

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ  
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E  
INFORMÁTICA INDUSTRIAL

EDUARDO PETERS

COPROCESSADOR PARA ACELERAÇÃO DE APLICAÇÕES  
DESENVOLVIDAS UTILIZANDO PARADIGMA ORIENTADO A  
NOTIFICAÇÕES

DISSERTAÇÃO

CURITIBA  
2012

EDUARDO PETERS

**COPROCESSADOR PARA ACELERAÇÃO DE APLICAÇÕES  
DESENVOLVIDAS UTILIZANDO PARADIGMA ORIENTADO A  
NOTIFICAÇÕES**

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná, como requisito parcial para obtenção do grau de "Mestre em Ciências" – Área de Concentração: Engenharia de Automação e Sistemas.

Orientador: Prof. Dr. Volnei Antônio Pedroni  
Co-Orientador: Prof. Dr. Jean Marcelo Simão.

CURITIBA  
2012

---

Dados Internacionais de Catalogação na Publicação

---

- P481 Peters, Eduardo  
Coprocessador para aceleração de aplicações desenvolvidas utilizando paradigma orientado a notificações / Eduardo Peters. – 2012.  
94 f. : il. ; 30 cm
- Orientador: Volnei Antônio Pedroni.  
Coorientador: Jean Marcelo Simão.  
Dissertação (Mestrado) – Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. Curitiba, 2012.  
Bibliografia: f. 85-89.
1. Paradigma orientado a notificações. 2. Hardware. 3. Coprocessadores. 4. Software – Desenvolvimento. 5. Simulação (Computadores). 6. Engenharia elétrica – Dissertações. I. Pedroni, Volnei Antônio, orient. II. Simão, Jean Marcelo, coorient. III. Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. IV. Título.

CDD (22. ed.) 621.3

---

Biblioteca Central da UTFPR, Campus Curitiba

Título da Dissertação Nº 602

## **“Coprocessador para Aceleração de Aplicações Desenvolvidas Utilizando Paradigma Orientado a Notificações”**

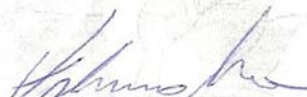
por

**Eduardo Peters**

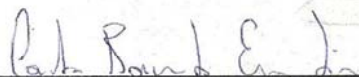
Esta dissertação foi apresentada como requisito parcial à obtenção do grau de MESTRE EM CIÊNCIAS – Área de Concentração: Engenharia de Automação e Sistemas, pelo Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial – CPGEI – da Universidade Tecnológica Federal do Paraná – UTFPR – Câmpus Curitiba, às 9h30min do dia 31 de julho de 2012. O trabalho foi aprovado pela Banca Examinadora, composta pelos professores:



Prof. Volnei Antônio Pedroni, Dr.  
(Presidente – UTFPR)



Prof. Fabiano Silva, Dr.  
(UFPR)



Prof. Carlos Raimundo Érig Lima, Dr.  
(UTFPR)

Visto da coordenação:



Prof. Ricardo Lüders, Dr.  
(Coordenador do CPGEI)

Aos meus pais Amauri e Elizabeth, que sempre me incentivaram e apoiaram em todas as fases de minha vida.

A minha noiva Mari Ane, que com carinho, compreensão e paciência ajudou-me a concretizar este sonho.

## AGRADECIMENTOS

Quero agradecer primeiramente a Deus, por tudo.

Agradeço a minha família, que me guiou durante toda a vida e sem a qual não teria chances de ter chegado até aqui.

Agradeço a minha noiva, Mari Ane Tromm, que me incentivou e apoiou durante toda a realização deste trabalho.

Ao meu orientador Volnei Antônio Pedroni, que com sabedoria e paciência me conduziu durante todo o período de mestrado. Pela sabedoria, direcionamentos e horas dispendidas na leitura de meus trabalhos.

Ao colega Ricardo Pereira Jasinski, a quem devo muito do meu conhecimento. Suas ideias e contribuições foram fundamentais para o desenvolvimento deste trabalho.

Ao Professor Jean Marcelo Simão, criador do Paradigma Orientado à Notificações, pelos direcionamentos e tempo dispendido na revisão de meus trabalhos.

Aos inventores do PON Professor Jean Marcelo Simão e Professor Paulo César Stadysz, bem como aos demais colaboradores da implementação em hardware do PON original, Robson Ribeiro Linhares, Fernando Augusto de Witt e Carlos Raimundo Erig Lima.

Agradeço aos colegas da Solvis, que pelas valiosas discussões contribuem grandemente ao meu aprendizado.

A todos os professores que passaram pela minha carreira acadêmica, pelo conhecimento repassado em suas aulas e UTFPR, pelas instalações e os materiais consumidos durante a realização desse trabalho.

Por fim, agradeço a todos os autores e todas as pessoas, que direta ou indiretamente contribuíram para a execução deste trabalho.

## RESUMO

Peters, Eduardo. Coprocessador para Aceleração de Aplicações Desenvolvidas Utilizando Paradigma Orientado a Notificações. 2012. 94 f. Dissertação - Programa de Pós-Graduação em Eng. Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2012.

Este trabalho apresenta um novo *hardware* coprocessador para acelerar aplicações desenvolvidas utilizando-se o Paradigma Orientado a Notificações (PON), cuja essência se constitui em uma nova forma de influência causal baseada na colaboração pontual entre entidades granulares e notificantes. Uma aplicação PON apresenta as vantagens da programação baseada em eventos e da programação declarativa, possibilitando um desenvolvimento de alto nível, auxiliando o reuso de código e reduzindo o processamento desnecessário existente das aplicações desenvolvidas com os paradigmas atuais. Como uma aplicação PON é composta de uma cadeia de pequenas entidades computacionais, comunicando-se somente quando necessário, é um bom candidato a implementação direta em *hardware*. Para investigar este pressuposto, criou-se um coprocessador capaz de executar aplicações PON existentes. O coprocessador foi desenvolvido utilizando-se linguagem VHDL e testado em FPGAs, mostrando um decréscimo de 96% do número de ciclos de *clock* utilizados por um programa se comparado a implementação puramente em *software* da mesma aplicação, considerando uma dada materialização em um *framework* em PON.

Palavras chave: Paradigma Orientado a Notificações, *Hardware* coprocessador, Aceleração de *software*.

## ABSTRACT

Peters, Eduardo. Coprocessor for Accelerating Applications Developed with Notification Oriented Paradigm. 2012. 94 f. Dissertação - Programa de Pós-Graduação em Eng. Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2012.

This work presents a new hardware coprocessor to accelerate applications developed using the Notification-Oriented Paradigm (NOP). A NOP application has the advantages of both event-based programming and declarative programming, enabling higher level software development, improving code reuse, and reducing the number of unnecessary computations. Because a NOP application is composed of a network of small computational entities communicating only when needed, it is a good candidate for a direct hardware implementation. In order to investigate this assumption, a coprocessor that is able to run existing NOP applications was created. The coprocessor was developed in VHDL and tested in FPGAs, providing a decrease of 96% in the number of clock cycles compared to a purely software implementation.

Keywords: Notification Oriented Paradigm, Hardware coprocessor, software accelerating.



## LISTA DE FIGURAS

Figura 1 - Comparação de vendas entre PCs e Sistemas Embarcados .....	14
Figura 2 - Preenchendo o espaço entre o processador e o ASIC .....	16
Figura 3 - <i>Chips</i> disponíveis para implementação de circuitos lógicos .....	21
Figura 4 - Relação entre as diferentes tecnologias de <i>hardware</i> .....	21
Figura 5 - Estrutura geral de uma FPGA .....	23
Figura 6 - Estrutura de uma LUT de duas entradas .....	24
Figura 7 - Flexibilidade dos dispositivos de <i>hardware</i> .....	25
Figura 8 - Implementação conceitual do barramento Avalon .....	31
Figura 9 – Modelo simplificado de um coprocessador .....	34
Figura 10 - Evolução das linguagens em relação aos paradigmas de programação .....	37
Figura 11 - Classificação dos atuais paradigmas .....	38
Figura 12. Exemplo da cadeia de notificações dos componentes do PON .....	41
Figura 13 - Modelo Clássico de Desenvolvimento .....	46
Figura 14 - Fluxo de desenvolvimento de sistemas baseados em HDLs .....	47
Figura 15 - Ambiente integrado de desenvolvimento para VHDL .....	48
Figura 16 - Ambiente integrado de desenvolvimento SBT .....	51
Figura 17 - Representação dos componentes implementados no periférico .....	53
Figura 18 - Representação geral do sistema .....	54
Figura 19 - Wizard para configuração do CoPON .....	56
Figura 20 - Representação em <i>hardware</i> de um Atributo .....	57
Figura 21 - Representação em blocos da Premissa .....	59
Figura 22 - Estrutura interna da Premissa .....	60
Figura 23 - Representação do bloco de <i>hardware</i> da Condição. ....	61
Figura 24 - Representação interna da Condição .....	62
Figura 25 - Representação da FIFO de armazenamento das Regras aprovadas. ....	63
Figura 26 - Representação geral do periférico. ....	64
Figura 27 – <i>Hardware</i> Acelerador .....	67
Figura 28 - Relação entre os objetos principais e colaboradores .....	68
Figura 29 – Código desenvolvido utilizando o <i>framework</i> PON original .....	70
Figura 31 – Acréscimo de <i>hardware</i> devido à adição do coprocessador .....	75
Figura 32 - Frequência máxima de operação ( $f_{MAX}$ ) .....	77
Figura 33 - SoC criado para execução das aplicações .....	79
Figura 34 – Número de ciclos de <i>clock</i> do momento da alteração de um Atributo até a aprovação de uma Regra para execução .....	81
Figura 35 – Número de ciclos de <i>clock</i> utilizados pelo sistema completo .....	82

## LISTA DE TABELAS

Tabela 1 – Comparação entre número de ciclos de <i>clock</i> utilizados desde a alteração de um Atributo até a aprovação de uma Regra para execução. ....	80
Tabela 2 – Comparação entre número de ciclos de <i>clock</i> utilizados pelo sistema em aplicações com e sem <i>hardware</i> acelerador .....	81

## LISTA DE QUADROS

Quadro 1 - Mapa de memória de um Atributo PON .....	57
Quadro 2 - Mapa de memória de uma Premissa PON.....	58
Quadro 3 - Operações realizadas pela Premissa.....	58
Quadro 4 - Mapa de memória de uma Condição .....	61
Quadro 5 - Mapa de memória do periférico.....	66
Quadro 6 - Configurações utilizadas para avaliação de custos de <i>hardware</i> .....	74

## LISTA DE ABREVIATURAS E SIGLAS

A/D	Analógico/Digital
ASIC	Application-specific integrated circuit
CPLD	Complex Programmable Logic Device
CPU	Unidade central de processamento
ES	Sistema Embarcado
FBE	Elemento da base de fatos
FPGA	Field Programmable Gate Array
HDL	Linguagem de descrição de hardware
IEP	Pipeline de execução de inteiros
IP	Propriedade intelectual
LISP	Processamento de Listas
LU	Unidade Lógica
LUT	Lookup Table
Opcode	Código de operação
PC	Contador de Programa
PF	Paradigma Funcional
PL	Paradigma Lógico
PLD	Dispositivo lógico programável
PON	Paradigma Orientado a Notificações
POO	Programação Orientada a Objetos
PP	Paradigma Procedimental
SBR	Sistemas Baseados em Regras
SOC	Sistema em um chip
SOPC	Sistema em um chip programável
VHDL	VHSIC hardware description language

## SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO</b>	<b>13</b>
1.1	APRESENTAÇÃO DO TEMA	13
1.2	OBJETIVOS	17
1.3	ESTRUTURA DA DISSERTAÇÃO	18
<b>2</b>	<b>CONCEITOS E FUNDAMENTAÇÃO TEÓRICA</b>	<b>19</b>
2.1	INTRODUÇÃO	19
2.2	DISPOSITIVOS LÓGICOS PROGRAMÁVEIS	19
2.3	CPU EM LÓGICA PROGRAMÁVEL	26
2.4	NIOS II	28
2.4.1	Barramento Avalon	30
2.4.2	Interface Avalon MM	33
2.5	COPROCESSADORES	34
2.6	PARADIGMAS DE PROGRAMAÇÃO	35
2.7	PARADIGMA ORIENTADO A NOTIFICAÇÕES	39
2.7.1	Elementos da Base de Fatos	41
2.7.2	Atributos	41
2.7.3	Premissas	42
2.7.4	Condições	42
2.7.5	Regras	43
2.7.6	Ações	43
2.7.7	Instigações	43
2.7.8	Métodos	44
2.8	REFLEXÃO	44
<b>3</b>	<b>DESENVOLVIMENTO</b>	<b>45</b>
3.1	INTRODUÇÃO	45
3.2	DESENVOLVIMENTO DO HARDWARE EM HDL	46
3.2.1	Especificação e Desenho	47
3.2.2	Micro Desenho	47
3.2.3	Codificação em Nível de Transferência de Registro – RTL	48
3.2.4	Simulação e Testes	49
3.2.5	Síntese, Alocação e Roteamento	49
3.3	DESENVOLVIMENTO DE SOFTWARE	50
3.4	ESTRUTURA DO PON EM HARDWARE	52
3.4.1	Atributos	56
3.4.2	Premissas	58
3.4.3	Condição	60
3.4.4	Regra	62
3.4.5	Interface e Lógica de Configuração	63
3.4.6	Interface com o NIOS II	64
3.4.7	Mapa de Memória	65
3.5	ALTERAÇÕES NO FRAMEWORK	67
3.6	FLUXO DE EXECUÇÃO DO COPROCESSADOR	70
3.7	REFLEXÃO	73
<b>4</b>	<b>RESULTADOS</b>	<b>74</b>
4.1	INTRODUÇÃO	74
4.1.1	Comparações de Utilização de Recursos de Hardware	74
4.1.2	Comparações de Desempenho de Aplicação	77
<b>5</b>	<b>CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS</b>	<b>83</b>

5.1 TRABALHOS FUTUROS.....	84
<b>REFERÊNCIAS.....</b>	<b>85</b>
<b>APÊNDICES .....</b>	<b>90</b>

# 1 INTRODUÇÃO

## 1.1 APRESENTAÇÃO DO TEMA

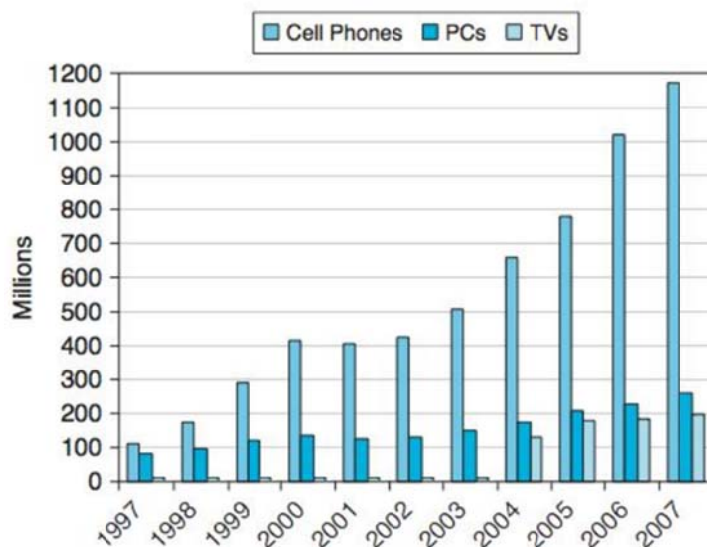
Grande parte dos sistemas computacionais atuais consiste em sistemas embarcados, os quais são sistemas computacionais especializados em desempenhar funções específicas.

Segundo Wolf (2002, p.136), “um Sistema Embarcado (ES) é qualquer computador que seja um componente de um sistema maior e que possui processamento próprio”. Por serem projetados para executar uma ou algumas poucas funcionalidades específicas, os sistemas embarcados utilizam a menor quantidade possível de *hardware* para desenvolver suas tarefas, garantindo assim um custo reduzido, menor consumo de energia e devido à sua especialização, maior eficiência.

No mundo atual o número de dispositivos embarcados é muito superior ao número de computadores de uso pessoal, como pode-se observar na Figura 1. Esta comparação é feita somente utilizando um tipo específico de sistema embarcado, os telefone celulares. Entretanto este número pode ser facilmente multiplicado se considerarmos outros tipos de sistemas embarcados, como eletrodomésticos, players multimídia, equipamentos para automação, eletrônica automotiva etc.

Em contrapartida, por se tratar de um sistema com recursos reduzidos, existem grandes limitações de processamento, consumo de memória e consumo de energia, tornando complexo o desenvolvimento de *software* para estes sistemas.

Neste âmbito, os atuais paradigmas de programação, com suas respectivas linguagens e *frameworks*, não contribuem para a construção de programas eficientes. Nestes, as expressões causais (e.g. comandos se-então) e os dados (e.g. variáveis) são tratados como entidades passivas, as quais são relacionadas por meio de pesquisa estabelecida pelo fluxo de execução dos programas. Isto causa inclusive um considerável desperdício de processamento (BANASZEWSKI, 2009).



**Figura 1 - Comparação de vendas entre PCs e Sistemas Embarcados**

Fonte: Patterson e Hannessy (2011)

O Paradigma Orientado a Notificações (PON)<sup>1</sup> é uma alternativa aos atuais paradigmas de programação (SIMÃO;STADZISZ, 2008) (BANASZEWSKI, 2009). Como neste paradigma as relações causais possuem uma conotação reativa, onde cada entidade que tem seu valor alterado tem a função de notificar as alterações às entidades afetadas pela sua mudança, há inclusive um menor desperdício de processamento.

Ademais, a programação em PON se torna mais fácil, por possuir um estilo de programação próximo à forma cognitiva humana, permitindo a representação do conhecimento de maneira natural ao ser humano, na forma de regras, visando alcançar esforços mínimos na programação em termos de escrita de código (BANASZEWSKI, 2009).

<sup>1</sup> Salienta-se que o PON é atualmente objeto de pedido de patente junto ao INPI (Instituto Nacional de Propriedade Industrial) sob Número Provisório 015080004262 e número efetivo PI0805518-1, via Agência de Inovação da UTFPR (SIMÃO; STADZISZ, 2008). Há ademais outros pedidos de patente relativos ao PON (SIMÃO; STADZISZ, 2009b) (SIMÃO et al, 2010). Há ainda um pedido de proteção industrial com subsequente pedido de patente (SIMÃO et al, 2012c). Assim sendo, a utilização do PON se submete ao respeito dos direitos relativos a esta potencial patente.



Por outro lado, por estar o PON materializado ainda sob a forma de um *framework*<sup>2</sup> para a linguagem de programação C++, o PON utiliza elevados recursos de *hardware*, com grande uso de memória de trabalho e processamento, dificultando assim seu uso em sistemas embarcados (LINHARES, 2011)(VALENÇA, 2011).

O desenvolvimento de sistemas totalmente customizados de acordo com a aplicação tem possibilitado desenvolver produtos de alta tecnologia com baixo custo e consumo reduzido de energia. Para isso, é necessário aperfeiçoar a lógica e o processador do sistema embarcado de acordo com a sua função. O consumo de energia também é reduzido devido a esta otimização, pois o processador pode operar em uma menor frequência, já que possui alto desempenho.

Uma das possibilidades para criação de sistemas embarcados, é a implementação de sistemas embarcados completos em lógica programável, contendo processador, periféricos de entrada e saída e *hardware* customizado à aplicação. Com a crescente evolução tecnológica dos dispositivos de lógica programável (e.g. FPGAs), disponibilizando uma grande quantia de unidades lógicas em um único circuito integrado, torna-se possível criar dispositivos altamente complexos em um único *chip*.

Neste âmbito, as implementações com *software* permitem maior flexibilidade que as realizadas em *hardware*, porém as implementações em *hardware* permitem obter maior desempenho. Assim, fica a cargo do projetista optar por uma ou outra, ou ainda um misto das duas, de acordo com os requisitos do sistema.

De fato, uma alternativa para a implementação de sistemas embarcados é o uso da computação reconfigurável (COMPTON; HAUCK, 2002, p.172). A Figura 2 apresenta um gráfico comparativo, relacionando o desempenho e flexibilidade das FPGAs em relação ao ASIC e aos microprocessadores de propósito geral.

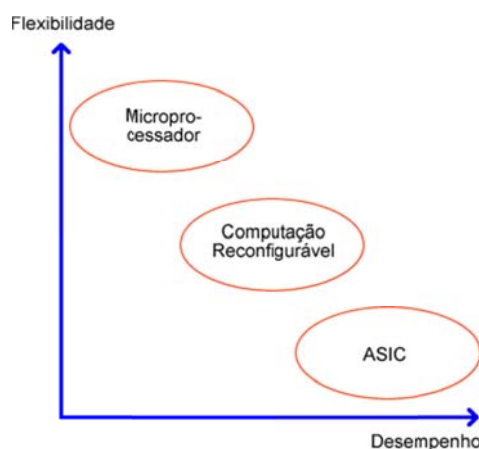
Além disso, existem algumas evidências mostrando que o número de transistores por FPGA está crescendo mais do que em microprocessadores, sendo comparado com a curva de crescimento da tecnologia de memória, indicando que

---

<sup>2</sup> A versão do *framework* utilizada no desenvolvimento deste trabalho foi proposta por (BANASZEWSKI, 2009). Salienta-se que este *framework* está em constante desenvolvimento e já existe uma nova versão com melhorias no desempenho e utilização de recursos.

em um futuro próximo poderão existir FPGAs com desempenho semelhante aos encontrados hoje nos processadores.

O uso da computação reconfigurável possibilita constante melhoria do *hardware* dos sistemas computacionais, uma vez que o *hardware* é descrito através de uma linguagem de descrição de *hardware* (HDL), sem que sejam necessárias alterações físicas de *hardware*, resultando em sistemas eficientes, compactos, de baixo consumo de energia e de custo reduzido.



**Figura 2 - Preenchendo o espaço entre o processador e o ASIC**

Fonte: adaptado de HARTENSTEIN (2001, p. 642)

A computação reconfigurável objetiva preencher a lacuna entre *hardware* e *software*, alcançando um desempenho muito maior que o *software*, e mantendo um grau de flexibilidade muito maior que o *hardware* (COMPTOM; HAUCK, 2000, p.171). Uma das abordagens para o uso da computação reconfigurável pode se dar através do uso de microprocessadores de uso geral, juntamente com *hardware* customizado. Dessa forma pode se obter o benefício de se desenvolver *software* com ferramentas padronizadas e obter um desempenho ótimo de partes específicas do *software*.

Um coprocessador é um conjunto especial de circuitos, desenvolvido para executar tarefas específicas mais rapidamente que um circuito microprocessador básico poderia fazer para executar a mesma tarefa. Um coprocessador é responsável por tarefas de processamento especializado, permitindo que o

processador seja liberado dessas tarefas, enquanto o processamento pesado e específico fica a cargo do coprocessador.

Nesse contexto surge a oportunidade de criar uma solução de sistema embarcado em uma FPGA, contendo um processador customizado com um periférico capaz de acelerar aplicativos desenvolvidos seguindo o Paradigma Orientado a Notificações, tornando-o uma alternativa possivelmente viável para o uso em sistemas embarcados dessa natureza.

