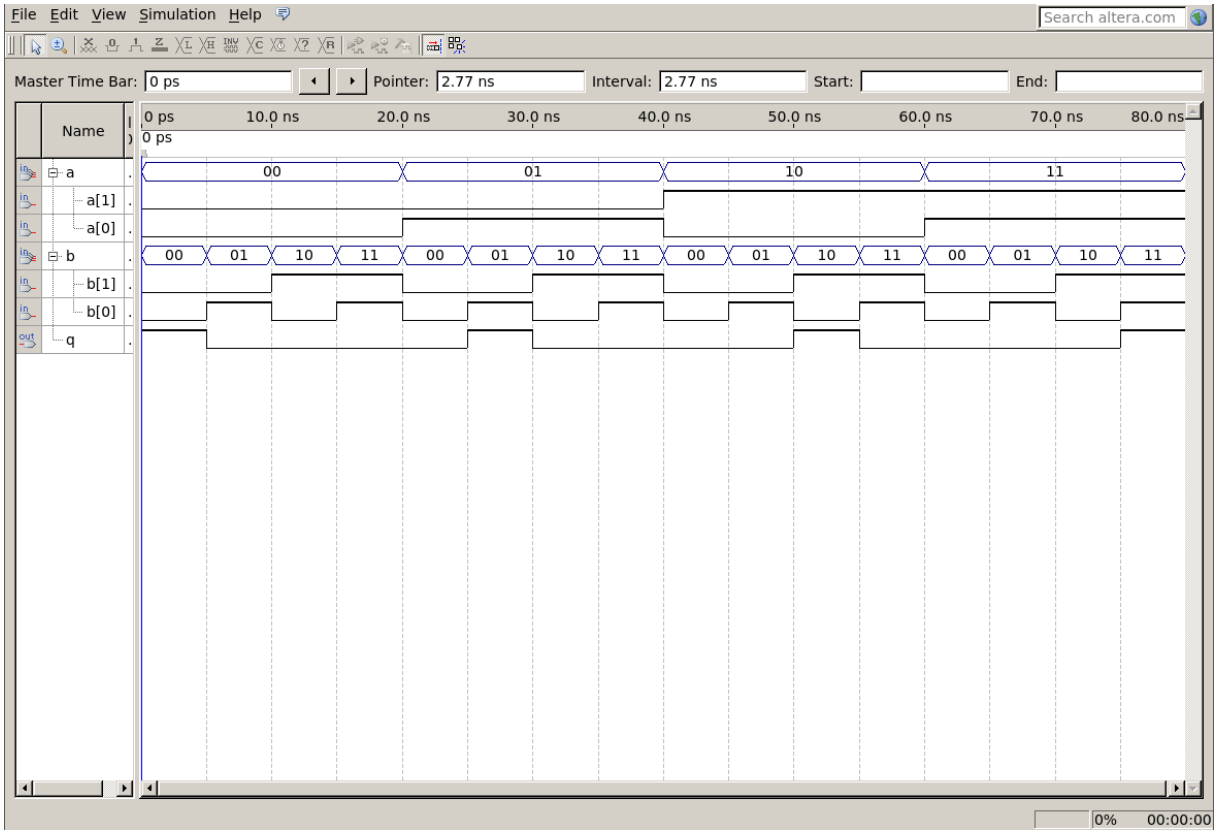
#### 4.4.1.2 Utilizando *testbenches*

A maneira mais escalável de testar um componente é a criação de um *testbench*. Um *testbench* é um código VHDL que possui como propósito o teste de um ou mais componentes desenvolvidos. Por ser escrita em linguagem de descrição de hardware, o usuário tem uma autonomia muito maior na realização de testes.

O Quartus possui ferramentas para agilizar o desenvolvimento de *testbenches*. O *Test Bench Template Writer* é uma ferramenta que inicia o processo de criação de um *testbench*, declarando uma entidade, mapeando componentes e criando sinais necessários para testar a entidade raiz do projeto atual. O *Test Bench Template Writer* pode ser executado no menu `Processing`, debaixo de `Start`, clicando na opção `Start Test Bench Template Writer`. Será criado um arquivo com extensão `.vht` contendo a estrutura básica do código necessário para a realização dos testes. A figura 22 contém um exemplo de código de *testbench* gerado pelo Quartus e alterado pelo usuário para inclusão de casos de teste.

Outra ferramenta disponível para o Quartus é o *NativeLink*, que permite a integração

Figura 21 – Visualizando o resultado da simulação lógica de forma de onda do Quartus.




Fonte: Elaborado pelo autor.

com outras ferramentas de desenvolvimento. A seguir será ilustrado o uso da integração do Quartus com o Modelsim-Altera para a execução de simulações com *testbenches*.

Com o uso de *testbenches* e do *NativeLink* é possível a fácil utilização do Modelsim para realização de simulações. O Modelsim é uma ferramenta à parte do Quartus capaz de simular com precisão circuitos em larga escala desenvolvidos em diversas linguagens de descrição de hardware, como o VHDL (MENTOR GRAPHICS, 2012). O Modelsim é a ferramenta recomendada pela Altera para a simulação precisa de circuitos de maior complexidade (ALTERA, 2007d).

Uma das dificuldades da utilização do Modelsim provém de sua complexidade. Por ser altamente robusto e baseado em comandos TCL, a barreira de entrada para seu uso é grande, já que um usuário iniciante necessita aprender os comandos para realizar suas tarefas (MENTOR GRAPHICS, 2015). Apesar de presente, a interface gráfica do Modelsim é bastante limitada em seus recursos, fazendo o uso de *scripts* uma necessidade.

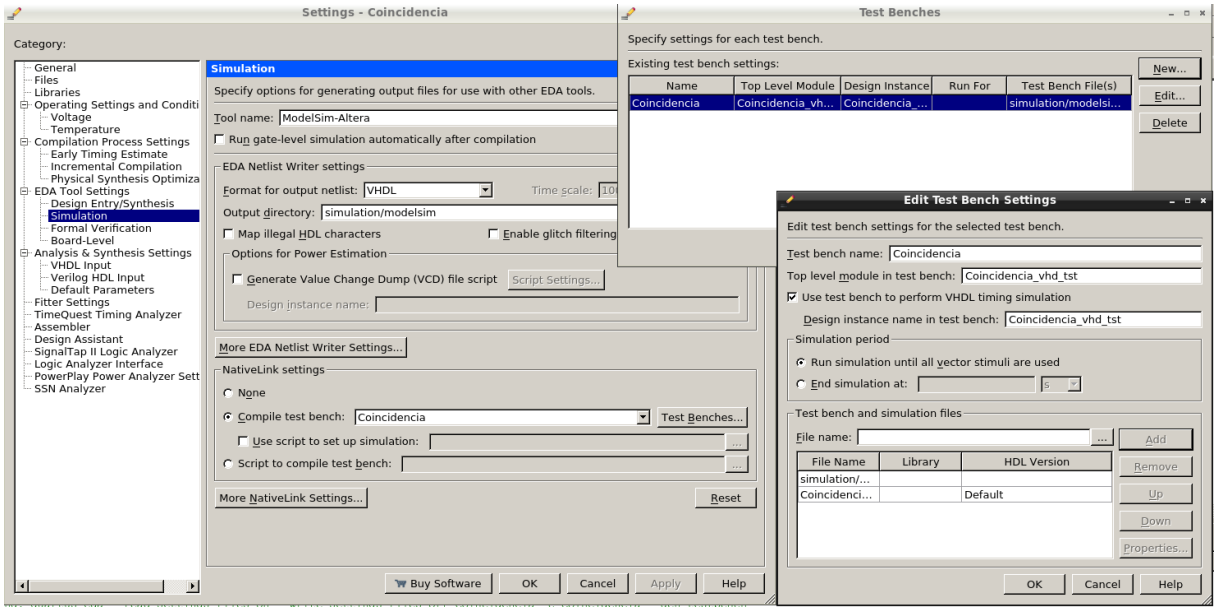
Com a utilização de *testbenches* e do *NativeLink* o Quartus automatiza esse processo, gerando um *script* básico com comandos informando ao Modelsim quais arquivos deve compilar, qual a entidade que este deve simular e que este deve ao final exibir as formas de onda produzidas pela simulação para o usuário.

