UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
UNIVERSIDADE DO ESTADO DO RIO GRANDE DO NORTE
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DISSERTAÇÃO DE MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

Dênis Freire Lopes Nunes

ESTENDENDO O CONJUNTO DE INSTRUÇÕES DA IPNOSYS PARA IMPLEMENTAÇÃO DE SOFTWARE *PIPELINING*

Mossoró/RN
2016

Dênis Freire Lopes Nunes

ESTENDENDO O CONJUNTO DE INSTRUÇÕES DA IPNOSYS PARA IMPLEMENTAÇÃO DE SOFTWARE *PIPELINING*

Dissertação apresentada ao Mestrado em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação da Universidade Federal Rural do Semi-Árido e Universidade do Estado do Rio Grande do Norte como requisito para obtenção do título de Mestre em Ciência da Computação.

Linha de Pesquisa:
Projeto de Sistemas e Circuitos

Orientador:
Silvio Roberto F. de Araújo, Prof. Dr.

Coorientador:
Leonardo Augusto Casillo, Prof. Dr.

Mossoró/RN
2016

Dados Internacionais de Catalogação na Publicação (CIP)
BIBLIOTECA CENTRAL ORLANDO TEIXEIRA - CAMPUS MOSSORÓ
Setor de Informação e Referência

N972e Nunes, Denis Freire Lopes.

Estendendo o conjunto de instruções da IPNoSys para implementação de software pipelining / Denis Freire Lopes Nunes. - Mossoró, 2016.
96f: il.

Orientador: Prof. Dr. Silvio Roberto Fernandes de Araújo
Co-Orientador: Prof. Dr. Leonardo Augusto Casillo

Dissertação (MESTRADO EM CIÊNCIA DA COMPUTAÇÃO) - Universidade Federal Rural do Semi-Árido. Pró-Reitoria de Pesquisa e Pós-Graduação

1. Arquitetura de computadores. 2. IPNoSys. 3. Software pipelining. I. Título

RN/UFERSA/BOT/050 CDD 004.22

Dênis Freire Lopes Nunes

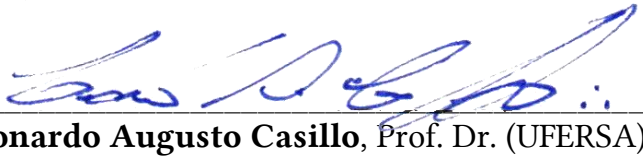
ESTENDENDO O CONJUNTO DE INSTRUÇÕES DA IPNOSYS PARA IMPLEMENTAÇÃO DE SOFTWARE *PIPELINING*

Dissertação apresentada ao Mestrado em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação da Universidade Federal Rural do Semi-Árido como requisito para obtenção do título de Mestre em Ciência da Computação.

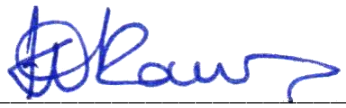
Linha de Pesquisa:
Projeto de Sistemas e Circuitos

Defendida em: 25 / 02 / 2016.

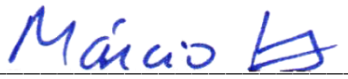
Silvio Roberto Fernandes de Araújo, Prof. Dr. (UFERSA)
Orientador



Leonardo Augusto Casillo, Prof. Dr. (UFERSA)
Coorientador



Karla Darlene Napumocemo Ramos, Prof.ª Dr. ª (UERN)
Membro Interno



Márcio Eduardo Kreutz, Prof. Dr. (UFRN - DIMAp)
Membro Externo

Mossoró/RN
2016

AGRADECIMENTOS

Primeiramente a minha família, em nome da minha mãe. Que em tudo me guia para ser a pessoa que sou hoje. Que sempre se sacrificou para que eu pudesse ter um futuro melhor que o dela. Se eu consegui chegar até aqui, é crédito dela.

Ao meu orientador, Prof. Silvio. Que por diversas vezes sentou comigo e "colocou a mão na massa" para debugar código e resolver erros. Acredito que eu tenha ganhado um novo amigo nessa jordana acadêmica. Espero poder compartilhar de outros açaís no futuro.

Ao meu mestre e que me iniciou na paixão pelo hardware. Talvez ele nem saiba a importância que teve no início dessa jornada. Muito obrigado por tudo, Prof. Leonardo Casillo! A cada vez que você perguntava "E aí? Cadê o texto?" eu corria mais pra terminar.

A família LAACOSTE. Principalmente a Álamo e Juliene. Que dividiram esses anos de trabalhos~~, sofrimento~~ e aprendizagem. Muito sucesso para vocês, amigos!

Por fim, aos amigos que torceram por mim. Que me incentivam com palavras de apoio. Que estiveram ao meu lado durante esse percurso. Fábio, Kelânia, Alysson e Rodolfo, não sei como agradecer a tudo que fizeram por mim. E Eriana Rebouças, queria saber colocar em palavras o quanto eu estou agradecido por tudo que tem me feito. Meu muito obrigado!

"Computadores são servos excelentes e eficientes,
mas eu não tenho nenhum desejo de servir a eles."

Mr. Spock
Star Trek: The Original Series
S02E24: "The Ultimate Computer"

RESUMO

A evolução das arquiteturas de computadores está convergindo no sentido de melhorar o desempenho das aplicações por meio do paralelismo. Foi nesse contexto que surgiu a IPNoSys. Esta arquitetura foi baseada em NoC, o que garante um bom cenário para execução paralela. Contudo, ela ainda foi pouco explorada em paralelismo em nível de instrução. Este trabalho teve como objetivo criar instruções, para a ISA da IPNoSys, que irão proporcionar a paralelização de aplicações que contenham laços de repetição através da técnica de software *pipelining*. Esta técnica visa dividir instruções, que estejam dentro de laços de repetição, entre os núcleos disponíveis. Depois de implementada, a nova IPNoSys com software *pipelining* mostrou, que é possível ter uma aceleração superior à dez vezes em aplicações com instruções lógicas e aritméticas, e superior a três vezes em aplicações com acesso memória.

Palavras-chave: Software *Pipelining*; IPNoSys; Paralelismo em Nível de Instrução

ABSTRACT

The evolution of computer architectures are converging to improve applications performance by means off parallelism. It was in this context that the IPNoSys was created. This architecture was based on NoC, which ensures a good scenario for parallel execution. However, it was still little explored in instruction level parallelism. This work aimed to create instructions, to the IPNoSys' ISA, which will provide parallelization of applications that containing repeating loops through software pipelining technique. This technique aims to divide instructions, which are within repetition of ties between the available cores. After implemented, the new IPNoSys with pipelining software showed that it is possible to have a higher acceleration to ten times in applications with logical and arithmetic instructions, and more than three times in applications with memory access.

Keywords: Software pipelining; IPNoSys; Instruction Level Parallelism

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE GRÁFICOS

LISTA DE ABREVIATURAS E SIGLAS

CISC	*Complex Instruction Set Computer*
DLP	*Data-Level Paralelismo*
E/S	Entrada e Saída
ILP	*Instruction-Level Paralelism*
IP	*Intelectual Property*
IPR	*Instruction per* RPU
IPNoSys	*Integrated Processing* NoC *System*
ISA	*Instruction Set Architecture*
MAU	*Memory Access Unit*
MIMD	*Multiple Instruiction Multiple Data*
MIPS	*Microprocessor Without Interlocked Pipeline Stages*
MISD	*Multiple Instruction Single Data*
MPSoC	*MultiProcessor System-on-Chip*
mW	Miliwatt
NoC	*Network-on-Chip*
PDL	*Package Description Language*
RISC	*Reduced Instruction Set Computer*
RPU	*Routing and Processing Unit*
SIMD	*Single Instruction Multiple Data*
SISD	*Single Instruction Single Data*
SOC	*System-on-Chip*
SP	Software *Pipelining*
SU	*Synchronization Unit*
TLP	*Thread-Level Paralelism*
UC	Unidade de Controle
ULA	Unidade Lógica e Aritmética
VLIW	*Very Long Instruction Word*

SUMÁRIO

1 INTRODUÇÃO

Desde a década de 70, as arquiteturas de computadores iniciaram uma corrida para acompanhar a crescente demanda de processamento das aplicações (DE ROSE; NAVAUX, 2003). Arquiteturas e técnicas de processamento paralelo começaram, então, cada vez mais, a serem exploradas no intuito de manter a evolução no desempenho. Projetos de computadores com mais de uma unidade funcional foram sendo desenvolvidos. Hoje, arquiteturas multicores e multicomputadores já são presentes em dispositivos domésticos.

Com os avanços tecnológicos na fabricação de circuitos integrados, também foi possível miniaturizar componentes. Essa tecnologia possibilitou inserir vários componentes como processador, memória, dispositivos de entrada e saída em um único chip. Essas arquiteturas foram denominadas de sistemas-em-chip (SoC). A colocação de mais processadores, interligados por um sistema de comunicação, em um SoC, deu origem a um tipo especial de arquitetura chamada de sistemas-em-chip multiprocessados (MPSoCs). Fazer a comunicação de núcleos processantes dentro dos MPSoCs é um dos desafios encontrados pelos projetistas. A transmissão de mensagens pode gerar gargalos e prejudicar o desempenho.

Uma proposta para interconexão em MPSoCs é a rede-em-chip (NoC) (BENINI; DE MICHELI, 2002). O funcionamento das NoCs faz uma analogia às redes de computadores convencionais. Os núcleos processantes são conectados a roteadores e estes, a enlaces. Esse modelo integra uma alta escalabilidade com a possibilidade de transmissão de mensagens em paralelo. As NoCs podem ser configuradas para trabalhar com diversos algoritmos de roteamento, topologias, tipo de chaveamento de mensagens, controle de fluxo, etc.

O trabalho de Araújo (2012) apresentou um novo modelo de NoC, intitulado de *Integrated Processing* NoC *System,* IPNoSys. Esta arquitetura trouxe um novo componente que uniu processadores e roteadores em um único elemento chamado de RPU (Unidade de processamento e Roteamento). Esta arquitetura dispõe de um alto poder de processamento quando comparada às arquiteturas semelhantes, principalmente no processamento de aplicações com alta capacidade de paralelização.

Na IPNoSys, é possível explorar o paralelismo em nível de instrução (ILP) e thread (TLP), entretanto, o modelo de computação atual tira melhor proveio do TLP.

Uma das técnicas para prover paralelismo em nível de instrução é a software *pipelining*, que objetiva a execução simultânea de instruções que compõem um laço de repetição. Este trabalho visa a implementação de técnicas de paralelismo em nível de instrução na arquitetura IPNoSys, a fim de possibilitar um aumento de desempenho na execução de aplicações.

1.1 Motivação

O número de trabalhos publicados mostra que a arquitetura IPNoSys vem ganhando destaque no meio acadêmico. No entanto, ela ainda foi pouco explorada em certos aspectos, como no paralelismo em nível de instrução. Como uma NoC, a IPNoSys apresenta um excelente cenário para aplicações paralelas. Contudo, o modelo atual de programação favorece o paralelismo em nível de *thread*.

Como será visto na seção 2.4, cada elemento processante da IPNoSys atua apenas sobre uma instrução do pacote. Isso faz com que os núcleos passem mais tempo transmitindo do que processando. Esse ambiente pode ser modificado para se ajustar a técnica de software *pipelining* de modo aumentar a granularidade do paralelismo e diminuir o tempo de ociosidade dos núcleos.

Criar possibilidade de paralelizar laços pode trazer um aumento de desempenho, principalmente em aplicações com complexidade na ordem de $O(n^2)$ ou superior. Nessas aplicações grande parte do tempo gasto é originado dos laços de repetição (CORMEN et al, 2002).

A motivação deste trabalho é contribuir com o projeto IPNoSys implementando, na arquitetura, recursos que possam prover novas formas de paralelismo em nível de instrução, através da técnica de software *pipelining*.

1.2 Objetivos

O objetivo geral deste trabalho consiste na implementação de uma técnica de paralelismo que visa melhorar o desempenho de aplicações que contenham laços de repetição na rede em chip IPNoSyS. Para isso, será considerada a técnica do software *pipelining*, a qual irá criar um modelo de execução que permita utilizar os recursos da arquitetura para exploração de paralelismo em nível de instruções.

Os objetivos específicos são de criar novas instruções para implementar a técnica de software *pipelining* e manter total compatibilidade com os programas da versão original da IPNoSys (sem software *pipelining*).

Ao final desse trabalho, o desempenho em aplicações que apresentem laços de repetição deve ser melhorado de forma significativa devido a exploração de software *pipelining*.

1.3 Organização Desse Trabalho

Este texto está dividido da seguinte forma: no Capítulo 2, é feita uma revisão bibliográfica sobre arquiteturas paralelas e técnicas de paralelismo, sendo apresentadas as técnicas de *pipeline* e software *pipelining*, além da arquitetura IPNoSys, utilizada no caso de estudos deste trabalho.

No Capítulo 2.5, um breve resumo de alguns trabalhos relacionados, com ênfase em uma implementação anterior de software *pipelining* para IPNoSys.

No Capítulo 3, é apresentada a implementação da técnica de software *pipelining* na IPNoSys desta dissertação.

Em seguida, tem-se os testes executados e os resultados obtidos. E, por último, as considerações finais e as referências utilizadas.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são abordados temas relevantes para este trabalho. A princípio, será feita uma definição sobre técnicas e arquiteturas paralelas. A seguir, será apresentada a arquitetura alvo utilizada como base para desenvolvimento da aplicação descrita por este trabalho. E, por fim, uma explanação sobre a técnica de software *pipelining*.

2.1 Processamento Paralelo

Em meados de 1950, Von Neumann idealizou o primeiro protótipo de um computador de propósito geral. Essa máquina continha uma memória principal, que armazena dados e instruções, uma unidade lógica e aritmética (ULA), uma unidade de controle (UC) e um equipamento de entrada e saída. A unidade de controle operava juntamente com um conjunto de instruções. As instruções são sequências de bits que informam à unidade de controle que operação a ULA irá executar e quais os dados que serão utilizados.

Na medida em que os computadores popularizaram-se, novas operações foram adicionadas ao conjunto de instruções, assim criando os chamados Computadores com um Conjunto de Instrução Complexo (*Complex Instruction Set Computer* – CISC). Segundo Tanenbaum (2013), os microprocessadores CISC se tornaram muito versáteis na execução de aplicações, pois seu grande conjunto de instruções facilita a programação. No entanto, a grande quantidade de instruções do CISC tornou-se inconveniente para várias aplicações mais específicas que precisavam apenas de algumas poucas instruções.

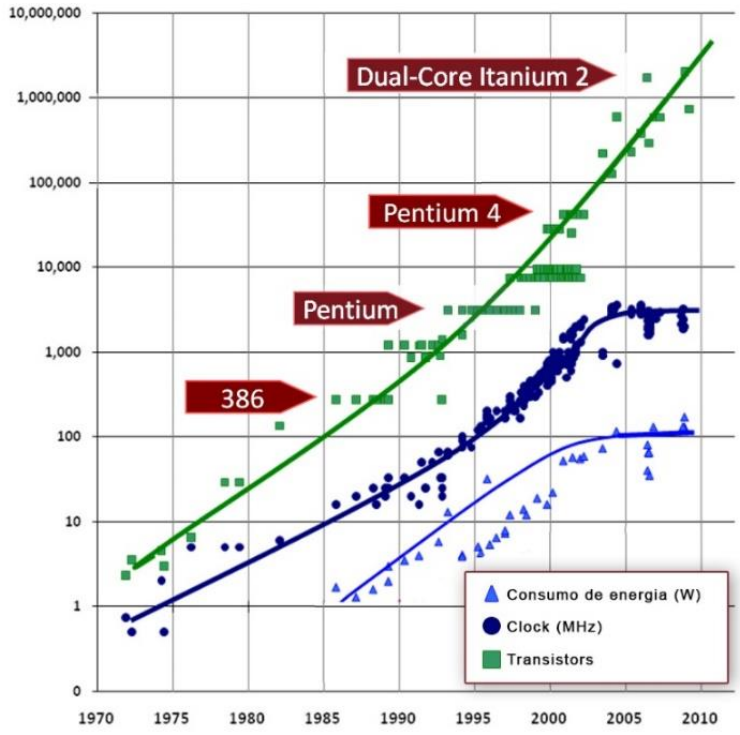
Com uma quantidade reduzida de instruções, os processadores do tipo RISC (*Reduced Instruction Set Computer* ou Computador com um Conjunto Reduzido de Instruções) apostaram na ideia de que instruções mais complexas podem ser efetuadas a partir do conjunto de outras instruções mais simples. Dessa forma, com poucas instruções, a unidade de controle do processador torna mais simples e a velocidade de execução de cada instrução se torna mais rápida. No cenário atual, Calazans (2008)

afirma que muitas arquiteturas são consideradas híbridas, por possuírem características tanto CISC como RISC.

Com a evolução, os computadores têm sido contemplados pelo aumento na velocidade do processador, diminuição no tamanho dos componentes, aumento no tamanho da memória e aumento da velocidade de entrada e saída (STALLINGS, 2013). Entretanto, desde a década passada, os avanços tecnológicos e evolução das aplicações levaram os microprocessadores a se aproximar de um limiar de velocidade de processamento (*clock*), devido às restrições físicas do aumento de temperatura e consumo de energia.

Em contrapartida, à estagnação da velocidade do *clock*, o número de transistores num chip de silício continua aumentando e obedecendo à Lei de Moore (MOORE, 1965). A Figura 1, retirada de Sutter (2005), mostra essa evolução nos processadores da Intel Corporation©.

Figura 1 - Produção de CPU's Intel© entre 1970 e 2010



Fonte: Adaptado de Sutter (2005)[1]

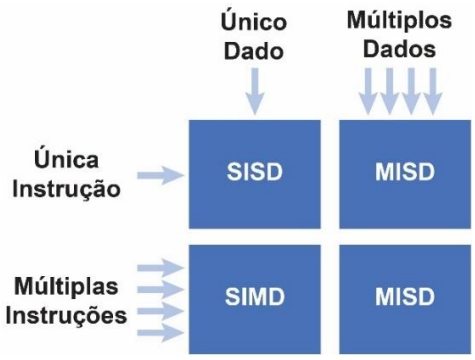
[1] Apesar da fonte datar de 2005, a figura foi atualizada em 2009 na página on-line do artigo.

Na curva superior, no gráfico da Figura 1, é possível ver a evolução sempre crescente do número de transistores dentro de um chip e na curva do meio, em azul, a evolução da taxa de frequência do *clock*. É possível perceber que os transistores continuam crescendo quase linearmente, enquanto a frequência do *clock* se estabilizou por volta do ano de 2005. Como o consumo de energia tem como um dos fatores a velocidade processamento, a curva do consumo de energia (azul claro) acompanha o do *clock*.

Como aumentar a frequência do *clock* faz o aquecimento e o consumo de energia aumentarem numa proporção maior (TORRES, 2013), a solução encontrada para ampliar o desempenho foi projetar arquiteturas que conseguissem executar mais instruções sem aumentar a velocidade do *clock*. Isso foi possível fazendo duas instruções ou mais, ou partes delas, serem executadas ao mesmo tempo, em paralelo. É importante ressaltar que, nas arquiteturas paralelas, o tempo de execução individual das instruções não é menor. Mas como há execução paralela, duas ou mais instruções são executadas numa mesma unidade de tempo, diminuindo assim a vazão de execução de instruções em um programa.

A ideia de processamento paralelo não é recente. Os trabalhos de Flynn (1966) e Flynn (1972) já descreviam modelos de arquiteturas paralelas. Nesses artigos, Flynn descreveu quatro categorias de máquinas que são aceitas até hoje como a melhor classificação para arquiteturas paralelas. São elas: *Single Instruction Single Data* (SISD), *Single Instruction Multiple Data* (SIMD), *Multiple Instruiction Multiple Data* (MIMD), *Multiple Instruction Single Data* (MISD). A Figura 2 resume as quatro classificações.

Figura 2 - Classificação de Arquiteturas Paralelas Segundo Flynn



Fonte: Adaptado de <http://www-usr.inf.ufsm.br/~sandro/elc139/tarefa1.php>

Os computadores SISD são as máquinas clássicas não paralelas da arquitetura de Von Neumann, com um único fluxo de instrução e um único fluxo de dados.

As SIMD funcionam com um único fluxo de instrução e múltiplos fluxos de dados. Correspondem ao caso das arquiteturas vetoriais em que a mesma operação é executada sobre múltiplos operandos e processadores.

Já as máquinas MIMD são o caso dos multiprocessadores, em que várias instruções podem ser executadas ao mesmo tempo em unidades de processamento diferentes, controladas por unidades de controle independentes (uma para cada unidade de processamento). Essa arquitetura permite execução de instruções diferentes para cada dado.

Por último, computadores MISD são os que executam diferentes instruções sobre a mesma posição da memória ao mesmo tempo. Essa arquitetura é impraticável pois não existem aplicações reais para ela.

Além da classificação de Flynn, também é possível encontrar na literatura uma classificação segundo o compartilhamento da memória. De acordo com De Rose e Navaux (2003), podemos afirmar que arquiteturas em que os processadores compartilham um único espaço de endereçamento de memória são chamadas de multiprocessadores. Por outro lado, arquiteturas que possuem replicação do conjunto processador/memória são chamadas de multicomputadores. Neste último caso, a comunicação entre os processos é efetuada por mensagens, através de uma rede de interconexão (HESS, 2003).

Atualmente, os multiprocessadores têm evoluído para um modelo genérico incluindo vários núcleos com CPU e memória, comunicando-se entre si através de uma rede de interconexão. Esse novo conceito de arquitetura foi chamado de *System-on-Chip* (SoC) por Culler, Singh e Gupta (1998).

2.1.1 Sistemas em Chip

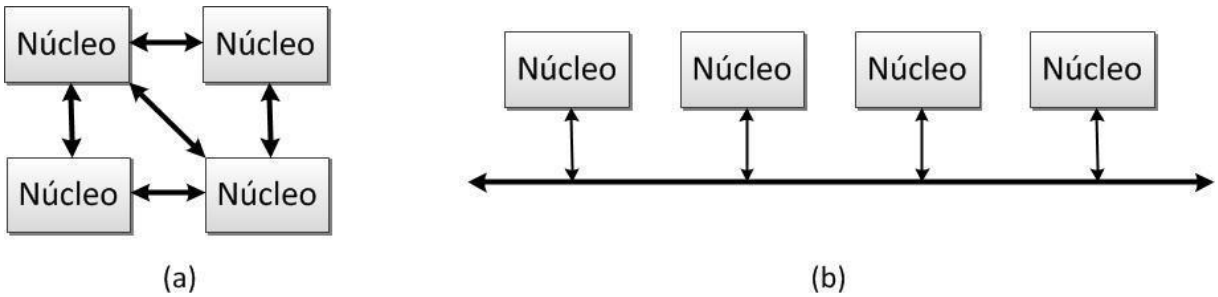
São arquiteturas compostas por um processador agregado a coprocessadores específicos para diversas finalidades, como processamento de imagens, sinais, gerenciamento de entrada e saída, etc.

A união das tecnologias SoC e multiprocessadores tornou possível a criação dos MPSoC (*MultiProcessor System-on-Chip*) (WOLF; JERRAYA; MARTIN, 2008). A tecnologia de MPSoCs não consiste apenas em inserir diversos processadores em um único chip. O conceito de MPSoC define que os processadores devem ser otimizados para a aplicação alvo, e blocos computacionais desnecessários à aplicação são removidos para economizar energia e área do circuito integrado (JERRAYA; WOLF, 2004).

2.1.2 Redes em Chip

Com vários processadores interconectados, o desafio dos projetistas tornou-se evitar a perda de desempenho durante a troca de informação entre as unidades processantes (ou *Intelectual Property* – IP). A arquitetura de comunicação mais simples utilizada por MPSoCs é a de canais ponto a ponto (Figura 3a), em que cada IP está ligado a cada um dos outros IPs. No entanto, esse modelo torna-se inviável no contexto de escalabilidade. As arquiteturas de barramento (Figura 3b) são uma solução para o aumento da rede, pois interligam todos os IPs através de uma única ligação. Contudo, este tipo de conexão tende a causar atrasos no envio de mensagens devido à disputa pelo canal de transmissão.

Figura 3 - Estruturas de Interconexão: (a) ponto-a-ponto; (b) barramento

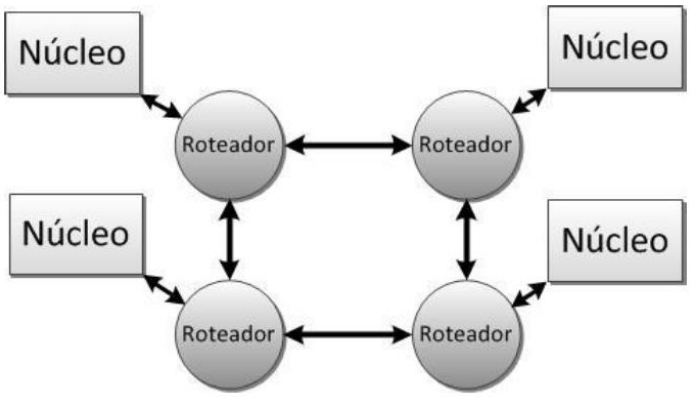


Fonte: Araújo (2012)

Uma proposta de comunicação entre os IPs são as Redes-em-Chip (*Network-on-Chip* – NoC) (BENINI; DE MICHELI, 2002). As NoCs fazem analogia a redes de computadores tradicionais. Elas podem ser definidas como uma estrutura de roteadores ligados ponto a ponto e conectados aos IPs da MPSoC, de modo a prover a comunicação entre eles. As informações ou mensagens são trocadas através de pacotes de dados. O uso de NoCs permite gerenciar a eficiência energética, comunicação e a escalabilidade do SoC (BENINI; DE MICHELI, 2002). A Figura 4 mostra a ligação entre núcleos por uma NoC.

O aumento da complexidade das SoCs trouxe problemas com relação ao consumo de energia, latência na sincronização, vazão de comunicação. Esses fatores motivaram o uso das NoCs para facilitar a separação entre comunicação e computação.

Figura 4 - Modelo de uma NoC



Fonte: Araújo (2012)

Os primeiros trabalhos sobre a ideia de NoCs datam de 1992 (TEWKSBURY; UPPULURI; HORNAK, 1992). Assim como nas redes de computadores, as NoCs podem ser caracterizadas por topologia, chaveamento, controle de fluxo, arbitragem, roteamento e memorização (ZEFERINO, 2003) (LEE et al, 2007).

Segundo Concer, Iamundo e Bononi (2009), as NoCs são definidas por um conjunto de três elementos: interfaces de rede, roteadores e enlaces (*links*). Esses componentes juntos são responsáveis por uma arquitetura de comunicação de chaveamento de pacotes, os quais encapsulam as mensagens ou as informações a serem transmitidas de um núcleo fonte até um destino através da infraestrutura

oferecida pela rede. O pacote é formado por um conjunto de palavras contendo um cabeçalho, que normalmente carrega informações do pacote, da aplicação e dos roteadores de origem e destino e a mensagem propriamente dita.

As NoCs podem ser configuradas para funcionarem com diferentes topologias, algoritmos de roteamento, chaveamento, controle de fluxo, memorização, etc. Tal característica torna fácil encontrar exemplos de implementações de NoCs na literatura. Um dos primeiros trabalhos sobre NoC foi a arquitetura SPIN (*Scalable Programmable Interconction Network*) de Adriahantenaina et al (2003), que utiliza uma topologia de árvore gorda (*fat tree*). Outra arquitetura bem citada é a SoCIN (*SoC Interconnection Network*) de Zeferino e Susin (2003). Outros exemplos são a HERMES (MORAES et al, 2004) e a STORM (REGO, 2006).

