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

ROBSON RIBEIRO LINHARES

CONTRIBUIÇÃO PARA O DESENVOLVIMENTO DE UMA ARQUITETURA DE COMPUTAÇÃO PRÓPRIA AO PARADIGMA ORIENTADO A NOTIFICAÇÕES

QUALIFICAÇÃO DE DOUTORADO

CURITIBA
2013

ROBSON RIBEIRO LINHARES

CONTRIBUIÇÃO PARA O DESENVOLVIMENTO DE UMA ARQUITETURA DE COMPUTAÇÃO PRÓPRIA AO PARADIGMA ORIENTADO A NOTIFICAÇÕES

Qualificação de doutorado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do título de "Doutor em Ciências" – Área de Concentração: Engenharia de Computação.

Orientador: Prof. Dr. Paulo Cézar Stadzisz
Co-orientador: Prof. Dr. Jean Marcelo Simão

CURITIBA

2013

RESUMO

LINHARES, Robson Ribeiro. **Contribuição para o desenvolvimento de uma arquitetura de computação própria ao paradigma orientado a notificações.** 2013. 268 f. Texto de qualificação (Doutorado em Engenharia Elétrica e Informática Industrial). Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI). Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, 2013.

O contexto atual da engenharia de *software* carece de técnicas para concepção, produtiva e com qualidade, de *software* que faça uso eficiente do potencial de execução paralelo disponibilizado pelo *hardware* dos sistemas computacionais modernos. Dentro deste contexto desenvolveu-se recentemente o Paradigma Orientado a Notificações (PON), cuja essência é o processo de inferência otimizado, baseado na colaboração ativa entre pequenas entidades lógico-causais por meio de notificações pontuais. As características teóricas do PON lhe permitem explorar a questão de paralelização e/ou distribuição de forma mais simples e eficiente do que paradigmas de computação mais usuais como o Paradigma Imperativo ou o Paradigma Declarativo. No entanto, a dinâmica de execução do PON, baseada em notificações, não é eficientemente realizada pelo *hardware* dos sistemas computacionais atuais, fundamentalmente baseado no modelo de von Neumann / Turing (e similares) de execução sequencial. De forma a abordar esta deficiência, este trabalho de pesquisa apresenta uma contribuição para o desenvolvimento de uma arquitetura de computação, denominada ARQPON, que é própria para a execução de *software* desenvolvido segundo o PON. Para tanto, analisa-se a fundamentação teórica sobre arquiteturas paralelas, sob o ponto de vista das vantagens e desvantagens dos diversos modelos de execução paralela e dos correspondentes recursos utilizados para sincronização e implementação de concorrência. Em seguida, propõe-se uma versão preliminar da ARQPON, em termos da sua estrutura de blocos funcionais e da correspondente dinâmica de funcionamento, visando realizar avaliações qualitativas e quantitativas que permitam validar os conceitos empregados na elaboração da arquitetura, bem como avaliar criticamente a própria aplicabilidade e viabilidade do PON como um paradigma promissor para o projeto de sistemas computacionais.

Palavras-chave: arquiteturas de computadores, arquiteturas paralelas, paradigmas de programação, paradigma orientado a notificações.

ABSTRACT

LINHARES, Robson Ribeiro. **Contribution to development of a computing architecture proper to the notification oriented paradigm.** 2013. 268 f. Texto de qualificação (Doutorado em Engenharia Elétrica e Informática Industrial). Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI). Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, 2013.

The current context of software engineering lacks techniques for the productive and quality design of software that makes efficient use of the parallel execution capabilities provided by the *hardware* of the modern computing systems. In this context, the Notification Oriented Paradigm (NOP) has been recently developed, whose essence is its optimized inference process, based on active collaboration among small causal-logical entities by means of precise notifications. The theoretical characteristics of NOP allow it to exploit parallelization / distribution in a simpler and more efficient way than more commonly used programming paradigms, such as the Imperative Paradigm and the Declarative Paradigm. However, the dynamics of execution under NOP, based on notifications, is not efficiently performed by the *hardware* of current computing systems, which are fundamentally based on the von Neumann / Turing (and similar) model of sequential execution. In order to address this drawback, this research presents a contribution to development of a computing architecture, named ARQPON, which is suitable for execution of software developed according to NOP concepts. Therefore, the fundaments of parallel architectures are initially analyzed, from the point of view of advantages and disadvantages presented by several different models of parallel execution and their corresponding resources used for implementation of synchronization and concurrency. Subsequently, it is proposed a preliminary version for the ARQPON in terms of its functional block structure and its corresponding dynamics of execution, aiming at performing qualitative and quantitative evaluations which will allow the validation of the concepts used for its elaboration, as well as performing a critical assessment of the applicability and viability of NOP itself as a promising paradigm for design of computing systems.

Keywords: computer architecture, parallel architectures, programming paradigms, notification oriented paradigm.

LISTA DE ILUSTRAÇÕES

FIGURA 1 – POSSÍVEIS CONFIGURAÇÕES PARA O AMBIENTE DE DESENVOLVIMENTO E EXECUÇÃO DE PROGRAMAS PON 26
FIGURA 2 - ESQUEMA DE COLABORAÇÕES ENTRE AS ENTIDADES DO METAMODELO DE CONTROLE DISCRETO 35
FIGURA 3 - EXEMPLO DE REDUNDÂNCIA NO PARADIGMA IMPERATIVO 38
FIGURA 4 - RELAÇÃO ENTRE PON, PI E PD 41
FIGURA 5 - METAMODELO DE NOTIFICAÇÕES DO PON 44
FIGURA 6 – SUBCONJUNTO DE REGRAS PARA UM SISTEMA DE CONTROLE DE TRÁFEGO VIÁRIO 45
FIGURA 7 - DIAGRAMA DE OBJETOS PARA A IMPLEMENTAÇÃO DA CADEIA DE NOTIFICAÇÕES DO EXEMPLO 49
FIGURA 8 – METAMODELO DE NOTIFICAÇÕES DO PON COM CLASSES PARA RESOLUÇÃO DE CONFLITOS E DETERMINISMO 52
FIGURA 9 – TAXONOMIA DE COMPUTADORES PARALELOS 62
FIGURA 10 – EXEMPLIFICAÇÃO DOS DIFERENTES TIPOS DE CONFLITOS DE DADOS EM *PIPELINE* 69
FIGURA 11 – ARQUITETURA DO ARM CORTEX-A9 71
FIGURA 12 – GRÁFICO DA EVOLUÇÃO DO TEMPO TÍPICO DE ACESSO À MEMÓRIA VERSUS TEMPO DE CICLO EM CPU 74
FIGURA 13 – EXEMPLOS DE *MULTITHREADING* DE GRANULARIDADE GROSSA, FINA E SIMULTÂNEO 78
FIGURA 14 – ESQUEMA DE UMA MÁQUINA RAW 81
FIGURA 15 – EXEMPLO DE FUNÇÃO IMPLEMENTADA SEGUNDO O MODELO DE FLUXO DE DADOS 91
FIGURA 16 – ARQUITETURA ELEMENTAR DE FLUXO DE DADOS 94
FIGURA 17 – DIAGRAMA EM BLOCOS DO SISTEMA DF-KPI 97
FIGURA 18 – ORGANIZAÇÃO DA WAVECACHE 100
FIGURA 19 – ESTRUTURAS DE DADOS DO MODELO TAM 103
FIGURA 20 – EXEMPLO DE ALOCAÇÃO DE RECURSOS EM ETS 110
FIGURA 21 – EXEMPLO DE ATUAÇÃO DE SEQUÊNCIA DE INSTRUÇÕES SOBRE *I-STRUCTURES* 111
FIGURA 22 - EXEMPLO GENÉRICO DE REDE NEURAL 120
FIGURA 23 – ARQUITETURA DA SIMULAÇÃO DO SISTEMA DE TELEFONE IMPLEMENTADO SEGUNDO O PON EM *HARDWARE* 123
FIGURA 24 – ARQUITETURA DO SOC UTILIZANDO COPROCESSADOR PON 125
FIGURA 25 – ARQUITETURAS DE MULTIPROCESSADOR (A), MULTIPROCESSADOR COM CACHES INDIVIDUAIS (B) E MULTICOMPUTADOR (C) 126
FIGURA 26 – RELAÇÃO ENTRE NÍVEIS DE HIERARQUIA DE MEMÓRIA 130
FIGURA 27 – EXEMPLOS DE TOPOLOGIAS UTILIZANDO BARRAMENTO COMPARTILHADO PARA INTERCONEXÃO 136
FIGURA 28 – EXEMPLOS DE *CROSSBAR* E REDE ÔMEGA PARA UM ARRANJO DE 4 NÚCLEOS 138
FIGURA 29 – ESQUEMA GERAL DE UMA NOC PARA 16 NÚCLEOS ESTRUTURADA EM GRADE 139
FIGURA 30 – ARQUITETURAS PARA UTILIZAÇÃO DE DISPOSITIVO DE LÓGICA RECONFIGURÁVEL EM UM SISTEMA 146
FIGURA 31 – ETAPAS DO PROCESSO DE DESENVOLVIMENTO EM *HARDWARE* RECONFIGURÁVEL 149
FIGURA 32 – CENÁRIOS DE SIMULAÇÃO DO PROBLEMA "MIRA-ALVO" 155
FIGURA 33 – ETAPAS DO MÉTODO DE PESQUISA ADOTADO 170
FIGURA 34 - CRONOGRAMA DE EXECUÇÃO DAS ATIVIDADES 176
FIGURA 35 – MODELO LÓGICO DA ARQPON 182
FIGURA 36 – CAMADAS DE INTERCONEXÃO DA ARQPON V1.0 189

FIGURA 37 – ORGANIZAÇÃO MULTI-MESTRE COM BARRAMENTO E HÍBRIDA 192
FIGURA 38 – BLOCOS ARQUITETURAIS DA ARQPON 213
FIGURA 39 – SUBSISTEMA DE MEMÓRIA DA ARQPON 215
FIGURA 40 – ESTRUTURA INTERNA DE UM PP 221
FIGURA 41 – ESTRUTURA INTERNA DE UM CP 222
FIGURA 42 – ESTRUTURA INTERNA DE UM MP 224
FIGURA 43 – ESTRUTURA INTERNA DO S/CS 225
FIGURA 44 – BLOCOS FUNCIONAIS ENVOLVIDOS NA ROTINA DE *STARTUP* 236
FIGURA 45 – CENÁRIO DE EXECUÇÃO DA ARQPON 241

LISTA DE TABELAS

TABELA 1 – SUMARIZAÇÃO DE MODELOS E IMPLEMENTAÇÕES DE FLUXO DE DADOS............... 107
TABELA 2 – AVALIAÇÃO DA UTILIZAÇÃO DE MÉTRICAS COMPARATIVAS DO PON EM FUNÇÃO DO ESCOPO DE COMPARAÇÃO 162
TABELA 3 – REQUISITOS FUNCIONAIS DA ARQPON 178
TABELA 4 – REQUISITOS DE INTERFACE LÓGICA DA ARQPON 178
TABELA 5 – REQUISITOS DE INTERFACE FÍSICA DA ARQPON 179
TABELA 6 – REQUISITOS OPERACIONAIS DA ARQPON 180
TABELA 7 – APLICABILIDADE DAS TÉCNICAS DE INTERCONEXÃO À ARQPON........................ 191
TABELA 8 – CÓDIGOS DAS OPERAÇÕES RELACIONAIS EFETUADAS PELAS *PREMISES* 205
TABELA 9 – CÓDIGOS DAS OPERAÇÕES LÓGICAS EFETUADAS PELAS *CONDITIONS* 207
TABELA 10 – CÓDIGOS DAS OPERAÇÕES EFETUADAS PELOS *METHODS*................................... 209
TABELA 11 – MAPEAMENTO DAS REGRAS DO CENÁRIO DE EXECUÇÃO EM ELEMENTOS DO METAMODELO DO PON 238

LISTA DE SIGLAS, ACRÔNIMOS E ABREVIATURAS

ALU	*Arithmetical and Logical Unit*
API	*Application Programming Interface*
ARQPON	Arquitetura de Processador para o Paradigma Orientado a Notificações
ASIC	*Application-Specific Integrated Circuit*
CISC	*Complex Instruction Set Computing*
CMP	*Chip Multi Processor*
COMA	*Cache Only Memory Architecture*
CON	Controle Orientado a Notificações
CORBA	*Common Object Request Broker Architecture*
COW	*Cluster Of Workstations*
CP	*Condition Processor*
CPLD	*Complex Programmable Logic Device*
CPU	*Central Processing Unit*
CSMA	*Change-Sensitive Memory Access*
DARPA	*Defense Advanced Research Projects Agency*
DCOM	*Distributed Component Object Model*
DLP	*Data Level Parallelism*
DMA	*Direct Memory Access*
DON	Desenvolvimento Orientado a Notificações
DOR	Desenvolvimento Orientado a Regras
DRAM	*Dynamic Random Access Memory*
DSM	*Distributed Shared Memory*
EDGE	*Explicit Data Graph Execution*
EEMBC	*Embedded Microprocessor Benchmark Consortium*
EEPROM	*Electrically Erasable and Programmable Read Only Memory*
ETS	*Explicit Token Store*
FBE	*Fact Base Element*
FLOPS	*Floating-point Operations Per Second*
FPCA	*Field-Programmable Computer Array*
FPGA	*Field-Programmable Gate Array*
FPU	*Floating Point Unit*

GAL	*Generic Array Logic*
ILP	*Instruction Level Parallelism*
IMT	*Interleaved Multi Threading*
ISA	*Instruction Set Architecture*
JVM	*Java Virtual Machine*
LLP	*Loop Level Parallelism*
MFLOPS	*Millions of Floating-point Operations Per Second*
MIMD	*Multiple Instruction Multiple Data*
MIPS	*Millions of Instructions Per Second*
MISD	*Multiple Instruction Single Data*
MLP	*Memory Level Parallelism*
MMU	*Memory Management Unit*
MP	*Method Processor*
MPI	*Message Passing Interface*
MPP	*Massive Parallel Processor*
NoC	*Network on Chip*
NUCA	*Non Uniform Cache Array*
NUMA	*Non-uniform Memory Access*
OO	Orientação a Objetos
P2ON	Processador para o Paradigma Orientado a Notificações
PAL	*Programmable Array Logic*
PC	*Personal Computer*
PD	Paradigma Declarativo
PGAS	*Partitioned Global Address Space*
PI	Paradigma Imperativo
PL	Paradigma Lógico
PLA	*Programmable Logic Array*
PLD	*Programmable Logic Device*
PON	Paradigma Orientado a Notificações
POO	Programação Orientada a Objetos
POSIX	*Portable Operating Systems Interfaces*
PP	Paradigma Procedimental
PP	*Premise Processor*
PVM	*Parallel Virtual Machine*

RAM *Random Access Memory*
RAW *Read After Write*
RISC *Reduced Instruction Set Computing*
RMI *Remote Method Invocation*
RNA Rede Neural Artificial
ROM *Read Only Memory*
RUP *Rational Unified Process*
S/CS *Scheduler / Conflict Solver*
SBR Sistemas Baseados em Regras
SDF *Scheduled Data Flow*
SDRAM *Synchronous Dynamic Random Access Memory*
SISAL *Streams and Iterations in a Single Assignment Language*
SIMD *Single Instruction Multiple Data*
SISD *Single Instruction Single Data*
SMP *Symmetric Multi Processor*
SMT *Simultaneous Multi Threading*
SPE *Synergistic Processor Element*
SPEC *Standard Performance Evaluation Corporation*
SPLD *Simple Programmable Logic Device*
SRAM *Synchronous Random Access Memory*
STL *Standard Template Library*
TAM *Threaded Abstract Machine*
TCC *Transactional Memory Coherence and Consistency*
TLP *Thread Level Parallelism*
TTDA *Tagged Token Dataflow Architecture*
UMA *Uniform Memory Access*
UML *Unified Modelling Language*
ULA Unidade Lógica Aritmética
VHDL *Very High Speed Integrated Circuits Hardware Description Language*
VLIW *Very Long Instruction Word*
WAR *Write After Read*
WAW *Write After Write*

SUMÁRIO

1 INTRODUÇÃO **14**
1.1 CONTEXTUALIZAÇÃO 14
1.2 OBJETIVOS 19
1.3 MOTIVAÇÕES 24
1.4 CONTRIBUIÇÕES E RESULTADOS ESPERADOS 25
1.5 ORGANIZAÇÃO DO DOCUMENTO 27
2 FUNDAMENTAÇÃO TEÓRICA **28**
2.1 CARACTERIZAÇÃO DE *SOFTWARE* SEQUENCIAL E *SOFTWARE* PARALELO 28
2.2 FUNDAMENTOS DO PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON) 34
2.2.1 Origens do PON *34*
2.2.2 PON comparativamente a outros paradigmas de programação *36*
2.2.2.1 Paradigma Imperativo 37
2.2.2.2 Paradigma Declarativo 39
2.2.2.3 Relação entre o PON e os Paradigmas Imperativo e Declarativo 41
2.2.3 O metamodelo de notificações do PON *44*
2.2.3.1 Resolução de conflitos 50
2.2.4 Estado de desenvolvimento do PON *53*
2.2.4.1 Evolução da teoria sobre o PON 53
2.2.4.2 *Framework* PON C++ 54
2.2.4.3 Distribuição e paralelização do PON 56
2.2.4.4 Desenvolvimento Orientado a Notificações (DON) 56
2.2.5 Adequação do PON para desenvolvimento de software paralelo/ distribuído *57*
2.3 CONCEITOS DE ARQUITETURAS DE COMPUTADORES PARALELAS 60
2.3.1 Categorização de arquiteturas de computadores *60*
2.3.2 Arquiteturas paralelas baseadas em processadores von Neumann *64*
2.3.2.1 Fundamentação teórica do modelo de von Neumann 65
2.3.2.2 Implementações de paralelismo baseadas em von Neumann 66
2.3.2.3 Modelos e linguagens de programação para arquiteturas paralelas von Neumann 83
2.3.2.4 Considerações sobre arquiteturas paralelas von Neumann 87
2.3.3 Arquiteturas paralelas baseadas em fluxo de dados *89*
2.3.3.1 Fundamentação teórica do modelo de fluxo de dados 90
2.3.3.2 Modelos e implementações de fluxo de dados 93
2.3.3.3 Gerenciamento de memória e sincronização 108
2.3.3.4 Modelos e linguagens de programação para fluxo de dados 112
2.3.3.5 Considerações sobre arquiteturas baseadas em fluxo de dados 114
2.3.4 Outros modelos de arquitetura paralela *117*
2.3.4.1 Multicomputadores 117
2.3.4.2 Redes neurais 119
2.3.4.3 Arquiteturas heterogêneas e/ou sensíveis ao domínio da aplicação 120
2.3.5 Organização de memória *126*
2.3.5.1 Hierarquias de memória 128
2.3.6 Topologias de interconexão *135*
2.3.7 Arquitetura do conjunto de instruções *139*
2.3.7.1 Categorização de instruções no modelo de von Neumann 140
2.3.7.2 Categorização de instruções no modelo de fluxo de dados 143

2.4 IMPLEMENTAÇÃO DE *HARDWARE* EM LÓGICA RECONFIGURÁVEL 145
2.4.1 Categorias de dispositivos de lógica programável e modelos de programação.......... 147
2.4.2 Metodologias para desenvolvimento de hardware em lógica reconfigurável 149
2.5 CRITÉRIOS DE BENCHMARKING E DIRETRIZES PARA A COMPARAÇÃO DE DESEMPENHO E AVALIAÇÃO DE IMPLEMENTAÇÃO DO PON PERANTE OUTRAS ARQUITETURAS 151
2.5.1 Considerações sobre avaliações comparativas no contexto do PON 154
2.6 CONSIDERAÇÕES SOBRE O CAPÍTULO 162
3 APRESENTAÇÃO DA PROPOSTA DE TESE 166
3.1 DESCRIÇÃO DA PROPOSTA DE TESE 166
3.2 RELEVÂNCIA DA PROPOSTA DE DOUTORADO 167
3.3 ORIGINALIDADE DA PROPOSTA DE DOUTORADO 168
3.4 MÉTODO DE PESQUISA ADOTADO 169
3.5 PLANEJAMENTO E CRONOGRAMA DAS ATIVIDADES 172
3.5.1 Atividades já desenvolvidas no doutorado 172
3.5.2 Atividades a serem desenvolvidas no doutorado... 174
3.6 CRONOGRAMA DE EXECUÇÃO DAS ATIVIDADES 176
4 DESENVOLVIMENTO DO TRABALHO 177
4.1 LEVANTAMENTO DE REQUISITOS PARA A ARQPON 177
4.1.1 Requisitos funcionais 177
4.1.2 Requisitos de interface lógica 178
4.1.3 Requisitos de interface física 179
4.1.4 Requisitos operacionais 179
4.2 PROJETO DA ARQPON 180
4.2.1 Modelo lógico 181
4.2.1.1 Visão estrutural 181
4.2.1.2 Visão dinâmica 183
4.2.2 Considerações iniciais sobre o modelo de programação 184
4.2.3 Aspectos arquiteturais relevantes 186
4.2.3.1 Processador de granularidade fina 186
4.2.3.2 Topologia de interconexão 188
4.2.3.3 Escalabilidade 194
4.2.3.4 Definição do “método PON mínimo” 195
4.2.3.5 Gerenciamento de informação de contexto 196
4.2.3.6 Semântica de acesso à memória 198
4.2.3.7 Interfaces externas (dispositivos de E/S) 199
4.2.3.8 Interface com processador von Neumann 200
4.2.4 Definição da ISA 200
4.2.4.1 Tamanho das instruções 202
4.2.4.2 Tipos de dados 202
4.2.4.3 Detalhamento das instruções 203
4.2.4.4 Limitações e restrições 211
4.2.5 Definição da microarquitetura 212
4.2.5.1 Visão geral 212
4.2.5.2 Subsistema de memória 214
4.2.5.3 Barramentos / redes de interconexão 218
4.2.5.4 Conjunto de PP (*Premise Processor*) 220
4.2.5.5 Conjunto de CP (*Condition Processor*) 221
4.2.5.6 Conjunto de MP (*Method Processor*) 223
4.2.5.7 *Scheduler / Conflict Solver* (S/CS) 224

4.2.5.8 Rotina de *startup* 234
4.2.5.9 Arbitragem de barramentos 237
4.2.6 Cenário de execução *238*
4.3 CONSIDERAÇÕES RELATIVAS À UTILIZAÇÃO DE *HARDWARE* RECONFIGURÁVEL PARA IMPLEMENTAÇÃO DA ARQPON 242
4.4 CONSIDERAÇÕES SOBRE O CAPÍTULO 244
5 CONCLUSÕES.......... 251
6 REFERÊNCIAS 255

1 Introdução

1.1 Contextualização

Atualmente é inegável a importante contribuição dos sistemas computacionais para a sociedade. Desde as aplicações que executam em *Personal Computers* (PCs), realizando tarefas simples de usuário tais como edição de texto, navegação em internet e outras, até aplicações corporativas complexas ou que necessitam de processamento centralizado em servidores, todas desempenham papel importante e muitas vezes essencial em processos e operações nas mais diversas áreas.

Some-se ainda a este contexto a crescente importância dos sistemas embarcados, que são projetados para desempenhar um conjunto reduzido e dedicado de funcionalidades. Estes sistemas estão presentes desde em aplicações de relativa menor escala e maior porte (sistemas de controle e automação industrial, sistemas de segurança, sistemas de monitoramento) até em aplicações de grande escala para usuários finais (brinquedos, *smartphones* e outros aparelhos eletrônicos portáteis).

Muito em função desta crescente presença dos sistemas computacionais em atividades cada vez mais especializadas e complexas, é também crescente a complexidade do *software* executado por tais sistemas. Tal complexidade demanda a utilização de técnicas cada vez mais sofisticadas para o desenvolvimento de *software*. No entanto, existe uma percepção (BARBARÁN; FRANCISCHINI, 1998) (ASANOVIC et al., 2006) (HISTORY, 2012) de que estas técnicas não evoluem em taxa suficiente para manter os níveis de produtividade requeridos pela indústria. Este efeito, que está relacionado ao conceito de "crise do *software*", já foi detectado em épocas anteriores da história da computação, em função do estágio de desenvolvimento da técnica naquela ocasião, e tentou-se mitigá-lo muitas vezes por meio da proposição de inovações tecnológicas, p. ex., transição do uso de linguagens de programação de baixo nível para alto nível (HENNESSY; PATTERSON, 2007). Contudo, na prática a "crise do *software"* nunca foi reduzida ao longo do tempo por mais que o surgimento de novas técnicas ou produtos criasse uma ilusão de aumento de produtividade.

Paralelamente, *software* mais complexo geralmente requer *hardware* mais complexo e de melhor desempenho para ser executado, mantendo-se os patamares de tempo de execução dentro de valores aceitáveis para cada aplicação. De certa forma, o aumento na complexidade do *hardware* é viabilizado pelos avanços tecnológicos que fundamentam a Lei

de Moore (MOORE, 1965), segundo a qual a densidade de integração de componentes eletrônicos nos circuitos integrados é duplicada aproximadamente a cada 24 meses.

No entanto, segundo estudo elaborado em 2008 pelo *Exascale Working Group,* com apoio da DARPA (*Defense Advanced Research Projects Agency*) (KOGGE et al., 2008), já não se pode mais assumir que a Lei de Moore implica também na duplicação do desempenho de computação a cada 18-24 meses. Isto se deve ao fato de que o aumento na velocidade do *clock* dos processadores, o qual é um dos principais fatores que impulsionam a evolução no desempenho dos microprocessadores nos últimos 20 anos (BORKAR; CHIEN, 2011), já não ocorre nas mesmas taxas da Lei de Moore. O motivo pelo qual isso ocorre é que o principal efeito do aumento da velocidade de *clock* é a exacerbação de problemas de dissipação de potência em arquiteturas com altíssima escala de integração (*gate leakage, subthreshold leakage*), o que se torna uma barreira ao crescimento do poder de processamento (AGERWALA; CHATTERJEE, 2005) (GSCHWIND, 2006).

Além de citarem a velocidade do *clock*, Borkar e Chien (2011) citam outros dois fatores como impulsionadores da evolução do desempenho de microprocessadores: o aumento de capacidade das memórias *cache* e a evolução das técnicas de microarquitetura. Em relação a este último, embora diversas técnicas tenham sido propostas e aprimoradas, geralmente voltadas a recursos mais sofisticados de *pipelining* de instruções que permitam o aumento da quantidade de instruções executadas por ciclo de *clock* (conhecido por ILP, ou *Instruction Level Parallelism*), o desenvolvimento e aplicação de tais técnicas tem sido substituído progressivamente pelo uso de arquiteturas com múltiplos núcleos de execução (*multicore*) (BOBDA et al., 2010). Isso se deve ao fato de que, além das técnicas de ILP aparentemente não estarem evoluindo o suficiente para melhorar, por si só, o desempenho relativo de execução (SCOTT, 2007) (KOGGE et al., 2008) (CHEN et al., 2008), a disponibilidade de espaço para integração de circuitos mais complexos nos *chips* tem cada vez mais favorecido a replicação dos núcleos de execução e a consequente disseminação das chamadas *arquiteturas paralelas*.

As arquiteturas paralelas têm por característica dispor de várias unidades de processamento que cooperam para a execução de um determinado *software*, aproveitando a sua capacidade de processamento de forma simultânea. Isto tende a melhorar o desempenho da execução do *software* em comparação com uma arquitetura que disponha de uma única unidade de execução semelhante, desde que o *software* seja desenvolvido de tal maneira a facilitar a exploração do paralelismo (ou simultaneidade) intrínseco da sua lógica. Disponibilizar múltiplos núcleos de execução em um único *chip* é uma das formas de se

viabilizar uma arquitetura paralela, embora não seja a única visto que outras formas de organização arquitetural podem surtir efeito semelhante, p. ex. a computação em *cluster* ou em grade (TANENBAUM, 2007).

Com a disponibilidade de arquiteturas paralelas, cresceu a necessidade de se dispor de modelos e técnicas que permitam implementar *software* que aproveite esta característica de paralelismo potencial de execução. No entanto, conforme citado por Eschmann *et al* (2002), a elaboração de *software* paralelo esbarra em dificuldades de abstração por parte dos programadores, dado que a distribuição lógica das instruções passa a ser não somente temporal (execução de instruções distribuída sequencialmente ao longo do tempo), mas também espacial (execução de instruções distribuída em vários locais ao mesmo tempo). Além disso, há uma carência de modelos de programação paralela que sejam difundidos e de simples utilização, quando comparados com os modelos disponíveis para programação sequencial (KLAUER et al., 2002).

Somem-se a estes fatores os efeitos da anteriormente citada “crise do *software”* e percebe-se um campo de pesquisa para o qual ainda há uma significativa demanda de contribuição. Neste âmbito, técnicas inovadoras de desenvolvimento de *software* podem se constituir em boas alternativas. Em particular, paradigmas de programação distintos do Paradigma de programação Imperativo (PI) utilizado atualmente, o qual é conceitualmente fundamentado na execução de sequências de instruções segundo uma dada lógica, podem propor conceitos mais facilmente utilizáveis e adaptáveis à realidade das características dos sistemas de computação paralela.

Um paradigma de programação que se mostra promissor para este fim é o Paradigma Orientado a Notificações (PON). Este paradigma foi concebido por Simão (2005) (SIMÃO; STADZISZ, 2009) a partir da proposta de um metamodelo de controle discreto aplicado à manufatura inteligente, o qual define uma forma de colaboração entre entidades de manufatura que posteriormente foi estendida e aplicada à inferência do ponto de vista de *software* genérico (BANASZEWSKI et al., 2007). O conceito-chave deste mecanismo de inferência é a *notificação,* que viabiliza e materializa a interação entre os elementos do metamodelo do PON e que permite, teoricamente, a habilitação simultânea e paralela da execução de instruções programadas segundo o PON (encapsuladas no conceito de *Method*). Esta possibilidade de expressão do paralelismo de forma intrínseca ao modelo de programação, aliada à utilização de conceitos oriundos dos Sistemas Baseados em Regras (SBR) do Paradigma Declarativo (PD), combinados com conceitos do PI, torna o PON uma

alternativa viável para concepção de *software* paralelo e, potencialmente, até mais interessante do que as técnicas já existentes, do ponto de vista de abstração.

Por ser um paradigma relativamente recente, o PON tem sido objeto de diversos esforços de pesquisa e desenvolvimento. Estes esforços visam desenvolver e aprimorar os conceitos, técnicas e ferramentas a ele associadas, de tal maneira a viabilizar o ciclo de desenvolvimento de aplicações PON desde a concepção até a execução. Mais especificamente, os temas de pesquisa já abordados ou sendo abordados em relação ao PON incluem: desenvolvimento de um perfil UML para a formalização de projeto de *software* PON, segundo o que se denominou DON (*Design Orientado a Notificações*) (WIECHETECK; STADZISZ; SIMÃO, 2011) (WIECHETECK, 2011); estudo de técnicas de balanceamento de carga de *software* PON para ambientes distribuídos e *multicore* (BELMONTE, 2012); diversos estudos comparativos do PON com outros paradigmas de programação, do ponto de vista qualitativo e quantitativo (BANASZEWSKI, 2009) (BATISTA et al., 2011) (RONSZCKA et al., 2011) (LINHARES et al., 2011) (SIMÃO et al., 2012c); uso de padrões para a concepção de aplicações PON (RONSZCKA, 2012); desenvolvimento e aprimoramento de um *framework* de *software* para a implementação de aplicações PON utilizando abstrações da linguagem C++ (BANASZEWSKI, 2009) (VALENÇA et al., 2011) (VALENÇA, 2012); e modelagem e implementação de aplicação PON diretamente em *hardware* utilizando elementos lógicos de dispositivos de lógica reconfigurável (*Field Programmable Gate Arrays,* ou FPGAs) (WITT et al., 2011) (PETERS, 2012).

No entanto, com exceção deste último, todos os esforços de implementação e de avaliação de aplicações PON dependeram da execução destas aplicações em plataformas de computação providas de núcleos de processadores convencionais, ou seja, construídos a partir de arquitetura baseada no modelo de execução sequencial de instruções (modelo de *Von Neumann*). Este modelo claramente limita a materialização plena dos conceitos do PON, na medida em que força a sequencialização de todo ou parte do mecanismo de inferência por notificações, o qual deveria ser passível de execução totalmente paralela.

Este efeito é comprovado claramente ao se analisar a implementação atual do *framework* PON (VALENÇA, 2012), que se constitui em uma API (*Application Programming Interface,* ou Interface de Programação de Aplicações) que implementa os conceitos do PON utilizando abstrações do Paradigma Orientado a Objetos (POO). Este *framework* realiza a inferência por meio de um processo de busca sequencial em estruturas de dados e subsequente ativação de métodos das entidades associadas. Isto permite somente

emular o processo de notificação do PON, ainda com a limitação de que os métodos (conceito de *Method*) são implementados por meio de métodos segundo o POO, os quais são fundamentados em lógica sequencial. Ou seja, a aplicação resultante se torna um híbrido de PON e PI (POO).

Os dados de medição de tempo de execução apresentados por Valença (2012) demonstram que tais estruturas de dados utilizadas atualmente pelo *framework* PON contribuem com *overhead* significativo. Isto limita, portanto, os ganhos teóricos de aplicações PON em relação a aplicações PI no que tange ao desempenho.

O principal fator que limita a implementação do *framework* PON a uma abordagem predominantemente sequencializada é, justamente, a indisponibilidade de uma plataforma ou ambiente de execução, particularmente no nível de *hardware,* que ofereça e implemente as abstrações adequadas para permitir a paralelização do mecanismo de notificações com a granularidade requerida pelo PON e de forma escalável. O trabalho realizado por Witt *et al* (2011) se constitui em um primeiro esforço para implementação das abstrações do PON em *hardware,* porém adaptado para o contexto específico da aplicação em foco (no caso, sistema simulador de telefonia). Ainda mais recentemente, Peters (2012) propôs uma generalização do mecanismo de notificações do PON em *hardware* por meio de um coprocessador implementado em FPGA (CoPON), responsável por propagar as notificações e ativar rotinas específicas executando em um processador sequencial, também implementado em FPGA. Tais rotinas específicas implementam o conceito de *Method* do PON segundo o PI, de forma semelhante à técnica utilizada pelo *framework* do PON em *software.*

Ambos os trabalhos seguem uma tendência de se modelar e implementar sistemas em arquiteturas heterogêneas, otimizadas para a aplicação conforme mencionado por Sun *et al* (2006). No entanto, nenhuma das abordagens citadas implementa o PON em *hardware* em sua plenitude, no sentido de que a primeira depende de reconfiguração específica para uma aplicação, incluindo a escala (quantidade) de elementos do metamodelo do PON necessários para implementá-la, e a segunda é uma implementação parcial e híbrida do PON pois delega a execução dos *Methods* para uma arquitetura do tipo Von Neumann convencional. Embora ambas as abordagens aproveitem as vantagens oferecidas pelas FPGAs, no sentido de permitir a adequação do *hardware* exatamente à aplicação que se pretende desenvolver, na prática elas mapeiam cada elemento da cadeia de notificação do PON em um circuito lógico distinto e específico; ou seja, a escalabilidade de ambas as abordagens é muito dependente da disponibilidade de elementos lógicos no dispositivo reconfigurável, que é limitada.

Levando tudo o que foi descrito em consideração, conclui-se que o ambiente de execução do PON ainda carece de desenvolvimento no sentido de prover uma estrutura de *hardware* que de fato aproveite as características do paradigma, incluindo o seu paralelismo intrínseco. Além disso, este ambiente deve ser genérico o suficiente para não impor a necessidade de reconfiguração sempre que uma nova aplicação PON precisar ser executada e deve prever mecanismos que permitam a execução escalável dos programas PON em termos de tamanho, funcionalidade e complexidade.

1.2 Objetivos

O objetivo deste trabalho, submetido a exame de qualificação, é propor um modelo de arquitetura de processador mais adaptado à execução de programas desenvolvidos segundo o PON (a partir deste ponto referida como *ARQPON*) do que o modelo de arquitetura de Von Neumann. O grau de adaptação e as decisões a serem tomadas para efetuá-la serão orientadas pela teoria já desenvolvida em relação ao PON, incluindo as suas similaridades e diferenças em relação a paradigmas tradicionais de programação tais como o Paradigma Imperativo (PI) e o Paradigma Declarativo (PD).

Como objetivos específicos propõe-se os itens listados a seguir.

1) **Investigar e discutir características de arquiteturas paralelas convencionais (baseadas em Von Neumann), eventualmente aplicando os conceitos à ARQPON.**

Diversas técnicas têm sido empregadas na construção de arquiteturas paralelas convencionais para otimizar algum aspecto relevante quanto ao seu desempenho, ao mesmo tempo em que levam em consideração aspectos relacionados, tais como escala de integração e consumo de energia. Este último, em particular, é um problema cada vez mais relevante, dado que os sistemas de computação, de forma geral, estão se tornando cada vez menores e mais portáteis (BORKAR; CHIEN, 2011).

As técnicas citadas procuram evidenciar o paralelismo em pelo menos um de três níveis possíveis: no nível de instrução (ILP), por meio do aumento do número de instruções executadas por ciclo de *clock*; no nível de dados (DLP, ou *Data Level Parallelism*), por meio da manipulação simultânea de conjuntos cada vez maiores de bits; e no nível de linha de execução (TLP, *ou Thread Level Parallelism*) (HENNESSY; PATTERSON, 2007) (HURSON; KAVI, 2008), por meio da definição de processos ou sequências de instruções

que, embora sequenciais entre si, possam ser executadas em paralelo com outras sequências de instruções. Alguns exemplos de técnicas de paralelismo modernamente utilizadas podem ser citados, tais como: *pipelines* com diversos recursos de otimização; núcleos de execução superescalares; *multithreading*; múltiplos núcleos (*multicore*), homogêneos ou heterogêneos; acesso à memória uniforme (UMA) e não uniforme (NUMA); memórias *cache* transacionais; entre outros que podem ser investigados.

Tais técnicas e recursos têm sido constantemente pesquisados e aprimorados, com resultados publicados e discutidos tanto no âmbito acadêmico quanto industrial. Portanto, pretende-se, no âmbito deste objetivo específico, investigar, entender e aproveitar os conceitos fundamentais relacionados àquelas técnicas e até mesmo exemplos de implementação, adaptando-os às características e conceitos requeridos para a ARQPON.

2) **Investigar e discutir características de arquiteturas paralelas estilo fluxo de dados (dataflow), eventualmente aplicando os conceitos à ARQPON.**

O mesmo argumento utilizado para o estudo de arquiteturas paralelas Von Neumann também se aplica ao estudo de arquiteturas baseadas no modelo de fluxo de dados (*dataflow*). Este modelo apresenta similaridades conceituais com o PON, em particular por permitir extrair naturalmente as propriedades de concorrência e paralelismo das aplicações (SAKAI et al., 1989).

Dentre os recursos e técnicas utilizados em arquiteturas *dataflow* a serem investigados e, eventualmente, aproveitados na definição da ARQPON pode-se citar: modelos de *matching* estático e dinâmico; utilização de *frames* de ativação; exploração do princípio da localidade nos acessos à memória; utilização de *I-Structures*; e híbridos entre fluxo de dados e Von Neumann.

3) **Definir os componentes básicos da ARQPON, baseado em uma visão de alto nível do paradigma.**

O PON define um metamodelo para os elementos que compõem a cadeia de notificações. Este metamodelo evoluiu a partir da abordagem holônica para recursos computacionais proposta por Simão (2005) e implementada na ferramenta ANALYTICE II para modelagem de elementos de manufatura, posteriormente se tornando o pilar conceitual do PON no sentido de definir os papéis e relações a partir dos quais se materializa o fluxo de notificações.

A ARQPON, de acordo com o seu objetivo genérico de ser o mais adaptada possível à filosofia do PON, deve, portanto, mapear explicitamente o metamodelo da cadeia de notificações em blocos arquiteturais de *hardware.* As relações entre estes blocos devem ser configuráveis para cada aplicação PON que se deseje executar, levando em consideração a limitação da quantidade de recursos disponíveis e permitindo que estes recursos sejam realocados e escalonados dinamicamente, de acordo com técnicas e políticas a serem definidas.

4) **Realizar uma implementação da ARQPON, por meio do detalhamento dos circuitos que compõem cada bloco e da sua implementação em FPGA, em escala apropriada às limitações do *hardware*, porém suficiente para demonstrar os conceitos aplicados.**

Segundo Asanovic *et al* (2006), FPGAs têm se tornado cada vez mais uma alternativa para a implementação de protótipos de arquiteturas de processadores. Isto ocorre em função de fatores tais como o contínuo aumento da capacidade, permitindo a prototipação de lógicas cada vez mais complexas, e a flexibilidade de reconfiguração, permitindo que testes sejam rapidamente reformulados e refeitos após avaliações dos testes anteriores. No entanto, FPGAs apresentam algumas limitações no que diz respeito ao desempenho (velocidade do *clock*) e também à eficiência de implementação de certas lógicas quando comparadas com a implementação equivalente em ASICs (*Application Specific Integrated Circuits*).

Feitas estas considerações, como este objetivo é especificamente demonstrar que a ARQPON efetivamente implementa os conceitos do PON e permite a execução de programas elaborados sob este paradigma, opta-se por realizar a implementação em FPGA. A partir deste ponto referir-se-á a esta implementação pela sigla P2ON (*Processador do Paradigma Orientado a Notificações*).

Questões relativas a desempenho e limitações de recursos do P2ON deverão ser propriamente abordadas na discussão sobre a comparação quantitativa com outras arquiteturas (ver objetivo específico a seguir) e na discussão sobre escalabilidade.

5) **Projetar uma aplicação relevante, 100% orientada a notificações, e implementá-la em baixo nível segundo as regras de implementação (conjunto de instruções, etc.) impostas pela ARQPON.**

Conforme já citado, diversos trabalhos sobre o PON foram publicados nos quais casos de estudo envolvendo aplicações PON foram utilizados em análises comparativas com

outros paradigmas (BANASZEWSKI, 2009) (BATISTA et al., 2011) (RONSZCKA et al., 2011) (LINHARES et al., 2011). No entanto, como tais casos de estudo foram implementados em C++ e compilados juntamente com um *framework* que abstrai os conceitos do PON e os implementa na forma de estruturas de dados, nenhuma das implementações é, de fato, 100% orientada a notificações.

Dado que um compilador PON ainda não está disponível (ver Seção 1.3), pretende-se especificar uma aplicação PON e implementá-la em baixo nível, utilizando-se do conjunto de instruções a ser proposto na definição da ARQPON. Esta aplicação PON possui os seguintes requisitos fundamentais:

- A aplicação PON deve ser concebida de tal maneira a estimular a ocorrência de notificações paralelas.
- A aplicação PON deve ser concebida de tal maneira a requerer mais elementos conceituais do metamodelo do PON do que os elementos correspondentes de *hardware* disponíveis na implementação do P2ON. Ou seja, a aplicação deve exercitar a capacidade do P2ON de escalonar dinamicamente elementos do metamodelo do PON, alocando-os aos elementos de *hardware* disponíveis.

Em particular, o primeiro requisito permite comprovar que a ARQPON é capaz de materializar o PON em sua plenitude, eliminando da sua execução as limitações típicas impostas por arquiteturas predominantemente sequenciais. O segundo requisito, por sua vez, permite avaliar se de fato a ARQPON prevê corretamente o escalonamento e realocação de recursos de *hardware* do P2ON para execução da cadeia de notificações do PON de forma funcional.

6) Efetuar análises qualitativas da ARQPON por meio de testes da aplicação PON projetada.

Estando disponíveis a implementação do P2ON e a implementação da aplicação PON, pretende-se efetuar análises qualitativas da ARQPON. Alguns exemplos de parâmetros qualitativos que devem ser analisados são: atendimento dos requisitos funcionais da aplicação PON pelo P2ON e consequentemente, pela ARQPON; adequação do conjunto de instruções da ARQPON ao desenvolvimento de outras aplicações; limitações da ARQPON no que diz respeito à materialização do paralelismo e escalabilidade.

7) **Investigar uma aplicação comum de avaliação de desempenho, cuja natureza explore as vantagens propostas pelo PON, e implementar esta aplicação segundo as regras impostas pela ARQPON.**

Diversas aplicações ou *suites* de aplicações de *benchmarking* são utilizadas, no domínio de arquitetura de computadores, para avaliações quantitativas de desempenho. Alguns exemplos são: SPEC CPU2006 (STANDARD..., 2006), EEMBC (EMBEDDED..., 2012), programas que implementam métodos baseados em cálculo numérico tais como somatório, resposta ao impulso finito, resposta ao impulso infinito, transformada rápida de Fourier e algoritmos de encriptação (LIU; FURBER, 2005); dentre outros.

No entanto, a escolha da aplicação ou *suite* mais adequada para avaliar determinada arquitetura deve ser influenciada por características da arquitetura. Keeton *et al* (1998) sustentam, por exemplo, que alguns recursos arquiteturais que auxiliam na execução de *software* técnico e científico, no estilo dos testes propostos pela *suite* SPEC, não auxiliam na execução de aplicações comerciais baseadas em acesso a banco de dados. Asanovic *et al* (2006) questionam, por sua vez, a validade de testes de *benchmark* comumente utilizados (por exemplo, *suite* SPEC) para avaliar processamento paralelo, dado que o estilo de programação é diferente; ainda, propõem um conjunto de aplicações (apelidadas de "anões") teoricamente mais adequado à avaliação de sistemas paralelos. Disto se conclui que dados obtidos de avaliação de *benchmark* devem ser submetidos a uma reflexão crítica sobre a sua validade, dado o contexto em que a avaliação foi aplicada.

No âmbito deste objetivo específico, portanto, será efetuada a seleção de uma aplicação comum que seja adequada para uma avaliação quantitativa da ARQPON. Esta seleção será baseada em métricas que indiquem quais os índices de desempenho da ARQPON mais adequados para esta análise, levando em conta as especificidades do paradigma. Em seguida esta aplicação será implementada para execução no P2ON e os dados de desempenho serão medidos e analisados.

8) **Comparar a aplicação comum executando no processador PON com a sua execução em outras plataformas, do ponto de vista das métricas de desempenho selecionadas.**

Uma vez implementada a aplicação comum de avaliação de desempenho proposta para execução no P2ON, esta aplicação também será implementada para execução em outras plataformas. Estas plataformas podem ou não explorar algum tipo de paralelismo, porém serão selecionadas criticamente de maneira a permitir a discussão e a comparação com os resultados obtidos no P2ON, segundo as métricas anteriormente estabelecidas.

1.3 Motivações

Partindo do princípio de que o PON pode vir a ser um paradigma relevante e promissor para o desenvolvimento de sistemas computacionais, como apontam as pesquisas realizadas e em andamento, torna-se importante dar continuidade ao desenvolvimento teórico e técnico sobre os seus conceitos. Experimentos como os já realizados em relação à implementação do PON diretamente em FPGA demonstram que este paradigma pode ser útil também para diversificar e, até mesmo, ampliar a expressividade das linguagens de descrição de *hardware*, o que expande suas fronteiras para além do desenvolvimento de *software* propriamente dito.

Tendo em vista esta necessidade de continuidade na evolução do ambiente de desenvolvimento e execução de aplicações PON, a principal motivação deste trabalho é justamente contribuir para a materialização do PON em sua plenitude. Ou seja, deve ser possível que um sistema computacional tenha o seu *software* concebido segundo os princípios do PON desde o início, já nas fases de análise e modelagem, e que este *software* seja materializado e finalmente executado mantendo-se a essência do paradigma. Esta reflexão é referendada desde 1990 pelas considerações realizadas por Arvind e Nikhil quando da proposição da arquitetura *tagged-token* de *dataflow* no MIT, segundo os quais o problema da computação paralela não pode ser resolvido somente em um nível, sendo necessário haver uma sinergia entre a linguagem, o compilador e a arquitetura de execução (ARVIND; NIKHIL, 1990).

O esforço de elaboração da ARQPON é importante no sentido de consolidar o PON como uma alternativa tecnológica para a construção de *software,* particularmente podendo atender à crescente necessidade de técnicas inovadoras de construção de sistemas paralelos. Em particular, a proposição do desenvolvimento da ARQPON a partir da demanda oferecida pelo PON é o inverso do que ocorre atualmente em relação ao desenvolvimento de *software* paralelo segundo o Paradigma Imperativo, no qual são as particularidades e restrições dos novos dispositivos e arquiteturas de *hardware* que geram a demanda por novas técnicas de desenvolvimento de *software.*

1.4 Contribuições e resultados esperados

A Figura 1 descreve possíveis configurações para o ambiente de desenvolvimento e execução de aplicações PON. Nesta figura é apresentada uma visão sistêmica para cada uma das possíveis configurações, desde a camada representativa do modelo da aplicação PON até a camada representativa do dispositivo que executa a aplicação PON. As camadas relativas a *software* são representadas por retângulos bidimensionais, ao passo que as camadas relativas a *hardware* são representadas por retângulos com efeito de tridimensionalidade.

A Figura 1 (a) mostra os blocos componentes da versão atual do ambiente na qual os programas PON são compilados juntamente com o *framework* PON implementado em C++. Perceba-se que, abaixo do *framework* e já incluindo a sua implementação, tanto o *software* gerado quanto o *hardware* do ambiente de execução são baseados no Paradigma Imperativo, portanto impedindo a plena realização de algumas das características do PON.

A Figura 1 (b) descreve o ambiente na versão experimentada por Witt *et al* (2011), na qual a aplicação PON é implementada em lógica reconfigurável por meio do mapeamento direto dos elementos da cadeia de notificações e dos *Methods* do PON em *hardware*. Perceba-se que esta versão de ambiente, embora inteiramente orientada ao PON, depende de que a aplicação PON modelada seja diretamente implementada em nível de descrição de *hardware,* portanto suprimindo algumas etapas (e, consequentemente, níveis de abstração) intermediários que são úteis para o processo de desenvolvimento de *software*.

A Figura 1 (c), por sua vez, descreve o ambiente incluindo a contribuição de Peters (2012) no que diz respeito a implementar diretamente em *hardware* os elementos da cadeia de notificações do PON, na forma de um coprocessador. Esta solução também é híbrida entre PON e PI, visto que a execução dos *Methods* do PON ocorre no Processador Von Neumann por meio de lógica sequencial, cuja ativação é escalonada por meio do fluxo de notificações no coprocessador PON.

Finalmente, a Figura 1 (d) descreve o ambiente incluindo a contribuição deste trabalho. Nesta versão, todo o processo de desenvolvimento do *software* é orientado a PON, desde a modelagem da aplicação até o código executável gerado pelo compilador PON. Este código é então executado pelo processador PON (P2ON), implementado em lógica reconfigurável, conforme as premissas definidas pela ARQPON.

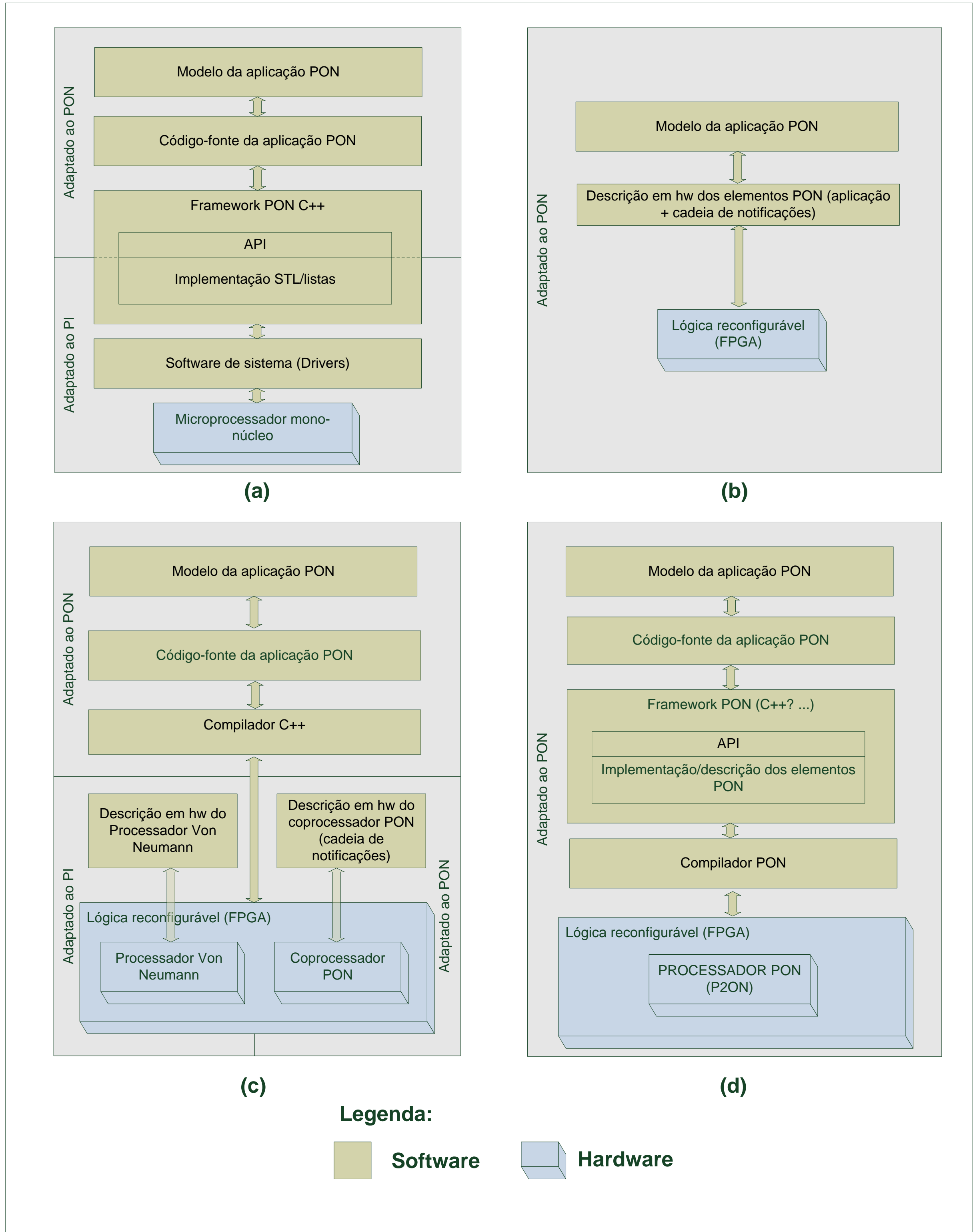


Figura 1 – Possíveis configurações para o ambiente de desenvolvimento e execução de programas PON. (Fonte: autoria própria).

Vale ressaltar que o bloco correspondente ao compilador PON ainda não foi desenvolvido, sendo uma proposta de trabalho futuro; no entanto, não ter o compilador PON

disponível não impede o desenvolvimento da ARQPON, dado que esta pode ser validada com uma aplicação PON implementada em baixo nível. Assim, a definição do conjunto de instruções da ARQPON é uma parte importante da contribuição, pois define a interface para a qual o futuro compilador PON deverá compilar as aplicações PON.

Além disso, a ARQPON é uma alternativa de arquitetura paralela que segue uma filosofia distinta das demais arquiteturas já propostas e implementadas. O estudo das características e do potencial da ARQPON, conforme se pretende neste trabalho, tende a levantar discussões e eventualmente propor soluções e caminhos aplicáveis a outros desenvolvimentos similares, mesmo segundo paradigmas distintos.

1.5 Organização do documento

Este documento está organizado em 5 capítulos. O Capítulo 2 descreve toda a fundamentação teórica pesquisada para elaboração deste trabalho, abrangendo principalmente a teoria do PON e conceitos fundamentais de arquiteturas de computadores, com ênfase em arquiteturas paralelas.

O Capítulo 3 apresenta a proposta do trabalho de doutorado, com detalhamento da proposta de tese, considerações sobre a relevância e originalidade do trabalho e uma descrição do método de pesquisa adotado. Adicionalmente, este capítulo contém uma descrição das atividades já desenvolvidas e por desenvolver e, finalmente, uma proposta de cronograma de trabalho.

O Capítulo 4 contém o desenvolvimento do trabalho até o presente momento. Este desenvolvimento engloba o levantamento de requisitos, premissas e aspectos arquiteturais relevantes para a ARQPON, bem como a proposta preliminar do modelo arquitetural, na forma de diagramas de blocos e da descrição destes diagramas e dos algoritmos envolvidos na sua dinâmica de funcionamento.

O Capítulo 5 apresenta as conclusões desta proposta de tese.

2 Fundamentação teórica

Neste capítulo são apresentados os fundamentos teóricos utilizados como embasamento para o desenvolvimento deste trabalho de qualificação de doutorado.

Em função do objetivo geral deste trabalho estar fortemente relacionado ao domínio de arquiteturas de computadores, aliado à pesquisa no domínio do Paradigma Orientado a Notificações (PON), enfatiza-se neste capítulo conceitos relacionados a arquiteturas de computadores e ao desenvolvimento de *software* que dizem respeito diretamente ao cumprimento daquele objetivo geral e dos seus objetivos específicos.

2.1 Caracterização de *software* sequencial e *software* paralelo

A fundamentação relativa a arquiteturas de computadores, relevante para a concepção da ARQPON, não pode ser abordada somente do ponto de vista de *hardware*. De fato, existe uma necessidade de sinergia entre o *hardware* e o *software*, em relação aos recursos que aquele oferece e de que forma este os aproveita, tanto em aplicações que apresentam características de execução sequencial quanto em aplicações que pretendam fazer uso de paralelismo. Levando-se então em consideração que alguns conceitos relativos à concepção de *software* para execução sequencial são diferentes dos conceitos relativos à concepção de *software* para execução paralela, propõe-se, nesta seção, apresentar uma caracterização dos dois tipos.

Denomina-se *software sequencial* todo *software* que é concebido segundo técnicas de programação sequencial. Estas, por sua vez, geralmente são aplicadas partindo-se da premissa de que o *software* será executado por uma máquina computacional que aloca todos os seus recursos de *hardware* disponíveis para processamento e execução de uma única instrução em determinado instante de tempo. Ou seja, não existe a execução simultânea de várias instruções nem mesmo de partes de uma instrução.

Uma arquitetura que possua uma única unidade central de processamento (CPU, ou *Central Processing Unit*) seguindo o modelo de execução sequencial é comumente denominada de *arquitetura sequencial.* A característica de execução sequencial de instruções por uma única CPU implica em que o desempenho de execução de um *software* sequencial seja dependente do número de instruções que a CPU é capaz de executar em determinada

unidade de tempo. Utiliza-se comumente o termo MIPS (*millions of instructions per second,* ou milhões de instruções por segundo) para quantificar este desempenho em arquiteturas modernas, embora outras métricas possam ser utilizadas (ver Seção 2.5 para maiores detalhes).

O aumento do número de MIPS executadas por uma CPU é influenciado pelos seguintes fatores:

- Aumento na frequência do *clock* da CPU.
- Utilização de técnicas mais sofisticadas de paralelização executadas em nível de *hardware,* ou seja, que paralelizem a execução de partes do *software* sequencial de forma transparente para o modelo de programação. Portanto, neste caso descarta-se a premissa de que todos os recursos de *hardware* são alocados simultaneamente para a execução de uma única instrução, porém ainda deve ser possível executar corretamente o *software* que foi concebido segundo uma abordagem sequencial.

Dentre as técnicas de paralelização em nível de *hardware* que mais têm sido utilizadas em arquiteturas de processadores, destacam-se a paralelização em nível de bit (DLP, ou *data level parallelism*) e paralelização em nível de instrução (ILP, ou *instruction level parallelism*).

O conceito de DLP se refere à possibilidade de se processar simultaneamente conjuntos múltiplos de dados, por meio do uso de barramentos e registradores com grande capacidade de bits. A ILP, por sua vez, é baseada na disponibilização de recursos de *hardware* que permitam dividir a execução de uma instrução em vários estágios e executar diversos estágios de diferentes instruções simultaneamente (conhecido por *pipelining*) (HENNESSY; PATTERSON, 2007). Determinadas características de gerenciamento do *pipeline* de instruções, com o uso de recursos tais como execução fora de ordem e execução especulativa de *branches* após predição dinâmica, podem melhorar sobremaneira o aproveitamento simultâneo dos múltiplos estágios de execução do *pipeline,* aumentando assim o valor médio de MIPS da CPU.

No entanto, os fatores citados anteriormente não têm evoluído de forma satisfatória nos últimos anos. De fato, o aumento da frequência do *clock* é limitado em função de características físicas dos componentes utilizados na fabricação de microprocessadores (BORKAR; CHIEN, 2011), enquanto que as técnicas de DLP e ILP também chegaram a um limite de desenvolvimento tecnológico, em função de que a complexidade extra que agregam

ao *hardware* já não se traduz mais em um aumento significativo no desempenho de execução (CHEN et al., 2008).

Em contraposição, denomina-se *software paralelo* os programas que são explicitamente compostos, total ou parcialmente, por comandos ou sequências de comandos que podem apresentar execução simultânea com outros comandos ou sequências de comandos. Ou seja, *software* paralelo não é composto por uma única linha de execução (*thread*) obrigatoriamente executada sequencialmente no tempo, mas sim por potenciais múltiplas linhas de execução que podem ser executadas simultaneamente umas às outras. Esta visão do *software* paralelo sendo composto por múltiplas linhas de execução é denominada de TLP (*thread level parallelism,* ou paralelismo em nível de *thread*).

Seguindo raciocínio similar ao aplicado na definição de arquitetura sequencial, denomina-se *arquitetura paralela* um ambiente, composto por múltiplas unidades de execução/processamento (CPUs) interconectadas, capaz de executar efetivamente de forma simultânea e concorrente as múltiplas *threads* definidas por um *software* paralelo. Ou seja, uma arquitetura paralela deve idealmente executar *software* paralelo atendendo as premissas a partir das quais ele foi concebido.

A concepção de *software* paralelo pode seguir premissas de programação paralela ou de programação concorrente. De acordo com Catanzaro et al. (2010), programação paralela engloba o conjunto de técnicas utilizadas para aumentar o desempenho de determinada computação (por exemplo, processamento de um vetor de dados) por meio da paralelização das operações utilizadas durante esta computação. Dentre estas técnicas pode-se destacar o uso de instruções que aproveitem um eventual LLP (*Loop Level Parallelism,* ou Paralelismo em Nível de Laço) (HAMMOND; NAYFEH; OLUKOTUN, 1997), o qual é implementado por algumas arquiteturas por meio de execução paralela de iterações de laço cujos dados sejam independentes entre si.

Já as técnicas de programação concorrente são utilizadas na resolução de problemas nos quais as múltiplas linhas de execução do programa potencialmente concorrem por recursos, tanto em ambientes monoprocessados quanto multiprocessados. Como exemplo pode-se citar a concepção de *software* para sistemas embarcados, que tipicamente reagem e tratam eventos que podem ocorrer enquanto efetuam outro tipo de processamento.

De qualquer maneira, técnicas de programação paralela e concorrente não são necessariamente exclusivas entre si, havendo frequentemente uma utilização conjunta de ambas. O aproveitamento do paralelismo oferecido pela arquitetura em qualquer dos níveis possíveis (TLP, ILP etc.) deve levar em consideração possíveis problemas de concorrência e

sincronização, sendo uma questão a ser abordada e considerada cuidadosamente quando da concepção do *software*. Pode-se citar, como exemplo, o acesso a variáveis compartilhadas que, em um programa que utiliza técnicas de TLP, pode requerer algum tipo de sincronização ou exclusão mútua por meio do uso de primitivas tais como semáforos ou *mutexes*.

Além disso, *software* sequencial também pode ser paralelizado até certo ponto. Segundo Ferlin (2008), esta paralelização pode ser executada implicitamente de forma automática ou semi-automática pelo próprio compilador, por meio de algumas fases adicionais no processo de compilação tais como a detecção de paralelismo e alocação de recursos. Ainda, Ferlin (1997)(2008) descreve um conjunto de técnicas de paralelização implícita e lista um conjunto de compiladores que oferecem este recurso, o qual é explorado adequadamente mediante a utilização de linguagens de programação apropriadas (p. ex. SISAL) (SISAL, 2012) (HURSON; KAVI, 2008). Esta abordagem está de acordo com a reflexão efetuada por Scott (2007), segundo a qual idealmente o programador não deveria se preocupar com questões relativas a potencial paralelismo quando do desenvolvimento de *software*, devendo estas questões ser abordadas pelas ferramentas (compiladores, etc.) durante o processo de geração do *software* executável. No entanto, a transformação de *software* sequencial em paralelo incorre no consequente impacto em *overhead* de sincronização e comunicação entre as diversas *threads* oriundas do processo de paralelização (KLAUER et al., 2002), além do que a identificação das partes do *software* que são paralelizáveis pode não ser simples, inclusive devido a questões relativas ao paradigma de desenvolvimento utilizado.

Em relação à forma como detalhes arquiteturais influenciam na visão que o desenvolvedor aplica na concepção do *software* paralelo, a diferenciação mais visível ocorre entre arquiteturas baseadas em multiprocessadores e arquiteturas baseadas em multicomputadores. Segundo Tanenbaum (2007), a principal diferença entre os dois tipos de sistemas é que os sistemas baseados em multiprocessadores mantêm um espaço compartilhado de memória para utilização por todas as unidades paralelas de execução, ao passo que sistemas baseados em multicomputadores mantêm espaços de memória dedicados para cada unidade de execução. Esta organização de memória influencia nas técnicas e estratégias que devem ser implementadas no *software* paralelo para sincronização e comunicação entre as diferentes unidades de execução.

Além disso, embora não se constitua de fato em paralelismo real, existem técnicas para execução de *software* paralelo em arquiteturas sequenciais. Estas técnicas estão fundamentadas na sequencialização escalonada das múltiplas linhas de execução (atualmente *threads* pertencentes a processos) que compõem o *software* paralelo, geralmente efetuada por

um sistema operacional. Isto permite que, em determinado instante de tempo, uma determinada linhas de execução (aqui chamada de *thread*) esteja ocupando exclusivamente os recursos computacionais de *hardware* segundo alguma política de escalonamento. Embora o *software* não seja, de fato, executado em paralelo segundo esta abordagem, ainda assim deve ser concebido segundo as premissas de programação concorrente no que diz respeito à comunicação e sincronização entre diferentes *threads* e aos problemas de concorrência que podem advir desta organização.

A principal dificuldade em relação ao desenvolvimento de *software* paralelo se refere ao alto grau de abstração necessário para se estruturar a lógica de forma coerente com os objetivos a serem alcançados por meio da paralelização. De fato, conforme lembrado por Irwin e Shen (2005), a lógica aplicada no desenvolvimento de *software* de maneira geral é influenciada pela forma de pensar dos seres humanos, e estes têm dificuldade natural em abstrair concorrência ou paralelismo, o que dificulta inclusive a análise e a prova de correção de *software* paralelo.

Ainda no que tange às dificuldades em se desenvolver *software* paralelo, Culler e Arvind (1988) reforçam que a execução de programas paralelos demanda mais recursos e gerenciamento mais complexo destes recursos do que a execução de programas sequenciais. Como exemplos pode-se citar a necessidade de se conhecer o modelo de consistência de memória para garantir que os acessos são feitos na ordem correta, a necessidade de se utilizar estruturas algorítmicas mais complexas para explorar o paralelismo e a possibilidade de execução não determinística que pode levar a problemas de concorrência tais como *deadlocks* (GUPTA; SOHI, 2011). Todas estas questões demandam uma relação de compromisso, segundo a qual um programa paralelizável deveria aproveitar ao máximo o potencial de execução paralela oferecido pelo sistema computacional que o executa, para justificar a sua paralelização, ao mesmo tempo procurando minimizar a utilização de recursos como a memória e a complexidade no gerenciamento destes recursos.

Finalmente, do ponto de vista de desempenho, em teoria, o *software* paralelo pode aproveitar a disponibilidade de múltiplas unidades de execução para aumentar o número de MIPS, quando comparado a *software* sequencial com o mesmo propósito. No entanto, as dificuldades e particularidades citadas anteriormente podem influenciar negativamente o desempenho, principalmente no que diz respeito à necessidade de abstrações mais complexas e de mecanismos de comunicação/sincronização adequados.

Em adição a estas questões, deve-se levar em consideração que um algoritmo originalmente implementado de forma sequencial não é, necessariamente, totalmente

paralelizável. Amdahl definiu quantitativamente o ganho de desempenho obtido pela paralelização de parte de um *software* sequencial, em função do percentual executável em paralelo e do ganho de desempenho (*speedup*) teórico obtido para a parte paralelizável, dando origem ao que se convencionou chamar de "Lei de Amdahl" (HENNESSY; PATTERSON, 2007). Esta é representada pela Equação 1.

Equação 1 – Cálculo do *speedup* total segundo a Lei de Amdahl

$$Speedup_{total} = \frac{1}{fp/Speedup_{teórico} + (1 - fp)}$$

Nesta equação, o *speedup* teórico corresponde ao número de unidades de execução disponíveis para a execução paralela e *fp* corresponde à fração paralelizável (0 a 1, correspondendo a 0 e 100%).

A Lei de Amdahl propõe que, à medida que a fração paralelizável se aproxima de 100%, o *speedup* total é mais próximo do *speedup* teórico. Por exemplo, uma arquitetura paralela com 10 unidades de execução que executasse um determinado *software* 100% em paralelo, sem interdependências entre as partes paralelas que degradassem o desempenho individual, geraria um *speedup* de 10 vezes em relação a uma única unidade executando o mesmo *software* sequencialmente.

No entanto, à medida que a fração executável em paralelo se aproxima de 0%, o *speedup* total se aproxima de um, ou seja, não há ganho. A diminuição da fração executável em paralelo pode ocorrer não somente pela dificuldade ou impossibilidade de se paralelizar o *software* por completo, mas também por interdependências existentes entre as partes paralelizáveis, requerendo pontos de sincronização que forcem a execução sequencial em determinados trechos.

A razão entre o *speedup* total obtido e o número de unidades disponíveis para execução em paralelo é denominada "eficiência" e apresenta uma noção de quão eficiente é a paralelização do ponto de vista de disponibilização de unidades extra para a execução do *software*. De fato, um valor de eficiência próximo de 1 indica que a ocupação das unidades paralelas na execução do programa é próxima de 100%.

Tendo em vista esta conceituação apresentada, as seções a seguir discorrem sobre temas que estão intimamente relacionados ao contexto do *software* paralelo. Mais especificamente, apresenta-se o Paradigma Orientado a Notificações (PON), enfatizando-se a

sua adequação para o desenvolvimento de *software* paralelo ou distribuído, bem como conceitos de arquitetura de computadores com enfoque nas tecnologias envolvidas para viabilizar a computação paralela.

2.2 Fundamentos do Paradigma Orientado a Notificações (PON)

As seções a seguir apresentam a teoria fundamental sobre o Paradigma Orientado a Notificações (PON), ou em inglês *Notification Oriented Paradigm (NOP)*, seguida de uma contextualização deste paradigma no escopo deste trabalho.

2.2.1 Origens do PON

O Paradigma Orientado a Notificações (PON) tem suas origens apresentadas na tese de doutorado de Simão (2005). Nesta tese o autor discorre sobre mecanismos de controle para sistemas de produção, propondo finalmente uma abordagem na qual um metamodelo de controle discreto é utilizado para a emulação e também a simulação de sistemas de manufatura, permitindo assim simular os chamados Sistemas Holônicos de Manufatura. Este metamodelo permite organizar as colaborações entre as entidades "inteligentes" de manufatura por meio de uma abordagem holônica, na qual cada entidade de manufatura é integrada a um sistema computacional por meio de um recurso virtual e tem suas colaborações regidas por um sistema de controle "inteligente".

Estes recursos virtuais, por sua vez, são apresentados individualmente como uma unidade (hólon) que possui atributos e métodos, representando os estados da entidade de manufatura e os serviços que pode executar. O controle destes recursos virtuais por um sistema "inteligente" de controle de manufatura, por sua vez, é expresso por meio de relações causais, constituídas por entidades regras (*Rules*) aplicáveis aos atributos e métodos de um recurso virtual.

A Figura 2 exemplifica de forma genérica o metamodelo proposto. Resumidamente, segundo este metamodelo, cada entidade *Rule* é decomposta em uma entidade *Condition* que permite a avaliação causal e a execução de ações (*Actions*) em função do valor da respectiva condição. Cada *Condition,* por sua vez, é decomposta em entidades que tratam de operações lógicas (*Premises*), as quais operam sobre valores dos atributos dos recursos virtuais. Cada

atributo, assim como os métodos que compõem um recurso virtual, são representados respectivamente por entidades do tipo *Attribute* e *Method*; os métodos, por sua vez, são ativados por diferentes instigações *(Instigations*) disparadas pelas *Actions* contidas nas *Rules.*

Este esquema constitui-se em um mecanismo de inferência que pode ser aplicado no domínio de controle dos sistemas de manufatura. Embora as relações estruturais anteriormente citadas entre os elementos do metamodelo sejam primordialmente de agregação, a dinâmica deste mecanismo de inferência é baseada exclusivamente em *notificações.* Ou seja, dada a dependência entre duas entidades A e B quaisquer do metamodelo, a ativação da entidade B dependente somente ocorre quando a entidade A da qual depende lhe envia uma notificação, sendo que esta notificação é proveniente, por sua vez, da própria ativação da entidade A. Devido a esta dinâmica denominou-se este mecanismo de CON (*Controle Orientado a Notificações*).

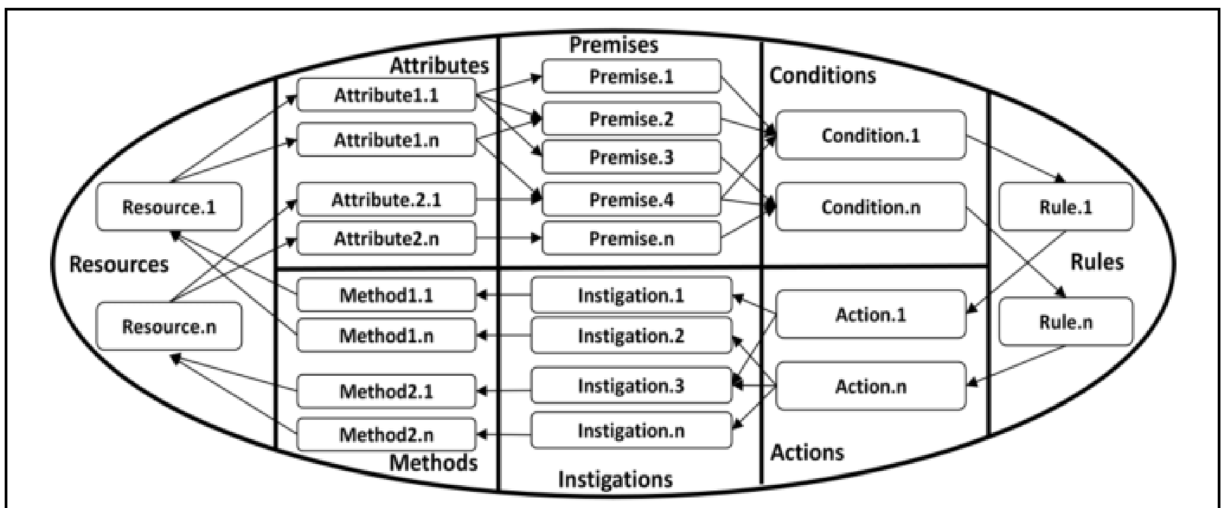


Figura 2 - Esquema de colaborações entre as entidades do metamodelo de controle discreto (Fonte: BANASZEWSKI, 2009, p. 11)

Segundo esta dinâmica, a ativação das entidades que compõem uma *Condition* (entidades *Attribute* e *Premise*) ocorre exclusivamente pela alteração do valor ou estado de alguma delas. Mais especificamente, quando um *Attribute* tem um valor alterado, este notifica as *Premises* que dele dependem sobre esta alteração; a *Premise* realiza o cálculo de seu valor lógico e, caso este tenha sido alterado, notifica as *Conditions* que dela dependem sobre esta alteração; por fim, a *Condition¸* ao ser avaliada com valor lógico verdadeiro, ativa a *Action* da *Rule* na qual está inserida.

A ativação de uma *Action* dispara o ciclo de notificação do ponto de vista de atuação. Ou seja, as *Instigations* associadas à *Action* são notificadas e disparam os *Methods* correspondentes. Estes *Methods*, por sua vez, tipicamente atuam sobre os *Attributes* dos recursos virtuais aos quais pertencem, e esta atuação pode causar alterações nos *Attributes,* as quais causam o reinício do ciclo de inferência por notificação.

O metamodelo de controle por notificações, da forma como foi apresentado, foi aplicado à simulação de sistemas de manufatura na ferramenta ANALYTICE II. Esta aplicação foi submetida a diversos testes e análises, tendo sido os resultados documentados em diversos trabalhos (SIMÃO, 2005) (SIMÃO et al., 2008) (SIMÃO, 2001) (SIMÃO; STADZISZ, 2002) (SIMÃO; STADZISZ; MOREL, 2006).

Além da aplicação a sistemas discretos de controle de manufatura, os autores vislumbraram o metamodelo de notificações como uma alternativa para um modelo a ser aplicado à concepção de controle discreto em geral, inferências discretas em geral e mesmo *software* de maneira genérica. Dado que o mecanismo de notificações e as entidades que o compõem influenciam a visão e a abordagem que o programador deve aplicar para a concepção do programa, denominou-se esta aplicação do metamodelo de notificações à construção de programas de *Paradigma Orientado a Notificações (PON)* (SIMÃO; STADZISZ, 2008).

As seções a seguir apresentam um posicionamento e comparação do PON com outros paradigmas de notificação, bem como detalham a sua aplicação.

2.2.2 PON comparativamente a outros paradigmas de programação

No intuito de analisar criticamente e comparativamente o PON como paradigma de programação, faz-se necessário inicialmente introduzir os conceitos de outros paradigmas aos quais ele pode ser comparado. De fato, o PON utiliza e melhora alguns conceitos apresentados pelo Paradigma Imperativo (PI) e pelo Paradigma Declarativo (PD), portanto serão enfatizados estes dois últimos para termos de comparação.

2.2.2.1 Paradigma Imperativo

Para fins da análise apresentada nesta seção, o Paradigma Imperativo pode ser considerado englobando o Paradigma Procedimental (PP) e o Paradigma Orientado a Objetos (POO) como subparadigmas, embora este último seja considerado por alguns autores como um paradigma distinto (WATT, 2004). Ambos os paradigmas PP e POO apresentam abstrações estruturais significativamente diferentes, com o POO apresentando abstrações mais ricas e naturais para o ser humano, conforme Pressman (1995). No entanto, apesar das diferenças estruturais, ambos se caracterizam por definir *software* como uma sequência lógica de instruções que, quando executada, processa e altera os estados (dados) do programa de acordo com aquela lógica. Sendo assim, pode-se afirmar que construir um *software* segundo o PI significa, essencialmente, programar o *hardware* de um computador sobre "como fazer" o processamento dos estados de tal maneira a se chegar ao resultado esperado.

Esta forma de programação tornou-se popular entre os programadores ao longo dos anos, muito em função da sua adequação direta ao modelo sequencial de von Neumann aplicado na construção da maioria dos microprocessadores dos sistemas computacionais (ver Seção 2.3.2.1). Esta adequação simplificou inicialmente a construção de ferramentas de programação, tais como compiladores, e estimulou naturalmente a definição de linguagens de programação que oferecem as abstrações próprias para programação segundo o PI.

No entanto, a forma de programação segundo o PI induz intrinsecamente o uso de linguagens de programação com muito detalhamento técnico e lógicas complexas para construção dos programas. A forma de programação segundo o PI também acopla naturalmente a lógica do programa aos estados por ele processados, pois mistura o processamento de relações causais (que são dependentes dos dados que representam os estados e de relações lógicas entre eles) com a atuação propriamente dita sobre os dados. Este acoplamento se torna uma deficiência do PI no que diz respeito à possibilidade de distribuição e paralelização, ou seja, de permitir que vários elementos processadores distintos processem partes do programa de forma simultânea e independente, dada esta forte dependência entre estados e lógica do programa (SIMÃO; STADZISZ, 2008) (SIMÃO; STADZISZ, 2009) (SIMÃO et al., 2012a).

Além disso, a estruturação de um programa em sequência de instruções induz o programador a introduzir redundâncias, visto que tem que levar em consideração o aspecto temporal da execução. Ou seja, não é possível na prática, embora talvez fosse logicamente

coerente, elaborar uma sequência lógica de instruções com ocorrência simultânea, dado que o conceito de tempo é intrínseco ao funcionamento do *hardware* que processa as instruções. Isto pode levar o programador a ter que repetir determinada avaliação causal de estado já efetuada pelo fato de que o seu resultado lógico foi utilizado, porém abandonado em um instante de tempo anterior devido à lógica típica de funcionamento do paradigma. Em adição a este fato, a execução contínua da sequência de instruções, característica do PI, pode levar a reavaliações desnecessárias de estados que não foram alterados desde a última avaliação, o que se constitui em outra forma de redundância.

A Figura 3 a seguir exemplifica o conceito de redundância em PI. Nesta figura, as avaliações do *estado1* comparando com *valor1,* tanto na linha 2 quanto na linha 10, são idênticas. No entanto, mesmo não tendo sido alterado o valor de *estado1* nas linhas intermediárias, o valor lógico da primeira avaliação não pode ser aproveitado, dada a estrutura imposta pelo programa. Isto se constitui no que é conhecido por *redundância estrutural*, ou seja, avaliações idênticas repetidas em função de questões estruturais do programa.

Ainda na Figura 3, o laço de repetição cujo cabeçalho é apresentado na linha 1 força a repetida reavaliação de todas as relações contidas nele, independente dos dados envolvidos nas relações terem sido potencialmente alterados ou não. Isto se constitui no que é conhecido por *redundância temporal*, ou seja, avaliações repetidas em função do comportamento temporal do programa, muitas vezes desnecessariamente.

```
1  Enquanto (estado4 menor que valor5)
2     Se (estado1 igual a valor1)
3        então
4           Procedimento1
5     Fim se
6     Se (estado2 igual a valor2 E estado3 igual a valor3)
7        então
8           Procedimento2
9     Fim se
10    Se (estado1 igual a valor1 OU estado1 igual a valor4)
11       então
12          Procedimento3
13    Fim se
14 Fim Enquanto
```

Figura 3 - Exemplo de redundância no Paradigma Imperativo

(Fonte: autoria própria)

Finalmente, a Figura 3 também permite avaliar o acoplamento existente no PI que dificulta a distribuição. Embora não detalhados, os procedimentos *Procedimento1* a

Procedimento3 podem alterar os valores dos estados definidos pelas variáveis *estado1* a *estado4;* estas, no entanto, são utilizadas diretamente nas avaliações das relações causais. Ou seja, existe uma dependência muito forte tanto dos procedimentos quanto das relações causais para com os estados (dados), em operações primitivas tais como operadores relacionais e de atribuição, o que dificultaria sobremaneira a distribuição do código. Ainda, a sequência de relações causais que constitui a lógica do programa também define uma forma de dependência, no sentido de lhes impor uma ordem de avaliação.