## 1.2 OBJETIVOS

Como uma aplicação PON é composta de uma cadeia de pequenas entidades computacionais inteligentes, comunicando-se somente quando necessário, torna-se um bom candidato à implementação em *hardware*. Para investigar este pressuposto, o objetivo principal dessa dissertação é estudar e discutir a construção um *hardware* periférico, capaz de acelerar a execução de *softwares* criados seguindo o Paradigma Orientado a Notificações, tornando dessa forma, viável o uso pleno do PON em sistemas embarcados construídos utilizando-se processadores genéricos.

Serão apresentadas descrições de cada componente da cadeia de notificações e sua implementação correspondente em *hardware*, bem como a integração dos componentes em um sistema, passível de uso em processadores comerciais.

Para isso, foram definidos os objetivos específicos listados a seguir:

- Analisar o PON através da bibliografia, conceitos e aplicações existentes e definir quais componentes podem efetivamente trazer resultados satisfatórios se implementados em *hardware*
- Modelar o sistema de aceleração a partir da implementação existente em *software*
- Criar e testar o *hardware* proposto para aceleração do PON

- Integrar o *hardware* de aceleração do PON a um processador comercial existente
- Realizar as alterações no *framework* atual, para que o mesmo utilize o novo acelerador
- Realizar estudos de caso, a fim de avaliar os ganhos e impactos causados pela adição do novo componente.

### 1.3 ESTRUTURA DA DISSERTAÇÃO

Para apresentar as tecnologias necessárias e demonstrar o desenvolvimento e comparações da implementação de um periférico acelerador do PON, esta dissertação está organizada em cinco capítulos.

O primeiro capítulo introduz o tema desta dissertação ao leitor.

O segundo capítulo traz uma breve revisão da literatura pertinente ao trabalho, apresentando os conceitos necessários ao entendimento dos sistemas embarcados, a tecnologia dos dispositivos lógicos programáveis, utilizada para implementação do periférico para o processador NIOS II e a teoria envolvida na criação do paradigma orientado a notificações.

O terceiro capítulo apresenta o desenvolvimento propriamente dito do periférico acelerador, ou seja, toda a especificação e maneira como estruturou-se periférico.

No quarto capítulo apresenta-se uma comparação de desempenho entre aplicações desenvolvidas utilizando o periférico acelerador do PON e as mesmas aplicações utilizando o PON em sua forma tradicional, ambos executados no processador Altera NIOS II, sintetizado em uma FPGA.

No quinto capítulo são discutidos os resultados e apresentadas as conclusões do trabalho e trabalhos futuros.

## 2 CONCEITOS E FUNDAMENTAÇÃO TEÓRICA

### 2.1 INTRODUÇÃO

Com a crescente evolução tecnológica das FPGAs, disponibilizando uma grande quantidade de unidades lógicas (LU) em um único circuito integrado, torna-se possível implementar dispositivos altamente complexos em um único *chip*. Uma das possibilidades é a implementação de sistemas microprocessados completos em lógica programável.

Outrossim, com a atual complexidade no desenvolvimento dos mesmos, aspira-se por uma maneira mais eficaz e natural de construir *softwares* para este tipo de sistema computacional.

O Paradigma Orientado a Notificações surge como uma alternativa aos paradigmas atuais de desenvolvimento, utilizados para a criação de sistemas embarcados, mas enfrenta problemas quanto ao uso de recursos como memória e processamento em sistemas dessa natureza.

Apresenta-se nesse capítulo a fundamentação teórica necessária ao entendimento do desenvolvimento de um sistema coprocessador, que visa tornar a utilização do PON viável em sistemas embarcados.

### 2.2 DISPOSITIVOS LÓGICOS PROGRAMÁVEIS

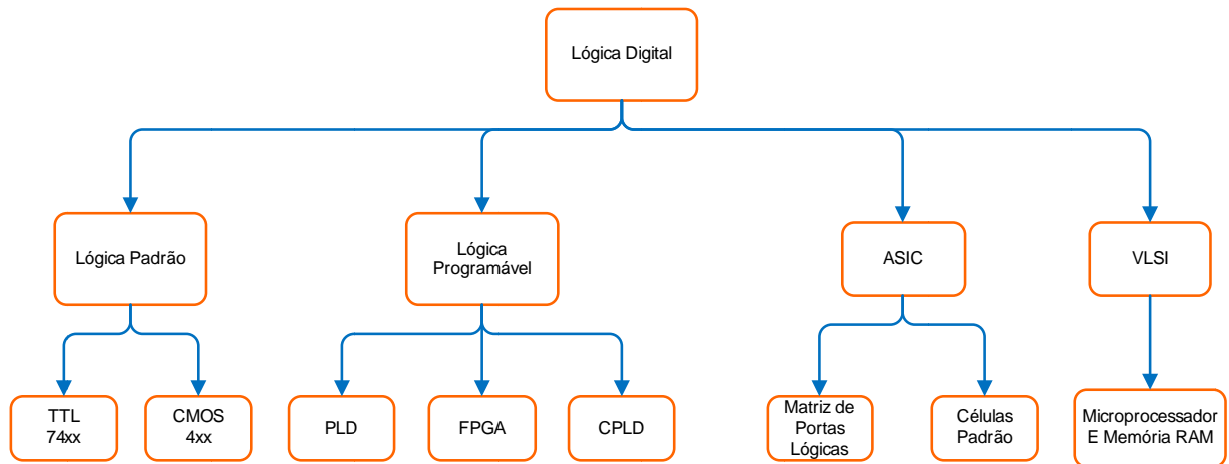
Além dos computadores em si, também a maneira como os sistemas digitais são elaborados tem evoluído drasticamente nas últimas décadas. Até a década de 60, os sistemas digitais eram construídos a partir de componentes discretos, tais como transistores e resistores. Com o surgimento de circuitos integrados (*chips*), foi possível integrar muitos transistores dentro de uma única pastilha de silício, dando subsídios à criação de uma vasta gama de produtos.

Tradicionalmente, os *chips* realizam operações pré-definidas pelo fabricante. Assim, para a elaboração de um circuito com lógica definida pelo usuário, há necessidade do uso de tipos diferentes de *chips*. Isto implica em diversas desvantagens ao sistema, como aumento da área necessária para a conexão, perda de desempenho devido ao atraso existente nas portas de entrada/saída e aumento do consumo de energia.

Para contornar estes problemas, surgiram outros tipos de circuitos integrados, os quais possuem as funcionalidades definidas não mais pelo fabricante, mas sim pelo usuário. Dentre estes, encontram-se os Circuitos Integrados de Aplicação Específica (ASIC), Dispositivos Lógicos Programáveis (PLD), Dispositivos Lógicos Programáveis Complexos (CPLD), *Field-Programmable Gate Array* (FPGA), Matriz Lógica Programável (PLA) etc.

Esta evolução permitiu que uma grande variedade de *chips* baseada em sistemas transistorizados fosse construída nestas pastilhas de silício, as quais são apresentadas na Figura 3.

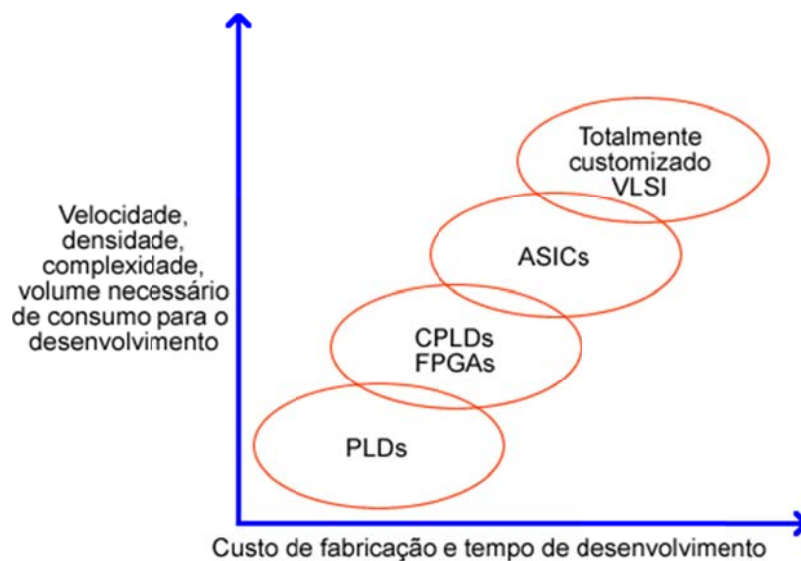
A funcionalidade existente em um chip do tipo ASIC é definida pelo usuário, mas implementada pelo fabricante devido à tecnologia necessária para este processo, a qual possui um custo elevado. Esta funcionalidade é fixa, ou seja, após o *chip* ter sido fabricado não é possível realizar atualizações em suas funções de *hardware*. Apesar desta desvantagem, este tipo de circuito integrado geralmente tem baixo consumo de energia, seu desempenho é alto devido à sua especificidade e o custo do sistema tende a reduzir-se quando produzido em larga escala.



**Figura 3 - Chips disponíveis para implementação de circuitos lógicos**

Fonte: (adaptado de HAMBLEN; FURMAN, 2001, p.38)

Já os PLDs, FPGA e CPLDs permitem ao próprio usuário definir e implementar em campo a funcionalidade do chip. Deste modo, o custo de desenvolvimento e o tempo necessário para a obtenção do produto são inferiores aos de um ASIC. Porém, há um pior desempenho e um aumento de densidade e complexidade do chip (HAMBLEN; FURMAN, 2001, p.39). Através da Figura 4, pode-se observar a relação entre as tecnologias mencionadas anteriormente.



**Figura 4 - Relação entre as diferentes tecnologias de hardware**

Fonte: (adaptado de HAMBLEN; FURMAN, 2001, p.39)

Dispositivos lógicos programáveis (PLD) são dispositivos configuráveis após a fabricação, utilizados para construir circuitos lógicos. Ao contrário de um ASIC, que tem uma função fixa após a fabricação, um PLD tem uma função indefinida quando fabricada e antes de sua utilização este deve ser programado, para que suas funções sejam estabelecidas.

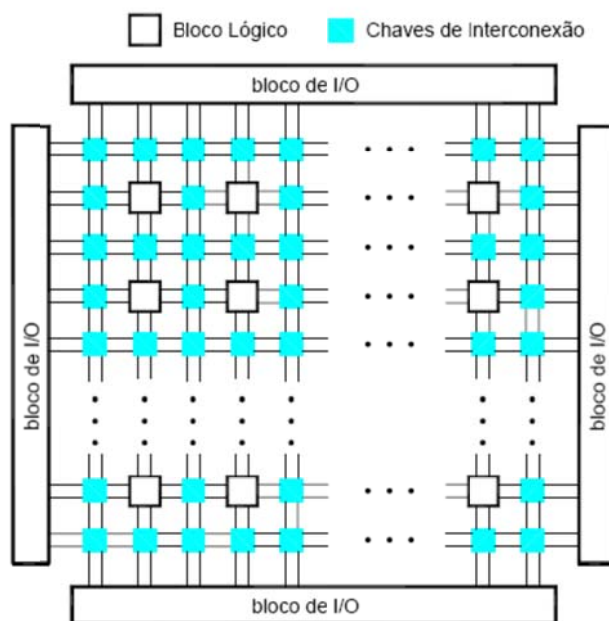
Os PLDs foram introduzidos nos anos 70 e são *chips* de propósito geral que podem ser utilizado para implementar quaisquer tipos de circuitos lógicos (BROWN; VRANESIC, 2005, p.94). Dentro do chip há uma coleção de elementos lógicos fundamentais, que podem ser interligados e configurados de diferentes maneiras.

O PLD pode ser visto como um circuito sem funcionalidade definida, que contém portas lógicas e chaves (*switches*) programáveis. As chaves programáveis permitem conectar as portas lógicas internas ao PLD para implementar quase todos os tipos de circuitos lógicos existentes, definindo assim a sua funcionalidade.

Uma FPGA (*Field Programmable Gate Array*) é um dispositivo que não implementa a função lógica no método tradicional; nela, cada elemento lógico individual é na verdade uma pequena matriz de memória, programada diretamente com a função desejada a partir de uma tabela verdade (BERGER, 2002, p.177).

A estrutura geral de uma FPGA é apresentada na Figura 5, apresentando três tipos de recursos: blocos de I/O para conectar os pinos de entrada e saída do dispositivo, blocos lógicos dispostos num arranjo bidimensional e um conjunto de chaves de interconexão organizadas como canais de roteamento horizontal e vertical entre as linhas e colunas dos blocos lógicos.





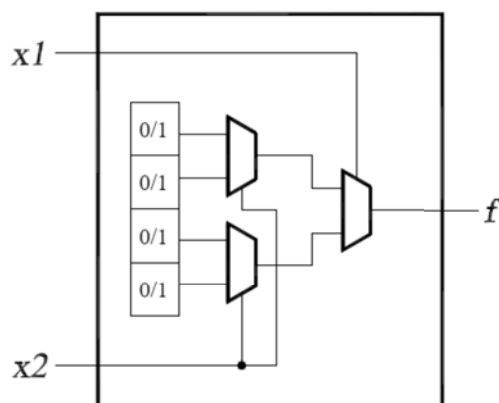
**Figura 5 - Estrutura geral de uma FPGA**

Fonte: (BROWN; VRANESIC, 2005, p.106)

Os canais de conexão entre os blocos lógicos são constituídos por condutores internos e pelas chaves existentes na FPGA. Cada bloco lógico em um FPGA tipicamente possui um pequeno número de entradas e uma saída. Os mais comuns são os baseados em *Lookup Tables* (LUTs), que contêm células de armazenamento usadas para implementar uma pequena função lógica. LUTs de vários tamanhos podem ser criadas, onde o tamanho é definido pelo número de entradas. A Figura 6 apresenta a estrutura de uma pequena LUT com duas entradas ( $x_1$  e  $x_2$ ) e uma saída ( $f$ ), a qual é capaz de implementar qualquer função de duas variáveis. Devido à tabela verdade de duas variáveis possuir quatro combinações possíveis, esta LUT possui quatro células de armazenamento. As variáveis de entrada  $x_1$  e  $x_2$  são usadas como seletores de entrada dos três multiplexadores, que dependendo dos valores de  $x_1$  e  $x_2$  selecionam o conteúdo de uma das quatro células como saída da LUT. As FPGA disponíveis no mercado geralmente possuem LUTs de quatro a oito entradas.

Quando um circuito lógico é implementado em um FPGA, os blocos lógicos são programados para realizarem as funções necessárias e as chaves de interconexão são programadas para definir o canal de comunicação entre estes

blocos lógicos. As células de armazenamento nas LUTs em uma FPGA são voláteis; deste modo, todas as vezes que o dispositivo for energizado deverá ter suas LUTs programadas. Frequentemente, uma PROM (*Programmable Read-Only Memory*) ou uma EPROM (*Erasable Programmable Read-Only Memory*) é incluída no circuito da placa onde se encontra a FPGA, com a finalidade de automaticamente carregar a configuração da FPGA quando o chip é energizado, através de um *hardware* específico para esta função.



**Figura 6 - Estrutura de uma LUT de duas entradas**

Fonte: (BROWN; VRANESIC, 2005, p.107)

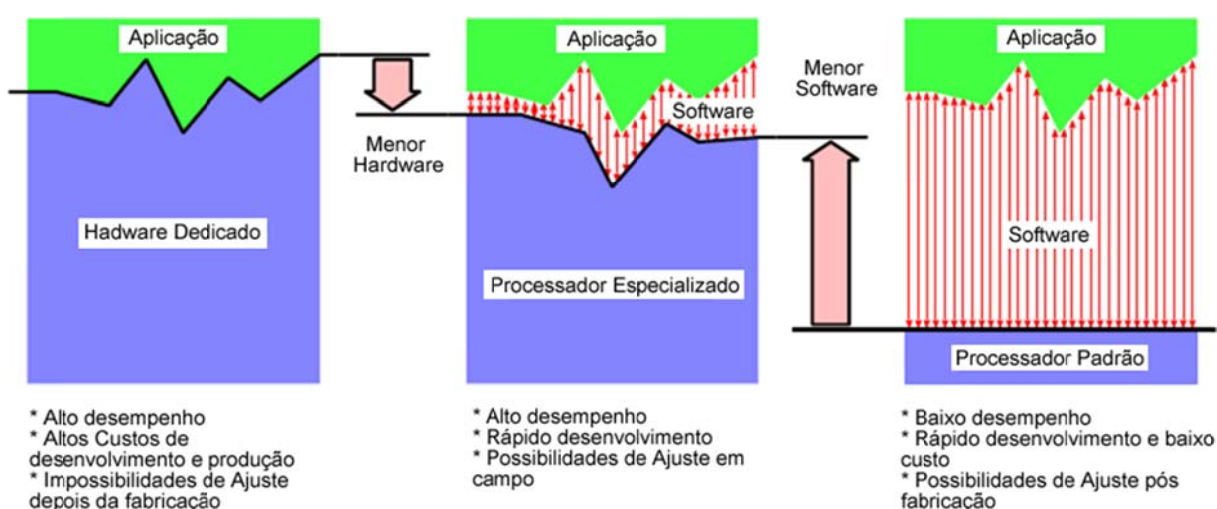
Atualmente existem diversos fabricantes de dispositivos FPGA, dentre os quais podemos citar as empresas Altera, Xilinx, Actel, Cypress Semiconductor, QuickLogic e Lattice/Vantis Corporation. As empresas Altera Corp. e Xilinx Inc. possuem a maior fatia deste mercado, oferecendo uma grande variedade de dispositivos de alto desempenho, alta densidade e relativamente baixo custo.

O uso de *hardware* reconfigurável possibilita que o circuito integrado seja configurado em campo de acordo com a aplicação, além de permitir atualizações do *hardware* já criado. Esta flexibilidade permite obter sistemas de alto desempenho, otimizados de acordo com a aplicação e com tempo de desenvolvimento baixo se comparado aos sistemas discretos e aos implementados em ASIC.

A computação reconfigurável objetiva preencher a lacuna entre *hardware* e *software*, alcançando um desempenho muito maior que o *software* e mantendo um grau de flexibilidade muito maior que o *hardware* (COMPTON; HAUCK, 2000,

p.171). A Figura 7 apresenta a relação entre a aplicação e o dispositivo de *hardware* existente nas três alternativas descritas anteriormente, demonstrando algumas das vantagens e desvantagens de cada tipo de implementação.

Como a computação reconfigurável está baseada nas linguagens de descrição de *hardware* (HDLs) é possível que o *hardware* seja descrito através de arquivos-fonte. Desta forma, modificando somente tais arquivos há a possibilidade de se atualizar todo o *hardware* do sistema. Esta atualização pode ser feita tanto na inicialização do dispositivo como em tempo de execução.



**Figura 7 - Flexibilidade dos dispositivos de *hardware***

Fonte: adaptado de lenne, Pozzi e Vuletic (2002, p.2)

Tecnologias mais recentes de FPGAs permitem uma operação denominada reconfiguração dinâmica ou em tempo de execução. Esta operação é caracterizada pela habilidade de substituição de contextos de *hardware* ou subsistemas digitais para rapidamente modificar (durante a operação) as funcionalidades dos componentes em operação e as suas interconexões (ESKINAZI *et al*, 2005). Esta abordagem permite que um novo *hardware* possa ser criado para suprir uma deficiência de *software* durante a execução do programa.

## 2.3 CPU EM LÓGICA PROGRAMÁVEL

Nos últimos anos, os sistemas embarcados, também conhecidos como computação embarcada vêm se tornando um novo modelo para o projeto e implementação de dispositivos eletroeletrônicos modernos, tais como câmeras filmadoras, telefones celulares, dispositivos de controle automotivos e equipamentos médicos, dentre outros. Estes dispositivos possuem processadores que podem alcançar níveis de desempenho em sua tarefa específica semelhantes ao de um poderoso computador, o que certamente não seria alcançado com o uso de um simples microprocessador de propósito geral.

Com a crescente evolução tecnológica das FPGAs, disponibilizando uma grande quantidade de unidades lógicas (LU) em um único circuito integrado, torna-se possível implementar dispositivos altamente complexos em um único *chip*. Uma das possibilidades é a implementação de sistemas microprocessados completos em lógica programável.

A CPU é o centro de todo sistema computacional, seja ele um sistema de processamento genérico, um sistema embarcado, microcontrolador ou qualquer outro. O trabalho da CPU é executar rigorosamente as instruções de um programa, na sequência definida, para uma aplicação específica. As operações elementares realizadas por uma CPU são: leitura de informações de entrada, leitura da memória de trabalho, transformações sobre os dados nela armazenados e escrita de informações na memória e periféricos. Um programa computacional (*software*) é composto por uma sequência destas operações básicas, e é responsável por garantir que a CPU realize as tarefas desejadas.

As ações de um processador de uma forma geral se resumem às ações buscar instruções na memória, decodificá-las e executá-las<sup>3</sup>, sendo processadas repetidamente:

1. Buscar a instrução apontada pelo contador de programa (PC<sup>4</sup>) da memória e carregá-la no registrador de instrução.

---

<sup>3</sup> Do Inglês *Fetch, Decode e Execute*, geralmente encontrados na literatura.

2. Mudar o PC, apontando-o para a próxima instrução da memória.
3. Decodificar a instrução, determinando o seu tipo, operandos etc.
4. Se a instrução usa operandos (dados) da memória, determinar os seus endereços.
5. Buscar os dados de memória e carregá-los nos registradores.
6. Executar a instrução.
7. Armazenar resultados (em registradores ou memória).

Os programas usam códigos para fornecer instruções para a CPU. Estes códigos são chamados de *códigos de operação*, ou *opcodes*<sup>5</sup>. Cada *opcode* instrui a CPU a executar uma sequência específica de ações.

A implementação de CPUs ou de sistemas microprocessados em dispositivos programáveis, além de apresentar todas as vantagens inerentes à reconfigurabilidade, traz uma série de benefícios em relação ao emprego de CPUs convencionais (JASINSKI, 2004, p.2). Notam-se esses benefícios a partir do aumento de desempenho do sistema com a implementação de rotinas diretamente em *hardware* ao invés da tradicional implementação via *software*.

A CPU Nios II 6.0, por exemplo, permite que a instrução de multiplicação seja realizada através de rotinas de *hardware*, proporcionando um ganho de até 80 ciclos de *clock* se comparado à execução por *software* (Altera, 2012b).

Modelos mais recentes de CPUs, criados especificamente para a implementação em lógica programável, são facilmente adaptáveis a aplicações específicas. Além da possibilidade de se acrescentar novas instruções, outras já existentes podem ser configuradas para executarem mais rapidamente ou utilizarem menos recursos de *hardware* (computação espacial x temporal) (JASINSKI, 2004, p.5).

Hoje em dia podemos encontrar no mercado alguns processadores desenvolvidos especialmente para o uso em FPGA. Tais processadores, em sua maioria, são comercializados pelos próprios fabricantes de dispositivos lógicos

---

<sup>4</sup> Apesar da similaridade com a sigla designada para Computadores Pessoais (PC), optou-se pela denominação para contador de programa, visto que é o termo comumente encontrado na literatura técnica.

