# Paradigma Orientado a Notificações para a Síntese de Lógica Reconfigurável

Ricardo Kerschbaumer[1,3,4], Jean M. Simão[1,2,3], Robson R. Linhares[1,2,3], Paulo C. Stadzisz[1,2,3], Carlos R. Erig Lima[1,2,3]

1 – Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI/UTFPR)
2 – Programa de Pós-Graduação em Computação Aplicada (PPGCA, DAINF - DAELN, UTFPR)
3 – Universidade Tecnológica Federal do Paraná (UTFPR) – Curitiba – PR, Brasil (41 33104760)
4 – Instituto Federal Catarinense (IFC) – Luzerna – SC, Brasil
ricardo@luzerna.ifc.edu.br, {jeansimao, linhares, stadzisz, erig}@utfpr.edu.br

***Resumo* - A demanda por capacidade de processamento é cada vez maior. Limitações técnicas impedem que a frequência de *clock* dos processadores atuais continuam aumentando para atender esta demanda. Algumas das alternativas mais utilizadas para contornar estas limitações são aumentar a eficiência dos programas, com novas ferramentas de programação e utilizar técnicas de processamento paralelo, seja em processadores com vários núcleos, em sistemas distribuídos ou em *hardware* digital. O Paradigma Orientado a Notificações (PON) vem apresentado resultados positivos, reduzindo o impacto de problemas como a redundância temporal e estrutural presente nos principais paradigmas de programação. Por outro lado, aplicações desenvolvidas em *hardware* digital, *FPGAs* (Field Programmable gate array) por exemplo já permitem fisicamente o paralelismo efetivo, o que lhes confere excelente desempenho. Mas o processo de desenvolvimento destas aplicações não é nada intuitivo, exigindo grande habilidade do desenvolvedor. Neste contexto, este artigo apresenta um estudo de caso onde são utilizadas as premissas do PON no desenvolvimento de uma aplicação em *hardware* digital, mais especificamente em uma *FPGA*. A aplicação escolhida para implementação foi um ordenador paralelo, cujo objetivo é receber um vetor de números inteiros e ordená-los. A título de comparação foram escolhidas duas técnicas de ordenação, cada uma delas implementada segundo as premissas do PON, em VHDL diretamente e em VHDL via máquina de estados, totalizando assim seis implementações. Não é objetivo deste artigo discutir as características do PON e suas implicações e sim, aplicá-lo no desenvolvimento de *hardware* digital de forma a tornar o desenvolvimento nesta plataforma mais fácil e intuitivo. Os resultados obtidos foram positivos, pois o PON possibilitou que se desenvolvesse as aplicações com grande facilidade, sem comprometer significativamente o desempenho em relação às outras técnicas aplicadas.**

***Palavras Chaves – Paradigma Orientado a Notificações; Projeto de Hardware Digital; Paralelismo; FPGA.***

## I. INTRODUÇÃO

Os principais paradigmas de programação atuais, Paradigma Imperativo (PI) e Paradigma Declarativo (PD) apresentam limitações no que diz respeito a acompanhar os avanços tecnológicos das plataformas de *hardware* correntes, principalmente quando se fala em paralelismo [1]. As principais técnicas de programação utilizadas atualmente, inclusive em sistemas de inteligência computacional, tais como, Paradigma Orientado a Objetos (POO) do PI e Sistemas Baseados em Regras (SBR) do PD, levam a um forte acoplamento de expressões causais e a redundâncias decorrentes de seus processos de avaliação mesmo quando técnicas de orientação a eventos e a dados são usadas [2][3][7]. Assim, as dificuldades encontradas atualmente no desenvolvimento de *softwares* que trabalhem de forma paralela ou distribuída, seja no desenvolvimento de sistemas de Inteligência Computacional ou em qualquer outra área, motivam a busca por alternativas que eliminem ou reduzam estas desvantagens [4][5][13].

Diferente dos paradigmas atuais, o Paradigma Orientado a Notificações (PON) vem mostrando bons resultados na tentativa de minimizar as redundâncias e o acoplamento no desenvolvimento de *software*. O PON surgiu de uma teoria de controle discreto e inferência, sendo concebido para reduzir ou eliminar algumas das deficiências dos atuais paradigmas, principalmente no que diz respeito a avaliações causais desnecessárias e acopladas. Isso acontece porque o PON evita o processo de inferência (cálculo lógico-causal) monolítica utilizando um mecanismo baseado na relação entre entidades computacionais notificantes [2][3].

As implementações apresentadas neste artigo utilizam os princípios do PON para desenvolver *hardware* digital, de forma a comparar suas características de desempenho e facilidade de implementação com formas tradicionais de desenvolver tais aplicações.

Na próxima seção será apresentado com mais detalhes o PON. Na seção seguinte serão apresentadas as implementações realizadas e na sequência os resultados obtidos e as conclusões.