Para este trabalho, foi escolhida uma NoC descrita por Araújo (2012) intitulada de *Integrated Processing* NoC *System*, IPNoSys, que será descrita em detalhes na seção 2.4.

2.2 Pipeline

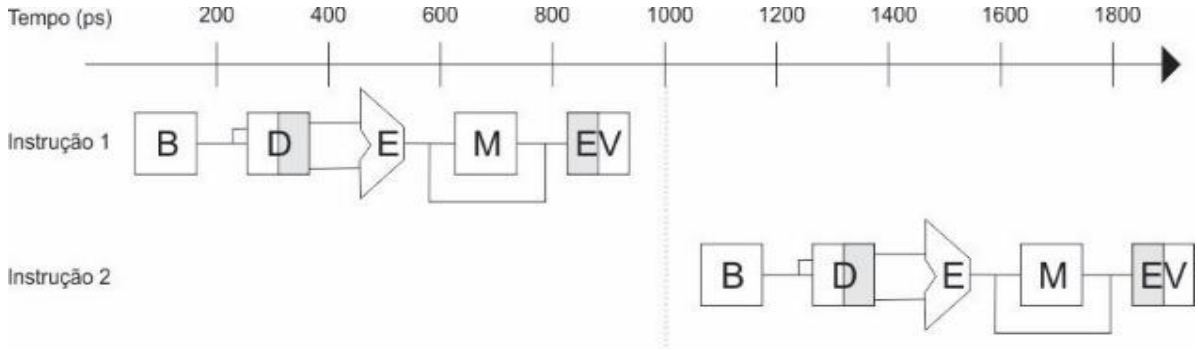
O *pipeline* foi adicionado aos computadores a partir do processador 486 da Intel, embora seu conceito inicial de divisão de instruções já tenha sido desenvolvido desde o processador 8086, o qual continha duas unidades distintas que processavam partes diferentes da instrução (VASCONCELOS, 2002). Esta ideia se baseia na criação de um semiparalelismo, dividindo a instrução em diferentes estágios que podem ser executados em paralelo, com o objetivo de aproveitar as unidades que ficariam ociosas em algum momento do processamento.

De forma geral, a técnica de *pipeline* não aumenta a velocidade de execução de uma instrução. O que ela faz é aumentar o início da execução das próximas instruções, de modo análogo a uma linha de produção. O termo "semiparalelismo" foi descrito por Patterson e Hennessy (2014) devido ao fato que duas (ou mais) instruções não são executadas em paralelo, mas que, em um determinado momento, elas estão simultaneamente em processo de execução.

Para ser compatível com *pipeline*, o processador deve ter as unidades funcionais do caminho de dados bem divididas, de modo que a saída de um estágio deve alimentar o estágio seguinte. Um dos projetos precussores do *pipeline* foi o MIPS (*Microprocessor Without Interlocked Pipeline Stages*) (PATTERSON; HENNESSY, 2014).

O trabalho de Nunes (2012) desenvolveu um processador baseado em MIPS contendo cinco estágios, sendo eles: Busca (identificado pela letra B), Decodificação (D), Execução (E), Acesso à Memória (M) e Escrita de Volta (EV). A Figura 5 mostra um exemplo de uma instrução sendo executada neste processador, mas sem utilização de *pipeline*.

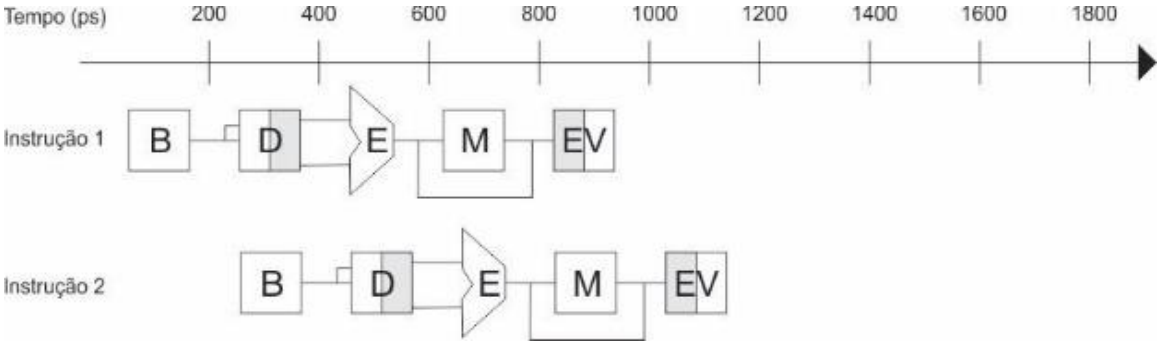
Figura 5 - Exemplo de execução de instruções em uma arquitetura sem *pipeline*



Fonte: Nunes (2012)

Pode-se observar que a instrução 2 só começa o estágio de Busca quando a instrução 1 finaliza o estágio de Escrita de Volta. No mesmo processador utilizando *pipeline*, o estágio de Busca da instrução 2 pode ser iniciado assim que o estágio de Busca da instrução 1 tiver sido concluído. A Figura 6 exemplifica este cenário.

Figura 6 - Exemplo de execução de instruções em uma arquitetura com *pipeline*



Fonte: Nunes (2012)

Assumindo condições ideais, o desempenho de um processador com *pipeline* será proporcional ao número de estágios que ele contém e pode ser calculado da seguinte forma:

$$Tempo\ do\ programa\ com\ pipeline = \frac{Tempo\ do\ programa\ sem\ pipeline}{Nº\ de\ estágios\ do\ pipeline}$$

No entanto, há situações em que a instrução seguinte não pode ser iniciada antes do término da instrução anterior. Estas situações são denominadas de dependências e podem ser de três tipos:

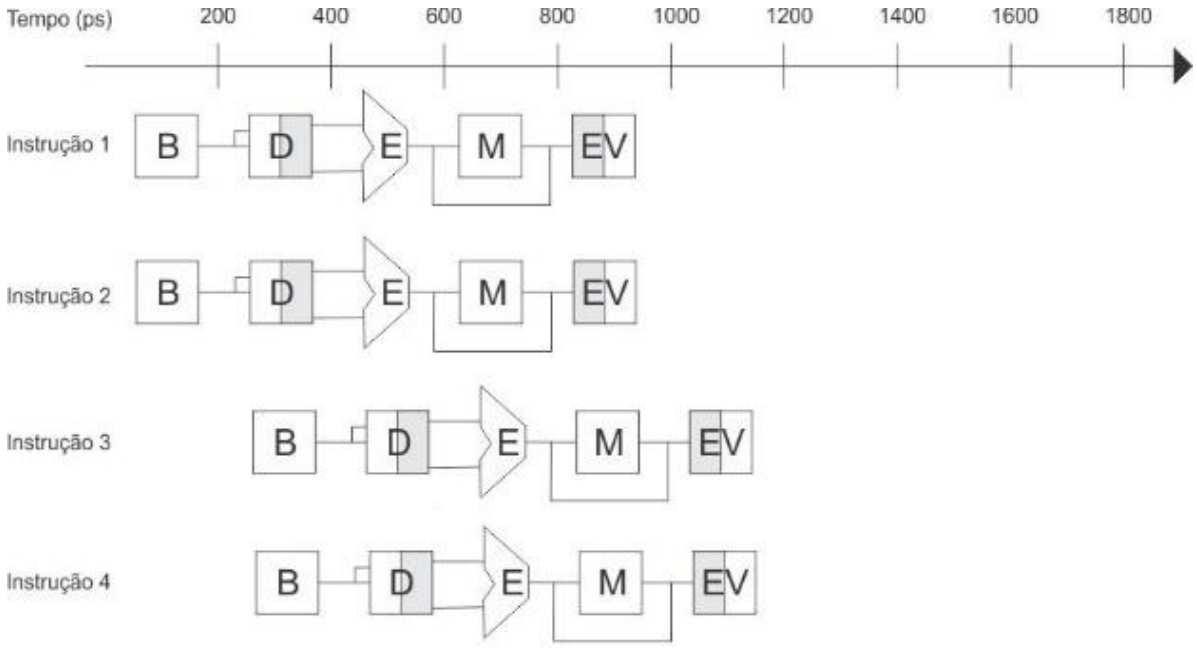
a) Dependências de Dados: acontece quando a próxima instrução necessita do valor do resultado da instrução anterior.
b) Dependências de Controle: as instruções de desvio são as causadoras desse tipo de dependência. Quando as instruções são executadas com *pipeline*, ao surgir um desvio, as instruções seguintes, que já haviam entrado no *pipeline*, são descartadas. Isso causa uma perda de desempenho.
c) Dependência Estrutural: acontece quando duas instruções no *pipeline* tentam acessar o mesmo recurso ao mesmo tempo.

As situações de dependências fazem com que o processador não consiga preencher os estágios do *pipeline*, resultando em perda do desempenho. Existem algumas técnicas para tratamento de dependências, mas a implementação desses mecanismos gera uma complexidade maior para a unidade de controle (PILLA; SANTOS; CAVALHEIRO, 2009). Contudo, a técnica de *pipeline* ainda é indiscutivelmente vantajosa no ganho de desempenho para processadores. Diversas famílias de processadores comerciais atuais, inclusive a Intel Core®, implementam o *pipeline* em suas arquiteturas (TORRES, 2013).

Quando as unidades funcionais do *pipeline* são replicadas, podemos ter instruções sendo executadas de forma paralela em todos os estágios. Para processadores desse tipo, damos o nome de superescalares (PATTERSON; HENNESSY, 2014). No entanto, é fácil notar que o ganho de desempenho dos processadores superescalares está ligado à quantidade de réplicas das unidades funcionais e às dependências das instruções. A Figura 7 contém o modelo de um processador

superescalar com suas unidades funcionais duplicadas. As regras de *pipeline* nessa arquitetura são as mesmas. Pode-se observar que apenas nessa arquitetura é que, de fato, as instruções são executadas em paralelo.

Figura 7 - Processador Superescalar



Fonte: Autoria própria

Máquinas superescalares têm seu paralelismo em nível de instrução (duas ou mais instruções são executadas ao mesmo tempo). Desta forma, elas são classificadas como arquiteturas paralelas ILP (*Instruction Level Paralelism*). Além do ILP, é possível encontrar arquiteturas com paralelismo em nível de thread (*Thread-Level Paralellism* – TLP) explorado por arquiteturas multicores e em nível de dados (*Data-Level Paralelism* – DLP) utilizado em processadores vetoriais (SANKARALINGAM et al, 2003).

2.3 Software *Pipelining*

A técnica de software *pipelining*, descrita por Lam (1988), afirma que computadores com arquiteturas do tipo VLIW (*Very Long Instruction Word*) possuem componentes funcionais que possibilitam que instruções dentro de um laço de

repetições possam ser executados de forma simultânea, similar ao *pipeline*. De acordo com Jones e Allan (1990), o conceito de software *pipelining* é descrito como conseguir iniciar a segunda iteração de um laço de repetição antes que a primeira iteração tenha finalizado. Para efeito didático, tomemos o algoritmo descrito pela Figura 8 para ilustrar o conceito do software *pipelining*.

Figura 8 - Exemplo de Laço de Repetição

```
1  algoritmo exemplo 1
2     x, i, op1, op2, op3 : inteiro
3     para i de 0 ate x faca
4        op1 = i
5        op2 = op1 + 1
6        op3 = op2 + 2
7     fimpara
8  fimalgoritmo
```

Fonte: Autoria Própria

Em arquiteturas convencionais, cada instrução do algoritmo da Figura 8 seria executada de forma sequencial, uma após a outra. Na Seção 2.2, foi possível verificar que, utilizando o *pipeline*, pode-se iniciar a instrução seguinte antes de finalizar a instrução anterior, por meio do *pipeline* convencional.

No entanto, para o exemplo descrito, o *pipeline* convencional não pode ser empregado, pois há uma dependência de dados entre as instruções. A instrução na linha 6 depende do resultado da instrução da linha 5, que depende da instrução da linha 4. Com isso, a unidade funcional de decodificação não pode ler os dados na memória antes deles serem escritos como resultado da instrução anterior. Devido aos problemas de dependências e considerando o valor de iterações x = 5, o algoritmo será executado como mostra a Figura 9.

Figura 9 - Execução Sequencial do Algoritmo Exemplo

Ciclos \ Iterações do Laço	1	2	3	4	5
1	op1=i				
2	op2=op1+1				
3	op3=op2+2				
4		op1=i			
5		op2=op1+1			
6		op3=op2+2			
7			op1=i		
8			op2=op1+1		
9			op3=op2+2		
10				op1=i	
11				op2=op1+1	
12				op3=op2+2	
13					op1=i
14					op2=op1+1
15					op3=op2+2

Fonte: Autoria Própria

Através do software *pipelining*, as instruções dentro do laço serão reorganizadas entre suas unidades funcionais, de modo que possam ser executadas de forma semelhante a um *pipeline* (JONES; ALLAN, 1990), como mostrado pela Figura 10.

Figura 10 - Execução do Algoritmo com Software *pipelining* (Ciclos/Iteração)

Ciclos \ Iterações do Laço	1	2	3	4	5
1	op1=i				
2	op2=op1+1	op1=i			
3	op3=op2+2	op2=op1+1	op1=i		
4		op3=op2+2	op2=op1+1	op1=i	
5			op3=op2+2	op2=op1+1	op1=i
6				op3=op2+2	op2=op1+1
7					op3=op2+2

Fonte: Autoria Própria

Existem várias formas de implementação de software *pipelining*, incluindo até modelos heurísticos para reorganização das instruções pelo compilador (RAU, 1994). Contudo, diversos autores mostram que a implementação da técnica de software *pipelining* garante um ganho de desempenho em aplicações que envolvem laços de repetição, tais como Jones e Allan (1990), Rau (1994), Goldberg et al (2002), Thies,

Chandrasekhar e Amarasinghe (2007) e Wang et al (2010). Adicionalmente, Govindarajan, Altman e Gao (1996) dizem que o esforço de implementar a técnica de software *pipelining* vale a pena devido à sua simplicidade de descrição na arquitetura, se comparada ao ganho de desempenho proporcionado.

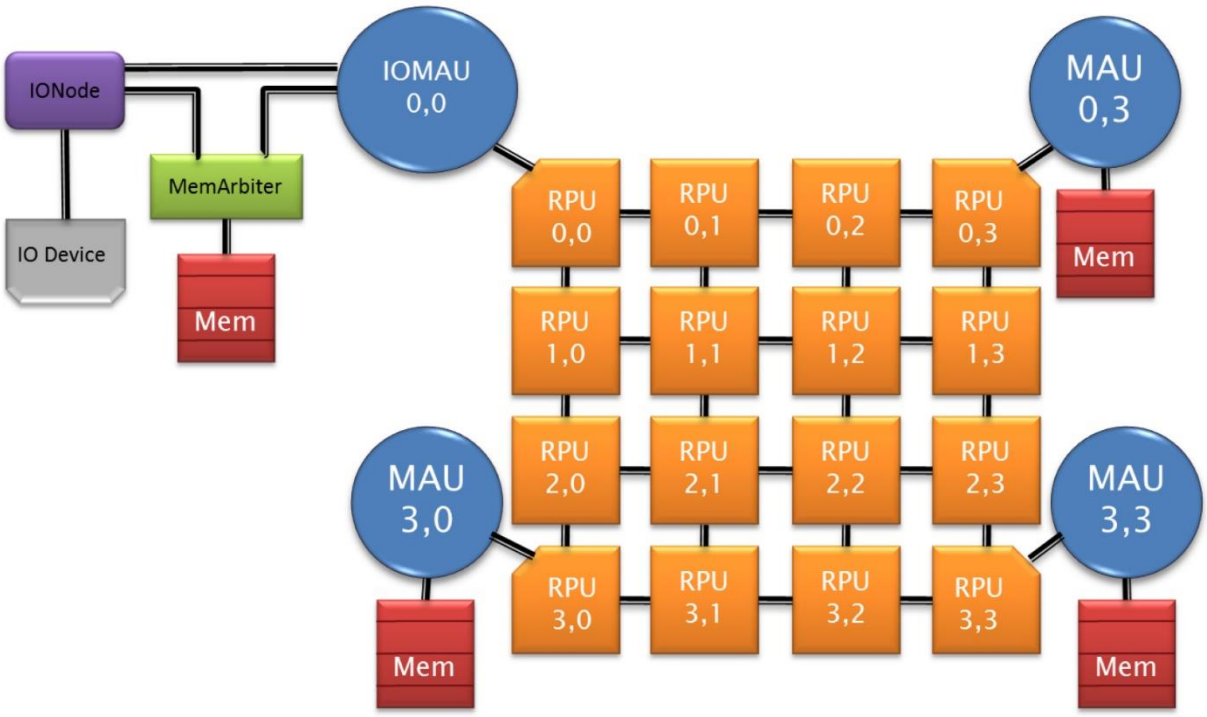
2.4 IPNoSys

Neste trabalho foi utilizada a rede em chip IPNoSys, que foi modificada para implementação da técnica de software *pipelining*. Os detalhes relevantes desta arquitetura em seu formato original são apresentados nesta seção.

2.4.1 Visão Geral

A *Integrated Processing NoC System*, ou IPNoSys (Figura 11), é uma arquitetura proposta por Araújo (2012), baseada em NoC.

Figura 11 - Arquitetura IPNoSys



Fonte: Araújo (2012)

A principal diferença entre IPNoSys e outras NoCs é a alteração dos roteadores a fim de prover, além da transmissão de pacotes, o processamento de dados. Isso é possível inserindo-se uma unidade lógica e aritmética (ULA) e uma unidade de sincronização (*sinchronization unit* - SU) no roteador. Essa estrutura modificada foi descrita em Araújo et al (2009) e chamada de Unidade de Processamento e Roteamento ou RPU (*Routing and Processing Unit*).

De modo geral, a IPNoSys é uma rede de RPUs com topologia grelha-2D de dimensão quadrada, ou seja, o número de linhas é igual ao de colunas. A ordem da rede (número de linhas e colunas) pode ser configurada de acordo com a aplicação. Cada RPU é ligada à sua vizinha através de um barramento.

Nos quatro cantos da rede existem, ligadas a cada RPU adjacente, uma Unidade de Acesso à Memória – MAU (*Memory Access Unit*). Uma das MAUs é diferenciada para prover o gerenciamento de entrada e saída (E/S). Ela é ligada ao IONode, que faz a ponte com os dispositivos de E/S. No modelo IPNoSys, o processamento ocorre quando uma MAU injeta pacotes na rede. Ao chegar na RPU, as instruções vão sendo processadas e roteadas para o próximo destino.

A relação das MAUs com as RPUs torna possível que todas injetem pacotes simultaneamente na rede, permitindo a execução paralela de aplicações. Os pacotes no IPNoSys utilizam o modelo *backpacker* (TIEG, 2010), em que os dados estão sempre presentes junto com as instruções, em analogia aos mochileiros, que carregam tudo que o que lhes é essencial. Com isso, é possível reduzir a quantidade de acessos à memória. Mais detalhes sobre os pacotes do IPNoSys serão abordados na Seção 2.4.3. O modelo de programação da IPNoSys se classifica como "tudo explícito", sendo responsabilidade do programador fazer toda a configuração para execução dos pacotes (SKILLICORN; TALIA, 1998).

Os pacotes do IPNoSys são sequências de instruções e dados que contém a aplicação a ser executada. Os pacotes são inseridos na rede pelas MAUs e são transferidos para as RPUs através de um algoritmo de roteamento. Cada RPU no caminho do pacote retira pelo menos uma instrução, a executa, insere os resultados no

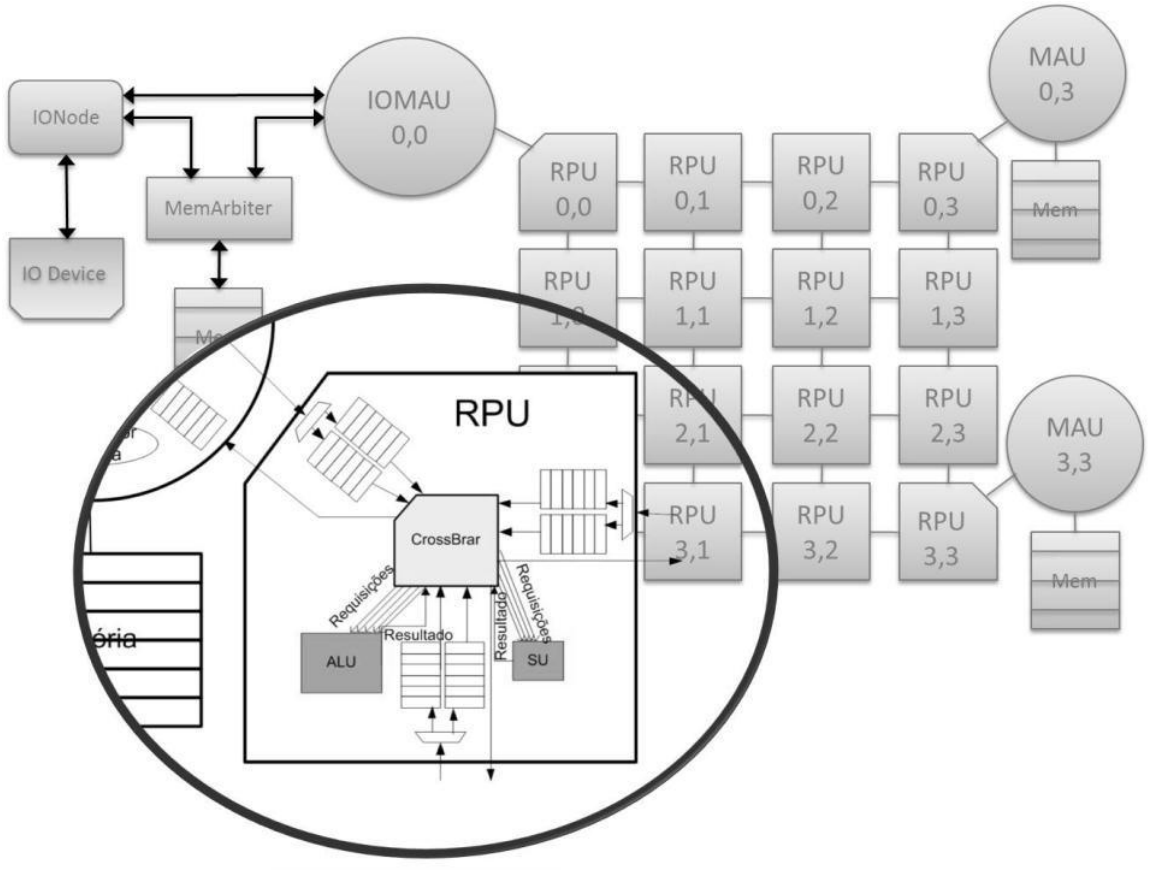
pacote (se houver) e o roteia para a próxima RPU. A cada RPU que o pacote passa, vai diminuindo até ser consumido por completo.

Para efeito de simulação, a arquitetura foi totalmente descrita utilizando SystemC (ARAÚJO, 2012). O SystemC é uma extensão da linguagem C++ que possibilita a simulação de sinais e processos de hardware, semelhante às linguagens de descrição de hardware (BLACK et al, 2005).

2.4.2 Unidade de Roteamento e Processamento

A RPU é o elemento responsável pelo roteamento dos pacotes e execução de instruções. Sua estrutura é composta por uma Unidade Lógica e Aritmética, responsável por executar as instruções descritas nos pacotes das aplicações, uma Unidade de Sincronismo (SU – *Synchronization Unit*), *buffers*, *crossbar* e árbitros (um associado a cada porta), como detalhado na Figura 12.

Figura 12 - Unidade de Processamento e Roteamento



Fonte: Araújo (2012)

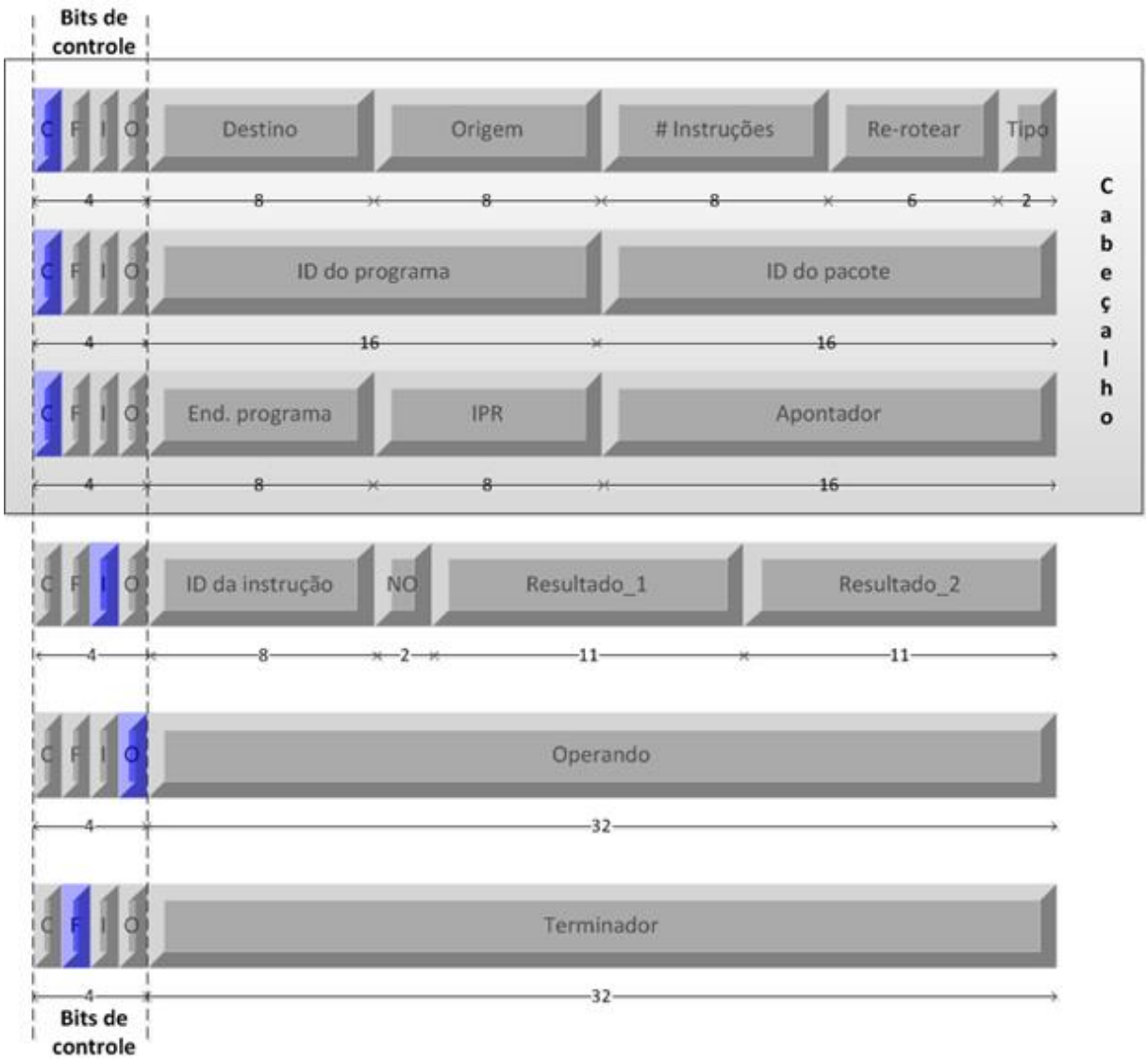
Ao chegar na RPU, o pacote é armazenado em um *buffer* na entrada. As RPUs mais internas à rede têm quatro entradas, enquanto as mais externas possuem apenas três. Cada entrada possui pelo menos dois buffers, um para cada canal virtual. Depois de armazenado, o árbitro correspondente analisa o conteúdo do pacote a fim de verificar o seu tipo. Caso seja um pacote regular, a instrução e os operandos são retirados do pacote e enviados para a ULA. Caso seja um pacote de outro tipo, o *crossbar* se responsabiliza de chavear para uma das portas de saída. Se o *buffer* de entrada da próxima RPU estiver cheio, inicia-se um processo chamado de execução localizada. Como a RPU não pode transmitir, ela mesma se encarrega de executar as próximas instruções até que consiga transmitir.