<sup>5</sup> Do inglês *Operation Code*, código de operação.

programáveis, sob a forma de núcleos de propriedade intelectual (IPs), desenvolvidos à partir de HDLs. Dentre os mais conhecidos cores voltados ao uso em FPGA, pode-se citar o PowerPC™, MicroBlaze™ e PicoBlaze™, desenvolvidos pela empresa Xilinx (XILINX, 2007), além das plataformas NIOS e Excalibur desenvolvidas pela norte-americana Altera (ALTERA, 2012a).

## 2.4 NIOS II

Uma unidade central de processamento (CPU), comumente chamado de processador, é o principal componente dos computadores digitais. É a parte do computador responsável pela manipulação e transformação dos dados ou informações. Sua função é executar programas armazenados na memória principal, buscando suas instruções, examinando-as e executando-as uma após a outra (TANENBAUM, 1990).

Ao contrário dos processadores comumente utilizados, o processador NIOS II é um soft-core (núcleo de *hardware* implementado em arquivos-fonte HDL), pronto para ser utilizado em uma FPGA. Dessa forma ele pode ser configurado e expandido pela adição de novos componentes de *hardware*.

As principais características deste processador são vistas em Altera (2012b) e podem ser definidas como:

- Arquitetura RISC
- Pipeline opcional de 5 ou 6 estágios
- Arquitetura *load-store*
- Instruções de 32 bits
- Endereçamento de 32 bits
- 32 níveis de interrupção
- 32 registradores de propósito geral
- I/O mapeado em memória
- Unidade de gerenciamento de memória (MMU)
- Unidade de proteção de memória (MPU)

- Módulo de depuração JTAG

O processador NIOS II implementa uma arquitetura *Reduced Instruction Set Computer* (RISC), constituído por um pequeno e otimizado conjunto de instruções, em contraste com as máquinas CISC.

Segundo Heath (1991), em máquinas *Complex Instruction Set Computers* (CISC) 80% das instruções geradas e executadas usam somente 20% do set de instruções. Desta forma, ainda segundo Heath, se as instruções complexas forem sintetizadas em sequências de instruções simples, o *hardware* utilizado para implementá-las também pode ser reduzido em complexidade, sendo possível o desenvolvimento de processadores com maior desempenho, menor quantidade de transistores e conseqüentemente menor custo.

Três doutrinas básicas podem definir a filosofia RISC (HEATH,1991):

- Todas as instruções devem ser executadas em um único ciclo de *clock*;
- A memória deve ser acessada somente via instruções de carga e armazenamento;
- Todas as unidades de execução devem estar fisicamente conectadas, sem nenhum microcódigo.

Outra das importantes características do processador NIOS II é seu *pipeline* de 5 ou 6 estágios. *Pipeline* pode ser definido como uma técnica de implementação de processadores que permite a sobreposição temporal das diversas fases de execução das instruções, aumentando assim o número de instruções executadas simultaneamente e a taxa de instruções iniciadas e terminadas por unidade de tempo. Quando carregada uma nova instrução, ela primeiramente passa pelo primeiro estágio, que trabalha nela durante apenas um ciclo, passando-a adiante, sendo processada sucessivamente pelos demais estágios do processador, enquanto os estágios anteriores já se ocupam com a execução da próxima instrução.

Uma das vantagens da técnica de *pipeline* é que o primeiro estágio não precisa ficar esperando a instrução passar por todos os demais para carregar a próxima, e pode carregar uma nova instrução assim que termina a primeira etapa. O *pipeline* não reduz o tempo gasto para completar cada instrução individualmente,

mas melhora o rendimento, pois várias tarefas são executadas simultaneamente usando recursos computacionais diferentes. Estas instruções são colocadas em uma fila de memória (dentro da CPU) e são executadas sequencialmente.

Os tempos de execução em processadores que utilizam técnicas de *pipeline* podem variar. Operações de ponto flutuante são muito sensíveis aos valores de dados, mas um *pipeline* de execução de inteiros (IEP) também pode introduzir variações dependentes dos dados. Em geral, o tempo de execução de uma instrução no *pipeline* não depende somente da instrução, mas também das instruções ao redor do *pipeline* (WOLF, 2001).

Existem atualmente três versões disponíveis do processador NIOS II, que apesar da diferenciação quanto a consumo de recursos e desempenho, compartilham o mesmo conjunto de instruções e nenhuma alteração no *software* é necessária para utilização em qualquer das três versões:

- Nios II/f - Desenvolvido priorizando o desempenho de execução em detrimento ao uso de *hardware*. Possui um pipeline de 6 estágios, cache de instruções, cache de memória e predição de salto dinâmica.
- Nios II/s - É o core padrão, desenvolvido visando um menor uso de portas lógicas, mantendo uma performance razoável. Possui pipeline de 5 estágios, cache de instruções e predição de salto estática.
- Nios II/e - Otimizado para pequeno uso de recursos de *hardware*. Não contém pipeline e cache.

O processador NIOS II utiliza portas separadas para instrução e dados. A porta de instruções somente executa operações de leitura para o carregamento das instruções. Já a porta de dados executa ambos, leitura e escrita, em memória ou periféricos, dependendo da instrução a ser realizada.

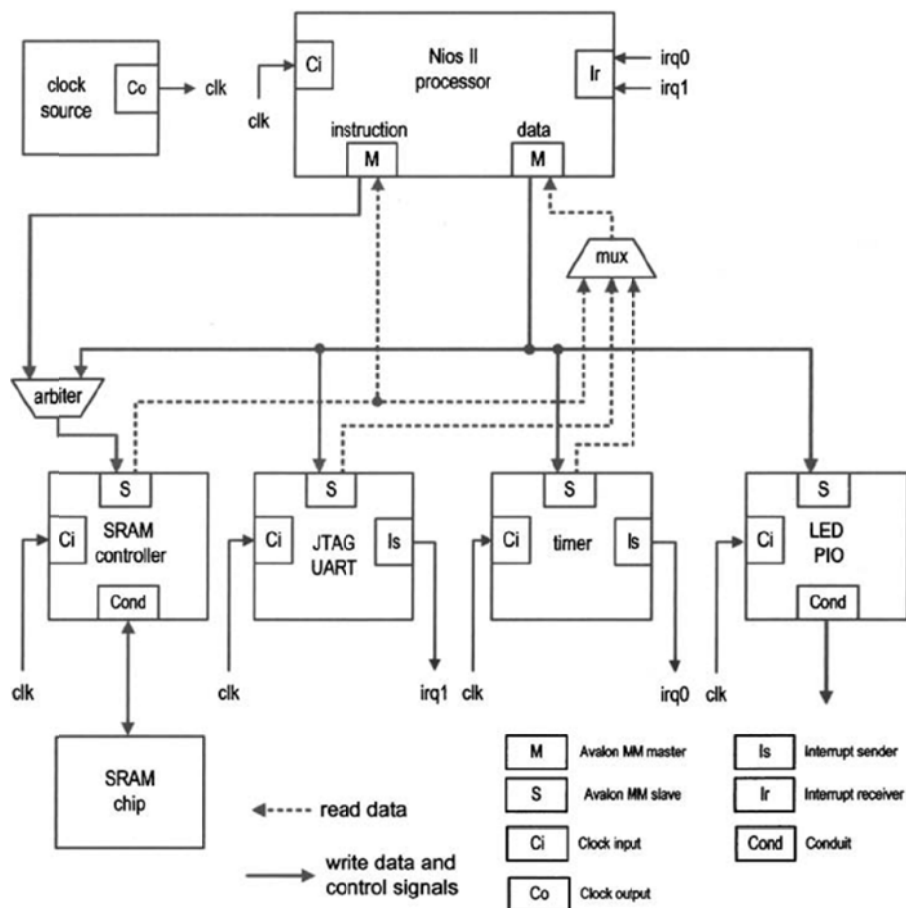
#### 2.4.1 Barramento Avalon

As duas portas, de instruções e de dados podem utilizar módulos de memória separados ou compartilhar os mesmos módulos de memória.



Em um sistema tradicional, o processador, a memória principal, os aceleradores e os dispositivos de I/O são conectados em um barramento compartilhado comum a todos. Como o barramento é um recurso compartilhado, o mesmo se torna um gargalo para a transferência de dados.

O processador Nios II, por sua vez, utiliza uma abordagem diferente. A plataforma *System on a Programmable Chip Builder* (SOPC Builder) da Altera provê a interconexão através de uma coleção de decodificadores, multiplexadores e árbitros, definindo caminhos concorrentes de transferência de dados, conforme mostrado na Figura 8.



**Figura 8 - Implementação conceitual do barramento Avalon**

Fonte: Chu (2011)

A interconexão é automaticamente gerada pelo *software* SOPC Builder no momento da compilação do *hardware* e se beneficia das características de

programabilidade da FPGA, gerando um *hardware* descentralizado e escalável, beneficiando assim o desempenho do sistema.

Para o desenvolvimento de um periférico de I/O ou um *hardware* acelerador, pode-se criar um circuito invólucro ao *hardware* funcional, para que o mesmo possa seguir a especificação do barramento Avalon, e dessa forma utilizá-lo como um componente integrado ao SOPC Builder.

O padrão Avalon consiste nos seguintes tipos de interface:

- Interface mapeada em memória (Avalon MM): Esta interface define uma conexão mestre-escravo baseada em endereço, onde um componente mestre utiliza um endereço para ler e escrever em um componente escravo.
- Interface de streaming Avalon (Avalon-ST): Define um link de comunicação unidirecional entre dois componentes, onde um componente Avalon-ST fonte transmite dados continuamente para outro Avalon-ST consumir.
- Interface escrava mapeada em memória *tristate*: É um caso especial de Avalon MM, utilizada para conexões a componentes *tristate* externos à FPGA, como memórias SRAM e Flash.
- Avalon *clock*: Define uma interface para sinais de *clock* e reset usados por um componente.
- Avalon interrupt: Define uma interface para que componentes escravos possam enviar eventos de interrupção para um componente mestre.
- Avalon conduit: Interface criada para agrupar e exportar dados para fora de um sistema SOPC Builder.

Como um periférico acelerador necessita comunicação baseada em endereços de memória, sempre sendo iniciada pelo dispositivo mestre, neste caso o processador NIOS II, a interface Avalon MM se mostra adequada a esta função.

### 2.4.2 Interface Avalon MM

Esta interface basicamente define uma coleção de sinais e propriedades para padronizar a comunicação entre os componentes. Geralmente o processador NIOS II atua como único componente mestre e a maioria dos periféricos e aceleradores são componentes escravos. Como mestre ele pode iniciar a comunicação e solicitar a leitura ou escrita de dados em componentes escravos que, por sua vez, respondem a solicitação do mestre.

A interface de comunicação é composta por 22 sinais, dentre os quais os mais relevantes para o entendimento do periférico são:

- `read (read_n)`: sinal composto por um bit e usado para leitura de dados pelo mestre em um componente escravo.
- `write (write_n)`: sinal composto por um bit e usado para escrita de dados pelo mestre em um componente escravo.
- `address`: sinal com largura variável (1 a 32 bits) utilizado como *offset* de memória em um componente escravo. Cada valor identifica uma posição de memória interna ao componente.
- `readdata`: sinal com largura variável (8 a 1024 bits) onde os dados são entregues pelo componente escravo em uma operação de leitura.
- `writedata`: sinal com largura variável (8 a 1024 bits) onde os dados são entregues pelo componente mestre em uma operação de escrita.
- `chipsselect`: sinal composto por um bit e utilizado pelo dispositivo mestre para selecionar um componente escravo para uma operação.

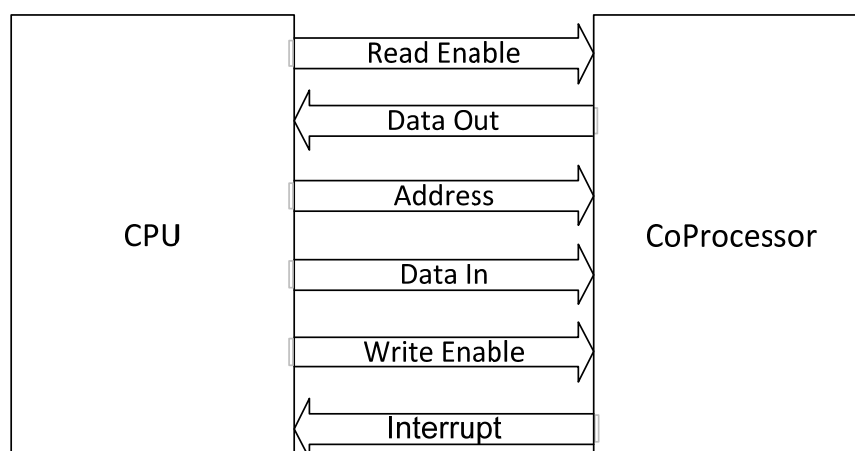
## 2.5 COPROCESSADORES

Um coprocessador pode ser definido como qualquer *hardware* que pode auxiliar um processador a desenvolver uma tarefa, ou um conjunto de tarefas de forma otimizada, trazendo dessa forma um benefício em desempenho ou economia de recursos. Ao assumir tarefas que consomem tempo de processamento, os coprocessadores podem acelerar a performance geral de um sistema (PATTERSON;HANNESY, 2011).

Geralmente um coprocessador é um elemento passivo no sistema, recebendo dados de uma fonte, geralmente o próprio processador ou de acesso direto à sua memória e respondendo ao processador através de uma interrupção.

A Figura 9 apresenta a estrutura simplificada de um coprocessador mapeado em memória. Nesta figura pode-se observar os sinais utilizados para a comunicação com o mesmo. De forma genérica, o processador coloca um dado e um endereço no barramento e indica ao coprocessador com um sinal de *write\_enable*.

O coprocessador executa o tratamento dos dados recebidos e quando possui algum dado pronto indica ao processador com um sinal de interrupção. Quando o processador estiver disponível, informa ao barramento o endereço de leitura do coprocessador e indica a leitura através de um sinal e *read\_enable*.



**Figura 9 – Modelo simplificado de um coprocessador**

Fonte: própria

## 2.6 PARADIGMAS DE PROGRAMAÇÃO

Em Ciência da Computação, o termo paradigma é empregado como a maneira de compreender um problema do mundo real, para transformar este problema em uma solução computacional (BANASZEWSKI, 2009). Assim, um paradigma de programação determina a visão que o programador possui sobre a estruturação e execução do programa.

Neste âmbito, as linguagens de programação evoluem através do tempo. Linguagens já existentes evoluem agregando novas características e novas linguagens surgem apresentando características completamente diferentes das habituais de acordo com os conceitos de um novo paradigma de programação (BERGIN e GIBSON, 1996) (BANASZEWSKI, 2009).

Em meados de 1940, no início do advento dos computadores eletrônicos, a forma de programação era bastante precária, tornando-se uma atividade tediosa e bastante árdua. Os programadores precisavam compreender os detalhes da arquitetura do computador e adaptar a forma de pensar nos problemas computacionais de acordo com o processo de execução das instruções pelos computadores (BANASZEWSKI, 2009).

De forma geral, nessa época os programas eram escritos diretamente em linguagem de máquina, deixando todo o entendimento de *hardware* e fluxo de execução do *software* por conta do programador.

Desde então as linguagens evoluíram muito, passando pelo surgimento da linguagem Assembly, em meados da década de 50, apresentando-se como forma de suprir a insatisfação dos programadores com a linguagem de máquina, chegando às linguagens de alto-nível, que oferecem comandos de manipulação mais próximos à linguagem humana, de forma que não seja necessário um conhecimento aprofundado da arquitetura dos computadores para criação de *software* (BANASZEWSKI, 2009).

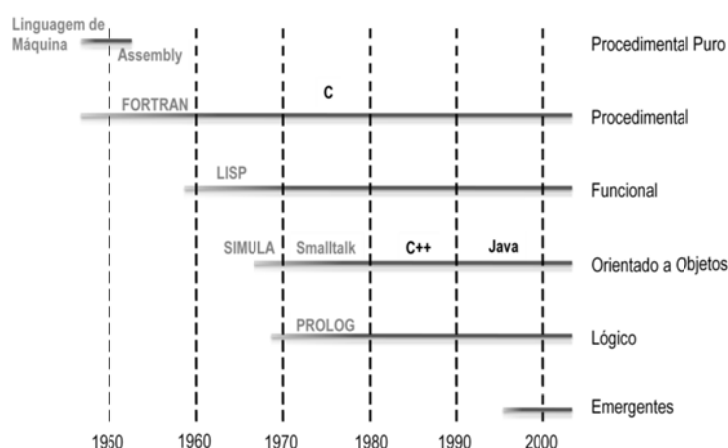
A partir dessa ideia de linguagem de alto-nível, diversas linguagens surgiram, como COBOL, ALGOL e algum tempo depois - durante a década de 1960

- as linguagens estruturadas (C, PASCAL, BASIC). Estas linguagens tentavam suprir o problema de legibilidade e complexidade de código criado pelo comando "goto", usada nas linguagens não estruturadas para criar laços de repetição e desviar o fluxo do programa, o que os tornava de difícil entendimento e conseqüentemente manutenção complexa.

Em 1960, surge a linguagem LISP (abreviação de LISt Processing – Processamento de Listas), proposta como uma solução inovadora na programação, a qual permitiu melhoras significativas na abstração em relação às linguagens convencionais (e.g. recursividades, laços condicionais, *garbage collector*) (BANASZEWSKI, 2009).

Durante a década de 1970, surgem evoluções nas linguagens imperativas e declarativas, juntamente com os sistemas baseados em regras, permitindo a representação do mundo real por meio de objetos computacionais, de forma semelhante à maneira como um humano percebe o mundo, oferecendo uma maior proximidade da forma de estruturar os problemas computacionais à cognição humana (NEWEL; SIMON, 1972) (BANASZEWSKI, 2009).

Com esse pressuposto, diversas linguagens imperativas surgiram, como SIMULA, Smalltalk, Object Pascal, C++ e Java, que ainda são largamente utilizadas nos dias de hoje. Em paralelo à melhoria das linguagens imperativas, surge uma nova linguagem declarativa, a linguagem PROLOG (PROgrammation in LOGic) permitindo desenvolver soluções computacionais da mesma maneira que se organiza o conhecimento para raciocinar. Esta linguagem é baseada em texto escrito em linguagem supostamente “natural”, onde as relações lógicas são representadas por meio de Regras. Na Figura 10, pode-se observar a evolução de destas linguagens, relacionadas com seus respectivos paradigmas.

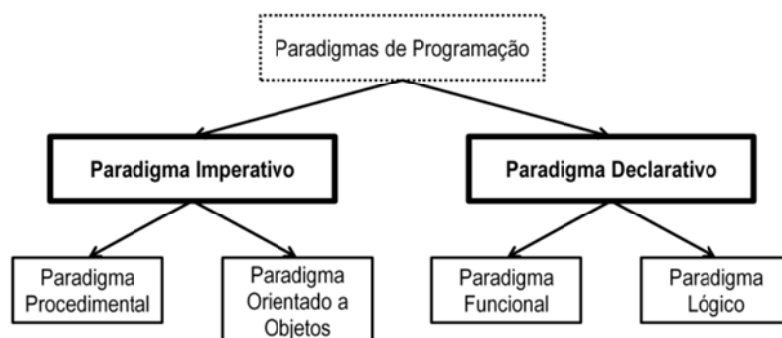


**Figura 10 - Evolução das linguagens em relação aos paradigmas de programação**

Fonte: Banaszewsk (2009)

A programação declarativa demandaria menos esforços de programação do que a programação imperativa. Por outro lado, perde-se flexibilidade na programação, principalmente para otimizar o código em termos de acesso ao *hardware* e desempenho. Isto porque as linguagens declarativas demandam buscas propriamente ditas, para correlacionar as expressões causais e os estados de entidades computacionais (e.g. variáveis ou atributos de objetos), avaliando por meio de algoritmos e estruturas de dados, que são caras computacionalmente (BANASZEWSKI, 2009).

Isto dito, o conjunto de paradigmas de programação estabelecidos é chamado de paradigmas dominantes. Entre os Paradigmas dominantes estão o Paradigma Procedimental (PP), o Paradigma Funcional (PF), o Paradigma Orientado a Objetos (POO) e o Paradigma Lógico (PL) (BANASZEWSKI, 2009). Na Figura 11 pode-se ver uma dada interpretação de relação entre os paradigmas existentes atualmente.



**Figura 11 - Classificação dos atuais paradigmas**

Fonte: (adaptado de BANASZEWSKI, 2009)

O Paradigma Imperativo consiste na organização sequencial do *software* por meio de comandos em uma linguagem procedimental ou orientada a objetos, os quais são executados sequencialmente pelo mecanismo interno destas linguagens. Basicamente, este mecanismo consiste em "buscas" sobre variáveis, vetores e listas e aos comandos de decisão e fluxo de execução.

Devido à sequencialidade da busca e a passividade dos buscados nas linguagens do PI, as linhas de código se tornam interdependentes e há problemas de redundância na execução dos programas (BANASZEWSKI, 2009). Estas redundâncias em sistemas compostos por muitas expressões de decisão possuem efeitos no desempenho da aplicação, podendo chegar a níveis inaceitáveis (SIMÃO; STADZISZ, 2008).

Comumente, nos programas criados com os conceitos do Paradigma Imperativo, o código referente à solução propriamente dita encontra-se disperso entre os comandos e expressões da própria linguagem, tornando o entendimento do código uma tarefa árdua, além de desviar a atenção do programador para a linguagem e não para o problema a ser resolvido.

O Paradigma Declarativo, por sua vez, é caracterizado pela interação mais simplificada do programador com o computador por meio de mecanismos que escondem as particularidades de implementação. No Paradigma Declarativo, o programador se concentra mais na organização do conhecimento sobre a resolução



do problema computacional do que na implementação do mesmo (BANASZEWSKI, 2009).

Porém, para oferecer estas facilidades o Paradigma Declarativo perde em velocidade de execução para o Paradigma Imperativo e em certas flexibilidades, especialmente no acesso direto ao *hardware* (SCOTT, 2000). Dessa forma, uso destas soluções pode limitar a criatividade do programador e também o acesso aos componentes de *hardware* ou ainda impossibilitar a construção de códigos mais eficientes, uma vez que os comandos declarativos são implementados para uso genérico (BANASZEWSKI, 2009).

Outrossim, todos estes paradigmas levam ao forte acoplamento de expressões causais e redundâncias decorrentes das suas avaliações. Estas limitações dificultam a execução paralela ou distribuída de programas e frequentemente comprometem o seu desempenho pleno mesmo em sistemas monoprocessados. Assim, existem motivações para buscas de alternativas à tais paradigmas com objetivo de suprir as desvantagens apresentadas (BANASZEWSKI, 2009).

Com este foco, foi concebido o Paradigma Orientado a Notificações (PON), introduzido embrionariamente em (SIMÃO, 2001) e (SIMÃO, 2005) e detalhado em (SIMÃO;STADZISZ, 2008) (SIMÃO;STADZISZ, 2008) (BANASZEWSKI, 2009), propondo soluções efetivas para as deficiências apresentadas e se apresentando como um paradigma com grande potencial de aplicabilidade (SIMÃO et al, 2012d).