## II. PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON)

O PON é uma proposta para uma nova técnica de desenvolvimento computacional. Esta nova solução possui características que a tornam capaz de lidar com algumas das limitações dos principais paradigmas atuais, i.e. o Imperativo (PI) e o Declarativo (PD) [2][3].

Estes paradigmas, PI e PD, utilizam-se de mecanismos monolíticos de inferência (implícitos ou explícitos), normalmente realizando buscas sobre elementos passivos a fim

Desenvolvido com o Apoio da CAPES
CAPES - *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior*.

de realizar avaliações lógico-causais. No caso do PI as avaliações lógico-causais são realizadas testando continuamente variáveis ou estruturas de dados o que gera redundâncias temporais, pois estas variáveis são testadas mesmo que seu valor não tenha sido alterado. Também são geradas redundâncias estruturais, pois podem ocorrer os mesmos testes sobre as mesmas variáveis em diferentes partes do programa [2].

Já os sistemas de inferência implementados nos sistemas baseados em regras (RETE, HAL etc.), são baseados em estruturas pesadas de dados que geram sobrecarga de processamento e mesmo algum nível de redundâncias [8][14].

O PON propõe um processo de inferência baseado em entidades ativas que interagem através de notificações precisas e diretas. Desta forma, o grande acoplamento de código e as redundâncias presentes nos paradigmas atuais não se apresentam no PON [3]. As entidades propostas para o PON são desacopladas por definição, o que também facilita sua execução em paralelo [2][3][7].

### *A. Estrutura e processo de inferência do PON*

Assim como nos sistemas baseados em regras, no PON as expressões causais são representadas por simples regras causais. Porém no PON cada regra é tratada como uma entidade computacional chamada *Rule*.

O PON também possui elementos que são avaliados, por notificações, pelas *Rules*. Estes elementos são chamados *Fact_Base_Element* (*FBE*). Uma *FBE* é composta por atributos. Cada atributo é representado e tratado no PON por uma entidade do tipo *Attribute*. Cada *Rule*, por sua vez, é constituída por dois tipos de entidades, chamadas de *Condition* e *Action* respectivamente. Estruturalmente a *Condition* e a *Action* trabalham juntas para compor o conhecimento causal da regra. A parte responsável pela tomada de decisões é a *Condition*, enquanto a *Action* é responsável por executar as tarefas *Rule*.

O processo de análise dos estados dos *Attribute*s é um processo de inferência realizado pelas *Conditions* existentes nas *Rules*. Neste processo, as *Conditions* têm a colaboração de entidades do tipo *Premises*. Quando o cálculo lógica da *Condition* sobre cada uma das suas *Premises* é inferida como verdadeira, esta *Condition* se torna verdadeira e a regra a qual ela pertence ativa sua *Action*. As *Actions* são compostas por *Instigations*. Estas *Instigations* por sua vez são responsáveis por chamar o *Method*. Estes *Methods* são responsáveis por executar os serviços da *FBE*. Normalmente a chamada de um método causa a alteração de algum atributo, alimentando assim o processo de inferência.

É a colaboração ativa de entidades realizada por meio de notificações diretas que torna o processo de inferência do PON inovador [2]. Cada *Premise* representa um valor booleano obtido a partir do estado de um ou dois *Attributes* e é composta por: (a) uma referência para o valor de um *Attribute*, chamado de *Reference*, que é recebido através de uma notificação; (b) um operador lógico, chamado de *Operator*, útil para realizar comparações lógicas; e (c) outro valor chamado *Value* que pode ser uma constante ou ainda uma referência ao valor de outro *Attribute*, neste caso também recebido por notificação [3].

Uma *Premise* também colabora com as *Conditions*, pois quando seu valor se torna verdadeiro as *Conditions* interessadas são notificadas. Assim, cada *Condition* notificada calcula seu valor booleano pela conjunção dos valores de todas as *Premises* que ela tem interesse [3].

Quando *Premises* que integram uma dada *Condition* são satisfeitas, à luz do seu conectivo lógico, esta *Condition* é ela mesma satisfeita, e notifica a respetiva *Rule* para que potencialmente seja executada. A colaboração entre as entidades do PON por meio de notificações pode ser observada no esquema ilustrado na Fig. 1. Neste esquema, o fluxo de notificações é simbolizado pelas setas ligadas a retângulos que, por sua vez, simbolizam as entidades do PON [7].