Figura 22 – Exemplo de um *testbench* gerado pelo Quartus e editado pelo usuário. Alguns casos de teste foram omitidos por brevidade de código. Existem outras abordagens para a geração de sinais periódicos, porém para este exemplo optou-se por uma implementação mais explícita dos casos de teste para clareza do leitor.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY Coincidencia_vhd_tst IS
END Coincidencia_vhd_tst;
ARCHITECTURE Coincidencia_arch OF Coincidencia_vhd_tst IS
   SIGNAL a : STD_LOGIC_VECTOR(1 DOWNTO 0);
   SIGNAL b : STD_LOGIC_VECTOR(1 DOWNTO 0);
   SIGNAL q : STD_LOGIC;
   COMPONENT Coincidencia
      PORT (
         a : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
         b : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
         q : OUT STD_LOGIC
      );
   END COMPONENT;
BEGIN
   i1 : Coincidencia
   PORT MAP (
      a => a,
      b => b,
      q => q
   );
   init : PROCESS
   BEGIN
      -- Casos de teste inseridos manualmente
      a <= "00";
      b <= "00";
      WAIT FOR 10 ns;
      -- Se o sinal ‘q’ nao for ‘1’, a simulacao terminara com erro
      ASSERT q = ’1’;
      a <= "01";
      b <= "00";
      WAIT FOR 10 ns;
      ASSERT q = ‘0’;
      -- Omitindo varios casos de teste por brevidade
      a <= "10";
      b <= "11";
      WAIT FOR 10 ns;
      ASSERT q = ‘0’;
      a <= "11";
      b <= "11";
      WAIT FOR 10 ns;
      ASSERT q = ‘1’;
      -- Encerrando a execucao
      ASSERT false SEVERITY failure;
      WAIT;
   END PROCESS init;
END Coincidencia_arch;
```

Fonte: Elaborado pelo autor.

Figura 23 – Captura de tela exemplificando a configuração de um *testbench* no *NativeLink*.




Fonte: Elaborado pelo autor.

Nas definições de projeto, acessadas no menu `Assignments`, opção `Settings`, estão contidas todas as definições exclusivas do projeto, como arquivos inclusos no projeto, entidade raiz, preferências do compilador, entre outros. Na opção `Simulation`, debaixo da seção `EDA Tool Settings` é possível configurar os *testbenches a serem utilizados*. Em `NativeLink Settings` há a opção `Compile test bench`, com um menu de seleção de *testbenches* criados e um botão para a criação, edição e remoção destes. Uma janela será então aberta permitindo ao usuário especificar os arquivos necessários para a execução do teste e a entidade raiz da simulação. Um exemplo dessa configuração pode ser visto na figura 23.

Assim, ao clicar no botão `RTL Simulation` ou executar a *task* de mesmo nome, será iniciado o Modelsim-Altera, seram compilados os arquivos de código informados e será executada a simulação contida na entidade apontada como raiz do *testbench*. Caso algum comando `Assert` possua valor falso, a linha que causou o erro será indicada no console do Modelsim, indicando qual caso de teste falou a verificação. Ao fim da simulação, o *script* gerado pelo *NativeLink* abrirá o visualizador de ondas permitindo ao usuário visualizar a resposta de forma de onda dos sinais utilizados no *testbench*.

### 4.4.2 Análise temporal

Diferente da análise lógica, que requer que apenas a etapa de *“Analisys & Synthesis”* seja executada, a análise temporal requer que as etapas *“Fitter (Place & Route)”* e *“EDA Netlist Writer”* também sejam executadas. A etapa *“Analisys & Synthesis”* gera a lógica por trás do circuito, portanto é suficiente por si só para simulações lógicas, ou simulações em nível RTL (do inglês *Register-transfer level*), uma camada de abstração acima da necessária para

simulações temporais.

As etapas a seguir transpõe essa camada de abstração para que a simulação temporal, ou a nível de porta lógica (em inglês *gate level*). O processo de *"Fitter (Place & Route)"* adequa o circuito desejado às características do chip utilizado, tentando alocar os pinos da forma especificada pelo usuário caso este o faça. A seguir, o *"EDA Netlist Writer"* produz os *netlists*, ou seja, a relação interligada entre todos os elementos no circuito final com a associação de pinos, elementos de circuito, suas conexões, etc. Além disso, essa etapa também escreve um arquivo SDF (*Standard Delay Format*), contendo as informações de atraso de ligação e de portas para o *netlist* produzido, que de acordo com o fabricante se assemelham bastante às que serão encontradas no circuito real. Assim o simulador possui a capacidade de incorporar informações de atraso na simulação e produzir um resultado mais próximo do produzido ao gravar o circuito em um chip físico (ALTERA, 2007d).

É de grande importância realizar a análise temporal de um circuito, já que além de determinar a frequência máxima em que o circuito pode operar os atrasos podem alterar grandemente a resposta de lógica. Isso deve-se ao fato que os atrasos não são simétricos entre todas as entradas, fazendo com que a alteração de alguns valores seja processada e propagada mais rapidamente do que de outros, o que temporariamente é equivalente ao circuito executar corretamente com uma entrada diferente da atual. Além disso, deve se considerar que esses atrasos tendem a se compor com a adição de componentes sequenciais que dependem da resposta anterior e também possuem seu atraso interno.

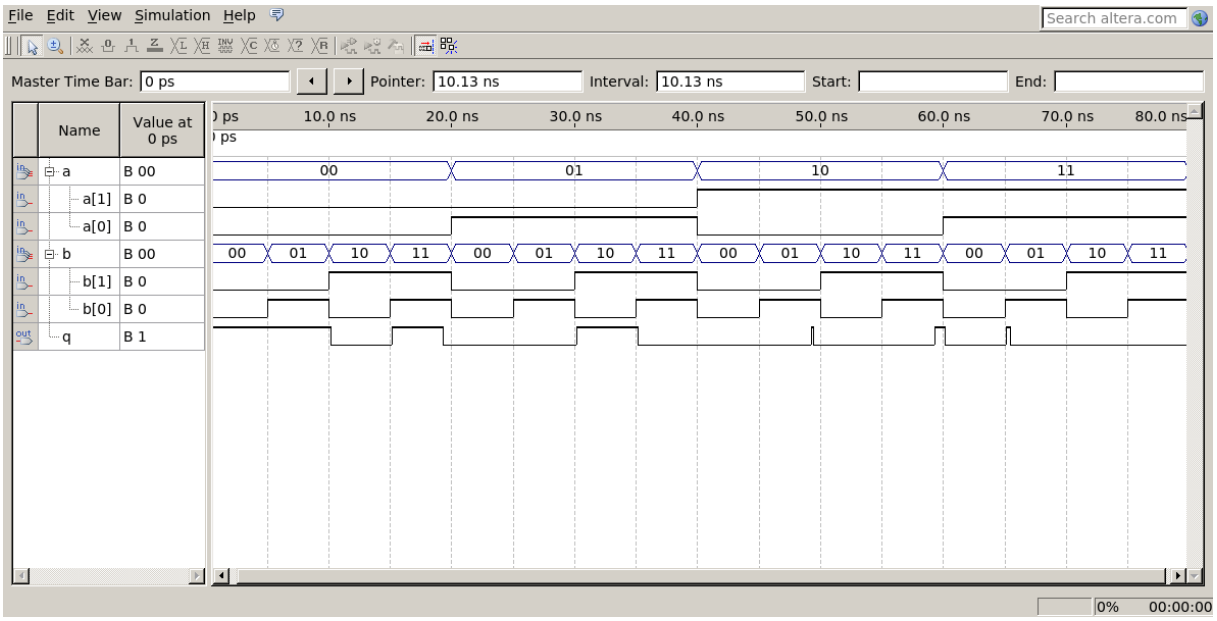
#### 4.4.2.1 Análise temporal de onda com Quartus II

Assim que o processo *"EDA Netlist Writer"* for completado o Quartus já é capaz de realizar simulações temporais. Este processo é executado automaticamente ao realizar uma compilação completa, porém pode ser forçado através do menu `Processing`, opção `Start`, clicando em *Start EDA Netlist Writer*. Então, da mesma forma que foi realizada a análise lógica de forma de onda na seção 4.4.1.1, pode-se realizar uma simulação temporal, clicando no botão `Run Timing Simulation` em vez do botão `Run Functional Simulation`.

Nas figuras 24 e 25 podemos avaliar a resposta temporal do circuito `Coincidencia`, cujo código-fonte está contido na figura 10. Ao comparar a resposta da figura 24, em que a entrada é alterada a cada 5 nanosegundos, com a análise lógica com atrasos ideais realizada na figura 21 é possível perceber a disparidade nas respostas produzidas. Não só é possível observar o tempo de atraso entre a variação das entradas e a produção do resultado, sendo o resultado esperado para os sinais de entrada definidos aos 25 nanosegundos só produzido após a marca dos 30 nanosegundos, como em certos momentos observamos resultados difíceis de explicar sem uma análise mais aprofundada, como acontece pouco antes à marca de 50 nanosegundos.

Em contraste, a figura 25, na qual a entrada do componente é alterada a cada 40 nanosegundos, a resposta já se assemelha muito mais à resposta ideal. Aumentando o tempo em que a entrada permanece estável, reduzimos proporcionalmente o período de instabilidade das saídas. Aqui é possível observar com mais clareza o que está ocorrendo com a variação

Figura 24 – Visualizando o resultado da simulação temporal de forma de onda do Quartus com entradas variando em alta frequência.




Fonte: Elaborado pelo autor.

das entradas: uma alteração no bit menos significativo leva aproximadamente 5 nanosegundos para ser computada, enquanto uma alteração no bit mais significativo leva aproximadamente 10 nanosegundos para ser computada. Essa latência possuem uma íntima relação com os conceitos abordados na seção 3.1.2.1.2 sobre latência, tempo de *setup* e tempo de *hold*.

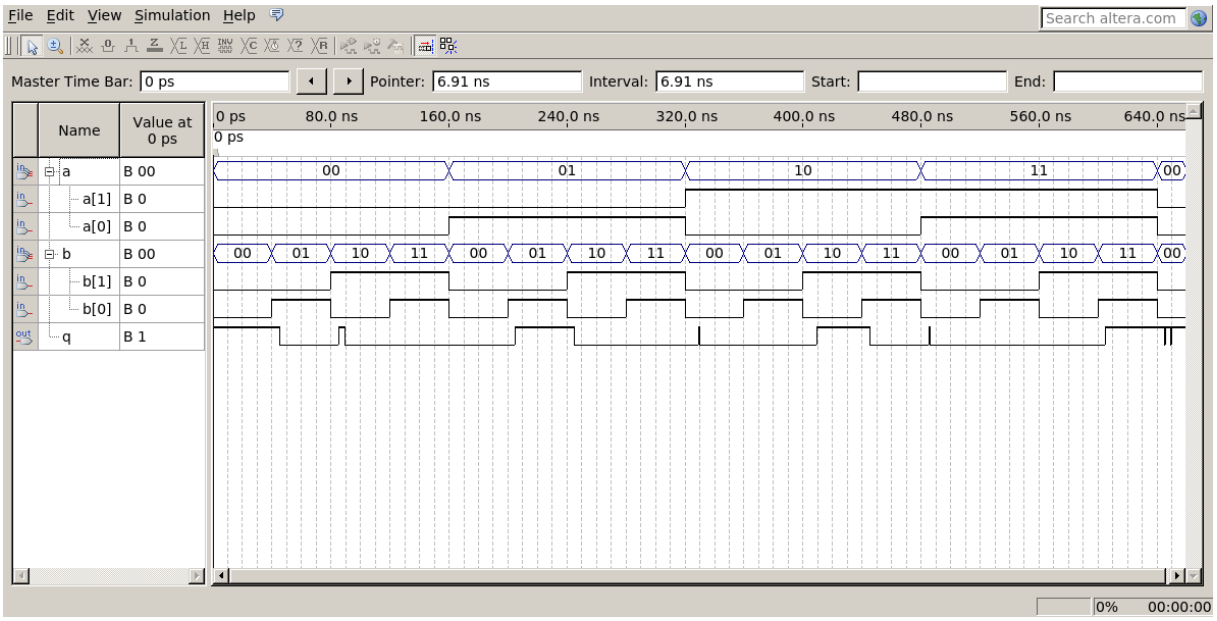
Isso pode ser confirmado visualizando o relatório de atrasos de propagação produzido pelo TimeQuest após a compilação completa, que indica que para a entrada ‘a’ o bit 1, mais significativo, possui um atraso de propagação para a saída ‘q’ de 9.454 picosegundos, enquanto o bit 0, menos significativo, possui atraso de 5.407 picosegundos no pior caso. De forma semelhante, a entrada ‘b’ possui atrasos no pior caso para os bits 1 e 0 de 9.280 picosegundos e 5.133 picosegundos, respectivamente.

O TimeQuest é uma ferramenta da Altera contida no Quartus que permite a análise mais detalhada das relações temporais em certos caminhos determinados pelo usuário. Seu uso permite não só a análise como a adição de restrições temporais para determinados caminhos síncronos que tentarão ser alcançadas quando o processo *“Fitter (Place & Route)”* for realizar a adaptação da lógica programada para os elementos físicos do dispositivo utilizado. A utilização do TimeQuest está além do escopo deste trabalho.

#### 4.4.2.2 Modelsim

De forma semelhante ao realizado na seção 4.4.1.2 para a análise lógica utilizando o Modelsim, o processo para a realização de simulações temporais utilizando este através do *NativeLink* envolve clicar no botão `Gate Level Simulation`. Porém, diferente da análise

Figura 25 – Visualizando o resultado da simulação temporal de forma de onda do Quartus com entradas variando em frequência relativamente menor.




Fonte: Elaborado pelo autor.

lógica, há grandes chances que ocorra um erro na simulação ou que seja realizada apenas a análise lógica.

O procedimento executado pelo Quartus ao clicar no botão `Gate Level Simulation` é a criação de um *script* de extensão `.do`, a abertura do Modelsim-Altera e a invocação do comando `do <nome do arquivo>.do` na linha de comando deste. Além disso ele também realiza o *backup* das versões antigas do *script*, adicionando a extensão `.bak` e um número ao final para indicar a versão. Sendo assim, ao clicar novamente no botão `Gate Level Simulation` as alterações realizadas no *script* serão desfeitas e passarão a estar apenas em um dos arquivos de backup.

Para garantir a execução correta da simulação temporal, primeiramente deve-se alterar os arquivos que compõe o *testbench*. Os arquivos de código-fonte com extensão `.vhd` devem ser substituídos pelos arquivos de *netlist* gerados com extensão `.vho` no diretório `simulation/modelsim`. Cabe ressaltar que o diretório e a extensão dos arquivos são definidos baseado nas opções do `EDA Netlist Writer`, como pode ser visto na figura 23. Caso a configuração `Format for output netlist` seja `Verilog`, o formato dos arquivos será `.vo`. O diretório é definido na configuração `Output directory`.

Após os ajustes na configuração do *testbench*, clique no botão `Gate Level Simulation` para gerar novamente o *script* com as configurações adequadas para a compilação e simulação com o Modelsim. O arquivo gerado estará no diretório configurado, por padrão `simulation/modelsim`, com nome `<nome_do_projeto>_run_msim_gate_<vhdl|verilog>.do`. Para evitar a perda das alterações no arquivo, é recomendado realizar uma cópia do *script*

Figura 26 – Exemplo de um *script* gerado pelo Quartus para a execução de simulações temporais com o Modelsim para o testbench definido na figura 22.