## 2.7 PARADIGMA ORIENTADO A NOTIFICAÇÕES

O Paradigma Orientado a Notificações (PON) é uma alternativa aos atuais paradigmas de programação (eg. imperativo, declarativo e orientado a objetos), se inspirando-se e evoluindo a partir dos mesmos, com ênfase nos conceitos dos Sistemas Baseados em Regras (SBR), os quais oferecem (até então) modelos de programação com maior proximidade à cognição humana (NEWEL; SIMON, 1972).

PON visa eliminar algumas das principais deficiências dos atuais paradigmas, como a existência de avaliações causais desnecessárias e fortemente acopladas.

Atualmente, os conceitos do PON estão materializados como um *framework* para uma linguagem de programação imperativa, a linguagem C++, por meio da qual os elementos do PON (eg. Atributos, Premissas, Condições, Regras etc) são representados como objetos C++. Entretanto, por ser inspirado em conceitos do paradigma declarativo, o *framework* do PON trás maiores facilidades de programação à linguagem C++, as quais são similares àquelas ofertadas na programação declarativa.

A programação em PON, por seguir os conceitos dos Sistemas Baseados em Regras, se torna mais fácil, se aproximando à forma cognitiva humana, permitindo a representação do conhecimento de forma natural ao ser humano, o PON também visa alcançar esforços mínimos na programação em termos de escrita de código (BANASZEWSKI, 2009).

Essencialmente, a programação em PON consiste em conceber os relacionamentos entre elementos da base de fatos (FBEs) por meio de Regras (Rules), onde o programador somente precisa compor as Regras sem necessariamente se preocupar em como elas vão interagir para que o fluxo de execução seja estabelecido, principalmente se uma ferramenta *wizard* for utilizada (BANASZEWSKI, 2009).

Uma aplicação em PON é formada a partir da combinação de diversos objetos desacoplados que interagem entre si para implementar as funcionalidades desejadas. Os principais componentes do PON são: elementos da base de fatos (*Fact Base Elements*, ou FBEs), Atributos (*Attributes*), Premissas (*Premises*), condições (*Conditions*), Regras (*Rules*), ações (*Actions*), instigações (*Instigations*) e métodos (*Methods*) (BANASZEWSKI, 2009).

Na Figura 12 pode-se observar a cadeia de notificações criada, onde um objeto do tipo Atributo se comunica com objetos do tipo Premissa, dando início à cadeia de notificações. As Premissas se comunicam com as Condições, notificando-as quando necessário. As condições, por sua vez, quando verdadeiras ativam as Regras, que dão início às Ações, Instigações e Métodos, conforme necessário.

Todos os objetos somente são executados, quando um objeto correlacionado o notifica, minimizando o ciclo de execução e evitando processamento desnecessário.

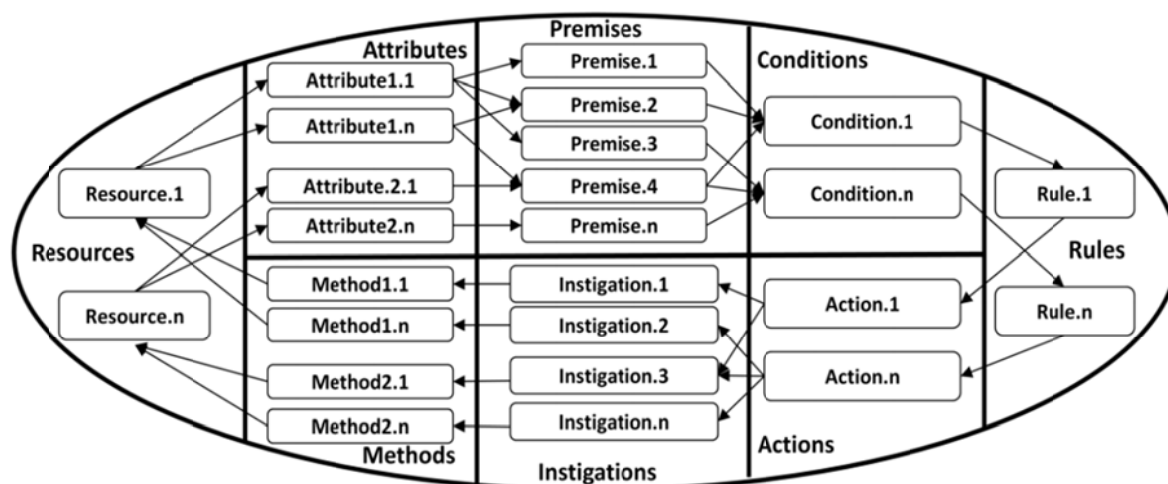


Figura 12. Exemplo da cadeia de notificações formada pelos componentes do PON

Fonte: Banaszewski (2009).

### 2.7.1 Elementos da Base de Fatos

Em PON, um Elemento da Base de Fatos descreve estados e serviços de entidades (reais ou computacionais) de um dado problema. Essencialmente, cada FBE representa os estados, serviços e entidades por meio de Atributos e Métodos (SIMÃO et al, 2012d).

### 2.7.2 Atributos

Os Atributos do PON diferem das demais linguagens por ter uma conotação ativa. Além de armazenar um valor, como nas demais linguagens, ele também é responsável por notificar as estruturas pertinentes (Premissas) sobre alterações em seu valor, dando início a cadeia de notificações do PON. Ainda pode-se dizer, em termos gerais, que o conjunto de Atributos de um sistema compõe o seu estado atual.

### 2.7.3 Premissas

Uma Premissa em PON consiste em uma operação lógica sobre o valor de um Atributo e um valor constante ou valor de outra premissa. Esta operação lógica é realizada no momento em que um Atributo notifica à Premissa sobre a alteração em seu valor armazenado. Quanto a operação lógica realizada, uma Premissa pode ser configurada para as operações lógicas igual a, diferente de, maior que, maior ou igual a, menor que e menor ou igual a (SIMÃO et al, 2012d).

No momento de sua alteração, as Premissas precisam notificar suas estruturas subsequentes, neste caso as condições, repassando seu estado lógico e dando continuidade à cadeia de notificação do PON (SIMÃO et al, 2012d).

### 2.7.4 Condições

Uma Condição representa uma expressão lógico-causal sobre os valores de saída das Premissas. Cada entidade Condição notificada reavalia o seu estado lógico de acordo com o valor recém-notificado pela Premissa em questão juntamente com os valores notificados previamente pelas demais Premissas conectadas. Assim, quando todas as Premissas que compõem uma Condição apresentam o estado lógico verdadeiro, a entidade Condição é satisfeita e sua Regra relacionada é aprovada. A operação realizada pode ser uma conjunção ou uma disjunção lógica sobre as saídas das Premissas, representando um conjunto de Premissas ativas ou não ativas (SIMÃO et al, 2012d).

Assim como os Atributos e as Premissas, as condições também precisam repassar às estruturas subsequentes, neste caso as Regras, o valor de saída da operação realizada, além de uma notificação quando este valor for alterado (SIMÃO et al, 2012d).

### 2.7.5 Regras

Um objeto Regra é a composição de dois objetos, uma Condição e uma Ação, tendo por finalidade descrever um aspecto específico da lógica de negócios da aplicação implementada. Em termos de componentes do PON, uma Regra simplesmente conecta uma Condição a uma ação, que será executada quando a Condição for verdadeira (SIMÃO et al, 2012d).

Ao invés de serem executadas imediatamente após suas condições se tornarem verdadeiras, as Regras são apenas aprovadas para execução, e sua execução é controlada por um objeto *Scheduler*, o qual pode considerar outros fatores, como prioridades relativas entre as Regras ou estratégias de resolução de conflitos (SIMÃO et al, 2012d) (BANASZEWSKI, 2009).

### 2.7.6 Ações

Uma ação corresponde a um conjunto de tarefas que serão executadas quando uma Regra for aprovada para execução e um parâmetro configurável, indicando se estas tarefas deverão ser executadas simultaneamente ou sequencialmente. Estas tarefas estão encapsuladas em outro componente do PON, a instigação (SIMÃO et al, 2012d).

### 2.7.7 Instigações

Uma instigação no PON possui duas responsabilidades, podendo tanto redefinir o estado de um Atributo, reiniciando a cadeia de notificações, quanto iniciar a execução de um método (SIMÃO et al, 2012d).

### 2.7.8 Métodos

Um método na atual implementação do PON equivale a um método (função) da linguagem C++. Esta funcionalidade possibilita a integração de uma aplicação em PON com código imperativo tradicional.

Desta forma, o programador possui liberdade para realizar praticamente qualquer tipo de operação dentro de um método, independentemente de sua complexidade ou duração. Na implementação atual em C++, um método não possui valor de retorno; porém, isto é contornado devido à facilidade de interagir com os componentes do PON dentro do próprio método (por exemplo, redefinindo o valor de um Atributo ao término de sua execução).

## 2.8 REFLEXÃO

Em suma, os conceitos do PON surgem para tornar a programação mais simples. Além de permitir a estruturação do conhecimento em forma natural ao ser humano. Proporciona a composição deste conhecimento de forma simplificada e transparente das particularidades do código imperativo. No mais, quando estas particularidades se fazem necessárias, o PON possibilita o seu uso, porém o faz de forma a evitar a ocorrência de certos problemas natos a este tipo de programação, como o acoplamento, dispersão do conhecimento e execuções desnecessárias (BANASZEWSKI, 2009).

Por ser composta de uma cadeia de pequenas entidades computacionais desacopladas, comunicando-se assincronamente, somente quando há necessidade, o PON torna-se é um bom candidato a implementação direta em *hardware*. No capítulo seguinte, tal implementação é descrita.

### 3 DESENVOLVIMENTO

#### 3.1 INTRODUÇÃO

Para eliminar os gargalos associados à implementação puramente em *software* do PON, implementou-se um *hardware* coprocessador (CoPON) capaz de executar toda a cadeia de inferências (Atributos, Premissas, condições e partes das Regras).

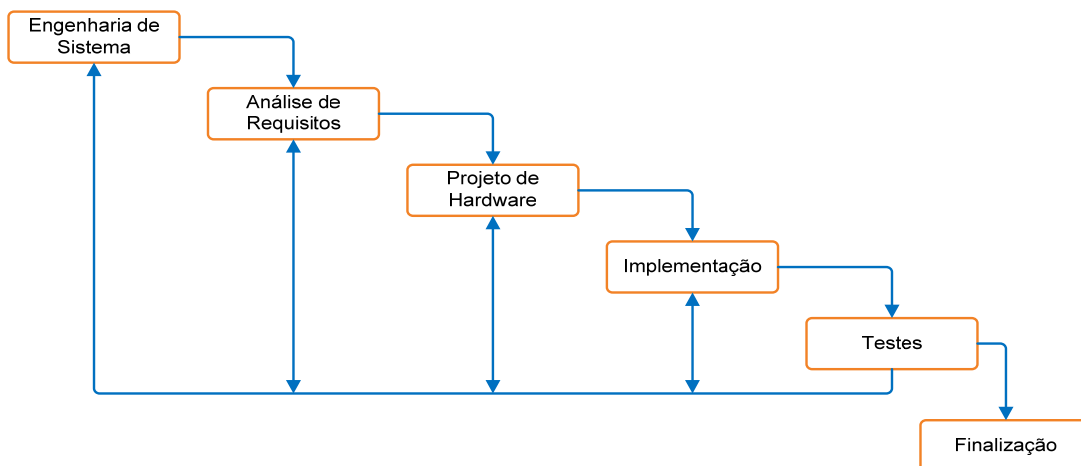
A estrutura fundamental do periférico PON se baseia em converter os componentes não dinâmicos do paradigma em componentes de *hardware*. Com isso, têm-se a intenção de substituir todo o processamento da cadeia de notificações e das operações lógicas por um *hardware* equivalente, liberando o processador dessa tarefa. A cadeia de notificações e as expressões lógicas possuem um alto custo computacional quando executadas puramente em *software*, já que necessitam estruturas complexas para que possam ser executadas sobre uma linguagem de programação genérica. Dessa forma substituí-las por um *hardware* capaz de desempenhar a mesma função poderá ocasionar um grande ganho na eficiência geral do sistema.

Este capítulo apresenta uma série de metodologias e ferramentas utilizadas no desenvolvimento do Coprocessador para Paradigma Orientado a Notificações (CoPON), bem como o funcionamento do *hardware* propriamente dito.

Optou-se pela criação de um *hardware* totalmente configurável quanto ao número de componentes PON de cada tipo presentes no sistema final. No momento da criação do sistema no aplicativo Altera SOPC Builder, configura-se o número de Atributos, Premissas, condições e Regras que o sistema suportará. Os valores máximos somente são limitados pela disponibilidade física de unidades lógicas e memória na FPGA.

O desenvolvimento da arquitetura do sistema envolveu diversas etapas complementares entre si, numa sequência de atividades similares às aplicadas em

projetos de engenharia. Utilizou-se para este caso o Modelo Clássico de Desenvolvimento ou Modelo em Cascata, o qual é demonstrado através da Figura 13 abaixo.



**Figura 13 - Modelo Clássico de Desenvolvimento**

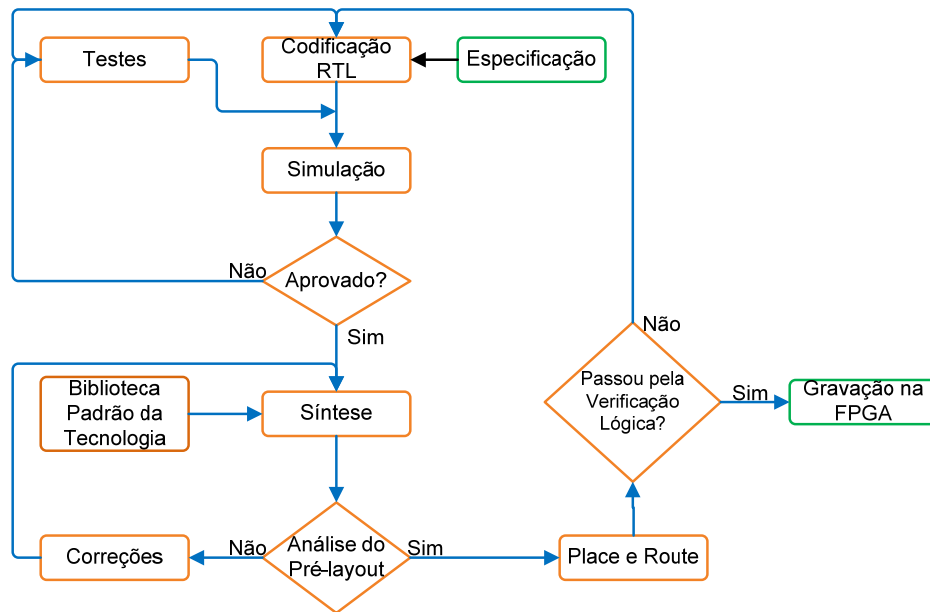
Fonte: Própria

Este modelo propõe uma abordagem sistemática e sequencial ao desenvolvimento do sistema, que começa pela definição de requisitos do sistema e passa pela análise, projeto, codificação e teste. Durante qualquer das etapas, pode ser necessário retornar para etapas anteriores e reiniciar o ciclo.

### 3.2 DESENVOLVIMENTO DO HARDWARE EM HDL

O modelo de desenvolvimento utilizado para sistemas baseados em linguagens de descrição de *hardware* pode ser sumarizado através da Figura 14. Nele pode-se observar as diversas etapas envolvidas na criação do sistema, desde a sua especificação até a finalização do projeto.





**Figura 14 - Fluxo de desenvolvimento de sistemas baseados em HDLs**

Fonte: adaptado de Lee (2003)

As etapas mais importantes do desenvolvimento dos sistemas em lógica programável e apresentadas na figura anterior são descritos abaixo.

### 3.2.1 Especificação e Desenho

Neste estágio, são definidos os parâmetros mais importantes do sistema que se está planejando. Nele são definidas de forma genérica todas as funções que o sistema suportará, suas interfaces externas e suas respostas externas.

A partir dos parâmetros definidos, são definidos vários blocos com a lógica do sistema e como estes se comunicarão.

### 3.2.2 Micro Desenho

Nesta fase é descreve-se o funcionamento de cada um dos blocos. O micro desenho contém detalhes sobre máquinas de estado, contadores, registradores

internos e possíveis formas de onda de saída. Esta é uma das fases que demanda mais tempo no desenvolvimento.

### 3.2.3 Codificação em Nível de Transferência de Registro – RTL

Nesta etapa, o micro desenho é convertido em arquivos HDL com a síntese da linguagem utilizada, seja ela Verilog, VHDL ou outra qualquer. Todo o trabalho de geração dos arquivos-fonte é executado nesta etapa, convertendo todo o *hardware* planejado em arquivos de texto com a descrição do sistema.

Para o desenvolvimento desse sistema, utilizou-se a linguagem VHDL e o ambiente integrado de desenvolvimento (IDE) para *Hardware*, Sigasi 2.0, mostrado na Figura 15.

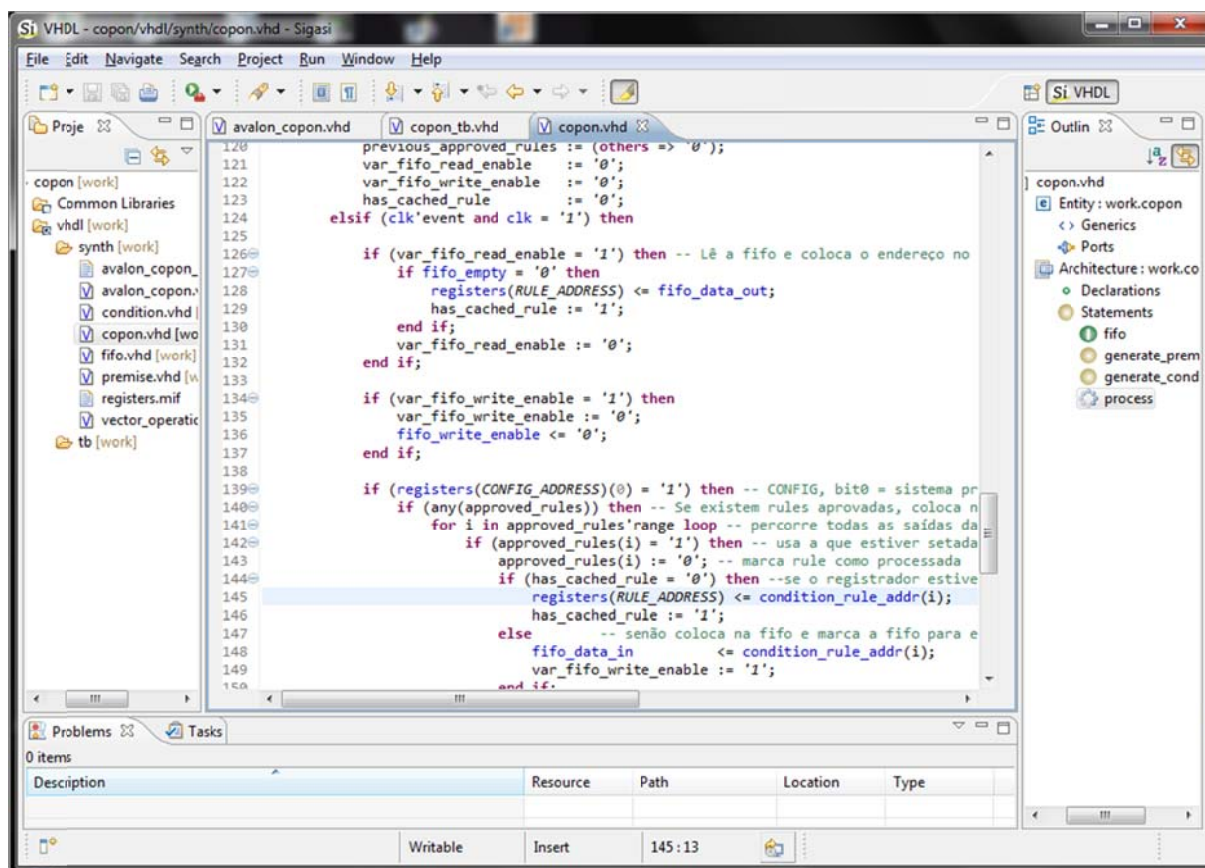


Figura 15 - Ambiente integrado de desenvolvimento para VHDL

Fonte: Própria

Esta IDE provê diversos recursos úteis para a codificação, como análise de erros de código em tempo real e ferramentas para refatoração de código.

### 3.2.4 Simulação e Testes

Simulação é o processo de verificação das características funcionais e temporais dos modelos criados. Para tanto usualmente cria-se arquivos de testes (*testbenches*), com os sinais de *clock*, *reset* e demais dados das entradas do bloco a ser testado, de forma que se possa analisar o funcionamento do bloco através das saídas do mesmo. Segundo Lee (2003), de 60% a 70% do tempo despendido com o desenvolvimento ocorre na verificação do sistema.

Cada componente do PON utilizado para a criação do periférico foi simulado e testado individualmente. Além dos testes individuais, também fez-se necessário a criação de um teste de integração do sistema, onde todos os sinais de entrada do periférico, que na prática serão gerados pelo NIOS II, foram simulados, garantindo a comunicação e integração de todos os componentes como um sistema.

Para a realização dos testes, utilizou-se o simulador de VHDL ModelSim Altera Starter Edition, versão 10.0c, e scripts auxiliares desenvolvidos em linguagem Ruby, para automatização de tais testes.

### 3.2.5 Síntese, Alocação e Roteamento

Síntese é o processo em que o compilador transforma o sistema descrito através de arquivos HDL na tecnologia do dispositivo de lógica programável a ser utilizado. Neste processo, são geradas as listas de ligações com os componentes que serão utilizados.

Alocação e roteamento é a etapa posterior à síntese, onde todas as portas lógicas e *flip-flops*<sup>6</sup> são distribuídas. A distribuição dos sinais de relógio e *reset* são feitas e depois disso, cada bloco é roteado dentro da FPGA.

Cada fabricante de FPGAs possui seu sistema proprietário para síntese e, nesse caso, por se tratar de um periférico para o processador NIOS II da Altera, utilizou-se a ferramenta Quartus II 11.1 para esta tarefa.

### 3.3 DESENVOLVIMENTO DE SOFTWARE

Para o desenvolvimento do *software* para o processador NIOS II, bem como todo o SoC gerado, utilizou-se as ferramentas fornecidas pela fabricante da FPGA. A suíte de ferramentas utilizada, *Nios II Software Build Tools (SBT)* é composta por compilador, *linker* e *debugger*, disponíveis na IDE Eclipse, com o *plugin Nios II SBT for Eclipse™*, conforme apresentado na Figura 16.

As ferramentas presentes nesta suíte, além de permitirem o desenvolvimento do código da aplicação propriamente dito, criam todas as interfaces de *hardware* necessárias ao desenvolvimento, através da ferramenta *Board Support Package (BSP)*. Esta ferramenta utiliza como entrada o arquivo “.sopcinfo”, gerado no momento da criação do sistema, criando através dele arquivos de cabeçalho contendo o mapa de memória do sistema e funções de acesso ao *hardware*.