A título de menção, o próprio PI, em particular, é utilizado como elemento estruturante de paradigmas ditos emergentes, tais como o Paradigma Orientado a Eventos (POE) e o Paradigma Funcional (PF). Nestes, embora não haja necessariamente um laço principal de execução, existe um mecanismo de disparo dos procedimentos (eventos no POE e funções no PF), sendo que estes são essencialmente construídos segundo o PI, herdando suas vantagens e desvantagens. Quando não feitos em PI, estes procedimentos seriam feitos em PD, objeto da próxima seção (BANASZEWSKI, 2009) (SIMÃO et al., 2012a).

2.2.2.2 Paradigma Declarativo

Por sua vez, o Paradigma Declarativo (PD) é constituído essencialmente pelo Paradigma Funcional (PF) e pelo Paradigma Lógico (PL) (SCOTT, 2000). Diferentemente do PI, o PD permite que o programador se concentre mais na organização do conhecimento sobre o problema; ou seja, construir um *software* segundo o PD significa, essencialmente, definir “o que fazer”.

Em um programa PD não existe necessariamente uma distinção clara entre o que é “dado” e o que é “instrução”. Ao invés disso, tanto os estados (fatos) de um programa quanto as relações causais (regras) que determinam a atuação sobre os estados são declarados como dados, em um nível de abstração elevado (p. ex. utilizando-se linguagens PD de alto nível tais como Prolog e CLIPS) a ponto de independer da forma técnica pela qual as relações serão efetivamente aplicadas sobre os estados. Este processo, geralmente, é realizado por um mecanismo de inferência, separado do programa e operando de forma transparente para ele.

Particularmente, os Sistemas Baseados em Regras (SBR) são implementados segundo os conceitos do PD. Estes sistemas são compostos por um conjunto de regras (*rules*), que são aplicadas a elementos da base de fatos (*fact base elements*) por meio de um

mecanismo de inferência. As regras são expressas segundo uma linguagem própria de PD e, eventualmente, utilizando-se ferramentas que auxiliem nesta expressão (p. ex. *Wizard CON*).

Conceitualmente, os SBRs se fundamentam no processamento cognitivo humano (NEWELL; SIMON, 1972 *apud* BANASZEWSKI, 2009), o qual tem acesso ao conhecimento sobre determinado problema (representado pelos elementos da base de fatos) e define regras para processamento deste conhecimento (relações de causa e efeito), de tal forma a produzir uma resposta apropriada para o problema. Ou seja, as regras representam conclusões sobre os estados dos atributos da base de fatos, determinando a forma como estes atributos são alterados (BANASZEWSKI et al., 2007).

A arquitetura de implementação de um SBR geralmente define uma Base de Fatos (repositório, ou memória, para armazenar os fatos), uma Base de Regras (repositório, ou memória, para armazenar a definição das regras) e uma máquina de inferência. Esta última é separada das bases e define um modelo de processamento que aplica as regras sobre os fatos, gerando novos fatos ou atualizando os existentes (ciclo de inferência). Exemplos clássicos de algoritmos otimizados utilizados em máquinas de inferência são o RETE (FORGY, 1982), o TREAT (MIRANKER, 1987), o LEAPS (MIRANKER; LOFASO; GADBOIS, 1990) e o HAL (LEE; CHENG, 2002).

O ciclo de inferência é geralmente composto por 3 fases distintas: *matching* (casamento), durante a qual os fatos são avaliados e as regras correspondentes são habilitadas, de acordo com o resultado da avaliação; *selection*, durante a qual as regras ativadas são ordenadas segundo alguma estratégia de priorização e/ou resolução de conflito; e *execução,* durante a qual as regras são executadas e a base de fatos é atualizada.

A fase de *matching* é particularmente relevante para o desempenho dos SBR, dado que esta fase ocupa aproximadamente 90% do seu tempo total de execução (MIRANKER; LOFASO, 1991). Como o *matching* é realizado dentro do ciclo de inferência, ele implica em avaliar fatos e regras repetidamente, muitas vezes de forma redundante porque nem sempre ocorrem alterações nos elementos da base de fatos entre dois ciclos de inferência subsequentes quaisquer. Algoritmos de inferência como o RETE e o HAL possuem estratégias para minimizar a ocorrência de redundâncias.

Do ponto de vista das questões de acoplamento e distribuição mencionadas anteriormente em relação ao PI, os problemas relativos a estas questões persistem no domínio dos SBR. Isto se deve ao fato dos mecanismos de inferência serem monolíticos e centralizados, ainda que com variações de implementação entre os diferentes algoritmos (p. ex. base de fatos centralizada em classes utilizada pelo HAL, responsável por ativar as

inferências). Banaszewski (2009) cita algumas iniciativas para paralelização ou distribuição de SBR, consistindo em estratégias tais como replicação de fatos, regras ou máquinas de inferência entre os diferentes nós ou unidades de processamento. No entanto, tais iniciativas são limitadas justamente pelas características intrínsecas da arquitetura e funcionamento conceitual das inferências realizadas pelos SBRs.

2.2.2.3 Relação entre o PON e os Paradigmas Imperativo e Declarativo

O PON, por sua vez, aproveita a forma de expressão dos SBR do Paradigma Declarativo, no sentido de representação do conhecimento na forma de fatos e regras, aliada à flexibilidade de programação e alguns recursos de abstração definidos pela Orientação a Objetos do Paradigma Imperativo, tais como classes e objetos (BANASZEWSKI, 2009). Esta relação é apresentada na Figura 4.

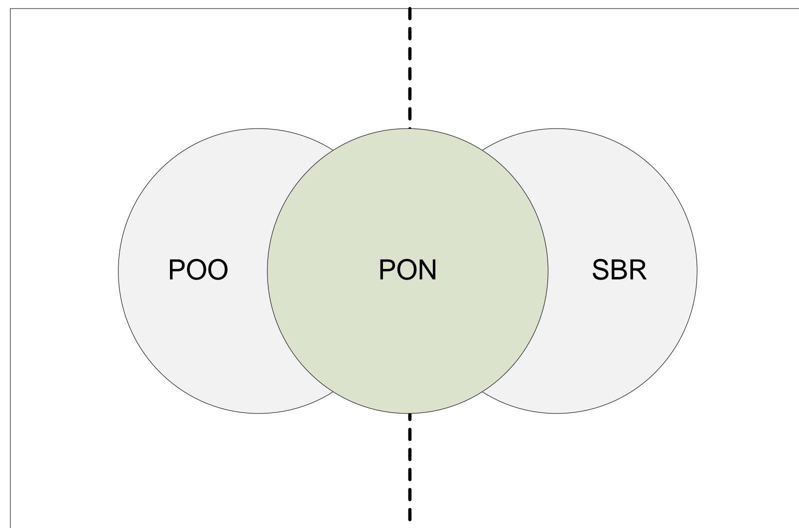


Figura 4 - Relação entre PON, PI e PD
(Fonte: adaptado de BANASZEWSKI, 2009, p. 90)

Ainda, a interseção do PON com o domínio dos SBR está relacionada à sua origem a partir do metamodelo de controle discreto para sistemas de manufatura, conforme apresentado na Seção 2.2.1. De fato, o conceito de FBE já havia sido aproveitado para a representação dos recursos virtuais do metamodelo da célula de manufatura, conforme aplicação nas simulações efetuadas por meio da ferramenta ANALYTICE II (SIMÃO, 2005).

Embora aproveitando conceitos dos dois paradigmas já citados, principalmente do ponto de vista de representação do conhecimento, o PON impõe uma nova forma de estruturação e de elaboração da lógica de programas que justifica a sua classificação como um paradigma em si. De fato, o modelo do PON impõe a expressão da dinâmica do programa e da sua lógica de causa e efeito, no nível mais básico de abstração possível (equivalente à avaliação de uma condição em PI), por meio de notificações potencialmente executadas em paralelo. Isto o diferencia fundamentalmente do comportamento dinâmico de programas PI, nos quais a lógica do programa está fortemente ligada e dependente da sequência de execução, e dos SBR, nos quais a sequência (e desempenho) de execução está abstraída e fortemente dependente do mecanismo de inferência subjacente.

Comparativamente aos PI e PD (particularmente SBR), o PON apresenta uma solução que tem o potencial de melhorar o funcionamento de programas do ponto de vista de desempenho e distribuição, justamente dois dos fatores que apresentam deficiências nas abordagens PI e PD (SBR), conforme explanado. Isto ocorre porque a essência do PON está materializada em dois dos seus conceitos fundamentais:

- Seu processo de inferência por notificações.
- Seu metamodelo de representação de relações causais, baseado em entidades coesas, reativas e desacopladas.

O processo de inferência por notificações viabiliza a propagação imediata de alterações ocorridas em estados (fatos) que são pertinentes ao contexto do programa. Além disso, e até mesmo mais importante do ponto de vista de desempenho global do sistema, a propagação ocorre somente *quando* os estados são alterados. Ou seja, em um programa PON não são efetuadas buscas sobre elementos, seja na forma de *matching* (como nos SBR do PD) ou na forma de avaliações cíclicas ou repetidas de relações causais (como no PI). Ao invés disso, o próprio programa PON é construído como uma rede de elementos reativos, encadeada de tal maneira que cada elemento notifique pontualmente somente os demais elementos aos quais interessa a sua alteração de estado, no tempo em que esta alteração ocorrer.

Estas características do PON viabilizam um ganho teórico em termos de aproveitamento do potencial de computação, o que tende a melhorar o desempenho relativo de aplicações PON quando comparadas a aplicações similares em PI ou PD. Em relação especificamente aos PD, Banaszewski (2009) apresenta uma análise assintótica comparativa entre o mecanismo de inferência do PON e os mecanismos de inferência considerados mais sofisticados no domínio dos SBR (tais como o RETE, TREAT, LEAPS e HAL), com o

objetivo de avaliar teoricamente a complexidade de cada algoritmo à medida que o volume de dados de entrada aumenta. Segundo esta análise, o mecanismo de notificações do PON é mais eficiente do que os algoritmos baseados em busca por co-relacionamento de dados e apresenta desempenho semelhante ao algoritmo do HAL (BANASZEWSKI, 2009).

O metamodelo de representação de relações causais do PON, por sua vez, é constituído por elementos ativos altamente desacoplados entre si. Cada elemento possui um papel bem definido e coeso na construção das relações lógico-causais, seja encapsulando os estados (atributos, fatos) a serem avaliados ou encapsulando as ações a serem executadas em função do resultado lógico daquela avaliação. Estes elementos são ligados uns aos outros explicitamente pelo programador durante a construção do programa, em função das dependências de envio ou recebimento de notificação existentes entre eles. Conforme Simão et al. (2012a), tal estrutura (e o correspondente mecanismo) é uma espécie de extrapolação do padrão de projeto *Observer*, no sentido de que é baseado nos mesmos fundamentos (elementos notificantes e elementos observadores), porém aplicado no nível básico das relações lógico-causais de um algoritmo.

Diferentemente do que ocorre em PI e PD, em função do acoplamento existente entre os dados e a lógica do programa nestes paradigmas, conforme já explanado, o desacoplamento entre os elementos do PON favorece teoricamente a distribuição destes elementos em diversos nós computacionais sem afetar a coesão do programa. Esta distribuição é viabilizada desde que haja meios técnicos para propagar as notificações entre estes diversos nós computacionais envolvidos na computação distribuída (ou paralela).

Ainda, do ponto de vista de distribuição, em particular no domínio dos PI existem tecnologias para facilitar esta tarefa tais como paralelizadores de código e *middleware* para distribuição (CORBA - *Common Object Request Broker Architecture*, DCOM – *Distributed Component Object Model*, RMI – *Remote Method Invocation*, etc.). No entanto, conforme mencionado na Seção 2.1, isto requer a definição explícita de troca de mensagens e sincronização entre as diversas partes distribuídas do programa, adicionando complexidade à sua implementação e impacto negativo no desempenho. Na abordagem segundo o PON, por sua vez, a troca de mensagens está intrinsecamente presente no programa, materializada no conceito de notificação, portanto não é necessário adicionar complexidade a um programa PON para torná-lo conceitualmente passível de distribuição e paralelização.

Sumarizando o que foi explanado anteriormente, pode-se então afirmar como características fundamentais dos programas desenvolvidos segundo o PON, além das características em que é similar aos programas OO e aos SBR: alta reatividade; facilidade

teórica de distribuição; equilíbrio entre generalidade (comportamento do mecanismo de notificações) e aplicabilidade (flexibilidade na composição de regras); e potencial de eliminação de redundâncias (ativação somente das entidades relacionadas a uma determinada notificação, dispensando a ativação de outras entidades).

2.2.3 O metamodelo de notificações do PON

A Figura 5 apresenta a estrutura do metamodelo de notificações do PON na forma de um diagrama de classes em UML.

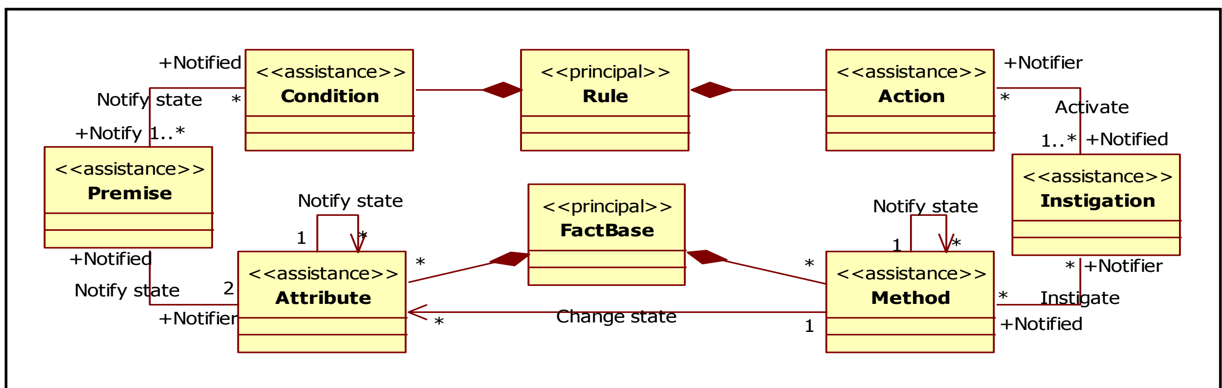


Figura 5 - Metamodelo de notificações do PON
(Fonte: baseado em SIMÃO; STADZISZ, 2008, p. 20)

No intuito de explicar a estrutura e a dinâmica do metamodelo de notificações do PON, faz-se útil a definição de um exemplo de implementação de um conjunto de relações causais segundo aquele paradigma, no qual sejam identificáveis os diferentes elementos do metamodelo. Para a elaboração deste exemplo definiu-se um subconjunto das regras necessárias para representar o funcionamento de um cruzamento em um sistema de controle de tráfego viário, com alguma inspiração no trabalho de Weber et al (2010). Este exemplo é apresentado na Figura 6.

No exemplo da Figura 6, o controle do cruzamento é efetuado por meio do controle de um par de semáforos, chamados de *SemaforoNS* (Norte-Sul) e *SemaforoLO* (Leste-Oeste). Estes semáforos sinalizam a permissão ou proibição de passagem de veículos nas ruas de sentido norte-sul e sentido leste-oeste que compõem o cruzamento, respectivamente.

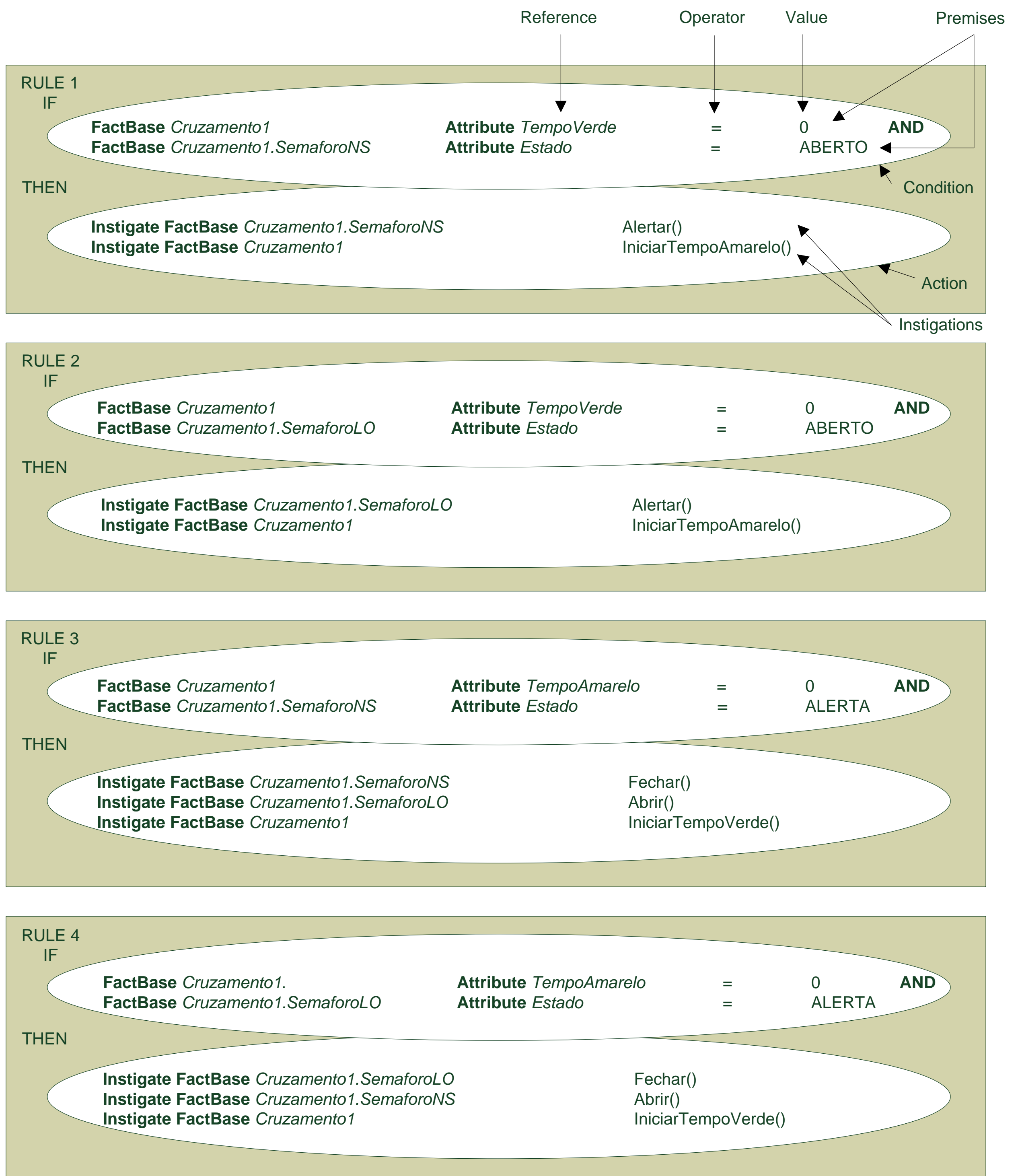


Figura 6 – Subconjunto de regras para um sistema de controle de tráfego viário

(Fonte: autoria própria)

Os semáforos e seus atributos (no exemplo, *Estado*), que compõem o *Cruzamento1*, e o próprio *Cruzamento1* e seus atributos (*TempoVerde* e *TempoAmarelo*) constituem-se nos Elementos da Base de Fatos (FBEs) utilizados na composição das regras *Rule 1* a *Rule 4*.

Cada uma das regras representa uma relação causal, na qual atributos dos FBEs são testados e ações são executadas em função dos valores daqueles atributos. Esta forma de representação é típica dos SBRs e aproveitada pelo PON, o qual por sua vez mapeia os elementos da cadeia de notificações para determinadas partes componentes das regras, conforme segue.

Conforme indicado na Figura 6, o teste condicional da *Rule1* compara o valor do atributo *TempoVerde*, do FBE *Cruzamento1,* com a constante 0 e compara o estado do semáforo *SemaforoNS* com o valor “ABERTO” (luz verde). Caso este teste resulte em valor lógico verdadeiro, a ação a ser tomada é a mudança de estado do semáforo *SemaforoNS* para “ALERTA” (luz amarela) e o início da contagem do tempo para a sua posterior mudança para “FECHADO” (luz vermelha).

Do ponto de vista estrutural, a construção segundo o PON do exemplo da Figura 6 depende da correta instanciação e ligação entre todos os elementos do metamodelo de notificações, de forma a se obter a cadeia de notificações para a aplicação mostrada no exemplo. A seguir, a título de explanação, apresenta-se o conjunto de etapas necessário para construção da *Rule1* segundo o PON. A numeração das etapas não reflete necessariamente uma ordem de execução/instanciação, que é dependente da forma como o metamodelo é implementado, mas sim uma ordem lógica que facilite o entendimento das relações estruturais do metamodelo.

1) Instanciação de objetos da classe *Attribute* do metamodelo do PON para representar os atributos *TempoVerde* e *Estado* dos respectivos FBEs (*Cruzamento1* e *Cruzamento1.SemaforoNS*).

2) Instanciação de objetos da classe *Premise* do metamodelo do PON para representar cada uma das relações lógicas que compõem o teste condicional. Cada uma das relações lógicas, por sua vez, é composta pela comparação de um atributo (representado pela indicação *Reference* na Figura 6) com um valor constante (representado pela indicação *Value*) por meio de um operador relacional (representado pela indicação *Operator*).

A dependência de construção de uma *Premise* em relação aos valores (atributo/constante) a serem testados está representada na Figura 5 por meio da relação de associação entre as classes *Premise* e *Attribute.* O próprio *Value*, de forma genérica, é considerado um *Attribute* para fins desta representação estrutural, por isso a cardinalidade 2 no lado da classe *Attribute.*

3) Instanciação de objetos da classe *Condition* do metamodelo do PON para representar o teste condicional completo da relação causal da regra. Uma condição é composta por um conjunto de premissas e o seu valor lógico é obtido por meio de uma operação lógica sobre o valor lógico de cada uma destas premissas.

A dependência de construção de uma *Condition* em relação às premissas que a ela estão associadas é representada na Figura 5 por meio da relação de associação entre as classes *Condition* e *Premise.*

4) Instanciação de objetos da classe *Method* do metamodelo do PON para representar cada uma das chamadas de método ativadas por uma instigação. Esta classe possui uma relação reflexiva, indicando que um método pode eventualmente ativar outro(s) método(s) para a sua execução. Além disso, a classe *Method* possui uma relação de associação com a classe *Attribute,* indicando que tipicamente a execução de um método causa a alteração de valor de um ou mais atributos. Esta alteração depende da interação entre métodos e atributos viabilizada pelo seu encapsulamento na classe *FactBase,* conforme será explicado a seguir.

5) Instanciação de objetos da classe *FactBase* do metamodelo do PON. Esta classe define, de fato, o modelo para materialização do conceito de FBE, conforme aproveitado da teoria dos SBRs, utilizando-se para tanto a definição do metamodelo de classe do POO, que viabiliza o encapsulamento coeso de dados e comportamento de determinada entidade (objeto) participante do sistema sendo modelado.

O objeto da classe *FactBase*, conforme Figura 5, é composto por objetos das classes *Attribute* e *Method*, representando os dados (atributos) e comportamento (métodos) de um FBE. Isto viabiliza a interação entre *Method* e *Attribute* conforme mostrado na mesma figura, indicando que os métodos de um FBE implementam a lógica de alteração de determinado(s) atributo(s) deste mesmo FBE. No exemplo da Figura 6, em relação à *Rule1,* o método *Alertar()* altera o valor do atributo *Estado* do objeto *Cruzamento1.SemaforoNS* da classe *FactBase.* O método *IniciarTempoAmarelo(),* por sua vez, altera o valor do atributo *TempoAmarelo* do objeto *Cruzamento1* da classe *FactBase.*

6) Instanciação de objetos da classe *Instigation* do metamodelo do PON para representar cada uma das instigações individuais que compõem uma *Action* de uma regra.

A dependência de materialização de uma *Instigation* em relação aos métodos que a compõem está representada na Figura 5 por meio da relação de associação entre as classes *Instigation* e *Method.* Isto significa, conceitualmente, que uma instigação é capaz de ativar a execução de 1 ou mais métodos, tipicamente executando no contexto de um mesmo FBE.

7) Instanciação de objetos da classe *Action* do metamodelo do PON para representar as ações a serem tomadas quando da aprovação de uma regra.

A dependência de materialização de uma *Action* em relação às instigações que a compõem está representada na Figura 5 por meio da relação de associação entre as classes *Action* e *Instigation.* Isto significa, conceitualmente, que uma ação é capaz de disparar a execução de uma ou mais instigações, tipicamente executando cada uma no contexto de um FBE distinto.

8) Instanciação de objetos da classe *Rule* do metamodelo do PON para representar o conceito de regra conforme ilustrado na Figura 5. Esta regra é composta por um objeto da classe *Condition*, representando o teste condicional da relação causal daquela regra, e por um objeto da classe *Action* representando o conjunto de ações a serem tomadas quando a regra é ativada, ou seja, quando o teste condicional resulta em valor lógico verdadeiro.

O mesmo raciocínio aplicado na implementação da *Rule1* pode ser aplicado à implementação das demais regras. No entanto, vale observar que muitos dos elementos instanciados a partir do metamodelo para a *Rule1* se repetem nas demais regras; por exemplo, a primeira *Premise* da *Rule1* (comparação do atributo *TempoVerde* com zero) é idêntica à primeira *Premise* da *Rule2.* Isto abre a possibilidade, prevista na teoria do PON, de reaproveitamento da premissa já instanciada para a *Rule1* na composição da *Rule2,* eliminando uma possível ocorrência de redundância estrutural.

A Figura 7 mostra um diagrama de objetos que representa a materialização e as ligações entre os elementos do metamodelo para o subconjunto completo de regras definido na Figura 6. Naquela figura, as ligações com o estereótipo "*contains*" indicam relações estruturais de agregação, já as ligações desenhadas como dependências da UML, sem

estereótipo, representam as ligações da cadeia de notificações do PON em si, com o sentido da ligação indicando o sentido da notificação. Finalmente, as ligações desenhadas como dependências da UML com estereótipo "*updates*" representam ações de atualização dos valores de atributos.

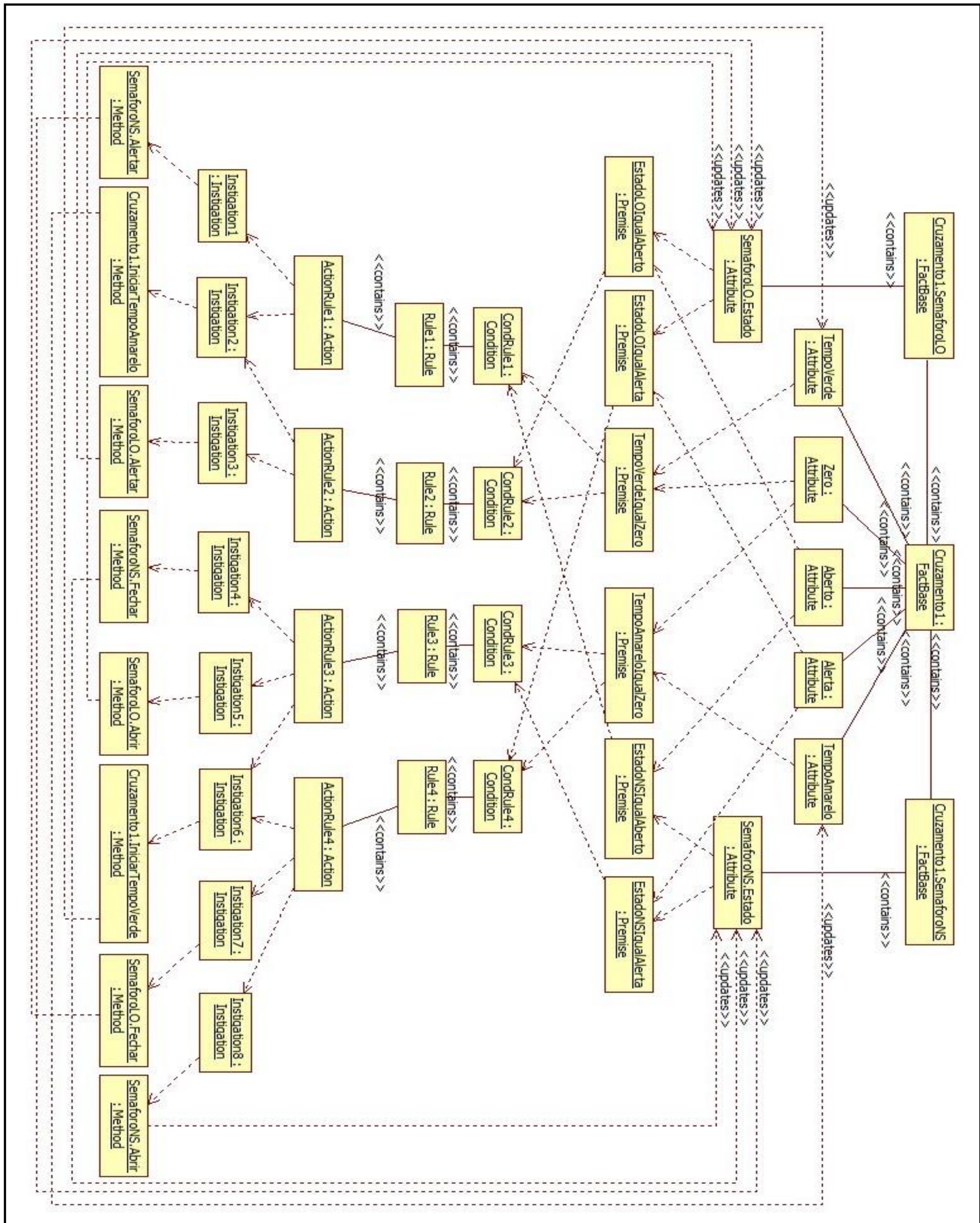


Figura 7 - Diagrama de objetos para a implementação da cadeia de notificações do exemplo (Fonte: autoria própria)

Na Figura 7 percebe-se claramente o reaproveitamento de elementos do metamodelo. De fato, a premissa *TempoVerdeIgualZero* é reaproveitada tanto pela condição *CondRule1* quanto pela condição *CondRule2;* raciocínio semelhante é aplicável à premissa *TempoAmareloIgualZero*, do ponto de vista das condições *CondRule3* e *CondRule4.* Ainda, a instigação *Instigation6* é reaproveitada pelas ações *ActionRule3* e *ActionRule4;* de fato, aquela instigação está ligada ao disparo do método *IniciarTempoVerde*, que é uma ação a ser executada tanto para a regra *Rule3* quanto para a regra *Rule4.*

Do ponto de vista dinâmico, a Figura 7 permite inferir a sequência de notificações a serem disparadas em função das ligações existentes entre os elementos. Por exemplo, eventuais alterações no atributo *TempoVerde* do FBE *Cruzamento1* causariam o disparo de notificação para a premissa *TempoVerdeIgualZero,* a qual, por sua vez, notificaria as condições *CondRule1* e *CondRule2* caso o seu valor lógico tivesse sido alterado em função da atualização de *TempoVerde*.

O recálculo do valor lógico, com resultado verdadeiro, para as condições *CondRule1* ou *CondRule2* causaria a ativação das regras *Rule1* e *Rule2*, respectivamente. Esta ativação notificaria as *Actions* correspondentes, as quais notificariam suas *Instigations* que posteriormente notificariam os *Methods*, disparando a execução do método correspondente. Cada um dos métodos, por sua vez, é responsável por atualizar um *Attribute*, e esta atualização gera uma nova notificação para as *Premises* relacionadas, reiniciando o ciclo.

2.2.3.1 Resolução de conflitos

Devido à possibilidade de aprovação simultânea e paralela de um conjunto de regras concorrentes e conflitantes (i. e., disparam ações que atuam sobre os mesmos atributos dos mesmos FBEs), o metamodelo de notificações do PON depende de um mecanismo de resolução de conflitos. Este mecanismo não está explícito na Figura 7 por ser implementável de diferentes formas (p. ex. por meio de um objeto escalonador de regras aprovadas), porém a sua essência seria controlar o fluxo de notificações a partir da *Action* definida por cada *Rule* aprovada, segundo determinada política, de tal maneira que as instigações de métodos correspondentes não ocorram de forma conflitante com as de outras regras também aprovadas. Desta forma, a resolução de conflitos pode ser intrínseca ao funcionamento da cadeia de notificações, portanto ocorrendo de maneira mais integrada do que as implementações de semáforos e monitores comumente utilizadas em programas implementados segundo o PI.

Neste sentido, um exemplo de resolvedor de conflitos seria um objeto escalonador de *Rules* aprovadas.

Banaszewski (2009) lista um conjunto de políticas de resolução de conflitos aplicadas ao PON: BREADTH, baseada em escalonamento FIFO (*first in first out*), DEPTH; baseada em escalonamento LIFO (*last in first out*); e PRIORITY, baseada em prioridades relativas das regras. É pertinente ressaltar, no entanto, que as políticas citadas são definidas no escopo de ambientes monoprocessados, ou seja, nos quais as regras são executadas sequencialmente após a sua aprovação, de acordo com a ordem definida pela política de resolução de conflitos.

No que diz respeito a ambientes multiprocessados, Simão (2005) e Simão e Stadzisz (2010) propõem uma estratégia de resolução de conflitos e garantia de determinismo baseada em:

- sincronização de acesso a *Attributes* (especializados para *Attribute-Exclusive*) por meio de *flags,* de forma a evitar condições de corrida decorrentes de acessos concorrentes por suas *Premises* conectadas.
- confirmação de recebimento de notificações, que se constituem basicamente de contra-notificações (notificações no sentido contrário).

Estas contra-notificações permitem a determinada entidade indicar ao emissor que recebeu uma notificação, o que viabiliza que o emissor da notificação tome decisões em relação a resolver eventuais conflitos entre múltiplas regras que forem aprovadas em decorrência do seu envio. Neste âmbito, Simão define a entidade *Exclusive-Premise*, derivada de *Premise,* a qual contém um contador do número de *Conditions* que se declararam (por meio de contra-notificações) com valor lógico verdadeiro, cada uma causando a aprovação da sua *Rule.*

De posse da quantidade de *Conditions* que causam aprovação de *Rules* e sendo esta quantidade maior do que um, esta *Exclusive-Premise* pode realizar um procedimento de obtenção do acesso ao *Attribute-Exclusive.* Caso seja bem sucedida, a *Exclusive-Premise* pode então resolver os conflitos ou delegar esta resolução a outra entidade, renotificando as *Conditions* em função do escalonamento determinado na resolução.

Mecanismo semelhante viabiliza a garantia de determinismo, por meio da especialização das classes *Attribute, Premise* e *Condition* nas classes *Deterministic-Attribute, Deterministic-Premise* e *Deterministic-Condition.* Embora esta especialização não seja

obrigatória, o principal objetivo destas classes especializadas é definir um protocolo, também por meio de contra-notificações, que permite aos objetos notificantes se assegurar de que todos os objetos notificados foram efetivamente atualizados a respeito de novos fatos.

Para tanto, ao serem notificadas, as *Deterministic-Conditions* contra-notificam as *Deterministic-Premises* a elas associadas, que por sua vez contra-notificam os *Deterministic-Attributes* a elas associados. As *Deterministic-Conditions* verificam então, de alguma forma (p. ex., por meio de uma nova notificação dos *Deterministic-Attributes* envolvidos), se todas as atividades de notificação referentes àquela atualização de *Attribute* se encerraram, para então notificar a respectiva *Rule* a respeito de sua aprovação, se foi o caso. A Figura 8 ilustra de que forma as classes relativas a resolução de conflito e garantia de determinismo se inserem na estrutura do metamodelo de notificações do PON.

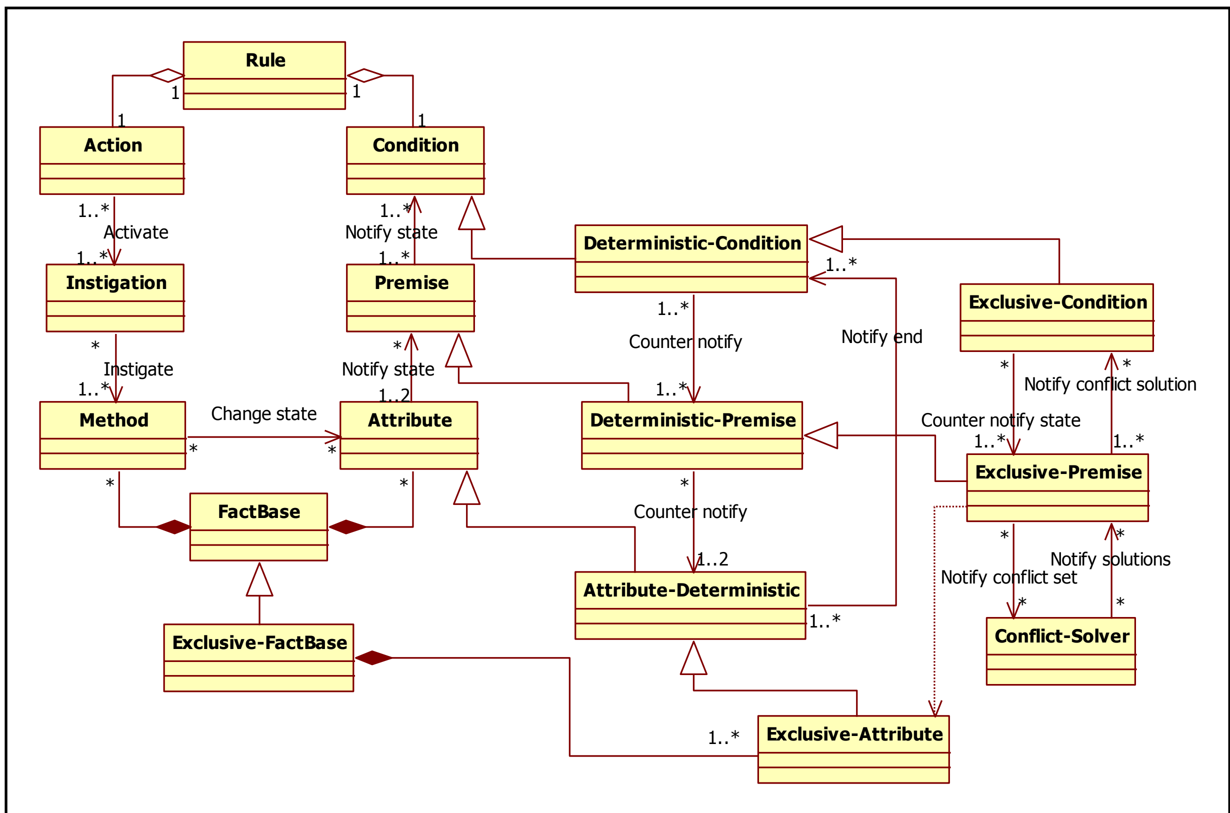


Figura 8 – Metamodelo de notificações do PON com classes para resolução de conflitos e determinismo (Fonte: adaptado de SIMÃO; STADZISZ, 2010, p. 28)

Os autores citam que o mecanismo de contra-notificações, embora não agregue *overhead* significativo no processo de inferência quando comparado com mecanismos de *matching* com a mesma finalidade, permite também melhorar a robustez da cadeia de notificações no que diz respeito a falhas de comunicação, que podem ocorrer com maior

probabilidade se a cadeia de notificações estiver distribuída em uma rede de computadores. Ainda, o uso de contra-notificações permite a elaboração de *handshaking* mais sofisticado entre os emissores e receptores das notificações, o que contribui para o determinismo das avaliações causais na medida em que permite garantir que todas as regras sejam atualizadas a respeito de novos fatos.

2.2.4 Estado de desenvolvimento do PON

A seguir um resumo do estado atual no que diz respeito ao desenvolvimento da teoria e técnicas relacionadas ao PON.

2.2.4.1 Evolução da teoria sobre o PON

A teoria básica do metamodelo de notificações do PON é apresentada nas Seções 2.2.1, 2.2.2.3 e 2.2.3. Alguns conceitos fundamentais do PON, entretanto, foram estendidos e adaptados posteriormente. Estas adaptações visaram principalmente uma adequação a um modelo de execução mais compatível com as tecnologias em microcomputadores pessoais, principalmente em função de particularidades do *framework* PON C++ (detalhado na subseção a seguir).

Valença (2012) evoluiu e implementou o conceito de renotificação, o qual consiste em forçar que uma entidade da classe *Attribute* (re)inicie o ciclo de notificações mesmo que o seu valor não tenha sido alterado. Este recurso permite que uma *Rule* seja reavaliada e, eventualmente, reativada ciclicamente, o que pode diminuir o *overhead* de atualização (alteração) de *Attributes* que seria necessário para efetivação deste comportamento cíclico.

Ronszcka (2012), por sua vez, estendeu e adaptou alguns conceitos fundamentais do metamodelo de notificações do PON, com vistas a facilitar a concepção / programação de aplicações PON sob o viés de padrões e, também, a implementar algumas otimizações do mecanismo do ponto de vista dinâmico. Neste sentido, a principal contribuição foi a definição dos conceitos de regras dependentes, de atributos impertinentes e de métodos PON para operações aritméticas.

As regras dependentes permitem a definição de *Rules* que não são ativadas apenas pelo mecanismo de notificação convencional (*Condition* com valor lógico *true*), mas sim por

isto e pela ativação de outra regra da qual dependem (*Master Rule*). Este conceito visa diminuir a quantidade de notificações enviadas por *Conditions* e *Premises* que são compartilhadas por múltiplas *Rules,* partindo do princípio de que estas *Conditions* e *Premises* poderiam notificar uma única *Rule* (*Master Rule*) e esta notificar as demais *Rules* que também dependem das mesmas *Conditions* e *Premises.*

O conceito de atributos impertinentes, por sua vez, permite determinar que a geração de notificações por certos *Attributes* seja desabilitada temporariamente. Esta técnica evita que *Attributes* que têm o seu valor alterado com muita frequência gerem notificações demasiadas ("impertinentes") a *Premises*, que por sua vez dependem também de outros *Attributes* cuja variação é muito menos frequente. Nesta situação, o *Attribute* de variação mais frequente somente teria a sua notificação habilitada quando o(s) *Attribute(s)* de variação menos frequente notificassem a *Premise*, que solicitaria o valor atual do *Attribute* impertinente e habilitaria a geração de notificação por este atributo.

Finalmente, a definição dos métodos PON para operações aritméticas afeta diretamente a implementação do *framework* PON C++ (ver subseção a seguir), permitindo criar *Methods* de maneira mais simplificada para a realização de operações simples, tais como cálculos aritméticos. Estes métodos são criados por meio da instanciação de classes específicas do *framework* segundo uma organização estrutural ditada pelo padrão de projeto *Composite,* eliminando a necessidade de implementação destes métodos segundo o POO (PI) conforme ocorria nas versões anteriores do *framework*.

2.2.4.2 *Framework* PON C++

O *framework* PON C++, proposto prototipalmente por Simão e refeito no trabalho de Banaszewski (2009) e Simão et al. (2012c), consiste em uma estrutura orientada a objetos que define classes, na linguagem C++, correspondentes aos elementos do metamodelo de notificações do PON. O objetivo deste *framework* é oferecer uma interface de programação (API) de relativamente fácil utilização para a implementação de aplicações segundo o PON, definindo as abstrações necessárias para compor os FBEs e suas respectivas *Rules* (VALENÇA, 2012). As origens do *framework* PON C++ remontam ao *framework* que implementa o metamodelo de controle orientado a notificações (CON) na ferramenta ANALYTICE II (SIMÃO, 2005).

O *framework* em si, por ser um conjunto de classes cujos métodos são materializados segundo o PI, tem sua implementação amplamente baseada em busca sobre estruturas de dados (originalmente fornecidas pela STL – *Standard Template Library* – do conjunto de bibliotecas C++) para avaliação de relações lógico-causais e decisão sobre o envio de notificações (que, por sua vez, são implementadas na forma de chamada de métodos). Esta forma de implementação do *framework* é desvantajosa à filosofia do PON, pois depende fundamentalmente de estruturas de dados e percurso sequencial sobre estas estruturas efetuado em PI, porém apresenta como vantagem permitir uma rápida prototipação de aplicações PON para teste sobre uma plataforma de computação convencional (PC ou mesmo sistemas embarcados para os quais estejam disponíveis compiladores C++). Em função destas características, o *framework* PON C++ se insere na arquitetura de execução de uma aplicação PON conforme mostrado na Figura 1(a).

Dadas algumas questões de implementação do *framework* PON, em particular o *overhead* causado pela sua implementação baseada em estruturas de dados, Valença (2012) efetuou uma série de otimizações com o objetivo de melhorar o desempenho de execução de aplicações sobre o *framework*. O resultado deste trabalho é uma versão do *framework* baseada em uma variedade de estruturas mais otimizadas do que as suas equivalentes baseadas em STL, p. ex. *arrays* (PONVECTOR), listas (PONLIST) e tabelas *hash* (PONHASH). Esta versão do *framework* PON C++ apresenta ganhos de desempenho em diversas aplicações quando comparado à versão original do *framework* apresentada por Banaszewski (2009).

Os resultados obtidos com a avaliação de desempenho de aplicações experimentais, implementadas utilizando o *framework* PON C++, foram e têm sido extensivamente documentados e publicados (BANASZEWSKI, 2009) (BATISTA et al., 2011) (RONSZCKA et al., 2011) (LINHARES et al., 2011) (SIMÃO et al., 2012a). Estes experimentos apresentam visões críticas sobre a aplicabilidade do paradigma e da solução proposta pelo *framework* que têm sido utilizadas na evolução do estado da arte do PON. Adicionalmente, foram iniciados esforços para a especificação de uma linguagem e eventualmente para a construção de um compilador próprio para o PON.

2.2.4.3 Distribuição e paralelização do PON

No que tange a aproveitar e viabilizar as características intrínsecas de distribuição e paralelização do PON, alguns estudos foram e têm sido efetuados tanto em nível de *hardware* quanto em nível de *software.*

Em nível de *software,* Weber et al. (2010) propuseram um ambiente *multithread* para a execução concorrente dos elementos do metamodelo de notificações do CON, onde cada entidade CON (*Rule*, *Condition, Premise* etc.) pode executar uma *thread* em particular. Esta solução foi desenvolvida também para ser aplicada ao PON, porém não foi ainda incluída nas versões atuais do *framework* PON C++. Além disso, Belmonte (2012) estendeu a implementação de Banaszewski do *framework* para incluir a execução de *Rules* em *threads* separadas.

Mais precisamente, Belmonte, Simão e Stadzisz (2012) desenvolvem um estudo relativo a balanceamento de carga em sistemas multiprocessados (i. e. com múltiplos processadores e memória compartilhada) de forma a otimizar a paralelização / distribuição da execução dos elementos do metamodelo de notificações do PON, com o objetivo de melhorar o desempenho de execução (BELMONTE, 2012).

Em nível de *hardware*, pode-se citar os trabalhos já desenvolvidos por Witt et al. (2011), Jasinski (2012) e Peters (2012). Todos estes trabalhos objetivaram viabilizar a implementação de parte (somente cadeia de notificações) ou de toda uma aplicação PON diretamente em *hardware*, particularmente por meio da configuração de um dispositivo de lógica reconfigurável (FPGA). A Seção 2.3.4.3.1 descreve com maiores detalhes os resultados destes trabalhos. Em particular, o trabalho realizado por Witt et al. motivou um pedido de patente (SIMÃO et al., 2012b).

2.2.4.4 Desenvolvimento Orientado a Notificações (DON)

Wiecheteck (2011) propôs um método para projeto de *software* PON, denominado DON (*Desenvolvimento Orientado a Notificações*). Este estudo foi motivado pelo fato das abordagens correntes de modelagem, tanto no domínio do PD (SBR) quanto no domínio do PI, não satisfazerem as necessidades específicas do PON.

A solução proposta é baseada em um perfil UML que expressa os conceitos PON e viabiliza a sua aplicação na etapa de modelagem de um processo de engenharia de *software*. Em particular, o DON se baseia no processo iterativo do RUP (*Rational Unified Process*), adaptando partes dele ao PON. Além disso, o estudo abordou de que maneira os conceitos de Redes de Petri podem ser utilizados para auxiliar ou mesmo guiar a modelagem de *software* PON, sendo que isto foi inspirado em trabalhos precedentes de Simão no tocante ao CON (SIMÃO, 2005)(SIMÃO; STADZISZ, 2009).

O desenvolvimento do DON gerou como resultado uma semi-formalização do PON (incluindo seu *framework*), por meio da categorização e mapeamento dos elementos do metamodelo do PON no perfil UML proposto. Em adição, o DON também resultou em uma proposta para as etapas que constituem o processo de projeto de *software* PON, a qual motivou um pedido de patente que além de contemplar o DON, o generalizou para o Desenvolvimento Orientado a Regras (DOR) (SIMÃO; STADZISZ; WIECHETECK, 2012) .

2.2.5 Adequação do PON para desenvolvimento de *software* paralelo/ distribuído

Segundo Arvind e Nikhil (1990) e Chen et al. (2008), uma linguagem/técnica que seja adequada para o desenvolvimento de *software* paralelo deve apresentar algumas características fundamentais:

a) Capacidade de isolar a programação de detalhes arquiteturais do ambiente de execução (número de unidades de execução, topologia de interconexão entre estas unidades, entre outras).
b) Paralelismo implícito na semântica da técnica/linguagem.
c) Nível adequado de abstração, favorecendo a simplicidade de maneira a diminuir a curva de aprendizado, porém sem comprometer o desempenho. Adicionalmente, segundo Asanovic et al. (2006), níveis muito baixos de abstração tendem a favorecer o desempenho em detrimento da produtividade, sendo o inverso também verdadeiro.
d) Determinismo implícito, sem a necessidade de se dispor de mecanismos explícitos de escalonamento e sincronização embutidos na lógica do programa.

Em relação à característica listada como (a), o PON é completamente utilizável em aplicações comuns, executando em ambiente paralelo ou distribuído, porque não há diferença, em termos conceituais, se um elemento da cadeia a ser notificado está na mesma região de memória, no mesmo computador ou na mesma rede do elemento notificante. A forma de propagação das notificações é função do acoplamento existente entre os elementos da cadeia, que pode ser interpretado sob dois pontos de vista distintos:

- Do ponto de vista estrutural, o “acoplamento” entre entidades PON, no sentido estrito de conexão entre entidades, tende a ser relativamente maior do que entre objetos POO, para a mesma aplicação, dada a menor granularidade dos elementos do PON.
- Do ponto de vista de organização lógica, o metamodelo de notificações conforme proposto (Figura 5) restringe as relações (e, portanto, os acoplamentos) entre os diferentes elementos da cadeia do PON, em função do seu papel no processo de inferência. Esta restrição não existe no POO (PI), portanto pode-se concluir que o acoplamento do PON, sob este ponto de vista, é menor do que no POO.

Como consequência, o menor acoplamento lógico do PON em relação ao POO facilita teoricamente a execução distribuída dos elementos do metamodelo de notificações. Isto ocorre porque as restrições nas relações entre elementos, impostas pelo metamodelo de notificações, tendem a simplificar as interfaces de comunicação e sincronização entre os elementos.

Em relação às características listadas como (b) e (c), estas são atendidas pelo PON devido a este apresentar características intrínsecas, tais como o relativo desacoplamento entre as entidades do metamodelo, que favorecem a paralelização/distribuição. Isto o torna teoricamente mais simples de ser utilizado para construção de *software* paralelo do que conceitos de outros paradigmas (BELMONTE; SIMÃO; STADZISZ, 2012), sendo esta simplicidade também reforçada pelo uso de abstrações de relativamente alto nível para a programação, baseadas nos SBR.

Ao mesmo tempo, o modelo de programação do PON objetiva também evitar a ocorrência de redundâncias temporais e estruturais nas aplicações, o que teoricamente favorece o desempenho das aplicações. Vale, neste ponto, a ressalva de que o *framework* do PON em C++, descrito na Seção 2.2.4.2, apresenta *overhead* que impacta no seu desempenho que não é devido necessariamente ao nível das abstrações que apresenta ao programador, mas, sobretudo, à forma de implementação destas abstrações.

Complementarmente a esta questão, Asanovic et al (2006) argumentam que aumentar o paralelismo explícito das aplicações será o principal método para aumentar o desempenho da computação em multiprocessadores. Embora este argumento pareça conflitante com a característica listada como (b), na prática, ele denota uma preocupação em se conceber o *software* desde o início levando-se em consideração que este possui múltiplas *threads* e as implicações da execução concorrente destas linhas. Levando-se em consideração este ponto de vista, embora aparentemente conflitante com a característica (b), ainda assim a abstração de alto nível apresentada pelo PON permite que o desenvolvedor considere explicitamente as questões de concorrência entre regras, caso julgue necessário, inclusive fazendo uso para tanto de técnicas de modelagem que ajudem a explicitar o paralelismo, tais como Redes de Petri.

Em relação à característica listada como (d), o PON pretende prover determinismo por meio de mecanismos adequados de resolução de conflitos oriundos de concorrência. Banaszewski (2009) discorre sobre técnicas já desenvolvidas e implementadas no *framework* do PON de resolução centralizada e descentralizada de conflitos, porém focadas na execução de programas PON em ambientes monoprocessados. Simão (2005), por sua vez, propõe embrionariamente uma possível estratégia de resolução de conflitos e garantia de determinismo descentralizada em ambientes multiprocessados baseada em contra-notificações no tocante ao CON. Isto é melhorado / evoluído e apresentado subsequentemente em pedido de patente (SIMÃO; STADZISZ, 2010).

Esta preocupação com a resolução de conflitos e garantia de determinismo é coerente com a visão de Denning e Dennis (2010), os quais ressaltam a importância de se obter *determinicidade* na computação paralela, que consiste em garantir que um conjunto de tarefas executadas em paralelo em memória compartilhada sempre produza o mesmo conjunto de saídas para um determinado conjunto de entradas, independente das velocidades relativas de execução das tarefas que processam estas entradas. Para tanto, deve-se garantir inclusive que tarefas conflitantes executem sempre na mesma ordem, o que pode ser obtido no PON justamente por meio do mecanismo de resolução de conflitos.

De fato, na ocorrência de regras conflitantes disparadas simultaneamente, o mecanismo de resolução de conflitos do PON deveria, para uma determinada política de resolução de conflitos, sempre garantir a execução destas regras na mesma ordem ou eventualmente executar somente a regra vencedora do conflito e descartar as demais, dependendo da estratégia que se pretenda implementar. O mecanismo de resolução de conflitos é executado com o apoio do mecanismo de contra-notificações, conforme proposto

por Simão (2005). Ao bem da verdade, o mecanismo de garantia de determinismo (útil inclusive para a resolução de conflitos) também se dá baseado em princípios de contra-notificação (SIMÃO; STADZISZ, 2010)(SIMÃO et al, 2010).

Adicionalmente, a execução distribuída ou paralela de regras do PON depende de um mecanismo para distribuir as regras aprovadas entre as diferentes unidades de processamento distribuídas que estejam aptas a recebê-las (ociosas). Este mecanismo pode também favorecer a escalabilidade, permitindo que uma quantidade N de regras aprovadas seja escalonada para execução em uma quantidade menor do que N de unidades de processamento. Ainda, as políticas de escalonamento são influenciadas pela avaliação de questões tais como balanceamento de carga, conforme analisado por Belmonte, Simão e Stadzisz (2012). De fato, o correto escalonamento de elementos do PON para as unidades adequadas de processamento é fundamental para se otimizar os ganhos potenciais de desempenho de *software* implementado segundo aquele paradigma, quando comparado com implementações PI.

2.3 Conceitos de arquiteturas de computadores paralelas

Nesta seção são apresentados conceitos relativos a arquiteturas de computadores considerados relevantes para a elaboração e proposição da ARQPON. Dado o caráter de paralelismo intrínseco do PON, que se pretende explorar particularmente na ARQPON, enfatiza-se os conceitos e técnicas relativos ao desenvolvimento de arquiteturas paralelas.

2.3.1 Categorização de arquiteturas de computadores

Com o intuito de categorizar diferentes arquiteturas de computadores, tanto sequenciais quanto paralelas, Flynn (1966) propôs uma taxonomia que leva em consideração o modelo de execução de instruções e como estas instruções processam os dados. Baseando-se nesta taxonomia, um sistema computacional pode ser classificado em uma das quatro categorias listadas a seguir.

1) SISD (*Single Instruction, Single Data*)

Nesta categoria se enquadram os sistemas computacionais capazes de executar uma única linha (ou fluxo) de execução, a qual processa sequencialmente um único fluxo de dados.

Portanto, esta categoria define a maioria dos processadores sequenciais de núcleo[1] único (single core).

2) SIMD (*Single Instruction, Multiple Data*)

Nesta categoria se enquadram os sistemas computacionais capazes de buscar uma única *thread* e executar as instruções sobre múltiplos conjuntos de dados. Para tanto, o sistema deve dispor de múltiplos estágios de execução no *pipeline* (ver Seção 2.3.2.2.1) capazes de executar simultaneamente o mesmo fluxo de controle, porém sobre conjuntos de dados independentes.

Na categoria SIMD se enquadram os processadores vetoriais e superescalares (ver Seção 2.3.2.2).

3) MISD (*Multiple Instruction, Single Data*)

Nesta categoria se enquadram os sistemas computacionais providos de múltiplas unidades de execução (estágios de *pipeline* ou até núcleos completos), cada uma buscando e executando uma operação independente porém processando o mesmo fluxo de dados. Hennessy e Patterson (2007) reportam que, embora definida em teoria, nenhum processador que se enquadrasse nesta categoria foi desenvolvido e entrou em uso comercial.

Alguns autores consideram que ILP obtido via *pipelining* é uma forma de MISD, dado que múltiplos estágios funcionais operam sobre os mesmos dados referentes a uma instrução. No entanto contra este argumento existe a interpretação de que os estágios de um *pipeline* não efetuam busca de instruções, de tal forma que não podem ser considerados núcleos de processamento de fato. Outros autores classificam como MISD os chamados *arrays sistólicos,* que consistem basicamente em redes de elementos computacionais conectados em grade, os quais executam uma determinada operação sobre um fluxo de dado, obtido de um vizinho na grade, e disponibilizam o resultado para um outro vizinho (KUNG, 1982) (MAK, 1986).

---

[1] Um "processador sequencial de núcleo único" pode ser definido pelo termo CPU, conforme já explanado anteriormente. Embora este termo possa eventualmente ser utilizado no contexto de computação paralela (TANENBAUM, 2007), ele é antigo e tem uma conotação mais relacionada ao conceito dos processadores sequenciais. Neste trabalho será utilizado o termo "núcleo" (de processamento ou execução), no contexto da computação paralela, por estar mais relacionado ao conceito de "unidade de execução completa de instruções potencialmente operando em paralelo com outras unidades de execução completa de instruções".

4) MIMD (*Multiple Instruction, Multiple Data*)

Nesta categoria se enquadram os sistemas computacionais compostos por múltiplas unidades de execução, cada qual buscando e executando um fluxo independente de instruções e processando um fluxo independente de dados. Por ser a categoria mais abrangente e flexível, do ponto de vista de viabilizar diferentes formas de paralelismo implementadas em *software*, esta categoria e suas variações é o foco de estudo deste trabalho.

Tanenbaum (2007) estende esta classificação taxonômica (

Figura 9), propondo uma subdivisão dos SIMD em processadores de vetores (dedicados a aplicar a mesma instrução sobre elementos de um vetor, em paralelo) e processadores matriciais (execução da mesma instrução por Unidades Lógico-Aritméticas distintas).

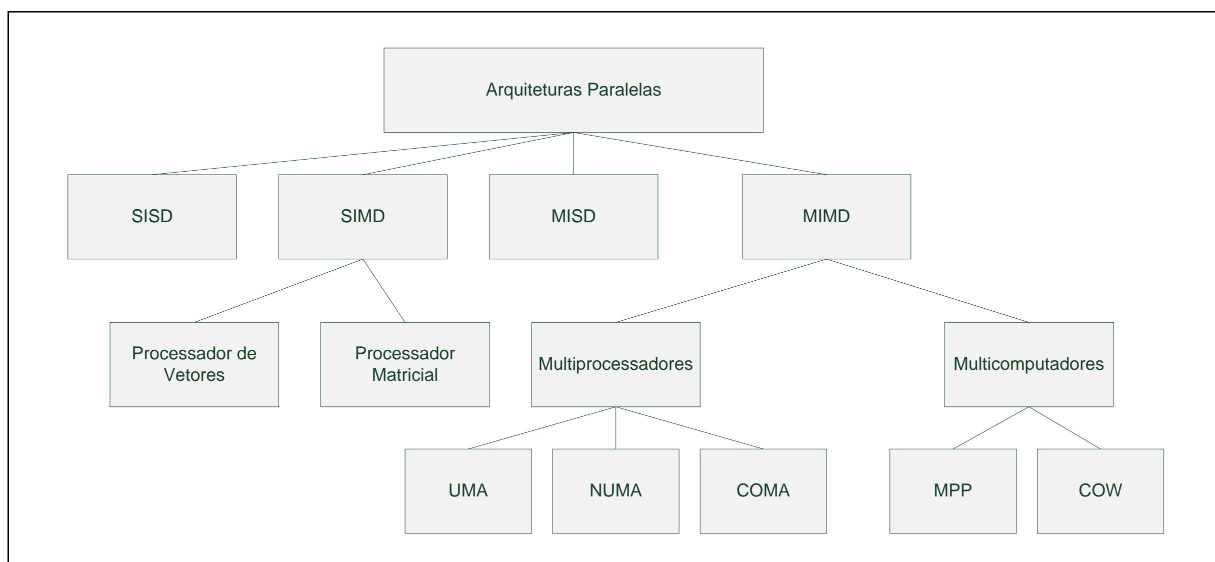


Figura 9 – Taxonomia de computadores paralelos

(Fonte: adaptado de TANENBAUM, 2007, p. 346)

Além disso, propõe-se uma subdivisão mais ampla, em termos de subníveis, para a categoria MIMD, justamente por esta apresentar a maior diversidade de variações. No nível imediatamente inferior ao MIMD classifica-se as arquiteturas paralelas em *multiprocessadores* e *multicomputadores*.

- Multiprocessadores: nesta categoria se encaixam as implementações *multicore*, nas quais múltiplos núcleos de execução executam *software* paralelo cooperativamente e

implementam comunicação e sincronização por meio de uma área de memória comum.

- Multicomputadores: nesta categoria se encaixa uma variação das arquiteturas *multicore*, na qual cada unidade de execução possui um espaço de memória dedicado e se comunica com as demais por meio de algum canal de comunicação de alta velocidade.

Os multiprocessadores estão subdivididos no que diz respeito à forma como a memória compartilhada é implementada: UMA (*Uniform Memory Access,* ou acesso uniforme à memória), NUMA (*Non Uniform Memory Access,* ou acesso não-uniforme à memória) e COMA (*Cache Only Memory Access,* ou Acesso Somente à Memória Cache). A Seção 2.3.5 discorre sobre as diferentes formas de compartilhamento de memória.

Os multicomputadores, por sua vez, estão subdivididos no que diz respeito à sua implementação utilizando *hardware* dedicado ou estações de trabalho comuns: MPP (*Massive Parallel Processors,* ou Processadores de Paralelismo Massivo) e COW (*Cluster Of Workstations*), ou Grupo de Estações de Trabalho, respectivamente. A Seção 2.3.4.1 discorre sobre as diferentes categorias de multicomputadores em maiores detalhes.

De forma complementar à visão taxonômica anterior, Tanenbaum (2007) apresenta outra visão no que diz respeito ao nível de abstração e à forma de implementação do paralelismo em arquiteturas paralelas:

- Arquiteturas com paralelismo de baixo nível: nesta categoria se encaixam as diferentes implementações de ILP, tais como arquiteturas superescalares, *pipelines* e arquiteturas baseadas em palavras longas com múltiplos operandos (VLIW, *very long instruction word*) (ver Seção 2.3.2.2.1). Em arquiteturas com paralelismo de baixo nível define-se as unidades de execução como sendo diferentes estágios de processamento existentes dentro de um núcleo (p. ex. diferentes unidades lógicas-aritméticas). Pode-se relacionar este nível de paralelismo ao nível implementado em SIMD, ou mesmo MISD conforme anteriormente mencionado.
- Arquiteturas com co-processadores: nesta categoria se encaixam arquiteturas nas quais o processador principal é auxiliado por um co-processador na execução de instruções específicas. Este co-processador geralmente implementa *hardware* que permite acelerar a execução de suas instruções específicas, ao mesmo tempo em que as paraleliza com a execução de outras instruções pelo processador principal. Pode-se

considerar esta uma forma de SIMD, dado que as instruções do co-processador compõem o mesmo fluxo de controle porém operam paralelamente sobre dados distintos.

- Arquiteturas com múltiplos processadores no mesmo *chip*, que são os multiprocessadores MIMD conforme já explanado.
- Arquiteturas com múltiplos computadores, que correspondem à definição de multicomputador MPP MIMD.
- Arquiteturas com múltiplos nós de processamento distribuídos, que correspondem à definição de multicomputador COW MIMD.

Independente da granularidade do paralelismo, todas estas categorias foram a princípio desenvolvidas com base no *modelo de von Neumann*, o qual depende de um contador de programa para determinar a ordem de execução de instruções. Contudo, surgiram também alternativas a este modelo, baseadas na exploração do paralelismo por meio de mudanças no paradigma imposto pelo modelo de von Neumann. Neste âmbito, um modelo que foi e tem sido ainda objeto de pesquisa é o chamado *modelo de fluxo de dados (dataflow)*, no qual a disponibilidade de dados a serem processados é determinante para a evolução da execução.

As seções a seguir apresentam fundamentos teóricos de arquiteturas paralelas baseadas tanto no modelo de von Neumann quanto no modelo de fluxo de dados. Com base nestes fundamentos efetua-se uma análise crítica a respeito da sua utilidade tanto na definição da ARQPON quanto na implementação do P2ON.

Dado que a proposição do objeto deste trabalho diz respeito a uma arquitetura de **processador** para execução do PON, enfatiza-se então na fundamentação teórica o estudo e potencial aproveitamento de conceitos e técnicas de arquitetura de multiprocessadores. Portanto, os conceitos e técnicas relacionados a multicomputadores não serão abordados com profundidade, apenas apresentados brevemente na seção sobre outros modelos de arquitetura paralela (ver Seção 2.3.4).

2.3.2 Arquiteturas paralelas baseadas em processadores von Neumann

As subseções a seguir apresentam a fundamentação relativa a arquiteturas paralelas baseadas em processadores von Neumann.

São apresentados a teoria em torno do modelo de von Neumann, as diferentes formas de implementação de paralelismo alinhadas com este modelo, as questões de programação relativas ao modelo e uma reflexão sobre as vantagens e desvantagens de utilização de arquiteturas segundo este modelo.

2.3.2.1 Fundamentação teórica do modelo de von Neumann

Conforme exposto na Seção 2.1, a concepção de *software* sequencial parte do princípio de que este será executado sequencialmente, ou seja, instrução após instrução, seguindo alguma lógica de controle que defina a ordem relativa das instruções.

Este modelo de execução passou a ser tradicionalmente denominado de *modelo de von Neumann*, por ter sido inspirado na proposta elaborada por John von Neumann em 1945 para a arquitetura do computador vN-EDVAC (GODFREY; HENDRY, 1993). Na verdade, o modelo de von Neumann seria a interpretação técnica mais bem aceita do conceito de máquina de Turing devido à sua capacidade de processamento aberta (i. e., circuitaria não dedicada a uma aplicação). Isto dito, as observações sobre o modelo de von Neumann também se aplicariam em geral ao modelo de Turing.

Segundo o modelo de von Neumann, a execução de instruções ocorre de forma sequencial segundo o controle exercido por um *contador de programa* contido em uma CPU. Além disso, dois outros elementos fundamentais do modelo de von Neumann são a memória de programa, onde são buscadas as instruções que são executadas pela CPU, e uma memória global de armazenamento de dados que pode ser a mesma memória onde está armazenado o programa ou mesmo uma memória com endereçamento independente (modelo Harvard).

O valor do contador de programa é utilizado, em uma arquitetura von Neumann, para indicar o endereço da memória de programa a partir do qual a próxima instrução será buscada para execução. Esta característica de busca de instruções em memória nos mesmos moldes da busca de dados foi o grande diferencial conceitual do modelo de von Neumann em relação a modelos anteriores, nos quais não existia este conceito.

Além disso, o valor do contador de programa é atualizado seguindo a lógica do fluxo de controle ditada pela lógica de computação das próprias instruções (TANENBAUM, 2007). Ou seja, a princípio as instruções armazenadas em endereços consecutivos são executadas em sequência, com o contador de programa sendo incrementado a cada instrução, a menos que

uma destas instruções comande uma alteração no contador de programa (*branch*) fazendo com que ele passe a buscar instruções sequencialmente a partir de outro endereço de memória.

Mesmo sendo o modelo de von Neumann fundamentalmente baseado em execução sequencial, este permite a exploração de paralelismo em diversos níveis de granularidade. Por exemplo, em uma máquina de núcleo simples, com um único contador de programa, o paralelismo pode ser explorado por meio de técnicas de ILP. Multiprocessadores ou multicomputadores, por sua vez, podem definir diferentes *threads* a serem executadas por cada núcleo de processamento, sendo que cada uma destas linhas utiliza um valor próprio de contador de programa para controlar a sua sequência de execução. Denning e Dennis (2010) citam que as gerações atuais de supercomputadores ainda são fortemente dependentes de múltiplos núcleos em paralelo baseados no modelo de von Neumann.

2.3.2.2 Implementações de paralelismo baseadas em von Neumann

A seguir são apresentadas implementações de paralelismo em arquiteturas baseadas no modelo de von Neumann, categorizadas em função da granularidade do paralelismo.

2.3.2.2.1 Implementações baseadas em ILP

A forma mais comum de exploração de ILP, presente na maioria dos núcleos de processamento modernos, é a utilização de *pipelines* de múltiplos estágios.

Conceitualmente, um *pipeline* de múltiplos estágios permite que partes de instruções diferentes, consecutivas no tempo, sejam processadas simultaneamente, uma por cada estágio. Isto torna a execução mais rápida quando comparada a uma arquitetura sem *pipeline*, na qual uma instrução deveria passar por todos os estágios de processamento e ser finalizada antes da próxima instrução começar a ser executada.

Uma implementação típica de *pipeline,* bastante utilizada em diversas arquiteturas RISC (ver Seção 2.3.7), é composta pelos seguintes estágios:

- Busca (*fetch*) de instruções em memória.
- Decodificação (*decode*) da instrução, de tal maneira a identificar os operandos e enviá-los para as unidades adequadas do estágio seguinte.

- Execução (*execute*), durante o qual as unidades adequadas processam os operandos e preparam as operações de acesso à memória ou reserva de registradores que forem necessárias para a operação dos próximos estágios.
- Acesso à memória, caso previamente escalonado pela unidade de execução.
- *Write back,* que consiste em atualizações de registradores que tenham sido previamente escalonadas pela unidade de execução.

Exemplos de implementações semelhantes incluem o núcleo ARM7TDMI (ARM, 1999), com 3 estágios de *fetch, decode* e *execute* sendo que este último engloba acesso à memória ou *writeback;* a implementação do núcleo MIPS conforme apresentada por Patterson e Hennessy (2009); e a implementação do núcleo ARM9TDMI (ARM, 2012).

Considerando uma situação ideal, na qual os N estágios de um *pipeline* levam o mesmo tempo para efetuar o seu processamento e na qual estes estágios podem executar diferentes etapas de N instruções simultaneamente, o tempo gasto entre o início da execução de duas instruções consecutivas seria definido pela Equação 2 (PATTERSON; HENNESSY, 2009):

Equação 2 – Tempo entre execução de instruções em arquitetura com *pipeline*

$$T_{ip} = \frac{T_{inp}}{N}$$

Na Equação 2, T_{ip} significa o tempo entre instruções *pipelined,* T_{nip} significa o tempo entre instruções não *pipelined* e N é o número de estágios do *pipeline.*

De outra forma, considerando que cada estágio de *pipeline* idealmente efetua o seu processamento em 1 ciclo de *clock,* é possível iniciar a execução de uma nova instrução a cada novo ciclo. Portanto, na média, considerando uma sequência com uma quantidade qualquer de instruções, o número de ciclos de *clock* gastos por cada instrução (CPI) nesta condição ideal de *pipelining* tende a 1.

Contudo, na prática é comum ocorrerem situações nas quais o *pipeline* fica impossibilitado de manter o tempo teórico entre instruções calculado conforme a equação anterior, ou seja, nas quais em determinados instantes existem menos do que N estágios executando alguma etapa de alguma instrução. Estas situações são denominadas de conflitos (*hazards*) e podem ser de três tipos, conforme descrito a seguir.

1) Conflitos estruturais

Os conflitos estruturais ocorrem quando o *hardware* do processador não é capaz de suprir a demanda por recursos de 2 ou mais estágios simultaneamente. Um exemplo típico seria a situação em que o estágio de busca tenta acessar a memória, para buscar a próxima instrução, ao mesmo tempo em que o estágio de acesso à memória tenta acessá-la para escrever ou ler um dado. Para esta situação em particular, possíveis soluções arquiteturais incluem memórias *cache* separadas para dados e instruções (arquitetura Harvard) (LINHARES, 2001), mantendo a independência entre os dois tipos de acesso.

2) Conflitos de dados

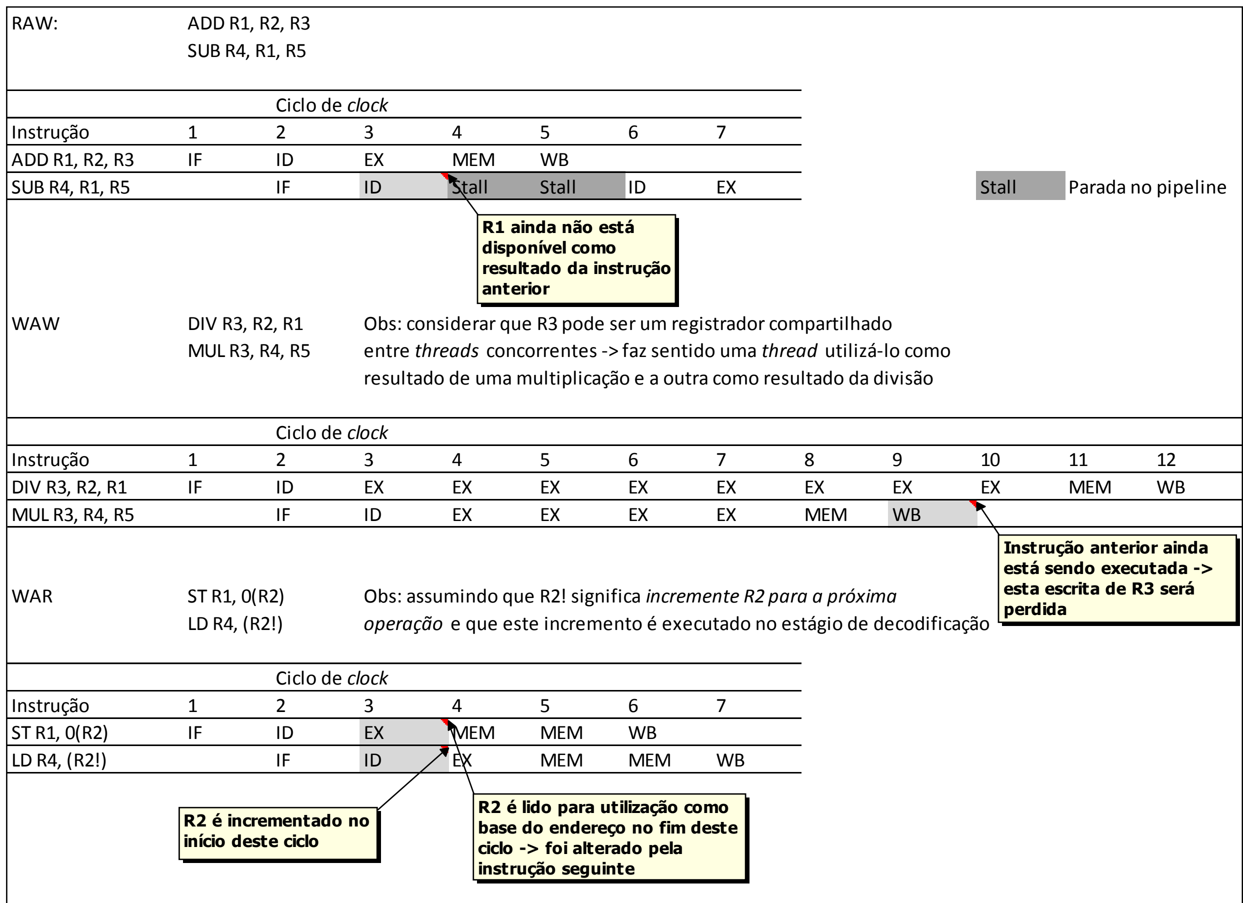
Os conflitos de dados ocorrem quando existe uma dependência entre instruções próximas, no que diz respeito ao valor de determinado dado, de tal maneira que a sua execução no pipeline alteraria a ordem temporal de acesso àquele dado, alterando a lógica do programa (HENNESSY; PATTERSON, 2007). A Figura 10 apresenta esquematicamente os três tipos possíveis de conflitos de dados.

Os conflitos de dados podem ser dos subtipos: RAW (*Read After Write*), no qual a instrução X_i tenta ler um dado antes que a instrução X_{i-1} o tenha atualizado (portanto, fazendo com que X_i leia um valor desatualizado); WAW (*Write After Write*), no qual a instrução Xi tenta escrever um dado antes que a instrução X_{i-1} o tenha escrito (portanto, mantendo o resultado de X_{i-1}, o qual está desatualizado); e WAR (*Write After Read*), no qual a instrução X_i tenta escrever um dado antes que a instrução X_{i-1} o tenha lido (portanto, fazendo com que X_{i-1} leia o novo valor).

A possibilidade de ocorrência de um determinado tipo de conflito de dados depende da arquitetura do *pipeline.* Em geral, em *pipelines* semelhantes ao modelo básico de 5 estágios previamente apresentado, a ocorrência de RAW é possível (pois o estágio de *write back* é posterior ao de acesso à memória) e a ocorrência de WAW é possível em determinadas implementações capazes de processar operações de ponto flutuante, que tipicamente demoram mais tempo do que a execução dos demais estágios.

A forma mais comum de se evitar conflitos de dados é alterar a ordem de execução, durante a geração do programa, das instruções que causam os conflitos, tomando-se o cuidado para que a lógica de execução do programa se mantenha. Este trabalho geralmente é efetuado pelo compilador e facilitado em arquiteturas RISC (*Reduced Instruction Set Computing,* ou Computação com Conjunto Reduzido de Instruções), nas quais a relativa simplicidade do funcionamento das instruções, quando comparada a arquiteturas CISC (*Complex Instruction*

Set Computing, ou Computação com Conjunto Complexo de Instruções), permite uma maior flexibilidade no reordenamento da execução. Oportunamente, aborda-se os conceitos de RISC e CISC na Seção 2.3.7.



RAW: ADD R1, R2, R3
SUB R4, R1, R5

		Ciclo de *clock*					
Instrução	1	2	3	4	5	6	7
ADD R1, R2, R3	IF	ID	EX	MEM	WB		
SUB R4, R1, R5		IF	ID	Stall	Stall	ID	EX

Stall: Parada no pipeline

WAW: DIV R3, R2, R1
MUL R3, R4, R5

Obs: considerar que R3 pode ser um registrador compartilhado entre *threads* concorrentes -> faz sentido uma *thread* utilizá-lo como resultado de uma multiplicação e a outra como resultado da divisão

		Ciclo de *clock*										
Instrução	1	2	3	4	5	6	7	8	9	10	11	12
DIV R3, R2, R1	IF	ID	EX	EX	EX	EX	EX	EX	EX	EX	MEM	WB
MUL R3, R4, R5		IF	ID	EX	EX	EX	EX	MEM	WB			

WAR: ST R1, 0(R2)
LD R4, (R2!)

Obs: assumindo que R2! significa *incremente R2 para a próxima operação* e que este incremento é executado no estágio de decodificação

		Ciclo de *clock*					
Instrução	1	2	3	4	5	6	7
ST R1, 0(R2)	IF	ID	EX	MEM	MEM	WB	
LD R4, (R2!)		IF	ID	EX	MEM	MEM	WB

Figura 10 – Exemplificação dos diferentes tipos de conflitos de dados em *pipeline*
(Fonte: autoria própria)

No entanto, técnicas de construção da arquitetura, tais como a atualização adiantada de registradores (*forwarding*) e o escalonamento dinâmico (*dynamic scheduling*) para a execução de instruções fora de ordem também podem minimizar a ocorrência de conflitos de dados. Ainda, pode-se implementar em *hardware* o algoritmo de Tomasulo (HENNESSY; PATTERSON, 2007), o qual permite escalonar e efetuar o procedimento de renomeação de registradores (*register renaming*); este procedimento visa a eliminação de conflitos de dados por meio de registradores auxiliares providos pelo *hardware*, os quais são temporariamente (e dinamicamente) renomeados para substituir os registradores principais enquanto estes ainda estão alocados para utilização por uma instrução anterior do *pipeline* (WALL, 1991).

3) Conflitos de controle

Os conflitos de controle ocorrem quando o *pipeline* inicia a execução de uma instrução de desvio (*branch*) condicional, a qual pode potencialmente alterar o fluxo de execução do programa. Caso isto ocorra, se o *pipeline* tiver iniciado a execução das instruções subsequentes à instrução de desvio, estas deverão ser descartadas para se iniciar a execução das instruções existentes no endereço de destino do desvio.