```
# Exibe no Modelsim os comandos executados
transcript on

# Cria ou recria uma biblioteca chamada 'gate_work'
if {[file exists gate_work]} {
   vdel -lib gate_work -all
}
vlib gate_work

# Associa a biblioteca padrao 'work' a biblioteca criada
vmap work gate_work

# Compila os arquivos necessarios
vcom -93 -work work {Coincidencia.vho}

vcom -93 -work work
    {/home/elias/test/Coincidencia/simulation/modelsim/Coincidencia.vht}
vcom -93 -work work
    {/home/elias/test/Coincidencia/simulation/modelsim/Coincidencia.vho}

# Define os parametros da simulacao
vsim -t 1ps +transport_int_delays +transport_path_delays -sdftyp
    /Coincidencia_vhd_tst=Coincidencia_vhd.sdo -L cycloneii -L gate_work -L work
    -voptargs="+acc" Coincidencia_vhd_tst

# Define as janelas a serem abertas
add wave * # Visualizacao de forma de onda para todos os sinais no elemento raiz
view structure # Visualizacao da hierarquia de estruturas
view signals # Visualizacao da lista de sinais

# Executa a simulacao
run -all
```

Fonte: Elaborado pelo autor.

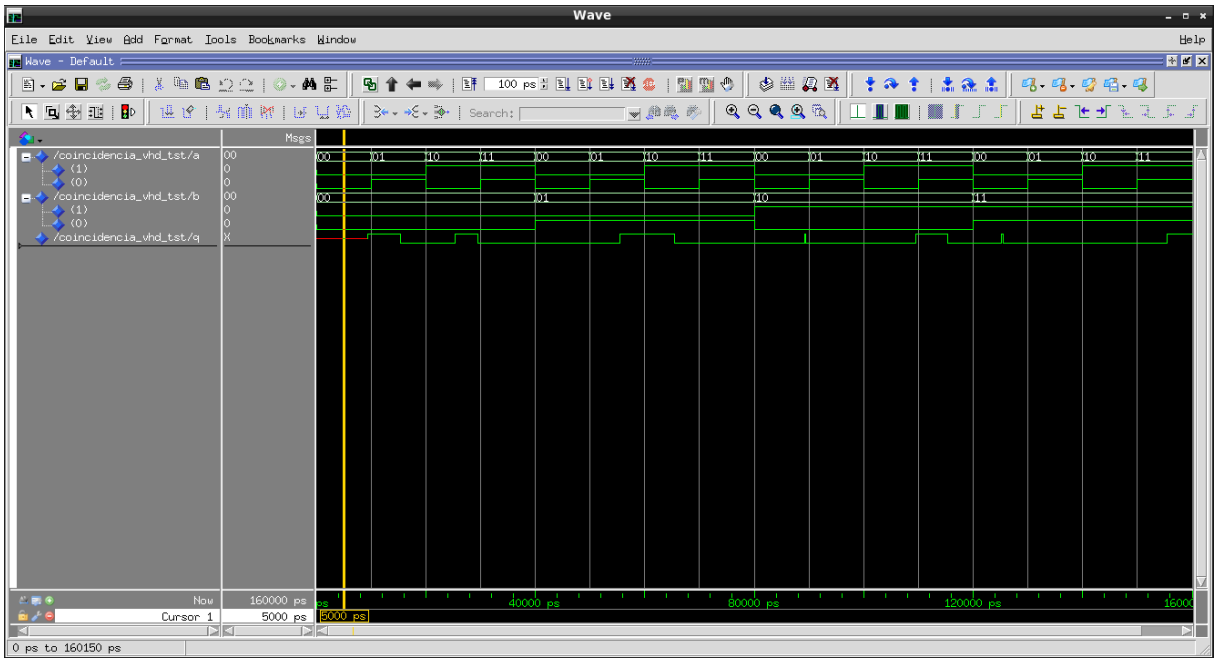
gerado e que as alterações sejam feitas neste.

Um problema comum na utilização de *testbenches* com a simulação do Modelsim através do *NativeLink* é o carregamento das informações de delay no arquivo `<entidade raiz>.sdo` para a entidade que abriga o *testbench*, fazendo com que haja um erro ao procurar os devidos *netlists* para realizar a associação dos atrasos. Deve então ser realizada a adaptação do arquivo para apontar qual entidade possui de fato os valores de atraso gerados.

Para o *testbench* da figura 22, temos na figura 26 o *script* gerado pelo Quartus, com comentários explicativos adicionados pelo autor.

No código-fonte do *testbench* utilizado, verifique o nome do marcador atribuído à instância entidade raiz. Em um *testbench* gerado pelo *Test Bench Template Writer* terá por padrão o nome `i1`, como pode ser observado na figura 22. Sendo assim, o comando `vsim` na figura

Figura 27 – Visualizando o resultado da simulação temporal de forma de onda do *testbench* da figura 22 no Modelsim.




Fonte: Elaborado pelo autor.

deve ser alterado para `vsim -t 1ps +transport_int_delays +transport_path_delays -sdftyp /Coincidencia_vhd_tst/i1=Coincidencia_vhd.sdo -L cycloneii -L gate_work -L work -voptargs="+acc"Coincidencia_vhd_tst`.

Ao realizar a chamada pelo *NativeLink* o arquivo será renomeado para um arquivo de *backup*, outro *script* igual ao original será gerado e executado, causando o mesmo erro. Para evitar isso, a simulação deve ser executada diretamente no Modelsim. Para isso, abra o Modelsim com o comando `vsim` no terminal. No console do Modelsim, utilize o comando `cd` (do inglês *change directory*, mudar de diretório) para navegar até o diretório que contém o *script* de extensão `.do` alterado anteriormente. Basta então executar o comando `do <nome do arquivo>`, passando como parâmetro o nome do arquivo com o *script* que executará a simulação. Para o *script* da figura 26 com a modificação anteriormente citada a forma de onda resultante pode ser vista na figura 27.

### 4.4.3 Simulação do LEON3 com Modelsim

É possível realizar a simulação do LEON3 no Modelsim a partir do código-fonte. Essa simulação permite a depuração do funcionamento do processador ao utilizar os recursos disponíveis no Modelsim para realizar a verificação do projeto.

Para realizar essa simulação, deve-se codificar o programa a ser executado em linguagem C. Esse código é então compilado utilizando o BCC e transformado em um binário de memória ROM utilizando o MKPROM. Essa memória ROM é carregada no *testbench* e a

simulação executada utilizando o Modelsim. Para facilitar esse processo, existem alguns procedimentos do `Makefile` do LEON3 que invocam os comandos necessários para a realização do fluxo descrito.

Para que a simulação seja possível, é recomendado habilitar a configuração `Accelerated UART tracing` no menu `VHDL Debugging` através do comando `make xconfig` na pasta do projeto.

Por padrão, o código-fonte do programa a ser executado na simulação está localizado no arquivo `systest.c` e executará um teste funcional do processador emitindo alguns relatórios. Esse programa pode ser alterado de acordo com o que o usuário deseja testar. Para realizar a compilação deste programa, deve ser executado o comando `make soft`. Após compilado, o programa estará armazenado no arquivo `ram.srec`, que é carregado pelo `testbench.vhd` e executado.

Caso deseje compilar manualmente o programa, após o fazer com os programas disponibilizados pelo BCC utilize o programa `sparc-elf-objcopy` para transformar o binário em um arquivo `srec`. Um exemplo desse processo, compilando o arquivo `test.c` e gerando uma imagem no arquivo `test.srec` pode ser obtida através dos comandos a seguir:

```
sparc-elf-gcc test.c -msoft-float -g -O2 -o test
mkprom2 -nocomp -msoft-float -freq 50 -ramsize 1024 test -o test.srec
```

Neste exemplo, foram utilizados alguns parâmetros úteis para a compilação, descritos abaixo:

- `sparc-elf-gcc -msoft-float`: Simula operações de ponto flutuante. Ideal para processadores sem hardware dedicado para essas operações;
- `sparc-elf-gcc -g`: Produz símbolos para facilitar a depuração do código;
- `sparc-elf-gcc -O2`: Realiza a otimização do código com foco em performance;
- `mkprom2 -nocomp`: Gera uma imagem sem compressão, aumentando o desempenho da execução;
- `mkprom2 -msoft-float`: Informa que o código foi compilado utilizando o este parâmetro;
- `mkprom2 -freq`: Informa a frequência base em que o processador estará trabalhando;
- `mkprom2 -ramsize`: Informa a quantidade de memória RAM disponível.

O nome do arquivo que armazena o conteúdo a ser carregado na RAM da simulação pode ser alterado no arquivo `testbench.vhd`, fazendo com que seja possível carregar um arquivo de memória arbitrário desejado pelo usuário.

O arquivo `prom.srec` contém instruções de inicialização do processador, que podem ser alteradas copiando o arquivo `/software/leon3/prom.S` para a pasta do *design* e realizando as alterações necessárias. O comando `make soft` realizará então a compilação deste

arquivo, mas caso deseje utilizar outros arquivos, como códigos C por exemplo, a compilação manual deste pode ser feita da mesma forma que descrita anteriormente. O nome do arquivo a ser carregado na ROM para realizar a inicialização do processador também pode ser alterado no arquivo `testbench.vhd`.

Por fim, para evocar o Modelsim e carregar o *testbench* modelo com o programa deve ser executado o comando `make vsim-lauch`. Assim que o Modelsim for aberto, o usuário poderar alterar as configurações que julgar necessário, como adicionar janelas, rastrear formas de onda, etc. Para executar a simulação, basta rodar no console do Modelsim a instrução `run -all`.

### 4.4.4 Teste do LEON3 com GRMON

Para gravar o circuito do LEON3 no dispositivo FPGA deve ser utilizado o método descrito na seção 4.3.3. A partir do momento que a transferência tiver ocorrido com sucesso, pode ser iniciado o uso do software GRMON para a execução de programas no LEON3.

Ao iniciar o GRMON deve ser especificada a forma de conexão com a placa através de um parâmetro *flag*. Caso a conexão seja via USB, o programa deve ser aberto através do comando `grmon -usb`. Caso seja via JTAG, a mesma conexão utilizada para transferir o LEON3 para a placa, o GRMON deve ser iniciado com o comando `grmon -altjtag`.

Dentro do console do GRMON, basta utilizar o comando `load <nome do executável>` para carregar o arquivo binário compilado com o BCC (`sparc-elf-gcc, sparc-elf-g++`, etc) para a memória RAM. Após o programa terminar de ser transferido, o comando `run` iniciará sua execução.

Para realizar o carregamento do programa para a memória flash é necessário utilizar o comando `flash`. Os comandos `flash unlock all` e `flash lock all` desbloqueiam e bloqueiam, respectivamente, a escrita na memória flash. O comando `flash erase all` limpa todo o conteúdo armazenado na memória flash, e o comando `flash load <nome do executável>` carrega o executável para a memória flash do sistema (COBHAM GAISLER, 2018b).

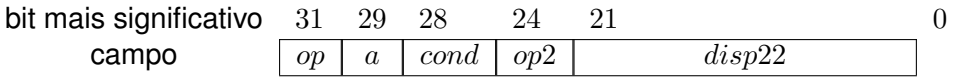
## 4.5 Estudo de caso

Nessa seção será mostrado um exemplo de modificação realizada no LEON3 e a análise de seus impactos no processador. Utilizando os conceitos apresentados até então, será possível realizar o processo da concepção da ideia à análise de resultados de dados obtidos em um processador físico.

### 4.5.1 Proposta

O processador LEON3 possui por padrão a opção de habilitar a predição de desvios (GAISLER et al., 2007). A predição implementada é estática, sempre assumindo o desvio como sendo tomado caso esteja ativo nas configurações. Caso esteja desativado, o processador funcionará como se possuísse predição de desvios assumindo o desvio como não tomado.

Figura 28 – Representação do formato de instruções 2 proposto na arquitetura SPARC.




Fonte: Elaborado pelo autor baseado em SPARC International, Inc. (1992).

O menu de configurações do LEON3 recomenda que seja ativada sempre a predição de desvios, já que afirmam melhorar o desempenho do processador em até 20% com um pequeno aumento na área de circuito e potência dissipada (COBHAM GAISLER, 2018a).

Assumir que desvios serão tomados permite um ganho de desempenho em caso de estruturas de repetição, já que estas tendem a realizar o desvio múltiplas vezes, uma para cada iteração, e não realizar o desvio apenas uma vez, ao sair do laço de repetição. Para estruturas condicionais não há uma vantagem tão clara entre assumir como tomado ou não o desvio, dependendo bastante do programa executado e podendo variar de desvio para desvio, podendo o desempenho inclusive influenciado por decisões do compilador.

A proposta desta seção é alterar a predição de desvios para, caso habilitada nas configurações, assumir como tomados desvios para um endereço anterior ao contador de programa e assumir como não tomado caso contrário. Assim estruturas de repetição irão se beneficiar de assumir o desvio como sendo tomado, enquanto estruturas condicionais executarão normalmente sem predição de desvios.

### 4.5.2 Implementação

Segundo (SPARC International, Inc., 1992), para instruções de desvio condicionais o desvio é calculado relativo ao contador de programa atual. As instruções de desvio condicional seguem o formato 2 de instruções, possuindo um campo para imediatos com sinal do bit 21 ao bit 0, representado na figura 28.

Para calcular o endereço do contador de programa após o desvio, o valor do imediato é somado ao valor do contador de programa atual. O valor resultante é o endereço da palavra a ser carregada. Por ser um processador 32 bits, as palavras tem tamanho de 4 bytes. Sendo assim, para realizar a requisição à memória, o valor calculado para o contador de programa é multiplicado por quatro realizando uma operação *shift* lógico para a esquerda de 2 bits.

Em estruturas de repetição o desvio condicional será para um valor menor que o contador de programas atual. Sendo assim, o valor do imediato a ser somado ao contador de programas será negativo nessas instruções condicionais. Como o LEON3 implementa números negativos através da utilização de complemento de 2 (GAISLER et al., 2007), para saber o sinal do imediato basta conferir o bit mais significativo deste. Caso o bit possua valor baixo o número é positivo, caso possua valor alto o número ali representado é negativo.

A implementação da predição de desvios no LEON3 é realizada no componente `IU3`, que comporta o *pipeline* de inteiros do processador. O código-fonte deste componente se

Figura 29 – Alteração realizada no código do LEON3 para a implementação proposta na seção 4.5.1. A alteração foi realizada na linha 3896 do arquivo `/lib/gaisler/leon3v3/iu3.vhd`.