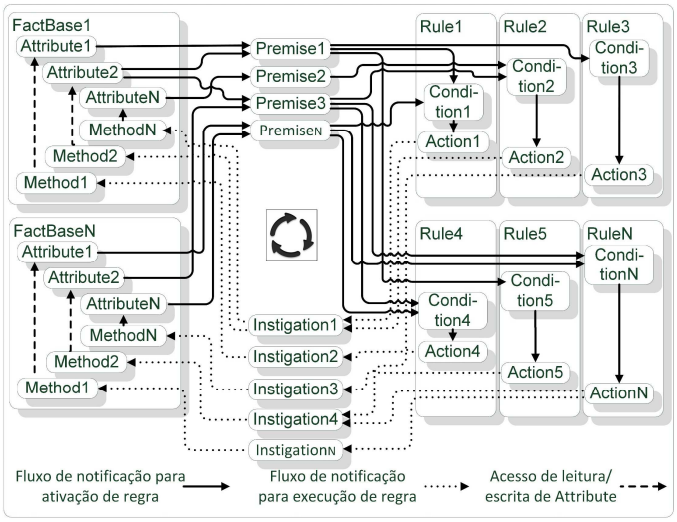


Fig. 1. Cadeia de notificações das regras. [7]

### *B. A Natureza do PON*

As redundâncias temporais são resolvidas no PON, eliminando pesquisas sobre elementos passivos. Isso é possível porque os *Attribute*s são capazes de notificar pontualmente apenas as *Premises* que estão interessadas em seu estado e isso só acontece quando esse estado é alterado. Isso evita avaliações e reavaliações desnecessárias [9].

De forma parecida as redundâncias estruturais são resolvidas no PON quando uma *Premise* é compartilhada entre duas ou mais *Conditions*. Assim, cada *Premise* realiza o cálculo lógico apenas uma vez e compartilha seu resultado com todas as *Conditions* interessadas, evitando reavaliações.

Com estas características o PON evita as redundâncias temporais e estruturais que são verificadas no PI e até mesmo em PD proporcionando uma melhora no desempenho [9]. Ainda, observando as características do PON pode-se notar que ele também permite o "desacoplamento" (ou acoplamento mínimo) de suas entidades. Estas entidades ativas e desacopladas são interessantes para implementação de *softwares* [7]. Assim é possível desenvolver softwares mais otimizados e fáceis de distribuir [2][3]. As aplicações desenvolvidas a partir do PON tendem a ter melhor desempenho economizando recursos de processamento [9].

Neste sentido, o PON é aplicável também ao desenvolvimento do *software* para ambientes *multi-core* por permitir de maneira menos complicada a distribuição fina de processamento. Ao bem da verdade, potencialmente, ele é aplicável no desenvolvimento de aplicações não só paralelas,

mas também distribuídas, devido à natureza desacoplada de seus elementos [9][7].

De fato, em termos de inferência, não importa se uma entidade é notificada na mesma região de memória, no mesmo computador ou na mesma sub-rede. Uma entidade notificadora (por exemplo, um *Attribute*) pode ser executada em uma máquina ou processador enquanto a entidade "cliente" (por exemplo, uma *Premise*) pode ser executada em outro. Para o notificador, seria “apenas” necessário saber o endereço da entidade cliente [9].

### C. Plataforma de Execução do PON

O PON vem sendo implementado em várias plataformas. Inicialmente foi desenvolvido um *framework* em C++. Neste *framework* as entidades do PON são implementadas através de objetos [11]. Existem também esforços para desenvolver um compilador PON que é capaz de gerar código específico de PON em C ou C++ a partir de uma linguagem PON de alto nível [10]. De forma geral estas abordagens geram código assaz otimizado, mas ainda sequencial para implementar o processo de notificações do PON [10]. Na sequência, esta tecnologia irá incorporar a questão de processamento paralelo para plataformas *multi-core* e distribuídas.

Entretanto, as abordagens realizadas puramente em *software*, mesmo em ambiente com certo paralelismo, não se aproximam tanto do modelo dinâmico teórico do PON quanto as abordagens que se utilizam de implementações em *hardware*. Mas mesmo as abordagens em *hardware* apresentadas dependem de algumas soluções de projeto e implementação que permitam explorar as propriedades do PON.

Uma forma de se implementar aplicações que se aproximem ainda mais do modelo dinâmico teórico do PON é desenvolver estas aplicações diretamente com componentes de *hardware* digital. Uma implementação deste gênero que esta sendo desenvolvida é chamada de PON HD (Hardware Digital) [6].

Mais detalhadamente, no PON HD cada elemento do PON tem sua implementação em *hardware*. Um *FBE* agrega os *Attributes* e *Methods*, e dessa forma este bloco é simplesmente utilizado para fins de organização na estrutura do projeto. Os *Methods* no PON HD apenas alteram o valor de determinado *Attribute* após receber uma notificação. Os *Attributes* são responsáveis por armazenar um determinado valor e notificar a *Premise* a respeito dessa mudança. Assim no PON HD os *Attributes* são implementados por meio de registradores construídos a partir de simples *flip-flops*. O *hardware* é projetado de tal forma que mudanças nos *Atributes* são notificadas as *Premises* [6]. Com a mudança no estado de determinado *Attribute*, as *Premises* devem executar o calculo lógico sobre determinada expressão causal relacionada a ele. Dessa forma, interpretou-se as *Premises* em *hardware* como blocos comparadores assíncronos [6].