Pode-se dizer que o árbitro é o elemento principal de controle da RPU. Ele é responsável por identificar a instrução, enviá-la para ULA, SU ou MAU. Após a execução de uma instrução, o árbitro passa apenas a transmitir o pacote, sempre verificando se é o momento de inserir algum resultado gerado pela instrução que executou. Ele faz isso verificando se a posição de inserção do resultado coincide com o contador de palavras transmitidas. Os árbitros ficam localizados nas portas de saídas e controlam as disputas pelos canais virtuais (das saídas) utilizando o algoritmo *round-robin*, como em um roteador tradicional.

2.4.3 Formato do Pacote

O Pacote no IPNoSys é formado por palavras de 32 bits, com quatro bits extras (bits de controle) usados para identificar o tipo da palavra (Figura 13). As três primeiras palavras de um pacote são compostas por seu cabeçalho. As palavras seguintes são as instruções e operandos da aplicação. Por fim, a última palavra do pacote contém a identificação de fim do pacote. O cabeçalho e o fim de pacote sempre contém três e uma palavra, respectivamente, enquanto que as de instruções e operandos podem variar dependendo da aplicação.

Figura 13 - Formato do pacote IPNoSys



Fonte: Araújo (2012)

No IPNoSys são encontrados quatro tipos de pacotes. Os Regular são os que carregam as instruções e operandos para serem executados nas RPUs. Além dele, existem os pacotes de controle, que são pacotes endereçados às MAUs a fim de efetuar alguma operação de leitura e escrita na memória ou sincronização; o pacote *Caller*, que faz uma operação similar a uma chamada de função; e o pacote Interrompido, que informa que a RPU está aguardando algum evento (como a entrada/saída) ou houve algum problema na execução do pacote regular que estava sendo processado. Os pacotes regulares da aplicação são gerados pelo compilador. Os outros tipos são gerados automaticamente pela arquitetura em tempo de execução da aplicação.

Em cada palavra do pacote, apenas um bit de controle pode estar ativo. Esses bits são identificados pelas letras “C” (cabeçalho), “F” (fim de pacote), “I” (instrução) e “O” (operandos), na Figura 13. Nas três palavras de cabeçalho são encontrados campos relativos a origem e destino do pacote, quantidade de instruções e outros detalhes do programa. Para este trabalho, as palavras de cabeçalho e fim de pacote foram ignoradas, sendo necessário apenas um aprofundamento dos campos das palavras de instrução e operandos. A palavra de instrução é definida por quatro campos (além dos bits de controle):

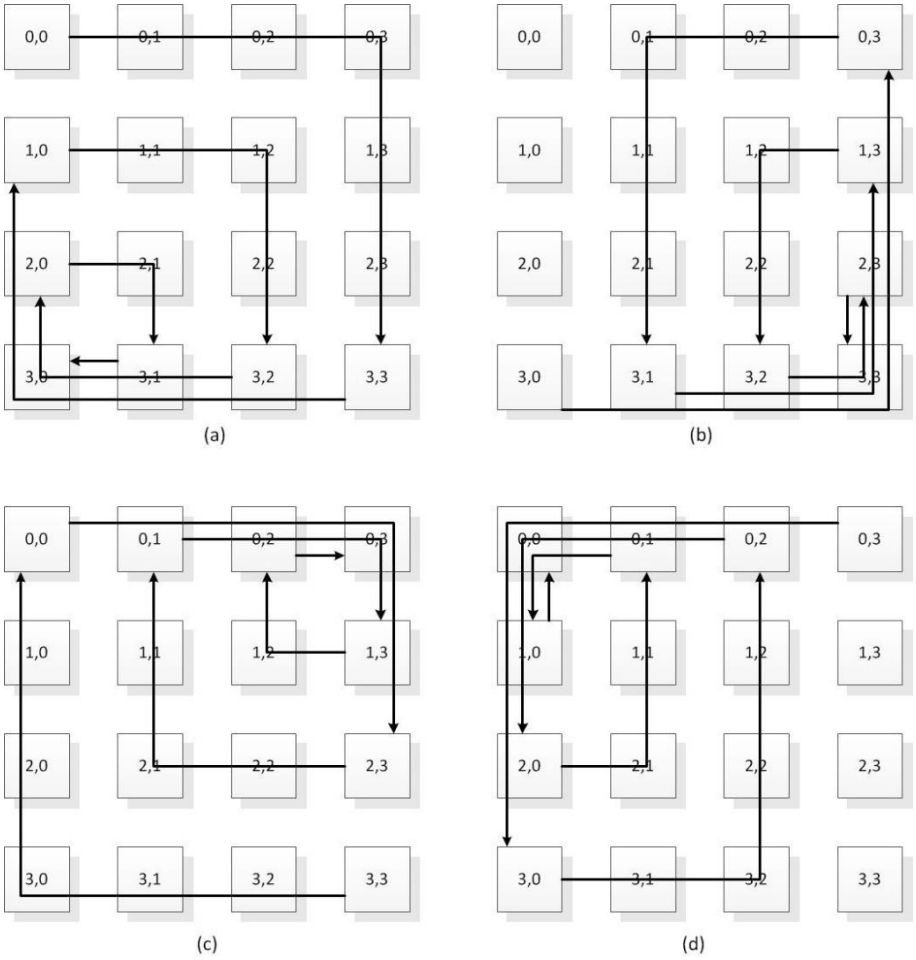
a) *ID da Instrução* (8 bits): Identifica a instrução a ser executada.
b) *NO* (2 bits): Número de operandos utilizados para executar a instrução.
c) *Resultado_1* (11 bits): em pacotes de controle, este campo é utilizado para indicar o endereço da MAU que deverá executar a instrução à qual está associado. Nos demais pacotes, ele é utilizado para indicar a posição em que o resultado da operação deve ser inserido no mesmo pacote.
d) *Resultado_2* (11 bits): este campo pode indicar a quantidade de operandos, menos um, envolvidos na instrução (se o campo NO contiver o valor “11_b”); ou uma segunda posição no pacote em que o resultado da instrução também deve ser inserido.

Nas palavras de operandos, os 32 bits são utilizados como operandos da instrução. Nas palavras de instrução, apenas o bit de controle “I” deve estar ativado, enquanto que nas palavras de operando, apenas o bit “O”.

2.4.4 Roteamento

Na arquitetura IPNoSys, a RPU retira uma instrução do pacote e transfere o resto do pacote para a próxima RPU. Portanto, deve haver um caminho longo suficiente na rede para que o pacote possa caminhar entre as RPUs até ele ser consumido por completo. O algoritmo de roteamento utilizado para prover esse recurso foi denomidado de *Spiral Complement* e apresentado em Araújo et al (2009). O *Spiral Complement* define uma série de caminhos em espirais. A Figura 14 mostra os caminhos percorridos por um pacote que é injetado por cada uma das MAUs.

Figura 14 - Roteamento *Spiral Complement*



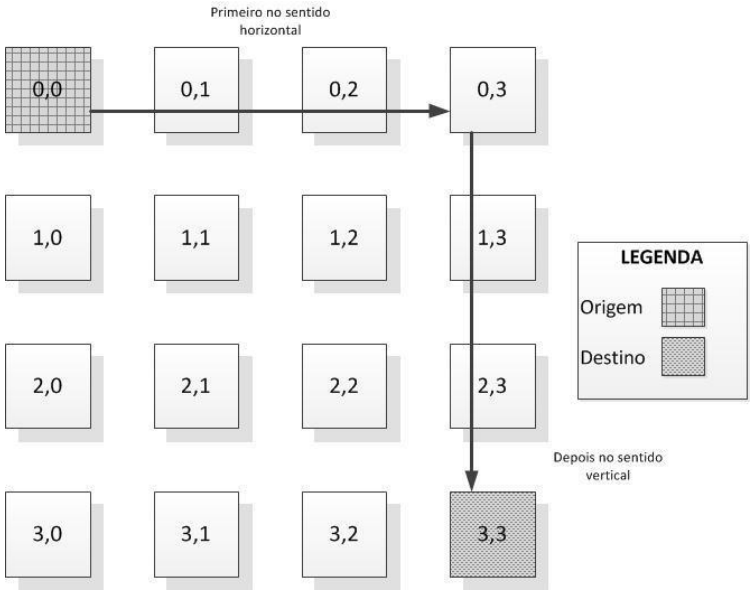
Fonte: Araújo (2012)

O modelo em espirais contribui para que o pacote percorra o máximo de caminhos possíveis. Entretanto, pacotes com um número alto de instruções podem chegar até o fim do caminho e ainda haver instruções para serem executadas. Nesses casos, ao chegar no fim do percurso, o pacote pode entrar numa situação de *deadlock*, em que ainda há instruções para serem executadas mas não existe RPU disponível para execução. Nesses casos, a fim de evitar o *deadlock*, a RPU entra em modo de execução localizada. Na execução localizada, a RPU para de transmitir pacotes e começa a executar o resto das instruções no pacote até que possa começar a transmitir novamente ou até que o pacote chegue ao fim.

O algoritmo *Spiral Complemet* é usado para rotear apenas os pacotes regulares. Os outros tipos de pacote, que têm como função apenas comunicação, são roteados pelo algoritmo de roteamento XY. Nesse algoritmo, o caminho do pacote é dividido em

duas partes. Aos sair da origem, o pacote caminha horizontalmente (eixo X) até a coluna onde se encontra o seu destino. A partir daí, o pacote percorre a rede no sentindo vertical (eixo Y) até chegar em seu objetivo (Figura 15).

Figura 15 - Roteamento XY



Fonte: Araújo (2012)

Outros algoritmos de roteamento foram desenvolvidos para IPNoSys a fim de tentar maximizar a quantidade de saltos antes de entrar em execução localizada. Os trabalhos de Cruz (2013) e Filho et al (2014), por exemplo, fizeram estudos de caso comparando o *Spiral Complement* com outros algoritmos de roteamento. Entretanto, tais algoritmos não serão abordados neste trabalho.

2.4.5 Programabilidade

Para se executar aplicações no sistema IPNoSys, é necessário descrever o programa através da Linguagem de Descrição de Pacotes (PDL – *Package Description Language*). A ISA (*Instruction Set Architecture,* ou conjunto de instruções*)* define 32 instruções, que são detalhadas na Tabela 1, divididas em 4 instruções aritméticas, 4 lógicas, 2 de deslocamentos, 4 de sincronização, 3 de acesso a memória, 6 de desvios condicionais, 1 incondicional, 2 auxiliares, 2 de E/S, 2 de sistema e 2 de chamada de procedimento.

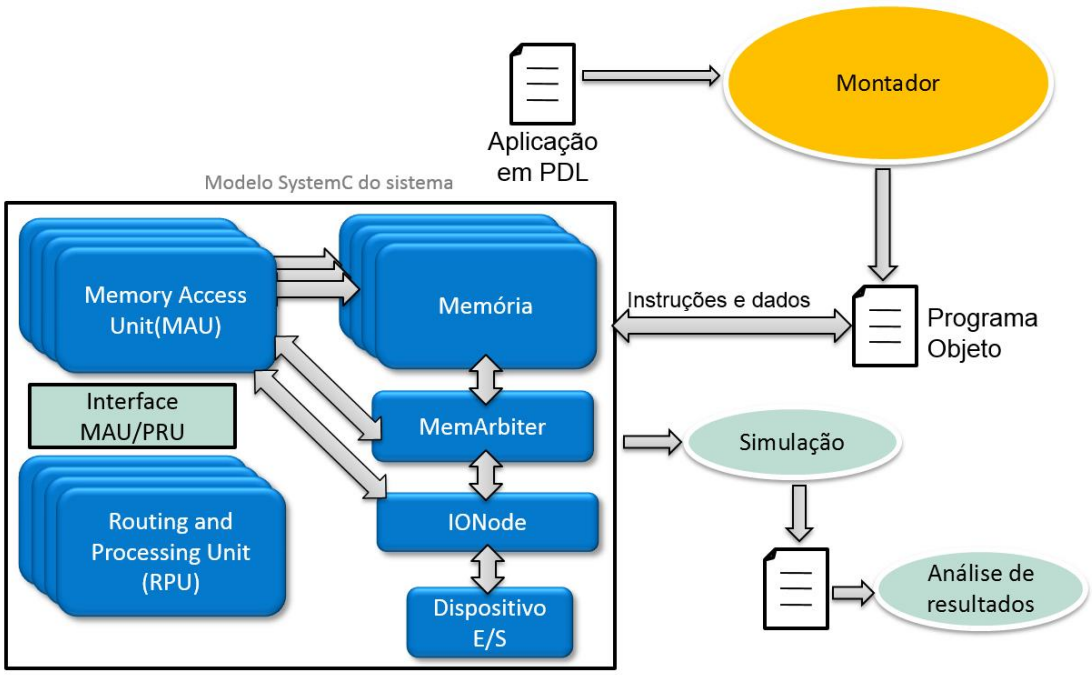
Tabela 1 - Conjunto de Instrução da IPNoSys

CodOp	Instrução	Tipo	Operandos	Descrição
0	**ADD**	Aritmética	2	Soma 2 inteiros
1	**SUB**	Aritmética	2	Subtrai 2 inteiros
2	**MUL**	Aritmética	2	Multiplica 2 inteiros
3	**DIV**	Aritmética	2	Divide 2 inteiros
4	**NOT**	Lógica	1	Negação de 1 valor
5	**AND**	Lógica	2	Conjunção de 2 valores
6	**OR**	Lógica	2	Disjunção de 2 valores
7	**XOR**	Lógica	2	Ou-exclusivo de 2 valores
8	**RSHIFT**	Deslocamento	2	Desloca *n* bits de um valor à direita
9	**LSHIFT**	Deslocamento	2	Desloca *n* bits de um valor à esquerda
10	**LOAD**	Acesso à Memória	1	Solicita um valor da memória
11	**STORE**	Acesso à Memória	Vários	Armazena um valor na memória
12	**EXEC**	Sincronização	1	Ordena a injeção de um pacote
13	**SYNEXEC**	Sincronização	Vários	Ordena a injeção de um pacote após sincronização
14	**SYNC**	Sincronização	1	Sinal de sincronização para um pacote
15	**RELOAD**	Acesso à Memória	1	Retorna um valor carregado da memória
16	**BE**	Condicional	2	Desvia se igual
17	**BNE**	Condicional	2	Desvia se diferente
18	**BL**	Condicional	2	Desvia se menor
19	**BG**	Condicional	2	Desvia se maior
20	**BLE**	Condicional	2	Desvia se menor ou igual
21	**BGE**	Condicional	2	Desvia se maior ou igual
22	**JUMP**	Incondicional	0	Desvia incondicionalmente
23	**COPY**	Auxiliar	1	Copia 1 valor para outra instrução no mesmo pacote
24	**NOP**	Auxiliar	0	Sem operação
25	**SEND**	Sincronização	2	Envia um valor para um ser inserido em um pacote.
26	**IN**	Entrada/Saída	3	Recebe bytes do controlador de E/S
27	**OUT**	Entrada/Saída	3	Envia bytes ao controlador de E/S
28	**WAKEUP**	Sistema	1	Ordena a reinjetar um pacote antes interrompido
29	**NOTIFY**	Sistema	1	Notifica estado de um pacote
30	**CALL**	Procedimento	Vários	Faz a chamada de uma função/pacote
31	**RETURN**	Procedimento	2	Retorna o resultado de uma função para o chamador

Fonte: Araújo (2012)

Utilizando este conjunto de instruções, é possível desenvolver aplicações que podem estar dispostas em um ou mais pacotes, cada um contendo instruções e operandos. A PDL é uma linguagem equivalente ao *assembly* e de modo análogo deve ser montada, traduzida e carregada na memória para que seja executada. Para simulação de programas na arquitetura, é utilizado um ambiente de programação e simulação descritos em SystemC (Figura 16). O próprio ambiente implementa um montador que traduz o programa descrito em PDL para o código objeto da arquitetura.

Figura 16 - Ambiente de Programação e Simulação IPNoSys



Fonte: Araújo (2012)

O programa em PDL deve ser escrito em um editor de texto qualquer. O montador converte o código em PDL no código objeto que é passado como parâmetro para o simulador, o qual inicializa as memórias e prossegue a execução da aplicação. Durante a simulação, vários dados estatísticos são produzidos e armazenados em um arquivo texto.

Segundo Araújo (2012), a IPNoSys também pode ser comparada às arquiteturas VLIW. Computadores VLIW possuem núcleos que executam um grupo de instruções em paralelo. As instruções são unidas em uma única palavra. O compilador é responsável por juntar as instruções e garantir que não haja dependências entre elas.

Mais detalhes sobre o IPNoSys podem ser encontrados consultando Araújo (2012), Araújo et al (2009) e Filho et al (2014).

2.5 Trabalhos Relacionados

Como visto na Seção 2.3, a técnica de software *pipelining* traz um ganho sugestivo a muitas arquiteturas que a implementam para certas aplicações. Com isso, não é difícil encontrar autores que tenham trabalhado o tema.

Em 2007, Douillet e Gao (2007) implementaram a técnica de software *pipelining* em uma arquitetura *multicore*. O método utilizado consistia em separar as instruções livres de dependências em *threads* que eram entregues aos núcleos disponíveis para processamento. Quando uma instrução com dependência era encontrada, um sinal de *wait* era ativado e a aplicação era executada de forma sequencial, mesmo se houvesse núcleos disponíveis. Os resultados mostraram que a implementação teve um resultado favorável mesmo em aplicações com muitas dependências.

No trabalho desenvolvido por Gao et al (2009), foi desenvolvida uma técnica para explorar software *pipelining* em arquiteturas reconfiguráveis. O algoritmo foi implementado na linguagem C e convertido por meio de um software específico para simulação e testes na ferramenta ModelSim® da Altera©. Os resultados provaram que as arquiteturas com software *pipelining* tiveram um desempenho em torno de 35% melhor em comparação com os modelos originais.

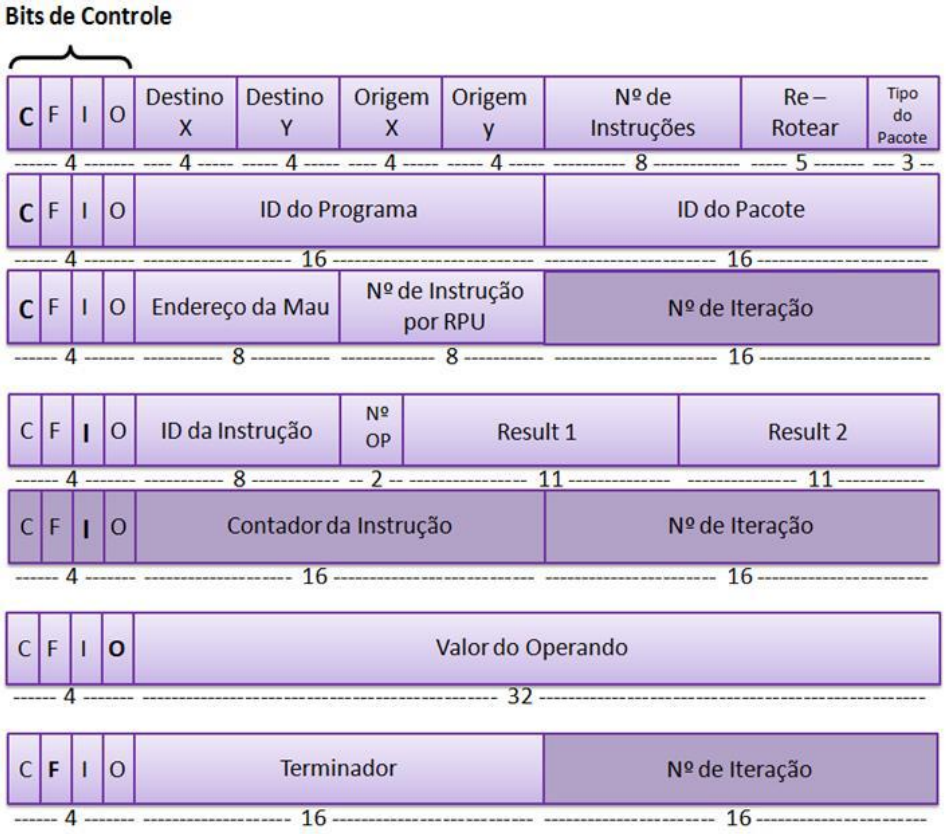
Em Wei et al (2012), foi desenvolvido um método de software *pipelining* para arquiteturas com recursos limitados. O algoritmo proposto escalona instruções nos processadores a fim de reduzir a taxa de comunicação e acesso a memória. Esta nova técnica foi comparada com outros modelos de ILP e obteve um resultado similar no desempenho e melhor na eficiência no uso de memórias e comunicação.

2.5.1 Software *Pipelining* e IPNoSys

Mais recentemente, os trabalhos de Medeiros (2014) e Medeiros (2015) apresentaram uma técnica de software *pipelining* para a arquitetura IPNoSys. Para efeito de validação da técnica, foi desenvolvido um simulador em C++ para a IPNoSys.

O modelo proposto modificava a estrutura do pacote e o algoritmo de roteamento para criar o efeito desejado. As modificações feitas nos pacotes foram a substituição do campo *Apontador* no cabeçalho do pacote pelo campo *Nº de Iteração* e a inclusão de uma nova palavra entre cada instrução e seu operando. Essa nova palavra contém os campos *Contador da Instrução* e *Nº de Iteração.* A Figura 17 mostra o novo pacote criado.

Figura 17 - Formato de Pacote Modificado Para Software *Pipelining*



Fonte: Medeiros (2014)

O modelo de execução proposto funciona da seguinte forma: A RPU retira uma instrução do pacote e verifica a próxima palavra. Se for um operando, a instrução é executada normalmente. Mas se for uma palavra com o as instruções de laço, a RPU executa a operação e envia a instrução e seus operandos para a próxima RPU para que a mesma seja executada novamente.

Os testes foram feitos a partir de uma aplicação genérica com um laço de repetição. As quantidades de iterações e tamanho da rede foram modificadas a fim de

se ter um quadro geral de várias situações possíveis. Os resultados mostraram que todas as simulações com software *pipelining* obtiveram um desempenho melhor do que a implementação com laço tradicional.

Apesar dessa proposta ter se mostrado bem melhor na execução de laços do que no modelo da IPNoSys original, o modelo proposto por Medeiros (2014) apresenta algumas fragilidades. O fato de se incluir uma nova palavra a cada instrução do laço e executar transmissões de pacotes em cada iteração são fatores negativos a esse modelo pois, segundo Araújo (2012), transmitir pacotes é o principal fator de queda de desempenho da IPNoSys. Outro ponto negativo é a limitação na quantidade de repetições possíveis para cada instrução. Como a instrução executa novamente a cada RPU que passa, a quantidade de execuções fica limitada a quantidade de RPUs no caminho do pacote. Como solução, o autor utilizou um algoritmo de roteamento implementado por Cruz (2013). Esse novo roteamento cria um caminho em zigue-zague de que o pacote possa sair de uma RPU, percorrer toda a rede e voltar a RPU original. Essa rota gera uma espécie de caminho "infinito" que o pacote pode percorrer e ter suas instruções executadas repetidamente.

Vale ressaltar também que a comparação dos resultados apresentados por Medeiros (2014) com a arquitetura original é prejudicada, visto que os testes não foram executados no mesmo simulador SystemC desenvolvido por Araújo (2012), e sim em um simulador próprio em linguagem de alto nível.

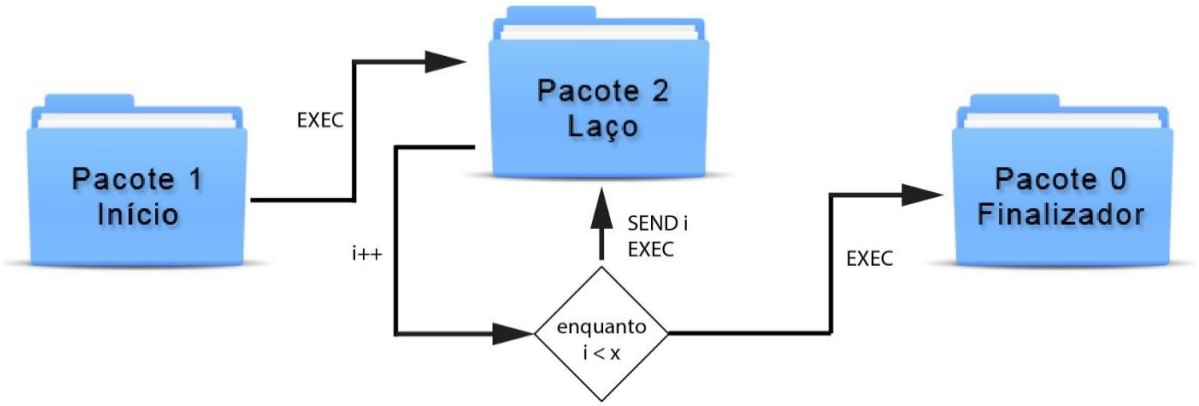
Diferentemente de Medeiros (2014) e Medeiros (2015), o trabalho apresentado nesta dissertação modificou a arquitetura da IPNoSys (montador e simulador originais) para que esta possa reconhecer e executar a técnica de software *pipelining* em aplicações reais, sem a alteração da estrutura do pacote ou algoritmo de roteamento e tentando reduzir a quantidade de transmissão entre as RPUs. A nova arquitetura apresentada tem compatibilidade completa com os programas desenvolvidos para a IPNoSys original e possivelmente, com desempenho melhorado nos laços de repetição, assim como aponta Medeiros (2014).

3 SOFTWARE *PIPELINING* NA IPNOSYS

O modelo de execução original da IPNoSys executa laços de repetição reinjetando o mesmo pacote na rede repetidamente. Esse processo se dá pela utilização das instruções EXEC e SEND ao fim do pacote. A instrução EXEC é responsável por fazer uma chamada a uma determinada MAU para injetar um novo pacote na rede, que no caso de laço de repetição, é o mesmo pacote que acabou de ser finalizado. A instrução SEND por sua vez, serve para enviar resultados entre pacote. A cada iteração do laço, instruções SENDs enviam os resultados para o novo pacote e a instrução EXEC chama o pacote para rede. Esse procedimento se repete conforme a quantidade de vezes definida pelo laço de repetição. Tal operação, apesar de funcional, tem o desempenho prejudicado pela quantidade de mensagens trocadas entre as RPUs e a MAU, para leitura e gravação de valores na memória e requisições para reinjetar o pacote.

É importante ressaltar que o laço deve ser composto por três pacotes. O primeiro com o código antes do laço de repetição que deve chamar o segundo pacote. O segundo pacote é o laço propriamente dito. No final deste, há uma tomada de decisão para determinar se o laço chegou ao fim ou se deve ser repetido novamente. Se houver uma nova iteração, o valor da variável de controle é incrementado e o pacote é novamente injetado na rede. Caso o laço tenha sido concluído, é chamado o terceiro pacote que representa o código após o laço. Na Figura 18 é mostrado esse procedimento. As instruções EXEC e SEND na Figura 18 representam pacotes de controle que são enviados das RPUs até a MAU que a executará tantas vezes quanto for a lógica do laço de repetição.