```
-- Codigo original
-- BPRED <= '0' when bp = 0 else not r.d.rexen when bp = 1 else not (r.w.s.dbp
    or r.d.rexen);
BPRED <= '0' when bp = 0 else (not r.d.rexen) and
    std_ulogic(r.d.inst(conv_integer(r.d.set))(21)) when bp = 1 else not
    (r.w.s.dbp or r.d.rexen);
```

Fonte: Elaborado pelo autor.

encontra no arquivo `/lib/gaisler/leon3v3/iu3.vhd`.

O componente `IU3` possui entre seus parâmetros genéricos o campo `bp`. Este parâmetro possui o valor definido pelo usuário ao realizar a configuração do uso da predição de desvios condicionais, armazenando o valor `'1'` quando esta estiver ativa e `'0'` caso contrário.

Para adicionar a condição de que o imediato da instrução de desvio deve ser negativo para que a predição de desvios aconteça deve-se então realizar a operação `E` lógica entre o sinal que habilita a predição de desvios e o bit 21 da instrução, sinal que informa se o imediato é negativo através de valor alto.

Para realizar essa alteração deve ser modificada a linha 3896 do arquivo `/lib/gaisler/leon3v3/iu3.vhd` conforme a figura 29. O sinal `BPRED` armazena valor alto quando a predição de desvios estiver habilitada para o ciclo atual. Para isso, são utilizadas as informações sobre a configuração do processador e sobre as instruções no estágio de decodificação e escrita.

A adição realizada ao código acessa os sinais retornados pela memória cache de instruções decodificado, utilizando a informação dos *sets* para saber em qual via associativa houve o *hit* com o endereço do contador de programa, recuperando assim a instrução correta. Então é utilizado o bit 21 dessa instrução com uma operação `E` lógico com o valor original do sinal, garantindo que este só possuirá valor alto se o imediato da instrução for negativo.

### 4.5.3 Testes

Para a verificação sobre os prós e contras da implementação da técnica proposta de predição de desvios será realizada a comparação do processador modificado com sua versão não modificada com ambas as configurações, predição de desvios estática ativada e desativada.

Coletou-se dados a partir das informações dos relatórios pós compilação e *fitting* produzido pelo Quartus II 13.0, o relatório de potência produzido pelo *PowerPlay Power Analyzer* e a execução em hardware do *benchmark* Dhrystone versão 1.1 através do GRMON.

Deve ser ressaltado que os testes foram realizados na placa Terasic DE2-115 de posse

do autor deste trabalho. O processo para desenvolvimento, compilação e testes é idêntico ao da placa Altera DE2, sendo ambas as placas de desenvolvimento criadas pela Altera. A diferença entre as duas placas é o chip utilizado, que na placa DE2 é o EP2C35F672C6 da família Cyclone II, enquanto o da DE2-115 é o EP4CE115F29C7 da família Cyclone IV. Todos os testes foram realizados na mesma placa, então a comparação entre as implementações do processador permanece válida, mesmo que os valores em si não correspondam aos obtidos utilizando o chip EP2C35F672C6.

Para a realização de testes comparativos de desempenho, foi utilizado o *benchmark* Dhrystone. O código-fonte deste se encontra na figura 31 nos anexos do trabalho.

O código-fonte do *benchmark* Dhrystone foi compilado utilizando o `sparc-elf-gcc` versão 4.4.2 com os parâmetros `-g -O2` para a otimização do código e geração de informações para depuração. Sendo assim, os resultados podem variar ao utilizar programas de diferentes naturezas e programas ou configurações diferentes no processo de compilação. Para a execução do programa a frequência de clock do processador foi configurada para 100MHz.

Os relatórios do *PowerPlay Power Analyzer* foram gerados utilizando a abordagem *vectorless*, que possui precisão reduzida quanto aos valores exatos por não ter sido informado ao cálculo a porcentagem aproximada de transistores mudando de estado por ciclo de clock. Portanto, os valores absolutos podem ou não corresponder com os medidos em um uso real, mas a relação entre eles é indicadora do impacto relativo da alteração, já que todos os relatórios foram produzidos sob as mesmas condições.

### 4.5.4 Resultados e análise

A seguir serão descritos os resultados das métricas utilizadas para a comparação entre as três implementações do LEON3 no chip Altera EP4CE115F29C7.

#### 4.5.4.1 Uso de chip

Nas figuras 1, 2 e 3 podem ser vistos os resultados do relatório de uso de chip produzido pelo Quartus II após a compilação e *fitting* do circuito no chip, para o processador com predição de desvios desabilitada, habilitada e modificada conforme a seção 4.5.2, respectivamente.

Como pode ser observado, o uso de elementos lógicos combinacionais aumentou com a implementação proposta em relação ao processador sem predição de desvios, ficando em um valor intermediário entre este e o processador com predição de desvios habilitada. Porém o uso de registradores se mostra um pouco maior do que ambas as implementações.

A partir dessas informações é possível concluir que, a não ser em situações com restrições específicas, não há variação significativa na utilização de recursos e área de chip ao realizar a modificação proposta.

Tabela 1 – Relatório produzido pelo Quartus II 13.0 sobre o uso de chip do LEON3 com predição de desvios desabilitada.

| Recurso do chip | Total utilizado | Total disponível | Porcentual utilizado |
|---|---|---|---|
| Total de elementos lógicos | 23.688 | 114.480 | 21% |
| Total de funções combinacionais | 20.333 | 114.480 | 18% |
| Registradores lógicos dedicados | 9.472 | 114.480 | 8% |

Fonte: Elaborado pelo autor.

Tabela 2 – Relatório produzido pelo Quartus II 13.0 sobre o uso de chip do LEON3 com predição de desvios habilitada.

| Recurso do chip | Total utilizado | Total disponível | Porcentual utilizado |
|---|---|---|---|
| Total de elementos lógicos | 23.974 | 114.480 | 21% |
| Total de funções combinacionais | 20.815 | 114.480 | 18% |
| Registradores lógicos dedicados | 9.491 | 114.480 | 8% |

Fonte: Elaborado pelo autor.

Tabela 3 – Relatório produzido pelo Quartus II 13.0 sobre o uso de chip do LEON3 com predição de desvios modificada.

| Recurso do chip | Total utilizado | Total disponível | Porcentual utilizado |
|---|---|---|---|
| Total de elementos lógicos | 23.840 | 114.480 | 21% |
| Total de funções combinacionais | 20.698 | 114.480 | 18% |
| Registradores lógicos dedicados | 9.519 | 114.480 | 8% |

Fonte: Elaborado pelo autor.

#### 4.5.4.2 Dissipação de potência estimada

Nas figuras 4, 5 e 6 podem ser vistos os resultados do relatório de uso de chip produzido pelo *PowerPlay Power Analyzer* com as estimativas de dissipação térmica de potência pelo circuito, para o processador com predição de desvios desabilitada, habilitada e modificada conforme a seção 4.5.2, respectivamente.

Através desses relatórios é possível observar um perfil de dissipação de potência bem parecido entre as três implementações do LEON3, onde a variação de energia entre os mesmos é baixíssima, ainda mais considerando que essas estimativas possuem um erro relativamente alto devido ao método *vectorless* de cálculo. A diferença mais significativa é na dissi-

pação térmica de potência dinâmica do chip, indicando que a implementação com predição de desvios modificada possui uma dissipação de potência pouco menor que as outras duas implementações.

A partir dessas informações é possível concluir que, a não ser em situações com restrições específicas, não há variação significativa na dissipação de potência ao realizar a modificação proposta.

Tabela 4 – Relatório produzido pelo *PowerPlay Power Analyzer* sobre a dissipação de potência estimada do LEON3 com predição de desvios desabilitada.

| Tipo de dissipação térmica de potência | Dissipação estimada (em mW) |
|---|---|
| Dissipação total | 698,37 |
| Dissipação dinâmica do chip | 473,23 |
| Dissipação estática do chip | 105,86 |
| Dissipação de entradas e saídas | 119,28 |

Fonte: Elaborado pelo autor.

Tabela 5 – Relatório produzido pelo *PowerPlay Power Analyzer* sobre a dissipação de potência estimada do LEON3 com predição de desvios habilitada.

| Tipo de dissipação térmica de potência | Dissipação estimada (em mW) |
|---|---|
| Dissipação total | 702,78 |
| Dissipação dinâmica do chip | 476,33 |
| Dissipação estática do chip | 105,88 |
| Dissipação de entradas e saídas | 120,57 |

Fonte: Elaborado pelo autor.

#### 4.5.4.3 Desempenho do *benchmark* Dhrystone

O *benchmark* Dhrystone 1.1 foi utilizado para produzir um panorama do desempenho entre as diferentes implementações do LEON3. A execução deste foi realizada com todos os processadores configurados para uma frequência de clock de 100MHz. O código-fonte foi configurado para realizar 500000 para maior precisão dos resultados, reduzindo o impacto do *overhead* de execução no resultado final.

A figura 7 contém um quadro comparativo com os resultados da execução do *benchmark* nas três implementações do processador sob as mesmas condições. A partir desses resultados, pode ser observado que o processador com a predição de desvios modificada teve um desempenho intermediário entre a versão com predição de desvios desabilitada e a predi-

Tabela 6 – Relatório produzido pelo *PowerPlay Power Analyzer* sobre a dissipação de potência estimada do LEON3 com predição de desvios modificada.

| Tipo de dissipação térmica de potência | Dissipação estimada (em mW) |
|---|---|
| Dissipação total | 690,04 |
| Dissipação dinâmica do chip | 463,97 |
| Dissipação estática do chip | 105,83 |
| Dissipação de entradas e saídas | 120,24 |

Fonte: Elaborado pelo autor.

ção de desvios habilitada. Isso confirma a hipótese que desvios para instruções anteriores à atual se beneficiam caso o hardware assuma os desvios como sendo tomados. Porém o desempenho desta versão modificada ficou aquém da versão que assume que todos os desvios são tomados.

A partir dessas informações é possível concluir que, para programas com perfil de fluxo de controle parecidos com o Dhrystone 1.1 compilado conforme descrito na seção 4.5.3, a predição de desvios padrão do LEON3 habilitada é a melhor configuração para o processador, salvo restrições específicas que possam ser impostas ao projeto. Isso se deve ao fato dessa implementação entregar melhor performance com pequenos aumentos na área de chip e potência dissipada.

Tabela 7 – Quadro comparativo dos resultados do *benchmark* Dhrystone 1.1 após 500000 iterações a 100MHz.

| Implementação do LEON3 | Dhrystones executados por segundo |
|---|---|
| Predição de desvios desabilitada | 114068 |
| Predição de desvios habilitada | 121065 |
| Predição de desvios modificada | 118110 |

Fonte: Elaborado pelo autor.

# 5 Considerações finais

*"Só conseguimos ver uma pequena distância à frente,*
*mas podemos ver que há muito a ser feito."*[1] .
*Alan Turing (Tradução nossa)*

Ao longo deste trabalho buscou-se construir uma referência sintetizada do desenvolvimento de hardware. Essa referência busca ser relevante tanto para estudantes que desejam realizar pesquisa quanto para aqueles que desejam utilizar um processador para alguma aplicação com outros fins. Apesar de não ser uma referência exaustiva, é esperado que aqueles interessados em adentrar essa área encontrem neste trabalho uma introdução com o conteúdo necessário para gerar no leitor um embasamento suficiente para que este possa expandir e aprofundar seu conhecimento na área de forma independente.

Devido à sua disponibilidade no Campus VII do CEFET-MG, foi decidido utilizar como base para o desenvolvimento o dispositivo EP2C35F672C6, da família Cyclone II da Altera. Portanto todas as etapas práticas relativas ao uso de ferramentas é restrito ao uso das ferramentas da Altera, em especial as compatíveis com tal dispositivo. Assim, muitos detalhes sobre como devem ser executadas as etapas no processo de prototipação de hardware se tornam específicas para aqueles que utilizarem tais tecnologias, porém as ideias do desenvolvimento e os conceitos aplicados permanecem os mesmos independentemente da tecnologia utilizada.

Da mesma forma, a decisão pela utilização do LEON3 como processador base para estudos sobre arquitetura de computadores restringe a amplitude do escopo deste trabalho. O LEON3 foi escolhido pela sua disponibilidade em código aberto e pela gama de softwares de suporte disponibilizados pela Cobham Gaisler para auxiliar o desenvolvimento o utilizando. Porém, ao fazer seu uso o usuário fica restrito pelos aspectos legais de sua licença e de seus softwares, que são cruciais para que o desenvolvimento seja bem sucedido. Também ocorre uma restrição técnica, já que o processador implementa a arquitetura SPARC em linguagem VHDL, fazendo com que qualquer pessoa que o utilize possua conhecimentos específicos sobre estes assuntos.

Essa restrição do escopo permite de certa forma que o objetivo do trabalho seja atingido, já que um problema detectado para todos aqueles que desejam utilizar a prototipação de hardware, tanto para fins acadêmicos quanto comerciais, é a separação entre a literatura com conhecimentos gerais sobre o assunto e a literatura específica sobre os aspectos técnicos necessários para conduzir um projeto real, sem as abstrações empregadas durante o processo didático em sala de aula.

Encontrar um ponto de entrada é uma das maiores barreiras encontradas pelos interessados na área devido a tal separação, com boa parte da literatura específica assume que

[1] *"We can only see a short distance ahead, but we can see plenty there that needs to be done."*

o leitor possua uma grande quantidade de conhecimentos prévios, enquanto a literatura geral apresenta apenas vislumbres do conteúdo prático, não tocando em pontos necessários para o desenvolvimento de um projeto real. Além disso, grande parte do conhecimento específico está distribuído por várias fontes, muitas informais, não compilados em um guia unificado. Isso não é uma crítica à toda a literatura no assunto, apenas uma constatação de que existe um público prejudicado por não ser o foco das publicações. Essa dificuldade é maior ainda para vários indivíduos quando consideramos que o material específico se encontra em língua estrangeira.

Através do contato direto e da capacidade de manipular um processador real, um aluno poderá aprofundar seus conhecimentos sobre arquitetura de computadores além daquilo visto em sala de aula. Diferente dos modelos simplificados, um processador real expõe o estudante a particularidades de um circuito real, à complexidade e as formas de mantê-la sob controle e a soluções de engenharia criadas com objetivo prático pelos desenvolvedores do LEON3, sem preocupação com o aspecto didático deste.

Além disso, também foi abordado como o pesquisador pode utilizar os recursos dos softwares de desenvolvimento modernos para melhor desenvolver, analisar e otimizar seu circuito. Essas ferramentas permitem a realização virtual de testes e medições antes só possíveis em um protótipo físico, permitindo a investigação de diversas propriedades a custo e tempo reduzidos. Isso permite que esforços de pesquisa sejam utilizados para diversas áreas que podem se beneficiar de computação, tanto a arquitetura de computação em si com a exploração de técnicas que possam melhorar o desempenho, quanto de outras áreas em que um processador de uso geral ou especializado para aquela aplicação podem gerar melhor resultados.

Essas possibilidades se mostram promissoras, já que está ocorrendo a expansão das aplicações de sistemas embarcados com o fim de utilizar a computação como forma de otimizar diversos processos. Com conexões em redes móveis, as aplicações se tornam cada vez maiores e sistemas especialistas podem viabilizar projetos antes tecnicamente inviáveis.

A chamada computação ubíqua é uma realidade presente no dia-a-dia e que expande os possíveis eixos de pesquisa a serem explorados. Com possibilidades mais flexíveis de computação, os esforços interdisciplinares para aplicação de computação em resolução de problemas de outras áreas faz com que as possibilidades sejam limitadas quase que apenas pela capacidade dos pesquisadores de identificar problemas solucionáveis com técnicas computacionais e com a devida aplicação dos conhecimentos das áreas envolvidas para produzir uma solução de engenharia capaz de resolver o problema.

Uma das maiores dificuldades de engenharia para qualquer problema de computação sempre é na utilização de equipamentos de hardware que se adequem às restrições impostas pelo problema. Com a possibilidade de produção de hardware personalizado criada por tecnologias como o FPGA, a possibilidade de utilizar e customizar processadores amplamente utilizados e testados como o LEON3 e a utilização de técnicas de computação e de softwares auxiliares, possuir equipamentos de hardware que se adequam às restrições se torna factível e viável a um custo reduzido desde que haja pessoal qualificado para poder realizar tais adaptações.

Com este trabalho, espera-se que ser esse alguém qualificado esteja ao alcance de qualquer um com conhecimentos básicos de sistemas digitais e arquitetura de computadores. Após compreender os conceitos e técnicas aqui expostos, cabe ao pesquisador expandir seus conhecimentos através da literatura teórica e prática para seus desejados fins, mas com um ponto de partida claro e uma introdução gradual que dê ao estudante uma curva de aprendizado adequada.

# Referências


AGARWAL, A.; LANG, J. H. *Foundations of Analog and Digital Electronic Circuits*. San Francisco: Elsevier, 2005. Citado nas páginas 14 e 29.

ALTERA. *DE2 Development and Education Board*. [S.l.], 2006. Disponível em: <https://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=China&No=30&FID=ab73908ea64e51be175534c8101942b7>. Citado na página 18.

ALTERA. *Cyclone II Device Handbook*. [S.l.]: Altera Corporation, 2007. Citado nas páginas 36 e 37.

ALTERA. *Quartus II Handbook Version 9.0 - Volume 1: Design and Synthesis*. San Jose: Altera Corporation, 2007. v. 1. Disponível em: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/qts/archives/quartusii_handbook_9.0.pdf>. Citado na página 22.

ALTERA. *Quartus II Handbook Version 9.0 - Volume 2: Design Implementation and Optimization*. San Jose: Altera Corporation, 2007. v. 2. Disponível em: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/qts/archives/quartusii_handbook_9.0.pdf>. Citado na página 22.

ALTERA. *Quartus II Handbook Version 9.0 - Volume 3: Verification*. San Jose: Altera Corporation, 2007. v. 3. Disponível em: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/qts/archives/quartusii_handbook_9.0.pdf>. Citado nas páginas 22, 41, 61, 63 e 66.

ALTERA. *Quartus II Handbook Version 9.0 - Volume 4: SOPC Builder*. San Jose: Altera Corporation, 2007. v. 4. Disponível em: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/qts/archives/quartusii_handbook_9.0.pdf>. Citado na página 22.

ALTERA. *Quartus II Handbook Version 9.0 - Volume 5: Embedded Peripherals*. San Jose: Altera Corporation, 2007. v. 5. Disponível em: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/qts/archives/quartusii_handbook_9.0.pdf>. Citado na página 22.

ALTERA. *Quartus II, Version 13.0.1*. [S.l.]: Altera Corporation, 2013. Citado nas páginas 40, 41, 44, 60 e 86.

ASHENDEN, P. *The Designer's Guide to VHDL*. Adelaide: Elsevier Science, 2001. (Systems on Silicon). ISBN 9780080477152. Citado na página 35.

BENGTSSON, D.; FÅNG, R. *Developing a LEON3 template design for the Altera Cyclone-II DE2 board*. 2011. Dissertação (Mestrado), 2011. Citado na página 22.

CEFET-MG. *Projeto Para Criação Do Curso De Engenharia De Computação*. Timóteo: Centro Federal de Educação Tecnológica de Minas Gerais Campus VII, 2008. Citado na página 15.

CHU, P. P. *RTL hardware design using VHDL: coding for efficiency, portability, and scalability*. Hoboken: John Wiley & Sons, 2006. Citado nas páginas 21, 29, 35, 36, 49 e 54.

COBHAM GAISLER. *GRLIB IP Core User's Manual*. [S.l.]: Cobham Gaisler, 2016. Citado nas páginas 33 e 39.

COBHAM GAISLER. *GRLIB IP Library User's Manual*. [S.l.]: Cobham Gaisler, 2016. Citado nas páginas 32, 33 e 34.

COBHAM GAISLER. *BCC User's Manual*. [S.l.]: Cobham Gaisler, 2017. Citado na página 43.

COBHAM GAISLER. *GRLIB release 2018.3-b4226*. Cobham Gaisler, 2018. Disponível em: <https://www.gaisler.com/index.php/downloads/leongrlib>. Citado nas páginas 58 e 73.

COBHAM GAISLER. *GRMON 2 User's Manual*. [S.l.]: Cobham Gaisler, 2018. Citado nas páginas 42 e 72.

COBHAM GAISLER. *MKPROM2 User's Manual*. [S.l.]: Cobham Gaisler, 2018. Citado na página 43.

D'AMORE, R. *VHDL: descriçao e síntese de circuitos digitais*. Rio de Janeiro: LTC, 2005. ISBN 9788521614524. Citado na página 35.

DANĚK, M. et al. *UTLEON3: Exploring Fine-Grain Multi-Threading in FPGAs*. New York: Springer, 2012. Citado na página 22.

GAISLER, C. Leon3 processor. *Nanoscale Integration and Modeling (NIMO) Group*, 2010. Citado na página 22.

GAISLER, J. et al. Grlib ip core user's manual. *Gaisler, Aeroflex*, 2007. Citado nas páginas 22, 72 e 73.

HENNESSY, J. L.; PATTERSON, D. A. *Computer architecture: a quantitative approach*. [S.l.]: Elsevier, 2011. Citado nas páginas 25, 26 e 32.

HOROWITZ, P.; HILL, W. *The Art of Electronics*. Cambridge: Cambridge University Press, 2015. Citado na página 30.

IEEE. Ieee standard vhdl language reference manual. *ANSI/IEEE Std 1076-1993*, p. 1–288, June 1994. Citado nas páginas 35 e 54.

KADY, S. E.; KHATER, M.; ALHAFNAWI, M. Mips, arm and sparc-an architecture comparison. In: *Proceedings of the World Congress on Engineering*. [S.l.: s.n.], 2014. v. 1. Citado na página 30.

KLEITZ, W. *Digital Electronics: A Practical Approach with VHDL*. [S.l.]: Pearson, 2011. Citado na página 21.

MACK, C. A. Fifty years of moore's law. *IEEE Transactions on semiconductor manufacturing*, IEEE, v. 24, n. 2, p. 202–207, 2011. Citado na página 14.

MANO, M. M. R.; CILETTI, M. D. *Digital Design: With an Introduction to the Verilog HDL*. [S.l.]: Pearson, 2012. Citado na página 21.

MAXFIELD, C. *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows (Edn Series for Design Engineers)*. Oxford: Newnes, 2004. Citado nas páginas 15, 35 e 36.

MENTOR GRAPHICS. *ModelSim User's Manual*. Wilsonville, 2012. Citado nas páginas 22, 41 e 63.

MENTOR GRAPHICS. *ModelSim Command Reference Manual*. Wilsonville, 2015. Citado nas páginas 22 e 63.

OKLOBDZIJA, V. G. *Digital Design and Fabrication (Computer Engineering Series)*. [S.l.]: CRC Press, 2007. Citado na página 21.

PATTERSON, D. A.; HENNESSY, J. L. *Computer organization and design: the hardware/-software interface*. [S.l.]: Newnes, 2013. Citado nas páginas 24, 25, 26 e 28.

PIAGET, J. et al. *Knowledge and Development: Volume 1 Advances in Research and Theory*. New York: Springer, 1977. Citado na página 15.

SPARC International, Inc. *The SPARC Architecture Manual*. Menlo Park, EUA: SPARC International, Inc., 1992. Citado nas páginas 31, 32, 33 e 73.

TANENBAUM, A. S.; ZUCCHI, W. L. *Organização estruturada de computadores*. [S.l.]: Pearson Prentice Hall, 2009. Citado nas páginas 23, 25 e 36.

WEICKER, R. P. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, New York, v. 27, n. 10, 1984. Citado na página 102.

# Anexos

Figura 30 – Lista de *megafunctions* disponíveis no Quartus II 13.0 SP1 Web Edition por padrão, como exibidas no *MegaWizard Plug-in Manager*. Com a fonte de cor preta estão os componentes disponíveis para uso, enquanto em cinza estão os componentes indisponíveis, seja por incompatibilidade de hardware ou ausência de direitos para uso. Para melhor visualização, a listagem foi dividida em três colunas.


Fonte: Altera (2013)

Figura 31 – Código-fonte utilizado do *benchmark* Dhrystone.