Uma *Condition*, por sua vez é composta por uma ou várias *Premises*. Com a mudança no estado lógico das *Premises* relacionadas, cada *Condition* deve calcular o seu valor lógico baseado em expressões lógicas sobre as *Premises*. Dessa forma, contemplou-se que as entidades de circuito assíncrono que mais representam esse comportamento são as portas lógicas E e OU [6]. No PON HD, uma *Rule* deve basicamente analisar o estado lógico da *Condition* que a compõe. Caso o valor desta seja verdadeiro, ela deve ativar a *Action*, assim no PON HD uma *Rule* é apenas uma conexão entre a saída da *Condition* e a entrada do *Action*, com a função adicional de evitar que a *Action* fique ativa por mais de um ciclo de *clock* [6].

Visto que o modelo inicial PON HD não aborda questões como a implementação de um mecanismo de determinismo nem técnicas de sincronização de regras executadas em paralelo, uma *Action* basicamente é responsável por repassar o sinal de execução proveniente de uma *Rule* para as *Instigations* [6]. Uma *Instigation* repassa o sinal de execução de uma ou mais *Actions* para um *Method*. Assim, este bloco foi modelado como uma porta lógica OU. Como foi mencionado o *Method* altera os *Attributes* reiniciando assim o ciclo de notificações [6].

O PON HD atual já é capaz de aproveitar completamente o paralelismo de execução das entidades PON. Esta abordagem tem a vantagem de permitir ao desenvolvedor utilizar todo o potencial do PON [7]. O estudo de caso apresentado neste artigo se utiliza desta abordagem, implementando toda a cadeia de notificações do PON em *hardware* digital utilizando VHDL.

Outrossim, além do PON HD em si, no que diz respeito a implementações em *hardware*, tem-se uma abordagem onde o PON é desenvolvido parte em *software* e parte em *hardware*. Nesta implementação foi criado um co-processador, usando uma versão do PON HD, que é responsável pelas avaliações logico-causais. O restante do processamento é realizado por um núcleo Von Neumann. Esta arquitetura de co-processador PON e núcleo von Neumann foi implementada em lógica reconfigurável e reduziu significativamente o número de ciclos de *clock* necessários para executar as tarefas [12].

Ainda com relação a implementações do PON em *hardware*, foi desenvolvida uma arquitetura de computador específica para o PON, chamada NOCA (*Notification Oriented Computer Architecture*). Nesta arquitetura as entidades do PON são executadas paralelamente, a partir de instruções lidas de um *software* de baixo nível [7]. A NOCA foi concebida para executar *software* PON de qualquer nível de complexidade a partir da memória. Esta arquitetura é organizada como um multiprocessador de granularidade fina executando instruções de forma hierárquica por meio de conjuntos de núcleos especializados. Um protótipo desta arquitetura foi implementado e apresentou ganhos efetivos em avaliações comparativas de desempenho [7].

## III. IMPLEMENTAÇÕES DO ORDENADOR

Para que se possa comparar mais efetivamente o processo de desenvolvimento do ordenador paralelo em *hardware* digital desenvolvido segundo o PON com as formas tradicionais de implementação, foram selecionados dois algoritmos. Estes algoritmos serão chamados de ordenador 1 e ordenador 2. Cada um deles foi desenvolvido em VHDL, mas utilizando diferentes abordagens. As abordagens utilizadas foram: segundo o PON HD, na qual os preceitos do PON foram utilizados; VHDL puro, na qual os ordenadores foram desenvolvidos sem seguir metodologias auxiliares somente codificação VHDL tradicional; e máquina de estados, onde a codificação dos ordenadores aconteceu seguindo o mecanismo de uma máquina de estados.

O objetivo das implementações foi comparar as diferentes formas de se elaborar o código VHDL, suas vantagens e

desvantagens e não necessariamente encontrar o melhor algoritmo de ordenação. Isto dito, foram realizadas seis implementações nas quais um conjunto de 110 valores inteiros de 16 bits foi ordenado em *hardware*.

### A. *Implementação do ordenador 1*

O primeiro algoritmo de ordenação opera completamente em paralelo. Todos os elementos do vetor são comparados com seus vizinhos e as trocas necessárias para a ordenação são realizadas no mesmo ciclo. Porém, neste algoritmo é necessário tomar um cuidado especial, um elemento não pode ser trocado com seus dois vizinhos ao mesmo tempo, o que geraria um problema de concorrência. Assim foi implementado um circuito de bloqueio que evita este problema. A Fig. 2 apresenta esta topologia.