Os conflitos de controle podem ser minimizados com o uso de diversas técnicas arquiteturais, entre elas: implementação de desvios atrasados (*delayed branches*), os quais permitem que instruções úteis sejam executadas após a decodificação do desvio, porém antes da sua execução (atualização do contador de programa), sem ser descartadas por um eventual desvio tomado; predição de desvio (*branch prediction*), a qual permite que um determinado caminho seja escolhido antes da execução da instrução de desvio, utilizando-se como critério de escolha a probabilidade de o desvio ocorrer (que pode ser codificada na própria instrução) e descartando-se a execução das instruções posteriores caso ele não venha a ser efetivamente tomado.

Em relação a este último, a capacidade do *pipeline* de executar instruções de um determinado fluxo sem ter a certeza de que este fluxo será tomado, posteriormente podendo descartar o resultado se de fato não for tomado, é chamado de *execução especulativa.*

Wall (1991) efetuou experimentos de simulação para demonstrar o efeito de recursos típicos de ILP, tais como renomeação de registradores e predição de desvios, no grau de paralelismo que é possível de se obter na execução de diversos programas. Para tanto, definiu paralelismo como o número de instruções executáveis em paralelo dividido pelo número de ciclos de *clock* requeridos, sendo que cada instrução individualmente executa em um ciclo de *clock* (portanto, já considerando efeitos de *pipelining*).

Os experimentos demonstraram que, mesmo com configurações ideais (e não realistas) de *hardware* (número potencialmente infinito de unidades de execução, predição de desvio com janela de até 2048 instruções pendentes), o grau médio de paralelismo obtido ficaria em torno de 7. Além disso, o autor concluiu que o principal fator que afeta o paralelismo é a capacidade do *hardware* de efetuar predições de desvio corretas e com a maior janela possível de instruções pendentes.

Complementarmente aos conceitos fundamentais de *pipeline* apresentados, uma alternativa para a exploração de ILP parte da premissa de que é possível diminuir o número de ciclos de *clock* por instrução (CPI) para um valor abaixo de um, por meio do processamento

de mais do que uma instrução por ciclo de *clock.* Para tanto, um *pipeline* como o apresentado anteriormente deve ser modificado para, sob determinadas circunstâncias, permitir o início de mais do que uma instrução por ciclo de *clock.* Esta característica define o conceito de processador de múltipla emissão (*multiple-issue processors*).

A emissão de múltiplas instruções é obtida por meio da multiplicação do número de unidades funcionais de cada tipo no *pipeline.* Por exemplo, um processador com núcleo ARM Cortex-A9 é capaz de despachar simultaneamente até 4 instruções de sua fila interna de execução para os estágios subsequentes, que implementam conjuntos de 4 unidades lógico-aritméticas e unidades de ponto flutuante (ARM, 2009). A Figura 11 mostra os estágios de múltipla emissão do *pipeline* do ARM Cortex-A9 no canto superior direito.

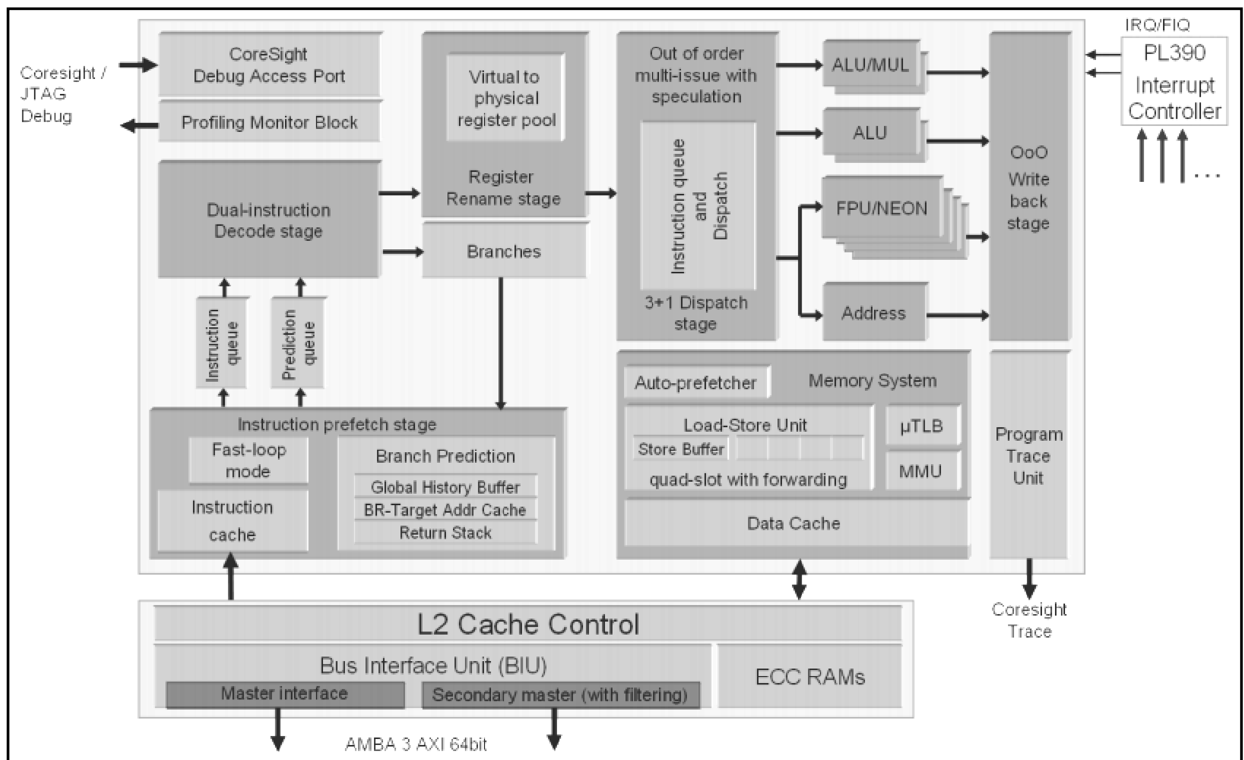


Figura 11 – Arquitetura do ARM Cortex-A9
(Fonte: adaptado de ARM, 2009)

Do ponto de vista de interface de programação, um processador de múltipla emissão pode ser superescalar ou implementar VLIW (*Very Long Instruction Words*). A diferença fundamental entre as duas abordagens é que, em um processador superescalar, as instruções são convencionais (ou seja, semelhantes às de processadores com *pipelines* mais simples), porém emitidas em conjunto, sendo que o *hardware* fica encarregado de detectar conflitos e

coordenar o despacho e execução paralelos para os estágios múltiplos, o que aumenta a sua complexidade.

Já nas arquiteturas que implementam VLIW o paralelismo está explícito no *opcode* de cada instrução, que define um conjunto de operações que são capazes de ocupar múltiplas unidades de execução paralelas simultaneamente. O gabarito (quantidade de bits) de cada instrução é geralmente maior do que instruções de arquiteturas com *pipelines* mais simples justamente por conter todas as informações (operandos e *opcodes*) a respeito das operações que se deseja executar em paralelo. Em função disso, é tarefa do compilador gerar instruções VLIW de tal forma que não causem conflitos durante a sua execução no *pipeline*, simplificando esta questão do ponto de vista de *hardware.*

Um exemplo de *pipeline* superescalar é o *pipeline* do ARM Cortex-A9 com suas múltiplas unidades de execução (ALUs e FPUs mostradas na Figura 11, canto superior direito), utilizado em uma ampla gama de aplicações embarcadas com requisitos de baixo consumo de energia. Um exemplo de *pipeline* que implementa VLIW é o da CPU Trimedia, utilizada principalmente em aplicações que exigem processamento massivo de multimídia (áudio, vídeo, etc.) (HENNESSY; PATTERSON, 2007) (TANENBAUM, 2007).

Na prática, Tullsen, Eggers e Levy (1998) demonstraram que o ganho em desempenho em arquiteturas de múltipla emissão pode ficar muito aquém do ganho que seria obtido em teoria em função do número de unidades em paralelo, devido principalmente aos seguintes fatores: interdependências entre instruções; ocorrência de operações de acesso a memória de longa latência; e dificuldade de montar sequências de instruções que possam aproveitar a capacidade de múltipla emissão sem depender de execução especulativa, dado que em média a cada 7 instruções de um programa sequencial uma delas é um *branch* (UNGERER; SILC; ROBIC, 1998).

Além dos fatores já mencionados, conforme citado por Swanson et al. (2003), a escalabilidade de processadores superescalares tende a ser afetada em função do avanço da escala de integração. Isto acontece porque processadores superescalares dependem de uma infraestrutura razoavelmente complexa de interconexão, lógica de controle complexa e estruturas inerentemente centralizadas que dependem de confiabilidade, a qual é prejudicada pelo aumento da escala de integração.

2.3.2.2.2 *Implementações baseadas em DLP e MLP*

Conforme proposto por Glew (1998) e posteriormente formalizado e analisado qualitativa e quantitativamente por Chou, Fahs e Abraham (2004), paralelismo em nível de memória (*memory-level parallelism*, ou MLP) se refere ao conjunto de técnicas implementadas em *hardware* para permitir que múltiplos acessos à memória, oriundos tipicamente de faltas no acesso à memória *cache* (*cache misses*), estejam pendentes ao mesmo tempo. Isto permite que estes acessos sejam atendidos de forma sobreposta, minimizando o tempo médio de atendimento.

O uso de MLP pode trazer resultados em termos de aumento de desempenho melhores do que algumas técnicas de ILP, particularmente para aplicações comerciais que fazem acesso intenso à memória externa e que sofrem, portanto, o efeito da latência relativa cada vez maior nestes acessos (CHOU; FAHS; ABRAHAM, 2004).

Este efeito é coloquialmente conhecido por "*memory wall*" e deve-se, principalmente, às diferentes estratégias de evolução adotadas por fabricantes de semicondutores para diferentes tipos de produtos, geralmente em função de orientações econômicas e/ou mercadológicas. Ou seja, fabricantes de processadores priorizam o aumento do desempenho por ciclo de *clock* dos seus produtos, ao passo que fabricantes de memórias (incluindo, neste caso, também dispositivos de memória magnética tais como *hard disks*) priorizam o aumento da capacidade de armazenamento dos dispositivos como consequência do aumento da densidade, com pequenos ganhos de desempenho (MCKEE, 2004).

Como resultado, a diferença de desempenho entre memórias e processadores tende a aumentar ao longo do tempo, conforme mostrado na Figura 12, sendo que a razão entre ciclo de acesso à memória e ciclo de execução em processadores (CPUs) era próxima de 1000 por volta do ano de 2007 (KOGGE et al., 2008).

A abordagem de DLP (*Data-Level Parallelism,* ou paralelismo em nível de dados) é aplicada em vários contextos, desde cálculos vetoriais massivos presentes em aplicações científicas até o processamento de dados de mídia. Isto ocorre porque, nestas aplicações, muitas vezes é possível explorar o paralelismo potencial existente entre operações aritméticas executadas em iterações independentes de laços (BURGER et al., 2004).

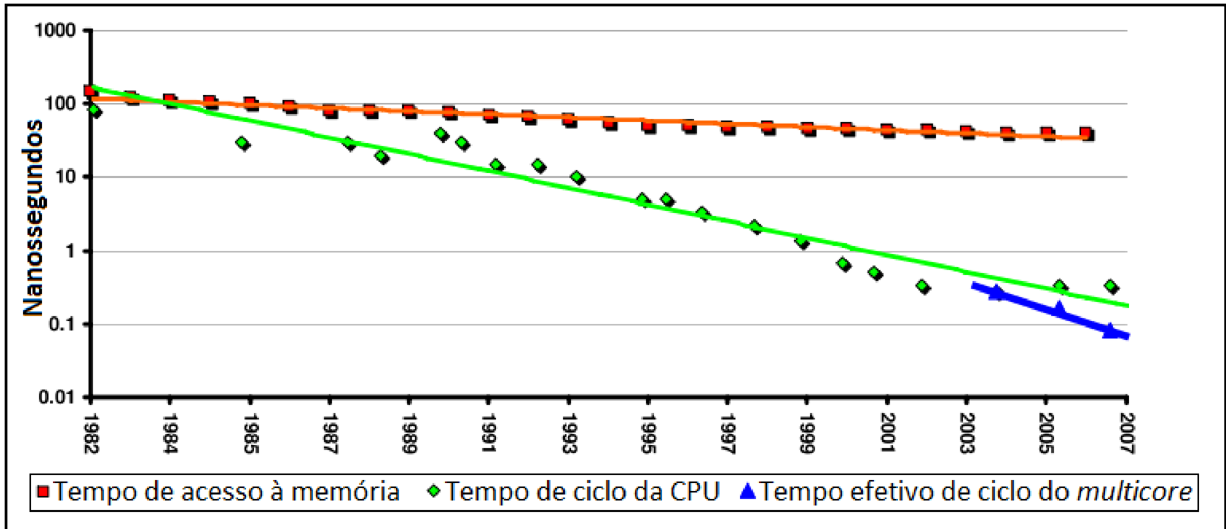


Figura 12 – Gráfico da evolução do tempo típico de acesso à memória versus tempo de ciclo em CPU (Fonte: adaptado de KOGGE et al., 2008, p. 103)

O DLP é geralmente implementado em arquiteturas SIMD, na medida em que estas propõem alimentar múltiplas unidades do *pipeline* com dados providos por uma mesma instrução. Um exemplo é a implementação das extensões SSE (*Streaming SIMD Extensions*) da arquitetura x86, introduzidas como parte do conjunto de instruções do Pentium III (e estendidas em versões posteriores) para permitir múltiplas operações simultâneas com números de ponto flutuante (PATTERSON; HENNESSY, 2009) (INTEL, 2012).

No entanto, conforme apresentado e discutido qualitativamente por Talla, John e Burger (2003), em particular para aplicações de processamento de mídia, o aumento do número de unidades de execução para aumentar o grau de DLP em arquiteturas SIMD não é a abordagem mais correta, dado que existem partes daqueles tipos de programas que tem seu paralelismo melhor explorado utilizando-se outras técnicas (tais como ILP).

2.3.2.2.3 Implementações baseadas em TLP

Dentre as técnicas que procuram explorar o paralelismo em nível de linha de execução (thread-level parallelism, ou TLP), pode-se destacar como relevantes para este estudo as técnicas de multithreading (explícito ou implícito), multiprocessamento simétrico (symmetric multiprocessing, ou SMP) e não-simétrico, máquinas RAW e Arquiteturas Associativas Auto-Distribuídas (*Self Distributed Associative Architecture,* ou SDAARC).

1) ***Multithreading***

A técnica de *multithreading* consiste basicamente em prover suporte de *hardware,* na arquitetura do processador, para a definição e gerenciamento de um conjunto de *threads* e multiplexação do uso do *pipeline* (superescalar ou não) por este conjunto. Ou seja, em determinado instante de tempo, diferentes estágios do *pipeline* podem estar ocupados processando os contextos de diferentes *threads* (UNGERER; SILC; ROBIC, 1998) (KONGETIRA; AINGARAN; OLUKOTUN, 2005) (UNGERER; ROBIC; SILC, 2003), sendo que a política de chaveamento de contexto também é implementada em *hardware.* Isto significa que os chaveamentos de contexto em uma arquitetura *multithreaded* ocorrem com frequência muito maior do que chaveamentos de contexto comandados por *software* (p. ex., por um kernel com suporte a múltiplas *threads*), podendo inclusive ocorrer a cada ciclo de *clock* (conforme será descrito a seguir, na categorização de arquiteturas *multithreaded*).

As principais motivações para a concepção da técnica de *multithreading* são as seguintes:

- Operações de alta latência de acesso à memória (por exemplo, oriundas de *cache misses*) executadas no contexto de uma *thread* podem fazer com que a execução das instruções subsequentes desta *thread* seja bloqueada até que o acesso seja finalizado, devido a dependências. No entanto, este bloqueio geralmente não influencia o fluxo de execução de outras *threads* independentes, que podem aproveitar a disponibilidade do *pipeline* minimizando a ocorrência de ciclos ociosos.
- Embora arquiteturas superescalares modernas permitam que sequências de até 32 instruções sejam emitidas por ciclo de *clock*, o aproveitamento ótimo do *pipeline* somente ocorrerá se o compilador souber aproveitar o ILP potencial da sequência de instruções em questão. No entanto, estudos anteriores sobre ILP (WALL, 1991) (UNGERER; ROBIC; SILC, 2003) reportam de 4 a 7 instruções por ciclo como sendo os números máximos reais de paralelismo obtidos, mesmo com o uso de técnicas tais como *branches* especulativos. Este problema de aproveitamento de ILP é minimizado em uma arquitetura *multithreaded*, dado que o paralelismo é obtido em uma granularidade não tão baixa, intercalando instruções de diferentes *threads* que teoricamente possuem menos interdependências.

Uma arquitetura *multithreaded* deve prover um conjunto de registradores dedicado para manter o contexto de cada uma das N *threads* que é capaz de executar simultaneamente.

Isto é necessário para minimizar o *overhead* decorrente do tempo de chaveamento de contexto, dado que este ocorre com muito mais frequência do que em outras arquiteturas. A disponibilidade de múltiplos conjuntos de registradores não elimina as penalidades oriundas do *flush* de estágios do *pipeline* que geralmente é necessário para as trocas de contexto. Contudo, geralmente estas penalidades são bastante inferiores ao tempo em que a *thread* sendo chaveada permaneceria bloqueada caso seu contexto fosse mantido, sendo então vantajoso efetuar o chaveamento.

A política de *multithreading* pode variar, dependendo da frequência e dos critérios utilizados para determinação do chaveamento de contexto. Em função desta característica pode-se identificar quatro categorias ou modelos de arquiteturas *multithreaded*, conforme segue.

***Multithreading* de granularidade fina** (ou IMT, *Interleaved Multithreading* (UNGERER; ROBIC; SILC, 2003)): neste modelo o chaveamento é executado após a busca de cada instrução (ou conjunto de instruções, se a arquitetura for superescalar).

Geralmente neste modelo a busca de uma instrução de determinada *thread* somente é efetuada quando a instrução anterior daquela mesma *thread* é finalizada. Esta providência simplifica sobremaneira o *hardware* do *pipeline,* dado que elimina a necessidade de detecções de conflitos. No entanto, a eficiência deste modelo depende de que o número de *threads* sendo escalonado seja pelo menos igual ao número de estágios do *pipeline,* do contrário este pode não ser totalmente ocupado em função do bloqueio eventual de alguma das *threads* devido a transações de acesso à memória. De fato, experimentos com IMT tais como o processador Niagara da Sun (KONGETIRA; AINGARAN; OLUKOTUN, 2005) demonstram que esta técnica prioriza o *throughput* de *threads* e não de instruções, ou seja, é mais adequada para aplicações em que exista de fato um grande número de *threads* para serem executadas em paralelo (FREITAS; ALVES; NAVAUX, 2009).

Para minimizar a perda de desempenho em função da não ocupação completa do *pipeline* devido a um número reduzido de *threads,* arquiteturas como a Tera (ALVERSON et al., 1990), lançada comercialmente como Cray MTA-2, implementam a técnica de verificação antecipada de dependências (*explicit-dependence lookahead*). Esta técnica consiste em incluir no *opcode* de cada instrução um conjunto de bits indicando quantas instruções a seguir podem ser inseridas e executadas no *pipeline* sem gerar conflitos com a

instrução atual. Ou seja, esta técnica permite uma melhor utilização do *pipeline* por uma única *thread* ou por um conjunto reduzido de *threads* sem torná-lo necessariamente mais complexo.

Outros exemplos de implementações de IMT são as arquiteturas P-RISC (NIKHIL, 1989), HEP (SMITH, 1986 *apud* NIKHIL, 1989) e Horizon; estas duas últimas serviram como base para a arquitetura do computador Cray MTA (UNGERER; ROBIC; SILC, 2003). O P-RISC apresenta instruções nativas para criação de *threads* que podem, inclusive, ser utilizadas para simular o comportamento de programas de fluxo de dados (ver Seção 2.3.3.1), no sentido de sincronizar a execução de determinada *thread* em função da disponibilidade de dados de entrada, bloqueando esta *thread* e escalonando outras segundo IMT enquanto os dados de entrada não estiverem disponíveis.

***Multithreading* de granularidade grossa** (ou *block interleaving*): neste modelo o chaveamento é executado quando ocorrem instruções que causam longa latência de acesso à memória.

Algumas das estratégias utilizadas para a decisão de se realizar o chaveamento são: chavear em todas as leituras de dados da memória (*switch on load*); chavear quanto tentar efetuar a leitura/escrita e o acesso não puder ser completado imediatamente, p. ex., em uma ocorrência de *cache miss* (*switch on use*). É pertinente ressaltar que, nesta última estratégia, o uso de memórias *cache* tende a diminuir o número de chaveamentos, pois filtra boa parte das operações com alta latência.

***Multithreading* simultâneo** (*simultaneous multithreading,* ou SMT) ou superescalar: consiste em uma variação de uma arquitetura superescalar na qual são aplicadas técnicas de *multithreading*. Ou seja, em uma arquitetura que implementa *multithreading* simultâneo os diferentes estágios de execução superescalares do *pipeline* podem ser ocupados por instruções de diferentes *threads* simultaneamente (UNGERER; ROBIC; SILC, 2003).

Experimentos comparativos realizados por Tullsen, Eggers e Levy (1998), em arquiteturas SMT com suporte de *hardware* para até 8 *threads* executando simultaneamente de forma superescalar, demonstram que arquiteturas SMT apresentam ganho no *throughput* de instruções quando comparadas a arquiteturas com múltiplos núcleos com características semelhantes (mesma largura de banda de emissão de instruções). Isto ocorre em função do particionamento dinâmico das unidades de execução disponíveis entre as instruções a serem

executadas, ou seja, o mapeamento existente entre uma *thread* e a unidade de execução designada para executá-la é definido dinamicamente em função da situação de execução global dos diversos contextos simultâneos, minimizando conflitos.

O conceito de SMT é aplicado em algumas arquiteturas da Intel (p. ex. famílias de processadores baseadas nos núcleos *Intel Core i3, i5* e *i7*, bem como *Intel Xeon*) e batizado, neste contexto, de *Hyperthreading* (PILLA; SANTOS; CAVALHEIRO, 2009) (INTEL, 2012).

A título de comparação, a Figura 13 exemplifica o funcionamento do *pipeline* de uma arquitetura *multithreaded,* diferenciando os efeitos de *multithreading* de granularidade fina, de granularidade grossa e simultâneo. O *pipeline* considerado para o exemplo é superescalar de dupla emissão, ou seja, idealmente capaz de executar duas instruções simultaneamente. A dimensão horizontal corresponde ao tempo (p. ex., cada ciclo de *clock*), ao passo que a dimensão vertical corresponde ao *slot* de emissão do *pipeline.* O exemplo apresenta a execução de 3 *threads,* cada uma representada por um diferente tom de cinza.

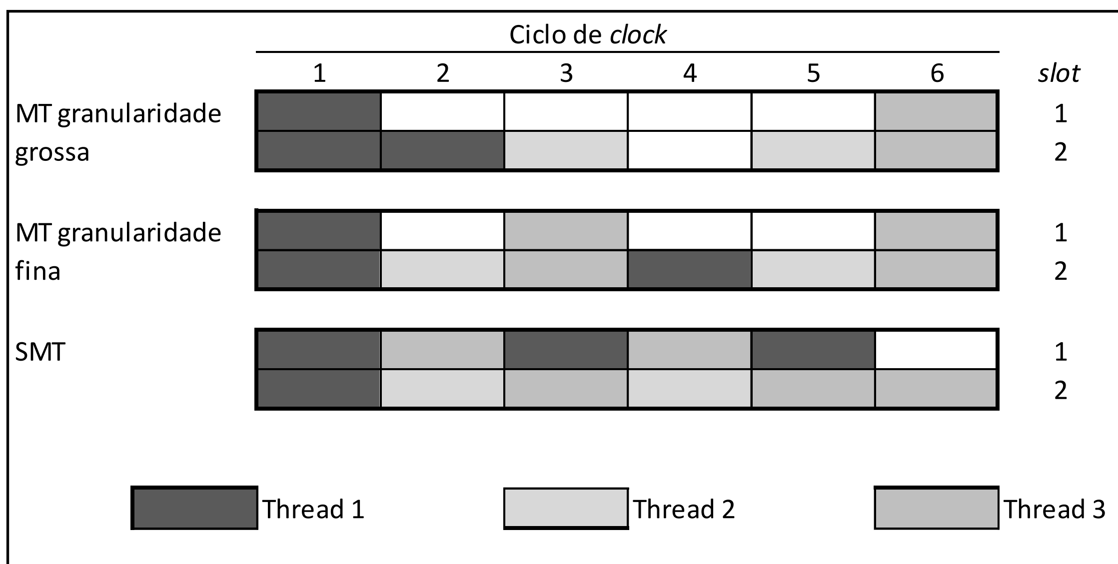


Figura 13 – Exemplos de *multithreading* de granularidade grossa, fina e simultâneo
(Fonte: adaptado de HENNESSY; PATTERSON, 2007, p. 725)

Percebe-se, por este exemplo, que as soluções de *multithreading* de granularidade fina e grossa tendem a subutilizar os estágios do *pipeline* (retângulos brancos na figura), embora este efeito seja um pouco menos acentuado com granularidade fina porque eventuais bloqueios ou conflitos intra-*thread* são mascarados; por exemplo, a Thread 2 não pode ocupar

dois estágios simultaneamente, gerando uma parada no *pipeline* no ciclo de *clock* 4 da versão com granularidade grossa, no entanto isto não ocorre na granularidade fina porque a cada ciclo uma nova *thread* é buscada para ocupar os estágios do *pipeline.*

Na versão SMT, por sua vez, a ocupação é mais homogênea dado que, em um mesmo ciclo, *threads* diferentes podem ocupar os diferentes *slots* de emissão do *pipeline.*

***Multithreading* implícito**: neste modelo o *hardware* do processador é capaz de isolar e gerar dinamicamente linhas de execução concorrentes, a partir de uma única *thread,* e executar estas linhas especulativamente de forma concorrente à *thread* original (UNGERER; ROBIC; SILC, 2003). Ou seja, a capacidade de execução concorrente implícita ao programa é identificada dinamicamente, sem o auxílio de um compilador.

Um exemplo de arquitetura que implementa *multithreading* implícito é a arquitetura do processador *multithreaded* especulativo proposto por Marcuello, Gonzalez e Tubella (1998). Esta arquitetura é capaz de efetuar a busca de várias janelas de instruções, não necessariamente contíguas, baseadas na análise de *branches* que são altamente predizíveis (p. ex. *branches* que finalizam laços) e executar especulativamente estas janelas de instruções.

As arquiteturas que implementam *multithreading* implícito tendem a ser complexas, pois necessitam mapear dependências de dados entre as diversas janelas de instruções e efetuar predições dos valores destes dados para evitar latências ocasionadas pela espera pelo cálculo efetivo dos seus valores.

2) **Multiprocessamento simétrico**

A técnica de multiprocessamento simétrico (*symmetric multiprocessing,* ou SMP), também conhecida por CMP (*chip multiprocessing*), consiste em implementar dois ou mais núcleos de processamento idênticos (em uma mesma pastilha, no caso dos CMP) interligados a uma memória compartilhada por meio de um barramento de alto desempenho.

Neste modelo de arquitetura, cada núcleo é capaz de executar qualquer *thread* disponível para execução, inclusive efetuando trocas entre os diversos núcleos para fins de balanceamento de carga. Esta abordagem de implementação de paralelismo é mais adequada para aplicações que exibem mais TLP do que ILP, tais como servidores comerciais que trabalham com conjuntos grandes de dados e baixa localidade de acesso, gerando altas taxas de *cache miss* (KONGETIRA; AINGARAN; OLUKOTUN, 2005). Ou seja, é possível obter

bons ganhos em execução no modelo SMP mesmo utilizando núcleos com *pipelines* mais simples, com menor grau de ILP.

Ungerer, Robic e Silc (2003) citam, a favor das técnicas de multiprocessamento baseadas em CMP, que desenvolver novos núcleos ou adaptar núcleos existentes para uma filosofia tal como *multithreading* tende a ser relativamente caro. Sendo assim, a tendência é que multiprocessadores modernos aproveitem núcleos já existentes sem alterações, interligando-os segundo a técnica CMP de tal forma a implementar o multiprocessamento.

Exemplos de SMP estão atualmente disponíveis em diversos microprocessadores *multicore* utilizados comercialmente, tais como: linha *Intel Core i3, i5* e *i7,* utilizada em PCs e *notebooks* e baseada em 2 a 4 núcleos Intel 64 (INTEL, 2012); e *Samsung Exynos 4 Quad,* utilizado em *smartphones* e baseado em 4 núcleos ARM Cortex-A9 (SAMSUNG, 2012).

3) **Multiprocessamento não simétrico**

Um exemplo de arquitetura que implementa multiprocessamento não simétrico, baseado em um conjunto de processadores não idênticos (p. ex. co-processadores), é a IBM Cell.

A arquitetura Cell, proposta pela IBM (GSCHWIND, 2006), pode ser considerada uma arquitetura baseada em conceitos do modelo de von Neumann, porém posicionada em relação a outras arquiteturas de maneira a favorecer o desempenho e a eficiência na dissipação de potência em detrimento da programabilidade / generalidade (SCOTT, 2007). Esta arquitetura foi concebida visando o seu uso em uma ampla gama de plataformas e aplicações, principalmente aquelas com necessidade de alto desempenho e responsividade ao usuário e às comunicações em rede, tais como plataformas de jogos e multimídia (p. ex. Playstation 3).

Kahle et al. (2005) citam que arquitetura do Cell é baseada na cooperação entre um núcleo central de 64 bits, concebido segundo a arquitetura POWER (PPE), com um conjunto de oito SPEs (*Synergistic Processor Elements*). O PPE permite emissão dual no modo SMT (duas *threads* simultaneamente) e execução de instruções de manipulação de vetores de dados (SIMD), ao passo que os SPEs são otimizados para interface com memória por meio de DMA. Os SPEs são interligados por meio de uma NoC (ver Seção 2.3.6), implementada em anel, capaz de sustentar taxas de intercomunicação de até 200 GB/s a um *clock* de 3.2 GHz (GSCHWIND, 2006).

Cada SPE opera sobre uma memória local *on-chip*, para a qual são copiados dados e instruções que serão utilizados por aquele SPE e sobre a qual o SPE tem controle explícito.

Segundo Kahle et al. (2005), esta organização arquitetural tem como principal objetivo combater os efeitos da "*memory wall"* que afetam o desempenho.

Do ponto de vista de programação, a arquitetura do Cell favorece modelos nos quais os SPEs sejam utilizados como aceleradores para execução de funções que são críticas em termos de desempenho, ou mesmo concentrem a execução da maior parte do *software* e sejam controlados/ sincronizados por funcionalidade executada pelo PPE.

4) **Máquinas RAW**

As máquinas RAW (WAINGOLD et al., 1997) são baseadas em uma arquitetura relativamente simples, cujos detalhes de implementação do *hardware* são conhecidos e amplamente acessíveis por meio do seu conjunto de instruções. Isto permite que os compiladores para máquinas RAW possam alocar recursos da melhor forma possível para cada aplicação, dado que possuem informações sobre a estrutura interna do *hardware* e acesso à alocação/programação dos seus recursos com baixa granularidade. A Figura 14 ilustra a organização lógica interna de uma máquina RAW com os blocos básicos anteriormente descritos.

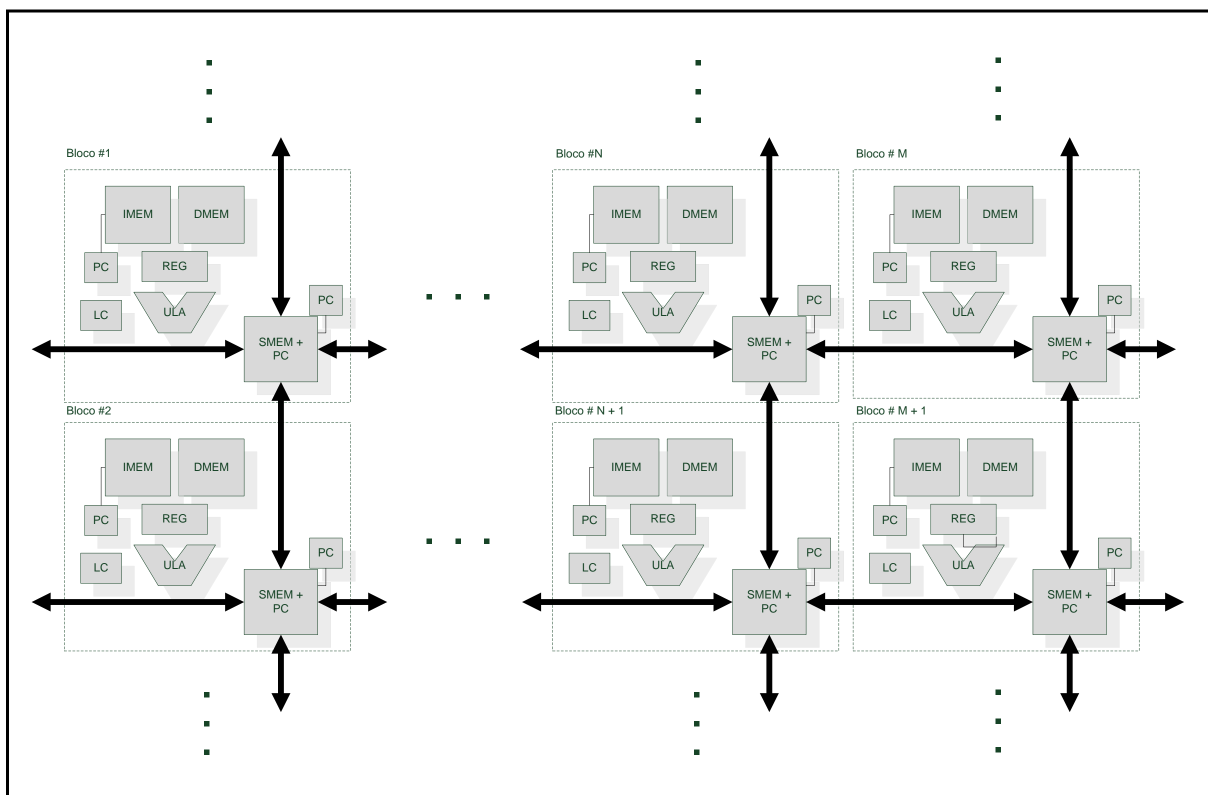


Figura 14 – Esquema de uma máquina RAW
(Fonte: adaptado de WAINGOLD et al., 1997, p. 87)

O *hardware* de uma máquina RAW é geralmente composto por blocos (*tiles*). Cada um dos blocos contém uma Unidade Lógico-Aritmética (ULA), memórias de dados (DMEM) e de instruções (IMEM), registradores (REG), alguma lógica reconfigurável (LC) e uma chave (*switch*) programável com memória (SMEM) e contador de programa próprios. Os blocos estão interconectados aos seus blocos vizinhos por meio de uma rede *on-chip* (NoC, ver Seção 2.3.6) em uma granularidade muito mais fina do que a obtida em um multiprocessador comum, viabilizando, por exemplo, a comunicação direta em alta velocidade entre ULAs de diferentes blocos.

Waingold et al. (1997) apresentaram a expectativa de que em 10 ou 15 anos a evolução tecnológica (aumento da densidade, técnicas de compilação mais avançados, entre outras) favoreceria a relação custo-benefício das arquiteturas RAW em relação a arquiteturas tradicionais e, desta forma, incentivaria a aplicação do seu conceito no desenvolvimento de microprocessadores modernos. No entanto, passados 15 anos desde a publicação do seu trabalho (em 1997), o conceito das arquiteturas RAW não se consolidou comercialmente.

Wentzlaff et al. (2007) e a empresa Tilera aproveitaram o conceito de *tiles* introduzido por Waingold et al. (1997) para propor o *Tile Processor* (Processador de Blocos). O processador de blocos pode ser considerado uma máquina RAW no sentido de que utiliza uma rede 2D *mesh* para interconexão entre os blocos, sendo esta rede acessível pelo usuário no sentido de permitir que dados sejam programaticamente transferidos entre os blocos, sem a intervenção de *software* de sistema, por meio do uso de uma API específica (iLib). Segundo os autores, este é um dos grandes diferenciais em relação às arquiteturas típicas de multicomputador nas quais a comunicação entre diferentes *threads* é efetuado unicamente por meio de memória compartilhada.

5) **Arquiteturas Associativas Auto-Distribuídas**

As Arquiteturas Associativas Auto-Distribuídas (*Self Distributed Associative Architecture,* ou SDAARC) (ESCHMANN et al., 2002) consistem em *hardware* multiprocessado no qual cada núcleo de processamento executa uma *microthread*, que se constitui em uma sequência de instruções com alto índice de localidade no acesso à memória (portanto, com potencial de minimizar latências relativas a *cache misses*).

As *microthreads* são determinadas em tempo de compilação, a partir de *software* de alto nível composto por uma única *thread*, e relacionadas umas às outras por meio de um grafo de fluxo de dados que expressa o paralelismo extraído automaticamente na etapa de compilação. Neste grafo, os nós representam as *microthreads* e os arcos representam o canal

de comunicação entre as *microthreads.* Um escalonador implementado em *hardware* distribui as *microthreads* entre os núcleos de processamento disponíveis levando em consideração as relações de dependência entre elas, permitindo o paralelismo e buscando otimizar, com esta distribuição, os custos de comunicação.

O armazenamento em memória de uma implementação SDAARC segue o modelo COMA (*Cache-Only Memory Architecture*, ou Arquitetura de Memória Somente de *Cache*). Segundo este modelo, blocos de dados / instruções não possuem um endereço permanente em memória, podendo ser movimentados para o espaço de memória próximo de cada processador à medida que são utilizados por aquele processador. Esta organização é levada em consideração pelo escalonador quando da distribuição e movimentação dos dados e instruções de cada *microthread.*

2.3.2.3 Modelos e linguagens de programação para arquiteturas paralelas von Neumann

Esta seção apresenta conceitos relacionados aos modelos de programação utilizados para a concepção de *software* paralelo segundo o modelo de von Neumann.

Um modelo de programação adequado para a implementação de *software* paralelo executável em arquiteturas von Neumann deve levar em consideração as seguintes questões (KOGGE et al., 2008):

- Formas ou técnicas para particionamento (segmentação) do programa em partes que sejam paralelizáveis e gerenciamento destas partes.
- Formas ou técnicas de efetuar comunicação e sincronização entre as partes paralelizáveis, permitindo a sua cooperação.

Do ponto de vista de particionamento da execução, este pode ser efetuado por meio da definição implícita ou explícita das *threads* que compõem o programa. A definição implícita ocorre por meio do uso de paralelizadores automáticos, que geralmente são compiladores que incorporam ao processo de compilação as fases de detecção de paralelismo e alocação de recursos. Ferlin (2008) lista alguns exemplos de compiladores paralelizadores e também cita a linguagem SISAL (SISAL, 2012) como exemplo de linguagem que dá suporte a paralelização implícita na etapa de compilação. As técnicas a serem aplicadas na paralelização implícita são altamente dependentes de características da arquitetura da

plataforma de *hardware* (superescalar, SMP, etc.) sobre a qual será executado o *software* paralelizado.

A seu turno, a definição explícita de *threads* se dá por meio de APIs específicas da linguagem/plataforma na qual se está desenvolvendo o programa. Alguns exemplos disto são os eu seguem: POSIX (*Portable Operating System Interfaces*) (BARNEY, 2012), especificação implementada por diversos sistemas operacionais baseados no UNIX, tais como o Linux e o Solaris; *Java Threads API*, conjunto de pacotes e classes em linguagem Java para a implementação de programas com múltiplas *threads* e a sua posterior execução em uma Máquina Virtual Java (*Java Virtual Machine* - JVM) (DEITEL; DEITEL, 2010); *Microsoft Windows API,* a qual define funções em linguagem C para a manipulação de *threads* (MICROSOFT, 2012).

Em todos os exemplos anteriores é obrigação do desenvolvedor definir as *threads* explicitamente em alto nível, porém são as camadas inferiores da plataforma (principalmente o sistema operacional) que se encarregam de escalonar a sua execução. Esta pode ocorrer de forma compartilhada em um único núcleo de processamento (portanto havendo concorrência) ou ser dividida entre um conjunto de múltiplos núcleos disponíveis (portanto havendo paralelismo). Dito de outra forma, o desenvolvedor não exerce um controle preciso sobre como a execução concorrente e/ou paralela ocorre de fato, muito embora em alguns sistemas operacionais, p. ex. Microsoft Windows, haja formas de se sugerir um núcleo preferencial para execução de cada uma das *threads*.

Em contraposição, foram desenvolvidas outras tecnologias/linguagens de programação que, embora ainda baseadas em abstrações de alto nível, permitem um controle mais efetivo sobre a execução paralela. Estas linguagens geralmente oferecem formas de otimização da comunicação entre *threads*, principalmente do ponto de vista de exploração da localidade no acesso à memória; ou seja, permitem maximizar o número de acessos a porções da memória que apresentem o menor tempo de acesso possível (o que ocorre, por exemplo, no processamento em arquiteturas do tipo NUMA).

Isto ocorre por meio de particionamento explícito das regiões de memória compartilhada, utilizando-se modelos tais como PGAS (*Partitioned Global Address Space).* Segundo este modelo, um conjunto de *threads* concorrentes acessa um espaço de memória comum, porém particionado, no qual pode-se explorar localidade por meio de alocação de dados específicos de cada *thread* em partições específicas ou definir dados compartilhados por meio de *arrays* globais (com fragmentos em múltiplas partições). Além disso, no modelo

PGAS é possível definir dados que fazem referência a outros dados em partições diferentes (EL-GHAZAWI, 2009).

Alguns exemplos de linguagens de programação que apresentam abstrações de alto nível para a programação paralela são: UPC (*Unified Parallel C*) (EL-GHAZAWI, 2009) (CHAUVIN et al., 2005); HPF (*High Performance Fortran*) (DIGITAL..., 2012); OpenMP (*Open Multi-Processing*) (PAS, 2005) (PAS, 2009); IBM X10 (HUDAK, 2011) (DUESTERWALD et al., 2010); Cray Chapel (CHAMBERLAIN; CALLAHAN; ZIMA, 2007); Sun Fortress (FELDMAN, 2008) (ORACLE, 2012); Cilk (RANDALL, 1998) (LEISERSON, 2006); Co-Array Fortran (CAF); e Titanium for Java. Em particular, UPC, X10, Chapel CAF e Titanium oferecem suporte à programação segundo o modelo PGAS.

Neste ponto é pertinente lembrar que o conceito de processo, muito utilizado no contexto da computação paralela, nada mais é do que a abstração representada por um conjunto de *threads* executando em um espaço bem determinado e protegido de endereçamento de memória e com um conjunto de recursos (p. ex. *handles* de arquivos) explicitamente e, muitas vezes, exclusivamente alocados para si. Portanto, ao se definir o modelo de programação, do ponto de vista de particionamento e de comunicação, automaticamente se define também o modelo de processo.

Uma variação do modelo de particionamento e comunicação baseado em processos é apresentada pelo Servo (ZEA; SARTORI; KUMAR, 2008). Este se constitui em um modelo de programação orientado a serviço, segundo o qual um programa é decomposto em componentes que oferecem serviços potencialmente executáveis em paralelo. A comunicação não é definida explicitamente entre componentes, mas sim entre serviços, os quais são ofertados e/ou requisitados pelos componentes.

Os mecanismos arquiteturais e de tempo de execução para implementação do Servo devem oferecer a funcionalidade de controle e descoberta de serviços, de tal forma a oferecer esta abstração para os componentes do programa. Isto também permite, conceitualmente, a implementação de facilidades tais como a migração e replicação de serviços durante a execução.

Do ponto de vista de comunicação e sincronização, pode-se categorizar os modelos de programação como sendo baseados em troca de mensagens ou baseados em compartilhamento de memória.

O modelo de troca de mensagens é, tipicamente, implementado em multicomputadores (ver Seção 2.3.1), os quais não possuem um espaço de endereçamento

comum de memória e, portanto, dependem de um mecanismo distinto para implementar comunicação e sincronização entre *threads.* Duas das especificações mais comuns para implementação do modelo de troca de mensagens são a MPI (*Message Passing Interface*) (ARGONNE, 2012) e a PVM (*Parallel Virtual Machine)* (OAK, 2012). Ambas definem um conjunto de APIs e ferramentas de *software* que permitem a implementação de comunicação entre *threads,* de forma explícita ou transparente ao ambiente de execução paralela subjacente.

O modelo de troca de mensagens apresenta como pontos positivos a possibilidade de se controlar a comunicação (e, por consequência, a distribuição de dados e eventualmente até mesmo da carga de processamento) programaticamente de forma intrinsecamente sincronizada, o que é garantido pela utilização da API de mensagens. Como pontos negativos, o uso de troca de mensagens requer a inserção de código específico para chamadas das APIs no código de aplicação, o que pode induzir a erros de programação. Além disso, a comunicação em si acarreta *overhead*, que é tão mais significativo quanto menores em tamanho e maiores em quantidade forem as transações de comunicação (EL-GHAZAWI, 2009).

O modelo baseado em compartilhamento de memória, por sua vez, é tipicamente implementado em multiprocessadores (ver Seção 2.3.1). Isto ocorre porque este modelo depende conceitualmente de que os diversos núcleos de processamento da arquitetura tenham acesso a um mesmo espaço de endereçamento de memória, de tal maneira que as diversas *threads* sendo executadas possam efetuar escritas e leituras em dados que são compartilhados entre elas. Este compartilhamento viabiliza que a comunicação e a sincronização entre as *threads* seja efetuada justamente por meio da coordenação entre as escritas/leituras dos dados compartilhados.

O modelo baseado em compartilhamento de memória apresenta como pontos positivos a simplicidade de codificação na aplicação, visto ser fundamentado em primitivas de escrita e leitura de dados. Como pontos negativos, a manipulação de dados compartilhados requer a definição de mecanismos explícitos de sincronização (para evitar problemas de concorrência tais como *race conditions,* ou atualizações perdidas), além do que a abstração de um espaço de endereçamento único não necessariamente leva em conta questões de localidade de acesso, portanto impedindo a exploração desta questão para melhorar o desempenho.

Em relação a este último ponto negativo, dado que a arquitetura da memória compartilhada nos multiprocessadores pode variar, tanto em número de níveis (presença ou não de *cache* e de quantidade de níveis) quanto na relevância da localidade de acesso (p. ex.

em arquiteturas NUMA), a estratégia de compartilhamento de memória deve levar estas questões em consideração. Neste aspecto, pode-se citar novamente as ferramentas (linguagens) que dão suporte a PGAS.

Comparando-se os modelos de comunicação e sincronização sob o viés da arquitetura, as técnicas de troca de mensagens tendem a simplificar o *hardware* porque concentram os problemas relativos a manter a coerência e consistência dos dados, decorrentes da concorrência, na implementação do protocolo de baixo nível de troca de mensagens. As técnicas de compartilhamento de memória, por sua vez, tendem a exigir *hardware* mais complexo para manter a coerência e consistência entre múltiplos acessos concorrentes à área de memória compartilhada. Em arquiteturas mais avançadas este mecanismo se traduz, principalmente, na movimentação de linhas de memória *cache* entre os diferentes nós de processamento (núcleo + memória cache) utilizando-se protocolos de coerência de *cache* (HAMMOND et al., 2004). A Seção 2.3.5.1 discorre sobre estes protocolos.

2.3.2.4 Considerações sobre arquiteturas paralelas von Neumann

A característica marcante do modelo de execução de von Neumann é a busca de instruções armazenadas em memória. Sendo assim, o desempenho de execução de programas sequenciais ou paralelos em máquinas von Neumann é dependente do desempenho de acesso à memória.

Esta dependência traz uma dificuldade, inerente à característica de taxa de transferência limitada de dados entre processador e memória, que é conhecida como “gargalo de von Neumann” (KOGGE et al., 2008). Como consequência deste gargalo, os computadores modernos gastam boa parte do seu tempo de processamento movimentando dados e aguardando pelo término destas operações, ao invés de executar operações de computação de fato. Também em função desta característica, as aplicações com pior desempenho relativo geralmente são aquelas que necessitam de mais memória para funcionar, em todos os níveis da hierarquia de memória (KOGGE et al., 2008).

Murphy (2007) pondera que o desempenho de programas em arquiteturas von Neumann, em particular de múltiplos núcleos, é muito mais afetado pela latência de acesso à memória (tempo entre requisição de acesso e disponibilidade dos dados) do que pela banda (velocidade de transferência do conjunto de bytes que compõem uma requisição de acesso), pois os processadores tendem a não manter o barramento de memória ocupado todo o tempo

devido ao tempo gasto com cálculos de endereçamento (ou seja, tendem a não aproveitar recursos de *pipelining* oferecidos pelos chips de memória para otimizar acessos sequenciais sem intervalos entre si). Além disso, programas típicos que trabalham essencialmente com dados inteiros são mais afetados do que programas típicos que trabalham com ponto flutuante, pois tendem a possuir mais *branches* e, por consequência, gerar um maior número de *cache misses*.

Do ponto de vista de evolução do desempenho, a dependência do modelo de von Neumann em relação a acessos a memória torna-se um problema em função da questão da *memory wall* (ver Seção 2.3.2.2.2). Esta questão afeta até mesmo as arquiteturas de processadores mais recentes, pois embora os fabricantes procurem inserir cada vez mais núcleos de processamento em um mesmo multiprocessador, a disparidade cada vez maior entre o desempenho de um núcleo de processamento e o tempo de acesso à memória torna difícil disponibilizar os dados aos núcleos na taxa em que eles podem processá-los. Este problema é exacerbado em aplicações que processam grandes quantidades de dados, p. ex. algumas aplicações de engenharia envolvendo a identificação de padrões (MOORE, 2008).

Para tentar minimizar este problema, diversas técnicas de otimização do tempo de acesso à memória tem sido propostas e desenvolvidas, ressaltando-se o uso de técnicas que aproveitem a questão da localidade espacial dado que a latência no acesso à memória é influenciada justamente pela localidade (proximidade física) das memórias em relação ao núcleo de processamento que as acessa. Sendo assim, aumentar a localidade é uma providência que pode trazer ganhos sensíveis para o desempenho do acesso (BORKAR; CHIEN, 2011) (ASANOVIC et al., 2006).

Neste sentido, destaca-se o desenvolvimento de soluções com espaços de endereçamento não uniformes (MCKEE, 2004) e a evolução das tecnologias de memória *cache* (ver Seção 2.3.5.1). No entanto, particularmente no domínio dos multicomputadores, Catanzaro et al. (2010) citam que o aumento do número de núcleos de processamento leva a uma necessidade de se ajustar os protocolos de coerência de *cache* (ver Seção 2.3.5.1) para esta nova realidade, na qual as operações de sincronização de memória compartilhada tendem a ser mais intensivas em função da escala.

Além das técnicas que procuram aumentar a localidade espacial, algumas arquiteturas de processadores empregam técnicas não convencionais para tentar tornar mais eficiente o acesso à memória, tais como os processadores com controle de memória *on-chip* (p. ex. IBM Cell), máquinas RAW e máquinas baseadas em processadores integrados aos chips de DRAM (PIM, ou *processing-in-memory*). No entanto, segundo Kogge et al. (2008),

embora promissoras do ponto de vista teórico, nenhuma destas técnicas alcançou viabilidade comercial.

Do ponto de vista de modelos de programação adaptados a arquiteturas paralelas von Neumann, estes direcionam a estruturação (manual ou automática) dos programas em diferentes *threads*, dado que os modelos impõem a execução sequencial de cada uma destas linhas segundo um contador de programa. Esta abordagem apresenta como vantagem facilitar o desenvolvimento de uma aplicação que pretende aproveitar o paralelismo, no sentido de que permite ao desenvolvedor dividir o programa em diferentes sequências lógicas a serem executadas em paralelo, abordando cada uma delas individualmente.

No entanto, conforme já citado, a paralelização em várias *threads* impõe formas de comunicação e sincronização entre elas (HAAVIND, 2008) (UNGERER; SILC; ROBIC, 1998). Além disso, o melhor aproveitamento do paralelismo disponível é uma relação de compromisso entre *threads* pequenas o suficiente para otimizar o uso dos recursos de *hardware* disponíveis, minimizando conflitos (p. ex. em arquiteturas *multithreading*), e entre *threads* grandes o suficiente para minimizar o *overhead* gerado pela sincronização e comunicação (ARVIND; NIKHIL, 1990).

Mesmo considerando-se especificamente as técnicas de ILP, a eficiência da execução é fortemente afetada pela ordem da execução das instruções, a princípio determinada pelo programador quando da construção de um algoritmo. Isto ocorre porque, embora o compilador e mesmo o *pipeline* de execução tenham a capacidade de reordenar as instruções de forma a obter um melhor desempenho, em essência a ordem final de execução (término das instruções) é determinada pela semântica original do algoritmo (UNGERER; SILC; ROBIC, 1998). Este problema é minimizado quando o algoritmo é implementado em linguagem ou técnica de alto nível apropriada, de maneira a definir fluxos de execução paralelos e relativamente independentes que não dependem de reordenação entre si para serem finalizados.

2.3.3 Arquiteturas paralelas baseadas em fluxo de dados

As seções a seguir apresentam a fundamentação teórica a respeito de arquiteturas baseadas em fluxo de dados (*dataflow*). Mais precisamente, são abordados a fundamentação teórica e os diferentes modelos e implementações de fluxo de dados, as questões relativas a concorrência e sincronização, a interação de arquiteturas de fluxo de dados com as hierarquias

de memória, os modelos e linguagens de programação para fluxo de dados e, finalmente, uma reflexão sobre a utilização das técnicas de fluxo de dados para concepção de *software* paralelo / concorrente.

2.3.3.1 Fundamentação teórica do modelo de fluxo de dados

A principal diferença entre o modelo de fluxo de dados e o modelo de von Neumann reside no fato de que as arquiteturas de fluxo de dados dependem da disponibilidade dos dados para buscar e executar as instruções, ao passo que as arquiteturas von Neumann dependem da disponibilidade das instruções (ou da possibilidade de buscá-las e executá-las) para posteriormente buscar e processar os dados (CARLSTRÖM; BODÉN, 2004).

Diferentemente do modelo de von Neumann, no qual a evolução da execução do programa é ditada pela variação no contador de programa (o que se denomina tecnicamente de fluxo de controle), no modelo de fluxo de dados (ou *dataflow*) o fator determinante da evolução da execução do programa é o fluxo de dados entre instruções, conforme a própria denominação do modelo já indica. Ou seja, enquanto no modelo de von Neumann o *software* concorrente é estruturado como um conjunto de *threads* [2], no modelo de fluxo de dados a estruturação parte do princípio de que a execução paralela ocorre, em alto nível, potencialmente entre instruções individuais.

Isto se deve ao fato de que, no modelo de fluxo de dados, a sincronização entre instruções é implícita ao modelo e ocorre por meio das relações de dependência de uma instrução para com os seus operandos, gerados por outra(s) instrução(ões) (IANNUCCI, 1988). Ou seja, no modelo de fluxo de dados uma instrução somente é habilitada para ser executada quando todos os operandos (*tokens*) dos quais depende estão disponíveis. Estes operandos são manipulados pela instrução, gerando resultados (*tokens*) de saída dos quais, por sua vez, outras instruções dependem para ser habilitadas.

A Figura 15 exemplifica, na forma de um grafo, a dinâmica de um programa de fluxo de dados que calcula o resultado de uma função com dois argumentos de entrada (*a* e *b*) e dois resultados de saída (*x* e *y*). As instruções são representadas na forma dos nós do grafo, ao

---

[2] Alguns autores, por exemplo Iannucci (1988), utilizam o termo *task* (tarefa) alternativamente a *thread.* No entanto, do ponto de vista da definição de qual seria a "unidade básica de execução paralela" em um modelo de programação de alto nível (portanto, excluindo ILP) os dois termos são equivalentes.

passo que os arcos representam o fluxo de dados entre as instruções. Além disso, um arco implementa, implicitamente, um *buffer* no qual operandos (*tokens*) gerados no nó de origem do arco podem ficar armazenados, aguardando ser consumidos pelo nó de destino do arco.

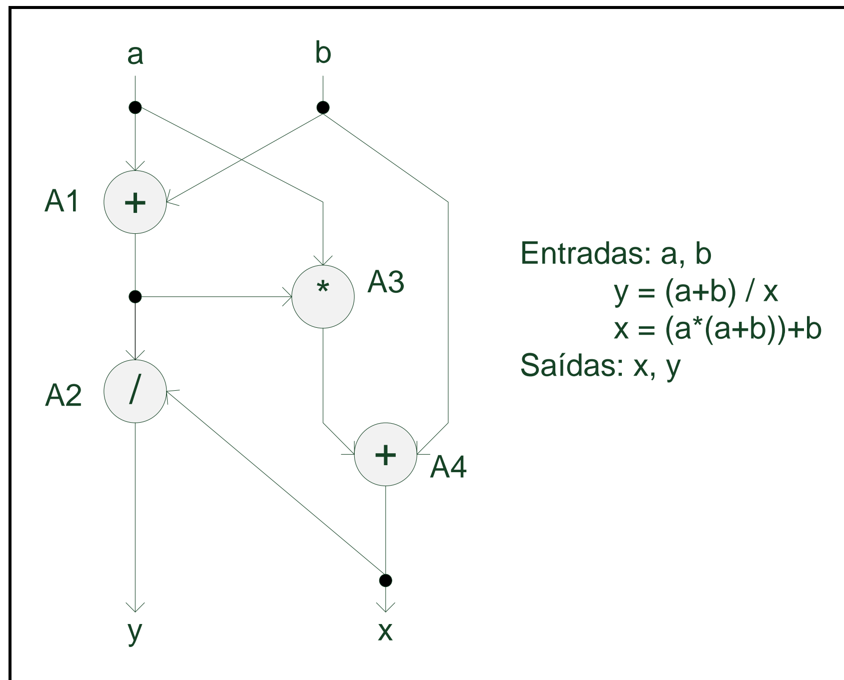


Figura 15 – Exemplo de função implementada segundo o modelo de fluxo de dados (Fonte: adaptado de Dennis; Misunas, 1974)

Na função mostrada na Figura 15, percebe-se que, embora *x* e *y* sejam valores de saída, existe uma dependência do cálculo de *y* em relação ao valor de *x*. Esta dependência está explícita na construção da operação A2, a qual possui como arcos de entrada os valores de saída de L3 (correspondendo ao resultado de *a+b*) e o valor de saída de A4 (correspondendo ao cálculo de *x*). Pode-se afirmar ainda que, uma vez que *a* e *b* estejam disponíveis nos respectivos arcos de entrada, a computação de *a+b* é executada e o resultado fica disponível imediatamente para A2 (que corresponde à equação de *y* na formulação da função apresentada no topo da figura) e para A3 (que corresponde a parte da equação de *x* na formulação da função apresentada no topo da figura). Ou seja, se colabora simultaneamente para a ativação de A2 e A3 sem uma definição programática de sequência de execução entre estas duas operações, conforme ocorreria em um programa segundo o modelo de von Neumann.

Como potencialmente várias instruções podem ser habilitadas simultaneamente, em função justamente do fluxo dos dados dos quais dependem, pode-se dizer que o modelo de fluxo de dados implementa uma forma de ILP (FERLIN, 2008). Este paralelismo é, porém, de

alto nível e perceptível pelo programador quando da concepção do *software* no sentido de que, por definição, as operações que compõem um programa de fluxo de dados, embora codificadas sequencialmente, podem ser executadas de forma simultânea. Além disso, o modelo de fluxo de dados se enquadra na categoria MIMD da taxonomia de Flynn, dado que cada instrução individual potencialmente depende de um conjunto de dados diferente para habilitar a sua execução.

As arquiteturas que implementam fluxo de dados podem ser classificadas em dois grandes grupos (HURSON; KAVI, 2008):

- Fluxo de dados estático: qualquer nó (expressão) de fluxo de dados pode ser executado somente se todos os *tokens* dos quais depende estiverem disponíveis nos seus arcos de entrada e não houver *tokens* em qualquer arco de saída aguardando para ser consumidos. Ou seja, somente uma instância deste nó pode estar habilitada em um determinado instante de tempo.
- Fluxo de dados dinâmico: podem haver múltiplos *tokens* nos arcos de saída, correspondendo a diferentes instâncias de execução do nó de fluxo de dados. As diferentes instâncias de execução, geralmente, são distinguidas umas das outras por meio de *tags*, que assinalam o contexto no qual aquele *token* foi gerado; assim, um nó fica habilitado para execução quando todos os arcos de entrada possuem *tokens* com a mesma *tag*.

Ainda, do ponto de vista de tecnologia de implementação, uma aplicação de fluxo de dados pode ser construída puramente em *software,* sobre plataformas usuais como von Neumann, puramente em *hardware* dedicado a uma aplicação de fluxo de dados específica ou adaptado ao modelo de execução de fluxo de dados, ou mesmo uma combinação desses. Particularmente, quando implementado em *software*, utiliza-se de mecanismos de inferência monolíticos principalmente quando se trata de cálculo lógico-causal. Por outro lado, quando implementado em *hardware*, pode-se fazer uso de inferência realizada pela própria arquitetura computacional. Em todo caso, segundo os esforços de revisão de literatura constantes deste trabalho, considera-se o mecanismo de inferência como monolítico.

Com base nos conceitos do modelo de fluxo de dados e na sua evolução teórica e técnica, diferentes implementações de arquitetura foram posteriormente elaboradas e eventualmente construídas e avaliadas. Estes aspectos são considerados na seção a seguir.

2.3.3.2 Modelos e implementações de fluxo de dados

Esta seção apresentada diferentes implementações do modelo de fluxo de dados. A lista de implementações não tem a intenção de ser completa, mas sim de apresentar um panorama de diferentes técnicas e conceitos de concepção relacionados a fluxo de dados. Estudos mais detalhados e completos sobre o histórico das arquiteturas de fluxo de dados podem ser encontrados em Ungerer, Silc e Robic (1998) e, mais recentemente, em Hurson e Kavi (2008). Além disso, a apresentação é resumida, porém destacando, quando conveniente, elementos arquiteturais ou de concepção que sejam relevantes para a proposição da ARQPON.

1) **Implementação elementar**

A implementação elementar foi proposta por Dennis e Misunas (1974) e segue, arquiteturalmente, a organização mostrada na Figura 16. Esta é uma implementação que se enquadra no grupo de fluxo de dados estático, na qual cada célula (unidade de execução) de instrução é capaz de armazenar o código de uma instrução e os seus operandos (máximo 2). O código de uma instrução inclui o seu opcode e também os endereços onde o resultado da operação deverá ser escrito (correspondendo aos arcos de saída).

Uma vez que uma célula de instrução seja habilitada (ou seja, todos os seus operandos estão disponíveis), esta notifica a rede de arbitragem, a qual decidirá em que ordem as diferentes unidades de processamento executarão as diferentes células de instrução habilitadas. A rede de arbitragem, portanto, implementa uma forma de resolução de conflitos.

As unidades de processamento recebem pacotes de operação da rede de arbitragem e executam estes pacotes, gerando pacotes de dados que correspondem aos resultados das operações (correspondendo aos *tokens* de saída) e que são encaminhados para os endereços correspondentes pela rede de distribuição. Esta, por sua vez, atualiza as células de instrução que são destino dos pacotes de dados gerados pelas instruções anteriores, reiniciando o ciclo.

Dennis e Misunas (1974) ainda apresentam uma variação da implementação elementar na qual uma rede de controle (não mostrada na Figura 16) é adicionada. Esta rede de controle atua sobre as células de instrução de forma a permitir a definição de operadores de controle em adição aos operadores de dados inicialmente definidos.

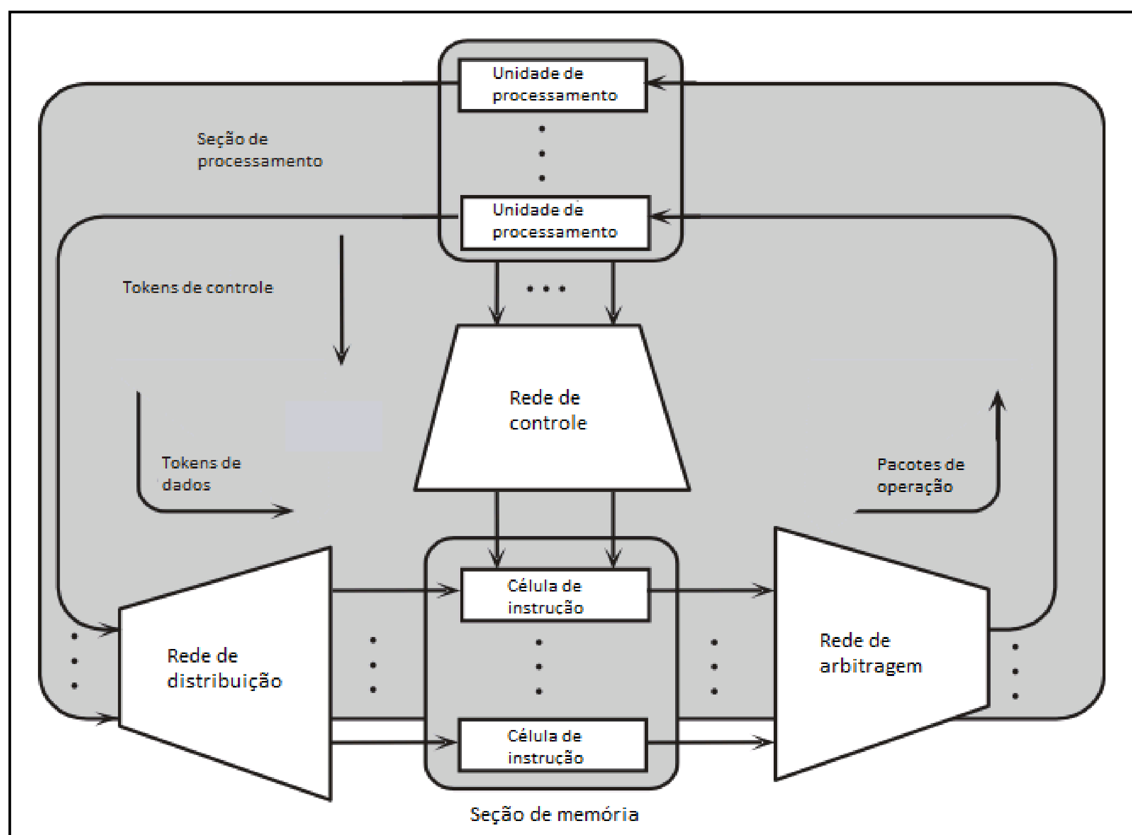


Figura 16 – Arquitetura elementar de fluxo de dados
(Fonte: adaptado de HURSON; KAVI, 2008, p. 4)

2) **MIT *Tagged Token Dataflow Architecture***

Arvind e Nikhil (1990) descrevem a implementação da *Tagged Token Dataflow Architecture* (TTDA), desenvolvida por seu grupo de pesquisa no Massachussets Institute of Technology (MIT). O TTDA é uma arquitetura de fluxo de dados dinâmica, ou seja, permite que múltiplos resultados (*tokens*) estejam presentes em um arco de saída. Esta característica permite, sob certas circunstâncias, a paralelização de múltiplas iterações de laços, uma vez que cada iteração é executada pelo mesmo subgrafo de operações e o resultado de cada iteração corresponde a um novo *token* de saída. Para tanto, cada *token* recebe uma *tag,* indicando a qual contexto (iteração) aquele *token* pertence.

Uma vez que *tokens* de diferentes *tags* devem ser distinguíveis uns dos outros, para que seja possível determinar se todos os *tokens* com a mesma *tag* estão disponíveis para ativação de uma operação, é necessário que os *tokens* tenham a sua disposição um espaço de memória adequado para o seu armazenamento. A identificação de *tokens* com a mesma *tag* é efetuada, no TTDA, por meio de busca e *matching* em uma memória com semântica de memória associativa (ou seja, na qual cada *token* pode ocupar qualquer posição da memória e procura-se determinar se um *token* está efetivamente na memória). Esta característica pode

influenciar negativamente o desempenho e motivou a proposição de diferentes estratégias de *matching*, conforme será abordado a seguir (Seção 2.3.3.3).

O modelo da memória utilizada pelo TTDA é baseado no conceito de *I-Structure* (NIKHIL; PINGALI; ARVIND, 1989). Este conceito permite a implementação de acessos à memória na forma de transações divididas (*split-phase*) e será abordado na Seção 2.3.3.3.

3) **Manchester *Dataflow Computer***

O computador de fluxo de dados desenvolvido na Universidade de Manchester (*Manchester Dataflow Computer*) (GURD; KIRKHAM; WATSON, 1985) é uma implementação dinâmica (baseada em *tags*) que consiste de uma fila de *tokens* repassados a uma unidade de *matching*, a qual agrupa os *tokens* em pares (para instruções diádicas, ou de dois operandos) ou unidades (para instruções monádicas, ou de um operando).

Em seguida, os *tokens* agrupados são enviados para um repositório de instruções, onde é selecionada, por meio de busca, a instrução correspondente que deve consumir os *tokens* agrupados, e o pacote contendo os *tokens* em conjunto com as *tags* e com a instrução é repassado para uma unidade de processamento, que processa a instrução e gera novos *tokens.* Estes são reencaminhados para a fila de *tokens* ou, então, repassados como saída por meio de um módulo de E/S.

4) **Monsoon**

A arquitetura Monsoon (PAPADOPOULOS, 1988) (PAPADOPOULOS; CULLER, 1990) foi construída em conjunto pelo MIT e pela Motorola. Seu principal diferencial em relação a modelos de fluxo de dados anteriores (principalmente o TTDA) é a implementação do modelo de armazenamento explícito de *tokens* (*explicit token store,* ou ETS), o qual melhora o desempenho de execução da aplicação quando comparado com arquiteturas tais como a TTDA, particularmente na fase de *matching* de operandos. Detalhes conceituais sobre o ETS são apresentados na Seção 2.3.3.3.

Do ponto de vista arquitetural, Monsoon é uma máquina composta de múltiplos elementos de processamento conectados entre si por uma rede de chaveamento e também a um conjunto de módulos de memória organizados na forma de *I-Structures* (ver Seção 2.3.3.3). Cada elemento de processamento possui um *pipeline* de 8 estágios, sendo que cada estágio necessariamente é ocupado por uma *thread* diferente, o que corresponde, aproximadamente, ao conceito de *multithreading* de granularidade fina (KLAUER et al., 2002) conforme apresentado na Seção 2.3.2.1.

5) **XPP**

O processador XPP-III (XPP, 2006) foi proposto pela empresa XPP Technologies como uma arquitetura de fluxo de dados adaptativa. Esta arquitetura é composta por um conjunto reconfigurável de elementos de processamento de *array* (PAEs) e por uma rede de comunicação entre estes elementos.

A arquitetura do XPP é dita *fluxo de dados de granularidade grossa (coarse-grained dataflow*) pelo fato que os PAEs podem executar, além de operações primitivas típicas de fluxo de dados e movimentação de memória de forma assíncrona (executadas pelos ALU-PAEs e RAM/IO-PAEs, respectivamente), também sequências de operações VLIW no estilo do modelo de von Neumann (executadas pelos FNC-PAEs).

6) **DF-KPI**

O computador DF-KPI foi desenvolvido na Universidade Técnica de Kosice na Eslováquia (ÁDÁM, 2010) e constitui-se em um sistema de fluxo de dados dinâmico (baseado em tags) que mescla localmente um modelo de fluxo de controle (seguindo os princípios de von Neumann) e globalmente um modelo de fluxo de dados. A concepção deste computador é baseada no conceito de computação em blocos (tile computing) (MADOŠ; BALÁŽ, 2011), o qual pode ser caracterizado por um grande número de elementos de processamento interconectados bidirecionalmente em rede, geralmente, em uma organização na forma de grade (grid ou mesh) na qual existe interconexão direta entre vizinhos.

A Figura 17 mostra o diagrama de blocos do DF-KPI. O modelo de fluxo de controle é uma implementação baseada em *pipeline* para cada um dos CPs (*coordinating processors*) definidos na arquitetura, composto por estágios de *load, fetch, operate, matching* e *copy.* Estes fazem uso do Repositório de Quadros e da Fila de Dados para efetuar a carga de instruções e operandos, respectivamente.

O sistema DF-KPI propõe uma estratégia de *direct matching* para os operandos, segundo a qual cada operando é referenciado em tempo de compilação e associado em tempo de execução à operação que o utiliza por meio de um endereço, permitindo otimização na operação de busca e *matching* de operandos disponíveis para ativação de determinada operação. Esta estratégia é correlata à estratégia de ETS implementada pela arquitetura Monsoon e faz uso da unidade de Repositório de Quadros para armazenamento dos quadros de memória definidos para o *matching* de operandos.

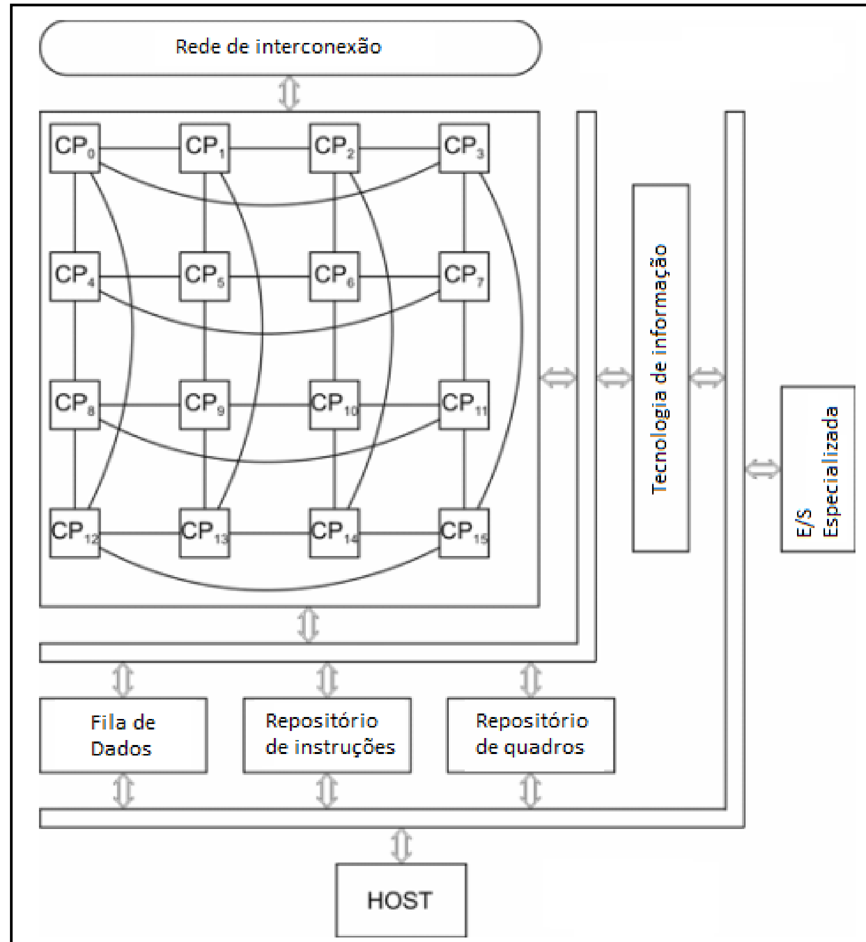


Figura 17 – Diagrama em blocos do sistema DF-KPI
(Fonte: adaptado de ÁDÁM, 2010, p. 78)

A Figura 17 ainda mostra alguns blocos acessórios (Tecnologia de Informação e E/S Especalizada), de tal forma a inserir a implementação do DF-KPI em um sistema computacional completo (ÁDÁM, 2010), bem como o *Host* utilizado para configuração e controle.

Madoš e Baláž (2011) descrevem uma implementação deste computador, consistindo de uma grade de 8 x 8 elementos com uma rede de interconexão com banda de 31 Tbps. Cada elemento de processamento implementa VLIW em 3 vias.

7) **EM-4**

A máquina EM-4 foi desenvolvida no Laboratório de Eletrotécnica em Tsukuba, Japão (SAKAI et al., 1989). Esta máquina é um exemplo de *threaded dataflow,* conforme classificado por Ungerer, Silc e Robic (1998), no sentido de permitir que sequências de instruções dependentes segundo o modelo de fluxo de dados sejam executadas conforme a

disponibilidade dos dados, aproveitando melhor os estágios de *pipeline* disponíveis para o núcleo de processamento selecionado.

A máquina EM-4 apresenta mais de 1000 elementos de processamento (PE) construídos em um único chip, denominado EMC-R. Cada um dos PEs é composto por um *pipeline* RISC simples, conectado a um conjunto de registradores, sendo que os estágios deste *pipeline* são melhor aproveitados por meio de atribuição de pesos diferenciados no grafo de fluxo de dados, criando o conceito de blocos fortemente conectados que são executados em sequência pelo mesmo PE.

O esquema de *matching* é direto, ou seja, cada função é associada a um *segmento de operandos* em memória convencional, dentro do qual os operandos são procurados para a verificação da habilitação ou não da função para execução (de forma também análoga à ETS).

8) **SDF (*Scheduled Dataflow*)**

Kavi, Giorgi e Arul (2001) propõem uma arquitetura denominada fluxo de dados escalonado (*Scheduled Dataflow,* ou SDF). Esta arquitetura pode ser considerada uma abordagem baseada em *multithreading,* no sentido de que prevê recursos de *hardware* para execução de múltiplas *threads* potencialmente em paralelo. No entanto, diferentemente do que acontece em um modelo *multithreading* von Neumann tradicional, as *threads* do SDF são necessariamente pequenas e simples (granularidade fina), não-bloqueantes e têm seu início sincronizado pela disponibilidade dos operandos que utilizam; ou seja, a sincronização e escalonamento das diferentes tarefas executadas pela aplicação é orientado a fluxo de dados (de forma análoga à do computador EM-4).

Segundo os autores, esta abordagem é uma alternativa a implementações superescalares e VLIW que dependem de *hardware* complexo para o controle de conflitos em *pipeline* e reordenação e emissão de múltiplas instruções. Isto ocorre porque as aplicações são compostas por *threads* von Neumann porém baseiam a sincronização em fluxo de dados, o que simplifica o *hardware* e permite desacoplar a execução no *pipeline* dos acessos à memória. Estes acessos são executados por processadores de sincronização (SPs) e ocorrem no início e no fim das *threads* para obtenção dos dados de entrada e geração dos resultados de saída, respectivamente.

9) **EDGE (*Explicit Data Graph Execution*)**

O EDGE (BURGER et al., 2004) é um modelo de arquitetura que permite o mapeamento, no próprio conjunto de instruções, de dependências de dados entre instruções a

serem executadas consecutivamente. Ou seja, isto elimina a necessidade de escalonamento dinâmico de instruções e construção dinâmica de tais grafos em *hardware*, o que geralmente torna mais complexas as implementações de *pipelines* de processadores modernos.

Neste caso, a característica de fluxo de dados reside no fato de que existe uma comunicação direta de dados gerados por uma instrução produtora para uma instrução consumidora (portanto, uma habilitação para execução), efetuada por meio de uma rede de roteamento de operandos, sem armazenamentos intermediários em memória ou registradores. O modelo de interconexão entre os elementos de processamento torna o EDGE mais um exemplo de arquitetura em blocos (*tiled architecture*).

O modelo EDGE é aplicado na arquitetura TRIPS, a qual serviu como base para um protótipo implementado na University Of Texas at Austin (BURGER et al., 2004). Este protótipo implementa 16 elementos de processamento, cada um incluindo uma unidade de inteiros, uma unidade de ponto flutuante, um roteador de operandos e um buffer de 128 bytes para manter múltiplas instruções e operandos pendentes. A ocupação do buffer de 128 bytes é determinada estaticamente, em tempo de compilação, porém a ordem de execução relativa é determinada dinamicamente em função da disponibilidade dos operandos (característica do modelo de fluxo de dados) (HURSON; KAVI, 2008).

10) **D2NOW (*Data-Driven Network Of Workstations*)**

O D2NOW (KYRIACOU; EVRIPIDOU; TRANCOSO, 2006) é uma implementação de uma arquitetura DDM (*Data Driven Multithreading).* O fundamento deste tipo de arquitetura é escalonar *threads* com base na disponibilidade de dados, por exemplo, ativar uma *thread* para execução somente se os dados de que ela depende já estiverem armazenados na *cache.* Esta política, denominada de *CacheFlow,* permite aproveitar melhor o tempo correspondente às latências geradas por acessos a memória, de forma similar a abordagens tais como a SDF.

O D2NOW é baseado em uma rede de processadores Pentium convencionais (HURSON; KAVI, 2008) interconectados por meio de comunicação de granularidade fina, ou seja, transferências de *tokens* de dados para efetuar a sincronização (aproveitando portanto, neste aspecto, o modelo de fluxo de dados). Este mecanismo de sincronização é implementado por uma unidade de sincronização de *threads* (TSU, ou *Thread Synchronization Unit*), que consiste em um módulo externo de *hardware* interligando diretamente ao barramento de interconexão dos processadores.

11) **WaveScalar**

A arquitetura WaveScalar (SWANSON et al., 2003) é similar ao EDGE no sentido de disponibilizar múltiplas unidades de execução e uma rede de roteamento de fluxo de dados entre elas. No entanto, WaveScalar difere da EDGE por ser projetada para alocar dinamicamente as instruções para os elementos de processamento disponíveis.

Esta arquitetura provê uma semântica de acesso à memória semelhante à de máquinas von Neumann (baseada em *loads/stores* executados ordenadamente), porém sem depender de contadores de programa para execução sequencial. Isto é obtido por meio do mapeamento de um programa von Neumann convencional em conjuntos denominados de *waves,* os quais são alocados posteriormente para os elementos de execução da arquitetura.

WaveScalar possui uma arquitetura de interconexão construída hierarquicamente, a qual agrupa unidades de execução para compartilhar recursos (por exemplo, ULAs e barramentos de comunicação) e interliga estas unidades segundo diferentes estratégias de conexão, dependendo do nível na hierarquia. Para tanto, define-se o conceito de *WaveCache*, que é uma grade de nós que agregam ULAs e blocos de memória *cache* (Figura 18), de forma semelhante a um computador de blocos (*tile computing*). O objetivo é manter o conjunto de instruções atual e os seus respectivos dados próximos da unidade de execução (esquerda da Figura 18) e executá-los *in place*, ou seja, sem movimentações entre níveis superiores de memória e registradores.