Figura 18 - Laço de Repetição Tradicional no IPNoSys



Fonte: Autoria Própria

De acordo com o apresentado na Seção 2.3, é possível utilizar a técnica de software *pipelining* para se conseguir um ganho de desempenho. Como forma de contribuir para o projeto IPNoSys, este trabalho implementou da técnica de software *pipelining* para a arquitetura em questão, com a avaliação do ganho de desempenho adquirido. A nova versão do IPNoSys software *pipelining* é dotada de mecanismos que executam instruções dentro de um laço de repetição de forma paralela, extraindo o máximo de desempenho do ILP.

3.1 Visão Geral

Para a implementação da técnica de software *pipelining*, criou-se duas novas instruções para a ISA da IPNoSys, chamadas de LOOP e RT (Tabela 2). A instrução LOOP serve para configurar as RPUs, delimitando quantas instruções pertencem ao laço e informando a quantidade de repetições as RPUs devem executar tais instruções. E a instrução RT transmite dados entre as RPUs quando há dependência de dados entre as instruções.

Tabela 2 - Instruções LOOP e RT

CodOp	Instrução	Tipo	Operandos	Descrição
32	**LOOP**	Controle de Fluxo	1	Define a operação de loop utilizando software *pipelining*
33	**RT**	Auxiliar	1	Carrega dados entre as instruções dentro de um laço de repetição

Fonte: Autoria Própria

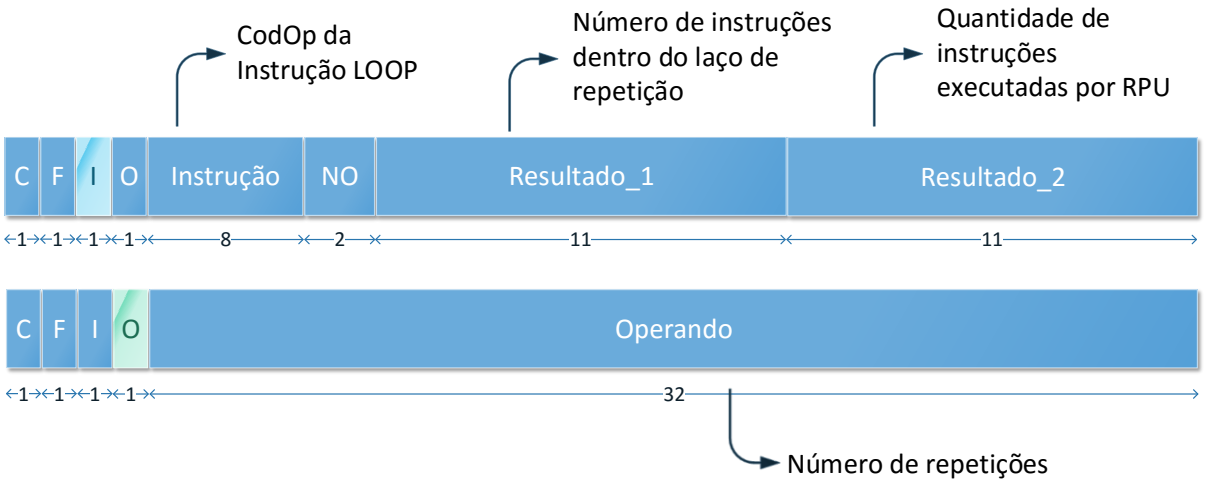
3.1.1 Instrução LOOP

A instrução LOOP utiliza dois parâmetros para configurar as RPUs. No espaço destinado ao *Resultado_1*, deve ser informada quantas das próximas instruções fazem parte do laço de repetição. Já no espaço dedicado ao *Resultado_2*, quantas instruções cada RPU no caminho do pacote vai executar (detalhamento na seção 3.2.1). A única palavra de *operando*, contém a quantidade de repetições que o laço deve executar. A Figura 19 contém a visão do programador (a) e os campos da palavra da instrução LOOP (b).

Figura 19 - PDL da Instrução LOOP (a) e formato da palavra (b)

```
1   LOOP qi ipr;   // Número de instruções do laço e IPR_LOOP
2        qr;       // Número de repetições
```

(a)



(b)

Fonte: Autoria Própria

Essa estratégia de informar a quantidade de instruções que fazem parte do laço serve para delimitar as instruções dentro dele. A Figura 20 mostra um algoritmo genérico com a sintaxe da instrução LOOP em PDL. Nela, de acordo com os parâmetros de LOOP, as instruções "$inst2$" e "$inst3$" seriam executadas dez vezes.

Figura 20 - LOOP Exemplo 01

```
1  inst1
2  LOOP 2 1;    // Quantidade de instruções do laço e IPR
3       10;     // Quantidade de repetições
4  inst2        // Inst. executada 10 vezes pela RPU A
5  inst3        // Inst. executada 10 vezes pela RPU B
6  inst4        // Inst. Fora do laço de repetição
```

Fonte: Autoria Própria

Enquanto o valor definido no campo *Resultado_1* diz a quantidade de instruções que fazem parte do laço, o valor do campo *Resultado_2* define a quantidade de instruções que serão executadas por cada RPU. Se esse valor for omitido pelo programador, o compilador define esse valor como "1" por padrão. Mais detalhes da utilidade desse campo serão mostrados nas sessões seguintes.

3.1.2 Instrução RT

O acrônimo RT foi usado para simbolizar o termo "retransmitir". A função dessa instrução é enviar o resultado de uma operação dentro do laço quando há dependência de dados com outras instruções que também estão dentro do laço. Observando o algoritmo da Figura 21, podemos perceber que a variável $op2$ depende de $op1$ e a variável $op3$, depende de $op2$. Em um cenário de execução com software *pipelining* proposto por este trabalho, as instruções das linhas 4, 5 e 6 ficariam em unidades funcionais diferentes. De modo que a unidade que estivesse executando a instrução da linha 4, deve enviar o resultado da operação para a unidade que esteja executando a linha 5 e assim por diante.

Figura 21 - Exemplo de Laço de Repetição Com Dependência de Dados

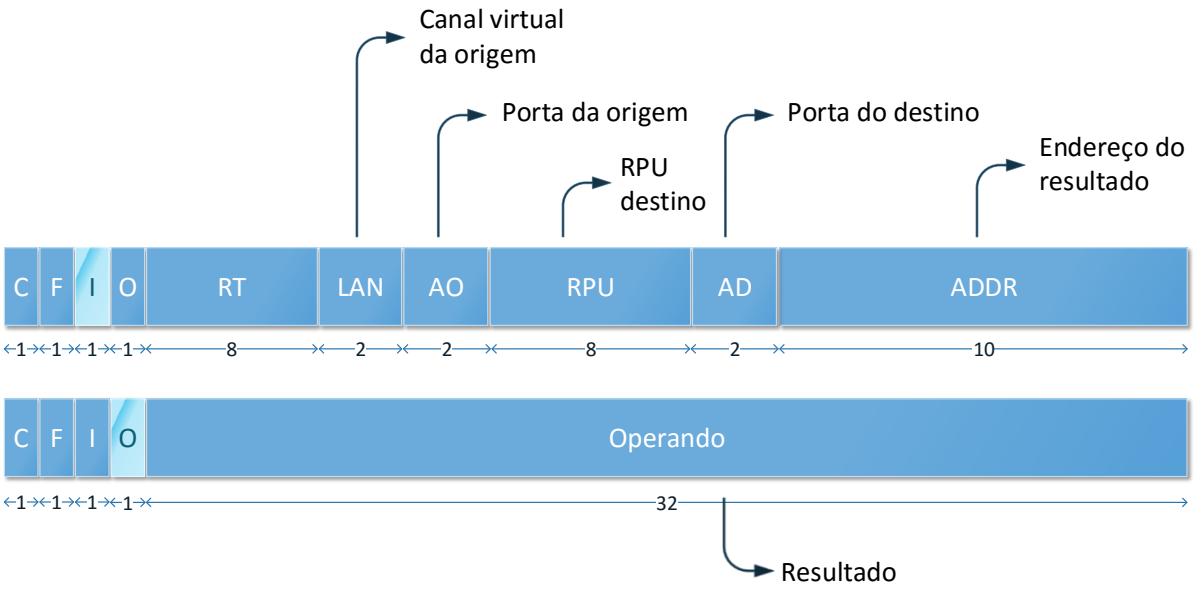
```
1  algoritmo exemplo 1
2     x, i, op1, op2, op3 : inteiro
3     para i de 0 ate x faca
4        op1 = i
5        op2 = op1 + 1
6        op3 = op2 + 2
7     fimpara
8  fimalgoritmo
```

Fonte: Autoria própria

A instrução RT é utilizada para encaminhar o resultado da operação de uma RPU para outra quando se está dentro de um laço de repetição. O formato da instrução RT é mostrado na Figura 22. Assim como as instruções RETURN, RELOAD, NOTIFY etc, a instrução RT não está disponível para o programador. O árbitro que está executando o pacote é que cria a instrução em tempo de execução de forma transparente para o usuário.

Os campos da palavra da instrução RT são diferenciados em relação as outras instruções da IPNoSys. O fato se deve a quantidade de informações que a RPU necessita para que um dado seja salvo em um de seus *buffers* de resultado. Deste modo, o pacote contendo a instrução RT deve transportar as informações do endereço da RPU e a porta do destino que irá receber o dado, a porta e o canal virtual de origem, o endereço da palavra e o dado propriamente dito. Para isso, o árbitro que está executando a instrução que irá propagar o resultado identifica a RPU destino e envia para a Unidade de Sincronização (SU) duas palavras como mostra o diagrama da Figura 22.

Figura 22 - Palavras da Instrução RT do árbitro para SU

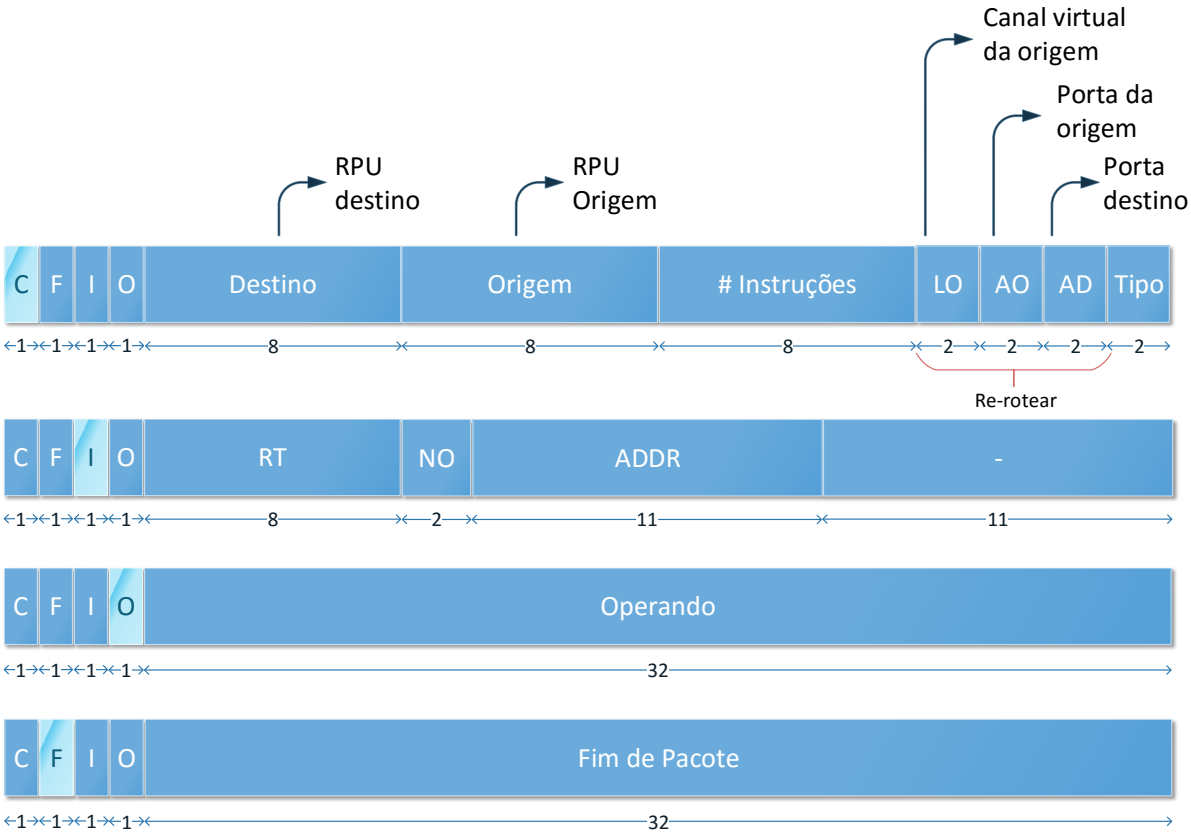


Fonte: Autoria própria

Ao chegar na SU, as duas palavras são adicionadas a uma palavra de cabeçalho e uma palavra de fim de pacote para criar um pacote de controle (Figura 23). A união destas 4 palavras forma um pacote de controle que é injetado na rede e roteado até a

RPU que estará executando a operação que necessitará do dado. As informações para roteamento são retiradas da palavra do RT e colocadas no cabeçalho. Depois do pacote pronto, a palavra RT fica semelhante as palavras das demais instruções da IPNoSys.

Figura 23 - Pacote de Controle com a Função RT



Fonte: Autoria própria

O campo "re-rotear" do cabeçalho não é utilizado nos pacotes de controle e foi convenientemente aproveitado para acondicionar as informações dos canais virtuais e portas. A palavra de instrução RT foi colocada no formato padrão das outras palavras da IPNoSys para evitar modificações no momento da decodificação quando ela chega na RPU destino.

É importante chamar atenção para o campo destinado ao endereço da palavra do dado (ADDR), que no momento em que o árbitro envia a instrução RT para a SU (Figura 22), é colocado em um campo com 10 bits. E no momento em que o pacote é criado pela SU, o ADDR volta a ter os 11 bits que as palavras de IPNoSys tem. Essa "compressão" da informação em 10 bits não gera problemas pois segundo Araújo

(2012), um pacote regular da IPNoSys pode conter até 256 instruções. Apenas as instruções STORE e SYNEXEC utilizam mais do que dois operandos. Com isso, num caso extremo onde todas as instruções da IPNoSys têm 4 palavras (uma de instrução e três operandos), o pacote teria $256\ \text{instruções} + (3 * 256\ \text{operandos})$. Ou seja, um total de 1.024 palavras de instruções onde poderiam ser inseridos resultados. Deste modo, são necessários apenas 10 bits ($2^{10} = 1.024$) no campo *Result_1* e *Result_2* para identificar todos os endereços possíveis em um pacote.

3.2 Implementação

A implementação do software *pipelining* na IPNoSys foi dividida em duas partes. A primeira foi a criação de mecanismos para a execução de instruções repetidas vezes, como em um laço de repetição. A segunda parte foi a identificação e tratamento das dependências de dados.

3.2.1 Definindo o Laço de Repetição

Para reconhecer e executar um laço de repetição em modo de software *pipelining*, as RPUs são configuradas para retirarem uma instrução que faz parte do laço e enviar o restante do pacote para próxima RPU. Esse procedimento deve ocorrer n vezes, em que n é a quantidade de instruções dentro do laço. Ao fim desse processo, cada RPU ficará com apenas uma única instrução (por padrão). Após retirar sua instrução e transmitir o restante do pacote, a RPU deve iniciar o processo de execução a quantidade de vezes definida no laço. Para a configuração das RPUs, a instrução LOOP é propagada junto com o resto do pacote. Para isso, cada RPU do caminho retira a instrução LOOP; decrementa o valor de n; retira a próxima instrução, que será a instrução a ser executada pelo laço; insere a instrução LOOP no pacote, com o n decrementado, e envia o pacote para a próxima RPU. A Figura 24 contém o algoritmo utilizado na implementação.

Figura 24 - Algoritmo de Execução do LOOP

```
1  instrucao = pop_buffer()
2  se (instrucao = LOOP) entao
3  |  m = result_1  //quantidade de repetições
4  |  n = operandA  //quatidade de instruções dentro do laço
5  |  prox_instrucao <- pop_buffer()
6  |  n--
7  |  se (n > 0) entao
8  |  |  pacote_loop <- monta_pacote(loop, m, n)
9  |  |  push(pacote_loop)
10 |  |  transmite()
11 |  fimse
12 |  enquanto (i < m) faca
13 |  |  executa(prox_instrucao)
14 |  |  i++
15 |  fimenquanto
16 fimse
```

Fonte: Autoria Própria

Além da alteração no simulador do IPNoSys, também foi necessária a adaptação no montador para que as instruções de LOOP sejam reconhecidas nos arquivos de código-fonte PDL.

Por questões arquiteturais apenas algumas instruções podem ser executadas em modo de software *pipelining*. São elas: as instruções lógicas e aritméticas (ADD, SUB, MULT, DIV, AND, OR, XOR, RSHIFT, LSHIFT e NOT), instruções de acesso a memória (LOAD e STORE) e as auxiliares (COPY e NOP). As demais instruções não têm escopo definido para execução dentro de um laço de repetição utilizando a instrução LOOP. Dessa forma, um compilador poderia identificar se as instruções contidas dentro do laço são compatíveis com a instrução LOOP, caso contrário implementaria o laço no formato original da arquitetura.

Para comparar o escopo do PDL, as Figura 25 e Figura 26 trazem, respectivamente, a implementação de um algoritmo de um contador simples para versão original da IPNoSys e para a versão IPNoSys SP[2]. A versão da IPNoSys original (Figura 25) contém os três pacotes necessários para criação do laço (Figura 18). O

[2] Para simplificar a escrita, deste ponto em diante, a arquitetura proposta por esta dissertação será referenciada sempre por IPNoSys SP (software *pipelining*).

pacote "*inicio*" (linhas 6 a 18) inicializa o programa e chama o pacote "*laco*". O pacote "*laco*" (linhas 19 a 49) contém o código que será executado repetidamente e, em seguida, uma tomada de decisão para saber se o pacote deve executar de novo ou deve ser chamado o pacote "*fim*". O pacote "*fim*", por sua vez, guarda o resultado na memória e encerra o programa. É fácil notar que boa parte código é composto por instruções auxiliares necessárias ao controle do laço (SENDs e EXECs). O contador propriamente dito é restrito apenas a instrução ADD (linhas 25 a 27).

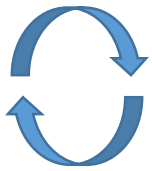
Figura 25 - PDL de um Laço de Repetição na IPNoSys Original

```
1   PROGRAM loop_comum
2   DATA
3         n = 10
4         soma
5
6   PACKAGE inicio
7   ADDRESS MAU_0
8         LOAD MAU_0 n1;
9              n;
10        SEND MAU_0 i;
11             prog, laco
12             n1;
13        SEND MAU_0 s;
14             prog, laco
15             1;
16        EXEC MAU_0;
17             prog, laco;
18  END

    //-------------------->

50  PACKAGE fim
51  ADDRESS MAU_0
52        COPY s1;
53             s = 0;
54        STORE MAU_0;
55             s1
56             soma;
57        EXIT;
68  END
59  END_PROGRAM

19  PACKAGE laco
20  ADDRESS MAU_0
21        COPY i1;
22             i = 0;
23        COPY r1;
24             s = 0;
25        ADD result result2
26             r1
27             1;
28        SUB isomado x;
29             i1
30             1;
31        BE fim;
32             x
33             0;
34        SEND MAU_0 i;
35             prog, laco
36             isomado;
37        SEND MAU_0 s;
38             prog, laco
39             result;
40        EXEC MAU_0;
41             prog, laco;
42        JUMP rpt;
43  fim:  SEND MAU_0 s;
44             prog, fim
45             result2;
46        EXEC MAU_0;
47             prog, fim;
48  rpt:  NOP;
49  END
```



Fonte: Autoria Própria

Em contrapartida, a Figura 26 apresenta o PDL para o mesmo algoritmo, mas agora utilizando a instrução LOOP.

Figura 26 - PDL de um Laço de Rpetição na IPNoSys SP

```
1   PROGRAM loop_SP
2   DATA
3         n = 10
4         soma
5   PACKAGE inicio
6   ADDRESS MAU_0
7         LOAD MAU_0 loop;
8               n;
9         LOOP 1 1;
10              loop;
11        ADD  s1 result;
12              s1 = 1
13              1;
14        STORE MAU_0;
15              result
16              soma;
17        EXIT;
18  END
19  END_PROGRAM
```

Fonte: Autoria Própria

Enquanto o laço de repetição na IPNoSys original (Figura 25) necessitou de três pacotes e 59 linhas de código, a versão com SP (Figura 26) é escrita com apenas um pacote e 19 linhas de código. O contador, representado pela instrução ADD (linhas 11 a 13) irá executar a quantidade de vezes determinada pelo operando da instrução LOOP (linhas 9 e 10). Como não há mais necessidade de outros pacotes, as instruções de controle são dispensadas.

Para mostrar o passo-a-passo da instrução LOOP, a Figura 28 mostra um código em PDL para resolução do fatorial de um número qualquer. O PDL foi baseado no algoritmo de fatorial descrito na Figura 27. A versão utilizando o laço de repetição da IPNoSys original para o mesmo fatorial pode ser consultado no Apêndice A.

Figura 27 - Algoritmo de Fatorial

```
1  ALGORITMO fatorial
2  var
3       a, b, i, loop: inteiro
4  inicio
5  |    a <- 5
6  |    b <- 1
7  |    loop < a - 1
8  |    para i de 1 ate loop faca
9  |    |    b <- b * i
10 |    fimpara
11 |    escreva(b)
12 fimalgoritmo
```

Fonte: Autoria Própria

Figura 28 - PDL de um Fatorial

```
1  PROGRAM fatorial
2  DATA
3       a = 5 // valor que se deseja saber o fatorial
4       b     // endereço onde o resultado será salvo
5  PACKAGE pac
6  ADDRESS MAU 0
7       LOAD MAU 0 fat;
8                  a;
9       COPY       fat1 fat2;
10                 fat;
11      SUB        loop;
12                 fat1
13                 1;
14      LOOP       2 1;
15                 loop;
16      ADD        fat3 fat4;
17                 fat3 = 1
18                 1;
19      MUL        result1 result2;
20                 result1 = 1
21                 fat4;
22      STORE MAU 0;
23                 result2
24                 b;
25      EXIT;
26 END
27 END_PROGRAM
```

Fonte: Autoria própria

O código em PDL (Figura 28) inicia com a seção de dados (linhas 3 e 4). Na linha 3, tem-se o número que desejamos saber o fatorial. E na linha 4, o resultado da operação. Após a seção de dados, inicia-se o pacote com as instruções a serem executadas.

Levando em consideração a IPNoSys SP de ordem 4 (4x4 ou 16 RPUs), a execução do programa fatorial ficaria assim: A RPU 0,0 retiraria a instrução LOAD e seu operando do pacote e solicitaria a MAU_0 o dado referente ao endereço de memória a. A MAU_0, através de uma instrução RELOAD, enviaria o valor do endereço a para a RPU 0,0. Com a chegada do RELOAD, a RPU 0,0 inicia o processo de transmissão do resto do pacote para a próxima RPU. Sendo assim, a RPU 0,1 recebe a instrução COPY e seu operando. Executa a operação, que é copiar o valor do endereço de fat para $fat1$ e $fat2$, e transmite o resto do pacote. A RPU 0,2 recebe a instrução SUB, executa, subtraindo 1 do valor do endereço de $fat1$ e colocando o resultado em $loop$, e transmite o resto do pacote.

Nesse ponto encontramos a instrução LOOP, a ser executada pela RPU 0,3. O LOOP foi definido como contendo duas instruções que irão executar a quantidade de vezes definido dentro do endereço $loop$. Desse modo, a RPU 0,3 retira a instrução LOOP e seu operando do pacote, retira a próxima instrução (ADD) e seus operandos e guarda as informações em registradores de configuração. Em seguida a RPU 0,3 decrementa o valor da quantidade de instruções dentro do LOOP (pois ela já retirou o ADD) e transmite uma nova instrução LOOP para a próxima RPU.

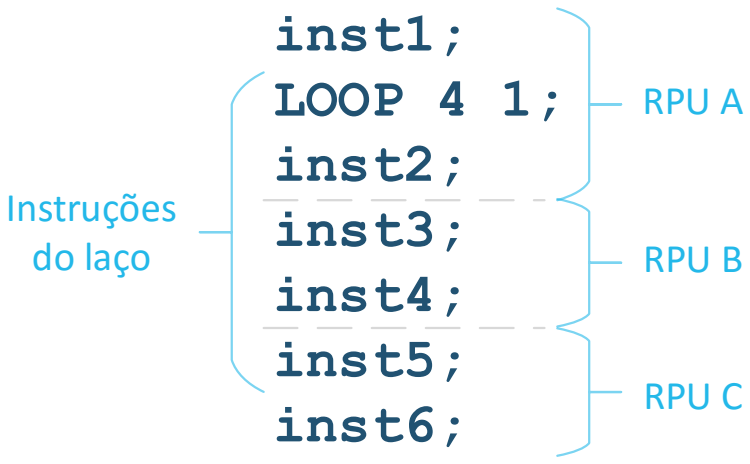
Após salvar a instrução ADD e seus operandos dentro de registradores e enviar o LOOP, a RPU 0,3 inicia o processo de execução da instrução ADD e transmissão do resto do pacote. A execução e transmissão é feita de modo paralelo já que são executados por unidades distintas dentro da RPU. De modo a garantir consistência nos dados, as RPUs que começam a trabalhar em modo software *pipelining* só transmitem instruções que estejam dentro do laço de repetição. Isso impede que alguma instrução que esteja depois do laço execute antes das instruções que estejam dentro do laço finalizarem a execução. Sendo assim, como esse exemplo contém duas instruções dentro laço, a RPU 0,3 retira a ADD, transmite o LOOP que agora só informa

uma instrução dentro do laço, e transmite a instrução MUL. O resto do pacote fica aguardando o término do laço para poder ser transmitida.

A RPU 1,3, a próxima do caminho, recebe a instrução LOOP, informando apenas uma instrução dentro do laço, e em seguida recebe a instrução MUL. O resto do pacote só chega na RPU 1,3 quando a RPU 0,3 terminar de executar a instrução ADD a quantidade de vezes definida dentro do laço. Nesse momento, as RPUs 0,3 e 1,3 estão executando as instruções ADD e MUL respectivamente, de forma paralela. Nesse ponto acontece de fato o software *pipelining*. Com o término da execução do ADD pela RPU 0,3 o resto do pacote é enviado a RPU 1,3. Quando a RPU 1,3 finalizar a execução da instrução MUL, o pacote segue para a RPU 2,3, que recebe a instrução STORE e finaliza o programa.

O cabeçalho de um pacote regular contém um campo chamado IPR (Instruções por RPU). Esse valor é definido pelo programador e define quantas instruções cada RPU no caminho do pacote vai executar. Para manter total compatibilidade, foi necessário que esse mecanismo fosse suprimido das RPUs que forem executar instruções no modo loop. Obrigar as RPUs a executarem mais de uma instrução pode ocasionar cenários em que instruções que não fazem parte do laço dividam RPUs que estão configuradas para trabalhar com software *pipelining*. A Figura 29 mostra um modelo de programa em PDL de uma execução hipotética com o IPR definido com o valor "2" (duas instruções por RPU).