```
/***** hpda:net.sources / homxb!gemini / 1:58 am Apr 1, 1986*/
/* EVERBODY:  Please read "APOLOGY" below. -rick 01/06/85
*       See introduction in net.arch, or net.micro
*
*  "DHRYSTONE" Benchmark Program
*
*  Version: C/1.1, 12/01/84
*
*  Date:    PROGRAM updated 01/06/86, RESULTS updated 03/31/86
*
*  Author:    Reinhold P. Weicker, CACM Vol 27, No 10, 10/84 pg. 1013
*       Translated from ADA by Rick Richardson
*       Every method to preserve ADA-likeness has been used,
*       at the expense of C-ness.
*
*  Compile: cc -O dry.c -o drynr    : No registers
*       cc -O -DREG=register dry.c -o dryr : Registers
*
*  Defines: Defines are provided for old C compiler's
*       which don't have enums, and can't assign structures.
*       The time(2) function is library dependant; Most
*       return the time in seconds, but beware of some, like
*       Aztec C, which return other units.
*       The LOOPS define is initially set for 50000 loops.
*       If you have a machine with large integers and is
*       very fast, please change this number to 500000 to
*       get better accuracy. Please select the way to
*       measure the execution time using the TIME define.
*       For single user machines, time(2) is adequate. For
*       multi-user machines where you cannot get single-user
*       access, use the times(2) function. If you have
*       neither, use a stopwatch in the dead of night.
*       Use a "printf" at the point marked "start timer"
*       to begin your timings. DO NOT use the UNIX "time(1)"
*       command, as this will measure the total time to
*       run this program, which will (erroneously) include
*       the time to malloc(3) storage and to compute the
*       time it takes to do nothing.
*
*  Run:    drynr; dryr
*
*  Results: If you get any new machine/OS results, please send to:
*
*          ihnp4!castor!pcrat!rick
*
*       and thanks to all that do. Space prevents listing
*       the names of those who have provided some of these
*       results. I'll be forwarding these results to
*       Rheinhold Weicker.*/
```