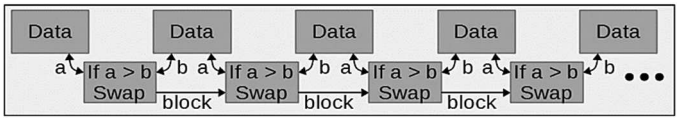


Fig. 2. Topologia do ordenador 1.

A codificação em VHDL puro foi realizada de forma a gerar um circuito digital que utiliza comparadores para realizar as comparações de forma paralela e sinais e variáveis para bloquear as trocas proibidas. A Fig. 3 apresenta o principal trecho do código desta implementação.

```
t_ended:='1';
-- Ordenação
FOR i IN 0 to N-1 LOOP
   if(ram(i)>ram(i+1) and swap='0') THEN
      swap:='1';
      ram(i+1) <= ram(i);
      ram(i) <= ram(i+1);
      t_ended:='0';
   ELSE
      swap:='0';
   END IF;
END LOOP;
-- Fin ordenação
IF t_ended='1' THEN
   ended<='1';
ELSE
   ended<='0';
END IF;
```

Fig. 3. Trecho do código VHDL do ordenador 1

Na codificação utilizando máquina de estados, por sua vez, foram criados três estados, conforme Fig. 4.

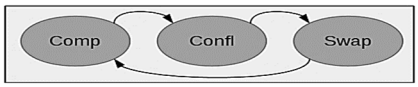


Fig. 4. Máquina de estados do ordenador 1.

No estado "comp" todos os elementos são comparados com seus vizinhos. No estado "confl" são verificados os conflitos nas trocas necessárias e no estado "swap" as trocas não conflitantes são realizadas. Cada um dos três estados é realizado em um único ciclo de *clock*.

Finalmente na implementação PON cada um dos elementos, que constituem este paradigma, foi implementado no código em VHDL.

Como pode ser verificado na Fig. 5, o *FBE* é composto pelos *Attributes* que armazenam os valores a ser ordenados, indicadores de que houve mudança no valor e o *Method* que realiza a troca dos dados.

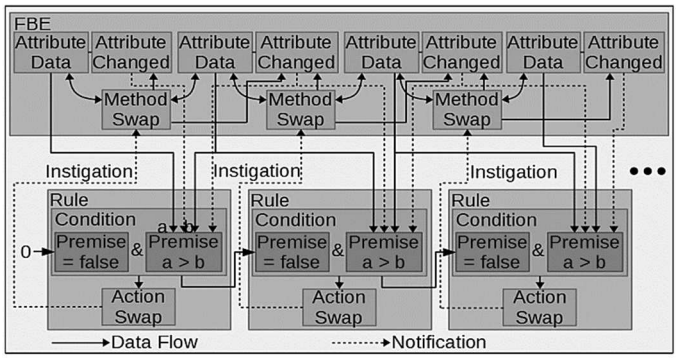


Fig. 5. Implementação PON do ordenador 1.

As *Rules* por sua vez são compostas cada uma por uma única *Condition*, que realiza a função lógica "E" entre duas *Premises* para executar a ação de troca dos valores. Estas duas *Premises* são: (1) o par anterior não pode ter sido trocado e (2) o elemento a esquerda dever ser maior que o elemento da direita. É importante salientar que todas as *Rules* operam em paralelo, realizando todas as comparações ao mesmo tempo.

```
FOR i IN 0 to N-1 LOOP -- Inicio PON
   if(changed(i)='1' or changed(i+1)='1') THEN -- Rule
      IF(ram(i)>ram(i+1) ) THEN -- premise A > B
         premiseAgB(i):='1';
      ELSE
         premiseAgB(i):='0';
      END IF;
      IF(swapped='1') THEN -- premise previous swapped
         premisePSwapped(i):='1';
      ELSE
         premisePSwapped(i):='0';
      END IF;-- Condition
      IF(premisePSwapped(i)='0' and premiseAgB(i)='1') THEN
         actionSwap(i):='1';
      ELSE
         actionSwap(i):='0';
      END IF;IF(actionSwap(i)='1') THEN -- method Swap
         ram(i+1) <= ram(i);
         ram(i) <= ram(i+1);
         changed(i) <= '1';
         changed(i+1) <= '1';
         t_ended:='0';
      ELSE
         IF (swapped='0') THEN
            changed(i) <= '0';
         END IF;
         IF (i=(N-1)) THEN
            changed(N) <= '0';
         END IF;
      END IF;
   ELSE
      swapped:='0';
   END IF;
END LOOP;
```

Fig. 6. Trecho de código da implementação em PON HD.

Nesta implementação a instigação faz apenas a conexão entre a ação e o método. A Fig. 6 apresenta o principal trecho do código VHDL desta implementação.