Figura 29 - Execução Hipotética com IPR = 2

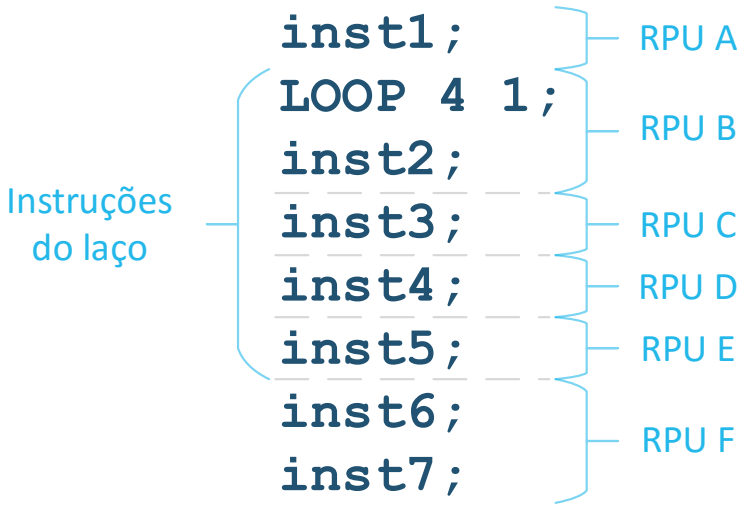


Fonte: Autoria própria

É possível observar a partir da Figura 29 que as instruções $inst1$ e $inst6$ estão, respectivamente, antes e depois do laço e são executadas por RPUs que foram configuradas pela instrução LOOP.

Devido a esse problema, quando a RPU identifica a instrução LOOP, o valor do IPR configurado pelo cabeçalho do pacote é modificado para "1" durante a execução do LOOP. As RPUs que executam as instruções antes e depois do laço ficam com seus valores de IPR inalterados. A forma de execução do PDL da Figura 29 de forma correta é mostrada na Figura 30.

Figura 30 - Execução com IPR = 2

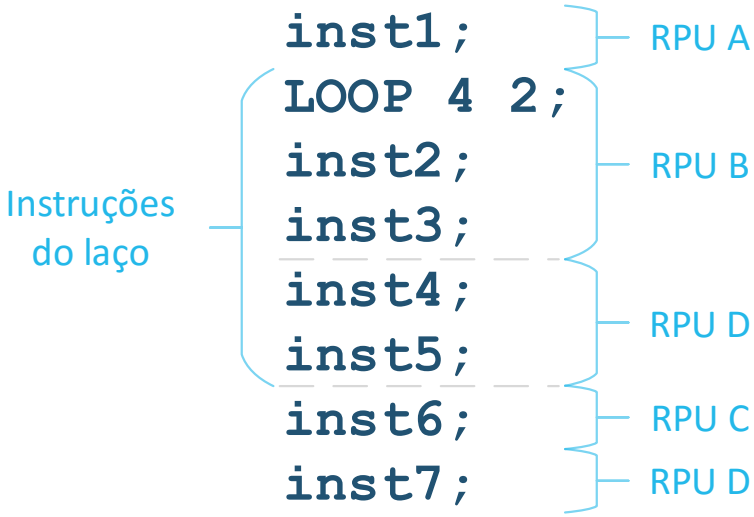


Fonte: Autoria própria

Obrigar cada RPU do caminho executar apenas uma instrução do laço limita a quantidade de instruções que podem ser colocadas dentro do laço. Devido a isso, foi adicionado o argumento IPR_LOOP do laço (o campo *Result_2* assume este papel nesta instrução) na definição da instrução LOOP (Figura 19). Esse campo tem o mesmo objetivo do campo IPR do cabeçalho, mas voltado apenas para as instruções que estejam dentro do laço. Por padrão, como visto em todos os exemplos até agora, esse valor é "1"[3], mas o programador tem possibilidade de alterar esse número, fazendo as RPUs que irão executar as instruções dentro do laço executem mais de uma instrução. Um exemplo disso pode visto na Figura 31.

[3] O valor "1" para o IPR_LOOP também é assumido se o programador suprimir essa informação para o montador.

Figura 31 - Execução com IPR_LOOP = 2



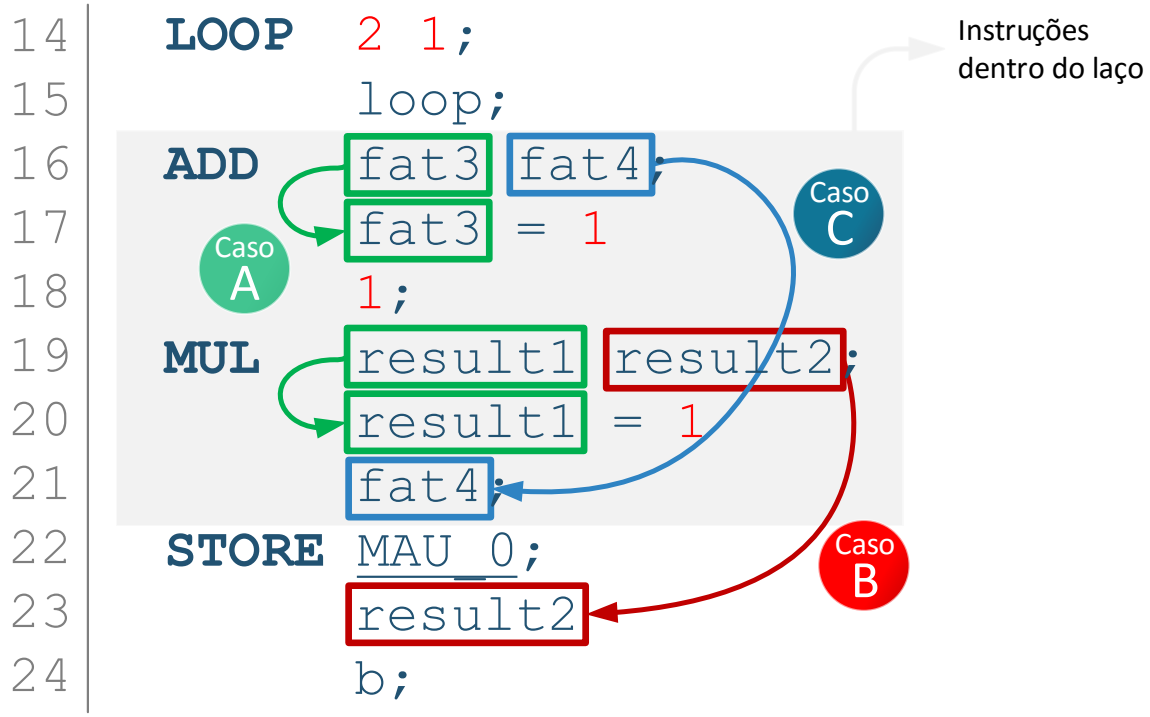
Fonte: Autoria própria

Os valores de IPR do cabeçalho e IPR_LOOP podem ser combinados para satisfazer a necessidade do programador/compilador e obter o melhor desempenho dependendo da aplicação.

3.2.2 Dependência de Dados

Ainda utilizando o programa da Figura 28 em que temos o código PDL para execução de um fatorial, podemos notar que existem dependências de dados entre as instruções que constituem o laço de repetição. Como exemplo pode-se notar o dado referenciado pelo endereço $fat4$. O valor de $fat4$ é atualizado pela instrução ADD e depois utilizado pela instrução MUL em uma RPU diferente. Caso esse tipo de cenário não seja tratado, o dado na instrução MUL pode chegar desatualizado ou até mesmo nem chegar. O modelo implementado por este trabalho identifica e trata três tipos diferentes de dependência de dados. Os três tipos são mostrados na Figura 32. São eles: dependência de dados dentro da própria instrução (caso A - em verde), dependência de dados entre instruções que estão dentro e fora do laço (caso B - em vermelho) e dependência de dados entre instruções diferentes dentro do laço (caso C - em azul). Por questões didáticas, a partir deste ponto os casos serão identificados apenas como dependência de dados do tipo A, B e C respectivamente.

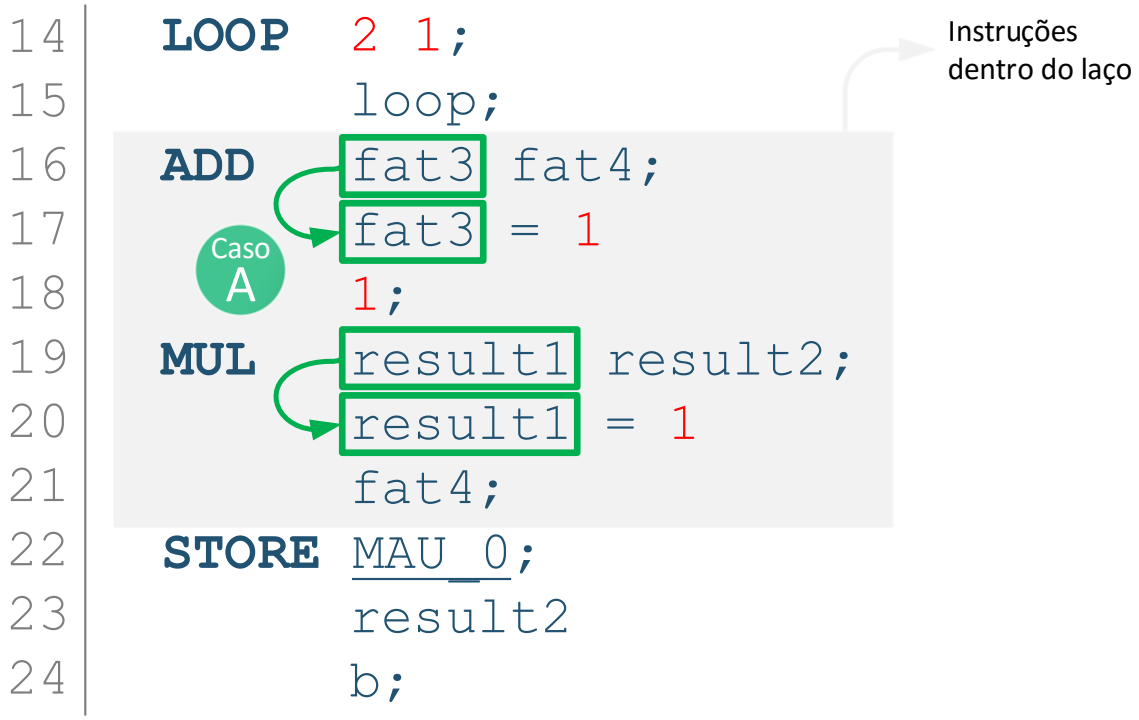
Figura 32 - Dependências de Dados Dentro do LOOP



Fonte: Autoria Própria

As dependências do tipo A e B foram as mais simples de serem resolvidas. A do tipo A, como ocorre apenas dentro da mesma RPU, o árbitro se encarrega de identificar a atualizar o valor no endereço especificado a cada iteração do laço. A Figura 33 destaca as dependências do tipo A no exemplo da Figura 32, nas linhas 16, 17 e 18 temos um contador simples. Nesse caso, há necessidade do programador inicializar o operando explicitamente com um valor *default* (como em $fat3$ e $fat4$ na Figura 33). Na primeira iteração a instrução ADD realiza a soma de $fat3$ com "1" e envia o resultado para os endereços $fat3$ e $fat4$. Como $fat3$ pertence a instrução ADD, o valor é atualizado dentro do próprio árbitro, e novo valor é utilizado a cada iteração no lugar do valor *default* ou o valor da iteração anterior.

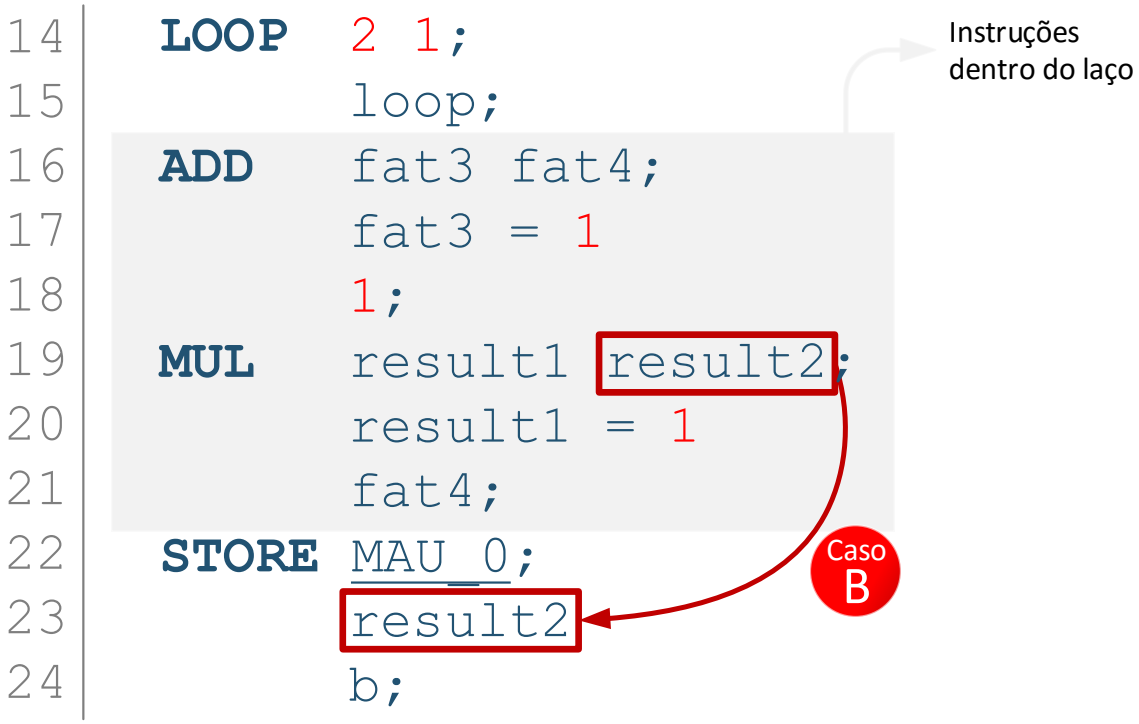
Figura 33 - Dependência de Dados do Tipo A



Fonte: Autoria Própria

Por outro lado, o caso B, onde a dependência ocorre com uma instrução dentro e outra fora do laço, foi solucionado evitando que o valor atualizado do endereço fosse propagado a cada iteração. Observando as instruções MUL e STORE (linhas 19 a 24) do exemplo da Figura 34 podemos notar o valor referenciado por $result2$. Esse valor é o resultado da multiplicação do resultado da operação anterior (ADD) por $result1$. A cada iteração do laço, o valor de $result2$ é atualizado. No final do laço $result2$ contém o valor do fatorial. De modo que só é necessário fazer o STORE desse valor final. Os valores intermediários que $result2$ assume não são necessários para as instruções que o utilizam após o laço. Com isso, quando o árbitro identifica esse tipo de cenário, ele desconsidera a existência do endereço $result2$ até a última iteração do laço. Quando o laço termina, o valor de $result2$ é gravado no *buffer* e propagado junto com o resto do pacote para as próximas RPUs como no modelo original da arquitetura visto na seção 2.4.2.

Figura 34 - Dependência de Dados do Tipo B

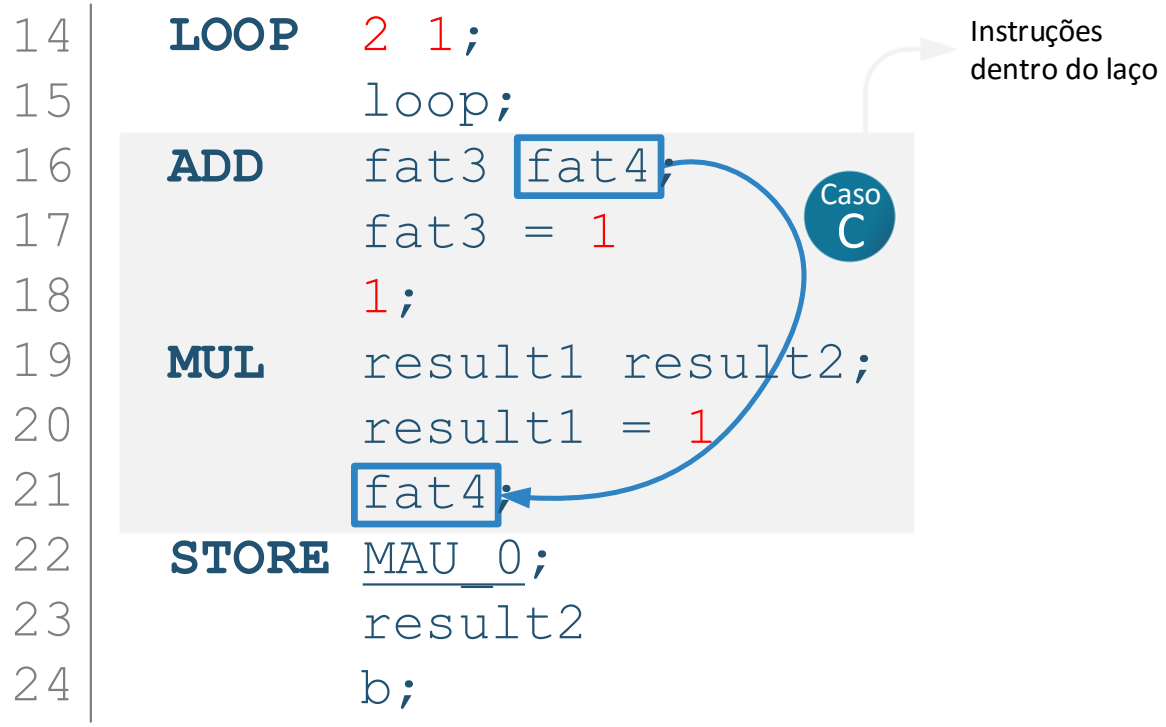


Fonte: Autoria Própria

A dependência de dados do tipo C é a mais complexa das três, pois ela ocorre entre instruções dentro do laço. Isso obriga a RPU a propagar o resultado de sua operação a cada iteração do laço para uma RPU destino que utilizará esse dado na sua operação. Esse caso é mostrado pelo endereço $fat4$ nas instruções ADD e MUL (linhas 16 a 21) no exemplo da Figura 35.

Como forma de solucionar esse problema, foi implementada a instrução auxiliar RT. A cada iteração do laço, o árbitro identifica o endereço da RPU destino comparando o endereço da palavra atual com a palavra de destino. Com essa informação o árbitro envia o as informações para a Unidade de Sincronização. Essa, por sua vez, cria um pacote de controle com a instrução RT e transmite para a RPU destino.

Figura 35 - Dependência de Dados do Tipo C



Fonte: Autoria Própria

Na execução do algoritmo da Figura 36, existe a dependência de dados do tipo "C". Uma amostra de como esse seria executado dentro no modelo de software *pipelining* proposto pode ser visto na Figura 37.

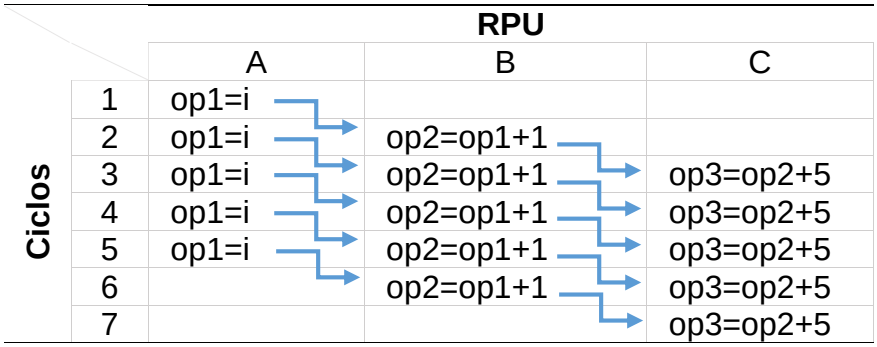
Figura 36 - Laço de Repetição Com Dependência de Dados Tipo "C"

```
1  algoritmo exemplo_1
2     x, i, op1, op2, op3 : inteiro
3     para i de 0 ate 5 faca
4     |   op1 = i
5     |   op2 = op1 + 1
6     |   op3 = op2 + 5
7     fimpara
8  fimalgoritmo
```

Fonte: Autoria própria

A seta azul na Figura 37 representa a propagação do resultado feito através da instrução RT. A cada iteração, o resultado é enviado para a RPU que está executando a instrução que utilizará o dado.

Figura 37 - Pipeline Preenchido



Fonte: Autoria própria

O algoritmo de tratamento de dependências de dados foi resumido na Figura 38. Esse algoritmo é executado ao fim de cada iteração do laço pelo árbitro que está no modo LOOP.

Figura 38 - Algoritmo Tratamento de Dependências de Dados

```
1  ALGORITMO trata_dependencia
2      se end_resultado <= end_palavra_atual entao
3          salvar_dado_no_buffer() // Caso A
4      senao se end_resultado > (end_palavra_atual +
                   (3 * qtd_inst_loop)) entao
5          se ultima_iteracao_do_loop() entao
6              salvar_dado_no_buffer() // Caso B
7          fimse
8      senao
9          rpu = identificar_RPU_destino()
10         enviar_RT(resultado, rpu); // Caso C
11     fimse
12 fimalgoritmo
```

Fonte: Autoria própria

Observando a linha 4 do algoritmo (Figura 38), nota-se que para entrar no caso Tipo B, deve-se verificar se o endereço do resultado (entenda-se como a posição da palavra do pacote onde o resultado deve ser inserido) é maior do que o endereço da palavra atual somado com o valor da quantidade de instruções dentro do laço multiplicado por três. Essa multiplicação ocorre para que seja possível delimitar a quantidade de palavras que existe dentro do laço.

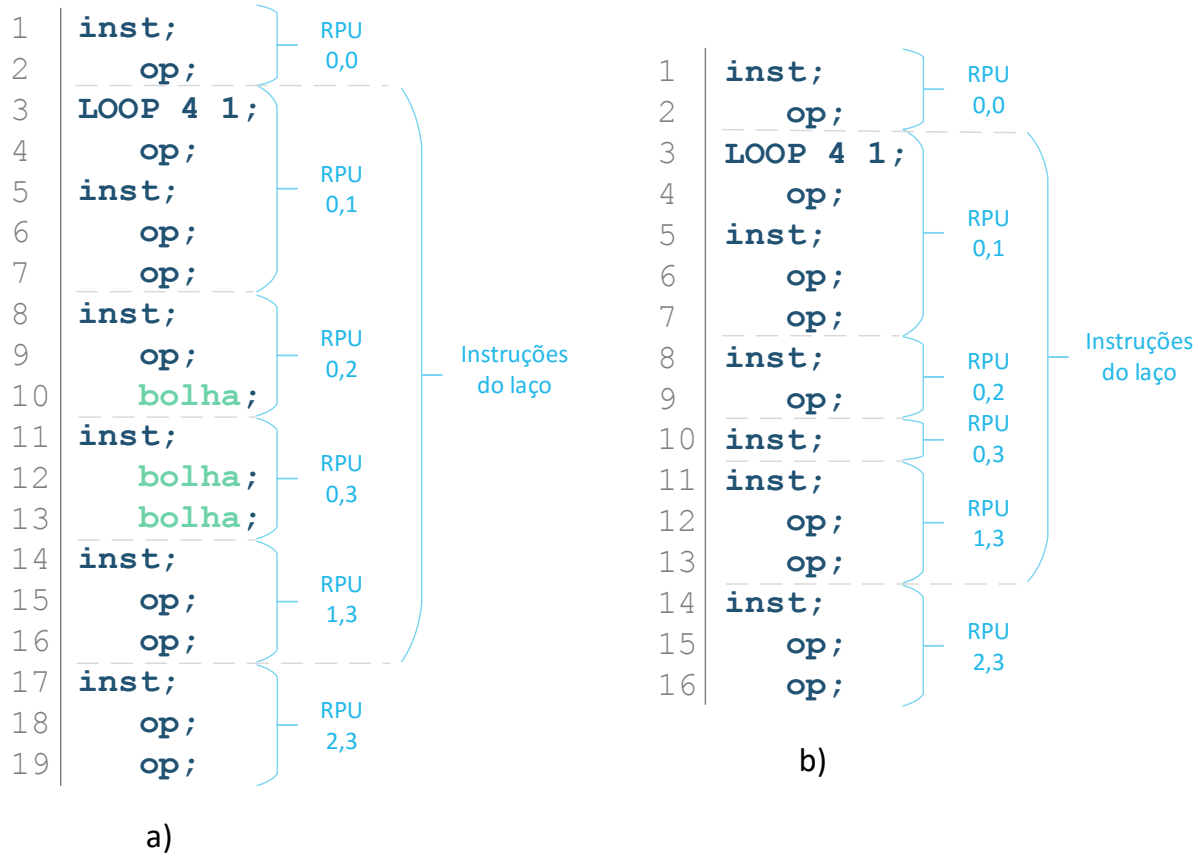
A IPNoSys padrão possui instruções com quantidades de palavras variadas. A instrução NOP possui apenas uma palavra (sem operandos). Enquanto um ADD possui 3 (uma palavra de instrução e duas de operandos). Quantidades variadas de palavras por instrução impede que seja possível prever qual o endereço da última palavra que faz parte do laço ou qual RPU receberá qual palavra através da instrução RT. Esse cenário obrigou a fixar o número de palavras por instrução dentro do laço em 3. Esse caso também é totalmente transparente ao programador, sendo responsabilidade do *assembler* inserir "bolhas" entre as instruções de modo a obter um número fixo de palavras. Se um programa estiver sendo escrito e for necessário uma instrução NOP dentro de um laço, o *assembler* automaticamente insere duas "bolhas" (palavras com o operando "0") logo após a instrução NOP, da mesma forma insere para as demais instruções dentro do laço para que todas tenham exatamente 3 palavras (incluindo a palavra de instrução).

Durante a execução de uma instrução que está dentro do laço, a RPU sempre irá retirar três palavras do pacote. Como o árbitro sabe o número de operandos de cada instrução, as palavras a mais são identificadas como bolhas e descartadas. Por exemplo: uma instrução LOAD tem apenas um operando. Mas se esse LOAD estiver dentro de um laço, o *assembler* adiciona um segundo operando com valor "0". Durante a execução, a RPU irá retirar as três palavras (instrução LOAD e os dois operandos). Mas ela sabe que o LOAD só necessita de um operando. Desta forma, o segundo operando (a bolha) é descartada.

Esse artifício faz com que seja possível prever qual RPU receberá qual instrução e consequentemente, qual RPU o RT deve enviar o operando. Por exemplo: em um cenário em que a RPU 0,1 recebe a instrução LOOP informando que há 4 instruções dentro do laço. A RPU 0,2 está executando a palavra de endereço 10 e precisa mandar o resultado para o endereço 16, como cada RPU retira 3 palavras do pacote, ela consegue identificar que as palavras 11, 12 e 13 ficarão de posse da RPU 0,3, as palavras 14, 15 e 16 com a RPU 1,3 (Figura 39a). Baseado nisso, a instrução RT é endereçada para a RPU 1,3 e o dado enviado. A Figura 39b mostra o mesmo programa sem as bolhas. É possível ver que, devido a diferença entre o "tamanho" das instruções, a palavra de endereço 16 está na RPU 2,3, fora do laço. Vale observar que o *assembler* só

irá inserir as bolhas nas instruções que estiverem dentro do laço e que possuam menos que 3 operandos. As instruções fora do laço continuam com seu escopo inalterado.