```
/* Note:   I order the list in increasing performance of the
*       "with registers" benchmark. If the compiler doesn't
*       provide register variables, then the benchmark
*       is the same for both REG and NOREG.
*
* PLEASE:    Send complete information about the machine type,
*       clock speed, OS and C manufacturer/version. If
*       the machine is modified, tell me what was done.
*       On UNIX, execute uname -a and cc -V to get this info.
* 80x8x NOTE: 80x8x benchers: please try to do all memory models
*       for a particular compiler.
*
* APOLOGY (1/30/86):
*    Well, I goofed things up! As pointed out by Haakon Bugge,
*    the line of code marked "GOOF" below was missing from the
*    Dhrystone distribution for the last several months. It
*    *WAS* in a backup copy I made last winter, so no doubt it
*    was victimized by sleepy fingers operating vi!
*
*    The effect of the line missing is that the reported benchmarks
*    are 15% too fast (at least on a 80286). Now, this creates
*    a dilema - do I throw out ALL the data so far collected
*    and use only results from this (corrected) version, or
*    do I just keep collecting data for the old version?
*
*    Since the data collected so far *is* valid as long as it
*    is compared with like data, I have decided to keep
*    TWO lists- one for the old benchmark, and one for the
*    new.  This also gives me an opportunity to correct one
*    other error I made in the instructions for this benchmark.
*    My experience with C compilers has been mostly with
*    UNIX 'pcc' derived compilers, where the 'optimizer' simply
*    fixes sloppy code generation (peephole optimization).
*    But today, there exist C compiler optimizers that will actually
*    perform optimization in the Computer Science sense of the word,
*    by removing, for example, assignments to a variable whose
*    value is never used. Dhrystone, unfortunately, provides
*    lots of opportunities for this sort of optimization.
*
*    I request that benchmarkers re-run this new, corrected
*    version of Dhrystone, turning off or bypassing optimizers
*    which perform more than peephole optimization. Please
*    indicate the version of Dhrystone used when reporting the
*    results to me.
*
* RESULTS BEGIN HERE*/
```

```
/*---------------DHRYSTONE VERSION 1.1 RESULTS
    BEGIN-------------------------
*
* MACHINE  MICROPROCESSOR OPERATING COMPILER DHRYSTONES/SEC.
* TYPE          SYSTEM          NO REG  REGS
* ------------------------- ------------ ----------- ---------------
* Apple IIe 65C02-1.02Mhz DOS 3.3  Aztec CII v1.05i 37  37
* -     Z80-2.5Mhz CPM-80 v2.2 Aztec CII v1.05g 91 91
* -     8086-8Mhz  RMX86 V6 Intel C-86 V2.0 197  203LM??
* IBM PC/XT 8088-4.77Mhz COHERENT 2.3.43 Mark Wiiliams 259  275
* -     8086-8Mhz  RMX86 V6 Intel C-86 V2.0 287  304 ??
* Fortune 32:16 68000-6Mhz V7+sys3+4.1BSD cc   360  346
* PDP-11/34A  w/FP-11C UNIX V7m cc   406  449
* Macintosh512 68000-7.7Mhz Mac ROM O/S DeSmet(C ware) 625 625
* VAX-11/750  w/FPA   UNIX 4.2BSD cc   831  852
* DataMedia 932 68000-10Mhz UNIX sysV cc    837  888
* Plexus P35  68000-12.5Mhz UNIX sysIII cc   835  894
* ATT PC7300  68010-10Mhz UNIX 5.0.3 cc   973  1034
* Compaq II 80286-8Mhz MSDOS 3.1 MS C 3.0  1086  1140 LM
* IBM PC/AT   80286-7.5Mhz   Venix/286 SVR2 cc           1159   1254 *15
* Compaq II 80286-8Mhz MSDOS 3.1 MS C 3.0  1190  1282 MM
* MicroVAX II -    Mach/4.3 cc   1361  1385
* DEC uVAX II -    Ultrix-32m v1.1  cc    1385  1399
* Compaq II 80286-8Mhz MSDOS 3.1 MS C 3.0  1351  1428
* VAX 11/780  -    UNIX 4.2BSD cc  1417  1441
* VAX-780/MA780    Mach/4.3 cc   1428  1470
* VAX 11/780  -    UNIX 5.0.1 cc 4.1.1.31 1650 1640
* Ridge 32C V1 -   ROS 3.3    Ridge C (older) 1628  1695
* Gould PN6005 -   UTX 1.1c+ (4.2) cc    1732  1884
* Gould PN9080 custom ECL UTX-32 1.1C cc  4745 4992
* VAX-784  -    Mach/4.3 cc   5263  5555 &4
* VAX 8600 -    4.3 BSD    cc    6329  6423
* Amdahl 5860 -    UTS sysV cc 1.22       28735  28846
* IBM3090/200 -    ?    ?        31250  31250
*
*
*---------------DHRYSTONE VERSION 1.0 RESULTS BEGIN-------------------------
*
* MACHINE  MICROPROCESSOR OPERATING COMPILER DHRYSTONES/SEC.
* TYPE          SYSTEM          NO REG  REGS
* ------------------------- ------------ ----------- ---------------
* Commodore 64 6510-1MHz C64 ROM   C Power 2.8  36    36
* HP-110 8086-5.33Mhz MSDOS 2.11 Lattice 2.14  284  284
* IBM PC/XT 8088-4.77Mhz PC/IX   cc     271  294
* CCC 3205 -    Xelos(SVR2)   cc     558  592
* Perq-II  2901 bitslice Accent S5c  cc (CMU) 301 301
* IBM PC/XT 8088-4.77Mhz COHERENT 2.3.43 MarkWilliams cc 296 317
* Cosmos 68000-8Mhz UniSoft   cc     305  322
* IBM PC/XT 8088-4.77Mhz Venix/86 2.0 cc    297  324
* DEC PRO 350 11/23         Venix/PRO SVR2 cc             299     325
* IBM PC 8088-4.77Mhz MSDOS 2.0  b16cc 2.0   310  340
* PDP11/23 11/23          Venix (V7)     cc              320     358
* Commodore Amiga  ?    Lattice 3.02  368  371
* PC/XT       8088-4.77Mhz   Venix/86 SYS V cc             339     377
* IBM PC 8088-4.77Mhz MSDOS 2.0  CI-C86 2.20M  390  390
* IBM PC/XT 8088-4.77Mhz PCDOS 2.1 Wizard 2.1  367  403
* IBM PC/XT 8088-4.77Mhz PCDOS 3.1 Lattice 2.15  403  403 @*/
```

```
/* Colex DM-6 68010-8Mhz Unisoft SYSV cc    378  410
* IBM PC 8088-4.77Mhz PCDOS 3.1 Datalight 1.10 416  416
* IBM PC NEC V20-4.77Mhz MSDOS 3.1 MS 3.1      387  420
* IBM PC/XT 8088-4.77Mhz PCDOS 2.1 Microsoft 3.0 390  427
* IBM PC NEC V20-4.77Mhz MSDOS 3.1 MS 3.1 (186)  393  427
* PDP-11/34 -   UNIX V7M cc   387  438
* IBM PC 8088, 4.77mhz PC-DOS 2.1 Aztec C v3.2d 423 454
* Tandy 1000  V20, 4.77mhz  MS-DOS 2.11 Aztec C v3.2d 423  458
* Tandy TRS-16B 68000-6Mhz Xenix 1.3.5 cc  438 458
* PDP-11/34 -   RSTS/E     decus c     438  495
* Onyx C8002  Z8000-4Mhz IS/1 1.1 (V7) cc    476  511
* Tandy TRS-16B 68000-6Mhz Xenix 1.3.5 Green Hills 609 617
* DEC PRO 380 11/73         Venix/PRO SVR2 cc             577    628
* FHL QT+  68000-10Mhz Os9/68000 version 1.3 603  649 FH
* Apollo DN550 68010-?Mhz AegisSR9/IX cc 3.12  666  666
* HP-110 8086-5.33Mhz MSDOS 2.11 Aztec-C    641  676
* ATT PC6300  8086-8Mhz  MSDOS 2.11 b16cc 2.0   632  684
* IBM PC/AT 80286-6Mhz PCDOS 3.0 CI-C86 2.1 666  684
* Tandy 6000  68000-8Mhz Xenix 3.0  cc     694  694
* IBM PC/AT 80286-6Mhz Xenix 3.0 cc    684  704 MM
* Macintosh 68000-7.8Mhz 2M Mac Rom   Mac C 32 bit int 694 704
* Macintosh 68000-7.7Mhz -    MegaMax C 2.0  661  709
* Macintosh512 68000-7.7Mhz Mac ROM O/S DeSmet(C ware) 714 714
* IBM PC/AT 80286-6Mhz Xenix 3.0 cc    704  714 LM
* Codata 3300 68000-8Mhz UniPlus+ (v7) cc   678  725
* WICAT MB 68000-8Mhz System V WICAT C 4.1 585 731 ~
* Cadmus 9000 68010-10Mhz UNIX   cc    714  735
* AT&T 6300   8086-8Mhz      Venix/86 SVR2  cc             668    743
* Cadmus 9790 68010-10Mhz 1MB SVR0,Cadmus3.7 cc   720  747
* NEC PC9801F 8086-8Mhz  PCDOS 2.11 Lattice 2.15  768   -  @
* ATT PC6300  8086-8Mhz  MSDOS 2.11 CI-C86 2.20M  769  769
* Burroughs XE550 68010-10Mhz Centix 2.10 cc  769 769 CT1
* EAGLE/TURBO 8086-8Mhz     Venix/86 SVR2  cc             696    779
* ALTOS 586 8086-10Mhz Xenix 3.0b cc      724  793
* DEC 11/73 J-11 micro Ultrix-11 V3.0 cc  735  793
* ATT 3B2/300 WE32000-?Mhz UNIX 5.0.2 cc    735  806
* Apollo DN320 68010-?Mhz AegisSR9/IX cc 3.12  806  806
* IRIS-2400 68010-10Mhz UNIX System V cc  772  829
* Atari 520ST 68000-8Mhz    TOS            DigResearch    839    846
* IBM PC/AT 80286-6Mhz PCDOS 3.0 MS 3.0(large) 833  847 LM
* WICAT MB 68000-8Mhz System V WICAT C 4.1 675 853 S~
* VAX 11/750 -     Ultrix 1.1 4.2BSD cc   781  862
* CCC 7350A  68000-8MHz UniSoft V.2 cc   821  875
* VAX 11/750 -     UNIX 4.2bsd cc   862  877
* Fast Mac 68000-7.7Mhz -     MegaMax C 2.0  839  904 +
* IBM PC/XT 8086-9.54Mhz PCDOS 3.1 Microsoft 3.0  833  909 C1
* DEC 11/44      Ultrix-11 V3.0 cc   862  909
* Macintosh 68000-7.8Mhz 2M Mac Rom   Mac C 16 bit int 877 909 S
* CCC 3210 -     Xelos R01(SVR2) cc     849  924
* CCC 3220 -             Ed. 7 v2.3     cc      892  925
* IBM PC/AT 80286-6Mhz Xenix 3.0 cc -i    909  925
* AT&T 6300 8086, 8mhz MS-DOS 2.11 Aztec C v3.2d 862 943
* IBM PC/AT 80286-6Mhz Xenix 3.0 cc    892  961
* VAX 11/750 w/FPA   Eunice 3.2 cc    914  976
* IBM PC/XT 8086-9.54Mhz PCDOS 3.1 Wizard 2.1  892  980 C1
* IBM PC/XT 8086-9.54Mhz PCDOS 3.1 Lattice 2.15  980  980 C1*/
```