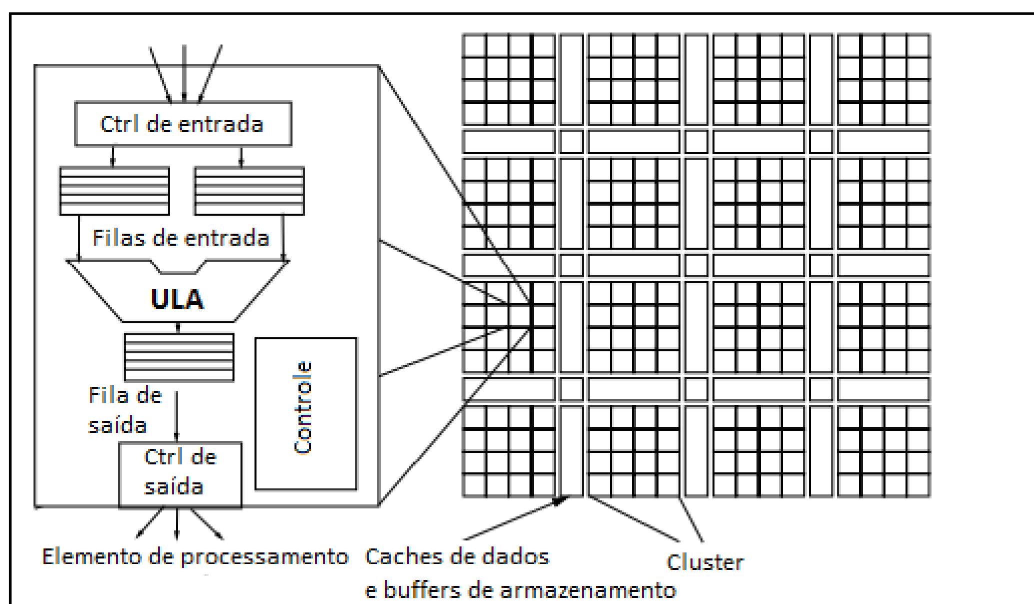


Figura 18 – Organização da WaveCache

(Fonte: adaptado de SWANSON et al., 2003, p. 5)

Esta é uma forma de explorar localidade em fluxo de dados, no sentido de utilizar a possibilidade de predição estática do fluxo de dados (levantamento prévio de produtores e consumidores) para tentar manter os dados de entrada de um determinado processador perto dele fisicamente. Os autores citam que esta abordagem de aumento da localidade espacial tornou-se viável ao longo dos anos em função do avanço tecnológico na construção de dispositivos de memória (principalmente aumento da densidade).

12) **Híbrido von Neumann – Dataflow**

Iannucci (1988) argumenta a respeito das vantagens e deficiências de arquiteturas que seguem puramente os modelos de von Neumann ou de fluxo de dados. Em particular, faz uma crítica ao *overhead* desnecessário de comunicação de *tokens* entre cada instrução presente no modelo de fluxo de dados, que pode desprivilegiar a baixa latência em algumas situações, e ao modelo de sincronização de tarefas paralelas relativamente ineficiente das máquinas de von Neumann.

Em função disso, Iannucci propõe uma implementação híbrida, que aproveita características de ambos os modelos com vistas a privilegiar baixa latência e facilitar sincronização. Esta implementação apresenta o conceito de *continuação,* que é uma combinação de um contador de programa e um registrador de endereço base e que representa um ponto de onde instruções podem ser obtidas para execução por um dos *pipelines* disponíveis. Cada continuação possui um estado (pronto, executando, etc.) e pode ter sua execução suspensa caso faça referência a um endereço de memória marcado como "vazio". A memória é organizada na forma de *I-Structure*, portanto oferece a semântica necessária para se verificar se um determinado endereço contém um dado válido ou não.

Resumidamente, a proposta de Iannucci consiste de uma implementação *multithreading* na qual o chaveamento é determinado pela ocorrência de um bloqueio, gerador de alta latência no acesso a memória. Isto permite destacar a sinergia que existe entre o modelo de fluxo de dados e o modelo de von Neumann, no que diz respeito a definir políticas de sincronização e chaveamento de contexto que sejam dependentes do fluxo de dados.

13) **RISC *dataflow* (P-RISC)**

A arquitetura denominada P-RISC (*Parallel RISC*), proposta por Nikhil (1989), consiste de uma implementação de fluxo de dados RISC no sentido de que procura manter a interface de programação no formato RISC (ou seja, capaz de executar *software* von Neumann convencional), porém agregando características de modelos de fluxo de dados.

Estas características são: suporte a mútiplas *threads;* sincronização explícita entre as múltiplas *threads* por meio de instruções específicas da arquitetura (*fork* e *join*); e alterações no modelo de memória para implementar *I-Structures* e transações de memória no estilo *split-phase* (UNGERER; SILC; ROBIC, 1998).

A arquitetura P-RISC é similar ao modelo híbrido proposto por Iannucci (1988), com a principal diferença de que a sincronização em P-RISC não ocorre potencialmente a cada instrução mas sim por meio das instruções específicas de sincronização (NIKHIL, 1989) .

14) **TAM (*Threaded Abstract Machine*)**

TAM (UNGERER; SILC; ROBIC, 1998) é um modelo de execução de programas, o qual é abstrato e pode ser implementado em uma arquitetura baseada ou não em modelo de fluxo de dados. No entanto, vale a pena ser citado nesta seção por ser fundamentado em programação para fluxo de dados, ou seja, pode ser utilizado como um passo intermediário para execução de programas de fluxo de dados em arquiteturas convencionais (CULLER et al, 1993).

Compilar para TAM envolve traduzir um programa expresso na forma de grafo de fluxo de dados em um conjunto de *threads.* O escalonamento destas *threads* é efetuado parte durante a compilação e parte durante a execução em um modelo *self-associative* (ou seja, as próprias *threads* tomam decisões a respeito da ordem de escalonamento), levando em consideração o compartilhamento de recursos no nível de hierarquia de armazenamento mais próximo do processador e utilizando instruções, no nível de aplicação, que permitem escalonamento e sincronização explícitos. TAM propõe que as características das *threads* não sejam dependentes de características do *hardware* paralelo (p. ex. número de processadores disponíveis) mas que possuam um nível de abstração próprio, com maior flexibilidade (CULLER et al., 1993).

Para tanto, TAM faz uso de estruturas de dados denominadas quadros de ativação, as quais armazenam informações relativas a um conjunto de *threads* habilitadas para execução dos seus respectivos blocos de código. Isto permite o escalonamento dinâmico e possibilita o armazenamento de dados locais ao conjunto de *threads* que pertence àquele quadro, explorando o princípio da localidade de acesso. A última *thread* escalonada de um determinado quadro é responsável por explicitamente iniciar o escalonamento do próximo quadro disponível para execução.

A Figura 19 apresenta esquematicamente as estruturas de dados utilizadas no modelo TAM. A fila de quadros prontos, que permeia a árvore de ativação, é percorrida para que o

próximo quadro de ativação seja selecionado. Este contém o vetor de continuação, que indica a localização das *threads* que estão ativadas e que podem ser escalonadas na sequência. Os segmentos de código ainda contém *inlets,* que são manipuladores de mensagens gerados durante a compilação e utilizados para a passagem de argumentos para as *threads* e atualizações do vetor de continuação.

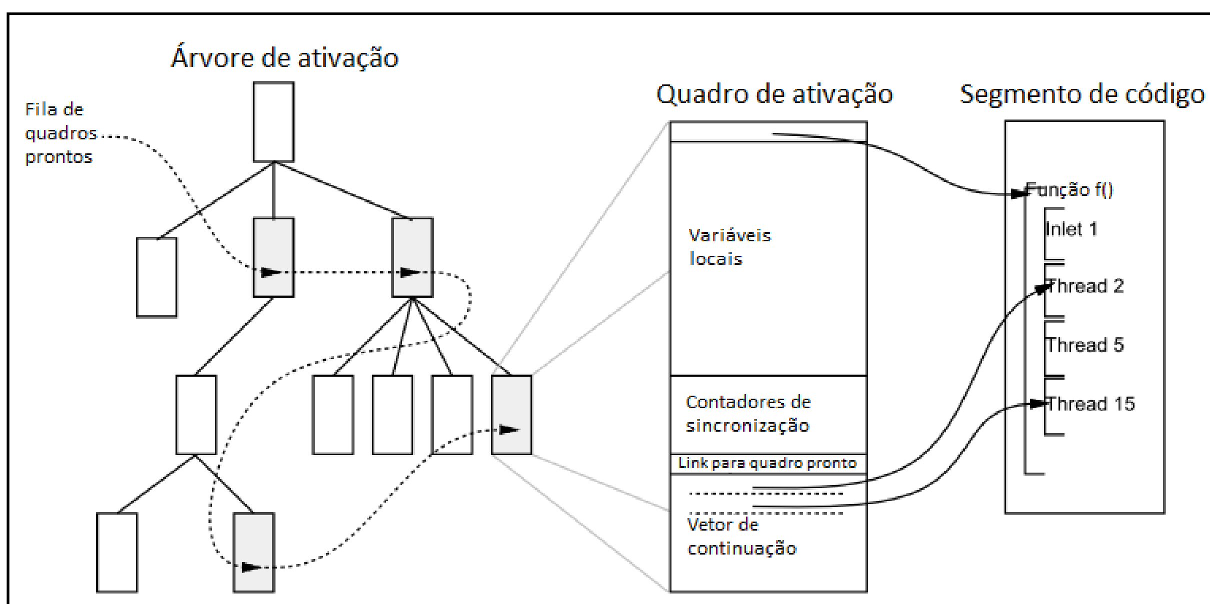


Figura 19 – Estruturas de dados do modelo TAM
(Fonte: adaptado de CULLER et al., 1993, p. 5)

Uma implementação da arquitetura TAM é o multiprocessador CM-5 (CULLER et al., 1993). Este multiprocessador é composto por núcleos baseados no processador Sparc RISC, interconectados por duas redes disjuntas que permitem fluxo de mensagens bidirecional.

15) **PRADA**

PRADA (FERLIN et al., 2011) é uma arquitetura paralela reconfigurável baseada em fluxo de dados. Propõe que um bloco de controle centralizado gerencie o envio de *templates* (instruções compostas por operações lógicas / aritméticas) para um conjunto de elementos de processamento (PEs) em paralelo, à medida que os operandos das operações descritas nos *templates* estejam disponíveis (portanto, segundo o modelo de fluxo de dados).

Embora materializado em lógica reconfigurável, um dispositivo PRADA apresenta PEs idênticos e genéricos, todos capazes de executar as mesmas operações lógicas ou aritméticas definidas pelo conjunto de instruções da arquitetura. Isto é apresentado pelos

autores como uma vantagem sobre outras arquiteturas (WaveScalar, etc.) nas quais existe um pré-mapeamento efetuado estaticamente que limita a distribuição de operações entre os PEs disponíveis. Os *templates* a serem processados pelos PEs são gerados manualmente pelo usuário e enviados ao PRADA por um computador externo (*host*) ao qual ele está conectado.

Experimentos comparativos descritos em Ferlin et al. (2011) demonstram que PRADA pode apresentar desempenho superior a arquiteturas capazes de um número maior de MIPS, notadamente nas aplicações comparadas que necessitam de uma grande quantidade de processamento sobre uma relativamente pequena quantidade de dados (criptografia e filtro digital, que são problemas de computação numérica). No entanto, em função da tecnologia utilizada para implementação (FPGA), existem limitações no que diz respeito à taxa de *clock* disponível, o que faz com que as comparações de desempenho sejam fundamentadas no número de ciclos de *clock* necessário para execução das aplicações e não no tempo absoluto.

16) **WASMII**

WASMII (LING; AMANO, 1993) é uma proposta de implementação de uma arquitetura de fluxo de dados utilizando como base uma FPGA a ser reconfigurada dinamicamente por meio de módulos (páginas) de RAM disponibilizados dentro do *chip*. Este conjunto de RAMs define um "*hardware* virtual", a ser reconfigurado sobre o *hardware* real oferecido pela FPGA.

A arquitetura do WASMII foi direcionada para o modelo de fluxo de dados justamente pelo fato de que as páginas de RAM que definem a reconfiguração do circuito não tem a capacidade de armazenar dados, somente processá-los. Sendo assim, a sincronização da reconfiguração necessária depende somente de que todos os *tokens* tenham sido removidos da página de RAM a ser substituída. Além disso, a característica de fluxo de dados permite que *chips* WASMII sejam interconectados para formar um sistema altamente paralelizável.

Ling e Amano (1993) citam como possíveis aplicações do WASMII a execução de algoritmos para resolução de problemas científicos que possuam característica iterativa. O corpo de instruções que compõem uma iteração, neste caso, é convenientemente alocado em páginas de RAM específicas do WASMII, que podem ser intercambiadas dinamicamente permitindo a utilização de um *hardware* real mais simplificado do que seria necessário para implementação estática da lógica de resolução do problema.

17) **Outras implementações**

Liu e Furber (2005) apresentam uma implementação de coprocessador de fluxo de dados, acoplado a um processador convencional RISC von Neumann. A iniciativa tem como principal objetivo reduzir o consumo de energia, que segundo os autores é potencializado em arquiteturas puramente von Neumann pela necessidade de se efetuar busca e decodificação de cada instrução executada sequencialmente, o que aumenta o uso da banda de acesso à memória. Experimentos realizados para algumas aplicações de *benchmark* envolvendo processamento intensivo de laços (p. ex. criptografia) demonstram que esta abordagem pode diminuir o consumo de energia em magnitudes de até 20 vezes.

Farabet et al. (2011) apresentam uma aplicação de visão computacional, implementada em lógica reconfigurável e segundo os princípios do modelo de fluxo de dados. Consideravelmente relevante nesta implementação é a definição dos *processadores de operação*, que são módulos simples (operações aritméticas, FIFOs, etc.) responsáveis pelo processamento e interconectáveis a outros processadores de forma reconfigurável, por meio de multiplexadores de roteamento. Os processadores de operação são organizados em uma estrutura matricial (portanto, similar ao modelo de *tile computing*), simplificando assim a arquitetura de interconexão.

Teifel e Manohar (2004) apresentam duas diferentes implementações do modelo de fluxo de dados baseadas em computação assíncrona, na qual as computações lógicas não são sincronizadas por um sinal de *clock*. Para tanto, define-se um conjunto de *pipelines* assíncronos (elementos processadores) que se comunicam por meio de portas de comunicação, sobre as quais são enviados *tokens* que correspondem ao fluxo de dados. As implementações são efetuadas em FPGA, portanto garantindo que os canais de comunicação entre elementos processadores possam ser definidos em tempo de projeto e programados no dispositivo reconfigurável para a execução da aplicação. No entanto, segundo Ferlin (2008) isto se apresenta como uma desvantagem desta proposta, uma vez que não existe uma flexibilidade dos elementos processadores para executar operações de outras aplicações, a menos que a FPGA seja reprogramada para tanto.

MONARCH (VAHEY et al., 2006) é um exemplo de implementação de arquitetura dita polimórfica, a qual possui um dispositivo reconfigurável (FPCA, ou *Field Programmable Computer Array*) que pode ser reconfigurado estática ou dinamicamente para atender a demandas de aplicações. A FPCA foi otimizada para processamento de sinal, fornecendo estruturas que implementam fluxo de dados, porém pode ser reconfigurada para implementar 6 processadores convencionais SIMD. Além disso, a FPCA é interconectada a um *array* de 32

núcleos de processamento RISC organizados em blocos (*tile computing*), sendo que cada um dos núcleos RISC possui 2 MB de DRAM dedicada de alta velocidade.

Gupta e Sohi (2011) propõem um modelo de execução para permitir a paralelização dinâmica de *software* sequencial e execução em uma plataforma *multicore,* utilizando sincronização baseada no modelo de fluxo de dados. A idéia dos autores é permitir que os conceitos de ILP superescalar sejam aproveitados em uma escala maior, ou seja, disponibilizando múltiplos núcleos para execução concorrente ao invés de múltiplos estágios de *pipeline.* Para tanto, os autores definem o conceito de FLP (*Function Level Parallelism*), que consiste em paralelizar funções, distribuindo-as em múltiplos núcleos e sincronizando o início da sua execução com a disponibilidade dos operandos de entrada (que são encapsulados em objetos), seguindo o modelo de fluxo de dados.

Os objetos são relacionados a múltiplos *tokens*, tanto de leitura quanto de escrita, para poder atender à característica de dados (objetos) mutáveis imposta pelos programas sequenciais von Neumann. Os *tokens* devem ser adquiridos de forma exclusiva pelo sistema de execução para serem utilizados (lidos ou escritos) por uma função, após o que são liberados e podem ser direcionados para funções que aguardam por aquele operando ao qual o *token* está relacionado (sincronização de início de execução). Os autores citam que as dependências entre funções podem ser mapeadas tanto estática quanto dinamicamente, dada a implementação de modelo de fluxo de dados para sincronização.

Experimentos comparativos, realizados em uma plataforma que implementa o modelo puramente em *software*, demonstram que ganhos em desempenho podem ser obtidos em relação a implementações semelhantes de *software* efetuadas utilizando-se APIs de programação concorrente (PThreads), mesmo levando-se em consideração o *overhead* *i*mposto pela implementação puramente em *software.*

2.3.3.2.1 Resumo

A Tabela 1 sumariza os diferentes modelos e implementações de fluxo de dados apresentados na seção anterior, listando resumidamente as suas principais características.

Tabela 1 – Sumarização de modelos e implementações de fluxo de dados

Modelo / implementação	Características
Fluxo de dados elementar	
TTDA	Dinâmico, com *I-Structures*
Manchester	Dinâmico, seleção de instruções em um repositório
Monsoon	ETS com *I-Structures*
PRADA	Reconfigurável, com número fixo de PEs e *templates* alocados dinamicamente para eles.
Farabet	*Tile computing* com PEs simples e NoC
Teifel e Manohar	Computação assíncrona, reconfigurável em FPGA porém não genérico (necessita de reconfiguração para cada aplicação).
WASMII	Reconfigurável por meio de páginas de RAM intercambiáveis que implementam fluxo de dados – não implementam memória
Híbrido Von Neumann de granularidade grossa	
XPP	Instruções VLIW + fluxo de dados
DF-KPI	*Tile computing,* fluxo de controle em *threads,* fluxo de dados para sincronização entre *threads.*
EM-4	ETS, fluxo de dados para sincronização entre *threads*
D2NOW	Rede de computadores Pentium convencionais, fluxo de dados para sincronização entre *threads*
P-RISC	Baseado no híbrido de Iannucci porém chaveamento ocorre em função de instruções de sincronização
MONARCH	Rede de fluxo de dados interconectável dinamicamente a 32 núcleos RISC
Gupta e Sohi	Granularidade de funções, sincronizadas pela disponibilidade dos argumentos, mapeadas estática ou dinamicamente a partir de um grafo de fluxo de dados
Híbrido Von Neumann de granularidade fina	

SDF	*Threads* pequenas, não bloqueantes, sincronização pela disponibilidade de todos os operandos conforme fluxo de dados
Híbrido de Iannucci	*Multithreading* com chaveamento nos pontos de bloqueio (onde se aguarda operandos), usa *I-Structures.*
Execução de grafo de fluxo de dados	
EDGE	Instruções produtoras enviam operandos diretamente para instruções consumidoras por meio de rede de roteamento (*tiled computing*), alocação de instruções para unidades de execução é estática.
WaveScalar	Semelhante ao EDGE porém alocação das instruções é dinâmica. Permite agrupamento de ULAs e memória *cache* para privilegiar localidade espacial (WaveCache)
TAM	Grafo de fluxo de dados é compilado para um conjunto de *threads* auto-associativas (auto escalonáveis). Quadros de ativação armazenam conjunto de *threads* ativadas.

2.3.3.3 Gerenciamento de memória e sincronização

Nesta seção são apresentadas algumas soluções arquiteturais propostas e implementadas no contexto de gerenciamento de memória e sincronização em arquiteturas de fluxo de dados.

A arquitetura Monsoon (PAPADOPOULOS, 1988) implementa o modelo de armazenamento e sincronização para *tokens* denominado de *explicit token store* (ETS). Segundo este modelo, um quadro de memória (*memory frame)* é alocado estaticamente para cada iteração de laço, em um endereço indicado por um ponteiro que é associado à *tag* daquela iteração. Este quadro de memória contém locais onde os *tokens* de saída das instruções do bloco correspondente à *tag* serão armazenados e de onde os *tokens* de entrada das mesmas instruções serão lidos. Desta maneira, não é necessária a busca e *matching* em todo o espaço de uma memória associativa para determinação da presença ou ausência (e valor) de todos os *tokens* que ativam a operação de fluxo de dados, mas sim somente no espaço correspondente ao quadro de memória. Isto melhora o desempenho de execução da

aplicação quando comparado com arquiteturas tais como a TTDA, além do que a alocação estática e explícita de espaço de memória para *tokens* evita situações nas quais a geração de *tokens* pudesse exceder o espaço disponível para um determinado elemento de processamento, o que poderia causar um *deadlock* na execução do programa de fluxo de dados.

Papadopoulos e Culler (1990) citam que a estratégia de alocação de quadros de memória do ETS remete a uma máquina von Neumann primitiva, no sentido de que esta também depende de quadros de memória para empilhamento e desempilhamento de argumentos quando da execução de funções. No entanto, a execução de um programa no Monsoon gera uma árvore de quadros de memória ativados, ao invés de uma pilha, devido à característica intrínseca de ativação paralela de instruções apresentada pelo modelo de fluxo de dados.

Além disso, os quadros de memória Monsoon possuem bits de presença para determinar se o quadro está de fato ativado ou não. Estes quadros são (re)verificados pelo mecanismo de inferência quando da geração de um novo *token*, eventualmente causando o disparo das operações de cálculo que deles dependem de acordo com o modelo de fluxo de dados. Portanto, diferentemente do modelo de inferência do PON, o modelo baseado em ETS ainda depende de um mecanismo centralizado para coordenar e iniciar as ações de cálculo.

A Figura 20 apresenta um exemplo de alocação de instruções e de dados na memória de programa e no quadro de ativação, respectivamente. Quando o *token* com valor 3.57 é gerado na saída da instrução *s* (lado direito), no contexto de execução *c,* este *token* é automaticamente direcionado para a instrução *s+1*, a qual é buscada no respectivo endereço da memória de programa (segunda linha da figura). O campo *r* indica que um dos operandos desta instrução está no deslocamento *+1* a partir do início do quadro de ativação indicado pelo contexto *c;* este endereço corresponde à segunda linha do quadro de ativação da figura (*c+1*), na qual o valor *3.57* que chegou no *token* é depositado, tendo o valor do bit de presença *q* alterado para *P* (“presente”). Isto permite que, quando o segundo *token* de entrada da instrução *s+1* chegar, a posição *c+1* seja verificada e, uma vez constatado que já está ocupada, dispare a execução da instrução *s+1* (que é uma multiplicação entre o segundo *token* recém-chegado e o *token* já armazenado na posição *c+1*).

A execução da instrução *s+1*, por sua vez, gera *tokens* que serão encaminhados para as instruções armazenadas *+1* e *+n* posições de memória adiante (correspondentes ao campo *dests*), portanto para as instruções *s+2* e *s+n+1*. O *token* direcionado para a instrução *s+2* desencadearia prontamente a sua execução, visto que esta instrução já tem o *token* recebido de *s* (lado esquerdo) armazenado na posição correspondente a *c+0* (pois seu *r* é igual a 0).

O modelo de memória baseado em *I-Structure* (NIKHIL; PINGALI; ARVIND, 1989) foi bastante utilizado historicamente nas implementações de modelo de fluxo de dados. Segundo os autores, este modelo permite introduzir uma noção limitada de estado no grafo de fluxo de dados, em contraposição aos *tokens* simples trafegados conforme o modelo elementar.

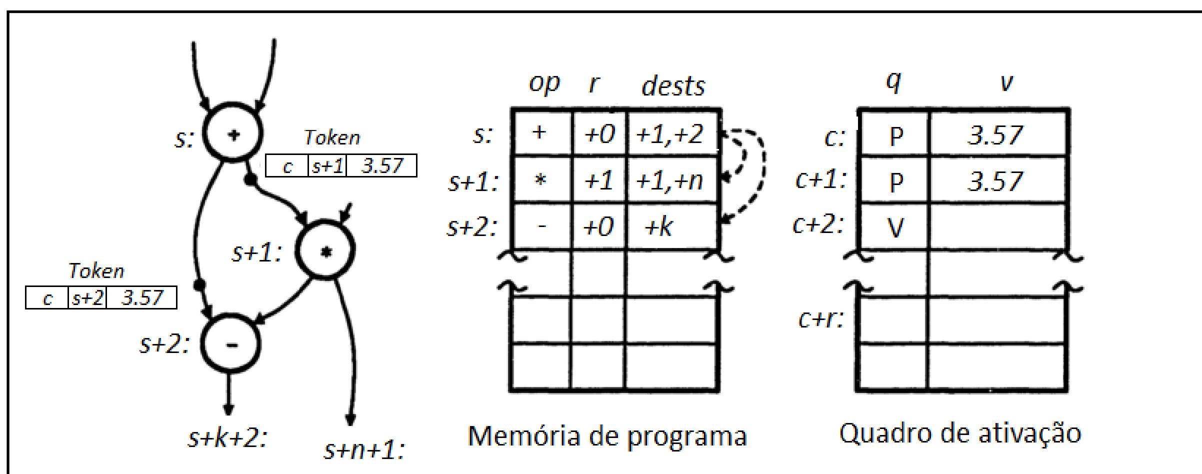


Figura 20 – Exemplo de alocação de recursos em ETS
(Fonte: adaptado de PAPADOPOULOS, 1988, p. 45)

Uma posição qualquer de memória em uma *I-Structure* contém bits extra de presença, que permitem implementar os estados "presente", "ausente" ou "aguardando" (o estado inicial é "ausente"). Uma posição é marcada como "presente" quando um *token* é escrito naquela posição, como resultado da execução de alguma operação.

Caso uma operação de leitura de um determinado *token* (efetuada pelo mecanismo de *matching*) tente ler uma posição marcada como "presente", o *token* é enviado como resultado daquela operação. Caso esteja marcada como "ausente", a operação de leitura é enfileirada e a posição é marcada como "aguardando".

Uma vez que uma posição esteja marcada como "aguardando", se houver uma operação de escrita nela, então o *token* é enviado como resultado para todas as operações de leitura aguardando na fila e o estado é alterado para "presente".

A

Figura 21 apresenta uma sequência de instruções que faz acesso a dados em uma *I-Structure* e o resultado da execução destes acessos, juntamente com o estado final da memória de dados e da fila de leituras postergadas. Arvind e Nikhil (1990) mencionam que as operações de leitura em *I-Structures* são divididas em fases (*split-phase*). Isso significa que o

pedido de leitura para a *I-Structure* é desvinculado do atendimento deste pedido, o que é mais evidente nas transações em que as operações de leitura são finalizadas após o estado mudar de "aguardando" para "presente". Isto permite que o processador execute quaisquer outras instruções habilitadas enquanto a leitura não puder ser finalizada, o que se constitui em mais uma técnica favorecedora do paralelismo.

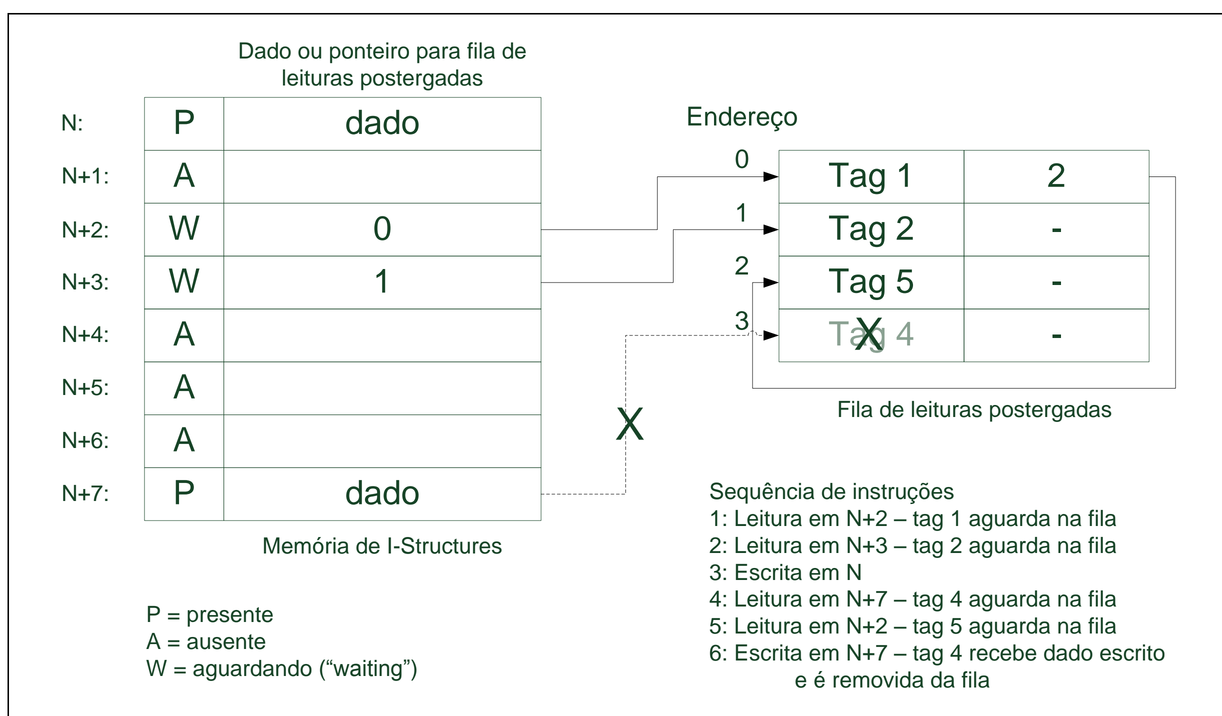


Figura 21 – Exemplo de atuação de sequência de instruções sobre *I-Structures*
(Fonte: adaptado de ARVIND; NIKHIL, 1990, p. 306)

Ainda em relação à questão da dinâmica de acesso às *I-Structures*, estas apresentam uma semântica *não estrita* no sentido de que uma determinada estrutura de dados (p. ex. um *array*) pode ter seu endereço acessado e, eventualmente, partes de seu conteúdo, mesmo antes de todo o conteúdo ter sido gerado (dado que cada elemento tem seu acesso regulado de acordo com o protocolo definido para *I-Structures*). Esta característica permite um paralelismo semelhante ao de um *pipeline,* na medida em que viabiliza que uma estrutura seja produzida e consumida parcialmente (em analogia aos estágios do *pipeline*) (NIKHIL; PINGALI; ARVIND, 1989).

Najjar, Böhm e Miller (1994) apresentam um estudo no qual ponderam sobre a ênfase que se dava, à época, à exploração de paralelismo ou de localidade de *thread* em arquiteturas de fluxo de dados, com baixa ou alta granularidade. Objetivou-se, com esta

ênfase, obter um balanço entre a complexidade da arquitetura e a tolerância à latência de acesso a memória, no entanto sem necessariamente levar em consideração questões de localidade no acesso a dados. Um exemplo são algumas arquiteturas de granularidade fina, que são fortemente dependentes de circulação de *tokens* e acesso a *I-Structures* em transações de fase dividida (ver Seção 2.3.5.1), muitas vezes gerando um tráfego de dados desnecessário.

Levando-se esta questão em consideração, os autores propõem uma variação do conceito de *I-Structure* denominado de *Célula de Vetor* (*Vector Cell,* ou *V-Cell).* A idéia de uma célula de vetor é agrupar um conjunto de dados (*chunk*) e fazer com que este conjunto possua semântica de acesso semelhante ao de uma *I-Structure,* ao invés de implementar esta semântica individualmente para cada dado conforme efetuado em outras implementações de *I-Structure.*

Esta organização apresenta como vantagens manter a tolerância a latências, ao mesmo tempo em que permite a implementação de *pipelines* vetorados (estilo superescalar) para o processamento dos dados visto que estes são acessados em conjunto. Ainda, o conceito de transação de fase dividida persiste nesta arquitetura porém as transações são efetuadas sobre células inteiras, e não somente sobre dados individuais. Para não sobrecarregar as etapas de sincronização de fluxo de dados, o *matching* é efetuado somente sobre manipuladores (*handles*) das células.

Os autores ponderam ainda que a arquitetura com células de dados é um híbrido entre von Neumann (no sentido de permitir a execução vetorada de dados dentro de uma célula) e fluxo de dados (no sentido de que a sincronização do acesso a células segue o modelo de transação dividida e de sincronização por presença dos dados).

2.3.3.4 Modelos e linguagens de programação para fluxo de dados

A seguir apresenta-se resumidamente a conceituação da programação no modelo de fluxo de dados, bem como um conjunto de modelos e linguagens de programação desenvolvidas especificamente para este tipo de programação.

Johnston et al. (2004) listaram recursos e características típicos de uma linguagem de programação voltada ao modelo de fluxo de dados. Estes recursos e características incluem:

- ausência de efeitos colaterais, tais como a atualização de variáveis globais (o que leva também a localidade de efeito);

- escalonamento efetuado em função de dependências de dados;
- atribuição única de variáveis (*tokens*), dado que estas serão consumidas posteriormente;
- ausência de histórico nos procedimentos, devido à atribuição única citada anteriormente.

De maneira geral, as linguagens de programação desenvolvidas para o modelo de fluxo de dados incluem as características listadas acima, o que as aproxima muito do paradigma de programação funcional, porém contendo elementos (principalmente sintáticos) dos programas imperativos, tais como a definição de laços. Exemplos incluem linguagens tais como TDFL, LAU, LUCID, Id, VAL, LAPSE, SISAL, entre outras (JOHNSTON et al., 2004).

VAL (*Value-Oriented Algorithmic Language)* é uma linguagem de programação desenvolvida no MIT (HURSON; KAVI, 2008), com o objetivo de representar grafos de fluxo de dados na forma de texto em uma linguagem de programação de alto nível. Apresenta construções (operações) comuns de fluxo de dados, porém suporte limitado para operações de entrada/saída e recursividade.

Id (*Irvine Dataflow Language*), desenvolvida para gerar código executável para a plataforma TTDA (ARVIND; NIKHIL, 1990), é uma linguagem de programação de alto nível interpretada e com tipagem implícita (i. e., tipos de dados são determinados à medida em que os dados são definidos e utilizados), permitindo tipos de dados escalares e estruturas. Variáveis são uma abstração utilizada somente por conveniência, dado que na prática cada expressão do fluxo de dados gera um novo dado como resultado a partir dos seus dados de entrada, ou seja, não existe atualização de dados já existentes (conceito essencial do modelo).

SISAL (*Streams and Iterations in a Single Assignment Language*) (SISAL, 2012) (HURSON; KAVI, 2008) é citada como a linguagem de programação de fluxo de dados com maior aceitação pelo fato de gerar código C otimizado como código intermediário e, portanto, permitir que os seus programas sejam compilados por um compilador C convencional e executados em plataformas convencionais. Além de tipos de dados semelhantes aos da linguagem Id, SISAL implementa o conceito de fluxo (*stream*), que consiste em uma sequência de valores gerados por uma expressão de fluxo de dados e consumidos na mesma ordem. Uma característica citada como importante é a possibilidade de interoperar com código desenvolvido em outras linguagens, por meio da definição de interfaces de funções e de módulos em fluxo de dados que implementam aquelas interfaces.

Chen et al. (2008) apresentam um modelo de programação inspirado em fluxo de dados (conceitos de componentes - unidades de computação - e canais de comunicação - fluxo de dados - entre eles) e sua implementação sobre um ambiente de tempo de execução Java. No entanto, este modelo depende de abstrações de *software* de alto nível e não é necessariamente executável em uma arquitetura própria para execução segundo modelo de fluxo de dados.

2.3.3.5 Considerações sobre arquiteturas baseadas em fluxo de dados

Segundo Sakai et al. (1989), o modelo de fluxo de dados seria teoricamente o mais adequado para viabilização de computação paralela pelos seguintes motivos:

- capacidade de extrair naturalmente concorrência na computação;
- ser adequado à integração em larga escala (VLSI) em *hardware* por ser constituído de um grande número de elementos de processamento (nós do grafo de fluxo de dados) idênticos;
- permitir que linguagens de fluxo de dados ofereçam abstrações mais fáceis de se utilizar para o desenvolvimento de programas concorrentes.

No entanto, Ling e Amano (1993) apontam que o funcionamento completo em fluxo de dados das implementações até então existentes era limitado justamente pela limitação dos recursos disponíveis para a representação do grafo de fluxo de dados. Esta limitação persiste, embora bem menos drástica considerando-se a evolução de escala de integração e, portanto, da possibilidade de implementação de máquinas de fluxo de dados com um número cada vez maior de elementos de processamento. Ainda, apesar desta limitação, Ling e Amano ponderavam que as máquinas de fluxo de dados tendem a ser relativamente menos complexas do que máquinas von Neumann de tecnologia similar.

Dada a similaridade que existe entre os conceitos de fluxo de dados de granularidade fina e *multithreading,* ambos sofrem de problemas semelhantes no que tange à execução de *software* predominantemente sequencial . De fato, conforme citado por Ungerer, Silc e Robic (1998), embora arquiteturas de fluxo de dados de granularidade fina acabem por não depender de lógica complexa para resolver conflitos de dados ou controle, pois não há paralelismo em nível de instrução dentro de uma unidade de execução, esta característica implica no envio de

uma instrução ao *pipeline* somente após a anterior ter sido executada, o que acaba impactando negativamente no desempenho.

Além disso, variáveis estáticas de escopo global, que são uma das características arquiteturais mais interessantes do modelo de von Neumann, não são implementadas no modelo de fluxo de dados justamente por este não definir o conceito de dados mutáveis e propagar os resultados do processamento no próprio fluxo de comunicação e execução. Isto aproxima a programação em fluxo de dados de um modelo de programação funcional, o qual segundo Gupta e Sohi (2011) pode ser difícil de utilizar e ineficiente para muitas aplicações.

Segundo Culler, Eicken e Schauser (1992), a argumentação persuasiva em favor de fluxo de dados apresentada originalmente por Arvind e Iannucci (1987) mostra problemas porque a abordagem é muito simplista em relação aos seguintes aspectos:

- A granularidade fina do fluxo de dados não leva corretamente em consideração a complexidade das hierarquias de memória. Este argumento está muito ligado ao problema da "*memory wall*" e, portanto, das latências envolvidas em acessos a dados com granularidade muito fina.
- A política de escalonamento de programas em fluxo de dados, que consiste basicamente em executar uma instrução quando possível (ou seja, quando os dados de entrada estiverem presentes) geralmente não é a melhor política na maioria das circunstâncias. Este argumento está muito ligado à necessidade de se levar em consideração o escalonamento de operações de acesso à memória, dado que estas podem consumir muito tempo para serem executadas.

Além destes argumentos, os autores ponderam que não se pode aumentar o grau de paralelismo (número de elementos de processamento) de uma arquitetura de fluxo de dados sem aumentar os custos de sincronização. Isto ocorre porque o grau de paralelismo apresenta um impacto na quantidade de registradores de acesso rápido que podem ser disponibilizados para cada elemento de processamento para armazenamento de *tokens*, implicando em acessos mais frequentes a níveis de memória mais lentos.

Johnston et al. (2004), por sua vez, apresentam argumentos de diferentes pesquisadores, elaborados na década de 1990, os quais acreditavam já à época que a granularidade fina de execução no modelo essencial de fluxo de dados levava a um consumo excessivo de recursos para emissão e execução de cada instrução, quando comparado ao modelo de von Neumann. Esta característica é particularmente prejudicial para programas com baixo grau de paralelismo ou para programas puramente sequenciais. Após esta

constatação, optou-se por redirecionar o estudo do modelo de fluxo de dados para a elaboração de arquiteturas híbridas, que pudessem fazer uso de conceitos do modelo de von Neumann e do modelo de fluxo de dados de forma combinada. Um exemplo de híbrido entre fluxo de dados e von Neumann é apresentado na Seção 2.3.3.2 (IANNUCCI, 1988).

Finalmente, no que diz respeito aos conceitos de arquitetura de fluxo de dados, embora estes conceitos tenham atingido relativamente baixo sucesso comercial e tenham ficado muito restritos ao campo de pesquisa acadêmica, muitas das técnicas e tecnologias desenvolvidas nesta área são utilizadas em arquiteturas de processadores modernas. Como exemplo pode-se citar técnicas complexas de detecção de conflitos de dados em *pipelines,* paralelismo dinâmico e execução fora de ordem. Contudo, algumas questões ainda devem ser resolvidas para que arquiteturas modernas possam utilizar do conceito de fluxo de dados de maneira mais efetiva:

- Manipulação de estruturas de dados de forma eficiente. Neste sentido, a necessidade do *matching* de operandos é um dos fatores que desfavorecem o modelo de fluxo de dados em relação ao desempenho. No entanto, técnicas tais como *explicit token store,* a utilização de quadros de ativação e o aumento da granularidade para tentar diminuir o número de pontos de sincronização (NAJJAR; BÖHM; MILLER, 1994) tendem a minimizar este fator.
- Alocação dinâmica de instruções de fluxo de dados para as unidades de execução disponíveis.
- Utilização eficiente de memória *cache* em arquiteturas de fluxo de dados.

Os problemas relacionados ao *matching* em arquiteturas de fluxo de dados, em particular, estão muito relacionados à forma como se realiza o controle da execução, tanto de operações aritméticas sobre os dados quanto de relações causais entre eles. Isto ocorre porque, embora a inferência seja disparada pela disponibilidade dos operandos (portanto, sob demanda), esta disponibilidade tem que ser verificada por meio de busca em um repositório de *tokens;* esta busca pode ser efetuada por meio de diversas técnicas, inclusive utilizando-se de máquinas de inferências típicas de SBR tais como as baseadas no algoritmo RETE (SIMÃO et al, 2012a).

Neste aspecto, uma arquitetura que faça uso do mecanismo de notificações do PON, conforme anteriormente explanado, pode apresentar vantagens, em relação às abordagens de fluxo de dados, justamente por não depender conceitualmente de repositórios de *tokens*, mas sim de notificações enviadas pontualmente e endereçadas diretamente aos seus consumidores.

Isto ocorre porque o processo de inferência no PON é fundamentalmente distribuído e colaborativo entre as entidades ativas do metamodelo.

Por outro lado, alguns avanços tecnológicos têm viabilizado a retomada de novos projetos e pesquisas em arquiteturas de fluxo de dados, tais como o aumento na densidade dos *chips*, a disponibilidade cada vez maior de espaços de memória *cache* e, também, o aumento na sofisticação de compiladores capazes de extrair o máximo de paralelismo por meio de análises de interdependências (HURSON; KAVI, 2008).

Em relação à programação voltada ao modelo de fluxo de dados, diversos autores (HAAVIND, 2008) (SWANSON et al., 2003) citam a dificuldade desta prática quando comparada à programação para o modelo de von Neumann. Em particular, uma das maiores dificuldades é a ausência de algumas propriedades e construções que são extremamente úteis em linguagens de programação como C, C++ e Java, tais como estruturas de dados mutáveis e variáveis globais.

2.3.4 Outros modelos de arquitetura paralela

Nesta seção são apresentadas, complementarmente, outras concepções de arquitetura paralela além das já apresentadas nas seções anteriores, as quais foram particularmente enfatizadas por serem mais diretamente aplicáveis à premissa de concepção da ARQPON como uma arquitetura baseada em multiprocessadores.

2.3.4.1 Multicomputadores

De forma correlata aos multiprocessadores, existem as arquiteturas paralelas baseadas fundamentalmente em multicomputadores, que também se encaixam na categoria MIMD conforme proposto por Flynn (1966). Conforme já citado (ver Seção 2.3.1), o principal diferencial entre multicomputadores e multiprocessadores reside no fato de que os primeiros possuem um espaço independente de memória alocado para cada nó de processamento. Cada nó, por sua vez, pode possuir múltiplos núcleos que compartilham a sua memória à maneira de um multiprocessador, porém este nó se comunica com os demais nós do multicomputador tipicamente por meio de mecanismos de troca de mensagem.

Os multicomputadores podem ser diferenciados entre processadores paralelos massivos (MPPs) e *clusters* de estações de trabalho (COWs) (TANENBAUM, 2007). Os MPPs são formados por CPUs padronizadas interligadas por uma rede proprietária de alta velocidade, dimensionada para privilegiar largura de banda e baixa latência de comunicação. Este tipo de organização apresenta como características gerais o alto desempenho, principalmente de E/S, e uma preocupação com a tolerância a falhas, dada a grande probabilidade de ocorrência de defeitos em função de uma organização arquitetural dependente do funcionamento paralelo de um grande número de CPUs. Exemplos de MPPs são a implementação do IBM BlueGene, baseada em CPUs PowerPC superescalares de emissão dual, e a RedStorm do Sandia National Laboratory, baseada em CPUs AMD Opteron.

Os COWs, por sua vez, são arranjos de computadores comuns, interligados via uma rede de comunicação utilizando como interface placas de rede convencionais. Apresentam como principal característica o baixo custo em relação aos MPPs, dado a dependência de "*hardware* de prateleira", o que segundo Tanenbaum (2007) tende a aumentar a sua popularização em relação ao uso de MPPs. Um exemplo clássico de COW é a implementação do mecanismo de busca do Google, cuja arquitetura define diferentes papéis para cada nó do *cluster* (manipulador de consultas, servidores de índice, servidores de anúncios, etc.) e um alto grau de replicação, de tal maneira a aumentar a disponibilidade e diminuir os efeitos das falhas que ocorrem nos componentes dos nós.

Uma categoria correlata aos COW são os computadores em grade (*grid computers*), cuja grande diferença em relação aos COW é o fato de que são compostos por computadores comuns distribuídos em grandes áreas geográficas e interconectados por meio de WAN (*Wide Area Network*). Um exemplo de projeto de computação em grade foi o SETI@home (*Search for Extraterrestrial Intelligence,* ou Busca por Inteligência Extraterrestre), o qual fazia uso de PCs comuns cadastrados junto ao projeto pelos seus proprietários / usuários. A idéia do SETI, que é um princípio válido para qualquer sistema de computação em grade, foi aproveitar a grande parcela de tempo ocioso dos PCs comuns para distribuição e paralelização do processamento dos sinais recebidos por radiotelescópios. Isto lhe permitiu operar, em 2006, a 257 TeraFLOPS (trilhões de operações de ponto flutuante por segundo), utilizando para tanto mais de 5 milhões de computadores espalhados por mais de 200 países (PATTERSON; HENNESSY, 2009).

As principais ressalvas em relação ao uso de multicomputadores residem justamente nos pontos negativos do modelo baseado em APIs de troca de mensagens (ver Seção 2.3.2.3),

que é tipicamente implementado por este tipo de arquitetura para comunicação e sincronização.

2.3.4.2 Redes neurais

Outra concepção de arquitetura que apresenta paralelismo é a das redes neurais artificiais (RNA), que são inspiradas no funcionamento e organização do cérebro humano e dos seus neurônios. Suas principais aplicações são a classificação e o reconhecimento de padrões, os quais são viabilizados pela capacidade intrínseca das RNAs de aprender por meio da experiência, mediante técnicas de treinamento da rede.

As RNAs apresentam por característica operar simultaneamente sobre um conjunto de dados ou padrões, que são ponderados (mediante fatores denominados "pesos sinápticos") e fornecidos como entrada a um conjunto de elementos processadores (os neurônios). Cada neurônio, por sua vez, aplica uma função de transferência e gera uma saída, que eventualmente será fornecida como entrada a outro conjunto de neurônios ou como saída do sistema. O treinamento da rede geralmente tem por objetivo ajustar os pesos sinápticos de tal maneira a refinar a saída da rede em função da aplicação desejada (por exemplo, gerar a classificação de um determinado padrão de entradas).

A Figura 22 mostra um exemplo de rede neural genérica com 3 camadas de neurônios. Dada a característica de dispersão do processamento de uma RNA em um conjunto de neurônios processadores, pode-se afirmar que este processamento apresenta um alto grau de paralelismo. Isto motivou a implementação em *hardware* de diversas arquiteturas de redes neurais, baseadas em diversas tecnologias de construção tais como VLSI (*Very Large Scale of Integration*), ASICs (*Application-Specific Integrated Circuits*) e FPGAs (LIAO, 2001). Segundo Liu e Liang (2005), a tecnologia de FPGAs é adequada para RNAs implementadas em *hardware* dadas as características de flexibilidade, reconfigurabilidade e relativamente baixo custo (quando comparado a ASICs) deste tipo de dispositivo, aliadas às necessidades do processo de aprendizagem e, até mesmo, de reconfiguração típico da teoria de redes neurais.

Em relação a sua aplicabilidade ao modelo de execução do PON, embora a propagação de dados ponderados possa apresentar uma dinâmica semelhante à de propagação de notificações, estas são utilizadas no PON principalmente como propagadoras de resultados de relações lógico-causais. Ou seja, o contexto de aplicação dos dados e a forma como eles são processados e seu significado interpretado é distinta nos dois casos.

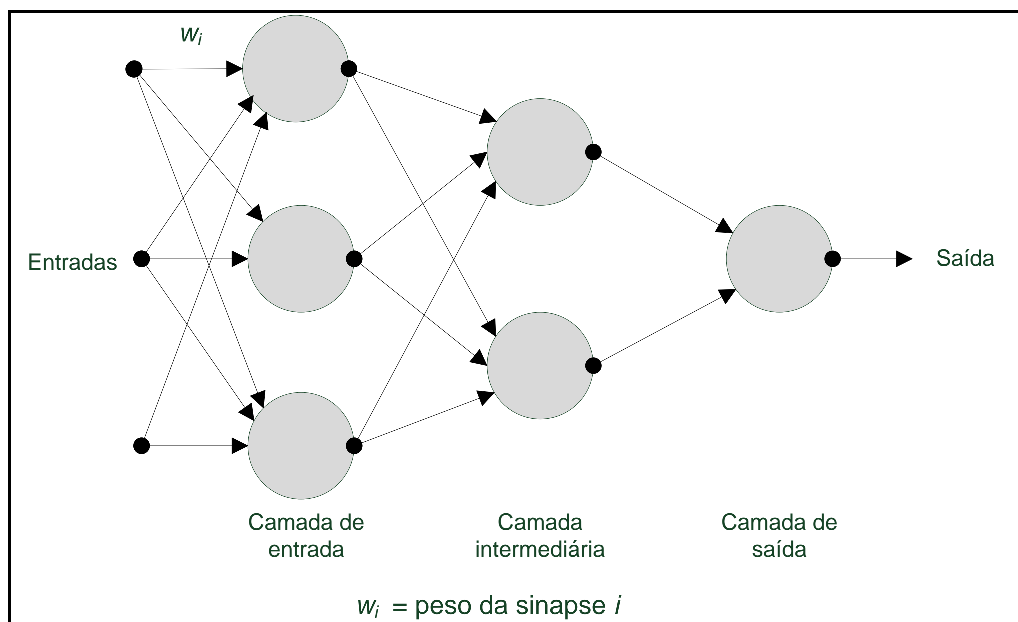


Figura 22 - Exemplo genérico de rede neural

(Fonte: autoria própria)

2.3.4.3 Arquiteturas heterogêneas e/ou sensíveis ao domínio da aplicação

Cavin et al. (2008) apresentam uma revisão de arquiteturas emergentes que são sensíveis ao domínio da aplicação, as quais devem utilizar como um dos preceitos de sua concepção a capacidade de suportar aplicações que são fundamentalmente paralelas. Neste grupo são incluídos os seguintes tipos de arquitetura:

- Arquiteturas heterogêneas, formadas por núcleos especializados em alguma tarefa (p. ex. processamento de sinal) associados a núcleos convencionais de processamento. Nesta categoria se enquadram as arquiteturas com co-processadores conforme proposto por Tanenbaum (2007) ou ainda o co-processador PON conforme proposto por Peters (2012).
- Arquiteturas mórficas ou bioinspiradas, adaptadas para atuar em determinado conjunto de problemas, frequentemente aproveitando e implementando conceitos de áreas tais como a biologia. Estas arquiteturas podem fazer uso da física dos dispositivos computacionais para mesclar computação analógica com digital, obtendo os benefícios de desempenho e relativamente baixo consumo da primeira com a capacidade de expressão lógica e de controle da computação analógica oferecida pela segunda. Além disso, apresentam o potencial de imitar o cérebro humano em aspectos tais como

predição de memória (i. e. adaptabilidade dos seus circuitos a dados passados de tal maneira a processar mais eficientemente dados futuros). Exemplos de arquiteturas mórficas são as redes não-lineares celulares (*cellular nonlinear networks*), que consistem de *arrays* mistos de elementos de processamento e sensores, interconectados seguindo uma organização dependente do problema e voltados principalmente para aplicações de processamento de sinal.

De forma geral, técnicas arquiteturais heterogêneas e/ou sensíveis ao domínio da aplicação têm sido empregadas para abordar a questão da necessidade de aumento do desempenho. Embora arquiteturas *multicore* homogêneas talvez careçam de maior desenvolvimento em tecnologias (linguagens) de programação paralela e também em tecnologias de gerenciamento de memória para melhorar o desempenho (CAVIN et al., 2008), arquiteturas *multicore* heterogêneas podem aproveitar circuitos especializados (processadores digitais de sinais, processadores de I/O, etc.) para melhorar desempenho de aplicações que estejam alinhadas com as funcionalidades que estes circuitos especializados oferecem.

No contexto do PON, algumas técnicas e modelos de circuitos e estruturas arquiteturais que foram desenvolvidos podem ser categorizados, em algum aspecto, como heterogêneos e/ou sensíveis ao domínio da aplicação. A seção a seguir detalha estes desenvolvimentos.

2.3.4.3.1 Modelos de arquitetura baseados no PON

Conforme apresentado na Seção 2.2.4, alguns esforços já foram realizados para viabilizar a implementação em *hardware* de estruturas que permitam a execução mais fidedigna de aplicações concebidas segundo o PON. Ou seja, objetivou-se, com estes esforços, propor técnicas ou elaborar modelos de circuitos e de estruturas arquiteturais de *hardware* que permitissem construir ou simular um ambiente no qual o mecanismo de notificações pudesse, de fato, executar de forma potencialmente paralela. Isto minimizaria as questões de *overhead* de execução apresentadas pelas abordagens nas quais as notificações são implementadas por meio de *software* executando sequencialmente em uma arquitetura de computação convencional.

Witt et al. (2011) propuseram um conjunto de circuitos sequenciais / combinacionais que implementam em *hardware* os elementos do metamodelo de notificações (WITT et al.,

2011)(SIMÃO et al., 2012b). Estes circuitos podem ser replicados e interconectados em FPGA de forma a implementar a cadeia de notificações do PON para uma aplicação específica. A maneira como este modelo se insere no ambiente de desenvolvimento e execução de aplicações PON é ilustrada na Figura 1 (c) (ver Seção 1.4).

Segundo este modelo, a execução da cadeia depende exclusivamente da propagação de sinais digitais entre os circuitos que representam os elementos do metamodelo. Esta propagação pode ocorrer de forma simultânea, partindo de elementos independentes entre si (p. ex., de uma suposta premissa P1 para uma suposta condição C1 e de uma suposta premissa P2 para a mesma condição C1, desde que as premissas P1 e P2 sejam independentes), portanto permitindo a ocorrência do paralelismo teórico previsto no PON. Além disso, dado que a propagação das notificações ocorre em um baixíssimo nível de abstração (propagação de sinais digitais síncronos nos circuitos do modelo), o desempenho de propagação nesta abordagem tende a ser muito superior ao desempenho obtido na abordagem na qual o mecanismo de notificações é implementado em *software*, a qual depende de buscas em estruturas de dados conforme já explicado anteriormente.

Os ganhos em desempenho foram atestados por meio da implementação de um caso de estudo que consistiu em um simulador de sistema de telefonia, conforme já havia sido implementado em *software* por Linhares et al. (2011), cuja máquina de estados foi modelada levando-se em consideração os eventos (retirada e recolocação do telefone no gancho, etc.) e estados (ocioso, tocando, chamando, em conversação, etc.) comuns deste tipo de sistema.

O modelo de máquina de estados do sistema telefônico foi então adaptado para o modelo de notificações do PON por meio do mapeamento dos estados e das ocorrências de eventos para *Attributes,* cuja variação dispara a cadeia de notificações e atualiza o valor do estado para cada um dos terminais telefônicos envolvidos na simulação. Este mapeamento foi efetuado utilizando-se um modelo de Rede de Petri, o qual permitiu identificar mais claramente as *Rules* definidas no sistema e as ligações que compõem a cadeia de notificações.

Em seguida, o modelo de notificações obtido foi implementado em *hardware*, conforme a arquitetura apresentada na Figura 23. Posteriormente, comparou-se o desempenho desta implementação com o desempenho da implementação em *software* efetuada por Linhares et al. (2011), comprovando-se a ampla superioridade da implementação em *hardware* no que tange à velocidade de propagação das notificações (aumento de desempenho em até 10.000 vezes).

Como desvantagem, o modelo desenvolvido por Witt et al. (2011) apresenta baixa escalabilidade, dado que mapeia diretamente os elementos da cadeia de notificações de

determinada aplicação para instâncias dos circuitos digitais correspondentes em *hardware.* Em consequência, como cada uma destas instâncias consome uma determinada quantidade de elementos lógicos disponíveis na FPGA, a quantidade de instâncias (e, portanto, de elementos de cadeia de notificações) seria limitada pela capacidade do dispositivo de FPGA. Ainda, a passagem da execução de uma aplicação para outra requer a reconfiguração da FPGA, o que diminui a flexibilidade no uso desta abordagem em relação a um modelo de implementação em *software* (o qual teria seu código binário simplesmente carregado da memória pela plataforma / sistema operacional) e, adicionalmente, inviabiliza a sua implementação em *hardware* não (re)programável (p. ex. ASIC).

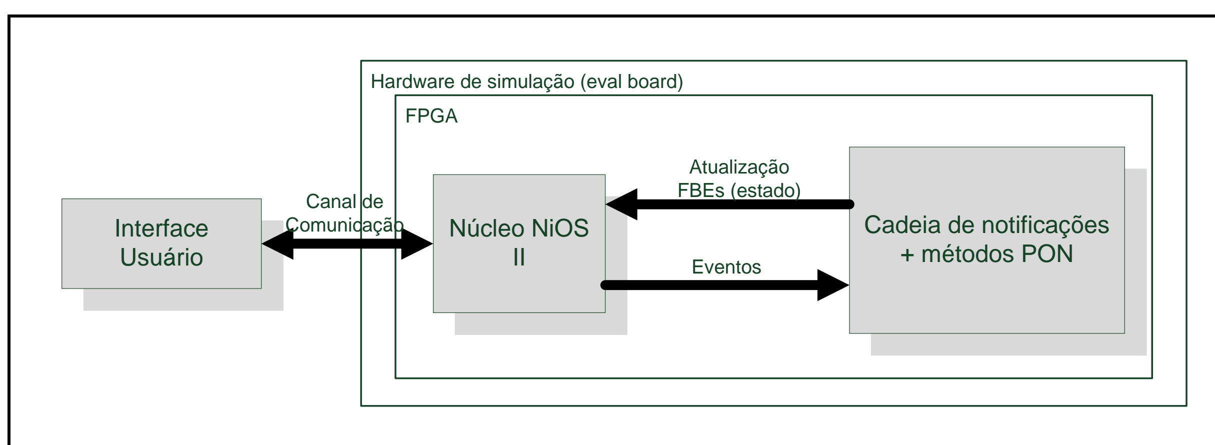


Figura 23 – Arquitetura da simulação do sistema de telefone implementado segundo o PON em *hardware* (Fonte: autoria própria)

Jasinski (2012) propôs um *framework* que permite efetuar a descrição de uma aplicação PON e compilar esta descrição. O resultado da compilação é um código VHDL, o qual descreve um circuito sequencial / combinacional que implementa a aplicação PON em questão em *hardware*, podendo este código ser utilizado para simulações ou até mesmo para sintetização de um circuito real em uma FPGA. A descrição dos elementos do PON suportados pelo *framework* (*Attributes, Premises, Conditions* e *Instigations*) para concepção da aplicação é efetuada na linguagem YAML, a qual permite efetuar descrições segundo uma estrutura hierárquica de maneira mais legível do que o seu equivalente na linguagem XML convencional. Os circuitos gerados para implementar a aplicação PON são baseados nos elementos do metamodelo do PON em *hardware* desenvolvidos por Witt et al. (2011).

O *framework* desenvolvido por Jasinski apresenta algumas restrições do ponto de vista de implementação de conceitos do PON:

- Não executa resolução de conflitos entre regras, portanto a própria aplicação deve ser concebida de tal maneira a ser isenta de conflitos;
- Não permite a definição de prioridades para regras, embora a ordem de declaração no arquivo YAML crie um sistema simples de prioridade (regras mais próximas ao início do arquivo têm maior prioridade);
- Não suporta métodos desenvolvidos segundo o PI;
- Não suporta a execução de instigações em série, apenas em paralelo;
- Suporta somente versões simplificadas de métodos (comparação de atributos com constantes).

Peters (2012), por sua vez, propõe implementar a cadeia de notificações do PON em um co-processador, cuja finalidade é acelerar a propagação das notificações até a ativação de métodos, portanto também objetivando melhorar o desempenho em relação à implementação em *software.*

O *hardware* do coprocessador é configurável, em tempo de projeto, em relação à quantidade de instâncias de elementos PON de cada tipo suportado (*Attribute, Premise, Condition* e *Rule*) que serão utilizados para a construção da aplicação PON em questão. Já os *Methods* do PON são codificados em *software* segundo o PI, utilizando-se do *framework* PON C++ já desenvolvido (BANASZEWSKI, 2009) (VALENÇA, 2012), e executados por um núcleo von Neumann. Como optou-se por sintetizar o coprocessador do PON em FPGA, em particular para os dispositivos Altera, utilizou-se o processador Altera NiOS II para implementação do núcleo von Neumann executor dos *Methods,* também sintetizado no dispositivo FPGA.

O resultado do processo de desenvolvimento de uma aplicação PON segundo esta abordagem é um SoC (*system on chip*) constituído pelo núcleo NiOS II e pelos seus periféricos (memória, etc.), incluindo o coprocessador PON configurado para a aplicação específica em termos de quantidade de elementos da cadeia de notificações e dos relacionamentos entre eles. A interação do coprocessador PON com o *software* executando no núcleo NiOS II se dá por meio de mapeamento do coprocessador em memória, permitindo o acesso pelo *software* a filas internas (FIFOs) configuradas que contêm, por exemplo, informações sobre as regras aprovadas, e também por meio de sinais de interrupção utilizados pelo coprocessador para notificar o núcleo NiOS II sobre a ativação de uma regra.

A Figura 24 ilustra a arquitetura do sistema, ressaltando a interconexão entre o processador NiOS II e os seus periféricos (entre eles o coprocessador PON).

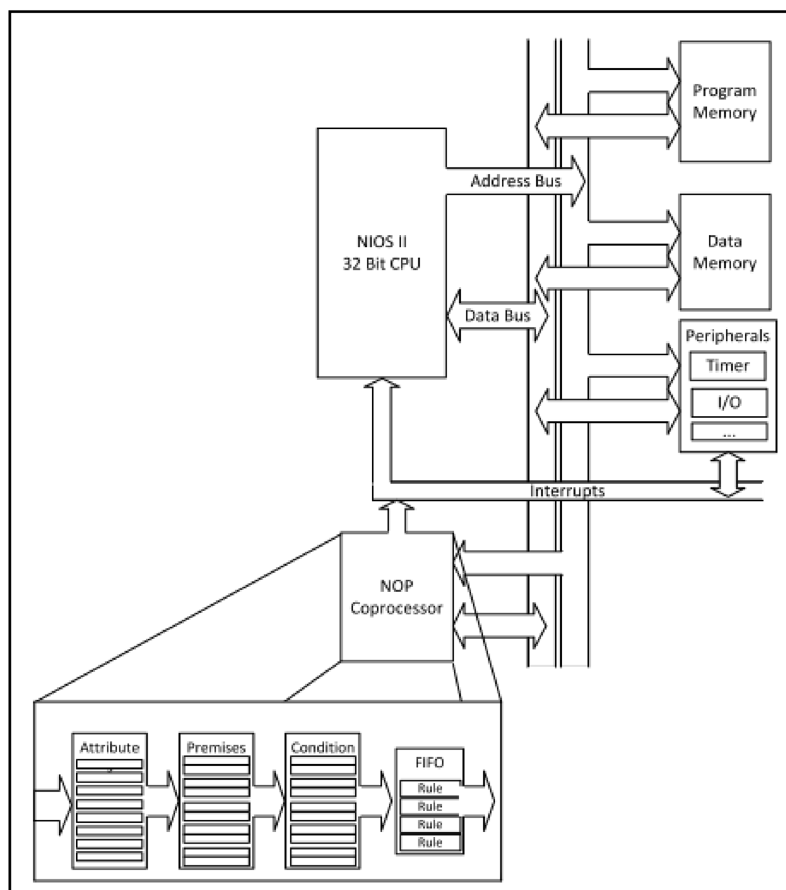


Figura 24 – Arquitetura do SoC utilizando coprocessador PON
(Fonte: Peters, 2012)

Como desvantagem, o modelo proposto por Peters é essencialmente híbrido, no sentido de que depende necessariamente de um núcleo von Neumann (o processador NiOS II) para execução da lógica dos *Methods,* a qual ocorre de forma sequencial. Ainda que o modelo pudesse ser estendido para múltiplos núcleos, cada um deles apresentaria a complexidade de um núcleo von Neumann completo, diferentemente da solução de Witt et al. que mapeia os elementos de metamodelo de notificações para circuitos relativamente simples. Além disso, apresenta as mesmas limitações já discutidas em relação ao trabalho de Witt et al. no que diz respeito a escalabilidade e flexibilidade

2.3.5 Organização de memória

Nesta seção apresenta-se uma visão de diferentes tecnologias e modelos utilizados para concepção e organização de memória em arquiteturas paralelas.

Do ponto de vista arquitetural, a principal questão diz respeito à memória disponível ser única e compartilhada entre os diferentes núcleos de processamento (como nos chamados multiprocessadores) ou distribuída e de acesso individual por cada núcleo (como nos chamados multicomputadores). A Figura 25 demonstra a diferença entre os modelos na forma de diagramas de blocos. Este assunto foi introduzido na Seção 2.3.1 e é detalhado a seguir.

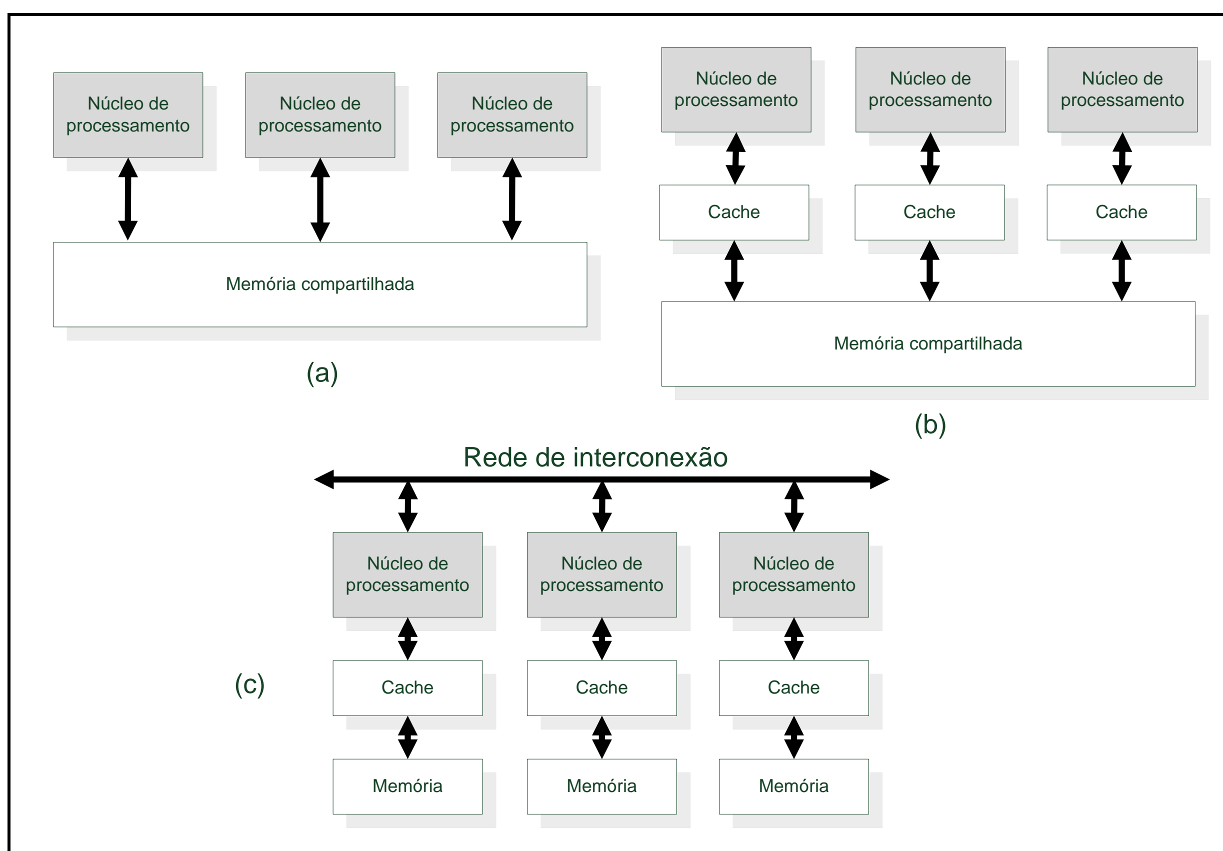


Figura 25 – Arquiteturas de multiprocessador (a), multiprocessador com caches individuais (b) e multicomputador (c)
(Fonte: autoria própria)

Todas estas organizações apresentam vantagens e desvantagens. No caso de memória compartilhada, conforme mostrado na Figura 25 (a) e (b), todos os núcleos de processamento podem acessar diretamente qualquer posição de memória por meio de instruções de escrita / leitura do seu conjunto de instruções. Isto tende a simplificar a construção do *software*

paralelo na medida em que não existe uma dependência programática de APIs específicas ou outras formas de comunicação para se implementar o compartilhamento de dados, bastando que estes possuam um endereço conhecido e acessível por todos os núcleos de processamento que pretendem acessá-los. A estes sistemas se denomina SMPs (*symmetric shared-memory processors,* ou Processadores de Memória Compartilhada Simétrica) (HENNESSY; PATTERSON, 2007).

A principal desvantagem da construção arquitetural com memória compartilhada em SMPs reside na dificuldade de escalabilidade. Isto ocorre porque cada núcleo extra que pudesse ser inserido no projeto do multicomputador deveria ser interligado à memória compartilhada, o que tornaria o *hardware* mais complexo e tenderia a diminuir o desempenho de acesso à memória de cada núcleo individualmente, dado que a banda de acesso à memória seria também dividida entre os vários núcleos.

Por outro lado, no caso da memória distribuída, conforme mostrado na Figura 25 (c), a comunicação entre processos ou *threads* executados por diferentes núcleos deve ser, obrigatoriamente, implementada via troca de mensagens, o que torna o *software* mais complexo (ver Seção 2.3.2.3 para maiores detalhes). No entanto, este modelo arquitetural apresenta como vantagem a maior escalabilidade, dado que é relativamente mais simples inserir novos computadores na rede de interconexão do multicomputador. Isto ocorre porque a rede de interconexão é tipicamente uma rede multiponto, sobre a qual executam protocolos de rede que são mais simples de se escalar por *software.*

Os modelos híbridos, por sua vez, procuram aproveitar a característica de distribuição dos sistemas multicomputadores, porém oferecendo para a camada de aplicação um espaço lógico de endereçamento único. Nestes modelos, caso o *software* de aplicação deseje acessar um dado que não se encontra fisicamente presente na memória local do núcleo de processamento onde a *thread* corrente executa, é responsabilidade do sistema operacional buscar o dado no local onde ele se encontra (memória de outro núcleo) e disponibilizá-lo para a *thread* corrente; esta funcionalidade pode ser obtida por meio do tratamento de uma exceção de falta de página, semelhante ao que ocorre nas MMUs (*Memory Management Units*) que implementam memória virtual. A estes sistemas costuma-se denominar DSMs (*Distributed Shared Memory,* ou Memória Compartilhada Distribuída) (HENNESSY; PATTERSON, 2007) (TANENBAUM, 2007).

Independente do modelo arquitetural de distribuição / compartilhamento de memória, existe uma questão ortogonal a ser tratada que é relativa à *localidade* de acesso aos dados; ou seja, quão distante está um determinado dado do núcleo de processamento que pretende

acessá-lo. Neste contexto, o conceito de distância é mais relevante se for entendido como o tempo dispendido para a realização / finalização de uma operação de escrita ou leitura.

No caso do modelo de memória compartilhada a questão da localidade é influenciada pela forma de implementação do espaço de endereçamento lógico. Os SMPs geralmente são arquiteturas do tipo UMA (*Uniform Memory Access,* ou Acesso Uniforme à Memória), o que significa que o tempo de acesso a qualquer posição da memória é uniforme. Já os DSMs são arquiteturas do tipo NUMA (*Non-Uniform Memory Access,* ou Acesso Não-Uniforme à Memória), o que significa que o tempo de acesso a diferentes posições de memória pode variar dependendo da localização do dado (memória do próprio núcleo ou de outro núcleo do multicomputador).

Isto evidencia que, no caso de sistemas de multicomputadores (DSMs ou de troca de mensagens), favorecer a localidade significa, principalmente, limitar o número de acessos a regiões de memória que estão no espaço de endereçamento dos demais núcleos. Em todos os casos, levar em consideração a localidade de acesso quando da distribuição / paralelização do *software* pode ajudar a melhorar o desempenho, na medida em que permite limitar o conjunto de dados a ser processado por cada *thread* e minimizar acessos mais lentos (FATAHALIAN et al., 2006).

2.3.5.1 Hierarquias de memória

Do ponto de vista de memória compartilhada, um fator relevante diz respeito à existência ou não de diferentes níveis de hierarquia de memória (por exemplo, níveis de memória *cache*) e até que ponto estes níveis são compartilhados ou não. A Figura 25 (b) mostra um exemplo no qual a memória *cache*, de nível único, é individual para cada núcleo de processamento, no entanto a memória principal se mantém compartilhada entre diversos núcleos.

Conforme resumido por Linhares (2001), denomina-se memória *cache* aos níveis de memória existentes entre a unidade de busca da instrução do processador e a memória principal do sistema (no caso dos multiprocessadores, a memória RAM compartilhada). O objetivo das memórias *cache* é maximizar o desempenho de acesso a dados em memória, oferecendo tempos de acesso mais rápidos do que os tempos de acesso à memória principal. Isto se obtém às custas de uma maior proximidade ao núcleo de processamento, o que implica

em uma densidade menor e, consequentemente, em um espaço de armazenamento muitas vezes menor do que o oferecido pela memória principal.

As memórias *cache* implementam políticas de substituição de seus blocos de dados por blocos de dados do nível imediatamente inferior, sendo que a substituição ocorre sempre que é efetuado um acesso a um endereço contido em um bloco que não está na *cache* (evento denominado de *cache miss*). As políticas de substituição procuram privilegiar os dados que são mais frequentemente acessados pelo núcleo de processamento, geralmente levando em consideração questões de localidade espacial (i. e., manter em *cache* dados de endereços próximos aos correntemente sendo acessados) e localidade temporal (i. e. manter em *cache* dados de endereços que foram recentemente acessados). Em função desta característica, *cache misses* tendem a ser mais numerosos em aplicações com baixa localidade espacial no que diz respeito ao acesso à memória (p. ex., acesso intensivo a bases de dados), o que tende a prejudicar arquiteturas voltadas principalmente a ILP, que são mais afetadas por longas latências no acesso a dados ((KONGETIRA; AINGARAN; OLUKOTUN, 2005).

A memória *cache* em si pode ser dividida em vários níveis, conforme mostrado na Figura 26. O objetivo desta organização é prover uma variação mais gradual entre o nível de alta capacidade de armazenamento com tempo de acesso relativamente alto (caso dos níveis de memória principal - memória RAM - e de armazenamento secundário) e o nível de baixa capacidade de armazenamento com tempo de acesso relativamente baixo (caso dos registradores embutidos no núcleo de processamento). Cada nível intermediário de *cache* também implementa políticas de substituição de dados para privilegiar localidade espacial e localidade temporal.

Segundo Kogge et al. (2008), uma tendência do ponto de vista tecnológico é a de que a capacidade de se ampliar os diferentes níveis de memória *cache* para minimizar o efeito crescente da diferença de desempenho entre os processadores e os dispositivos de memória principal esteja chegando a um limite. Isso significa que muito provavelmente o desempenho das memórias *cache* está chegando a um ponto de estagnação.

Além disso, em uma arquitetura como a mostrada na Figura 25 (b), a existência de *caches* individuais para cada núcleo cria a possibilidade de que, em determinado instante, múltiplas *caches* possuam cópias dos mesmos dados, Portanto, faz-se necessário neste tipo de organização que os conteúdos das diversas *caches* de cada núcleo sejam coerentes entre si, sendo que esta coerência é obtida por meio de protocolos específicos a serem abordados na sequência, os quais naturalmente agregam complexidade ao projeto da arquitetura.

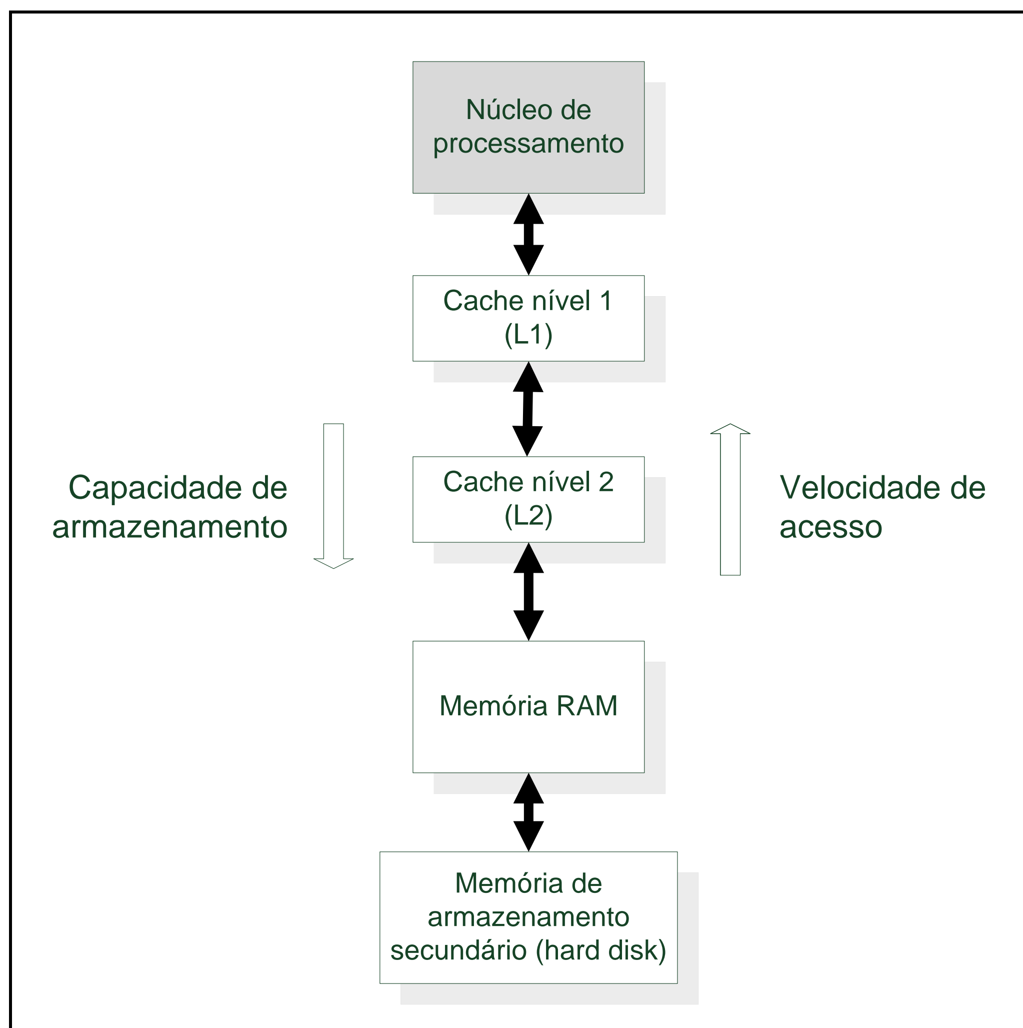


Figura 26 – Relação entre níveis de hierarquia de memória

(Fonte: adaptado de LINHARES, 2001, p. 13)

Devido ao aumento na capacidade dos diversos níveis de memória *cache,* aliado ao aumento da velocidade do *clock,* não se pode mais assumir para todas as implementações que o tempo de acesso a qualquer dado em determinado nível de memória *cache* é constante. Em função desta característica vem sendo desenvolvida pesquisa com foco em arquiteturas que implementam memórias *cache* do tipo NUCA (*Non-Uniform Cache Arrays,* ou Arrays de Cache Não Uniforme), as quais não apresentam uma estrutura hierárquica constante e uniforme do ponto de vista do processador.

O princípio básico que norteia uma arquitetura NUCA seria manter dados que são mais acessados por um processador fisicamente próximos a ele, sendo que estes dados que estão mais próximos do processador apresentam tempos de acesso menores do que os que estão mais distantes. Esta premissa implica em que os bancos de memória *cache* tenham

tempos de resposta diferenciados, ou seja, não necessariamente a resposta de um banco de memória *cache* ocorrerá simultaneamente à dos demais bancos (FREITAS; ALVES; NAVAUX, 2009).

Kim, Burger e Keckler (2002) apresentam diversos modelos de arquitetura de NUCA, incluindo propostas de políticas de substituição / realocação dinâmicas que levam em consideração o princípio da localidade para aumentar o desempenho médio de acesso, por meio do reposicionamento dos dados mais acessados em blocos com menor tempo de acesso. Vale ressaltar que estas políticas são distintas das implementadas em *caches* de múltiplos níveis, nas quais o conceito de níveis hierárquicos é mais evidente e não depende, necessariamente, de questões de localidade e tempo de acesso como nas NUCA.

Wang e Wang (2006) apresentam o conceito de memória associativa por *thread.* Segundo este modelo, as posições de armazenamento de determinado nível da hierarquia de memória (p. ex. blocos de *cache*) agregam informação explícita a respeito de qual *thread* está utilizando aquela posição. Segundo os autores esta técnica ajuda a melhorar o desempenho médio do acesso à memória por minimizar a ocorrência de contenções, no sentido de que evita que o conteúdo de um bloco de memória em *hardware* seja periodicamente substituído (p. ex. pela ocorrência de um *cache miss*) em função do chaveamento de contexto entre diferentes *threads.* Ou seja, a política de substituição de blocos de memória privilegia que dados da mesma *thread* sejam substituídos entre si quando necessário, minimizando as interferências entre *threads* que ocorrem principalmente em aplicações de intenso acesso à memória tais como processamento de multimídia e operações de bancos de dados.