As linguagens disponíveis para criação de *software* para esta plataforma são Assembly e C/C++, sendo que a utilizada para este projeto foi a C++, mantendo assim a compatibilidade com o *framework* já desenvolvido para o PON.

---

<sup>6</sup> Os flip-flops são elementos de circuito que podem apresentar em seu funcionamento apenas dois estados estáveis. Não existem estados intermediários entre estes dois estados.

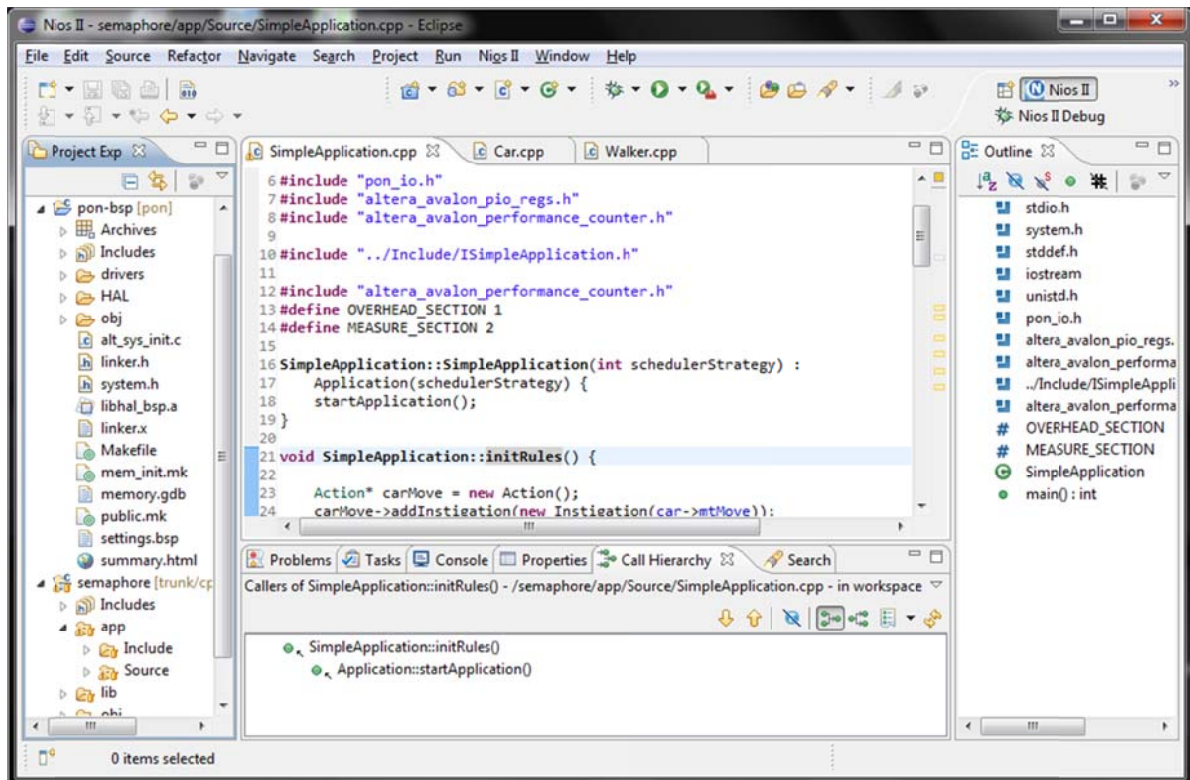


Figura 16 - Ambiente integrado de desenvolvimento SBT

Fonte: Própria

Para validação do sistema, criaram-se dois conjuntos de *softwares*, um orientado totalmente a teste funcional do sistema e outro que visa validar o sistema comparando-o com implementações anteriores do PON.

O *software* de teste funcional tem por finalidade estressar todas as funções existentes no *hardware*, utilizando a mesma lógica utilizadas nos testes de *hardware*. Com este *software* todo o *hardware* desenvolvido é previamente testado, garantindo assim seu funcionamento de acordo com o projeto especificado.

Para o segundo grupo de aplicações, alguns dos programas anteriormente escritos utilizando o *framework* PON foram portados e executados na plataforma NIOS II. Para comparação, criou-se um *software* equivalente funcionalmente utilizando o coprocessador, e uma variação do *framework*, que tem por finalidade configurar e utilizar o coprocessador mantendo a mesma interface de aplicação (API) do PON puramente em *software*.

As alterações feitas no atual *framework* para o PON visam configurar e utilizar o coprocessador de forma transparente ao usuário. Dessa forma grande parte das aplicações já desenvolvidas utilizando-se o PON podem ser reutilizadas, com alterações mínimas no código.

Aplicações que utilizam apenas Atributos, Premissas, Condições e Regras simples são completamente suportadas. Já casos onde se utilizam recursos avançados do PON, como Sub-Condições não são suportados pela atual versão do CoPON. Em alguns casos podemos contornar as limitações através de artifícios de *software*, que devem ser analisados para cada situação.

### 3.4 ESTRUTURA DO PON EM *HARDWARE*

O fato de o PON ser representado com uma cadeia de entidades computacionais simples e imutáveis, operando em paralelo, traz indícios de que o mesmo possa ser adequado à implementação em *hardware*.

Analisando-se o atual *framework*, criado para materializar as ideias referentes ao paradigma orientado a notificações, pode-se concluir que Atributos, Premissas, condições e partes das Regras podem trazer benefícios se implementados em *hardware*, sem trazer prejuízos à usabilidade do *framework*. Os demais componentes podem assumir diferentes formas, e dependem exclusivamente da aplicação em questão para sua criação (JASINSKI, 2012).

Um Método, por exemplo, é composto diretamente por código C++ e pode servir de interface entre o código escrito utilizando o PON e código imperativo tradicional. Dessa maneira, se o mesmo for implementado em *hardware*, pode-se perder em flexibilidade no *framework* já a interligação entre *software* PON e código legado será dificultada.

Optou-se por desenvolver os componentes indicados na Figura 17 (Atributos, Premissas, condições e Regras), criando um *hardware* genérico que poderá ser utilizado por qualquer aplicação que siga o PON. O *hardware* gerado visa eliminar os gargalos associados com a implementação do PON puramente em

*software*, transferindo a carga de processamento diretamente para um coprocessador (CoPON).

Este coprocessador foi implementado como um periférico para o processador NIOS II e é capaz de executar toda a cadeia de inferências, reduzindo o uso de recursos de processamento pelo *software*. O SoC resultante é constituído do processador, módulos de memória, periféricos de I/O e o periférico acelerador, compartilhando o mesmo endereçamento de memória, permitindo um uso transparente do coprocessador pelo *software*, como pode-se ver, por exemplo na Figura 18. Nesta figura observa-se a forma com que o coprocessador é conectado ao processador, através dos barramentos de dados e de endereços. Pode-se ver também o compartilhamento entre os periféricos do canal de interrupção do sistema.

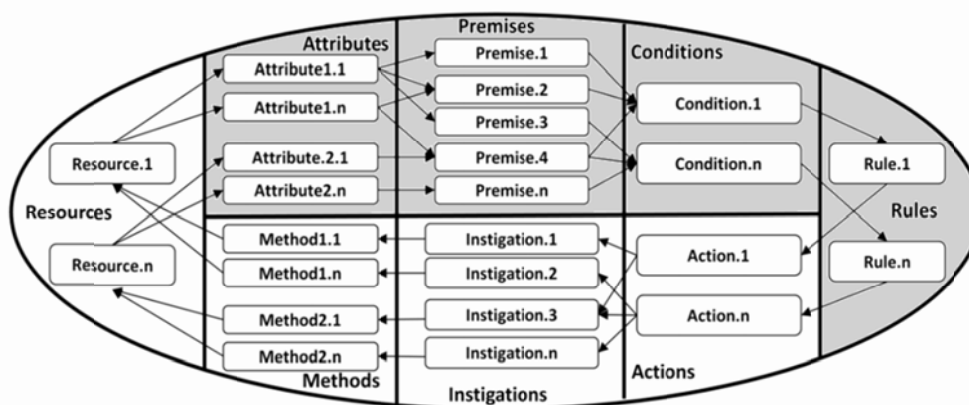
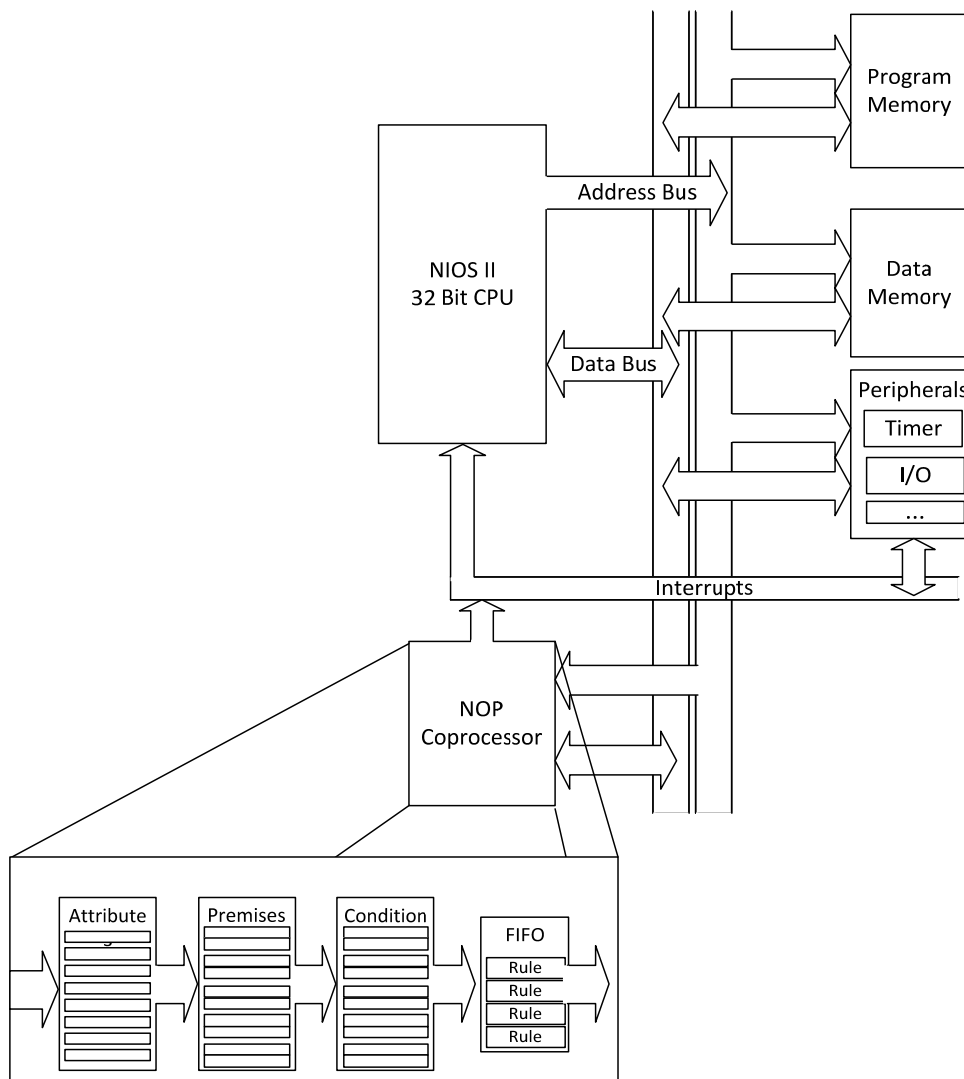


Figura 17 - Representação dos componentes implementados no periférico

Fonte: adaptado de Banaszewski (2009)



**Figura 18 - Representação geral do sistema**

Fonte: Própria

Como *hardware* possui a característica de ser configurável no momento da compilação do projeto do SoC no SOPC Builder, o número de Atributos, Premissas, condições e de Regras aprovadas armazenadas podem ser customizados, garantindo uma boa relação entre recursos utilizados e flexibilidade do sistema. A configuração do coprocessador é feita através da ferramenta *wizard* apresentada na Figura 19.

O campo ADDRESS\_WIDTH define quantos bits serão utilizados para endereçamentos dos componentes e está ligado diretamente ao número de componentes configurados.

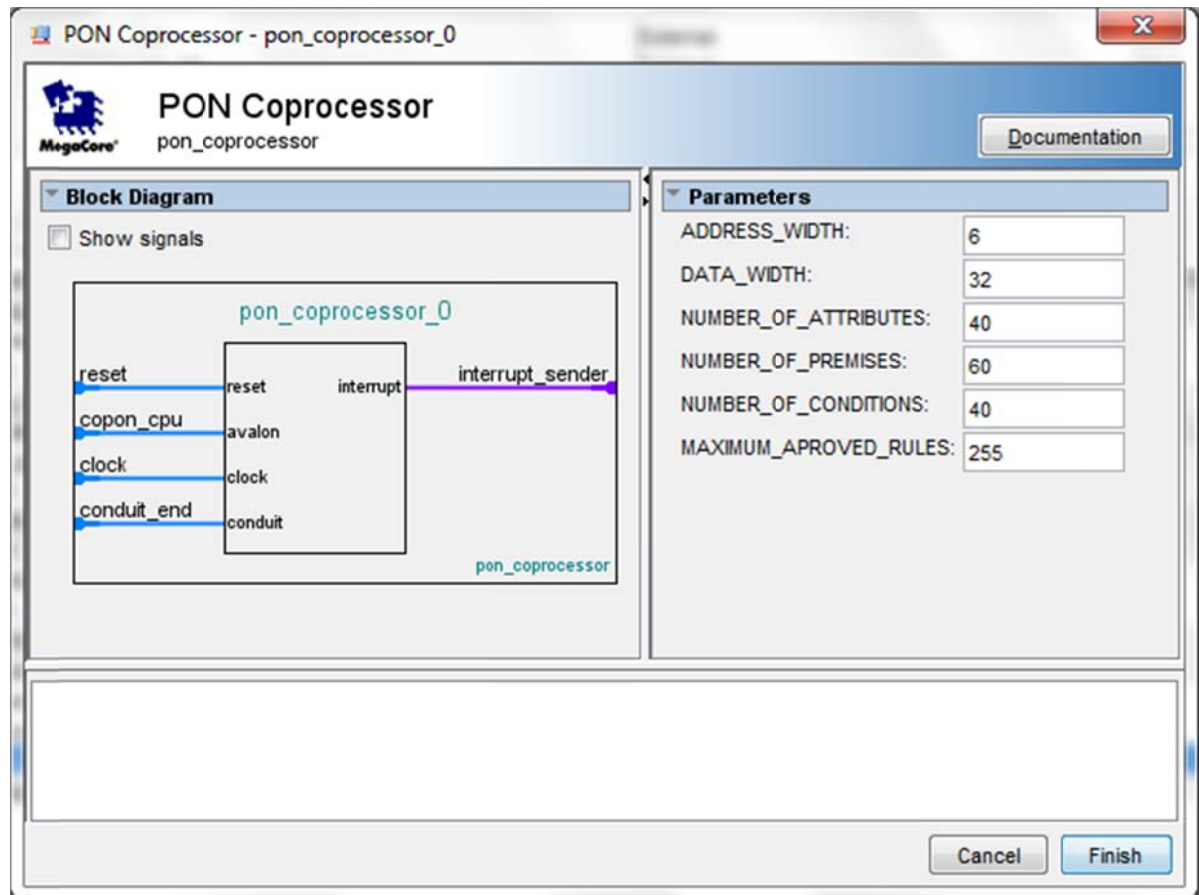


O campo `DATA_WIDTH` define a largura do campo binário de dados utilizado no processador e está relacionado diretamente com o tamanho dos dados do processador. No caso do processador NIOS II este valor é de 32 bits, portanto este é o valor padrão, mas assim como o campo de endereços, preferiu-se deixá-lo configurável, para garantir o funcionamento com outros periféricos diferentes do NIOS II, já que o CoPON cumpre as definições do barramento Avalon.

Os campos `NUMBER_OF_ATTRIBUTES`, `NUMBER_OF_PREMISES` e `NUMBER_OF_CONDITIONS` se relacionam respectivamente com o número de Atributos, Premissas e condições disponíveis no sistema.

O campo `MAXIMUM_APPROVED_RULES` define o número máximo de Regras aprovadas que serão armazenadas dentro do periférico para leitura em *software*. Este campo diz respeito ao tamanho da FIFO interna do periférico. Através de testes práticos, comprovou-se ser necessário um mínimo de 2 endereços armazenados, com um número ótimo igual ao dobro do número de condições configuradas no periférico. Como é um componente com alto custo de *hardware*, deve-se analisar o *software* para que se encontre um número ideal. Se o valor definido for muito baixo, corre-se o risco do periférico não poder aprovar uma Regra; se for muito alto, *hardware* desnecessário será gerado.

Por via de Regra, se o *software* for pequeno e rápido, podemos manter uma FIFO pequena, mas se o aplicativo possuir métodos lentos, deve-se preferir uma FIFO de maior tamanho.



**Figura 19 - Wizard para configuração do CoPON**

Fonte: própria

Cada um dos componentes criados possui características própria, portanto foram criados como entidades individuais, unidas por um sistema de controle. Cada entidade possui um suíte de testes que garante seu funcionamento de acordo com a especificação, bem como o sistema como um todo, que possui uma suíte de testes em *hardware* e em *software*, garantindo seu funcionamento de acordo com a especificação.

### 3.4.1 Atributos

Basicamente, um Atributo em PON oferece duas funcionalidades: armazenar um valor e monitorar seu estado, notificando a Premissa pertinente sobre a alteração de seu valor e repassando a esta Premissa seu novo valor.

Em *hardware*, o Atributo foi subdivido em duas partes, correspondentes à sua função lógica, ficando a cargo da entidade Atributo apenas armazenar o valor. Dessa forma o Atributo PON em *hardware* acaba sendo representado basicamente por um registrador, conforme apresentado no lado esquerdo da Figura 20.

O *hardware* responsável pela detecção de alterações no valor do Atributo difere da metodologia empregada no PON puramente em *software*. Por características de implementação, o mecanismo de detecção de alterações dos Atributos PON em *hardware* está alocado juntamente com o *hardware* da Premissa e é representado pelo lado direito da Figura 20.

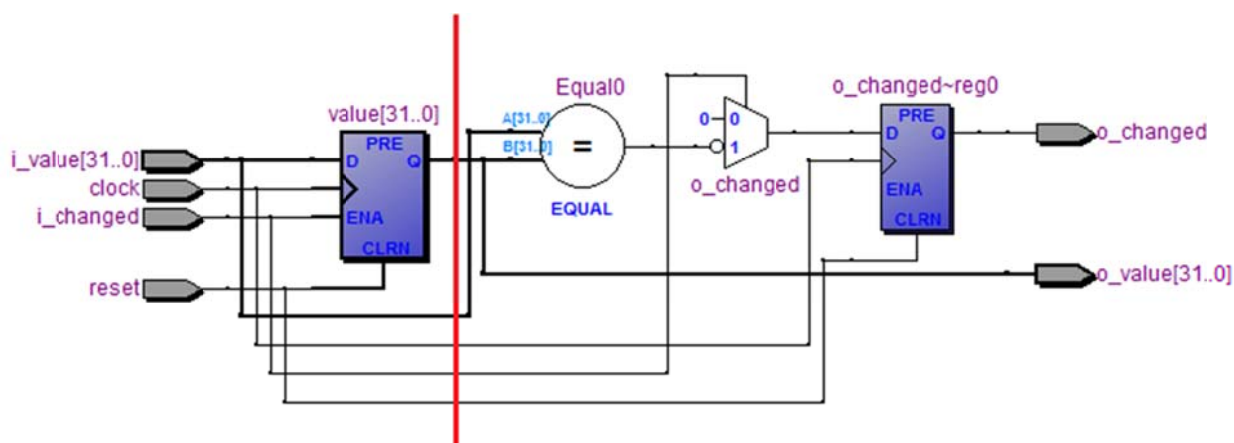


Figura 20 - Representação em *hardware* de um Atributo

Fonte: própria

Tanto na implementação em *hardware* (VHDL) como em *software* (C++), o módulo Atributo deve ser especializado de acordo com o tipo de dados que contém. Neste trabalho, limitou-se aos tipos que podem ser representados em um endereço de 32 bits de memória.

O mapa de memória utilizado por um Atributo PON é pode ser visto no Quadro 1.

Endereço	Conteúdo
Endereço base do Atributo	Valor do atributo (bits 0..31)

Quadro 1 - Mapa de memória de um Atributo PON

### 3.4.2 Premissas

Uma Premissa em PON consiste em uma operação lógica entre o valor de um Atributo e um valor constante, ou entre dos valores de Atributos. Em memória, a Premissa é representada por dois endereços de memória, conforme apresentado no Quadro 2. O primeiro endereço armazena o valor utilizado para comparação com um dos Atributos.

O segundo endereço de memória utilizado pela Premissa está reservado à configuração da Premissa. Este endereço é dividido em dois campos, sendo que o primeiro campo representa em seus 3 bits a operação que será realizada (operações lógicas igual a, diferente de, maior que, maior ou igual a, menor que e menor ou igual a), seguindo como base o Quadro 3. O restante desse campo binário está reservado ao relacionamento com o Atributo que será comparado, sendo que cada bit representará, do bit de menor valor ao bit de maior valor, um Atributo.

Endereço	Conteúdo	
Endereço base da Premissa	Valor da constante (bits 31..0)	
Endereço base da Premissa + 1	Referência ao Atributo (bits 31..4)	Seleção da operação (bits 3..0)

**Quadro 2 - Mapa de memória de uma Premissa PON**

Em suma, estes campos são os selecionadores de dois multiplexadores, um selecionando a operação que será realizada e outro selecionando o Atributo utilizado para a operação.

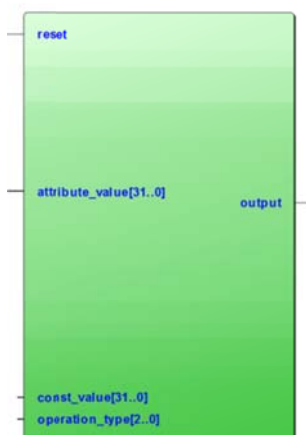
Símbolo do operador (VHDL)	Comparação (operador relacional)	Representação binária da operação
==	igual a	000
/=	diferente de	001
<	menor que	010
<=	menor ou igual a	011
>	maior que	100
>=	maior ou igual a	101

**Quadro 3 - Operações realizadas pela Premissa**

A Figura 21 mostra o bloco de *hardware* e seus sinais, gerados pela Premissa.

O sinais utilizados internamente pela Premissa são compostos de:

- reset: diretamente ligado ao sinal de reset do sistema
  - attribute\_value: o valor do Atributo que será comparado pela Premissa.
  - const\_value: valor que servirá de base de comparação com o valor do Atributo em questão.
  - operation\_type: tipo de operação que será realizada, seguindo como base o Quadro 3, já apresentado.
  - output: representa o resultado da operação lógica realizada pela Premissa.
- Os valores lógicos de todas as Premissas são agrupados em um único vetor, que será utilizado pelas condições.