```
/* Plexus P35 68000-10Mhz UNIX System III cc   984   980
* PDP-11/73 KDJ11-AA 15Mhz UNIX V7M 2.1 cc  862    981
* VAX 11/750  w/FPA   UNIX 4.3bsd cc   994   997
* IRIS-1400 68010-10Mhz UNIX System V cc  909 1000
* IBM PC/AT 80286-6Mhz Venix/86 2.1 cc   961  1000
* IBM PC/AT 80286-6Mhz PCDOS 3.0 b16cc 2.0  943  1063
* Zilog S8000/11 Z8001-5.5Mhz Zeus 3.2 cc  1011 1084
* NSC ICM-3216 NSC 32016-10Mhz UNIX SVR2 cc   1041  1084
* IBM PC/AT 80286-6Mhz PCDOS 3.0 MS 3.0(small) 1063 1086
* VAX 11/750  w/FPA   VMS     VAX-11 C 2.0   958  1091
* Stride 68000-10Mhz System-V/68 cc  1041 1111
* Plexus P/60 MC68000-12.5Mhz UNIX SYSIII Plexus   1111  1111
* ATT PC7300  68010-10Mhz UNIX 5.0.2 cc  1041  1111
* CCC 3230 -    Xelos R01(SVR2) cc    1040  1126
* Stride 68000-12Mhz System-V/68 cc  1063 1136
* IBM PC/AT   80286-6Mhz     Venix/286 SVR2 cc            1056    1149
* Plexus P/60 MC68000-12.5Mhz UNIX SYSIII Plexus   1111  1163 T
* IBM PC/AT 80286-6Mhz PCDOS 3.0 Datalight 1.10 1190 1190
* ATT PC6300+ 80286-6Mhz MSDOS 3.1 b16cc 2.0  1111  1219
* IBM PC/AT 80286-6Mhz PCDOS 3.1 Wizard 2.1 1136 1219
* Sun2/120 68010-10Mhz Sun 4.2BSD cc  1136 1219
* IBM PC/AT 80286-6Mhz PCDOS 3.0 CI-C86 2.20M 1219  1219
* WICAT PB 68000-8Mhz System V WICAT C 4.1 998 1226 ~
* MASSCOMP 500 68010-10MHz RTU V3.0 cc (V3.2) 1156 1238
* Alliant FX/8 IP (68012-12Mhz) Concentrix cc -ip;exec -i  1170 1243 FX
* Cyb DataMate 68010-12.5Mhz Uniplus 5.0 Unisoft cc 1162 1250
* PDP 11/70 -   UNIX 5.2 cc   1162  1250
* IBM PC/AT 80286-6Mhz PCDOS 3.1 Lattice 2.15 1250  1250
* IBM PC/AT 80286-7.5Mhz Venix/86 2.1 cc   1190  1315 *15
* Sun2/120 68010-10Mhz Standalone cc  1219  1315
* Intel 380 80286-8Mhz Xenix R3.0up1 cc  1250 1315 *16
* Sequent Balance 8000 NS32032-10MHz Dynix 2.0 cc 1250 1315 N12
* IBM PC/DSI-32 32032-10Mhz MSDOS 3.1 GreenHills 2.14 1282  1315 C3
* ATT 3B2/400 WE32100-?Mhz UNIX 5.2 cc   1315  1315
* CCC 3250XP  -    Xelos R01(SVR2) cc    1215  1318
* IBM PC/RT 032 RISC(801?)?Mhz BSD 4.2    cc            1248    1333 RT
* DG MV4000 -   AOS/VS 5.00 cc   1333  1333
* IBM PC/AT 80286-8Mhz Venix/86 2.1 cc   1275  1380 *16
* IBM PC/AT 80286-6Mhz MSDOS 3.0 Microsoft 3.0 1250 1388
* ATT PC6300+ 80286-6Mhz MSDOS 3.1 CI-C86 2.20M  1428  1428
* COMPAQ/286  80286-8Mhz     Venix/286 SVR2 cc            1326    1443
* IBM PC/AT   80286-7.5Mhz   Venix/286 SVR2 cc            1333    1449 *15
* WICAT PB 68000-8Mhz System V WICAT C 4.1 1169 1464 S~
* Tandy II/6000 68000-8Mhz Xenix 3.0 cc    1384  1477
* MicroVAX II -    Mach/4.3 cc   1513  1536
* WICAT MB 68000-12.5Mhz System V WICAT C 4.1 1246 1537 ~
* IBM PC/AT   80286-9Mhz     SCO Xenix V    cc            1540    1556 *18
* Cyb DataMate 68010-12.5Mhz Uniplus 5.0 Unisoft cc 1470 1562 S
* VAX 11/780  -    UNIX 5.2 cc   1515  1562
* MicroVAX-II -    -     -     1562  1612
* VAX-780/MA780    Mach/4.3 cc   1587  1612
* VAX 11/780  -    UNIX 4.3bsd cc   1646  1662
* Apollo DN660 -   AegisSR9/IX cc 3.12   1666  1666
* ATT 3B20 -    UNIX 5.2 cc   1515  1724
* NEC PC-98XA 80286-8Mhz PCDOS 3.1 Lattice 2.15  1724  1724 @
* HP9000-500  B series CPU  HP-UX 4.02 cc   1724  -
* Ridge 32C V1 -   ROS 3.3    Ridge C (older) 1776  -
* IBM PC/STD  80286-8Mhz MSDOS 3.0  Microsoft 3.0 1724 1785 C2*/
```

```
/* WICAT MB 68000-12.5Mhz System V WICAT C 4.1 1450 1814 S~
* WICAT PB 68000-12.5Mhz System V WICAT C 4.1 1530 1898 ~
* DEC-2065 KL10-Model B TOPS-20 6.1FT5 Port. C Comp. 1937 1946
* Gould PN6005 -   UTX 1.1(4.2BSD) cc   1675  1964
* DEC2060  KL-10   TOPS-20    cc    2000  2000 NM
* Intel 310AP 80286-8Mhz Xenix 3.0 cc   1893  2009
* VAX 11/785  -    UNIX 5.2 cc   2083  2083
* VAX 11/785  -    VMS     VAX-11 C 2.0  2083  2083
* VAX 11/785  -    UNIX SVR2  cc    2123  2083
* VAX 11/785  -             ULTRIX-32 1.1  cc     2083    2091
* VAX 11/785  -    UNIX 4.3bsd cc   2135  2136
* WICAT PB 68000-12.5Mhz System V WICAT C 4.1 1780 2233 S~
* Pyramid 90x -    OSx 2.3    cc    2272  2272
* Pyramid 90x FPA,cache,4Mb OSx 2.5   cc no -O 2777 2777
* Pyramid 90x w/cache    OSx 2.5    cc w/-O    3333  3333
* IBM-4341-II -    VM/SP3     Waterloo C 1.2 3333 3333
* IRIS-2400T  68020-16.67Mhz UNIX System V cc   3105 3401
* Celerity C-1200 ?   UNIX 4.2BSD cc   3485  3468
* SUN 3/75 68020-16.67Mhz SUN 4.2 V3 cc   3333 3571
* IBM-4341 Model 12 UTS 5.0   ?    3685  3685
* SUN-3/160   68020-16.67Mhz Sun 4.2 V3.0A cc    3381    3764
* Sun 3/180 68020-16.67Mhz Sun 4.2  cc   3333  3846
* IBM-4341 Model 12 UTS 5.0   ?    3910  3910 MN
* MC 5400  68020-16.67MHz RTU V3.0 cc (V4.0) 3952 4054
* Intel 386/20 80386-12.5Mhz PMON debugger Intel C386v0.2 4149 4386
* NCR Tower32 68020-16.67Mhz SYS 5.0 Rel 2.0 cc          3846  4545
* MC 5600/5700 68020-16.67MHz RTU V3.0 cc (V4.0) 4504 4746 %
* Intel 386/20 80386-12.5Mhz PMON debugger Intel C386v0.2 4534 4794 i1
* Intel 386/20 80386-16Mhz PMON debugger Intel C386v0.2 5304 5607
* Gould PN9080 custom ECL UTX-32 1.1C cc  5369 5676
* Gould 1460-342 ECL proc   UTX/32 1.1/c   cc             5342    5677 G1
* VAX-784  -    Mach/4.3 cc   5882  5882 &4
* Intel 386/20 80386-16Mhz PMON debugger Intel C386v0.2 5801 6133 i1
* VAX 8600 -    UNIX 4.3bsd cc   7024  7088
* VAX 8600 -    VMS     VAX-11 C 2.0  7142  7142
* Alliant FX/8 CE  Concentrix cc -ce;exec -c  6952  7655 FX
* CCI POWER 6/32   COS(SV+4.2) cc   7500  7800
* CCI POWER 6/32   POWER 6 UNIX/V cc   8236  8498
* CCI POWER 6/32   4.2 Rel. 1.2b cc   8963  9544
* Sperry (CCI Power 6)   4.2BSD     cc    9345   10000
* CRAY-X-MP/12   105Mhz   COS 1.14 Cray C       10204   10204
* IBM-3083 -    UTS 5.0 Rel 1 cc       16666   12500
* CRAY-1A      80Mhz   CTSS    Cray C 2.0    12100   13888
* IBM-3083 -    VM/CMS HPO 3.4 Waterloo C 1.2 13889 13889
* Amdahl 470 V/8   UTS/V 5.2      cc v1.23      15560   15560
* CRAY-X-MP/48   105Mhz   CTSS    Cray C 2.0    15625   17857
* Amdahl 580  -    UTS 5.0 Rel 1.2 cc v1.5       23076   23076
* Amdahl 5860      UTS/V 5.2      cc v1.23      28970   28970*/
```

```
/* NOTE
*   *   Crystal changed from 'stock' to listed value.
*   +   This Macintosh was upgraded from 128K to 512K in such a way that
*       the new 384K of memory is not slowed down by video generator accesses.
*   %   Single processor; MC == MASSCOMP
*   NM  A version 7 C compiler written at New Mexico Tech.
*   @   vanilla Lattice compiler used with MicroPro standard library
*   S   Shorts used instead of ints
*   T  with Chris Torek's patches (whatever they are).
*   ~   For WICAT Systems: MB=MultiBus, PB=Proprietary Bus
*   LM  Large Memory Model. (Otherwise, all 80x8x results are small model)
*   MM  Medium Memory Model. (Otherwise, all 80x8x results are small model)
*   C1  Univation PC TURBO Co-processor; 9.54Mhz 8086, 640K RAM
*   C2  Seattle Telecom STD-286 board
*   C3  Definicon DSI-32 coprocessor
*   C?  Unknown co-processor board?
*   CT1 Convergent Technologies MegaFrame, 1 processor.
*   MN  Using Mike Newtons 'optimizer' (see net.sources).
*   G1  This Gould machine has 2 processors and was able to run 2 dhrystone
*       Benchmarks in parallel with no slowdown.
*   FH  FHC == Frank Hogg Labs (Hazelwood Uniquad 2 in an FHL box).
*   FX  The Alliant FX/8 is a system consisting of 1-8 CEs (computation
*   engines) and 1-12 IPs (interactive processors). Note N8 applies.
*   RT  This is one of the RT's that CMU has been using for awhile. I'm
*   not sure that this is identical to the machine that IBM is selling
*   to the public.
*   i1  Normally, the 386/20 starter kit has a 16k direct mapped cache
*   which inserts 2 or 3 wait states on a write thru. These results
*   were obtained by disabling the write-thru, or essentially turning
*   the cache into 0 wait state memory.
*   Nnn This machine has multiple processors, allowing "nn" copies of the
*   benchmark to run in the same time as 1 copy.
*   &nn This machine has "nn" processors, and the benchmark results were
*   obtained by having all "nn" processors working on 1 copy of dhrystone.
*   (Note, this is different than Nnn. Salesmen like this measure).
*   ?   I don't trust results marked with '?'. These were sent to me with
*       either incomplete info, or with times that just don't make sense.
*   ?? means I think the performance is too poor, ?! means too good.
*       If anybody can confirm these figures, please respond.
*
*  ABBREVIATIONS
*  CCC   Concurrent Computer Corp. (was Perkin-Elmer)
*  MC Masscomp
*
*--------------------------------RESULTS
    END----------------------------------*/
```

```
/* The following program contains statements of a high-level programming
*  language (C) in a distribution considered representative:
*
*  assignments      53%
*  control statements    32%
*  procedure, function calls 15%
*
*  100 statements are dynamically executed. The program is balanced with
*  respect to the three aspects:
*     - statement type
*     - operand type (for simple data types)
*     - operand access
*        operand global, local, parameter, or constant.
*
*  The combination of these three aspects is balanced only approximately.
*
*  The program does not compute anything meaningfull, but it is
*  syntactically and semantically correct.
*
*/

/* Accuracy of timings and human fatigue controlled by next two lines */
/*#define LOOPS 5000    /* Use this for slow or 16 bit machines */
/*#define LOOPS 50000   /* Use this for slow or 16 bit machines */
#define LOOPS 500000    /* Use this for faster machines */