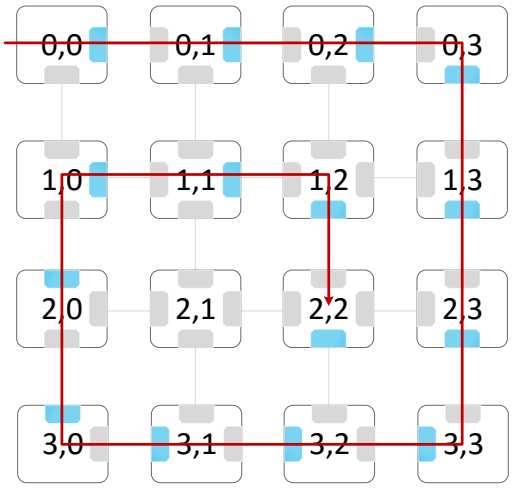
Figura 39 - Bolhas no Software *Pipelining*



Fonte: Autoria própria

Além de saber a RPU destino, o pacote com a instrução RT também deve saber qual árbitro que está com a instrução que usará o resultado propagado. Cada RPU contém 2, 3 ou 4 árbitros, cada um ligado a cada porta de saída. Para identificar o árbitro, é analisado o algoritmo de roteamento *Spiral Complement*. Como esse algoritmo é determinístico, é possível saber com antecedência o árbitro alvo verificando a porta. A Figura 40 mostra, em azul, as portas no caminho do *spiral complement* na IPNoSys 4x4 com o pacote saindo da MAU 0. Enquanto a Tabela 3 relaciona o endereço de todas as RPUs e árbitros (N – norte, S – sul, L – leste, O – oeste) no caminho dos quatro caminhos da *spiral complement* na IPNoSys 4x4.

Figura 40 - Portas no caminho do *Spiral Complement*



Fonte: Autoria própria

Tabela 3 - Tabela de predição de RPU e porta

	Spiral 0		**Spiral 1**		**Spiral 2**		**Spiral 3**	
#	**RPU**	**Porta**	**RPU**	**Porta**	**RPU**	**Porta**	**RPU**	**Porta**
0	0,0	L	3,0	L	3,3	O	0,3	O
1	0,1	L	3,1	L	3,2	O	0,2	O
2	0,2	L	3,2	L	3,1	O	0,1	O
3	0,3	S	3,3	N	3,0	N	0,0	S
4	1,3	S	2,3	N	2,0	N	1,0	S
5	2,3	S	1,3	N	1,0	N	2,0	S
6	3,3	O	0,3	O	0,0	L	3,0	L
7	3,2	O	0,2	O	0,1	L	3,1	L
8	3,1	O	0,1	O	0,2	L	3,2	L
9	3,0	N	0,0	S	0,3	S	3,3	N
10	2,0	N	1,0	S	1,3	S	2,3	N
11	1,0	L	2,0	L	2,3	O	1,3	O
12	1,1	L	2,1	L	2,2	O	1,2	O
13	1,2	S	2,2	N	2,1	N	1,1	S
14	2,2	S	1,2	N	1,1	N	2,1	S

Fonte: Autoria própria

Quando uma RPU recebe um pacote com a instrução RT, verifica se o destino do pacote é ela. Se sim, retira o resultado do pacote e coloca no buffer de resultados para que possa ser utilizado pela operação corrente.

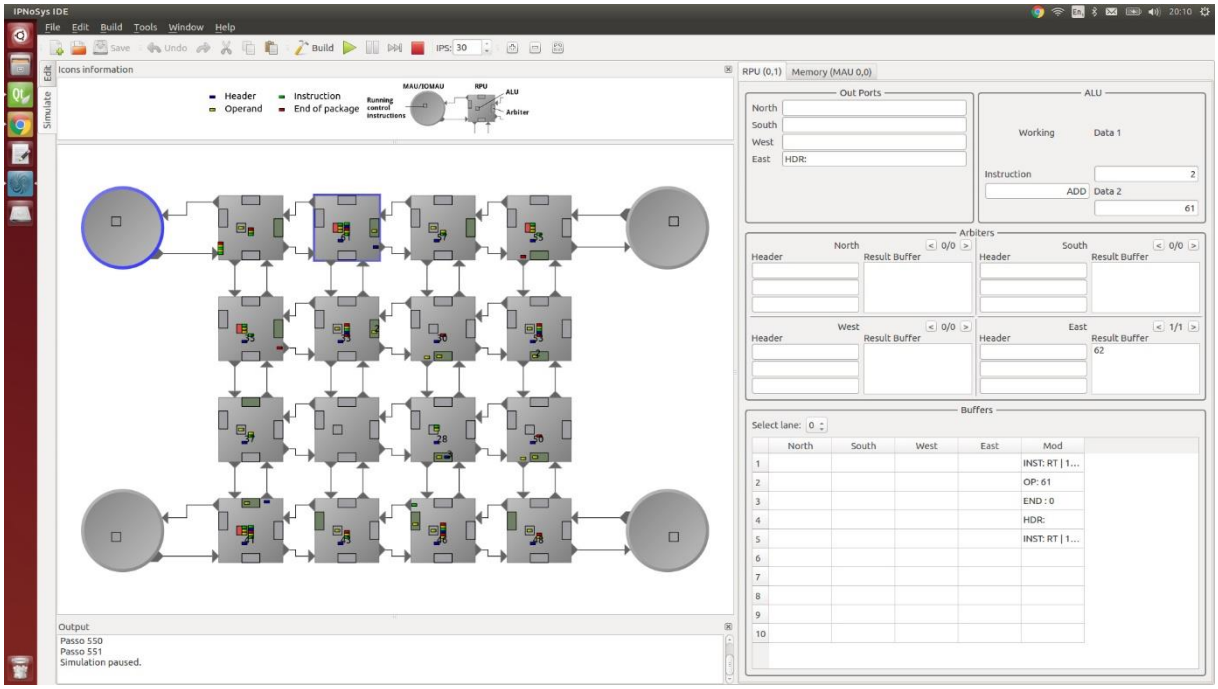
Com essas implementações é possível resolver todos os problemas de dependências de dados entre as instruções. As três resoluções apresentaram um bom desempenho na execução. Inclusive a implementação da resolução da dependência do tipo C, que inclui novas palavras e transmissão de dados, não demonstrou ser um gargalo durante os testes efetuados.

4 RESULTADOS OBTIDOS

Como forma de validar a arquitetura proposta, vários testes foram executados para averiguar seu desempenho. Primeiramente a IPNoSys com software *pipelining* foi comparada com a IPNoSys original. Em seguida, foi medido a perda de desempenho quando da necessidade de tratar as dependências de dados. Para esses testes foram utilizadas aplicações genéricas compostas por apenas uma *thread* com instruções lógicas e aritméticas. Por último, será mostrado os resultados referentes da execução de uma aplicação de uma multiplicação de matrizes.

As aplicações foram escritas em PDL e executadas numa rede 4x4 com auxílio da IPNoSys IDE, desenvolvida por Oliveira e Fernandes (2015). A IPNoSys IDE integra um editor de texto, o montador e o ambiente de simulação originais da IPNoSys (Figura 41) em que é possível acompanhar à execução do programa automática ou iterativa de forma animada.

Figura 41 - IPNoSys IDE



Fonte: Autoria própria

Ao fim da simulação a IDE mostra todas as estatísticas do processo geradas pelo simulador, incluindo: a quantidade de ciclos de *clock* necessários para execução

do programa, quantidade de palavras transmitidas e potência dissipada em miliwatts (mW). Essas informações foram utilizadas como métricas para as comparações deste trabalho.

4.1 IPNoSys SP *vs* IPNoSys Original

Para averiguar o ganho de desempenho que a nova versão tem em relação a versão original, cinco aplicações contendo 5, 10, 15, 30 e 60 instruções lógicas e aritméticas, com dependências de dados, foram executas variando a quantidade de iterações entre 10, 100, 1 mil e 10 mil vezes. Como visto na Seção 3.2.1, a IPNoSys 4x4 está limitada a executar apenas 15 instruções em modo software *pipelining*. As aplicações contendo 30 e 60 instruções foram executadas configurando o campo IPR_LOOP da instrução LOOP, para que as RPUs com o laço possam executar mais do que uma instrução. A aplicação contendo 30 instruções foi executada com IPR_LOOP configuração com valor 2 (2 instruções por RPU) e a aplicação com 60 instruções, com valor 4 (4 instruções por RPU). A Tabela 4 lista a quantidade de ciclos de *clock* de todas as execuções efetuadas[4]. A esquerda é relacionado o número de iterações executados e a cima, a quantidade de instruções de cada aplicação juntamente com a arquitetura que foi executada: IPNoSys com software *pipelining* (SP) ou a arquitetura original.

Tabela 4 - Relação de Ciclos de *Clock* IPNoSys SP *vs* Original

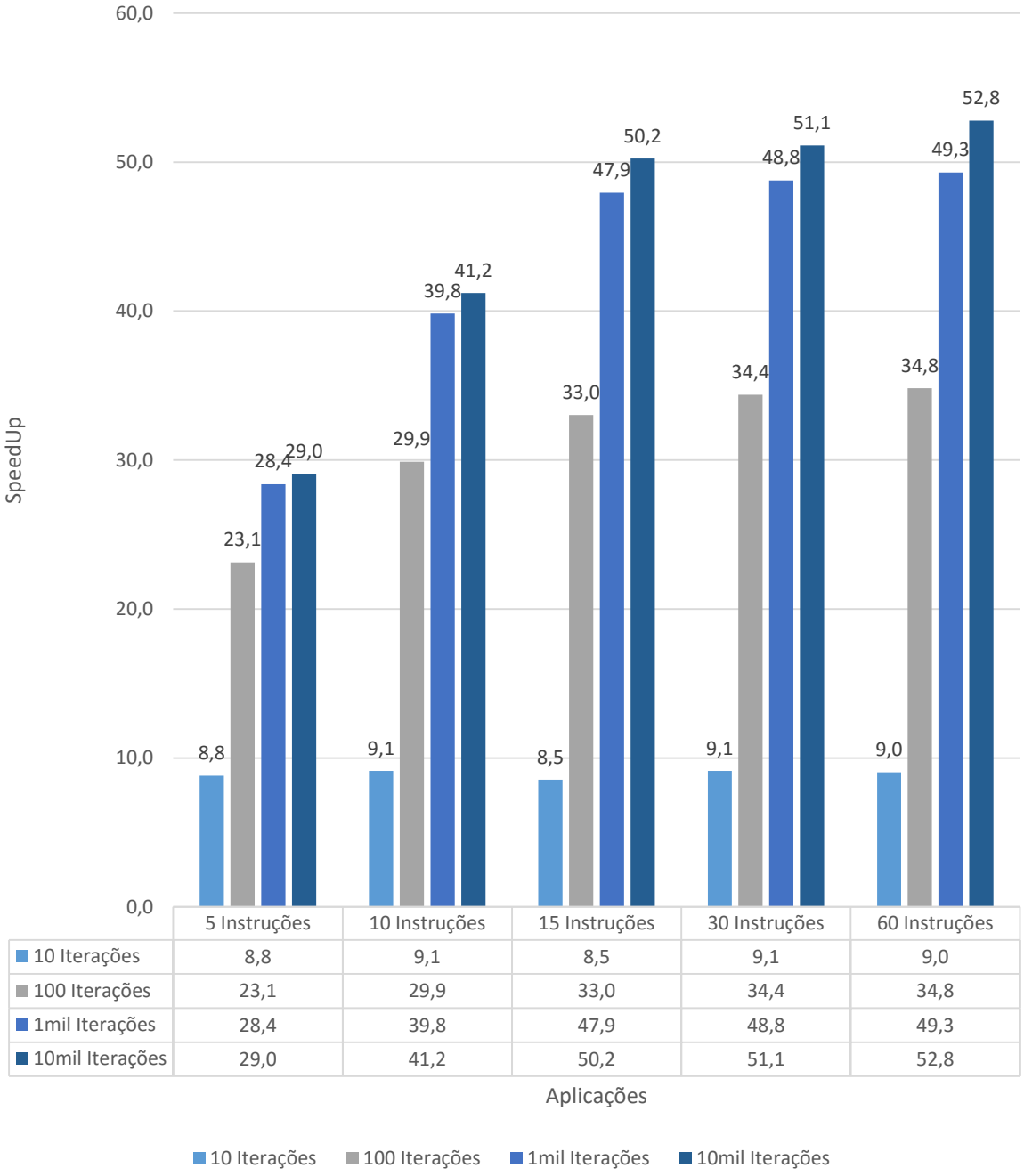
	5 Instruções		**10 Instruções**		**15 Instruções**		**30 Instruções**		**60 Instruções**	
#	**SP**	**Original**	**SP**	**Original**	**SP**	**Original**	**SP**	**Original**	**SP**	**Original**
10	301	2.652	398	3.632	510	4.356	707	6.456	1.179	10.656
100	1.021	23.622	1.118	33.422	1.233	40.716	1.795	61.716	2.979	103.716
1mil	8.221	233.322	8.318	331.322	8.433	404.316	12.595	614.316	20.979	$1.03x10^4$
10mil	80.221	$2.33x10^3$	80.318	$3.31x10^4$	80.433	$4.04x10^4$	120.095	$6.14x10^4$	190.079	$10.0x10^5$

Fonte: Autoria própria

[4] Alguns valores foram aproximados no formato de notação cientifica devido a ordem de grandeza ser muito alta.

Através destes testes é possível notar que a versão com software pipelining é, no mínimo, quase 10 vezes mais rápida do que na IPNoSys original. O Gráfico 1 mostra o *SpeedUp*[5] obtido pelo software *pipelining*.

Gráfico 1 – SpeedUp IPNoSys SP *vs* Original (*Clock*)



Fonte: Autoria própria

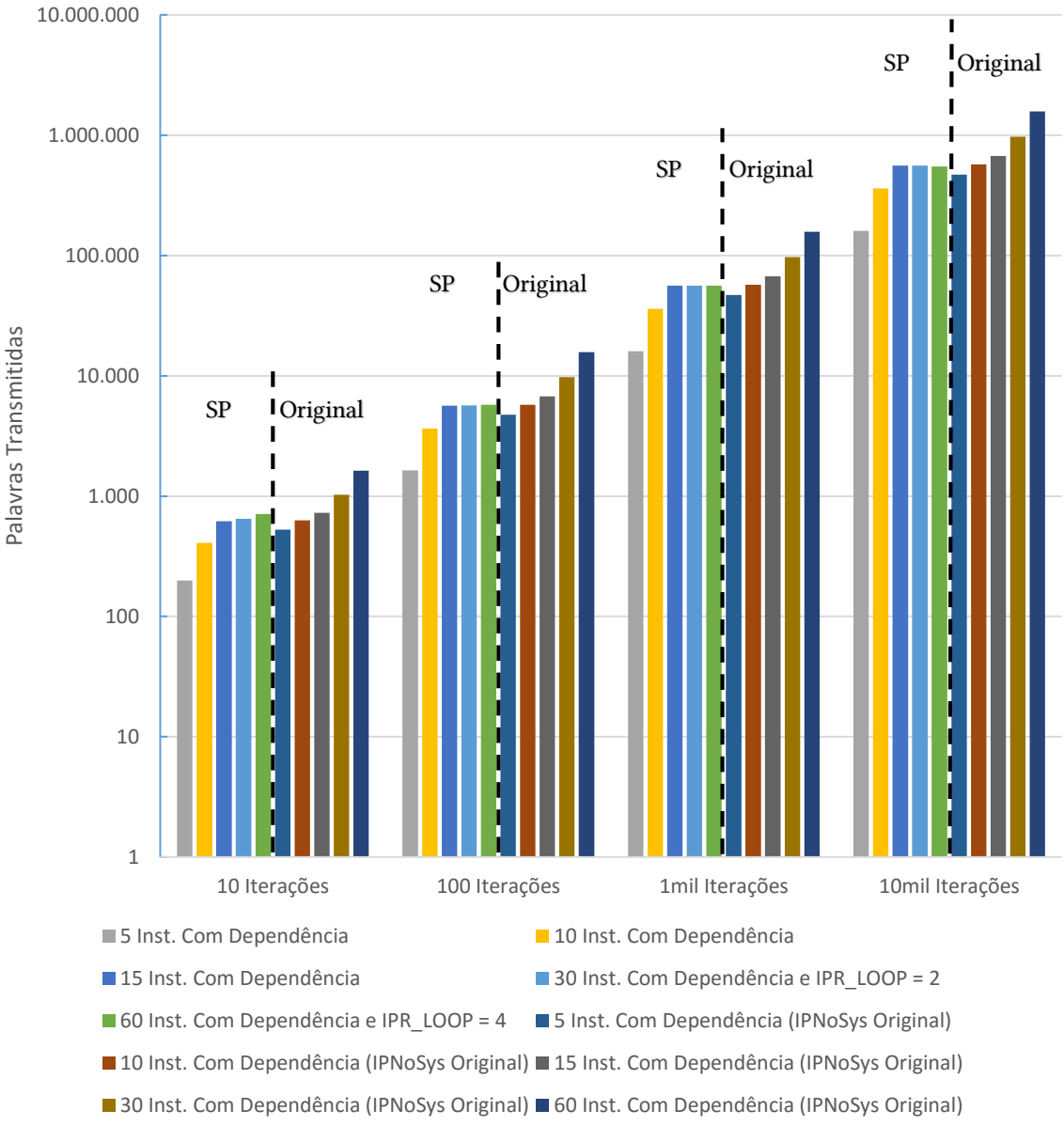
[5] Valor da quantidade de *clock* da versão original dividido pela quantidade de *clock* da versão com software *pipelining*.

A partir do Gráfico 1 nota-se que o ganho de desempenho do software *pipelining* cresce de acordo com o número de instruções e repetições, chegando a ser superior a 50 vezes quando a aplicação contém mais de 30 instruções e a partir de 1 mil iterações. Vale notar também que todas as instruções das aplicações contêm dependências de dados, o que obriga a versão com software *pipelining* inserir bolhas e transmitir os resultados através da instrução RT, prejudicando seu desempenho. Aplicações sem, ou com menos dependência de dados tendem a ser ainda mais rápidas, como será visto na seção 4.2.

Para poucas iterações o ganho em desempenho já é considerável, mas com aumento o número de repetições é que o custo benefício da implementação do software *pipelining* aparece. As maiores diferenças no *SpeedUp* estão entre as execuções de 10 e 100 iterações. Após 1 mil iterações o ganho o *SpeedUp* torna-se estável.

Para comparar o impacto na transmissão de mensagens, o Gráfico 2 apresenta o número de palavras transmitidas entre as RPUs e RPUs e MAU. No eixo vertical é relacionado o número de palavras transmitidas e, no eixo horizontal, as iterações. As aplicações com software *pipelining* (SP) e da versão original estão separadas pela linha tracejada.

Gráfico 2 – IPNoSys SP *vs* Original (Transmissão de Palavras)



Fonte: Autoria própria

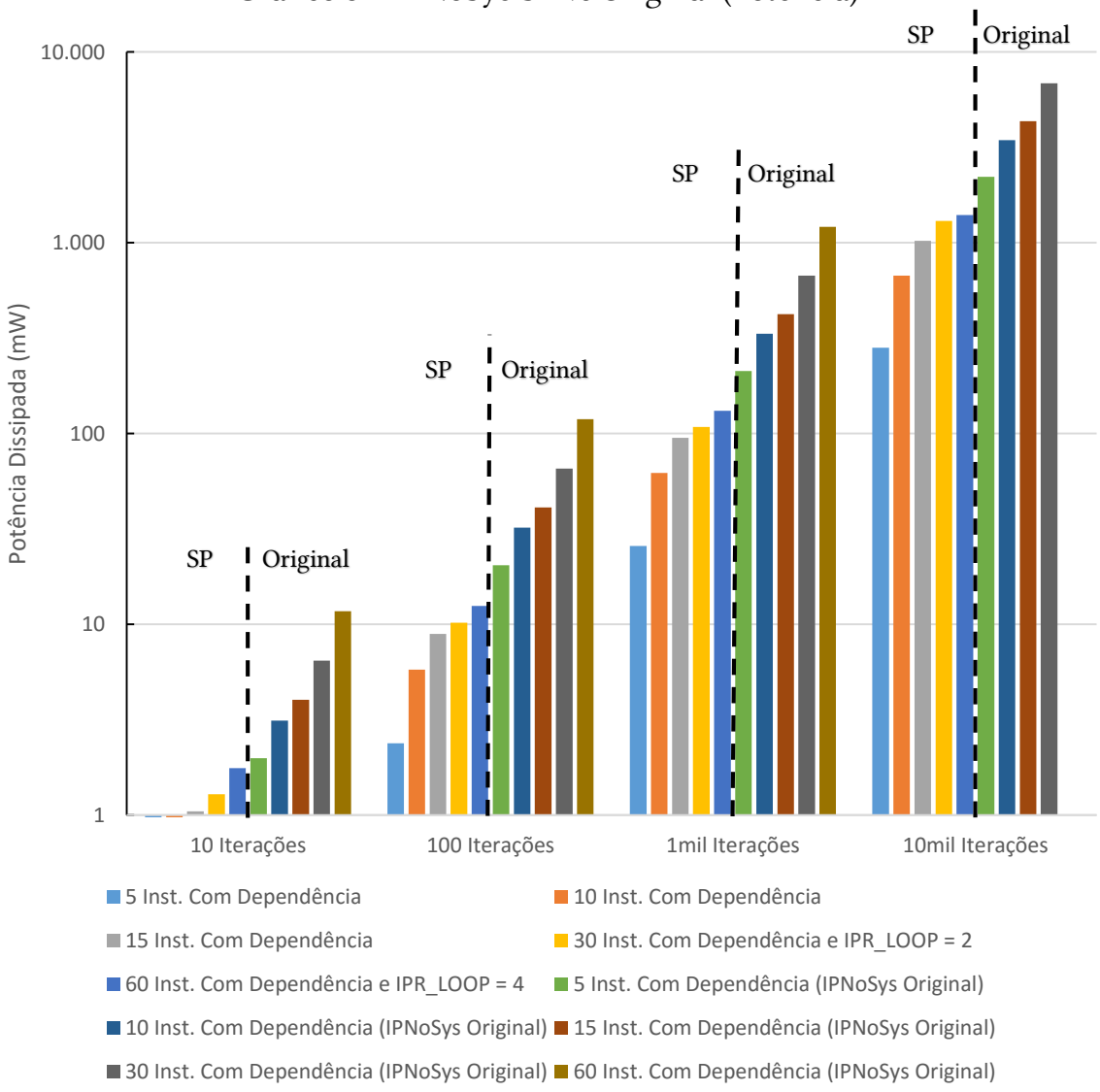
A partir da análise do Gráfico 2 é possível notar que a quantidade de transmissão de palavras cresce exponencialmente[6] com o aumento do número de iterações e instruções nas duas arquiteturas. Contudo, mesmo levando em conta que a cada iteração, a RPU configurada com software *pipelining* deve transmitir um novo pacote contendo a instrução RT, o número total de transmissões ainda é um pouco menor do que a versão original. No entanto a versão com software *pipelining* atinge o

[6] O gráfico está valorado em escala exponencial.

ápice de transmissão na aplicação com 15 instruções e sofre uma pequena queda nas versões de 30 e 60 instruções. Isso se deve ao fato de que acima de 15 instruções, há mais de uma instrução na mesma RPU, de modo que não é necessário utilizar os pacotes com instruções RT para propagar o resultado.

Em se tratando de uma arquitetura, vale averiguar a potência dissipada pelo processamento da aplicação. O Gráfico 3 apresenta os valores medidos nos testes feitos. No eixo vertical é relacionado a potência, e no eixo horizontal, as iterações. As aplicações com software *pipelining* (SP) e da versão original estão separadas pela linha tracejada.

Gráfico 3 – IPNoSys SP *vs* Original (Potência)



Fonte: Autoria própria

O valor de potência informado pelo simulador da IPNoSys é apenas dos *buffers* (de entrada e de resultado) e baseia-se no chaveamento de bits (mudança dos bits). Ou seja, quanto mais transmissões mais se usa *buffers* e mais potência é consumida. Com a quantidade de palavras transmitidas relativamente inferior, na IPNoSys com software *pipelining*, a potência dissipada também tende a ser mais baixa que a versão original. As aplicações com menos de 15 instruções e 10 iterações necessitaram de décimos de potência para serem completadas.

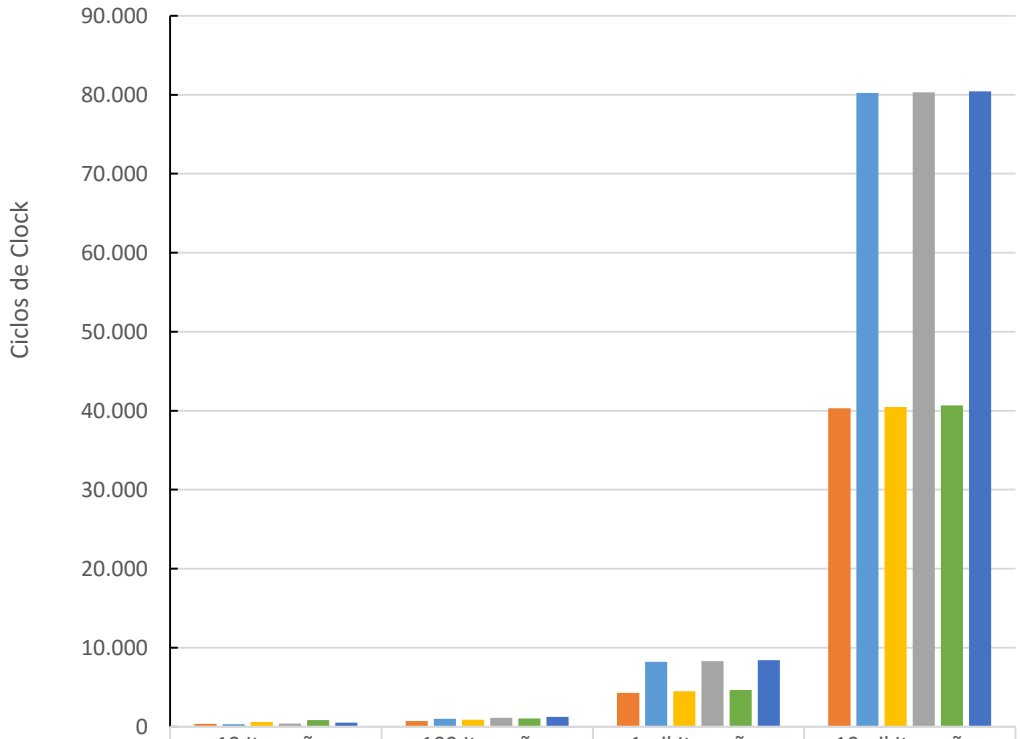
Os números mostram que em todos os testes efetuados, a versão com software *pipelining* oferece vantagem em relação a versão original. Apesar dos testes serem genéricos, é possível ter uma noção do ganho de desempenho adquirido, principalmente no que tange a velocidade de processamento.

4.2 IPNoSys SP Com e Sem Dependências de dados

Para verificar o impacto da estratégia de tratamento de dados utilizada na IPNoSys com software *pipelining*, dois grupos com três aplicações contendo 5, 10 e 15 instruções lógicas e aritméticas cada, foram executas variando a quantidade de iterações entre 10, 100, 1 mil e 10 mil vezes. O primeiro grupo de três aplicações não contém nenhuma dependência de dados, enquanto o segundo grupo, 100% de dependência. As métricas utilizadas foram quantidade de ciclos de *clock*, quantidade de mensagens transmitidas e potência.

Após 15 instruções, na IPNoSys 4x4, cada RPU executará mais de uma instrução. Com isso, a dependência acontece dentro da própria RPU, não havendo mais necessidade de se propagar o resultado. O Gráfico 4 mostra os ciclos de *clock* necessários para todos testes efetuados.