O modelo de programação Sequoia (FATAHALIAN et al., 2006) apresenta uma alternativa arquitetural classificada como "sensível à hierarquia de memória" (*memory hierarchy aware programming model*). Este modelo propõe uma linguagem de programação com abstrações próprias para movimentações e comunicação entre diferentes níveis de hierarquia, baseado na justificativa de que o programador pode extrair melhor desempenho se possuir um maior conhecimento a respeito da arquitetura da máquina computacional, de forma a organizar acessos a dados levando em consideração questões de localidade *versus* desempenho de acesso provido pelos diferentes níveis (*cache,* DRAM, etc.). Esta abordagem é distinta dos modelos de programação tradicionais, nos quais embora possa se definir níveis de particionamento entre diferentes núcleos paralelos de processamento (p. ex. esquemas de PGAS), geralmente não é possível explicitar a alocação de dados entre diferentes níveis hierárquicos de memória. Um exemplo de arquitetura que implementa instruções sensíveis à hierarquia de memória é a IBM Cell (KAHLE, 2005).

Uma forma de otimizar o uso de banda de acesso à memória é desvincular a operação de acesso, que envolve uma requisição ou endereçamento de posição de memória, da operação de disponibilização do dado sendo acessado propriamente dito, que corresponde à resposta do *hardware* que implementa a memória. Este tipo de técnica é chamado de transação dividida (*split transaction*) (WANG, 2001) e depende de *hardware* mais complexo, dado que procura viabilizar a existência de múltiplas requisições pendentes, o que exige formas de se armazenar estas requisições em *hardware* para realizar *matching* com as correspondentes respostas. Além disso, a implementação de transações divididas deve levar em consideração a possibilidade de requisições conflitantes para o mesmo conjunto de dados ou de requisições cujas respostas são geradas fora de ordem.

Finalmente, uma forma de se minimizar o número de *cache misses*, em acessos a dados, é implementar técnicas de *prefetching*. Estas consistem em antecipar a busca do dado em memória, em relação à instrução que vai utilizá-lo efetivamente, permitindo atender a um eventual *cache miss* antes do dado ser efetivamente processado. Wang (2001) discorre sobre técnicas de *prefetching* implementadas em *software,* por meio de instruções específicas que disparam o *prefetching* nos instantes adequados, inseridas em tempo de compilação; e também sobre técnicas de *prefetching* implementadas em *hardware*, as quais dependem de predições de padrões de acesso à memória baseadas em acessos anteriores e detectadas desta forma pelo *hardware* (portanto, efetuadas em tempo de execução).

2.3.5.1.1 Coerência de cache

Em arquiteturas multiprocessadas com *caches* individuais para cada núcleo, conforme a Figura 25 (b), existe a possibilidade de que um bloco de memória, presente em duas ou mais *caches,* seja atualizado na *cache* de um dos processadores e precise ser utilizado em seguida pelo(s) outro(s) processador(es) que também a mantém em *cache*. Neste caso, deve-se garantir que a alteração efetuada na *cache* do primeiro processador seja de alguma forma perceptível pelos outros processadores cujas *caches* também contêm aquele dado, ou seja, deve-se manter a coerência entre os conteúdos das diversas *caches.*

A coerência dos dados nas *caches* é obtida por meio da implementação de protocolos de coerência. Segundo Hennessy e Patterson (2007), manter a coerência depende de se utilizar alguma estratégia para monitorar os blocos de dados que são compartilhados por mais de uma

cache, sendo que em função desta estratégia pode-se identificar duas grandes categorias de protocolos:

- Protocolos baseados em diretório: dependem de uma área de dados centralizada e compartilhada (diretório) capaz de armazenar o *status* de compartilhamento de todos os blocos de dados.
- Protocolos baseados em monitoração (*snooping*): dependem de uma implementação que permita à *cache* individual de cada núcleo monitorar (*snoop*) o barramento de memória compartilhado, ao qual todas as outras *caches* também estão ligadas, para verificar se os acessos a memória sendo efetuados dizem respeito a algum bloco de dados que está armazenado nela própria.

A seguir descreve-se resumidamente alguns dos protocolos de monitoração comumente utilizados.

1) Protocolos baseados em invalidação ou atualização por escrita

Estes protocolos são implementados de tal maneira a garantir que uma atualização de bloco de dados em determinada *cache* cause uma invalidação ou uma atualização do mesmo bloco nas demais *caches.*

Hennessy e Patterson (2007) comentam que as versões que efetuam invalidação geralmente são mais eficientes porque tendem a gerar menos tráfego no barramento de memória compartilhada. Isto ocorre devido ao fato de que a invalidação somente precisa ocorrer uma vez para cada bloco de *cache* (e não para cada *byte*, no caso de atualização) e que múltiplas sobrescritas em sequência no mesmo bloco geram uma única invalidação (ao contrário da atualização, que deveria ocorrer para cada sobrescrita).

2) Protocolo MESI

Este protocolo recebe um acrônimo relativo às iniciais dos nomes dos 4 estados que podem ser atribuídos a um bloco de *cache* (M – *Modified,* E – *Exclusive,* S – *Shared* e I – *Invalid*) (PAPAMARCOS; PATEL, 1984).

Os diferentes estados do bloco de *cache* são utilizados para garantir que a *cache* de cada núcleo tenha uma ação precisa a tomar quando houver um acesso aos mesmos dados efetuado por um outro núcleo. Estas ações incluem efetuar a escrita efetiva do dado no nível de memória inferior, para viabilizar que a cópia atualizada seja armazenada nas *caches* dos demais núcleos, ou mesmo invalidar o conteúdo da sua própria *cache* devido a uma escrita

mais recente efetuada por outro núcleo. Ou seja, o protocolo MESI é uma implementação mais sofisticada de um protocolo de invalidação / atualização.

3) Protocolo de coerência transacional (TCC, *ou Transactional Memory Coherence and Consistency*)

Proposto por Hammond et al. (2004), este protocolo visa aproveitar a característica de sincronização e tolerância a latência de sistemas baseados em mensagens (MPI), porém mantendo a facilidade de programação provida por esquemas baseados em compartilhamento de memória ao mesmo tempo em que requer um *hardware* relativamente mais simples do que o *hardware* necessário para a implementação de outros protocolos de coerência de *cache.*

A dinâmica de funcionamento do TCC consiste em agrupar um conjunto de instruções de escrita na memória na forma de uma operação atômica, denominada "transação" (este agrupamento deve ser levado em consideração na construção do *software*, portanto de certa maneira afeta o modelo de programação). As transações são efetivadas na memória (*commited*) somente após o seu término e mediante uma arbitragem a respeito da permissão de escrita com os outros elementos (núcleos) que podem gerar transações. Além disso, uma transação é especulativa, no sentido de que pode ser desfeita (*rollback*) caso tenha sido efetuada sobre dados que já tinham sido modificados por outra transação.

A organização das escritas na forma de transações apresenta como vantagens a otimização do uso de banda de intercomunicação, dado que o número de pacotes / mensagens pequenas tende a diminuir, e a consequente diminuição da latência total das operações. Além disso, o controle de sequenciamento entre transações, ao invés de operações primitivas de leitura / escrita, garante implicitamente a sincronização entre as operações que compõem uma transação.

4) Protocolo Ecoli

Este protocolo foi implementado na arquitetura SDAARC, proposta por Eschmann et al. (2002). e tem por objetivo atender à característica de migração dinâmica de *microthreads* prevista por aquela arquitetura (ver Seção 2.3.2.2, subseção sobre implementações de TLP).

A arquitetura de memória do SDAARC é do tipo COMA (*Cache-Only Memory Architeture,* ou Arquitetura de Memória Somente com *Cache*), o que significa que toda a memória disponível é dividida entre os diferentes núcleos e tratada por cada um deles como se fosse uma *cache* (denominada "memória de atração"). Um *miss* em uma memória de atração causa o envio de uma requisição de acesso às demais memórias de atração, com a

consequente movimentação do bloco de dados que se deseja acessar. Hennessy e Patterson (2007) comentam que implementações de COMA são relativamente complexas e nenhuma máquina puramente COMA foi construída, embora algumas simplificações tenham sido propostas tais como o F-COMA (movimentação em um único nível de memória, chamado de nível "lar") e o S-COMA (movimentação fazendo uso de técnicas de memória virtual ao invés de *hardware* específico implementado na própria memória *cache*).

No contexto da arquitetura SDAARC implementando COMA, o protocolo Ecoli define um conjunto de estados (E – *Exclusive,* C – *Clone,* O – *Original,* L – *Leaving* e I – *Invalid*) com propósito semelhante ao dos estados do protocolo MESI, porém levando em consideração que não existe um nível inferior de memória (memória principal) no qual se efetue atualizações. Além disso, dado que as *microthreads* definidas na arquitetura SDAARC consomem argumentos e geram resultados (ou seja, em um esquema de fluxo de dados), o protocolo Ecoli não trata dos conceitos de escrita / leitura (*load / store*) convencionais, mas sim define o conceito de "aplicação" (*apply*) que consiste na movimentação de dados para passagem de resultados de uma *microthread* como argumento para a(s) *microthread(s)* que dela dependem (ESCHMANN et al., 2002).

2.3.6 Topologias de interconexão

A seguir apresentam-se conceitos relacionados à forma como os diferentes elementos de processamento são interconectados em uma arquitetura paralela (topologia). Dá-se foco ao nível físico de interconexão, visto que o nível lógico é abordado mais apropriadamente nas seções sobre modelos e linguagens de programação (ver Seções 2.3.2.3 e 2.3.3.4). Além disso, restringe-se o escopo às tecnologias de interconexão utilizadas em multiprocessadores, que são mais relevantes para possível aplicação na ARQPON dado o seu caráter de implementação na forma de multiprocessador.

A forma mais simples de interconexão de multiprocessadores no nível físico são os barramentos compartilhados. Estes barramentos transportam pacotes de dados entre os núcleos de processamento e também para outros dispositivos que estejam interligados ao barramento, em particular os dispositivos de memória. Os pacotes de dados são enviados pelos diferentes núcleos de processamento de acordo com uma política de arbitragem que evita colisões e consumidos pelo elemento de destino (controlador de memória ou outro núcleo de processamento).

A Figura 27 ilustra três topologias utilizando barramentos. A topologia ilustrada em (a) mostra duas CPUs e a memória compartilhada conectadas ao barramento; já a topologia ilustrada em (b) apresenta a alternativa na qual cada CPU possui um espaço de memória *cache* privativo para si, como nível de memória superior ao da memória compartilhada. Finalmente, a topologia ilustrada em (c) apresenta uma arquitetura na qual cada CPU tem acesso a uma porção de memória RAM privada, a qual é tipicamente utilizada para armazenar código e dados que não são tipicamente compartilhados com as demais CPUs, reduzindo assim o tráfego no barramento (TANENBAUM, 2007).

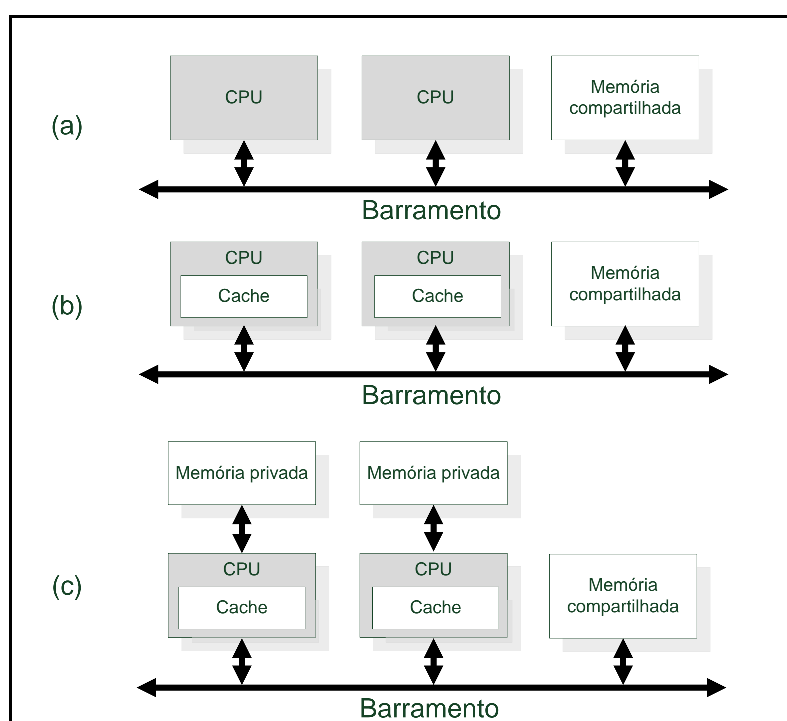


Figura 27 – Exemplos de topologias utilizando barramento compartilhado para interconexão (Fonte: adaptado de TANENBAUM, 2007, p. 349)

A principal desvantagem da construção arquitetural utilizando barramentos reside na dificuldade de escalabilidade, ou seja, tende a se tornar menos eficiente com o aumento do número de núcleos de processamento. Isto ocorre principalmente em função das limitações de banda impostas por contenções devidas a conflitos e à necessidade de arbitragem para utilização do barramento (TANENBAUM, 2007) (PATTERSON; HENNESSY, 2009) (FREITAS; ALVES; NAVAUX, 2009), aliadas à degradação do desempenho elétrico em função de capacitâncias parasitas que são adicionadas para cada núcleo que é conectado ao barramento (BJERREGAARD, 2006).

Exemplos de implementações de barramentos são a arquitetura AMBA (*Advanced Microcontroller Bus Architecture*) para ARM e a arquitetura Avalon da Altera. O AMBA é uma especificação de interconexão entre periféricos, tanto dentro de um mesmo *chip* quanto de dentro para fora, que define uma arquitetura no estilo mestre-escravo. Um mecanismo de arbitragem central é capaz de selecionar/priorizar um determinado mestre, o qual se comunica com o escravo de interesse por meio de ativação via multiplexadores. A especificação do AMBA define três subtipos de barramento: AHB (para interconexão com módulos internos de alto desempenho e alta frequência de *clock*), ASB (para interconexão com módulos internos com menor requisito de desempenho do que o AHB) e APB (para interconexão com periféricos de baixa potência) (ARM, 1999) (KALTE et al., 2002).

O barramento Avalon, por sua vez, é tipicamente utilizado como topologia de interconexão em dispositivos FPGA Altera, podendo ser manipulado e configurado por meio do *software SOPC Builder* para interconexão de dispositivos que possuam sinalização compatível com seu padrão. Este barramento opera em uma organização mestre-escravo e define tipos de interface distintos para dispositivos com endereçamento mapeado em memória, dispositivos externos interligados via barramento *tristate*, sinais de *clock,* sinais de interrupção e interfaces diretas de comunicação de dados (*streaming*) (ALTERA, 2011) (PETERS, 2012).

Uma alternativa para os barramentos é a utilização de topologias tais como *crossbar* ou *omega,* conforme apresentado na Figura 28. Nestas topologias a interconexão é efetuada por meio de comutadores (*switches*) distribuídos segundo algum arranjo espacial (no caso dos *crossbar*) ou em uma abordagem multi-estágios (no caso da *omega*), o que permite que cada núcleo P_i de processamento seja potencialmente interligado diretamente a cada outro núcleo P_j (arranjo totalmente conectado). Ainda assim, tais topologias tendem a se tornar excessivamente complexas para uma quantidade crescente de núcleos de processamento. De fato, uma topologia *crossbar* necessita de N^2 comutadores para interligar N núcleos, ao passo que uma topologia *omega* tende a introduzir atrasos cada vez maiores no tráfego de mensagens em função do aumento do número de estágios de chaveamento causado pelo aumento do número de núcleos de processamento.

Alguns exemplos de implementações utilizando *crossbars* incluem o Sun Niagara (KONGETIRA; AINGARAN; OLUKOTUN, 2005), a arquitetura de fluxo de dados COLT (BITTNER; ATHANAS; MUSGROVE, 1996 apud FERLIN et al., 2011) e o *Tile Processor* (WENTZLAFF et al., 2007).

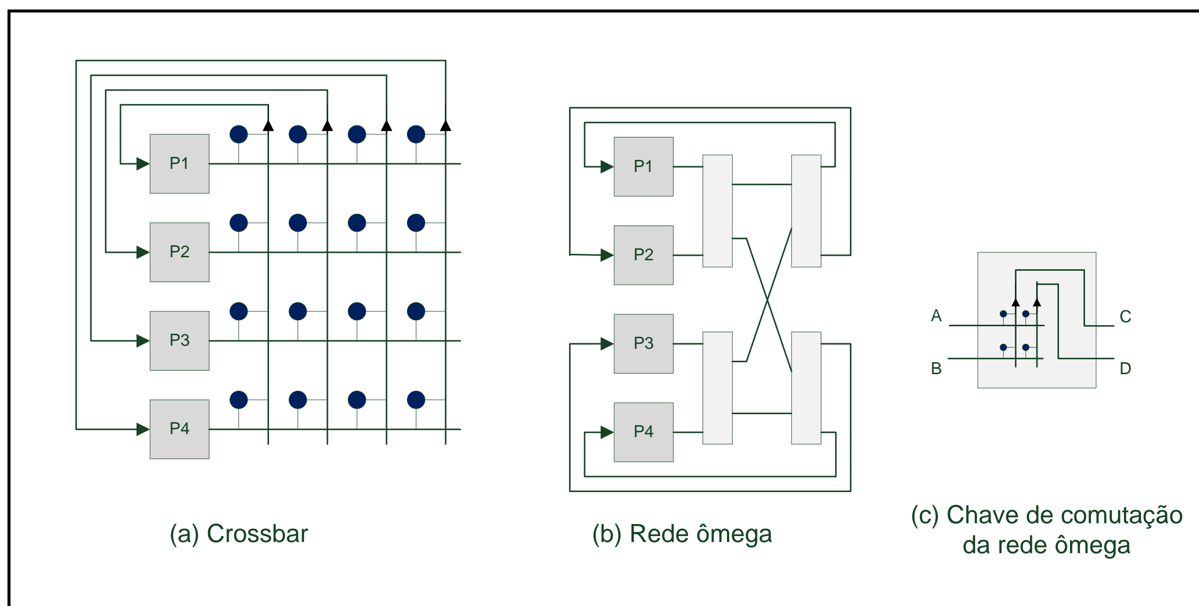


Figura 28 – Exemplos de *crossbar* e rede ômega para um arranjo de 4 núcleos (Fonte: adaptado de PATTERSON; HENNESSY, 2009 , p. 663)

Outra alternativa para implementação de interconexão são as chamadas *Redes no chip* (NoCs, ou *Networks on Chip*). Estas se fundamentam na disponibilização de uma interface de rede para cada núcleo de processamento, internamente ao *chip*, o que permite que este núcleo seja interconectado aos demais núcleos (e, eventualmente, a outros controladores heterogêneos, tais como controladores de memória) por intermédio de circuitos roteadores. A Figura 29 ilustra um exemplo de organização de *array* de 16 núcleos de processamento interligados por meio de uma NoC, indicando-se os componentes fundamentais da sua implementação.

Freitas, Alves e Navaux (2009) apresentam um estudo sobre algumas características de NoCs já implementadas, do ponto de vista de arquitetura, topologia de interconexão e categorização dos protocolos utilizados para comunicação. Além disso, apresentam uma análise comparativa de uso de NoCs em relação a barramentos e *crossbars*, adaptada de um trabalho anterior (BJERREGAARD, 2006 *apud* FREITAS; ALVES; NAVAUX, 2009). Esta análise aponta que as NoCs agregam complexidade ao projeto de *hardware*, dado que necessitam de circuitos mais elaborados, porém permitem desempenho superior de interconexão em relação a barramentos e *crossbars* à medida que o número de nós interconectados aumenta.

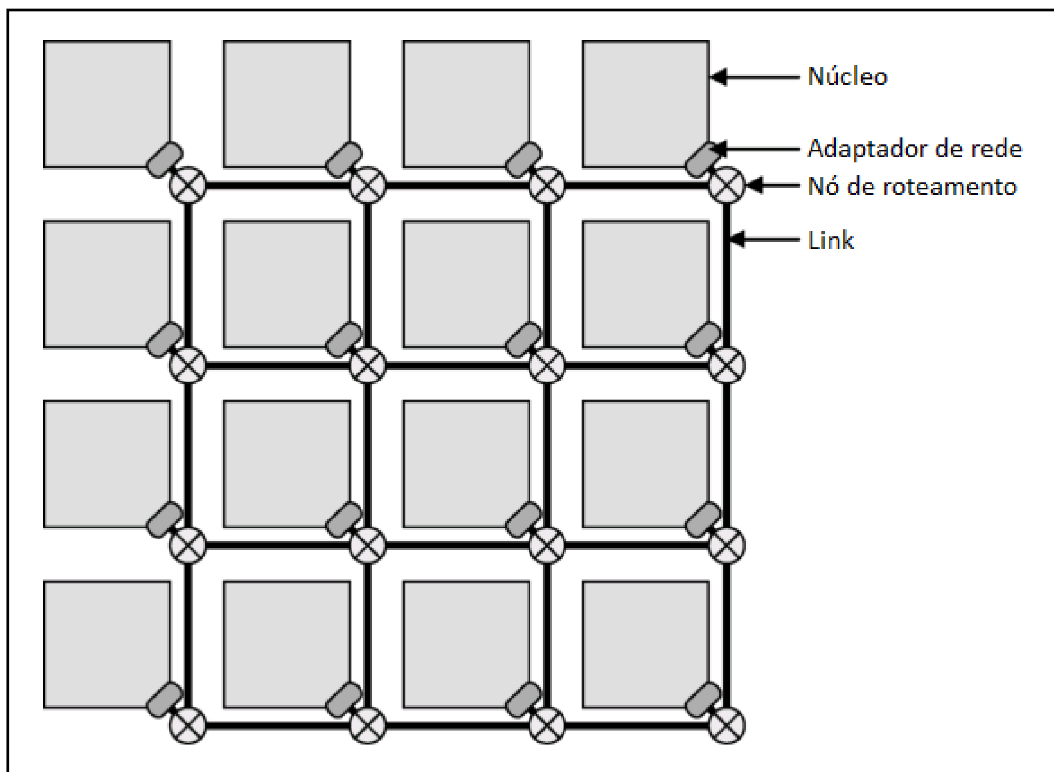


Figura 29 – Esquema geral de uma NoC para 16 núcleos estruturada em grade
(Fonte: adaptado de BJERREGAARD, 2006, p. 8)

Um exemplo de arquitetura que utiliza NoC é o IBM Cell, que implementa a interconexão entre os elementos de processamento (PPE e SPEs) por meio de uma rede em anel (ver Seção 2.3.2.2.3) (ASANOVIC et al., 2006).

2.3.7 Arquitetura do conjunto de instruções

A arquitetura do conjunto de instruções (ISA, ou *Instruction Set Architecture*) define a interface entre o *hardware* de um processador e o *software* de mais baixo nível que pode ser executado por aquele processador. Ou seja, a ISA representa uma abstração das funcionalidades que o *hardware* pode oferecer ao *software* (operações aritméticas, acesso a I/O, acesso à memória, etc.), independente da forma como o *hardware* implementa aquelas funcionalidades (PATTERSON; HENNESSY, 2009).

As seções a seguir apresentam considerações sobre as ISAs de máquinas baseadas no modelo de von Neumann e de máquinas baseadas no modelo de fluxo de dados. Para cada modelo categoriza-se as instruções em grandes grupos, de tal maneira que esta informação

possa ser utilizada para uma análise crítica da ISA mais adequada a ser projetada para a ARQPON.

2.3.7.1 Categorização de instruções no modelo de von Neumann

Em função da sua ISA, as arquiteturas de computadores baseadas no modelo de von Neumann são geralmente classificadas em dois grandes grupos, conforme descrito a seguir.

1) CISC (*Complex Instruction Set Computer*, ou Computador com Conjunto Complexo de Instruções)

As arquiteturas classificadas como CISC possuem ISAs com as seguintes características:

- Conjunto extenso de instruções, com diversas variações
- Instruções que executam operações complexas ou múltiplas operações;
- Instruções que implementam acessos múltiplos à memória (p. ex. leitura e atualização na mesma instrução);
- Múltiplos modos de endereçamento, permitindo que uma posição de memória seja acessada diretamente, por meio de referência em registrador, por meio de combinações de referências em registradores, etc.;
- Instruções com tamanho (número de *bits*) variável, com operandos anexados em seguida.
- Número reduzido de registradores.

O número reduzido de registradores e a possibilidade de se elaborar *software* com menor número de instruções mais complexas tem relação, respectivamente, com a necessidade de economia de recursos de *hardware* nos processadores e com a relativamente baixa capacidade das memórias. Estas são características das primeiras gerações de processadores, nas quais o modelo CISC foi amplamente empregado.

Além disso, o modelo CISC apresenta instruções com relativo alto grau de abstração (quando comparadas a RISC), o que facilitou a construção dos primeiros compiladores. Estas instruções são implementadas pelo processador na forma de microcódigo, composto por

microinstruções que são armazenadas em um pequeno bloco de memória, dentro do próprio processador, e interpretadas à medida que precisam ser processadas (PATTERSON, 1985).

Um exemplo de CISC é a arquitetura Intel x86. Esta arquitetura foi proposta inicialmente em 1978, tendo como principal característica ser baseada em um conjunto de registradores de propósito específico de 16 bits cada. O x86 original sofreu diversas alterações e tem evoluído até então, principalmente, com a adição de novas instruções à ISA (p. ex., instruções de extensão SSE, que viabilizam uma forma de SIMD) e a evolução de elementos arquiteturais tais como a quantidade e a largura de *bit* do conjunto de registradores (PATTERSON; HENNESSY, 2009).

A ISA mais recente para os processadores da família x86, que engloba as arquiteturas Intel 64 e IA-32, define em torno de 800 instruções. Este conjunto engloba desde os formatos básicos, de propósito geral, até as extensões tais como MMX (*Multi Media Extensions*), SSE1 a SSE4 (*Streaming SIMD Extensions*), AVX (*Advanced Vector Extensions*), VMX (*Virtual Machine Extensions*) e SMX (*Safer Mode Extensions*) (INTEL, 2012).

2) **RISC (*Reduced Instruction Set Computer*, ou Computador com Conjunto Reduzido de Instruções).**

À medida que o aumento da escala de integração viabilizou a construção de processadores e de memórias cada vez mais complexos e com maior capacidade, percebeu-se que alguns dos argumentos a favor do modelo CISC foram se tornando obsoletos. Além disso, a evolução tecnológica dos processadores possibilitou a construção de compiladores cada vez mais complexos, e, neste sentido, delegar cada vez mais decisões sobre o funcionamento do *software* para a etapa de compilação (p. ex., modo de armazenamento a ser utilizado para operações aritméticas) tornou-se mais viável e ampliou as possibilidades de se gerar *software* mais otimizado, não dependendo de uma ISA complexa para tanto.

Neste contexto optou-se então por transferir cada vez mais a complexidade (e, consequentemente, abstração) do modelo de execução para o *software*, implementando-se *hardware* que oferecesse uma ISA mais simples, porém aproveitando melhor os recursos arquiteturais e de armazenamento mais abundantes à disposição. Esta tendência deu origem ao modelo RISC (PATTERSON, 1985).

As arquiteturas classificadas como RISC possuem ISAs com as seguintes características:

- Conjunto reduzido de instruções, com algumas variações.

- Instruções que executam operações simples.
- Implementação do modelo *load/store* de acesso à memória, ou seja, uma determinada instrução pode ler dados da memória e armazená-los em registradores ou escrever na memória dados que estão armazenados em registradores, mas nunca as duas operações na mesma instrução.
- Modos de endereçamento somente indireto (por ponteiro armazenado em registrador), portanto de acordo com o modelo *load/store*.
- Instruções de tamanho (número de *bits*) fixo, com valores dos operandos representados nos *bits* da instrução.
- Grande número de registradores, principalmente de propósito geral.
- A simplificação das instruções permite, idealmente, que cada instrução demore um único ciclo de *clock* para executar.

Um exemplo de RISC é a arquitetura ARM7TDMI. Esta arquitetura implementa uma ISA composta por dois subconjuntos distintos, de instruções de 32 bits (denominado ARM) e de instruções de 16 bits (denominado THUMB). O acesso à memória segue o modelo *load/store* e utiliza os mesmos barramentos para acessar tanto dados quanto instruções. A ISA da arquitetura ARM7TDMI é formada por 70 instruções, sendo 34 do subconjunto ARM e 36 do subconjunto THUMB (ARM, 1999).

As arquiteturas baseadas em ISAs CISC ou RISC implementam instruções que diferem, do ponto de vista funcional, basicamente na complexidade (número de operações que executam individualmente) e nos diferentes modos de acesso à memória. Portanto, é possível categorizar estas instruções sem entrar no mérito do tipo de ISA específico ser RISC ou CISC ou se existem instruções no formato VLIW.

Tomando-se como referência a ISA do modo ARM da arquitetura ARM7TDMI (ARM, 1999), pode-se categorizar as instruções destes tipos de arquitetura nos seguintes grupos:

- Operações lógicas e aritméticas: ADD, AND, MUL, entre outras.
- Operações relacionais e de nível de bit: BIC, CMP, TST, entre outras.
- Operações de acesso à memória (modelo *load/store*): LDR, STR, entre outras.
- Operações de movimentação entre registradores: MOV, MSR, entre outras.
- Operações de desvio (*branch*): B, BL, BX.

- Operações de interação com dispositivos externos: MRC (*Move From Coprocessor Register to CPU Register)*, entre outras.

A arquitetura ARM7TDMI também define *flags* para execução condicional de determinadas instruções. No entanto, a execução condicional não determina uma categoria específica de instruções porque não apresenta diferenciação funcional em relação às instruções comuns, apenas permitindo que uma instrução comum seja efetivamente executada ou não.

A ISA das arquiteturas Intel 64 e IA-32, por sua vez, implementa também algumas instruções para controle do funcionamento de recursos específicos da CPU, tais como instruções para invalidação de memória *cache* (INVD, WBINVD)*,* instruções para pausa temporária de processadores lógicos (ou seja, de pausa na execução de *threads* específicas em *multithreading*) (HLT) e instruções para (re)sincronização de execução especulativa ou fora de ordem (LFENCE). Vale ressaltar, no entanto, que estas instruções são específicas para atender a determinadas questões arquiteturais (nos exemplos citados, presença de memória *cache* e de *pipeline* especulativo *multithreaded*) e, portanto, não implementam operações básicas do ponto de vista de computação no modelo von Neumann conforme as instruções anteriormente citadas para a arquitetura ARM7TDMI.

2.3.7.2 Categorização de instruções no modelo de fluxo de dados

Em uma arquitetura de fluxo de dados, a ISA é projetada partindo-se da premissa de que a dinâmica do programa consiste na propagação de *tokens* entre elementos de processamento. Portanto, em uma arquitetura puramente de fluxo de dados não se aplicam questões relacionadas ao modelo *load/store*, tais como abordadas em arquiteturas CISC ou RISC do modelo de von Neumann, visto que não existe o conceito de dados mutáveis (variáveis) que exigisse uma flexibilização do conjunto de instruções para levar em consideração diferentes estratégias de acesso à memória. Mesmo em arquiteturas que definem modelos explícitos de armazenamento e sincronização, tais como ETS e *I-Structures,* estes aspectos não são contemplados explicitamente nas ISAs correspondentes.

Dennis e Misunas (1974) propuseram, em sua arquitetura preliminar para um processador de fluxo de dados, que os *tokens* fossem categorizados em *tokens* de controle e *tokens* de dados. Os *tokens* de controle tem por objetivo aplicar regras de controle ao fluxo

dos *tokens* de dados, que por sua vez correspondem aos dados do fluxo convencional do modelo. Para a manipulação destes *tokens*, os autores definiram o seguinte conjunto de operações:

- *Operator* – define uma operação aritmética, que opera sobre 2 *tokens* de dados como argumentos e gera um *token* de dado como resultado.
- *Decider* – define uma operação relacional que opera sobre 2 *tokens* de dados como argumentos e gera um *token* de controle (*true* ou *false*) como resultado.
- *T-Gate* – define uma operação com 2 *tokens* como argumento, na qual 1 *token* de dados de entrada é propagado para a saída caso o outro *token*, de controle, tenha o valor *true.* Do contrário, o *token* de dado de entrada é absorvido.
- *F-Gate* – similar ao *T-Gate*, porém com a diferença de que o *token* de dado somente é propagado se o *token* de controle tiver o valor *false*, e absorvido caso contrário.
- *Merge* – define uma operação que recebe 2 *tokens* de dados e 1 *token* de controle como entrada. Dependendo do valor do *token* de controle, *true* ou *false,* o *token* de dado da direita ou da esquerda é propagado para a saída e o outro *token* é absorvido.
- *Boolean* – define uma operação lógica sobre 2 *tokens* de controle como argumentos, gerando 1 *token* de controle como resultado.

As operações definidas acima são mapeadas para 3 tipos de instruções, correspondentes a um *Operator,* um *Decider* e um *Boolean.* As operações de *gate* e *merge* são especificadas como opções nos códigos (*opcodes*) das instruções.

Como exemplo de implementação do modelo preliminar, a arquitetura reconfigurável de fluxo de dados PRADA (FERLIN et al., 2011) define um conjunto de 16 instruções básicas, que podem ser categorizadas nos seguintes grupos:

- Aritméticas, lógicas e relacionais.
- Condicional, que equivale a encaminhar um *token* para uma de duas saídas A e B, dependendo do valor de um *token* de controle. Esta implementação equivale a unir um *T-Gate* e um *F-Gate* da arquitetura preliminar de Dennis e Misunas (1974), ambos com os mesmos *tokens* de dado e de controle como entrada.
- Duplicação de *tokens.*
- Parada.

A arquitetura WaveScalar (SWANSON et al., 2003) propõe, em adição a instruções semelhantes às da arquitetura preliminar de fluxo de dados, instruções que permitem a execução ordenada de operações de memória (ou seja, segundo o modelo *load/store*). Portanto, a ISA desta arquitetura propõe algumas instruções de acordo com o modelo de von Neumann, de forma híbrida.

A arquitetura EDGE (BURGER et al., 2004) apresenta em sua ISA instruções que implementam comunicação direta do seu resultado (dado de saída) para outras instruções, que consomem este resultado (dado de entrada), sem passar por armazenamento intermediário em memória ou registradores. Assim, as instruções que recebem o resultado são escalonadas para execução segundo o modelo de fluxo de dados, ou seja, tão logo o resultado seja disponibilizado. Os *opcodes* das instruções EDGE, portanto, não fazem referência a registradores ou posições de memória como destino dos resultados, mas sim a outras instruções que vão consumir estes resultados.

2.4 Implementação de *hardware* em lógica reconfigurável

Segundo Compton e Hauck (2002) *apud* Ferlin (2008), existem duas formas básicas para se viabilizar a execução de um algoritmo por um circuito eletrônico. A primeira forma faz uso de um microprocessador, que neste contexto é um *hardware* com características genéricas capaz de realizar eletronicamente a lógica do algoritmo por meio da execução de um conjunto de instruções (ISA). A execução das instruções causa transições nos estados internos dos circuitos do microprocessador, geralmente em função de dados fornecidos como entrada, e gera como resultado os dados processados de saída. A segunda forma faz uso de circuitos integrados de aplicação específica (ASIC, ou *Application Specific Integrated Circuit*). Os ASICs apresentam uma funcionalidade fixa, na forma de um circuito eletrônico dedicado, que é definida pelo usuário e que não pode ser alterada após a sua fabricação.

A finalidade de um ASIC é implementar uma funcionalidade específica para determinada aplicação, geralmente com desempenho superior e menor consumo de energia do que uma implementação genérica em *software*. Os custos de produção de um ASIC, no entanto, são superiores por depender de um processo de fabricação próprio, embora estes custos possam ser diluídos e, eventualmente, se tornar atraentes para produção industrial em escala (PETERS, 2012).

A forma baseada em microprocessador é mais flexível do que a baseada em ASIC, no sentido de que é mais simples alterar a sequência de instruções a ser executada pelo processador (*software*) do que o circuito implementado no ASIC (*hardware*). No entanto, o caráter generalista do microprocessador resulta em mais *overhead,* no sentido de que este funciona baseado uma máquina de estados razoavelmente complexa para identificação, decodificação e execução de cada uma das instruções que compõem a lógica do algoritmo, o que pode causar impacto no desempenho de execução.

Como alternativa, a computação em *hardware* reconfigurável tem por objetivo aliar o melhor das duas abordagens. Ou seja, pretende oferecer flexibilidade de alteração e prototipação geralmente obtidos por meio de *software*, dado que o *hardware* é (re)programável, aliada a uma adaptabilidade a uma aplicação específica, permitindo o uso de uma quantidade menor de componentes e os consequentes ganhos em consumo de energia, desempenho e área física de circuito (PETERS, 2012).

Neste aspecto, um sistema que faça uso de *hardware* reconfigurável pode assumir diferentes arquiteturas ou configurações, dependendo da função do dispositivo de *hardware* reconfigurável dentro do sistema e do seu eventual acoplamento com um processador principal (FERLIN, 2008). A Figura 30 mostra dois exemplos a este respeito.

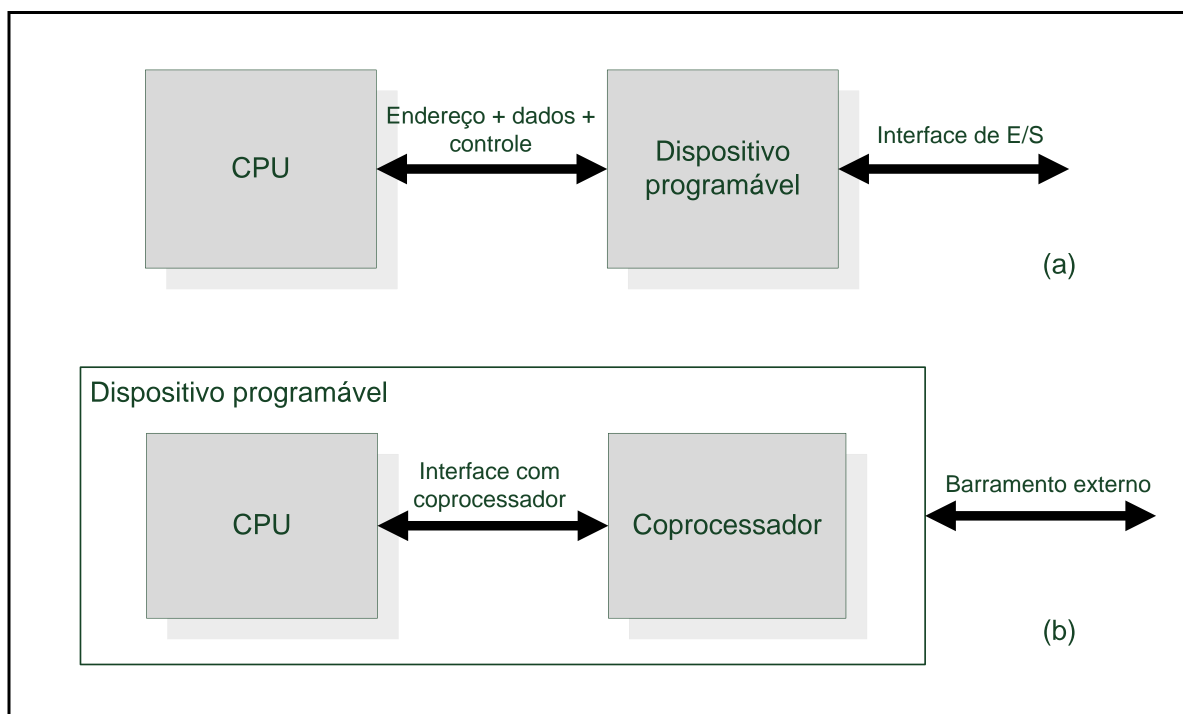


Figura 30 – Arquiteturas para utilização de dispositivo de lógica reconfigurável em um sistema (Fonte: autoria própria)

No exemplo (a) da Figura 30, um dispositivo de lógica reconfigurável é utilizado como unidade de processamento externa ao processador principal (p. ex. para implementação da sinalização de um protocolo específico da interface de E/S mostrada na figura). Já no exemplo (b) da Figura 30, um dispositivo de lógica reconfigurável é utilizado como processador principal de um sistema, dado que implementa o núcleo central de processamento (CPU) e um coprocessador, que é uma unidade funcional colaborativa que interfaceia com o núcleo central de processamento por meio de instruções específicas enviadas e sincronizadas por ele. Esta situação corresponde à arquitetura proposta por Peters (2012) para o coprocessador PON, que é implementado em lógica reconfigurável juntamente com um núcleo Altera NiOS II que desempenha o papel de um microprocessador von Neumann genérico.

2.4.1 Categorias de dispositivos de lógica programável e modelos de programação

Atualmente existe à disposição uma gama de dispositivos que podem ser considerados de lógica programável, ou seja, nos quais o *hardware* é diretamente programado / configurado para executar determinadas funções.

Tais dispositivos, conhecidos genericamente por PLDs (*Programmable Logic Devices*), foram inicialmente disponibilizados na forma de *arrays* de portas lógicas para a implementação de circuitos digitais combinacionais (PAL – *Programmable Array Logic* e PLA – *Programmable Logic Array*). Posteriormente, agregou-se a estes dispositivos elementos de lógica sequencial, tais como flip-flops, além de componentes mais sofisticados de lógica combinacional tais como multiplexadores (GAL – *Generic Array Logic* ou, de forma mais abrangente, SPLD – *Simple Programmable Logic Device*).

O próximo passo, em termos tecnológicos, foi aprimorar a programabilidade em campo dos dispositivos programáveis, ou seja, disponibilizar interfaces que facilitassem a sua programação *in circuit.* Surgiram então as CPLDs (*Complex PLDs*), as quais fazem uso de interface JTAG (padronizada pelo *Joint Action Test Group*) para a sua programação, geralmente com maior densidade, melhor desempenho e agregando uma maior gama de sinais de E/S.

Em seguida surgiram as FPGAs (*Field Programmable Gate Arrays*), que se constituem em um aprimoramento da tecnologia de CPLDs visando aplicações de maior

complexidade e melhor desempenho. A diferença fundamental entre CPLDs e FPGAs reside no fato de que a configuração (arranjo dos elementos lógicos), nas CPLDs, é não volátil, ou seja, implementada em memórias EEPROM, ao passo que nas FPGAs é volátil, implementada em memórias SRAM. Isso requer que as FPGAs contenham uma memória não volátil extra, embutida no *chip*, que é utilizada para a programação (volátil) da configuração quando da inicialização do dispositivo (PEDRONI, 2010).

Internamente, as FPGAs são compostas por blocos lógicos, utilizados para a configuração das funções lógicas, e as interconexões entre estes blocos. Os blocos lógicos, por sua vez, podem ser baseados em LUTs (*lookup tables*), as quais são compostas por células de armazenamento multiplexáveis que permitem implementar um mapeamento entre os valores binários de um conjunto de entradas e o valor binário de saída da função correspondente; este mapeamento e os valores binários de entrada, bem como o estado das chaves binárias de interconexão, se constituem então nos elementos configuráveis da FPGA para a implementação das funções lógicas.

Sob outro viés, dispositivos como FPGAs podem ser reconfigurados tanto estaticamente (antes do início da operação do *hardware* onde se inserem) quanto dinamicamente (durante a operação do *hardware* onde se inserem).

A reconfiguração estática ainda pode ser subdividida em (estritamente) estática e semi-estática. A principal diferença entre as duas abordagens é que, na reconfiguração estática, o dispositivo de *hardware* reconfigurável assume uma única configuração (programação) desde o início e não é mais reconfigurado ao longo da vida útil do sistema, ao passo que na reconfiguração semi-estática o dispositivo pode assumir uma diferente configuração a cada nova operação do *hardware* onde se insere (FERLIN, 2008).

Em relação à reconfiguração dinâmica, a sua principal característica é permitir a adaptação às demandas de processamento de uma aplicação, parcialmente implementada em *hardware,* à medida que estas demandas surgem durante a operação; por exemplo, a reconfiguração para implementação de um algoritmo de descompressão de imagem somente enquanto esta funcionalidade for necessária, liberando posteriormente o *hardware* da FPGA (elementos lógicos) para ser reconfigurado para outras funcionalidades. Todavia, Kalte et al. (2002) citam que, embora dispositivos mais recentes possam ser dinamicamente reconfigurados até mesmo parcialmente, os tempos de reconfiguração são relativamente altos e podem se constituir em um custo proibitivo.

2.4.2 Metodologias para desenvolvimento de *hardware* em lógica reconfigurável

O desenvolvimento de um sistema que possua parte do seu *hardware* implementado em lógica reconfigurável depende, inicialmente, de uma etapa na qual será definido claramente o seu particionamento. Ou seja, nesta etapa define-se que partes da funcionalidade do sistema (algoritmos) serão executadas no *hardware* reconfigurável e que partes serão executadas externamente ao *hardware* reconfigurável (possivelmente em *software* executando em um microprocessador ao qual ele esteja conectado). Este particionamento é uma das etapas do método de *codesign* aplicado ao desenvolvimento do *hardware* e do *software* componentes de um sistema (WOLF, 2003).

Uma vez efetuado o particionamento e definida a porção a ser implementada em *hardware* reconfigurável, na forma de um circuito eletrônico, é necessário definir uma metodologia de projeto deste circuito. Como, na prática, não é viável efetuar este desenvolvimento no nível dos *bits* que realizam a programação dos blocos lógicos reconfiguráveis, faz-se necessário algum tipo de abstração de mais alto nível do *hardware* reconfigurável (FERLIN, 2008). A Figura 31 apresenta uma visão do processo de desenvolvimento de um circuito em *hardware* reconfigurável.

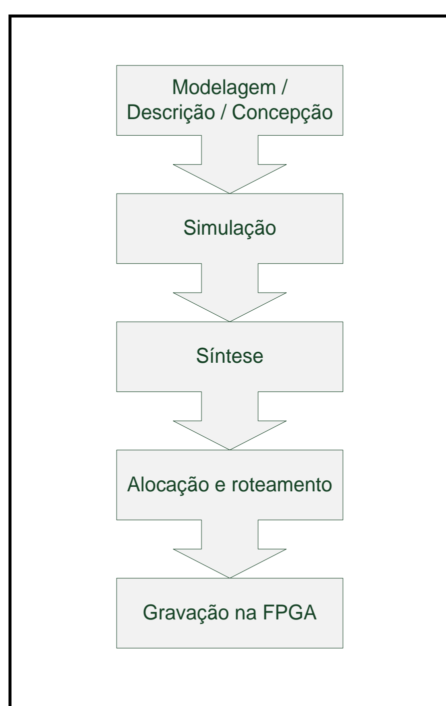


Figura 31 – Etapas do processo de desenvolvimento em *hardware* reconfigurável

(Fonte: adaptado de PETERS, 2012, p. 47)

Atualmente existem ferramentas que permitem a concepção do circuito de *hardware* a ser configurado na forma de diagrama esquemático. Este método depende de um desenho de circuito no qual se explicita os componentes discretos que o compõem (portas lógicas, *buffers,* etc., ou até componentes de mais alto nível como comparadores) e as suas interconexões, portanto exigindo um conhecimento razoável de projeto de *hardware* por parte do desenvolvedor. Segundo Ferlin (2008), esta abordagem tende a se tornar inadequada para projetos maiores justamente pela dificuldade em se gerenciar a representação gráfica de um grande número de componentes.

Outro tipo de abstração envolve o uso de linguagens específicas de descrição de *hardware,* tais como VHDL (*Very-High-Speed-Integrated-Circuits Hardware Description Language)* (PEDRONI, 2010), a qual permite que o desenvolvimento siga uma abordagem estrutural ou comportamental. Na abordagem estrutural o *hardware* é concebido com um foco na estrutura, ou seja, na construção do circuito em si, por meio da descrição dos componentes discretos de *hardware* e de suas interconexões de forma similar à concepção utilizando-se diagrama esquemático. Também em função desta característica, a abordagem estrutural exige-se do desenvolvedor um conhecimento mais aprofundado de projeto de *hardware.*

Na abordagem comportamental, por sua vez, opta-se por modelar a dinâmica desejada de funcionamento do *hardware* (p. ex. por meio da modelagem de máquinas de estados). Neste caso, uma ferramenta que viabilize a descrição VHDL comportamental deve gerar automaticamente o circuito que implementa aquela dinâmica, no entanto possivelmente gerando um resultado menos otimizado do que o resultado que seria gerado por um desenvolvedor experiente utilizando a abordagem estrutural.

Existe ainda a possibilidade de se desenvolver o *hardware* reconfigurável em linguagem de programação de alto nível, tal como C ou C++, ou utilizando-se de ferramentas de modelagem matemática tais como *Matlab*. Utilizando esta técnica, o desenvolvedor descreve o comportamento do sistema como um todo em linguagem de alto nível, cabendo então a uma ferramenta (compilador) efetuar o particionamento (orientado por diretivas de compilação), implementando parte do sistema em *software* e parte em *hardware* reconfigurável (FERLIN, 2008).

Finalmente, a modelagem pode ser efetuada fazendo-se uso de técnicas tais como a de Redes de Petri. Esta técnica permite a representação gráfica da dinâmica de um sistema em função de determinados estímulos ou entradas, evidenciando potenciais fluxos de execução paralelos. Antiqueira (2011) desenvolveu um método para implementação de Redes de Petri

em lógica reconfigurável, permitindo uma verificação formal e simulação deste modelo antes da sua implementação efetiva, bem como a transformação da Rede de Petri para uma representação do circuito lógico em linguagem VHDL.

Independente da técnica ou abordagem utilizada, em seguida o circuito descrito pode ser submetido a simulações, conforme mostra a próxima etapa na Figura 31. As simulações permitem avaliar previamente o seu comportamento, evitando que eventuais problemas ou falhas sejam detectados durante o teste no dispositivo de *hardware,* o que pode até mesmo causar mau funcionamento permanente no dispositivo ou em outros componentes a ele interligados. Uma vez que as simulações sejam bem sucedidas e as falhas tenham sido corrigidas, a próxima etapa do processo de desenvolvimento envolve a síntese do circuito em *hardware* a partir da sua descrição, que consiste em traduzir esta descrição para um conjunto de blocos lógicos e suas interligações (PETERS, 2012).

Em seguida a estas etapas ocorrem a alocação e o roteamento, que consistem em posicionar os blocos lógicos necessários dentro do dispositivo de lógica reconfigurável e rotear as suas interligações. A última etapa do processo envolve então a gravação do dispositivo reconfigurável, permitindo que este seja operado conforme os parâmetros de funcionamento do circuito nele programado.

Tanto a simulação quanto a síntese e as etapas posteriores são efetuadas por ferramentas de *software* adequadas. Exemplos de ferramentas ou ambientes disponíveis para esta finalidade no mercado incluem Altera Quartus II e Xilinx ISE, ambas permitindo simulação e síntese, dentre outras que oferecem todas ou parte das funcionalidades integradas (PEDRONI, 2010).

2.5 Critérios de *benchmarking* e diretrizes para a comparação de desempenho e avaliação de implementação do PON perante outras arquiteturas

Segundo Ferlin (2008), diversas métricas são comumente utilizadas para se avaliar desempenho em arquiteturas de computadores, tanto sequenciais quanto paralelas. Tais métricas podem ser comparativas, quando permitem que o resultado da avaliação seja comparado com os resultados de avaliações de outras implementações ou arquiteturas. Esta comparação visa principalmente embasar uma discussão a respeito de vantagens ou

desvantagens que a solução sendo avaliada apresenta em relação às soluções com as quais é comparado, do ponto de vista de desempenho. Para tanto, a avaliação comparativa de desempenho pode fazer uso de métricas relativas, tais como:

- Ganho de desempenho (*speedup*), que é a razão entre o desempenho de duas implementações sendo comparadas.
- Eficiência do paralelismo, que diz respeito ao ganho de desempenho obtido em função do número de elementos de processamento paralelos adicionados a uma determinada implementação de arquitetura.

Um requisito importante para execução de uma análise comparativa é que ela seja executada sobre as mesmas configurações de *hardware* e *software* para todas as arquiteturas a serem comparadas, com exceção do elemento que se pretende avaliar (p. ex. processador) (FERLIN, 2008). Do contrário, pode ocorrer uma distorção na comparação ou na percepção da ordem de grandeza do critério sendo avaliado.

O desempenho de um sistema computacional é geralmente definido como função da velocidade de processamento em determinado tipo de aplicação, ou seja, o número de operações efetuadas por esta aplicação por unidade de tempo. Tais operações podem ser categorizadas como cálculos de ponto flutuante, de onde surge o termo FLOPS (*Floating-Point Operations Per Second*), ou mesmo instruções de forma genérica, de onde surge o termo MIPS (*Million Instructions Per Second*). Estas métricas, em particular, viabilizam avaliações quantitativas, que oferecem uma noção de ordem de grandeza no que diz respeito à capacidade de computação.

As avaliações comparativas de sistemas computacionais, e de certa forma também as avaliações quantitativas, dependem de aplicações ditas de *benchmarking*. Estas são aplicações bem definidas, projetadas para executar operações que exercitem intensivamente o aspecto ou tipo de computação que se pretende avaliar. Exemplos são as *suites* SPEC2006 (STANDARD..., 2006), EEMBC (EMBEDDED..., 2012), LINPACK (PETITET et al., 2008) / LAPACK (LAPACK, 2012), SPLASH 2 (WOO et al., 1995), ANSYS Fluent (ANSYS, 2012) e alguns *benchmarks* de escopo específico tais como STREAM (avaliação de largura de banda de memória usando vetores de tamanho superior à capacidade da memória *cache*).

Um exemplo de avaliação comparativa é o relatório publicado por Dongarra (2011). Este relatório apresenta o desempenho de vários processadores na execução de aplicações específicas de cálculo de sistemas de equações, entre elas as da *suite* LINPACK. Os resultados são apresentados utilizando-se MFLOPS como unidade e permitem tanto uma

avaliação quantitativa (p. ex. para se determinar o processador de melhor desempenho para determinada aplicação) quanto comparativa.

Muitos das aplicações de *benchmark* anteriormente apresentadas podem ser utilizadas também no domínio de arquiteturas paralelas. No entanto, conforme citado por Patterson e Hennessy (2009), a forma de se viabilizar a execução paralela das aplicações, para se obter um resultado útil para comparação, pode diferir. Aplicações como as da *suite* LINPACK são fracamente escaláveis, o que significa que dependem que o tamanho do problema a ser resolvido aumente proporcionalmente ao número de elementos de processamento paralelos. Outras aplicações, p. ex. SPLASH 2, são fortemente escaláveis, o que significa que o tamanho do problema pode ser mantido mesmo com o aumento do número de elementos de processamento disponíveis (portanto, alocando uma porção menor da execução da aplicação para cada elemento de processamento).

Independente da escalabilidade, outra questão que se apresenta no *benchmarking* de arquiteturas paralelas é o quanto cada aplicação pode ser adaptada para uma diferente proposição de arquitetura ou até mesmo de tecnologia de compilação, dado que adaptações radicais podem invalidar uma análise comparativa (um hipotético ganho de desempenho poderia ocorrer em função de um melhoramento do *software* e não do *hardware* em si) (PATTERSON; HENNESSY, 2009). Neste sentido, Asanovic et al. (2006) propuseram um conjunto de problemas de *benchmark* em um nível de abstração mais alto (apelidados de “anões”). Os problemas propostos incluem cálculo de álgebra linear densa, matrizes esparsas, percurso de grafos e máquinas de estado finitas e são abordados como métodos computacionais, ou seja, independentes de uma implementação específica de algoritmo. O objetivo foi dar margem à utilização de novos *frameworks* ou até mesmo novos paradigmas de computação em *hardware,* permitindo que o sistema sendo avaliado implemente a solução para o problema de *benchmark* da forma mais adequada à sua tecnologia.

No domínio das arquiteturas de fluxo de dados, diversas medidas foram obtidas como parte do *benchmarking* efetuado por Gurd, Kirkham e Watson (1985) para o computador de fluxo de dados Manchester. As métricas utilizadas incluem MIPS, FLOPS, *speedup* e eficiência de paralelização (dado o paralelismo intrínseco que pode ser realizado em uma arquitetura de fluxo de dados).

Ferlin (2008) cita ainda que podem surgir potenciais dificuldades na utilização de algumas das métricas de avaliação de desempenho. Por exemplo, a avaliação comparativa utilizando MIPS ou FLOPS como métrica é altamente influenciada pelo modelo do conjunto de instruções das arquiteturas sendo comparadas, principalmente se for uma comparação entre

uma implementação RISC e uma CISC. De fato, uma implementação CISC tende a utilizar muito menos instruções do que uma implementação RISC para executar as mesmas tarefas, ainda que as instruções CISC tipicamente demorem mais ciclos de *clock* para executar, portanto uma avaliação baseada em número de instruções por unidade de tempo pode ser distorcida.

2.5.1 Considerações sobre avaliações comparativas no contexto do PON

Do ponto de vista do PON, dado que um programa é organizado na forma de regras que são habilitadas mediante um mecanismo de notificações, faz-se necessário aplicações de *benchmark* que levem em consideração esta característica. Em particular, aplicações que sejam baseadas na execução sequencial de um grande conjunto de operações matemáticas (tais como as da *suite* LINPACK) somente são úteis para este propósito se puderem ser (re)compostas na forma de regras. Isto pode ser inviável justamente por este tipo de aplicação não apresentar um conjunto de relações lógico-causais que justifique a sua modelagem em regras.

Alguns pesquisadores efetuaram avaliações de *benchmark* utilizando aplicações concebidas desde o princípio para serem baseadas em regras. Estas avaliações tiveram como objetivo comparar implementações no domínio dos SBR, com o foco em questões relativas ao desempenho dos mecanismos e modelos de inferência e execução das regras.

Lee e Cheng (2002) efetuaram comparações do mecanismo de inferência HAL, no domínio dos SBR, com outros mecanismos de inferência (RETE, TREAT, etc.) utilizando duas aplicações de *benchmark*, apelidadas de “Miss Manners” e “Waltz”. A primeira aplicação é uma simulação de alocação de convidados de um jantar para as respectivas mesas, respeitando regras relativas ao sexo (alocação de casais) e afinidade de *hobbies* de cada uma das pessoas, baseada na execução de um conjunto de 8 regras. A segunda aplicação, por sua vez, é um programa especializado em analisar linhas de um desenho bidimensional e extrapolar estas linhas como arestas de um objeto tridimensional, baseado na execução de um conjunto de 33 regras.

A métrica quantitativa utilizada em ambos os experimentos realizados por Lee e Cheng foi o número de operações executadas sobre elementos da memória de trabalho, que é definido como o somatório da quantidade de escritas e leituras efetuadas. Nesta definição, os

autores consideraram que operações de apagamento ou modificação são implementadas por meio de uma operação de leitura seguida de uma escrita.

Especificamente no escopo do PON, Banaszewski (2009) efetuou estudos comparativos utilizando uma aplicação no estilo *toy problem* denominada “Mira Alvo”. Esta aplicação consiste em um jogo no qual entidades que fazem o papel de mira (os “arqueiros”) interagem com entidades que fazem o papel de alvo (as “maçãs”) segundo determinadas regras. As regras consistem basicamente em assinalar que determinado arqueiro flechou uma determinada maçã quando as seguintes premissas forem atendidas: a maçã estiver pronta, for de determinada cor e tiver um identificador compatível com o do arqueiro correspondente (Figura 32 (a)). Outro cenário proposto incluiu uma entidade que faz o papel de elemento sincronizador (uma “arma de fogo”), cujo disparo permite que os arqueiros interajam com as respectivas maçãs (Figura 32 (b)).

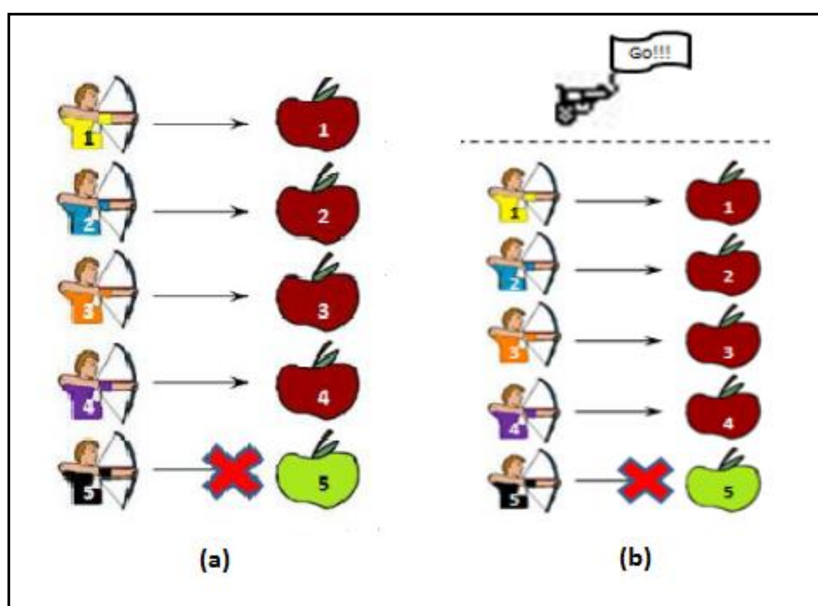


Figura 32 – Cenários de simulação do problema “mira-alvo”
(Fonte: adaptado de BANASZEWSKI, 2009, p. 151 e 163)

Ambos os cenários foram então implementados no PON (*framework* PON C++), no PI (código em linguagem C++) e no PD (*shell* CLIPS), utilizando-se técnicas e ferramentas apropriadas. Estes cenários foram exercitados exaustivamente com a substituição das maçãs (atingidas ou não) por outras, de maneira que determinado percentual de expressões causais fosse satisfeito.

Os resultados foram submetidos a análise comparativa, variando-se o percentual de expressões causais satisfeitas e utilizando-se como métrica quantitativa o tempo de execução

de cada uma das implementações. Com este procedimento, objetivou-se avaliar o impacto da ocorrência de redundâncias e do aumento do número de notificações no que diz respeito ao desempenho de cada uma das versões implementadas em cada um dos paradigmas.

Peters (2012), por sua vez, também executou avaliações comparativas de desempenho da aplicação mira alvo quando executada no processador NiOS II da Altera, utilizando o *framework* C++ do PON, e quando executada com o auxílio do co-processador PON que foi objeto da sua pesquisa. A métrica de desempenho utilizada para comparação também foi o tempo de execução, medido em ciclos de *clock*.

Outras aplicações de *benchmark* também utilizadas por Banaszewski e/ou Peters para avaliações comparativas do PON, nos respectivos contextos de suas pesquisas, são listadas a seguir:

- Sistema de condicionamento de ar de edifício, baseado em exemplo apresentado por Friedman-Hill (2003) *apud* Banaszewski (2009) e que consiste em um controle centralizado de regras de acionamento de bombas de calor e entradas de ar em função dos valores de temperaturas fornecidos por termômetros.
- Sistema de portão eletrônico. Este sistema foi inicialmente proposto por Wiecheteck (2011), como caso de estudo para a sua pesquisa relativa ao método DON (*Desenvolvimento Orientado a Notificações*), e consiste em um conjunto de regras utilizadas para controle de abertura e fechamento de um portão eletrônico em função da atuação em um controle remoto.
- Simulador de semáforo, proposto por Ronszcka (2011) e que consiste em um conjunto de regras que simulam o comportamento de veículos e pedestres em um sistema de tráfego, em função da variação do estado de um conjunto de semáforos de trânsito existentes no sistema.

Todas as aplicações baseadas em regras apresentadas nesta seção podem ser adaptadas ou até mesmo estendidas para utilização em avaliações tanto quantitativas como comparativas da ARQPON, sem requerer alterações radicais na sua lógica dado que já foram concebidas baseadas em regras.

Além disso, dada a granularidade de cada uma destas aplicações, na forma de múltiplas instâncias de FBEs e das regras que sobre eles atuam, pode-se considerar que elas são fortemente escaláveis. Ou seja, com o aumento do número de núcleos de processamento, pode-se alocar um número menor de regras para execução por cada núcleo, mantendo o número de regras total da aplicação.

Do ponto de vista de métricas quantitativas para a avaliação, algumas considerações relevantes são elaboradas e apresentadas nas subseções a seguir. Estas considerações fazem uso da seguinte terminologia:

- Abordagem / plataforma PON: conjunto das camadas de *software* e/ou *hardware* que implementa as abstrações do metamodelo do PON, viabilizando a execução de aplicações segundo aquele paradigma. As plataformas consideradas neste estudo são o *framework* C++ PON, a ARQPON, o co-processador PON e PON reconfigurável em *hardware*.
- Implementação de aplicação PON: uma determinada implementação de um sistema (*hardware* e/ou *software*) segundo o PON, definida pelo conjunto de instâncias dos elementos do metamodelo de notificações que interagem para a dinâmica do sistema.
- Implementação de aplicação em outro paradigma: uma determinada implementação de um sistema segundo outro paradigma de programação.

1) Utilização do tempo de execução como métrica quantitativa para comparação

O tempo de execução de um conjunto de regras, como parâmetro para cálculo do ganho de desempenho (*speedup*), tem sido utilizado como métrica quantitativa nos trabalhos que efetuam comparações entre diferentes implementações PON ou entre implementações PON e em outros paradigmas. Contudo, vale ressaltar que algumas dificuldades se apresentam na interpretação dos dados para uma análise comparativa.

Por exemplo, Banaszewski (2009) cita o fato de que as avaliações do *framework* PON, no contexto da sua dissertação, poderiam ter sido efetuadas sobre o sistema ANALYTICE II, dado que este já apresenta parte da sua estrutura lógica concebida em PON. Porém, isto não ocorreu porque partes do código do ANALYTICE II ainda são concebidas segundo o PI, o que poderia causar interferência e dificultar o mapeamento das redundâncias que se pretendia abordar.

Embora as aplicações propostas e citadas na subseção anterior sejam mais simples e especificamente implementadas para facilitar a avaliação comparativa do PON, dois problemas ainda persistem. O primeiro diz respeito ao fato de que, conforme já exposto, até então nenhuma das aplicações PON foi executada 100% de acordo com o modelo do PON; de fato, embora Peters (2012) tenha implementado uma abordagem na qual a dinâmica da cadeia de notificações é fiel ao metamodelo, isto não ocorre na execução dos métodos, pois estes são implementados segundo o PI. Em relação às versões implementadas com o *framework* C++

isso é ainda mais evidente, dadas as questões que dizem respeito à sua implementação com estruturas de dados.

O segundo problema diz respeito à concepção das aplicações PI em si, para as avaliações nas quais o PON é comparado com aquele paradigma. Conforme observado por Banaszewski, a escolha entre uma implementação PI o mais simplificada possível, com acesso direto aos atributos de um FBE, e uma implementação PI com o auxílio de estruturas de dados tais como *vector* da STL (*Standard Template Library*) do C++ impacta sobremaneira no desempenho. Embora a implementação do PON sobre *framework* C++ também sofra do mesmo impacto, este é um ponto que dificulta uma análise sobre a eficiência do PON em relação ao PI no que tange ao desempenho.

Aliado a este fato, algumas da aplicações de *benchmark* (p. ex. mira alvo) foram concebidas para ressaltar redundâncias temporais e estruturais. Embora Friedman-Hill (2003) e Forgy (1984), *apud* Banaszewski (2003), estimem que entre 1% e 20% das avaliações causais em programas sejam desnecessárias devido a redundâncias, algumas aplicações reais sofrem muito menos deste problema do que outras. Tome-se como exemplo aplicações orientadas a eventos gerados por interrupções de *hardware*, que são muito comuns no domínio dos sistemas embarcados, as quais somente executam determinados trechos de código quando necessário, em função de um estímulo externo que atua diretamente sobre o *hardware* do processador para desviar a execução sem necessitar de avaliações lógico-causais extra (e potencialmente redundantes) para esta verificação.

A avaliação quantitativa do PON efetuada por Linhares et al. (2011) no contexto de um simulador de sistema telefônico ilustra bem este fato, dado que aquele sistema apresenta características de orientação a eventos gerados por interrupções. Nesta avaliação, a versão implementada segundo o PON foi até 250 vezes mais lenta do que a versão PI, justamente por não haver redundâncias significativas e por a versão PON sofrer do excessivo *overhead* imposto pela implementação do *framework* PON em C++.

Avaliações comparativas entre diferentes abordagens / plataformas PON, por sua vez, tem se mostrado mais úteis, principalmente no sentido de permitir avaliar qual a viabilidade prática de implementação do modelo do PON e se as vantagens teóricas relacionadas ao modelo podem, de fato, ser obtidas. Neste sentido, as avaliações efetuadas com as implementações do PON em *hardware* apresentadas por Witt et al. (2011) e Peters (2012), quando comparadas à implementação utilizando o *framework* C++, permitem uma reflexão mais consistente do ponto de vista quantitativo no que tange aos valores de tempo de execução, demonstrando o impacto do *overhead* imposto pelo *framework* C++ executando em

plataforma sequencial. Anteriormente, esta reflexão já havia sido efetuada por Banaszewski (2009) quando da comparação quantitativa entre diferentes estágios de evolução do *framework* C++, as quais foram complementadas em função de evoluções posteriores do *framework* efetuadas por Valença (2012).

2) Utilização da quantidade de relações lógico-causais avaliadas por quantidade de regras executadas como métrica quantitativa para comparação

A quantidade de relações lógico-causais avaliadas por quantidade de regras executadas tende a ser uma métrica mais adequada do que avaliação de tempo de execução para comparação entre diferentes implementações de uma mesma aplicação PON. Isto se deve ao fato que esta métrica sofre menos interferência de questões relacionadas a diferenças no desempenho e na arquitetura da plataforma de execução, na hipótese em que as implementações fossem executadas em plataformas distintas. Além disso, quanto maior é o número de avaliações lógico-causais necessárias para execução de uma regra maior é a ocorrência de redundâncias, portanto esta métrica oferece um parâmetro simples para avaliação da implementação no que tange a minimização daquelas redundâncias.

De forma contrária, esta métrica não se aplicaria para comparações da mesma implementação em plataformas diferentes. Isto ocorre porque, em tese, as mesmas relações lógico-causais deveriam ser avaliadas, na mesma ordem, independentemente da plataforma/ambiente de materialização do PON onde se está executando o programa, desde que utilizado o mesmo critério de resolução de conflitos e desde que seja garantido o determinismo da execução.

Esta métrica também pode ser aplicada para a comparação de implementações PON com implementações em outros paradigmas. Porém, neste caso deve-se ter cuidado em definir claramente o que se entende por regra executada, já que não necessariamente este conceito é claro e bem definido em outros paradigmas.

No caso de comparações do PON com o PD, em particular um SBR, o conceito de regra é claro e, portanto, esta métrica de avaliação é capaz de avaliar comparativamente o mecanismo de inferência do PON *versus* o mecanismo/algoritmo de inferência que se utilizou para execução do SBR (HAL, RETE, etc.).

Já em uma comparação com uma aplicação PI, uma alternativa seria mapear determinadas funções/procedimentos da aplicação PI em uma sequência bem definida de regras do PON. Desta forma, o número de avaliações lógico-causais para execução da

sequência de regras PON seria comparado com o número de avaliações lógico-causais executadas na função PI escolhida.

A métrica utilizada por Lee e Cheng (2002), no âmbito da avaliação do HAL, apresenta semelhanças com esta métrica no sentido de que permite avaliar o impacto das redundâncias no número de operações de leitura de memória executadas (que são operações realizadas naturalmente na avaliação de relações lógico-causais).

3) Utilização do grau de paralelização como métrica quantitativa para comparação

O grau de paralelização é uma métrica interessante para se comparar diferentes plataformas de execução PON, dado que expõe com que fidelidade cada plataforma implementa a dinâmica de funcionamento da cadeia de notificações do PON. Define-se neste contexto o grau de paralelização como sendo a quantidade de possíveis ciclos de notificação simultâneos, ou seja, quantas regras podem potencialmente ser aprovadas e executadas em paralelo.

O grau de paralelização é dependente do funcionamento do ambiente no qual se aplica a distribuição / paralelização de elementos da cadeia de notificações. Por exemplo, no caso do *framework* C++ monoprocessado esta distribuição é somente temporal, ou seja, cada elemento da cadeia de notificações deve ser executado (i. e. propagar notificação se necessário) sequencialmente aos demais elementos. Logo, o grau de paralelização do ambiente utilizando o *framework* C++ monoprocessado é 1.

Uma versão do *framework* PON C++ executada em ambiente multiprocessado exibiria um grau de paralelização igual ao número de elementos processadores. Embora tal implementação não tenha sido efetuada, um antecessor deste *framework* foi implementado na forma *multithread* para o CON, no contexto do sistema ANALYTICE II (WEBER et al., 2010). Nesta implementação, cada componente do CON é alocado para execução em uma *thread* específica, que poderia ser executada em paralelo com as demais desde que houvesse recursos de processamento para tanto e não houvesse interdependências entre as *threads*.

A implementação do co-processador PON efetuada por Peters (2012) apresenta um grau de paralelização potencial dependente da capacidade do dispositivo de FPGA utilizado para a sua configuração (disponibilidade física de unidades lógicas e memória), o mesmo ocorrendo na abordagem proposta por Witt et al. (2011). O grau de paralelização efetivamente implementado depende, em ambos os casos, da aplicação sendo desenvolvida, dado que a sua lógica influencia na possibilidade de execução efetivamente paralela das regras.

4) Utilização da eficiência de paralelização como métrica quantitativa para comparação

Conforme a Lei de Amdahl (ver Seção 2.1), a eficiência de paralelização é naturalmente limitada pela porção do programa que pode, de fato, ser executada em paralelo. Esta porção é influenciada pela natureza da aplicação e também pela capacidade da plataforma computacional em efetivar a execução paralela.

No contexto específico do PON, a eficiência pode ser utilizada como métrica para a avaliação da escalabilidade de uma solução que implemente paralelismo, seja baseada na aplicação diretamente configurada em *hardware* conforme Witt et al., seja baseada no co-processador PON conforme proposto por Peters ou seja baseada na ARQPON. A justificativa é que a eficiência tende a decrescer com o aumento da escala de paralelismo (número de elementos de processamento) disponível, dado que uma determinada aplicação PON é capaz de explorar paralelismo até certo limite. Este limite teórico corresponde ao número total de regras definidas, porém na prática é inferior a este valor visto que existem regras interdependentes.

A razão de decréscimo da eficiência em função do aumento da escala é, portanto, um parâmetro que permite uma análise comparativa entre diferentes implementações de plataforma PON que ofereçam paralelismo. Em particular, no contexto da ARQPON, esta métrica pode ser útil, nas fases de prototipação e experimentação, para comparação entre diferentes alternativas propostas para esta arquitetura.

Do ponto de vista de comparação entre implementações, a eficiência de paralelização pode ser utilizada como indicador da qualidade e da viabilidade da adaptação para o PON de determinada aplicação desenvolvida em outro paradigma. Ou seja, caso o objetivo da adaptação seja melhorar o desempenho de execução, explorando o paralelismo teórico intrínseco do PON, esta adaptação pode ser considerada melhor sucedida quanto maior for a eficiência obtida pela execução paralela do ciclo de notificações. Naturalmente, esta análise faz sentido para plataformas que viabilizem o paralelismo, excluindo, portanto, as implementações atualmente baseadas no *framework* PON C++ em ambiente monoprocessado.

A Tabela 2 sumariza os aspectos discutidos em relação às métricas de *benchmark.*

Tabela 2 – Avaliação da utilização de métricas comparativas do PON em função do escopo de comparação

Métrica	Escopo da comparação		
	Plataforma PON X Plataforma PON	Implementação PON X Implementação PON	Implementação PON X Impl. outro paradigma
Tempo de execução	Relevante como forma de se avaliar o grau de sucesso na implementação de conceitos do PON pelas plataformas.	Relevância depende do grau de orientação a notificações *versus* código sequencial.	Relevância depende do grau de orientação a notificações da implementação PON, útil para avaliação do impacto de redundâncias.
Relações lógico-causais / regra	Irrelevante (comportamento lógico no nível de regra deveria ser independente de implementação da plataforma).	Relevante na avaliação do impacto de redundâncias.	Relevante caso as relações lógico-causais e regras sejam mapeáveis em seus equivalentes na implementação em outro paradigma.
Grau de paralelização	Relevante como forma de se avaliar o grau de sucesso na implementação de conceitos do PON pelas plataformas.	Irrelevante.	Irrelevante.
Eficiência de paralelização	Relevante para avaliação de escalabilidade.	Relevância depende do grau de orientação a notificações *versus* código sequencial.	Relevante para se avaliar a eficiência da adaptação da aplicação em outro paradigma para a filosofia do PON.

2.6 Considerações sobre o capítulo

Este capítulo apresentou uma série de conceitos teóricos e tecnológicos relacionados com o objetivo desta pesquisa de doutorado. O objetivo deste estudo foi fazer um

levantamento atualizado das pesquisas e propostas existentes na literatura que servirão de referencial para as propostas apresentadas.

Inicialmente apresentou-se os conceitos básicos do Paradigma Orientado a Notificações (PON), bem como comparações entre estes conceitos e seus equivalentes no Paradigma Imperativo (PI) e no Paradigma Declarativo (PD). Além disso, apresentou-se uma reflexão sobre quão adequadas são as características do PON para o desenvolvimento de *software* que apresente paralelismo ou necessidade de distribuição. Este conjunto de fundamentos teóricos deve ser totalmente contemplado na proposição da ARQPON ou, pelo menos, a parte não contemplada deve ser levada em consideração nas decisões sobre a arquitetura de tal maneira que a ARQPON possa ser evoluída para contemplá-la em desenvolvimentos futuros.

Os conceitos relativos a arquiteturas de computadores, também apresentados neste capítulo, embasam a proposição da ARQPON na medida em que esta pode aproveitar-se de blocos arquiteturais e técnicas já exploradas em outras arquiteturas, acrescidas de um eventual aprimoramento para adaptação aos conceitos do PON. Neste contexto, enfatizou-se o estudo de características de arquiteturas paralelas, abordando-se os diversos níveis de paralelização possíveis (ILP, DLP, TLP) e suas técnicas de implementação.

Optou-se por aprofundar dois modelos distintos de concepção de arquiteturas paralelas: o modelo de von Neumann e o modelo de fluxo de dados (*dataflow*). O modelo de von Neumann, embora tenha sido originalmente concebido como fundamento para a execução sequencial de programas, foi estudado por ser também a base conceitual da grande maioria dos esforços envolvidos na evolução da computação paralela, incluindo as principais técnicas de implementação de paralelismo atualmente utilizadas. O modelo de fluxo de dados, por sua vez, foi abordado porque o seu mecanismo de execução apresenta similaridades com o conceito de propagação de notificações do PON, portanto, soluções arquiteturais derivadas das máquinas baseadas em fluxo de dados podem ser analisadas e, eventualmente, aplicadas na concepção da ARQPON.

A fundamentação teórica englobou também alguns conceitos relativos à utilização de *hardware* programável como plataforma de prototipação para a ARQPON, em particular o uso de lógica reconfigurável na forma de dispositivos de FPGA. Neste contexto, enfatizou-se questões relativas às técnicas de desenvolvimento para FPGA e como se pretende utilizar estas técnicas para a implementação dos protótipos da ARQPON (P2ON).

Finalmente, apresentou-se algumas considerações a respeito da realização de análises de *benchmarking*, tanto comparativas como qualitativas ou quantitativas, com foco em

arquiteturas paralelas e posteriormente na reflexão sobre como realizar *benchmarking* em aplicações PON. Estas considerações serão utilizadas como base para a concepção das aplicações de *benchmark* que serão utilizadas para avaliação da implementação da ARQPON, no contexto deste trabalho de pesquisa.

Feita esta revisão, pode-se traçar um panorama da evolução da computação paralela, iniciando-se com a proposição do modelo de von Neumann, o qual foi fundamental para viabilizar a computação moderna no sentido de permitir que *software* fosse desenvolvido em maior escala e de maneira mais flexível. O desempenho de arquiteturas baseadas neste modelo é primariamente influenciado pela frequência do *clock* da CPU, cujo aumento acompanhou a evolução das tecnologias de construção eletrônica. Adicionalmente, técnicas arquiteturais foram progressivamente desenvolvidas para otimização do tempo de execução de instruções individuais ou de grupos de instruções, o que, em conjunto com o aumento da frequência do *clock,* viabilizou a evolução do desempenho conforme a Lei de Moore. Entre estas técnicas, destacam-se as que objetivam a obtenção de paralelismo de execução nos diversos níveis, tanto entre instruções individuais (ILP) quanto entre grupos de instruções logicamente relacionadas (TLP).

No entanto, alcançou-se um estágio tecnológico no qual a evolução do desempenho de computação baseada unicamente na evolução do desempenho dos circuitos eletrônicos, particularmente em plataformas baseadas em silício, está desacelerando. Como a escala de integração continua aumentando no mesmo ritmo, passou-se a optar por soluções arquiteturais que aproveitassem a densidade de componentes disponível, em particular soluções nas quais os núcleos de processamento fossem replicados (*multi core*). Em consequência deste viés, é cada vez mais necessário conceber *software* que possa aproveitar do paralelismo oferecido pelo *hardware* de múltiplos núcleos.

Considerando que a exploração do paralelismo disponível requer técnicas adequadas de desenvolvimento de *software*, iniciativas tais como o modelo de fluxo de dados objetivaram, ao seu tempo, apresentar inovações que permitissem a paralelização de maneira distinta às formas baseadas no modelo de von Neumann. Estas inovações ocorreram em um cenário de interdependência entre *software* e *hardware*, ou seja, o modelo de programação de fluxo de dados e suas características dinâmicas e de semântica impulsionaram o desenvolvimento de técnicas de *hardware* compatíveis, tais como o uso de *I-Structures.*

Neste âmbito, o PON foi proposto como um paradigma de programação e de execução que apresenta conceitos e abstrações que o caracterizam como uma abordagem de

concepção de *software* com paralelismo intrínseco. Considerando-se a interdependência que deve haver entre *hardware* e *software*, o *software* desenvolvido segundo o PON depende de um ambiente de execução adequado para a exploração plena do paradigma, principalmente levando-se em consideração as peculiaridades do seu modelo de notificações. Este modelo é inovador, porém, realizável de maneira limitada por modelos tais como von Neumann, que é fundamentado em execução sequencial regida unicamente por um contador de programa. Ainda que o modelo de fluxo de dados apresente similaridades com o PON, do ponto de vista dinâmico, as aplicações de fluxo de dados são mais restritas a cálculos aritméticos na forma funcional, diferentemente do modelo de notificações do PON que pode ser aplicado, com as adaptações necessárias, aos problemas solucionáveis com aplicações comuns segundo o PI.