As três implementações realizam corretamente a ordenação dos dados, porém cada uma delas teve diferente dificuldade de implementação, utilizou diferentes quantidades de recursos da *FPGA* e teve diferentes características de desempenho.

### B. *Implementação do ordenador 2*

O segundo algoritmo de ordenação opera comparando e trocando os elementos ímpares do vetor em um ciclo e os elementos pares em outro ciclo. Assim todos os elementos do vetor são comparados. Porém, como a comparação é dividida em

pares e ímpares, não acontecem conflitos nas operações de troca. A Fig. 7 apresenta a topologia utilizada nesta implementação. Nesta implementação foi necessário um circuito chaveador para alternar entre os elementos pares e ímpares.

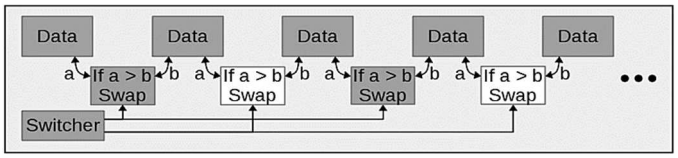


Fig. 7. Topologia do ordenador 2.

Para a codificação em VHDL puro foi utilizado um sinal que indica se são os elementos pares ou ímpares que estão sendo avaliados, bem como comparadores para comparar os valores dos dados e decidir se a troca é necessária ou não.

Na codificação utilizando máquina de estados, ao seu turno, foram criados quatro estados, conforme Fig. 8. No estado "comp1" os elementos pares são comparados com seu vizinho da direita para determinar se a troca é necessária. No estado "swap1" são trocados os elementos pares que se encontram fora do lugar. No estado "comp2" os elementos ímpares são comparados com seu vizinho da direita para determinar se a troca é necessária. No estado "swap2" são trocados os elementos ímpares que se encontram fora do lugar. Cada um dos quatro estados é realizado em um único ciclo de *clock*.

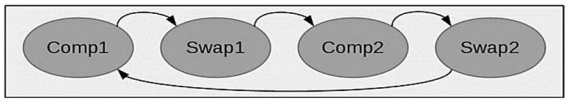


Fig. 8. Máquina de estados do ordenador 2.

Na implementação PON-HD, assim como no ordenador 1 cada um dos elementos, que constituem este paradigma, foi implementado no código. Como pode ser verificado na Fig. 9, os elementos que compõem o *FBE* são os seguintes: cada valor a ser ordenado é um *Attribute;* para cada valor tem-se também um *Attribute* que indica se este valor sofreu alteração; existe também um *Attribute* que indica qual o grupo de valores está sendo utilizado nas comparações, o grupo dos elementos pares ou o grupo dos elementos ímpares.

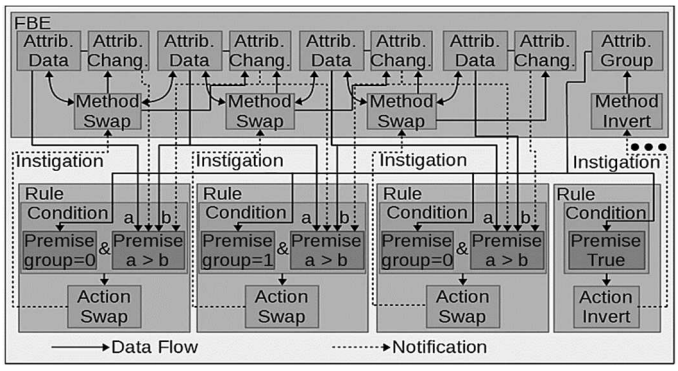


Fig. 9. A implementação PON do ordenador 2.

As *Rules* por sua vez são compostas cada uma por uma única condição, que realiza a função "E" entre duas *Premises* para executar a ação de troca dos valores. Estas duas *Premises* são o grupo ao qual o elemento pertence (par ou ímpar), que deve estar ativo, e o teste a respeito do elemento à esquerda ser maior que o elemento à direita. Nesta implementação a instigação faz apenas a conexão entre a ação e o método.

Assim como no ordenador 1, as três implementações realizam corretamente a ordenação dos dados, porém cada uma delas teve diferente dificuldade de implementação, utilizou diferentes quantidades de recursos da FPGA e teve diferentes características de desempenho.

## IV. RESULTADOS EXPERIMENTAIS

Foram utilizados vários conjuntos de dados para verificar o correto funcionamento de cada uma das implementações. O número de elementos do vetor de dados pode ser ajustado nos códigos conforme a necessidade, sendo limitado apenas pelo tamanho da *FPGA*. O *hardware* utilizado nos experimentos foi uma *FPGA* EP2C8Q208 da Altera, contendo 8256 elementos lógicos.