Gráfico 4 – IPNoSys SP Com*vs* Sem Dependências de Dados (*Clock*)



	10 Iterações	100 Iterações	1mil Iterações	10mil Iterações
5 Inst. Sem Dependência	347	700	4.300	40.300
5 Inst. Com Dependência	301	1.021	8.221	80.221
10 Inst. Sem Dependência	587	878	4.478	40.478
10 Inst. Com Dependência	398	1.118	8.318	80.318
15 Inst. Sem Dependência	827	1.061	4.661	40.661
15 Inst. Com Dependência	510	1.233	8.433	80.433

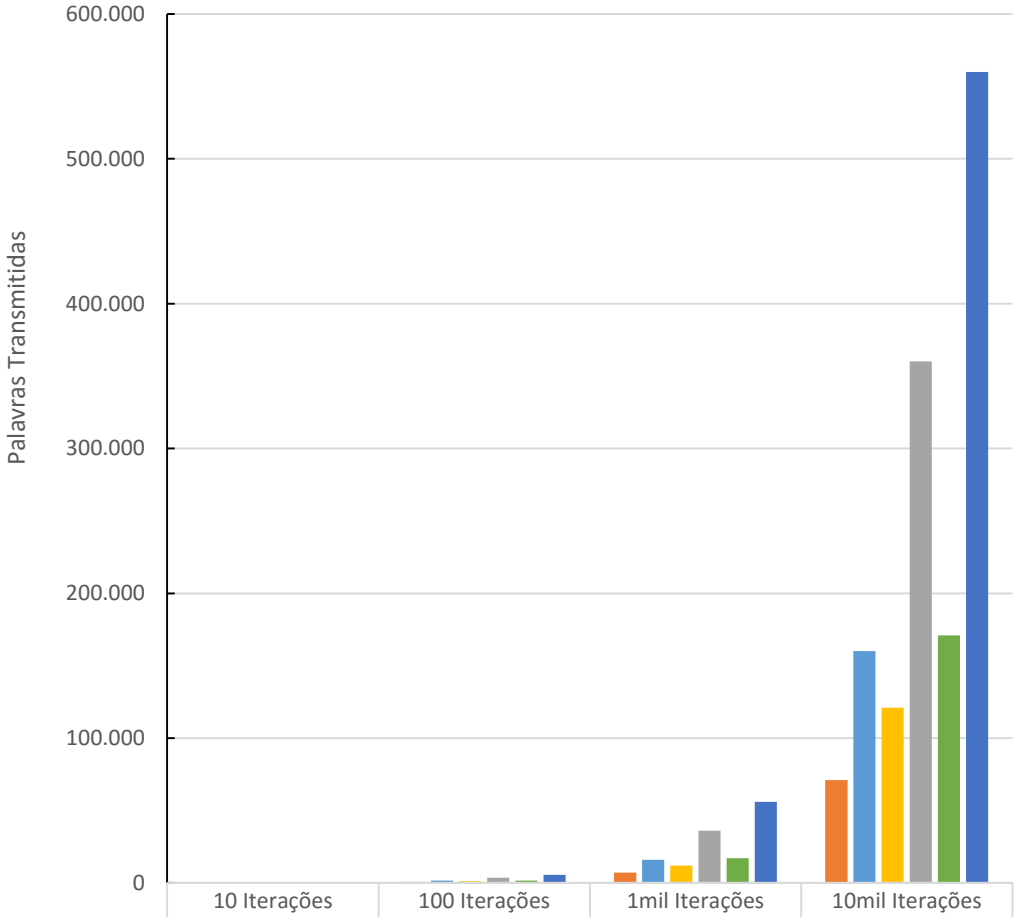
Fonte: Autoria própria

É interessante notar a partir desses dados que com poucas iterações, as aplicações com dependências são mais rápidas do que as sem. Isso acontece porque para efetuar uma operação, a RPU deve aguardar os dados chegarem. Quando as instruções não têm dependência, esse dado vem de uma das MAUs e caminha junto com o pacote. Na IPNoSys com software *pipelining*, quando há dependência, esse dado vem de uma RPU. Na RPU, logo após a execução da instrução o resultado é enviado para SU e é transmitido pelo RT. Só depois disso é que a RPU transfere a próxima palavra do pacote regular. Isso faz com que os operandos que são enviados pelo RT cheguem mais rápido ao seu destino dos que os operandos que estão caminhando junto com o pacote regular.

Com o aumento do número de iterações, o quadro se inverte. Pois a cada iteração, um pacote com a instrução RT deve ser transmitido. Os testes com 1 mil e 10 mil iterações mostraram que as aplicações sem dependências levam praticamente metade do tempo do que as aplicações com dependências.

A principal diferença entre as duas aplicações durante a execução é a quantidade de palavras criadas pela arquitetura. O Gráfico 5 mostra os valores medidos em todos os testes executados.

Gráfico 5 – IPNoSys SP Com *vs* Sem Dependências (Transmissão de Palavras)

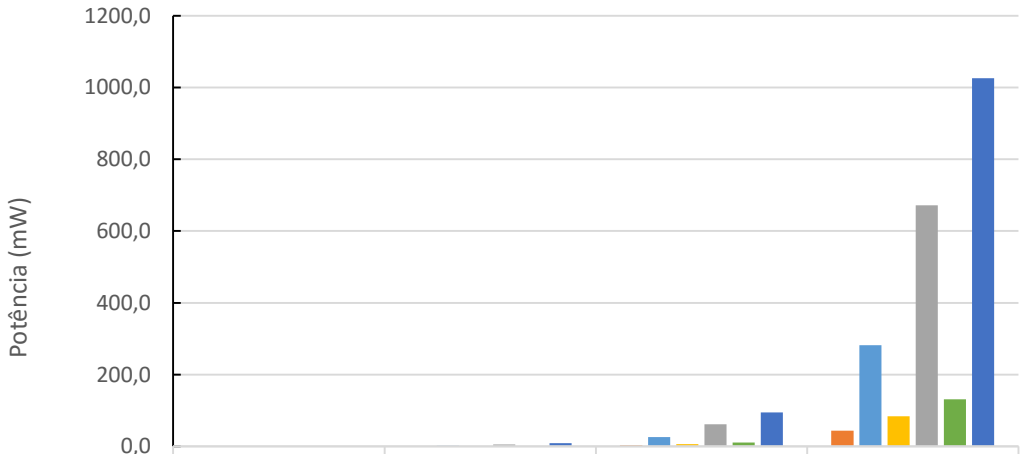


	10 Iterações	100 Iterações	1mil Iterações	10mil Iterações
5 Inst. Sem Dependência	71	710	7.100	71.000
5 Inst. Com Dependência	199	1.639	16.039	160.093
10 Inst. Sem Dependência	121	1.210	12.100	121.000
10 Inst. Com Dependência	409	3.649	36.049	360.049
15 Inst. Sem Dependência	171	1.710	17.100	171.000
15 Inst. Com Dependência	619	5.659	56.059	560.059

Fonte: Autoria própria

Os dados do Gráfico 5 mostram um crescimento linear na quantidade de transmissão de mensagens nas duas arquiteturas. Entretanto, a aplicação que contem dependência de dados cresce com um fator maior, sendo em torno de 5 vezes maior na aplicação com 15 instruções, com esse número podendo se distanciar mais com o aumento da quantidade de iterações. Por último foi averiguada a potência nas duas aplicações. O Gráfico 6 apresenta os dados de todas as execuções.

Gráfico 6 – IPNoSys SP Com *vs* Sem Dependências (Potência)



	10 Iterações	100 Iterações	1mil Iterações	10mil Iterações
5 Inst. Sem Dependência	0,2	0,4	3,6	43,6
5 Inst. Com Dependência	0,3	2,4	25,8	281,8
10 Inst. Sem Dependência	0,5	0,9	7,0	83,7
10 Inst. Com Dependência	0,7	5,8	62,1	671,5
15 Inst. Sem Dependência	0,9	1,6	11,0	131,2
15 Inst. Com Dependência	1,0	8,9	94,8	1025,4

Fonte: Autoria própria

Os números apresentados no Gráfico 6 mostram que a execução da aplicação com dependência gasta muito mais energia do que a versão sem. Isso porque o uso de *buffers* é reduzido. Na aplicação com 15 instruções e 10 mil repetições, o gasto é quase 10 vezes superior, podendo aumentar de acordo com o número de iterações.

Todas essas informações obtidas mostram que o tratamento de dependência de dados utilizando a estratégia da instrução RT gera uma grande perda de desempenho. Contudo, não é possível se ter aplicações com 100% de instruções sem dependências de dados, e num quadro geral, ainda é muito vantajoso em relação ao modelo original.

4.3 Multiplicação de Matrizes

Com intuito de testar a nova arquitetura com uma aplicação real, foi escolhido um algoritmo que tem como objetivo calcular a multiplicação de duas matrizes de ordem 64 ($A_{[64][64]} \times B_{[64][64]}$). Este algoritmo foi utilizado no trabalho de Araújo (2012) como forma de avaliar o desempenho da IPNoSys original naquele trabalho. A escolha do algoritmo de multiplicação de matrizes também se deu pela facilidade de ser paralelizado. É possível dividir a matriz alvo em quatro partes e criar uma *thread* para calcular cada parte em paralelo.

O algoritmo original, implementado em PDL, se baseia em dois laços aninhados para poder percorrer as matrizes e efetuar as operações. Enquanto o laço mais interno percorre as colunas da matriz, o laço mais interno percorre a linha. A cada iteração do laço mais interno, são efetuados acessos a memória para ler os valores das linhas e colunas das matrizes A e B, as operações lógicas e aritméticas e a escrita do resultado (matriz $C_{[64][64]}$). Os laços de repetição são baseados no modelo explicado pela Figura 18, com vários pacotes, interagindo entre si. Modelo esse bastante custoso devido ao tempo desperdiçado transmitindo palavras.

Como forma de comparar os resultados obtidos pela arquitetura original e a que implementa software *pipelining*, foi implementado um algoritmo para multiplicação de matrizes que pudesse se utilizar do paralelismo em nível de instrução. O novo algoritmo se diferencia do original pela utilização da instrução LOOP.

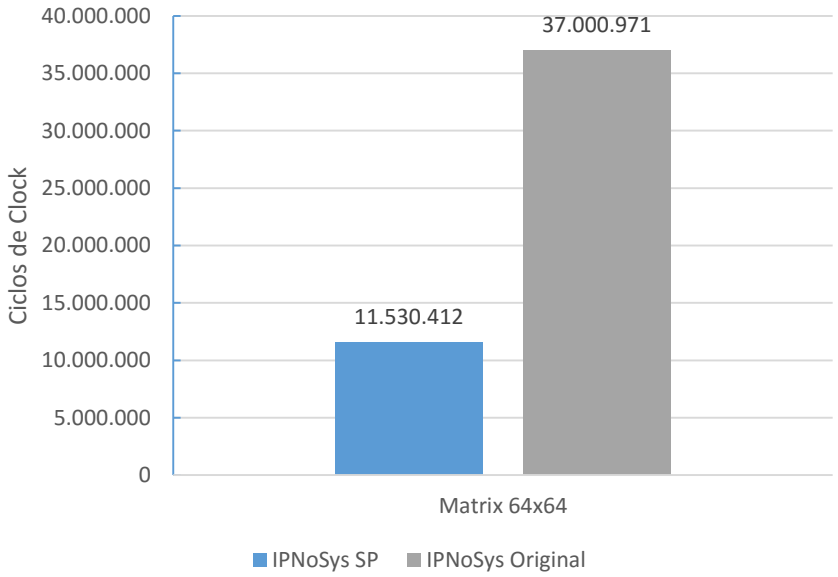
Para poder percorrer as duas matrizes, é essencial a utilização de laços aninhados. Como visto neste trabalho, a instrução LOOP simula apenas um laço. Não sendo possível, no mesmo pacote, a utilização de duas instruções LOOP seguidas uma da outra. Para se conseguir criar laços aninhados é necessário que os laços mais externos sejam baseados no modelo original do laço apresentado da Figura 18, no início do Capítulo 4, deixando sempre o laço mais interno com a instrução LOOP. Baseada nessa estratégia, um novo algoritmo para multiplicação de matrizes foi criado. Esse algoritmo contém dois laços. O mais externo contendo apenas algumas instruções de controle, e o mais interno com a instrução LOOP.

As métricas utilizadas são as mesmas dos testes anteriores: ciclos de clock, quantidades de palavras transmitidas e potência dissipada. Primeiramente será mostrado o cálculo da matriz com apenas uma *thread*, e depois, com 4 *threads*.

4.3.1 Multiplicação de Matrizes com 1 thread

A versão do algoritmo de multiplicação de matrizes com apenas uma *thread* é bem simples e não se utiliza de execução paralela na IPNoSys original. Isso faz com que o desempenho seja bem baixo se comparado com a versão apresentada por este trabalho. O PDL desta aplicação com software *pipelining* pode ser consultado no Apêndice B. O Gráfico 7 mostra os valores dos ciclos de *clock* para as duas versões.

Gráfico 7 – Multiplicação de Matrizes com 1 *Thread* (*Clock*)



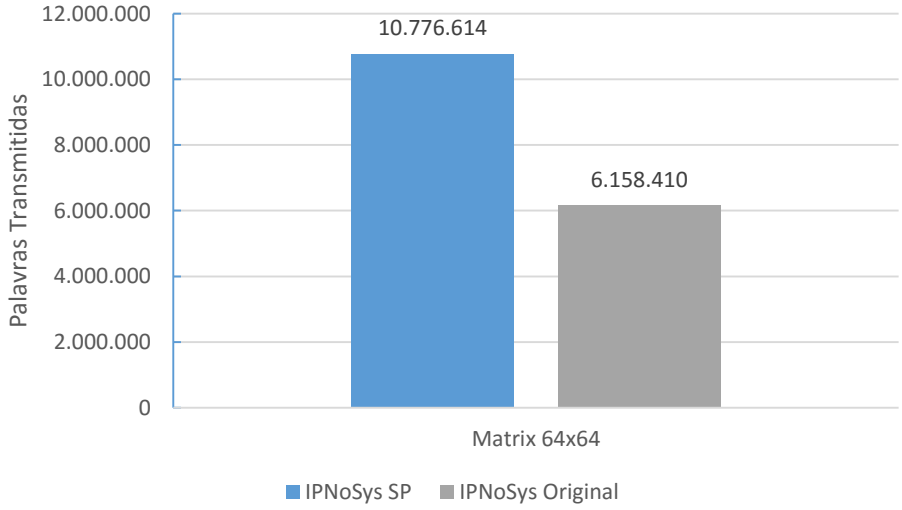
Fonte: Autoria própria

Apesar da diferença apresentada pelo Gráfico 7 parecer bem vantajosa, fica longe dos valores mostrados nos testes executados na Seção 4.1. Enquanto nos primeiros testes foram encontradas diferenças superiores a 10 vezes da IPNoSys com software *pipelinig*, neste exemplo o desempenho dela foi de cerca de três vezes melhor. Isso se deve ao fato do modelo da aplicação com laços aninhados, o qual é implementado de forma híbrida (laço externo com modelo original e laço interno com a instrução LOOP).

Segundo Araújo (2012), o caminho crítico da arquitetura IPNoSys é a instrução LOAD, e por isso é aconselhável usar a menor quantidade possível desse tipo de instrução para se obter um melhor desempenho. Essa instrução é utilizada para leitura de dados na memória. O motivo da perda de desempenho é porque quando um LOAD é decodificado por uma RPU, um pacote de controle é criado e roteado até uma MAU com a solicitação do dado. A MAU recupera esse dado da memória e envia um pacote de volta até a RPU destino. Enquanto esse procedimento acontece, na IPNoSys original, o restante do pacote que continha o LOAD fica estacionado sem poder prosseguir, na espera do dado. Na IPNoSys com software *pipelining*, as instruções anteriores ao LOAD continuam executando, mas as posteriores que precisam desse dado, ficam aguardando. Quando o dado chega é enviado através da instrução RT. Como na multiplicação de matrizes, a cada iteração do laço, é necessário fazer duas leituras da memória (coluna da matriz A e linha da matriz B), muito tempo é desperdiçado devido a instrução LOAD.

Com relação a transmissão de mensagens também é fácil notar que por causa da instrução RT com valores carregados por LOADs, a versão com software *pipelining* tem um número superior se comparada com a original, contudo esse valor é apenas o dobro do apresentado pela versão original. O Gráfico 8 mostra os resultados das duas execuções.

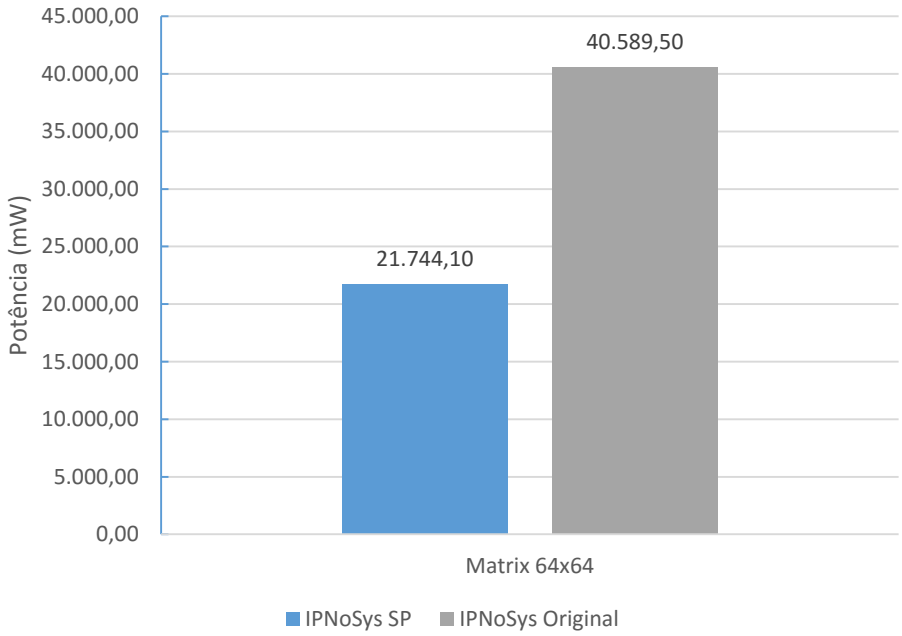
Gráfico 8 – Multiplicação de Matrizes com 1 *Thread* (Palavras Transmitidas)



Fonte: Autoria própria

Por último foi averiguada o gasto de energia estre as duas versões. Como a IPNoSys com software *pipelining* tem sua execução mais rápida, é de se esperar que se gaste menos energia também, já que seus núcleos funcionarão por menos tempo. O Gráfico 9 mostra os valores da potência dissipada nas duas arquiteturas. A IPNoSys SP gastou pouco mais da metade do que a versão original.

Gráfico 9 – Multiplicação de Matrizes com 1 *Thread* (Potência)



Fonte: Autoria própria

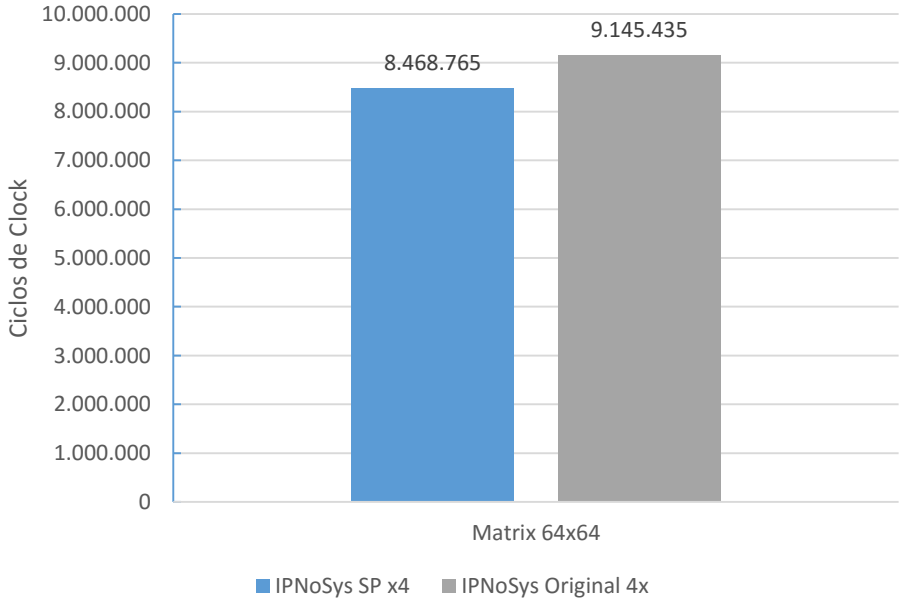
Os números registrados por esses testes mostram uma boa vantagem da implementação do software *pipelining* na IPNoSys, ganhando tanto em desempenho como em gasto de energia.

4.3.2 Multiplicação de Matrizes com 4 threads

Uma grande vantagem da IPNoSys é poder paralelizar aplicações através da divisão em *threads*. É possível ter até quatro *threads* (pacotes) sendo executadas pela rede. Motivados por isso, o algoritmo de multiplicação de matrizes na IPNoSys SP foi adaptado para executar tanto com paralelismo em nível de *thread* como em nível de instrução.

Contudo, nas duas versões da IPNoSys, a quantidade de instruções em cada pacote é um empecilho para este tipo de execução. Com quatro pacotes trafegando na rede, eles acabam se chocando e ficam impossibilitado de transmitir palavras para a próxima RPU pois a mesma está ocupada com uma instrução de outro pacote, fazendo a RPU entrar em modo de execução localizada. Dessa forma, poucas RPUs no caminho de um pacote ficam disponíveis para se criar paralelismo em nível de instrução. O Gráfico 10 contém os valores dos ciclos de *clock* das execuções para as duas versões.

Gráfico 10 – Multiplicação de Matrizes com 4 *Threads* (*Clock*)

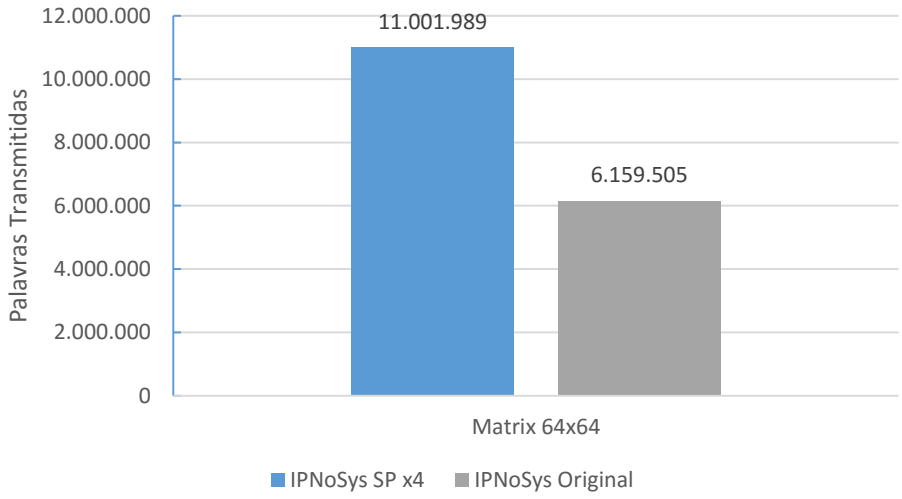


Fonte: Autoria própria

Devido a impossibilidade de se extrair o máximo de paralelismo em nível de instrução, a vantagem da versão com software *pipelining* é menor, como se pode notar a partir do Gráfico 10. A diferença foi em torno de 700 mil ciclos a menos.

Com relação a transmissão de mensagens não há impacto nas duas versões da arquitetura, pois, a quantidade de instruções são as mesmas da versão com apenas uma *thread*. Só que na versão com 4 *threads* elas só estão divididas em pacotes diferentes.

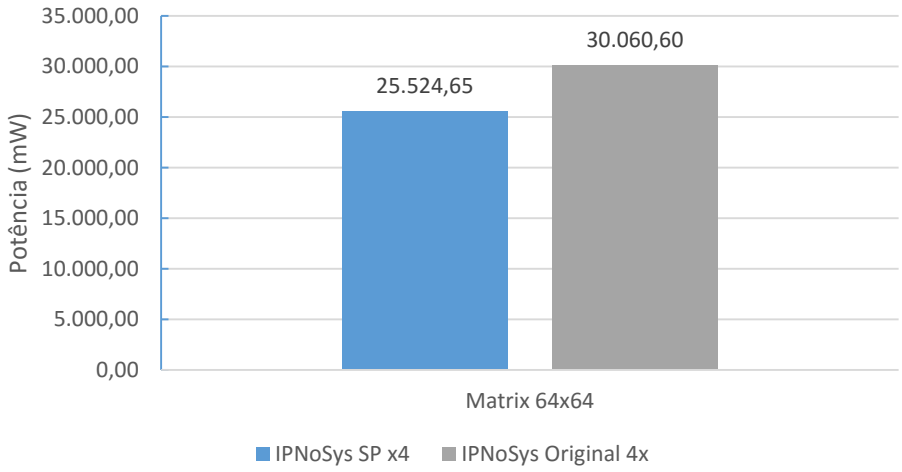
Gráfico 11 – Multiplicação de Matrizes com 4 *Threads* (Palavras Transmitidas)



Fonte: Autoria própria

A última métrica verificada foi o gasto de energia das duas arquiteturas. O Gráfico 12 apresenta os valores da comparação. A versão com software *pipelining* tem um gasto um pouco menor.

Gráfico 12 – Multiplicação de Matrizes com 4 *Threads* (Potência)



Fonte: Autoria própria

Se comparado aos valores da execução do algoritmo com apenas uma *thread*, pode-se notar que a versão com software *pipelining* tem um aumento no gasto de energia, enquanto a versão original tem uma redução. Isso acontece porque os pacotes da aplicação com 4 *threads* na IPNoSys original entram rapidamente em execução

localizada. Isso diminui a quantidade de transmissões e consequentemente o gasto de energia. Já na IPNoSys SP, os pacotes também entram em execução localizada, mas continuam transmitindo o RT. Esse passo faz com que seja verificado um aumento na quantidade de potência dissipada.

Como visto, a versão com software *pipelining*, tem um impacto bastante positivo na IPNoSys, acelerando as aplicações e reduzindo o gasto com energia em todos os testes efetuados.

5 CONSIDERAÇÕES FINAIS

Este trabalho mostrou que é possível unir a técnica de software *pipelining* na IPNoSys, uma arquitetura que implementa os conceitos de NoC, e obter um aumento de desempenho na execução de aplicações que contenham laços de repetição.

Para criar o efeito de paralelismo, duas novas instruções foram adicionadas a ISA da IPNoSys. Uma delas, chamada de LOOP, delimita um conjunto de instruções que estejam dentro de laço. A instrução LOOP também serve para configurar as RPUs para reterem uma ou mais instrução que esteja dentro do laço e execute-as repetidas vezes.

Foi diagnosticado a necessidade de tratamento de dependências de dados em três situações diferentes. Na primeira situação a dependência acontece dentro da própria RPU. Nesse cenário, o dado é apenas salvo no seu buffer de resultados para ser utilizado na próxima iteração. O segundo tipo de dependência, acontece entre uma instrução que está fora do laço e outra instrução que está dentro. Nesse caso, a RPU que executa a instrução dentro do laço aguarda a última iteração do laço para salvar o dado no buffer.

O último cenário de dependência de dados acontece entre instruções que estão dentro do laço de repetição. Para este caso foi criada mais uma nova instrução (RT) que é adicionada em um pacote de controle e roteado até a RPU destino juntamente com o resultado da operação. A instrução RT não é disponível ao programador, sendo criada automaticamente pela RPU a cada iteração do laço.