**Figura 21 - Representação em blocos da Premissa**

Fonte: Própria

Como descrito, a Premissa necessita duas configurações para seu funcionamento. Na Figura 22 pode-se observar a estrutura de componentes internos da Premissa, gerados após a sua sintetização. Cinco comparadores são necessários para a criação de todas as operações lógicas demonstradas no Quadro 3 (os comparadores de igualdade e desigualdade compartilham o mesmo comparador). O valor de 3 bits corresponde à entrada do multiplexador, que selecionará a operação a ser efetuada. Outro multiplexador, não mostrado na figura, será responsável pela seleção do Atributo que servirá de base para comparação. Como no caso dos Atributos, a Premissa está limitada a valores de 32 bits.

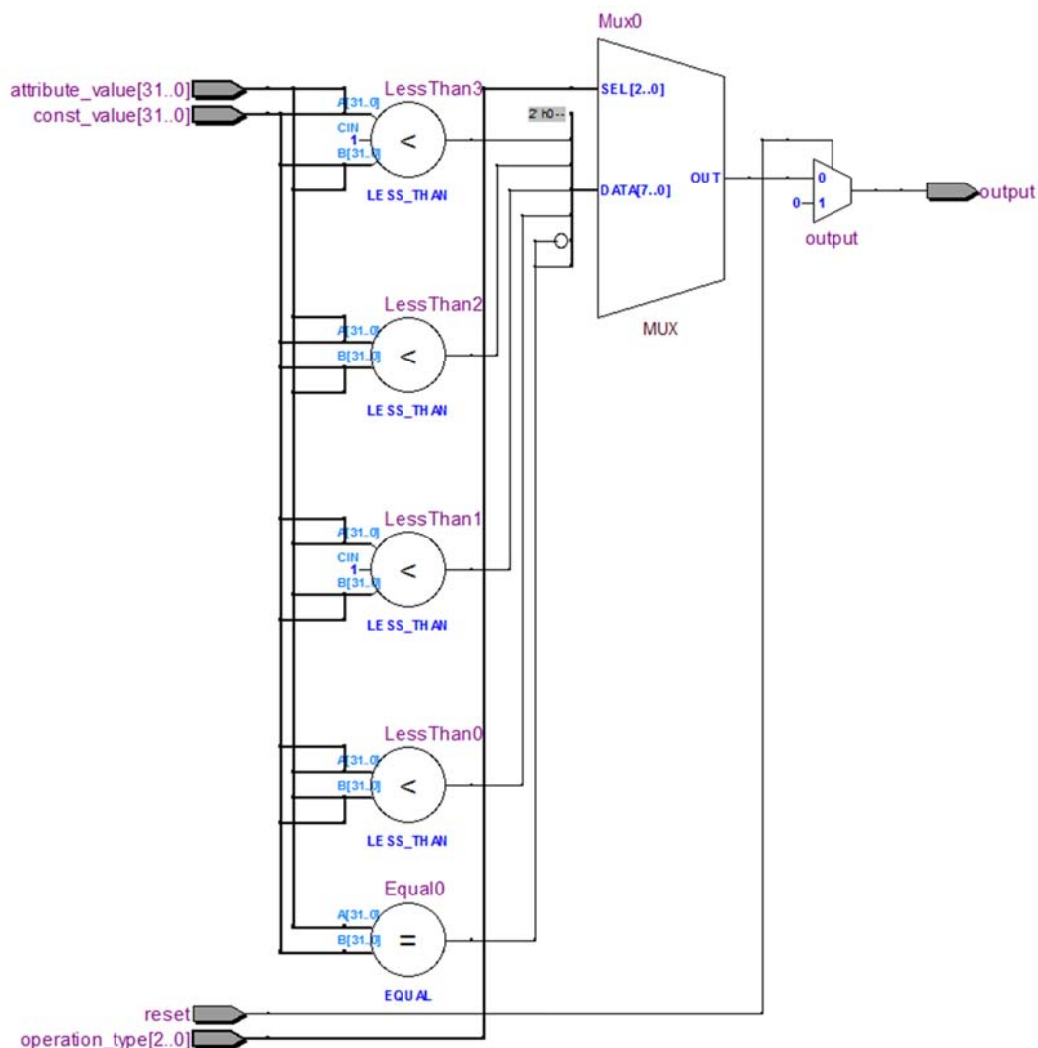


Figura 22 - Estrutura interna da Premissa

Fonte: Própria

### 3.4.3 Condição

A Condição é representada como uma operação lógica cuja entrada é o resultado das operações lógicas realizadas pelas Premissas. Quando uma Condição se torna verdadeira, a Regra associada a ela precisa ser agendada para execução.

Assim, a Condição em *hardware* também é composta por dois endereços de memória, o primeiro endereço faz a correspondência com a sequencia de Premissas que devem estar ativas para a sua aprovação, sendo a Premissa de número 1 representada pelo bit de menor valor. O segundo endereço de memória abriga o

endereço da Regra correspondente à Condição; dessa forma, quando a Condição for verdadeira, a Regra alocada neste endereço será executada. O mapa de memória utilizado por uma Condição PON é mostrado no Quadro 4.

Endereço	Conteúdo
Endereço base da Condição	Máscara de bits da Premissa
Endereço base da Condição + 1	Endereço da Regra

**Quadro 4 - Mapa de memória de uma Condição**

Na Figura 23 observamos a representação do bloco de *hardware* correspondente à Condição. Os sinais representados neste bloco são:

- reset: associado diretamente ao sinal de reset do sistema.
- active\_premises: representa o estado lógico de todas as Premissas do sistema.
- needed\_premises: vetor cujo conteúdo lista as Premissas que deverão estar ativas para que a Condição seja aprovada.
- rule\_addr: endereço da Regra que será executada caso a Condição seja verdadeira.



**Figura 23 - Representação do bloco de *hardware* da Condição.**

Fonte: Própria

A estrutura interna de uma Condição é composta por um comparador entre os valores de saída das Premissas e uma máscara de bits que indica quais devem ser as Premissas ativas para a aprovação da Regra. A estrutura gerada após a sintetização do *hardware* está representada na Figura 24.

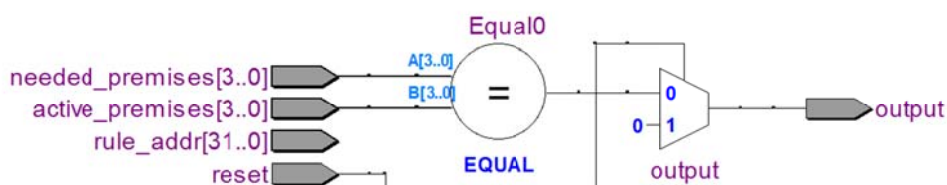


Figura 24 - Representação interna da Condição.

Fonte: Própria

#### 3.4.4 Regra

A Regra em PON é a ligação entre uma Condição que quando aprovada deverá executar um conjunto de instigações. Esse é o ponto de encontro entre o *hardware* do acelerador e o *software* desenvolvido utilizando PON.

Como implementação proposta utiliza rotinas de *software* para implementar as ações e agendamentos não existe nenhum *hardware*, associado diretamente à Regra.

Outrossim, necessita-se armazenar as Regras aprovadas pelo *hardware* para que as mesmas possam ser futuramente utilizadas pelo *software* e indicar ao *software* o evento de aprovação da Regra. Para tanto, o sistema armazena todos os endereços das Regras aprovadas em uma FIFO, desenvolvida para esse fim, e representada na Figura 25.

Essa FIFO é capaz de alocar e de ler um valor armazenado em um ciclo de *clock*, para que dessa forma não se torne um gargalo no sistema.

Os sinais utilizados são:

- clk: conectado diretamente ao sinal de *clock* do sistema.
- reset: conectado diretamente ao sinal de reset do sistema.
- dataout: vetor que contém um endereço de Regra aprovado.
- read\_enable: quando colocado em nível lógico alto, fornece um novo endereço de Regras, que poderá ser lido no vetor *dataout*.
- datain: vetor de entrada de um novo endereço de Regra aprovada.



- `write_enable`: quando colocado em nível lógico alto, faz com que o valor de `datain` seja armazenado em memória.
- `empty`: sinal que representa que não há nenhuma Regra armazenada na fifo.

A versão negada do sinal `empty` é utilizada em conjunto com o estado interno do componente para gerar um sinal ao periférico, indicando que existe uma Regra aprovada. Esse sinal pode ser usado em *software* para disparar uma interrupção no processador.



**Figura 25 - Representação da FIFO utilizada para armazenamento das Regras aprovadas.**

Fonte: Própria

### 3.4.5 Interface e Lógica de Configuração

Cada um dos componentes tem um número de bits de configuração que precisam ser configurados pela aplicação, conforme mencionado anteriormente. Para que todos os componentes criados possam interagir e realmente criar a cadeia funcional de notificações, fez-se necessário o desenvolvimento de uma extensa lógica de controle, capaz de gerenciar um número variável de componentes e criar a interface de dados entre o periférico e o barramento Avalon.

Todos os registradores de *cache*, que tornam o periférico capaz de executar todas as operações de leitura e gravação de registradores em apenas um ciclo de *clock*, são gerenciados por este componente.

Outra função do componente é processar o sinal de “Regra aprovada”, gerado pela Condição e juntamente com os dados de cache criar a lógica de interrupção, utilizada pelo sistema.

### 3.4.6 Interface com o NIOS II

O processador Altera NIOS II utiliza a estrutura de interconexão (barramento) Avalon. Este barramento é constituído de uma coleção de decodificadores, multiplexadores e árbitros, para prover um caminho de transferência concorrente para os dados e instruções.

Para criar a comunicação entre o processador e o periférico acelerador do PON, criou-se um circuito invólucro que segue a especificação para periféricos mapeados em memória do barramento Avalon (Avalon MM Peripheral). Dessa forma permitindo que o periférico seja convertido em um componente SOPC e utilizado diretamente dentro do programa Altera SOPC Editor. Com isto, o periférico pode ser integrado e configurado com um componente normal, para ser integrado ao SoC.

A Figura 26 representa o periférico, como este é visto pelo restante do sistema.



Figura 26 - Representação geral do periférico.

Além dos sinais utilizados pelo barramento Avalon MM, o periférico também oferece um sinal de interrupção, ligado ao controlador interno de interrupções do processador e um sinal *rule\_ready*, que indica que há uma Regra aprovada e pode ser utilizada para acionar outros dispositivos de *hardware* do sistema. Com ela, pode-se, por exemplo, controlar o dispositivo gerador de *clock* do sistema, para que quando exista uma Regra pronta para ser processada aumente a frequência do sistema, dessa forma economizando energia durante o período de inatividade e garantindo um bom desempenho para as ações a serem tomadas quando há uma Regra para ser processada.

### 3.4.7 Mapa de Memória

O acesso às entidades criadas para construção da cadeia de notificações em *hardware* é feito diretamente no espaço de memória do sistema. O Quadro 5 mostra uma representação do mapa de memória, como o mesmo é acessado pelo *software*.

Todos os endereços são atribuídos dinamicamente no momento da instanciação do coprocessador no Altera SOPC Builder e dependem diretamente do número de componentes do PON configurados.

O primeiro endereço, *status\_register*, contém dados de status do periférico, como indicação de Regra pronta para ser executada (bit 0) e FIFO cheia (bit 1).

O segundo endereço, *config\_register*, contém dados de configuração e sua principal função é ligar e desligar a cadeia de notificação (bit 0).

O terceiro endereço contém o valor do primeiro Atributo do sistema. O espaço de memória dos Atributos é igual ao número de Atributos configurados no periférico.

As Premissas iniciam-se no endereço subsequente aos reservados para os Atributos. Cada uma das Premissas configuradas possui dois endereços sequenciais de memória, sendo o primeiro o valor da Premissa e o segundo sua configuração. O número de endereços de memória reservados às Premissas, conseqüentemente, é igual a duas vezes o número de Premissas.

Assim como nas Premissas, cada uma das condições também necessita de dois endereços de memória, um para configuração da Condição e um para armazenar o endereço da Regra que será executada, quando esta Condição se tornar verdadeira. As condições estão alocadas nos endereços subsequentes às Premissas e o número de endereços necessários também equivale a duas vezes o número de condições configuradas.

O último endereço do periférico é reservado para leitura das Regras aprovadas e armazenadas na FIFO do periférico. Cada endereço de Regra é lido sequencialmente, na ordem em que as mesmas foram aprovadas.

Endereço	Conteúdo
0x04001000	status register
0x04001001	config register
0x04001002	attribute[1]
0x04001003	attribute[2]
0x04001004	attribute[3]
0x04001005	premise[1]
0x04001006	premise_config[1]
0x04001007	premise[2]
0x04001008	premise_config[2]
0x04001009	condition[1]
0x0400100A	condition_config[1]
0x0400100B	condition[2]
0x0400100C	condition_config[2]
0x0400100D	rule_ready

**Quadro 5 - Mapa de memória do periférico**

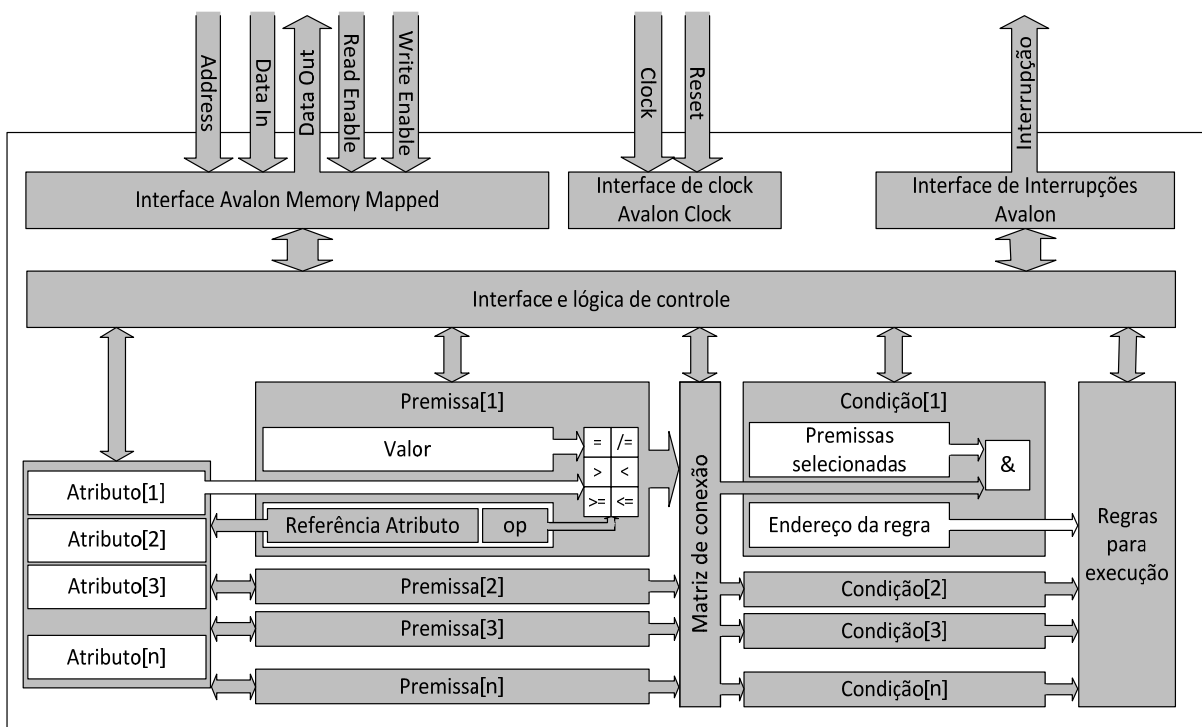
Na Figura 27 pode-se observar o sistema completo, com a relação entre os diversos componentes internos e seu princípio de funcionamento, juntamente com as interfaces de ligação entre o componente e o SoC.

As interfaces com o barramento Avalon mostradas encapsulam o sistema. Dessa forma, para que o coprocessador seja utilizado em outro processador, somente é necessário adaptar-se este encapsulamento a outro tipo de barramento. Essa interface é responsável pela comunicação de dados, interrupções e *clock* com o processador.

A interface de controle é responsável pela interligação e controle dos diferentes componentes do sistema, e sua ligação com a lógica do barramento. Na metade inferior da figura, observamos os componentes do PON criados em *hardware* e a ligação entre os mesmos.

Conforme descrito as Premissas tem função de avaliar os estados dos Atributos e checar qualquer alteração em seus valores, executando então a lógica interna da Premissa. As Premissas estão conectadas as Condições através de uma matriz de conexão, dessa forma as Condições avaliam os estados notificados pelas Premissas e quando aprovadas armazenam na FIFO o endereço correspondente a Regra que será executada.

Com a aprovação de uma Regra, a lógica de controle dispara uma interrupção, que será enviada via Interface de Interrupções Avalon para o processador que poderá tratá-la e executar o restante da cadeia de notificações.



**Figura 27 – Hardware Acelerador**

Fonte: própria

### 3.5 ALTERAÇÕES NO FRAMEWORK

O periférico gerado tem toda sua interface com o *software* baseada em operações de leitura e escrita em registradores de memória, seguindo o mapa de memória já apresentado.

Na materialização atual do PON, sob a forma de um *framework*, o mecanismo de notificações consiste no mecanismo interno de execução das instâncias do paradigma proposto, o qual determina o fluxo de execução das aplicações. Por meio deste mecanismo, as responsabilidades de um programa são divididas entre os objetos do modelo, os quais cooperam por meio de notificações, informando uns aos outros as parcelas de suas contribuições a fim de formar o fluxo de execução do programa (SIMÃO et al, 2012d).

As relações pelas quais os objetos colaboram é apresentada no diagrama de classe da UML na Figura 28. Neste, os objetos das classes Rule e FBE (Elemento da Base de Fatos) se apresentam em extremidades opostas e se relacionam com o auxílio dos objetos colaboradores conforme as conexões modeladas. Pode-se constatar a modelagem de dois fluxos opostos de notificações: o fluxo ativo e o passivo em relação aos objetos da extremidade. Por exemplo, o fluxo de notificações originado no FBE é ativo em relação ao FBE e passivo em relação ao Rule. De maneira oposta, o fluxo de notificações originado na Rule é ativo em relação à Rule e passivo em relação ao FBE. Estes fluxos são formados pela cooperação entre os objetos colaboradores destas extremidades.

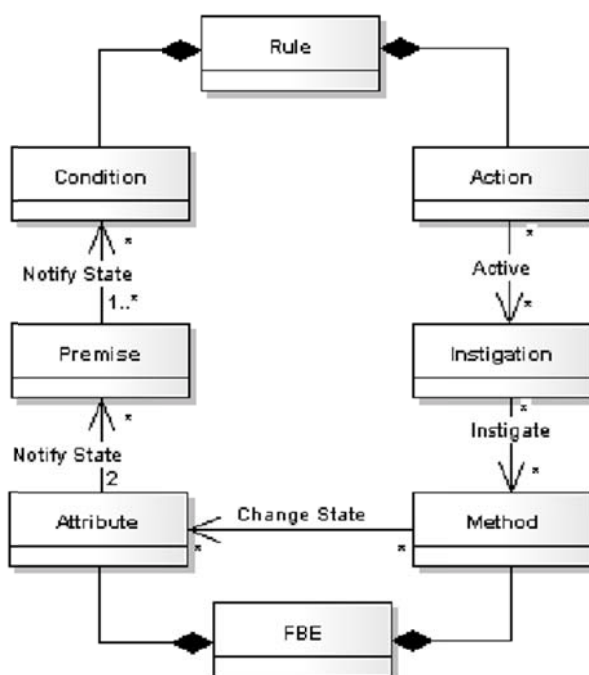


Figura 28 - Relação entre os objetos principais e colaboradores

Fonte: adaptado de Banaszewski (2009).

Na implementação do *framework*, para utilização com CoPON, manteve-se o mesmo conjunto de classes, apenas alterando suas funcionalidades, já que a cadeia de notificações entre a instância da FBE e a instância da Rule não é mais necessária, sendo executada em *hardware*.

Como o sistema é baseado em registradores para todas as funções relativas ao periférico, fica a cargo da instância da classe correspondente ao componente do periférico, configurar os registradores relacionados com sua instância em *hardware*. Para tanto, no método construtor de cada classe, implementou-se um mecanismo de escrita nos registradores do periférico.

Dessa forma, toda a configuração do periférico fica transparente ao programador, deixando a cargo dele apenas a tarefa de sinalizar ao *hardware* quando o mesmo já possui todos os dados necessários e pode começar a processá-los.

A interface de aplicação de programação (API) desenvolvida manteve o mesmo padrão utilizado na implementação completamente em *software* do PON, reduzindo ao máximo as alterações necessárias para o funcionamento das aplicações legadas no novo sistema.

Nas Figura 29 e Figura 30 podemos comparar dois trechos de código, os quais declaram os objetos necessários para a aplicação Controle Remoto, que será apresentada no próximo capítulo. Podemos verificar que as mudanças são mínimas e dizem respeito, principalmente, a partes ainda não implementadas no coprocessador, como diferentes tipos de condições.

```
// Cria os Atributos
Attribute* remoteControl = new Attribute(RemoteControl::RELEASED);
Attribute* gateStatus = new Attribute(Gate::OPENED);

// Cria as condições, juntamente com suas Premissas relacionadas
Condition* cond1 = new Condition(Condition::CONJUNCTION);
cond1->addPremise(new Premise(remoteControl-
>bIsPressed, Boolean::TRUE_NOP, Premise::EQUAL));
cond1->addPremise(new Premise(gate->GateStatus, Gate::CLOSED, Premise::EQUAL));

Condition* cond2 = new Condition(Condition::CONJUNCTION);
cond2->addPremise(new Premise(remoteControl-
>bIsPressed, Boolean::TRUE_NOP, Premise::EQUAL));
```

```

cond2->addPremise(new Premise(gate->GateStatus, Gate::OPENED, Premise::EQUAL));

// Cria as ações
Action* action1 = new Action();
action1->addInstigation(new Instigation(remoteControl->bIsPressed, false));
action1->addInstigation(new Instigation(gate->mtOpenGate));

Action* action2 = new Action();
action2->addInstigation(new Instigation(remoteControl->bIsPressed, false));
action2->addInstigation(new Instigation(gate->mtCloseGate));

// Cria as Regras
Rule* rule1 = new Rule("Open gate", scheduler, cond1, action1, false);
Rule* rule2 = new Rule("Close gate", scheduler, cond2, action2, false);

```

**Figura 29 – Código desenvolvido utilizando o *framework* PON original**

```

// Cria os Atributos
Attribute* remoteControlButton = new Attribute(RemoteControl::RELEASED);
Attribute* gateStatus = new Attribute(Gate::OPENED);

// Cria as condições e suas Premissas relacionadas
Condition* cond1 = new Condition();
cond1->addPremise(new Premise(remoteControlButton, RemoteControl::PRESSED, Premise::EQUAL));
cond1->addPremise(new Premise(gateStatus, Gate::CLOSED, Premise::EQUAL));

Condition* cond2 = new Condition();
cond2->addPremise(new Premise(remoteControlButton, RemoteControl::PRESSED, Premise::EQUAL));
cond2->addPremise(new Premise(gateStatus, Gate::OPENED, Premise::EQUAL));

// Cria as ações
Action* action1 = new Action();
action1->addInstigation(new Instigation(remoteControl->mtReleaseButton));
action1->addInstigation(new Instigation(gate->mtOpenGate));

Action* action2 = new Action();
action2->addInstigation(new Instigation(remoteControl->mtReleaseButton));
action2->addInstigation(new Instigation(gate->mtCloseGate));

// Cria as Regras
Rule* rule1 = new Rule("Open gate", scheduler, cond1, action1, false);
Rule* rule2 = new Rule("Close gate", scheduler, cond2, action2, false);

// Inicia o coprocessador
write_register(CONFIG_ADDRESS, 1);

```

**Figura 30 – Código desenvolvido utilizando o *framework* PON modificado para utilizar o CoPON**

### 3.6 FLUXO DE EXECUÇÃO DO COPROCESSADOR

Os componentes do PON, executados no coprocessador são configurados através dos registradores específicos, que são automaticamente escritos durante a criação dos objetos C++ do PON.