Portanto, a fundamentação teórica apresentada e a reflexão sobre cada um destes fundamentos sugere uma oportunidade de desenvolvimento de pesquisa sobre novas arquiteturas de computação que possam suprir as demandas específicas do PON. Tais arquiteturas, embora possam fazer uso de parte da tecnologia já desenvolvida para resolver questões arquiteturais pontuais, devem propor um modelo lógico próprio e distinto, tanto do ponto de vista estrutural quanto comportamental. Como consequência, tais arquiteturas podem contribuir para o estado da arte no que tange ao problema geral de paralelização da computação, tanto pelo modelo lógico inovador a ser proposto, em nível de *hardware,* quanto pela contribuição no que diz respeito a enriquecer a discussão sobre quão promissor é o PON como técnica de programação paralela.

O Capítulo 3 a seguir apresenta a proposta de tese, incluindo o método de pesquisa a ser utilizado para a proposição da ARQPON e realização dos demais objetivos específicos.

3 Apresentação da proposta de tese

Este capítulo apresenta um detalhamento da proposta de tese em desenvolvimento. As subseções a seguir abordam a proposta de tese, a sua relevância e originalidade como um trabalho de doutoramento e o método de pesquisa utilizado para o seu desenvolvimento. Complementarmente relata-se o estado das atividades já desenvolvidas e ainda por desenvolver, bem como se apresenta um cronograma relacionado a estas atividades.

3.1 Descrição da proposta de tese

Este trabalho de doutorado se propõe a investigar e definir um modelo de arquitetura de processador para programas concebidos segundo o Paradigma Orientado a Notificações (PON). Este modelo é denominado de ARQPON e o(s) processador(es) a ser(em) implementado(s) segundo esta arquitetura é(são) denominado(s) de P2ON.

Diferente de outros trabalhos já desenvolvidos, objetivando definir um ambiente de *hardware* próprio para a execução de uma aplicação específica PON (WITT et al, 2011) (PETERS, 2012), a ARQPON se propõe a ser um ambiente que permita a concretização dos princípios de execução de *software* concebido segundo o PON de forma independente de uma aplicação específica. Para tanto, a ARQPON deve ser definida baseando-se nos seguintes princípios / premissas:

- A ARQPON deve ser genérica, no sentido de permitir que o P2ON execute diferentes aplicações desenvolvidas segundo o PON sem necessitar de alterações ou reconfigurações de sua estrutura de *hardware* para cada nova aplicação a ser executada.
- A ARQPON deve mapear em *hardware*, na forma dos seus blocos constituintes, os diferentes elementos conceituais do metamodelo do PON, procurando favorecer ao máximo a execução do comportamento destes elementos de acordo com o metamodelo teórico.
- A ARQPON deve ser escalável, no sentido de permitir implementações do P2ON em graus variados de complexidade (ou seja, com quantidades diferentes dos blocos básicos constituintes da arquitetura e correspondentes interconexões), porém mantendo os princípios básicos de funcionamento da arquitetura.

Uma vez definida a ARQPON segundo as premissas citadas anteriormente, pretende-se avaliá-la qualitativa e quantitativamente. Estas avaliações permitirão uma análise crítica a respeito da viabilidade de tal modelo de execução para o PON.

3.2 Relevância da proposta de doutorado

Historicamente, o conjunto de conhecimentos relacionados à engenharia de *software* evoluiu por meio da proposição e aprimoramento de técnicas, linguagens, padrões, processos, etc., visando melhorar a produtividade e qualidade do desenvolvimento de *software*. Durante esta evolução, diversos paradigmas de desenvolvimento de *software* foram propostos, tais como o paradigma funcional, o paradigma declarativo, o paradigma imperativo e suas variações, cada um deles obtendo maior ou menor sucesso em termos de adequação aos problemas de engenharia de *software* que foram surgindo e consequente aceitação pela comunidade acadêmica e industrial.

Neste contexto, o PON é proposto como um novo paradigma, com potencial de permitir um salto em produtividade na engenharia de *software*. Isto ocorre justamente em função das suas características, como o paralelismo naturalmente implícito na concepção do *software* e facilidade de distribuição, o que inclusive encontra respaldo em uma percepção da comunidade acadêmica sobre a importância cada vez maior do desenvolvimento ou aprimoramento de técnicas e modelos de computação, possivelmente diferentes do modelo de von Neumann, que sejam mais adaptados à realidade das novas aplicações eminentemente paralelas que surgirão nos próximos anos (KOGGE et al., 2008) (CATANZARO et al., 2010).

O objeto da presente proposta de doutorado está relacionado a este contexto, visto que, conforme apresentado na Figura 1 (d), a ARQPON se insere como elemento central em um conjunto de técnicas e ferramentas que compõem um ambiente completo para concepção e execução de aplicações concebidas segundo o PON.

Até o presente momento, cada uma das soluções já propostas para o ambiente de aplicação PON sofre de alguma deficiência. Isto ocorre ou porque o referido ambiente suprime alguma etapa importante do processo de desenvolvimento (caso da Figura 1 (b)) ou porque tenta emular o comportamento dinâmico do PON em um paradigma de programação / execução distinto (caso da Figura 1 (a)) ou não puramente orientado a notificações (caso da Figura 1 (c)).

As premissas e fundamentos da ARQPON objetivam eliminar estas deficiências, oferecendo um ambiente de execução mais adaptado ao modelo de execução proposto pelo paradigma. Além disso, conforme já citado na Seção 3.1, tais características da ARQPON permitem uma avaliação crítica mais apurada da viabilidade do próprio paradigma, justamente por permitir que aplicações sejam concebidas e executadas inteiramente orientadas a notificações, sem fazer uso de camadas intermediárias que simplesmente emulam os modelos do PON.

3.3 Originalidade da proposta de doutorado

O PON é um paradigma de programação relativamente recente, tendo os seus fundamentos inicialmente estabelecidos na tese de doutorado de Simão em 2005. A partir de então, investiu-se em pesquisas com o objetivo de desenvolver técnicas, desde relacionadas à concepção e *design* de *software,* passando pela implementação e compilação em linguagem de alto nível, até a concretização em nível de execução, que viabilizassem o desenvolvimento de aplicações segundo o PON.

Inicialmente, houve um foco na implementação de *software* PON, com o desenvolvimento e posterior aprimoramento do *framework* PON C++. Isto permitiu a concretização dos fundamentos do paradigma na forma de abstrações de *software,* porém ainda gerando código executável segundo o paradigma imperativo em máquinas computacionais baseadas no modelo de von Neumann.

Tendo em vista a necessidade de um ambiente de execução mais adaptado à abordagem do PON, propôs-se posteriormente técnicas para se implementar uma aplicação PON em *hardware,* inicialmente por meio do uso de blocos básicos em FPGA correspondentes aos elementos do metamodelo (SIMÃO et al, 2012b) e mais tarde por meio de um coprocessador da cadeia de notificações (PETERS, 2012) reconfigurável para uma determinada aplicação e integrável a um núcleo von Neumann para execução das regras. Ambas as técnicas baseadas em *hardware,* no entanto, geram como resultado um *hardware* específico e configurado para a execução de uma única aplicação PON.

Comparativamente, ambientes de programação e execução convencionais, por exemplo os baseados em von Neumann para execução de *software* PI (POO), dispõem de arquiteturas de *hardware* genéricas. A flexibilidade de execução de diferentes aplicações, neste caso, é obtida por meio da alteração do *software* de aplicação que reside em memória.

Portanto, objetivando de forma semelhante um ambiente de execução que seja adaptado à abordagem do paradigma e, ao mesmo tempo, com as características de generalidade e flexibilidade descritas, conclui-se que os desenvolvimentos no âmbito do PON anteriormente listados não são suficientes. A proposta da ARQPON é, justamente, ser genérica no sentido de que um processador construído segundo a ARQPON não necessita ser reimplementado para executar uma nova aplicação PON ou uma versão alterada da aplicação correntemente sendo executada. Ao invés disso, a aplicação PON é construída e eventualmente alterada em *software*, fazendo-se uso do conjunto de declarações de dados e instruções de baixo nível propostas pela ARQPON, armazenada em memória e buscada por aquele processador para consequente execução.

Adicionalmente, a concepção da ARQPON totalmente voltada aos princípios do PON acrescenta um caráter extra de originalidade à pesquisa, no sentido de que permite expor claramente uma potencial deficiência conceitual do modelo de von Neumann, que é a não categorização ou hierarquização do conjunto básico de instruções. A ARQPON pode apresentar uma alternativa para esta deficiência provendo um nível de abstração de conjunto de instruções correspondente ao nível de abstração dos elementos do metamodelo do PON, que é relativamente mais elevado. Pretende-se avaliar as implicações desta característica e qual a sua influência na execução de uma aplicação.

3.4 Método de pesquisa adotado

O método utilizado para a realização deste projeto de pesquisa é apresentado na Figura 33. A primeira etapa consistiu na definição do tema de pesquisa. O tema escolhido foi o “Paradigma Orientado a Notificações”, delimitando-se o escopo para alguns aspectos ainda não cobertos pelas pesquisas anteriores ou que poderiam ser aprimorados, tais como: análise temporal de programas segundo o PON; estudo de técnicas de implementação de *software* PON; estudo de um compilador de código PON otimizado; estudo de técnicas de implementação de programas PON utilizando paralelismo em *hardware*.

O tema citado e os diferentes aspectos que poderiam ser abordados foram mencionados na Proposta de Tese de Doutorado inicialmente apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), servindo como base para o início do trabalho de pesquisa.

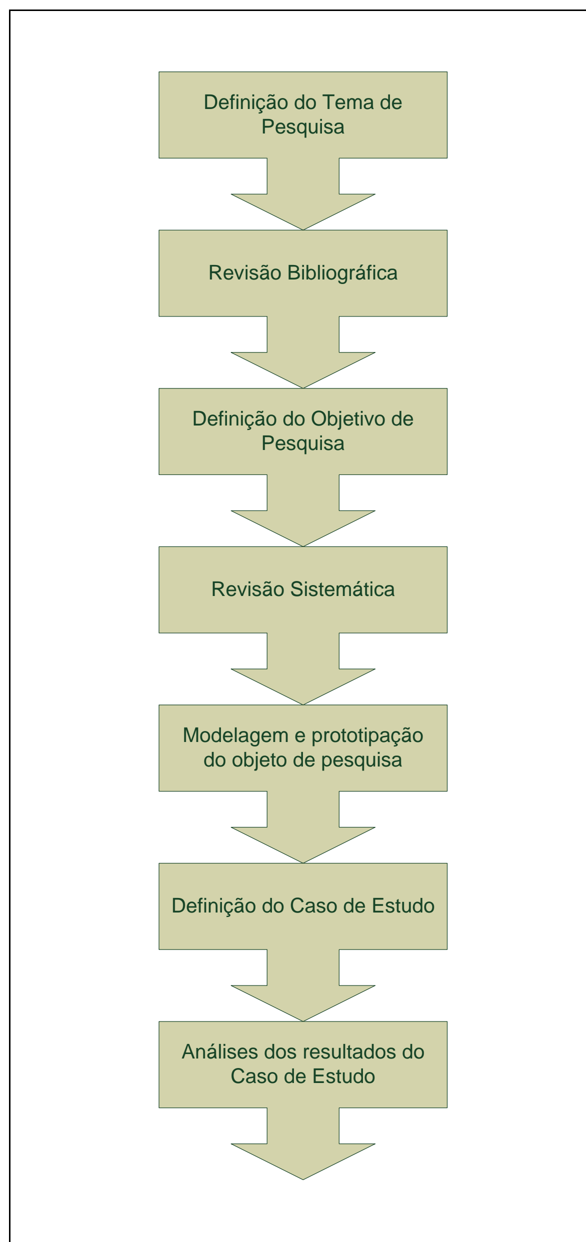


Figura 33 – Etapas do método de pesquisa adotado

(Fonte: autoria própria)

Em seguida, efetuou-se uma revisão bibliográfica focada principalmente na pesquisa já existente sobre o PON. Esta revisão foi complementada inicialmente com estudos especiais sobre o PON, durante os quais experimentos preliminares de implementação de *software* PON foram executados. Estes experimentos objetivaram adquirir um maior grau de conhecimento sobre o tema e entender melhor de que forma os diversos aspectos contemplados na proposta de tese poderiam ser ou não abordados.

Como resultado desta etapa, refinou-se o objetivo de pesquisa até que este fosse definido precisamente, resultando em um escopo reduzido no qual se decidiu por focar a

pesquisa na implementação do PON em *hardware* por meio de uma arquitetura de processador, conforme já explicitado anteriormente. Este processo de refinamento do objetivo de pesquisa, em função do conhecimento adquirido na revisão bibliográfica, é mencionado por Wazlawick (2008) como um processo possível dentro de uma metodologia de pesquisa na área de computação.

Em seguida procedeu-se com uma revisão sistemática (GREEN, 2005), com a finalidade de direcionar e aprofundar a revisão bibliográfica inicial para assuntos mais diretamente relacionados ao tema do objetivo de pesquisa. Nesta etapa estudou-se artigos e livros relacionados a arquiteturas de computadores em diferentes modelos (von Neumann, fluxo de dados e outros modelos de interesse), bem como conceitos relacionados ao desenvolvimento de *software* concorrente ou paralelo, à implementação de *hardware* em lógica reconfigurável e a critérios de *benchmarking* aplicáveis a *software* paralelo e, em particular, ao domínio do PON. A sistemática de seleção de material bibliográfico para esta pesquisa envolveu os seguintes critérios:

- Busca por palavras-chave em bases de pesquisa relevantes para as áreas de Ciência da Computação e Engenharia de Computação, tais como *IEEE Explore* e *ACM Library*.
- Busca por material bibliográfico relativamente recente (de preferência, produzido nos últimos dez anos). Neste aspecto abriu-se uma exceção para material mais antigo sobre o modelo de fluxo de dados, devido à maior disponibilidade deste material em datas anteriores aos últimos dez anos.
- Busca por material com maior número de referências, neste caso eventualmente selecionando artigos clássicos anteriores aos últimos dez anos que pudessem, entretanto, colaborar com os fundamentos dos assuntos sendo pesquisados.

A sistemática de extração, análise e interpretação dos dados e informações buscados envolveu, por sua vez, a leitura cuidadosa do material selecionado, marcação dos trechos mais importantes e posterior compilação e síntese destes trechos. A ferramenta Mendeley Desktop (MENDELEY, 2012), versão 0.9.8.2, foi usada como ambiente de suporte para a organização, seleção e marcação de texto do material selecionado.

A etapa seguinte envolve a modelagem do objeto de pesquisa. Durante esta etapa as informações coletadas na revisão sistemática, principalmente relacionadas a técnicas arquiteturais adequadas para a construção de arquiteturas paralelas e adaptáveis aos conceitos do PON, são submetidas a uma análise e selecionadas para a criação de uma nova arquitetura

denominada ARQPON. Desta atividade se obtém como resultado uma concepção para a ARQPON na forma de diagramas arquiteturais, conforme mencionado na Seção 3.5.1. Na sequência da pesquisa esta concepção será submetida a uma primeira avaliação com o objetivo de efetuar eventuais correções e melhoramentos, obtendo-se então o modelo definitivo da ARQPON para fins deste trabalho.

A definição do caso de estudo será efetuada com o objetivo de apoiar as análises qualitativas, quantitativas e comparativas da ARQPON. Estas análises serão efetuadas sobre dados experimentais obtidos por meio da execução do caso de estudo no protótipo da ARQPON em diferentes circunstâncias, comparando-se também com a sua execução em outras plataformas (ver Seção 3.5.2), consistindo, portanto, em uma etapa baseada em pesquisa experimental (WAZLAWICK, 2008).

3.5 Planejamento e cronograma das atividades

Esta seção apresenta as atividades que foram planejadas para a realização deste projeto de pesquisa, separadas em atividades já desenvolvidas e atividades ainda por desenvolver.

3.5.1 Atividades já desenvolvidas no doutorado

Esta subseção apresenta cada uma das atividades planejadas e já desenvolvidas até a defesa de qualificação deste projeto de pesquisa.

Atividade 1 – Obtenção de créditos

Esta atividade englobou a obtenção de créditos, conforme requisito estabelecido pelo Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI) para obtenção do título de “Doutor em Ciências”. A quantidade mínima de créditos requerida pelo Programa é de 48 (quarenta e oito).

Até o momento, a situação de obtenção de créditos é a seguinte:

- 23 (vinte e três) créditos obtidos como validação de créditos de curso de mestrado, realizado também no CPGEI.

- 18 (dezoito) créditos obtidos com a participação em disciplinas ofertadas pelo CPGEI e pelo PPGIA (programa conveniado da PUC-PR).
- 2 (dois) créditos obtidos com a atividade de estágio docência.

Restam, portanto, 5 (cinco) créditos a serem obtidos para o cumprimento dos requisitos. Para esta finalidade, existe um requerimento de 6 (créditos), em andamento no CPGEI, referente à publicação de 3 (três) trabalhos em anais de congresso internacional.

Atividade 2 – Levantamento Bibliográfico

Conforme estabelecido pelo método de revisão sistemática, esta atividade englobou ampla revisão da literatura disponível nos tópicos de interesse deste projeto de pesquisa: Fundamentos de Software Sequencial e Paralelo, Arquiteturas de Computadores, Critérios de *Benchmark* para Arquiteturas de Computadores e Paradigma Orientado a Notificações.

Dentro do contexto de arquiteturas de computadores, enfatizou-se o estudo comparativo entre arquiteturas von Neumann e fluxo de dados, com foco em suas vantagens e desvantagens, de forma a fundamentar a proposição da ARQPON do ponto de vista das tecnologias disponíveis.

A síntese do levantamento bibliográfico é apresentada no capítulo sobre Fundamentação Teórica (Capítulo 2).

Atividade 3 – Esboço preliminar da ARQPON

Esta atividade compreendeu duas etapas distintas:

- Levantamento das premissas / diretrizes para a ARQPON. Estas premissas foram elaboradas com base nos objetivos e requisitos da ARQPON dentro da plataforma de execução do PON, levando-se em consideração os aspectos científicos e tecnológicos de arquiteturas de computadores obtidos por meio da fundamentação teórica.
- Elaboração do diagrama em blocos preliminar para a ARQPON. Este diagrama leva em consideração as premissas levantadas anteriormente e propõe soluções arquiteturais, cujos méritos e deficiências serão avaliados na continuação da pesquisa.

Atividade 4 – Definição e implementação de aplicação para a avaliação do esboço preliminar da ARQPON

Durante esta atividade será definida uma aplicação simples, concebida segundo a abordagem do PON, cujo objetivo será permitir o levantamento de dados para uma avaliação crítica do esboço elaborado para a ARQPON. Esta aplicação deverá ser implementada segundo as premissas (conjunto de instruções, modelo de programação, etc.) da arquitetura, levando em consideração as questões apresentadas na Fundamentação Teórica referentes a métricas e escalabilidade.

Atividade 5 – Projeto e implementação de protótipo para a ARQPON

Nesta atividade será projetado e implementado um protótipo de P2ON para o esboço da ARQPON. Tal protótipo será implementado utilizando um *kit* de desenvolvimento FPGA como plataforma e priorizando uma implementação que contenha cada um dos blocos ou subsistemas constituintes da arquitetura, em escala mínima e suficiente para permitir a avaliação crítica conforme a atividade a seguir.

Atividade 6 – Elaboração do texto de qualificação

Esta atividade compreendeu a elaboração deste documento, o qual será submetido a exame de qualificação.

3.5.2 Atividades a serem desenvolvidas no doutorado

A seguir uma lista das atividades ainda a serem desenvolvidas para complementar o estudo proposto neste projeto de pesquisa.

Atividade 1 – Avaliação crítica da ARQPON

Nesta atividade deverá ser efetuada uma avaliação da solução previamente elaborada para a ARQPON, de forma a averiguar os seus méritos e deficiências do ponto de vista conceitual. Esta avaliação deverá ser efetuada utilizando-se critérios qualitativos e quantitativos, a serem selecionados durante esta atividade, e aplicados aos dados obtidos por ocasião da execução da aplicação de *benchmark* projetada durante a Atividade 1.

Com base nesta avaliação, o esboço da ARQPON será eventualmente refinado de forma a se obter o modelo de arquitetura definitivo.

Atividade 2 – Reimplementação do protótipo da ARQPON

Uma vez refinado o modelo de arquitetura, será efetuado um reprojeto e reimplementação do protótipo em maior escala. Isto objetiva aproveitar ao máximo a capacidade (quantidade de unidades lógicas e memória) oferecida pelo *kit* de desenvolvimento FPGA utilizado como plataforma e, desta forma, avaliar a escalabilidade da ARQPON.

Atividade 3 – Definição e implementação de aplicação de *benchmarking* para o protótipo da ARQPON

Nesta atividade será escolhida e implementada uma aplicação, de maior porte do que a aplicação escolhida para a Atividade 1, com o objetivo de avaliar a ARQPON segundo critérios qualitativos e quantitativos (possivelmente os mesmos selecionados durante a Atividade 3).

Atividade 4 – Realização de *benchmarking* comparativo entre o P2ON e outras plataformas

Nesta atividade os dados qualitativos e quantitativos obtidos na Atividade 5, para a versão da ARQPON escolhida, serão comparados a dados semelhantes obtidos para outras plataformas. Para tanto, a aplicação definida na Atividade 5 deverá ser portada para cada uma destas plataformas, o que envolverá adaptações a diferentes *frameworks* e paradigmas.

As plataformas a serem envolvidas no estudo comparativo são as seguintes:

- Aplicação concebida segundo o PON e implementada em *hardware* reconfigurado especificamente para materializar esta aplicação, de maneira semelhante ao trabalho efetuado por Witt et al (2011).
- Aplicação concebida segundo o PON e implementada para execução em *hardware* reconfigurado com o coprocessador desenvolvido por Peters (2012). Esta implementação envolve, portanto, a implementação de parte da aplicação (*Methods*) segundo o PI para execução em um núcleo von Neumann Altera NiOS II.
- Aplicação concebida segundo o PON e implementada em *software* utilizando a versão mais recente do *framework* PON C++ otimizada por Valença (2012).

- Aplicação concebida segundo o PI e implementada para execução em um núcleo von Neumann Altera NiOS II, configurado em FPGA com circuitos auxiliares para medição de tempo e implementação de eventuais interfaces de E/S.

Atividade 5 – Elaboração de artigos

Durante a execução das demais atividades os resultados obtidos serão formatados para a elaboração de artigos e envio para publicação.

Atividade 6 – Elaboração e defesa da tese

Esta atividade engloba documentar os resultados das demais atividades na forma de um documento de Teste de Doutorado, bem como a posterior defesa da Tese perante uma banca examinadora.

3.6 Cronograma de execução das atividades

A Figura 34 apresenta o cronograma envolvendo as etapas ainda a serem desenvolvidas deste projeto de pesquisa.

	2013						2014	
Atividade	Jul	Ago	Set	Out	Nov	Dez	Jan	Fev
1								
2								
3								
4								
5								
6								

Figura 34 - Cronograma de execução das atividades

(Fonte: autoria própria)

4 Desenvolvimento do trabalho

Neste capítulo é apresentado o desenvolvimento preliminar da arquitetura de processador para o PON (ARQPON).

Este desenvolvimento engloba a enumeração e discussão das premissas da ARQPON, conforme fundamentos do PON apresentados no Capítulo 2 e objetivos gerais e específicos da pesquisa apresentados nos Capítulos 1 e 3. Com base nas premissas enumeradas, propõe-se, então, uma alternativa preliminar para a ARQPON, com ênfase na visão de blocos arquiteturais e algum detalhamento do ponto de vista estrutural e funcional de cada bloco.

No texto das seções a seguir, referencia-se a versão preliminar específica como ARQPON v1.0. A partir desta versão preliminar pretende-se efetuar avaliações e validações para gerar outra(s) versão(ões) até a conclusão da pesquisa de doutorado. Outros aspectos genéricos ou propostos como ideias de desenvolvimento futuro são referenciados como ARQPON de maneira genérica.

4.1 Levantamento de requisitos para a ARQPON

Com base nos objetivos e motivações descritos para a elaboração da ARQPON na Seção 1.2, levantou-se a lista de requisitos apresentada a seguir.

4.1.1 Requisitos funcionais

A Tabela 3 apresenta os requisitos funcionais para concepção da ARQPON. O requisito RF-01 é consonante com a principal motivação para a concepção da ARQPON, que é viabilizar a existência de um ambiente de execução adequado para aproveitar as características potenciais de execução paralela / distribuída do PON. Já o RF-02 abre a possibilidade de construção de um processador híbrido com base na ARQPON, de forma semelhante à proposta de Peters (2012), o que vai de encontro à ideia de que nem todas as construções de *software* podem ser eficientemente implementadas unicamente segundo a filosofia do PON.

Tabela 3 – Requisitos funcionais da ARQPON

Identificador	Descrição
RF-01	A ARQPON deverá ser capaz de executar aplicação dada que seja composta pelos elementos do metamodelo de notificações do PON
RF-02	A ARQPON deverá ser capaz de executar aplicação dada que seja composta por elementos do metamodelo de notificações do PON e também por *Methods* implementados por meio de funções segundo o modelo de von Neumann.

Esta conclusão também é referendada por Kogge *et al.* (2008), segundo os quais uma dificuldade que um modelo de programação inovador deve enfrentar é a necessidade de se manter o máximo de compatibilidade retroativa com código desenvolvido segundo os modelos de programação tradicionais. Esta necessidade é justificada pelo fato de que reimplementar aplicações que já existem e estão em funcionamento envolve um investimento significativo e indesejado.

4.1.2 Requisitos de interface lógica

A Tabela 4 apresenta os requisitos de interface lógica para concepção da ARQPON.

Tabela 4 – Requisitos de interface lógica da ARQPON

Identificador	Descrição
RIL-01	A ARQPON deverá definir uma interface de programação / ISA que mapeie os elementos do metamodelo de notificações do PON
RIL-01.1	A ARQPON deverá definir um ou mais tipos de instruções que descrevam a lógica de uma *Premise* do PON.
RIL-01.2	A ARQPON deverá definir um ou mais tipos de instruções que descrevam a lógica de uma *Condition* do PON.
RIL-01.3	A ARQPON deverá definir um ou mais tipos de instruções que descrevam a lógica de um *Method* do PON.
RIL-01.4	A ARQPON deverá definir um ou mais tipos de instruções que descrevam a lógica de um *Attribute* do PON.

O requisito RIL-01 e os seus sub-requisitos são consequência do objetivo de que a ARQPON materialize os elementos do metamodelo de notificações e os execute segundo os conceitos do PON. Como isso implica que a ARQPON defina, em nível de *hardware,* uma estrutura lógica para execução de *Premises, Conditions* e *Methods* (conforme o modelo lógico mostrado na Seção 4.2.1), cada um dos elementos da cadeia que se enquadram nestas classes deve ser passível de definição em nível de instrução.

4.1.3 Requisitos de interface física

A Tabela 5 apresenta os requisitos de interface física para concepção da ARQPON.

Tabela 5 – Requisitos de interface física da ARQPON

Identificador	Descrição
RIF-01	A ARQPON deve definir uma interface física para interconexão com um dispositivo de memória externo.
RIF-02	A ARQPON deve definir uma ou mais interfaces físicas para interconexão com dispositivos periféricos de E/S.

O requisito RIF-01 permite que um processador construído segundo a ARQPON seja capaz de executar programas PON de crescente nível de complexidade, uma vez que os dados e instruções daqueles programas podem ser armazenados em um dispositivo de memória com capacidade não limitada à capacidade da memória que pode ser implementada internamente ao processador. Já o requisito RIF-02 atende à necessidade de se fornecer uma ou mais interfaces de E/S para conexão a periféricos (p. ex., controladores de barramentos externos, controladores de vídeo, etc.) que venham a compor o sistema computacional juntamente com o processador da ARQPON.

4.1.4 Requisitos operacionais

A Tabela 6 apresenta os requisitos operacionais para concepção da ARQPON. Os requisitos RO-01 e RO-02 definem a característica de execução paralela da ARQPON pelas

várias unidades de processamento conforme a dinâmica de execução ditada pelo metamodelo de notificações. Já o requisito RO-03, de forma complementar ao RIF-01, está relacionado ao suporte de execução de programas PON não limitado à quantidade de unidades de processamento disponíveis fisicamente no *hardware,* mas sim à quantidade de memória disponível.

Tabela 6 – Requisitos operacionais da ARQPON

Identificador	Descrição
RO-01	A ARQPON deve definir um determinado número de unidades de processamento capazes de execução paralela das instruções definidas pela ARQPON.
RO-02	As unidades de processamento da ARQPON devem ser capazes de propagar o fluxo de notificações entre si conforme previsto no metamodelo de notificações do PON.
RO-03	A ARQPON deve ser capaz de executar uma aplicação PON composta por mais elementos do metamodelo de notificações do que o número de unidades de processamento disponíveis para sua execução.

4.2 Projeto da ARQPON

Segundo Hennessy e Patterson (2007) (p. 9), existem três dimensões que devem orientar o projeto de uma arquitetura de computador:

- Arquitetura do conjunto de instruções (ISA): refere-se à interface de programação propriamente dita e a outros aspectos relevantes para a programação, tais como registradores internos, modos de endereçamento e tamanho das *words*.
- Organização: refere-se aos blocos arquiteturais (sistema de memória, unidades de processamento, topologia de interconexão) e como eles são organizados, em alto nível, para a execução das funcionalidades do computador.
- *Hardware*: refere-se aos detalhes específicos de implementação, tais como o projeto da lógica em baixo nível e do encapsulamento do dispositivo.

O projeto da ARQPON v1.0 leva em consideração as duas primeiras dimensões, dado que a terceira diz respeito a uma implementação específica, a qual será abordada quando

da etapa de prototipação da ARQPON v1.0 (P2ON v1.0). Complementarmente a estas dimensões, faz-se útil elaborar um modelo lógico para a ARQPON, dado que este modelo pode ser utilizado para orientar a elaboração da ISA e da organização segundo os conceitos do PON.

As seções a seguir apresentam o projeto da ARQPON v1.0 levando em consideração as duas dimensões de projeto definidas e os modelos lógico e de programação complementares.

4.2.1 Modelo lógico

A Figura 35 apresenta o modelo lógico da ARQPON. As subseções a seguir descrevem este modelo do ponto de vista estrutural e dinâmico.

4.2.1.1 Visão estrutural

Na Figura 35 as classes à esquerda (pacote *PON metamodel*) representam os elementos do metamodelo de notificações e as relações estruturais entre eles. Estes elementos são materializados, quando da implementação de *software* PON no nível de instrução da ARQPON, pelos elementos representados pelas classes ao centro (pacote *Low-level software*). Estes elementos, por sua vez, estão estereotipados como “instruções PON”, em consonância com o requisito RIL-01 para a ARQPON (ver Seção 4.1.2), e apresentam ligações entre si que mapeiam os fluxos de notificação/atualização próprios do modelo de execução do PON. Deve-se perceber que os elementos *Action, Instigation, Rule* e *FBE* não são traduzidos em instruções porque, na prática, embora sejam elementos agregadores ou roteadores que compõem as abstrações de alto nível de um *software* PON baseado em regras, estes elementos não realizam qualquer tipo de operação, em tempo de execução, que justifique o seu mapeamento para instruções de baixo nível específicas.

As relações de dependência entre as classes do pacote *Low-level software* e as classes do pacote *ARQPON* descrevem de que forma cada elemento do *software* PON de baixo nível (instrução PON) deve ser processado por uma implementação da ARQPON. Ou seja, *Premises, Conditions* e *Methods* devem ser, dependendo de sua classe, processados por um conjunto de processos paralelos de *hardware* específicos dentro da implementação da

ARQPON (*NPP* processos para *Premises, NCP* processos para *Conditions* e *NMP* processos para *Methods,* respectivamente). Este modelo atende ao requisito operacional RO-01.

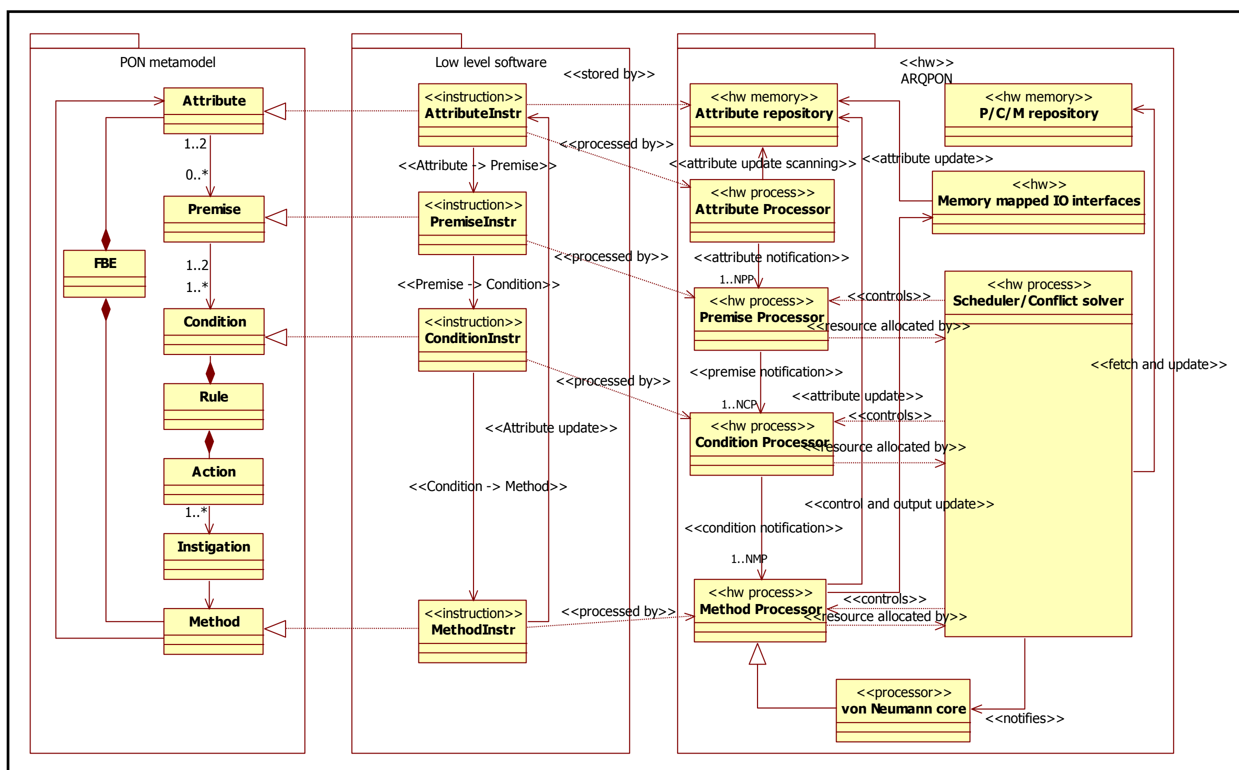


Figura 35 – Modelo lógico da ARQPON

(Fonte: autoria própria)

Dado que as quantidades *NPP, NCP* e *NMP* de processos de *hardware* são finitas, a satisfação do requisito RO-03 é obtida por meio de um processo de *hardware* responsável por alocar recursos e escalonar a execução das instruções PON nos processos de *hardware* específicos para *Premises, Conditions* e *Methods.* Este processo, apresentado na Figura 35 como *Scheduler / Conflict Solver,* deve realizar o trabalho de alocação levando em consideração cada uma das notificações que são trafegadas entre os diferentes processos de *hardware,* de tal maneira que possa verificar, via acesso ao repositório de *Premises, Conditions* e *Methods,* quais são os elementos que devem ser alocados para consumir a notificação em questão e alocá-los de fato.

A relação de extensão entre o núcleo von Neumann e o processo de *hardware* específico para processamento de *Methods* indica que uma aplicação PON pode ser logicamente implementada de forma híbrida. Ou seja, parte da funcionalidade desta aplicação seria implementada estritamente utilizando *Methods* PON, para execução pelo processo de *hardware* responsável pelos *Methods*, e parte seria implementada utilizando funções segundo

o modelo de von Neumann, a serem executadas pelo núcleo von Neumann apresentado na figura.

4.2.1.2 Visão dinâmica

Do ponto de vista dinâmico, cada conjunto de processos de *hardware* processa o conjunto de elementos do PON que lhe compete e propaga múltiplas notificações simultaneamente para o conjunto de processos imediatamente abaixo na Figura 35, conforme o modelo do PON. Os processos de *hardware* responsáveis pela execução de *Methods,* adicionalmente, efetuam atualizações no repositório de *Attributes* quando instigados. Estas atualizações, quando resultam em alteração no valor de *Attributes,* geram novas notificações que são detectadas pelo processador de *Attributes* e propagadas para os processadores de *Premises,* reiniciando um "ciclo" de notificações.

O *hardware* de E/S, descrito na Figura 35 como *Memory Mapped IO Interfaces,* pode ser acessado e controlado pelos processos de *hardware* responsáveis pelos *Methods.* Além disso, o *hardware* de E/S pode ser configurado para efetuar atualizações no repositório de *Attributes* em função de novos dados de entrada, iniciando um novo ciclo de notificações conforme já explanado.

Cada conjunto de notificações é também repassado ao *Scheduler / Conflict solver*, o qual utiliza esta informação para tomar decisões sobre alocação e desalocação, bem como para implementar o algoritmo de resolução de conflitos e habilitar ou bloquear a execução dos processos sendo notificados em função desta resolução. Em particular, a alocação de elementos aos processos é disparada pela detecção, por parte do *Scheduler / Conflict Solver*, de uma notificação para um ou mais destinatários que não estão alocados a nenhum processo de *hardware.* Cabe, então, ao *Scheduler / Conflict solver* obter os elemento(s) destinatário(s) da notificação no repositório de P/C/M, alocar este(s) elemento(s) no(s) processo(s) selecionado(s) e replicá-la somente a este(s) elemento(s), dado que os demais elementos, que já estavam alocados previamente a processos de *hardware,* já teriam tido a oportunidade de consumir esta notificação.

O *Scheduler / Conflict solver* pode, ainda, notificar o núcleo von Neumann da necessidade de execução de uma função implementada segundo o modelo de von Neumann. Ao executar esta função o núcleo von Neumann pode acessar e/ou alterar valores de

Attributes, de forma semelhante aos demais processadores de *Methods,* eventualmente iniciando um novo ciclo de notificações.

4.2.2 Considerações iniciais sobre o modelo de programação

Conforme apresentado na Figura 35, propõe-se que cada *Premise, Condition* e *Method* do *software* PON a ser executado na implementação da ARQPON seja mapeado para uma instrução. Portanto, a definição da ISA deve levar em consideração esta categorização e os dados requeridos para a descrição de cada um dos elementos do PON citados na forma de instrução (ver Seção 4.2.4).

Do ponto de vista de fluxo de controle, o PON é fundamentalmente diferente de outros modelos de execução tais como von Neumann e fluxo de dados. Enquanto em um programa von Neumann o fluxo de controle é ditado pela evolução do valor do contador de programa e em um programa de fluxo de dados o fluxo de controle é ditado pela disponibilidade dos dados para execução das instruções, em um programa PON o fluxo de controle é ditado pela propagação de notificações de alteração de *Attributes* e consequentes alterações dos valores de *Premises* e *Conditions* a eles associadas. Sendo assim, dado que o fluxo de controle PON é dependente do correto mapeamento das dependências de notificação entre os elementos, conclui-se que as instruções da ISA da ARQPON devem definir explicitamente este mapeamento.

Do ponto de vista de fluxo de dados, programas von Neumann acessam dados em posições de memória, calculadas estaticamente ou dinamicamente caso alguma informação de escopo ou contexto seja necessária para localização dos dados (caso do uso de variáveis automáticas de função ou atributos de objeto, por exemplo). Já programas puramente de fluxo de dados não definem o conceito de variáveis da mesma forma que os programas von Neumann, visto que os dados são produzidos pela execução das instruções e imediatamente repassados e consumidos como parâmetros para execução de outras instruções que deles dependem.

Programas PON, por sua vez, definem *Attributes* que representam o estado dos FBEs e, portanto, dependem de uma implementação que armazene esta informação de estado na forma de variáveis que não sejam desalocadas ou consumidas. Sendo assim, faz-se necessário reservar espaço em memória para armazenamento dos dados associados a um *Attribute.* Além disso, *Attributes* também fazem parte da cadeia de notificações do PON e, portanto, são

elementos ativos que possuem ligações com outros elementos (*Premises*) e algumas propriedades que influenciam no seu comportamento lógico (p. ex. ser impertinente, exclusivo ou determinístico). Isto requer que todas estas informações também sejam mapeadas em uma ou mais instruções específicas, a serem executadas pelo processo de *hardware* responsável pelos *Attributes* (ver Figura 35).

Para interagir com dispositivos de E/S, a ARQPON pode definir o mapeamento de registradores em memória. Isto facilita a definição dos *Methods,* dado que estes podem tratar da escrita/leitura de valores dos dispositivos periféricos da mesma forma como tratam escritas/leituras de *Attributes*, porém sem necessariamente disparar o fluxo de notificações da aplicação.

Feitas estas considerações, pode-se categorizar o modelo de programação da ARQPON como um novo modelo de execução do PON que (como o próprio PON) reaproveita algumas características do modelo imperativo e do modelo declarativo, pelas seguintes razões:

- As instruções propostas são imperativas no sentido de que descrevem “o que” fazer de forma assertiva, quando do recebimento de notificação pelo elemento que é descrito por aquela instrução.
- As instruções propostas também são declarativas justamente por serem mapeáveis diretamente para os elementos da cadeia de notificações, descrevendo a sua estrutura e/ou seu comportamento lógico e as relações que eles apresentam com os demais elementos (descrição do fluxo de controle).

A Seção 4.2.4 descreve detalhadamente a ISA proposta para a ARQPON v1.0, levando em consideração todos os detalhes do modelo de programação abordados anteriormente. Na Seção 4.2.3 a seguir, por sua vez, são abordados outros aspectos arquiteturais relevantes, relacionados à fundamentação teórica apresentada no Capítulo 2 e que servem como orientação para o detalhamento da ARQPON v1.0.

4.2.3 Aspectos arquiteturais relevantes

4.2.3.1 Processador de granularidade fina

O modelo de execução definido pelo PON implica a propagação potencialmente paralela de notificações entre os elementos da cadeia que são conectados para a implementação da lógica causal (*Attributes, Premises, Conditions, Actions, Instigations* e *Methods*). Ou seja, é possível que, em determinado instante de tempo, elementos de diferentes classes na cadeia de notificações estejam alocados para os processos de *hardware* respectivos e executando suas operações de forma simultânea e independente.

O paralelismo potencial de uma aplicação PON é melhor explorado quanto maior for o número de diferentes operações prontas para execução que possam ser imediatamente executadas sem necessitar compartilhar o recurso de processamento com outras operações. Para tanto, parte-se do princípio de que cada elemento da cadeia de notificações do PON pode ter seu comportamento executado por uma unidade de processamento específica (representada por um processo de *hardware* conforme o modelo lógico apresentado na Figura 35), exclusivamente alocada para execução desta tarefa, porém passível de realocação (efetuada pelo processo *Scheduler / Conflict solver*, também apresentado na Figura 35) para a execução do comportamento de outros elementos conforme seja necessário.

Considerando-se esta alocação exclusiva, requer-se um baixo grau de complexidade de cada unidade de processamento, dado o baixo grau de complexidade das operações executadas pelos elementos da cadeia do PON quando consideradas individualmente. De fato, *Premises* executam operações relacionais simples, *Conditions* executam operações lógicas simples e *Actions* e *Instigations* são simplesmente elementos propagadores de notificação. Em relação aos *Methods,* estes podem executar lógicas complexas (iterações, por exemplo), porém ainda assim é possível desenvolver aplicações PON nas quais aquelas lógicas sejam decompostas em operações mais simples. Uma vez estabelecida a operação mínima que deve ser executada por um método PON ("método mínimo"), pode-se projetar o elemento de processamento de métodos de tal maneira que este seja simples o suficiente para somente executar a operação mínima. Uma reflexão sobre esta questão é apresentada na Seção 4.2.3.4.

A implementação de múltiplas unidades básicas de processamento é uma estratégia que tem sido levada em consideração no que diz respeito à evolução das arquiteturas de computadores paralelos. Segundo Kogge *et al* (2008), aplicações que demandem alto desempenho (p. ex., sistemas de manipulação de grandes quantidades de dados) podem

necessitar de *hardware* que suporte a execução de até bilhões de *threads* em paralelo. Ainda, Kogge *et al* (2008) mencionam que esta multiplicidade demanda modelos de programação adequados (sem propor nenhum especificamente), no que se pode enquadrar então os fundamentos do PON aplicados à ARQPON conforme discorrido anteriormente. Por sua vez, Asanovic *et al* (2006) argumentam que tais núcleos podem ser simples o suficiente para facilitar técnicas de verificação formal que são de difícil aplicabilidade em arquiteturas mais complexas, além do que estudos anteriores mostram que *pipelines* de núcleos mais complexos acarretam custos em termos de dissipação de energia que nem sempre compensam os ganhos teóricos em desempenho.

Dado que as operações das *Premises,* das *Conditions* e dos *Methods* são executadas por múltiplas unidades de processamento, alocadas exclusivamente para execução de uma destas operações por vez e que cada uma destas operações individualmente é de baixa complexidade, conclui-se que faz sentido especializar as unidades de processamento em função do tipo de operação. Esta especialização objetiva aproveitar melhor os recursos de *hardware* disponíveis, pois permite que estes recursos sejam alocados em um número maior de unidades de processamento de implementação relativamente mais simples em *hardware.* Segundo Ungerer, Robic e Silc (2003), núcleos de implementações do tipo *interleaved multithreading* (IMT), que executam instruções de várias *threads* de maneira intercalada, tendem a ser mais simples porque nestes núcleos não é necessário *hardware* específico para a detecção de conflitos em *pipeline* (não existe exploração de ILP pois duas instruções consecutivas de uma mesma *thread* nunca estão em execução simultânea no *pipeline*). Embora não sejam implementações de IMT, os núcleos propostos para execução de *Premises, Conditions* e *Methods* também apresentam a propriedade de não necessitar detecção de conflitos em *pipeline,* visto que a própria estrutura do *software* PON implica que cada um destes núcleos execute uma operação por vez, excitada por uma notificação recebida conforme a dinâmica da cadeia de notificações.

Para a especialização das unidades de processamento da ARQPON define-se a seguinte terminologia:

- PP (*Premise Processor* ou Processador de Premissa): unidade de processamento responsável por efetuar o cálculo lógico de uma operação relacional envolvendo os valores de um ou dois *Attributes*
- CP (*Condition Processor* ou Processador de Condição): unidade de processamento responsável por efetuar o cálculo lógico de uma operação lógica envolvendo os valores lógicos de duas *Premises* ou *SubConditions.*

- MP (*Method Processor* ou Processador de Método): unidade de processamento responsável por executar a operação mínima executável por um método PON em resposta a um valor lógico verdadeiro notificado por uma *Condition* (ver Seção 4.2.3.4 para mais detalhes).

Partindo-se desta terminologia, a implementação física dos conjuntos de processos de *hardware* utilizados para processamento das *Premises, Conditions* e *Methods* (Figura 35) é obtida por meio do agrupamento dos PPs, CPs e MPs, respectivamente.

Em função das características citadas de execução independente, com alto grau de paralelismo e, ao mesmo tempo, heterogênea no sentido de se constituir de diferentes tipos de operação (avaliações relacionais, lógicas e/ou atualizações de atributos), pode-se classificar esta proposta para a ARQPON como uma proposta de arquitetura com granularidade de paralelismo fina (TANENBAUM, 2007). Ou seja, o processador construído segundo a ARQPON deve ser composto por muitos elementos de processamento relativamente simples, eventualmente com baixo poder de processamento e interagindo por meio de comunicação de relativamente alta velocidade. A seção a seguir discorre sobre a questão de comunicação/interconexão entre as múltiplas unidades de processamento.

4.2.3.2 Topologia de interconexão

Por ser proposta na forma de uma arquitetura multiprocessada, a ARQPON deve definir uma topologia de interconexão entre as diversas unidades de processamento que a compõem.

O modelo lógico de execução orientada a notificações do PON implica a propagação de resultados do cálculo de *Premises* para parâmetros de *Conditions,* bem como da avaliação de resultados dos cálculos de *Conditions* para a consequente ativação de *Methods*. Sendo assim, pode-se afirmar que existem, logicamente, camadas de interconexão entre *Premises* e *Conditions* e entre *Conditions* e *Methods,* as quais são apresentadas no modelo lógico da Figura 35 na forma de *links* portadores de notificações entre os conjuntos de processos de *hardware* relacionados a *Premises, Conditions* e *Methods,* conforme o requisito operacional RO-02.

Em função desta característica, propõe-se, para a ARQPON v1.0, a definição de camadas físicas de interconexão hierárquicas distintas entre os grupos de PP e CP e entre os

grupos de CP e MP que implementam fisicamente o modelo lógico de processos de *hardware*. A Figura 36 demonstra este aspecto da arquitetura.

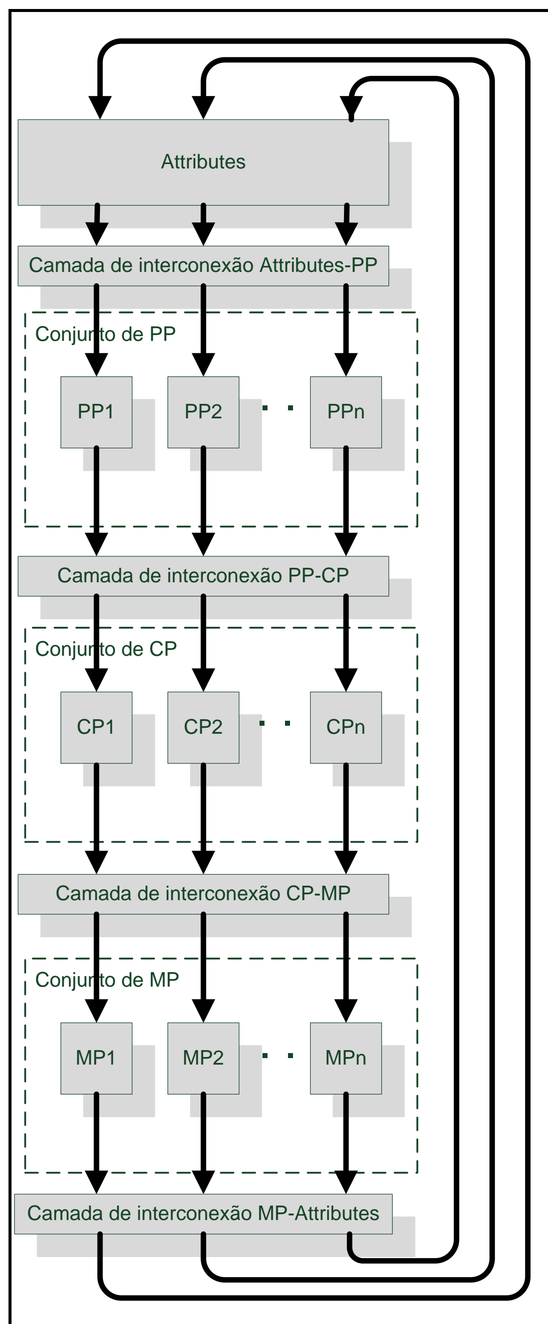


Figura 36 – Camadas de interconexão da ARQPON v1.0

(Fonte: autoria própria)

Dado o comportamento desejado de realocação de unidades de processamento para diferentes elementos da cadeia em tempo de execução (p. ex. poder realocar um determinado CP para diversas *Conditions* da aplicação PON à medida que for necessário, em atendimento ao requisito operacional RO-03), é necessário que a implementação física da camada de

interconexão suporte roteamento dinâmico entre PP e CP e entre CP e MP. Além disso, segundo o metamodelo da cadeia de notificações do PON, cada *Premise* pode potencialmente notificar N *Conditions* que dela dependem, e cada *Condition* pode potencialmente notificar a execução de N *Methods* (por meio do mapeamento via *Instigations*), portanto é necessário que a implementação física da camada de interconexão suporte também o roteamento entre 1 elemento transmissor e N elementos receptores das notificações.

Ainda, dado que uma *Premise* deseja notificar uma *Condition* não necessariamente estando ciente de qual CP ela está ocupando (pelo fato da alocação ser dinâmica, efetuada pelo *Scheduler / Conflict solver*), o endereço da unidade de processamento de destino da notificação não é necessariamente conhecido no início da comunicação. Esta questão poderia ser solucionada por meio de consulta em uma tabela centralizada que contivesse o mapeamento entre endereço lógico e unidade de processamento. No entanto, isto agregaria complexidade ao protocolo de propagação de notificações e possivelmente *overhead* de comunicação, com impacto no desempenho.

Outra opção consiste em utilizar uma camada de interconexão que suporte *snooping*, no qual a *Premise* enviaria a notificação por meio de um canal comum e esta notificação seria monitorada e consumida por um ou mais CPs que identificassem que a informação é relevante para eles. O *Scheduler / Conflict Solver*, adicionalmente, também deveria ser capaz de consumir esta informação simultaneamente para verificar se alguma *Condition* que deveria ser notificada não está alocada em nenhum CP e proceder a alocação, caso necessário, encaminhando posteriormente a notificação para as *Conditions* recém-alocadas. Embora não seja totalmente coerente com o modelo de notificações do PON, no qual cada elemento notificante possui uma ligação e acessa (endereça) diretamente cada um dos seus elementos notificados, esta abordagem tende a ser relativamente eficiente do ponto de vista de utilização de banda, dado que uma mesma notificação pode ser consumida simultaneamente por vários elementos. Além disto, esta abordagem simplifica sobremaneira a camada de interconexão por não depender de conexões elétricas diretas entre cada elemento notificante e cada elemento notificado.

Finalmente, em relação ao uso concorrente da camada de interconexão, é necessária uma topologia que suporte múltiplos nós capazes de gerar e transmitir notificações potencialmente de forma simultânea, conforme o modelo do PON.

Consideradas estas questões, pode-se avaliar a aplicabilidade à ARQPON de cada uma das técnicas arquiteturais comuns para interconexão entre multiprocessadores apresentadas na Seção 2.3.6. A Tabela 7 apresenta esta avaliação.

Tabela 7 – Aplicabilidade das técnicas de interconexão à ARQPON

Técnica de interconexão	**Roteamento dinâmico**	**Comunicação 1 para N**	**Comunicação com endereço de destino ignorado (utilizando *snooping*)**	**Comunicação simultânea de múltiplos nós**
Barramento	Suportado (mensagem sempre alcança as interfaces de todos os nós da rede)	Suportado (por meio de *multicast* com lista de endereços ou *broadcast*)	Suportado (desde que nós de destino saibam identificar que são destinatários da mensagem)	Suportado (desde que sincronizada por meio de um árbitro de barramento multi-mestre)
Comutador *crossbar*	Suportado (por meio da ativação/desativação dinâmica dos *switches*)	Suportado (por meio da replicação da mensagem variando-se a comutação)	Não suportado	Suportado, com algumas limitações em função do caminho desejado
Rede Omega	Suportado (por meio da ativação/desativação dinâmica dos *switches*)	Suportado (por meio da replicação da mensagem variando-se a comutação)	Não suportado	Suportado, com algumas limitações em função do caminho desejado
Network on Chip	Suportado (por meio de alterações de endereçamento)	Suportado (por meio de *multicast* com lista de endereços ou *broadcast*)	Suportado por meio de *broadcast* de pacotes	Suportado por meio de contenção nos roteadores

De acordo com a Tabela 7, as características desejadas para as camadas de interconexão da ARQPON restringem a sua implementação a barramentos ou NoCs.

Ponderando-se entre as limitações elétricas apresentadas por barramentos, em função do número de processadores interconectados, e entre a relativamente mais alta complexidade de implementação dos elementos de uma NoC, propõe-se para a ARQPON v1.0 uma arquitetura de interconexão baseada em barramento multi-mestre para uma quantidade limitada de unidades de processamento. Esta arquitetura pode ser eventualmente evoluída para um modelo híbrido (subconjuntos de unidades de processamento compartilhando um barramento e interconectados a outros subconjuntos por meio de *bridges* com papel de roteadores/repetidores, semelhante ao que ocorre em uma NoC) caso necessário. A Figura 37 descreve como seria a organização de barramento multi-mestre (a) e a organização híbrida (b).

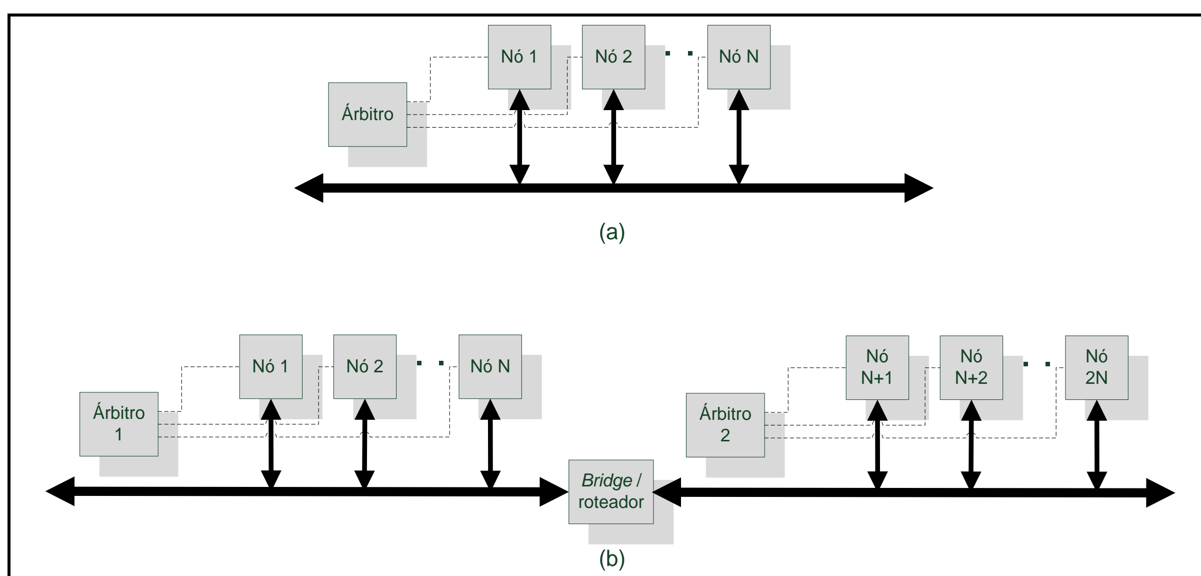


Figura 37 – Organização multi-mestre com barramento e híbrida

(Fonte: autoria própria)

Do ponto de vista de desempenho, a arquitetura baseada em barramento multi-mestre apresenta as seguintes desvantagens:

- A presença de múltiplos mestres requer uma etapa de arbitragem para cada operação de transmissão, a qual é realizada por um árbitro de barramento e que geralmente introduz uma latência crescente em função do número de mestres (BJERREGAARD, MAHADEVAN, 2006).
- A banda é compartilhada entre todos os mestres conectados ao barramento.

A questão do compartilhamento da banda, do ponto de vista de propagação de notificações, é minimizada pelo fato de que os múltiplos processadores (PP, CP e MP) estão distribuídos em camadas de interconexão diferentes, portanto há dois barramentos disponíveis (entre PP e CP e entre CP e MP), o que duplica a banda disponível. Já a complexidade e a latência agregadas pela existência dos árbitros de barramento serão ponderadas e avaliadas quando dos testes realizados com protótipos do P2ON v1.0.

A título de comparação, a abordagem aplicada por Peters (2012), na concepção do co-processador PON, foi mapear a dependência de *Conditions* em relação a *Premises* por meio de um mapa de bits programado para cada *Condition*, no qual os bits correspondentes a cada *Premise* que interessa estão ligados. Caso esta solução fosse considerada para a ARQPON, a realocação dinâmica de uma *Premise* em um PP implicaria atualizar dinamicamente os mapas de bits de todas as *Conditions* que estão alocadas nos CP. Além disso, seria possível que, para algumas das *Conditions* alocadas em determinado instante, as *Premises* correspondentes não estivessem todas alocadas em PPs; neste caso, o mapa de bits já não seria suficiente porque seria capaz de representar um mapeamento de fluxo de notificação somente para as *Premises* correntemente alocadas.

Seleção do padrão de barramento para a ARQPON

A princípio seria mais interessante, do ponto de vista de projeto, selecionar uma arquitetura de barramento existente e padronizada, tais como as arquiteturas Avalon e AMBA já mencionadas como exemplos na Seção 2.3.6. A vantagem desta abordagem seria que, para estes tipos de arquiteturas, geralmente existe uma especificação consolidada e eventualmente até mesmo circuitos ou módulos IP (*Intellectual Property*) à disposição que podem ser configurados e reaproveitados para a prototipação do P2ON v1.0 em FPGA. Ainda, o barramento Avalon em particular é diretamente configurável via ferramenta *SOPC Builder* da Altera, o que facilita a prototipação.

Ambos os barramentos citados e outros comumente utilizados neste tipo de aplicação, embora geralmente configuráveis para uma topologia multi-mestre, dependem que cada mestre enderece diretamente os seus escravos (ALTERA, 2002) (MITIĆ, STOJČEV, 2006). Isto impõe uma restrição à propagação de notificações nas camadas de interconexão da ARQPON, no sentido de que cada gerador de notificação (p. ex., PP) deveria endereçar diretamente cada consumidor de notificação ao qual está logicamente conectado (p. ex. CP). Isto levaria à necessidade de um mapeamento de endereços via tabela, conforme discutido

anteriormente, e inviabilizaria a ideia de implementação de *snooping* pelos nós receptores que fossem pertinentes.

Uma implementação utilizando barramento Avalon, em particular, também seria menos viável pelo fato de que, neste barramento, cada escravo é fisicamente conectado a cada um dos mestres que podem se comunicar com ele por meio de uma conexão dedicada (via multiplexadores) (ALTERA, 2002). No caso da ARQPON, como os PPs podem, potencialmente, notificar qualquer condição alocada em qualquer CP, isto levaria à necessidade de uma rede de *NPP x NCP* interconexões, o que poderia se tornar muito custoso em termos de *hardware.* O mesmo raciocínio vale para a camada de interconexão entre CP e MP. No entanto, é pertinente ressaltar que esta solução oferece uma maior largura de banda por permitir a comunicação paralela entre diferentes pares de mestre e escravo.

Considerando então um balanço entre complexidade de protocolo, custo de implementação em *hardware,* reaproveitamento de soluções já existentes, flexibilidade e adequabilidade à abordagem da ARQPON, optou-se preliminarmente por propor uma solução customizada de barramento, com as peculiaridades inerentes a cada uma das camadas de interconexão da ARQPON. No entanto, não se descarta a adoção de um barramento padronizado, dependendo das deficiências na abordagem customizada a serem levantadas em uma primeira avaliação do protótipo do P2ON v1.0.

As características dos barramentos utilizados em cada uma das camadas de interconexão da ARQPON são expostas na Seção 4.2.5.3.

4.2.3.3 Escalabilidade

Do ponto de vista de escalabilidade, a divisão em múltiplas unidades de processamento simples é um facilitador. Isto ocorre porque, uma vez que se deseje aumentar a capacidade de processamento, é mais simples aumentar a quantidade de unidades de processamento replicando as unidades simples que já existem do que projetar novas unidades de processamento mais complexas tentando otimizar o desempenho de execução (UNGERER; SILC; ROBIC, 2003). Além disso, como o modelo de programação da ARQPON, à luz do próprio PON, é fundamentado na distribuição do processamento, não seriam necessárias adaptações significativas em termos de implementação de *software.*

Em contraposição, a arquitetura de interconexão pode se tornar um fator dificultador da escalabilidade. Em particular, a utilização de barramentos, conforme proposto na Seção

4.2.3.2, apresenta limitações elétricas que podem inviabilizar a ampliação de escala para utilização de muitos núcleos, além do que a banda utilizável por cada núcleo tende a diminuir visto que todos os núcleos compartilham o mesmo canal de comunicação (ver Seção 2.3.6). No entanto, conforme também proposto na Seção 4.2.3.2, uma alternativa para ampliação da escala seria uma arquitetura de interconexão híbrida, com seções de barramento interligadas por *bridges* segundo os princípios de uma NoC.

O impacto exato dos aspectos discutidos na escalabilidade será melhor levantado e analisado quando das primeiras avaliações do protótipo do P2ON v1.0.

4.2.3.4 Definição do "método PON mínimo"

Conforme as Seções 2.3.7.1 e 2.3.7.2, as instruções definidas pelas ISAs de arquiteturas comuns von Neumann e de fluxo de dados são agrupáveis em um número reduzido de categorias. Cada uma destas categorias pode ser mapeada para uma abstração do PON, conforme apresentado a seguir.

Em relação às ISAs von Neumann:

- operações lógicas relacionais: efetuadas pelas *Premises* do PON e, implicitamente, pelos *Attributes* para decidir se uma notificação de alteração deve ser gerada ou não.
- operações lógicas *booleanas*: efetuadas pelas *Conditions* do PON.
- operações de acesso à memória: efetuadas pelos *Methods* para escrita/leitura de dados, bem como pelos *Attributes* para decidir sobre o envio de notificação ou não.
- operações aritméticas e de bit: executadas pelos *Methods* para cálculo de valores a serem escritos nos *Attributes.*
- operações de movimentação de registradores: não são definidas explicitamente, pois as movimentações de dados são efetuadas diretamente por meio de leituras e escritas nos *Attributes.*
- operações de desvio: não são definidas no PON, pois o fluxo de controle é determinado inteiramente pelo fluxo de notificações.
- operações de E/S: podem ser consideradas um caso particular de acesso à memória, nas arquiteturas em que os registradores dos periféricos são mapeados em memória.

Em relação às ISAs de fluxo de dados (modelo elementar), estas definem as operações *Operator, Decider* e *Boolean* que mapeiam para operações aritméticas e de bit, relacionais e lógicas do modelo de von Neumann, respectivamente. As demais operações definidas pelo modelo elementar são abstrações da operação de desvio, eventualmente aplicando alguma lógica condicional.

Conclui-se, portanto, que em um modelo de execução PON conforme proposto pela ARQPON os *Methods* devem ser capazes de executar, em nível de máquina, operações aritméticas, de bit e de escrita/leitura de dados em memória. Sendo assim, a operação executada pelo "método mínimo" da ARQPON pode ser descrita segundo um dos seguintes formatos genéricos:

$$RES \leftarrow OPN1\ op\ OPN2$$
$$RES \leftarrow op\ OPN1$$

Nestes formatos, *OPN1* e *OPN2* são valores de operandos obtidos da memória (valores de *Attributes* ou de interfaces de entrada mapeadas em memória), *op* é uma operação aritmética ou de bit e *RES* é o resultado da operação a ser escrito na memória (na forma de uma atualização de valor de *Attribute* ou como valor a ser fornecido a uma interface de saída mapeada em memória). No primeiro formato, *op* se refere a uma operação binária (dependente de dois operandos), ao passo que no segundo formato *op* se refere a uma operação unária (dependente de um único operando).

Nas situações em que o processamento de um método envolva executar uma sequência de operações, cada uma delas conforme um dos formatos apresentados, a regra PON que o define pode ser decomposta em um conjunto de regra-mestre (*master rule*) e de regras dependentes, de tal maneira que o *Method* definido para cada uma das regras do conjunto esteja em conformidade com a definição de método mínimo.

4.2.3.5 Gerenciamento de informação de contexto

A informação de contexto constitui-se, genericamente, em um conjunto de dados que é processado por uma instância específica de uma tarefa e que permite distingui-la de outras instâncias específicas da mesma tarefa.

Tomando-se como exemplo a aplicação mira alvo (ver Seção 2.5.1), a informação de contexto poderia ser utilizada para permitir que todos os FBEs que correspondem aos arqueiros pudessem compartilhar a mesma definição do método “atirar flecha”, porém sendo a invocação do método particular para o contexto de cada FBE. Isto implicaria o método acessar somente os *Attributes* que fazem parte do FBE em cujo contexto está sendo invocado.

O conceito de contexto está presente em programas segundo o PI, tanto em arquiteturas sequenciais quanto em arquiteturas paralelas. Nas arquiteturas sequenciais de Programação Procedimental (PP-PI) pode-se caracterizar o conteúdo do *frame* atual da pilha de chamada de funções como uma informação de contexto, dado que armazena argumentos e dados locais da instância atual de função sendo executada. Nas arquiteturas sequenciais de Programação Orientada a Objetos (POO-PI), por sua vez, o encapsulamento de atributos em objetos também define uma forma de contexto (ou escopo de objeto).

Em arquiteturas paralelas, particularmente nas que implementam TLP, cada uma das *threads* possui uma informação de contexto atrelada a si. Isto permite que *threads* distintas executem o mesmo código, porém atuando sobre um conjunto de dados distinto e particular de cada *thread.* Geralmente o contexto em TLP é implementado por meio de áreas de pilha distintas, em uma extensão do que ocorre para chamadas de funções em programas *single threaded.* No entanto, arquiteturas que implementam *multithreading* oferecem conjuntos de registradores independentes para o gerenciamento de cada contexto, dado que o chaveamento entre *threads* é realizado em nível de *hardware.*

O conceito de contexto, da forma apresentada, é implementado no *framework* C++ do PON por meio da representação dos FBEs em classes e objetos e por meio do armazenamento da referência (ponteiro) para estes objetos, que é efetuado por cada objeto da classe *Method.* Portanto, embora o objeto da classe *Method*, correspondente a um elemento do metamodelo de notificações, seja único e exclusivo para cada contexto, o código que ele executa é compartilhado e executado no contexto de um FBE específico.

Para o modelo da ARQPON, como as operações executadas por cada *Premise, Condition* e *Method* estão essencialmente atreladas a uma instância específica de objeto da cadeia de notificações, fazê-los compartilhar o seu código entre a execução de diferentes contextos criaria uma dificuldade em relação ao mapeamento do recebimento e envio de notificação, que poderia ser variável dependendo do contexto em que se estivesse executando. Sendo assim, para fins de simplificação da ARQPON v1.0, propõe-se que não haja um mecanismo de gerenciamento de contexto implícito na arquitetura de execução, ou seja, cada possível instância de *Premise, Condition* ou *Method* deve ser codificada explicitamente na

forma da instrução a ser especificada (ver Seção 4.2.4). A utilização da mesma semântica de *Premise, Condition* ou *Method* por diferentes FBEs implica, portanto, na replicação das instruções correspondentes para cada um dos FBEs envolvidos.

No caso do disparo de métodos no núcleo von Neumann (ver Seção 4.2.3.8), a definição da instrução PON específica deve prever um campo para definição de um argumento numérico, que pode ser utilizado pelo método von Neumann como informação de contexto.

4.2.3.6 Semântica de acesso à memória

Diferentemente do modelo de fluxo de dados, no qual a execução das operações é sincronizada pela presença e consumo dos dados de entrada (parâmetros), no PON a execução das operações de *Premises, Conditions* e *Methods* é sincronizada pela geração e consumo de notificações. Sendo assim, uma semântica de acesso à memória tal qual a oferecida por *I-Structures* não é de muita utilidade para a ARQPON, pois oferece uma forma de sincronização de acesso baseada na presença ou não do dado sendo lido.

Em relação à forma de endereçamento da memória, dado que se pretende codificar um *Method* utilizando-se a definição de método mínimo, não é conveniente implementar um modelo *load/store* no sentido de que este depende de uma sequência de instruções para efetivar o acesso (carga de endereço em registrador e posterior acesso à memória – *load/store* – na posição indicada pelo endereço carregado). Desta forma, optou-se inicialmente por implementar endereçamento direto e imediato (endereços codificados explicitamente no *opcode*) para as instruções executadas pelos MPs, de maneira semelhante ao endereçamento direto implementado em algumas arquiteturas CISC.

Finalmente, as posições de memória correspondentes a *Attributes* necessitam uma semântica particular de acesso, segundo a qual uma alteração no valor do *Attribute* é percebida e convertida em uma notificação a ser enviada para os PPs, iniciando o fluxo de notificações. Em uma memória convencional, a alteração no valor de *Attribute* implica a execução de uma sequência de:

- leitura do valor atual;
- comparação com o valor a ser escrito;
- se for diferente, escrita do novo valor e geração de notificação.

Ou seja, no melhor caso ocorrerá um acesso de leitura à memória e, no pior caso, um acesso de leitura e um de escrita. Em ambos os casos é necessário a execução de uma comparação de igualdade, que é uma operação relacional.

Propõe-se, para a ARQPON, uma arquitetura de memória na qual a atualização no valor de *Attribute* implique a execução de uma sequência de:

- escrita do novo valor e comparação simultânea;
- se for diferente, geração de notificação.

Ou seja, em qualquer caso ocorrerá um acesso de escrita à memória, porém simultâneo à comparação com o valor atual. Também, neste caso, é necessária a execução de uma operação relacional de comparação de igualdade para geração de notificação, com exceção das situações em que o *Attribute* seja configurado como renotificante, nas quais a notificação deve ser gerada independentemente de alteração no valor.

Esta semântica de acesso à memória é denominada, a partir deste ponto, de *CSMA* (*Change-Sensitive Memory Access,* ou Acesso à Memória Sensível a Alteração). A sua implementação efetiva depende da tecnologia a ser utilizada para prototipação, haja visto que o comportamento desejado com CSMA possivelmente implique em alterações no projeto eletrônico dos circuitos que implementam os *bits* de memória.

4.2.3.7 Interfaces externas (dispositivos de E/S)

Conforme explicitado no modelo de programação (Seção 4.2.2), a ARQPON define um mapeamento em memória para os registradores que controlam o funcionamento das interfaces para os dispositivos de E/S. Isto permite que a implementação de método mínimo suportada pelo MP possa também ser utilizada para a interação com os dispositivos de E/S, dado que a escrita/leitura de registradores de dispositivos de E/S se traduz em acessos diretos à memória.

Ainda, como os dispositivos de E/S podem ser categorizados como *FBEs*, os dados fornecidos por eles ao P2ON devem poder ser abstraídos como *Attributes* do PON. Isto é obtido por meio da declaração de *Attributes* comuns que são configurados para serem atualizados diretamente pelos dispositivos de E/S, permitindo a geração de notificações para as *Premises* conectadas e assim emulando, de certa forma, o mecanismo de interrupções de

hardware que é comumente implementado em arquiteturas baseadas no modelo de von Neumann.

4.2.3.8 Interface com processador von Neumann

Conforme o modelo lógico apresentado na Figura 35, e também atendendo ao requisito RF-02, a ARQPON prevê que o *Scheduler / Conflict solver* possa notificar um núcleo von Neumann para execução de métodos mais complexos do que os suportados pelos MP, em situações nas quais isto se mostre mais conveniente. Isto permite à ARQPON executar *software* híbrido de forma semelhante ao que ocorre na arquitetura proposta por Peters (2012) para o CoPON. No entanto, diferentemente deste, mesmo com um núcleo von Neumann disponível, a ARQPON continua sendo capaz de executar parte dos *Methods* segundo o modelo básico do PON, por meio dos MPs.

O núcleo von Neumann deve ser completamente integrável ao barramento de escrita/leitura de *Attributes,* inclusive fazendo uso da semântica de acesso descrita na Seção 4.2.3.6, e interligado a uma memória de armazenamento de código binário para os métodos von Neumann. Esta memória é preferencialmente separada da memória responsável por armazenar os códigos da *Premises, Conditions* e *Methods,* a serem executados pelos PP, CP e MP, para não interferir na dinâmica de *fetching* de *Premises, Conditions* e *Methods* que é particular do modelo do PON.

Para que o *Scheduler / Conflict solver* possa gerenciar a alocação do núcleo von Neumann de forma independente dos demais MP e o mais desacoplada possível da implementação do núcleo, propõe-se a definição de uma instrução PON específica para o disparo de um método von Neumann. Esta instrução será executada pelo *Scheduler / Conflict Solver* e terá como objetivo iniciar o controle do processo de execução no núcleo von Neumann. Para fins de gerenciamento de informação de contexto (ver Seção 4.2.3.5) e/ou de passagem de parâmetros, esta instrução deve prever a definição de um argumento numérico.

4.2.4 Definição da ISA

A partir da análise de granularidade efetuada na Seção 4.2.3.1, propôs-se que a execução das instruções na ARQPON seja distribuída nos PP, CP e MP. Sendo assim, a

definição da ISA deve contemplar um mapeamento de cada instrução para o tipo de unidade de processamento capaz de executá-la.

Conforme o modelo de programação apresentado na Seção 4.2.2, a ISA da ARQPON v1.0 deve definir as seguintes instruções:

- Instrução para declaração de *Attribute* (ATTRIBUTE-DECL)
- Instrução para operação de *Premise* (PREMISE-OP)
- Instrução para operação de *Condition* (CONDITION-OP)
- Instrução para operação de *Method* (METHOD-OP)
- Instrução para disparo de método von Neumann (METHOD-VN-OP)

Além disso, pode-se partir de algumas premissas relacionadas à teoria do PON e a características/limitações da ARQPON:

- ATTRIBUTE-DECL deve prever bit para sinalizar que um *Attribute* é exclusivo (resolução de conflitos).
- ATTRIBUTE-DECL deve prever um bit para sinalizar que um *Attribute* é “não notificante”, ou seja, que não deve gerar notificação mesmo quando alterado. Esta é uma outra forma de sinalizar que o *Attribute* é impertinente porém em escopo global, ou seja, independente da sua pertinência ou não em relação às *Premises.*
- ATTRIBUTE-DECL deve prever um bit para sinalizar que um *Attribute* é renotificante, ou seja, que deve gerar notificação sempre que atualizado, independente de ter sido alterado ou não. Este bit é menos prioritário do que o bit que sinaliza que o *Attribute* é não notificante.
- PREMISE-OP deve prever bits para indicar se os *Attributes* que constituem seus operandos são “impertinentes” para aquela *Premise* ou não, ou seja, que não devem ser avaliados mesmo quando geram notificações de alteração.
- PREMISE-OP e CONDITION-OP devem prever um bit para definição de *Exclusive-Premise* e *Exclusive-Condition.*
- ATTRIBUTE-DECL, PREMISE-OP e CONDITION-OP devem prever bits para definir *Deterministic-Premises, Deterministic-Attributes* e *Deterministic-Conditions.* Todos estes elementos aguardam contra-notificações dos elementos que notificaram, conforme o mecanismo de determinismo e resolução de conflito descrito na Seção 4.2.5.7.2.

As subseções a seguir discorrem sobre questões arquiteturais relacionadas à ISA e apresentam descrição detalhada de cada uma das instruções definidas.

4.2.4.1 Tamanho das instruções

Conforme apresentado no modelo lógico da ARQPON (Seção 4.2.1), o escalonador de *Premises, Conditions* e *Methods* deve ser capaz de identificar quais são os destinatários de cada notificação. Sendo assim, um determinado *Attribute* deve codificar, na instrução que o define, todas as *Premises* que deveria notificar; cada *Premise* deve fazer o mesmo em relação às *Conditions* e cada *Condition* deve fazer o mesmo em relação aos *Methods.*

Portanto, conclui-se que o tamanho em bits de cada instrução é variável e dependente do tamanho da lista de elementos notificados em função da execução desta instrução. Excluindo-se a lista de elementos notificados, cada instrução da ISA da ARQPON v1.0 ocupa até 5 palavras de 32 bits cada para os demais campos (incluindo campos reservados ou não utilizados).

4.2.4.2 Tipos de dados

Os dados processados pela ARQPON não podem ser considerados simplesmente agrupamentos de bits que codificam valores, visto que a definição de um *Attribute* engloba uma série de outras propriedades além do seu próprio valor. Sendo assim, é mais conveniente e simples que a ARQPON v1.0 defina um único formato de dado de 32 bits (*word*) para codificação do valor de um *Attribute,* sendo que este dado pode representar:

- Número inteiro com sinal (*int*) de 32 bits, utilizado em operações aritméticas efetuadas pelos *Methods.*
- Palavra (*word*) de 32 bits, utilizado em operações de bit efetuadas pelos *Methods.*

A definição de um tamanho único de 32 bits para os dados facilita o detalhamento de cada instrução da ISA e simplifica o processo de decodificação. No entanto, isto ocorre às custas de um aproveitamento não otimizado do espaço de memória disponível. Em decorrência, a ARQPON v1.0 não define tipos de dados que não sejam inteiros, tais como números de ponto fixo ou ponto flutuante.

4.2.4.3 Detalhamento das instruções

As subseções a seguir apresentam a codificação de cada uma das instruções da ISA da ARQPON. Adicionalmente, cada instrução define um mnemônico em *assembly* que pode ser utilizado para codificação em baixo nível de um programa PON na forma textual. As seguintes regras são utilizadas para descrição ou composição de um mnemônico:

- Vírgulas são utilizadas para separação de cada campo do mnemônico.
- Campos com valor constante e literal são escritos em letras maiúsculas e estilo normal.
- Campos que denotam uma variável ou valor numérico são escritos em letras maiúsculas e estilo itálico.
- Caracteres entre colchetes ([ ]) no mnemônico denotam campos opcionais, que podem ser suprimidos porém mantendo-se as vírgulas separadoras. Caso sejam suprimidos, assume-se que o valor do campo correspondente é 0.

As tabelas de codificação das instruções, apresentadas nas seções a seguir, possuem 1 linha superior na qual se apresenta o tamanho em bits de cada campo e 1 linha inferior na qual se apresenta o código de cada campo que é posteriormente detalhado na forma de texto.

4.2.4.3.1 Instrução ATTRIBUTE-DECL

Mnemônico:

ATTRIBUTE-DECL, [E], [D],[N],[R], *NP,Value,[*AP_0*,*AP_1*,...,*AP_{NP-1}*]*

Codificação:

3	1	1	1	1	9	16	32	NP*32		
001	E	D	N	R	Reserved	NP	Value	AP_0	AP_1	AP_{NP-1}

Onde:

E, D, N, R – *flags* (0 para desativado, 1 para ativado) que indicam o valor das propriedades de Exclusivo (E), Determinístico (D), Notificante (N) e Renotificante (R).