Os resultados apresentados mostraram que foi possível obter uma diminuição no número de ciclos de *clocks* e consequentemente aceleração da execução. Em aplicações com grande número de operações lógica aritméticas. A melhoria foi superior a dez vezes. E em aplicações com instruções de acesso a memória, um ganho de desempenho até três vezes.

Devido ao tratamento de dependências, o número de palavras transmitidas é bastante elevado na versão com software *pipelining*. Contudo, nos testes efetuados esse fato não prejudicou o desempenho das aplicações.

A nova arquitetura com software *pipelining* também mostrou um melhor desempenho em sua eficiência energética dos *buffers*. Parte disso se deve ao fato da execução ser mais rápida do que a versão original e por permitir menos transmissões entre as RPUs e consequentemente menos uso dos *buffers*.

Também foi mostrado que apesar da proposta de software *pipelining* ser caracterizada pela distribuição das instruções entre RPUs, o que limitaria a quantidade de instruções dentro do laço, uma solução inspirada no IPR original da arquitetura contorna esse problema. Outra limitação superada é a de laços aninhados, conseguida por meio de uma solução que combina o novo modelo (com a instrução LOOP) no laço mais interno e o modelo tradicional nos laços mais externos, podendo ainda obter um melhor desempenho que apenas o modelo tradicional.

Por fim, o modelo proposto de software *pipelining* manteve total compatibilidade com o conjunto de instruções e modelo de computação tradicional da IPNoSys, o que permite explorar o paralelismo em TLP, como já era feito no modelo original, mas também em ILP de forma mais eficiente.

O projeto tratado por esta dissertação gerou dois artigos. Um foi aceito, e será apresentado no IBERCHIP 2016 e o outro foi submetido para o periódico IEEE *Latin America* e aguarda resultado.

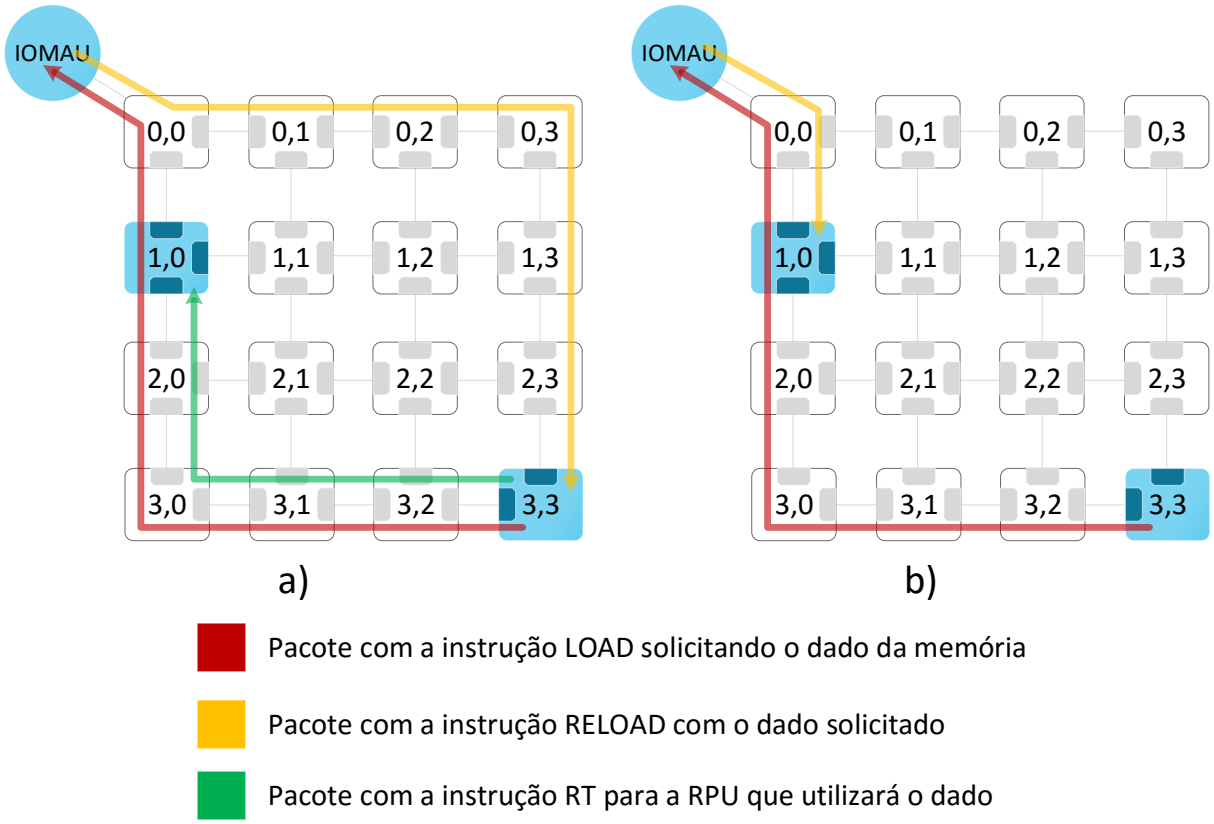
5.1 Trabalhos Futuros

Os resultados obtidos deixam evidente a melhoria proporcionada através da criação de paralelismo em nível de instrução do software *pipelining*. Entretanto alguns pontos precisam ser melhorados em trabalhos futuros. Dentre eles, a possibilidade de se criar laços de repetição, utilizando a instrução LOOP, por meio de desigualdades também. Na arquitetura apresentada neste trabalho, o laço se repete até que a quantidade de repetições seja igual ao informado no operando da instrução LOOP.

Outro fator que pode ser melhorado é a execução da instrução LOAD. Como visto, o acesso a memória é bastante lento se comparada as outras operações da IPNoSys, principalmente se o LOAD está dentro de um laço e deve propagar seu

resultado através do RT. Atualmente o processo se dá com a RPU criando e enviando um pacote de controle solicitando o dado até uma das MAUs. A MAU lê o dado na memória e envia para a RPU através de outro pacote de controle. E por último, a RPU recebe o dado e envia até a RPU que irá utilizá-lo através de outro pacote com a instrução RT (Figura 42a). Esse processo acontece a cada iteração do laço. Uma forma de melhorar esse mecanismo, seria no momento em que a MAU vai enviar o dado, ao invés de ser roteado para a RPU que fez a solicitação, a MAU enviaria diretamente para a RPU que irá necessitar do dado. Com isso, a instrução RT seria dispensada (Figura 42b), economizando tempo de processamento para criação do RT, banda para transmissão e ociosidade da RPU que aguarda para utilizar o dado.

Figura 42 - Procedimento da instrução LOAD (a) atual e (b) a ser proposto



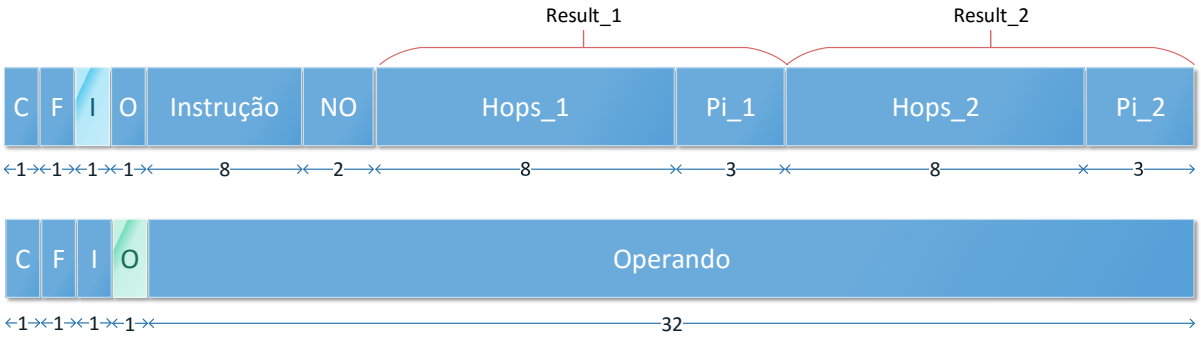
Fonte: Autoria própria

Outro mecanismo que pode ser melhorado é o caso da previsão de RPU na dependência de dados. Como forma de possibilitar a previsão, foi necessária a inserção de bolhas em instruções com menos de três operandos. Essa estratégia não é ótima

pois adiciona mais palavras na rede e aumenta a transmissão e armazenamento, que entre outras coisas aumenta a potência nos *buffers*.

Uma ideia para otimizar esse modelo e abolir as bolhas seria modificar os campos *Result*, que deixariam de indicar valores absolutos das posições das palavras onde os resultados devem ser inseridos para indicar a quantidade de *hops* (quantidade de instruções) à frente da que produz resultados onde deve ser inserido. Mas, para indicar a instrução que deve receber o resultado pode ser insuficiente pois para algumas instruções a ordem do operando importa (como DIV ou STORE por exemplo). Dessa forma o campo *Result* (que possui 11 bits) deveria ser segmentado em 2 partes: quantidade de *hops* (8 bits) e a ordem do operando (3 bits). Sabendo a quantidade de saltos (ou instruções) que o dado deve efetuar, é possível prever qual RPU utilizará o dado. Para isso funcionar é necessário modificar o *assembler* para que ele passe a contabilizar essa posição relativa de inserção do resultado ao invés da posição absoluta e fazer as adaptações nas MAUs e árbitros da RPU. A Figura 43 mostra os campos na nova palavra de instrução.

Figura 43 - Proposta para Nova Palavra de Instrução



Fonte: Autoria própria

Outro modelo a ser estudado é a possibilidade de criar laços aninhados utilizando instruções LOOPs seguidas. Esse modelo eliminaria a utilização de chamados de pacotes a cada iteração dos laços mais externos, reduzindo consideravelmente o tempo de execução de aplicações que se utilizam de vários laços.

Além disso, outros fatores que não foram abordados nesse trabalho podem ser estudados, como a modificação do tamanho da rede e aumento da quantidade de canais

virtuais. O aumento desses valores possibilita mais RPUs disponíveis para processamento. Esse fato pode afetar positivamente o desempenho de aplicações. Contudo, deve gerar um aumento também na potência dissipada.

Medeiros (2014) alterou o algoritmo de roteamento da IPNoSys para se adequar melhor ao seu projeto. Alterar o algoritmo de roteamento no desempenho de aplicações gera impacto no desempenho das aplicações (COSTA FILHO et al, 2014). É possível serem feitos estudos implementando o algoritmo de roteamento utilizado por Medeiros (2014) na arquitetura proposta por este trabalho e os resultados podem ser comparados.

Portanto, este trabalho abriu caminho para o desenvolvimento de diversas implementações e estudos baseado na IPNoSys com software *pipelining*. Pois o ganho de desempenho é bastante significativo e há diversas possibilidades de novas modificações.

REFERÊNCIAS

ADRIAHANTENAINA, Adrijean; CHARLERY, Hervé; GREINER, Alain; *et al.* SPIN: A scalable, packet switched, on-chip micro-network. *In*: **Design, Automation and Test in Europe, DATE**. [s.l.]: IEEE, **Anais...** 2003, p. 70–73. Disponível em: <http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=1253808>. Acesso em: 22 jun. 2015.

ARAÚJO, Silvio Roberto Fernandes de. **Projeto de Sistemas Integrados de Propósito Geral Baseados em Redes em Chip – Expandindo as Funcionalidades dos Roteadores para Execução de Operações: A plataforma IPNoSys**. 209 f. (Tese de Doutorado) - Universidade Federal do Rio Grande do Norte, 2012.

BENINI, L.; DE MICHELI, G. Networks on chips: a new SoC paradigm. *In*: **Computer**. [s.l.]: IEEE Computer Society Press, **Anais...** 2002, v. 35, p. 70–78. Disponível em: <http://dl.acm.org/citation.cfm?id=619071.621885>. Acesso em: 25 maio 2015.

BLACK, David C; BUNTON, Bill; DONOVAN, Jack; *et al.* **Systemc: From the ground up**. 1. ed. New York: Springer Science, 2005.

CALAZANS, Ney Laert Vilar. **Organização de Computadores**. Porto Alegre/RS: [s.n.], 2008.

CONCER, N.; IAMUNDO, S.; BONONI, L. aEqualized: A novel routing algorithm for the Spidergon Network On Chip. *In*: **Design, Automation & Test in Europe Conference & Exhibition**. Nice: [s.n.], **Anais...** 2009, p. 749–754. Disponível em: <http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=5090764>. Acesso em: 7 jun. 2015.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; *et al.* **Algoritmos: Teoria e Prática**. 3. ed. Rio de Janeiro/RJ: Camups, 2002.

COSTA FILHO, Raimundo Valter; FERNANDES, Silvio Roberto; NUNES, Denis Freire Lopes; *et al.* Exploração de Espaço de Projeto do Roteamento na Arquitetura IPNoSys. **HOLOS**, v. 4, p. 175–184, 2014. Disponível em: <http://www2.ifrn.edu.br/ojs/index.php/HOLOS/article/view/1909>. Acesso em: 15 jun. 2015.

CRUZ, M. O. **Roteamento em Zigue-zague para a plataforma IPNoSyS**. 71 f. (Trabalho de Conclusão de Curso) - Universidade Federal do Rio Grande do Norte, 2013.

CULLER, David; SINGH, Jaswinder Pal; GUPTA, Anoop. **Parallel Computer Architecture: A Hardware/Software Approach**. 1. ed. [s.l.]: Morgan Kaufmann, 1998. Disponível em: <http://www.dte.eis.uva.es/Docencia/ETSII/SMP/archivos/archibak/culler.pdf>.

DE ROSE, César A F; NAVAUX, Philippe O A. **Arquiteturas Paralelas**. 1. ed. Porto Alegre/RS: Sagra Luzzatto, 2003. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/erad-rs/2001/002.pdf>.

DOUILLET, Alban; GAO, Guang R. Software-Pipelining on Multi-Core Architectures. *In*: **16th International Conference on Parallel Architecture and Compilation Techniques**. Washington, DC: IEEE Computer Society, **Anais...** 2007, p. 39–48. Disponível em: <http://dl.acm.org/citation.cfm?id=1299109>. Acesso em: 21 abr. 2015.

FERNANDES, Silvio Roberto; OLIVEIRA, B. C.; COSTA, M.; *et al.* Processing while routing: a network-on-chip-based parallel system. **IET Computers & Digital Techniques**, v. 3, n. 5, p. 525 – 538, 2009. Disponível em: <http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=5200576>. Acesso em: 5 fev. 2015.

FLYNN, Michael J. Some Computer Organizations and Their Effectiveness. **IEEE Transactions on Computers**, v. C-21, n. 9, p. 948–960, 1972. Disponível em: <http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=5009071>. Acesso em: 16 jun. 2015.

FLYNN, Michael J. Very high-speed computing systems. **Proceedings of the IEEE**, v. 54, n. 12, p. 1901–1909, 1966. Disponível em: <http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=1447203>. Acesso em: 15 jun. 2005.

GAO, Lei; ZARETSKY, David; MITTAL, Gaurav; *et al.* A software pipelining algorithm in high-level synthesis for FPGA architectures. *In*: **10th International Symposium on Quality of Electronic Design**. San Jose, CA: IEEE, **Anais...** março de 2009, p. 297–302. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4810311>. Acesso em: 21 abr. 2015.

GOLDBERG, Benjamin; CRUTCHER, Emily; HUNEYCUTT, Chad; *et al.* Software bubbles: using predication to compensate for aliasing in software pipelines. *In*: **International Conference on Parallel Architectures and Compilation Techniques**. [s.l.: s.n.], **Anais...** 2002, p. 211–221. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1106019>.

GOVINDARAJAN, R.; ALTMAN, E.R.; GAO, G.R. Co-scheduling hardware and software pipelines. **2nd International Symposium on High-Performance Computer Architecture**, p. 52–61, 1996.

HESS, Cassiano Ricardo. **MDX-cc: Ambiente de Programação Paralela Aplicado a Cluster de Clusters**. 131 f. (Dissertação de Mestrado) - Pontifícia Universidade Católica do Rio Grande do Sul, 2003. Disponível em: <http://hdl.handle.net/10923/1655>. Acesso em: 22 jun. 2015.

JERRAYA, Ahmed Amine; WOLF, Wayne. **Multiprocessor Systems-on-Chips**. 1. ed. [s.l.]: Elsevier Science, 2004.

JONES, R B; ALLAN, V H. Software pipelining: a comparison and improvement. *In*: **23rd Annual Workshop and Symposium of Microprogramming and Microarchitecture**. Orlando, FL: IEEE Computer Society, **Anais...** 1990, p. 46–56. Disponível em: <http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=151426>. Acesso em: 15 jun. 2015.

LAM, Monica. Software pipelining: an effective scheduling technique for VLIW machines. **ACM SIGPLAN Notices**, v. 23, n. 7, p. 318–328, 1988. Disponível em: <http://dl.acm.org/citation.cfm?id=54022&CFID=686343476&CFTOKEN=20582674>. Acesso em: 5 nov. 2014.

LEE, Hyung Gyu; CHANG, Naehyuck; OGRAS, Umit Y; *et al.* On-chip communication architecture exploration. **ACM Transactions on Design Automation of Electronic Systems**, v. 12, n. 3, p. 23–es, 2007.

MEDEIROS, Aparecida Lopes de. **Implementação da técnica de Software Pipeline na Rede em Chip IPNoSys**. 92 f. (Dissertação de Mestrado) - Universidade Federal do Rio Grande do norte, 2014.

MEDEIROS, Aparecida Lopes de; MEDEIROS, Ana Luisa; KREUTZ, Márcio Eduardo. Exploração de Paralelismo em Nível de Instruções e de Tarefas em Uma Arquitetura de Processamento Não Convencional. **Revista Brasileira de Computação Aplicada**, v. 7, n. 3, p. 120–133, 2015.

MOORE, George Edward. Cramming More Components Onto Integrated Circuits. **Eletrônics**, p. 114–117, 1965.

MORAES, Fernando Gehm; CALAZANS, Ney Laert Vilar; MELLO, Aline Vieira de; *et al.* HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip. **Integration, the VLSI Journal**, v. 38, n. 1, p. 69–93, 2004. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167926004000185>. Acesso em: 25 maio 2015.

NUNES, Denis Freire Lopes. **Proposta de Implementação de um Pipeline em um Processador Reconfigurável Baseado em MIPS**. 66 f. (Trabalho de Conclusão de Curso de Bacharelado) - Universidade Federal Rural do Semi-Árido, 2012.

OLIVEIRA, Lucas V. A.; FERNANDES, Silvio Roberto. IPNoSys IDE Ambiente de Desenvolvimento e Simulação Integrado para uma Arquitetura não Convencional. **International Journal of Computer Architecture Education**, v. 4, n. 1, p. 21–24, 2015. Disponível em: <http://www2.sbc.org.br/ceacpad/ijcae/v4_n1_dec_2015/IJCAE_v4_n1_dez_2015_paper_6_vf.pdf>.

PATTERSON, David A; HENNESSY, John L. **Organização e Projeto de Computadores: A Interface Hardware/Software**. 4. ed. São Paulo/SP: Elservier, 2014.

PILLA, Mauricio; SANTOS, Rafael Ramos dos; CAVALHEIRO, Gerson Geraldo H. Introdução à Programação para Arquiteturas MultiCore. *In*: **IX Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul**. Caxias do Sul/RS: [s.n.], **Anais...** 2009, p. 71–102. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/erad/2009/005.pdf>. Acesso em: 15 jun. 2015.

RAU, B. R. Iterative Module Scheduling: An Algorithm for Software Pipelining Loops. *In*: **MICRO-27**. Palo Alto/CA: IEEE, **Anais...** 1994, p. 63–74.

REGO, Rodrigo Soares de Lima Sá. **Projeto e Implementação de uma Plataforma MP-SoC usando SystemC**. 142 f. (Dissertação de Mestrado) - Universidade Federal do Rio Grande do Norte, 2006.

SANKARALINGAM, Karthikeyan; NAGARAJAN, Ramadass; LIU, Haiming; *et al.* Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. **ACM SIGARCH Computer Architecture News**, v. 31, n. 2, p. 422–433, 2003. Disponível em: <http://dl.acm.org/citation.cfm?id=859667&CFID=686343476&CFTOKEN=20582674>. Acesso em: 12 maio 2015.

SKILLICORN, D. B.; TALIA, D. Models and languages for parallel computation. **ACM Computing Surveys**, v. 30, n. 2, p. 123–169, 1998.

STALLINGS, Willian. **Arquitetura e Organização de Computadores**. 8. ed. São Paulo/SP: Pearson Pratice Hall, 2013.

SUTTER, H. The free lunch is over: A fundamental turn toward concurrency in software. **Dr. Dobb's Journal**, v. 33, n. 3, p. 1–9, 2005. Disponível em: <http://www.mscs.mu.edu/~rge/cosc2200/homework-fall2013/Readings/FreeLunchIsOver.pdf>. Acesso em: 9 maio 2015.

TANENBAUM, Andrew S. **Organização estruturada de computadores**. 6. ed. São Paulo/SP: Pearson Pratice Hall, 2013. Disponível em: <https://feevale.bv3.digitalpages.com.br/users/publications/9788576050674?>.

TEWKSBURY, S.K.; UPPULURI, M; HORNAK, L.A. Interconnections/micro-networks for integrated microelectronics. *In*: **GLOBECOM '92 - Communications for Global Users: IEEE**. [s.l.]: IEEE Computer Society, **Anais...** 1992, v. 1, p. 180–186. Disponível em: <http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=276496>. Acesso em: 21 maio 2015.

THIES, William; CHANDRASEKHAR, Vikram; AMARASINGHE, Saman. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. *In*: **40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)**. Chicago, USA: IEEE Computer Society, **Anais...** 2007, p. 356–369. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4408240>. Acesso em: 20 jun. 2015.

TIEG, Steve. A new computing paradigm for the 21st century. **EETimes VIrtual Conference: System-on-Chip 2.0**, 2010.

TORRES, Gabriel. **Hardware - Versão Revisada e Atualizada**. 1. ed. [s.l.]: Nova Terra, 2013.

VASCONCELOS, Laércio. **Hardware Total**. 1. ed. São Paulo/SP: Makron Books, 2002.

WANG, Bo; SHANG, Shifeng; FANG, Qiming; *et al.* Parallel task developing based on software pipeline in multicore system. *In*: **International Symposium on Parallel and Distributed Processing with Applications, ISPA 2010**. Taipei: IEEE Computer Society, **Anais...** 2010, p. 542–549.

WEI, Haitao; YU, Junqing; YU, Huafei; *et al.* Software Pipelining for Stream Programs on Resource Constrained Multicore Architectures. **IEEE Transactions on Parallel and Distributed Systems**, v. 23, n. 12, p. 2338–2350, 2012. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6143921>. Acesso em: 21 abr. 2015.

WOLF, Wayne; JERRAYA, Ahmed Amine; MARTIN, Grant. Multiprocessor system-on-chip (MPSoC) technology. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 27, n. 10, p. 1701–1713, 2008.

ZEFERINO, Cesar Albenes. **Redes-em-Chip: Arquiteturas e Modelos para Avaliação de Área e Desempenho**. 242 f. (Tese de Doutorado) - Universidade Federal do Rio Grande do Sul, 2003. Disponível em: <https://www.lume.ufrgs.br/bitstream/handle/10183/4179/000397636.pdf?sequence=1>. Acesso em: 5 jun. 2015.

ZEFERINO, Cesar Albenes; SUSIN, Altamiro Amadeu. SoCIN: a parametric and scalable network-on-chip. *In*: **16th Symposium on Integrated Circuits and Systems Design**. [s.l.]: IEEE, **Anais...** 2003, p. 169–174. Disponível em: <http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=1232824>. Acesso em: 22 jun. 2015.

APÊNDICE A – PDL DO FATORIAL UTILIZANDO LAÇO COMUM

```
1  PROGRAM fatorial
2  DATA
3       a = 5 // valor que se deseja saber o fatorial
4       b     // endereço onde o resultado será salvo
5  PACKAGE pac_1 // pacote de inicialização
6  ADDRESS MAU_0
7       LOAD MAU_0 f;
8            a;
9       SEND MAU_0 fat;
10           prog_0, laco
11           f;
12      SEND MAU_0 result;
13           prog_0, laco
14           1;
15      EXEC MAU_0; // chamando a primeira iteração
16           prog_0, laco;
17 END
18
19 PACKAGE laco // pacote que contém o laço de repetição
20 ADDRESS MAU_0
21      COPY fat1 fat2;
22           fat = 0;
23      COPY result1;
24           result = 0;
25      MUL  result2 result3;
26           result1
27           fat1;
28      SUB  fat3 fat4;
29           fat2
30           1;
31      BNE  voltar; // verificando se é a última iteração
32           fat3
33           0;
34      SEND MAU_0 result;
35           prog_0, pac_0
36           result2;
37      EXEC MAU_0; // chamando o pacote final
38           prog_0, pac_0;
39      JUMP fim;
40 voltar:
41      SEND MAU_0 fat;
42           prog_0, laco
43           fat4;
44      SEND MAU_0 result;
45           prog_0, laco
46           result3;
```

```
47        EXEC MAU_0; // chamando o pacote laço novamente
48             prog_0, laco;
49   fim:
50        NOP;
51   END
52
53   PACKAGE pac_0 // pacote com o fim do programa
54   ADDRESS MAU_0
55        COPY result1;
56             result = 0;
57        STORE MAU_0;
58             result1
59             b;
60        EXIT;
61   END
62   END_PROGRAM
```

APÊNDICE B – PDL DA MULT. DE MATRIZES COM 1 THREAD

```
1   PROGRAM matriz_1_thread
2   DATA
3   // 12288 endereços de memória das 3 matrizes
4   PACKAGE pac_1
5   ADDRESS MAU_0
6         SEND MAU_0 endA;
7               matriz_1_thread, laco
8               0;
9         SEND MAU_0 endB;
10              matriz_1_thread, laco
11              4096;
12        SEND MAU_0 endC;
13              matriz_1_thread, laco
14              8191;
15        EXEC MAU_0;
16              matriz_1_thread, laco;
17  END
18
19  PACKAGE laco
20  ADDRESS MAU_0
21        COPY endA1
22              endA = 0;
23        COPY endB1;
24              endB = 0;
25        COPY endC1;
26              endC = 0;
27        LOOP 8 1;
28              64;
29        ADD x1 x2;      // ===== início do laço
30              x1 = 0
31              64;
32        ADD y1 y2;
33              y1 = 0
34              1;
35        ADD endA3;
36              endA1
37              x2;
38        ADD endB3;
39              endB1
40              y2;
41        LOAD MAU_0 ca;
42              endA3;
43        LOAD MAU_0 lb;
44              endB3;
45        MUL r1;
46              ca
```

```
47              lb;
48        ADD r2 r3;      // ===== fim do laço
49              r2 = 0
50              r1;
51        ADD endC2 endC3;
52              endC1
53              1;
54        STORE MAU 0;
55              r3
56              endC2;
57        COPY i endC4;
58              endC3;
59        BNE voltar;
60              12288
61              i;
62        EXEC MAU 0;
63              matriz_1_thread, final;
64        JUMP fim;
65  voltar:
66        SEND MAU 0 endA;
67              matriz_1_thread, laco
68              endA4;
69        SEND MAU 0 endB;
70              matriz_1_thread, laco
71              endB4;
72        SEND MAU 0 endC;
73              matriz_1_thread, laco
74              endC4;
75        EXEC MAU 0;
76              matriz_1_thread, laco;
77  fim: NOP;
78  END
79
80  PACKAGE final
81  ADDRESS MAU 0
82        EXIT;
83  END
84  END_PROGRAM
```