A distribuição inicial dos dados no vetor seguiu dois padrões. Um padrão utilizou dados com ordem inversa, ou seja, com ordenação contrária a desejada. O outro padrão utilizou vários grupos de dados distribuídos aleatoriamente. Os mesmos grupos de dados foram apresentados a cada uma das seis implementações para ordenação. A Tabela 1 apresenta os resultados obtidos com vetores contendo 110 elementos. Utilizou-se 110 elementos pois com esta quantidade a capacidade total da *FPGA* disponível chegou próxima de ser ultrapassada.

TABELA 1 RESULTADOS OBTIDOS NOS EXPERIMENTOS

| | Ordenador 1 | | | Ordenador 2 | | |
|---|---|---|---|---|---|---|
| | VHDL | M. S. | PON | VHDL | M. S. | PON |
| Ciclos de *clock*[1] | 110 | 330 | 110 | 110 | 221 | 110 |
| Ciclos de *clock*[2] | 121 | 363 | 121 | 109 | 219 | 109 |
| *Clock* máx. (MHz) | 27,8 | 41,5 | 16,2 | 57,2 | 55,4 | 50,1 |
| Funções Comb. | 6282 | 6073 | 6493 | 6780 | 6883 | 6220 |
| Registradores | 2263 | 2104 | 1990 | 2263 | 1997 | 1991 |
| Elementos Lógicos | 6460 | 6115 | 7061 | 6960 | 6893 | 6751 |

[1] Experimento com 110 elementos inversamente ordenados
[2] Experimento com 110 elementos aleatórios

Na Tabela 1 foram compilados os dados de dois conjuntos de experimentos, o primeiro com 110 elementos inversamente ordenados e o segundo com 110 elementos aleatórios. Para estes experimentos foram verificados os seguintes dados: número de ciclos de *clock* necessários para a ordenação; a máxima frequência de operação estimada do circuito; e o número de funções combinacionais, registradores e elementos lógicos utilizados na *FPGA*. Assim é possível comparar o desempenho e os recursos de *hardware* utilizados por cada implementação. Para confirmar o funcionamento das implementações e a fidelidade dos resultados, todas estas implementações foram sintetizadas e testadas em *hardware*, os resultados foram verificados utilizando-se a ferramenta *SignalTap II Logic Analyzer* da Altera, que permitiu monitorar os dados no interior da *FPGA*. Também foi desenvolvido um *software* simulador em C++ que permitiu verificar cada etapa do funcionamento de cada um dos algoritmos.

Observando os resultados é possível notar que nos dois algoritmos implementados as versões em PON-HD e em VHDL

puro utilizam o mesmo número de ciclos de *clock* para realizar a ordenação. Isto demonstra que mesmo que se tenha utilizado técnicas diferentes na concepção do código, o *hardware* inferido pelo compilador opera de forma semelhante nas duas implementações. A versão em máquina de estados (M. S.), por sua vez, utilizou um ciclo de *clock* para cada estado, o que resultou na utilização de um número maior de ciclos de *clock* para realizar a ordenação. Ainda com relação à máxima frequência de operação, a implementação em VHDL puro teve o melhor desempenho. Isto já era esperado pois, apesar da complexidade de implementação, não existem elementos extras como nas versões em PON-HD e máquina de estados para sobrecarregar o circuito.

Os recursos de *hardware* da *FPGA* utilizados nas implementações são bastante parecidos, com a versão em PON-HD utilizando menos registradores. É possível verificar que as *Rules* são executadas realmente em paralelo, pois só desta forma seria possível realizar as ordenações no número de ciclos de *clock* em que elas aconteceram.

Outro resultado muito importante e que não é facilmente mensurável é a dificuldade de implementação de cada uma das soluções. A implementação em VHDL pura é claramente a mais difícil de se realizar e depende fortemente da experiência e habilidade do desenvolvedor. As implementações em PON e máquina de estados, por utilizar mecanismos que ajudam a dividir o problema, são mais fáceis de realizar. Como cada elemento do PON exerce uma função simples e bem definida no processo, é mais simples construir cada um destes elementos, mesmo em VHDL. No PON o projetista tem a preocupação de criar as regras que determinam o comportamento do sistema. O código surge como consequência da decomposição e tradução dos elementos do PON.

## V. CONCLUSÃO

Os resultados mostram que a utilização do PON no desenvolvimento de aplicações em *hardware digital* é uma alternativa válida, uma vez que torna o processo de desenvolvimento mais fácil, sem exigir mais *hardware* que as demais alternativas experimentadas. É muito importante ressaltar que esta é a percepção desta equipe de desenvolvedores e é influenciada pela sua experiência e habilidade nas técnicas empregadas. Porém, os resultados obtidos em outros experimentos envolvendo o PON encontraram resultados parecidos, apoiando a ideia de que o PON facilita o desenvolvimento do *hardware* [11][12].