NP – número de *Premises* conectadas a este *Attribute*

Value – valor numérico do *Attribute.*

AP_0 a AP_{NP-1} – endereços (32 bits) de cada uma das *Premises* conectadas a este *Attribute.*

Observação 1: o *flag* Notificante (N), se desligado, indica que o *Attribute* é não notificante (impertinente), ou seja, alterações no valor deste *Attribute* não geram notificações.
Observação 2: o *flag* Renotificante (R), se ligado, indica que o *Attribute* gera notificações sempre que atualizado, independente se o seu valor foi alterado ou não.

4.2.4.3.2 Instrução PREMISE-OP

Mnemônico:
PREMISE-OP, [E], [D],[R],[I1], [I2], [*A/V*], *Op*, *NC,A/V AT1,A/V AT2,*[AC_0,AC_1,...,AC_{NC-1}]

Codificação:

2	1	1	1	1	1	4	2	1	1	1	1	7	8
00	E	D	R	I1	I2	Op	A/V	V1	V2	LR	INI	Reserved	NC

32	32	NC*32	
A/V AT1	A/V AT2	AC_0	AC_{NC-1}

Onde:

E, D, R – *flags* (0 para desativado, 1 para ativado) que indicam o valor das propriedades de Exclusivo (E), Determinístico (D) e Renotificante (R).

I1 e I2 – *flags (*0 para desativado, 1 para ativado) que indicam se o *Attribute* da esquerda é impertinente (I1) e/ou se o *Attribute* da direita é impertinente (I2) para esta *Premise.*

A/V – bits que indicam se os campos de A/V AT1 e AT2, respectivamente, devem ser tratados como endereços de *Attributes* (valor 0) ou valores constantes (valor 1).

V1 e V2 – bits indicando se os valores dos operandos desta *Premise* já foram inicializados, permitindo portanto o seu cálculo lógico (utilizado para a avaliação inicial da *Premise* durante a rotina de *startup,* ver Seção 4.2.5.8).

LR – *flag* indicando qual o último valor lógico calculado para esta *Premise* (ver Seção 4.2.5.7.1). Se for omitido assume-se que o seu valor é 0 (falso).

INI – *flag* indicando se esta *Premise* já foi inicializada (utilizado para a avaliação inicial da *Premise* durante a rotina de *startup,* ver Seção 4.2.5.8)

NC – número de *Conditions* conectadas a esta *Premise.*

A/V AT1 e A/V AT2 – endereços dos *Attributes* da esquerda e da direita, respectivamente (ou valor constante se o bit correspondente de A/V for 1).

AC_0 a AC_{NC-1} – endereços (32 bits cada) de cada uma das *Conditions* conectadas a esta *Premise.*

Op – código da operação relacional efetuada pela *Premise,* conforme a Tabela 8.

Tabela 8 – Códigos das operações relacionais efetuadas pelas *Premises*

Código	Mnemônico	Operação	Semântica (linguagem C)
0000	==	Igualdade	_opn1 == opn2_
0001	!=	Desigualdade	_opn1 != opn2_
0010	>	Maior que	_opn1 > opn2_
0011	<	Menor que	_opn1 < opn2_
0100	>=	Maior ou igual a	_opn1 >= opn2_
0101	<=	Menor ou igual a	_opn1 <= opn2_
0110	&	Mascarado != 0	_(opn1 & opn2) != 0_

Observação 1: _opn1_ e _opn2_ devem ser interpretados da seguinte forma:

Se A/V == 00: _opn1 = *(A/V AT1)_ e _opn2 = *(A/V AT2)_

Se A/V == 01: _opn1 = *(A/V AT1)_ e _opn2 = A/V AT2_

Se A/V == 10: _opn1 = A/V AT1_ e _opn2 = *(A/V AT2)_

Se A/V == 11: _opn1 = A/V AT1_ e _opn2 = A/V AT2_

Observação 2: o *flag* Renotificante (R), se ligado, indica que a *Premise* gera notificações sempre que notificada pelo(s) *Attribute(s)* pertinente(s), independente se o seu valor lógico foi alterado ou não.

4.2.4.3.3 Instrução CONDITION-OP

Mnemônico:

CONDITION-OP, [E], [D],[R],[S],*Prio, Op*, *NM,Add P/S1, Add P/S2,[*AM_0*,*AM_1*,...,*AM_{NM-1}*]*

Codificação:

2	1	1	1	1	4	4	2	1	1	1	1	1	3	8
01	E	D	R	S	Op	Prio	VP	P1	P2	LR	NP	US	Reserved	NM

32	32	NM*32	
Add P/S1	Add P/S2	AM_0	AM_{NM-1}

Onde:

E, D, R – *flags* (0 para desativado, 1 para ativado) que indicam o valor das propriedades de Exclusivo (E), Determinístico (D) e Renotificante (R).

S – *flag* (0 para desativado, 1 para ativado) que indica se esta é uma *SubCondition* (*Condition* cujo valor lógico é utilizado como argumento para uma outra *Condition*, o que permite encadeamento de operações lógicas).

Prio – valor indicativo da prioridade desta condição (pode ser utilizado para resolução de conflitos). O valor 0b1111 (0xF) indica a maior prioridade, ao passo que o valor 0b0000 indica a menor prioridade.

VP – bits indicando se o valor de P1 (bit 0) e de P2 (bit 1) são válidos (utilizados para a alocação e desalocação de *Conditions,* ver Seção 4.2.5.7.1).

P1 – bit indicando o último valor lógico notificado pela *Premise* da esquerda (utilizado para a alocação e desalocação de *Conditions,* ver Seção 4.2.5.7.1).

P2 – bit indicando o último valor lógico notificado pela *Premise* da direita (utilizado para a alocação e desalocação de *Conditions,* ver Seção 4.2.5.7.1).

LR – *flag* indicando qual o último valor lógico calculado para esta *Condition* (ver Seção 4.2.5.7.1).

NP – bit indicando qual das *Premises* (0 para P/S1 e 1 para P/S2) gerou a notificação que ativou o *fetch* desta instrução (utilizado para resolução de conflito, ver Seção 4.2.5.7.2).

US – bit ligado durante o *fetch* para indicar que a carga da *Condition* ocorreu durante o processo de resolução de conflito e, portanto, deve seguir o algoritmo correspondente (ver Seção 4.2.5.7.2).

NM – número de *Methods* conectados a esta *Condition* (caso o bit S esteja ativado, indica o número de *Conditions* que dependem desta *SubCondition*).

Add P/S1 e Add P/S2 – endereços das *Premises* ou *SubConditions* da esquerda e da direita, respectivamente.

AM_0 a AM_{NM-1} – endereços (32 bits cada) de cada um dos *Methods* conectadas a esta *Condition* (ou de *Conditions* conectadas a esta *SubCondition* se o bit S estiver ativado).

Op – código da operação lógica efetuada pela *(Sub)Condition,* conforme a Tabela 9.

Tabela 9 – Códigos das operações lógicas efetuadas pelas *Conditions*

<table>
<tr><th>Código</th><th>Mnemônico</th><th>Operação</th><th>Semântica (linguagem C)</th></tr>
<tr><td>0000</td><td>&&</td><td>AND</td><td>opn1 && opn2</td></tr>
<tr><td>0001</td><td>!(&&)</td><td>NAND</td><td>!(opn1 && opn2)</td></tr>
<tr><td>0010</td><td>||</td><td>OR</td><td>opn1 || opn2</td></tr>
<tr><td>0011</td><td>!(||)</td><td>NOR</td><td>!(opn1 || opn2)</td></tr>
<tr><td>0100</td><td>^</td><td>XOR</td><td>(opn1 && !opn2) || (!opn1 && opn2)</td></tr>
<tr><td>0101</td><td>!&&</td><td>NOT first AND sec</td><td>!opn1 && opn2</td></tr>
<tr><td>0110</td><td>&!&</td><td>first AND NOT sec</td><td>opn1 && !opn2</td></tr>
<tr><td>0111</td><td>!||</td><td>NOT first OR sec</td><td>!opn1 || opn2</td></tr>
<tr><td>1000</td><td>|!|</td><td>first OR NOT sec</td><td>opn1 || !opn2</td></tr>
<tr><td>1001</td><td>!</td><td>NOT</td><td>!opn1</td></tr>
<tr><td>1010</td><td>!!</td><td>BYPASS</td><td>opn1</td></tr>
</table>

Observação 1: *opn1* e *opn2* correspondem aos valores lógicos das *Premises/SubConditions* referenciadas por *Add P/S1* e *Add P/S2*, respectivamente.

Observação 2: as operações NOT e BYPASS são implementadas nas formas *opn1* !&& TRUE e *opn1* && TRUE. Ou seja, NOT é a operação lógica de negação e BYPASS é a negação da negação, ambas aplicadas somente à *Premise* da esquerda. Para ambas as operações, o campo *Add P/S2* é indiferente.

Observação 3: o *flag* Renotificante (R), se ligado, indica que a *Condition* gera notificações sempre que notificada por uma de suas *Premises*, independente se o seu valor lógico foi alterado ou não.

4.2.4.3.4 Instrução METHOD-OP

Mnemônico:

METHOD-OP, [E],[D],[MM],[K],[*A/V*],*Op*,[IC=*ITCOUNT*],*NDM,Add C/MM,A/V AT opn1, A/V AT opn2,Add AT res,[*ADM_0*,*ADM_1*,...,*ADM_{NDM-1}*]*

Codificação:

2	1	1	1	1	4	2	5	13	2
10	E	D	MM	K	Reserved	A/V	Op	ITCOUNT	NDM

32	32	32	32	NDM*32		
Add C/MM	A/V AT opn1	A/V AT opn2	Add AT res	ADM_0	ADM_1	ADM_{NDM-1}

Onde:

E, D – *flags* (0 para desativado, 1 para ativado) que indicam o valor das propriedades de Exclusivo (E) e Determinístico (D).

MM – *flag* (0 para desativado, 1 para ativado) que indica se este é um *Método Mestre.* Caso afirmativo, os campos NDM e ADM_0 a ADM_{NDM-1} indicam a quantidade e os endereços dos métodos dependentes que são notificados após a execução deste método mestre.

K – bit utilizado para implementar a estratégia de re-execução continuada de método (*keeper*). Ou seja, quando este bit está ligado o MP gera uma auto-notificação para re-execução deste método até que o valor lógico da *Condition* que o notificou inicialmente seja alterado para falso.

A/V – bits que indicam se os campos de A/V AT opn1 e A/V AT opn2, respectivamente, devem ser tratados como endereços de *Attributes* (valor 0) ou valores constantes (valor 1).

ITCOUNT - quando o *Method* atua sobre variáveis escalares (p. ex. *arrays*) de forma iterativa (ou seja, executando a mesma operação iterativamente sobre cada elemento consecutivo dos *arrays* de operando e gerando resultado sobre o *array* de resultado), ITCOUNT indica o número de iterações do *Method* a ser executado. Se for maior do que 1, o *Scheduler / Conflict solver* tenta escalonar ITCOUNT instâncias do *Method* nos MP para executar as ITCOUNT iterações em paralelo. Os endereços dos *Attributes* de operandos e resultado (*End AT opn1, End AT opn2* e *End AT res*, respectivamente) são utilizados como

base e automaticamente incrementados durante o *fetch* para referenciar os atributos corretos a serem utilizados por cada iteração. Para maiores detalhes ver Seção 4.2.5.7.1.

NDM – número de *Methods* dependentes conectados a este *Method* mestre.

Add C/MM – endereço da *Condition* ou *Method* mestre que notifica este *Method.*

A/V AT opn1 e A/V AT opn2 – endereços dos *Attributes* utilizados como operandos para a execução da operação do *Method* (ou valores constantes caso o bit correspondente em A/V seja 1).

Add AT res – endereço do *Attribute* utilizado para armazenar o resultado da execução da operação do *Method.*

ADM_0 a ADM_{NDM-1} – endereços (32 bits cada) de cada um dos *Methods* dependentes da execução deste *Method* mestre (se o bit MM estiver ativado).

Op – código da operação a ser efetuada pelo *Method,* conforme a Tabela 10.

Tabela 10 – Códigos das operações efetuadas pelos *Methods*

<table>
<tr><th>Código</th><th>Mnemônico</th><th>Operação</th><th>Semântica (linguagem C)</th></tr>
<tr><td>00000</td><td>=</td><td>Atribuição</td><td>res = opn1</td></tr>
<tr><td>00001</td><td>&</td><td>Bitwise AND</td><td>res = opn1 & opn2</td></tr>
<tr><td>00010</td><td>!&</td><td>Bitwise NAND</td><td>res = ~(opn1 & opn2)</td></tr>
<tr><td>00011</td><td>|</td><td>Bitwise OR</td><td>res = opn1 | opn2</td></tr>
<tr><td>00100</td><td>!|</td><td>Bitwise NOR</td><td>res = ~(opn1 | opn2)</td></tr>
<tr><td>00101</td><td>^</td><td>Bitwise XOR</td><td>res =(~opn1 & opn2) | (opn1 & ~opn2)</td></tr>
<tr><td>00110</td><td>!^</td><td>Bitwise XNOR</td><td>res = (opn1 & opn2) | (~opn1 & ~opn2)</td></tr>
<tr><td>00111</td><td>~</td><td>Bitwise NOT</td><td>res = ~opn1</td></tr>
<tr><td>01000</td><td><<</td><td>Shift left</td><td>res = opn1 << opn2</td></tr>
<tr><td>01001</td><td>>></td><td>Shift right</td><td>res = opn1 >> opn2</td></tr>
<tr><td>01010</td><td>+</td><td>Adição com sinal</td><td>res = opn1 + opn2</td></tr>
<tr><td>01011</td><td>-</td><td>Subtração com sinal</td><td>res = opn1 – opn2</td></tr>
<tr><td>01100</td><td>*+</td><td>Multiplicação com sinal (word mais significativa)</td><td>res = (opn1 * opn2) >> 32</td></tr>
<tr><td>01101</td><td>*-</td><td>Multiplicação com sinal (word menos significativa)</td><td>res = (opn1 * opn2) & 0xFFFFFFFF</td></tr>
</table>

01110	/	Divisão com sinal	*res = opn1 / opn2*
01111	RDATREF	Leitura de referência a partir de *Attribute*	*res = *(opn1 + opn2)*
10000	WRATREF	Escrita em referência a partir de *Attribute*	**(Add AT res) = *(opn1 + opn2)*
10001	SETMEMREF	Alteração de bits em memória para '1'	**(Add AT res) = *(A/V AT opn1) | opn2*
10010	CLRMEMREF	Alteração de bits em memória para '0'	**(Add AT res) = *(A/V AT opn1) & ~opn2*
10011	RDMEMREF	Leitura de posição de memória	*res = *(A/V AT opn1)*
10100	WRMEMREF	Escrita em posição de memória	**(Add AT res) = opn1*
10101	INC	Incremento	*res = opn1 + 1*
10110	DEC	Decremento	*res = opn1 - 1*

Observação 1: *opn1* e *opn2* devem ser interpretados da seguinte forma:

Se A/V == 00: *opn1 = (*(A/V AT opn1)).valor* e *opn2 = (*(A/V AT opn2)).valor*

Se A/V == 01: *opn1 = (*(A/V AT opn1)).valor* e *opn2 = A/V AT opn2*

Se A/V == 10: *opn1 = A/V AT opn1* e *opn2 = (*(A/V AT opn2)).valor*

Se A/V == 11: *opn1 = A/V AT opn1* e *opn2 = A/V AT opn2*

res deve ser interpretado como: *(*(Add AT res)).valor*

Observação 2: operações sobre as propriedades dos *Attributes, Premises, Conditions* e *Methods* podem ser implementadas por meio das operações RDMEMREF, WRMEMREF, SETMEMREF, CLRMEMREF, RDATREF e WRATREF.

Observação 3: nas operações em que *opn2* não é utilizado o valor do campo *A/V AT opn2* é indiferente.

4.2.4.3.5 Instrução METHOD-VN-OP

Mnemônico:
METHOD-VN-OP, *Add MVN, Arg*

Codificação:

2	30	32
11	Add MVN (30 bits MSB)	Arg

Onde:

Add MVN – 30 bits mais significativos do endereço do método von Neumann a ser disparado (na memória de métodos von Neumann).

Arg – valor numérico a ser passado como argumento para a execução do método von Neumann.

Observação: o início do método von Neumann, na memória de métodos von Neumann, deve estar contido em um endereço múltiplo de 4 bytes (alinhamento em 32 bits), visto que os 2 bits menos significativos do endereço são preenchidos com 0 por *default.*

4.2.4.4 Limitações e restrições

Para esta proposta de ISA para a ARQPON v1.0 são consideradas as seguintes limitações e restrições:

- As operações implementadas pelos *Methods* suportam somente cálculos com aritmética de números inteiros para simplificar a arquitetura interna dos MP. Isto se constitui inicialmente em uma limitação funcional, pois segundo Kogge *et al* (2008) as operações de ponto fixo geralmente predominam no domínio de sistemas embarcados em função da característica predominante de processamento de dados fornecidos por estímulos (sensores) externos, que geralmente oferecem precisão limitada. No entanto, a implementação com números inteiros pode ser posteriormente estendida para ponto fixo ou até mesmo ponto flutuante sem

prejuízo para o modelo conceitual da ARQPON (ou seja, concentrando a sua execução nos *Methods*).

- Conforme a definição das instruções apresentada na Seção 4.2.4.3, os *Attributes* e *Premises* limitam o número máximo de elementos PON notificados a 2^{16} (65536) e as *Conditions* limitam este número a 2^{8} (256). Embora esta seja uma limitação de implementação do metamodelo de notificações do PON, dado que este não limita conceitualmente a cardinalidade das relações que viabilizam as notificações, as quantidades limite apresentadas devem ser suficientes para a maioria das aplicações. Além disso, um elemento do PON que necessite notificar mais elementos do que a quantidade limite pode ser replicado, abrindo então a possibilidade de multiplicar o número de elementos que podem ser efetivamente notificados.
- Conforme explanado na Seção 4.2.3.5, a ARQPON não implementa informação de contexto. Sendo assim, qualquer aplicação elegível para execução em uma implementação da ARQPON deve definir todos os seus FBEs em um contexto (escopo) global, com ciclo de vida igual ao ciclo de vida da aplicação. As instruções da ISA da ARQPON v1.0, portanto, não possuem qualquer campo que permita codificar uma informação específica de contexto, com exceção de METHOD-VN-OP que define um argumento para o método von Neumann que poderia, eventualmente, ser utilizado com aquela finalidade.

4.2.5 Definição da microarquitetura

4.2.5.1 Visão geral

A Figura 38 apresenta o diagrama em blocos arquiteturais da visão geral da ARQPON. Para fins de simplificação do texto, referir-se-à aos diversos barramentos ou redes de interconexão como *Redes,* embora continuem válidas as considerações descritas na Seção 4.2.3.2 sobre as possíveis topologias de interconexão.

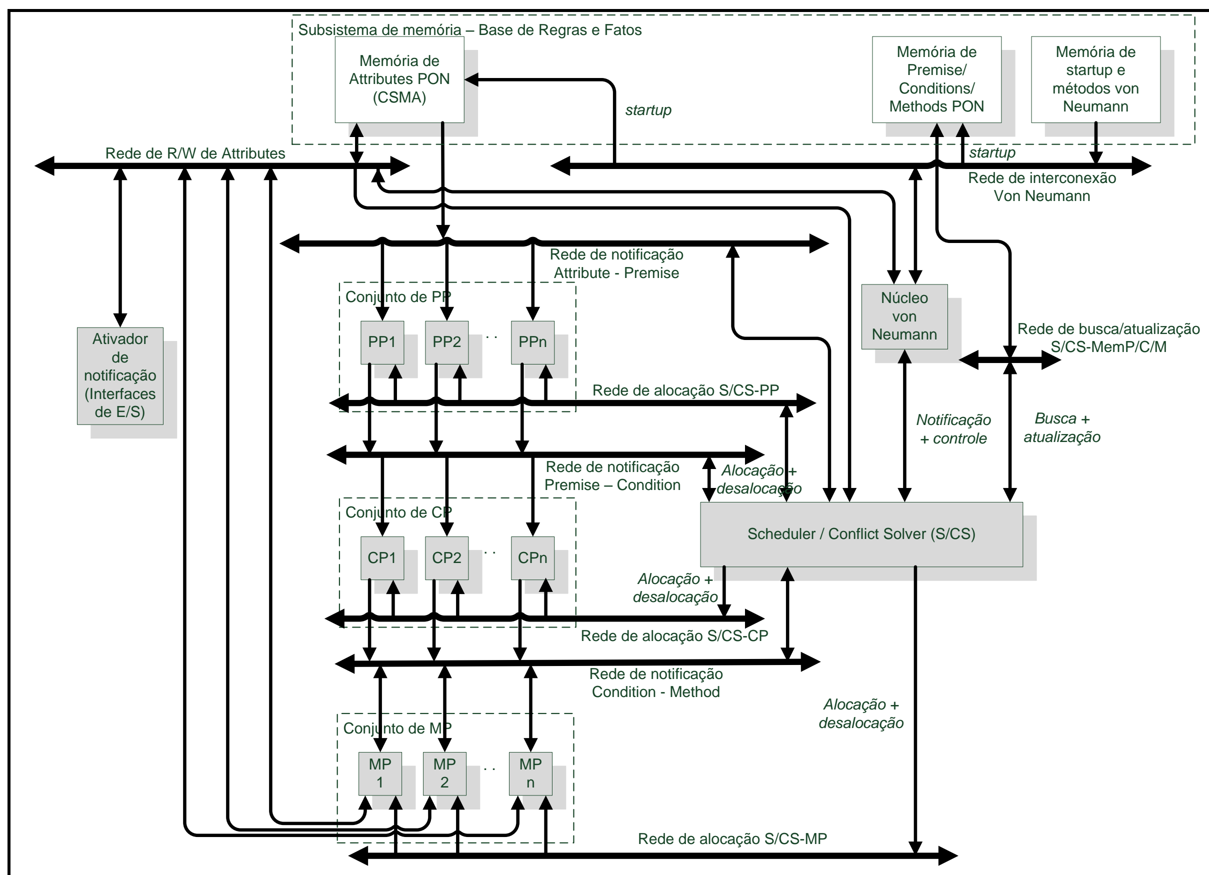


Figura 38 – Blocos arquiteturais da ARQPON

(Fonte: autoria própria)

Os conjuntos de blocos MP, PP e CP mapeiam, respectivamente, para os processos de execução de *Methods, Premises* e *Conditions* do modelo lógico (Figura 35). Cada um dos blocos MP, PP e CP, por sua vez, representa um módulo processador do elemento respectivo da cadeia de notificações do PON.

Os blocos que representam espaços de memória, apresentados na parte de cima da figura, são agrupados como *Subsistema de memória – Base de Regras e Fatos* pois têm por objetivo armazenar as definições de *Attributes* e instruções correspondentes a *Premises, Conditions* e *Methods* PON. Além disso, inclui-se a região de memória reservada para os métodos implementados segundo o modelo de von Neumann e também para as rotinas de *startup* (ver Seção 4.2.5.8).

O bloco *Scheduler / Conflict Solver* mapeia para o processo correspondente no modelo lógico. A partir deste ponto, referir-se-à a este bloco pela sigla *S/CS.*

A separação das memórias em três grandes blocos está relacionada aos seguintes fatores:

- Semântica de acesso aos *Attributes* é diferenciada (segue o modelo CSMA proposto na Seção 4.2.3.6).
- Propõe-se implementar políticas de *caching* para *Attributes* e para *Premises/Conditions/Methods,* visto que a busca destes elementos em memória é uma etapa do processo de propagação de notificação que pode ser muito custosa em função da velocidade de acesso à memória principal ("*memory wall*"). Além disso, embora não detalhado nesta figura, os barramentos conectados a estas duas *caches* são distintos, visto que a *Rede de notificação Attribute-Premise* é responsável por propagar notificações de alteração de *Attributes* e a *Rede de busca/atualização S/CS-MemP/C/M* é utilizado pelo *Scheduler / Conflict solver* para busca destes elementos em memória e posterior alocação aos núcleos de processamento correspondentes.
- A memória de métodos von Neumann pode ser interconectada a um barramento distinto dos demais barramentos do S/CS para viabilizar a utilização de um padrão de barramento conhecido (p. ex. Avalon) no interfaceamento com o núcleo von Neumann.

As subseções a seguir detalham a arquitetura interna dos blocos apresentados e a configuração dos barramentos / redes de interconexão.

4.2.5.2 Subsistema de memória

A Figura 39 apresenta o detalhamento dos blocos estruturais correspondentes ao subsistema de memória da ARQPON.

Em função das características dos barramentos de interconexão (Seção 4.2.5.3), todos os acessos de memória possuem largura de 32 bits. Sendo assim, o endereçamento da memória pode ser implementado tanto com endereços distintos para cada *byte* como com endereços distintos para cada *word* de 32 bits.

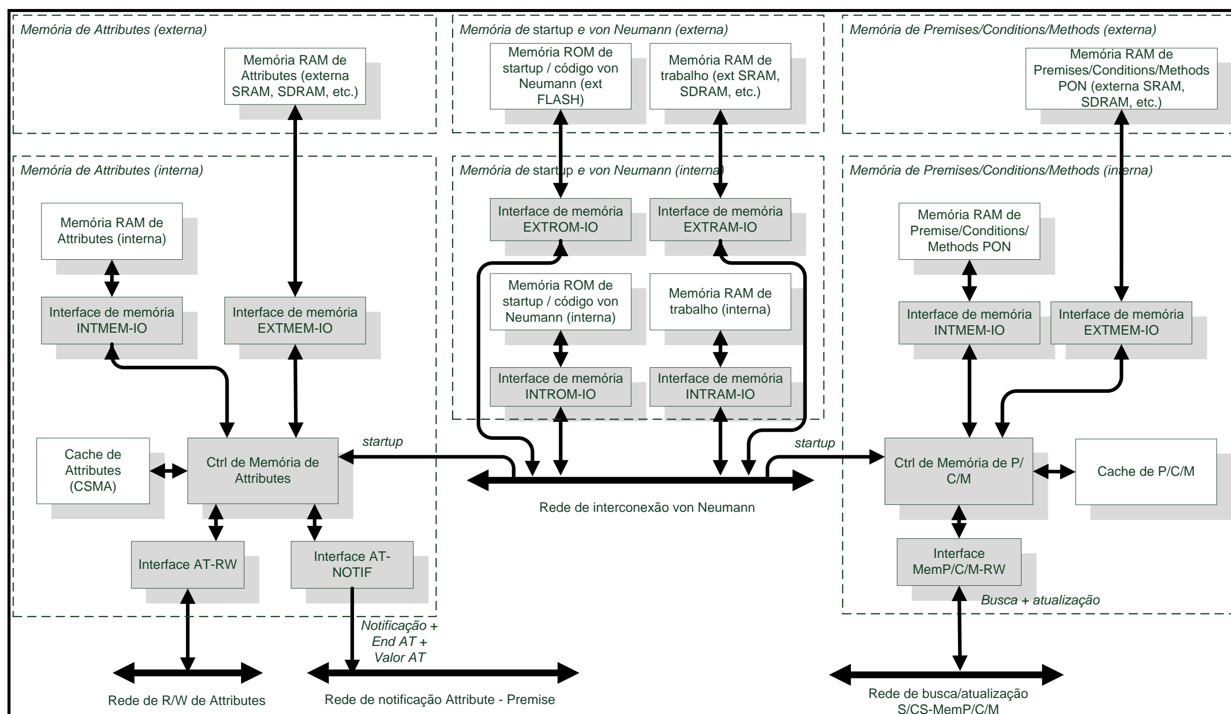


Figura 39 – Subsistema de memória da ARQPON

(Fonte: autoria própria)

4.2.5.2.1 Memória de Attributes

O agrupamento denominado *Memória de Attributes* corresponde aos blocos funcionais e memórias que compõem o repositório de *Attributes* conforme o modelo lógico, e que implementam a lógica de atualização dos *Attributes* e geração de notificação.

O bloco *Memória RAM de Attributes* corresponde ao dispositivo de armazenamento de dados propriamente dito. Este pode ser interno ao *chip* ou externo, dependendo neste último caso de uma implementação de *Interface de Memória* (EXTMEM-IO) compatível com o padrão de interface da memória externa (controlador SRAM, SDRAM, etc.). A implementação com memória externa atende ao requisito RIF-01 (ver Seção 4.1.3).

O bloco *Ctrl de memória de Attributes* é responsável por implementar a lógica de funcionamento da *Cache de Attributes* e as transações de escrita/leitura realizadas com a *Memória de Attributes* via *Interface de Memória.* Além disso, a interface deste bloco com a *Rede de interconexão von Neumann* viabiliza a configuração inicial dos *Attributes* efetuada pelo núcleo von Neumann executando a etapa de *startup* (ver Seção 4.2.5.8).

O bloco *Cache de Attributes (CSMA)* implementa uma *cache* de nível 1 para os *Attributes* e também a lógica de acesso sensível a alteração (CSMA), que permite otimizar a verificação de alteração de *Attribute* e geração de notificação (ver Seção 4.2.3.6). As notificações, quando ocorrem, são encaminhadas via *Interface AT-NOTIF* para a *Rede de notificação Attribute-Premise,* juntamente com o endereço do *Attribute* e o seu valor atualizado, ficando disponíveis para consumo pelas *Premises* via *snooping* (ver Seção 4.2.3.2).

O bloco *Interface AT-RW* é utilizado como interface para operações de leitura e escrita de *Attributes* que são encaminhadas via *Rede de R/W de Attributes.* Estas operações são geradas pelos blocos que dependem da leitura dos valores dos *Attributes* (MP) e pelos blocos que atualizam valores de *Attributes* (MP, *Núcleo von Neumann* e *Interfaces de E/S*).

4.2.5.2.2 Memória de startup e métodos von Neumann

O bloco correspondente à *Memória de startup e métodos von Neumann* representa os dispositivos físicos que armazenam o seguinte conteúdo:

- *Setup:* corresponde à parte da aplicação PON formada por declarações de *Attributes* (ATTRIBUTE-DECL) e operações de *Premises, Conditions* e *Methods* (PREMISE-OP, CONDITION-OP, METHOD-OP e METHOD-VN-OP), os quais estão gravados em memória não volátil e precisam ser inicializados nas respectivas memórias RAM. A rotina de *startup* é descrita em detalhes na Seção 4.2.5.8.
- *Métodos von Neumann:* correspondem ao código binário das funções que são implementadas segundo o modelo von Neumann e disparadas pela cadeia de notificações da aplicação PON, também armazenados em memória não volátil mas eventualmente podendo ser copiados para a *Memória RAM de trabalho* durante a inicialização. Esta funcionalidade está descrita na Seção 4.2.3.8.
- *Memória de trabalho:* corresponde à área de memória RAM (pilha) utilizada pelos métodos von Neumann para alocação de dados locais (exceto aqueles que representam o papel de *Attributes*) e empilhamento de chamadas de funções.

As *Interfaces de Memória* para este bloco estão conectadas à *Rede de operação von Neumann.* Isto possibilita que o *Núcleo von Neumann* busque os métodos von Neumann para execução e acesse os seus dados locais, bem com acesse os dados de *startup* e efetue a sua

inicialização tanto na *Memória de Attributes* quanto na *Memória de P/C/M,* ambas as quais também estão conectadas ao referido barramento.

4.2.5.2.3 Memória de Premises / Conditions / Methods PON

O bloco correspondente à *Memória de Premises / Conditions / Methods PON* apresenta uma interface com dispositivo físico externo ao processador semelhante à da *Memória de Attributes.* Esta interface pode inclusive ser interligada ao mesmo dispositivo físico externo ao processador, o que viabilizaria a implementação da memória de elementos PON em um único *chip.* No entanto, esta abordagem tende a aumentar a latência dos acessos visto que ambos os tipos de acesso compartilhariam o mesmo meio físico.

A *Cache de P/C/M* é uma memória que apresenta uma semântica comum de memórias *cache* (detalhes sobre capacidade da *cache,* política de substituição e largura de blocos não fazem parte do escopo de definição da arquitetura), controlada a partir do *Ctrl de memória de P/C/M.* Este controle, por sua vez, possui uma conexão com o S/CS via *Rede de busca/atualização S/CS – MemP/C/M,* por meio da qual este efetua as operações de busca (*fetch*) de instruções, para o processo de alocação nos respectivos processadores, e também a operação de persistência de valores de *Premises* conectadas a *Conditions* (para maiores detalhes sobre este último ver Seção 4.2.5.7.1).

De forma distinta do modelo de von Neumann, a dinâmica do fluxo de controle da ARQPON não implica instruções executadas repetidamente serem buscadas e alocadas continuamente nos estágios de execução do processador – uma vez que uma *Premise, Condition* ou *Method* é alocada no respectivo processador, somente precisará ser desalocada se o S/CS precisar daquele processador para a alocação de outra instrução. Esta abordagem tende a responder bem a questões de redundância temporal e estrutural e, portanto, também a questões de localidade temporal - ou seja, a manutenção das instruções nas respectivas unidades funciona como uma forma de *caching* de primeiro nível (L1). Desta forma, pode-se afirmar que a *Cache de P/C/M* é uma *cache* de nível 2 (L2), sendo opcional a sua implementação.

4.2.5.3 Barramentos / redes de interconexão

A seguir apresenta-se uma descrição de características de cada um dos barramentos / redes de interconexão definidos na ARQPON v1.0. Todos os barramentos de dados e endereços de memória possuem 32 bits de largura, o que facilita a compatibilidade com as definições da ISA (Seção 4.2.4), em particular nas situações em que valores de dados e valores de endereços devem trafegar no mesmo barramento, tais quais nas interfaces utilizadas para propagação de notificações.

R/W de Attributes:
Largura: 32 bits dados + 32 bits endereço + controle
Topologia: multi-mestre (*S/CS, MPs, Núcleo von Neumann* e *Interfaces de E/S* com arbitragem centralizada).
Endereçamento: ponto a ponto
Descrição: utilizado para acessos de escrita/ leitura na memória de *Attributes.*

Operação von Neumann:
Largura: 32 bits dados + 32 bits endereço
Topologia: mestre único (*Núcleo von Neumann*)
Endereçamento: ponto a ponto
Descrição: utilizado para acesso de *startup* e busca de instruções pelo *Núcleo von Neumann.*

Attribute-Premise:
Largura: 32 bits
Topologia: multi-mestre (*Ctrl de Memória de Attributes* e *S/CS* com arbitragem centralizada).
Endereçamento: modo *snooping* pelos escravos.
Descrição: utilizado para propagação de notificações entre a *Memória de Attributes* e os PPs e o *S/CS,* bem como para reenvio de notificação pelo *S/CS* (ver Seção 4.2.5.7).

Premise-Condition:
Largura: 32 bits
Topologia: multi-mestre (PPs, CPs e *S/CS* com arbitragem centralizada).
Endereçamento: modo *snooping* pelos escravos.

Descrição: utilizado para propagação de notificações entre os PPs e os CPs e o *S/CS,* bem como para reenvio de notificação pelo *S/CS* (ver Seção 4.2.5.7) e para envio de notificação de CPs com *SubConditions* para outros CPs.

Condition-Method:

Largura: 32 bits

Topologia: multi-mestre (CPs, MPs e *S/CS* com arbitragem centralizada).

Endereçamento: modo *snooping* pelos escravos.

Descrição: utilizado para propagação de notificações entre os CPs e os MPs e o *S/CS,* bem como para reenvio de notificação pelo *S/CS* (ver Seção 4.2.5.7) e envio de notificação por MP configurado no modo *Master Method*.

S/CS-PP:

Largura: 32 bits dados + N bits endereço (dependendo da quantidade de PPs) + controle

Topologia: mestre único (*S/CS*).

Endereçamento: ponto a ponto.

Descrição: utilizado para alocação de *Premise* no respectivo PP.

S/CS-CP:

Largura: 32 bits dados + N bits endereço (dependendo da quantidade de CPs) + controle

Topologia: mestre único (*S/CS*).

Endereçamento: ponto a ponto.

Descrição: utilizado para alocação e desalocação de *Condition* no respectivo CP.

S/CS-MP:

Largura: 32 bits dados + N bits endereço (dependendo da quantidade de MPs) + controle

Topologia: mestre único (*S/CS*).

Endereçamento: ponto a ponto.

Descrição: utilizado para alocação de *Method* no respectivo MP.

S/CS-MemP/C/M:

Largura: 32 bits dados + 32 bits endereço

Topologia: mestre único (*S/CS*).

Endereçamento: ponto a ponto.

Descrição: utilizado para a busca de instruções de *Premise, Condition* e *Method* na respectiva memória pelo *S/CS* e também para a persistência de valores de *Premises* nas *Conditions* correspondentes.

4.2.5.4 Conjunto de PP (*Premise Processor*)

A Figura 40 apresenta a estrutura interna de um PP. Os blocos preenchidos na cor branca representam o conjunto de registradores implementados por um PP. Estes registradores armazenam as informações de relacionamento do PP com os *Attributes* que constituem os seus operandos, bem como o endereço da *Premise* alocada, o operador implementado pela *Premise*, os *flags* para indicação do funcionamento no modo *Exclusive* ou *Deterministic* (ver Seção 4.2.5.7) e controle de *Attributes* impertinentes e operandos por valor ou endereço, os valores atuais dos operandos e o valor atual do resultado lógico da operação relacional; estes dois últimos, em particular, permitem verificar a necessidade da geração de notificação, em função de alteração no resultado lógico da operação da *Premise.*

A *Interface de PP-NOTIF_IN* ligada à *Rede de notificação Attribute-Premise* é responsável pela operação de *snooping* neste barramento, de tal forma que o PP possa detectar se uma notificação gerada por um *Attribute* é pertinente para esta *Premise.* Para tanto, o endereço do *Attribute* propagado e os bits I1 e I2 do *opcode* da instrução PREMISE-OP são levados em consideração (ver Seção 4.2.4.3.2).

A *Interface de PP-ALLOC* ligada à *Rede de alocação S/CS-PP* é responsável por consumir e decodificar o *opcode* da PREMISE-OP, buscado pelo *S/CS* e direcionado para este PP.

A *Interface de PP-NOTIF_OUT* ligada à *Rede de notificação Premise-Condition* é responsável por gerar uma eventual notificação de alteração no resultado lógico da *Premise,* propagando este resultado para as *Conditions* de interesse via barramento citado. Vale ressaltar que o resultado lógico da *Premise* somente é calculado após ambos os operandos estarem carregados e disponíveis no PP.

As conexões ou sinais necessários para controle e sincronização dos blocos funcionais do PP não são mostradas explicitamente na figura.

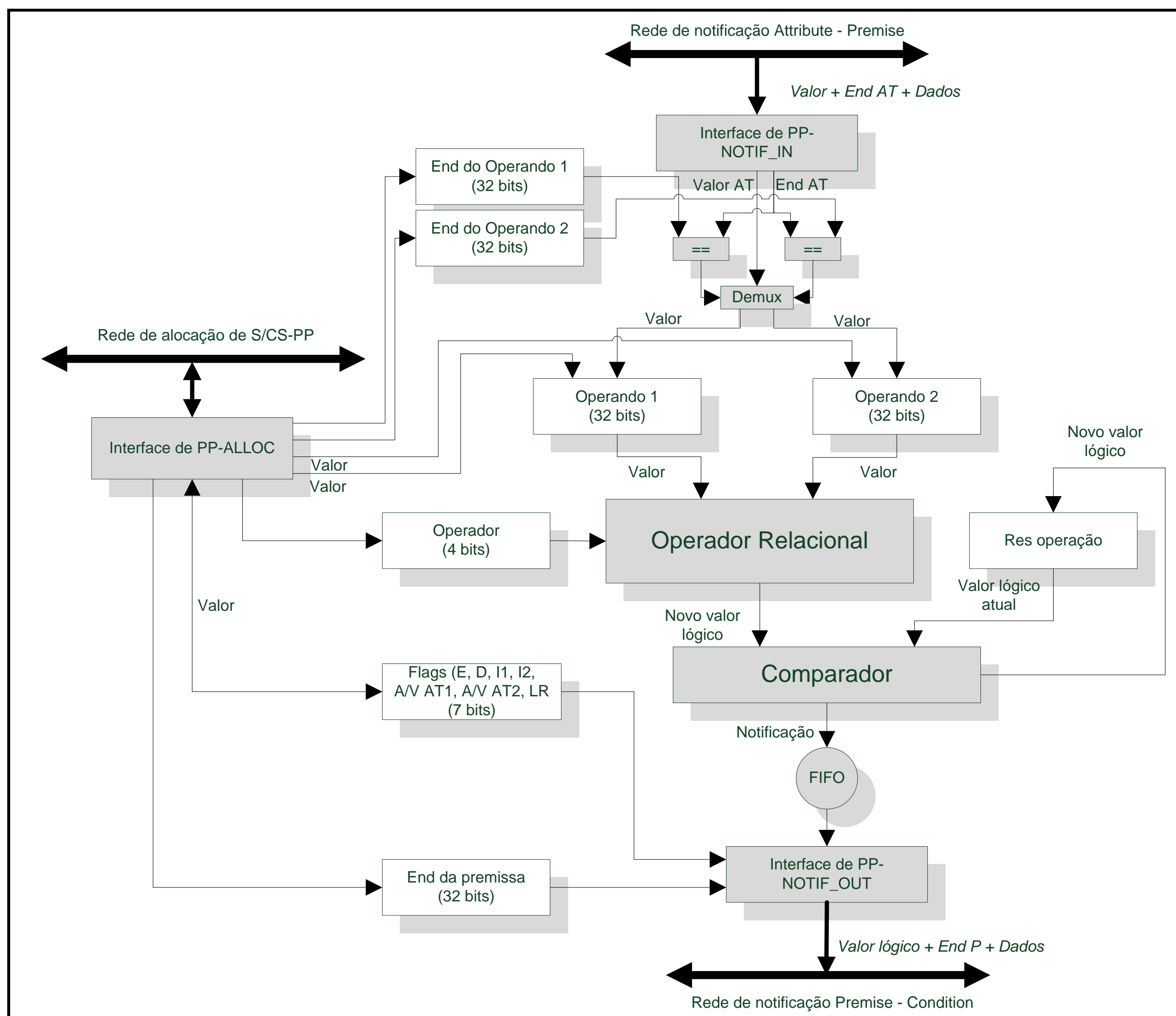


Figura 40 – Estrutura interna de um PP
(Fonte: autoria própria)

4.2.5.5 Conjunto de CP (*Condition Processor*)

A Figura 41 apresenta a estrutura interna de um CP. Os blocos preenchidos na cor branca representam o conjunto de registradores implementados por um CP. Estes registradores armazenam as informações de relacionamento do CP com as *Premises* que constituem os seus operandos, bem como o endereço da *Condition* alocada, o operador lógico implementado pela *Condition*, os *flags* para indicação do funcionamento no modo *Exclusive* ou *Deterministic* (ver Seção 4.2.5.7) ou se é *SubCondition*, o valor da prioridade para resolução de conflitos (ver Seção 4.2.5.7.2), os valores atuais dos operandos e o valor lógico anterior calculado para a *Condition*.

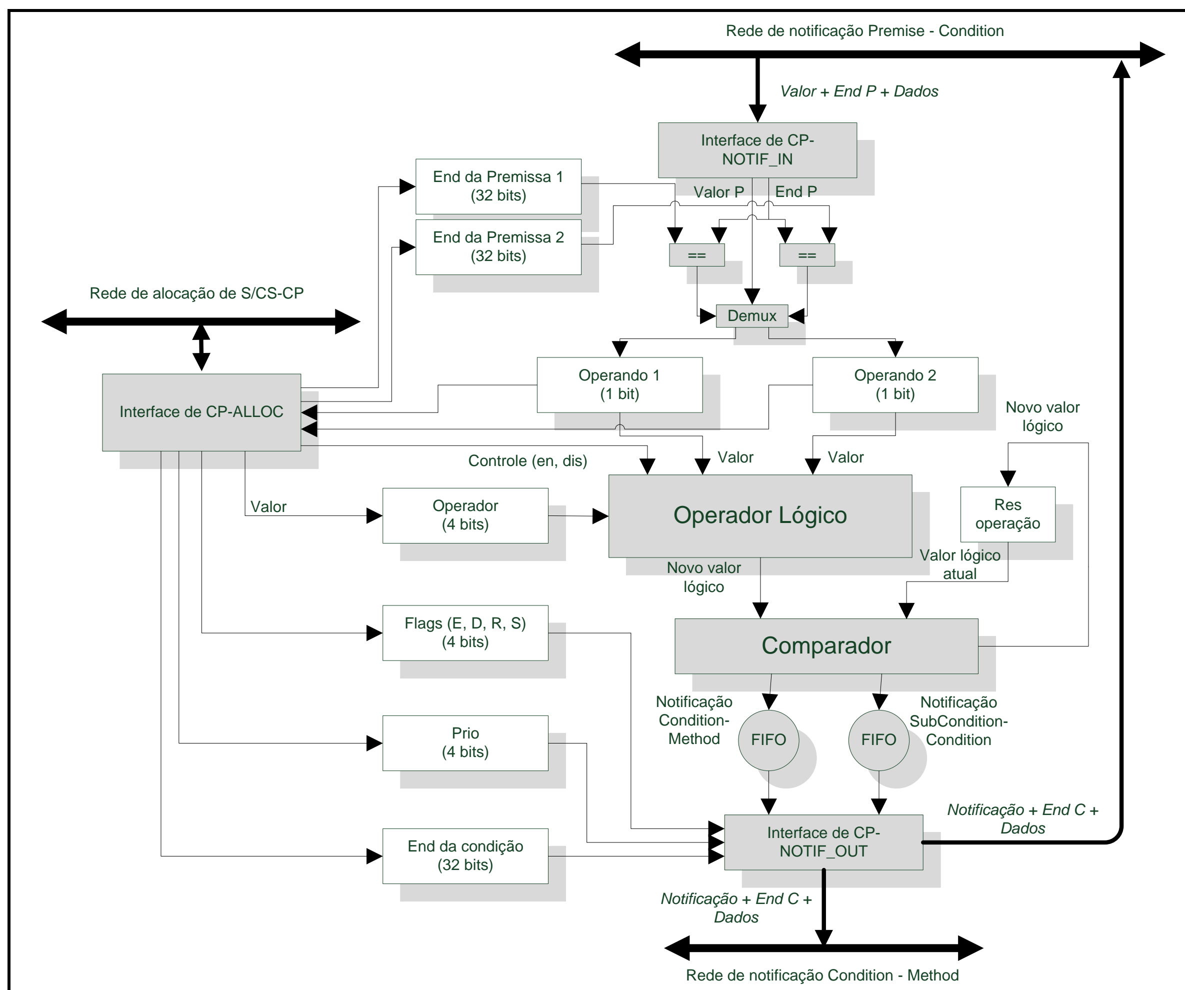


Figura 41 – Estrutura interna de um CP
(Fonte: autoria própria)

Uma vez que ambos os operandos estiverem carregados e disponíveis no CP, uma notificação recebida por ele dispara o cálculo do novo valor lógico da *Condition,* a partir dos valores atuais dos operandos e do operador. O resultado deste cálculo é então comparado ao valor lógico atual e, se for diferente, faz com que o CP gere e enfileire notificações para os *Methods* ou para outras *Conditions* (caso o flag de *SubCondition* esteja ativado).

A *Interface de CP-NOTIF_IN* ligada à *Rede de notificação Premise-Condition* é responsável pela operação de *snooping* neste barramento, de tal forma que o CP possa detectar se uma notificação gerada por uma *Premise* é pertinente para esta *Condition.* Para tanto, o endereço da *Premise* propagada é levado em consideração.

A *Interface de CP-ALLOC* ligada ao *Rede de alocação S/CS-CP* é responsável por consumir e decodificar o *opcode* da CONDITION-OP, buscado pelo *S/CS* e direcionado para este CP. Além disso, por meio desta interface ocorre o processo de desalocação de *Condition*,

durante o qual os valores lógicos atuais dos operandos desta *Condition* são enviados ao *S/CS* para serem posteriormente persistidos na *Memória de P/C/M*.

A *Interface de CP-NOTIF_OUT* ligada à *Rede de notificação Condition-Method* é responsável por propagar as notificações enfileiradas para os *Methods* de interesse via barramento citado. Além disso, esta interface também é ligada à *Rede de notificação Premise-Condition* para propagar notificações de *SubConditions* para as *Conditions* que delas dependem.

As conexões ou sinais necessários para controle e sincronização dos blocos funcionais do CP não são mostradas explicitamente na figura.

4.2.5.6 Conjunto de MP (*Method Processor*)

A Figura 42 apresenta a estrutura interna de um MP. Os blocos preenchidos na cor branca representam o conjunto de registradores implementados por um MP. Estes registradores armazenam as informações de relacionamento do MP com a *Condition* ou com o *Master Method* que dispara a sua execução, bem como o endereço do *Method* alocado, a operação implementada pelo *Method*, os endereços/valores dos operandos e o endereço do *Attribute* que recebe o resultado, os *flags* para indicação do funcionamento no modo *Exclusive* ou *Deterministic* (ver Seção 4.2.5.7) e para indicar se é *Master Method* ou não, e registradores auxiliares para armazenamento temporário dos valores dos operandos para utilização pela ULA.

A *Interface de MP-NOTIF_IN* ligada à *Rede de notificação Condition-Method* é responsável pela operação de *snooping* neste barramento, de tal forma que o MP possa detectar se uma notificação gerada por uma *Condition* é pertinente para este *Method.* Para tanto, o endereço da *Condition* propagada é levado em consideração.

A *Interface de MP-ALLOC* ligada à *Rede de alocação de S/CS-MP* é responsável por consumir e decodificar o *opcode* do METHOD-OP, buscado pelo *S/CS* e direcionado para este MP. Vale ressaltar que instruções do tipo METHOD-VN-OP não são enviadas aos MP, porém direcionadas pelo *S/CS* para execução pelo *Núcleo von Neumann* (ver Seção 4.2.3.8 para maiores detalhes).

A *Interface de MP-MM_NOTIF_OUT* ligada à *Rede de notificação Condition-Method* é responsável por gerar notificação de fim de execução caso o método alocado no MP seja um *Master Method* ou o flag K esteja ligado. Esta notificação poderá então ser

consumida (via *snooping*) por métodos dependentes deste método ou para re-execução dele próprio se o flag K estiver ligado.

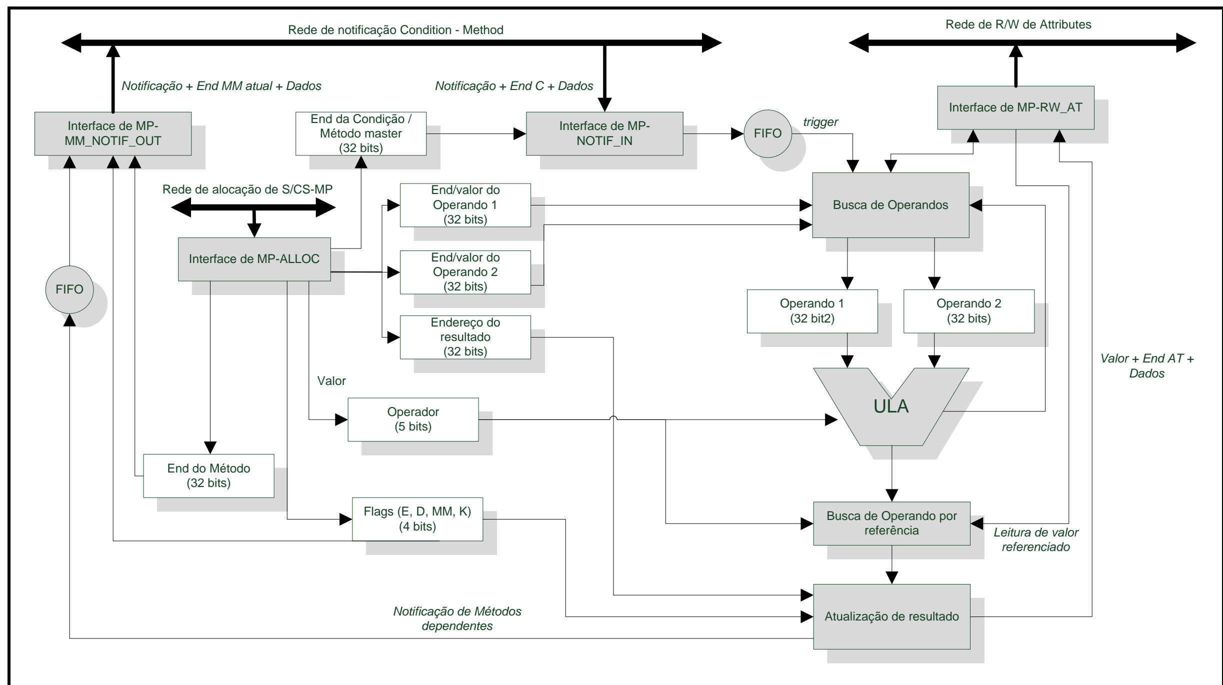


Figura 42 – Estrutura interna de um MP

(Fonte: autoria própria)

As conexões ou sinais necessários para controle e sincronização dos blocos funcionais do MP não são mostradas explicitamente na figura.

4.2.5.7 *Scheduler / Conflict Solver* (S/CS)

A Figura 43 apresenta a estrutura interna do bloco de *S/CS.* O S/CS é dividido internamente em três subseções funcionais, cada uma delas responsável pela tarefa de escalonamento de um dos conjuntos de elementos do PON para seus respectivos processadores (*Premises* para PPs, *Conditions* para CPs e *Methods* para MPs). Cada uma das seções possui um bloco de *hardware* dedicado ao processamento do algoritmo de alocação para aquela seção (*P Scheduler, C Scheduler* e *M Scheduler,* respectivamente).

Além disso, cada subseção define blocos de *hardware* dedicados ao interfaceamento com os barramentos/redes por onde trafegam notificações (*Interface de SCS-NOTIF_AP, Interface de SCS-NOTIF_PC* e *Interface de SCS-NOTIF_CM*, respectivamente) e blocos de

hardware dedicados ao interfaceamento com os barramentos/redes por onde são enviados os *opcodes* para cada uma das unidades de processamento (*Interface de SCS-ALLOC_PP, Interface de SCS-ALLOC_CP* e *Interface de SCS-ALLOC_MP,* respectivamente). As três subseções ainda compartilham um bloco de *hardware* dedicado ao interfaceamento com o barramento ao qual está conectada a memória de *Premises, Conditions* e *Methods* (*Interface de SCS-RW_MemP/C/M*).

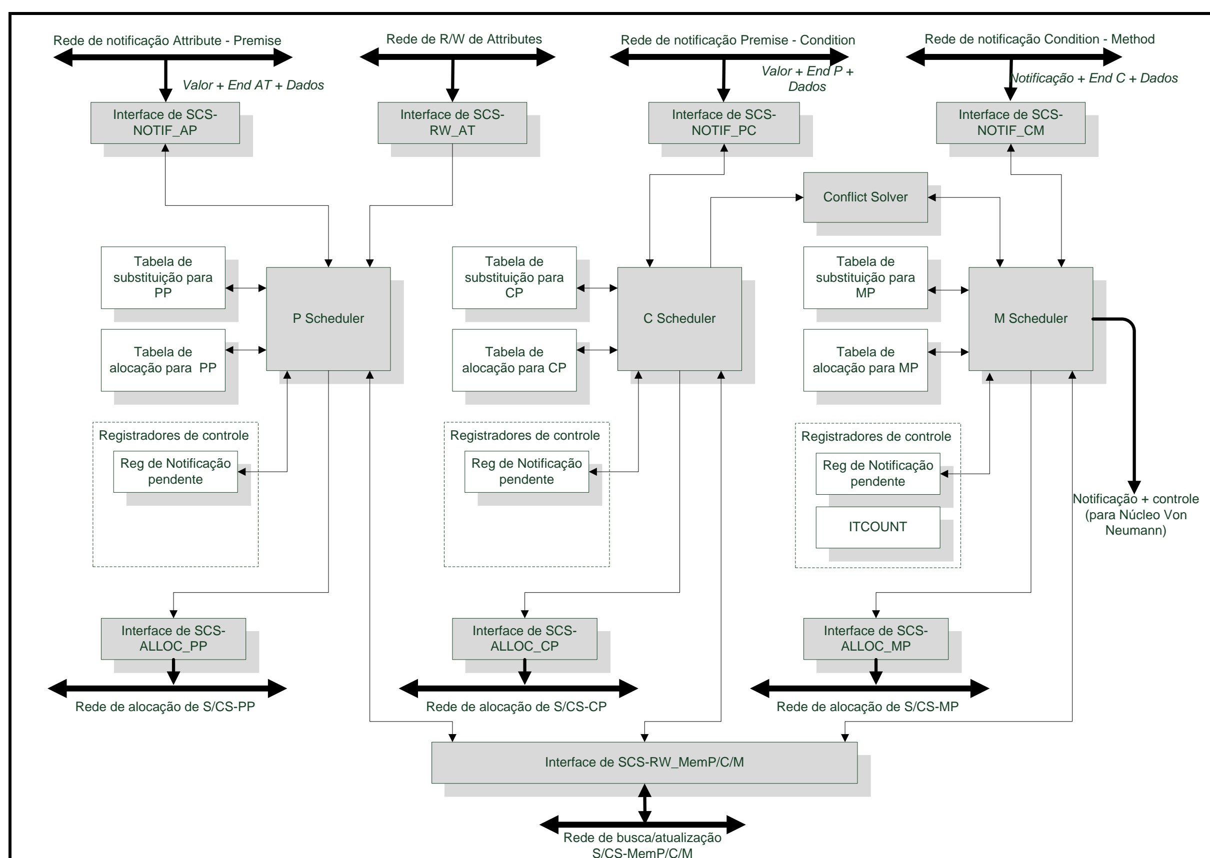


Figura 43 – Estrutura interna do S/CS
(Fonte: autoria própria)

Os blocos preenchidos na cor branca representam registradores e tabelas utilizadas pelo S/CS. Em particular, o grupo de *Registradores de Controle* define explicitamente o *Registrador de Notificação Pendente,* que é utilizado como registrador auxiliar pelo algoritmo de alocação (ver Seção 4.2.5.7.1). Além disso, a subseção responsável pelos MPs também define explicitamente o registrador *ITCOUNT* como auxiliar para a lógica de execução paralela de operações iterativas em MP (ver Seção 4.2.5.7.1).

As tabelas denominadas *Tabela de substituição* e *Tabela de alocação,* para cada um dos grupos de PPs, CPs e MPs, indicam estruturas de dados internas ao S/CS que armazenam informações utilizadas para implementação das políticas de substituição e alocação (ver Seção 4.2.5.7.1).

O bloco *P Scheduler,* em particular, define a *Interface S/CS-RW_AT* com a *Rede de R/W de Attributes.* Esta interface é utilizada no processo de realocação de uma *Premise* em seu PP, de tal maneira a recuperar o valor atual do *Attribute* não gerador da notificação que motivou a alocação (para maiores detalhes ver Seção 4.2.5.7.1).

O bloco *M Scheduler*, em particular, define uma interface com o *Núcleo von Neumann*. Esta interface é utilizada para o envio de notificação de início de execução de um determinado método pelo núcleo von Neumann, quando da execução da instrução METHOD-VN-OP pelo *M Scheduler.*

Os blocos *C Scheduler* e *M Scheduler* interfaceiam com o bloco *Conflict Solver.* Este bloco é responsável por executar o algoritmo de resolução de conflito, conforme apresentado na Seção 4.2.5.7.2.

4.2.5.7.1 Alocação para PP, CP e MP

O Algoritmo 1 define a sequência de operações executadas pelo S/CS para alocação de *Premises* em PP.

Algoritmo 1 – Alocação / desalocação de elementos do PON de/em processadores

---

Descrição: este algoritmo descreve as operações, realizadas pelo S/CS e pelos PP, para implementação do mecanismo de alocação e desalocação de *Premises*.

Funcionamento:

1: o S/CS recebe uma notificação de *Attribute* por meio da *Rede de notificação Attribute-Premise*.

2: o S/CS busca a instrução ATTRIBUTE-DECL respectiva na *Memória de Attributes.*

3: Se a próxima *Premise* conectada ao *Attribute* não está alocada a nenhum PP, o S/CS executa os seguintes sub-passos:

3.1 escolhe um PP para receber a *Premise*

3.2 desaloca a *Premise* alocada neste PP, recebe deste PP o valor lógico atual da *Premise* sendo desalocada e envia para este PP o código da *Premise* a ser alocada por meio da *Rede de alocação de S/CS-PP*.

3.3 envia para o PP escolhido, juntamente com a *Premise* recém-alocada, o valor do *Attribute* notificante (obtido no Passo 1) e o valor do outro *Attribute* envolvido na *Premise* (caso o outro operando não seja constante). Este último é buscado por meio de acesso à *Rede de R/W de Attributes*.

3.4 comanda no PP escolhido o início do processamento da notificação.

4: o S/CS repete o passo 3 e seus sub-passos para cada uma das *Premises* não alocadas.

Lógica semelhante à do Algoritmo 1 é utilizada para a alocação de *Conditions* em CP e de *Methods* em MP, com as seguintes diferenças:

- Para desalocação de uma *Condition* de um CP, o passo 3 implica também obter e armazenar o valor lógico atual de ambas as *Premises* que são utilizadas para esta *Condition* (ver Seção "Impacto da desalocação no mecanismo de notificações" adiante para uma explicação mais detalhada).

A política de alocação e substituição utilizada no passo 3 do Algoritmo 1, tanto para PP quanto para CP e MP, deve levar em consideração:

- Estado atual de alocação de cada processador (se está ou não alocado e para qual elemento da cadeia)
- Interdependências existentes entre elementos da cadeia (p. ex., *Conditions* que devem ser alocadas simultaneamente porque são notificadas pela mesma *Premise*).
- Informações que permitam estabelecer uma política de substituição (p. ex. tempo de alocação, para avaliação de localidade temporal).

Estas informações devem ser gerenciadas pelo S/CS utilizando-se as tabelas de substituição e de alocação para cada grupo de processadores, conforme apresentado na Figura 43. Uma proposta preliminar para a estrutura/conteúdo destas tabelas seria a seguinte:

- Tabela de substituição: N registros contendo o identificador numérico de um processador, sendo o arquivo classificado por tempo (do mais antigo para o mais novo). Esta estrutura permitiria uma política de substituição baseada em LRU

(*least recently used*). O número N de registros corresponde ao número de processadores disponíveis de cada tipo (NPP, NCP ou NMP).

- Tabela de alocação: N registros, organizados por identificador numérico do processador, contendo o endereço do elemento ao qual cada processador está alocado e *bits* auxiliares para a operação do S/CS (indicando, p. ex., se não deve ser desalocado em um processo de alocação por haver interdependências com outros elementos sendo alocados).

Em tese, um *Attribute* não deveria notificar um número de *Premises* maior do que o número de PPs implementados pelo P2ON, uma *Premise* não deveria notificar um número de *Conditions* maior do que o número de CPs implementados pelo P2ON e uma *Condition* não deveria notificar um número de *Methods* maior do que o número de MPs implementados pelo P2ON. Este conjunto de restrições se aplicaria para viabilizar que o S/CS pudesse alocar todos os elementos da cadeia notificados nos seus respectivos processadores, de tal maneira que a notificação propagada sensibilizasse todos os elementos pertinentes simultaneamente, conforme a lógica do programa PON. Estas restrições são complementares à restrição apresentada pela ISA da ARQPON v1.0 em relação ao tamanho da lista de elementos notificados por cada instrução (Seção 4.2.4.4), sendo válidas, portanto, as mesmas considerações do ponto de vista de adaptação das aplicações PON à restrição.

Caso alguma das restrições fosse violada, poder-se-ia considerar que o S/CS é incapaz de executar aquela operação de alocação, gerando um erro fatal de tempo de execução a ser tratado apropriadamente. Alternativamente, pode-se considerar a implementação de filas de elementos notificados pendentes, internas ao S/CS, uma para cada classe de elemento (P/C/M). Estas filas também sofreriam de restrições de tamanho, porém devido à sua implementação em memória a utilização de recursos de *hardware* seria menor do que o decorrente da adição de mais unidades de processamento. Além disso, o gerenciamento das filas, alocação dos elementos enfileirados e regeneração de notificações para estes elementos demandaria a implementação de lógica extra no S/CS.

Em relação a restrições de desalocação, nenhum elemento deve ser desalocado enquanto estiver sob operação de resolução de conflito (ver Seção 4.2.5.7.2).

Impacto da desalocação no mecanismo de notificações

Conceitualmente, tanto *Premises* quanto *Conditions* mantêm internamente cópias dos valores dos seus operandos (*Attributes* e *Premises/SubConditions*¸ respectivamente), com o intuito de otimizar a execução das relações lógico-causais que implementam. Esta otimização ocorre no sentido de que, quando um determinado *Attribute* que é operando de uma *Premise* envia uma notificação de alteração, o seu novo valor pode ser imediatamente comparado ao valor do outro *Attribute* envolvido na *Premise,* que está armazenado internamente a ela, sem a necessidade de que este outro valor seja buscado no objeto que implementa o *Attribute.* O mesmo raciocínio vale para *Conditions* em relação aos valores lógicos das *Premises* que lhe servem como operandos.

Como a desalocação de *Premises* e *Conditions* dos PPs e CPs poderia, consequentemente, também desalocar as cópias internas dos valores respectivos de *Attributes* e *Premises,* a ARQPON deve implementar uma estratégia de persistência destes valores. Em relação aos valores de *Attributes,* como estes já estão armazenados na *Memória de Attributes,* deve-se implementar uma estratégia de leitura deste valor quando da alocação de uma *Premise* no respectivo PP. Já os valores das *Premises* envolvidos em uma *Condition* devem ser persistidos à parte pois não se constituem em dados que já estão armazenados, mas sim no resultado lógico atual das operações relacionais implementadas pelas *Premises* envolvidas.

A ARQPON propõe resolver estes problemas da seguinte maneira:

- Ao desalocar uma *Premise* o S/CS obtém o seu valor lógico atual, armazenado internamente, via operação de leitura na *Rede de alocação de S/CS-CP.* Este valor é armazenado nos bits VP e LR da instrução PREMISE-OP correspondente na *Memória de P/C/M,* via acesso de escrita do S/CS na *Rede de busca / atualização S/CS-MemP/C/M.* Desta forma, quando necessitar realocar futuramente esta *Premise* o S/CS pode repassar ao PP também o valor calculado para o seu último resultado lógico e a sua validade (bit VP), evitando assim que uma notificação seja gerada desnecessariamente caso o resultado da *Premise,* em virtude da notificação recebida, não tenha sofrido alteração ou não seja válido.
- Ao alocar uma *Premise,* o S/CS deve buscar o valor do *Attribute* que não gerou a notificação motivadora da alocação. O S/CS efetua esta operação por meio de uma leitura na *Rede de R/W de Attributes* e fornece o valor atual do *Attribute* ao PP na operação de alocação efetuada via *Rede de alocação de S/CS-PP.*

- Ao desalocar uma *Condition*, o S/CS obtém o seu valor lógico atual e o valor atual das suas *Premises,* armazenados internamente, via operação de leitura na *Rede de alocação de S/CS-CP*. Estes valores são armazenados nos bits LR, P1 e P2 e no campo VP da instrução CONDITION-OP correspondente na *Memória de P/C/M,* via acesso de escrita do S/CS na *Rede de busca / atualização S/CS-MemP/C/M.* Desta forma, quando necessitar realocar futuramente esta *Condition* o S/CS pode repassar ao CP também o valor atual da *Premise* que não gerou a notificação motivadora da (re)alocação e o último valor lógico calculado para a *Condition*, evitando assim que uma notificação seja gerada desnecessariamente caso o resultado da *Condition,* em virtude da notificação recebida, não tenha sofrido alteração.

Execução paralela de operações iterativas em MP

O paralelismo oferecido pela multiplicidade de MPs da ARQPON v1.0 pode ser aproveitado para otimizar a execução de operações iterativas, semelhante ao que ocorre em arquiteturas superescalares ou VLIW por meio da emissão de múltiplas instruções para vários núcleos de processamento simultaneamente.

Para tanto, propõe-se implementar o contador *ITCOUNT*, no *opcode* da instrução *METHOD-OP* (ver Seção 4.2.4.3.4), cujo objetivo é indicar ao S/CS que se deseja alocar métodos para execução de uma operação iterativa no seguinte formato:

**(addrres + i) = *(addropn1 + i) OP *(addropn2 + i)*

Onde *addrres, addropn1* e *addropn2* são os endereços base dos conjuntos escalares (*arrays*) de atributos (resultado e operandos, respectivamente) sobre os quais se deseja executar a operação *OP*. O valor *i* é um inteiro representando o índice da iteração; este inteiro é iniciado com 0 e automaticamente incrementado pelo S/CS, durante o processo de alocação, de tal maneira que cada *Method* alocado a um MP referencie o *i*-ésimo *opn1,* o *i*-ésimo *opn2* e o *i*-ésimo *res,* viabilizando a execução da *OP* em paralelo para as *i* iterações.

O número de iterações executável simultaneamente é limitado ao número de MPs disponíveis. Caso o valor de *ITCOUNT* seja superior ao número de MPs, o S/CS define os registradores auxiliares *Registrador de Notificação Pendente* e *ITCOUNT* (Figura 43)*,* os quais armazenam a notificação disparadora da operação iterativa e o número restante de

iterações a ser executado, respectivamente. Isto viabiliza que a quantidade de iterações definidas seja efetuada em etapas, limitada a execução paralela à quantidade de MPs disponíveis.

Alternativamente, em uma implementação simplificada da ARQPON, os registradores citados podem ser omitidos e instruções *METHOD-OP* com *ITCOUNT* superior ao número de MPs gerariam um erro de tempo de execução.

A execução de operações iterativas em paralelo implementa uma semântica não estrita de acesso e modificação de *arrays,* dado que as alterações são efetuadas independentemente em partes distintas do *array* e propagadas de forma assíncrona entre si, via mecanismo de notificações.

4.2.5.7.2 Resolução de conflito

Conforme descrito por Simão e Stadzisz (2010), conflitos existentes entre *Rules* PON mutuamente exclusivas que porventura venham a ser aprovadas simultaneamente podem ser solucionados por meio de um mecanismo de resolução de conflitos. Para tanto, define-se que as *Conditions* das *Rules* mutuamente exclusivas possuem um atributo "exclusivo" (tornando-as *Exclusive-Conditions*), e que *Premises* que colaborem para a aprovação de duas ou mais *Exclusive-Conditions* também possuam um atributo "exclusivo" (o que as torna *Exclusive-Premises).* Ainda, cada *Exclusive-Premise* está relacionada a pelo menos um *Exclusive-Attribute,* que é componente de um *Exclusive-FactBase,* o qual é representado pelos elementos factuais potencialmente geradores de conflitos.

A ARQPON v1.0 realiza a resolução de conflitos por meio de operações a serem executadas pelo escalonador/resolutor de conflito (S/CS). Esta é uma solução prática, no âmbito da estrutura da ARQPON, pelo fato de que não depende da geração explícita de contra-notificações, conforme proposto conceitualmente por Simão e Stadzisz (2010).

De fato, a confirmação de uma única *Exclusive-Condition* de um grupo de *Conditions* conflituosas é conceitualmente efetuada por meio de contra-notificação para a *Exclusive-Premise* comum e decisão de resolução por esta mesma premissa. Esta confirmação pode ser realizada pelo S/CS, dado que este tem acesso (via *snooping* da *Rede de notificação Premise-Condition*) a todo o fluxo de notificações de *Premises* bem como também tem acesso (via *snooping* da *Rede de notificação Condition-Method*) ao resultado lógico de todas as *Conditions* aprovadas. Sendo assim, o S/CS pode confirmar a propagação do resultado lógico

"verdadeiro" somente da *Condition* que for vencedora da resolução de conflito, sendo o critério de resolução também de responsabilidade do S/CS. Para a ARQPON v1.0, propõe-se que este critério seja baseado unicamente no valor do campo de prioridade da instrução CONDITION-OP.

O Algoritmo 2 define a sequência de operações executadas pela ARQPON para a resolução de conflitos.

Algoritmo 2 – Resolução de conflito

Descrição: este algoritmo descreve as operações, realizadas pelo S/CS, pelos CPs e pelos MPs, para implementação do mecanismo de resolução de conflito a partir de uma notificação gerada por uma *Premise* com o *flag Exclusive* ligado.

Funcionamento:

1: os CPs nos quais estão alocadas as *Conditions* correspondentes à *Premise* notificante mantêm um *flag* indicando que estão "sob resolução".

2: o S/CS armazena o endereço da *Premise* mais o seu contador de *Conditions* (NC), por meio do *snooping* na *Rede de notificação Premise-Condition,* para posterior verificação.

3: ao final da execução, se o *flag* "sob resolução" estiver ligado o CP obrigatoriamente gera notificação, independente de o valor lógico da *Condition* ser TRUE ou FALSE, porém com a marcação do *flag* "sob resolução" embutida na notificação.

4: os MP de interesse, ao verificarem o *flag* "sob resolução" ligado, não executam imediatamente.

5: o S/CS, ao receber as notificações dos CPs "sob resolução" via *snooping* da *Rede de notificação Condition-Method,* efetua uma contagem comparando com o valor de NC armazenado. As notificações geradas pelos CPs incluem o endereço da *Premise Exclusive* originadora do conflito, de tal maneira que este endereço possa ser utilizado para *matching* do valor de NC armazenado previamente pelo S/CS.

6: ao verificar que todas as *Conditions* relacionadas à *Premise Exclusive* de fato executaram, o S/CS verifica quais delas apresentaram valor lógico TRUE e escolhe apenas uma para ser ativada de fato de acordo com um critério de resolução (valor de prioridade).

7: o S/CS replica a notificação da *Condition* vencedora na *Rede de notificação Condition-Method* porém com o *flag* "sob resolução" desligado.

8: os MPs executam os métodos relacionados à notificação gerada.

4.2.5.7.3 Determinismo

O mecanismo de garantia de determinismo descrito por Simão e Stadzisz (2010) tem por principal objetivo garantir que todos os elementos da cadeia do PON afetados por uma alteração de valor de *Attribute* tenham oportunidade de avaliá-la. Ou seja, este mecanismo garante que o ciclo de inferência funcione de maneira uniforme, no sentido de impedir que eventuais caminhos de notificação mais rápidos, excitados pela alteração em um *Attribute* A1, gerem alterações de outros *Attributes* que afetem a propagação de notificações em caminhos mais lentos também excitados pela alteração no *Attribute* A1, os quais deveriam também ter sido previamente avaliados por completo caso a velocidade de propagação das notificações de alteração de A1 fosse a mesma em todos os caminhos.

Do ponto de vista da ARQPON v1.0, o problema que motivou o mecanismo de garantia de determinismo é bastante pertinente, dado que em uma determinada implementação de *hardware* a execução das avaliações relacionais de diferentes *Premises*, das avaliações lógicas de diferentes *Conditions* e das operações de diferentes *Methods* pode diferir em termos de tempo de execução. Ainda, dado que na ARQPON existe o problema do escalonamento / alocação de *Premises, Conditions* e *Methods* em seus respectivos processadores, a etapa de alocação pode introduzir uma disparidade ainda maior no tempo de propagação da notificação em diferentes caminhos, visto que as notificações certamente se propagariam com maior velocidade em um caminho nos quais todos os elementos já estivessem previamente alocados em seus processadores.

A garantia de determinismo é resolvida na ARQPON v1.0 por meio da liberação, pelo S/CS, da propagação das notificações somente *depois* de ter verificado que todos os notificados estavam alocados nos seus respectivos processadores. Este mecanismo resolve o problema de diferença de velocidade de propagação de notificação, devido a alguns elementos estarem previamente alocados e outros não, pois se o determinismo estiver ativado, se força que as notificações sejam propagadas em “ondas” síncronas (camada a camada), ainda assim mantendo o paralelismo.

A escolha da propagação imediata de uma notificação ou do aguardo pela liberação do S/CS ocorre por meio de um *flag* nas notificações geradas por P/C/M, denominado *N/NF*. Este *flag* indica se é a notificação de alteração de fato ou se é uma notificação final (NF) gerada pelo S/CS para garantir determinismo nas P/C/M marcadas com o *flag Deterministic*.

O Algoritmo 3 define a sequência de operações relativas à garantia de determinismo na camada de notificação entre *Attributes* e *Premises.* Mecanismo semelhante é implementado na camada entre *Premises* e *Conditions* e na camada entre *Conditions* e *Methods.*

Algoritmo 3 – Garantia de determinismo em *Premises*

Descrição: este algoritmo descreve as operações, realizadas pelo S/CS e pelos PPs, para implementação do mecanismo de garantia de determinismo em *Premises* do PON marcadas com o *flag Deterministic*.

Funcionamento:

1: o *Attribute* marcado como *Deterministic* gera uma notificação com o *flag N/NF* assumindo o valor "N".

2: os PPs contendo *Premises* interessadas na notificação do *Attribute* e que possuam o *flag Deterministic* ignoram a notificação que contém o *flag N/NF* com o valor "N".

3: o S/CS, via *snooping* da *Rede de notificação Attribute-Premise,* percebe a notificação, efetua as operações de alocação e, ao final da alocação, percebendo que o *Attribute* é *Deterministic,* gera uma (re)notificação com o *flag N/NF* assumindo o valor "NF".

4: os PPs contendo *Premises* interessadas consomem a notificação que contém o *flag N/NF* com o valor "NF".

4.2.5.8 Rotina de *startup*

A rotina de *startup* é executada pelo núcleo von Neumann e tem por objetivo auxiliar o processo de descarga de uma aplicação para o P2ON, permitindo que a plataforma de desenvolvimento (p. ex. PC) interfaceie diretamente e somente com o núcleo von Neumann durante este processo. Isto viabiliza que a descarga da aplicação seja realizada por meio de uma interface e protocolo padronizados, tais como JTAG.

Após o *reset,* os blocos e subsistemas do P2ON se apresentam nos seguintes estados:

- Memória de *Attributes*: desabilitada (inacessível pelo núcleo von Neumann) e com mecanismo de notificações desativado.
- Memória de *P/C/M PON:* desabilitada (inacessível pelo núcleo von Neumann).
- Conjuntos de PP, CP e MP: não alocados.

- S/CS: tabelas de substituição zeradas (nenhum registro válido) e tabelas de alocação com todos os registros indicando processadores desalocados.
- Núcleo von Neumann: operacional e pronto para receber comandos da plataforma de desenvolvimento via interface JTAG.
- Memória de *startup* / métodos von Neumann: operacional e em espaço de endereçamento acessível pelo núcleo von Neumann, inclusive para execução de comandos via JTAG.

A Figura 44 ilustra os blocos funcionais envolvidos na rotina de *startup.* O código binário da aplicação PON deve ser descarregado para a *Memória de startup / métodos von Neumann* via interface JTAG com o Núcleo von Neumann. Este código binário deve conter os seguintes elementos:

- estruturas de dados que correspondem às instruções da ISA da ARQPON v1.0 a serem posteriormente copiadas e posicionadas nas memórias de *Attributes* e de P/C/M via *Interface de startup de AT* e *Interface de startup de PCM*.
- códigos dos métodos von Neumann, compilados para endereços absolutos.
- código da rotina de *startup* compilado no endereço inicial de execução do núcleo von Neumann.

Ao ser executado, logo em seguida ao *reset*, o código da rotina de *startup* deve realizar as operações descritas no Algoritmo 4.

Algoritmo 4 – Rotina de *startup*

Descrição: este algoritmo descreve as operações a serem realizadas pela rotina de *startup* logo após a descarga da aplicação PON na *Memória de startup/ métodos von Neumann* e posterior sinal de *reset.*

Funcionamento:

1: habilita a memória de P/C/M para acesso de escrita/leitura.

2: preenche a memória de P/C/M com as instruções PREMISE-OP, CONDITION-OP, METHOD-OP e METHOD-VN-OP a partir da estrutura de dados correspondente, via *Interface de startup de PCM*. Os bits de validade dos valores lógicos das instruções PREMISE-OP (V1, V2 e INI) e CONDITION-OP (VP) devem indicar que os seus valores lógicos são inválidos.

3: habilita a memória de *Attributes* para acesso de escrita/leitura.

4: preenche a memória de *Attributes* com as instruções ATTRIBUTE-DECL a partir da estrutura de dados correspondente, via *Interface de startup de AT.*
5: habilita o mecanismo de notificações no controlador da memória de *Attributes.*
6: para cada instrução ATTRIBUTE-DECL, força a geração de uma notificação de *startup* na *Rede de notificação Attribute-Premise.*
7: se necessário, altera o valor de um *Attribute* (via *Rede de R/W de Attributes*), iniciando desta maneira o funcionamento do mecanismo de notificações.

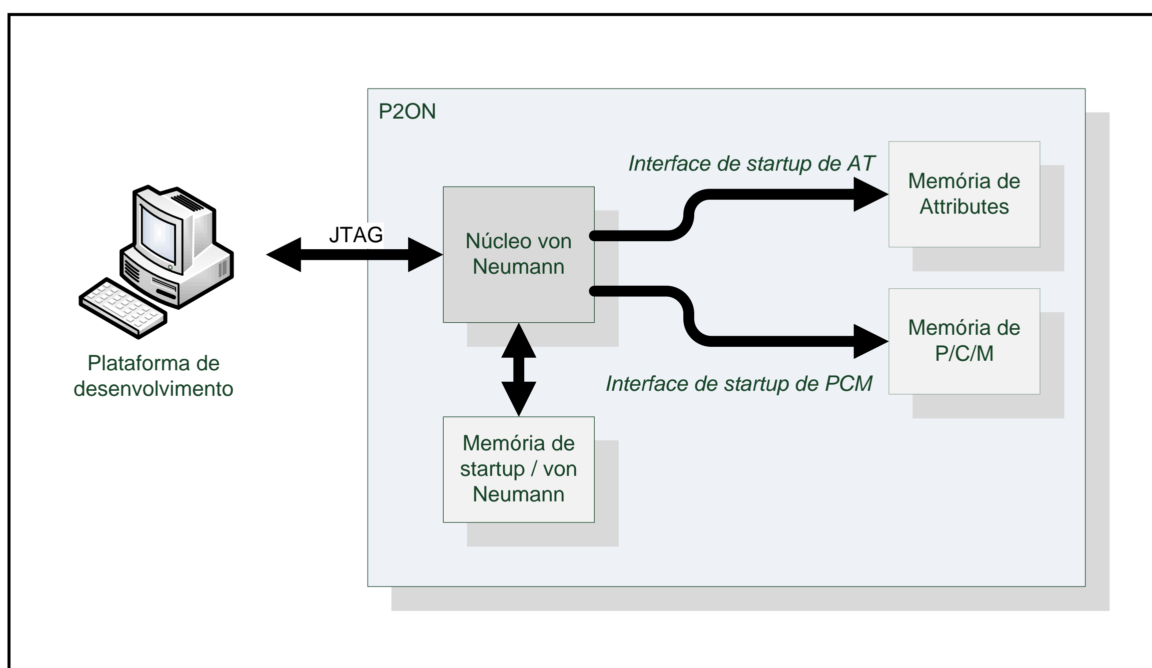


Figura 44 – Blocos funcionais envolvidos na rotina de *startup*
(Fonte: autoria própria)

Em particular, a operação de notificação forçada descrita no passo 6 deve ser executada pela *Memória de Attributes* para viabilizar que os valores iniciais dos *Attributes* sejam propagados para as *Premises* que deles dependem, alterando os valores dos bits V1 e V2 da instrução PREMISE-OP correspondente para 1. Uma vez que ambos os bits V1 e V2 sejam ligados e o *flag* INI tenha o valor inicial 0, o PP executa o cálculo lógico da *Premise* e necessariamente propaga notificação para as *Conditions* que dela dependem, alterando posteriormente o valor do *flag* INI para 1. A notificação gerada, por sua vez, altera o valor do bit do campo VP das instruções CONDITION-OP correspondentes às *Conditions* notificadas, de tal maneira que estas possam calcular seu valor lógico inicial e propagar notificação para os *Methods* que delas dependem tão logo ambos os bits do campo VP estejam setados.