Cada Atributo recebe em seu único registrador um inteiro correspondente ao valor do Atributo. A configuração de cada atributo é feita automaticamente pelo framework, usando para sua criação em software a seguinte síntese:

```
//Attribute* novoAtributo = new Attribute(valorInicial);
Attribute* remoteControlButton = new
Attribute(RemoteControl::RELEASED);
```

Cada uma das Premissas recebe o valor que servirá como base de comparação em seu primeiro registrador. No segundo registrador de cada premissa, iniciando se no quarto bit, carrega-se uma referência ao atributo com qual será comparada, onde cada bit, representa um atributo do coprocessador. Nos três bits que restaram, configura-se a operação que será realizada pela premissa que está sendo configurada, seguindo os valores apresentados no Quadro 3.

A síntese para criação da premissa, conforme definido pelo framework e a seguinte:

```
//Premise* premissa = new Premise(ponteiroParaAtributo,
valorDeComparacao, codigoDaOperação);
Premise* p1 = new
Premise(remoteControlButton, RemoteControl::PRESSED, Premise::EQUAL);
```

Para cada condição deve-se configurar, em seu primeiro registrador, quais premissas deverão estar ativas para que a condição se torne verdadeira. Neste campo cada bit representa uma premissa do sistema, iniciando-se pelo bit menos significativo.

No segundo registrador do atributo, deve-se armazenar o endereço de memória da regra que será executada, caso a condição se torne verdadeira. Este endereço será utilizado por um ponteiro C/C++ para execução da regra em software.

A síntese para criação da Condição segue a seguinte forma:

```
// Cria-se a condição e adiciona-se as premissas necessárias
Condition* cond1 = new Condition();
cond1->addPremise(p1);
cond1->addPremise(p4);
```

Quando o coprocessador estiver configurado adequadamente, inicia-se a execução do mesmo através da ativação do primeiro bit do registrador de configuração do coprocessador.

No código do programa, pode-se executar essa tarefa da seguinte maneira:

```
write_register(ENDERECO_INICIAL_COPON + 1, 1);
```

Durante a execução do software, quando o coprocessador possuir uma regra aprovada, será gerada uma interrupção para o processador. Esta interrupção deverá ser tratada pelo *software*, executando o método "approved()" da regra aprovada.

Como apenas o endereço da regra está armazenado no coprocessador, precisa-se buscá-lo e assim recuperar o objeto da Regra. Uma das maneiras de se executar essa tarefa, é apresentada a seguir:

```
Rule *objpointer;  
unsigned int rule = read_register(RULE_ADDRESS);  
objpointer = (Rule*)rule;  
objpointer->approved();
```

A partir do momento em que uma Regra é aprovada, toda a execução do PON passa a ser em software, seguindo o fluxo do *framework* PON já existente.

Além da interrupção, pode-se também chegar à existência de uma regra aprovada através de *polling*. O registrador STATUS do coprocessador mantém a informação de regras aprovadas em seu primeiro bit.

Dessa forma, em *software* onde o uso de interrupções não é possível, ou onde já se utilize algum sistema de *polling* ou qualquer tipo de loop infinito, pode-se ler esse registrador e dessa forma acompanhar a aprovação das regras. O tratamento das Regras aprovadas, é o mesmo dos sistemas que se utilizam de interrupções.

Os demais componentes do PON seguem as mesmas sínteses utilizadas pelo framework padrão do PON.

Um exemplo de código funcional de uma aplicação pode ser vista no APÊNDICE 1. Por motivos de entendimento e legibilidade do código, esta versão não utiliza interrupções e sim *polling*.

### 3.7 REFLEXÃO

O coprocessador apresentado neste capítulo, apesar de limitado em alguns aspectos (e.g. sub-Condições, atributos de ponto flutuante), é capaz de executar muitas das aplicações já criada seguindo o PON.

Para seu desenvolvimento, além do método apresentado, houve uma abordagem alternativa utilizando-se somente circuitos combinacionais e técnicas de sistemas assíncronos. Essa abordagem não se mostrou útil por ter um elevado consumo de *hardware* e um complexidade superior a alternativa apresentada.

Um dos cuidados tomados no desenvolvimento do *hardware* foi de se manter uma cobertura completa de testes, para que dessa forma o trabalho possa ter continuidade, com o desenvolvimento de novas funcionalidade para acompanhar o desenvolvimento do PON. Com o desenvolvimento dos testes, buscou-se além de assegurar que as funcionalidades estejam corretas, documentar todo o sistema.

Com a utilização dos conceitos de *Test-driven development* (TDD) e *Behavior Driven Development* (BDD), auxiliados por *scripts* criados em linguagens de alto-nível e pacotes VHDL desenvolvidos para tal fim, pôde-se chegar a resultados de testes que além de auxiliarem na localização de erros, documentam o funcionamento do sistema.

## 4 RESULTADOS

### 4.1 INTRODUÇÃO

A fim de validar o *hardware* apresentado no Capítulo 3, foram portadas três aplicações construídas originalmente utilizando o Paradigma Orientado a Notificações.

Este trabalho não se ateve a comparações entre paradigmas, mas optou-se pela utilização das aplicações previamente desenvolvidas para esse fim, por permitir comparações com trabalhos anteriores, devidamente estabelecidos e testados.

#### 4.1.1 Comparações de Utilização de Recursos de *Hardware*

O coprocessador apresentado foi implementado em *hardware* utilizando linguagem VHDL e sintetizado em duas FPGAs diferentes, sendo uma de baixo custo e outra de alto desempenho, a fim de avaliar as variações de frequência máxima ( $f_{MAX}$ ) e o acréscimo de utilização de *hardware* causado pela adição do coprocessador, quando comparado a um *hardware* semelhante, sem o coprocessador.

Para avaliar o acréscimo de área utilizada na FPGA, decorrente da adição do coprocessador, duas configurações do coprocessador foram utilizadas, uma contendo o mínimo de componentes necessários para o funcionamento do coprocessador, outra contendo a configuração padrão, utilizada também para as comparações de *software*. As diferentes configurações são sumarizadas através do Quadro 6.

	Configuração #1	Configuração #2
Número de Atributos	1	4
Número de Premissas	1	4
Número de condições	1	4

**Quadro 6 - Configurações utilizadas para avaliação de custos de *hardware***

As duas configurações apresentadas foram sintetizadas para a família Cyclone IV, da fabricante Altera, utilizando a FPGA de baixo custo EP4CE22F17C6.

Os resultados demonstraram um acréscimo médio de 27 elementos lógicos para cada Atributo adicionado ao periférico. Da mesma forma, 668 elementos lógicos são acrescentados em média para cada Premissa adicionada e 221 elementos lógicos para cada Condição. Para o sistema de controle e interface com o barramento Avalon são utilizados 860 elementos lógicos em média. Em contrapartida, o SoC utilizado como parâmetro de referência utiliza 2997 elementos lógicos fixos.

Quanto ao número de registradores, o sistema apresenta um acréscimo linear de 32 registradores para cada Atributo adicionado, 32 registradores para cada Premissa adicionada e 68 registradores para cada Condição adicionada ao sistema. O tamanho da FIFO que armazenará as Regras aprovadas também tem influencia sobre o número total de registradores, sendo este incrementado de 35 registradores em média a cada endereço adicionado à FIFO.

A Figura 31 apresenta uma comparação entre as duas configurações analisadas, quando confrontadas com um sistema sem o CoPON. Como pode-se observar foram adicionados 828 elementos lógicos e 550 registradores entre um sistema sem o coprocessador e um sistema com um coprocessador funcional.



**Figura 31 – Acréscimo de *hardware* devido à adição do coprocessador**

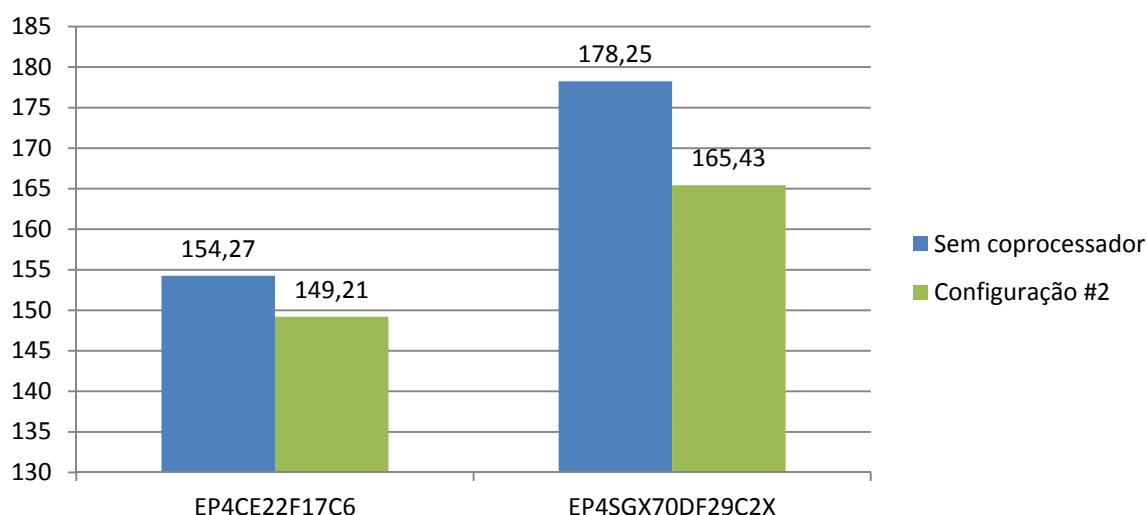
Comparativamente o sistema sem coprocessador utiliza cerca de 13% dos elementos lógicos (LE) de uma FPGA EP4CE22F17C6, que possui um total de 22320 LEs. Com a “Configuração 1” passa-se a utilizar 17% dos LEs da mesma FPGA e para a “Configuração 2” o uso é de 30% dos LEs. Isso implica em cerca de 15 Atributos, 15 Premissas e 15 Condições que podem ser utilizados nesta FPGA. Para a utilização de maiores números de elementos há a necessidade da utilização de uma FPGA maior, sendo que pode-se encontrar dispositivos de baixo custo com até 300K LEs e FPGAs de alto desempenho chegando próximos a 1M LEs (ALTERA, 2012d).

Para avaliação da influência da adição do novo *hardware* em termos de frequência máxima de operação ( $f_{MAX}$ ), o *hardware* foi sintetizado em uma FPGA de alto desempenho da família Stratix IV. da Altera, e em uma FPGA de baixo custo da mesma fabricante, pertencente à família Cyclone IV.

Em ambos os casos, o decréscimo da frequência máxima de operação foi pequeno, ficando abaixo dos 10% para a família Cyclone IV e abaixo de 15% para a família Stratix IV.

A Figura 32 apresenta um comparativo utilizando as FPGAs EP4CE22F17C6 (Cyclone IV) e EP4SGX70DF29C2X (Stratix IV). Utilizou-se uma configuração similar do coprocessador à utilizada para o desenvolvimento de *software*, apresentado a seguir, comparando-as com o sistema sem coprocessador.

Como se pode verificar não houve uma diminuição significativa na frequência máxima de operação. Indicando assim que o acréscimo de *hardware* não influenciou de forma negativa o desempenho geral do sistema, e que os ganhos de processamento poderão ser maiores que as perdas em frequência.



**Figura 32 - Frequência máxima de operação ( $f_{MAX}$ )**

Encerradas as comparações de custos de *hardware*, desenvolveu-se três aplicações diferentes para a comparação dos ganhos de desempenho na execução dos aplicativos.

#### 4.1.2 Comparações de Desempenho de Aplicação

Para validação do *hardware* construído e comparações de desempenho de execução, foram criadas três aplicações de testes, utilizando como base as aplicações apresentadas por Wiecheteck (2011), Banaszewski (2009), Ronszcka (2011) e Simão et al. (2012).

A aplicação intitulada Mira ao Alvo consiste na implementação do tradicional jogo de mira ao alvo. As entidades alvos são simbolicamente representadas como maçãs, e as entidades miras são representadas como arqueiros, sendo que há uma maçã para cada arqueiro. Assim, cada maçã e cada arqueiro recebe uma identificação única, sendo que cada arqueiro somente pode flechar sua respectiva maçã (BANASZEWSKI, 2009).

Cada um dos objetos apresenta características que denotam seu estado, apresentando-se como pronto ou não. As maçãs ainda recebem outros Atributos, um

indicando que a mesma já foi flechada e outro denotando sua coloração, vermelha ou verde.

O jogo consiste em um arqueiro, com Atributo pronto, acertar uma maçã com Atributos pronta e vermelha.

A segunda aplicação, intitulada Semáforo, consiste em um sistema que simula uma situação envolvendo um pedestre, um carro e um semáforo. Quando o semáforo estiver parado, o carro deverá parar e o pedestre deverá atravessar a rua; quando o semáforo estiver fechado o pedestre deverá parar e deixar que o carro continue seu trajeto (RONSZCKA, 2011).

O objeto carro possui um Atributo, especificando se o mesmo está parado ou se movendo. O objeto pedestre possui um Atributo semelhante, indicando se o mesmo está caminhando ou parado. A entidade semáforo possui um Atributo também com dois estados, indicando se o mesmo está aberto ou fechado.

O objetivo da aplicação consiste em monitorar o estado do Atributo do objeto semáforo, e quando o mesmo indicar fechado, disparar uma ação fazendo com que o pedestre se coloque em estado caminhando e o carro em estado parado. Quando o semáforo se apresentar no estado aberto, a ação deverá colocar o pedestre em estado parado e o objeto carro em estado indicando seu movimento.

A terceira aplicação, intitulada Portão Eletrônico, simula a ação de abrir e fechar um portão eletrônico através de um controle remoto (WIECHETECK, 2011).

Esta aplicação consiste em dois objetos e um Atributo para cada objeto. Os objetos são portão e controle, com estados aberto e fechado para o portão e pressionado e liberado para o controle.

A aplicação consiste em alterar o estado do portão quando o controle passar pelos estados pressionado e liberado, indicando uma ação do usuário.

Para o desenvolvimento das aplicações experimentais, utilizou-se a plataforma de *hardware* Altera DE0-Nano Development and Education Board, contendo uma FPGA Altera Cyclone IV 4C22. Este kit é composto por diferentes



componentes de *hardware* necessário para o desenvolvimento do sistema computacional, como memórias, interfaces de I/O e osciladores.

O SoC criado contém além do processador uma interface com a memória externa, um módulo de depuração para o processador, interfaces de I/O, um periférico utilizado para contagem dos ciclos de *clock* utilizados pela aplicação e o coprocessador propriamente dito. A configuração do SoC, feita através do SOPC Builder, consta na Figura 33.

Conn...	Name	Description	Clock	Base	End	IRQ
	cpu_0	Nios II Processor	[clk]			
	instruction_master	Avalon Memory Mapped Master	clk			
	data_master	Avalon Memory Mapped Master	[clk]			
	jtag_debug_module	Avalon Memory Mapped Slave	[clk]	0x04000800	0x04000fff	IRQ 0 - IRQ 31
	sdr*_0	SDRAM Controller	[clk]			
	s1	Avalon Memory Mapped Slave	clk_50	0x02000000	0x03ffffff	
	jtag_uart_0	JTAG UART	[clk]			
	avalon_jtag_slave	Avalon Memory Mapped Slave	clk	0x04001170	0x04001177	
	led_green	PIO (Parallel I/O)	[clk]			
	s1	Avalon Memory Mapped Slave	clk	0x04001140	0x0400114f	
button_pio	PIO (Parallel I/O)	[clk]				
s1	Avalon Memory Mapped Slave	clk	0x04001150	0x0400115f		
switch_pio	PIO (Parallel I/O)	[clk]				
s1	Avalon Memory Mapped Slave	clk	0x04001160	0x0400116f		
performance_counter_0	Performance Counter Unit	[clk]				
control_slave	Avalon Memory Mapped Slave	clk	0x04001100	0x0400113f		
sysid	System ID Peripheral	[clk]				
control_slave	Avalon Memory Mapped Slave	clk	0x04001178	0x0400117f		
pon_coprocessor_0	PON Coprocessor	[clock]				
copon_cpu	Avalon Memory Mapped Slave	clk	0x04001000	0x040010ff		

Figura 33 - SoC criado para execução das aplicações

Fonte: Própria

As medidas foram realizadas utilizando o periférico Altera Performance Counter. Este periférico provê uma maneira de se mensurar o número de ciclos de *clock* de um *software*, ou partes dele, com o mínimo de intrusão possível (ALTERA, 2012c). Em suma, este periférico não utiliza tempo do processador para sua execução e sua configuração utiliza um tempo fixo, que pode ser deduzida do tempo total gasto.

Dois tipos de medições foram feitas. A primeira consiste em avaliar o ganho, considerando somente a área de *software* diretamente afetada pelo coprocessador e considerando o sistema como um todo.

Na segunda abordagem, todo o sistema é considerado, o que mostra uma visão geral dos ganhos, mas em contrapartida considera os itens citados anteriormente. Dessa forma, as medições podem variar com a aplicação e com a maneira como o *software* é desenvolvido.

Na primeira abordagem, considerando somente a área de *software* diretamente afetada pelo coprocessador, disparou-se o contador de ciclos de *clock* no momento da alteração de um Atributo PON, encerrando o contador no momento em que uma Regra é aprovada e encaminha para o Scheduler.

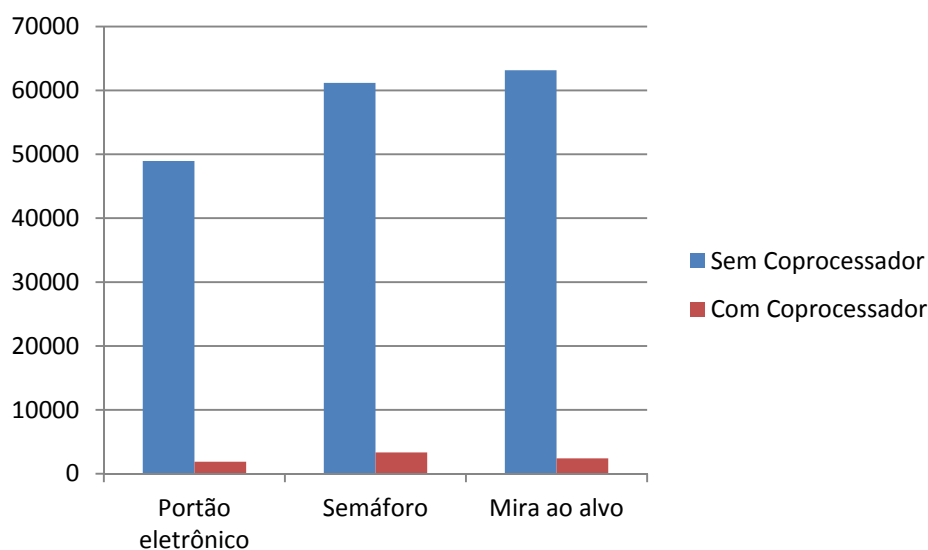
Esta abordagem propicia uma visão direta da redução, pois não considera o custo de criação dos objetos e outros *overheads* originados puramente pela implementação em C++.

Como demonstrado na Tabela 1, houve um ganho médio de 23,6 vezes o desempenho original da aplicação quando utilizado um coprocessador PON.

**Tabela 1 – Comparação entre número de ciclos de *clock* utilizados desde a alteração de um Atributo até a aprovação de uma Regra para execução.**

Aplicação	Sem		Ganho
	Coprocessador	Com Coprocessador	
Portão eletrônico	48957	1884	25,99
Semáforo	61185	3324	18,41
Mira ao Alvo	63170	2399	26,33
<b>Média</b>	<b>57770,67</b>	<b>2535,67</b>	<b>23,57</b>

A Figura 34 mostra a comparação entre os diferentes cenários, utilizando CoPON e não o utilizando, nas três aplicações utilizadas para comparação.



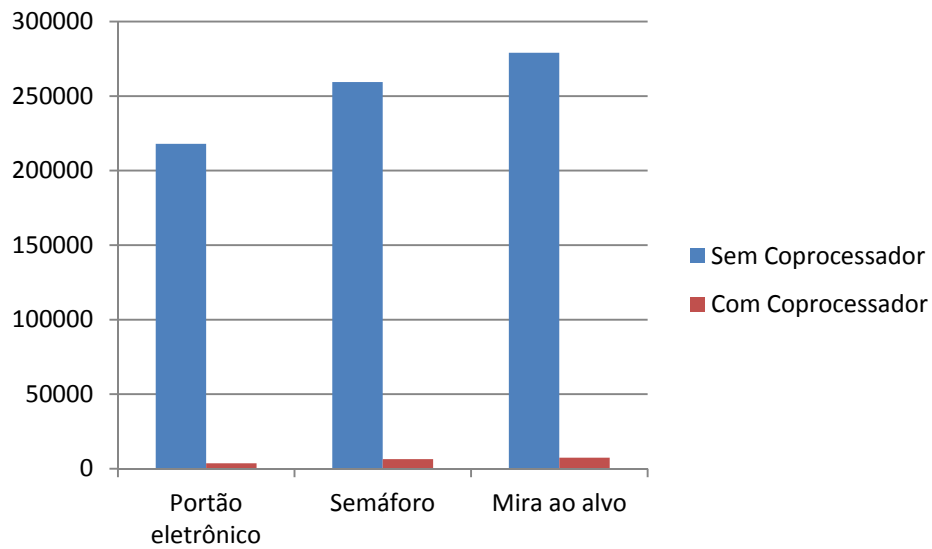
**Figura 34 – Número de ciclos de *clock* do momento da alteração de um Atributo até a aprovação de uma Regra para execução**

Na segunda abordagem, considerando o sistema completo, para as três aplicações utilizadas como base de comparação, chegamos aos valores encontrados na Tabela 2. O ganho médio foi de 45,7 vezes o desempenho do PON original.

**Tabela 2 – Comparação entre número de ciclos de *clock* utilizados pelo sistema em aplicações com e sem *hardware* acelerador**

Aplicação	Sem Coprocessador	Com Coprocessador	Ganho
Portão eletrônico	217978	3681	59,22
Semáforo	259372	6416	40,43
Mira ao Alvo	279018	7466	37,37
<b>Média</b>	<b>252122,67</b>	<b>5854,33</b>	<b>45,67</b>

Na Figura 35, podemos observar um gráfico comparando às três aplicações, com e sem a utilização do coprocessador da cadeia de notificações do PON.