/* Compiler dependent options */
#undef  NOENUM        /* Define if compiler has no enum's */
#undef  NOSTRUCTASSIGN  /* Define if compiler can't assign structures */

/* define only one of the next three defines */
/*#define GETRUSAGE   /* Use getrusage(2) time function */
#define TIMES      /* Use times(2) time function
/*#define TIME     /* Use time(2) time function */

/* define the granularity of your times(2) function (when used) */
/*#define HZ  60    /* times(2) returns 1/60 second (most) */
#define HZ 60   /* times(2) returns 1/60 second (most)
/*#define HZ  100     /* times(2) returns 1/100 second (WECo) */

/* for compatibility with goofed up version */
/*#define GOOF     /* Define if you want the goofed up version */

#ifdef GOOF
char  Version[] = "1.0";
#else
char  Version[] = "1.1";
#endif

#ifdef  NOSTRUCTASSIGN
#define structassign(d, s) memcpy(&(d), &(s), sizeof(d))
#else
#define structassign(d, s) d = s
#endif
```

```
#ifdef  NOENUM
#define Ident1  1
#define Ident2  2
#define Ident3  3
#define Ident4  4
#define Ident5  5
typedef int Enumeration;
#else
typedef enum {Ident1, Ident2, Ident3, Ident4, Ident5} Enumeration;
#endif

typedef int OneToThirty;
typedef int OneToFifty;
typedef char CapitalLetter;
typedef char String30[31];
typedef int Array1Dim[51];
typedef int Array2Dim[51][51];

struct  Record
{
   struct Record   *PtrComp;
   Enumeration   Discr;
   Enumeration   EnumComp;
   OneToFifty    IntComp;
   String30   StringComp;
};

typedef struct Record  RecordType;
typedef RecordType * RecordPtr;
typedef int   boolean;

#define NULL    0
#define TRUE    1
#define FALSE   0

#ifndef REG
#define REG
#endif

extern Enumeration Func1();
extern boolean  Func2();

#ifdef TIMES
#include <sys/param.h>
#include <sys/types.h>
#include <sys/times.h>
#endif
#ifdef GETRUSAGE
#include <sys/time.h>
#include <sys/resource.h>
#endif
```

```
main()
{
   Proc0();
   exit(0);
}

/*
* Package 1
*/
int     IntGlob;
boolean    BoolGlob;
char    Char1Glob;
char    Char2Glob;
Array1Dim  Array1Glob;
Array2Dim  Array2Glob;
RecordPtr  PtrGlb;
RecordPtr  PtrGlbNext;

Proc0()
{
   OneToFifty    IntLoc1;
   REG OneToFifty  IntLoc2;
   OneToFifty    IntLoc3;
   REG char   CharLoc;
   REG char   CharIndex;
   Enumeration   EnumLoc;
   String30   String1Loc;
   String30   String2Loc;
   extern char   *malloc();

   register unsigned int i;
   #ifdef TIME
   long       time();
   long       starttime;
   long       benchtime;
   long       nulltime;

   starttime = time( (long *) 0);
   for (i = 0; i < LOOPS; ++i);
   nulltime = time( (long *) 0) - starttime; /* Computes o'head of loop */
   #endif
   #ifdef TIMES
   time_t        starttime;
   time_t        benchtime;
   time_t        nulltime;
   struct tms    tms;
```

```
times(&tms); starttime = tms.tms_utime;
for (i = 0; i < LOOPS; ++i);
times(&tms);
nulltime = tms.tms_utime - starttime; /* Computes overhead of looping */
#endif
#ifdef GETRUSAGE
struct rusage starttime;
struct rusage endtime;
struct timeval nulltime;

getrusage(RUSAGE_SELF, &starttime);
for (i = 0; i < LOOPS; ++i);
getrusage(RUSAGE_SELF, &endtime);
nulltime.tv_sec = endtime.ru_utime.tv_sec - starttime.ru_utime.tv_sec;
nulltime.tv_usec = endtime.ru_utime.tv_usec - starttime.ru_utime.tv_usec;
#endif

PtrGlbNext = (RecordPtr) malloc(sizeof(RecordType));
PtrGlb = (RecordPtr) malloc(sizeof(RecordType));
PtrGlb->PtrComp = PtrGlbNext;
PtrGlb->Discr = Ident1;
PtrGlb->EnumComp = Ident3;
PtrGlb->IntComp = 40;
strcpy(PtrGlb->StringComp, "DHRYSTONE PROGRAM, SOME STRING");
#ifndef GOOF
strcpy(String1Loc, "DHRYSTONE PROGRAM, 1'ST STRING"); /*GOOF*/
#endif
Array2Glob[8][7] = 10; /* Was missing in published program */

/*****************
-- Start Timer --
*****************/
#ifdef TIME
starttime = time( (long *) 0);
#endif
#ifdef TIMES
times(&tms); starttime = tms.tms_utime;
#endif
#ifdef GETRUSAGE
getrusage (RUSAGE_SELF, &starttime);
#endif
for (i = 0; i < LOOPS; ++i)
{

   Proc5();
   Proc4();
   IntLoc1 = 2;
   IntLoc2 = 3;
   strcpy(String2Loc, "DHRYSTONE PROGRAM, 2'ND STRING");
   EnumLoc = Ident2;
   BoolGlob = ! Func2(String1Loc, String2Loc);
```

```
            while (IntLoc1 < IntLoc2)
            {
               IntLoc3 = 5 * IntLoc1 - IntLoc2;
               Proc7(IntLoc1, IntLoc2, &IntLoc3);
               ++IntLoc1;
            }
            Proc8(Array1Glob, Array2Glob, IntLoc1, IntLoc3);
            Proc1(PtrGlb);
            for (CharIndex = 'A'; CharIndex <= Char2Glob; ++CharIndex)
               if (EnumLoc == Func1(CharIndex, 'C'))
                  Proc6(Ident1, &EnumLoc);
            IntLoc3 = IntLoc2 * IntLoc1;
            IntLoc2 = IntLoc3 / IntLoc1;
            IntLoc2 = 7 * (IntLoc3 - IntLoc2) - IntLoc1;
            Proc2(&IntLoc1);
         }

         /*****************
         -- Stop Timer --
         *****************/

         #ifdef TIME
         benchtime = time( (long *) 0) - starttime - nulltime;
         printf("Dhrystone(%s) time for %ld passes = %ld\n",
         Version,
         (long) LOOPS, benchtime);
         printf("This machine benchmarks at %ld dhrystones/second\n",
         ((long) LOOPS) / benchtime);
         #endif
         #ifdef TIMES
         times(&tms);
         benchtime = tms.tms_utime - starttime - nulltime;
         printf("Dhrystone(%s) time for %ld passes = %ld\n",
         Version,
         (long) LOOPS, benchtime/HZ);
         printf("This machine benchmarks at %ld dhrystones/second\n",
         ((long) LOOPS) * HZ / benchtime);
         #endif
         #ifdef GETRUSAGE
         getrusage(RUSAGE_SELF, &endtime);
         {
            double t = (double)(endtime.ru_utime.tv_sec
            - starttime.ru_utime.tv_sec
            - nulltime.tv_sec)
            + (double)(endtime.ru_utime.tv_usec
            - starttime.ru_utime.tv_usec
            - nulltime.tv_usec) * 1e-6;
            printf("Dhrystone(%s) time for %ld passes = %.1f\n",
            Version,
            (long)LOOPS,
            t);
            printf("This machine benchmarks at %.0f dhrystones/second\n",
            (double)LOOPS / t);
         }
         #endif
      }
```

```
Proc1(PtrParIn)
REG RecordPtr PtrParIn;
{
   #define NextRecord (*(PtrParIn->PtrComp))

   structassign(NextRecord, *PtrGlb);
   PtrParIn->IntComp = 5;
   NextRecord.IntComp = PtrParIn->IntComp;
   NextRecord.PtrComp = PtrParIn->PtrComp;
   Proc3(NextRecord.PtrComp);
   if (NextRecord.Discr == Ident1)
      {
      NextRecord.IntComp = 6;
      Proc6(PtrParIn->EnumComp, &NextRecord.EnumComp);
      NextRecord.PtrComp = PtrGlb->PtrComp;
      Proc7(NextRecord.IntComp, 10, &NextRecord.IntComp);
   }
   else
      structassign(*PtrParIn, NextRecord);

   #undef  NextRecord
}

Proc2(IntParIO)
OneToFifty *IntParIO;
{
   REG OneToFifty  IntLoc;
   REG Enumeration    EnumLoc;

   IntLoc = *IntParIO + 10;
   for(;;)
   {
      if (Char1Glob == 'A')
      {
         --IntLoc;
         *IntParIO = IntLoc - IntGlob;
         EnumLoc = Ident1;
      }
      if (EnumLoc == Ident1)
         break;
   }
}

Proc3(PtrParOut)
RecordPtr  *PtrParOut;
{
   if (PtrGlb != NULL)
      *PtrParOut = PtrGlb->PtrComp;
   else
      IntGlob = 100;
   Proc7(10, IntGlob, &PtrGlb->IntComp);
}
```

```
Proc4()
{
   REG boolean BoolLoc;

   BoolLoc = Char1Glob == 'A';
   BoolLoc |= BoolGlob;
   Char2Glob = 'B';
}

Proc5()
{
   Char1Glob = 'A';
   BoolGlob = FALSE;
}

extern boolean Func3();

Proc6(EnumParIn, EnumParOut)
REG Enumeration EnumParIn;
REG Enumeration *EnumParOut;
{
   *EnumParOut = EnumParIn;
   if (! Func3(EnumParIn) )
      *EnumParOut = Ident4;
   switch (EnumParIn)
   {
      case Ident1:  *EnumParOut = Ident1; break;
      case Ident2:  if (IntGlob > 100) *EnumParOut = Ident1;
      else *EnumParOut = Ident4;
      break;
      case Ident3:  *EnumParOut = Ident2; break;
      case Ident4:  break;
      case Ident5:  *EnumParOut = Ident3;
   }
}

Proc7(IntParI1, IntParI2, IntParOut)
OneToFifty IntParI1;
OneToFifty IntParI2;
OneToFifty *IntParOut;
{
   REG OneToFifty IntLoc;

   IntLoc = IntParI1 + 2;
   *IntParOut = IntParI2 + IntLoc;
}
```

```
Proc8(Array1Par, Array2Par, IntParI1, IntParI2)
Array1Dim  Array1Par;
Array2Dim  Array2Par;
OneToFifty IntParI1;
OneToFifty IntParI2;
{
   REG OneToFifty IntLoc;
   REG OneToFifty IntIndex;

   IntLoc = IntParI1 + 5;
   Array1Par[IntLoc] = IntParI2;
   Array1Par[IntLoc+1] = Array1Par[IntLoc];
   Array1Par[IntLoc+30] = IntLoc;
   for (IntIndex = IntLoc; IntIndex <= (IntLoc+1); ++IntIndex)
      Array2Par[IntLoc][IntIndex] = IntLoc;
   ++Array2Par[IntLoc][IntLoc-1];
   Array2Par[IntLoc+20][IntLoc] = Array1Par[IntLoc];
   IntGlob = 5;
}

Enumeration Func1(CharPar1, CharPar2)
CapitalLetter CharPar1;
CapitalLetter CharPar2;
{
   REG CapitalLetter CharLoc1;
   REG CapitalLetter CharLoc2;

   CharLoc1 = CharPar1;
   CharLoc2 = CharLoc1;
   if (CharLoc2 != CharPar2)
      return (Ident1);
   else
      return (Ident2);
}

boolean Func2(StrParI1, StrParI2)
String30 StrParI1;
String30 StrParI2;
{
   REG OneToThirty    IntLoc;
   REG CapitalLetter CharLoc;
```

```
      IntLoc = 1;
      while (IntLoc <= 1)
         if (Func1(StrParI1[IntLoc], StrParI2[IntLoc+1]) == Ident1)
         {
            CharLoc = 'A';
            ++IntLoc;
         }
      if (CharLoc >= 'W' && CharLoc <= 'Z')
         IntLoc = 7;
      if (CharLoc == 'X')
         return(TRUE);
      else
      {
         if (strcmp(StrParI1, StrParI2) > 0)
         {
            IntLoc += 7;
            return (TRUE);
         }
         else
            return (FALSE);
      }
}

boolean Func3(EnumParIn)
REG Enumeration EnumParIn;
{
   REG Enumeration EnumLoc;

   EnumLoc = EnumParIn;
   if (EnumLoc == Ident3) return (TRUE);
      return (FALSE);
}

#ifdef  NOSTRUCTASSIGN
memcpy(d, s, l)
register char *d;
register char *s;
register int  l;
{
   while (l--) *d++ = *s++;
}
#endif
/* ---------- */
```

Fonte: Weicker (1984)