4.2.5.9 Arbitragem de barramentos

Embora não apresentados explicitamente na Figura 38, cada barramento / rede de interconexão configurado como multi-mestre (ver Seção 4.2.5.3) possui um bloco funcional ligado a si responsável pela arbitragem. Ou seja, caso haja mais do que um mestre requisitando simultaneamente o acesso ao barramento, o bloco de arbitragem tem por objetivo eleger um dos mestres e garantir a ele o acesso efetivo, fazendo com que os demais mestres tenham o acesso suspenso temporariamente.

A política de eleição de determinado mestre, durante a etapa de arbitragem, pode ser implementada de diversas maneiras e obedecendo a diferentes critérios. Como em geral os algoritmos de arbitragem dependem de alguma forma de priorização para resolver conflitos de requisições simultâneas, propõe-se adotar a seguinte regra de arbitragem inspirada no trabalho de Wang (2001):

- Nos barramentos onde o S/CS concorra com outros processadores, o S/CS sempre é mais prioritário.
- Para cada barramento multi-mestre, atribui-se um identificador de prioridade a cada processador (com exceção do S/CS) de acordo com a seguinte regra:
    - Attribute-Premise: Ctrl de Memória de Attribute com identificador 0.
    - *Premise-Condition:* PP_0 a PP_{NPP-1} com identificadores 0 a NPP-1, respectivamente, e CP_0 a CP_{NCP-1} com identificadores NPP a (NPP + NCP - 1), respectivamente.
    - *Condition-Method:* MP_0 a MP_{NMP-1} com identificadores 0 a NMP-1, respectivamente, e CP_0 a CP_{NCP-1} com identificadores NMP a (NMP + NCP -1), respectivamente.
    - *R/W de Attributes:* MP_0 a MP_{NMP-1} com identificadores 0 a NMP-1, respectivamente, *Núcleo von Neumann* com identificador NMP e *Interfaces de E/S* com identificador NMP + 1.
- O processador de prioridade mais alta varia segundo a técnica *round-robin*, a cada ciclo de *clock.* Por exemplo, no ciclo 0 o processador de identificador 0 é o mais prioritário, em seguida o de identificador 1 e assim sucessivamente.
- A cada operação de arbitragem, o acesso ao barramento é garantido ao processador mais prioritário naquele ciclo de *clock* que esteja requisitando o barramento.

No caso da ARQPON, dado que os barramentos multi-mestre são aqueles utilizados para propagação de notificações e atualização de atributos, não é conveniente priorizar sempre um ou mais mestres específicos visto que, conceitualmente, o metamodelo de notificações do PON parte do princípio de que as notificações se propagam simultaneamente, ou seja, sem qualquer noção de priorização ou sequencialização. Assim, a política com variação do nível de prioridade mais alto em estilo *round-robin* tende a ser mais justa, pois altera periodicamente as prioridades relativas dos diferentes mestres.

4.2.6 Cenário de execução

Esta seção apresenta um cenário completo de execução de um conjunto de regras na ARQPON v1.0. O seu objetivo é complementar a explanação do modelo lógico da Seção 4.2.1, tanto do ponto de vista estrutural quanto dinâmico, esclarecendo de que forma este modelo mapeia para o conjunto de blocos e subsistemas que compõem a microarquitetura e para as suas correspondentes interações.

Considere-se as duas regras apresentadas a seguir em pseudocódigo, as quais não são conflitantes e não requerem execução determinística. Nestas regras identifica-se os elementos do metamodelo do PON conforme apresentado na Tabela 11.

Regras do exemplo:

Se (AT1 = 10) e (AT2 = 20) então
AT2 ← AT4 + AT5
AT3 ← AT3 + 1
Fim se
Se (AT1 = 30) e (AT2 = 40) então
AT2 ← 0
Fim se

Tabela 11 – Mapeamento das regras do cenário de execução em elementos do metamodelo do PON

Classe do elemento	**Lista de elementos**
Attribute	AT1, AT2, AT3, AT4 e AT5
Premise	P1: AT1 = 10

	P2: AT2 = 20 P3: AT1 = 30 P2: AT2 = 40
Condition	C1: P1 *E* P2 C2: P3 *E* P4
Method	M1: AT2 ← AT4 + AT5 M2: AT3 ← AT3 + 1 M3: AT2 ← 0

Considere-se, em adição, as pré-condições (situação inicial) de execução a seguir:

- P2ON possui 2 PPs, 2 CPs e 2 MPs.
- Valor inicial de AT1 é 30.
- Valor inicial de AT2 é 20.
- PP1 contém P1 alocada, com valor lógico anterior igual a *false.*
- PP2 contém P4 alocada, com valor lógico anterior igual a *false.*
- CP1 contém C1 alocada, com valor lógico anterior igual a *false* (valor de P1 igual a *false,* valor de P2 igual a *true*).
- CP2 contém C2 alocada, com valor lógico anterior igual a *false* (valor de P3 igual a *true,* valor de P4 igual a *false*).
- MP1 contém M1 alocado.
- MP2 está desalocado.

A Figura 45 apresenta o cenário de execução proposto a partir da sequência de interações entre os diversos blocos funcionais da ARQPON gerada pela alteração do valor inicial de AT1 para 10. Esta figura não está em escala na dimensão do tempo, porém ainda assim permite observar o paralelismo apresentado pelos blocos funcionais da ARQPON.

A mensagem 1 representa a notificação de alteração do *Attribute* AT1, gerada pela *Memória de Attributes.* Esta notificação é repassada ao PP1, ao PP2 e ao P-Scheduler do S/CS (mensagens 2, 5 e 7) por meio da *Rede de notificação Attribute-Premise*, sendo que o PP1 efetivamente efetua o cálculo da *Premise* P1 nele alocada (mensagem 8), dado que esta *Premise* está conectada ao *Attribute* AT1 (é dependente dele). PP2, por sua vez, ignora a notificação (mensagem 6) dado que possui uma *Premise* (P4) não dependente de AT1 nele alocada.

O P-Scheduler, por sua vez, efetua as operações de busca das *Premises* conectadas (mensagem 3), verificação de tabela de alocação (mensagem 4) e busca na *Mem P/C/M* da *Premise* não alocada (P3) que também é dependente de AT1 (mensagem 9). Isto dispara a desalocação de P4 de PP2, alocação de P3 e posterior salvamento do estado de P4 na *MemP/C/M* (mensagens 15, 19 e 21, respectivamente). A alocação de P3, finalmente, dispara o cálculo lógico de P3 (mensagem 20).

O cálculo lógico de P1 resulta em alteração do valor de *false* para *true*, o que causa uma notificação na *Rede de notificação Premise-Condition* (mensagem 10). O cálculo lógico de P3, por sua vez, resulta em alteração inversa (*true* para *false*), o que também causa uma notificação (mensagem 23).

A notificação de P1 é consumida pelos CP1, CP2 e C-Scheduler do S/CS (mensagens 17, 13 e 11). O C-Scheduler busca a lista de *Conditions* conectadas a P1 e verifica o estado da tabela de alocação (mensagens 12 e 14), concluindo que a única *Condition* dependente de P1 (C1) já está alocada. O processador CP1, no qual C1 está alocada, efetua o cálculo lógico (mensagem 18) em função do novo valor de P1 (*true*) e resulta em *true*, o que causa posterior geração de notificação na *Rede de notif Condition-Method* (mensagem 22). O processador CP2, por sua vez, por ter alocada a *Condition* C2 que não é dependente de P1, ignora a notificação (mensagem 16).

A notificação de P3 dispara sequência semelhante, sendo inicialmente consumida pelos CP1, CP2 e C-Scheduler do S/CS (mensagens 33, 28 e 25). O C-Scheduler busca a lista de *Conditions* conectadas a P3 e verifica o estado da tabela de alocação (mensagens 26 e 27), concluindo que a única *Condition* dependente de P3 (C2) já está alocada. O processador CP2, no qual C2 está alocada, efetua o cálculo lógico (mensagem 32) em função do novo valor de P3 (*false*) e resulta em *false*, o que não causa posterior geração de notificação na *Rede de notif Condition-Method* devido ao valor lógico de C2 não ter sido alterado (mensagem 38). O processador CP1, por sua vez, por ter alocada a *Condition* C1 que não é dependente de P3, ignora a notificação (mensagem 36).

A notificação gerada por C1 é consumida pelos MP1, MP2 e M-Scheduler do S/CS (mensagens 34, 29 e 24). O processador MP1, tendo em si alocado o *Method* M1 que é dependente de C1, executa este *Method* (mensagem 37), causando acessos de leitura e escrita aos *Attributes* AT4, AT5 e AT2 (mensagens 42 e 43) e a geração de uma notificação de alteração de AT2 (mensagem 44). O processador MP2, por sua vez, estando desalocado, ignora a notificação recebida de C1 (mensagem 30).

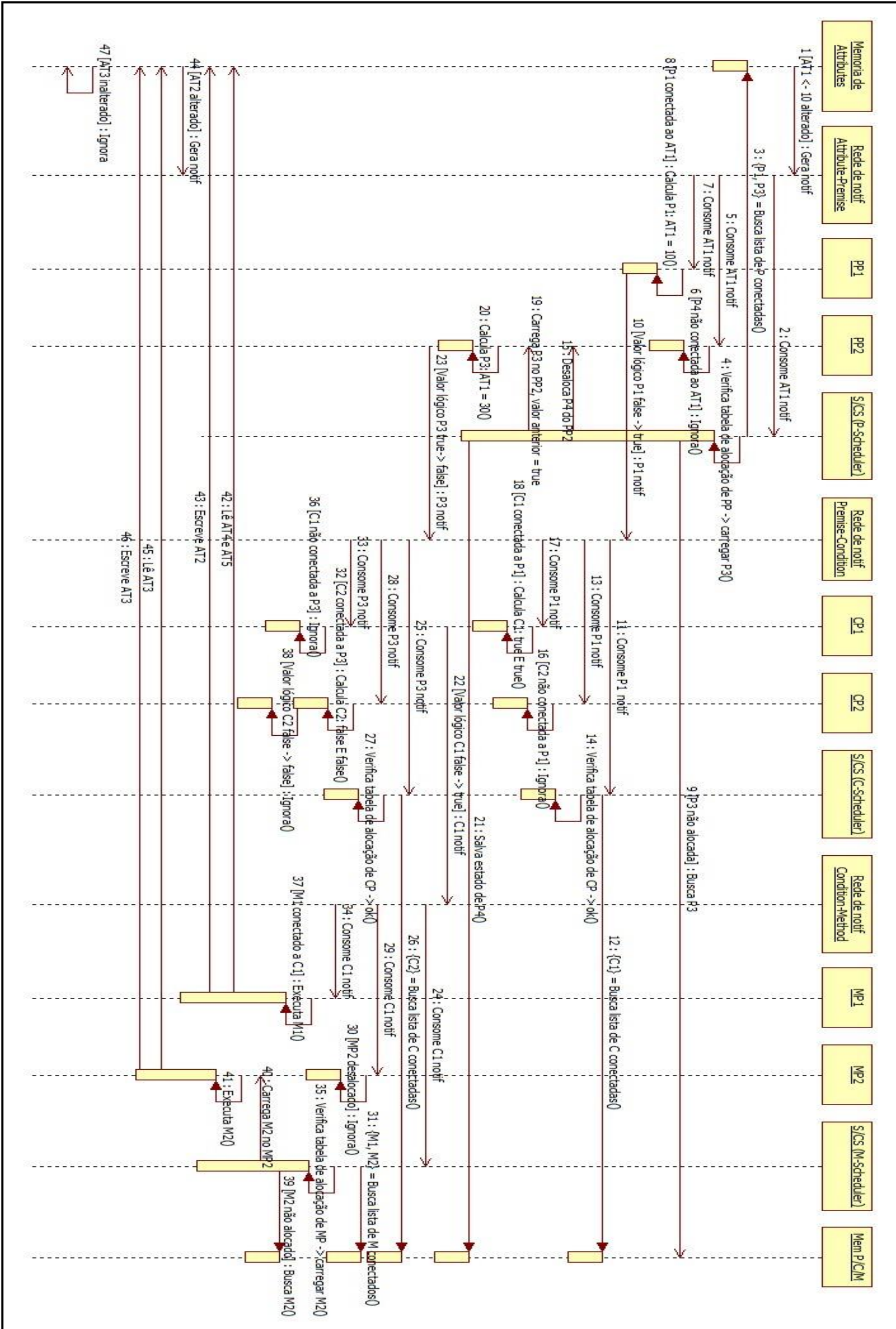


Figura 45 – Cenário de execução da ARQPON

(Fonte: autoria própria)

Finalmente, o M-Scheduler busca a lista de *Methods* conectados (mensagem 31), verifica a necessidade de alocação de M2 (mensagem 35), busca M2 na *MemP/C/M* (mensagem 39) e carrega M2 no MP2 (mensagem 40), disparando a sua posterior execução (mensagem 41). Esta execução causa acessos de leitura e escrita ao *Attribute* AT3 (mensagens 45 e 46), em consequência alterando o valor deste *Attribute* e gerando a correspondente notificação (mensagem 47).

As notificações geradas pelas alterações de AT2 e AT3 (mensagens 44 e 47, respectivamente) iniciam um novo ciclo de potenciais notificações e sua execução por parte dos diferentes processadores e do S/CS. O paralelismo existente na execução por estes diversos processadores é evidente na figura, assim como também se evidencia que operações de escrita/leitura em memória, principalmente na *MemP/C/M,* podem se constituir em gargalos devido à concorrência potencial entre estes acessos.

4.3 Considerações relativas à utilização de hardware reconfigurável para implementação da ARQPON

Segundo Asanovic et al. (2006), FPGAs têm se tornado cada vez mais uma alternativa para a implementação de protótipos de arquiteturas de processadores, em função de fatores tais como:

- Contínuo aumento da capacidade em novas versões de dispositivos, permitindo a prototipação de lógicas cada vez mais complexas.
- Flexibilidade de reconfiguração, permitindo que testes sejam rapidamente reformulados e refeitos após avaliações dos testes anteriores. Para os pesquisadores da área esta é uma vantagem que pode compensar até mesmo as limitações de desempenho das FPGAs.
- Disponibilidade de módulos ou subsistemas prontos, sejam *open source* ou *IP-cores*, que podem compor sistemas maiores e facilitar a sua implementação.

No entanto, segundo Hauser e Wawrzynek (1997) o uso de FPGAs implica algumas limitações, tais como:

- a FPGA escolhida para determinada plataforma pode não conter número suficiente de elementos lógicos para codificar determinada aplicação e, mesmo que seja

possível reconfigurá-la em tempo de execução, esta operação pode ser muito custosa para o desempenho.

- FPGAs têm capacidade limitada de armazenamento de dados, o que pode requerer a implementação de sistemas de acesso a memória externa.
- algumas lógicas, quando implementadas em FPGA (p. ex. ULAs) são mais complexas e menos eficientes do que as mesmas lógicas implementadas em ASICs, dadas as características funcionais das LUTs presentes nas FPGAs, o que impacta no desempenho.

Do ponto de vista da implementação da ARQPON (ou seja, do P2ON), certamente esta poderá ser futuramente efetuada em ASIC, pois será um caminho natural para uma versão para a qual se pretenda uma maior escala de produção, aproveitando-se das características positivas dos ASICs anteriormente mencionadas.

No entanto, a prototipação inicial do P2ON, no escopo deste trabalho de doutorado, justifica a sua implementação em FPGA principalmente pela flexibilidade de reconfiguração anteriormente citada, que é útil nesta fase na qual se pretende corrigir erros ou validar e aprimorar a implementação em iterações curtas e rápidas. Naturalmente, as também citadas limitações de desempenho e de densidade impostas pela implementação em FPGA devem ser levadas em consideração quando da análise e comparação dos resultados obtidos na etapa de *benchmarking.*

A utilização de módulos prontos, *open source* ou *IP-cores*, não está descartada na implementação do P2ON justamente pela agilidade que confere ao processo de prototipação. No entanto, a concepção da ARQPON não é afetada porque esta se propõe primordialmente a ser um modelo arquitetural, portanto definindo os seus blocos fundamentais e as relações existentes entre eles independentemente de implementações específicas.

Ainda no que tange a limitações, a capacidade reduzida de memória das FPGAs impacta na complexidade do protótipo de P2ON a ser implementado. No entanto, dada a escalabilidade pretendida na concepção da ARQPON, o protótipo de P2ON deve ser adaptável à quantidade de recursos disponíveis. Além disso, o modelo da ARQPON deve prever interfaces para a conexão de subsistemas externos, tais como expansões de memória.

Em relação à abordagem de implementação a ser utilizada, o grau de complexidade que a ARQPON pode vir a apresentar, principalmente em função da utilização de recursos avançados de *pipelining* e de NoC ou da previsão de extensões para estes recursos, a princípio sugere que uma concepção estrutural em VHDL venha a ser a mais adequada. Esta opção pela

abordagem estrutural se justifica pela busca da otimização de desempenho, que é facilitada nesta abordagem em relação à abordagem comportamental justamente pelo maior controle que o desenvolvedor exerce em relação à composição do circuito de *hardware* gerado. No entanto, em função principalmente da quantidade e complexidade das máquinas de estado a serem projetadas nos subsistemas da ARQPON, pode-se optar por uma abordagem predominantemente comportamental de maneira a facilitar o desenvolvimento das versões e protótipos iniciais.

A opção por VHDL, por sua vez, se justifica também pelo grau de complexidade, que desencoraja uma abordagem baseada unicamente no desenho de diagrama esquemático. Além disso, o desenvolvimento em VHDL também permite maior controle sobre a composição do circuito quando comparado a abordagens baseadas em linguagens de programação de alto nível.

Finalmente, como o P2ON é uma implementação genérica e não alterável da ARQPON (ou seja, contém uma quantidade fixa e pré-determinada de circuitos que desempenham as funções dos diferentes elementos da cadeia de notificações do PON), optou-se por utilizar uma estratégia de reconfiguração estritamente estática. No entanto, não se descarta as opções de reconfiguração semi-estática ou dinâmica em futuras versões da ARQPON e implementações do P2ON.

4.4 Considerações sobre o capítulo

Este capítulo apresentou o desenvolvimento da versão preliminar da ARQPON, conforme proposto como etapa de qualificação do desenvolvimento desta tese de doutorado.

Foram apresentados em detalhes os requisitos a partir dos quais a ARQPON deveria ser concebida. De forma complementar a estes requisitos, concebeu-se o modelo lógico desejado para a ARQPON bem como se discorreu sobre aspectos arquiteturais relevantes que deveriam orientar a sua proposição.

Em seguida, a ARQPON foi proposta em termos da sua interface de programação (ISA) e dos blocos componentes da sua microarquitetura. Para cada um destes blocos discorreu-se sobre detalhes da sua estrutura interna, com ênfase no mapeamento deste bloco para o modelo lógico da ARQPON e na interação funcional e estrutural com os demais blocos.

Apresentada então esta proposta para a versão preliminar da ARQPON, embora sendo inovadora em si por ser totalmente adequada a um paradigma emergente que é o PON, pode-se analisá-la e categorizá-la à luz da fundamentação teórica sobre arquiteturas de computadores anteriormente apresentada.

Do ponto de vista de técnicas de implementação de paralelismo, a ARQPON não propõe implementar ILP na forma de *pipelining* justamente para simplificar a construção das unidades de processamento, em particular os MPs. Isto viabiliza uma quantidade maior de unidades de processamento no mesmo dispositivo de *hardware,* favorecendo a execução paralela no nível de granularidade dos elementos do metamodelo de notificações do PON.

Em relação a TLP, pode-se afirmar que o conceito de *thread* no modelo de notificações do PON não é claro ou explícito, dado que o grafo gerado a partir das conexões entre os elementos da cadeia de notificações é composto por múltiplas "ramificações de notificação" que não são desconexas o suficiente umas das outras para serem consideradas como *threads* independentes (embora a execução de um elemento do PON isoladamente possa ser considerada uma *microthread*, a sua granularidade – em nível de instrução – não contribui para esta análise).

No entanto, apesar desta dissimilaridade, considerando-se que a execução de uma iteração completa do ciclo de notificações (*Attribute-Premise-Condition-Action-Instigation-Method-Attribute*), por uma ramificação qualquer da cadeia, é semelhante a uma *thread* pelo fato de ser uma linha de execução sequencial, pode-se posicionar a ARQPON em relação às técnicas de TLP da seguinte forma:

- Semelhante a modelos *multithreaded*, em relação à otimização do acesso à memória. Isto ocorre porque uma unidade de processamento pode efetuar acesso à memória enquanto as demais estão processando notificações pendentes, portanto aproveitando o tempo de latência no acesso.
- Semelhante a modelos IMT, pois um determinado processador de P/C/M é alocado dinamicamente para execução da instrução (*Premise, Condition* ou *Method*) correspondente, uma instrução por vez, independente da ramificação do ciclo de notificações na qual esteja executando. É pertinente a ressalva, no entanto, que cada processador isoladamente não pode ser considerado IMT por não implementar um *pipeline* que executa instruções de diferentes *threads* simultaneamente.
- Semelhante a modelos SMT, pois os conjuntos de PP, CP e MP podem executar simultaneamente diferentes ramificações do ciclo de notificações. O controle do

despacho para diferentes unidades é efetuado pelo S/CS, que seria equivalente ao controlador do processo de emissão múltipla em uma arquitetura SMT von Neumann. Ainda, gargalos que limitem o despacho superescalar em paralelo não são um problema para a ARQPON porque, diferentemente de programas von Neumann, em programas PON não existe o conceito de "janela de instruções" que é executável simultaneamente (intervalo entre *branches*) e um tamanho típico para esta janela (de acordo com estudo efetuado por Ungerer, Silc e Robic (1998), igual a 7 para programas von Neumann). Idealmente, em um programa PON quaisquer *Methods* de *Rules* ativadas podem ser executados paralelamente depois de resolvidas questões de determinismo e conflitos.

- ARQPON não é classificável como SMP, visto que os PP, CP e MP são diferentes entre si (portanto, não simétricos), no entanto os PPs, CPs e MPs não podem ser tampouco considerados co-processadores porque a execução é totalmente descentralizada. A interconexão lógica entre os conjuntos de PPs, CPs e MPs e a divisão de tarefas entre eles, para execução de uma determinada iteração do ciclo de notificações, apresenta semelhança com o funcionamento dos *arrays* sistólicos (mencionado na Seção 2.3.1).

Do ponto de vista de organização de memória, pode-se considerar que a ARQPON define 3 níveis hierárquicos:

- Memória principal: particionada em 3 blocos físicos (*Memória de Attributes, Memória de P/C/M* e *Memória de Setup / von Neumann*), cada um com seu controlador específico.
- Memória *cache* de nível 2 (L2): corresponde às *caches* de *Attribute* e P/C/M, sendo que a primeira implementa a semântica CSMA. O fato da *cache* ser única e centralizada pode ser um gargalo potencial para a versão preliminar da ARQPON porém simplifica o projeto dado que elimina a necessidade de implementação de protocolos de coerência de *cache*.
- Memória *cache* de nível 1 (L1): consiste nos registradores dos PP, CP e MP responsáveis por armazenar o valor atual dos operandos utilizadas por cada processador específico. Este nível só é viável em função do conceito do PON de alocação de uma instrução específica para um processador específico, ou seja, o processador pode efetuar *caching* tanto da operação a ser executada quanto dos

> dados envolvidos na avaliação de notificação. Ainda, esta alocação permite realizar uma característica que diferencia sobremaneira a ARQPON de arquiteturas von Neumann convencionais, que é o fato de a etapa de busca (*fetch*) de instrução estar desvinculada das demais etapas da sua execução; isto permite explorar ainda melhor questões de localidade temporal que já foram otimizadas ao se diminuir as redundâncias temporais quando da concepção do *software* segundo o PON.

A semântica de memória da ARQPON não é do tipo *load/store* porque isto dependeria de que os registradores se mantivessem inalterados entre a operação que os atualiza e a operação de acesso à memória que deles depende (*load* ou *store*). No entanto, no modelo proposto para a ARQPON não haveria como garantir que estas duas operações seriam executadas pelo mesmo MP, mesmo sendo na forma de regras dependentes, e como cada MP possui um conjunto de registradores próprio não haveria como um MP acessar a informação previamente armazenada nos registradores do outro MP.

Uma consequência desta característica é que não se pode garantir manutenção de contexto em um mesmo MP para duas instruções consecutivas; isto justifica o fato de os MP não possuírem registradores auxiliares acessíveis explicitamente, tais como os acumuladores e registradores de propósito geral de arquiteturas RISC e CISC von Neumann, e implica em que qualquer operando ou resultado de método, mesmo que intermediário e somente utilizado por uma sequência de *master rule* e dependentes, deva ser armazenado na memória de *Attributes.* Este pode ser considerado um ponto fraco da ARQPON v1.0, no entanto é consistente com o modelo teórico do PON no sentido de manter toda e qualquer informação de estado da aplicação na forma de *Attributes* de FBEs.

Em relação a semânticas de memória implementadas em arquiteturas de fluxo de dados, o conceito de *Explicit Token Store* não se aplica à ARQPON porque o disparo das instruções não é função da disponibilidade de dados e sim de notificações, portanto não é necessário efetuar *matching* de dados. De fato, o S/CS precisa efetuar operações de busca em tabelas para efetuar a alocação de instruções nos processadores correspondentes, porém esta busca está limitada à quantidade de processadores de cada tipo, que tende a ser muito menor do que a quantidade de *tokens* potencialmente aguardando por *matching* em uma arquitetura de fluxo de dados. Ainda, em função do tamanho relativamente pequeno da tabela de alocação, a busca pode ser implementada paralelamente em *hardware* mediante circuitos adequados, de forma análoga ao *matching* efetuado para determinação de *hit* ou *miss* em uma memória cache, por exemplo.

Em relação a *I-Structures,* de certa forma a semântica CSMA implementa o disparo de instruções em função do resultado de uma operação de acesso à memória, de forma análoga ao que ocorre em *I-Structures* quando existem operações de acesso pendentes. Pode-se considerar que a lista de *Premises* conectadas a um *Attribute* apresenta alguma similaridade semântica com a lista de acessos pendentes implementada por uma *I-Structure*, embora exista uma lista para cada *Attribute* dado a estrutura de grafo da cadeia de notificações.

Do ponto de vista de topologia de interconexão, conforme já explanado na Seção 4.2.3.2, dado que conceitualmente o número de PPs, CPs e MPs pode ser grande (em função da sua baixa complexidade), arranjos de interconexão ponto-a-ponto consumiriam muitos recursos de *hardware*. Soluções com NoC e barramento são mais viáveis, preferindo-se esta última inicialmente para diminuir a complexidade. No entanto, em versões posteriores, com números maiores de processadores, deve-se estudar a hipótese de implementar NoC ou um modelo híbrido (Figura 37). A proposta de uso de múltiplos barramentos com finalidades distintas (propagação de notificação, *fetch* de instruções, atualização de dados, etc.) encontra um paralelo na implementação de 5 redes distintas pelo *Tile Processor* (WENTZLAFF et al*.,* 2007), embora o esquema de comutação implementado neste seja na forma de *crossbar* em um *grid* 2D.

Em relação à ISA proposta para a ARQPON v1.0, embora o gabarito das instruções seja relativamente extenso (a definição de um *Attribute* pode ocupar até mais do que 1024 bytes), é importante ressaltar que cada instrução embute em si não somente a definição da sua operação (relacional, lógica ou aritmética) mas também informação de fluxo de controle relativa à parte da estrutura da cadeia de notificações da qual aquela instrução participa. Sendo assim, não faz sentido avaliar, do ponto de vista de complexidade ou tamanho em *bits,* a definição da ISA da ARQPON à luz de ISAs RISC ou CISC de arquiteturas von Neumann ou ISAs de arquiteturas de fluxo de dados, visto que a semântica de cada instrução da ARQPON é substancialmente mais rica.

Um conceito importante aplicado à ISA da ARQPON v1.0, que é derivado da análise das ISAs von Neumann e de fluxo de dados, é a identificação do que seria a operação de *Method* mais simples, denominada "método mínimo". Esta definição permite projetar o MP da forma mais simples possível, atendendo ao aspecto desejado de granularidade fina e à funcionalidade requerida pelos *Methods* PON.

O mecanismo de determinismo e resolução de conflito, por sua vez, é projetado na ARQPON v1.0 valendo-se de verificações efetuadas pelo S/CS e (re)propagação de notificações. Embora esta decisão de projeto aumente a complexidade da lógica do S/CS, o

mecanismo não deve agregar *overhead* significativo para a execução em si uma vez que não existem contra-notificações de fato, conforme proposto por Simão e Stadzisz (2010) no mecanismo conceitual, mas sim uma confirmação de notificação pelo S/CS.

Como complemento às características apresentadas, algumas questões que não foram contempladas na proposta da ARQPON v1.0 e que podem ser futuramente consideradas, dependendo dos resultados da avaliação a ser efetuada sobre a implementação da versão preliminar:

- Inclusão de informação de contexto na ISA, conforme discorrido na Seção 4.2.3.5, permitindo a criação de escopos locais e ciclos de vida variáveis para os FBEs da aplicação PON.
- Inclusão na ISA de campos que permitam a alocação estática de instruções PON para seus respectivos processadores, ou seja, que permitam às próprias instruções conter informações sobre como o S/CS deve atuar na alocação de instruções para os respectivos PP, CP ou MP, de tal maneira a otimizar o desempenho (p. ex. minimizando o número de desalocações e alocações). As técnicas utilizadas para esta alocação estática (implementadas, por exemplo, por um compilador PON) estão fora do escopo deste trabalho.
- Implementação de mecanismos mais sofisticados de resolução de conflitos, por exemplo codificados em métodos von Neumann que seriam disparados pelo S/CS. Neste caso, deveria haver um mecanismo que permitisse ao processador von Neumann informar o resultado da resolução de conflito ao S/CS para que este pudesse proceder com o protocolo.
- Implementação de um nível de *cache* intermediário entre os atuais níveis 1 e 2, que permitisse manter conjuntos de instruções em *buffers* específicos para cada unidade de processamento de forma similar à proposta pela arquitetura EDGE/TRIPS (BURGER et al., 2004). Isto poderia otimizar o desempenho por diminuir os acessos à *cache* compartilhada de nível imediatamente superior, porém tornaria o gerenciamento de alocação pelo S/CS mais complexo.

No que tange a questões de *software,* segundo Kogge *et al.* (2008), permitir a execução de *software* legado é uma característica importante e desejável para um modelo inovador de arquitetura paralela, conforme discutido quando da proposição do requisito RF-02 para a ARQPON (ver Seção 4.1.1). Sendo assim, prever na ARQPON uma interface com

um núcleo von Neumann embutido, viabilizando a execução híbrida de *software* PON e von Neumann, é importante no sentido de permitir a execução de *software* legado com uma quantidade menor de adaptações, ainda assim podendo aproveitar das características inovadoras de um modelo de execução adaptado ao PON.

Finalmente, segundo Sankaralingam *et al.* (2003), arquiteturas de granularidade fina executam melhor aplicações que apresentam paralelismo de dados e comportamento mais regular (p. ex. *loops* em vetores), ao passo que arquiteturas de granularidade grossa executam melhor aplicações irregulares e com muito código de fluxo de controle, tais como compressão e compilação. Embora a ARQPON seja essencialmente de granularidade fina, conforme discutido anteriormente, é possível que aplicações irregulares e com muito fluxo de controle sejam eficientemente executadas pela ARQPON dado que estas aplicações podem apresentar características totalmente diferentes quando reimplementadas segundo o PON. Uma análise mais aprofundada da reimplementação de aplicações von Neumann em PON ainda é objeto de estudo futuro, porém algumas conclusões a este respeito certamente poderão ser obtidas a partir da análise das aplicações de *benchmark* quando da avaliação do protótipo do P2ON v1.0.

Feitas estas considerações, o Capítulo 5 a seguir apresenta as conclusões deste trabalho de qualificação.

5 Conclusões

A evolução tecnológica dos dispositivos de *hardware* para computação, de certa forma em consonância com os preceitos da Lei de Moore, alcançou um ponto em que se tornou muito mais viável investir em arquiteturas de computadores que aproveitem a cada vez maior densidade de circuitos eletrônicos disponível do que aumentar a velocidade do *clock* conforme ocorria tradicionalmente, dado limitações físicas e tecnológicas de construção. Isto tem favorecido a construção de arquiteturas de processadores de múltiplos núcleos, os quais são capazes de executar paralelamente múltiplas linhas de execução (*threads*) de *software.*

No entanto, a construção de *software* intrinsecamente paralelo requer modelos de programação que favoreçam a expressão deste tipo de abstração. Neste aspecto a pesquisa na área de computação paralela carece de ideias inovadoras, que proponham modelos distintos dos tradicionais (tais quais a programação imperativa para arquiteturas von Neumann.) e que imprimam um novo impulso ao desenvolvimento de *software* para arquiteturas paralelas. A inovação pode ocorrer de várias formas, seja abordando as deficiências de abstração de paralelismo dos modelos de programação atuais, seja propondo alternativas que minimizem os efeitos de barreiras tecnológicas do *hardware* (p. ex *memory wall*), porém levando em consideração a tendência de paralelização crescente do *hardware* e seus efeitos na construção do *software.*

Neste âmbito, o Paradigma Orientado a Notificações (PON) se constitui em uma inovação em relação aos modelos e técnicas de programação tradicionais. Isto porque o PON propõe a concepção de *software* na forma de regras aplicadas sobre fatos, porém com um mecanismo de inferência otimizado, baseado em pequenas entidades lógico-causais que colaboram por notificações pontuais. Este modelo tem o potencial de diminuir ou eliminar redundâncias de processamento, tipicamente presentes inclusive em *software* construído segundo o Paradigma Imperativo (PI), que é largamente utilizado tanto na academia quanto na indústria. Além disso, a organização estrutural desacoplante do mecanismo de notificações do PON favorece a execução distribuída e/ou paralela de determinado *software* construído segundo este paradigma.

Tanto o estado da arte quanto da técnica de desenvolvimento de *software* PON, no momento, embora em contínuo esforço de pesquisa, carecem de um ambiente de *hardware* adequado à exploração de suas características conceituais. Ou seja, é necessário desenvolver um ambiente de *hardware* que mapeie de forma mais fidedigna possível os elementos

conceituais do mecanismo de inferência do PON e os execute também de forma próxima ao modelo conceitual, com a propagação paralela e simultânea de notificações.

Algumas pesquisas foram desenvolvidas visando a propor uma alternativa de ambiente de execução para o PON (PETERS, 2012) [135] (WITT et al., 2011) [97], porém com o viés de se desenvolver um circuito de *hardware* customizado para uma determinada aplicação, eventualmente com uma característica híbrida de execução PON e PI. Estas soluções apresentam escalabilidade limitada e são dependentes de um processo de reconfiguração de *hardware* (tipicamente dispositivos de FPGA) para a aplicação específica, portanto de forma menos flexível do que técnicas tradicionais de desenvolvimento de aplicações que são orientadas puramente a *software*.

Neste contexto, a principal contribuição desta proposta de doutorado é definir a ARQPON. Esta é uma arquitetura de processador, concebida como um sistema de blocos funcionais que são fixos para determinada implementação deste processador (P2ON) e que se organizam de tal maneira a viabilizar a execução de uma aplicação PON de forma mais fiel ao seu modelo conceitual no que diz respeito às características de desacoplamento, distribuição e paralelismo. Diferentemente das propostas anteriormente apresentadas, a ARQPON pretende ser genérica e escalável o suficiente para permitir a execução de aplicações PON de tamanho e complexidade variados, implementadas puramente na forma de *software* constituído pelo conjunto de instruções definido para a ARQPON.

Em função de suas características, a ARQPON se insere em um processo de desenvolvimento tradicional de aplicações, que envolve o projeto/implementação de *software,* compilação, carga do *software* na plataforma de execução e posterior execução deste *software* pelo processador da ARQPON. De fato, a evolução da tecnologia dos dispositivos reconfiguráveis os torne cada vez mais atraentes para utilização em cenários nos quais o *hardware* pudesse ser reconfigurado toda vez que se necessitasse executar uma nova aplicação (reconfiguração semi-estática ou dinâmica), com os potenciais ganhos decorrentes desta especificidade. Apesar deste fato, a proposta da ARQPON como apresentada se encaixa no contexto tecnológico atual, no qual a sua implementação na forma de um ASIC seria economicamente e tecnologicamente mais viável caso o PON seja adotado como modelo de desenvolvimento de aplicações em grande escala. Isto não exclui a possibilidade de utilização de implementações de P2ON juntamente com dispositivos de lógica reconfigurável em plataformas computacionais, dada a aplicabilidade da teoria do PON e, portanto, de sua estrutura de execução, em aplicações dedicadas tais como as usualmente desempenhadas por sistemas embarcados.

A motivação deste projeto de pesquisa está no aprimoramento do estado da técnica do PON, em um primeiro momento. Entretanto, primordialmente, também se objetiva aprimorar o estado da arte, haja visto que a materialização do PON em uma forma o mais próxima possível do modelo conceitual e a avaliação do comportamento de aplicações PON neste ambiente poderá revelar aspectos ainda não estudados do paradigma, tais como a sua real capacidade de processamento lógico-causal distribuído. A relevância do projeto, por sua vez, está em tentar apresentar uma contribuição para a questão já mencionada dos rumos e tendências da pesquisa na área de computação paralela, na forma de um modelo que é novo conceitualmente e não somente tecnologicamente. Ainda, a pesquisa potencialmente serve como meio de reflexão sobre o PON se apresentar como um novo paradigma de computação.

Por sua vez, a execução do projeto de doutorado é viável justamente pela característica da ARQPON ser naturalmente escalável. Ou seja, a prototipação da ARQPON e avaliação de aplicações, conforme proposto para a continuidade dos trabalhos, pode ser efetuada em uma escala reduzida, de tal maneira a facilitar a etapa de prototipação que tende ser a mais complexa e trabalhosa. Ainda, embora haja a intenção de se elaborar um protótipo o mais otimizado possível, visando principalmente a demonstrar um possível ganho de desempenho em relação a outras plataformas de implementação de *software* PON, este não é um critério determinante para validação ou invalidação do objeto de pesquisa. De fato, uma avaliação crítica da proposta do modelo de execução e da sua aplicabilidade à filosofia do PON, auxiliando em obter uma visão mais clara dos potenciais do paradigma, é relativamente mais importante do que a avaliação da qualidade do protótipo da ARQPON em si.

Do ponto de vista de inovação, o próprio contexto do PON onde a ARQPON se insere é um tema relativamente novo de pesquisa. Aliado a este fato, as premissas básicas de generalidade e flexibilidade a partir das quais a ARQPON foi concebida são inovadoras por si só, pois se constituem em evolução da própria plataforma de execução de *software* PON. Ainda, a execução hierarquizada das instruções pela ARQPON, em um nível de abstração relativamente mais elevado, é diferente do que se propõe tradicionalmente com o modelo de von Neumann, permitindo uma discussão a respeito desta característica e suas implicações.

Em suma, diferentemente de von Neumann / Turing (e afins), a ARQPON tipifica as primitivas de processamento em factuais (processamento e controle de *Attributes*), lógicas, causais (processamento de *Premises* e *Conditions* pelos PPs e CPs) e procedimentais (processamento de *Methods* pelos MPs). Isto permite dar tratamento particular e otimizado, à luz de notificações, para elementos que naturalmente são diferenciados. Ao bem da verdade, o

modelo de von Neumann / Turing considera todos esses elementos de forma homogênea, dentro de um fluxo de execução de instruções, negligenciando suas particularidades.

A etapa seguinte deste projeto envolve a prototipação da ARQPON e sua avaliação executando uma aplicação PON relativamente simples. Esta etapa objetiva analisar a proposta preliminar da ARQPON e efetuar eventuais correções ou melhoramentos para, em seguida, poder efetuar avaliações mais completas envolvendo aplicações mais robustas e comparações com outras implementações, tanto seguindo o modelo do PON (baseadas no *framework* PON C++ ou em implementação de *hardware* específica) quanto seguindo o modelo do PI. Os resultados finais obtidos contribuirão para uma avaliação mais clara e embasada não só da eficácia do modelo arquitetural proposto mas também, de maneira mais ampla, da aplicabilidade do PON e do seu conjunto de técnicas para o desenvolvimento de sistemas computacionais em geral.

6 Referências

ÁDÁM, N. **Single Input Operators of the DF KPI System**. Acta Polytechnica Hungarica, *7*(1), 73-86, 2010.

AGERWALA, T.; CHATTERJEE, S. **Computer architecture: challenges and opportunities for the next decade.** *Micro, IEEE* , vol.25, no.3, p. 58- 69, Maio-Junho 2005.

ALTERA CORPORATION. **AN184: Simultaneous Multi-Mastering with the Avalon Bus System.** 2002. Disponível em: <http://extras.springer.com/2001/978-0-306-47635-8/an/an184.pdf>. Acesso em 28/12/2012.

ALTERA CORPORATION. **Avalon Interface Specifications.** Revisão 11.0, 2011.

ALVERSON, R.; CALLAHAN, D.; CUMMINGS, D.; KOBLENZ, B.; PORTERFIELD, A.; SMITH, B. **The Tera Computer System.** Proceedings of the 4th international conference on Supercomputing (ICS '90) (p. 1-6). New York, NY, USA: ACM, 1990.

ANSYS Fluent Benchmarks. Disponível em: <http://www.ansys.com/Support/Platform+Support/Benchmarks+Overview/ANSYS+Fluent+Benchmarks>. Acesso em 17/10/2012.

ANTIQUEIRA, P. A. **Implementação de Modelos de Redes de Petri em *Hardware* de Lógica Reconfigurável**. 2011. Dissertação de Mestrado, CPGEI, UTFPR. Curitiba, Brasil, 2011.

ARGONNE NATIONAL LABORATORY. **The Message Passing Interface (MPI) standard**. Disponível em: <http://www.mcs.anl.gov/research/projects/mpi/>. Acesso em 29 ago. 2012.

ARM (ADVANCED RISC MACHINES LIMITED). **ARM7TDMI (Thumb) Data Sheet.** Revisão B, Jan 1999.

ARM (ADVANCED RISC MACHINES LIMITED). **AMBA Specification**. Revisão 2.0, 1999.

ARM (ADVANCED RISC MACHINES LIMITED). **The ARM Cortex-A9 Processors.** 2009. Disponível em: <http://www.arm.com/files/pdf/armcortexa-9processors.pdf>. Acesso em 22/08/2012.

ARM (ADVANCED RISC MACHINES LIMITED). **ARM9TDMI Technical Reference Manual (Revision 3).** Disponível em: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337g/CACCIFED.html>. Acesso em 21/08/2012.

ARVIND; NIKHIL, R.S. **Executing a program on the MIT tagged-token dataflow architecture.** IEEE Transactions on Computers, vol.39, no.3, p.300-318, Mar 1990.

ASANOVIC, K., BODIK, R., CATANZARO, B. C., GEBIS, J. J., HUSBANDS, P., KEUTZER, K., et al. **The Landscape of Parallel Computing Research : A View from Berkeley**. 2006. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf>.

BANASZEWSKI, R. F.; STADZISZ, P. C.; TACLA, C. A.; SIMÃO, J. M. **Notification Oriented Paradigm (NOP): A Software Development Approach based on Artificial Intelligence Concepts**. Logic Applied To Technology, 2007.

BANASZEWSKI, R. F. **Paradigma Orientado a Notificações : Avanços e Comparações**. Dissertação de Mestrado. CPGEI, UTFPR. Curitiba, Brasil, 2009.

BARBARÁN, G. C.; FRANCISCHINI, P. G. **Indicadores de Produtividade na Indústria de Software**. ENEGEP Encontro Nacional de Engenharia de Produção, 78(2), 7, 1998.

BARNEY, B. **POSIX Thread Programming**. 2012. Disponível em: <http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf>. Acesso em 25/10/2012.

BATISTA, M. V.; BANASZEWSKI, R. F.; RONSZCKA, A. F.; VALENCA, G. Z.; LINHARES, R. R.; STADZISZ, P. C.; TACLA, C. A.; SIMAO, J. M. **Uma Comparação entre o Paradigma Orientado a Notificações (PON) e o Paradigma Orientado a Objetos (POO) realizado por meio da implementação de um Sistema de Vendas**. Em: III Congreso Internacional de Computación y Telecomunicaciones - COMTEL, 2011, Lima, Perú. Memoria: COMTEL 2011. Lima, Perú : Fondo Editorial de la UIGV, p. 53-56, 2011.

BELMONTE, D.; SIMÃO, J. M.; STADZISZ, P. C. **Proposta de um Método para Distribuição da Carga de Trabalho Usando o Paradigma Orientado a Notificações (PON)**. Revista SODEBRAS, 7(84), p. 10-17, 2012.

BELMONTE, D. **Método para Distribuição da Carga de Trabalho dos Softwares PON em Multicore**. Trabalho de Qualificação de Doutorado, CPGEI, UTFPR. Curitiba, Brasil, 2012.

BJERREGAARD, T.; MAHADEVAN, S. **A Survey of Research and Practices of Network-on-Chip**. ACM Computing Surveys, 38(1), 2006.

BITTNER, R. A.; ATHANAS, P. M.; MUSGROVE, M. D. **Colt : An Experiment in Wormhole Run-Time Reconfiguration**. Proceedings of SPIE Photonics East (p. 187-195). Boston, MA, USA, 1996.

BOBDA, C.; MAHR, P.; ANDRES, B.; ISHEBABI, H. **Application-driven architecture synthesis of on-chip Multiprocessor systems**. 2010 International Conference on High Performance Computing & Simulation, p. 591-598, 2010.

BORKAR, S.; CHIEN, A. A. **The Future of Microprocessors**. Communications of the ACM, 54(5), p. 67-77, 2011.

BURGER, D.; KECLER, S. W.; MCKINLEY, K. S.; DAHLIN, M.; JOHN, L. K.; LIN, C., et al. **Scaling to the End of Silicon with EDGE Architectures**. IEEE Computer, 37(7), p. 44-55, 2004.

CARLSTRÖM, J.; BODÉN, T. **Synchronous Dataflow Architecture for Network Processors**. IEEE Micro, 24(5), p. 10-18, 2004.

CATANZARO, B.; FOX, A.; KEUTZER, K.; PATTERSON, D.; SU, B.-Y.; SNIR, M., et al. **Ubiquitous Parallel Computing From Berkeley, Illinois and Stanford**. IEEE Micro, 30(2), p. 41-55, 2010.

CAVIN, R.; HUTCHBY, J. A.; ZHIRNOV, V.; BREWER, J. E.; BOURIANOFF, G. **Emerging Research Architectures**. Computer, 41(5), p. 33-37, 2008.

CHAMBERLAIN, B. L.; CALLAHAN, D.; ZIMA, H. P. **Parallel Programmability and the Chapel Language**. International Journal of High Performance Computing Applications, 21(3), p. 291-312, 2007.

CHAUVIN, S.; SAHA, P.; CANTONNET, F.; ANNAREDDY, S.; EL-GHAZAWI, T. **UPC Manual**. 2005. Disponível em: <http://upc.gwu.edu/downloads/Manual-1.2.pdf>. Acesso em 23/05/ 2012.

CHEN, Y.; FAN, B.; ZHONG, L.; WU, C. **Diva: A Dataflow Programming Model and its Runtime Support in Java Virtual Machine**. 13th Asia-Pacific Computer Systems Architecture Conference (ACSAC '08). Hsinchu, Taiwan, 2008.

CHOU, Y.; FAHS, B.; ABRAHAM, S. **Microarchitecture Optimizations for Exploiting Memory-Level Parallelism**. Proceedings of the 31st International Symposium on Computer Architecture. Washington, DC, USA: IEEE Computer Society, 2004.

COMPTON, K.; HAUCK, S. **Reconfigurable Computing : A Survey of Systems and Software**. ACM Computing Surveys, 34(2), p. 171-210, 2002.

CULLER, D. E.; ARVIND. **Resource Requirements of Dataflow Programs**. Proceedings of the 15th Annual International Symposium on Computer architecture (ISCA '88) (p. 141-150). Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1988.

CULLER, D. E.; EICKEN, T. V.; SCHAUSER, K. E. **Two Fundamental Limits on Dataflow Multiprocessing**. Technical Report UCB/CSD-92-716, EECS Department, University of California, Berkeley, 1992.

CULLER, D. E.; GOLDSTEIN, S. C.; SCHAUSER, K. E.; EICKEN, T. V. **TAM - A Compiler Controlled Threaded Abstract Machine**. Journal of Parallel and Distributed Computing Practices, 18(3), p. 347-370, 1993.

DENNING, P. J.; DENNIS, J. B. (2010). **The Resurgence of Parallelism**. Communications of the ACM 53, p. 30-32, 2010.

DENNIS, J. B.; MISUNAS, D. P. **A preliminary architecture for a basic data-flow processor**. Em: *Proceedings of the 2nd annual symposium on Computer architecture* (ISCA '75). ACM, New York, NY, USA, p. 126-132, 1974.

DEITEL, P. J.; DEITEL, H. M. **Java – Como Programar**. 8ª Edição. Prentice Hall, 2010.

DIGITAL EQUIPMENT CORPORATION. **Digital High Performance Fortran 90 HPF and PSE Manual**. Disponível em: <http://www.mun.ca/hpc/hpf_pse/manual/hpf.htm#book-toc>. Acesso em 23/05/2012.

DONGARRA, J. J. **Performance of Various Computers Using Standard Linear Equations Software**. Technical Report, University Of Tennessee. Knoxville, TN, 2011.

DUESTERWALD, E.; GEAY, E.; SARASWAT, V.; GROVE, D. **An Overview of the X10 Programming Language and X10 Development Tools**. SuperComputing 2010. Disponível em: <http://x10-lang.org/documentation/tutorials.html?id=160>.

EL-GHAZAWI, T. **UPC Tutorial**. 2009. Apresentação no PGAS 09. Disponível em: <http://www2.hpcl.gwu.edu/pgas09/tutorials/upc_tut.pdf>. Acesso em 23/05/2012.

EMBEDDED MICROPROCESSOR BENCHMARK CONSORTIUM. **EEMBC – The Embedded Microprocessor Benchmark Consortium**. Disponível em: <http://www.eembc.org>. Acesso em 26/06/2012.

ESCHMANN, F.; KLAUER, B.; MOORE, R.; WALDSCHMIDT, K. **SDAARC : An Extended Cache-Only Memory Architecture**. IEEE Micro, 22(3), p. 62-70, 2002.

FARABET, C.; MARTINI, B.; CORDA, B.; AKSELROD, P.; CULURCIELLO, E.; LECUN, Y. **NeuFlow : A Runtime Reconfigurable Dataflow Processor for Vision**. Fifth IEEE Workshop on Embedded Computer Vision (ECV'11 @ CVPR'11). Colorado Springs, 2011.

FATAHALIAN, K.; HORN, D. R.; KNIGHT, T. J.; LEEM, L.; HOUSTON, M.; PARK, J. Y. et al. **Sequoia : Programming the Memory Hierarchy**. Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC '06). New York, NY, USA, 2006.

FELDMAN, M. **Sun's Fortress Language: Parallelism By Default**. 2008. HPC Wire. Disponível em: <http://www.hpcwire.com/hpcwire/2008-07-16/suns_fortress_language_parallelism_by_default.html>. Acesso em 24/05/2012.

FERLIN, E. P. **Avaliação de métodos de paralelização automática**. 1997. Dissertação (Mestrado em Física Aplicada) - Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos, 1997. Disponível em: <http://www.teses.usp.br/teses/disponiveis/76/76132/tde-09102008-111750/>. Acesso em 03/12/2012.

FERLIN, E. P. **Arquitetura Paralela Reconfigurável Baseada em Fluxo de Dados Implementada em FPGA**. 2008. Tese de Doutorado. CPGEI, UTFPR, Curitiba, Brasil.

FERLIN, E. P.; LOPES, H. S.; LIMA, C. R. E.; PERRETTO, M. **PRADA: a high-performance reconfigurable parallel architecture based on the dataflow model**. International Journal of High Performance Systems Architecture, 3(1), p. 41-55, 2011.

FLYNN, M. J. **Very High-speed Computing Systems**. Proceedings of the IEEE, 54(12), p. 1901-1909, 1966.

FORGY, C. **RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem**. Artificial Intelligence, 19(1), p. 17-37, 1982.

FREITAS, H. C. DE; ALVES, M. A. Z.; NAVAUX, P. O. A. **NoC e NUCA - Conceitos e Tendências para Arquiteturas de Processadores Many-Core**. ERAD - Escola Regional de Alto Desempenho, p. 5-37, 2009.

FRIEDMAN-HILL, E. **Jess in Action: Rule Based System in Java**. Greenwich, CT, USA: Manning Publications Co, 2003.

GLEW, A. **MLP yes ! ILP no !** ASPLOS Wild and Crazy Idea Session ’98, 1998.

GODFREY, M. D.; HENDRY, D. F. **The Computer As Von Neumann Planned It**. IEEE Annals of the History Of Computing, 15(1), p. 11-21, 1993.

GREEN, S. **Systematic reviews and meta-analysis**. Singapore Medical Journal, 46(6), p. 270-274, 2005.

GSCHWIND, M. **Chip Multiprocessing and the Cell Broadband Engine**. Proceedings of the 3rd conference on Computing frontiers (CF ’06) (p. 1-8). New York, NY, USA: ACM, 2006.

GUPTA, G.; SOHI, G. S. **Dataflow Execution of Sequential Imperative Programs on Multicore Architectures**. Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44 ’11) (p. 59-70). New York, NY, USA: ACM, 2011.

GURD, J. R.; KIRKHAM, C. C.; WATSON, I. **The Manchester Prototype Dataflow Computer**. Communications of the ACM, 28(1), 1985.

HAAVIND, R. **Editorial: It’s springtime for creative engineers**. Solid State Technology, 51(4), p. 10-10, 2008.

HAMMOND, L.; NAYFEH, B. A.; OLUKOTUN, K. **A Single-Chip Multiprocessor**. IEEE Computer, 30(9), p. 79-85, 1997.

HAMMOND, L.; WONG, V.; CHEN, M.; CARLSTROM, B. D.; DAVIS, J. D.; HERTZBERG, B.; et al. **Transactional memory coherence and consistency**. 31st Annual International Symposium on Computer Architecture. Washington, DC, USA, 2004.

HAUSER, J. R.; WAWRZYNEK, J. **Garp : A MIPS Processor with a Reconfigurable Coprocessor**. Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM ’97), 1997.

HENNESSY, J. L.; PATTERSON, D. A. **Computer architecture: a quantitative approach**. 4a Ed. Amsterdam; Boston: Morgan Kaufmann, 2007.

HEWITT, C. **Middle History Of Logic Programming**. 2012. CoRR, abs/0904.3.Disponível em: <http://arxiv.org/abs/0904.3036>.

HISTORY Of Software Productivity. Disponível em: <http://www.*software*metrics.com/Articles/history.htm >. Acesso em 27/06/2012.

HUDAK, D. **Module 1 : X10 Overview. The X10 Language and Methods for Advanced HPC Programming**. 2011. Disponível em: <http://www.osc.edu/~dhudak/Site/Home_files/x10-tutorial.pdf>. Acesso em 24/05/2012.

HURSON, A. R.; KAVI, K. M. **Dataflow Computers : Their History and Future**. 2008. Wiley Online Library. John Wiley & Sons, Inc. Disponível em: http://onlinelibrary.wiley.com/doi/10.1002/9780470050118.ecse102/full>.

IANNUCCI, R. A. (1988). **Toward a dataflow / von neumann hybrid architecture**. SIGARCH Computer Architecture News, 16(2), p. 131-140, 1988.

INTEL CORPORATION. **Intel ® 64 and IA-32 Architectures Software Developer s Manual**. 2012. Disponível em: <http://www.intel.com/content/www/us/en/processors/architectures-*software*-developer-manuals.html>.

INTEL CORPORATION. **Mobile 3rd Generation Intel® Core™ Processor Family: Product Brief**. Disponível em: <http://www.intel.com.br/content/www/br/pt/processors/core/3rd-gen-core-family-mobile-brief.html>. Acesso em 10/11/2012.

IRWIN, M. J.; SHEN, J. P. **Revitalizing Computer Architecture Research**. 2005. Computing. Disponível em: <http://www.cra.org/uploads/documents/resources/rissues/computer.architecture_.pdf>.

JASINSKI, R. P. **Framework para Geração de *Hardware* em VHDL a Partir de Modelos em PON (Paradigma Orientado a Notificações**). Relatório da disciplina de Lógica Reconfigurável por *Hardware*. Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. Universidade Tecnológica Federal do Paraná, 2012.

JOHNSTON, W. M.; HANNA, J. R. P.; MILLAR, R. J. **Advances in dataflow programming languages**. ACM Computing Surveys, 36(1), p. 1-34, 2004.

KAHLE, J. A.; DAY, M. N.; HOFSTEE, H. P.; JOHNS, C. R.; MAEURER, T. R.; SHIPPY, D. **Introduction to the Cell multiprocessor**. IBM Journal Of Research and Development, 49(4), p. 589-604, 2005.

KALTE, H.; LANGEN, D.; VONNAHME, E.; BRINKMANN, A.; RÜCKERT, U. **Dynamically Reconfigurable System-on-Programmable-Chip**. Proceedings of the 10th Euromicro conference on Parallel, distributed and network-based processing (EUROMICRO-PDP'02), p. 235-242, 2002.

KAVI, K. M.; GIORGI, R.; ARUL, J. **Scheduled Dataflow: Execution Paradigm, Architecture, and Performance Evaluation**. IEEE Transactions on Computers, 50(8), p. 834-846, 2001.

KEETON, K.; PATTERSON, D. A.; HE, Y. Q.; RAPHAEL, R. C.; BAKER, W. E. **Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads**. Proceedings of the 25th annual international symposium on Computer architecture (ISCA '98), p. 15-26. Washington, DC, USA: IEEE Computer Society, 1998.

KIM, C.; BURGER, D.; KECKLER, S. W. **An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches**. Proceedings of the 10th international conference on Architectural support for programming languages and operating systems (ASPLOS-X), p. 211-222. New York, NY, USA: ACM, 2002.

KLAUER, B.; ESCHMANN, F.; MOORE, R.; WALDSCHMIDT, K. **The CDAG: a data structure for automatic parallelization for a multithreaded architecture**. Proceedings 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing, p. 219-226. IEEE Comput. Soc., 2002.

KOGGE, P.; BERGMAN, K.; BORKAR, S.; CAMPBELL, D.; CARLSON, W.; DALLY, W.; et al. **ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems**. Setembro de 2008, DARPA. Disponível em: <http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf>.

KONGETIRA, P.; AINGARAN, K.; OLUKOTUN, K. **Niagara: A 32-Way Multithreaded SPARC Processor**. IEEE Micro, (25), p. 21-29, 2005.

KUNG, H. T. **Why Systolic Architectures ?** Computer, 15(1), p. 37-46, 1982.

KYRIACOU, C.; EVRIPIDOU, P.; TRANCOSO, P. **Data-Driven Multithreading Using Conventional Microprocessors**. IEEE Transactions On Parallel And Distributed Systems, 17(10), p. 1176-1188, 2006.

LAPACK – Linear Algebra PACKage. Disponível em: <http://www.netlib.org/lapack/>. Acesso em 17/10/2012.

LEE, P.-Y.; CHENG, A. M. K. **HAL: a faster match algorithm**. IEEE Transactions on Knowledge and Data Engineering, 14(5), p. 1047-1058, 2002.

LEISERSON, C. E. **Multithreaded Programming in Cilk - Lecture 1 (p. 1-59)**. 2006. Disponível em: <http://supertech.csail.mit.edu/cilk/lecture-1.pdf>. Acesso em 24/05/2012.

LIAO, Y. **Neural Networks in *Hardware*: A Survey**. Technical Report. University Of California. Davis, CA, USA, 2001.

LING, X.-P.; AMANO, H. (1993). **WASMII: a Data Driven Computer on a Virtual *Hardware***. Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (p. 33-42), 1993.

LINHARES, R. R. **Modelamento de *Hardware* Visando À Estimação do Tempo de Execução de Programas**. 2001. Dissertação de Mestrado, CPGEI, CEFET-PR. Curitiba, Brasil, 2001.

LINHARES, R. R.; RONSZCKA, A. F.; VALENCA, G. Z.; BATISTA, M. V.; WITT, F. A.; LIMA, C. R. E.; SIMAO, 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**. En: III Congreso Internacional de Computación y Telecomunicaciones - COMTEL, 2011, Lima, Perú. Memoria: COMTEL 2011. Lima, Perú : Fondo Editorial de la UIGV, 2011. p. 103-106.

LIU, Y.; FURBER, S. **A low power embedded dataflow coprocessor**. Proceedings of the IEEE Computer Society Annual Symposium on VLSI (p. 246-247), 2005.

LIU, J.; LIANG, D. **A Survey of FPGA-Based *Hardware* Implementation of ANNs**. ICNN&B ’05. International Conference on Neural Networks and Brain, 2005, 2, p. 915-918.

MADOŠ, B.; & BALÁŽ, A. **Data Flow Graph Mapping Techniques of Computer Architecture with Data Driven Computation Model**. Electrical Engineering, p. 355-359, 2011.

MARCUELLO, P.; GONZALEZ, A.; TUBELLA, J. **Speculative Multithreaded Processors**. International Conference on Supercomputing, p. 77-84, 1998.

MAK, V. W. K. **A Survey Of Concurrent Architectures**. Technical Report. Stanford, CA, USA, 1986.

MCKEE, S. A. **Reflections on the Memory Wall**. Em ACM (Eds.), Proceedings of the 1st Conference on Computing Frontiers (CF ’04). New York, USA, 2004.

MENDELEY LTD. **Overview | Mendeley**. Disponível em: <http://www.mendeley.com/features/>. Acesso em 06/11/2012.

MICROSOFT CORPORATION. **Process And Thread Functions**. 2012. Disponível em: <http://msdn.microsoft.com/en-us/library/windows/desktop/ms684847(v=vs.85).aspx>. Acesso em 25/10/2012.

MIRANKER, D. P. **TREAT : A Better Match Algorithm for AI Production Systems**. Sixth National Conference On Artificial Intelligence - AAAI 87 (p. 42-47), 1987.

MIRANKER, D. P.; BRANT, D. A.; LOFASO, B.; GADBOIS, D. **On The Performance Of Lazy Matching In Production Systems**. 8th National Conference on Artificial Intelligence AAAI (p. 685-692). AAAI Press / The MIT Press, 1990.

MIRANKER, D. P.; LOFASO, B. **The organization and Performance of a TREAT-based Production System Compiler**. IEEE Transactions on Knowledge and Data Engineering, 3(1), p. 3-10, 1991.

MITIĆ, M.; STOJČEV, M. **An Overview of On-Chip Buses**. 2006. University of Niš. Vol. 19, p. 405-428. Disponível em: <http://facta.junis.ni.ac.rs/eae/fu2k63/stojcev.pdf>. Acesso em 28/12/2012.

MOORE, G. E. **Cramming More Components onto Integrated Circuits**. Electronics, 38(8), p. 114-117, 1965.

MOORE, S. K. **Multicore Is Bad News For Supercomputers**. IEEE Spectrum Online. November 2008. Disponível em: <http://spectrum.ieee.org/computing/*hardware*/multicore-is-bad-news-for-supercomputers>. Acesso em 26/11/2012.

MURPHY, R. **On the Effects of Memory Latency and Bandwidth on Supercomputer Application Performance**. Proceedings of the 2007 IEEE 10th International Symposium on

Workload Characterization (IISWC '07) (p. 35-43). Washington, DC, USA: IEEE Computer Society, 2007.

NAJJAR, W. A.; BÖHM, A. P. W.; MILLER, W. M. **A Model for Dataflow Based Vector Execution**. Proceedings of the 8th international conference on Supercomputing (ICS '94), p. 11-22, 1994.

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

NIKHIL, R. S. **Can dataflow subsume von Neumann computing?** Proceedings of the 16th annual international symposium on Computer architecture (ISCA '89) (p. 262-272). New York, NY, USA: ACM, 1989.

NIKHIL, R. S.; PINGALI, K. K.; ARVIND. **I-Structures : Data Structures for Parallel Computing**. ACM Transactions on Programming Languages and Systems, 11(4), p. 598-632, 1989.

OAK RIDGE NATIONAL LABORATORY. **PVM – Parallel Virtual Machine**. Disponível em: <http://www.csm.ornl.gov/pvm/>. Acesso em 29/08/2012.

ORACLE CORPORATION. **Fortress FAQ**. Disponível em: <http://labs.oracle.com/projects/plrg/Fortress/faq.html>. Acesso em 24/05/2012.

PAPADOPOULOS, G. M. **Implementation of a General Purpose Dataflow Multiprocessor**. Electrical Engineering. MIT Press Cambridge, MA, 1988.

PAPADOPOULOS, G. M.; CULLER, D. E. **Monsoon: an Explicit Token-Store Architecture**. Proceedings of the 17th annual international symposium on Computer Architecture (ISCA '90) (p. 82-91). New York, NY, USA: ACM, 1990.

PAPAMARCOS, M. S.; PATEL, J. H. **A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories**. Proceedings of the 11th annual international symposium on Computer architecture (ISCA '84) (p. 348-354). New York, NY, USA: ACM, 1984.

PAS, R. V. D. **An Introduction Into OpenMP**. IWOMP 2005. Eugene, Oregon, USA, 2005. Disponível em:< http://www.nic.uoregon.edu/iwomp2005/iwomp2005_tutorial_openmp_rvdp.pdf>.

PAS, R. V. D. **An Overview of OpenMP**. NTU Talk 2009. Singapore, 2009. Disponível em: <http://openmp.org/mp-documents/ntu-vanderpas.pdf>.

PATTERSON, D. A. **Reduced instruction set computers**. Communications of the ACM, 28(1), 1985.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization & Design: The *Hardware* / Software Interface**. 4a Edição. Morgan Kaufmann Publishers, Inc. San Francisco, CA, USA, 2009

PEDRONI, V. **Eletrônica Digital Moderna com VHDL**. Rio de Janeiro, RJ: Elsevier, 2010.

PETERS, E. **Coprocessador para Aceleração de Aplicações Desenvolvidas Utilizando Paradigma Orientado a Notificações**. 2012. Dissertação de Mestrado, CPGEI, UTFPR. Curitiba, Brasil, 2012.

PETITET, A.; WHALEY, R. C.; DONGARRA, J.; CLEARY, A. **HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers**. 2008. Disponível em: <http://www.netlib.org/benchmark/hpl/>. Acesso em 17/10/2012.

PILLA, M. L.; SANTOS, R. R.; CAVALHEIRO, G. G. **Introdução à programação para arquiteturas multicore**. ERAD - Escola Regional de Alto Desempenho. Caxias do Sul, Brasil, 2009.

PRESSMAN, R. **Engenharia de Software**. São Paulo: Makron Books, 1995.

RANDALL, K. H. **Cilk: Efficient Multithreaded Computing**. PhD Thesis. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1998.

RONSZCKA, A. F. **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.

RONSZCKA, A. F.; BELMONTE, D. L.; VALENCA, G. Z.; BATISTA, M. V.; LINHARES, R. R.; TACLA, C. A.; STADZISZ, P. C.; SIMAO, J. M. **Comparações quantitativas e qualitativas entre o Paradigma Orientado a Objetos e o Paradigma Orientado a Notificações sobre um simulador de jogo**. Em: III Congreso Internacional de Computación y Telecomunicaciones - COMTEL, 2011, Lima, Perú. Memoria: COMTEL 2011. Lima, Perú : Fondo Editorial de la UIGV, 2011. p. 61-64.

RONSZCKA, A. F. **Contribuição Para a Concepção de Aplicações no Paradigma Orientado a Notificacões (PON) Sob o Viés de Padrões**. 2012. Dissertação de Mestrado, CPGEI, UTFPR. Curitiba, Brasil, 2012. Disponível em

http://files.dirppg.ct.utfpr.edu.br/cpgei/Ano_2012/dissertacoes/CPGEI_Dissertacao_608_2012.pdf.

SAKAI, S.; YAMAGUCHI, Y.; HIRAKI, K.; KODAMA, Y.; YUBA, T. **An Architecture of a Dataflow Single Chip Processor**. Proceedings of the 16th annual international symposium on Computer architecture (ISCA ’89) (p. 46-53). New York, NY, USA: ACM, 1989.

SAMSUNG CORPORATION. **Samsung Exynos**. Disponível em: <http://www.samsung.com/global/business/semiconductor/minisite/Exynos/products4quad.html>. Acesso em 10/11/2012.

SANKARALINGAM, K.; NAGARAJAN, R.; LIU, H.; KIM, C.; HUH, J.; BURGER, D., et al. **Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture**. Proceedings of the 30th annual international symposium on Computer architecture (ISCA ’03) (p. 422-433). New York, NY, USA: ACM, 2003.

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

SCOTT, S. **Future Supercomputer Architectures**. Frontiers of Extreme Computing (p. 1-22), 2007.

SIMÃO, J. M. **Proposta de uma Arquitetura de Controle para Sistemas Flexíveis de Manufatura Baseada em Regras e Agentes**. 2001. Dissertação de Mestrado, Universidade Tecnológica Federal do Paraná - UTFPR, Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial - CPGEI, Curitiba, 2001.

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

SIMÃO, J. M. **A Contribution To The Development Of A HMS Simulation Tool And Proposition Of A Meta-Model For Holonic Control**. 2005. Tese de doutorado. CPGEI, CEFET-PR. Curitiba, Brasil, 2005.

SIMÃO, J. M.; STADZISZ, P.C.; MOREL, G. **Manufacturing Execution System for Customized Production**. Journal of Material Processing Technology , 179 (1-3), 268, 2006.

SIMÃO, J. M. ; STADZISZ, P. C. **Paradigma Orientado a Notificações (PON) - Uma Técnica de Composição e Execução de Software Orientada a Notificações**. 2008, Brasil. Patente: Privilégio de Inovação. Número do registro: PI08055181, data de depósito: 26/11/2008, título: "PEDIDO DE PATENTE: Paradigma Orientado a Notificações (PON) Uma Técnica de Composição e Execução de Software Orientada a Notificações." , Instituição de registro:INPI - Instituto Nacional da Propriedade Industrial. Instituição(ões) financiadora(s): Universidade Tecnológica Federal do Paraná.

SIMÃO, J. M.; STADZISZ, P. C.; BANASZEWSKI, R. F.; TACLA, C. A. **Holonic Manufacturing Execution Systems for Customised and Agile Production: Manufacturing Plant Simulation**. V Congresso Brasileiro de Engenharia de Fabricação, 2008.

SIMÃO, J. M.; STADZISZ, P. C. **Inference Based on Notifications: A Holonic Metamodel Applied to Control Issues**. IEEE Transactions on Systems, Man and Cybernetics. Part A, Systems and Humans, v. 39, p. 238-250, 2009.

SIMÃO, J. M.; STADZISZ, P. C. **Mecanismo de Resolução de Conflito e Garantia de Determinismo para o Paradigma Orientado a Notificações (PON)**. 2010, Brasil. Patente: Privilégio de Inovação. Número do registro: PI10002960, data de depósito: 26/02/2010, título: "PEDIDO DE PATENTE: Mecanismo de Resolução de Conflito e Garantia de Determinismo para o Paradigma Orientado a Notificações (PON)" , Instituição de registro:INPI - Instituto Nacional da Propriedade Industrial. Instituição(ões) financiadora(s): Universidade Tecnológica Federal do Paraná.

SIMÃO, J. M. ; BANASZEWSKI, R. F. ; TACLA, C. A. ; 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**. 2010, Brasil. Patente: Privilégio de Inovação. Número do registro: PI10037365, data

de depósito: 25/03/2010, título: "PEDIDO DE PATENTE: Mecanismo de Inferência Otimizado do Paradigma Orientado a Notificações (PON) e Mecanismos de Resolução de Conflitos para Ambientes Monoprocessados e Multiprocessados Aplicados ao PON" , Instituição de registro:INPI - Instituto Nacional da Propriedade Industrial. Instituição(ões) financiadora(s): Universidade Tecnológica Federal do Paraná, 2010.

SIMÃO, J. M.; TACLA, C. A.; STADZISZ, P. C.; BANASZEWSKI, R. F. **Notification Oriented Paradigm (NOP) and Imperative Paradigm : A Comparative Study**. Journal of Software Engineering and Applications, 5(6), p. 402-416, 2012a.

SIMÃO, J. M. ; LINHARES, R. R. ; WITT, F. A. ; LIMA, C. R. E. ; STADZISZ, P. C. **Paradigma Orientado a Notificações em *Hardware* Digital**. 2012, Brasil. Patente: Privilégio de Inovação. Número do registro: BR102012026429, data de depósito: 16/10/2012, título: "PEDIDO DE PATENTE: Paradigma Orientado a Notificações em Hardware Digital" , Instituição de registro:INPI - Instituto Nacional da Propriedade Industrial. Instituição(ões) financiadora(s): UTFPR, 2012b.

SIMÃO, J. M., STADZISZ, P. C., TACLA, C. A., LINHARES, R. R., BELMONTE, D. L., & BANASZEWSKI, R. F. **Comparações entre duas materializações do Paradigma Orientado a Notificações (PON): Framework PON Prototipal versus Framework PON Primário**. Em: IV Congreso Internacional de Computación y Telecomunicaciones - COMTEL 2012. Lima, Peru: Universidad Inca Garcilaso de la Vega, 2012, p. 13-20, 2012c.

SIMÃO, J. M.; STADZISZ, P. C.; WIECHETECK, L. V. B. **Perfil UML para o Paradigma Orientado a Notificações (PON), Perfil UML para o Paradigma Orientado a Regras (POR), Método de Desenvolvimento Orientado a Notificações (DON) e Método de Desenvolvimento Orientado a Regras (DOR)**. 2012, Brasil. Patente: Privilégio de Inovação. Número do registro: BR1020120264307, data de depósito: 16/10/2012, título: "PEDIDO DE PATENTE: Perfil UML para o Paradigma Orientado a Notificações (PON), Perfil UML para o Paradigma Orientado a Regras (POR), Método de Desenvolvimento Orientado a Notificações (DON) e Método de Desenvolvimento Orientado a Regras (DOR)" , Instituição de registro:INPI - Instituto Nacional da Propriedade Industrial. Instituição(ões) financiadora(s): UTFPR, 2012.

SISAL Lives. Disponível em: <http://sisal.sourceforge.net>. Acesso em 15/08/2012.

SMITH, B. **A pipelined, shared resource MIMD computer**. In *Advanced computer architecture*, Dharma P Agrawal (Ed.). IEEE Computer Society Press, Los Alamitos, CA, USA p. 39-41, 1986.

STANDARD PERFORMANCE EVALUATION CORPORATION. **SPEC CPU2006**. Disponível em: <www.spec.org/cpu2006/>. Acesso em 26/06/2012.

SUN, F.; RAVI, S.; RAGHUNATHAN, A.; JHA, N. K. **Application-Specific Heterogeneous Multiprocessor Synthesis Using Extensible Processors**. Computer-Aided Design, 25(9), p. 1589-1602, 2006.

SWANSON, S.; MICHELSON, K.; SCHWERIN, A.; OSKIN, M. **WaveScalar**. Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36). Washington, DC, USA: IEEE Comput. Soc., 2003.

TALLA, D.; JOHN, L. K.; BURGER, D. (2003). **Bottlenecks in Multimedia Processing with SIMD style Extensions and Architectural Enhancements**. IEEE Transactions on Computers, 52(8), p. 1015-1031, 2003.

TANENBAUM, A. S. **Organização Estruturada de Computadores**. 5a edição. São Paulo, SP: Pearson, 2007.

TEIFEL, J.; MANOHAR, R. **An Asynchronous Dataflow FPGA Architecture**. IEEE Transactions on Computers, 53(11), p. 1376-1392, 2004.

TULLSEN, D. M.; EGGERS, S. J.; LEVY, H. M. **Simultaneous Multithreading : Maximizing On-Chip Parallelism**. 25 years of the international symposia on Computer architecture (selected papers) (ISCA '98) (p. 533-544), 1998.

UNGERER, T.; SILC, J.; ROBIC, B. **Asynchrony in parallel computing: from dataflow to multithreading**. Journal of Parallel and Distributed Computing Practices, 1, p. 1-33, 1998.

UNGERER, T.; ROBIC, B.; SILC, J. **A Survey of Processors with Explicit Multithreading**. ACM Computing Surveys, 35(1), p. 29-63, 2003.

VAHEY, M.; GRANACKI, J.; LEWINS, L.; DAVIDOFF, D.; DRAPER, J.; STEELE, C.; GROVES, G.; KRAMER, M.; LACOSS, J.; PRAGER, K.; KULP, J.; CHANNELL, C. **MONARCH: a first generation polymorphic computing processor**. Abstract approved for public release. Proceedings of 10th Annual Workshop on High Performance Embedded Computing, Boston, MA, USA, 2006.

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**. Em: III Simpósio de Computação Aplicada, 2011, Passo Fundo - RS. III Simpósio de Computação Aplicada, 2011.

VALENÇA, G. Z. **Contribuição para Materialização do Paradigma Orientado a Notificações (PON) Via Framework e Wizard**. 2012. Dissertação de Mestrado, Programa de Pós-Graduação em Computação Aplicada (PPGCA), UTFPR. Curitiba, Brasil, 2012.

WAINGOLD, E.; TAYLOR, M.; SRIKRISHNA, D.; SARKAR, V.; LEE, W.; LEE, V., et al. **Baring It All to Software: Raw Machines**. Computer. Cambridge, MA, USA, 1997.

WALL, D. W. **Limits of Instruction-Level Parallelism**. SIGARCH Computer Architecture News, 19(2), p. 176-188, 1991.

WANG, L. **Performance Study of Chip Multiprocessors with Integrated DRAM**. 2001. Master Thesis. Department of Electrical and Computer Engineering, Queen's University. Kingston, Ontario, Canada, 2001.

WANG, S.; WANG, L. **Thread-Associative Memory for Multicore and Multithreaded Computing**. Proceedings of the 2006 international symposium on Low power electronics and design (ISLPED '06) (p. 139-142), 2006.

WATT, D. **Programming Language Design Concepts**. J. Willey & Sons, 2004.

WAZLAWICK, R. S. **Metodologia de Pesquisa para Ciência da Computação**. Rio de Janeiro, RJ: Elsevier, 2008.

WEBER, L.; BELMONTE, D. L.; BANASZEWSKI, R. F.; STADZISZ, P. C.; SIMÃO, J. M. **Viabilidade De Controle Orientado A Notificações (CON) Em Ambiente Concorrente Baseado Em Threads**. SICITE, Brasil, 2010. Disponível em: <http://eventos.cp.utfpr.edu.br/index.php/sicite/2010/paper/view/1252>. Acesso em 25/10/2012.

WENTZLAFF, D.; GRIFFIN, P.; HOFFMANN, H.; BAO, L.; EDWARDS, B.; RAMEY, C., et al. **On-Chip Interconnection Architecture of the Tile Processor**. IEEE Micro, 27(5), p. 15-31, 2007.

WIECHETECK, L. V. B.; STADZISZ, P. C.; SIMÃO, J. M. **Um Perfil UML para o Paradigma Orientado a Notificações (PON)**. Em: III Congreso Internacional de Computación y Telecomunicaciones - COMTEL 2011, 2011, Peru - Lima. III Congreso Internacional de Computación y Telecomunicaciones - COMTEL 2011, 2011.

WIECHETECK, L. V. B. **Método para Projeto de Software usando o Paradigma Orientado a Notificações – PON**. 2011. Dissertação de Mestrado. CPGEI, UTFPR. Curitiba, Brasil, 2011. Disponível em http://files.dirppg.ct.utfpr.edu.br/cpgei/Ano_2011/dissertacoes/CPGEI_Dissertacao_578_2011.pdf.

WITT, F. A.; SIMAO, J. M.; LINHARES, R. R.; STADZISZ, P. C.; LIMA, C. R. E. **Comparação entre o Paradigma Orientado a Objetos (POO) e o Paradigma Orientado a Notificações (PON) em um Controle Discreto em Lógica Reconfigurável**. Em: XVI SICITE - Seminário de Iniciação Científica e Tecnológica da UTFPR, 2011, Ponta Grossa - PR. Anais do XVI SICITE, 2011.

WOLF, W. **A Decade of *Hardware* / Software Codesign**. Computer, 36(4), p. 38-43, 2003.

WOO, S. C.; OHARA, M.; TORRIE, E.; SINGH, J. P.; GUPTA, A. **The SPLASH-2 Programs : Characterization and Methodological Considerations**. SIGARCH Computer Architecture News, 23(2), p. 24-36, 1995.

XPP TECHNOLOGIES. **XPP-III Processor Overview White Paper**. 2006. Disponível em: <http://www.cs.washington.edu/education/courses/cse591n/06au/papers/XPP-III_overview_WP.pdf>.

ZEA, N.; SARTORI, J.; KUMAR, R. **Servo: A Programming Model for Many-core Computing**. ACM SIGARCH Computer Architecture News, 36(2), p. 28-37, 2008.