**Figura 35 – Número de ciclos de *clock* utilizados pelo sistema completo**

Estas medições demonstram um grande ganho de desempenho quando utilizamos a aceleração por *hardware* para o PON, o que pode torna-lo viável a sistemas embarcados criados utilizando lógica programável.

## 5 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Neste trabalho apresentou-se um *hardware* periférico coprocessador, capaz de acelerar a cadeia de notificações do Paradigma Orientado a Notificações. A arquitetura proposta foi desenvolvida em linguagem VHDL e implementada tanto em FPGAs de baixo custo como em FPGAs de alto desempenho.

O PON surge como um novo paradigma promissor, tornando o desenvolvimento de aplicativos mais natural, ao se aproximar da forma humana de raciocínio.

A implementação atual do PON carece de melhorias de desempenho por utilizar uma linguagem imperativa para implementá-lo, o que o torna de alto custo computacional, dificultando seu uso em sistemas embarcados e SoCs.

Ao final de todas as etapas realizadas ao longo deste projeto de pesquisa (revisão bibliográfica, levantamento de requisitos, definição da arquitetura, adaptação do processador, criação do coprocessador, adaptação dos demais *cores*, desenvolvimento da aplicação exemplificada, testes de *hardware* e *software* e comparações), chegou-se ao principal resultado do trabalho: a implementação de um coprocessador para o PON funcional, incluindo projetos de *hardware* e *software*, capaz de ser prontamente utilizado por diversos sistemas embarcados.

Em média, o coprocessador PON (CoPON), foi capaz de uma aceleração de 23x, se comparado à implementações puramente em *software* deste paradigma quando considerado somente a cadeia de inferências, e um ganho médio de 45x se considerarmos toda a aplicação PON.

O acréscimo total de *hardware* para a implementação deste coprocessador foi de apenas 24% se comparado a uma implementação tradicional com o processador NIOS II.

Dessa forma, o Coprocessador PON desenvolvido se mostrou viável para a utilização em sistemas embarcados utilizando o processador NIOS II. Como o coprocessador em si não depende de características internas do processador e se

comunica com o sistema através de um invólucro contendo a lógica para o barramento utilizado, o mesmo poderá ser utilizado em outros tipos de processadores, bastando a lógica do barramento ser reescrita. Com a finalização deste projeto, criou-se uma plataforma de desenvolvimento apta para ser utilizada em diversos projetos de sistemas embarcados e SoCs.

## 5.1 TRABALHOS FUTUROS

Uma sugestão para a continuidade deste trabalho é a alocação de registradores diretamente na memória principal do processador, permitindo assim maior facilidade de desenvolvimento do *framework* e disponibilizando um maior número de Atributos com um *hardware* menor.

Outra possibilidade de estudo é o uso de sistemas computacionais como a linha Altera E6X5C, que integra em um mesmo chip um processador x86 padrão e uma FPGA da família Stratix (INTEL, 2012).

Também como possibilidade de estudo futuro, pode se citar a tecnologia OpenCL, que vêm sendo desenvolvida pela Altera e permitirá a geração e uso transparente de *hardware* pelo *software*, da mesma forma como hoje são criados aplicativos para execução em GPUs. Dessa forma, pode-se estudar a criação de um acelerador para o PON utilizando-se essa tecnologia, o que o tornaria mais portátil a outros sistemas.

## REFERÊNCIAS

- ALTERA, Altera Intellectual Property: IP MegaStore. Disponível em: <<http://www.altera.com/products/ip/ipm-index.html>>. Acesso em: 17 abril de 2012a.
- ALTERA , Altera Nios II Processor Reference, Disponível em: <[http://www.altera.com/literature/hb/nios2/n2cpu\\_nii5v1.pdf](http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf)>, 2010, Acessado em: 30 de junho de 2012b).
- ALTERA , Altera Profiling Nios II Systems , Disponível em: <http://www.altera.com/literature/an/an391.pdf><http://www.altera.com/literature/an/an391.pdf>, Acessado em: 23 de junho de 2012c.
- ALTERA , Stratix V FPGAs, Disponível em: <http://www.altera.com/devices/fpga/stratix-fpgas/stratix-v/stxv-index.jsp>, Acessado em: 23 de junho de 2012d.
- BANASZEWSKI, R. F. Paradigma Orientado a Notificações: Avanços e Comparações. Dissertação de Mestrado, UTFPR - CPGEI, Curitiba, 2009. Disponível em: [http://arquivos.cpgei.ct.utfpr.edu.br/Ano\\_2009/dissertacoes/Dissertacao\\_500\\_2009.pdf](http://arquivos.cpgei.ct.utfpr.edu.br/Ano_2009/dissertacoes/Dissertacao_500_2009.pdf), Acessado em: 13 de dezembro de 2011.
- BERGER, A. Embedded systems design. Kansas: CMP Books, 2002.
- BERGIN, T., GIBSON, R. History of Programming Languages (2a Edição). Addison Wesley, 1996.
- BROWN, Stephen.; VRANESIC, Zvonko. Fundamentals of digital logic with vhdl design. 2 ed. Toronto: Mc Graw Hill, 2005.
- CHU, Pong. Embedded SoPC Design with Nios II Processor and VHDL Examples. New Jersey, USA, John Wiley & Sons, 2011., 2011
- COMPTON, Katherine.; HAUCK, Scott. Reconfigurable computing: a survey of systems and *software*. ACM Computing Surveys, New York, volume 34, no. 2, 2002. p.171-210.
- ESKINAZI, Remy; et al. FPGAs Dinamicamente Reconfiguráveis: Fluxo de Projeto e Vantagens na Concepção de Circuitos Integrados. In: II Congresso Brasileiro de Tecnologia – CONBRATEC, Recife, 10 a 12 de Novembro de 2005.
- HABINC, Sandi. Using VHDL Cores in System-On-A-Chip Developments. ESA SP-439: European Space Components Conference : ESCCON 2000, European Space Agency, Vol. 439, p.159-167, June 2000.
- HAMBLEEN, James. O.; FURMAN, Michael. D. Rapid prototyping of digital systems. 2

ed. Boston: Kluwer Academic Publishers, 2001.

HARTENSTEIN, Reiner. A decade of reconfigurable computing: a visionary retrospective. Proceedings of the conference on Design, automation and test in Europe table of contents, Germany, 2001, page 642–649.

HEATH, Steve. Microprocessor Architectures and Systems: RISC, CISC & DSP. Manchester. Butterworth-Heinemann Ltd. 1991.

IENNE, Paolo.; POZZI, Laura.; VULETIC, Miljan. On the limits of automatic processor specialization by mapping dataflow sections on ad-hoc functional units. Technical report CS 01/376, Swiss Federal Institute of Technology Lausanne Processor Architecture Laboratory, 2002.

INTEL, Processador Intel® Atom™ série E6x5C. Disponível em: 2012  
[http://www.intel.com/p/pt\\_BR/embedded/hwsw/hardware/atom-e6x5c/overview](http://www.intel.com/p/pt_BR/embedded/hwsw/hardware/atom-e6x5c/overview).  
 Acessado em: 30 de junho de 2012.

J. M. Simão, P. C. Stadzisz, C. R. Erig Lima, F. A. de Witt, R. R. Linhares. "Paradigma Orientado a Notificações em Hardware Digital". Pedido de Proteção Industrial e Pedido de Patente enviados à Agência de Inovação da UTFPR respectivamente em 11/05/2012 e 17/07/2012, Curitiba - PR, Brasil - Aguardando Aprovação da Agência para eventual envio para o INPI.

JASINSKI, Ricardo P. Implementação de uma CPU Tolerante a Falhas em Lógica Programável. 2004. f.169. Dissertação (Mestrado em Engenharia Elétrica e Informática Industrial). CPGEI, CEFET-PR, Centro Federal de Educação Tecnológica do Paraná, Curitiba.

JASINSKI, Ricardo P. *Framework* para Geração de *Hardware* em VHDL a partir de modelos em PON. Relatório interno restrito para a disciplina de Lógica Reconfigurável por Hardware. Prof. Dr. C. R. Erig Lima. 2012.

LEE, Weng F. Verilog Coding for Logic Synthesis. New Jersey: A John Wiley & Sons, 2003

LINHARES, R. R. ; RONSZCKA, A. F. ; VALENÇA, G. Z. ; Batista, M. V. ; Lima, C. R. E. ; Witt, F. A. ; SIMÃO, J. M. ; STADZISZ, P. C. . Comparações entre o Paradigma Orientado a Objetos e o Paradigma Orientado a Notificações sob o contexto de um simulador de sistema telefônico.. In: III Congreso Internacional de Computación y Telecomunicaciones - COMTEL 2011, 2011, Lima - Peru.

NEWELL, A., SIMON, H. A. Human Problem Solving. Englewood Cliffs, NJ, USA: Prentice-Hall, 1972.

PATTERSON, David A.; HENNESSY, John L. Computer Organization and Design: The *Hardware/Software* Interface - 4th edition. San Francisco. Morgan Kaufman.2011

RONSZCKA, Adriano Francisco. Como Programar com o Paradigma



Orientado a Notificações. Estudos Especiais – PON, Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná, 2011.

SCOTT, M. L. Programming Language Pragmatics. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 2000.

SIMÃO, J. M. “A Contribution to the Development of a HMS Simulation Tool and Proposition of a Meta-Model for Holonic Control”. Tese de Doutorado, CPGEI/UTFPR, 2005 – Disponível em:  
[http://arquivos.cpgei.ct.utfpr.edu.br/Ano\\_2005/teses/Tese\\_012\\_2005.pdf](http://arquivos.cpgei.ct.utfpr.edu.br/Ano_2005/teses/Tese_012_2005.pdf).

SIMÃO, J. M. “Proposta de uma Arquitetura de Controle para Sistemas Flexíveis de Manufatura Baseada em Regras e Agentes”. Dissertação de Mestrado, CPGEI/UTFPR, Curitiba, 2001.

SIMÃO, J. M., TACLA, C. A., STADZISZ, P. C., “Holonic Control Meta-Model”. IEEE Transactions on Systems, Man and Cybernetics. Part A, Systems and Humans, 2009.

SIMÃO, J. M.; STADZISZ, P. C. “Inference Process Based on Notifications: The Kernel of a Holonic Inference Meta-Model Applied to Control Issues”. IEEE Transactions on Systems, Man and Cybernetics. Part A, Systems and Humans, Vol. 39, Issue 1, 238-250, Digital Object Identifier 10.1109/TSMCA.2008.20066371, 2009.

SIMÃO, J. M.; STADZISZ, P. C. “Paradigma Orientado a Notificações (PON) – Uma Técnica de Composição e Execução de Software Orientado a Notificações”. Pedido de Patente submetida ao INPI/Brazil (Instituto Nacional de Propriedade Industrial) em 2008 e a Agência de Inovação/UTFPR em 2007. Nº INPI Provisório 015080004262. Nº INPI Efetivo PI0805518-1. Patente submetida ao INPI. Brasil, 2008.

SIMÃO, J. M.; STADZISZ, P. C. An Agent-Oriented Inference Engine applied for Supervisory Control of Automated Manufacturing Systems. In: J. Abe, & J. Silva Filho, Advances in Logic, Artificial Intelligence and Robotics (Vol. 85, pp. 234-241). Amsterdam, The Netherlands: IOS Press Books, 2002.

SIMÃO, J. M; STADZISZ, P. C, Mecanismo de Resolução de Conflito e Garantia de Determinismo para o Paradigma Orientado a Notificações (PON). Patente pendente submetida ao INPI/Brazil (Instituto Nacional de Propriedade Industrial) em 02/2010 e Agência de Inovação da UTFPR em 2009b. Número INPI: PI1000296-0.

SIMÃO, J. M; TACLA, C. A; BANASZEWSKI R. F; STADZISZ, P. C. Mecanismo de Inferência Otimizado do Paradigma Orientado a Notificações (PON) e Mecanismos de Resolução de Conflitos para Ambientes Monoprocessados e Multiprocessados Aplicados ao PON. Patente pendente submetida ao INPI/Brazil (Instituto Nacional de Propriedade Industrial) em 03/2010 e Agência de Inovação da UTFPR em 2010. Número INPI: PI1003736-5.

SIMÃO, J.M., STADZISZ, P.C., KUNZLE, L. Rule and Agent-oriented Architecture to

Discrete Control Applied as Petri Net Player. (G. Torres, J. Abe, M. Mucheroni, & C. P.E., Eds.) 4th Congress of Logic Applied to Technology - LAPTEC 2003, 101, p. 217, 2003.

SIMÃO, J.M., STADZISZ, P.C., TACLA, C. A., BANASZEWSKI, R. F. "Notification Oriented Paradigm (NOP) and Imperative Paradigm: A Comparative Study". A Journal of Software Engineering and Applications, JSEA (V. 5 N. 6, 2012, pp. 402-416) doi:10.4236/jsea.2012.56047. 2012a.  
<http://www.scirp.org/journal/PaperInformation.aspx?paperID=19842#abstract>.

SIMÃO, J.M.; BELMONTE, L.; VALENÇA, G. .Z; LINHARES, R. R.; BANASZEWSKI, R. F.; FABRO, J. A.; TACLA, C. A.; STADZISZ, P.C.; RONSZCKA, A. F.; BATISTA, V. "Notification Oriented and Object Oriented Paradigms comparison via Sale System". A Journal of Software Engineering and Applications, 2012b - Accepted Paper - July 2012.

SIMÃO, J.M.; BELMONTE, L.; VALENÇA, G. .Z; BATISTA, V.; LINHARES, R. R.; BANASZEWSKI, R. F.; FABRO, J. A.; TACLA, C. A.; STADZISZ, P.C.; RONSZCKA, A. F. "A Game Comparative Study: Object-Oriented Paradigm and Notification-Oriented Paradigm". A Journal of Software Engineering and Applications, 2012c

SIMÃO, Jean M., *et al*, "Notification Oriented Paradigm (NOP) and Imperative Paradigm: A Comparative Study", Journal of Software Engineering and Applications (JSEA) – Accepted Paper – Publication foresaw to JSEA 5.6 in June 2012d.

SIMSIE, L. Using programmable logic for embedded systems. Technical report, Altera Corporation, 2003. Disponível em:  
 <[http://whitepapers.osdn.com/detail/RES/1074018954\\_768.html](http://whitepapers.osdn.com/detail/RES/1074018954_768.html)>. Acesso em: 9 out. 2007.

TANENBAUM, Andrew S. Organização Estruturada de Computadores. Rio de Janeiro. LTC Editora, 3. Ed. 1990.

VALENÇA, G. Z. ; BANASZEWSKI, R. F. ; Ronszcka, A. F. ; Batista, M. V. ; Linhares, R. R. ; Fabro J. A. ; STADZISZ, P. C. ; SIMÃO, J. M. . Framework PON, Avanços e Comparações. In: III Simpósio de Computação Aplicada, 2011, Passo Fundo - RS. III Simpósio de Computação Aplicada, 2011

WIECHETECK, Luciana Vilas Boas. Método para projeto de software usando o paradigma orientado a notificações – PON. 2011. 196 f. Dissertação (Mestrado em Engenharia Elétrica e Informática Industrial) – Universidade Tecnológica Federal do Paraná, Curitiba, 2011.

WITT, F. A. Comparação de Desempenho de Aplicação em Paradigma Orientado a Objetos (POO) e em Paradigma Orientado a Notificações (PON) para a Implementação de um Controle Discreto em Lógica Reconfigurável. Relatório interno restrito de Atividade de Iniciação Científica do PIBIC/UTFPR. Curitiba – PR Brazil, Agosto 2011 (Orientador: Prof. Dr. J. M. Simão, Co-orientador, Prof. Dr. C. R. Erig Lima, Colaboradores: R. R. Linhares, P. C. Stadzisz,).

WITT, F. A; LINHARES, R. R; SIMÃO, J. M; STADZISZ, P. C; LIMA, C. R. E. Implementação do Paradigma Orientado a Notificações em Hardware. Relatório interno restrito para o curso de Lógica Reconfigurável do CPGEI/UTFPR. Curitiba – PR Brazil, Dezembro de 2010 (Prof. Dr. C. R. Erig Lima).

WOLF, Wayne Computers as Components: Principles of Embedded Computing System Design. Morgan Kaufmann. 2001. 662 páginas. ISBN 155860541X

WOLF, Wayne. Embedded computing - What is embedded computing? IEEE Computer, volume 35, no. 9, p.136–137, 2002.

## APÊNDICES

APÊNDICE A – Código exemplo de uma aplicação PON utilizando coprocessador.

```

/***** copon_memory_map.h *****/

#ifndef __COPON_MEMORY_MAP_H_
#define __COPON_MEMORY_MAP_H_

#define NUMBER_OF_ATTRIBUTES 4
#define NUMBER_OF_PREMISES 4
#define NUMBER_OF_CONDITIONS 4
#define MAXIMUM_APPROVED_RULES 16

#define REGISTERS_COUNT (2 + NUMBER_OF_ATTRIBUTES +
NUMBER_OF_PREMISES * 2 + NUMBER_OF_CONDITIONS * 2 + 2)

#define STATUS 0
#define CONFIG_ADDRESS 1
#define ATTR_BASE_ADDRESS 2
#define PREMISES_BASE_ADDRESS (ATTR_BASE_ADDRESS +
NUMBER_OF_ATTRIBUTES)
#define CONDITIONS_BASE_ADDRESS (PREMISES_BASE_ADDRESS +
NUMBER_OF_PREMISES * 2)
#define RULE_ADDRESS (CONDITIONS_BASE_ADDRESS +
NUMBER_OF_CONDITIONS * 2)

#define write_register(register, data)
    IOWR(PON_COPROCESSOR_0_BASE, register, data)
#define read_register(register)
    IORD(PON_COPROCESSOR_0_BASE, register)

#endif

/***** gate.cpp *****/

#include "../Include/IGate.h"
#include <iostream>
#include "system.h"
#include "io.h"
#include "../Include/copon_memory_map.h"

using namespace std;

/* GateStatus:

```

```
0: Open
2: Close */
```

```
Gate::Gate(){
    GateStatus = read_register(ATTR_BASE_ADDRESS + 1); //new
Integer(this, 2);
    mtOpenGate = new MethodPointer<Gate>(this,&Gate::OpenGate);
    mtCloseGate = new MethodPointer<Gate>(this,&Gate::CloseGate);
}
```

```
void Gate::OpenGate(){
    write_register(ATTR_BASE_ADDRESS + 1, 0); // GateStatus = 0
    cout << "OPEN" << endl;
}
```

```
void Gate::CloseGate(){
    write_register(ATTR_BASE_ADDRESS + 1, 2); // GateStatus = 0
    cout << "CLOSE" << endl;
}
```

```
/****** remote_control.cpp *****/
```

```
#include "../Include/IRemoteControl.h"
#include <stdio.h>
```

```
#include "system.h"
#include "io.h"
#include "../Include/coapon_memory_map.h"
```

```
RemoteControl::RemoteControl(){
    mtPressButton = new
MethodPointer<RemoteControl>(this,&RemoteControl::PressButton);
    mtReleaseButton = new
MethodPointer<RemoteControl>(this,&RemoteControl::ReleaseButton);
}
```

```
void RemoteControl::PressButton(){
    write_register(ATTR_BASE_ADDRESS, 1);
    printf("PressButton\n");
}
```

```
void RemoteControl::ReleaseButton(){
    write_register(ATTR_BASE_ADDRESS, 0);
    printf("ReleaseButton\n");
}
```

```
/****** simple_application.cpp *****/
```

```
#include <stdio.h>
```

```

#include "system.h"
#include <stddef.h>
#include <iostream>
#include <unistd.h>
#include "../Include/printer_helper.h"
#include "altera_avalon_pio_regs.h"
#include "../Include/ISimpleApplication.h"
#include "../Include/coapon_memory_map.h"
#include "altera_avalon_performance_counter.h"

using namespace std;

SimpleApplication::SimpleApplication(int
schedulerStrategy):Application(schedulerStrategy){
    startApplication();
}
Rule* rule1;
Rule* rule2;

void SimpleApplication::initRules(){

    // Criação dos atributos do sistema, com seu valor inicial
    Attribute* remoteControlButton = new
Attribute(RemoteControl::RELEASED);
    Attribute* gateStatus = new Attribute(Gate::OPENED);

    // Criação das premissas do sistema, cada uma recebendo como
parâmetro o atributo que será comparado,
    // o valor usado para comparação e a operação que será
realizada
    Premise* p1 = new
Premise(remoteControlButton,RemoteControl::PRESSED,Premise::EQUAL);
    Premise* p2 = new
Premise(remoteControlButton,RemoteControl::RELEASED,Premise::EQUAL);
    Premise* p3 = new
Premise(gateStatus,Gate::OPENED,Premise::EQUAL);
    Premise* p4 = new
Premise(gateStatus,Gate::CLOSED,Premise::EQUAL);

    // Criação das Condições, referenciando as premissas que
deverão estar ativas para que sejam aprovadas
    Condition* cond1 = new Condition();
        cond1->addPremise(p1);
        cond1->addPremise(p4);
    Condition* cond2 = new Condition();
        cond2->addPremise(p1);
        cond2->addPremise(p3);

    // Criação das Ações, que serão executadas pelas regras

```

```

        Action* action1 = new Action();
        action1->addInstigation(new Instigation(remoteControl-
>mtReleaseButton));
        action1->addInstigation(new Instigation(gate-
>mtOpenGate));
        Action* action2 = new Action();
        action2->addInstigation(new Instigation(remoteControl-
>mtReleaseButton));
        action2->addInstigation(new Instigation(gate-
>mtCloseGate));

        // Criação das regras, cada uma indicando a condição que deverá
estar ativa para sua execução
        // e a ação que será executada quando da sua aprovação
        rule1 = new Rule("Open gate", scheduler, cond1, action1,
false);
        rule2 = new Rule("Close gate", scheduler, cond2, action2,
false);

        // Inicia a execução do coprocessador
write_register(CONFIG_ADDRESS, 1);
}

void SimpleApplication::initFactBase(){
    gate = new Gate();
    remoteControl = new RemoteControl();
}

void SimpleApplication::initSharedPremises(){
}

void SimpleApplication::configureStartApplicationAction(){
}

void SimpleApplication::codeApplication(){
    int c=0;
    Rule *objpointer;

    while (c!=-1){
        cout << "Gate initial status: " << gate->GateStatus <<
endl;
        cout << "Press button? (1): ";
        cin >> c;
        if (c==1){
            remoteControl->PressButton();
            remoteControl->ReleaseButton();
            if (read_register(STATUS) == 1) {
                unsigned int rule =
read_register(RULE_ADDRESS);

```

```
        cout << "Rule: " << rule << endl;
        objpointer = (Rule*)rule;
        objpointer->approved();
    }
}

#include "sys/alt_irq.h"
#include "alt_types.h"

int main() {
    SimpleApplication simpleApplication(SchedulerStrategy::NO_ONE);
    return 0;
}
```