A utilização do PON implicou em uma redução na máxima frequência de operação, assim sua utilização em aplicações onde a frequência de *clock* é crítica se torna mais complexa. Porem como grande parte das aplicações desenvolvidas em hardware digital não opera nos limites de frequência da FPGA esta limitação não compromete os resultados obtidos.

O experimento também mostrou que a implementação em *hardware* digital dos elementos do PON permite a execução das *Rules* em paralelo e a avaliação de todas as *Rules* notificadas ao mesmo tempo, demonstrando assim os princípios de desacoplamento das entidades do PON.

Como trabalho futuro pode ser explorada a escalabilidade destas implementações em um *Hardware* que permita vetores ainda maiores de dados. Assim pode-se comparar vetores de diferentes tamanhos e verificar o comportamento do PON com relação a desempenho e utilização de recursos. Experimentos que verifiquem o consumo de energia das diferentes implementações também são importantes, pois como no PON apenas as entidades notificadas operam é possível que se tenha uma redução no consumo de energia em relação as outras implementações.

Existem ainda outros trabalhos em desenvolvimento, como por exemplo a geração automática de *hardware* digital utilizando PON. Onde o objetivo é gerar o código VHDL diretamente a partir de uma linguagem PON de alto nível. Considerando que o desenvolvimento de *hardware* digital é uma tarefa difícil, o uso do PON neste contexto se torna muito interessante.

## REFERÊNCIAS


[1] A.S. Tanenbaum, M. Van Steen. "Distributed Systems: Principles and Paradigms", Prentice Hall, 2002.

[2] J. M. Simão and P. C. Stadzisz, "Inference Based on Notifications: A Holonic Meta-Model Applied to Control Issues". IEEE Transactions on Systems, Man and Cybernetics, Part A. Vol. 39, Issue 1, Jan. 2009 Pg. 238-250

[3] J. M. Simão, P. C. Stadzisz, "Paradigma Orientado a Notificações (PON)—Uma Técnica de Composição e Execução de Software Orientada a Notificações". Patent pending submitted to INPI/Brazil in 2008 and UTFPR Innovation Agency 2007. INPI Number: PI0805518-1. http://www.patentesonline.com.br/paradigma-orientado-a-notificacoes-pon-uma-tecnica-de-composicao-e-execucao-de-software-234943.html.

[4] P. V. Roy and S. Haridi, "Concepts, Techniques, and Models of Computer Programming," MIT Press, Cambridge, 2004.

[5] M. Gabbrielli and S. Martini, "Programming Languages: Principles and Paradigms," Springer-Verlag, London, 2010.

[6] J. M. Simão, R. R. Linhares, F. A. Witt, C. R. E. Lima and P. C. Stadzisz, "Paradigma Orientado a Notificações em Hardware Digital". Patent pending submitted to INPI/Brazil in 2012 and UTFPR Innovation Agency in 2012.

[7] R. R. Linhares, J. M. Simão and P. C. Stadzisz. "NOCA – A Notification-Oriented Computer Architecture". IEEE Latin America Transactions, Vol. 13, Issue 5, May 2015.

[8] C. L. Forgy, "RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, Vol. 19, No. 1, 1982, pp. 17-37. doi:10.1016/0004-3702(82)90020-0.

[9] J. M. Simão, R. F. Banaszewski, C. A. Tacla, P. C. Stadzisz, "Notification Oriented Paradigm (NOP) and Imperative Paradigm: A Comparative Study," Journal of Software Engineering and Applications (JSEA), p.402-416, v.5, n.6, 2012.

[10] R. D. Xavier, "Paradigmas de desenvolvimento de software: comparação entre abordagens orientada a eventos e orientada a notificações". Master in Science Thesis at the Federal University of Technology – Paraná (UTFPR). Curitiba – Paraná (PR), Brazil, 2014 (September).

[11] J. M. Simão, D. L. Belmonte, A. F. Ronszcka, R. R. Linhares, G. Z. Valença, R. F. Banaszewski, J. A. Fabro, C. A. Tacla, P. C. Stadzisz, and M. V. Batista. "Notification Oriented and Object Oriented Paradigms Comparison via Sale System". Journal of Software Engineering and Applications (JSEA), Vol. 5, p. 695-710, 2012. http://dx.doi.org/10.4236/jsea.2012.59083.

[12] E. Peters, R. P. Jasinski, V. A. Pedroni, J. M. Simão. "A New Hardware Coprocessor for Accelerating Notification-Oriented Applications". International Conference on Field-Programmable Technology - FPT 2012 IEEE. Seoul, Korea (South).

[13] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," 2006.

[14] A. F. Ronszcka, R. F. Banaszewski, R. R. Linhares, C. A. Tacla, P. C. Stadzisz, J. M. Simão, "Notification-Oriented and Rete Network Inference: A Comparative Study". In: IEEE SMC 2015, Hong Kong – China. Accepted Paper, 2015.